Refunctionalization of Abstract Abstract Machines

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

Guannan Wei, James Decker, and Tiark Rompf Purdue University

Abstract Machines (CEK/CESK) ⇄ (refunctionalization / defunctionalization) CPS Interpreters ⇄ (back to direct-style / CPS transformation) Definitional Interpreters

Known as functional correspondence between *concrete* abstract machines and evaluators.

Defunctionalization transforms higher-order functions to first-order data type representations and their dispatching functions (Reynolds, 1972). Closure conversion is one example of defunctionalization. Refunctionalization is its left-inverse, transforming first-order data types back to higher-order functions. Refunctionalization and defunctionalization can be used to construct a *functional correspondence* between abstract machines and evaluators (Ager et al., 2003). This correspondence shows that abstract machines and evaluators can be inter-derived in a systematic way after identifying their first-order/higher-order representations of contexts/continuations.

Is there a functional correspondence between the ***abstract*** semantic artifacts (i.e., abstract interpreters)?

Abstract Abstract Machines [ICFP 10] ⟷ ? ⟷ Abstract Definitional Interpreters [ICFP 17]

0. Pushdown AAM

Abstracting Abstract Machine (AAM) is a methodology to derive sound abstract interpreters from concrete interpreters. For example, we can construct an abstract interpreter for a call-by-value lambda calculus by systematically applying a combination of abstractions (e.g., finite address space, store-allocated continuations, etc.) to a concrete CESK machine.

In this pearl, we start from a variant of AAM, the pushdown AAM, which uses an unbounded stack, and show the transformation to ADI.

Pushdown AAM

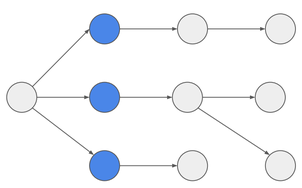
- Environment maps variables to addresses:
  `type Env = Map[String, Addr]`
- Store maps (finite) addresses to sets of abstract values:
  `type Store = Map[Addr, Set[Clos]]`
- Continuation keeps unbounded (same as a concrete CESK machine):
  `case class Frame(x: String, e: Expr, ρ: Env)`
- State has four components:
  `case class State(e: Expr, ρ: Env, σ: Store, κ: List[Frame])`
- State transition function and collecting function:
  `step  : State ⇒ Set[State]`
  `drive : List[State] × Set[State] ⇒ Set[State]`

Why Pushdown AAM?

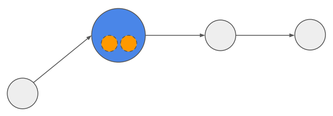
- Naturally corresponds to abstract definitional interpreters (Darais et al., 2017), which inherents the stack structure from the defining language.

1. Linearization

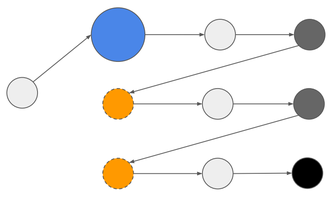
The linearization transforms the nondeterminism into another *meta*-continuation component of the state, and makes the state transition deterministic.



The classical AAM has a nondeterministic state transition -- one state may have multiple successors; these will be added to an additional worklist. A `drive` function controls the exploration of states by repeatedly popping up a state and getting its successors.



After linearization, the state transition becomes deterministic. At a fork point, we pick up one state as the successor, and save enough information at this fork point so that we can come back later and construct the remaining states.



When we reach an end of one computation path, there may still be remaining states at some fork point. By resuming to the most recent fork point and constructing a new successor, the reachable states will be explored like traversing a tree in depth-first order.

Before: `case class State(e: Expr, ρ: Env, σ: Store, κ: List[Frame])`
After: `case class State(e: Expr, ρ: Env, σ: Store, κ: List[Frame], mκ: List[NDCont])`
`case class NDCont(cls: List[Clos], argvs: Set[Clos], σ: Store, κ: List[Frame])`

2. Lightweight Fusion

Lightweight fusion combines the `step` and `drive` functions.

Before: `step  : State ⇒ Option[State]`
`drive : State × Set[State] ⇒ Set[State]`
After: `drive_step : State × Set[State] ⇒ Set[State]`

The fused function `drive_step` does both state-transition and state-collection, which looks like a "big-step" abstract interpreter, but still uses a first-order representation of machine states.

3. Disentanglement

- Identify first-order data types that represent contexts.
  `case class State(e: Expr, ρ: Env, σ: Store, κ: List[Frame], mκ: List[NDCont])`
- Identify code blocks that handle different cases of these data types.

```
def drive_step(nds: State, seen: Set[State]): Set[State] = { ...
  nds match {
    case State(Let(x, App(f, ae), e), ρ, σ, κ, mκ) if isAtomic(f) && isAtomic(ae) ⇒ ...
    case State(ae, ρ, σ, κ, mκ) if isAtomic(ae) ⇒
      κ match {                                          dispatching κ
        case Nil ⇒
          mκ match {                                     dispatching mκ
            case Nil ⇒ new_seen
            case NDCont(Nil,_,_,_)::mκ ⇒ ...
            case NDCont(cls, argv, σ, κ)::mκ ⇒ ...
          }
        case Frame(x, e, f_ρ)::κ ⇒ ... } } }
```

- Lift these code blocks to *top-level* individual functions.
  `continue  : State × Set[State] ⇒ Set[State]`
  `mcontinue : State × Set[State] ⇒ Set[State]`
  `drive_step` calls `continue` when an atomic expression needs to be returned; `continue` calls `mcontinue` when the object program's stack is empty; `mcontinue` halts when the nondeterministic stack is empty.

4. Refunctionalization

- Transforms first-order data types and their dispatching functions to higher-order functions, i.e., to CPS form.
- Types of the higher-order continuations and refunctionalized `aeval` function:

```
type Cont  = (State, Set[State], MCont) ⇒ Set[State]
type MCont = (State, Set[State]) ⇒ Set[State]
aeval : State × Set[State] × Cont × MCont ⇒ Set[State]
```

- After refunctionalization, an abstract interpreter written with two HO continuations:

```
def aeval(state: State, seen: Set[State], k: Cont, mk: MCont): Set[State] = {
  e match {
   case Let(x, App(f, ae), e) if isAtomic(f) && isAtomic(ae) ⇒
     val closures = atomicEval(f, ρ, σ).toList
     val Clos(Lam(v, body), c_ρ) = closures.head
     val α = alloc(v);                    val new_ρ = c_ρ + (v ↦ α)
     val argvs = atomicEval(ae, ρ, σ);    val new_σ = σ.join(α ↦ argvs)
     val new_k: Cont = ...   // A HO function takes result of body and then evaluates e
     val new_mk: MCont = ... // A HO function iterates over the target closures
     aeval(State(body, new_ρ, new_σ), new_seen, new_k, new_mk)
   case ae if isAtomic(ae) ⇒ k(state, new_seen, mk) } }
```

5. Back to Direct-Style

From extended CPS to direct-style, three choices:

- Use explicit side-effects and assignments.
- Use monads (Darais et al., 2017)
- ***Use delimited control operators (shift/reset).***

```
def aeval(state: State, seen: Set[State]): (State, Set[State]) @cps[Set[State]] = { ...
  e match {
    case Let(x, App(f, ae), e) if isAtomic(f) && isAtomic(ae) ⇒
      val closures = atomicEval(f, ρ, σ).toList
      val (Clos(Lam(v, body), c_ρ), c_seen) = choices(closures, new_seen)
      val v_α = alloc(v);  val new_ρ = c_ρ + (v ↦ v_α)
      val new_σ = σ.join(v_α ↦ atomicEval(ae, ρ, σ))
      val (bd_state, bd_seen) = aeval(State(body, new_ρ, new_σ), c_seen)
      val State(bd_ae, bd_ρ, bd_σ) = bd_state
      val x_α = alloc(x);  val new_ρ_* = ρ + (x ↦ x_α)
      val new_σ_* = bd_σ.join(x_α ↦ atomicEval(bd_ae, bd_ρ, bd_σ))
      aeval(State(e, new_ρ_*, new_σ_*), bd_seen)
    case ae if isAtomic(ae) ⇒ (state, new_seen)
  }
}
```

delimited continuation captured into choices.

- `choices` returns a closure nondeterministically, and captures the reset computation by internally using the `shift` operator.

What else?

- Co-inductive caching (Darias et al., 2017) to ensure termination.
- Polyvariant analysis by adding timestamp to addresses.
- Return a set of values instead of states, lift the fields of `State` to `aeval`.

Take-Home Message

- A constructive functional correspondence fills the gap between AAM and ADI.
- Linearization twists the worklist to a meta-continuation, then apply existing techniques (e.g., l.w. fusion, disentanglement, refunc., and delimited cont.) to the two-continuation-passing style abstract interpreter.

Abstract Abstract Machines [ICFP 10] → Linearization → Lightweight Fusion → Disentanglement → Refunctionalization → Back to Direct-Style → Abstract Definitional Interpreters [ICFP 17]