Staged Abstract Interpreters

Fast and Modular Whole-Program Analysis via Meta-Programming

Guannan Wei, Yuxuan Chen, Tiark Rompf
Purdue University

Abstract

It is well known that a staged interpreter is a compiler: specializing the interpreter to a given program produces an equivalent program that runs faster. It is even more widely known that an abstract interpreter is a program analyzer: tweaking the interpreter to run on a domain of abstract values produces a sound static analysis.

What happens when we combine these two ideas, and apply staging to an *abstract* interpreter? We obtain a sound static analysis, specialized for a given program, that runs faster. More surprisingly, we show that by applying the staged abstract interpreter to *open* programs and considering the free variables as dynamic inputs, we obtain a modular analysis that generates sound partial analysis results which can be composed and reused later without losing precision, even though the original abstract interpreter is a whole-program analysis algorithm.

Based on the idea of staged abstract interpreters, we show several case studies, including Boucher and Feeley's abstract compilation of 0-CFA, pushdown control-flow analysis with context-sensitivity and precise stores, and a numerical analysis on an imperative language.

We empirically evaluate the performance improvements on control-flow analysis of benchmark programs. The results show speedups up to 2.3x with staging on a monovariant analysis.

1 Introduction

Futamura projections [17, 18] reveal the close connection between compilers and interpreters. The first Futamura projection specifically shows that specializing an interpreter with respect to the input program yields an equivalent executable. Partial evaluation [25] was the first proposed approach to realize Futamura projections, which first identifies the binding-time of variables in the program, i.e., they can be known whether statically or dynamically, and then evaluates the static part, and generates a residual program that relies on the dynamic part. However, precisely analyzing binding-time for an arbitrary program is hard in general. As an alternative and pragmatic approach to specialization and partial evaluation, multi-stage programming (MSP for short) [44, 45] requires the stage annotations (i.e., binding-time annotations) from the programmers. These staging annotations identify which part of the input program should be specialized. As a classical example, we consider a power function:

```
def power(b: Rep[Int], e: Int): Int =
  if (e == 0) 1 else e * power(b, e-1)
```

If the program identifies that `b` will be known dynamically (as shown in its type `Rep[Int]` – a representation of `Int`) and `e` is statically known, say 5, then we can specialize the function `power` with respect to `e = 5` and generate a new specialized function that runs faster:

```
def power5(b: Int) = b * b * b * b * b
```

Abstract interpretation [11] as a semantic approach to construct sound static analysis by approximation is almost as old as Futamura projections. Some recent progress such as Abstracting Abstract Machines (AAM) uncovers a methodology to derive sound abstract interpreters from their concrete counterparts and has been applied to different variants of definitional interpreters and abstract machines [14, 21, 22]. Given the structural similarity between concrete and abstract interpreters, intellectually it is natural to raise the question whether it is possible to specialize an abstract interpreter, and how can we effectively do that having considered their different functionalities. This paper studies the confluence of two old ideas — Futamura project and abstract interpretation, but from the perspective of their recent realizations — multi-stage programming and definitional abstract interpreters. To be specific, we present the application of the first Futamura projection on abstract interpreters, and the approach of constructing optimizing abstract interpreters by multi-stage programming.

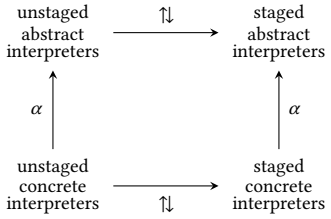

Figure 1. The confluence of staging and abstraction.

Multi-stage programming has been widely used to improve the performance in many domains, such as optimizing compilers or domain-specific languages [10, 34, 36, 42, 43], numerical computation [2, 29], generic programming [33, 50], data processing [26, 30], query compilation in databases [15, 46], etc. Likewise, specializing static analyses by partial

Preprint, November 2018.
Copyright held by the authors.

evaluation emerged in late 90s [4, 6, 7, 13], and indeed it is able to effectively remove the interpretive overhead of repeatedly traversing the abstract syntax tree. However, these previous works focused mostly on one particular analysis, or required to completely rewrite the analyzer. Hence, it is worth to investigate the idea from a modern perspective based on generative programming, especially for a general abstract interpreter that models direct-style λ-calculus, imperative features such as mutation, and different abstract domains. One technical challenge is the binding-time engineering when non-determinism, fixed-point iterations, different abstract domains and staging are introduced at the same time. In this paper, we present an end-to-end design and implementation of staging an abstract interpreter; that means not only the interpreter that traverses the abstract syntax tree, but also the data structure of abstract domains, abstract environment and heap, and the fixed-point iteration are staged.

On the other hand, the abstract interpreter, as a semantic artifact, should be written in a style that is easy for people to communicate the formulation and abstraction, but also can be implemented efficiently. As the slogan of multi-stage programming said, "abstraction without regret", we draw connection between high-level description and efficient implementation of abstract interpreters, just like the connection between concrete interpreters and compilers drawn by Futamura projection. Particularly, we show a easy but systematic way of adding stage annotations to the abstract interpreter without changing the code of interpreter skeleton, which is shared between four concrete or abstract, unstaged or staged interpreters. We use LMS framework for staging, which allows only use types to annotate the binding-time. Therefore, the proposed approach bridges the gap between designing sound static analysis and implementing efficient program analyzer.

When targeting higher-order programs, either staged concrete interpreters or staged abstract interpreters are able to compile a closure, i.e., specialize the call of interpreter with respect to the body expression of the lambda term without knowing the actual argument value. By generalizing this observation, we can actually specialize the abstract interpreter with any open programs, which unexpectedly leads to a modular analysis and improves scalability. A open program contains free variables, which represent other parts of the program and will be left as dynamic values. For instances, one challenge in static analysis of modern software is that the programs are usually shipped with large library code [47], for example, it has been shown that analyzing a simple "Hello World" program in Java depends on additional 3,000 classes in the library [31]. A precise whole-program analysis formulated as abstract interpreters can be very expensive due to the scale of the program and the inherent complexity of the algorithm. However, analyzing these libraries is sometimes unnecessarily repeated. When applying the staged abstract interpreter on such open programs, we leave the unknown arguments and calling contexts as dynamic values and generate partial analyzing result which is represented as a residual program. The partial analyzing result can be reused and composed with the analyzing result of application code when available later. Therefore we mechanically obtain a modular analysis through staging, even though the original algorithm is formulated as a whole-program analysis.

1.1 Contributions

This paper makes the following contributions:

- Starting from a generic definition of interpreter for a higher-order functional language, whose syntax and semantics are described in Section 2.1, we show its concrete interpreter (Section 2.3), staged concrete interpreter (Section 3), abstract interpreter (Section 4) and staged abstract interpreter (Section 5). Our design allows one can derive these four different kinds of interpreters by mechanically changing the underlying semantics or stage annotations.
- As shown in Section 5, we demonstrate that if we apply the abstract interpreters on open programs, it not only improves the *efficiency* but also the *scalability*, which are two major issues in static analysis.
- To demonstrate the applicability and usefulness of the proposed approach, we conduct three case studies Section 6:

  (1) We revisit Abstract Compilation [7] in Section 6.1, which was originally a closure generation technique applied on 0-CFA. We show that abstract compilation can be understood and easily implemented as an instance of our staged abstract interpreter. But unlike the abstract compilation, our approach does not require the implementers to rewrite the analysis.

  (2) We demonstrate pushdown control-flow analysis techniques (Section 6.2), such as context-sensitive analysis and more precise stores can be integrated to staged abstract interpreters without losing any precision.

  (3) We show that in an imperative language (Section 6.3), by replacing the abstract domain to an interval domain (as well as staged), the staged abstract interpreter performs numerical analysis.
- We empirically evaluate the performance improvement by staging an big-step abstract interpreter for control-flow analysis (Section 7). We compare our staged abstract interpreter with the unstaged version.

2 Preliminaries

In this section, we first describe the abstract syntax of the language we will analyze, then present a generic interface for interpreters, and instantiate an unstaged concrete interpreter. Then, starting from this unstaged concrete interpreter, we

proceed in two directions: by abstracting it we obtain an unstaged abstract interpreter (Section 4); by staging it we obtain a staged concrete interpreter (Section 3).

2.1 Abstract Syntax

We consider a tiny higher-order functional language L_λ, which is based on direct-style λ-calculus with numbers, arithmetics, recursions and conditionals. In Section 6.3, we will add more imperative features to the language. Since we are mostly interested in analyzing the dynamic behavior of the program, we disguised any static semantics and type system. We also assume that input programs are well-typed and variables are distinct. The abstract syntax is as follows:

```
abstract class Expr
case class Lit(i: Int)
case class Var(x: String)
case class Lam(x: String, e: Expr)
case class App(e1: Expr, e2: Expr)
case class AOp(op: String, e1: Expr, e2: Expr)
case class Rec(x: String, e: Expr, body: Expr)
case class If0(cnd: Expr, thn: Expr, els: Expr)
```

We give the concrete semantics of $L_\lambda$by showing a big-step definitional interpreter. The interpreter is a recursive function that takes the program AST, environment and store, and returns the evaluated value and the accompanied store. The environment is a mapping from identifiers to addresses, and the store is a mapping from addresses to values. We use the store to model recursion and mutation in concrete semantics; it is also useful for polyvariant analysis. This environment-and-store-passing style big-step interpreter is standard and can also be obtained by refunctionalizing [1, 49] a small-step CESK machine [16].

2.2 Generic Interface

Before showing the actual implementation of the concrete semantics, let us first consider a generic interface for the interpreters, which is semantics-agnostic and can be "interpreted" by either concrete or abstract, unstaged or staged semantics of our choice. The trait `Semantics` first declares several abstract type members, such as `Addr`, `Value`, `Env`, `Store` and returned type `Ans`. It also declares initial values for environments and stores, and few abstract methods that manipulate environments and stores such as `put` and `get`.

```
trait Semantics {
  type R[+_]
  type Ident = String;  type Addr;  type Value;  type Env
  type Store;  type Ans = (R[Value], R[Store])
  def get(ρ: R[Env], x: Ident): R[Addr]
  def put(ρ: R[Env], x: Ident, a: R[Addr]): R[Env]
  def get(σ: R[Store], a: R[Addr]): R[Value]
  def put(σ: R[Store], a: R[Addr], v: R[Value]): R[Store]
  def alloc(σ: R[Store], x: Ident): R[Addr]
  val ρ0: R[Env]; val σ0: R[Store]
  def close(ev: EvalFun)(λ: Lam, ρ: R[Env]): R[Value]
  def num(i: Lit): R[Value]
  def apply_closure(ev: EvalFun)
    (f: R[Value], arg: R[Value], σ: R[Store]): Ans
  def branch0(cnd: R[Value], thn: ⇒ Ans, els: ⇒ Ans): Ans
  def prim_eval(op: Symbol, v1: R[Value], v2: R[Value]): R[Value]
}  // to be continued
```

The actual semantics and operations of the interpreter are left to be implemented later, these cinclude operations such as lifting literal terms to values (`close`, `num`), applying a closure (`apply_closure`), branching (`branch0`), and arithmetics (`prim_eval`). We can choose to implement them in a way that is concrete or abstract, unstaged or staged. But, just based on these abstract operations, an expressive generic interpreter can be defined:

```
trait Semantics {  // continued
  type EvalFun = (Expr, R[Env], R[Store]) ⇒ Ans
  def eval(ev: EvalFun)(e: Expr, ρ: R[Env], σ: R[Store]): Ans =
    e match {
      case Lit(i) ⇒ (num(Lit(i)), σ)
      case Var(x) ⇒ (get(σ, get(ρ, x)), σ)
      case Lam(x, e) ⇒ (close(ev)(Lam(x, e), ρ), σ)
      case App(e1, e2) ⇒
        val (e1v, e1σ) = ev(e1, ρ, σ)
        val (e2v, e2σ) = ev(e2, ρ, e1σ)
        apply_closure(ev)(e1v, e2v, e2σ)
      case Rec(x, f, body) ⇒
        val α = alloc(σ, x)
        val ρ_* = put(ρ, x, α)
        val (fv, fσ) = ev(f, ρ_*, σ)
        val σ_* = put(fσ, α, fv)
        ev(body, ρ_*, σ_*)
      case If0(cnd, thn, els) ⇒
        val (cndv, cndσ) = ev(cnd, ρ, σ)
        branch0(cndv, ev(thn, ρ, cndσ), ev(els, ρ, cndσ))
      case AOp(op, e1, e2) ⇒
        val (e1v, e1σ) = ev(e1, ρ, σ)
        val (e2v, e2σ) = ev(e2, ρ, e1σ)
        (prim_eval(op, e1v, e2v), e2σ)
    }
  def fix(ev: EvalFun ⇒ EvalFun): EvalFun =
    (e, ρ, σ) ⇒ ev(fix(ev))(e, ρ, σ)
  def eval_top(e: Expr): Ans = eval_top(e, ρ0, σ0)
  def eval_top(e: Expr, ρ: R[Env], σ: R[Store]): Ans =
    fix(eval)(e, ρ, σ)
}
```

The implementation of the skeleton interpreter traverses the abstract syntax tree and should be straightforward to understand. It is worth noting that the interpreter is written in open-recursive style – it can not refer to itself directly, instead, `eval` takes an additional parameter `ev` referring to itself; this allows us to instrument the interpreter from the outside. Accordingly, the function `close` for lifting lambda term to closure values is also written in this style. We define a default implementation `fix` to close the recursion by recursively using the result of `fix(ev)` for `ev`. Finally, the top-level evaluation function `eval_top` is defined.

Stage Polymorphism. Readers may notice that an abstract higher-kinded type `R[+T]` is defined in the trait, i.e. a type-level function whici will be used to specify the binding-time, i.e., staged or not; the annotation + indicates that the type parameter `T` is covariant. The LMS framework we use here

allows us to specify binding-times purely based on types. All occurrences of type `Env`, `Store`, etc. are wrapped with `R` essentially because they may potentially be dynamic when staging is introduced. Obviously, the unstaged higher-kinded type then can be simply instantiated as an identity type function: `type R[+T] = T`. In Section 3, `R` will be instantiated as a staging annotation.

2.3 Concrete Instantiation

Now we can instantiate a concrete interpreter. We first concretize the abstract types as follows. Since the interpreter is concrete, for the propose that properly models the heap, the type `Int` is used for address space `Addr`. The environment and store are just ordinary `Maps` in Scala. A value can be either a tagged number `NumV` or a closure `CloV` that contains a lambda term and an environment.

```
trait Concrete extends Semantics {
  type Addr = Int;   sealed trait Value
  case class CloV(λ: Lam, ρ: Env) extends Value
  case class NumV(i: Int) extends Value
  type Env = Map[Ident, Addr];  type Store = Map[Addr, Value]
}
```

we proceed to implement the components that define the semantics. As mentioned before, type `R` is concretized as an identity type function, thus all types are just ordinary types.

```
object ConcInterp extends Concrete {
  type R[+T] = T
  val ρ0 = Map[Ident,Addr](); val σ0 = Map[Addr,Value]()
  def get(ρ: Env, x: Ident): Addr = ρ(x)
  def put(ρ: Env, x: Ident, a: Addr): Env = ρ + (x ↦ a)
  def get(σ: Store, a: Addr): Value = σ(a)
  def put(σ: Store, a: Addr, v: Value): Store = σ + (a ↦ v)
  def alloc(σ: Store, x: Ident): Addr = σ.size + 1
  def close(ev: EvalFun)(λ: Lam, ρ: Env): Value = CloV(λ, ρ)
  def num(i: Lit): Value = NumV(i.i)
  def apply_closure(ev: EvalFun)
    (f: Value, arg: Value, σ: Store): Ans = f match {
      case CloV(Lam(x, e), ρ) ⇒
        val α = alloc(σ, x); val ρ_* = put(ρ, x, α)
        val σ_* = put(σ, α, arg); ev(e, ρ_*, σ_*)
    }
  def branch0(cnd: Value, thn: ⇒ Ans, els: ⇒ Ans): Ans =
    cnd match { case NumV(i) ⇒ if (i == 0) thn else els }
  def prim_eval(op: Symbol, v1: Value, v2: Value): Value =
    op match {
      case '+ ⇒ v1.asInstanceOf[NumV].i + v2.asInstanceOf[NumV].i
      ...
    }
}
```

Finally, `get` and `put` directly manipulate the map of environment and store. An allocation function is used when we need to update the store: in the concrete case, `alloc` always returns a fresh address of the store. `close` and `num` lift syntactic literals to values. `apply_closure` takes the function value, argument value and the latest store, then extends the environment and store after allocating a fresh address, and continue evaluating the body `e` of the lambda term.

3 From Interpreters to Staged Interpreters

3.1 Multi-Stage Programming in LMS

Lightweight modular staging (LMS) [35] is a multi-stage programming framework implemented as a Scala library that enables dynamic code generation in a type-safe manner. Different from the syntactic approach in MetaOCaml [28] that uses quotations, LMS distinguishes binding-time solely based on types. LMS provides a type constructor `Rep[T]` where `T` can be an arbitrary type, indicating an expression will be known at next stage. All operations acting on a `Rep[T]` expression will be residualized in the generated code. However, we need to provide intermediate representation and code generation support for type `T` to LMS. Fortunately, the LMS framework already provides such support for primitive types and commonly used data structure such as arrays and maps; implementing such support for custom class is also straightforward. Let's go back and see how LMS can be used to specialize the power function we mentioned in Section 1.

```
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(x: Rep[Int]): Rep[Int] = power(x, 5)
}
```

In the code shown above, `power` takes two arguments where `b` is declared as `Rep[Int]` type meaning that `b` is a representation of `Int` but whose value will be available at the next stage. Meanwhile, the result of the `power` function will be available at the next stage as well, thus the return value of `power` is also of type `Rep[Int]`. Then we specialize `power` in function `snippet` by providing `x` to `5`. The generated code for `power5` is a function that takes only one argument which corresponds to `b`, and the body of this function multiples `b` five times.

3.2 Staged Concrete Semantics

In the staging part, we share the same concrete type instantiation, i.e. trait `Concrete`, but we reimplement the staged version for the concrete operations in trait `RepConcInterpOps`. Meanwhile, it is also extended from `LMSOps` which provides necessary supports for the staged version of primitive types and data structures. To stage the interpreter, we first identifies the syntactic input is known statically, and the interpreter returns a pair of values and store, which both becomes dynamic values. Thus the staged stores propogate the binding-time to environments, because they share the same address component.

```
trait RepConcInterpOps extends Concrete with LMSOps {
  type R[+T] = Rep[T]
  val ρ0: Rep[Env] = Map[Ident,Addr]()
  val σ0: Rep[Store] = Map[Addr,Value]()
  def get(ρ: Rep[Env], x: Ident): Rep[Addr] = ρ(x)
  def put(ρ: Rep[Env], x: Ident, a: Rep[Addr]): Rep[Env] =
    ρ + (unit(x) ↦ a)
  def get(σ: Rep[Store], a: Rep[Addr]): Rep[Value] = σ(a)
  def put(σ: Rep[Store], a: Rep[Addr], v: Rep[Value]): Rep[Store] =
```

```
    σ + (a ↦ v)
  def alloc(σ: Rep[Store], x: Ident): Rep[Addr] = σ.size + 1
  def num(i: Lit): Rep[Value] = lift(NumV(i))
  def branch0(cnd: Rep[Value], thn: ⇒ Ans, els: ⇒ Ans): Ans = {
    //FIXME: the cast is ugly, patten match?
    val i = cnd.asInstanceOf[Rep[NumV]].i
    if (i == 0) thn else els
  }
  def prim_eval(op: Symbol,
    v1: Rep[Value], v2: Rep[Value]): Rep[Value] = op match {
      case '+ ⇒ v1.asInstanceOf[Rep[NumV]].i +
                v2.asInstanceOf[Rep[NumV]].i
      ...
    }
  ...
}
```

The first notable change is that the abstract type member `R` is assigned to be `Rep`, and accordingly the affected types such as `Env` and `Store` become `Rep[Env]` and `Rep[Store]`. For `num`, we use the `lift` function which is a built-in in LMS that lifts a current-stage value to next stage. The interesting point is how we handle closures and function applications when staging is involved. In the unstaged version, function `close` is used to lift a literal lambda term to a closure, which is a pair of the syntactic lambda term and the enclosing environment. However, what we desire is a *compiled* closure, instead of a lifted `CloV` value like what we did for `NumV` — in other words, the syntactic term of the lambda expression should be eliminated and specialized away. The specialization of the interpreter with respect to the body of the lambda term proceeds under the assumptions that the argument and latest store will be provided later. We will see this is an important observation that enables us to specialize an abstract interpreter in a modular way. At the end of function `close`, we create a `CompiledClo` object that contains a staged function, which all will be emitted in the generated code.

```
def close(ev: EvalFun)(λ: Lam, ρ: Rep[Env]): Rep[Value] = {
  val Lam(x, e) = λ
  val f: Rep[(Value,Store)]⇒Rep[(Value,Store)] = {
    case (arg: Rep[Value],σ: Rep[Store]) ⇒
      val α = alloc(σ, x)
      ev(e, put(ρ, x, α), put(σ, α, arg))
  }
  CompiledClo(fun(f))
}
```

Next, `apply_closure` takes three arguments: the function value, argument value and the latest store, which are both of `Rep` type. This means basically there is nothing we can do at the current stage — so we create a new node `ApplyClosure` that contains these arguments in the intermediate representation graph, and in the code generation phase we can emit the code that performs the application. The additional `reflectEffect` is a function in LMS that handles side effects.

```
case class ApplyClosure(f: Rep[Value],
  arg: Rep[Value], σ: Rep[Store]) extends Def[(Value, Store)]
def apply_closure(ev: EvalFun)
  (f: Rep[Value], arg: Rep[Value], σ: Rep[Store]): Ans = {
    reflectEffect(ApplyClosure(f, arg, σ))
  }
```

Code Generation. When generating code for the next stage, values like `CompiledClo` and `NumV` are kept as they are because they are intended exist in the next stage; however, we need to treat `ApplyClosure` a little differently.

```
case ApplyClosure(f, arg, σ) ⇒
  emitValDef(sym, quote(f) +
                  ".asInstanceOf[CompiledClo].f(" +
                  quote(arg) + "," +
                  quote(σ) + ")")
```

Concretely, as shown in the code above, when emitting code for `ApplyClosure`, we just emit a function application where the function value is extracted from the `CompiledClo` object (we assume the input program is well-typed so that we can safely cast it to `CompiledClo`). Argument and store are the remaining rest fields in `ApplyClosure`.

4 From Interpreters to Abstract Interpreters

Based on the generic interface we presented in Section 2.1, we now describe the instantiation of an abstract interpreter. To keep the presentation simple, we use simple abstract domains, and just establish a context-/path-/flow-insensitive, store-widened analysis in this section — indeed, it is coarse but simple enough to setup a foundation for the staged abstract interpreter we will present later. In Section 6.2, we will see how to regain context-/path-/flow-sensitivity for pushdown control-flow analysis; in Section 6.3, we will see how to instantiate an interval abstract domain for numerical analysis.

Abstract Instantiation. Our instantiation roughly follows the Abstracting Abstract Machines methodology [21, 22] that transforms the concrete semantics to abstract semantics. The store now maps addresses to sets of abstract values, meaning that an address points to all possible values that may occur at runtime, where the abstract value is either a closure or a single abstract number `NumV`, which stands for the top element of the number lattice. The address space is also constrained to be finite, in the monovariant setting, we simply use variable names for address.

```
trait Abstract extends Semantics {
  case class Addr(x: Ident);  sealed trait AbsValue
  case class CloV(λ: Lam, ρ: Env) extends AbsValue
  case class NumV() extends AbsValue
  type Value = Set[AbsValue]
  type Env = Map[Ident, Addr];  type Store = Map[Addr, Value]
}
```

4.1 Stage Polymorphic Lattices

To effectively reuse the code between unstaged and staged variants, we also make the lattice structure stage polymorphic. A lattice type class parameterizes over an element type `E` and defines five operations for `R[E]`, such as meet, join,

and ordering relation; again, R is the higher-kinded type indicating the binding-time.

```
trait Lattice[E, R[+_]] {
  val bot: R[E];  val top: R[E]
  def ⊑(l1: R[E], l2: R[E]): R[Boolean]
  def ⊔(l1: R[E], l2: R[E]): R[E]
  def ⊓(l1: R[E], l2: R[E]): R[E]
}
```

For example, here we use powersets as the abstract domain, and the unstaged lattice of powersets can be easily implemented as follows:

```
implicit def SetLattice[T]: Lattice[Set[T], NoRep] =
  new Lattice[Set[T], NoRep] {
    lazy val bot: Set[T] = Set[T]()
    lazy val top: Set[T] = throw new NotImplementedError()
    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
  }
```

The ordering relation is to ask whether one set is a subset of the other, join is to union the two sets, and meet is to intersect the two sets. Accordingly, other structures used in the rest of the paper such as tuples (products) and maps can be lifted to lattices element-wise or point-wise. The code for them are elided.

4.2 Abstract Semantics

The operations `get` and `put` on abstract stores are slightly changed according to the abstract semantics: `get` uses the bottom element in the `Value` lattice if the queried address does not exist; `put` performs the update by joining with the existing values. function `alloc` simply invokes the `Addr` constructor. `close` and `num` lift syntactic literals to our abstract domain, i.e., a singleton set that contains only one `CloV` or `NumV` object.

```
object AbsInterp extends Abstract {
  type R[+T] = T
  val ρ0 = Map[Ident, Addr](); val σ0 = Map[Addr, Value]()
  def get(ρ: Env, x: Ident): Addr = ρ(x)
  def put(ρ: Env, x: Ident, a: Addr): Env = ρ + (x ↦ a)
  def get(σ: Store, a: Addr): Value =
    σ.getOrElse(a, Lattice[Value].bot)
  def put(σ: Store, a: Addr, v: Value): Store =
    σ + (a ↦ (v ⊔ get(σ, a)))
  def alloc(σ: Store, x: Ident): Addr = Addr(x)
  def close(ev: EvalFun)(λ: Lam, ρ: Env): Value = Set(CloV(λ, ρ))
  def num(i: Lit): Value = Set(NumV())
  def apply_closure(ev: EvalFun)
    (fs: Value, arg: Value, σ: Store): Ans = { var σ_* = σ
      val vs = for (CloV(Lam(x, e), ρ) <- fs) yield {
        val α = alloc(σ_*, x)
        val (v, vσ) = ev(e, put(ρ, x, α), put(σ_*, α, arg))
        σ_* = vσ ⊔ σ_*; v
      }
    (vs.reduce(Lattice[Value].⊔), σ_*)
  }
  def branch0(cnd: Value, thn: ⇒Ans, els: ⇒Ans): Ans =
    thn ⊔ els
  def prim_eval(op: Symbol, v1: Value, v2: Value): Value =
    Set(NumV())
} // to be continued
```

For branching, `branch0` joins the answers from two branches since we are doing a path-insensitive analysis. For arithmetics, `prim_eval` returns the top abstract number element. It is worth noting that `apply_closure` is where we handle the non-determinism of applications. The non-determinism happens because the abstract store maps addresses to sets of abstract values; when dereferencing an address from the store, we need to explore all possible values to achieve a sound analysis. In the case of `apply_closure`, the first argument `fs` may contains multiple target closures, so we use a `for` comprehension over all closures, apply them respectively and form a set result values `vs`; meanwhile, every time when evaluating the body expression `e` we use the latest store σ_* which is updated iteratively during the loop. Thus this is a store-widened analysis.

4.3 Fixpoint Iteration

We described the abstract semantics modularly in the last section, however, the abstract interpreter may not terminate for some input program. In this section, we use a memoization technique to ensure its termination. This technique is also widely used under the name of co-inductive caching algorithm [14, 49] or truncated depth-first evaluation [37].

The idea is to set up two caches called `in` and `out`, which are both mapping from the arguments of the interpreter (`Config`) to the result values of the interpreter (`Ans`). The `in` cache contains what we already have computed from the last iteration, the `out` cache is what we will have computed after this iteration joined with the previous one. When starting a new iteration, `in` is set to be the result of the last iteration, i.e., `out`; `out` is set to be empty. In the iteration, we first check whether `out` contains what we want, if yes then it is returned; otherwise, we retrieve what we have from `in`, compute the result for this iteration, and put the joined result back into `out`. Note that we also instrument the recursive call by putting `cached_ev` into `evev`. After one iteration, if `in == out`, then there is no more information to be gained, thus the iteration should end and we have reached the fixed point.

```
case class CacheFix(evev: EvalFun ⇒ EvalFun) {
  var in = Map[Config, Ans](); var out = Map[Config, Ans]()
  def cached_ev(e: Expr, ρ: Env, σ: Store): Ans = {
    val cfg: Config = (e, ρ, σ)
    if (out.contains(cfg)) out(cfg)
    else {
      val ans0 = in.getOrElse(cfg, Lattice[(Value, Store)].bot)
      out = out + (cfg ↦ ans0)
      val ans1 = evev(cached_ev)(e, ρ, σ)
      out = out + (cfg ↦ (ans0 ⊔ ans1)); ans1
    }
  }
  def iter(e: Expr, ρ: Env, σ: Store): Ans = {
    in = out; out = Map[Config, Ans](); cached_ev(e, ρ, σ)
    if (in == out) out((e, ρ, σ)) else iter(e, ρ, σ)
```

```
  }
}
override def eval_top(e: Expr, ρ: Env, σ: Store): Ans =
  CacheFix(eval).iter(e, ρ, σ)
```

Finally, we override the definition of `eval_top` by instantiating `CacheFix` with `eval` and starting the first iteration.

5 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 interpreter. We present a principled approach to derive staged abstract interpreter from its unstaged version. One guiding principle of our approach is that the code of the abstract semantics and the code that optimizes should be separated. This it is an advantage of using staging for abstract interpreters: the designer of the analysis has no need to rewrite the analysis, and the performance improvement comes almost for free, without any sacrifice of soundness or precision. Unsurprisingly, the staged abstract interpreter we present in this section has the same abstract semantics as the unstaged version we presented in Section 4.

5.1 Staged Lattices

In Section 4.1, we exploited the higher-kinded type `R` to leave space for staging lattices, now we instantiate the type `R` to `Rep` and still use powersets as an example to present its staged version.

```
trait RepLattice[A] extends Lattice[A, Rep]
implicit def RepSetLattice[T:Typ]: RepLattice[Set[T]] =
  new RepLattice[Set[T]] {
    lazy val bot: Rep[Set[T]] = Set[T]()
    lazy val top: Rep[Set[T]] = throw new NotImplementedError()
    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 type parameter `T:Typ` of `RepLattice` requires that the elements of sets is can also be staged. Otherwise, without knowing how to stage the elements in the set, we can not stage the set either. The methods defined operate on type `Rep[Set[T]]`, thus the underlying implementation such as `union` and `intersect` will be mapped to a node in the IR graph during staging and eventually emitted in the generated code. Again, other structures such as maps and tuples are implemented in a similar way.

5.2 Staged Abstract Semantics

We have seen how to obtain a staged concrete semantics based on types, now we take the same approach to obtain a staged abstract semantics. The basic operations are largely kept the same as in the unstaged version, except the types are changed to `Rep`. Besides, when we update the environment, the identifier `x` is known statically, but the environment map has type `Rep[Map[Ident,Addr]]`, so we apply `lift` to `x` to turn it as a next-stage value.

```
trait RepAbsInterpOps extends Abstract with LMSOps {
  type R[+T] = Rep[T]
  val ρ0: Rep[Env] = Map[Ident, Addr]()
  val σ0: Rep[Store] = Map[Addr, Value]()
  def get(ρ: Rep[Env], x: Ident): Rep[Addr] = ρ(x)
  def put(ρ: Rep[Env], x: Ident, a: Rep[Addr]):
    Rep[Env] = ρ + (lift(x) ↦ a)
  def get(σ: Rep[Store], a: Rep[Addr]): Rep[Value] =
    σ.getOrElse(a, RepLattice[Value].bot)
  def put(σ: Rep[Store], a: Rep[Addr], v: Rep[Value]):
    Rep[Store] = σ + (a ↦ RepLattice[Value].⊔(v, get(σ, a)))
  def alloc(σ: Rep[Store], x: Ident): Rep[Addr] = Addr(x)
  def num(i: Lit): Rep[Value] = Set(NumV())
  def branch0(cnd: Rep[Value], thn: ⇒ Ans, els: ⇒ Ans): Ans =
    thn ⊔ els
  def prim_eval(op: Symbol,
                v1: Rep[Value], v2: Rep[Value]): Rep[Value] =
    Set(NumV())
  ...
}
```

Once more, the way we handle closures is the same as in the staged concrete interpreter: the recursive call to `ev` with the body expression `e` is compiled and specialized, the wrapper function `f` will be a field value in a `CompiledClo` object and be generated for the next stage. At the end, we return a singleton set:

```
def close(ev: EvalFun)(λ: Lam, ρ: Rep[Env]): Rep[Value] = {
  val Lam(x, e) = λ
  val f: Rep[(Value, Store)]⇒Rep[(Value,Store)] = {
    case (args: Rep[Value], σ: Rep[Store]) ⇒
      val args = as._1; val σ = as._2; val α = alloc(σ, x)
      ev(e, put(ρ, x, α), put(σ, α, args))
    }
  Set[AbsValue](CompiledClo(fun(f)))
}
def apply_closure(ev: EvalFun)
  (f: Rep[Value], arg: Rep[Value], σ: Rep[Store]): Ans = {
    reflectEffect(ApplyClosure(f, arg, σ))
  }
```

When generating code for an application, we can not directly apply the callee. Instead, we emit code that calls a next-stage function `apply_closures_norep`. As its unstaged counterpart, function `apply_closures_norep` non-deterministically applies multiple target closures with the argument and latest store, and finally returns the joined value and a single store. We provide the definition of `apply_closures_norep` in the runtime supporting code.

```
case ApplyClosures(fs, arg, σ) ⇒
  emitValDef(sym, "apply_closures_norep(" +
                  quote(fs) + "," + quote(arg) +
                  "," + quote(σ) + ")")
```

5.3 Staged Fixpoint Iteration

Our fixed-point iteration again relies on two caches `in` and `out`, but the iteration no longer be done at the current stage. Because the `in` and `out` are both next-stage values, the test of whether `in` and `out` are equal is a generated expression in the next stage, and we can only know the comparison result at the next stage. In other words, we do not know how many iterations we need to reach the fixed-point. To achieve this, we need to stage a function value — `iter_aux` is generated as a recursive function of type `Rep[Unit ⇒ (Value,Store)]` and will be invoked at the next stage.

```
def iter(e: Expr, ρ: Rep[Env], σ: Rep[Store]):
Rep[(Value,Store)] = {
  def iter_aux: Rep[Unit ⇒ (Value,Store)] = fun { () ⇒
    in = out; out = Map[Config, (Value,Store)]()
    cached_ev(e, ρ, σ)
    if (in === out) out((unit(e), ρ, σ)) else iter_aux()
  }
  iter_aux() // generated code that invokes iter_aux()
}
```

However, the instrumented evaluation function that uses the `in` cache and updates the `out` cache can be completely eliminated by staging. Each recursive call to `cached_ev` will also be specialized if it is applied on subexpressions of the analyzed program.

```
def cached_ev(e: Expr, ρ: Rep[Env], σ: Rep[Store]):
Rep[(Value, Store)] = {
  val cfg: Rep[Config] = (unit(e), ρ, σ)
  if (out.contains(cfg)) { out(cfg) }
  else {
    val ans0 = in.getOrElse(cfg, RepLattice[(Value, Store)].bot)
    out = out + (cfg ↦ ans0)
    val ans1 = evev(cached_ev)(e, ρ, σ)
    out = out + (cfg ↦ (ans0 ⊔ ans1)); ans1
  }
}
```

5.4 Specialized Data Structures

Now we have already obtained an end-to-end staged abstract interpreter that is able to specialize an analysis. However, we treat the data structures such as `Maps` as black-boxes, which means any operations on a `Map` becomes code in the next stage. But, as we identified when introducing the generic interface, the keys of any environment maps are identifiers in the program, which are completely known statically. This leaves us a chance to further specialize the data structures. Assume the `Map[K,V]` is implemented as a hash map, if the keys K are known, then the indices can be computed statically. Thus the specialized map would be an array `Array[Rep[V]]` whose elements are next-stage values; all the accesses to the array is determined during staging.

Particularly, if we are specializing a monovariant analysis, the address space is equivalent to the identifiers, then the environment component can be entirely eliminated, and the store is a specialized map as array of `Rep[Value]` elements.

5.5 Modular Analysis for Free

One of the challenges of modern static analysis is program usually depends on large libraries programs [47]. Can we analyze programs and libraries separately and reuse the result without losing precision? Then we can reduce part of the overhead of repeatedly analyzing libraries for different programs. Indeed, some static analyzers compute summary for a function or a module, which can be reused later, however they are mostly too conservative or unsound, which both lead to imprecision.

The specialization of abstract interpreter provides a chance to obtain such partial analysis result in a mechanized way, but still keeps the analysis sound. As we see when compiling the closures, we can specialize the abstract interpreter with respect the body expression of the lambda term without knowing the actual argument. The programs with some unknown variables are open programs, which is exactly the case if we want to analyze programs in a modular way.

For a concrete analysis, for example, k-CFA ($k > 0$) is naturally a whole-program analysis, because it is inter-procedural and needs the last k calling contexts to distinguish different call sites, where the calling contexts are dynamic values. However, it is possible to analyze programs (libraries) separately through specializing an abstract interpreter that generates the analysis as the next-stage program and leave the unavailable programs and calling contexts as dynamic parameters, and then install these contexts when we have the whole program.

5.6 Discussion

In the literature of partial evaluation, Jones provided guidelines on what to do and not to do when specializing a concrete interpreter [24]. We borrow these guidelines and extend them to abstract interpreters. We discuss decisions we made to achieve this and examine some alternatives.

Big-step vs Small-step. What we implemented is a big-step, compositional abstract interpreter, where "compositional" means that every recursive call of our abstract interpreter is applied to proper substructures of the current syntactic parameters [24]. This compositionality ensures that specialization can be done by unfolding, as well as that the specialization procedure terminates. Nevertheless, it is also possible to specialize a small-step operational abstract semantics — Johnson et al. showed this in abstract compilation [7] style as one of their optimizations of Abstract Abstract Machines [23]. However, the generated abstract bytecode still requires another small-step abstract machine to execute, which is an additional engineering efforts. Another alternative approach for efficient specialization is to write the abstract interpreter in a big-step, monadic style [14].

Correctness and Soundness. Based on the assumption that LMS preserves the semantics during staging, we are confident that the staged abstract interpreter does the same analysis compared with the unstaged one. Moreover, the optimization done by staging does not compromise any soundness.

6 Case Study

In Section 5, we have shown that staging an abstract interpreter is feasible and can be systematic after correctly identifying the binding-times, despite the fact that the abstract interpreter is intended to be imprecise and easy to implement. In this section, we conduct several case studies to show that this approach is also useful and widely applicable to different analyses.

6.1 Abstract Compilation a la Staging

Boucher and Feeley introduced abstract compilation (AC) as a new implementation technique for abstract interpretation based static analysis [7]. The idea is inspired by partial evaluation similar to the present paper – the program can be known statically, the overhead of interpretation can be eliminated. In AC, the compiled analysis can be represented by either text or closures (higher-order functions); though the closures can be executed immediately, the textual programs need to be compiled and loaded first.

Specifically, Boucher and Feeley show how to compile a monovariant control-flow analysis [38, 39] for continuation-passing style (CPS) programs. Since the analyzed program is written in CPS, the analyzer is essentially a big-step control-environment abstract interpreter. Closure generation compiles the analysis as a closure taking an environment argument. The overhead of traversing the abstract syntax tree of input program also has been eliminated.

In this section, we show that Boucher and Feeley's abstract compilation can be understood and implemented as an instance of staging abstract interpreters. We first revisit the original implementation of abstract compilation of 0-CFA, and then reproduce their result by simply adding stage annotations. The generated program of our approach improves approximately the same extent of speed, but without changing a single line of the analyzer program (with the use of LMS). However, closure generation requires more engineering effort, specifically a whole-program conversion on the analyzer. Moreover, as shown in Section 5.4, our approach is able to not only remove the interpretive overhead, but also specialize the data structures used in the analysis, for example, the environment that maps variables to sets of lambda.

6.1.1 Closure Generation

The analysis presented by Boucher and Feeley is 0-CFA for a CPS language consisting of lambda terms, applications, `letrec` and primtive operators. The analyses for different syntactic constructs are decomposed into different functions, such as `analyzeCall` and `analyzeApp`. The idea of closure generation is to rewrite these functions. Where previously they may take both static arguments and dynamic arguments, after the rewrite only the static arguments is taken. In this case, the static arguments are syntactic terms; the dynamic arguments are stores. After written in AC style, functions like `compCall` (compiled version of `analyzeCall`) return a value of type `CompAnalysis`, i.e., a closure that takes a store and returns a store. The result of multiple calls on such functions, for example, `compCall` and `compArgs`, can be composed. The generated closure only takes stores, because the input program is specialized into the closure. The code is shown in Figure 2 (left).

6.1.2 Staged 0-CFA

On the other side, our approach does exactly the same thing through staging: the syntactic terms are static, and stores are dynamic, therefore the generated code just looks-up and updates the store. Figure 2 (right) shows the code written with LMS. In fact, the only changes are the type of `Store` is replaced with `Rep[Store]` indicating that the values of type `Store` will be known at the next stage. Indeed, additional engineering efforts are required to make this happen, including: the staged version `Map` which is already included in LMS; implicit `lift` function that transform a current-stage constant value to next stage; next-stage representation of the syntactic terms, i.e., proper `toString` functions of AST structs. We consider these efforts relatively small, and they do not interfere the actual analysis we desire.

6.2 Control-flow Analysis

The target language we presented in Section 2.1 is essentially a higher-order functional language. One fundamental analysis task for functional programs is control-flow analysis, i,e., determining which functions will possibly be applied at each call-site. The abstract interpreter we used in Section 4 and Section 5 is a store-widened 0-CFA-like abstraction, moreover it is also a pushdown control-flow analysis; in the last section, we also reviewed AC with finite-state 0-CFA. In this section, based on the existing staged abstract interpreter, we further develop the staging techniques with control-flow analyses, including recoverying a more precise store model and adding context-sensitivity to the analysis.

6.2.1 A More Precise Store Model

The store-widened analysis we implemented in Section 5 uses a single store to approximate the runtime store. It can be efficently computed in polynomial time together with 0-CFA-like abstraction, but sometimes we may desire a more precise result that distinguishes the final values and stores for different closure targets. To achieve this goal, we need to tweak our abstract interpreter and type instantiation. The answer type `Ans` is changed to a set of `VS`s where a `VS` is a

```
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) ⇒
      analysisAbsApp(σ.lookup(x), args, 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 App(f, args) ⇒ analyzeApp(f, args, analyzeArgs(args, σ))
    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_**)
  }
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)
  }
```

Figure 2. Comparison of AC (left) and SAI (right). Only core code are shown.

pair of sets of abstract values (such as closures or abstract numbers) and a store.

```
type VS = (Set[AbsValue], Store)
type Ans = R[Set[VS]]
```

Note that the type `Ans` uses our stage polymorphic type `R`, meaning that under staging the type `Ans` represents a next stage value. Then, our generic interpreter is also changed when handling function applications.

```
case App(e1, e2) ⇒
  val e1ans = ev(e1, ρ, σ)
  val e1vs  = choices(e1ans)
  val e2ans = ev(e2, ρ, e1vs.σ)
  val e2vs  = choices(e2ans)
  apply_closure(ev)(e1vs.v, e2vs.v, e2vs.σ)
```

The idea to handle the application is to explore all possible closures from `e1` and meanwhile use the latest store. What `choices` does is similar to McCarthy's `amb` operator [32]: it non-deterministically returns an element of type `VS` from its argument, e.g., `e1ans`, to its right-hand side receiver, i.e., `e1vs`. The function `choices` internally uses the delimited control operator `shift` to capture the continuation, which is the code block after its call site. This allows us to perform nesting depth-first evaluation for function application while still writing the program in direct-style [49]. Again, we can stage this part as we did for the naive abstract interpreter.

6.2.2 Context-Sensitivity

We add k-CFA-like context-sensitivity to the analysis by introducing an abstract timestamp, whose concrete instantiation is a finite list of expressions that track k recent calling contexts. The definition of abstract addresses is changed to a tuple of identifiers and the time it get 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 analysis with increasing precision.

```
type Time = List[Expr]
```

Every time when we call the `eval` function, we refresh the timestamp by calling function `tick`, which returns a new timestamp. Here we adopt a k-CFA-like allocation strategy, therefore the `tick` function can be implemented as appending the current expression being evaluated to the existing calling context, and then taking the first k elements from the list.

```
def tick(e: Expr, τ: R[Time]): R[Time] = (e :: τ).takewhile
def eval(ev: EvalFun)
  (e: Expr, ρ: R[Env], σ: R[Store], τ: R[Time]): Ans = {
    val τ_* = tick(e, τ)
    e match {
      case Lit(i) ⇒ ...
      ...
    }
  }
```

Accordingly, the type of return value is accompanied by the timestamp. For a recursive call of ev, which would also return the latest timestamp, and that timestamp will be used for the evaluation afterward.

```
type VST = (Value, Store, Time)
type Ans = R[Set[VST]]
```

Using other allocation strategies to achieve different sensitivities is also possible [19] and can be staged under our framework.

6.3 Numerical Analysis in Imperative Languages

Now we consider in a first-order imperative language, we may care more about the data-flow because the control-flow is relatively easy to obtain. In this section, we show the staging of other abstract domains, particularly an interval domain for numbers. It has been shown that specializing abstract domains with respect to the structure of analyzed program significantly improves the performance: a recent example is online decomposition of polyhedra [40, 41]. In this section, we first show how to support imperative language features in the generic abstract interpreter. Then we present a similar idea for the interval domain and show that the specialization is feasible by staging systematically.

6.3.1 Scaling to Imperative Languages

To evaluates an assignment, we first evaluate its right-hand side, and then put the value into the slot where the address of v points to. For simplicity, we elect to make the value of the assignment be an `void()` value.

```
case Assign(x, e) ⇒
  val (v, σ_*) = ev(e, ρ, σ)
  (void(), put(σ_*, get(ρ, x), v))
```

To evaluates a `while` loop statement, we evaluate the condition first. Then similar to how we treat `branch0`, we have a generic `branch` function but works on boolean values. For the `true` branch, we recursively call `ev` on a newly constructed expression `Seq(e, While(t, e))` meaning that first evaluates `e` and the repeat the loop. Otherwise, for the other branch, we simply return a void value and current store.

```
case While(t, e) ⇒
  val (tv, tσ) = ev(t, ρ, σ)
  branch(tv, ev(Seq(e, While(t, e)), ρ, tσ), (void(), tσ))
```

6.3.2 Staged Interval

Interval domain is a relative simple domain which consists of two numeric fields, an upper bound and a lower bound. Here we annotate the upper bound and lower bound fields to be of type `Rep[Double]`. The operations such as + of two intervals produce a new

```
case class Interval(lb: Rep[Double], ub: Rep[Double]) {
  def +(that: Interval): Interval = that match {
    case Interval(lb_, ub_) ⇒ Interval(lb + lb_, ub + ub_)
  }
  ...
}
```

7 Evaluation

To evaluate the performance can be improved through staging, we use the abstract interpreter demonstrated in the paper to analyze some Scheme programs. We first implement a desugaring transformation for a large subset of scheme that transforms into the small language we used in the paper. We use the same generic interface with a 0-CFA-like abstraction, and the unstage abstract interpreter forms the baseline.

All of our evaluation benchmarks were performed on an Ubuntu 16.04 LTS (Linux kernel 4.4.0) machine with 4 Intel Xeon Platinum 8168 CPU at 2.7GHz and 3 TiB of RAM. Although the machine has 96 cores and 192 threads in total, the abstract interpreters only use one thread to run all the benchmark programs.

Our evaluations show that the staged abstract interpreter performs well in some cases that are considered to be the worst scenarios for traditional k-CFA. The staged version outperforms the unstaged abstract interpreter by a significant margin (up to 230% performance gain) on these difficult ones.

7.1 Benchmarks

The benchmark programs we used are collected from previous papers [6, 23, 48] and exisiting artifacts [1].

We used the following benchmark programs:

- **fib**: Calculates the n-th fibonacci number.
- **rsa**: The RSA public key encryption algorithm.
- **kcfa3**: A difficult benchmark for k-CFA.
- **church**: Church numerals with additions and mutiplications.
- **fermat**: Fermat and Solovay-Strassen primality testing.
- **kcfa-worst-case-n**: Benchmark programs that are supposed to be tough cases for k-CFA; the number n indicates the depth of nesting lambda terms.

7.2 Performance

Considering that our abstract interpreters are implemented in Scala which will be affected by the JIT warm up times, we run all experiments 10 times and report the statistical median values of the running times.

Figure 3 shows the benchmark evaluation results. The "unstaged" and "staged" columns show the median time to finish the analysis (in milliseconds). The "Δ of median" column shows the performance change from the unstaged abstract interpreter to the staged version.

As we can see from the table, the staged version significantly outperforms the unstaged especially for the difficult ones. However, for some benchmarks, we observe degraded performance. We conjecture this happens due to the large size of the generated code is unoptimized – as what we test here is a simple implementation almost identical to what

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

Program	unstaged	staged	Δ of median
fib	1.020	2.046	-50.14%
rsa	35.928	145.306	-75.27%
kcfa3	2.934	5.671	-48.27%
church	884.800	545.736	+62.13%
fermat	15.673	68.790	-77.22%
kcfa-worst-16	320.689	301.564	+6.34%
kcfa-worst-32	3,685.615	1,854.461	+98.74%
kcfa-worst-64	49,422.666	14,849.725	+232.82%

Figure 3. Evaluation result.

we described in the previous sections, we believe the performance still can be improved if more engineering effort is taken, such as optimizing code generation and avoiding code duplication.

8 Related Work

Optimizing Static Analysis Through Specialization The idea in this paper is closely inspired by abstract compilation [7]. Boucher and Feeley presented abstract compilation techniques as an efficient implementation of the monovariant flow analysis (*0*-CFA) for programs written in continuation-passing style. The key idea is to remove the interpretation overhead on traversing the syntax tree by partial evaluation. Specifically, they proposed two similar kinds of abstract compilation techniques. The first one is to generate specialized analysis as a textual program, which then can be loaded and executed by `eval` or other similar mechanisms. The second one is to use closures, i.e., functions that remember their environments, as a representation of specialized analysis. As we show in the case study, compiling the analysis generates higer-order functions on-the-fly with respect to the analyzed program, then the generated closure can be applied immediately in the higher-order host language.

Johnson et al. adapt the idea of closure generation to optimize small-step abstract interpreter in state-transition style [23]. The analyzed program is firstly compiled to a intermediate representation called "abstract bytecode", which are actually higher-order functions, and then be executed on a abstract abstract machine for that IR.

Damian [13] provides a formal treatment to abstract compilation and Shiver's CFA, as well as proofs to establish correctness and certified specialized analyzers [13]. Amtoft applyies partial evaluation for constraint-based control flow analysis [4]. Split the analysis to be multiple stages is also studied other than control-flow analysis, though the formulation may very different. For example, Hardekopf and Lin apply staging to flow-sensitive pointer analysis [20]. The first stage is to analyze the program code to obtain a sparse representation, and then the second stage conducts the flow-sensitive analysis.

Singh et al. optimizes polyhedra abstract domain through online decomposition [40, 41]. The idea is to decompose a large polyhedra into several smaller one; one abstract transformer usually only depends a limit number of variables, then we just need to compute the resulting polyhera on those smaller polyhedra that contains these related variables. The decomposition can be considered as a specialization of abstract domain with respect to the program static structure.

Abstract Interpreters Abstract interpretation was proposed as a semantic-based approach to construct sound static analysis by approximation [11]. As semantic artifacts, the Abstracting Abstract Machines (AAM) [21, 22] approach shows that abstract interpreters can be derived systematically from concrete semantic artifacts. The big-step abstract interpreters we presented in this paper are also inspired by the AAM framework, which are later adopted for big-step abstract interpreters [14, 49]. Cousot and Cousot [12] also proposed an abstract interpretation framework for modular analysis. Calcagno et al. [8] developed compositional shape analysis by using bi-abduction. The big-step abstract interpreter we presented in this paper is inspired by [14, 49]. Keidel et al. [27] demonstrate a similar implementation of definitional abstract interpreter using arrows.

Meta-Programming The Lightweight Modular Staging framework relies on types as stage annotations, and the staging procedure is modular. Other notable implementations of MSP exist in ML family, for examples, MetaML [45] and MetaOCaml [9, 28]. Compared with the LMS approach in Scala, MetaML/MetaOCaml use term-level annotations such as brackets, escape, and run. Notwithstanding, we use LMS in this paper, the idea of staging an abstract interpreter still applies with other MSP implementations. Partial evaluation as an automatic technique is studied comprehensively [24, 25]. Futamura projections reveals close relations between interpreters and compilers [17, 18]. Asai [5] studies the compilation of reflective language using MSP and MetaOCaml. Amin and Rompf shows how to collapse a tower of concrete interpreters by MSP and LMS framework.

9 Conclusion

In this paper, we show the feasibility of using multi-staging programming as a systematic approach to optimize abstract interpreters, while minimally modifying the abstract interpreter and keeping the analysis sound. We presented an end-to-end design and implementation of an staged abstract interpreter for a small higher-order stateful language, which models the core part of modern programming languages. The design that uses abstracting definitional interpreters captures the commonality between abstract interpreters and concrete interpreters; the implementation that uses staging polymorphism captures the commonality between unstage programs and staged programs.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

[1] 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

[2] 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

[3] Nada Amin and Tiark Rompf. 2017. Collapsing Towers of Interpreters. *Proc. ACM Program. Lang.* 2, POPL, Article 52 (Dec. 2017), 33 pages. https://doi.org/10.1145/3158140

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

[5] Kenichi Asai. 2014. Compiling a reflective language using MetaOCaml. In *Generative Programming: Concepts and Experiences, GPCE'14, Vasteras, Sweden, September 15-16, 2014*, Ulrik Pagh Schultz and Matthew Flatt (Eds.). ACM, 113–122. https://doi.org/10.1145/2658761.2658775

[6] 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

[7] 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

[8] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2009. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 289–300. https://doi.org/10.1145/1480881.1480917

[9] 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

[10] 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

[11] 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

[12] 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

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

[14] 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

[15] 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

[16] Matthias Felleisen and Daniel P. Friedman. 1987. A Calculus for Assignments in Higher-Order Languages. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*. ACM Press, 314–325. https://doi.org/10.1145/41625.41654

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

[18] 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

[19] Thomas Gilray, Michael D. Adams, and Matthew Might. 2016. 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

[20] 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

[21] 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

[22] 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

[23] 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

[24] 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.

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

[26] 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

[27] 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

[28] 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

[29] 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

[30] 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

[31] Sulekha Kulkarni, Ravi Mangal, Xin Zhang, and Mayur Naik. 2016. Accelerating program analyses by cross-program training. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 359–377. https://doi.org/10.1145/2983990.2984023

[32] John McCarthy. 1963. A Basis for a Mathematical Theory of Computation1). In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 35. Elsevier, 33 – 70. https://doi.org/10.1016/S0049-237X(08)72018-4

[33] 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

[34] 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

[35] 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

[36] 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

[37] 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.

[38] 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

[39] 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

[40] Gagandeep Singh, Markus Püschel, and Martin Vechev. 2017. 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

[41] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2017. 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

[42] 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

[43] 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

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

[45] 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

[46] 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

[47] 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

[48] 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

[49] 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

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