Refunctionalization of Abstract Abstract Machines (Functional Pearl)

Filling the Gap Between Abstract Abstract Machines and Abstract Definitional Interpreters

ANONYMOUS AUTHOR(S)

Abstracting abstract machines (AAM) is a systematic methodology for constructing abstract interpreters that are derived from concrete small-step abstract machines. Recent progress applies the same idea on definitional interpreters, and obtains big-step abstract definitional interpreters (ADI) written in monadic style. Yet, the relations between small-step abstracting abstract machines and big-step abstracting definitional interpreters are not well studied.

In this paper, we show their correspondence and how to syntactically transform small-step abstract abstract machines into big-step abstract definitional interpreters. The transformations include linearization, fusing, disentangling, refunctionalizing, and un-CPS to direct-style with delimited controls. Linearization expresses non-deterministic choices by first-order data types, after which refunctionalization sequentializes the evaluation order by higher-order functions. All transformations properly handle the collecting semantics and the nondeterminism of abstract interpretation.

Following the idea that in deterministic languages evaluation contexts in reduction semantics are defunctionalized continuations, we further show that in nondeterministic languages, evaluation contexts are refunctionalized to extended continuations style. Remarkably, we reveal how precise call/return matching in control-flow analysis is obtained by refunctionalizing a small-step abstract abstract machine with proper caching.

1 INTRODUCTION

Defining a language by building an interpreter for it can be traced to the very early days of programming languages research [Landin 1966; Reynolds 1972]. Nowadays, even an undergraduate student in computer science is able to build toy languages through interpreters. But building a sound abstract interpreter remained an esoteric and difficult task until very recently.

Van Horn and Might proposed the Abstracting Abstract Machines (AAM) methodology which provides a recipe for constructing sound abstract interpreters for higher-order functional languages from concrete abstract machines [Van Horn and Might 2010, 2012]. Given a concrete small-step abstract machine, (e.g, the CESK machine, Krivine's machine, etc.), by allocating continuations in the store and bounding both the value addresses and continuation addresses to be finite, we obtain an abstract interpreter with a finite state space which can be used for performing sound static analysis. One can further instantiate different polyvariant control flow analyses by using different address allocators [Gilray et al. 2016a].

Applying the same idea to big-step definitional interpreters, Darais et al. built abstract definitional interpreters (ADI) that are written in monadic style [Darais et al. 2017]. One of the advantages of a monadic interpreter is that it is modular and composable. By changing the underlying monads, the definition of the interpreter is not modified, but we can recover different semantics, including the concrete semantics and various abstract semantics such as context-sensitivity and abstract garbage collection [Sergey et al. 2013].

Broadly speaking, abstract abstract machines and abstract definitional interpreters are different forms of abstract interpreters. They are obtained by applying a combination of abstractions to their concrete counterparts, abstract machines and definitional interpreters, respectively. An interesting

2018. 2475-1421/2018/9-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

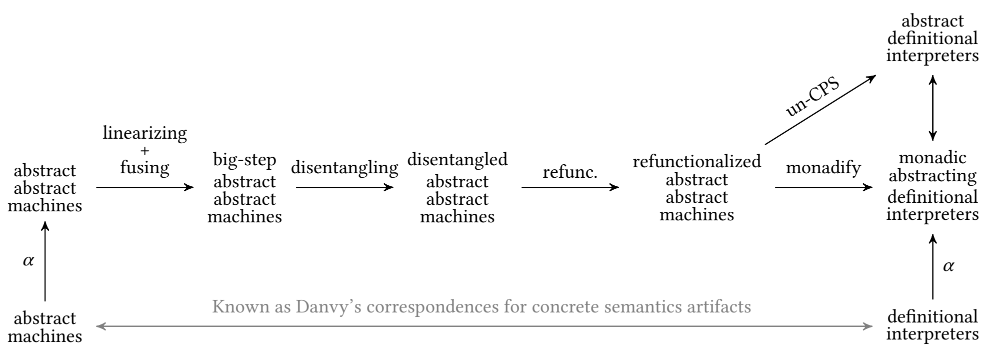

Fig. 1. Transformations from AAM to ADI

question, and the subject of this paper, is how we can interderive these abstract semantics artifacts from the respective other one.

In the concrete world, the relations among reduction semantics, abstract machines, definitional interpreters, and monadic interpreters have been intensively studied by Danvy and his collaborators [Ager et al. 2003, 2004, 2005; Biernacka and Danvy 2009; Danvy 2006, 2008, 2009; Danvy and Nielsen 2001, 2004]. The concrete abstract machines implement structural operational semantics in continuation-passing style, where the reduction contexts are defunctionalized continuations. One can derive definitional interpreters by refunctionalizing the evaluation contexts of abstract machines, and by defunctionalizing the higher-order functions, one may obtain abstract machines in the reverse direction.

In the domain of abstract semantics artifacts, by contrast, the relations between small-step abstract abstract machines and big-step abstract definitional interpreters, as well as the question of deriving one from the other, are not well studied. The fundamental difference between concrete semantics artifacts and abstract semantics artifacts is nondeterminism. In addition to ensuring termination, abstract semantics artifacts are usually equipped with a cache of explored state that is guaranteed to reach the least fixed-point eventually. In this functional pearl, we provide a constructive answer to the question of interderiving abstract semantics artifacts and relate AAM and ADI by presenting a series of syntactical transformations on the program from small-step abstract abstract machines to big-step abstract definitional interpreters.

In addition, the abstract abstract machines with unbounded stack naturally correspond to abstract definitional interpreters. We show that after refunctionalizing an AAM with unbounded stack, and with the proper caching algorithm, the pushdown control-flow analysis can be obtained.

1.1 Contributions

We begin by reviewing some background necessary for this work in Section 2, as well as introducing some of the basic code structures used throughout the paper. We then address the main contribution of this paper, which is the filling of the gap between small-step AAM and big-step ADI by developing a series of systematic transformations. Figure 1 shows the transformations.

Those transformations are summarized here, with their associated section:

- We show **linearization** in Section 3. By expressing the nondeterministic choices as a first-order data type, we linearize the execution of abstract abstract machines; the transition of machine states therefore becomes deterministic. Notably, this introduces another layer of controls that will also be refunctionalized later.

- In Section 4, we then present the **fusing** transformation. The fusing transformation simply combines the single-step function `step` and the driving function into one, but keeps all the machine state representations.
- Section 5 discusses **disentangling**, which disassembles the fused AAM to be several individual functions, with each function handling one data type represented by a continuation.
- **Refunctionalization** is shown in Section 6 which sequentializes the order of abstract execution by higher-order functions. For clarity, we first present the refunctionalized AAM without caching. We then adopt a different caching algorithm to guarantee the termination of abstract interpretation. In this section, we also review pushdown control-flow analysis and examine how computable and precise call/return matching is obtained through these transformations.
- The last transformation we show is that of transforming the refunctionalized AAM to a **direct-style** interpreter (Section 7) by using delimited controls.

These transformations are used throughout the paper, with refunctionalization and defunctionalization of abstract interpreters playing important roles for the call stack of the analyzed language. By refunctionalization, the call stack of the analyzed language is blended into the call stack of the defining language. This provides another perspective to explain why Darais et al.'s abstract definitional interpreters is able to inherit the pushdown control-flow property from its defining language.

We complete this paper by discussing related work in Section 8, followed by concluding thoughts in Section 9.

2 BACKGROUND

2.1 A-Normal Form λ-Calculus

Traditionally, continuation-passing style (CPS) is a popular intermediate representation for analyzing functional programs because it exposes control transfer explicitly and simplifies analysis [Shivers 1988, 1991]. Here, we choose to use λ-calculus in administrative normal form (ANF) [Flanagan et al. 1993], which is a direct-style intermediate representation, as our target for clarity, but without losing simplicity and generality. The transformations we will show in the rest of this paper also work on abstract machines for plain direct-style λ-calculus languages. Although we only show the core calculus language, it can be easily extended to support recursive bindings (such as `letrec`), conditionals, primitive types, and operations on primitive types. These cases would be trivial to implement, so we elide them in this paper.

To begin, we present the concrete syntax of a call-by-value λ-calculus language in ANF.

```
  e ∈ Exp  ::= ae | (let ([x (ae ae)]) e)
 ae ∈ AExp ::=  x | lam
lam ∈ Lam  ::= (lambda (x) e)
  x ∈ Var  (variable names)
```

In ANF, an expression is either an atomic expression or a `let` expression. A restriction exists which states that all function applications must be administrated within a `let` expression and then bound to a variable name under the current environment. Both the operator and operand of function applications are atomic expressions. An atomic expression *ae* is either a variable or a literal `lambda` term, either of which can be evaluated in a single step. We also assume that all the variable names in the program are unique.

The abstract syntax in Scala is shown as follows. We assume that the source program conforms to the ANF convention, and we do not enforce it in the term structure of Scala constructs.

```
sealed trait Expr
case class Var(x: String) extends Expr
case class App(e1: Expr, e2: Expr) extends Expr
case class Lam(x: String, body: Expr) extends Expr
case class Let(x: String, e: App, body: Expr) extends Expr
```

2.2 CESK Machine

2.2.1 Machine Components. The CESK machine is an abstract machine for describing semantics of and evaluating λ-calculus [Felleisen and Friedman 1987]. The CESK machine models program execution as state transitions in a small-step fashion. As its name suggests, a machine state has four components: 1) *Control* is the expression currently being evaluated. 2) *Environment* is a map that contains the address of a variable in the lexical scope. 3) *Store* models the heap of a program as a map from addresses to values. The address space consists of numbers (0-indexed). In our toy language, the only category of value is a closure, i.e., a function paired with an environment. 4) *Continuation* represents the program stack. In this paper, we instantiate the stack as a list of frames because the ANF simplifies the evaluation context. For direct style λ-calculus, the continuations of CEK/CESK machines can be presented by variants of a data type.

The Scala representations for the components of the CESK machine are as follows:

```
type Addr = Int
type Env = Map[String, Addr]
type Store = Map[Addr, Storable]

abstract class Storable
case class Clos(v: Lam, env: Env) extends Storable

case class Frame(x: String, e: Expr, env: Env)
type Kont = List[Frame]

case class State(e: Expr, env: Env, store: Store, k: Kont)
```

It is worth noting that the continuation class `Kont` is defined as a list of frames, where a frame can be considered an evaluation context in reduction semantics. We represent frames using the `Frame` class, which stores the information of a single call-site, i.e., the information that can be used to resume the interrupted computation. A `Frame` constitutes a variable name `x` to be bound later, a control expression to which the program may resume, and its environment.

2.2.2 Single-Step Transition. Before describing how the machine evaluates expressions, we must first define several helper functions. As mentioned in Section 2.1, atomic expressions are either a variable or a literal `lambda` term. As such, the atomic expression evaluator `atomicEval` handles these two cases and evaluates atomic expressions to closures in a straightforward way. The `alloc` function generates a fresh address, and always allocates a unique integer in the domain of `store`. The `isAtomic` function is used as a predicate to determine if the expression is atomic.

```
def atomicEval(e: Expr, env: Env, store: Store): Storable = e match {
  case Var(x) ⇒ store(env(x))
  case lam @ Lam(x, body) ⇒ Clos(lam, env)
}
def alloc(store: Store): Addr = store.keys.size + 1
```

```
def isAtomic(e: Expr): Boolean = e.isInstanceOf[Var] || e.isInstanceOf[Lam]
```

We can now faithfully describe the state transition function `step`, which when given a machine state, determines its successor state.

```
def step(s: State): State = s match {
  case State(Let(x, App(f, ae), e), env, store, k) if isAtomic(ae) ⇒
    val Clos(Lam(v, body), env_c) = atomicEval(f, env, store)
    val addr = alloc(store)
    val new_env = env_c + (v ↦ addr)
    val new_store = store + (addr ↦ atomicEval(ae, env, store))
    val frame = Frame(x, e, env)
    State(body, new_env, new_store, frame::k)
  case State(ae, env, store, k) if isAtomic(ae) ⇒
    val Frame(x, e, env_k)::ks = k
    val addr = alloc(store)
    val new_env = env_k + (x ↦ addr)
    val new_store = store + (addr ↦ atomicEval(ae, env, store))
    State(e, new_env, new_store, ks)
}
```

As shown and previously discussed, we examine the only two cases which the state may be.

- In the first `case` statement shown in the previous code, the control of the current state matches as a `Let` expression, with its right-hand side a function application. By calling the `atomicEval` evaluator, we obtain the closure for which the callee `f` stands. The successor state's control then transfers to the `body` expression of the closure with an updated environment and store. The new environment is extended from the closure's environment and mapped from `v` to a fresh address `addr`. The new store is extended with `addr` mapping to the value of `ae`, which in turn is evaluated from `atomicEval`. Finally, a new frame `frame` is pushed onto the stack `k`, where `frame` contains the variable name `x` at the left-hand position of the `Let`, the body expression of `Let`, and the lexical environment of the body expression.
- If the state is not a `Let` expression, then it must be an atomic expression, as seen in the above code. In this scenario, we begin by extracting the top frame of all available continuations. The control (i.e., an atomic expression) of the current state is the evaluated term that is being bound to the variable `x` from the top frame. The environment and store are updated with `x` mapping to the closure value of `ae`. Finally, the successor state is transferred to expression `e` from the top frame, which is the body of a `Let` expression, with the updated environment, store, and the rest of the stack `ks`.

2.2.3 Valuation. To run the program, we first use the `inject` function (below) to construct an initial machine state given an expression `e`. The initial state contains an empty environment, store, and stack.

```
def inject(e: Expr): State = State(e, Map(), Map(), Nil)
```

The `drive` function is then used to evaluate to a final state by iteratively applying `step` on the current state until a state is reached ion which the control is an atomic expression and the continuation stack is empty. Naturally, we can then extract the value from the final state at last.

```
def drive(s: State): State = s match {
  case State(ae, _, _, Nil) if isAtomic(ae) ⇒ s
  case _ ⇒ drive(step(s))
}
def eval(e: Expr): State = drive(inject(e))
```

2.3 Abstracting Abstract Machines

Abstracting abstract machines (AAM) is a systematic methodology that derives sound abstract interpreters for higher-order functional languages from concrete abstract machines [Van Horn and Might 2010, 2012]. An abstracting abstract machine implements computable abstract semantics which approximates the runtime behaviors of programs. Since the state space of concrete execution is possibly infinite, the key insight of AAM approach when analyzing programs is to allocate both bindings and continuations on the store, and then bound the addresses space to be finite. Since each component of state is finite, the abstracted machine-state space is also finite, and therefore computable.

In this section, we derive the abstracting abstract machine from concrete CESK machines, and also show how to instantiate useful k-call-sensitive control flow analysis.

2.3.1 Machine Components. Similar to CESK machines, the machine state of AAM has a control expression, an environment, a store, and continuation, as well as a timestamp. However, there are several notable differences between AAM's store and CESK machine's store. In AAM, the store maps addresses to sets of values; it stores all possible values for a particular address. As such, dereferencing addresses becomes nondeterministic. Also, the store performs *joining*, rather than overwriting, when updating elements. Furthermore, the continuations are likewise allocated on the store instead of formed into a linked list, and the continuations are in a state which then turns into a continuation address.

For clarity, we divide the store into two separate stores: the binding stores `BStore`, and the continuation store `KStore`. The binding store maps binding addresses to sets of closure values, whereas the continuation store maps continuation addresses to sets of continuations. We then define a generic class `Store[K,V]` that performs joining when updating elements in a store (below). By parameterizing `Store[K,V]` with `[BAddr, Storable]` and `[KAddr, Cont]`, we obtain `BStore` and `KStore`, respectively.

We note that both the value store and continuation store are monotonic; it continuously grows and never shrinks. This property guarantees that a fixed point of store can always be reached through Kleene's iteration when analyzing the program

```
case class Store[K,V](map: Map[K, Set[V]]) {
  def apply(addr: K): List[V] = map(addr).toList
  def update(addr: K, d: Set[V]): Store[K,V] = {
    val oldd = map.getOrElse(addr, Set())
    Store[K, V](map ++ Map(addr ↦ (d ++ oldd)))
  }
  def update(addr: K, sd: V): Store[K,V] = update(addr, Set(sd))
}
type BStore = Store[BAddr, Storable]
type KStore = Store[KAddr, Cont]
```

The codomain of binding stores `Storable` is the same as previously defined for CESK machines. The codomain of continuation stores `Cont`, on the other hand, is comprised of a `Frame` object and a continuation address `KAddr`. To mimic the runtime call stack, `KAddr` plays the role of representing the remaining stack frames. But since the continuation store may contain multiple continuations, the dereferencing of continuation addresses is also nondeterministic.

```
abstract class Storable
case class Clos(v: Lam, env: Env) extends Storable

case class Frame(x: String, e: Expr, env: Env)
```

```
case class Cont(frame: Frame, kaddr: KAddr)
```

As a consequence, the components of states are also changed: the store is divided into binding stores and continuation stores; the continuation stack becomes an address. By dereferencing this address in a continuation store, we can retrieve the actual transfer of controls. The definition of environment `Env` remains the same.

```
case class State(e: Expr, env: Env, bstore: BStore, kstore: KStore, k: KAddr, time: Time)
```

2.3.2 Allocating Addresses. Up to this point, we have not described allocating addresses in stores, nor handling the time stamp `Time`. In abstract interpretation, however, these are key ingredients to achieve analyses with different sensitivities, as well as to perform a finite state space analysis[Gilray et al. 2016a]. To effectively approximate the runtime behavior, we introduce a finite program contour `time` that encodes the program execution history. We use a list of execution contexts (expressions) to represent this, and as we will see in Section 2.3.4, by applying different `tick` functions on the timestamp, we are able to obtain a family of analyses.

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

As previously mentioned, the space of addresses must be finite in AAM. Binding addresses are parameterized by variable names and the creation time of the binding, both of which are finite. Continuation addresses `KAddr` has two variants: 1) `Halt` which corresponds to the empty stack, and 2) `ContAddr` consists of the target expressions of callee, which are also finite.

```
case class BAddr(x: String, time: Time)

abstract class KAddr
case object Halt extends KAddr
case class ContAddr(tgt: Expr) extends KAddr
```

We introduce two helper functions, `allocBind` and `allocKont`, which will be used to allocate binding addresses and continuation addresses.

```
def allocBind(x: String, time: Time): BAddr = BAddr(x, time)
def allocKont(tgtExpr: Expr): KAddr = ContAddr(tgtExpr)
```

2.3.3 Single-Step Transition. Since dereferencing an address becomes nondeterministic, our `atomicEval` function (below) is also nondeterministic. Given an atomic expression `e`, `atomicEval` returns a set of storable values (i.e., closures) to the caller. If the expression is simply a `lambda` term, the returned set is a singleton.

```
def atomicEval(e: Expr, env: Env, bstore: BStore): Set[Storable] = e match {
  case Var(x) ⇒ bstore(env(x))
  case lam@Lam(x, body) ⇒ Set(Clos(lam, env))
}
```

The structure of function `step` is similar to the concrete CESK machine, except the nondeterminism which makes `step` return a list of reachable successor states. We have two cases to consider (code shown below):

- If the current control expression is a `Let`, then the result of `App(f, ae)` will be bound to variable `x`. In this case, we retrieve the set of closures that `f` may represent. For each closure in the set, we perform nearly the same operations as in the concrete CESK machines, with an important difference: the continuation is allocated on the store `kstore`, so a new continuation address `new_kaddr` must be constructed and a new frame `Frame(x, e, env)` paired with the current continuation address kaddr is merged into new_kaddr. Finally, a list of successor states is generated.

- In the second case, an atomic expression ae sits on the control position of the state. Here, the value of ae is being returned to its caller. In order to accomplish this, we dereference the continuation address kaddr and obtain a set of continuations conts. For each continuation in the set, we construct an environment based on the environment env_f of the frame, and bind x to a newly created binding address baddr. We must also update the store with baddr and the values that ae represents. In every generated state, the control becomes the expression e in the frame, and as we can tell from the name, the continuation address f_kaddr also comes from the frame.

```
def step(s: State): List[State] = {
  val new_time = tick(s)
  s match {
    case State(Let(x, App(f, ae), e), env, bstore, kstore, kaddr, time) ⇒
      val closures = atomicEval(f, env, bstore).toList
      for (Clos(Lam(v, body), env_c) <- closures) yield {
        val baddr = allocBind(v, new_time)
        val new_env = env_c + (v ↦ baddr)
        val new_bstore = bstore.update(baddr, atomicEval(ae, env, bstore))
        val new_kaddr = allocKont(body)
        val new_kstore = kstore.update(new_kaddr, Cont(Frame(x, e, env), kaddr))
        State(body, new_env, new_bstore, new_kstore, new_kaddr, new_time)
      }
    case State(ae, env, bstore, kstore, kaddr, time) if isAtomic(ae) ⇒
      val conts = kstore(kaddr).toList
      for (Cont(Frame(x, e, env_f), f_kaddr) <- conts) yield {
        val baddr = allocBind(x, new_time)
        val new_env = env_f + (x ↦ baddr)
        val new_store = bstore.update(baddr, atomicEval(ae, env, bstore))
        State(e, new_env, new_store, kstore, f_kaddr, new_time)
      }
  }
}
```

2.3.4 k-Call-Sensitive Instantiation. In k-call-sensitive analysis, a history of the last k call sites is used as a finite program contour. The history is represented as a list of expressions and embedded in the allocated addresses.

Before transferring to successor states, we must use the tick function to refresh the timestamp, and then use this new timestamp for successors when allocating addresses. The tick function returns the k front-most expressions given the current state and its time history.

```
def k: Int = 0
def tick(s: State): Time = (s.e :: s.time).take(k)
```

If we instantiate k to be 0, the history degenerates to an empty list, and we obtain a monovariant analysis (i.e., it does not differentiate values at different call sites). In this case, the address space collapses to the space of variable names. Note that regarding the ambiguity in k-CFA[Gilray et al. 2016a], the code here actually implements call+return sensitivity.

2.3.5 Collecting Semantics. Similar to the CESK machines, to run (analyze) a program we first use the inject function to construct the initial state given to the program. Note that the initial continuation store has a built-in mapping that maps the continuation address for `Halt` to an empty set of continuations. We also provide an empty program contour as our initial time.

```
def inject(e: Expr): State =
  State(e, Map(),
        Store[BAddr, Storable](Map()),
        Store[KAddr, Cont](Map(Halt ↦ Set())),
        Halt, List())
```

However, in contrast to the concrete CESK machine, the `drive` function performs collecting semantics instead of the valuation semantics. That is, for the purpose of analyzing programs, the function `drive` collects all the intermediate machine states as the program is abstractly executing. The following code shows a variant of the worklist algorithm to find the fixed-point of the set of states. Function `drive` always applies function `step` to the head element `hd` of the worklist `todo` if `hd` is unseen. It then inserts the result of `step` to the rest of worklist, and in the meantime adds `hd` to the explored states set. If the worklist is empty, `drive` simply returns the set of reachable states up to the current execution point.

```
def drive(todo: List[State], seen: Set[State]): Set[State] = todo match {
  case Nil ⇒ seen
  case hd::tl if seen.contains(hd) ⇒ drive(tl, seen)
  case hd::tl ⇒ drive(step(hd).toList ++ tl, seen + hd)
}

def analyze(e: Expr): Set[State] = drive(List(inject(e)), Set())
```

Finally, a user may invoke the `analyze` function to obtain all reachable states for a given program.

2.4 One Step Back: Unabstracted Stack

In this section, we describe a variant of AAM that allows the stack to be unbounded which uses a precise call stack as we did in the concrete CESK machine. Instead of allocating continuations in the store and embedding addresses of continuations in states, we intend to use a list of frames to explicitly model the stack. By doing so, we recover the call stack as precise as runtimes (so called pushdown control flow analysis), but since the stack is unbounded, the analysis is potentially not computable. For readers who are not familiar with pushdown analysis, we have a detailed discussion in Section 6.3. The reason we show it here is to establish an artifact with precise call/return matching used for the next step of transformation.

In the definition of `State`, the continuation store disappears, and component `konts` becomes a list of frames. The other components remain unchanged.

```
case class State(e: Expr, env: Env, bstore: BStore, konts: List[Frame], time: Time)
```

The state transition function `step` shown below is still nondeterministic, but the only nondeterminism happening is when dereferencing the callee `f` from the function application `App(f, ae)`.

```
def step(s: State): List[State] = {
  val new_time = tick(s)
  s match {
    case State(Let(x, App(f, ae), e), env, bstore, konts, time) if isAtomic(ae) ⇒
      for (Clos(Lam(v, body), env_c) <- atomicEval(f, env, bstore).toList) yield {
        val frame = Frame(x, e, env)
        val baddr = allocBind(v, new_time)
        val new_env = env_c + (v ↦ baddr)
        val new_store = bstore.update(baddr, atomicEval(ae, env, bstore))
```

```
        State(body, new_env, new_store, frame::konts, new_time)
      }
    case State(ae, env, bstore, konts, time) if isAtomic(ae) ⇒
      konts match {
        case Nil ⇒ List()
        case Frame(x, e, env_f)::konts ⇒
          val baddr = allocBind(x, new_time)
          val new_env = env_f + (x ↦ baddr)
          val new_store = bstore.update(baddr, atomicEval(ae, env, bstore))
          List(State(e, new_env, new_store, konts, new_time))
      }
  }
}
```

In the first case of pattern matching, we may have multiple choices of closure for callee `f`. For each closure in the set, a new frame is constructed and pushed onto the stack. The code for handling the second case (atomic expressions) is the same as the concrete CESK machines.

Note on Computability. Unfortunately, even though other components in the states are finite, this AAM with an unbounded stack is still not computable. This is because the unbounded stack can grow to arbitrary depth which implies that the state space is possibly infinite; the analysis may therefore not terminate for all programs if we simply enumerate reachable states. To see this, consider a program that has two mutually recursive functions:

```
(letrec ([f1 (lambda (x)
               (let ([x1 (f2 x)]) x1))]
         [f2 (lambda (y)
               (let ([y1 (f1 y)]) y1))])
  (let ([z (f1 1)])
    z))
```

Function `f1` and `f2` mutually invoke each other, so the stack will alternate pushing frames `f1` and `f2` onto the top of current stack. However, no two existing stack components are identical in the state space.

3 LINEARIZATION

In the previous section, we show that by keeping an unabstracted stack in the state space, we can recover the precise call/return match. In this and following sections, we begin describing the transformations step by step. Our base machine for now is the AAM with unbounded stack, though as we will later transform the stack to higher-order functions representing continuations, it does not matter what kind of AAM we start from. This is because transforming to higher-order functions forces us to sequentialize the order of abstract evaluation, and neither requiring construction of a frame on stack, nor to allocate continuations in the store. Thus, our choice to start from an AAM with unbounded stack is motivated simply because it has an equivalent stack model to abstract definitional interpreters (our final target).

In Danvy's paper *Defunctionalized Interpreters for Programming Languages*, he mentions that for deterministic languages, a reduction semantics is a structural operational semantics in continuation style, where the reduction context is a defunctionalized continuation [Danvy 2008]. This poses a problem, as the underlying semantics of AAM is fundamentally nondeterministic. But the evaluation context (i.e., the `Frame` in our program) is *not* nondeterministic, and the worklist `todo` actually implicitly handles the nondeterminism. Thus, if we simply refunctionalize the frames to functions, it does not help us move toward abstract definitional interpreters.

In order to address this problem, our first step of transformations is to linearize all nondeterministic choices. This step removes all nondeterminism from the `step` function. Likewise, the `Frame` saves the information of the caller in concrete executions. We then introduce another layer of controls, and define a case class `NDCont` that saves the information at a fork point when we have multiple target closures. We also add a new field `ndk` to the definition of state, which we now call `NDState`. For clarity of presentation in differentiating between the two continuations, we elect to call the first a *normal continuation*, and the second as a *nondeterministic continuation*.

```
case class NDCont(cls: List[Clos], argvs: Set[Storable], store: BStore, time: Time, frames: List[Frame])
case class NDState(e: Expr, env: Env, bstore: BStore, konts: List[Frame], time: Time, ndk: List[NDCont]) {
  def toState: State = State(e, env, bstore, konts, time)
}
```

Each `NDCont` object contains a list of closures that are possible functions to be invoked, a set of values that will be bind to the function's formal argument, and the store, time, and frames at the fork point. In the definition of `NDState`, an auxiliary function `toState` is added that converts itself to `State`.

Function `step` now becomes of type `NDState ⇒ NDState`. The following code shows the first case of matching an instance of `NDState` in the `step` function:

```
case NDState(Let(x, App(f, ae), e), env, bstore, konts, time, ndk) ⇒
  val closures = atomicEval(f, env, bstore).toList.asInstanceOf[List[Clos]]
  val Clos(Lam(v, body), c_env) = closures.head
  val frame = Frame(x, e, env)
  val new_frames = frame::konts
  val baddr = allocBind(v, new_time)
  val new_env = c_env + (v ↦ baddr)
  val argvs = atomicEval(ae, env, bstore)
  val new_store = bstore.update(baddr, argvs)
  val new_ndk = NDCont(closures.tail, argvs, bstore, new_time, new_frames)::ndk
  NDState(body, new_env, new_store, new_frames, new_time, new_ndk)
```

By atomically evaluating `f`, we obtain a set of closures, with the transition being deterministic in regards to the first element of the closure set. As such, we prepare a new environment, a new store, and a new frame list only for the first closure in that set. A new nondeterministic continuation `new_ndk` is also constructed, which contains the rest of closures, the values of argument `ae`, the store, the time, and the new frame list. We use the store before updating, because for different closures they may form different binding addresses `baddr`. We also use the new frames, because all closures at this fork point share the same stack and return point.

In the second case (shown below), we will see how `NDCont` deals with nondeterminism.

```
case NDState(ae, env, bstore, konts, time, ndk) if isAtomic(ae) ⇒
  konts match {
    case Nil ⇒ ndk match {
      case NDCont(Nil, _, _, _, _)::ndk ⇒
        NDState(ae, env, bstore, konts, time, ndk) /* transfer to the most recent fork point */
      case NDCont(cls, argvs, bstore, time, frames)::ndk ⇒
        val Clos(Lam(v, body), c_env) = cls.head
        val baddr = allocBind(v, time)
        val new_env = c_env + (v ↦ baddr)
        val new_store = bstore.update(baddr, argvs)
        val new_ndk = NDCont(cls.tail, argvs, bstore, tile, frames)::ndk
        /* resume the fork point with the next closure */
```

```
      NDState(body, new_env, new_store, frames, time, new_ndk)
    }
    case Frame(x, e, f_env)::konts ⇒
      val baddr = allocBind(x, newTime)
      val new_env = f_env + (x ↦ baddr)
      val new_store = bstore.update(baddr, atomicEval(ae, env, bstore))
      NDState(e, new_env, new_store, konts, newTime, ndk) /* normal return */
  }
```

If the normal continuation `konts` is an empty list, then it means we have reached the end of the computation (i.e., a halt); otherwise, we should return the values of `ae` to its caller which is contained in the top frame of stack.

However, since we have added nondeterminism into state, an empty list of frames means we have the reached the halt of *one computation path*, and we should determine the next state indicated by `NDCont`. Thus, we have a pattern matching on `ndk`:

- If the closure set is an empty set, then we have tried all the closures of this fork point and should move to the next fork point. Since the `NDCont` is represented by a list, and we always append new elements to its front, we are resuming to the most recent fork point by popping up the front-most element.
- Otherwise, we pop a closure from the set, then build a new environment and a new store for the body expression of the closure; we use the same frame list and time that are copied from the fork point rather than current one. The nondeterministic continuation `new_ndk` is also updated by the removal of that closure.

```
def drive(nds: NDState, seen: Set[State]): Set[State] = {
  nds match {
    case NDState(ae, _, _, Nil, _, Nil) if isAtomic(ae) ⇒ seen
    case nds ⇒
      val s = nds.toState
      if (seen.contains(s)) drive(step(nds), seen)
      else drive(step(nds), seen + s)
  }
}
```

The `drive` function is also changed: there is no worklist anymore, as all the potentially unexplored states are embedded into the continuation for nondeterminism. The termination of the analysis occurs when we reach an `NDState` in which the expression is atomic and both the normal continuation and nondeterminism continuation are empty lists. This corresponds to the case that `todo` list is empty.

At this point, we have obtain a linearized abstract abstract machine. If we imagine that the classical AAM explores a graph of reachable states, then the linearized AAM flattens the graph to a linear sequence.

4 FUSING

Fusing is a transformation that combines the `step` and `drive` functions into a single function. The fused function `drive_step` is essentially formed by merging the functionality of `step` into the `drive` function. `drive_step` takes an `NDState` and a set of explored states as an argument and returns a set of reachable states once it terminates.

```
def drive_step(nds: NDState, seen: Set[State]): Set[State] = {
  nds match {
    case NDState(ae, _, _, Nil, _, Nil) if isAtomic(ae) ⇒ seen
```

```
    case nds ⇒
      val s = nds.toState
      val new_seen = if (seen.contains(s)) seen else seen+s
      val newTime = tick(nds)
      val new_ndstate = nds match {
        case NDState(Let(x, App(f, ae), e), env, bstore, konts, time, ndkonts) ⇒
          ......
        case NDState(ae, env, bstore, konts, time, ndkonts) if isAtomic(ae) ⇒
          ......
      }
      drive_step(new_ndstate, new_seen)
  }
}
```

With this, have a single function to perform both abstract evaluation and the collection of intermediate states. Given `inject(e)` as an initial state and an empty set as the initial set of states reached, we can easily define the entrance function `analyze` as follows:

```
def analyze(e: Expr): Set[State] = drive_step(inject(e), Set())
```

5 DISENTANGLING

With fusing complete, our AAM appears similar to a "big-step" interpreter, although it still has the machine state representation inside. In the disentangling transformation, we identify the first-order data types which represent evaluation contexts and lift the code blocks that handle these evaluation contexts to be individual functions.

Since there are two layers of continuations, we obtain three mutually recursive functions: `drive_step`, which plays the same role as before; `continue`, which is called from `drive_step` when encountering an atomic expression, and handles the evaluation context (`Frame`) of the analyzed language; and `ndcontinue`, which dispatches nondeterministic continuations `NDCont`, and which is invoked from `continue` when the normal stack is empty.

```
def drive_step(nds: NDState, seen: Set[State]): Set[State] = {
  nds match {
    case NDState(ae, _, _, Nil, _, Nil) if isAtomic(ae) ⇒ seen
    case nds ⇒
      val s = nds.toState
      val new_seen = if (seen.contains(s)) seen else seen + s
      val new_time = tick(nds)
      nds match {
        case NDState(Let(x, App(f, ae), e), env, bstore, konts, time, ndk) ⇒
          ......
          drive_step(NDState(body, new_env, new_store, new_frames, new_time, new_ndk), new_seen)

        case NDState(ae, env, bstore, konts, time, ndk) if isAtomic(ae) ⇒
          continue(nds, new_seen)
      }
  }
}
```

The above code shows the skeleton of `drive_step`. At the end of the first case of pattern matching, we invoke `drive_step` recursively[1]. The code for the second case is replaced entirely by a function

[1] The code for constructing new environments, stores, and frames is elided for simplicity of presentation.

call to continue. The astute reader will notice a vague shape of interpreter in continuation style has emerged.

```
def continue(nds: NDState, seen: Set[State]): Set[State] = {
  val NDState(ae, env, bstore, konts, time, ndk) = nds
  val new_time = tick(nds)
  konts match {
    case Nil ⇒ ndcontinue(nds, seen)
    case Frame(x, e, f_env)::konts ⇒
      val baddr = allocBind(x, new_time)
      val new_env = f_env + (x ↦ baddr)
      val new_store = bstore.update(baddr, atomicEval(ae, env, bstore))
      drive_step(NDState(e, new_env, new_store, konts, new_time, ndk), seen)
  }
}
```

continue is mainly what we have seen for the second case in drive_step, except that ndcontinue is called when the normal continuation is empty. Since we are analyzing a language in ANF, the first-order data representation of continuations is simplified to a Scala List. An empty list Nil represents the halt of a computation path, whereas a cons :: means the top frame of the list holds the binding variable and evaluation context. An analogy to this in standard CEK machines would be an abstract class (e.g., cont) with three variants: halt, arg(value, cont), and arg(exp, env, cont).

```
def ndcontinue(nds: NDState, seen: Set[State]): Set[State] = {
  val NDState(ae, env, bstore, konts, time, ndk) = nds
  ndk match {
    case NDCont(Nil, _, _, _, _)::ndk ⇒
      drive_step(NDState(ae, env, bstore, konts, time, ndk), seen)
    case NDCont(cls, argvs, bstore, time, frames)::ndk ⇒
      val Clos(Lam(v, body), c_env) = cls.head
      val baddr = allocBind(v, time)
      val new_env = c_env + (v ↦ baddr)
      val new_store = bstore.update(baddr, argvs)
      drive_step(NDState(body, new_env, new_store, frames, time,
                         NDCont(cls.tail, argvs, bstore, time, frames)::ndk),
                 seen)
  }
}
```

The shape for ndcontinue is similar to continue; after all, they both use a List representation for continuations. When the closure set is empty, we recursively call drive_step with the rest of ndk, since in this case the normal stack konts is still empty, so the state will be dispatched to ndcontinue again. Otherwise, we invoke drive_step with the body expression of the next closure object.

6 REFUNCTIONALIZATION

Refunctionalization transforms first-order data types representing evaluation contexts to higher-order functions. Besides sequentializing the order of abstract evaluation by functions, this transformation can be also regarded as allocating continuation functions of the defined language in the heap of the metalanguage.

In this section, several other notable changes have been made:

- We do not have the concept of state (or NDState) anymore. The components of the states are lifted as arguments of the evaluation function.

- Since our final target is an abstract definitional interpreter (though still nondeterministic), starting from this section our abstract semantic artifact returns a set of final values instead of collected states.
- As we are getting close to achieving abstract definitional interpreters, the name of `drive_step` is updated to `aeval` (for *abstract eval*).

For clarity, we first show how to refunctionalize normal continuations and nondeterministic continuation without collecting semantics. Then, adding an appropriate caching algorithm to make sure the analysis will terminate at some fixed-point is a simple transformation of the program to cache-passing style. Finally, we examine how computable pushdown control flow analysis is established through refunctionalization and caching.

6.1 First Try

To properly represent final values, we introduce the case class `VS` which contains a set of storable values, a timestamp, and a store.

```
case class VS(vals: Set[Storable], time: Time, store: BStore)
```

An instance of `VS` represents the computational result of one path in the nondeterministic evaluation. The reason that we include a store is that there might be a case in which two paths have the same values but different accumulated side effects (e.g., memory allocations). Since the whole computation is nondeterministic, the type of the final result values is a collection `Set[VS]`. For convenience, we define it as a type `Ans`:

```
type Ans = Set[VS]
```

As previously mentioned, the components of states are lifted as arguments to `aeval`. Therefore, the pattern matching on the state becomes just a match on expression `e`.

```
type Cont = Ans ⇒ Ans
def aeval(e: Expr, env: Env, store: BStore, time: Time, continue: Cont): Ans = {
  val new_time = (e::time).take(k)
  e match {
    case Let(x, App(f, ae), e) ⇒
      ......
    case ae if isAtomic(ae) ⇒
      ......
  }
}
```

An additional argument `continue` which has type `Cont` is also introduced. Recall that in the disentangled AAM, the second case in `drive_step` makes a call to `continue`. Here, the reemerging `continue` is a function with type `Ans ⇒ Ans`. The function `aeval` does not return: instead, it calls `continue` with the result values.

Nondeterminism Abstractions. To handle the inevitable nondeterminism, we define a generic function `nd` as follows:

```
def nd[T,S](ts: Set[T], acc: S, f: (T, S, S ⇒ S) ⇒ S, g: S ⇒ S): S = {
  if (ts.isEmpty) g(acc)
  else f(ts.head, acc, (vss: S) ⇒ nd(ts.tail, vss, f, g))
}
```

Given a set of elements of type `T`, an initial accumulated value of type `S`, a function `f` that works on element type `T` and accumulated type `S`, and a function `g` that works on and returns a value of type S, nd applies f to each element in the set iteratively, accumulates the value and finally applies g on it.

If we look at continue and ndcontinue in the disentangled AAM, we will find that nd is an abstraction of those two functions. In function continue, there are two out calls to ndcontinue and drive_step, this is the reason why we have functions f and g in nd.

Readers familiar with functional programming may think of fold at this point; indeed, it is an implementation of fold in continuation style. But here we have two continuations, one for the next element in the set, and one for the final result. This is sometimes referred to as an extended continuation-passing style (ECPS) that has a continuation and a meta-continuation [Danvy and Filinski 1990].

Evaluation without Caching. Since our first version of aeval implements the nondeterministic evaluation, given the defined nd, the evaluation function aeval can be easily defined. In the case that e is a let expression, we invoke nd for twice and perform a nesting two-step, depth-first evaluation over possible values of App(f, ae). The values of branches from on fork point will be accumulated, and then returned to the upper fork point.

```
case Let(x, App(f, ae), e) ⇒
  val closures = atomicEval(f, env, store).asInstanceOf[Set[Clos]]
  nd[Clos, Ans](closures, Set[VS](), { case (clos, accouter, closnd) ⇒
    val Clos(Lam(v, body), c_env) = clos
    val baddr = allocBind(v, new_time)
    val new_env = c_env + (v ↦ baddr)
    val new_store = store.update(baddr, atomicEval(ae, env, store))
    aeval(body, new_env, new_store, new_time, (bodyvss: Set[VS]) ⇒ {
      nd[VS, Ans](bodyvss, Set[VS](), { case (vs, accinner, bdnd) ⇒
        val VS(vals, time, store) = vs
        val new_env = env + (x ↦ baddr)
        val new_store = store.update(baddr, vals)
        aeval(e, new_env, new_store, time,
              /* accumulate the values of one path, and call next body value*/
              (evss: Ans) ⇒ bdnd(accinner ++ evss))
      },
      /* accumulate the values of paths to accouter, and call next closure */
      (evss: Ans) ⇒ closnd(evss ++ accouter))
    })
  },
  continue)
```

The outer call to nd evaluates over the set of possible closures of f. For each such closure, we go into its body with a new environment and a new store. Remember that the function aeval returns a set of VS objects to its continuation, so the continuation argument bodyvss is a set of VS objects which represent values from multiple computation paths when evaluating this single body expression.

We next consider the inner call to nd, which evaluates over the set of body values bodyvss. For each VS, the timestamp and store in VS will be instantiated to aeval. The inner application of aeval returns a set of values of one computation path to its continuation, where the continuation will accumulate the result and transfer evaluation to the next body value.

```
case ae if isAtomic(ae) ⇒
  val ds = atomicEval(ae, env, store)
  continue(Set(VS(ds, new_time, store)))
```

If the expression is atomic, then we simply construct a new instance of VS and return it to the `continue` function.

At the end, the initial invocation to `aeval` is passed with an identity function as a halting continuation:

```
def analyze(e: Expr) =
  aeval(e, Map[String, BAddr](), Store[BAddr, Storable](Map()), List(), (ans ⇒ ans))
```

6.2 Caching Fixed-Points

Based on the refunctionalized AAM from the last section, we use the same nondeterministic operator `nd`, but further extend it to a fixed-point cached evaluation in this section. The fixed-point caching algorithm performs a sound over-approximation of all possible concrete reachable paths and values, and also prevents non-termination when analyzing diverged programs.

Configuration. We do not have the concept of an explicit stack or state, so to represent the cache we introduce a configuration `Config` as a state-like definition which borrows the components from state, sans continuations.

```
case class Config(e: Expr, env: Env, store: BStore, time: Time)
```

Cache. Recall that the latent assumption of the worklist algorithm in classical AAM is that if we have seen a state `s`, it means we also have seen the successors of state `s`. The caching algorithm we adopt here replays the same assumption but in a big-step manner: if we have seen the configuration `c`, then it means we also have seen the values that are evaluated by the configuration `c`. Here we will apply the fixed-point caching algorithm as described from Darais et al.'s ADI paper [Darais et al. 2017].

The case class `Cache` and its operations are defined as follow:

```
case class Cache(in: Store[Config, VS], out: Store[Config, VS]) {
  def inGet(config: Config): Set[VS] = in.getOrElse(config, Set())
  def outGet(config: Config): Set[VS] = out.getOrElse(config, Set())
  def outContains(config: Config): Boolean = out.contains(config)
  def outUpdate(config: Config, vss: Set[VS]): Cache = Cache(in, out.update(config, vss))
}
```

The `Cache` contains two maps `in` and `out` which are both mappings from configurations to sets of VS. The `in` cache contains mappings from the previous iteration of evaluation, and the `out` cache contains mappings after the current iteration of evaluation. Once we have evaluated a term to some values, we update the `out` cache. In the next iteration, we will use the `out` from the previous iteration as the `in` cache for the current iteration.

Cache-Passing Style. We can now transform our interpreter into cache-passing style. We begin by changing the return type `Ans` to a case class which contains a VS set and a cache.

```
case class Ans(vss: Set[VS], cache: Cache)
```

The skeleton of function `aeval` largely remains the same. Every place which has a value of `Ans` is also replaced as direct pattern matching so that we can use the fields in `Ans` immediately. The cache is used in a monotonic way: each function call to `aeval` or `nd` is passed with the latest cache from the most recent continuation's result.

Upon entering `aeval`, we first determine whether the `out` cache contains some values for the current configuration and use them immediately (if present). This corresponds to small-step AAM with a worklist: when we have seen a state, we can simply discard the state and continue working through the rest of the worklist `todo`.

If, however, we have not hit the `out` cache, we retrieve the values from the `in` cache, and update the `out` cache with values from the `in` cache. This retrieval from the `in` cache may return an empty set of values if there is no such mapping for the configuration from the previous iteration. In reality, we first conservatively assume that all computations can be diverged, and if not, we update its values in the mapping after evaluation. In terms of partial orders and lattices, it is the case that starts the Kleene iteration from the bottom of the lattice.

```
def aeval(e: Expr, env: Env, store: BStore, time: Time, cache: Cache, continue: Cont): Ans = {
  val config = Config(e, env, store, time)
  /* lookups the out cache, uses the values if contains this config */
  if (cache.outContains(config)) return continue(Ans(cache.outGet(config), cache))
  /* uses the values from in cache */
  val new_cache = cache.outUpdate(config, cache.inGet(config))
  val new_time = (e::time).take(k)
  e match {
    case Let(x, App(f, ae), e) ⇒
      val closures = atomicEval(f, env, store).asInstanceOf[Set[Clos]]
      nd[Clos, Ans](closures, Ans(Set[VS](), new_cache), { case (clos, Ans(accouter, cache), closnd) ⇒
        ......
        aeval(body, new_env, new_store, new_time, cache, { case Ans(bodyvss, bodycache) ⇒
          nd[VS, Ans](bodyvss, Ans(Set[VS](), bodycache), { case (vs, Ans(acc_vss, cache), bdnd) ⇒
            ......
            aeval(e, new_env, new_store, time, cache, { case Ans(evss, ecache) ⇒
              bdnd(Ans(accinner ++ evss, ecache))
            })
          },
          { case Ans(evss, cache) ⇒ closnd(Ans(evss ++ accouter, cache)) })
        })
      },
      /* updates the out cache after evaluation */
      { case Ans(vss, cache) ⇒ continue(Ans(vss, cache.outUpdate(config, vss))) })

    case ae if isAtomic(ae) ⇒
      val vs = Set(VS(atomicEval(ae, env, store), new_time, store))
      continue(Ans(vs, new_cache.outUpdate(config, vs))) /* updates the out cache after evaluation */
  }
}
```

After completing the evaluation, we obtain a set of `VS` objects and must update the `out` cache. In the first case of pattern matching, this happens inside the last continuation of the outer `nd` call. For the second case, we update the `out` cache before calling the `continue` continuation.

```
def analyze(e: Expr) = {
  def iter(cache: Cache): Ans = {
    val Ans(vss, new_cache) = aeval(e, Map[String, BAddr](), Store[BAddr, Storable](Map()), List(), cache, {
      val initConfig = Config(e, Map[String, BAddr](), Store[BAddr, Storable](Map()), List())
      case Ans(vss, cache) ⇒ Ans(vss, cache.outUpdate(initConfig, vss))
    })
    if (new_cache.out == cache.out) { Ans(vss, new_cache) }
    else { iter(Cache(new_cache.out, new_cache.out)) }
  }
  iter(mtCache)
}
```

Finally, the `analyze` function (as the entrance of the analysis) does a looping iteration to find the fixed-point on caches starting from an empty cache. If the `out` cache of this iteration is equivalent to the `out` cache from the last iteration, then we have reached a fixed-point that over-approximates the concrete evaluations, and thus can be returned. Otherwise, the next iteration with the new `out` cache will be activated. Note that the continuation of the initial call to `aeval` sets up the values for the initial configuration in the final cache.

6.3 Pushdown Control Flow Analysis, Revisited

In the previous section, we have established a computable pushdown control-flow analysis through refunctionalization and caching. In this section, we revisit the pushdown control flow problem and examine what we have done to overcome it.

The Problem with Return-flows. Pushdown control flow is a property in analysis that precisely models the runtime call stack of the analyzed program. A pushdown control flow analysis provides an as-exact-as-runtime return-flow when analyzing a program, but traditional control flow analysis collapses the state space into a finite space and thus causes imprecise stack modeling.

To see how traditional control flow analysis suffers from spurious return-flows, we can consider the following example:

```
(let ([id (lambda (z) z)])
  (let ([x (id 1)])
    (let ([y (id 2)])
      x)))
```

In k-CFA algorithm or the abstract abstract machine shown in Section 2.3, the call sites `(id 2)` and `(id 1)` share the same return flow, so the invocation of `(id 2)` returns to both call-sites `[x (id 1)]` and `[y (id 2)]`. Therefore, the returned value `2` for variable `y` is also propagated to variable `x`, causing imprecise analysis results to arise. This return-flow merging is inevitable, even when increasing the context-sensitivity. If we use the monovariant analysis (i.e., 0-CFA), the analysis result would be such that `x` and `y` point to value set `{1, 2}`, because the algorithm does not distinguish that `z` comes from a different call site. Under 1-CFA, the algorithm is able to distinguish that variable `z` of function `id` has two different values at two call sites, so variable `y` would not be polluted by `1`. However, variable `x` still points to value set `{1, 2}`, because two call-sites still share the same continuation 1-CFA is still unable to separate the return-flows.

Call/Return Matching through Refunctionalization and Caching. The refunctionalized AAM we obtained has the perfect match on return-flows even though we do not have a stack model. The higher-order functions representing continuations already connect all the execution in order. In fact, the continuations (call stack) of the analyzed language is blended into the call stack of our defining language (Scala) through refunctionalization, then the call/return of the analyzed language is naturally matched.

In fact, what we started is an AAM with an unbounded stack which already precisely matches the calls and returns. Refunctionalization makes the interpreter be a "big-step" semantics artifact, then the consequence is we have no places to save the context information. But refunctionalization not only forces us to sequentialize the order non-deterministic evaluation through higher-order functions representing continuations, but also drive us to use a new caching algorithm.

Indeed, the caching algorithm plays an important role to make the analysis computable.

An interesting implication of this is if we apply the caching algorithm with some necessary changes to small-step abstract machines with an unbounded stack, we are also able to establish a computable and precise call/return match.

7 TO DIRECT-STYLE

In the previous section, we presented a refunctionalized AAM utilizing the `nd` operator in extended continuation-passing style. To obtain definitional abstract interpreters, we now transform it further to direct-style.

We have multiple choices of how to proceed:

- By monadifying the continuations, we obtain the abstracting definitional interpreters in monadic style which are what Darais et al. describe [Darais et al. 2017].
- By representing the extended continuation-passing style with delimited controls, we derive a new form of abstract interpreters in direct-style.
- In fact, we may also just use the same caching algorithm but with side effects such as assignments and mutations to update the cache, and then achieve the same definitional interpreter with pushdown control flows. In this case, nondeterminism can be easily handled via `for` comprehension.

These coincidences should not be a surprise since existing literature is extraordinarily rich in showing the correspondence between monads and continuation-passing style as well as delimited controls since their very beginning era [Danvy and Filinski 1990, 1992; Moggi 1991; Wadler 1992].

In this section, we present the second version that uses delimited control operators.

7.1 Un-CPS by Delimited Controls

As previously identified, there are two layers of continuations: one for the normal stack of the analyzed language, and the other for the nondeterministic choices of closures. This corresponds to the extended continuation-passing style. But in our defining language (Scala), the delimited controls rely on the continuation-passing transformation which is only one layer.

To address this problem, we reintroduce two operators `nd` and `ndcps` for handling non-deterministic choices in a way that mimics extended continuation-passing style.

Nondeterminism Operators, Again. The first operator `nd` is simplified from the previous one, but specialized with type `Ans` and `Cache`. We may still keep the types to be generic, but the purpose of presentation clearly, exposing `Cache` explictly reveals that cache should be monotonically accumulated to the rest of computation. Now `nd` only takes a function `f` that applies on type `(T, Cache)` where value of type `T` comes from the set, and `nd` does recursive call on itself over elements in the set `ts`. We also move the accumulation operation (++) inside of `nd`.

```
def nd[T](ts: Set[T], acc: Ans, k: ((T, Cache)) ⇒ Ans): Ans = {
  if (ts.isEmpty) acc
  else nd(ts.tail, acc ++ k(ts.head, acc.cache), k)
}
```

The second operator `ndcps` is implemented with delimited control operator `shift` but simply calls our first operator `nd` in body.

```
def ndcps[T](ts: Set[T], acc: Ans): (T, Cache) @cps[Ans] = shift { f: (((T, Cache)) ⇒ Ans) ⇒
  nd(ts, acc, f)
}
```

The function `ndcps` takes two arguments: a set of elements with type `T`, and an initial accumulated value of type `Ans`. The `ndcps` iteratively returns an element from the set along with the latest cache , and the return type is also annotated by `@cps[Ans]`. This annotation denotes the final returned value of `ndcps` is type `Ans`. The `f` introduced by the `shift` operator is the delimited continuation from the call-site of `ndcps`.

Now we can use these two operators to redefine `aeval`. We add an annotation `@cps[Ans]` on the return type of the `aeval` function. Then, for every nondeterministic choice in the abstract interpreter, we can simply call `ndcps` on the set of possible choices and write the program sequentially as a concrete interpreter. `ndcps` also returns the latest cache to its left-hand side definition. The cache is still accumulated to the next call to `ndcps` or `aeval`, given the fact that the abstract interpreter should always use the latest cache.

```
def aeval(e: Expr, env: Env, store: BStore, time: Time, cache: Cache): Ans @cps[Ans] = {
  val config = Config(e, env, store, time)
  if (cache.outContains(config)) Ans(cache.outGet(config), cache)
  else {
    val new_time = (e::time).take(k)
    val new_cache = cache.outUpdate(config, cache.inGet(config))
    e match {
      case Let(x, App(f, ae), e) ⇒
        val closures = atomicEval(f, env, store).asInstanceOf[Set[Clos]]
        val (Clos(Lam(v, body), c_env), clscache) = ndcps[Clos](closures, Ans(Set[VS](), new_cache))
        val vbaddr = allocBind(v, new_time)
        val new_cenv = c_env + (v ↦ vbaddr)
        val new_cstore = store.update(vbaddr, aeval(ae, env, store))
        /* evaluates the function application App(f, ae) */
        val Ans(bodyvss, bodycache) = aeval(body, new_cenv, new_cstore, new_time, clscache)
        val (VS(vals, time, vsstore), vscache) = ndcps[VS](bodyvss, Ans(Set[VS](), bodycache))
        val baddr = allocBind(x, time)
        val new_env = env + (x ↦ baddr)
        val new_store = vsstore.update(baddr, vals)
        /* evaluates the body expression from the let */
        val Ans(finval, fincache) = aeval(e, new_env, new_store, time, vscache)
        Ans(finval, fincache.outUpdate(config, finval))

      case ae if isAtomic(ae) ⇒
        val vs = Set(VS(atomicEval(ae, env, store), new_time, store))
        Ans(vs, cache.outUpdate(config, vs))
    }
  }
}
```

The function `analyze` is also changed by adding a `reset` operator around the function call to `aeval`. The afterwards updating on cache becomes sequential.

```
def analyze(e: Expr) = {
  def iter(cache: Cache): Ans = {
    reset {
      val Ans(vss, anscache) = aeval(e, Map[String, BAddr](), Store[BAddr, Storable](Map()), List(), cache)
      val initConfig = Config(e, Map[String, BAddr](), Store[BAddr, Storable](Map()), List())
      val new_cache = anscache.outUpdate(initConfig, vss)
      if (new_cache.out == cache.out) { Ans(vss, new_cache) }
      else { iter(Cache(new_cache.out, new_cache.out)) }
    }
  }
  iter(mtCache)
}
```

Now we have arrive the end of this series of transformations. Starting from an small-step abstract abstract machine, eventually we obtain an big-step definitional interpreter with pushdown control-flows and written in direct-style.

8 RELATED WORK

Abstract Interpretation and Control-flow Analysis. Cousot and Cousot invented abstract interpretation as a sound approach to approximate a program's runtime behavior[Cousot and Cousot 1977]. Control flow analysis is one instance of abstract interpretation on functional programs that can be traced to Jones [Jones 1981]. Shivers introduces k-CFA which uses k recent calling contexts as program contour that differentiates values from different contexts [Shivers 1988, 1991]. The modern formulation of control flow analysis is the abstracting abstract machines methodology [Van Horn and Might 2010, 2012] which forms the starting point of this paper. Modern programming languages such as Java[Might et al. 2010] and Racket [Tobin-Hochstadt and Van Horn 2012] can also be modeled by the abstracting abstract machine approach.

Defunctionalization and Refunctionalization. Defunctionalization and refunctionalization build connections between abstract machines and interpreters. Reynolds first shows defunctionalization as a program transformation technique that can be used to transform higher-order functions to first-order functions when specifying a programming language by an interpreter [Reynolds 1972]. Refunctionalization is the reverse of this transformation.

Over the years, Danvy and his collaborators applied refunctionalization and defunctionalization to many different abstract machines and semantics, including the CEK machines, the CLS machines, the SECD machines, etc. [Ager et al. 2003, 2004, 2005; Biernacka and Danvy 2009; Danvy 2006, 2008, 2009; Danvy and Nielsen 2001, 2004] Other applications are also found to be applicable, such as Dyck word recognizer, and Dijkstra's shunting-yard algorithm [Danvy 2006].

For deterministic languages, Danvy shows a reduction semantics is a structural operational semantics in continuation style, where the reduction context is a defunctionalized continuation [Danvy 2008]. Our work further shows that for nondeterministic languages, there is another layer of continuations that controls the nondeterministic choices. After identifying and explicitly exposing (Section 3) that layer of nondeterministic continuations, the whole program can then be refunctionalized to extended continuation-passing style.

Fusion as one of the transformation in our paper is also studied by Ohori and Sasano [Ohori and Sasano 2007].

Pushdown Control Flow Analysis. In recent years, there are significant efforts [Earl et al. 2012; Gilray et al. 2016b; Johnson and Van Horn 2015; Vardoulakis and Shivers 2010] to achieve precise call/return match based on small-step abstracting abstract machines.

CFA2 is the first solution that solves the return-flows problem [Vardoulakis and Shivers 2010], but CFA2 has several drawbacks: it works only on continuation-passing style programs, and does not supports polyvariant analysis, in addition to an exponential time complexity. Pushdown control flow analysis (PDCFA) is a mechanism which maintains this precision through the use of a Dyke state graph representing all possible stacks contained within the unbounded-stack machine[Earl et al. 2010, 2012]. Similar to PDCFA, abstracting abstract control (AAC) is another strategy for maintaining stack precision [Johnson and Van Horn 2015]. AAC functions by utilizing continuations which are specific to both the source and target states of a call-site transition, which guarantees that no spurious merging will occur during returns.

Gilray et al. further proposed a polyvariant continuation-addresses allocator for small-step AAM to achieve pushdown analysis [Gilray et al. 2016b]. This method is both simple to implement and computationally inexpensive and so called Pushdown for Free (P4F). Based on the AAM we presented in Section 2.3, the only changes in the code required is not only keeping track of the entry-point expression of callee, but also holding the target environment when allocating a continuation addresses. No other pieces of code would need to be modified. Notably, this change only causes constant-factor increase in time complexity of the analysis if the store is widened.

Our work establish the pushdown control-flow analysis through refunctionalization and a proper caching algorithm. The call/return is naturally matched by the stack of the meta language.

Abstracting Definitional Interpreters. Reynold's seminal paper *Definitional interpreters for higher-order programming languages* [Reynolds 1972] shows that in definitional interpreters, the defined language can inherit properties from the defining language. With this insight, Darais et al. construct abstracting definitional interpreters which automatically inherit the pushdown control-flow property from its defining language because the defined language simply uses the stack the meta-language [Darais et al. 2017]. Darais et al.'s abstracting definitional interpreters work on direct-style programs, and is written in monadic style. One of the advantages of monadic abstract interpreters is modular and composable, therefore deploying different sensitivities or features is just applying a different monad. Prior to that, Sergey et al. show a monadic abstract interpreter for small-step semantics [Sergey et al. 2013].

This paper is greatly inspired by Darais et al.'s work. The difference on the surface is that ours works for A-Normal Form λ-calculus, instead of plain λ-calculus, although our work can also be easily adopted to handle a plain λ-calculus. But except of simply showing the final form of abstract definitional interpreters, we reveal that refunctionalization plays an important role for inheriting the stack from the metalanguage. The target of the series of transformations in this paper is abstract definitional interpreters, and we additionally demonstrate abstract interpreters written direct-style with delimited controls. We use the delimited controls `shift` and `reset` that are implemented in Scala [Rompf et al. 2009]. The correspondence of delimited controls (as well as continuations) and monads are well-known [Danvy and Filinski 1990, 1992; Moggi 1991; Wadler 1992].

9 CONCLUSION

In this functional pearl, we fill the gap between small-step abstract abstract machines and big-step abstract definitional interpreters by developing a series of syntactical transformations. Among these transformations, linearization turns a worklist into another layer of continuations; refunctionalization converts the first-order data types representing continuations to higher-order functions; and finally un-CPS with delimited control transforms the abstract interpreter into a sequence of expressions, which looks no different to a concrete interpreter.

We show the correspondence not only exists between concrete semantics artifacts but also exists between abstract semantics artifacts. An interesting open question would be whether there also exists a correspondence of static analyses formalized in different denotational style and operational style.

REFERENCES

Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. A Functional Correspondence Between Evaluators and Abstract Machines. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming (PPDP '03)*. ACM, New York, NY, USA, 8–19. https://doi.org/10.1145/888251.888254

Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. 2004. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Inform. Process. Lett.* 90, 5 (2004), 223 – 232. https://doi.org/10.1016/j.ipl.2004.02.012

Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. 2005. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science* 342, 1 (2005), 149–172.

Małgorzata Biernacka and Olivier Danvy. 2009. Towards compatible and interderivable semantic specifications for the Scheme programming language, Part II: Reduction semantics and abstract machines. In *Semantics and algebraic specification.* Springer, 186–206.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 238–252.

Olivier Danvy. 2006. Refunctionalization at Work. In *Proceedings of the 8th International Conference on Mathematics of Program Construction (MPC'06)*. Springer-Verlag, Berlin, Heidelberg, 4–4. https://doi.org/10.1007/11783596_2

Olivier Danvy. 2008. Defunctionalized Interpreters for Programming Languages. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*. ACM, New York, NY, USA, 131–142. https://doi.org/10.1145/1411204.1411206

Olivier Danvy. 2009. Towards compatible and interderivable semantic specifications for the Scheme programming language, Part I: Denotational semantics, natural semantics, and abstract machines. In *Semantics and algebraic specification*. Springer, 162–185.

Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90)*. ACM, New York, NY, USA, 151–160. https://doi.org/10.1145/91556.91622

Oliver Danvy and Andrzex Filinski. 1992. Representing control: A study of the CPS transformation. *Mathematical structures in computer science* 2, 4 (1992), 361–391.

Olivier Danvy and Lasse R. Nielsen. 2001. Defunctionalization at Work. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '01)*. ACM, New York, NY, USA, 162–174. https://doi.org/10.1145/773184.773202

Olivier Danvy and Lasse R Nielsen. 2004. Refocusing in reduction semantics. *BRICS Report Series* 11, 26 (2004).

David Darais, Nicholas Labich, Phúc C Nguyen, and David Van Horn. 2017. Abstracting definitional interpreters (functional pearl). *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 12.

Christopher Earl, Matthew Might, and David Van Horn. 2010. Pushdown control-flow analysis of higher-order programs. *arXiv preprint arXiv:1007.4268* (2010).

Christopher Earl, Ilya Sergey, Matthew Might, and David Van Horn. 2012. Introspective pushdown analysis of higher-order programs. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 177–188.

Mattias Felleisen and Daniel 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*. ACM, 314.

Cormac Flanagan, Amr Sabry, Bruce F Duba, and Matthias Felleisen. 1993. The essence of compiling with continuations. In *ACM Sigplan Notices*, Vol. 28. ACM, 237–247.

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)*. ACM, New York, NY, USA, 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 *ACM SIGPLAN Notices*, Vol. 51. ACM, 691–704.

James Ian Johnson and David Van Horn. 2015. Abstracting abstract control. *ACM SIGPLAN Notices* 50, 2 (2015), 11–22.

Neil D Jones. 1981. Flow analysis of lambda expressions. In *International Colloquium on Automata, Languages, and Programming*. Springer, 114–128.

Peter J Landin. 1966. The next 700 programming languages. *Commun. ACM* 9, 3 (1966), 157–166.

Matthew Might, Yannis Smaragdakis, and David Van Horn. 2010. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *ACM Sigplan Notices*, Vol. 45. ACM, 305–315.

Eugenio Moggi. 1991. Notions of computation and monads. *Information and computation* 93, 1 (1991), 55–92.

Atsushi Ohori and Isao Sasano. 2007. Lightweight Fusion by Fixed Point Promotion. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1190216.1190241

John C Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2*. ACM, 717–740.

Tiark Rompf, Ingo Maier, and Martin Odersky. 2009. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *ACM Sigplan Notices*, Vol. 44. ACM, 317–328.

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

O. 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

Sam Tobin-Hochstadt and David Van Horn. 2012. Higher-order Symbolic Execution via Contracts. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. ACM, New York, NY, USA, 537–554. https://doi.org/10.1145/2384616.2384655

David Van Horn and Matthew Might. 2010. Abstracting abstract machines. In *ACM Sigplan Notices*, Vol. 45. ACM, 51–62.

David Van Horn and Matthew Might. 2012. Systematic abstraction of abstract machines. *Journal of Functional Programming* 22, 4-5 (2012), 705–746.

Dimitrios Vardoulakis and Olin Shivers. 2010. CFA2: a context-free approach to control-flow analysis. In *European Symposium on Programming*. Springer, 570–589.

Philip Wadler. 1992. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1–14.