Cumulative Abstract Semantics

ANDREW FOX, University of Colorado Boulder, USA
CADE LUEKER, University of Colorado Boulder, USA

Cumulative abstract semantics provides a promising solution to the expression problem, but is still an underdeveloped theory. The reusability that this framework provides has the potential to lower the time and cost of creating abstract interpreters, as well as facilitating reuse of research. Syntax and semantics are inherently intertwined and difficult to define in isolation. Thereby an analyzer is forced to fuse the syntax, control flow directionality, and abstract domain. We build on cumulative abstract semantics as a theory. Our previous work constrained cumulative semantics to effect systems, we lift our theory to any higher ordered language. Demonstrating the decomposition of syntax and introduction of semantics do not require an a priori effect system, but can be expressed with type classes and continuations. We also provide a formalization of this framework, as well as two implementations in different languages for concrete interpretation to demonstrate it's scalability and use. We found that there was not a performance impact between the two languages, and neither us too much of a performance cost. We proved that this is indeed a valid framework, but it still requires more work to be done.

1 Introduction

Abstract interpretation frameworks face a fundamental tension between simplicity and modularity. While existing approaches enable the construction of static analyzers across new, but similar, abstract domains, there is no methodology for changing the directionality of analysis within the same engine. Existing solutions require significant code duplication between engines. The nascent *cumulative abstract semantics* propose an elegant solution to this problem, yet this approach remains largely unexplored beyond initial demonstrations [8]. A formalization establishing the relationship between cumulative semantics and traditional abstract interpretation is still missing, as well as separating the theory behind this approach and the language in which it is implemented. Furthermore, there exists no evidence supporting more realistic domain features such as state and iteration. Without these foundations, cumulative semantics remain a promising but unproven technique for building modular abstract interpreters.

Cumulative semantics reshape how abstract interpreters are constructed at a fundamental level. Traditional approaches require implementing a complete interpreter for each new analysis, duplicating substantial amounts for evaluation logic when either the semantic domain or flow of the engine changes. In contrast, cumulative abstract semantics decomposes evaluation logic into smaller, more generic, and reusable fragments. The separation of the control flow and the syntax from the abstract domain specific operations allows for a high degree of reuse across distinct instantiations of interpreters. Two analysis with the same flows can reuse the same control flow fragments, only using unique components for differing domain semantics. Similarly, the same domain operations can support both forward and backward analysis by composing with different control flow handlers. With the potential for automatic code generation, this framework also permits the extension of the syntax, allowing developers to easily update languages with new features or add new languages completely without affecting any of the old analyses. The reusability of this framework has the potential to substantially lower the cost of developing and maintaining new static analyses.

Achieving orthogonality between syntax and semantics in abstract interpretation is fundamentally difficult because interpreters must couple these concerns to produce any meaningful output. Typically the choice of control flow must be implemented based on the domain specific semantic

Authors' Contact Information: Andrew Fox, Andrew.Fox@colorado.edu, University of Colorado Boulder, Boulder, Colorado, USA; Cade Lueker, University of Colorado Boulder, Boulder, Colorado, USA, cade.lueker@colorado.edu.

representation of the target syntax, forcing this decision to percolate through the entirety of the analysis. This mutual dependence is an instance of the expression problem[16]: extending syntax requires modifying the evaluation fold, while adding semantic domains often necessitates revisiting traversal logic. State amplifies this challenge as impure statements modify the environment, which must threaded carefully through the entire interpretation while agnostic to a given domain. The difficulty is not merely compositional, but also ensuring that arbitrary combinations of syntax, control flow strategies, and abstract domains will work cohesively and independently of each other. Existing approaches address these challenges through complex abstractions that either obscure the evaluation structure or restrict expressiveness, demonstrating that modularity in abstract interpretation requires solving deep technical problems rather than applying standard design patterns.

1.1 Contribution

In our paper we present the following contributions:

- We expand on the idea of cumulative abstract semantics as a new manner to craft stateful interpreters in Section 2.
- We give a relationship between cumulative abstract semantics and traditional interpreters, offering a recipe for how to use them in Section 3.
- We demonstrate two different implementations: one in Lean and another in Effekt, and compare the two approaches in Section 4.

2 Overview

Expanding on the previous work of Lueker et al. [8] we demonstrate the core theory behind cumulative abstract semantics, without the previously assumed reliance on an a priori effect system. To provide an intuition preliminary definitions of cumulative abstract semantics, we illustrate iterations on evaluation of a simple language. We begin with an arithmetic language, to which we add conditionals and variables. Consider the following small arithmetic expression language:

$$\begin{array}{lll} \textit{Expressions} & e ::= \mathtt{cst}(n) & \text{integer constants} \\ & \mid e_1 + e_2 & \text{addition} \end{array}$$

While the paradigm of cumulative semantics is flexible enough to work across semantic representations, such as big step or denotational, this example demonstrates a piecewise denotational implementation of the type $[\![\cdot]\!]_0 : n$. This interpreter is dubbed *monolithic* because it has no options for extension without rewriting the evaluator, while mapping expressions to concrete values through induction on the syntax.

$$[\![\mathtt{cst}(n)]\!]_0 = n$$

$$[\![e_1 + e_2]\!]_0 = [\![e_1]\!]_0 + [\![e_2]\!]_0$$

2.1 Domain Parameterization via Introduction Interfaces

While the concrete interpreter is correct and complete, it lacks modularity. The evaluation logic is tightly coupled to the concrete integer domain, making it difficult to reuse any components of this interpreter for abstract analysis or alternative semantic interpretations. To address this limitation we refactor the evaluator over an arbitrary domain, $\mathcal{d}$. We also introduce the concept of *introduction* interfaces, which provide domain specific semantics to an evaluator. We define two introduction interfaces for our language, $c\grave{s}t$ and $\grave{+}$, for representing integer literals and addition in some abstract domain $\mathcal{d}$. The resulting *unsubstantiated* interpreter, $\breve{[\![\cdot]\!]}_1 : \mathcal{d} \setminus \langle c\grave{s}t, \grave{+} \rangle$, operates over an abstract domain by delegating all domain-specific operations to these handlers. To indicate which handlers still need substantiation, we use the with the $\backslash \langle\rangle$ syntax borrowed from the Koka [7] programming language.

$$\llbracket \breve{\texttt{cst}}(n) \rrbracket_1 = \grave{cst}(n)$$

$$\llbracket e_1 \,\breve{+}\, e_2 \rrbracket_1 = \llbracket \breve{e_1} \rrbracket_1 \,\dot{+}\, \llbracket \breve{e_2} \rrbracket_1$$

Introduction interfaces correspond to their syntax, so for every syntax in the language there is an introduction interface for it. By providing witnesses for both of these interfaces, we can instantiate this evaluator for any abstract domain. Here is a substantiation in the concrete domain (left) and in the interval domain (right):

$$\grave{cst}(n) := n \qquad\qquad \grave{cst}(n) := [n, n]$$

$$n_1 \,\dot{+}\, n_2 := n_1 + n_2 \qquad\qquad [l_1, u_1] \,\dot{+}\, [l_2, u_2] := [(l_1 + l_2), (u_1 + u_2)]$$

2.2 Abstract Domain Operators

We now extend the language with conditional expressions and booleans:

$$e ::= \quad \cdots \mid \texttt{if } (e_1)\ e_2 \texttt{ else } e_3 \qquad \text{conditionals}$$

The C-style conditional construct $\texttt{if } (e_1)\ e_2 \texttt{ else } e_3$ evaluates e_1 and based on whether the result is non-zero will evaluate e_2 or e_3 respectively. To support the abstract domain implementations we introduce the notion of *lowering* interfaces, which operate purely on domain elements and are defined at whim. These manifest themselves as standard abstract domain operators like $\grave{assume}$, $\grave{assumen}$, and $\grave{\sqcup}$. The two assume handlers evaluate the first value (or its negation) and based on the truth value in that domain, will return the second value or $\bot$. The join operator behaves as expected. The power of lowering interfaces lies in their composability and syntax independence. They define abstract operations that can be combined to express complex domain transformations while remaining agnostic to the specific control flow patterns that invoke them. A new interpreter of the type $\llbracket \breve{\cdot} \rrbracket_2 : \mathcal{d} \backslash \left\langle \grave{cst}, \dot{+}, \grave{if} \right\rangle$ leveraging these interfaces is defined as follows. Note how we are able to reuse our previously defined introduction interfaces, but not the old evaluator. This evaluator is unsubstantiated, but is universal for flow-sensitive abstract interpretation, being parameterized by the domain and its introduction and lowering handlers.

$$\llbracket \breve{\texttt{cst}}(n) \rrbracket_2 = \grave{cst}(n)$$

$$\llbracket e_1 \,\breve{+}\, e_2 \rrbracket_2 = \llbracket \breve{e_1} \rrbracket_1 \,\dot{+}\, \llbracket \breve{e_2} \rrbracket_1$$

$$\llbracket \texttt{if } (e_1)\ \breve{e_2} \texttt{ else } e_3 \rrbracket_2 = \grave{if}\ (\llbracket \breve{e_1} \rrbracket_2)\ \llbracket \breve{e_2} \rrbracket_2\ else\ \llbracket \breve{e_3} \rrbracket_2$$

$$\grave{if}\ (n_1)\ n_2\ else\ n_3 := \grave{assume}(n_1) n_2 \,\grave{\sqcup}\, \grave{assumen}(n_1) n_3$$

2.3 Control Flow Separation via Elimination Handlers

However, our evaluator $\llbracket \breve{\cdot} \rrbracket_2$ still bakes in a fixed control flow pattern for all constructs. We must expose the evaluation structure itself through *elimination* handlers, which eliminate the source syntax and access the interpretation ecosystem that is introduction and lowering interfaces. This gives the elimination handlers the opportunity to directly call introduction handlers, or modify the syntax and pass the new expression to a different elimination handler (through the evaluation function). These handlers must have access to the evaluation function, `rec`. We define elimination handlers in an abstract interpretation style for our three expression types as follows:

$$\acute{cst}(n) := \grave{cst}(n)$$

$$e_1 \mathbin{\acute{+}} e_2 := \mathtt{rec}(e_1) \mathbin{\grave{+}} \mathtt{rec}(e_2)$$

$$\acute{if}\ (e_1)\ e_2\ \mathit{else}\ e_3 := \grave{if}\ (\mathtt{rec}(e_1))\ \mathtt{rec}(e_1)\ \mathit{else}\ \mathtt{rec}(e_1)$$

With these handlers we define a new evaluator of the type: $[\![\breve{\cdot}]\!]_3 : \mathscr{d}\backslash\langle\grave{i},\acute{e}\rangle$. For brevity's sake we transition to using $\grave{i}$ and $\acute{e}$ to indicate that the introduction handlers or elimination handlers respectively are unsubstantiated.

$$[\![\breve{\mathtt{cst}(n)}]\!]_3 = \acute{cst}(n)$$

$$[\![e_1 \mathbin{\breve{+}} e_2]\!]_3 = e_1 \mathbin{\acute{+}} e_2$$

$$[\![\mathtt{if}\ (e_1)\ \breve{e_2}\ \mathtt{else}\ e_3]\!]_3 = \acute{if}\ (e_1)\ e_2\ \mathit{else}\ e_3$$

This interpreter delegates all control flow to the elimination handlers, making it universal to all analyses. In the concrete domain we could provide an elimination handler like the following that implements short circuiting logic, disregarding the introduction handler in the process:

$$\acute{if}\ (e_1)\ e_2\ \mathit{else}\ e_3 := \ \mathtt{if}\ \ (\ \mathtt{rec}(e_1) \neq 0)\ \ \mathtt{then}\ \mathtt{rec}(e_2)\ \mathtt{else}\ \mathtt{rec}(e_3)$$

2.4 State Threading

Until now our expression language has been pure, with no concept of state. With the following we add global state to our language:

$$\begin{array}{llll} e ::= & \cdots \mid & \mathtt{var}(x) & \text{variable recall} \\ \mid & & x \mathrel{\mathtt{:=}} e_1 & \text{assignment} \end{array}$$

With these new expressions, we must implement corresponding elimination handlers and introduction interfaces: $\acute{var}$, $\acute{:=}$, $\grave{var}$, and $\grave{:=}$. These operations introduce the need for a notion of state, which we intentionally will keep generic and refer to as σ. We need to modify the return type of our new evaluator, giving us a new unsubstantiated evaluation function of the type: $[\![\breve{\cdot}]\!]_4 : \sigma \longrightarrow \sigma \times \mathscr{d}\backslash\langle\grave{i},\acute{e}\rangle$. For simplicity we model this interpretation as a tuple of state and values, however, different approaches such as monadic state are equally valid.

$$[\![\breve{\mathtt{cst}(n)}]\!]_3 = \acute{cst}(n)$$

$$[\![e_1 \mathbin{\breve{+}} e_2]\!]_3 = e_1 \mathbin{\acute{+}} e_2$$

$$[\![\mathtt{if}\ (e_1)\ \breve{e_2}\ \mathtt{else}\ e_3]\!]_3 = \acute{if}\ (e_1)\ e_2\ \mathit{else}\ e_3$$

$$[\![\breve{\mathtt{var}(x)}]\!]_3 = \acute{var}(x)$$

$$[\![x \mathrel{\breve{:=}} e_1]\!]_3 = x \mathrel{\acute{:=}} e_1$$

This evaluation function (like $\breve{[\![\cdot]\!]}_2$) is simply a fold over the syntax of the language. With this, we can choose how to thread state as demonstrated by the following elimination handlers for concrete interpretation:

$$\acute{cst}(n) := \lambda\sigma.\left\langle \sigma, \grave{cst}(n) \right\rangle$$

$$e_1 \mathbin{\acute{+}} e_2 := \lambda\sigma.\left\langle \sigma'', \texttt{rec}(e_1)\sigma \mathbin{\grave{+}} \texttt{rec}(e_2)\sigma' \right\rangle$$

$$\acute{if}\ (e_1)\ e_2\ else\ e_3 := \grave{if}\ (\texttt{rec}(e_1))\ \texttt{rec}(e_1)\ else\ \texttt{rec}(e_1)$$

$$\acute{var}(x) := \lambda\sigma.\sigma \vdash \grave{var}(x)$$

$$x \mathbin{\acute{:=}} e_1 := \lambda\sigma.\sigma' \vdash x \mathbin{\grave{:=}} \texttt{rec}(e_1)\sigma$$

The evaluator for expressions and statements now operates in the context of these state-threading effects, allowing environment operations to be handled modularly. By keeping the state generic we allow for both relational and non relational domains. With the state threading we also gain the same accumulation of handlers that we did before state. We (naturally) had to reconstruct the elimination handlers with the addition of state, but got to keep all of our introduction handlers. Via this iterative decoupling of syntax and domain representation we arrive at a generic evaluation function that servers as a fully parametric abstract evaluation engine. One that can be substantiated for any abstract domain regardless of the directionality of control flow.

3 Technical

In this section we seek to formalize the definition of cumulative abstract semantics. To do this we first compare them with standard abstract interpretation strategies. The connection between the two approaches is formally characterized. We begin to explore the potential for automatic generation of interfaces and handler signatures by identifying patterns in the types of program nodes.

3.1 Formalization of Cumulative Abstract Semantics

The evaluation of a specific node follows the general path of $[\![\cdot]\!] \rightsquigarrow elim \rightsquigarrow intro$. An elimination or introduction interface is chosen based off of matching type signatures. This means eval is running, hits a node and asks the interpreter to find a way to fulfill the elim request. We continue to borrow Koka's notation of $\breve{eval}\ :\ \backslash\langle \acute{e}, \grave{i} \rangle$ to indicate this unsubstantiated interpreter needs to fill some elimination $\acute{e}$ calls and some introduction $\grave{i}$ calls. By applying a specific set of elimination and introduction handlers, we *substantiate* it, in other words fill out the missing implementation. The application looks like $\acute{E} \circ \grave{I}(\breve{eval}\langle \acute{e}, \grave{i} \rangle) = eval$. It is possible to incrementally apply them as well: $\acute{E}(\breve{eval}\langle \acute{e}, \grave{i} \rangle) = \breve{eval}\langle \grave{i} \rangle$, but this interpreter is still not fully substantiated, and thus unexecutable. The relationships among standard interpretation techniques are illustrated in Figure 1. The topmost, and most abstract, layer is an unsubstantiated interpreter: the simple fold over the syntax. This interpreter is not directly executable because it contains two classes of unresolved references: elimination ($\acute{e}$) and introduction ($\grave{i}$).

By applying a combination of elimination handlers (E) and introduction interfaces (I), we obtain a substantiated interpreter for a given abstract domain. Notably, by varying the elimination handlers while fixing the introduction handler, we can derive analyses that realize different control-flow directions within the same abstract domain. Although the figure explicitly depicts only forward and backward control flow, this approach generalizes to other forms of control-flow variation (e.g. flow sensitivity).

$$\begin{tikzcd}
 & & & \check{\mathtt{eval}}: \mathscr{d} \setminus \langle \acute{e}, \grave{i} \rangle \arrow[dl, "E"'] \arrow[dr, "E"] \arrow[dd, "E \circ I"] \arrow[ddll, bend right, "E \circ I"'] \arrow[ddrr, bend left, "E \circ I"] & & & \\
 & & \overleftarrow{\mathtt{eval}} : \mathscr{d} \setminus \langle \grave{i} \rangle \arrow[dl, "I"'] & & \overrightarrow{\mathtt{eval}} : \mathscr{d} \setminus \langle \grave{i} \rangle \arrow[dl, "I"'] \arrow[dr, "I"] & & \\
\cdots & \overleftarrow{\mathtt{eval}} : \hat{v} & & \overrightarrow{\mathtt{eval}} : v & & \overrightarrow{\mathtt{eval}} : \hat{v} & \cdots
\end{tikzcd}$$

Fig. 1. Relationship between cumulative semantics, extensible domain, and monolithic interpreters. The symbol $\mathscr{d}$ indicates an arbitrary, unspecified domain, while v and $\hat{v}$ indicate the concrete or a specific abstract domain respectively.

If we apply only elimination handlers, we obtain an interpreter with a fixed control-flow strategy that remains extensible with respect to the abstract domain. This layer closely resembles typical modular abstract interpreters, such as Sturdy, which support extensibility in the abstract domain but require substantially more effort to adapt or redesign the direction of control-flow [5]. Once these interpreters are further instantiated by introduction handlers, they become fully substantiated and executable.

3.2 Patterns for Signatures of Handlers

So far we have identified three classes of analysis components: elimination, introduction, and lowering. We defined both handler and semantic interfaces as typed, promised computations, fulfilled by either a substantiated handler or witness. The distinction between handlers and interfaces is that handlers have access to the `rec` continuation. These can be implemented with a variety higher ordered programming paradigms, such as through an algebraic effect and handler system, or through type classes. For each syntax there is always a corresponding pair of elimination handlers and introduction interfaces. Lowering interfaces are created whenever necessary or desired; because these are not tied to the language, there is freedom to define them however one wants. The other two forms of computational signatures are more formulaic, which is intentional for the purpose of automatic code generation. The goal of this framework is that given a grammar, the elimination and introduction interfaces will be automatically generated alongside the unsubstantiated evaluation function, providing a full, well-typed template for an analysis engine. From there the developer will implement handlers and witnesses as they see fit, reusing from their library of accumulated semantics when redundancies are encountered.

Elimination handler signatures have three components. They take a function for evaluation (explicitly or implicitly) that gives them the access to the power of recursion. They then take all of the same parameters as the associated program syntax does. And finally a state and the return value of the interpretation. Consider an `if` syntax node which has three parameters and the given eval function type:

$$\mathtt{if} : \mathit{Expr} \rightarrow \mathit{Stmt} \rightarrow \mathit{Stmt} \rightarrow \mathit{Stmt}$$

$$\mathit{eval} : \mathit{Prog} \rightarrow \sigma \rightarrow \sigma \times \delta$$

With these, the signature of the elimination handler would be the following:

$$\acute{if} : (Prog \to \sigma \to \sigma \times \delta) \to Expr \to Stmt \to Stmt \to \sigma \to \sigma \times \delta$$

This pattern holds true for all elimination handlers. One might be tempted to define one pattern for introduction interfaces as well. However, this design would conflict with the ethos of cumulative semantics. Exposing the entire state when, for example, defining the introduction witnesses for integer addition is conceptually inappropriate: what role would the global state play in determining the sum of two operands, and how would the addition operator decide which version of the state to propagate? Such concerns are questions better suited for control flow operators and their elimination handlers. Granting introduction handlers unrestricted access to state would erode the deliberate separation between elimination and introduction operators, thereby undermining the central purpose of elimination handlers: to determine how state is threaded through the program. A binary operator, at the introduction level, should not depend on or manipulate the global state. Instead, our framework introduces three canonical categories of program nodes, each with distinct interface patterns.

Computational nodes correspond to purely computational expressions (such as binary operators). They are purely mathematical expressions that simply need to be computed, and their introduction handlers operate solely on values from the semantic domain and do not manipulate state. Their type is then a δ for each program node in the syntax, and return δ. By not involving state we also gain the benefit of being able to reuse the introduction witness in different versions of evaluation , as we saw in Section 2.4.

State nodes are where we cannot make this separation. In a forwards, non-relational analysis, a 'get' operation does not modify the state; it only observes it. In a backwards analysis, however, a 'get' affects the abstract state. Thus, state-oriented nodes must combine both capabilities: producing values and potentially transforming the state, so their return type is always the return type of the program. Like computations, all program nodes in their parameter list are transformed into domain values, and have the same return type.

Control flow nodes (such as 'if' and 'while') present a choice for the developer, namely the semantic requirements of the domain that they are implementing. If they are implementing a denotational semantics, then the control flow nodes deal with denotational-semantic-style state transformations. Their elimination handlers are responsible for deconstructing the syntax into state transformers. Their introduction witnesses then thread the state through and dictate how to merge various transformations through joining, fix point iteration, widening, etc. This results in their introduction witnesses being passed a state transformation for each program node, and returning a state. If they are implementing a big step semantics then every program node becomes the return type of the evaluation, with control flow decisions residing in the elimination handlers and the introduction interfaces becoming trivial. Other semantic styles are as of yet still unexplored.

4 Evaluation

In this section, we examine two case studies of the cumulative abstract semantics: one realized in a native language with built-in support for algebraic effects and lexical handlers, Effekt [13], and another realized via type classes in a more conventional functional language, Lean. Both implementations adhere to an identical conceptual scheme and yield extensionally equivalent analyses, thereby enabling a comparison in terms of runtime performance and relative implementation complexity.

Our target language is a Python-like language, whose syntax is presented in Figure 2.

Identifiers	x		
Numbers	n	$\in$	$\mathbb{R}$
Booleans	b	$:=$	true | false
Concrete Values	v	$:=$	$n \mid b \mid ()$
Operators	op	$:=$	$+ \mid - \mid \div \mid * \mid == \mid > \mid < \mid$ and | or
Expressions	e	$:=$	$\mathrm{cst}(v) \mid \mathrm{var}(x) \mid \mathrm{binop}(e, e, op) \mid \mathrm{neg}(e)$
Statements	s	$:=$	$\mathrm{skip} \mid \mathrm{assign}(x, e) \mid \mathrm{if}(e, s, s) \mid \mathrm{seq}(s, s) \mid \mathrm{while}(e, s)$
Program node	p	$:=$	$s \mid e$

Fig. 2. A python-like AST

4.1 Typeclass Based

The cumulative semantics can be defined with type classes relying on the language's type resolution capabilities to select appropriate instances at compile time. To clarify nomenclature, an interface in this version will be a type class, while a witness will be an instance of that class. We select Lean4 to make this interpreter for several reasons: It is a clean, modern functional programming language that is growing in popularity, and compiles to performant C binaries. Additionally Lean provides the ability to reason about programs. The capability to carry proofs with our semantics opens the door to reusable soundness proofs, among others. Another practical advantage is with Lean one can get develop-time error checking if there is not an appropriate type class instance, eliminating any runtime errors where these handler may have been missing only to be discovered during testing.

4.1.1 Type Class Architecture. The unsubstantiated evaluator establishes the foundation for all subsequent instantiations. Its signature contains the set of elimination handlers that must be provided to produce an executable interpreter:

```
partial def eval {σ δ : Type}
  [CstE σ δ] [VarE σ δ] [BinopE σ δ] [NegE σ δ]
  [SkipE σ δ] [AssignE σ δ] [IfE σ δ] [SeqE σ δ] [WhileE σ δ]
  : Prog → σ → (δ × σ)
```

This evaluator remains parametric over both the state type σ and the abstract domain δ. The interpreter implements a state-passing denotational semantics, reflected in its return type of $\sigma \rightarrow \delta \times \sigma$. This function itself performs a straightforward structural recursion over the syntax, delegating to the listed typeclasses. The expression case shows this clearly:

```
| (.Exp e) => match e with
  | .Cst v => CstE.cst eval v
  | .Binop e1 e2 op => BinopE.binop eval e1 e2 op
  | .Neg e => NegE.neg eval e
```

Each syntatic construct invokes its corresponding elimination handler, passing the evaluator as the first argument to enable recursion. By passing itself, it maintains the same scope that it was called in, rather than defaulting to the global scope. This pattern extends to all language constructs.

4.1.2 Handler Signature Patterns. The type signature for handler pairs follow systematic patterns as established in Section 3.2. We examine representative examples from each of the three categories to illustrate how these patterns manifest via typeclasses.

Computational nodes, like binary operations, operate purely on domain values without modifying state. In Lean the signature looks like the following:

```
class BinopE (σ δ : Type) where
  binop : (Prog → σ → (δ × σ)) → Expr → Expr → Op → σ → (δ × σ)
class BinopI (δ : Type) where
  binop : δ → δ → Op → δ
```

The elimination handler(BinopE) accepts a continuation for evaluation, the raw syntactic parameters, and the current state. It promises to return an updated state-value pair. The introduction interface operates only on domain values. Each expression in its parameter list becomes a domain value δ, while the operator remains unchanged.

Syntax nodes that interact with state are demonstrated by the *assign* statement:

```
class AssignE (σ δ : Type) where
  assign : (Prog → σ → (δ × σ)) → Ident → Expr → σ → (δ × σ)
class AssignI (σ δ : Type) where
  assign : Ident → δ → σ → (δ × σ)
```

Unlike computational nodes, the introduction interface for assignment must accept a state, and returns a state-value tuple. Similar to computation nodes, state nodes transform expressions into domain values.

Control flow nodes transform statements into first class state transformations. This is why we refer to this implementation as somewhat denotational: because the evaluation is broken into state transformations that are then threaded together. Their introduction handlers accept the initial state, thread it through evaluation, returning only a state. The *sequence* handler illustrates this approach:

```
class SeqE (σ δ : Type) where
  seq : (Prog → σ → (δ × σ)) → Stmt → Stmt → σ → (δ × σ)
class SeqI (σ δ : Type) where
  seq: σ → (σ → σ) → (σ → σ) → σ
```

The introduction interface receives the initial state along with two state transformers, representing the semantic functions for each statement. This design allows the introduction witness to determine how state flows through sequential compositions: whether strictly left-to-right, with potential short-circuiting, or through some other control flow strategy—without being coupled to the specific syntax of the statements involved.

4.1.3 Concrete Evaluation. Substantiating the interpreter entails providing an instance for each required typeclass. We again examine binary operations, through the implementation of their witnesses:

```
instance {σ δ : Type} [BinopI δ] : BinopE σ δ where
  binop eval e1 e2 op ρ :=
    let (v1, ρ') := eval (.Exp e1) ρ
    let (v2, ρ'') := eval (.Exp e2) ρ'
    (BinopI.binop v1 v2 op, ρ'')

instance : BinopI ConcreteValue where
  binop v1 v2 op := match (v1, v2, op) with
    | (.Num n1, .Num n2, Op.Plus) => .Num (n1 + n2)
    | (.Num n1, .Num n2, Op.Minus) => .Num (n1 - n2)
    ...
```

The elimination instance remains generic over some state σ and domain δ, while the introduction instance is tied to the *ConcreteValue* domain. This genericity allows the same *BinopE* handler to be used across many domains simply by providing a new *BinopI* instance. In a similar fashion, we implement the Assign handlers:

```
instance {σ δ : Type} [AssignI σ δ] : AssignE σ δ where
  assign eval x e ρ :=
    let (v, ρ') := eval (.Exp e) ρ
    AssignI.assign x v ρ'

instance {σ δ : Type} [Inhabited δ] [Put σ δ] : AssignI σ δ where
  assign x v ρ := (default, Put.put x v ρ)
```

The elimination handler, as typical, decides the threading of state by evaluating the right hand side, then passing that state to the introduction handler. The introduction handler then modifies this state with the Put type class, which encapsulates the actual state update operation. This type class can be thought of as a lowering handler, preventing the need for reimplementation of assignment logic. This additional abstraction permits flexible state representations like simple maps, more sophisticated scoped states, or even relational states. The *Inhabited* type class, which allows the elimination handler to return a default value (unit) without knowing the domain, can also be thought of as a lowering handler. Something of note here is these specific instances are generic over any domain, and so could be reused for abstract interpretation.

Control flow handlers demonstrate how state transformations are assembled from syntax and packaged. This is clear in the conditional handler pair:

```
instance {σ δ : Type} [Inhabited δ] [IfI σ δ] [Assume σ δ] : IfE σ δ
    where
  if_ eval e t f ρ :=
    let (v, ρ') := eval (.Exp e) ρ
    let tk : σ → σ := λ σ => Assume.assume v (eval (.Stm t) σ).snd
    let fk : σ → σ := λ σ => Assume.assumef v (eval (.Stm f) σ).snd
    (default, IfI.if_  ρ' tk fk)

instance {σ δ : Type}  [Join σ] : IfI σ δ where
  if_ ρ tk fk :=
    (tk ρ) ⊔ (fk ρ)
```

The elimination handler constructs two state transforms: one that assumes the guard condition and evaluates the true branch, and another that assumes the negation and evaluates the false branch. It also evaluates the guard condition. With the new state and the two state transforms, it calls the introduction handler which determines how to join the two. This implementation shows a join of the two branches, giving us flow sensitive analysis. If we wanted to do a flow insensitive analysis it would be as simple as supplying a different elimination handler that has no assume calls in it.

The while loop presents the most complex control flow scenario, requiring fixpoint iteration:

```
partial def lfp {α : Type} (f : α → α) (x : α) [Bottom α] [LatOrder α] :
    α :=
  let rec aux (current : α) :=
    let next := f current
    if next ⊑ current then current else aux next
  aux x
```

```
instance {σ δ : Type} [Assume σ δ] [WhileI σ δ] [Inhabited δ] : WhileE σ
    δ where
  while_ eval e body ρ:=
    let k : σ → σ := fun σ =>
      let (v, σ') := eval (.Exp e) σ
      Assume.assume v (eval (.Stm body) σ').snd
    let invariant := WhileI.while_ ρ k
    let final := eval (.Exp e) invariant
    (default, Assume.assumef final.1 final.2)

instance{σ δ: Type} [Bottom σ] [LatOrder σ]: WhileI σ δ where
  while_ ρ cont := lfp cont ρ
```

The elimination handler constructs a state transform for one iteration of the loop: evaluate the guard, then execute the body and return the result with an assume. The introduction handler applies fixpoint iteration to this transform, starting from the inital state and continuing until convergence. The least fixpoint implementation requires that the state forms a lattice with bottom and ordering operations, which are both constraints captured in the type classes *Bottom* and *LatOrder*. This design successfully separates the syntactic structure of loops from the semantic strategy for computing their effect, allowing the same elimination handler to support different fixpoint strategies or widening operators through alternative introduction instances.

4.2 Effect Based

4.2.1 Handlers and Interfaces. As explored in our introductory work on Cumulative Semantics, the theory is cleanly expressed with algebraic effects and handlers [11]. The key insight being that effect handlers offer an implicit resume, eliminating the need for passing a continuation with every recursive call. Any language with an effect system supporting generic effect interfaces and multiple resumptions can express the ethos of cumulative semantics.

Previously we implemented our analyzer with Koka, but found that due to the need for several handlers with multiple resumptions performance dropped significantly. The Koka compiler is not optimized for such a task. Instead, we decided to implement with Effekt [13] due to its efficient compilation of effect handlers to CPS style code.

Due to the high level of abstraction provided by algebraic effects, implementing an analyzer that adheres to cumulative semantics with a direct style is extremely natural.

The subset of Python's Syntax is represented as an ADT named *Prog*, of two smaller ADT's. One for expressions, and one for statements. Elimination handlers are defined as an effect interface, parametrized by a generic domain D. They take in state, a piece of syntax, and or an *Ident*, represented by a *String*. They return a tuple of *Prog* and state, signaling that their handlers reduce or eliminate syntax while updating state. Due to the branching nature of syntax (*if*, *seq*, etc..) elimination handlers utilize multiple resumptions to pass each syntactic branch back into the evaluation function. As a result the elimination effects are grouped into a single interface, ensuring that when a new, processed syntax is resumed, the handler of this syntax is known immediately.

```
interface Elimination[D] {
  def varE(st: State[D], x: String): (Prog, State[D])
  ...
  def seqE(st: State[D], e1: Stmt, e2: Stmt): (Prog, State[D])
}
```

Additionally introduction interfaces are expressed with effects. Algebraic effects are powerful enough to emulate type classes. They are parameterized by the same generic domain type D, and will only resume once, allowing them to be defined individually. Introduction interfaces are tasked with introducing new values and states of a syntactic domain, as demonstrated by their type signatures.

```
effect plusI[D](st: State[D], d1: D, d2: D): (D, State[D])
effect asgnI[D](st: State[D], x: String, v: D): (D, State[D])
```

4.2.2 Unsubstantiated Evaluation. Once the elimination and introduction effects have been typed it is possible to create an unsubstantiated evaluation function. The type signature denotes the presence of the *Elimination* effect interface, as well as any introduction handlers for constant domain values. These can either be expressed as elimination handlers or directly as introduction handles as shown below. The *eval* function takes in a program and initial state, returning a generic domain value and state parameterized by that generic domain. The *do* notation calls an effect, the resumption of which will return a processed piece of syntax back into the recursive *eval* function call.

```
def eval[D](prog: (Prog, State[D])): (D, State[D]) /{
    intI[D], boolI[D], unitI[D], varI[D],
    Elimination[D] } =
  val (e, st) = prog
  e match {
    case E(e) => e match
      case Plus(e1, e2) => eval(do plusE(st, e1, e2))
      ...
```

4.2.3 Concrete Evaluation. There are two steps to achieving an executable evaluation. First we must substantiate the elimination Effects for concrete values. Then intro effects can be substantiated to produce values in the domain of the current evaluation. The syntax below expresses a function that wraps an effectful one, providing handlers to a specified effect. This wrapper can introduce new effects, even if they are called by another effect's handlers. Each elimination handler calls its corresponding introduction interface, which we account for in the type of the function.

```
def match_to_fold
  {prog: => (Val, State[Val]) / {Elimination[Val]}}:
  (Val, State[Val]) / {
    ...
    ifI[Val], whileI[Val], seqI[Val] } =
    { try {prog()}
      with Elimination[Val] {
        def plusE(st, e1, e2) =
          val (v1, st1) = resume((E(e1), st))
          val (v2, st2) = resume((E(e2), st1))
          do plusI(st2, v1, v2)
        ...
```

While the return type of the handler must match the type passed to its *resume*, it can be thought of as a returning a continuation to the recursive call of *eval*. Once evaluation has reached a leaf of the AST,the elimination handler delegates returning domain values to its introduction interface, passing in any updated values and states.

Evaluation becomes executable once all elimination and introduction effects are handled. Now with the elimination effects handled, a wrapper to give semantics to introduction interfaces fully substantiates *eval* is instantiated.

```
def run(prog: Prog, init: State[Val]): (Val, State[Val]) = {
  try {
    with match_to_fold()
    eval((prog, init))}
  ...
  with ifI[Val] {(stT, stF, g, t, f) =>
    g match {
      case B(true) => resume((t, stT))
      case B(false) => resume((f, stF))
      case _ => resume((U(), stF))
```

Introduction handlers choose the values and states a syntactic node should return. This behavior pins the domain of evaluation and directionality of control flow. As seen in the witness (effect handler) for if, the witness receives two states for each branch of the if, the value of evaluating the guard condition, the true branch, and the false branch. It then dictates which branch's state should be returned based on the guard value. Herein lies the value for relational analysis. The introduction handler can decide to combine the states of diverging branches instead of choosing only one.

The Effekt implementation of cumulative semantics is very explicit and due to the direct style facilitated by algebraic effects, the theory is not lost within the implementation. Both elimination handlers and introduction witnesses are expressed with the same construct creating a congruence that allows the developer to focus more on the theory of cumulative semantics over the implementation itself.

4.3 Comparison of the Two Implementations

To evaluate performance, we executed a simple while-loop program with varying iterations using the native Python interpreter, as well as the Lean4 and Effekt interpreters. Execution times were measured with the hyperfine benchmarking tool. The results are reported in Table 1. Although both cumulative semantic implementations introduce a non-negligible overhead, their performance remains within one order of magnitude of the Python interpreter, indicating reasonably good scaling behavior. The nearly constant run time of the Lean4 implementation for small input sizes suggests a substantial fixed overhead, likely associated with type-class resolution, but the subsequent growth in running time is moderate. In contrast, the Effekt-based implementation exhibited markedly superior performance, not yet exploding.

5 Conclusion

In this paper, we have developed an intuitive account of the purpose of elimination, introduction, and lowering instances/handlers. We subsequently formalized and refined these intuitions, presenting a systematic methodology for constructing the corresponding semantics. This methodology was then instantiated in two distinct programming languages with slightly different design structures. The resulting variations illustrate the expressiveness and flexibility of the proposed framework, demonstrating that it can be used in multiple ways to accommodate different interpretation styles. Our performance evaluation indicates that the framework incurs some overhead; however, in relative terms this overhead remains moderate and does not render the approach impractical.

Cumulative abstract semantics, as presented here, are subject to several limitations, the most significant being the lack of comprehensive validation. In particular, there is currently no complete

Implementation	Iterations (N)	Time (Mean ± σ)	User	System	Range
Effekt LLVM	100,000	64.8 ms ± 1.2 ms	64.3 ms	0.4 ms	61.5 ms . . . 66.8 ms
	10,000	7.5 ms ± 0.1 ms	7.2 ms	0.2 ms	7.2 ms . . . 8.3 ms
	1,000	1.7 ms ± 0.1 ms	1.4 ms	0.2 ms	1.6 ms . . . 2.4 ms
Lean4	100,000	126.7 ms ± 326.8 ms	20.1 ms	5.0 ms	22.2 ms . . . 1056.7 ms
	10,000	22.1 ms ± 0.3 ms	18.0 ms	1.9 ms	21.5 ms . . . 23.2 ms
	1,000	22.1 ms ± 0.3 ms	18.0 ms	1.9 ms	21.4 ms . . . 22.8 ms
Python	100,000	17.4 ms ± 1.5 ms	13.1 ms	3.1 ms	16.6 ms . . . 31.3 ms
	10,000	14.1 ms ± 0.3 ms	9.9 ms	3.1 ms	13.6 ms . . . 16.3 ms
	1,000	13.7 ms ± 0.3 ms	9.6 ms	3.1 ms	13.3 ms . . . 14.9 ms

Table 1. Benchmark Comparison of Loop Iterations

implementation for abstract interpretation; existing implementations are restricted to concrete interpreters.

To address these limitations, we envision several directions for future work. First, we plan to construct abstract interpreters in both object languages, supporting interval analysis and backward symbolic weakest-precondition analysis. In addition, we aim to establish correctness results in the presence of monadic state, which would in turn enable a principled treatment of error handling via monads and related constructs. Finally, it would be valuable to investigate, within Lean, methodologies for structuring and transporting soundness proofs so that they can be reused across different sound components.

6 Related Works

Modular Abstract Interpreters. The isolation of complex components in programming language definitions has been explored extensively in prior work. Horn and Might [3] introduced a technique for abstracting the program store, thereby transforming an infinite execution tree into a finite state graph. This foundational work was extended by Sergey et al. [14], who employed a monadic store to decouple store execution from program execution. This modularization simplified the interpretation process and enabled independent reasoning about specific components. Through the use of monadic state, they achieved orthogonalization of semantics and facilitated reusability across distinct interpreters. Our work adopts a similar approach by factoring state into a set of lowering handlers, which permits developers to define state representations flexibly and reuse them across multiple instantiations. This design does not constrain implementations to exclusively monadic state representations, permitting any style of state encapsulation desired. We extend this technique further by modularizing the program's interaction with state, such that both state representations and state access patterns (retrievals and updates) can be handled in any way. In their Sturdy framework, Keidel et al. [4, 5] employed arrow transformers to achieve gradual, automatic soundness proofs. While their approach provides soundness guarantees compositionally, it is limited to non-relational domains and necessitates a generic interpreter implementation for each target language, offering flexibility primarily along the abstract domain axis. We pursue an additional dimension of modularity beyond Sturdy by reducing the overhead required to extend syntax and by incorporating modularity for control flow as a fundamental design principle. More recently, Michelland et al. [9] advanced monadic modularity through their I-Tree-based framework, composing state and control flow monads to establish a sound metatheory formalized in Roq (Coq). Their approach supports modular analyses such as binding-time analysis, and we pursue similar modularity objectives while aiming to reduce implementation complexity and eliminate the need for intricate transformer stacks. Monat [10] developed the MOPSA abstract interpreter built around generic Ocaml modules to provide domain generality. MOPSA represents domains as Directed Acyclic Graphs, causing evaluated expressions to cascade through each implemented domain until one can perform a meaningful analysis. Domain composition is accomplished by taking the reduced product of two domains. Although MOPSA has generic domain representations, the directionality of control flow is fixed. Our work on cumulative semantics allows for both the generalization of abstract domain and multidirectional analysis, but does not facilitate the composition of domains during a single analysis present in MOPSA.

Effect-Based Interpreters. The application of algebraic and lexically scoped effects to interpreter construction represents a relatively recent development. Reinders [12] explored this theoretical direction in preliminary work, and Bunkenburg and Wu [1] developed that theory into practice, implementing a modular interpreter using effect handlers embedded in Haskell, representing one of the earliest implementations of this approach. While their work focused on concrete interpretation of the Curry language, Curry's inherent support for non-determinism parallels the non-determinism encountered in abstract interpretation. This demonstrated the utility of effect handlers for control flow manipulation, which our work adopts and generalizes to more comprehensive abstract domains and applications beyond the Curry language. Prior to the exploration of a priori effect systems Kisylov explored simulating effects with Free Monads [6]. In his work effects are represented as free monads with effectful operations denoting their effects as a coproduct of free monads in the type signature. Many effect systems leverage similar constructs in their implementations. However, for top level use it presents the overhead of manually injecting and lifting free monads. Our Lean implementation of cumulative semantics achieved similar modularity and functionality by directly passing continuations and leveraging typeclasses, removing the need for monadic encapsulation and the plumbing that it entails.

Type Class-Based Interpretation. Type classes have long been used to modularize semantics, as introduced in Haskell by Swierstra [15]. Our work builds on the idea of tagless final interpreters, where the initial encoding is an unsubstantiated interpreter and the final encoding is a substantiated one. In place of the Haskell modules used by Carette et al. [2], we use Lean's type classes. While their approach focuses on modularizing the syntax and semantic domain, our cumulative semantics further decouples the control-flow strategy (via elimination handlers) from the domain operations.

References

[1] Niels Bunkenburg and Nicolas Wu. 2024. Making a Curry Interpreter using Effects and Handlers. In *Proceedings of the 17th ACM SIGPLAN International Haskell Symposium, Haskell 2024, Milan, Italy, September 6-7, 2024*, Niki Vazou and J. Garrett Morris (Eds.). ACM, 68–82. doi:10.1145/3677999.3678279

[2] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.* 19, 5 (2009), 509–543. doi:10.1017/S0956796809007205

[3] David Van Horn and Matthew Might. 2010. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 51–62. doi:10.1145/1863543.1863553

[4] Sven Keidel and Sebastian Erdweg. 2019. Sound and reusable components for abstract interpretation. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 176:1–176:28. doi:10.1145/3360602

[5] Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg. 2018. Compositional soundness proofs of abstract interpreters. *Proc. ACM Program. Lang.* 2, ICFP (2018), 72:1–72:26. doi:10.1145/3236767

[6] Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, Ben Lippmeier (Ed.). ACM, 94–105. doi:10.1145/2804302.2804319

[7] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. *Electronic Proceedings in Theoretical Computer Science* 153 (June 2014), 100–126. doi:10.4204/EPTCS.153.8 arXiv:1406.2061 [cs].

[8] Cade Lueker, Andrew Fox, and Bor-Yuh Evan Chang. 2025. Towards Cumulative Abstract Semantics via Handlers. arXiv:2512.10861 [cs.PL] https://arxiv.org/abs/2512.10861

[9] Sébastien Michelland, Yannick Zakowski, and Laure Gonnord. 2024. Abstract Interpreters: A Monadic Approach to Modular Verification. *Proc. ACM Program. Lang.* 8, ICFP (2024), 602–629. doi:10.1145/3674646

[10] Raphaël Monat. 2021. *Static type and value analysis by abstract interpretation of Python programs with native C libraries. (Analyse statique, de type et de valeur, par interprétation abstraite, de programmes Python utilisant des librairies C)*. Ph. D. Dissertation. Sorbonne University, Paris, France. https://tel.archives-ouvertes.fr/tel-03533030

[11] Gordon D Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* Volume 9, Issue 4 (Dec. 2013). doi:10.2168/lmcs-9(4:23)2013

[12] Jaro S. Reinders. 2023. Towards Modular Compilation Using Higher-Order Effects. In *Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands (OASIcs, Vol. 109)*, Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:9. doi:10.4230/OASICS.EVCS.2023.22

[13] Philipp Schuster, Jonathan Immanuel Brachthäuser, Marius Müller, and Klaus Ostermann. 2022. A typed continuation-passing translation for lexical effect handlers. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 566–579. doi:10.1145/3519939.3523710

[14] Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. 2013. Monadic abstract interpreters. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 399–410. doi:10.1145/2491956.2491979

[15] Wouter Swierstra. 2008. Data types à la carte. *J. Funct. Program.* 18, 4 (2008), 423–436. doi:10.1017/S0956796808006758

[16] Philip Wadler. 1998. The Expression Problem. Java Genericity mailing list. https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt