Staged Abstract Interpreters

Fast and Modular Whole-Program Analysis via Meta-programming

GUANNAN WEI, YUXUAN CHEN, and TIARK ROMPF, Purdue University, USA

It is well known that a staged interpreter is a compiler: specializing an interpreter to a given program produces an equivalent executable that runs faster. This connection is known as the first Futamura projection. It is even more widely known that an abstract interpreter is a program analyzer: tweaking an interpreter to run on abstract domains produces a sound static analysis. What happens when we combine these two ideas, and apply specialization to an *abstract* interpreter?

In this paper, we present a unifying framework that naturally extends the first Futamura projection from concrete interpreters to abstract interpreters. Our approach derives a sound staged abstract interpreter based on a semantic-agnostic interpreter with type-level binding-time abstractions and monadic abstractions. By using different instantiations of these abstractions, the generic interpreter can flexibly behave in one of four modes: as an unstaged concrete interpreter, a staged concrete interpreter, an unstaged abstract interpreter, or a staged abstract interpreter. As an example of *abstraction without regret*, the overhead of these abstraction layers is eliminated in the generated code after staging. We show that staging abstract interpreters is practical and useful to optimize static analysis, all while requiring less engineering effort and without compromising soundness. We conduct three case studies, including a comparison with Boucher and Feeley's abstract compilation, applications to various control-flow analyses, and a demonstration that can be used for modular analysis. We also empirically evaluate the effect of staging on the execution time. The experiment shows an order of magnitude speedup with staging for control-flow analyses.

CCS Concepts: • **Software and its engineering → Automated static analysis**; *Functional languages*; *Interpreters*.

Additional Key Words and Phrases: multi-stage programming, abstract interpreters, control-flow analysis

ACM Reference Format:
Guannan Wei, Yuxuan Chen, and Tiark Rompf. 2019. Staged Abstract Interpreters: Fast and Modular Whole-Program Analysis via Meta-programming. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 126 (October 2019), 32 pages. https://doi.org/10.1145/3360552

1 INTRODUCTION

Abstract interpretation as a lattice-based approach to sound static analyses was proposed by Cousot and Cousot [1977, 1979]. Based on the notion of Galois connections, an analyzer can soundly approximate concrete program behavior at runtime by computing fixed points on an abstract domain. Despite the tremendous theoretical development of abstract interpretation over the years, constructing artifacts and analyzers that perform sound abstract interpretation for modern and expressive languages is considered complicated for a long time.

Recent progress on methodologies such as Abstracting Abstract Machines (AAM) [Horn and Might 2010, 2012] uncovers an operational approach to constructing abstract interpreters. By

Authors' address: Guannan Wei; Yuxuan Chen; Tiark Rompf, Department of Computer Science, Purdue University, USA, guannanwei@purdue.edu, chen1797@purdue.edu, tiark@purdue.edu.

2475-1421/2019/10-ART126
https://doi.org/10.1145/3360552

deriving semantics artifacts for abstract interpretation from their concrete counterparts (for example, abstract machines), soundness can be easily established by examining the transformation of semantic artifacts. This systematic approach to abstraction can be tailored to different language features (such as state, first-class control, exceptions, etc.) and sensitivity analyses [Darais et al. 2015; Gilray et al. 2016a,b]. It has also been applied to various small-step abstract machines [Horn and Might 2010, 2012; Sergey et al. 2013] and big-step definitional interpreters [Darais et al. 2017; Keidel et al. 2018; Wei et al. 2018]. Based on the idea of AAM, more pragmatically, several implementation strategies in high-level functional programming languages have emerged. Such techniques include monads and monad transformers [Darais et al. 2017; Sergey et al. 2013], arrows [Keidel et al. 2018], extensible effects [Github, Inc. 2019], etc. These pure approaches provide certain benefits, including the fact that the abstract interpretation artifacts can be built in a compositional and modular fashion. Furthermore, referential transparency allows programmers to confidently reason about the correctness. The soundness of an analysis can also be proven with less effort, whether by mechanized proofs [Darais and Van Horn 2016] or paper-based proofs [Keidel et al. 2018].

However, besides the intrinsic complexity of static analysis, there are additional abstraction penalties from these high-level implementation approaches. First, similar to concrete interpreters, an abstract interpreter analyzes a program by traversing its AST. This traversal incurs interpretive overhead, such as pattern matching on the AST nodes and recursive calls on subexpressions. If this requires only a single traversal, such overhead can be negligible. Usually, however, abstract interpreters iteratively analyze and traverse an AST multiple times to reach fixed-points; the accumulated overhead is therefore magnified. Second, it is common that abstract interpreters written in pure languages make extensive use of functional encodings of side-effects to implement the semantics of interpreted languages. For example, the list monad can be used to model nondeterminism of collecting interpreters. Although such pure approaches have their merits, they are significantly slower when compared with imperative, stateful implementations. The goal of this paper is to develop an approach for implementing abstract interpreters, which allows high-level abstractions during implementation for clarity, but passes the incurred overhead at runtime for efficiency.

Roughly at the same time abstract interpretation was proposed, Futamura [1971] observed a close connection between interpreters and compilers through a hierarchy of specializations, known since then as the Futamura projections. The first Futamura projection states that specializing an interpreter to an input program removes interpretative overhead and yields an equivalent executable that runs faster. Moreover, a procedure that can specialize an interpreter to any input program is equivalent to a compiler. In recent years, the idea of Futamura projections has attracted more attention, and has been successfully applied to many real-world scenarios, such as building JIT compilers for dynamic languages [Bolz et al. 2009; Marr and Ducasse 2015] and building query compilers for databases [Essertel et al. 2018; Tahboub et al. 2018]. The question here is can we apply Futamura projections to an abstract interpreter and obtain a compiler for abstract interpretation? The solution is not obvious, as we need to specialize a nonstandard semantics that extensively uses nondeterminism, operates on abstract domains, and computes fixed-points iteratively.

In this paper, we show that the first Futamura projection can be naturally extended to abstract interpreters. We present an abstraction-without-regret framework that eliminates performance penalties for monadic abstract interpreters, while keeping the implementation as close to their conceptual model as possible. In short, we borrow ideas and techniques from multi-stage programming and embedded DSLs, and apply them to abstract interpreters: 1) To remove the overhead from interpretation and effect layers, we specialize the abstract interpreter via staging, and then generate efficient low-level code that performs the actual analysis. 2) Inspired by tagless-final interpreters [Carette et al. 2009], we construct a generic interpreter that abstracts over binding-times [Amin and Rompf 2018; Ofenbeck et al. 2017] and different semantics, which allows the staged abstract

interpreter to be derived from its unstaged counterpart. As a result, the derived staged abstract interpreter has no intrusive changes to the underlying abstract semantics, thereby preserving soundness. In this sense, our approach allows for a complete absence of regret in terms of both performance and engineering effort. We elaborate these two main ideas in detail, as follows.

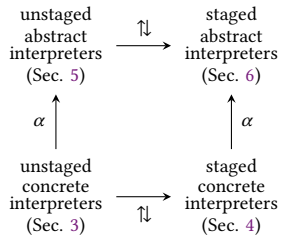

Fig. 1. The confluence of specialization and approximation.

Futamura Projections Meet Abstract Interpreters. The first Futamura projection shows that specializing an interpreter w.r.t. an input program yields an equivalent program. For instance, assume that `eval : Expr → Input → Value` is an interpreter for some language. Given a program e `: Expr`, by applying the specialization we obtain a specialized program $\texttt{eval}_e$ `: Input → Value`. The earlier approach to realizing Futamura projections is partial evaluation (PE) [Jones et al. 1993]. Offline PE relies on a separate binding-time analysis (*static* or *dynamic*), and online PE identifies the binding-times during the specialization. The underlying representation of binding-times can be viewed as a two-level semantics [Nielson 1989; Nielson and Nielson 1988, 1992] that distinguishes between compile-time computation and run-time computation. However, it is hard to precisely analyze binding-times given an arbitrary program. As an alternative approach, multi-stage programming (MSP) [Taha 1999; Taha and Sheard 1997] allows programmers to annotate the binding-times of variables and control which part of the program should be specialized. At the syntactic level, traditional MSP languages (e.g., MetaML/MetaOCaml) use quasi-quotations and escapes as annotations. However, the syntactic quasi-quotation is a weak two-level semantics, and does not always preserve the evaluation order of staged expressions. Besides the syntactic approach, staging annotations can also be purely type-based. One of the examples is the Lightweight Modular Staging framework [Rompf and Odersky 2010] in Scala, which as well provides a stronger guarantee of preserving the evaluation order by generating code in administrative normal form.

To apply the Futamura projections to abstract interpreters, we consider Darais et al. [2017]'s big-step monadic abstract interpreter as the unstaged baseline. Similar to two-level semantics [Nielson 1989], monads provide a good abstraction to hide the detail of abstract interpretation semantics. However, monads do not have a clear stage distinction. We further introduce binding-times and make the monads binding-time polymorphic. Our proposed framework adopts type-based multi-stage programming from the Lightweight Modular Staging framework, and implements the first Futamura projection of an abstract interpreter for a stateful higher-order language. By deriving staged monads that can be used to perform code generation, we obtain a staged abstract interpreter that can be used for specialization. Through staging, the generated code is specialized to the input program, and all the monadic operations are inlined and compiled down to low-level Scala code.

Generic Interpreter and Reinterpretation. Program specialization and abstract interpretation are two orthogonal concepts. To implement the confluence of them, we first construct a generic interpreter that is agnostic to both binding times and value domains used in the semantics. Later, the generic interpreter can be instantiated from these two dimensions (Figure 1):

- With a flat binding-time and concrete domains, it is an ordinary definitional interpreter based on big-step operational semantics;

- With two-level binding-times and concrete domains, it is a compiler that translates a program into another language;
- With a flat binding-time and abstract domains, it is a definitional abstract interpreter [Darais et al. 2017] that statically computes runtime properties;
- With two-level binding-times and abstract domains, it is an optimized program analyzer, but works in the fashion of compilation.

Although the four artifacts may look dissimilar at first glance, they are in fact all firmly rooted in the concrete semantics. This observation provides a way to abstract over the semantics and achieve the flexibility of reinterpreting the shared interpreter. The generic interpreter returns a value of monadic types, which vary based on semantics. The value domains of the interpreter and the effects such as state and nondeterminism can all be wrapped into this monadic type. The binding-time abstraction is represented by a higher-kinded type and injected into the monadic type. The binding-times control whether the interpreter produces values directly or generates code.

Applications and Evaluation. We evaluate the idea of staging an abstract interpreter through case studies and an empirical performance evaluation. 1) We compare our approach with abstract compilation [Boucher and Feeley 1996], an implementation technique for control-flow analyses. We show that by utilizing type-based stage annotations, we can achieve the same optimizations. Meanwhile, the analyzer does not need to change (modulo the addition of stage annotations), thereby requiring significantly less engineering effort. 2) We extend the basic staged abstract interpreter to cover different flow analyses, including a store-widened analysis, a context-sensitive analysis, and abstract garbage collection. 3) We show that staging an abstract interpreter enables modular compilation of an analysis to programs. Here we borrow the concept of modular analysis, and show that the compiled analysis is reusable. Therefore, the approach provides a modular way to create optimized analysis code by mechanized reuse of a whole-program analyzer. 4) We empirically evaluate the performance improvements gained by staging, showing an order of magnitude speedup.

Contributions. 1) Intellectually, we present a framework that naturally extends the first Futamura projection to abstract interpreters, showing a well-grounded approach to optimizing static analyses via meta-programming. 2) Practically, we show that staging an abstract interpreter is useful to improve performance and scalability of analyses by case studies and an empirical evaluation.

Organization. The paper is organized as follows:

- We begin by introducing our target language and reviewing monads in Scala, and then presenting the generic interpreter (Section 2), after which we review instantiations of concrete interpretation (Section 3) and staged concrete interpretation (Section 4).
- We present the unstaged abstract interpreter under the same framework by replacing the environment, store, and values with their abstract counterparts (Section 5). We then show that the combination of approximation and specialization, dubbed *staged abstract interpreters*, can be readily derived (Section 6). We also summarize the approach and discuss soundness properties after showing the four artifacts.
- We summarize our approach and discuss correctness (Section 7). We conduct three case studies (Section 8) and an empirical evaluation on performance improvements (Section 9). Finally we discuss related work and conclude.

2 PRELIMINARIES

In this section, we first describe the abstract syntax of the language. Then, we present the generic interpreter shared among the four different semantics, after which we instantiate the interpreter to the concrete one. It is worth noting that we choose to use Scala and a monadic style to demonstrate

this idea, though the approach is not restricted to this choice. Indeed, one can use imperative or direct-style in other MSP languages (e.g., MetaOCaml [Calcagno et al. 2003; Kiselyov 2014] or Template Haskell [Sheard and Jones 2002]) to construct such staged abstract interpreters, although the details are different.

2.1 Abstract Syntax

We consider a call-by-value λ-calculus in direct-style, extended with numbers, arithmetic, recursion, and conditionals. Other effects such as assignments can also be supported readily. Since we are interested in analyzing the dynamic behavior of programs, we elide the precise static semantics. We assume that programs are well-typed and variables are distinct. The abstract syntax is as follows:

```
abstract class Expr
case class Lit(i: Int) extends Expr                        // numbers
case class Var(x: String) extends Expr                     // variables
case class Lam(x: String, e: Expr) extends Expr            // abstractions
case class App(e1: Expr, e2: Expr) extends Expr            // applications
case class If0(e1: Expr, e2: Expr, e3: Expr) extends Expr  // conditionals
case class Rec(x: String, rhs: Expr, e: Expr) extends Expr // recursion
case class Aop(op: String, e1: Expr, e2: Expr) extends Expr // arithmetic
```

The abstract syntax we present can be seen as a deep embedding of the language: we use data-types to represent programs. This is the most natural choice for program analysis and allows us to use different interpretations over the AST; with the inheritance and overriding mechanisms in Scala, we may also add new language constructs and reuse existing interpretations.

2.2 Monads in Scala

A monad is a type constructor `M[_]: * → *` with two operations, often called `return` and `bind` [Moggi 1991; Wadler 1992]. Informally, `return` wraps a value into a monad `M`, and `bind` unwraps a monadic value and transforms it into a new monadic value. Pragmatically in Scala, we define the monad type class using trait `Monad` (Figure 2), which declares the `pure` [1] and `flatMap` operations. The trait itself takes the monad type constructor `M[_]` as an argument, which, as a higher-kinded type, takes a type and returns a type. The method `pure` promotes values of type `A` to values of type `M[A]`. The monadic `bind` operation is usually called `flatMap` in Scala. It takes a monad-encapsulated value of type `M[A]`, a function of `A ⇒ M[B]`, and returns values of type `M[B]`.

```
trait Monad[M[_]] {
  def pure[A](a: A): M[A]
  def flatMap[A,B](ma: M[A])(f: A ⇒ M[B]): M[B]
}
```

```
trait MonadOps[M[_], A] {
  def map[B](f: A ⇒ B): M[B]
  def flatMap[B](f: A ⇒ M[B]): M[B]
}
```

Fig. 2. trait `Monad` (left) and trait `MonadOps` (right)

Similar to Haskell's `do`-notation, Scala provides special syntactic support for monadic operations through `for`-comprehension. For example, an object of type `List[A]` is an instance of the list monad, with element type `A`. To compute the Cartesian product of two lists of numbers, we can use Scala's `for`-comprehension syntax:

```
val xs = List(1, 2); val ys = List(3, 4);
for { x ← xs; y ← ys } yield (x, y)   // List((1,3), (1,4), (2,3), (2,4))
```

The Scala compiler translates the above `for`-comprehension expression into an equivalent one using `flatMap` and `map`. The last binding of `for`-comprehensions is translated into a `map` operation, where the argument of `yield` becomes the body expression inside that `map` operation. The bindings before the last one are all translated into `flatMap`:

```
xs.flatMap { case x ⇒ ys.map { case y ⇒ (x, y) } }
```

[1] We elect to use `pure` as the name, since `return` is a keyword in Scala and `unit` is a built-in function in LMS.

Note that here the monadic object of type `List[_]` encapsulates the data internally. Therefore, it only exposes the simplified version of `flatMap`, where the monadic value of `M[A]` is not introduced as a function argument. The trait `MonadOps` (Figure 2) defines the simplified versions of monadic operations that are necessary for `for`-comprehension. The conversion between `Monad` and `MonadOps` can be achieved by using the implicit design pattern. In the rest of the paper, we use Scala's `for`-comprehension syntax and a few monads and monad transformers such as `ReaderT`, `StateT`, and `SetT` to write our interpreters. Monad transformers are type constructors of kind $(* \to *) \to (* \to *)$, which take a monad type as argument and produce another monad type. By using monad transformers, we can combine multiple monads into a single one. The implementation of monads and monad transformers follows the ones from Haskell.

2.3 Generic Interpreter

In this section, we present the generic evaluator interface in the style of a big-step definitional interpreter. The key idea is to keep both the binding-time type and returned monadic type abstract, so that they can be instantiated differently. We also need to abstract the primitive operations on those types. In later sections, we will instantiate the monadic type to perform concrete interpretation [Liang et al. 1995] or abstract interpretation [Darais et al. 2017; Sergey et al. 2013].

Basic Types. We start with some basic type definitions that are used in the interpreter. The identifiers in the program are represented by strings. To represent states, two required components for the interpreter are environments, represented by type `Env`, and stores, represented by type `Store`. An `Env` maps identifiers to addresses, and a `Store` maps addresses to values. An environment captures bound variables in the scope of current control, and a store models a persistent heap through the program run-time. Currently, the domains of addresses and values are still abstract; hence, they are declared as abstract types.

```
trait Semantics {
  type Ident = String; type Addr; type Value
  type Env = Map[Ident, Addr]; type Store = Map[Addr, Value]
  type R[_] // Binding-time as a higher-kinded type
  ...  }
```

Binding-time Abstraction. The binding-time type is declared as a higher-kinded type `R[_]` [Ofenbeck et al. 2017]. Now we can use `R` to annotate other data types in the interpreter. Later, the binding-time information is also injected into the monadic type `MonadOps`. If we instantiate `R` as the identity type (i.e., `type R[T] = T`), then the generic interpreter is a standard definitional interpreter that will execute the program. In Section 4, we instantiate `R` using LMS's built-in next-stage type annotation `Rep`, which makes the interpreter act as a compiler.

Monadic Operations. We define the return type of the interpreter `Ans` as a monadic type `AnsM[_]` wrapping the type `Value`. As mentioned in Section 2.2, in order to use the `for`-comprehension syntax certain operations must be added on the type `AnsM`. Here, we use a structural type `MonadOps` to require that type `AnsM` must (at least) implement `map` and `flatMap`. It is worth noting that `MonadOps` takes another type parameter `R[_]` as the binding-time, i.e., `MonadOps[R, AnsM, T]`. Inside `MonadOps`, `R[_]` annotates the data types `A` and `B` that are encapsulated by the monad, but not the monad type `M` itself. When the generic interpreter acts as a compiler, we will replace the monads with the ones that work on staged data values.

We also declare several operations to manipulate environments and stores. These methods return monadic values of type `AnsM[_]`, which may be parameterized over the environment or store type, or merely a `Unit` value for performing effects. For example, `local_env` installs a new environment

```
def eval(ev: Expr ⇒ Ans)(e: Expr): Ans = e match {
  case Lit(i) ⇒ num(i)
  case Var(x) ⇒ for {
    ρ ← ask_env
    σ ← get_store
  } yield get(σ, ρ, x)
  case Lam(x, e) ⇒ for {
    ρ ← ask_env
  } yield close(ev)(Lam(x, e), ρ)
  case App(e1, e2) ⇒ for {
    v1 ← ev(e1)
    v2 ← ev(e2)
    rt ← ap_clo(ev)(v1, v2)
  } yield rt
  case If0(e1, e2, e3) ⇒ for {
    cnd ← ev(e1)
    rt  ← br0(cnd, ev(e2), ev(e3))
  } yield rt
  case Let(x, rhs, e) ⇒ for {
    v  ← ev(rhs)
    ρ  ← ask_env
    α  ← alloc(x)
    _  ← set_store(α → v)
    rt ← local_env(ev(e))(ρ + (x → α))
  } yield rt
  case Aop(op, e1, e2) ⇒ for {
    v1 ← ev(e1)
    v2 ← ev(e2)
  } yield arith(op, v1, v2)
  case Rec(x, rhs, e) ⇒ for {
    α  ← alloc(x)
    ρ  ← ask_env
    v  ← local_env(ev(rhs))(ρ + (x → α))
    _  ← set_store(α → v)
    rt ← local_env(ev(e))(ρ + (x → α))
  } yield rt
}
```

Fig. 3. The generic interpreter, shared by the unstaged/staged + concrete/abstract interpreter.

ρ when evaluating the monadic value `ans`; `set_store` takes a pair of addresses and (potentially staged) values, and updates the store accordingly.

```
type MonadOps[R[_], M[_], A] = {
  def map[B](f: R[A] ⇒ R[B]): M[B]
  def flatMap[B](f: R[A] ⇒ M[B]): M[B]
}

type AnsM[T] <: MonadOps[R, AnsM, T]
type Ans = AnsM[Value]
```

```
// Environment operations
def ask_env: AnsM[Env]
def local_env(ans: Ans)(ρ: R[Env]): Ans
// Store operations
def get_store: AnsM[Store]
def put_store(σ: R[Store]): AnsM[Unit]
def set_store(av: (R[Addr], R[Value])): AnsM[Unit]
```

Primitive Operations. Next, we define a few primitive operations. First, we declare two versions of `alloc`. The first takes a store and an identifier and produces a fresh address of non-monadic type `R[Addr]`. Since the freshness of the address may depend on the store, which might be a next-stage value as indicated by its type, the type of addresses is consequently wrapped by `R[_]`. The other `alloc` is simply the monadic version of the addresses, and can therefore be used in monadic computations. An auxiliary method `get` retrieves the value of an identifier `x` through the environment and store.

```
def alloc(σ: R[Store], x: Ident): R[Addr];  def alloc(x: Ident): AnsM[Addr]
def get(σ: R[Store], ρ: R[Env], x: Ident): R[Value]
```

Other primitive operations in the interpreter handle the language constructs. The methods `num` and `close` deal with primitive values, which lift literal terms (e.g., lambdas) to our value representation (e.g., closures). Conditionals and arithmetic are handled by `br0` and `arith`, respectively. The method `ap_clo` is used for applying functions, by taking a function value and an argument value. Note that the `Env`, `Store`, and `Value` are all annotated by `R[_]`, as they are potentially next-stage values when the interpreter acts as a compiler.

```
def num(i: Int): Ans
def close(ev: Expr ⇒ Ans)(λ: Lam, ρ: R[Env]): R[Value]
def br0(test: R[Value], thn: ⇒ Ans, els: ⇒ Ans): Ans
def arith(op: Symbol, v1: R[Value], v2: R[Value]): R[Value]
def ap_clo(ev: Expr ⇒ Ans)(fun: R[Value], arg: R[Value]): Ans
```

The Interpreter. We can now construct the generic interpreter in monadic form, as shown in Figure 3. The basic idea of generic interpretation is to traverse the AST while maintaining the effects (such as environment and state updates). Note that the interpreter is written in open-recursive style: it cannot refer to itself directly. Instead, `eval` takes an additional parameter `ev` of type `Expr ⇒ Ans` that refers to itself. Consequently, method `close` that lifts λ-terms to closures and method `ap_clo` that applies functions also takes an extra `ev`, because further evaluation may happen inside them. To close the open recursion, we use a fixed-point operator `fix`. For concrete-interpretation instantiation, `fix` works like the Y combinator; for abstract-interpretation instantiation, it will instrument the interpreter by memoizing `ev`'s inputs and outputs, which ensures the termination of abstract interpretation. Finally, there is a top-level wrapper `run`. The return type `Result` depends on the kind of monads being used and is therefore also left abstract.

```
def fix(ev: (Expr ⇒ Ans) ⇒ (Expr ⇒ Ans)): Expr ⇒ Ans
type Result; def run(e: Expr): Result
```

3 A CONCRETE INTERPRETER

As the first step in our roadmap, we instantiate the generic interpreter for concrete execution in this section. The result is a standard definitional interpreter with environments and stores. It can also be obtained by refunctionalizing a standard CESK machine [Ager et al. 2003; Felleisen and Friedman 1987]. We first present the concrete components, i.e., the value domains, then show the monad stack for concrete interpretation, and finally sketch how the primitive operations are implemented.

Concrete Components. The two types we need to concretize are addresses `Addr` and values `Value`. The types `Env` and `Store` are derived automatically. To ensure the freshness of address allocations, we use type `Int` and always return a number that is greater than the size of the current store. A value can be either a tagged number `IntV`, or a closure `CloV` that contains a λ-term and an environment. The final result of the interpreter is a `Value` paired with a `Store`. To distinguish from the monadic values of type `Ans` produced by the interpreter, later we use term *grounded values* to denote such final values, which can be obtained by running the monads. The two elements in the type `Result` are annotated by the binding-time `R`, because they can be next-stage objects. We also define a standard fixed-point combinator to close the open-recursive function `ev`.

```
trait ConcreteComponents extends Semantics {
  type Addr = Int;  sealed trait Value
  case class IntV(i: Int) extends Value;  case class CloV(λ: Lam, e: Env) extends Value
  type Result = (R[Value], R[Store])
  def fix(ev: (Expr ⇒ Ans) ⇒ (Expr ⇒ Ans)): Expr ⇒ Ans = e ⇒ ev(fix(ev))(e)
}
```

Unstaged Monads. For concrete interpretation, the monad needs to model reader and state effects, which correspond to the environment and the store, respectively. We follow the monad transformer approach [Liang et al. 1995], and use the `ReaderT` and `StateT` monad transformers to compose the monad stack. In other words, the type `AnsM` is instantiated by layering the `ReaderT` and `StateT` transformers[2], where the `ReaderT` is parameterized by the type `Env`, and the `StateT` is parameterized by the type `Store`, and the inner-most monad `IdM` is merely the identity monad.

```
trait ConcreteSemantics extends ConcreteComponents {
  type R[T] = T;  type AnsM[T] = ReaderT[StateT[IdM, Store, ?], Env, T] // the monad stack
  ... }
```

Here, we sketch the basic idea of `ReaderT` and `StateT`. Readers may refer to [Chiusano and Bjarnason 2014; Liang et al. 1995] for more detail. A `ReaderT` monad transformer encapsulates

[2]The question mark syntax is a kind projector [Osheim 2019], such that `StateT[IdM,Store,?]` is equivalent to `({type M[T]=StateT[IdM,Store,T]})#M`

computation `R ⇒ M[A]`, where `R` is the environment type, and `M[_]` is the inner monadic type. Given a value of `R`, a `ReaderT` monad produces a transformed value of type `M[A]`. Similarly, a `StateT` monad encapsulates computation `S ⇒ M[(A, S)]`, where `S` is the state type, and `M[_]` is the inner monad type. Given a value of type `S`, a `StateT` monad produces a transformed value of type `M[(A, S)]`, where the new state (of type `S`) is accompanied with the result (of type `A`). Note that for the moment, the binding-time type `R` is the identity type; thus, these monads operate on unstaged data. We can also see this from the signature of `flatMap`: the argument function `f` takes an unstaged value of type `A` and produces a monadic value. In the following code, we elide operations other than `flatMap`.

```
case class ReaderT[M[_]: Monad, R, A](run: R ⇒ M[A]) {
  def flatMap[B](f: A ⇒ ReaderT[M, R, B]): ReaderT[M, R, B] =
    ReaderT(r ⇒ Monad[M].flatMap(run(r))(a ⇒ f(a).run(r))); ... }
case class StateT[M[_]: Monad, S, A](run: S ⇒ M[(A, S)]) {
  def flatMap[B](f: A ⇒ StateT[M, S, B]): StateT[M, S, B] =
    StateT(s ⇒ Monad[M].flatMap(run(s)) { case (a, s1) ⇒ f(a).run(s1) }); ... }
```

After defining the monad stack, the operations manipulating environments and stores can be defined by constructing the proper monadic value and lifting it to the top-level of our monad stack. To modify the store, for example, we construct a `StateT` value that updates the current store σ to σ + αv, which results in a `StateT[IdM, Store, Unit]` value. Then, we lift this `StateT` value to the top-level `ReaderT` type, i.e., `AnsM[Unit]`.

```
def set_store(αv: (Addr, Value)): AnsM[Unit] = liftM(StateTMonad.mod(σ ⇒ σ + αv))
```

Primitive Operations. Other primitive operations over the value domains can be implemented straightforwardly. We elide most of them but describe how we handle functions and applications. The reason is that λ-terms are data but are also part of the control flow. The way we treat them is simple now, but will become more involved when we stage the abstract interpreter. The method `close` denotes a λ-term to our representation of functions. At the moment, `close` takes a λ-term and an environment, and produces the defunctionalized representation of closures. The method `ap_clo` performs function applications, which takes a function value `fun` and an argument `arg`. It first extracts the syntactic λ-term and environment enclosed in `fun`, and allocates an addresses for the argument and updates the environment and store accordingly. Finally, by recursively calling `ev`, it evaluates the body expression of the λ-term under the new environment and store.

```
def close(ev: Expr ⇒ Ans)(λ: Lam, ρ: Env): Value = CloV(λ, ρ)
def ap_clo(ev: Expr ⇒ Ans)(fun: Value, arg: Value): Ans = fun match {
  case CloV(Lam(x, e), ρ: Env) ⇒ for {
    α ← alloc(x)
    _ ← set_store(α → arg)
    rt ← local_env(ev(e))(ρ + (x → α))
  } yield rt
}
```

We define the top-level `run` method as invoking the fixed-point operator with the evaluator and input expression `e`, and running the monad stack with the initial environment ρ_0 and store σ_0.

```
def run(e: Expr): Result = fix(eval)(e)(ρ0)(σ0)
```

4 FROM INTERPRETERS TO STAGED INTERPRETERS

In this section, based on the generic interpreter and concrete components we presented in Section 2, we show how to stage the concrete interpreter by changing the monad type and refactoring the primitive operations. We begin by briefly introducing the Lightweight Modular Staging framework in Scala, and then replay the same steps as we did for the unstaged counterpart. Finally, we briefly describe the code generation.

4.1 Multi-stage Programming with LMS

Lightweight Modular Staging (LMS) [Rompf and Odersky 2010] is a multi-stage programming framework implemented as a Scala library. LMS enables runtime code generation in a type-safe manner. Different from the approach of MetaML/MetaOCaml [Calcagno et al. 2003; Kiselyov 2014] which uses syntactic quotations, LMS distinguishes binding-times based on types and preserves the evaluation order via an internal ANF transformation of the staged program. To achieve typed-based MSP, LMS provides a type constructor `Rep[T]`, where `T` can be an arbitrary type, indicating that a value of type `T` will be known at a future stage. Operations acting on `Rep[T]` expressions will be residualized as generated code. A classic example for introducing multi-stage programming is the power function that computes b^x. Usually, it is implemented as a recursive function:

```
def power(b: Int, x: Int): Int = if (x == 0) 1 else b * power(b, x - 1)
```

If `x` is known at the current stage, we may specialize the power function to the value `x` by unrolling the recursive calls. In LMS, this is fulfilled by first adding the `Rep` type annotation to variables known at the next stage. In this case, `b: Rep[Int]` is known later, and `x: Int` is known currently, as shown below. The way we use this staged `power` function is to create a `DslDriver` and override the `snippet` method that supplies `5` as the currently known value for `x`.

```
new DslDriver[Int, Int] {
  def power(b: Rep[Int], x: Int): Rep[Int] = if (x == 0) 1 else b * power(b, x-1)
  def snippet(b: Rep[Int]): Rep[Int] = power(b, 5) // specialize the power to b^5
}
```

The LMS framework provides staging support for primitive data types such as `Int` and `Double`, and for commonly-used data structures such as lists and maps. The framework internally constructs a sea-of-nodes intermediate representation (IR) for the next-stage program when executing the current stage program [Rompf 2016]. For convenience, the conversion from expressions of type `Rep[_]` to their IR is done by using the implicit design pattern. As we will see later, implementing staging and code generation support for user-defined classes is also straightforward.

4.2 Staged Concrete Semantics

To implement the staged concrete interpreter, we replay the steps from the instantiation of the unstaged counterpart in Section 3. However, now we use the `Rep` type to annotate the value domains, environments and stores, and define the staged version of monads and primitive operations.

Staged Monads. We use the same monad stack structure as in the unstaged interpreter: a reader monad with a state monad. The difference here is that now the monads operate on staged values. For brevity, we call them *staged monads*. In the code snippet, we also use a `Rep` prefix on the name of constructors and types to differentiate them. But it is important to note that the instances of `ReaderT`/`StateT` are not staged, and that the monadic computation, such as functions of `R ⇒ M[A]`, are also not staged. Instead, the internal data that these monads operate on are staged, i.e., values of type `R` and `A` in the reader monad, and values of type `S` and `A` in the state monad. The following code snippet shows the idea. We use light gray to highlight where `Rep` types are added.

```
case class RepReaderT[M[_]: RepMonad, R, A](run: Rep[R] ⇒ M[A]) {
  def flatMap[B](f: Rep[A] ⇒ ReaderT[M, R, B]): RepReaderT[M, R, B] =
    RepReaderT(r ⇒ RepMonad[M].flatMap(run(r))(a ⇒ f(a).run(r))); ... }
case class RepStateT[M[_]: RepMonad, S, A](run: Rep[S] ⇒ M[(A, S)]) {
  def flatMap[B](f: Rep[A] ⇒ StateT[M, S, B]): RepStateT[M, S, B] =
    RepStateT(s ⇒ RepMonad[M].flatMap(run(s))(as ⇒ f(as._1).run(as._2))); ... }
```

The function `f` passed to the `flatMap` is also a value known at the current stage. Therefore, all invocations of `flatMap` can be reduced before generating code. The fact that we only stage data but not the monadic computation or monadic values is the reason that we can peel of the monad

stack in the generated code through staging. Now the type `AnsM[_]` is instantiated as the same structure as before, but using the `Rep` versions of monad transformers:

```
type R[T] = Rep[T]
type AnsM[T] = RepReaderT[RepStateT[RepIdM, Store, ?], Env, T]
```

Readers may notice that the conversion between unstaged monads and staged monads is merely changing the type of unstaged data to staged data, which in fact can be achieved without modifying the implementation of monads. This is true so far, but as we will see, it is not as straightforward for the nondeterminism monad (`SetT`) in the abstract interpreter, because not only the elements in the set are staged, but the whole set is also staged, and we have no knowledge about how many elements are in the set. Therefore, any traversals over the set has to be staged. In this section, we explicitly distinguish the two versions of monadic interfaces. Later, we will instantiate the staged set monad by manually fusing the fragment of the monad stack inside `SetT`.

Primitive Operations. Most of the primitive operations can be easily translated to their staged versions – we just need to change the types. As we mentioned before, we will illustrate in detail how functions and applications are handled. One problem here is what should we do for λ-terms when staging them? We cannot directly create a next-stage defunctionalized closure for it, because that means we still need to interpret over these closures at the next stage and the interpretation overhead has not been eliminated. The solution is to compile the λ-term with its environment by calling the evaluator at the current stage. The result is generating a next-stage Scala function. The evaluator is passed as `ev` to the method `close`. The following code implements this idea:

```
type ValSt = (Value, Store)
def emit_compiled_clo(f: (Rep[Value], Rep[Store]) ⇒ Rep[ValSt], λ: Lam, ρ: Exp[Env]): Rep[Value]
def close(ev: Expr ⇒ Ans)(λ: Lam, ρ: Rep[Env]): Rep[Value] = {
  val Lam(x, e) = λ
  val f: (Rep[Value], Rep[Store]) ⇒ Rep[ValSt] = { case (v: Rep[Value], σ: Rep[Store]) ⇒
      val α = alloc(σ, x)
      ev(e)(ρ + (unit(x) → α))(σ + (α → v))
  }; emit_compiled_clo(f, λ, ρ)
}
```

In `close`, we first create a current-stage function `f`, whose input and output are both next-stage values and stores. We then use `emit_complied_clo` to delay `f` to a next-stage Scala function. The method `emit_compiled_clo` produces current-stage representations of next-stage function values, i.e., of type `Rep[Value]`. Inside function `f`, we can access the evaluator via `ev`. However, `ev` produces a monadic value of type `AnsM`, which can only exist at the current stage. To connect current-stage monadic values and future-stage grounded values, the evaluator `ev` is supplied with not only the body expression `e` to be specialized, but also with the prepared environment and store. By doing this, we have *collapsed* the monadic values of type `AnsM` to grounded values (of type `Rep[ValSt]`) within staging; therefore in the future stage, there will be no monadic value.

```
def emit_ap_clo(fun: Rep[Value], arg: Rep[Value], σ: Rep[Store]): Rep[ValSt]
def ap_clo(ev: EvalFun)(fun: Rep[Value], arg: Rep[Value]): Ans = for {
  σ  ← get_store
  vs ← lift[ValSt](emit_ap_clo(fun, arg, σ))
  _  ← put_store(vs._2)
} yield vs._1
```

For function applications `ap_clo`, we have two *next-stage* values `fun` and `arg`. But what we can do with these values of type `Rep` is limited. In fact, `fun` does not have an intensional application operation we can use directly. We only know that it is a current-stage representation of functions produced by `close`, whose next-stage form is a Scala function. With this knowledge, we can

generate a next-stage function application for it. The method `emit_ap_clo` produces a current-stage representation of a next-stage function application, with required arguments such as the latest store σ. Meanwhile, we still want the method `ap_clo` implemented in monadic style, so that it can be smoothly connected with other parts of the interpreter. To achieve this, we lift the result values and stores from the future stage into our current-stage monad stack, with a binding `vs`. Finally, we use `put_store` to update the store with `vs._2` and return the value result `vs._1` into the current-stage monadic value.

Now, compared with the unstaged interpreter, we observe a key difference where we use the evaluation function `ev`. In the unstaged interpreter, the invocation of `ev` happens in `ap_clo`, i.e., at the time of application; while in the staged interpreter, we eagerly apply `ev` when denoting the λ-terms to our next-stage value domains, i.e., at the time of value-representation.

4.3 A Little Bit of Code Generation

In the previous section, we showed the types of `emit_compiled_clo` and `emit_ap_clo` without their concrete implementations. In this section, we briefly discuss the IR nodes generated by them and sketch the code generation. We define the IR nodes `IRCompiledClo` and `IRApClo` using `case class` extending from `Def[T]`. `Def[T]` is a built-in type in the LMS framework, representing next-stage value definitions of type `T`.

```
case class IRCompiledClo(f: Rep[(ValSt) ⇒ ValSt], λ: Lam, ρ: Rep[Env]) extends Def[Value]
case class IRApClo(fun: Rep[Value], arg: Rep[Value], σ: Rep[Store]) extends Def[ValSt]
```

The IR nodes are manipulated by the LMS passes and finally generated as next-stage Scala programs. To generate code, LMS provides an `emitNode` method for programmers to control over what code is generated for each kind of nodes. Here, we match `IRCompiledClo` and `IRApClo`, and generate definitions via `emitValDef` for them. For compiled closures, we pass the function `f` with its accompanying syntactic term and environment to the next-stage value representation `CompiledClo`. For applications, we know that `fun` is an instance of `CompiledClo` at the next stage, and it has a callable field `f`. Hence, we generate a next-stage function application `f(arg, σ)`. The invocations of `quote` are used to generate proper names for other next-stage values.

```
override def emitNode(sym: Sym[Any], rhs: Def[Any]) = rhs match {
  case IRCompiledClo(f, λ, ρ) ⇒ emitValDef(sym, s"CompiledClo(${quote(f)}, ${quote(λ)}, ${quote(ρ)})")
  case IRApClo(fun, arg, σ) ⇒ emitValDef(sym, s"$fun.f(${quote(arg)}, ${quote(σ)})")
  ... }
```

5 FROM INTERPRETERS TO ABSTRACT INTERPRETERS

After seeing the unstaged and staged concrete interpreter, we now turn our focus to abstract interpreters under the same framework. We first present a lattice representation with binding-time abstraction, as well as simple abstract domains, such as power sets. Then, we show the abstract components, namely, abstract values, abstract environments, and abstract stores. The abstract interpreter we construct in this section is similar to Darais et al. [2017]'s. For simplicity, it is context/path/flow-insensitive by our choice. In Section 8.2, we will instantiate it to a commonly-used context-sensitive analysis. Darais et al. [2015] showed how to achieve path- and flow-sensitivity by varying the monads, which is also applicable to our unstaged or staged abstract interpreter.

5.1 Stage Polymorphic Lattices

Many abstract domains in abstract interpreters can be formulated as complete lattices. A complete lattice over set S is a tuple $\langle S, \sqsubseteq, \top, \bot, \sqcup, \sqcap \rangle$, where $\top$ is the top element, $\bot$ is the bottom element, $\sqsubseteq$: $S \to S \to \mathit{Bool}$ is the ordering relation, $\sqcup : S \to S \to S$ is the join (least upper bound) operator, and $\sqcap : S \to S \to S$ is the meet (greatest lower bound) operator. The trait `SPLattice` (Figure

4) defines a type class of stage-polymorphic lattices. Similar to the `MonadOps` in Section 2.3, we introduce an additional higher-kinded type `R[_]` to the trait for annotating the binding-times, where `R[_]` wraps the data type `S` and `Boolean` in these operations.

```
trait SPLattice[S, R[_]] {
  val ⊤: R[S];  val ⊥: R[S]
  def ⊑(l1: R[S], l2: R[S]): R[Boolean]
  def ⊔(l1: R[S], l2: R[S]): R[S]
  def ⊓(l1: R[S], l2: R[S]): R[S]
}
trait Lattice[S] extends SPLattice[S, NoRep]
```

```
def SetLattice[T]: Lattice[Set[T]] = new Lattice[Set[T]] {
  lazy val ⊤: Set[T] = error("No representation for ⊤")
  lazy val ⊥: Set[T] = Set[T]()
  def ⊑(l1: Set[T], l2: Set[T]): Boolean = l1 subsetOf  l2
  def ⊔(l1: Set[T], l2: Set[T]): Set[T]  = l1 union     l2
  def ⊓(l1: Set[T], l2: Set[T]): Set[T]  = l1 intersect l2
}
```

Fig. 4. trait `SPLattice` and `Lattice` (left), and the power set instance (right)

In this section, we instantiate `SPLattice` with the flat binding-time type `R[T] = T` (i.e., `NoRep`); the result is the trait `Lattice[S]`. An example of the lattices we use in the rest of the paper is the power set abstract domain (shown in Figure 4). Two power sets are ordered by the subset relation. We use set union to compute their join, and set intersection to compute their meet. The bottom element of a power set is the empty set, and we do not have a representation of the top element for power set. Other lattices used in the paper, such as products and maps, can be implemented similarly or by lifting the existing lattices element-wise or point-wise. Non-relational numerical abstract domains such intervals can also be implemented in a stage-polymorphic way.

5.2 Abstract Semantics

In this section, we follow the abstracting definitional interpreters (ADI) approach [Darais et al. 2017] to refactor our monadic concrete interpreter to a monadic abstract interpreter.

Abstract Components. We first widen the concrete value domain to `AbsValue`. There are two variants of `AbsValue`: 1) numbers are lifted to a singleton object `IntTop`, which represents the set of all numbers; 2) closures remain the same. Then, the type `Value` is redefined as a set of `AbsValue` to account for approximation. The address space is constrained to be finite to ensure the computability of the analysis – for a simple monovariant analysis, we directly use variable names as addresses. After defining the `Value` and `Addr` type, the types of environments and stores are automatically lifted to their abstract versions, i.e., `Map[Ident, Addr]` and `Map[Addr, Set[AbsValue]]`, respectively.

```
trait AbstractComponents extends Semantics {
  sealed trait AbsValue
  case object IntTop extends AbsValue
  case class CloV(lam: Lam, env: Env) extends AbsValue
  type Value = Set[AbsValue]; case class Addr(x: String)
  ... }
```

It is important that the abstract stores now map addresses to sets of abstract values, indicating that an address may point to an over-approximated set of runtime values. This can be justified by the approximation nature and nondeterminism during the analysis. For example, when analyzing a conditional expression, we may not have enough information to decide which branch will be taken, thus a sound treatment is to explore both of the branches. Also, at some later point, we need to compute the join of two paths. Another source of nondeterminism comes from the abstract semantics of function application. For instance, consider an expression `f(a)`, if there are multiple possible target closures of `f`, then we have to apply all of them nondeterminitically.

To ensure the abstract interpreter terminates on all programs when computing the fixed-point, the ADI approach uses a co-inductive mechanism consisting of two caches that remember the input and output of the `eval` function. Here, we first provide the necessary type definitions, and describe the algorithm later. A `Cache` is a mapping from configurations `Config` to sets of value-store pairs,

where the configuration is a triple of the current expression being evaluated, environment and store. Intuitively, a cache memoizes the result values and stores for a given program configuration.

```
type Config = (Expr, Env, Store); type Cache = Map[Config, Set[(Value, Store)]]
```

Monads for Abstract Interpretation. Compared with the concrete interpreter that uses reader and state effects, the abstract interpreter further introduces the nondeterminism effect and another reader and state effect for the two caches. The nondeterminism effect is represented by the `Set[M[_], A]` monad transformer, where `M` is the inner monad type being transformed, and `A` is the type of elements in the set. We use a `ReaderT` for one cache that is not changed during one fixed-point iteration, and use a `StateT` for another cache that will be constantly updated during the analysis. Darais et al. [2017] discuss different permutations of the monad stack for abstract interpretation. The following `AnsM` type shows the monad stack (we use light gray to highlight what has been changed from the monad stack of concrete interpretation):

```
trait AbstractSemantics extends AbstractComponents {
  type R[T] = T
  type AnsM[T] = ReaderT[StateT[SetT[ReaderT[StateT[IdM, Cache, ?], Cache, ?], ?], Store, ?], Env, T]
  ... }
```

Similar to the concrete scenario, we sketch the `flatMap` implementation of the `SetT` transformer and omit the rest. The field `run` encapsulated by the monad is of type `M[SetT[A]]`, where `M[_]` is the inner monad, and `A` is the element type of the set. We first apply `flatMap` on the inner monad to obtain the set `s`. Then, we use the function `f` to transform every element of type `A` into a monadic value of type `Set[M, B]`. Finally, we fold all the transformed values into a single monadic value of type `Set[M, B]` by appending all of them. The initial value of the fold is an empty `SetT`.

```
  case class SetT[M[_]: Monad, A](run: M[Set[A]]) {
    def flatMap[B](f: A ⇒ SetT[M, B]): SetT[M, B] =
      SetT(Monad[M].flatMap(run) { (s: Set[A]) ⇒
        s.foldLeft(SetT.empty[M, B])((acc, a) ⇒ acc ++ f(a)).run
      }); ... }
```

This monad stack scheme (`AnsM[T]`) leads to a different type of grounded values. Recall that in the concrete setting, we only have a reader monad (for environments) and a state monad (for stores). Different from state effects, reader effects are not visible at the final result. Hence, together with the value produced by the interpreter, the final result type is just a pair of `Value` and `Store`. However, under the new monad stack for abstract interpretation, we have a `SetT` monad transformer inside the environment monad and store monad. Therefore, the type `Set` becomes the container type of the pairs of values and stores, i.e., `Set[(Value, Store)]`. Also, note that the reader and state monad for the caches both inhabit the nondeterminism monad; as the result, the final result type pairs the set of value-stores with the cache type from that state monad, as shown below.

```
type Result = (R[Set[(Value, Store)]], R[Cache])
```

Primitive Operations. The primitive operations are changed according to the new monad stack scheme and value domains. One of the notable changes is that the store update operator merges the new values with existing values. As we mentioned before, sometimes we have to explore both of the branches for conditionals: in method `br0`, we combine the results using the `mplus` operation from `MonadPlus`, which requires the value domains to be join-semilattices (in our case, `Set` and `Map`).

```
  def set_store(αv: (Addr, Value)): AnsM[Unit] = liftM(StateTMonad.mod(σ ⇒ σ ⊔ Map(αv)))
  def br0(test: Value, thn: ⇒ Ans, els: ⇒ Ans): Ans = ReaderTMonadPlus.mplus(thn, els)
```

The value representation of λ-terms is still a defunctionalized closure of type `CloV`, but we lift it to a singleton set of `AbsValue` (remember that type `Value` is an alias of `Set[AbsValue]`).

```
  def close(ev: Expr ⇒ Ans)(λ: Lam, ρ: Env): Value = Set(CloV(λ, ρ))
```

For function applications, `ap_clo` looks the same as the concrete interpreter. The difference is that now the first argument of `ap_clo` is a set of closures `funs`, so we cannot directly match it against with a syntactic `Lam` by pattern matching. Instead, we use an auxiliary function `lift_nd` that takes a set and lifts the elements in the set into the monad stack. Then we can straightforwardly implement the function `ap_clo` still in monadic style, where the closures come from the monads and thus the nondeterminism can be naturally handled. The light gray line shows what is added from its concrete counterpart.

```
def lift_nd[T](vs: Set[T]): AnsM[T]
def ap_clo(ev: Expr ⇒ Ans)(funs: Value, rand: Value): Ans = for {
  CloV(Lam(x, e), ρ) <- lift_nd[AbsValue](funs)
  α ← alloc(x)
  _ ← set_store(α → funs)
  rt ← local_env(ev(e))(ρ + (x → α))
} yield rt
```

Caching and Fixpoint Iteration. As we mentioned earlier, the ADI approach uses a two-cache mechanism to compute the least fixed-point and prevent non-termination. The caching algorithm is also called a *co-inductive* caching or *truncated depth-first evaluation* [Rosendahl 2013]. It has been used in other abstract interpreters or fixed-point computation [Darais et al. 2017; Rosendahl 2013; Wei et al. 2018]. The idea is to use an `in` cache and an `out` cache during the depth-first evaluation. The `in` cache stores the result from the last iteration, and the `out` cache is constantly updated during the current iteration. In the next iteration, the last `out` cache is used as the `in` cache, and an empty cache is plugged into the `out` slot. Once the `out` cache does not contain any new information compared with the `in` cache, the fixed-point is reached. In our monad stack, the `in` and `out` caches are modeled by the reader monad and state monad, respectively. We first define several methods to manipulate the two caches through the monad stack (the implementations are elided).

```
def ask_in_cache: AnsM[Cache]; def get_out_cache: AnsM[Cache]
def put_out_cache(out: R[Cache]): AnsM[Unit]
def set_out_cache(cfg: R[Config], vs: R[(Value, Store)]): AnsM[Unit]
```

The co-inductive caching algorithm is implemented as an instrumentation over the `eval` function (Figure 5), and it also closes the open recursion. The instrumentation works as follows. Initially, the two caches are both empty. During the iteration, we first check whether the `out` cache contains the configuration `cfg`, which represents the current desired computation. If `out` does contain `cfg`, then the result is directly returned throughout the monad stack. Otherwise, we first retrieve the result from `in` ($\bot$ if `in` does not contain `cfg`), and update this result from `in` into the `out` cache in the fashion of join. Then, we invoke `ev` to evaluate the result for this iteration, where `ev` takes `fix(ev)` as the self reference. After the evaluation, the store σ may have been changed, so we obtain the latest store and construct the result `(v, σ)`, which will be used to update the `out` cache.

The iteration terminates when the resulting `out` cache is equivalent to the input `in` cache, which indicates that there is no more fact have been discovered. Therefore, the iteration should end, and we have reached the least fixed-point.

```
def run(e: Expr): Result = fix(eval)(e)(ρ0)(σ0)(cache0)(cache0).run
```

Finally, we override the top-level `run` method by running the monadic value `fix(eval)(e)` with the initial environment ρ_0, initial abstract store σ_0, and initial caches `cache0`; all of which are empty.

6 FROM ABSTRACT INTERPRETERS TO STAGED ABSTRACT INTERPRETERS

In the previous sections, we have seen an unstaged abstract interpreter and a staged concrete interpreter. Now we begin describing the implementation of their confluence – a staged abstract

```
def fix(ev: (Expr ⇒ Ans) ⇒ (Expr ⇒ Ans)): Expr ⇒ Ans = e ⇒ for {
  ρ ← ask_env; σ ← get_store; in ← ask_in_cache; out ← get_out_cache; val cfg = (e, ρ, σ)
  rt ← if (out.contains(cfg)) for { // ask if out already contains the desired result
         (v, σ) ← lift_nd[(Value, Store)](out(cfg));  _ ← put_store(σ)
       } yield v
       else for {
         _ ← put_out_cache(out + (cfg → in.getOrElse(cfg, ⊥)))
         v ← ev(fix(ev))(e);  σ ← get_store;  _ ← update_out_cache(cfg, (v, σ))
       } yield v
} yield rt
```

Fig. 5. The unstaged co-inductive caching algorithm.

interpreter. Unsurprisingly, the staged abstract interpreter in this section uses the same abstract semantics as the unstaged version in Section 5. The approach we use to refactor the unstaged one to the staged abstract interpreter is modular, and does not sacrifice soundness or precision. The designer of the analyzer therefore does not have to rewrite the analysis. We first present the staged lattices and staged monads, and then discuss the staged version of primitive operations, especially `close`, `ap_clo` and `fix`. In the end, we discuss several optimizations.

6.1 Staged Lattices

In Section 5.1, we exploited the higher-kinded type `R` to achieve stage polymorphism. Now we instantiate the type `R` as `Rep` and still use the power-set as example to describe its staged version.

```
trait RepLattice[S] extends Lattice[S, Rep]
def RepSetLattice[T]: RepLattice[Set[T]] = new RepLattice[Set[T]] {
  lazy val ⊥: Rep[Set[T]] = Set[T]()
  lazy val ⊤: Rep[Set[T]] = error("No representation for ⊤")
  def ⊑(l1: Rep[Set[T]], l2: Rep[Set[T]]): Rep[Boolean] = l1 subsetOf  l2
  def ⊔(l1: Rep[Set[T]], l2: Rep[Set[T]]): Rep[Set[T]]  = l1 union     l2
  def ⊓(l1: Rep[Set[T]], l2: Rep[Set[T]]): Rep[Set[T]]  = l1 intersect l2
}
```

The trait `RepLattice` is shown by instantiating `Lattice` with the type `Rep`. For all type `T`, method `RepSetLattice` provides an instance of `RepLattice` for `Set[T]`, where the lattice operations are eventually delegated to the operations on the staged set data type, such as `subsetOf`, `union` and `intersect`. Compared with the unstaged version, we do not need to change the implementations, except the types – such as changing them from `Set[T]` to `Rep[Set[T]]`. We show the changes in light gray code. In the code generation part, these operations on staged sets emit their corresponding next-stage code. Again, other lattice structures such as products and maps can be implemented in a similar way.

6.2 Staged Abstract Semantics

Now we can implement the staged abstract semantics, with the binding-time type `R` instantiated as `Rep`. The types of abstract components are reused from the unstaged version.

Staged Monads for Abstract Interpretation. We use the same monad stack structure as the unstaged abstract interpreter, but replacing *all* the monad transformers with their staged versions. The following code snippet shows this change. We manually fuse the three inner transformers (`IdM` is omitted) into a single monad `RepSetReaderStateM[R, S, A]`, where `R` is the type parameter for the reader effects, and `S` is the type for the state. In our monad stack scheme, `R` and `S` are both instantiated with the type `Cache`. Similar to the unstaged version, the grounded result type is a pair of two staged values: `Rep[Set[(Value, Store)]]` and `Rep[Cache]`.

```
trait StagedAbstractSemantics extends AbstractComponents {
```

```
  type R[T] = Rep[T]
  type AnsM[T] = RepReaderT[RepStateT[RepSetReaderStateM[Cache, Cache, ?], Store, ?], Env, T]
  type Result = (Rep[Set[(Value, Store)]], Rep[Cache])
  ... }
```

Primitive Operations. When deriving the staged concrete interpreter from its unstaged counterpart, we notice that where the evaluation `ev(e)` is invoked are shifted from `ap_clo` to `close`, where `e` is the body expression of a λ-term. In other words, when staging, we eagerly specialize the interpreter and generate code every time we reach a λ-term, instead of lazily calling `ev` when the applications happen. Similarly, here we use the same way to handle staged λ-terms. Additionally, in the staged abstract interpreter, we have to handle nondeterminism incurred by the over-approximation of runtime behavior. In the rest of this part, we focus on discussing the closure representation and function application for the staged version.

The `close` method now is a mix of the *staged* concrete version and unstaged *abstract* version. We first build a current-stage function `f`, which takes four next-stage values, including the argument and latest store, and `in/out` caches additionally. Inside `f`, we collapse the monadic value of type `Ans` to grounded values of type `Result`, by providing the desired arguments, i.e., the new environment, new store, and two caches. The collapsing of monadic values happens at the current-stage, so the invocation of `ev` is unfolded at staging time. Finally, we generate a singleton set containing the compiled next-stage closure (`emit_compiled_clo`), which is represented by an IR node in LMS.

```
def emit_compiled_clo(f: (Rep[Value], Rep[Store], Rep[Cache], Rep[Cache])
                        ⇒ Rep[(Set[(Value,Store)], Cache)], λ: Lam, ρ: Exp[Env]): Rep[AbsValue]
def close(ev: Expr ⇒ Ans)(λ: Lam, ρ: Rep[Env]): Rep[Value] = {
  val Lam(x, e) = λ
  val f: (Rep[Value],Rep[Store],Rep[Cache],Rep[Cache]) ⇒ Rep[(Set[(Value,Store)],Cache)] = {
    case (arg, σ, in, out) ⇒
      val α = alloc(σ, x)
      ev(e)(ρ + (unit(x) → α))(σ ⊔ Map(α → arg))(in)(out)
  }; Set[AbsValue](emit_compiled_clo(f, λ, ρ)) }
```

The `ap_clo` method for function applications is also similarly mixing the two previous versions. We use the staged version of `lift_nd` to lift the next-stage set of functions into the monad stack. For each closure `clo` in the set, we generate a next-stage value, representing the function application `clo(arg)`, by using `emit_ap_clo`. Again, the method `emit_ap_clo` produces a current-stage representation of the future-stage application result. Finally, we reify the `out` cache, store, and values, which are all from the future-stage, back into the current-stage monadic value.

```
def emit_ap_clo(fun: Rep[AbsValue], arg: Rep[Value], σ: Rep[Store],
                in: Rep[Cache], out: Rep[Cache]): Rep[(Set[ValSt], Cache)]
def ap_clo(ev: Expr ⇒ Ans)(funs: Rep[Value], arg: Rep[Value]): Ans = for {
  σ   ← get_store;      clo ← lift_nd[AbsValue](funs)
  in  ← ask_in_cache;   out ← get_out_cache
  res ← lift_nd[(Set[ValSt], Cache)](Set(emit_ap_clo(clo, arg, σ, in, out)))
  _   ← put_out_cache(res._2);  vs ← lift_nd[ValSt](res._1);  _ ← put_store(vs._2)
} yield vs._1
```

Staged Caching and Fixpoint Iteration. The fixed-point iteration again relies on the two caches `in` and `out`, which are both staged maps now. Therefore, the query of whether the `in/out` cache contains the current configuration will produce next-stage Boolean values, i.e., of type `Rep[Boolean]`, and the branching condition cannot be determined statically. We have to generate code for the whole `if` expression. Figure 6 shows the staged version of `fix`. The variable `res` represents the next-stage result, consisting of a next-stage `if` expression. The true branch simply returns a pair of the query result form the `out` cache, and the `out` cache itself. The else branch constructs a

```
def fix_cache(e: Expr): Ans = for {
  ρ   ← ask_env;  σ ← get_store;  in ← ask_in_cache;  out ← get_out_cache
  cfg ← lift_nd[Config](Set((unit(e), ρ, σ)))
  res ← lift_nd[(Set[ValSt], Cache)](Set(
    if (out.contains(cfg)) (out(cfg), out)  // a next-stage value of type Rep[(Set[ValSt], Cache)]
    else { val m: Ans = for {               //   generated by the if/else
             _ ← put_out_cache(out + (cfg → in.getOrElse(cfg, ⊥)))
             v ← eval(fix_cache)(e);  σ ← get_store;  _ ← update_out_cache(cfg, (v, σ))
           } yield v
           m(ρ)(σ)(in)(out) }))
  _ ← put_out_cache(res._2);  vs ← lift_nd(res._1);  _ ← put_store(vs._2)
} yield vs._1
```

Fig. 6. The staged co-inductive caching algorithm.

monadic value `m` of type `Ans` first, which evaluates `e` under the new `out` cache. After this, we use a similar technique that eagerly collapses the monadic value `m` to grounded values, by providing its desired environment and other arguments. Finally, we have a current-stage representation of the future-stage values `res`, and we reify the content of `res` back into the current-stage monad stack.

6.3 A Little Bit of Code Generation, Again

The code generation for compiled closures and function applications is similar to their counterparts in the staged concrete interpreter. We have two IR nodes implemented as case classes; they also take additional caches as arguments. We elide the code generation part for these IR nodes.

```
case class IRCompiledClo(f: (Rep[Value], Rep[Store], Rep[Cache], Rep[Cache])
                         ⇒ Rep[(Set[ValSt], Cache)], λ: Lam, ρ: Rep[Env]) extends Def[AbsValue]
case class IRApClo(clo: Rep[AbsValue], arg: Rep[Value], σ: Rep[Store],
                   in: Rep[Cache], out: Rep[Cache]) extends Def[(Set[(Value, Store)], Cache)]
```

6.4 Optimizations

Our staging schema works by unfolding the interpreter over the abstract syntax tree of the input program. In practice, however, the staging schema would suffer from code explosion when analyzing (specializing) large programs, which increases compile time. If we generate next-stage programs running on the JVM, such large generated programs would also incur GC overhead at runtime. In this section, we present optimizations that largely mitigate these issues. Implementing all of those optimizations do not need to change the generic interpreter.

Specialized Data Structures. In the staged interpreters, all instances of `Env` and `Store` are staged. The data structures representing these components are treated as black-boxes, i.e., `Rep[Map]`, which means that the operations on a `Rep[Map]` directly become next-stage code, and we do not inspect any further inside. As we identified when introducing the generic interface, the keys of an `Env` are string-represented identifiers in the program, which are completely known statically. This observation gives us a chance to further specialize the data structures for environments. For example, let us assume that the `Map[K, V]` is implemented as a hash-map. If all the keys of type `K` are all known statically, then the indices for those keys can also be computed statically. Thus, in this case, the specialized map will be an array of type `Array[Rep[V]]`, whose elements are next-stage values, and the size of the array is known statically as the program has finite number of identifiers. As result, an access to the map is staged into an access to the array with a pre-computed index.

In particular, if we are specializing a monovariant analysis, the address space is equivalent to the set of all identifiers in the program. Utilizing this fact, the result of accesses to the environment can be computed statically and we may generate addresses directly during staging. After this, the store can be specialized as an array by using the way mentioned above.

Selective Caching. We observe that the two-fold co-inductive caching is used for every recursive call in our abstract interpreter. But this is unnecessary and redundant when generating code for atomic expressions such as literals or variables, because they always terminate. Borrowing the idea from the partition of expressions used in administrative normal form (ANF) λ-calculus [Flanagan et al. 1993], we can use a selective caching algorithm that does not generate caching code for atomic expressions:

```
def fix_select: Expr ⇒ Ans = e ⇒ e match {
  case Lit(_) | Var(_) | Lam(_, _) ⇒ eval(fix_select)(e)
  case _ ⇒ fix_cache(e)
}
```

Partially-static Data. Our treatment of binding-times is coarse-grained: `Exprs` are static, the rest of the components are all dynamic. But this is not always true, because the static data have to be used somewhere with the dynamic operations. Partially-static data is a way to improve binding-times and optimize the generated code. For example, to left-fold a static singleton set (often appears in `SetT`), e.g., `Set(x).fold(init)(f)` where `x` and `init` are staged values, a naive code generator would faithfully generate code that applies `fold` with the set and function `f`. But we can also utilize algebraic properties of `fold` to generate cheaper code, e.g., `Set(x).fold(init)(f) = f(init, x)`. Since the function `f` is known at the current stage, we completely eliminate the fold operation and function application. We apply several rewritings enabled by partially-static patterns, such as for `Set`, `Map`, and `Tuple`. This optimization greatly reduces the size of residual programs.

7 DISCUSSION

We have gradually presented the confluence of specialization and abstraction of concrete interpreters from an operational perspective. In this section, we review and summarize our recipe to achieve the staged abstract interpreter and discuss correctness and different design choices.

7.1 Summarizing the Approach

We summarize our approach to staging an abstract interpreter as follows:

- First, we construct a generic interpreter that abstracts over binding times, value domains, and primitive operations. In this paper, the generic interpreter is implemented in monadic style; therefore, the semantics can be encapsulated into monads.
- Then, we implement an unstaged abstract interpreter modularly using the appropriate monad stack. This step has been explored in the previous literature.
- Finally, we replace the monad stack with a staged monad stack, and refactor related primitive operations. Such staged monads operate on staged data types, i.e., next-stage values.

Monadic interpreters are known to be able to decouple the interpretation procedure and the underlying semantics. The key insight in this paper is that by making the monadic interpreter stage polymorphic [Amin and Rompf 2018; Ofenbeck et al. 2017], the abstract interpreter can be extended to generate efficient code. The underlying semantics and staging are two orthogonal dimensions. It is important to note that the computation encapsulated by the monads are not staged: only data (such as sets and maps) are staged. All the monadic computation, i.e., functions passed to the monadic bind operation `flatMap`, are statically known. This is why we can eliminate the monadic layer and its associated overhead.

What has been eliminated? In the generated code, all primitive operations (such as `eval`, `fix`, `ap_clo`, etc.) and monadic operations (such as `flatMap` and `map`) are eliminated. The residual program consists of statements and expressions that purely manipulate the environment, store, and

two caches, whose underlying representations are all `Map[K,V]`. We also have several operations on tuples and lists, which are residualized from the internal code fragments of the monads.

7.2 Correctness

Soundness is the central concern of static analysis, and as such, is vital for prospective users. Our approach does not interact with the soundness of the analysis, i.e, if the unstaged one does not produces false negative result, the staged one also does not. This soundness preservation indeed relies on the correctness of the staged implementations and the underlying MSP system. We now briefly examine how this is achieved. Note that the rationale listed here is based on empirical evidence; a rigorous proof of soundness preservation of our approach remains an open challenge.

- In general, syntactic MSP systems based on quotations (e.g., MetaML/MetaOCaml) cannot guarantee that the generated code preserves the meaning after staging. In other words, the generated code may not preserve the evaluation order or may contain undesired duplications of code. In this paper, we use the LMS framework, which 1) checks binding-times by checking types which is enabled by the Scala compiler, and 2) generates code in administrative normal form [Flanagan et al. 1993] that preserves the execution order within a stage [Rompf 2016].
- As shown in the previous sections, the generic interpreter is untouched and shared by the four artifacts. This allows the programmer to check the implementation of staged monads and primitive operations, as well as their correctness modularly.
- We build the staged monads and staged data structures in a correct-by-construct way that directly corresponds to their unstaged versions. For instance, the staged data structures we use here are simply black boxes that wrap the data structures in the Scala library.
- In our experiments, the staged abstract interpreter produces the same result as the unstaged one on all benchmark programs we tested.

7.3 Alternatives

Monadic-style vs Direct-style. We use a monadic interpreter throughout the paper, but using monadic-style is not the only choice for staging. One can inline the monadic operations and obtain an abstract interpreter in continuation-passing style, or even translate back to a direct-style that may use explicit side-effects such as mutations. In either case, we can still apply the staging idea to the abstract interpreter and remove the interpretation overhead. However, monads allow the staged abstract interpreter to be implemented in a modular and extensible way.

Big-step vs Small-step. We have implemented a big-step, compositional abstract interpreter in monadic style, where *compositional* means that every recursive call of `eval` is applied to the proper substructures of the current syntactic parameters [Jones 1996]. This compositionality ensures that specialization can be done by unfolding, as well as guarantees the termination of the specialization procedure. It is also possible to specialize a small-step operational abstract semantics through abstract compilation [Boucher and Feeley 1996] – as Johnson et al. [2013] presented for optimizing Abstract Abstract Machines. However, the generated abstract bytecode still requires another small-step abstract machine to execute, which is an additional engineering effort.

8 CASE STUDIES

In Section 6, we use a toy language to show that staging an abstract interpreter is feasible and that a staged artifact can be derived by combining the staged concrete interpreter and unstaged abstract interpreter. In this section, we conduct several case studies to further show that this methodology is practically useful and widely applicable to a diverse set of analyses.

```
type CompAnalysis = Store ⇒ Store
def compProgram(prog: Expr): CompAnalysis = compCall(prog)
def compCall(call: Expr): CompAnalysis = call match {
  case Letrec(bds, body) ⇒
    val C1 = compCall(body); val C2 = compArgs(bds.map(_.value))
    (σ: Store) ⇒ C1(C2(σ.update(bds.map(_.name),
       bds.map(b ⇒ Set(b.value.asInstanceOf[Lam])))))
  case App(f, args) ⇒
    val C1 = compApp(f, args); val C2 = compArgs(args)
    (σ: Store) ⇒ C1(C2(σ))
}
def compApp(f: Expr, args: List[Expr]): CompAnalysis = f match {
  case Var(x) ⇒ (σ: Store) ⇒
    analyzeAbsApp(args, σ(x), σ)
  case Op(_) ⇒ compArgs(args)
  case Lam(vars, body) ⇒
    val C = compCall(body)
    (σ: Store) ⇒ C(σ.update(vars, args.map(primEval(_, σ))))
}
def compArgs(args: List[Expr]): CompAnalysis = args match {
  case Nil ⇒ (σ: Store) ⇒ σ
  case (arg@Lam(vars, body))::rest ⇒
    val C1 = compCall(body); val C2 = compArgs(rest)
    (σ: Store) ⇒ C2(C1(σ))
  case _::rest ⇒ compArgs(rest)
}
```

```
def analyzeProgram(prog: Expr, σ: Rep[Store]): Rep[Store] =
  analyzeCall(prog, σ)
def analyzeCall(call: Expr, σ: Rep[Store]): Rep[Store] =
  call match {
    case Letrec(bds, body) ⇒
      val σ_* = σ.update(bds.map(_.name),
        bds.map(b ⇒ Set(b.value.asInstanceOf[Lam])))
      val σ_** = analyzeArgs(bds.map(_.value), σ_*)
      analyzeCall(body, σ_**)
    case App(f, args) ⇒ analyzeApp(f, args, analyzeArgs(args, σ))
  }
def analyzeApp(f: Expr, args: List[Expr], σ: Rep[Store]):
  Rep[Store] = f match {
    case Var(x) ⇒ analyzeAbsApp(args, σ(x), σ)
    case Op(_) ⇒ analyzeArgs(args, σ)
    case Lam(vars, body) ⇒
      val σ_* = σ.update(vars, args.map(primEval(_, σ)))
      analyzeCall(body, σ_*)
  }
def analyzeArgs(args: List[Expr], σ: Rep[Store]): Rep[Store] =
  args match {
    case Nil ⇒ σ
    case Lam(vars, body)::rest ⇒
      analyzeArgs(rest, analyzeCall(body, σ))
    case _::rest ⇒ analyzeArgs(rest, σ)
  }
```

Fig. 7. Comparison of AC (left) and SAI (right). Only core code are shown.

8.1 Abstract Compilation à la Staging

We first revisit a similar technique called *abstract compilation* (AC). AC was introduced by Boucher and Feeley [1996] as an implementation technique for abstract interpretation-based static analyses. Similar to the present paper, the idea of AC was inspired by partial evaluation: the program is known statically and the interpretive overhead can be eliminated. In AC, the compiled analysis can be represented by either text or closures (higher-order functions). The minute difference is that the closures can be executed immediately, while the textual programs need to be compiled and loaded.

Specifically, Boucher and Feeley showed how to compile a monovariant control-flow analysis [Shivers 1988, 1991] for continuation-passing style (CPS) programs. The analyzer is a big-step control-environment abstract interpreter. By applying AC, every function in the analyzer is refactored to return a closure only taking an environment argument. Therefore the overhead of traversing the AST of input program is eliminated.

We show that AC can be understood and implemented as an instance of staging abstract interpreters. We first revisit the original implementation of abstract compilation for 0-CFA, and then reproduce their result by simply adding stage annotations. The program generated by our approach provides approximately the same improvement with regards to speed, but without changing a single line of the analyzer program relying on type-based staging annotations. With Scala's type inference, we only need to add staging annotations for function argument types and return types. However, AC requires more engineering effort, i.e., rewriting the whole analyzer into the closure-generation form. Moreover, as shown in Section 6.4, our approach is able to not only remove the interpretive overhead, but also enable several optimizations such as specialized data structures.

Original AC. The analysis presented by Boucher and Feeley is a 0-CFA for a small CPS language consisting of abstractions, applications, recursion, and primitive operators. Figure 7 (left) shows the AC implementation: different syntactic constructs are handled by different functions (e.g., `compCall` and `compApp`). All original functions take a store as regular argument. After refactoring to AC, every function returns a value of type `CompAnalysis`, which is a function value that takes and returns stores. After the first run of the analysis, we have traversed the AST and obtained a

single closure that takes a store and returns a store. Therefore, the ASTs are treated as static terms, and the stores are dynamic terms. Finally, we may invoke the compiled closure with an initial store to complete the analysis.

AC through Staging. On the other side, our approach works equivalently through staging: the ASTs are static, whereas stores are dynamic. Therefore, we annotate the type `Store` with `Rep` (Figure 7 (right)). In fact, the only changes of our approach using the LMS framework are the types of `Store` to `Rep[Store]`, since they will be known at the next stage. There is no need for other changes. The generated programs consist entirely of looks-up and updating operations w.r.t the store. The functions at the current stage are eliminated in the residual program; therefore, we do not generate higher-order functions, which provides additional performance improvement. Nevertheless, the LMS framework provides support for necessary data structures, such as for `Map`. We consider these efforts reusable and completely orthogonal to the implementation of the analysis.

8.2 Control-Flow Analysis

To demonstrate that the SAI approach is applicable to various analyses, we now extend and refactor the staged abstract interpreter we presented in Section 6 to several variants of control-flow analyses (CFA). We first discuss the calling-context sensitive analysis for staged analyzers by tweaking the allocation strategy. Then, we describe how to implement store-widening with the staged monads, which is a common optimization in CFA. Finally, we integrate abstract garbage collection with staging. All of these extensions are orthogonal to staging.

8.2.1 Context-Sensitivity. The abstract addresses we introduced in Section 6 are identical to the variable names, which is a context-insensitive schema. Through tweaking the address allocation strategy, we can achieve various context-sensitive analyses [Gilray et al. 2016a]. Here, we demonstrate a calling-context sensitive analysis, i.e., k-CFA-like analysis [Horn and Might 2012], can be implemented in our staged abstract interpreter.

```
type Time = List[Expr]
case class KCFAAddr(x: Ident, time: Time) extends Addr
```

The timestamp is a finite list of expressions that tracks the k-most recent calling contexts. The definition of abstract addresses `KCFAAddr` is also changed to include an identifier and the time when it gets allocated, meaning that this address points to some values under such calling context. If k is 0, we obtain a monovariant analysis as demonstrated before; if $k > 0$, we obtain a family of analyses with increasing precision. Note that when $k = 0$, the address space is statically determined by the set of variables appeared in the program, and we may exploit it to optimize the generated code. But when $k > 0$, the address generation happens at the next stage, i.e., analysis time.

```
type AnsM[T] = RepReaderT[RepStateT[RepStateT[ // add Time with a RepStateT
                 RepSetReaderStateM[Cache, Cache, ?], Time, ?], Store, ?], Env, T]
type Result = (Rep[Set[(Value, Store, Time)]], Rep[Cache])
```

We then integrate the timestamp into our monad stack by adding a staged `StateT` monad. The result type is accordingly updated to contain the set of tuples, where the field of `Time` is added.

```
def tick(e: Expr): AnsM[Unit] = for {
  τ ← get_time; u ← liftM(StateTMonad.put((e::τ).take(k)))
} yield u
def alloc(x: String): AnsM[Addr] = for {
  τ ← get_time
} yield unit(KCFAAddr(x, τ)) // we use unit to turn a current-stage constant to a next-stage value
```

Every time when we call the `eval` function, we also refresh the timestamp by calling the `tick` function, which updates the timestamp in the state monad. The `tick` function is implemented as prepending the current control expression `e` being evaluated to the existing calling context, and

then taking the first k elements from the list. The type `Config` used in the caching algorithm is also changed to include timestamps.

8.2.2 Store-Widened Analysis. A commonly-used technique to improve the running time of control-flow analyses is store-widening. As shown by Darais et al. [2017, 2015], a store-widening analysis can be easily implemented in the monadic abstract interpreter by swapping the `StateT` and the `SetT` monad transformers. This optimization is orthogonal to staging. Within the monadic interpreter framework, we just need to swap the same *staged* version of monad transformers. After this adjustment, conceptually, we have the following monad stack:

```
type AnsM[T] = RepReaderT[RepSetT[RepStateT[
                 RepReaderT[RepStateT[IdM, Cache, ?], Cache, ?], Store, ?], ?], Env, T]
```

8.2.3 Abstract Garbage Collection. Abstract garbage collection is a technique to reclaim unreachable addresses in the store while performing the abstract interpretation [Might and Shivers 2006]. Darais et al. [2017] showed that, for big-step monadic abstract interpreters, we may add a `Reader` monad to track the set of root addresses and compute the reachable addresses every time when we obtain a value. Here, we adopt the idea with staged monads:

```
type AnsM[T] = RepReaderT[RepReaderT[
                 RepStateT[RepSetReaderStateM[Cache, Cache, ?], Store, ?],
               Set[Addr], ?], Env, T] //add a reader monad for Set[Addr]
```

The skeleton of a staged version of abstract garbage collection implementation is shown in Figure 8. We refactor the `fix` function with caching (used in Section 6) to `fix_gc` (the caching part is omitted in the presented code snippet). Several auxiliary functions are used: `ask_root` retrieves the current root addresses from the monad stack; `root` computes the reachable addresses given a value; and `gc` performs garbage collection according to the given reachable addresses and returns a new store. We also need to compute the root addresses for compound expressions, which is elided in the code snippet; for the details of this part, readers may refer to Darais et al. [2017].

```
def fix_gc(ev: (Expr ⇒ Ans) ⇒ (Expr ⇒ Ans))(e: Expr): Ans = for {
  ψ ← ask_roots ... // omit code for retrieving ρ, σ, in and out
  val res: Rep[(Set[(Value,Store)], Cache)] = ...  // omit code for checking caches
  _ ← put_out_cache(res._2);  vs ← lift_nd(res._1)
  σ ← gc(ψ ⊔ root(vs._2));     _ ← put_store(σ)  // gc performs abstract garbage collection
} yield v
```

Fig. 8. `fix_gc` for abstract garbage collection

8.3 Modular Staged Analysis for Free

One of the challenges of modern static analysis is that programs usually depend on large libraries, and analyzing these libraries is time-consuming [Toman and Grossman 2017]. If we are able to analyze programs and libraries separately and reuse the results without losing precision, we can reduce the overhead of repeatedly analyzing libraries. Some static analyzers compute a summary for a function or a module, which can be reused later. However, when lacking context information, those analyses can be too conservative or unsound. In this section, we show that meta-programming can provide a trade-off between summary-based modular analyses and slow whole-program analyses, by utilizing an existing whole-program analyzer.

As we have already seen in Section 6, the primitive operation `close` can denote a λ-term to a next-stage Scala function. The specialization of the staged abstract interpreter to a λ-term is also independent of other parts of the analyzed program.

```
def close(ev: Eval ⇒ Ans)(λ: Expr, ρ: Rep[Env]): Rep[Value]
```

To illustrate the idea concretely, we consider a library that contains a function `map` and use the Scheme language as example:

```
(define (map xs f) (if (null? xs) '() (cons (f (car xs)) (map (cdr xs) f))))
```

The specialization (0-CFA and without store-widening) of `close` to `map` generates the following next-stage Scala code, that is, the function `x39`:

```
val x39 = {(x40: List[Set[AbsValue]],          // list of argument values
            x41: Map[Addr, Set[AbsValue]],     // store
            x42: Map[Config, Set[ValSt]],      // in cache
            x43: Map[Config, Set[ValSt]]) ⇒    // out cache
  val x50 = List[Addr](ZCFAAddr("xs"), ZCFAAddr("f")) // addresses of arguments
  val x51 = x50.zip(x40)                              // addresses paired with their values
  val x61 = x51.foldLeft (x41) { case (x52, x53) ⇒    // preparing a new store that joins
    val x54 = x53._1;  val x55 = x53._2               //   the arguments and their values
      val x58 = x52.getOrElse(x54, x57)               //   into the latest store, initially x41
      val x59 = x58.union(x55);  x52 + (x54 ↦ x59) }
  ... }
```

We borrow the notation from traditional modular analysis [Cousot and Cousot 2002]: such a generated function is a modularly reusable artifact, i.e., independent of its context. The generated functions already eliminate the interpretation overhead and monadic overhead via staging, and can therefore be considered as partial summaries. However, we still need to run the generated function at the next stage to obtain the analysis result. At the next stage, we can provide different abstract arguments, stores, and caches, depending on the context where it is called.

Compared with traditional summary-based analysis, our partial summaries are represented by executable functions that are parameterized to the context information. In our approach, the transition from a whole-program analyzer to a modular-staged analyzer is mechanized and for-free via staging. We do not need to design any intermediate representation or data structures of summaries. In some scenarios, traditional summary-based analyses are able to carry more information within a module, e.g., the summary of their internal submodules. The staged approach presented here may still need to run through the fixed-point for submodules.

9 PERFORMANCE EVALUATION

To evaluate how much performance can be improved by the staging, we scale our toy staged abstract interpreter to support a large subset of the Scheme language and evaluate the analysis time against the unstaged versions. The result shows that the staged versions are, on average, 10+ times as fast as the unstaged version, as evidence of our *abstraction without regret* approach.

Implementation and Evaluation Environment. We implement the abstract interpreters for control-flow analysis, i.e., a value-flow analysis that computes which λ-terms can be called at each call-site. With other suitable abstract domains, the abstract interpreter can also be extended to other analyses. A front-end desugars Scheme to a small core language that makes the analyzer easy to implement. We implement two common monovariant 0-CFA-like analyses: one is equipped with store-widening (0-CFA-SW for short), the other one is not (0-CFA for short). It is traditionally expected that the store-widened analysis be faster than the alternative. Our prototype implementation currently generates Scala code. The generated Scala code will be compiled and executed on the JVM, though it is possible to generate C/C++ code in the future. In the experiments, we implement several optimizations mentioned in Section 6.4, specifically, including the selective caching scheme and IR-level rewriting rules (for pairs, lists, sets, and maps) that exploit partially-static data. These optimizations are useful to reduce the size of generated code and improve performance.

We use Scala 2.12.8 and Oracle JVM 1.8.0-191 [3], running on an Ubuntu 16.04 LTS machine. All our evaluations were performed on a machine with 4 Intel Xeon Platinum 8168 CPUs at 2.7GHz and 3

[3]The options for running the JVM are set to the following: `-Xms2G -Xmx8G -Xss1024M -XX:MaxMetaspaceSize=2G -XX:ReservedCodeCacheSize=2048M`

program	#AST	unstaged	staged	$\frac{\text{unstaged}}{\text{staged}}$	unstaged	staged	$\frac{\text{unstaged}}{\text{staged}}$
		w/o store-widening			w/ store-widening		
`fib`	32	3.288 ms	0.154 ms	21.33x	1.434 ms	0.098 ms	14.62x
`rsa`	451	238.171 ms	23.333 ms	10.20x	11.977 ms	1.197 ms	10.00x
`church`	120	61.001 s	4.277 s	14.26x	2.338 ms	0.534 ms	4.37x
`fermat`	310	23.540 ms	2.885 ms	8.05x	7.146 ms	0.915 ms	7.81x
`mbrotZ`	331	665.456 ms	66.070 ms	10.07x	11.008 ms	1.476 ms	7.45x
`lattice`	609	29.230 s	2.627 s	11.12x	16.432 ms	2.427 ms	6.76x
`kcfa-worst-16`	182	44.431 ms	3.211 ms	13.83x	4.425 ms	0.850 ms	5.20x
`kcfa-worst-32`	358	284.268 ms	9.065 ms	31.35x	10.109 ms	1.661 ms	6.08x
`kcfa-worst-64`	710	2.165 s	0.0425 s	50.85x	23.269 ms	3.312 ms	7.02x
`solovay-strassen`	523	5.078 s	0.766 s	6.62x	18.757 ms	3.142 ms	5.96x
`regex`	550	-	-	-	6.803 ms	1.088 ms	6.24x
`matrix`	1732	-	-	-	85.611 ms	9.297 ms	9.20x

Fig. 9. Evaluation result for monovariant control-flow analysis.

TB of RAM. Although the machine has 96 cores in total, the abstract interpreters are single-threaded. To minimize the effect of warm-up from the HotSpot JIT compiler, all the experiments are executed 20 times and we report the statistical median value of the running times. We set a 5 minute timeout.

Benchmarks. The benchmark programs we used in the experiment are collected from previous papers [Ashley and Dybvig 1998; Johnson et al. 2013; Vardoulakis and Shivers 2011] and existing artifacts [4] for control-flow analysis. Some of the benchmarks are small programs designed for experiments, such as the `kcfa-worst-`n series, which are intended to challenge k-CFA; while some other benchmarks are from real-world applications, for example, the RSA public key encryption algorithm `rsa`. In Figure 9, we report the number of AST nodes after parsing and desugaring (excluding comments) as a proper measurement of program size.

Result. Figure 9 shows the evaluation result comparing the performance improved by staging on the two monovariant CFAs. The *unstaged* and *staged* columns show the median time (excluding pre-compilation) to finish the analysis. The column $\frac{\text{unstaged}}{\text{staged}}$ shows the improvement. The dash - in some cells indicates timeout. As expected, the staged version significantly outperforms the unstaged version on all benchmarks. For 0-CFA without store-widening, we observe an average speedup of 17 times; for 0-CFA with store-widening, the average speedup is 7 times. The highest speedup among all the tests is observed on benchmark `kcfa-worst-64`. Although the program size of `kcfa-worst-64` is large, it has a rather simple structure which only involves booleans, λ-terms, and applications. We conjecture that the simple domain and structure allow the generated code to be efficiently optimized by the JVM. However, on the corresponding 0-CFA-SW, we do not observe a significant speedup. Due to the complexity of generated code and next-stage running platform (JVM), we leave the detailed empirical analysis of which static analysis can benefit more from the specialization as a future work.

10 RELATED WORK

Optimizing Static Analysis Through Specialization. The underlying idea in this paper is closely related to abstract compilation (AC) [Boucher and Feeley 1996]: removing the interpretative overhead of traversing the syntax tree by specialization. The residual programs of AC can be either textual (using quasi-quotations) or closures (using higher-order functions). Our approach is more flexible between representations and requires less engineering efforts to refactor the analyzer (Section 8.1).

[4] https://github.com/ilyasergey/reachability

Johnson et al. [2013] adapted closure generation to optimize a small-step abstract interpreter in state-transition style. The program is first compiled to an "abstract bytecode" IR. The IR consists of higher-order functions, which will be executed later on an abstract abstract machine for that IR. Damian [1999] provided a formal treatment for abstract compilation and control-flow analysis, as well as proofs to establish the correctness of a certified specialized analyzer. Amtoft [1999] applied partial evaluation for constraint-based control-flow analysis.

Splitting an analysis into multiple stages has also been studied for other forms of static analysis. Hardekopf and Lin [2011] applied staging to a flow-sensitive pointer analysis. The first stage is to analyze the programs and obtain a sparse representation, and then the second stage conducts a flow-sensitive analysis by refining the first one. Many other forms of static analysis are implemented by first collecting constraints from programs and then using a solver to discharge the proof obligations or find the fixed-point. Notable examples include SAT- and SMT-based program analysis [Gulwani et al. 2008] and Datalog-based points-to analysis [Smaragdakis and Balatsouras 2015]. Such analyses can be considered having stage distinctions. One piece of future work is to explore whether they can be implemented and optimized using meta-programming. Soufflé, one of the state-of-the-art Datalog engines commonly used in points-to analysis [Antoniadis et al. 2017], uses Futamura projections for efficient implementation [Jordan et al. 2016]. However, it only specializes the solving engine to Datalog rules, instead of a full pipeline from a program AST to a specialized solver.

Recent work shows that numerical abstract domains can be specialized to the static program structure, although not by ways of staging. For example, decomposing polyhedrons [Singh et al. 2017a,b] to smaller ones depending on the variables involved in a statement can significantly improve the running time and space consumption of abstract transformers.

Construction of Abstract Interpreters. Abstract interpretation is a semantics-based approach of sound static analyses by approximation [Cousot 1999; Cousot and Cousot 1977]. As for building semantic artifacts, the Abstracting Abstract Machines (AAM) [Horn and Might 2010, 2012] approach shows that abstract interpreters can be derived systematically from concrete semantic artifacts based on operational semantics. The AAM approach is closely related to control-flow analysis for higher-order languages [Midtgaard 2012; Shivers 1991]. Using monads to construct abstract interpreters was explored by Darais et al. [2017, 2015]; Sergey et al. [2013]. The unstaged abstract interpreter in this paper follows the abstracting definitional interpreters (ADI) approach [Darais et al. 2017]. Similar to ADI, reconstructing big-step abstract interpreters with delimited continuations was shown by Wei et al. [2018]. Keidel et al. [2018] presented a modular, arrow-based abstract interpreter that allows soundness proofs to be constructed compositionally. Cousot and Cousot [2002] proposed an abstract interpretation framework for modular analysis. In Section 8.3, we borrow the notation of modular analysis. However, the staged abstract interpreter does not generate complete summaries for modules; instead, it specializes the analysis modularly. Similarly, AC is considered as an example of this kind of modular analyses [Cousot and Cousot 2002].

Control-flow Analysis. Our case studies and evaluation focus on control-flow analysis for functional languages [Midtgaard 2012; Shivers 1991]. In general, k-CFA (where $k > 0$) is considered impractical to be used as a compiler pass, due to its EXPTIME lower bound [Van Horn and Mairson 2008]. However, several variants of 0-CFA were developed and shown to be useful in practice [Adams et al. 2011; Ashley and Dybvig 1998; Bergstrom et al. 2014; Reppy 2006].

Two-level Semantics. The idea of reinterpreting the semantics as abstract interpretation can be traced to Nielson's two-level semantics [Nielson 1989]. Using two-level semantics for code generation was also explored by Nielson and Nielson [1988]. Sergey et al. [2013]'s monadic abstract interpreter is closely related to the two-level semantics: the use of a generic interface with monads

and then reinterpreting it with different semantics is already two-level. Instead of focusing on semantics, this paper shows how a staged analyzer can be built and used to increase efficiency of static analyses. We augment the monadic abstract interpreter with binding-time polymorphism, using code generation to produce efficient low-level code. The presented work can be considered as a two-dimensional application of the two-level semantics.

Partial Evaluation and Multi-stage Programming. Partial evaluation is an automatic technique for program specialization [Jones 1996; Jones et al. 1993]. In this paper, we use mutli-stage programming to specialize programs. The Lightweight Modular Staging (LMS) framework [Rompf and Odersky 2010] we used in the paper relies on type-level stage annotations. Other notable MSP systems using syntactic quotations exist in the ML family, e.g., MetaML [Taha and Sheard 1997] and MetaOCaml [Calcagno et al. 2003; Kiselyov 2014]. Compared with other approaches using syntactic quotations, LMS provides guarantees for preserving evaluation orders within a stage. Ofenbeck et al. [2017] and Amin and Rompf [2018] discussed staging for generic programming as well as stage polymorphism in LMS. Multi-stage programming has been widely used to improve the performance in many other domains, such as optimizing compilers or domain-specific languages [Carette et al. 2009; Rompf et al. 2015, 2014; Sujeeth et al. 2014, 2013], numerical computation [Aktemur et al. 2013; Kiselyov 2018], generic programming [Ofenbeck et al. 2017; Yallop 2017], data processing [Jonnalagedda et al. 2014; Kiselyov et al. 2017], query compilation in databases [Essertel et al. 2018; Rompf and Amin 2015; Tahboub et al. 2018], etc.

As a source of inspiration of this paper, the Futamura projections reveal a close relation between interpreters and compilers, which was originally proposed by Futamura [1971] (reprinted as [Futamura 1999]). Amin and Rompf [2018] considered a tower of concrete interpreters and showed how to collapse them by using MSP – it would be interesting to explore this idea for multiple abstract interpreters [Cousot et al. 2019; Giacobazzi et al. 2015]. Carette and Kiselyov [2005] discussed a monad with let-insertion and memoization for code generation, using Gaussian Elimination in MetaOCaml as an example. In this work, let-insertion is handled by the ANF transformation in LMS, and the memoizing monad is just a combination of a reader monad and a state monad operating on staged data. Similar to the ideas in this paper, specializing monadic interpreters was explored by Danvy et al. [1991] and Sheard et al. [1999]. However, these work are different from the performance motivation here.

11 CONCLUSION

We present a simple and elegant framework that extends the first Futamura projection to abstract interpreters. To realize it, we design a generic interpreter that abstracts over binding times and value domains. By instantiating the generic interpreter with the staged monads for abstract semantics, the staged abstract interpreter can be readily derived. The meta-programming approach brings new insights and implementation techniques to optimize static analyzers via code generation. Optimizations can be achieved orthogonally to the interpreter and abstract semantics. In our evaluation, staging improves the performance of running time by an order of magnitude.

ACKNOWLEDGMENTS

We thank the reviewers of PLDI '19 and OOPSLA '19, as well as James Decker, Guanhong Tao, Grégory Essertel, Fei Wang, Qianchuan Ye, Yapeng Ye, and Le Yu for their feedback on early drafts. We thank David Van Horn and David Darais for their encouragement on this work. This work was supported in part by NSF awards 1553471, 1564207, 1918483, DOE award DE-SC0018050, as well as gifts from Google, Facebook, and VMware.

REFERENCES

Michael D. Adams, Andrew W. Keep, Jan Midtgaard, Matthew Might, Arun Chauhan, and R. Kent Dybvig. 2011. Flow-sensitive Type Recovery in Linear-log Time. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. ACM, New York, NY, USA, 483–498. https://doi.org/10.1145/2048066.2048105

Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*. ACM, 8–19. https://doi.org/10.1145/888251.888254

Baris Aktemur, Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2013. Shonan challenge for generative programming: short position paper. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM 2013, Rome, Italy, January 21-22, 2013*, Elvira Albert and Shin-Cheng Mu (Eds.). ACM, 147–154. https://doi.org/10.1145/2426890.2426917

Nada Amin and Tiark Rompf. 2018. Collapsing towers of interpreters. *PACMPL* 2, POPL (2018), 52:1–52:33. https://doi.org/10.1145/3158140

Torben Amtoft. 1999. Partial evaluation for constraint-based program analyses. *BRICS Report Series* 6, 45 (1999).

Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. 2017. [orting Doop to Soufflé: A Tale of Inter-engine Portability for Datalog-based Analyses. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP 2017)*. ACM, New York, NY, USA, 25–30. https://doi.org/10.1145/3088515.3088522

J. Michael Ashley and R. Kent Dybvig. 1998. A practical and flexible flow analysis for higher-order languages. *ACM Transactions on Programming Languages and Systems* 20, 4 (1998), 845–868. https://doi.org/10.1145/291891.291898

Lars Bergstrom, Matthew Fluet, Matthew Le, John Reppy, and Nora Sandler. 2014. Practical and Effective Higher-order Optimizations. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 81–93. https://doi.org/10.1145/2628136.2628153

Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the Meta-level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS '09)*. ACM, New York, NY, USA, 18–25. https://doi.org/10.1145/1565824.1565827

Dominique Boucher and Marc Feeley. 1996. Abstract Compilation: A New Implementation Paradigm for Static Analysis. In *Proceedings of the 6th International Conference on Compiler Construction (CC '96)*. Springer-Verlag, London, UK, UK, 192–207. http://dl.acm.org/citation.cfm?id=647473.727587

Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In *Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings (Lecture Notes in Computer Science)*, Frank Pfenning and Yannis Smaragdakis (Eds.), Vol. 2830. Springer, 57–76. https://doi.org/10.1007/978-3-540-39815-8_4

Jacques Carette and Oleg Kiselyov. 2005. Multi-stage Programming with Functors and Monads: Eliminating Abstraction Overhead from Generic Code. In *Generative Programming and Component Engineering*, Robert Glück and Michael Lowry (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 256–274.

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. https://doi.org/10.1017/S0956796809007205

Paul Chiusano and Rnar Bjarnason. 2014. *Functional Programming in Scala* (1st ed.). Manning Publications Co., Greenwich, CT, USA.

P. Cousot. 1999. The Calculational Design of a Generic Abstract Interpreter. In *Calculational System Design*, M. Broy and R. Steinbrüggen (Eds.). NATO ASI Series F. IOS Press, Amsterdam.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

P. Cousot and R. Cousot. 1979. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, San Antonio, Texas, 269–282.

Patrick Cousot and Radhia Cousot. 2002. Modular Static Program Analysis. In *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings (Lecture Notes in Computer Science)*, R. Nigel Horspool (Ed.), Vol. 2304. Springer, 159–178. https://doi.org/10.1007/3-540-45937-5_13

Patrick Cousot, Roberto Giacobazzi, and Francesco Ranzato. 2019. A^2I: Abstract^2 Interpretation. *Proc. ACM Program. Lang.* 3, POPL, Article 42 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290355

Daniel Damian. 1999. Partial evaluation for program analysis. *Progress report, BRICS PhD School, University of Aarhus* (1999).

Olivier Danvy, Jürgen Koslowski, and Karoline Malmkjær. 1991. Compiling monads. *Kansas State University, Manhattan, Kansas, Tech. Rep. CIS-92-3* (1991).

David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. 2017. Abstracting definitional interpreters (functional pearl). *PACMPL* 1, ICFP (2017), 12:1–12:25. https://doi.org/10.1145/3110256

David Darais, Matthew Might, and David Van Horn. 2015. Galois Transformers and Modular Abstract Interpreters: Reusable Metatheory for Program Analysis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 552–571. https://doi.org/10.1145/2814270.2814308

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

Grégory M. Essertel, Ruby Y. Tahboub, James M. Decker, Kevin J. Brown, Kunle Olukotun, and Tiark Rompf. 2018. Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 799–815. https://www.usenix.org/conference/osdi18/presentation/essertel

Mattias Felleisen and D. P. Friedman. 1987. A Calculus for Assignments in Higher-order Languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '87)*. ACM, New York, NY, USA, 314–. https://doi.org/10.1145/41625.41654

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI '93)*. ACM, New York, NY, USA, 237–247. https://doi.org/10.1145/155090.155113

Yoshihiko Futamura. 1971. Partial evaluation of ccomputation process-an approach to a compiler-compiler. *Systems, Compurters, Controls* 25 (1971), 45–50.

Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12, 4 (01 Dec 1999), 381–391. https://doi.org/10.1023/A:1010095604496

Roberto Giacobazzi, Francesco Logozzo, and Francesco Ranzato. 2015. Analyzing Program Analyses. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 261–273. https://doi.org/10.1145/2676726.2676987

Thomas Gilray, Michael D. Adams, and Matthew Might. 2016a. Allocation characterizes polyvariance: a unified methodology for polyvariant control-flow analysis. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 407–420. https://doi.org/10.1145/2951913.2951936

Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. 2016b. Pushdown control-flow analysis for free. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 691–704. https://doi.org/10.1145/2837614.2837631

Github, Inc. 2019. Semantic Analyzer. https://github.com/github/semantic.

Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. 2008. Program Analysis As Constraint Solving. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 281–292. https://doi.org/10.1145/1375581.1375616

Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011*. IEEE Computer Society, 289–298. https://doi.org/10.1109/CGO.2011.5764696

David Van Horn and Matthew Might. 2010. Abstracting abstract machines. In *Proceeding 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. https://doi.org/10.1145/1863543.1863553

David Van Horn and Matthew Might. 2012. Systematic abstraction of abstract machines. *J. Funct. Program.* 22, 4-5 (2012), 705–746. https://doi.org/10.1017/S0956796812000238

J. Ian Johnson, Nicholas Labich, Matthew Might, and David Van Horn. 2013. Optimizing Abstract Abstract Machines. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 443–454. https://doi.org/10.1145/2500365.2500604

Neil D. Jones. 1996. What not to do when writing an interpreter for specialisation. In *Partial Evaluation*, Olivier Danvy, Robert Glück, and Peter Thiemann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 216–237.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation.* Prentice Hall.

Manohar Jonnalagedda, Thierry Coppey, Sandro Stucki, Tiark Rompf, and Martin Odersky. 2014. Staged parser combinators for efficient data processing. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 637–653. https://doi.org/10.1145/2660193.2660241

Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 422–430.

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

Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml - System Description. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings (Lecture Notes in Computer Science)*, Michael Codish and Eijiro Sumii (Eds.), Vol. 8475. Springer, 86–102. https://doi.org/10.1007/978-3-319-07151-0_6

Oleg Kiselyov. 2018. Reconciling Abstraction with High Performance: A MetaOCaml approach. *Foundations and Trends® in Programming Languages* 5, 1 (2018), 1–101. https://doi.org/10.1561/2500000038

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 285–299. http://dl.acm.org/citation.cfm?id=3009880

Sheng Liang, Paul Hudak, and Mark P. Jones. 1995. Monad Transformers and Modular Interpreters. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 333–343. https://doi.org/10.1145/199448.199528

Stefan Marr and Stéphane Ducasse. 2015. Tracing vs. Partial Evaluation: Comparing Meta-compilation Approaches for Self-optimizing Interpreters. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 821–839. https://doi.org/10.1145/2814270.2814275

Jan Midtgaard. 2012. Control-flow Analysis of Functional Programs. *ACM Comput. Surv.* 44, 3, Article 10 (June 2012), 33 pages. https://doi.org/10.1145/2187671.2187672

Matthew Might and Olin Shivers. 2006. Improving Flow Analyses via Gamma-CFA: Abstract Garbage Collection and Counting. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP '06)*. ACM, New York, NY, USA, 13–25. https://doi.org/10.1145/1159803.1159807

Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

Flemming Nielson. 1989. Two-level semantics and abstract interpretation. *Theoretical Computer Science* 69, 2 (1989), 117 – 242. https://doi.org/10.1016/0304-3975(89)90091-1

Flemming Nielson and Hanne Riis Nielson. 1988. Two-level semantics and code generation. *Theoretical Computer Science* 56, 1 (1988), 59 – 133. https://doi.org/10.1016/0304-3975(86)90006-X

Flemming Nielson and Hanne Riis Nielson. 1992. *Two-level Functional Languages*. Cambridge University Press, New York, NY, USA.

Georg Ofenbeck, Tiark Rompf, and Markus Püschel. 2017. Staging for Generic Programming in Space and Time. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2017)*. ACM, New York, NY, USA, 15–28. https://doi.org/10.1145/3136040.3136060

Erik Osheim. 2019. Kind Projector. https://github.com/non/kind-projector.

John Reppy. 2006. Type-sensitive Control-flow Analysis. In *Proceedings of the 2006 Workshop on ML (ML '06)*. ACM, New York, NY, USA, 74–83. https://doi.org/10.1145/1159876.1159888

Tiark Rompf. 2016. The Essence of Multi-stage Evaluation in LMS. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.), Vol. 9600. Springer, 318–335. https://doi.org/10.1007/978-3-319-30936-1_17

Tiark Rompf and Nada Amin. 2015. Functional Pearl: A SQL to C Compiler in 500 Lines of Code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 2–9. https://doi.org/10.1145/2784731.2784760

Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Arvind K. Sujeeth, Manohar Jonnalagedda, Nada Amin, Georg Ofenbeck, Alen Stojanov, Yannis Klonatos, Mohammad Dashti, Christoph Koch, Markus Püschel, and Kunle Olukotun. 2015. Go Meta! A Case for Generative Programming and DSLs in Performance Critical Systems. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA (LIPIcs)*, Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.), Vol. 32. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 238–261. https://doi.org/10.4230/LIPIcs.SNAPL.2015.238

Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010*, Eelco Visser and Jaakko Järvi (Eds.). ACM, 127–136. https://doi.org/10.1145/1868294.1868314

Tiark Rompf, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, and Kunle Olukotun. 2014. Surgical precision JIT compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 41–52. https://doi.org/10.1145/2594291.2594316

Mads Rosendahl. 2013. Abstract Interpretation as a Programming Language. *Electronic Proceedings in Theoretical Computer Science* 129 (2013), 84–104. https://doi.org/10.4204/EPTCS.129.7 Published in David A. Schmidt's 60th Birthday Festschrift.

Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. 2013. Monadic Abstract Interpreters. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 399–410. https://doi.org/10.1145/2491956.2491979

Tim Sheard, Zine-El-Abidine Benaissa, and Emir Pasalic. 1999. DSL implementation using staging and monads. In *Proceedings of the Second Conference on Domain-Specific Languages (DSL '99), Austin, Texas, USA, October 3-5, 1999*, Thomas Ball (Ed.). ACM, 81–94. https://doi.org/10.1145/331960.331975

Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. *SIGPLAN Not.* 37, 12 (Dec. 2002), 60–75. https://doi.org/10.1145/636517.636528

Olin Shivers. 1988. Control Flow Analysis in Scheme. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI '88)*. ACM, New York, NY, USA, 164–174. https://doi.org/10.1145/53990.54007

Olin Shivers. 1991. The Semantics of Scheme Control-flow Analysis. In *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '91)*. ACM, New York, NY, USA, 190–198. https://doi.org/10.1145/115865.115884

Gagandeep Singh, Markus Püschel, and Martin Vechev. 2017a. A Practical Construction for Decomposing Numerical Abstract Domains. *Proc. ACM Program. Lang.* 2, POPL, Article 55 (Dec. 2017), 28 pages. https://doi.org/10.1145/3158143

Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2017b. Fast polyhedra abstract domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 46–59. http://dl.acm.org/citation.cfm?id=3009885

Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (April 2015), 1–69. https://doi.org/10.1561/2500000014

Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embedded Comput. Syst.* 13, 4s (2014), 134:1–134:25. https://doi.org/10.1145/2584665

Arvind K. Sujeeth, Austin Gibbons, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. 2013. Forge: generating a high performance DSL implementation from a declarative specification. In *Generative Programming: Concepts and Experiences, GPCE'13, Indianapolis, IN, USA - October 27 - 28, 2013*, Jaakko Järvi and Christian Kästner (Eds.). ACM, 145–154. https://doi.org/10.1145/2517208.2517220

Walid Taha. 1999. *Multi-stage programming: Its theory and applications.* Ph.D. Dissertation. Oregon Graduate Institute of Science and Technology.

Walid Taha and Tim Sheard. 1997. Multi-Stage Programming with Explicit Annotations. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97), Amsterdam, The Netherlands, June 12-13, 1997*, John P. Gallagher, Charles Consel, and A. Michael Berman (Eds.). ACM, 203–217. https://doi.org/10.1145/258993.259019

Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 307–322. https://doi.org/10.1145/3183713.3196893

John Toman and Dan Grossman. 2017. Taming the Static Analysis Beast. In *2nd Summit on Advances in Programming Languages (SNAPL 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.), Vol. 71. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 18:1–18:14. https://doi.org/10.4230/LIPIcs.SNAPL.2017.18

David Van Horn and Harry G. Mairson. 2008. Deciding kCFA is Complete for EXPTIME. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*. ACM, New York, NY, USA, 275–282. https://doi.org/10.1145/1411204.1411243

Dimitrios Vardoulakis and Olin Shivers. 2011. CFA2: a Context-Free Approach to Control-Flow Analysis. *Logical Methods in Computer Science* 7, 2 (2011). https://doi.org/10.2168/LMCS-7(2:3)2011

Philip Wadler. 1992. The Essence of Functional Programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '92)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/143165.143169

Guannan Wei, James Decker, and Tiark Rompf. 2018. Refunctionalization of Abstract Abstract Machines: Bridging the Gap Between Abstract Abstract Machines and Abstract Definitional Interpreters (Functional Pearl). *Proc. ACM Program. Lang.* 2, ICFP, Article 105 (July 2018), 28 pages. https://doi.org/10.1145/3236800

Jeremy Yallop. 2017. Staged generic programming. *PACMPL* 1, ICFP (2017), 29:1–29:29. https://doi.org/10.1145/3110273