# Refunctionalization of Abstract Abstract Machines

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

**Guannan Wei**, James Decker, Tiark Rompf
Department of Computer Science, Purdue University

# Definitional Interpreters for Higher-Order Programming Languages*

JOHN C.REYNOLDS**
*Systems and Information Science, Syracuse University*

| Order-of-application dependence: | Use of higher-order functions: yes | no |
|---|---|---|
| yes | direct interpreter for GEDANKEN | McCarthy's definition of LISP |
| no | Morris-Wadsworth method | SECD machine, Vienna definition |



John Reynolds, Definitional Interpreters for Higher-Order Programming Languages,
in Proceedings of the ACM Annual Conference, Volume 2, pages 717–740, August 1972.

# Definitional Interpreters for Higher-Order Programming Languages*

JOHN C.REYNOLDS**
*Systems and Information Science, Syracuse University*

| Order-of-application dependence: | Use of higher-order functions: yes | no |
|---|---|---|
| yes | direct interpreter for GEDANKEN | McCarthy's definition of LISP |
| no | Morris-Wadsworth method | SECD machine, Vienna definition |



John Reynolds, Definitional Interpreters for Higher-Order Programming Languages, in Proceedings of the ACM Annual Conference, Volume 2, pages 717–740, August 1972.

# Definitional Interpreters for Higher-Order Programming Languages*

JOHN C.REYNOLDS**
*Systems and Information Science, Syracuse University*

| Order-of-application dependence: | Use of higher-order functions: yes | no |
|---|---|---|
| yes | direct interpreter for GEDANKEN | McCarthy's definition of LISP |
| no | Morris-Wadsworth method | SECD machine, Vienna definition |



John Reynolds, Definitional Interpreters for Higher-Order Programming Languages,
in Proceedings of the ACM Annual Conference, Volume 2, pages 717–740, August 1972.

*de*functionalization: transform higher-order functions to first-order data types with their dispatching functions (e.g., closure conversion).

| Order-of-application dependence: | Use of higher-order functions: yes | no |
|---|---|---|
| yes | direct interpreter for GEDANKEN | McCarthy's definition of LISP |
| no | Morris-Wadsworth method | SECD machine, Vienna definition |



John Reynolds, Definitional Interpreters for Higher-Order Programming Languages, in Proceedings of the ACM Annual Conference, Volume 2, pages 717–740, August 1972.

*de*functionalization: transform higher-order functions to first-order data types with their dispatching functions (e.g., closure conversion).

*re*functionalization: the left-inverse of defunctionalization [Danvy et al.].

| Order-of-application dependence: | Use of higher-order functions: yes | no |
|---|---|---|
| yes | direct interpreter for GEDANKEN | McCarthy's definition of LISP |
| no | Morris-Wadsworth method | SECD machine, Vienna definition |



John Reynolds, Definitional Interpreters for Higher-Order Programming Languages, in Proceedings of the ACM Annual Conference, Volume 2, pages 717–740, August 1972.

# Functional Correspondence

- Refunctionalization / defunctionalization can be used to show the functional correspondence between small-step abstract machines and big-step evaluators.
- Idea: apply refunctionalization / defunctionalization to *control flow*.



Ager, Mads Sig, et al. A functional correspondence between evaluators and abstract machines,
Proc. of the 5th ACM SIGPLAN inter. conf. on Principles and practice of declarative programming. 2003.

# Functional Correspondence

- Example: refunctionalizing a CEK machine yields an interpreter in continuation-passing style.

```
         CEK Machine
State := ⟨Expr, Env, Kont⟩
Kont  := Halt
       | Ar⟨Expr, Env, Kont⟩
       | Fn⟨Lam, Env, Kont⟩
```

# Functional Correspondence

- Example: refunctionalizing a CEK machine yields an interpreter in continuation-passing style.

```
           CEK Machine
State := ⟨Expr, Env, Kont⟩
Kont  := Halt
       | Ar⟨Expr, Env, Kont⟩
       | Fn⟨Lam, Env, Kont⟩
```

evaluation context
(first-order data types)

# Functional Correspondence

- Example: refunctionalizing a CEK machine yields an interpreter in continuation-passing style.

CEK Machine

State := ⟨Expr, Env, Kont⟩

Kont := Halt
| Ar⟨Expr, Env, Kont⟩
| Fn⟨Lam, Env, Kont⟩

refunc. →

← defunc.

CPS Interpreter

eval : Expr × Env × (Val → Val) → Value

evaluation context
(first-order data types)

continuation
(higher-order functions)

# Functional Correspondence

- Example: refunctionalizing a CEK machine yields an interpreter in continuation-passing style.

**CEK Machine**

```
State := ⟨Expr, Env, Kont⟩
Kont  := Halt
       | Ar⟨Expr, Env, Kont⟩
       | Fn⟨Lam, Env, Kont⟩
```

refunc. →

← defunc.

**CPS Interpreter**

```
eval : Expr × Env × (Val → Val)
     → Value
```

- *refunc.* evaluation contexts = *higher-order* continuations
- *defunc.* continuations = *first-order* evaluation contexts



Ager, Mads Sig, et al. A functional correspondence between evaluators and abstract machines,
Proc. of the 5th ACM SIGPLAN inter. conf. on Principles and practice of declarative programming. 2003.

# Functional Correspondence

- Example: refunctionalizing a CEK machine yields an interpreter in continuation-passing style.

- Transform CPS interpreter back to direct-style, i.e., a definitional evaluator.

$$
\begin{array}{ccccc}
\text{Abstract Machines (CEK/CESK)} & \underset{\text{defunc.}}{\overset{\text{refunc.}}{\rightleftarrows}} & \text{CPS Interpreters} & \underset{\text{CPS trans.}}{\overset{\text{to direct-style}}{\rightleftarrows}} & \text{Definitional Interpreters}
\end{array}
$$



Ager, Mads Sig, et al. A functional correspondence between evaluators and abstract machines,
Proc. of the 5th ACM SIGPLAN inter. conf. on Principles and practice of declarative programming. 2003.

# Functional Correspondence

- Functional correspondence: independently designed concrete semantic artifacts can be inter-derived in a systematic way.

- Refunctionalization and defunctionalization plays an important role in the inter-derivation.

Abstract Machines (CEK/CESK) ⟷ Definitional Interpreters

Functional correspondence between
*concrete* abstract machines and evaluators

Ager, Mads Sig, et al. A functional correspondence between evaluators and abstract machines, Proc. of the 5th ACM SIGPLAN inter. conf. on Principles and practice of declarative programming. 2003.

Abstract Abstract Machines
[ICFP 10]

A recipe to derive small-step abstract interpreters from concrete interpreters.

Abstract Machines (CEK/CESK)

Definitional Interpreters

Functional correspondence between *concrete* abstract machines and evaluators

## Abstract Abstract Machines [ICFP 10]

A recipe to derive small-step abstract interpreters from concrete interpreters.

- finite state space
  $\text{State}^\# := \langle \text{Expr}, \text{Env}^\#, \text{Store}^\#, \text{Kont}^\# \rangle$
- nondeterministic state transition
  $\text{State}^\# \rightarrow \text{Set}[\text{State}^\#]$

## Abstract Machines (CEK/CESK)

## Definitional Interpreters

Functional correspondence between *concrete* abstract machines and evaluators

Abstract
Abstract
Machines
[ICFP 10]

A big-step, compositional,
monadic abstract interpreter.

Abstract
Definitional
Interpreters
[ICFP 17]

Abstract
Machines
(CEK/CESK)

Functional correspondence between
*concrete* abstract machines and evaluators

Definitional
Interpreters

Abstract
Abstract
Machines
[ICFP 10]

Abstract
Definitional
Interpreters
[ICFP 17]

Abstract
Machines
(CEK/CESK)

Functional correspondence between
*concrete* abstract machines and evaluators

Definitional
Interpreters

Abstract Abstract Machines
[ICFP 10]

?

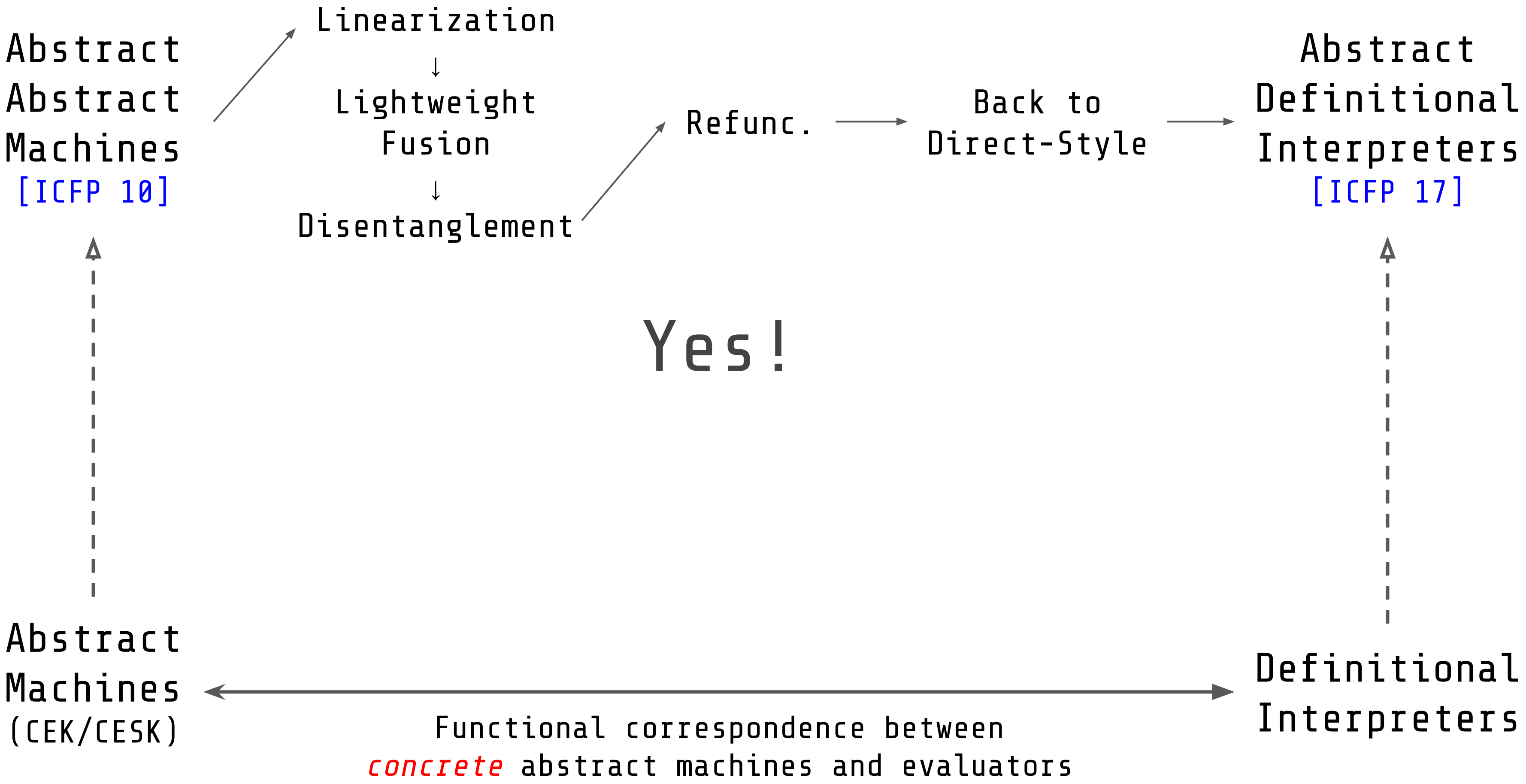
Abstract Definitional Interpreters
[ICFP 17]

Is there a functional correspondence between the *abstract* semantic artifacts?

Abstract Machines
(CEK/CESK)

Definitional Interpreters

Functional correspondence between *concrete* abstract machines and evaluators


Abstract Abstract Machines
[ICFP 10]
Linearization
↓
Lightweight Fusion
↓
Disentanglement
Refunc.
Back to Direct-Style
Abstract Definitional Interpreters
[ICFP 17]
Yes!
Abstract Machines (CEK/CESK)
Definitional Interpreters
Functional correspondence between
*concrete* abstract machines and evaluators

Abstract Abstract Machines [ICFP 10]

Linearization

↓

Lightweight Fusion

↓

Disentanglement

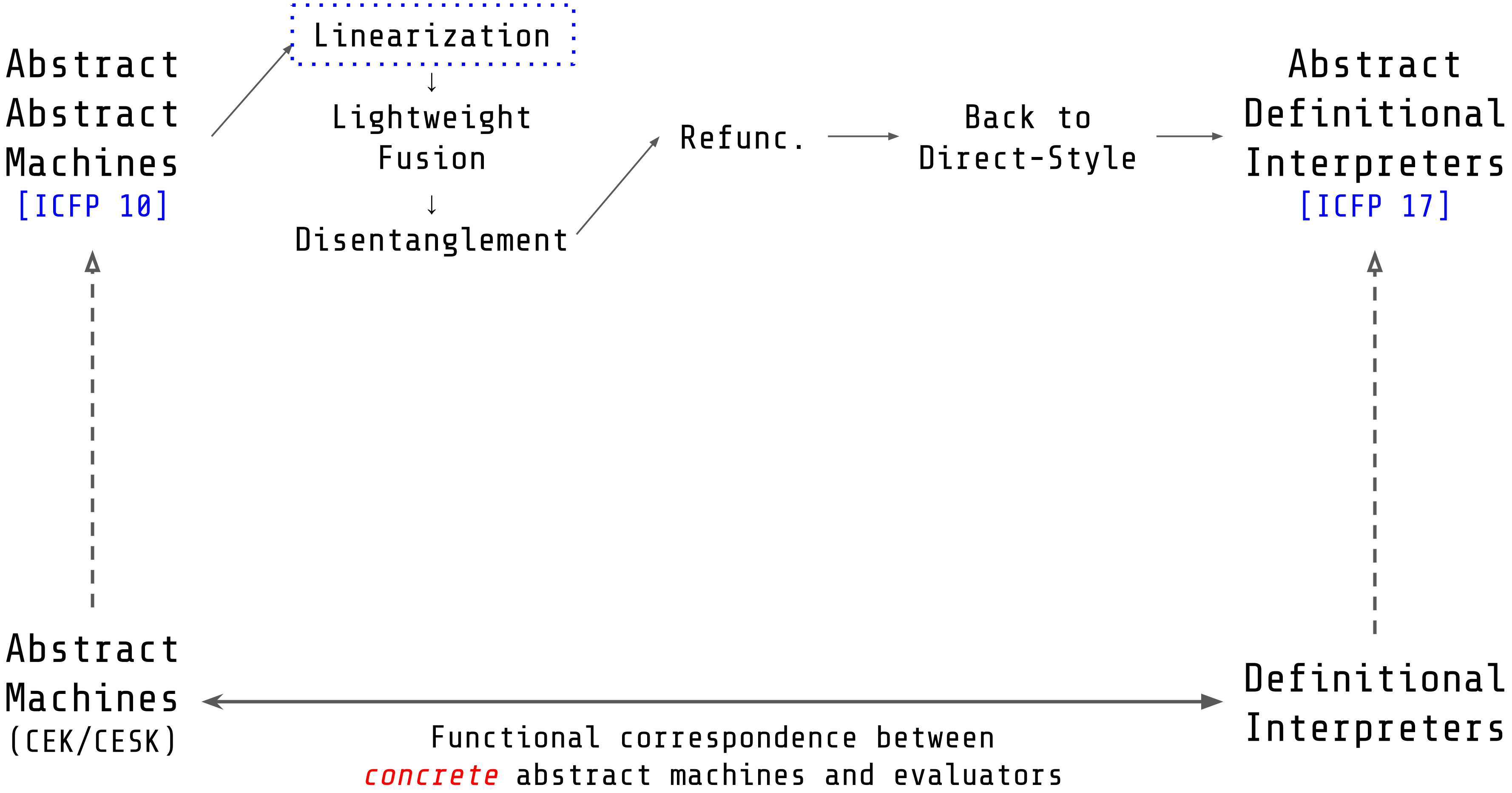
Refunc. → Back to Direct-Style → Abstract Definitional Interpreters [ICFP 17]

# Yes!

- A constructive answer from pushdown AAM to ADI.
- Refunctionalized AAM with two continuations.
- Back to direct-style using delimited control operators.

Abstract Machines (CEK/CESK) ⟷ Definitional Interpreters

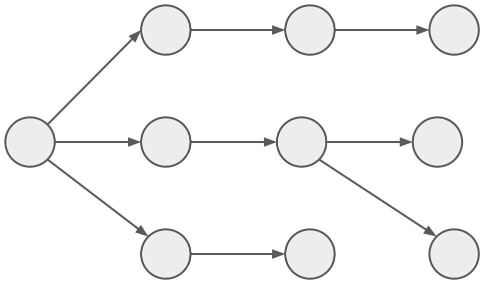
Functional correspondence between *concrete* abstract machines and evaluators


Abstract
Abstract
Machines
[ICFP 10]
Linearization
↓
Lightweight
Fusion
↓
Disentanglement
Refunc.
Back to
Direct-Style
Abstract
Definitional
Interpreters
[ICFP 17]
Abstract
Machines
(CEK/CESK)
Definitional
Interpreters
Functional correspondence between
*concrete* abstract machines and evaluators

# Linearization

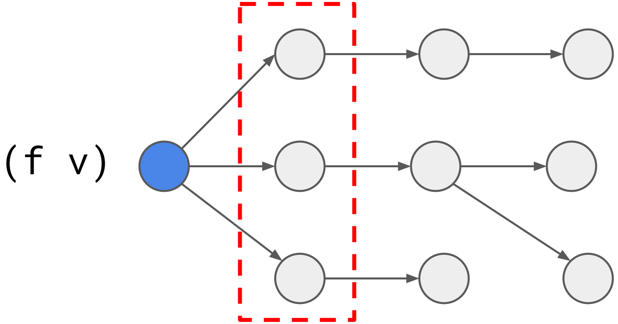
*Concrete* abstract machine (CEK) is deterministic...

# Linearization



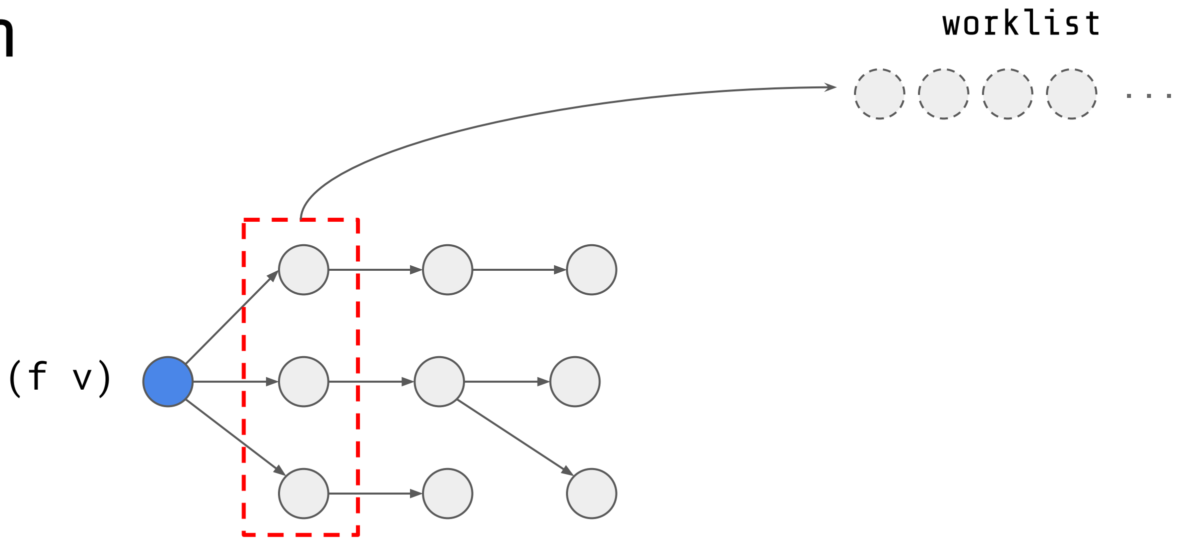
*Abstract* abstract machine (AAM) is *non*deterministic...

# Linearization



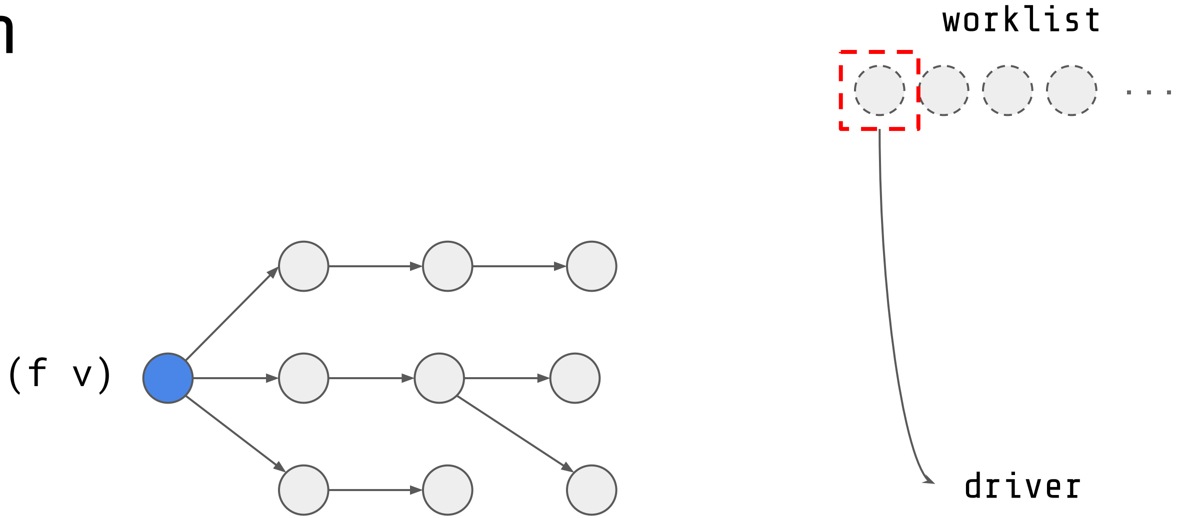

f may represent multiple target closures.

# Linearization



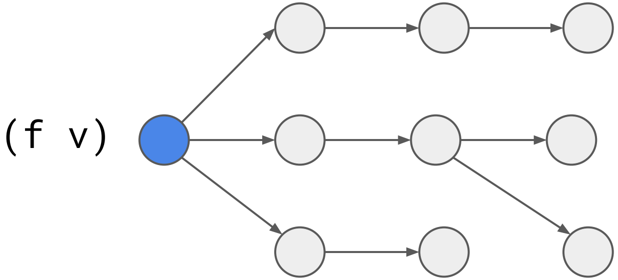

AAM adds the successors into a worklist.

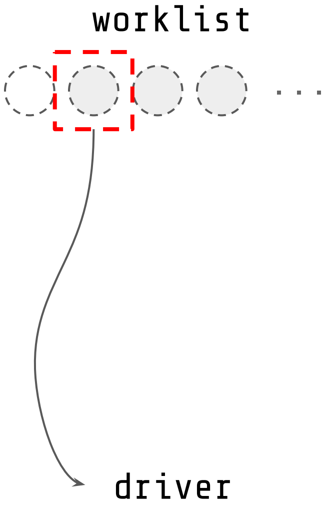
# Linearization




A driver function keeps popping up a state from the worklist, and asking "*Have I see you before?*", if not, "*Do you have successors?*".

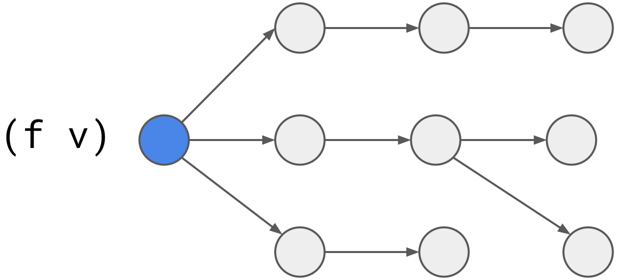
# Linearization

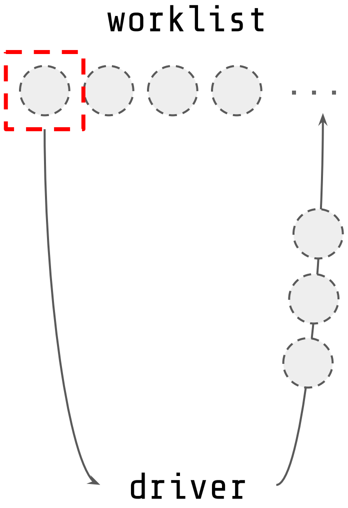



A driver function keeps popping up a state from the worklist, and asking "*Have I see you before?*", if not, "*Do you have successors?*".

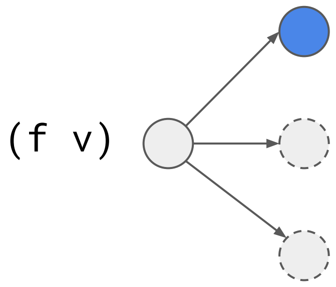
# Linearization



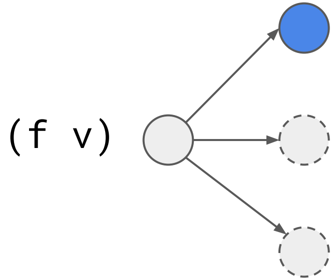

A driver function keeps popping up a state from the worklist, and asking "*Have I see you before?*", if not, "*Do you have successors?*".

# Linearization



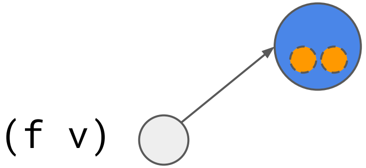

Linearization makes the state transition to be deterministic by using another *meta*-continuation to express the choices.

# Linearization



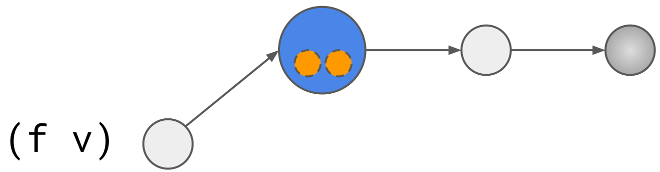

Pick a state as *the* successor state.

# Linearization



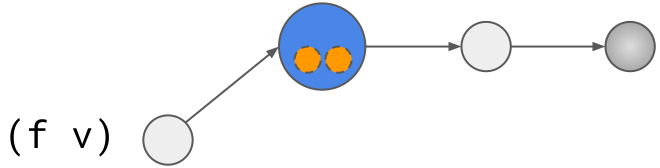

Save the information at the fork point into that meta-continuation of the state, so that we can come back later.

# Linearization



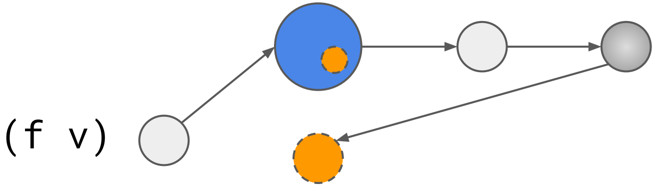

Continue working on this state, until we reach its end.

# Linearization



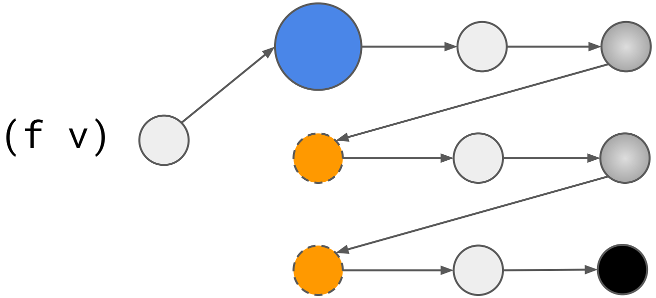

Remember we still have states left...

# Linearization




Resume to the most recent fork point,
and construct the next state.

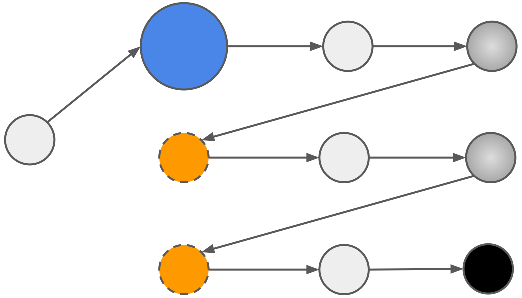
# Linearization



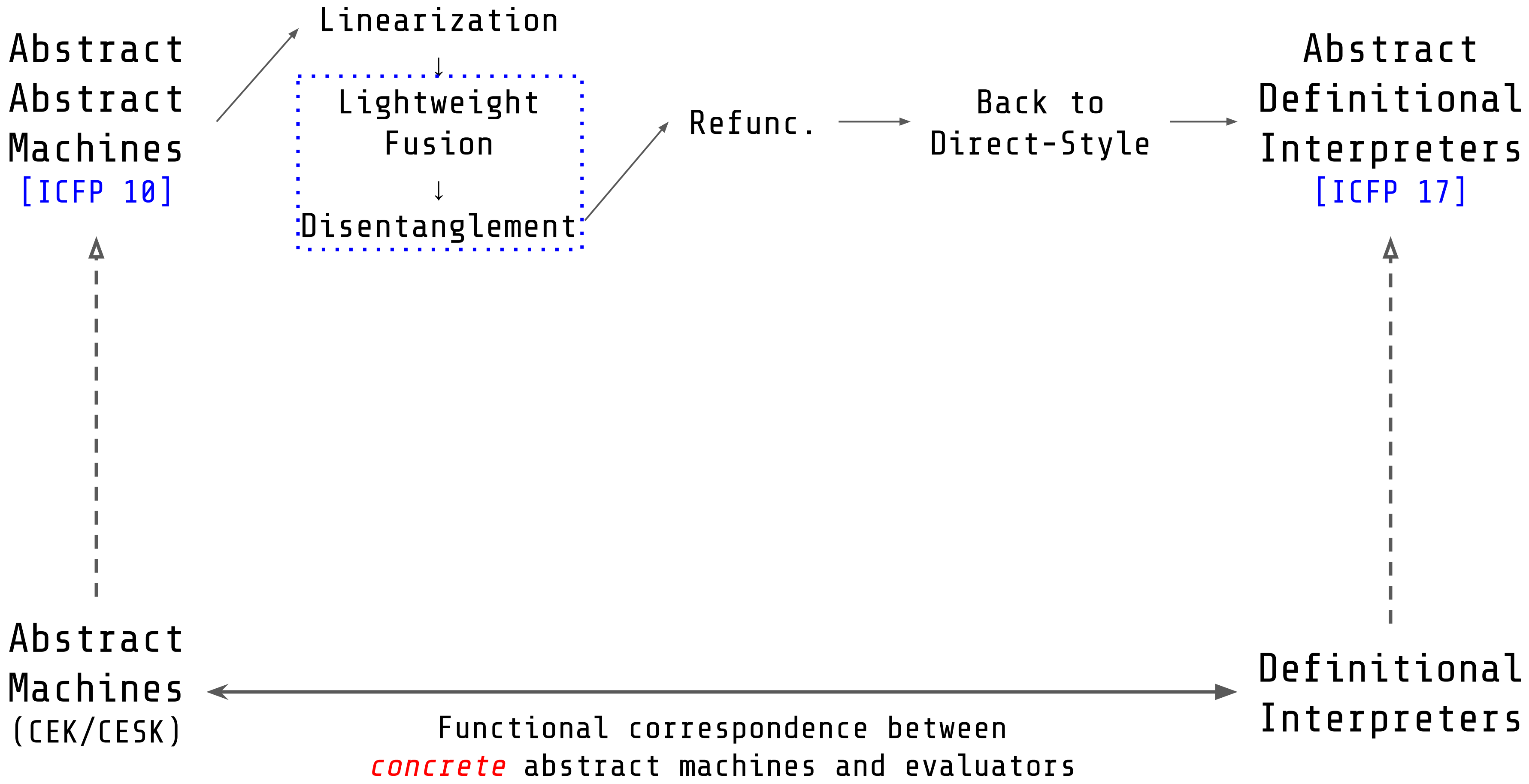

A driver function just keeps asking
"Do you have a successor?"...
Until no more states and no more saved
choices in all meta-continuations.

# Linearization

(f v)



- Now the abstract state has *two* continuations, both are represented by first-order types.
- Change the state definition
  from **⟨Expr, Env$^\#$, Store$^\#$, Kont⟩**
  to **⟨Expr, Env$^\#$, Store$^\#$, Kont, MKont⟩**


Abstract Abstract Machines
[ICFP 10]
Linearization
Lightweight Fusion
Disentanglement
Refunc.
Back to Direct-Style
Abstract Definitional Interpreters
[ICFP 17]
Abstract Machines
(CEK/CESK)
Definitional Interpreters
Functional correspondence between
concrete abstract machines and evaluators

# Fusion and Disentanglement

- Lightweight fusion and disentanglement further tweak the form of AAM and expose continuations explicitly.

# Fusion and Disentanglement

- Lightweight fusion and disentanglement further tweak the form of AAM and expose continuations explicitly.
- Fusion: merges the `step` function and the `drive` function into one, so the abstract interpreter is a single, recursive function.

# Fusion and Disentanglement

- Lightweight fusion and disentanglement further tweak the form of AAM and expose continuations explicitly.
- Fusion: merges the `step` function and the `drive` function into one, so the abstract interpreter is a single, recursive function.
- Disentanglement: lifts the code that dispatches those two data types representing continuations to be top-level functions.

```
aeval     : State × Cache ⇒ Cache
continue  : State × Cache ⇒ Cache
mcontinue : State × Cache ⇒ Cache
```

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

We have obtained the *de*functionalized form of AAM.

Abstract Machines (CEK/CESK) ↔ Definitional Interpreters

Functional correspondence between *concrete* abstract machines and evaluators

Abstract Abstract Machines
[ICFP 10]

Linearization
↓
Lightweight Fusion
↓
Disentanglement

Refunc.

Back to Direct-Style

Abstract Definitional Interpreters
[ICFP 17]

We have obtained the *de*functionalized form of AAM.

Continuations and their dispatching functions are exposed explicitly.

Abstract Machines
(CEK/CESK)

Definitional Interpreters

Functional correspondence between
*concrete* abstract machines and evaluators

# Refunctionalization

- Transforms the two first-order data type representing continuations and their dispatching functions to two higher-order functions.

- After which, the abstract interpreter is written in two-continuation-passing style.

# Refunctionalization

- Types of the first-order dispatching functions:

$$\text{State} : \langle \text{Expr}, \text{Env}^{\#}, \text{Store}^{\#}, \text{Kont}, \text{MKont} \rangle$$

$$\text{continue} : \text{State} \times \text{Cache} \Rightarrow \text{Cache}$$

$$\text{mcontinue} : \text{State} \times \text{Cache} \Rightarrow \text{Cache}$$

- Types of the higher-order continuations:

$$\text{State} : \langle \text{Expr}, \text{Env}^{\#}, \text{Store}^{\#}, \cancel{\text{Kont}, \text{MKont}} \rangle$$

```
type Cont  = (State, Cache, MCont) ⇒ Cache
type MCont = (State, Cache) ⇒ Cache
```

# Refunctionalization

- Types of the first-order dispatching functions:

$$\text{State} : \langle \text{Expr}, \text{Env}^{\#}, \text{Store}^{\#}, \text{Kont}, \text{MKont} \rangle$$

```
continue  : State × Cache ⇒ Cache
mcontinue : State × Cache ⇒ Cache
```

- Types of the higher-order continuations:

$$\text{State} : \langle \text{Expr}, \text{Env}^{\#}, \text{Store}^{\#}, \cancel{\text{Kont}, \text{MKont}} \rangle$$

```
type Cont  = (State, Cache, MCont) ⇒ Cache
type MCont = (State, Cache) ⇒ Cache
```

# Refunctionalization

```
def aeval(state: State, seen: Cache, k: Cont, mk: MCont): Cache = {
  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 App 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)
  }
}
```

# Refunctionalization

```
def aeval(state: State, seen: Cache, k: Cont, mk: MCont): Cache = {
  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 App 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)
  }
}
```

Abstract Abstract Machines [ICFP 10]

Linearization
↓
Lightweight Fusion
↓
Disentanglement

Refunc.

Back to Direct-Style

Abstract Definitional Interpreters [ICFP 17]

We have obtained a *re*functionalized AAM in CPS.

Abstract Machines (CEK/CESK)

Definitional Interpreters

Functional correspondence between *concrete* abstract machines and evaluators

# Back to Direct-Style

- From extended CPS to direct-style, three choices:
  - Use explicit side-effects and assignments.
  - Use monads [Darais et al. ICFP 17].
  - Use delimited control operators (shift/reset).

# Back to Direct-Style

- From extended CPS to direct-style, three choices:
  - Use explicit side-effects and assignments.
  - Use monads [Darais et al. ICFP 17].
  - *Use delimited control operators (shift/reset).*
    - shift to capture the continuation
    - reset to set the boundary

# Back to Direct-Style

- From extended CPS to direct-style, three choices:
  - Use explicit side-effects and assignments.
  - Use monads [Darais et al. ICFP 17].
  - *Use delimited control operators (shift/reset).*
    - shift to capture the continuation
    - reset to set the boundary
- After the transformation, the abstract interpreter looks almost no difference to a concrete interpreter.

# Back to Direct-Style

```
def aeval(state: State, seen: Cache): (State, Cache) @cps[Cache] = {
  ...
  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)
  }
}
```

# Back to Direct-Style

```
def aeval(state: State, seen: Cache): (State, Cache) @cps[Cache] = {
  ...
  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)
  }
}
```

Get a closure of f, nondeterministically.

# Back to Direct-Style

```
def aeval(state: State, seen: Cache): (State, Cache) @cps[Cache] = {
  ...
  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)
  }
}
```

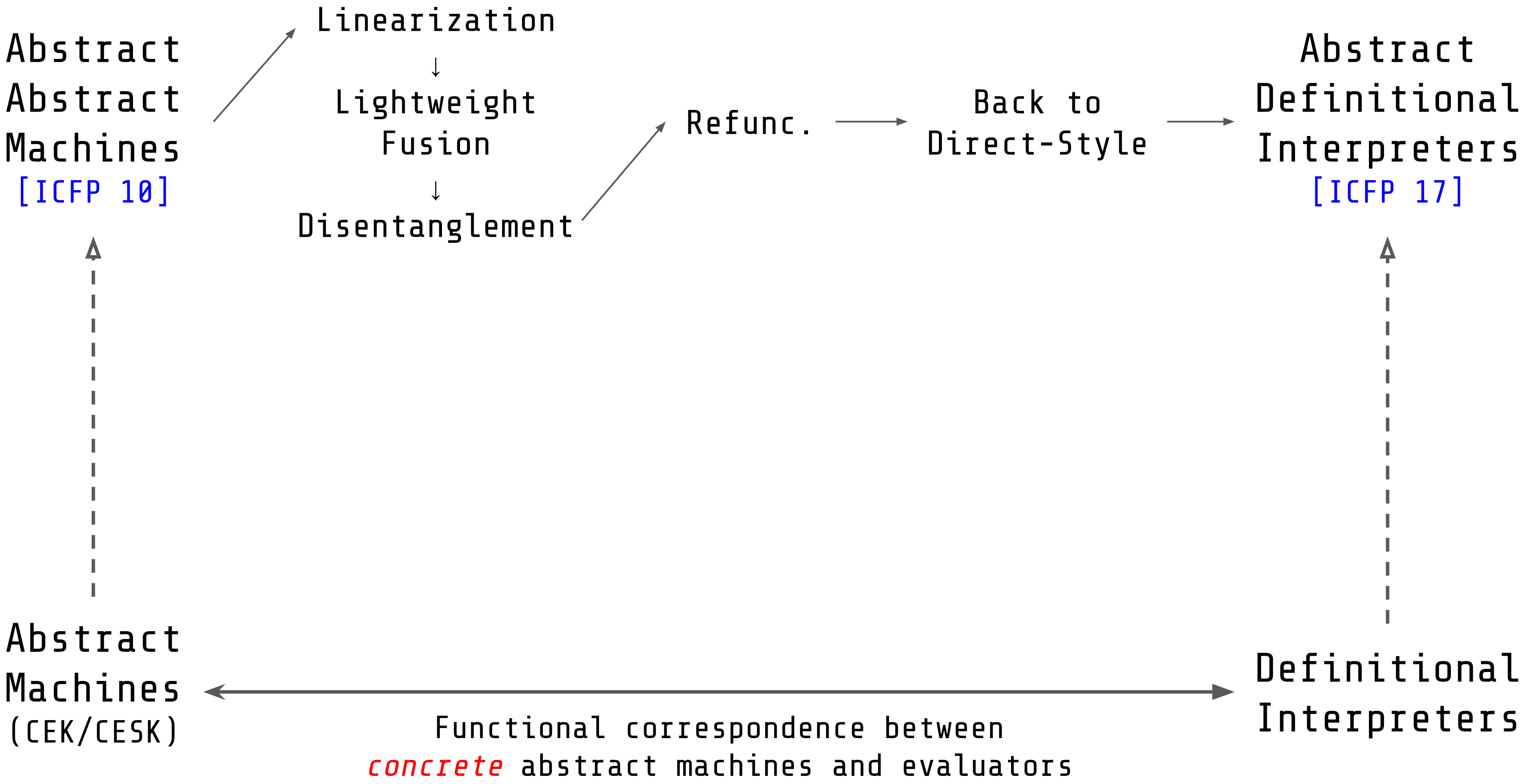
Get a closure of f, nondeterministically.

choices uses shift to capture the continuation, implicitly.

# What is still missing?

- The abstract interpreter may not terminate!
  *Solution*: Co-inductive caching [Darais et al. ICFP 17] that ensures reaching fixed-points.

- The aeval still returns a set of states.
  *Solution*: Only returns a set of final values instead of collected states.


Abstract Abstract Machines
[ICFP 10]
Linearization
↓
Lightweight Fusion
↓
Disentanglement
Refunc.
Back to Direct-Style
Abstract Definitional Interpreters
[ICFP 17]
Abstract Machines
(CEK/CESK)
Definitional Interpreters
Functional correspondence between
*concrete* abstract machines and evaluators

Abstract Abstract Machines [ICFP 10]

Linearization
↓
Lightweight Fusion
↓
Disentanglement

Refunc. → Back to Direct-Style → Abstract Definitional Interpreters [ICFP 17]

- Linearization transforms nondeterministic choices to another continuation.

Abstract Machines (CEK/CESK) ↔ Definitional Interpreters

Functional correspondence between *concrete* abstract machines and evaluators

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

- Linearization transforms nondeterministic choices to another continuation.
- Existing techniques for concrete functional correspondence, but we use it for *two* continuations.

Abstract Machines (CEK/CESK) ↔ Definitional Interpreters

Functional correspondence between *concrete* abstract machines and evaluators

Abstract Abstract Machines
[ICFP 10]

Linearization
↓
Lightweight Fusion
↓
Disentanglement

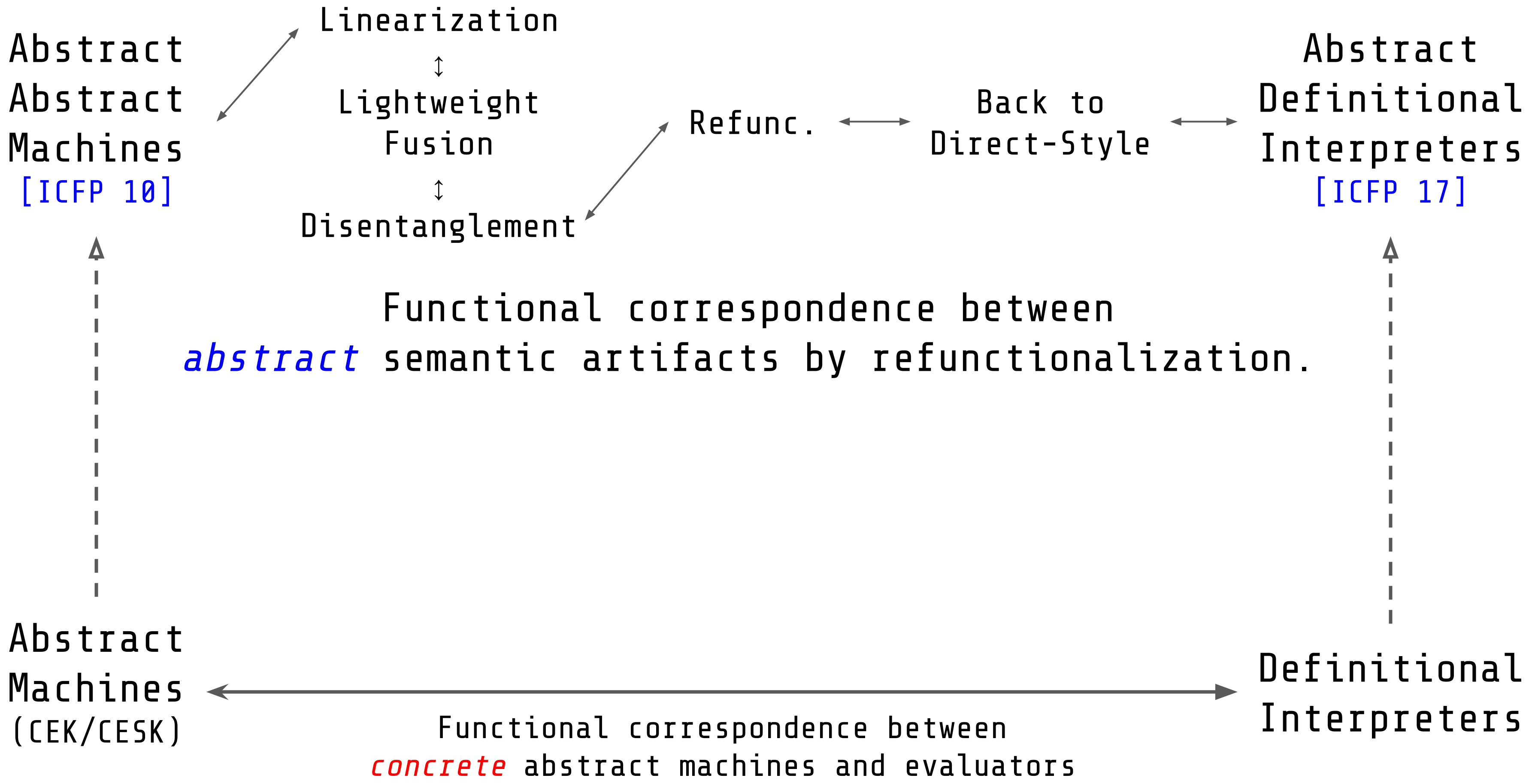
Refunc. → Back to Direct-Style → Abstract Definitional Interpreters
[ICFP 17]

- Linearization transforms nondeterministic choices to another continuation.
- Existing techniques for concrete functional correspondence, but we use it for *two* continuations.
- shift/reset to transform CPS back to direct-style.

Abstract Machines
(CEK/CESK)

Definitional Interpreters

Functional correspondence between
*concrete* abstract machines and evaluators


Abstract Abstract Machines
[ICFP 10]
Linearization
↕
Lightweight Fusion
↕
Disentanglement
Refunc.
Back to Direct-Style
Abstract Definitional Interpreters
[ICFP 17]
Functional correspondence between
*abstract* semantic artifacts by refunctionalization.
Abstract Machines
(CEK/CESK)
Definitional Interpreters
Functional correspondence between
*concrete* abstract machines and evaluators

Abstract
Abstract
Machines
[ICFP 10]

Linearization
↕
Lightweight
Fusion
↕
Disentanglement

Refunc. ⟷ Back to Direct-Style ⟷

Abstract
Definitional
Interpreters
[ICFP 17]

Functional correspondence between
*abstract* semantic artifacts by refunctionalization.

# Thanks!

# Questions?

Abstract
Machines
(CEK/CESK)

⟷

Definitional
Interpreters

Functional correspondence between
*concrete* abstract machines and evaluators