Refunctionalization of Abstract Abstract Machines

Bridging the Gap between Abstract Abstract Machines and Abstract Definitional Interpreters (Functional Pearl)

GUANNAN WEI, Purdue University, USA
JAMES DECKER, Purdue University, USA
TIARK ROMPF, Purdue University, USA

Abstracting abstract machines is a systematic methodology for constructing sound static analyses for higher-order languages, by deriving small-step *abstract* abstract machines (AAMs) that perform abstract interpretation from abstract machines that perform concrete evaluation. Darais et al. apply the same underlying idea to monadic definitional interpreters, and obtain monadic *abstract* definitional interpreters (ADIs) that perform abstract interpretation in big-step style using monads. Yet, the relation between small-step abstract abstract machines and big-step abstract definitional interpreters is not well studied.

In this paper, we explain their functional correspondence and demonstrate how to systematically transform small-step abstract abstract machines into big-step abstract definitional interpreters. Building on known semantic interderivation techniques from the concrete evaluation setting, the transformations include linearization, lightweight fusion, disentanglement, refunctionalization, and the left inverse of the CPS transform. Linearization expresses nondeterministic choice through first-order data types, after which refunctionalization transforms the first-order data types that represent continuations into higher-order functions. The refunctionalized AAM is an abstract interpreter written in continuation-passing style (CPS) with two layers of continuations, which can be converted back to direct style with delimited control operators. Based on the known correspondence between delimited control and monads, we demonstrate that the explicit use of monads in abstract definitional interpreters is optional.

All transformations properly handle the collecting semantics and nondeterminism of abstract interpretation. Remarkably, we reveal how precise call/return matching in control-flow analysis can be obtained by refunctionalizing a small-step abstract abstract machine with proper caching.

CCS Concepts: • **Theory of computation** → **Abstract machines**; • **Software and its engineering** → *Functional languages*; *Interpreters*;

Additional Key Words and Phrases: refunctionalization, abstract machines, control-flow analysis, Scala

ACM Reference Format:
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 (September 2018), 28 pages. https://doi.org/10.1145/3236800

1 INTRODUCTION

Implementing, and in some cases defining, a programming language by building an interpreter for it can be traced to the very early days of programming languages research [Landin 1966; McCarthy

Authors' addresses: Guannan Wei, Department of Computer Science, Purdue University, 305 N. University Street, West Lafayette, IN, 47907, USA, wei220@purdue.edu; James Decker, Department of Computer Science, Purdue University, 305 N. University Street, West Lafayette, IN, 47907, USA, decker31@purdue.edu; Tiark Rompf, Department of Computer Science, Purdue University, 305 N. University Street, West Lafayette, IN, 47907, USA, tiark@purdue.edu.

2475-1421/2018/9-ART105
https://doi.org/10.1145/3236800

1960; Reynolds 1972]. Nowadays, even undergraduate students in computer science learn how to implement their own programming languages by building an interpreter. However, implementing a sound static analysis by building an abstract interpreter [Cousot and Cousot 1977] remained an esoteric and difficult task until very recently.

Van Horn and Might [2010, 2012] 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. Given a concrete small-step abstract machine that models a store (e.g., the CESK machine, Krivine's machine, etc.), if we allocate continuations in the store and draw all allocated addresses (both for regular values and for continuations) from a finite set, then we obtain an abstract interpreter with a finite state space, which can be used for performing sound static analysis and is guaranteed to terminate. By using different address allocators [Gilray et al. 2016a], one can further instantiate different polyvariant control-flow analyses.

Applying the same idea to monadic definitional interpreters, Darais et al. [2017] showed how to build monadic abstract definitional interpreters (ADIs) in big-step style. 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 question, and the subject of this paper, is how one can derive one abstract semantic artifact given the other. The main contribution of this paper is to explain the functional correspondence between these abstract semantic artifacts, and to demonstrate the concrete syntactic transformation steps necessary to transform an abstract abstract machine into its corresponding abstract definitional interpreter. The reverse direction is comparatively easier, and can be obtained by following the respective inverse steps in the opposite order. This contribution fills an intellectual gap in our understanding of abstract semantic artifacts, and it also opens up further practical avenues for constructing static analysis tools based on principled and well-understood techniques.

In the concrete world, the relationships between 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 2006b, 2008, 2009; Danvy and Nielsen 2001, 2004]. These 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 reduction 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 semantic artifacts, by contrast, the relationship 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. One of the fundamental differences between concrete and abstract semantic artifacts is that a sound instance of the latter must consider facts about all possible execution paths, and thus introduce a layer of nondeterminism. In addition to ensuring termination, abstract semantic artifacts are usually equipped with a cache of reachable states; the cache is iteratively updated in a monotonic way, guaranteed to reach the least fixed-point eventually.

In addition, those abstract abstract machines with an unbounded stack naturally correspond to abstract definitional interpreters. We show that refunctionalizing an AAM with an unbounded stack (and utilizing the proper caching algorithm) leads to a pushdown control-flow analysis.

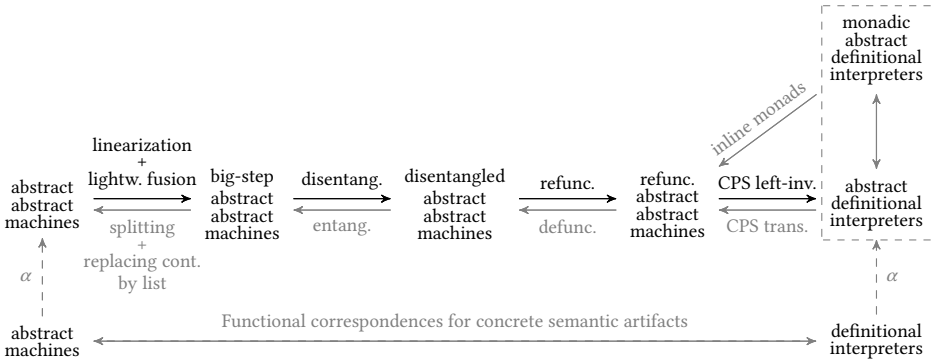

Fig. 1. Functional correspondence between and transformations from AAMs to ADIs

Contributions and Outline. We begin by reviewing necessary background in Section 2, as well as by introducing some of the basic code structures used throughout the paper. We then address the main contribution of this paper: bridging the gap between small-step AAMs and big-step ADIs by applying a series of well-known systematic transformations drawn from concrete semantic artifacts. Those transformations are shown in Figure 1 and summarized here, with full descriptions in their respective associated sections:

- **Linearization**: By expressing the nondeterministic choices as a first-order data type, we linearize the execution of abstract abstract machines; the transition of machine states thus becomes deterministic. Notably, this makes explicit another layer of control (Section 3).
- **Lightweight fusion** [Danvy and Millikin 2008a; Ohori and Sasano 2007]: We apply a fusion transformation to the linearized AAM, which combines the single-step function `step` and the driving function into one, but keeps all the machine state representations intact (Section 4).
- **Disentanglement**: We identify different first-order data types which represent different layers of continuations, and disassemble their handlers into separate functions (Section 5).
- **Refunctionalization** [Danvy 2006b; Danvy and Millikin 2009]: We apply refunctionalization to the disentangled AAM, which replaces the first-order data types representing continuations with higher-order functions, and associates dispatching logic with proper higher-order functions. We obtain an abstract interpreter written in continuation-passing style with an additional layer of continuations due to nondeterminism.
  For clarity, we first present a vanilla version of a refunctionalized AAM which simply converts the stack structures to higher-order functions but keeps other parts unchanged. We then adopt a caching algorithm [Darais et al. 2017] to guarantee the termination of abstract interpretation. At the end of this section, we review the pushdown control-flow analysis and examine how computable and precise call/return matching is obtained through these transformations (Section 6).
- **Direct-style transformation** [Danvy 1994]: We finally transform the refunctionalized AAM back to a direct-style interpreter by using delimited control operators (Section 7).

These transformations are used throughout the paper, with the refunctionalization and defunctionalization of abstract interpreters playing important roles for the stack model of the analyzed language. By refunctionalization, the call structure of the analyzed language is blended into the call structure of the defining language. This provides another perspective to explain why Darais

et al.'s abstract definitional interpreter is able to inherit the pushdown control-flow property from its defining language.

We discuss 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 a direct-style λ-calculus as our target where all "serious" expressions (i.e., function calls) are let-bound. This style is variously known as administrative normal form (ANF) [Flanagan et al. 1993], or monadic normal form [Danvy 2003; Moggi 1991]. 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 straightforward to implement without introducing new transformations related to the concerns of this paper, so we elide them here.

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

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

In ANF, an expression is either an atomic expression or a `let` expression. A restriction exists which states that all function applications (and only those) must be administered 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 is represented in Scala as follows. We assume that the source program conforms to the ANF convention, and as such, 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 the semantics of and evaluating a 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 addresses of all variables 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's execution context. In this paper, we instantiate the execution context as a call stack consisting of a list of frames.

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` corresponds to a reduction context in a reduction-based formulation of the semantics. An empty list represents an empty context, and corresponds to halt. Otherwise, the head frame in the list represents the innermost context, and the reduction result of this frame will be used to fill the "hole" of its following frame in the list. 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, and a control component to which the program may resume, as well as 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. The function `step` is a partial function that only handles non-final states, which must have a successor; the final case of a state is handled by function `drive`, which will be explained in Section 2.2.3.

```
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 non-final 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 the `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 control component 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 variable `x` from the top frame will be bound to the result of evaluating the atomic expression `ae` by updating the environment and store. Finally, the successor state is 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 a closed 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 which the control is an atomic expression and the stack structure is empty. Naturally, we can then extract the value from the final state.

```
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 abstract abstract machine implements a computable abstract semantics which approximates the run-time behaviors of programs. Since the state space of concrete execution is possibly infinite, the key insight of the AAM approach when analyzing programs is to allocate both variable bindings and continuations on the store, and then bound the addresses space to be finite. Since each component of the machine state is finite, the abstracted machine-state space is also finite, and therefore computable.

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

2.3.1 Machine Components. Similar to the concrete CESK machine, the machine state of the derived AAM has a control component, an environment, a store, a continuation, as well as a timestamp. However, there are several notable differences between an AAM's store and the 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 continuation component becomes an address that maps to a set of continuations in the store instead of directly embedded in the state.

For clarity, we divide the store into two separate stores: the binding store `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 the continuation store are updated monotonically; they grow continuously and never shrink.

```
case class Store[K,V](map: Map[K, Set[V]]) {
  def apply(addr: K): Set[V] = map(addr)
  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 co-domain of the binding store `Storable` is the same as previously defined for the CESK machine. The co-domain of the continuation store `Cont`, on the other hand, is comprised of a `Frame` object and a continuation address `KAddr`. To mimic the run-time call stack, `KAddr` plays the role of representing the remaining stack frames. But since the continuation store may contain multiple continuations, dereferencing of continuation addresses is also nondeterministic.

```
case class Frame(x: String, e: Expr, env: Env);      case class Cont(frame: Frame, kaddr: KAddr)
```

As a consequence, the components of machine states are also changed: the store is divided into a binding store and a continuation store; the continuation becomes an address that maps to a set of continuations in `KStore`. By dereferencing this address in a continuation store, we can retrieve the actual control-transfer destination. 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 described neither allocating of addresses in stores, nor handling of time stamps `Time`. In abstract interpretation, however, these are key ingredients to realize analyses with different sensitivities, as well as to perform a finite state space analysis [Gilray et al. 2016a]. To effectively approximate the run-time behavior, we introduce program *contours* `time` which are finite history of function calls till up to the current state [Shivers 1991]. The function `tick` is used to refresh the "time" and get a "new time". We use a finite list of expressions (which are drawn from the control component) to encode the calling context history, 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 states is finite when the space of addresses is finite. To make this happen, addresses of variable bindings are parameterized by variable names and the creation time of the binding, both of which are finite. Continuation addresses `KAddr` have two variants: 1) `Halt` which corresponds to the empty stack, and 2) `ContAddr` which consists of the call target expressions, also a finite set.

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

Given that the space of addresses is finite, we can conclude that there are only finite numbers of environments and stores because the numbers of variables and closures are also finite. This property guarantees a finite space of reachable states, and we can always realize a terminating analysis through Kleene's fixed-point iteration.

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 set of reachable successor states. We have two cases to consider (code shown below):

- If the current control component 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 machine, 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 set of successor states is generated.
- In the second case, an atomic expression `ae` sits on the control position of the state. Here, the values 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): Set[State] = {
  val new_time = s.tick
  s match {
    case State(Let(x, App(f, ae), e), env, bstore, kstore, kaddr, time) ⇒
      val closures = atomicEval(f, env, bstore)
      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) ⇒
      for (Cont(Frame(x, e, env_f), f_kaddr) <- kstore(kaddr)) 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. We implement `tick` as a public method of case class `State`.

```
def k: Int = 0
case class State(e: Expr, env: Env, bstore: BStore, kstore: KStore, kaddr: KAddr, time: Time) {
  def tick: Time = (e :: 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 machine, 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.

```
val mtTime = List();                        val mtStore = Store[BAddr, Storable](Map())
val mtEnv = Map[String, BAddr]();           val mtKStore = Store[KAddr, Cont](Map(Halt ↦ Set()))
def inject(e: Expr): State = State(e, mtEnv, mtStore, mtKStore, Halt, mtTime)
```

However, in contrast to the concrete CESK machine, the `drive` function performs collecting semantics instead of 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 work-list algorithm to find the fixed-point of the set of states. Function `drive` always applies function `step` to the head element `hd` of the work-list `todo` if `hd` is unseen. It then inserts the result of `step` to the rest of work-list, and in the meantime adds `hd` to the explored states set. If the work-list 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 model of the call stack as 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 structure as the same as at run-time (so-called pushdown control-flow analysis), but since the stack is unbounded, the state space is potentially infinite.

The pushdown AAM with an unbounded stack is also described by Van Horn and Might [2012]. We show it here is to maintain consistency of abstract semantics during transformations – since the eventual abstract definitional interpreters simply inherit a precise control-flow from their defining language, we also would like to start from a pushdown AAM that is also able to precisely match calls and returns. For readers who are not familiar with pushdown analysis, we provide a detailed discussion in Section 6.3.

In the definition of `State`, the continuation store and continuation address disappear; instead, a list of frames represents the stack. An empty list denotes that we have reached the halt. The other components remain unchanged as in the Section 2.3.

```
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 only when dereferencing the callee `f` from the function application `App(f, ae)`.

```
def step(s: State): Set[State] = {
  val new_time = s.tick
  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)) 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 ⇒ Set()
        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))
          Set(State(e, new_env, new_store, konts, new_time))
      } } }
```

In the first case of pattern matching, we may have multiple possible closures 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 machine.

Note on Termination. Unfortunately, even though other components in the state are finite, this version of AAM with an unbounded stack may still diverge when analyzing some programs. This is because the unbounded stack can grow to an 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 frames of `f1` and `f2` will be alternately pushed onto the top of the current stack. However, no two existing stack components are identical in the state space. One can decide the reachability of a state through Dyck state graph [Earl et al. 2010, 2012], but simply caching the explored states (i.e., `seen`) would not always make the analyzer terminate.

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. Now we begin describing the transformations step by step. Our base machine is the AAM with an unbounded stack; as we will later transform the stack structure to higher-order functions representing continuations, it does not matter what kind of AAM we start from. This is because it would not be possible to construct a new frame on the stack, nor to allocate continuations in the store after refunctionalization. Thus, our choice to start from an unbounded-stack AAM is motivated simply because it has an equivalent stack model to abstract definitional interpreters (our final target).

The underlying semantics of AAM is fundamentally nondeterministic: there are possibly multiple target closures when dereferencing an address, and we have to explore both branches if conditionals exist in the analyzed language. But the reduction context (i.e., the `Frame` in our program) is *not* nondeterministic, and the work-list implicitly handles the nondeterminism. Thus, if we simply refunctionalize the frames to higher-order functions, it does not help us to move toward abstract definitional interpreters.

In his paper *Defunctionalized Interpreters for Programming Languages*, Danvy observes that for deterministic languages, a reduction semantics can be seen as a structural operational semantics in continuation-passing style, where the reduction context is a defunctionalized continuation [Danvy 2008]. Therefore, refunctionalizing such a reduction semantics for a deterministic language yields an interpreter in continuation-passing style. Nevertheless, doing so for nondeterministic languages yields an interpreter with multiple layers of continuation [Danvy 2006b; Danvy and Millikin 2009], where the additional layer of continuation represents the nondeterministic choices.

We apply this idea to AAM, and thus our first transformation step 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 control, and define a case class `NDCont` that saves the information at a fork point when we have nondeterministic target closures. We also add a new field `ndk` of type `NDCont` 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 *continuation*, and the second a *meta-continuation* [1].

```
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)
  def tick: Time = ...
}
```

Here we also use first-order data types to explicitly represent the meta-continuation which controls the nondeterminism. Each `NDCont` object contains a list of closures that are possible functions to be invoked, a set of values that will be bound to the target function's formal argument, and the store, time, and frames at the fork point. In the definition of `NDState`, an auxiliary function `toState`

[1] These are also called *success continuation* and *failure continuation* [Danvy and Filinski 1990].

is added to convert itself to type `State`. `step` now becomes of type `NDState ⇒ Option[NDState]`. Returning a `None` value means reaching the halt.

```
def step(nds: NDState): Option[NDState] = {
  val new_time = nds.tick;  nds match { ... }
}
```

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
  Some(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 meta-continuation `new_ndk` is constructed, which contains the rest of closures, the values of evaluating `ae`, the store, the time, and the new frame list. We use the store before updating, because different closures may form different binding addresses `baddr`. We also use the *new* frames, because all the closures at this fork point share the same stack structure 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 Nil ⇒ None /* Halt */
      case NDCont(Nil, _, _, _, _)::ndk ⇒
        Some(NDState(ae, env, bstore, konts, time, ndk)) /* transfer to the front-most 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 */
        Some(NDState(body, new_env, new_store, frames, time, new_ndk))
    }
    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))
      Some(NDState(e, new_env, new_store, konts, new_time, ndk)) /* normal return */
  }
```

If the continuation `konts` is empty, then the machine has reached an end of the computation. Otherwise, we should return the values of `ae` to its caller which is contained in the top frame of the stack. However, since a meta-continuation `ndk` is introduced, we need to dispatch more cases when `konts` is empty:

- If the meta-continuation `ndk` is also empty, then the machine has reached the end of *all* computations. Besides, the control component `ae` now is an atomic value: `None` is returned.

- If the target closures set of the front-most NDCont is empty (Nil in the code), then we have computed all the closures at this fork point and should move to the next fork point. Since the meta-continuation nkd is represented by a list, and we always append new elements to its front, we resume to the next fork point by popping the front-most element and using the tail of that list.
- Otherwise, we pop a closure from the set, and build a new environment and store for evaluating the body expression of that closure. We use the same frame list and time which are copied from the fork point rather than current one. The meta-continuation new_ndk is updated by the removal of that popped closure, which guarantees the closure set will eventually shrink to empty.

```
def drive(nds: NDState, seen: Set[State]): Set[State] = {
  val s = nds.toState;  val new_seen = if (seen.contains(s)) seen else seen + s
  step(nds) match {
    case None ⇒ new_seen;  case Some(nnds) ⇒ drive(nnds, new_seen)
  } }
```

The drive function is also changed: there is no work-list anymore, as all the potentially unexplored states are embedded into the continuations. The termination of the analysis occurs when the value None is returned from calling step. Otherwise, we extract the next state nnds and recurse on function drive. The set of explored states is preserved as before.

At this point, we have obtained a linearized abstract abstract machine. Intuitively, if we imagine that the classical AAM explores a graph of reachable states, then the linearized version of AAM flattens the graph in depth-first order to a linear sequence of states. It has been observed that "*failure can be replaced by a list of successes*" [Wadler 1985] and that "*one programmer's lazy-list constructor is another programmer's success and failure continuations*" [Danvy 2006a]; the step function in AAM is the list constructor used in a lazy way, and here we turn it into an explicit data types representing continuation. The inverse transformation of this step is to apply Wadler's technique that converts the data types representing meta-continuations (failure continuations) back to a lazy list representation, where the elements in the list are plain machine states.

4 LIGHTWEIGHT FUSION

We next apply a lightweight fusion transformation that combines the step and drive functions into a single function. The fused function drive_step takes an NDState and a set of explored states seen as arguments and returns a set of reachable states once it terminates. It is essentially formed by merging the functionality of step into the drive function. Since each case in function step is statically known, we can inline the pattern matching on the Option type and replace it with returning new_seen when None is matched, or by calling drive_step recursively when Some is matched.

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

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

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

With fusion complete, our AAM appears similar to a "big-step" interpreter, although it still has the machine state representation inside. The inverse transformation of this step would be splitting the work-list algorithm apart to an individual function.

5 DISENTANGLEMENT

In the disentanglement transformation, we identify the first-order data types which represent reduction contexts and lift the code blocks that dispatch these reduction contexts to be individual functions. After disentanglement, we will obtain an abstract abstract machine in defunctionalized form.

Since there are two layers of continuations, we disentangle them into three mutually recursive functions: 1) `drive_step`, which plays the same role as before; 2) `continue`, which is invoked from `drive_step` when encountering an atomic expression, and dispatches the reduction context of the analyzed language; and 3) `ndcontinue`, which is invoked from `continue` when the normal stack is empty, and dispatches the meta-continuation `NDCont`.

```
def drive_step(nds: NDState, seen: Set[State]): Set[State] = {
  ...
  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 the `drive_step`. At the end of the first case of pattern matching, we invoke `drive_step` recursively on the new `NDState`. [2]. The code for the second case is replaced entirely by a function call to `continue`. The shape of an interpreter in continuation-passing style has emerged, even though the `continue` function is not higher-order for the moment.

```
def continue(nds: NDState, seen: Set[State]): Set[State] = {
  val NDState(ae, env, bstore, konts, time, ndk) = nds; val new_time = nds.tick
  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)
  } }
```

Function `continue` is identical to what we have seen for the second case in `drive_step`, except that the dispatching logic for empty `konts` is lifted to a separate function `ndcontinue`. It matches on the list representation of context: an empty list `Nil` represents the halt of a computation path, whereas a cons `::` means the top frame of the list holds the innermost context.

```
def ndcontinue(nds: NDState, seen: Set[State]): Set[State] = {
  val NDState(ae, env, bstore, konts, time, ndk) = nds
  ndk match {
    case Nil ⇒ seen
```

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

```
    case NDCont(Nil, _, _, _)::ndk ⇒
      /* drive_step() is equivalently replaced by ndcontinue() */
      ndcontinue(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)
      val new_ndk = NDCont(cls.tail, argvs, bstore, time, frames)::ndk
      drive_step(NDState(body, new_env, new_store, frames, time, new_ndk, seen))
  } }
```

Function ndcontinue is identical to the foregoing dispatching logic of meta-continuations when konts is empty, except that the first application of drive_step is reduced to ndcontinue (we can safely do this because ae is atomic and knots is empty). When ndk is empty, the current set of explored states seen is returned, after which the analysis terminates. Otherwise, a new NDState is constructed depending on the status of the closures set.

The purpose of disentanglement is to reveal the defunctionalized form of our AAM. Functions continue and ndcontinue that dispatch different data types representing continuations are disentangled from drive_step. The inverse transformation of this step would be simply inlining ndcontinue into continue, and then inlining continue into drive_step.

6 REFUNCTIONALIZATION

Refunctionalization transforms first-order data types representing evaluation contexts to higher-order functions, and associates the dispatching logic of continuations with proper higher-order functions. After this transformation, we will obtain an abstract interpreter written in extended continuation-passing style [Danvy and Filinski 1990]. This transformation can be also regarded as expressing the control flow of the defined language through control flow of the defining language. In this section, we first present a direct refunctionalization of the continuations. Next, we adopt a caching algorithm to obtain a computable analysis [Darais et al. 2017], and further prepare the interpreter for transformation to direct-style at a later point.

6.1 First Try

First of all, as we are getting close to achieving abstract definitional interpreters, the name of drive_step is updated to aeval (for *abstract eval*). Since there will be no first-order data types representing model of stacks, we introduce a configuration class Config as a state-like definition by eliminating konts and ndk from the definition of states. Accordingly, the abstract interpreter now returns Set[Config].

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

By looking at the definition of the first-order function continue from the last section, we may find it calls the meta-continuation ndcontinue inside of its body. This observation leads us to define the types of higher-order functions representing continuations and meta-continuations:

```
type Cont  = (Config, Set[Config], MCont) ⇒ Set[Config]
type MCont = (Config, Set[Config])        ⇒ Set[Config]
```

The continuation Cont is a higher-order function which takes a configuration, a set of explored configurations, as well as a meta-continuation as its arguments, while the meta-continuation MCont only takes a configuration and a set of explored configurations as arguments. Now, the skeleton of function aeval looks as follows:

```
def aeval(config: Config, seen: Set[Config], continue: Cont, mcontinue: MCont): Set[Config] = {
  val Config(e, env, store, time) = config;  val new_time = config.tick
```

```
  val new_seen = if (seen.contains(config)) seen else seen + config
  e match {
    case Let(x, App(f, ae), e) ⇒
      val closures = atomicEval(f, env, store).toList.asInstanceOf[List[Clos]]
      val Clos(Lam(v, body), c_env) = closures.head
      val baddr = allocBind(v, new_time);      val new_env = c_env + (v ↦ baddr)
      val argvs = atomicEval(ae, env, store);  val new_store = store.update(baddr, argvs)
      val new_cont: Cont = ...
      val new_mcont: MCont = ...
      aeval(Config(body, new_env, new_store, new_time), new_seen, new_cont, new_mcont)
    case ae if isAtomic(ae) ⇒ continue(config, new_seen, mcontinue)
  }
}
```

The continuation and meta-continuation are added as arguments to `aeval`. Before jumping into the details of refunctionalizing the stack structure and constructing `new_cont` and `new_mcont`, we have two important observations in the disentangled AAM where the continuations were represented by data types: 1) When constructing the `new_ndk`, we observe that it explicitly uses `new_frames` as one argument of its top `NDCont` object. The higher-order version of these continuations follows the same pattern. 2) In the meta-continuation dispatching function `ndcontinue`, the size of the closures set is decreasing every time while the other parts of meta-continuation are kept. This means in the higher-order version, we also need an ability to update the top frame of meta-continuation and keep the rest as the same.

Let us first look into `new_cont`. The body of `new_cont` is similar to the first-order dispatching function `continue`. The invocation of `aeval` uses the latest meta-continuation `m` instead of the outer `mcontinue`. Note that the new environment `new_env` is updated based on the outer environment; the atomic evaluation uses the latest environment `env_`.

```
val new_cont: Cont = { case (config @ Config(ae, env_, store, time), seen, m) ⇒
  val new_time = config.tick
  val baddr = allocBind(x, new_time);  val new_env = env + (x ↦ baddr)
  val new_store = store.update(baddr, atomicEval(ae, env_, store))
  aeval(Config(e, new_env, new_store, new_time), seen, continue, m) }
```

To address the circularity of the meta-continuations, we define an auxiliary function `makeMCont` to construct `new_mcont`, which takes a list of closures as the argument and returns the meta-continuation. Function `makeMCont` is a recursive function where the recursive invocation happens when constructing the meta-continuation based on the rest of closures set. Now we can dynamically generate the meta-continuation inside of a meta-continuation. Meanwhile, the `new_cont` we defined above is used here, simply because of different closures actually share the same continuation. The case where the empty closures set occurs is also interesting. At this case, the outer meta-continuation `mcontinue` is returned. This becomes clear thinking inductively; our base case is such that the closures set contains only one closure. `makeMCont` is thus invoked with an empty set, and should therefore be just the current meta-continuation `mcontinue`.

```
def makeMCont(cls: List[Clos]): MCont =
  if (cls.isEmpty) mcontinue else { (config, seen) ⇒
    val Clos(Lam(v, body), c_env) = cls.head;  val baddr = allocBind(v, new_time)
    val new_env = c_env + (v ↦ baddr);  val new_store = store.update(baddr, argvs)
    aeval(Config(body, new_env, new_store, new_time), seen, new_cont, makeMCont(cls.tail)) }
val new_mcont: MCont = makeMCont(closures.tail)
```

We present one final, though very important, note regarding the initial continuation and meta-continuation. The initial continuation will be invoked when one computation path reaches its end, so we invoke the meta-continuation `m` to continue exploring other paths. The initial meta-continuation will be invoked when all computation paths reach their ends, so we terminate with the explored configurations `seen`. They correspond to the cases where `konts` is empty and `konts` and `ndk` are both empty, respectively.

```
def analyze(e: Expr): Set[Config] =
  aeval(inject(e), Set(), (c, seen, m) ⇒ m(c, seen), (c, seen) ⇒ seen)
```

We have now completely refunctionalized an abstract abstract machine; the two layers of continuations are represented by higher-order functions. The inverse transformation of this step is known as *de*functionalization, which converts the higher-order functions representing continuations to first-order data types [Danvy 2008; Danvy and Nielsen 2001].

6.2 Simplification and Caching Fixed-Points

In this section, based on the refunctionalized AAM, we further present several other transformations toward a big-step abstract definitional interpreter:

- The function `aeval` is simplified by introducing and using a nondeterministic operator `nd` which makes `aeval` only uses one layer of continuation.
- The components of the configuration are lifted as arguments of the evaluation function.
- The refunctionalized AAM collects all possible configurations, but our final target is an abstract definitional interpreter that cares about final evaluation results. As such, starting from this section, our abstract semantic artifact returns a set of final values instead of collected configurations.
- The configurations still play an important role in caching, where a cache is a mapping from configurations to sets of final values. The caching algorithm ensures termination of the analysis, as well as performs a sound over-approximation of all possible values and reachable paths.

We begin discussing these changes separately, and finally see how they work together.

6.2.1 Nondeterminism Abstraction. We observe that in the refunctionalized AAM, `makeMCont` is invoked with the tail part of the closure set, and actually repeats the code that creates the new environment and store for the body expression of a closure. Based on this observation, we present a nondeterministic operator that abstracts this pattern:

```
type Ans = Set[Config]
def nd[T](ts: Iterable[T], acc: Ans, k: (T, Ans) ⇒ Ans): Ans =
  if (ts.isEmpty) acc else nd(ts.tail, acc ++ k(ts.head, acc), k)
```

As a matter of convenience, we name the return type of the abstract interpreter to be `Ans`, which is `Set[Config]` for the moment. Given a collection of elements of type `T`, an initial accumulator of type `Ans`, and a function `k` that works on elements of type `T`, `nd` chooses an element in the collection and applies `k` to it iteratively, and accumulates the values that returned from `k` when recurring on the rest of the collection. Eventually, `nd` returns the accumulated value `acc` when there is no element in the collection. The latest accumulated value `acc` is exposed to `k` when applied.

Now we may refactor the function `aeval` to one layer of continuation by using the `nd` operator. The use of two layers of continuations will be unchained to two functions, in that each of them has one continuation.

```
type Cont = (Config, Ans) ⇒ Ans
def aeval(config: Config, seen: Ans, continue: Cont): Ans = { ...  e match {
```

```
  case Let(x, App(f, ae), e) ⇒
    val closures = atomicEval(f, env, store).toList.asInstanceOf[List[Clos]]
    nd[Storable](closures, new_seen, { case (Clos(Lam(v, body), c_env), seen) ⇒
      val baddr = allocBind(v, new_time);      val new_env = c_env + (v ↦ baddr)
      val argvs = atomicEval(ae, env, store);  val new_store = store.update(baddr, argvs)
      aeval(Config(body, new_env, new_store, new_time), seen, { case (config, seen) ⇒
        val Config(ae, env_, store, time) = config;  val new_time = config.tick
        val baddr = allocBind(x, new_time);  val new_env = env + (x ↦ baddr)
        val new_store = store.update(baddr, atomicEval(ae, env_, store))
        aeval(Config(e, new_env, new_store, new_time), seen, continue) }) })
  case ae if isAtomic(ae) ⇒ continue(config, new_seen) } }
```

The circular meta-continuation is abstracted by the `nd` operator, and only one continuation remains at the level of `aeval`. When encountering a set of possible closures, we apply `nd` on the set with the latest explored configurations and an anonymous function that works on each closure. The beginning part (before calling `aeval`) of this anonymous function is essentially abstracted from `makeMCont`, which constructs new environment and store for the closure being evaluated. Then we apply `aeval` on the body expression of the closure with a continuation, which is inlined from `new_cont` and will apply `aeval` to `e` with the current continuation of the `let` expression.

As we can see, by using the `nd` operator, the evaluation function can be simplified to a single-layer continuation-passing style. The only continuation of `aeval` plays the same role as the one in a concrete CPS interpreter.

6.2.2 Values. Up to now, the abstract interpreter still collects and returns a set of configurations. But our target, the abstract definitional interpreters, should return a set of final *values* instead of collected configurations; the intermediate values are simply discarded. 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 potentially nondeterministic, the type of the final result values is a collection `Set[VS]`.

6.2.3 Cache-Passing Style. Recall that the latent assumption of the work-list algorithm in classical AAM is that if we have seen a state `s`, it means we also have seen all 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 from 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` cache from the previous iteration as the `in` cache. The iteration terminates when the `out` cache is identical to the `in` cache.

We can now transform our abstract interpreter into cache-passing style. We begin by redefining the return type `Ans` as a case class which contains a `VS` set and a cache. `Ans` also implements its own accumulating operation `++` which defines as a union of `Set[VS]`s and a join of caches. The join operation of caches is defined as joining its two store components.

```
case class Ans(vss: Set[VS], cache: Cache) {
  def ++(ans: Ans): Ans = Ans(vss ++ ans.vss, cache.join(ans.cache)) }
```

Accordingly, the `nd` operator is slightly changed to cache-passing style. The argument of type `Ans` in the continuation `k` is replaced by an argument of type `Cache`; every time when we apply `k`, the latest cache `acc.cache` is provided.

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

As previously mentioned, the components of configurations are lifted as arguments to `aeval`. Given the cache and the simplified one-layer continuation, the shape of function `aeval` now looks as follows:

```
def aeval(e: Expr, env: Env, store: BStore, time: Time, cache: Cache, continue: Cont): Ans = {
  val config = Config(e, env, store, time);  val new_time = config.tick
  if (cache.outContains(config)) continue(Ans(cache.outGet(config), cache))
  else {
    val new_cache = cache.outUpdate(config, cache.inGet(config))
    e match { ... } } }
```

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

If, however, we have not hit the `out` cache, we retrieve the values from the `in` cache, and update the `out` cache with the 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 this configuration from the previous iteration. In the initial iteration, we first conservatively assume that all computations can diverge, and if not, we update its values in the cache after the evaluation. In terms of partial orders and lattices, it is the case that starts the Kleene iteration from the bottom of the lattice. The `out` cache of this iteration will be used as the `in` cache for the next iteration.

6.2.4 Putting Them Together. With the skeleton of cache-and-continuation-passing style `aeval` in mind, we may now use the `nd` operator to fill in the details. The first case of our match is that `e` is a `let` expression:

```
case Let(x, App(f, ae), e) ⇒
  val closures = atomicEval(f, env, store)
  nd[Storable](closures, Ans(Set[VS](), new_cache), { case (cls, clscache) ⇒
    val Clos(Lam(v, body), c_env) = cls;   val vbaddr = allocBind(v, new_time)
    val new_cenv = c_env + (v ↦ vbaddr); val new_store = store.update(vbaddr, atomicEval(ae, env, store))
    aeval(body, new_cenv, new_store, new_time, clscache, { case Ans(bdvss, bdcache) ⇒
      nd[VS](bdvss, Ans(Set[VS](), bdcache), { case (vs, bdvsscache) ⇒
        val Val(vals, time, vsstore) = vs;  val baddr = allocBind(x, time)
        val new_env = env + (x ↦ baddr);  val new_store = vsstore.update(baddr, vals)
        aeval(e, new_env, new_store, time, bdvsscache, { case Ans(evss, ecache) ⇒
```

```
      continue(Ans(evss, ecache.outUpdate(config, evss)))
    }) }) }) })
```

We invoke `nd` twice and perform a nesting, two-step, depth-first evaluation over possible values of `App(f, ae)`. The outer application of `nd` evaluates over the set of possible target closures of `f`. For each such closure, we go into its body with a new environment and a new store. Recall that the function `aeval` returns a set of `VS` objects to its continuation, so the continuation argument `bdvss` represents a set of values from multiple computation paths when evaluating this single `body` expression.

We next consider the inner application of `nd`, which evaluates over the set of the function body's values/stores `bdvss`. For each `VS`, the timestamp and the store in `VS` will be instantiated to the next application of `aeval`. The environment `new_env` is built on the outer environment variable `env`, which is the environment of this `let` expression. The inner application of `aeval` evaluates `e` with the substituted environment and store, then returns a set of values of one computation path to its continuation, where we apply the `continue` function with these values and the updated cache. Note that 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 of `aeval` or `nd`.

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

If the argument expression `e` is atomic, which is the second case of match, we simply construct a new instance of `VS` and pass it to the `continue` function.

Once the evaluation is completed at each case, we obtain a set of `VS` objects and must update the `out` cache. In the first case of pattern matching, this happens at the innermost continuation of `aeval`, where we reach the end of one computation path. For the second case, we update the `out` cache before calling the `continue` function.

```
val mtCache = Cache(Store[Config, VS](Map()), Store[Config, VS](Map()))
def analyze(e: Expr) = {
  def iter(cache: Cache): Ans = {
    val Ans(vss, anscache) = aeval(e, mtEnv, mtStore, mtTime, cache, ans ⇒ ans)
    if (anscache.out == anscache.in) Ans(vss, anscache)
    else iter(Cache(anscache.out, Store[Config, VS](Map())))
  } iter(Cache.mtCache) }
```

Finally, the `analyze` function (as the entrance of the analysis) does a looping iteration to find the least fixed-point of the cache, starting from an empty cache. If the `out` cache of this iteration is equivalent to its `in` cache, which means no further information was discovered during this iteration, then we have reached a fixed-point of cache that has over-approximated the concrete evaluations, and thus the result can be returned. Otherwise, we install the latest `out` cache to the `in` cache position when initializing the next iteration; in the meantime, an empty cache will be used as the `out` cache.

Now, we have a big-step abstract interpreter written in plain continuation-passing style which utilizes the `nd` operator and guarantees termination by caching.

6.3 Pushdown Control-Flow Analysis, Revisited

In the previous section, we 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 of the analysis that precisely models the run-time call structure 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 the k-CFA algorithm, or in using 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. Even worse, if we use the monovariant analysis (i.e., 0-CFA), the analysis result would be such that `x` and `y` point to the value set `{1, 2}`. Because the algorithm does not distinguish incoming values from different call sites for `z` (the argument of `id`). Under 1-CFA, the algorithm is able to distinguish that variable `z` has two different values at two call sites, so variable `y` would not be polluted by `1`. However, variable `x` still points to the value set `{1, 2}`, as the two call-sites still share the same continuation and 1-CFA does not increase the ability to separate the return-flows.

Call/Return Matching through Refunctionalization and Caching. The refunctionalized AAM with caching precisely matches return-flows even though we do not have a stack model. The higher-order functions representing continuations already connect all the executions in order.

In fact, we started with an unbounded-stack AAM which already precisely matches the calls and returns by explicitly using the data types to model the stack. However, in the refunctionalized AAM, the higher-order functions representing continuations of the analyzed language are blended into the call stack of our defining language (Scala) through refunctionalization, and hence the call/return flow of the analyzed language is naturally matched. This is why we say the pushdown property of the analysis is inherited from the defining language [Darais et al. 2017]. Another consequence of refunctionalization is that we have no place to explicitly save the context information in the store or on the stack.

Meanwhile, refunctionalization not only forces the nondeterministic control flows to be higher-order, but also drives us to use a different caching algorithm. Indeed, the caching algorithm plays an important role in guaranteeing the analysis will eventually terminate. 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 BACK TO DIRECT-STYLE

In the previous section, we presented a refunctionalized AAM in extended continuation-passing style, and then simplified it by utilizing the `nd` operator. To obtain a definitional abstract interpreter in direct-style, we have multiple choices of how to proceed:

- By replacing the continuations with monads [Filinski 1994], we obtain the abstract definitional interpreters in monadic style which are similar to what Darais et al. [2017] describe.

- By representing the continuation-passing style with delimited control operators such as `shift` and `reset`, we derive a new form of abstract interpreters in direct-style. This transformation is the left-inverse of the CPS transform [Danvy 1994; Danvy and Lawall 1992].
- 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-flow. Friedman and Medhekar [2003], in a tutorial on abstract interpreters, use this style to model caches and updates. In this case, nondeterminism can be easily handled via `for` comprehensions.

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

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

7.1 Back to Direct-Style with Control Operators

We first transform the function `aeval` to direct-style by removing the additional continuation argument `continue` and sequentializing the expressions in `aeval`. To enable using delimited control operators inside of `aeval`, we also add an annotation `@cps[Ans]` on the return type, which tells the compiler to CPS transform this function. Returning to direct-style in the case where `e` appears as an atomic expression is straightforward: we simply unwrap the application of `continue`. The abstract evaluator `aeval` now looks as follows:

```
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 = config.tick; val new_cache = cache.outUpdate(config, cache.inGet(config))
    e match {
      case Let(x, App(f, ae), e) ⇒ ...
      case ae if isAtomic(ae) ⇒
        val vss = Set(VS(atomicEval(ae, env, store), new_time, store))
        Ans(vss, new_cache.outUpdate(config, vss))
      } } }
```

The case where `e` is a `let` expression is less straightforward. As previously identified, `nd` works on an unknown number of nondeterministic choices, and it takes a function `k` that works for every element in the collection. Here we present another function `choices` that uses delimited control operator `shift` to capture that `k` when the program is written in direct-style and pass it to `nd`.

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

The function `choices` takes two arguments: a set of elements of type `T`, and an initial cache. The return type of `choices` is `(T, Cache)` with an annotation `@cps[Ans]`, which denotes that `choices` will iteratively return an element of type `T` from the collection along with the latest cache once `f` is invoked, but the final returned value of the function is type `Ans`. The function `f` is the delimited continuation captured by the `shift` operator. Whenever we call `choices`, the subsequent computations that will be executed after its call-site constitute the delimited continuation `f`. Once the function `f` (which is the function `k` in `nd`) is invoked and returned, it returns to its call site where

nd will be invoked on the remaining elements of the collection. Now we can use this function to sequentialize expressions in the let case:

```
case Let(x, App(f, ae), e) ⇒
  val closures = atomicEval(f, env, store)
  val (Clos(Lam(v, body), c_env), clscache) = choices[Storable](closures, new_cache)
  val vbaddr = allocBind(v, new_time); val new_cenv = c_env + (v ↦ vbaddr)
  val new_store = store.update(vbaddr, atomicEval(ae, env, store))
  val Ans(bdvss, bdcache) = aeval(body, new_cenv, new_store, new_time, clscache)
  val (VS(vals, time, vsstore), vscache) = choices[VS](bdvss, bdcache)
  val baddr = allocBind(x, time); val new_env = env + (x ↦ baddr)
  val new_vsstore = vsstore.update(baddr, vals)
  val Ans(evss, ecache) = aeval(e, new_env, new_vsstore, time, vscache)
  Ans(evss, ecache.outUpdate(config, evss))
```

For every nondeterministic fork-point in the abstract interpretation, we can simply call choices on the set of possible choices and write the program sequentially as concrete interpreters, and the subsequent statements and expressions after the call site become the delimited continuation f. We have two applications of choices, which correspond to the two applications of nd in the refunctionalized form in the previous section. choices also returns the latest cache to its left-hand side when invoked. The cache will continue to accumulate for the following calls of choices or aeval, given the fact that the abstract interpreter should always use the latest cache.

analyze is also changed by calling the reset operator around the function application of aeval to set the delimiter.

```
def analyze(e: Expr) = {
  def iter(cache: Cache): Ans = {
    val Ans(vss, anscache) = reset { aeval(e, mtEnv, mtStore, mtTime, cache) }
    if (anscache.out == anscache.in) Ans(vss, anscache)
    else iter(Cache(anscache.out, Store[Config, VS](Map())))
  } iter(Cache.mtCache) }
```

We have finally arrived at the end of this series of transformations. The inverse transformation of the last step is essentially that of performing a CPS transformation on the function aeval and explicitly using function nd instead of function choices. Then, these continuations must be manifestly embedded at the call site of nd, instead of implicitly provided by the shift operator.

To conclude, starting from a small-step abstract abstract machine, eventually we obtain a big-step abstract definitional interpreter with pushdown control-flow, written in direct-style.

8 RELATED WORK

Defunctionalization and Refunctionalization. Reynolds [1972] proposed defunctionalization as a program transformation technique that can be used to transform higher-order functions to first-order data types. While the technique is very general, it was first presented as a key step in transforming higher-order definitional interpreters into their first-order counterparts. As a form of generalized closure conversion, defunctionalization is widely used in compilation and analysis [Cejtin et al. 2000; Consel 1993; Eisenberg and Stolarek 2014; Fourtounis et al. 2014; Pottier and Gauthier 2006]. Refunctionalization as a left inverse of defunctionalization, was introduced by Danvy [2006b]; Danvy and Millikin [2009].

Ager et al. [2003] observed that defunctionalization and refunctionalization can be used to show the functional correspondence between interpreters and abstract machines. A number of

independently designed small-step semantic artifacts can be obtained by closure-converting, CPS-transforming, and then defunctionalizing their big-step counterparts, and vice versa by refunctionalizing [Danvy and Millikin 2009]. Such semantic artifacts include the SECD machine [Ager et al. 2003; Danvy 2004; Danvy and Millikin 2008b], the CEK machine [Ager et al. 2003], the CLS machine [Ager et al. 2003], Categorical Abstract Machines [Ager et al. 2003], lazy abstract machines [Ager et al. 2004], monadic evaluators [Ager et al. 2005], as well as languages with richer constructs [Biernacka and Danvy 2009; Danvy 2008, 2009], etc.

Besides, Danvy and his collaborators observed the correspondence between evaluation contexts and continuations. For deterministic languages, Danvy [2008] showed that reduction contexts are defunctionalized (data type representing) continuations and reduction semantics is a structural operational semantics in continuation-passing style. Not unexpectedly, the correspondence also exists for nondeterministic languages. For example, Danvy and Nielsen [2001] observed that refunctionalizing a regular expression matcher with two stacks yields its counterpart with two layers of continuations. In addition, the operational semantics of control operators (e.g., `shift` and `reset`) can be given by an abstract machine with multi-layer continuations [Biernacka et al. 2005; Danvy and Filinski 1990].

But as mentioned above, defunctionalization and refunctionalization are more general concepts, with a wide range of uses. For example, Danvy and Nielsen [2001] showed that defunctionalization is dual to Church encoding, and further case studies where known higher-order and first-order implementations can be related through defunctionalization and refunctionalization include backtracking algorithms (e.g., regular expression matcher), Dyck word recognizer, Dijkstra's shunting-yard algorithm, and even Quicksort [Danvy 2006b; Danvy and Millikin 2009; Danvy and Nielsen 2001].

Lightweight Fusion. Lightweight fusion as one of the transformation in our paper was presented by Ohori and Sasano [2007]. As shown by Danvy and Millikin [2008a], lightweight fusion can be used to show the equivalence between small-step and big-step abstract machines.

Abstract Interpretation and Control-Flow Analysis. Cousot and Cousot [1977] discovered abstract interpretation as a sound approach to approximate a program's run-time behavior. Control-flow analysis is one instance of abstract interpretation on higher-order functional programs that can be traced to Jones [1981]. Shivers [1988, 1991] introduced k-CFA which uses k recent calling contexts as program contours that differentiate values from different contexts. A recent 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. The AAM approach has been successfully applied to programming languages such as Java [Might et al. 2010] and Racket [Tobin-Hochstadt and Van Horn 2012].

Pushdown Control-Flow Analysis. There have been significant efforts to achieve precise call/return matching based on small-step abstract abstract machines [Earl et al. 2012; Gilray et al. 2016b; Johnson and Van Horn 2014; Vardoulakis and Shivers 2010].

CFA2 is the first solution that solves the return-flow problem [Vardoulakis and Shivers 2010], but CFA2 has several limitations: it only works on continuation-passing style programs, and does not support polyvariant analysis, in addition to having 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 2014]. 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. [2016b] further proposed a polyvariant continuation-addresses allocator for small-step AAM to achieve pushdown analysis. 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 the 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 a constant-factor increase in time complexity of the analysis if the store is widened.

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

Abstracting Definitional Interpreters. Reynolds' seminal paper *Definitional interpreters for higher-order programming languages* [Reynolds 1972, 1998] showed that in definitional interpreters, the defined language inherits properties such as the order of evaluation from the defining language, unless these properties are made explicit in the interpreter. With this insight, Darais et al. constructed abstract definitional interpreters [Darais 2017; Darais et al. 2017] which automatically inherit the pushdown control-flow property from its defining language because the defined language simply uses the call-stack model of the meta-language. Darais et al.'s abstract definitional interpreters work on direct-style λ-calculus, and are themselves written in monadic style. One of the advantages of monadic abstract interpreters is modularity. Hence, deploying different sensitivities or features can be achieved by just applying different monads. Prior to that, Sergey et al. [2013] presented a monadic abstract interpreter for small-step semantics.

This paper is greatly inspired by the work of Darais et al. [2017]. A first surface-level difference is that our presentation uses A-Normal Form λ-calculus, instead of plain λ-calculus, although our work could be easily adapted to handle plain λ-calculus as well. Going through the transformation steps from AAMs to ADIs instead of only considering the final ADI result provides additional insights. In particular, it reveals that refunctionalization plays an important role for inheriting the call structure (i.e., the model of the stack) from the meta-language. In addition, we demonstrate that the explicit use of monads in ADIs is optional, by translating to direct-style ADIs with delimited control operators in addition to the monadic version of Darais et al. [2017]. We use the delimited control operators `shift` and `reset` [Danvy and Filinski 1990] as implemented in Scala [Rompf et al. 2009]. The correspondence of delimited continuations and monads is well known [Danvy and Filinski 1990, 1992; Filinski 1994; Moggi 1991; Wadler 1992].

9 CONCLUSION

In this Functional Pearl, we bridge the gap between small-step abstract abstract machines and big-step abstract definitional interpreters by applying a series of syntactic transformations to transform the former into the latter. Among these transformations, linearization turns a work-list into an additional layer of continuations, refunctionalization converts the first-order data types representing continuations to higher-order functions, and finally, the left-inverse of the CPS transformation converts the CPS abstract interpreter into a direct-style abstract interpreter with delimited control operators, which looks almost identical to the corresponding concrete interpreter.

This sequence of transformation demonstrates that a functional correspondence exists not only between concrete semantic artifacts, but also between abstract semantic artifacts. An interesting open question remains of whether there also exist correspondences between static analyses that are formalized in different denotational and operational styles.

ACKNOWLEDGMENTS

The authors are grateful to the anonymous reviewers for their helpful comments, and they are further indebted to Olivier Danvy for numerous invaluable comments on the draft. We also thank Kimball Germane and David Sherratt for their feedback, and Thomas Gilray for the discussion on pushdown control-flow analysis.

This work was supported in part by NSF awards 1553471 and 1564207, and DOE award DE-SC0018050.

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.

Malgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. 2005. An Operational Foundation for Delimited Continuations in the CPS Hierarchy. *Logical Methods in Computer Science* Volume 1, Issue 2 (Nov. 2005). https://doi.org/10.2168/LMCS-1(2:5)2005

Malgorzata 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, Essays Dedicated to Peter D. Mosses on the Occasion of His 60th Birthday (Lecture Notes in Computer Science)*, Jens Palsberg (Ed.), Vol. 5700. Springer, 186–206. https://doi.org/10.1007/978-3-642-04164-8_10

Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. 2000. Flow-Directed Closure Conversion for Typed Languages. In *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings (Lecture Notes in Computer Science)*, Gert Smolka (Ed.), Vol. 1782. Springer, 56–71. https://doi.org/10.1007/3-540-46425-5_4

Charles Consel. 1993. A Tour of Schism: A Partial Evaluation System for Higher-order Applicative Languages. In *Proceedings of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '93)*. ACM, New York, NY, USA, 145–154. https://doi.org/10.1145/154630.154645

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. 1994. Back to Direct Style. *Sci. Comput. Program.* 22, 3 (1994), 183–195. https://doi.org/10.1016/0167-6423(94)00003-4

Olivier Danvy. 2003. A New One-Pass Transformation into Monadic Normal Form. In *CC (Lecture Notes in Computer Science)*, Vol. 2622. Springer, 77–89.

Olivier Danvy. 2004. A Rational Deconstruction of Landin's SECD Machine. In *Implementation and Application of Functional Languages, 16th International Workshop, IFL 2004, Lübeck, Germany, September 8-10, 2004, Revised Selected Papers (Lecture Notes in Computer Science)*, Clemens Grelck, Frank Huch, Greg Michaelson, and Philip W. Trinder (Eds.), Vol. 3474. Springer, 52–71. https://doi.org/10.1007/11431664_4

Olivier Danvy. 2006a. *An Analytical Approach to Programs as Data Objects*. DSc thesis. Department of Computer Science, Aarhus University, Aarhus, Denmark.

Olivier Danvy. 2006b. Refunctionalization at Work. In *Mathematics of Program Construction, 8th International Conference, MPC 2006, Kuressaare, Estonia, July 3-5, 2006, Proceedings (Lecture Notes in Computer Science)*, Tarmo Uustalu (Ed.), Vol. 4014. Springer, 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, Essays Dedicated to Peter D. Mosses on the Occasion of His 60th Birthday (Lecture Notes in Computer Science)*, Jens Palsberg (Ed.), Vol. 5700. Springer, 162–185. https://doi.org/10.1007/978-3-642-04164-8_9

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

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

Olivier Danvy and Julia L. Lawall. 1992. Back to Direct Style II: First-Class Continuations. In *LISP and Functional Programming*. 299–310. https://doi.org/10.1145/141471.141564

Olivier Danvy and Kevin Millikin. 2008a. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. *Inform. Process. Lett.* 106, 3 (2008), 100 – 109. https://doi.org/10.1016/j.ipl.2007.10.010

Olivier Danvy and Kevin Millikin. 2008b. A Rational Deconstruction of Landin's SECD Machine with the J Operator. *Logical Methods in Computer Science* 4, 4 (2008). https://doi.org/10.2168/LMCS-4(4:12)2008

Olivier Danvy and Kevin Millikin. 2009. Refunctionalization at work. *Science of Computer Programming* 74, 8 (2009), 534 – 549. https://doi.org/10.1016/j.scico.2007.10.007 Special Issue on Mathematics of Program Construction (MPC 2006).

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. 2017. *Mechanizing Abstract Interpretation.* PhD Dissertation. Department of Computer Science, University of Maryland.

David Darais, Nicholas Labich, Phúc C. Nguyen, and David Van Horn. 2017. Abstracting Definitional Interpreters (Functional Pearl). *Proc. ACM Program. Lang.* 1, ICFP, Article 12 (Aug. 2017), 25 pages. https://doi.org/10.1145/3110256

Christopher Earl, Matthew Might, and David Van Horn. 2010. Pushdown Control-Flow Analysis of Higher-Order Programs. *CoRR* abs/1007.4268 (2010). arXiv:1007.4268 http://arxiv.org/abs/1007.4268

Christopher Earl, Ilya Sergey, Matthew Might, and David Van Horn. 2012. Introspective Pushdown Analysis of Higher-order Programs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. ACM, New York, NY, USA, 177–188. https://doi.org/10.1145/2364527.2364576

Richard A. Eisenberg and Jan Stolarek. 2014. Promoting Functions to Type Families in Haskell. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell (Haskell '14)*. ACM, New York, NY, USA, 95–106. https://doi.org/10.1145/2633357.2633361

Matthias 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.

Andrzej Filinski. 1994. Representing Monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*. ACM, New York, NY, USA, 446–457. https://doi.org/10.1145/174675.178047

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

Georgios Fourtounis, Nikolaos S Papaspyrou, and Panagiotis Theofilopoulos. 2014. Modular polymorphic defunctionalization. *Computer Science and Information Systems* 11, 4 (2014), 1417–1434.

Daniel P. Friedman and Anurag Medhekar. 2003. Tutorial: Using an Abstracted Interpreter to Understand Abstract Interpretation, Course notes for CSCI B621, Indiana University.

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 *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 691–704. https://doi.org/10.1145/2837614.2837631

James Ian Johnson and David Van Horn. 2014. Abstracting Abstract Control. In *Proceedings of the 10th ACM Symposium on Dynamic Languages (DLS '14)*. ACM, New York, NY, USA, 11–22. https://doi.org/10.1145/2661088.2661098

Neil D. Jones. 1981. Flow Analysis of Lambda Expressions (Preliminary Version). In *Automata, Languages and Programming, 8th Colloquium, Acre (Akko), Israel, July 13-17, 1981, Proceedings (Lecture Notes in Computer Science)*, Shimon Even and Oded Kariv (Eds.), Vol. 115. Springer, 114–128. https://doi.org/10.1007/3-540-10843-2_10

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

John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM* 3, 4 (April 1960), 184–195. https://doi.org/10.1145/367177.367199

Matthew Might, Yannis Smaragdakis, and David Van Horn. 2010. Resolving and Exploiting the k-CFA Paradox: Illuminating Functional vs. Object-oriented Program Analysis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 305–315. https://doi.org/10.1145/1806596.1806631

Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55 – 92. https://doi.org/10.1016/0890-5401(91)90052-4 Selections from 1989 IEEE Symposium on Logic in Computer Science.

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

François Pottier and Nadji Gauthier. 2006. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation* 19, 1 (01 Mar 2006), 125–162. https://doi.org/10.1007/s10990-006-8611-7

John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of 25th ACM National Conference*. ACM, Boston, Massachusetts, 717–740. Reprinted in Higher-Order and Symbolic Computation 11(4):363-397, 1998, with a foreword [Reynolds 1998].

John C. Reynolds. 1998. Definitional Interpreters Revisited. *Higher-Order and Symbolic Computation* 11, 4 (1998), 355–361.

Tiark Rompf, Ingo Maier, and Martin Odersky. 2009. Implementing First-class Polymorphic Delimited Continuations by a Type-directed Selective CPS-transform. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA, 317–328. https://doi.org/10.1145/1596550.1596596

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

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

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

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 *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. ACM, New York, NY, USA, 51–62. https://doi.org/10.1145/1863543.1863553

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 *Proceedings of the 19th European Conference on Programming Languages and Systems (ESOP'10)*. Springer-Verlag, Berlin, Heidelberg, 570–589. https://doi.org/10.1007/978-3-642-11957-6_30

Philip Wadler. 1985. How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Functional Programming Languages and Computer Architecture*, Jean-Pierre Jouannaud (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 113–128.

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