A Functional Correspondence between Monadic Evaluators and Abstract Machines for Languages with Computational Effects

Mads Sig Ager (mads@brics.dk)
Olivier Danvy (danvy@brics.dk)
Jan Midtgaard (jmi@brics.dk)
BRICS, University of Aarhus, Denmark

On the computational content of semantic specifications.

- Denotational semantics: a compositional evaluation function (e.g., from expressions and environments to expressible values).
- Big-step operational semantics: a relation (e.g., between expressions, environments, and expressible values).
- Small-step operational semantics: a transition function (from state to state).

Semantic specifications as computational objects.

- Denotational semantics: a compositional evaluation function – an evaluator.
- Big-step operational semantics: a relation – a logic program in general, and a functional program in particular.
- Small-step operational semantics: an abstract machine.

Semantic specifications as data objects.

Semantic specifications as data objects.

Here: as the object of program transformations.

Example of program transformations

- lambda lifting / lambda dropping
- closure conversion / Church encoding
- CPS transformation / DS transformation
- defunctionalization / refunctionalization
- data-stack introduction / elimination

Domain of discourse: the λ-calculus

Why: $\longrightarrow$ Idealized programming language.

$\longrightarrow$ Forty years of experience

to draw from and reflect on.

The point (already made last year)

evaluator

↓

closure conversion (to make it first order)

CPS transformation (to make it sequential)

defunctionalization (to make it first order)

abstract machine

Example in direct style

```
(*   fac : int -> int   *)
fun fac 0
    = 1
  | fac n
    = n * (fac (n - 1))

fun main n
    = fac n
```

Example in CPS

```
(*   fac : int * (int -> int) -> int   *)
fun fac (0, k)
    = k 1
  | fac (n, k)
    = fac (n - 1, fn v => k (n * v))

fun main n
    = fac (n, fn a => a)
```

The function space to defunctionalize

```
(*   fac : int * ( int -> int ) -> int   *)
fun fac (0, k)
    = k 1
  | fac (n, k)
    = fac (n - 1, fn v => k (n * v))

fun main n
    = fac (n, fn a => a)
```

The constructors

```
(*   fac : int * (int -> int) -> int   *)
fun fac (0, k)
    = k 1
  | fac (n, k)
    = fac (n - 1, fn v => k (n * v))

fun main n
    = fac (n, fn a => a)
```

The consumers

```
(*   fac : int * (int -> int) -> int   *)
fun fac (0, k)
      = k 1
  | fac (n, k)
      = fac (n - 1, fn v => k (n * v))

fun main n
      = fac (n, fn a => a)
```

The defunctionalized continuation

```
datatype cont = C0
              | C1 of cont * int

fun apply (C0, v)
    = v
  | apply (C1 (k, n), v)
    = apply (k, n * v)
```

Factorial in CPS, defunctionalized

```
fun fac (0, k)
      = apply (k, 1)
  | fac (n, k)
      = fac (n - 1, C1 (k, n))

fun main n
      = fac (n, C0)
```

Correctness

By structural induction on `n`,
using a logical relation over
the original continuation and
the defunctionalized continuation.

Factorial, defunctionalized

```
fun fac (0, k)
    = apply (k, 1)
  | fac (n, k)
    = fac (n - 1, C1 (k, n))

fun main n
    = fac (n, C0)
```

Factorial, as a transition system

```
fun fac (0, k)
      = apply (k, 1)
  | fac (n, k)
      = fac (n - 1, C1 (k, n))

fun main n
      = fac (n, C0)
```

Factorial, as a transition system

$$n \Rightarrow \langle\, n, C_0 \,\rangle_{fac}$$

$$\langle\, 0, k \,\rangle_{fac} \Rightarrow \langle\, k, 1 \,\rangle_{app}$$

$$\langle\, n, k \,\rangle_{fac} \Rightarrow \langle\, n - 1, C_1(n, k) \,\rangle_{fac}$$

$$\langle\, C_1(n, k), v \,\rangle_{app} \Rightarrow \langle\, k, n \times v \,\rangle_{app}$$

$$\langle\, C_0, v \,\rangle_{app} \Rightarrow v$$

Last year's experiment #1: CBN

canonical CBN evaluator for λ-terms

↓

closure conversion

CPS transformation

defunctionalization

↓

abstract machine

Last year's experiment #1: CBN

canonical CBN evaluator for λ-terms

↓

closure conversion

CPS transformation

defunctionalization

↓

Krivine's abstract machine

Krivine’s abstract machine

The abstract machine

of theoreticians.

Last year's experiment #2: CBV

canonical CBV evaluator for λ-terms

↓

closure conversion

CPS transformation

defunctionalization

↓

abstract machine

Last year's experiment #2: CBV

canonical CBV evaluator for λ-terms

↓ closure conversion

CPS transformation

defunctionalization

Felleisen et al.'s CEK abstract machine

The CEK abstract machine

The simplest abstract machine

of programming-language people.

Significance of the result

Krivine's machine and the CEK machine:

- The two best-known abstract machines for the λ-calculus.
- Developed and presented independently.

Other evaluators and abstract machines

- SECD, CLS, CAM, VEC, etc.
- call by need
- Featherweight Java, propositional Prolog, etc.

The correspondence holds.

Related work (1/2)

- Reynolds: "Definitional Interpreters, etc."
- ...much, much work,

  including textbooks such as

  "Essentials of Programming Languages"

Related work (2/2)

- Graunke, Findler, Krishnamurthi, and Felleisen: “Automatically Restructuring Programs for the Web” (ASE 2001)
- Schmidt: “State Transition Machines for Lambda-Calculus Expressions” (SDCG 1980)

This work

We build on Moggi's insight

as embodied in Wadler's interpreters.

One generic interpreter,

parameterized by a monad.

The style is in the monad.

The point

monadic evaluator + monad

↓

inlining (to make it ‘styled’)

closure conversion

CPS transformation

defunctionalization

↓

abstract machine

Several detailed examples

In the paper:

- The identity monad.

  Result: the CEK machine.

- A lifted state monad.

  Result: the CEK machine
  with error and state.

Stack inspection

- A security mechanism to allow code with different levels of trust to interact in the same execution environment.
- Before execution, the source code is annotated with permissions.
- During execution, the call stack is inspected to check whether the required permissions are available.

Stack inspection and proper tail recursion

Clements and Felleisen, ESOP 2003:

properly tail recursive stack inspection

with the CM machine.

Our observation

- Stack inspection can be characterized as a lifted state monad:

```
type 'a monad
= permission_table list
  -> ('a * permission_table list) lift
```

- The functional correspondence applies.

See Section 6 in the article.

A simpler monad for stack inspection

Only the top-most permission table is updated:

```
type 'a monad
= permission_table * permission_table list
  -> ('a * permission_table) lift
```

See Section 7 in the article.

Compound monads:
stack inspection + exceptions

See the BRICS tech report.

Conclusion: What

- The functional correspondence is compatible with monads.
- It makes it possible to mechanically construct abstract machines for languages with effects.

interpreter ⟵ inter derivation ⟶ abstract machine

Conclusion: How

Standard program transformations:

- Closure conversion.
- Data-stack introduction.
- CPS transformation.
- Defunctionalization.

Conclusion: How much

- Known, and not home-grown, machines.
- Variants of known machines.
- New machines.

Conclusion: How much

- Evaluation.
- Normalization.
- Logic programming.
- Imperative programming.
- Object-oriented programming.

Thank you.

Krivine's machine and the CEK machine

- The two best-known abstract machines for the λ-calculus.
- Developed and presented independently.

A “Wadlerian” classification

- Krivine: the logician.
- Felleisen: the computer scientist.

Limits of the classification

- Reynolds: the theoretician (CBN).
- Landin: the programmer (CBV).

A point remains, however

- Discovery (KAM).
- Invention (CEK).

Flashback

John Reynolds's warning

about evaluation-order independence.

A canonical evaluator

```
datatype term
= IND of int  (* de Bruijn index *)
| ABS of term
| APP of term * term

datatype expval
= FUN of denval -> expval
withtype denval = expval
```

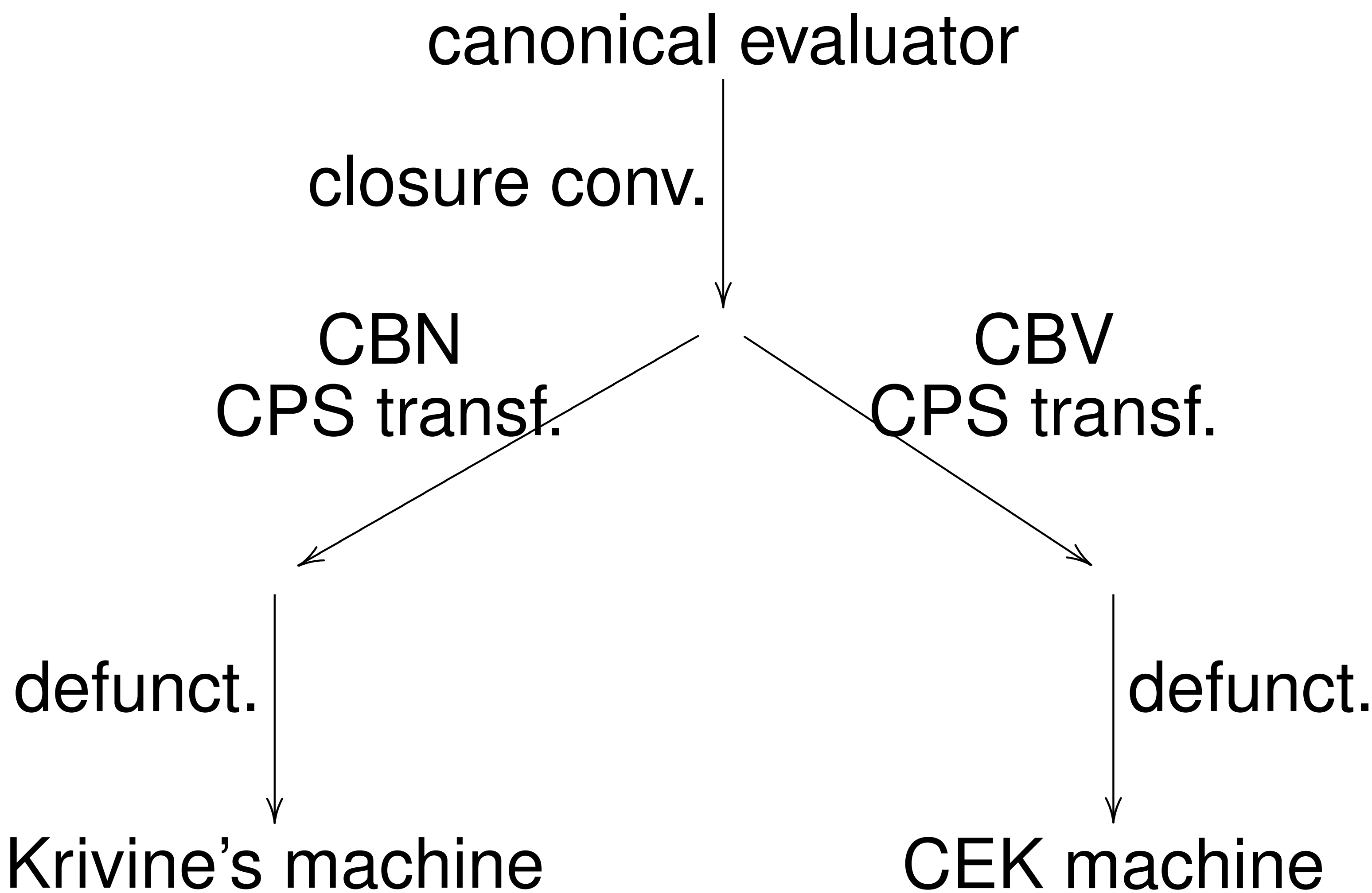
```
fun eval (IND n, e)
    = List.nth (e, n)
  | eval (ABS t, e)
    = FUN (fn v => eval (t, v :: e))
  | eval (APP (t0, t1), e)
    = let val (FUN f) = eval (t0, e)
      in f (eval (t1, e))
      end
```

John Reynolds's warning (1972)

Beware of the evaluation order

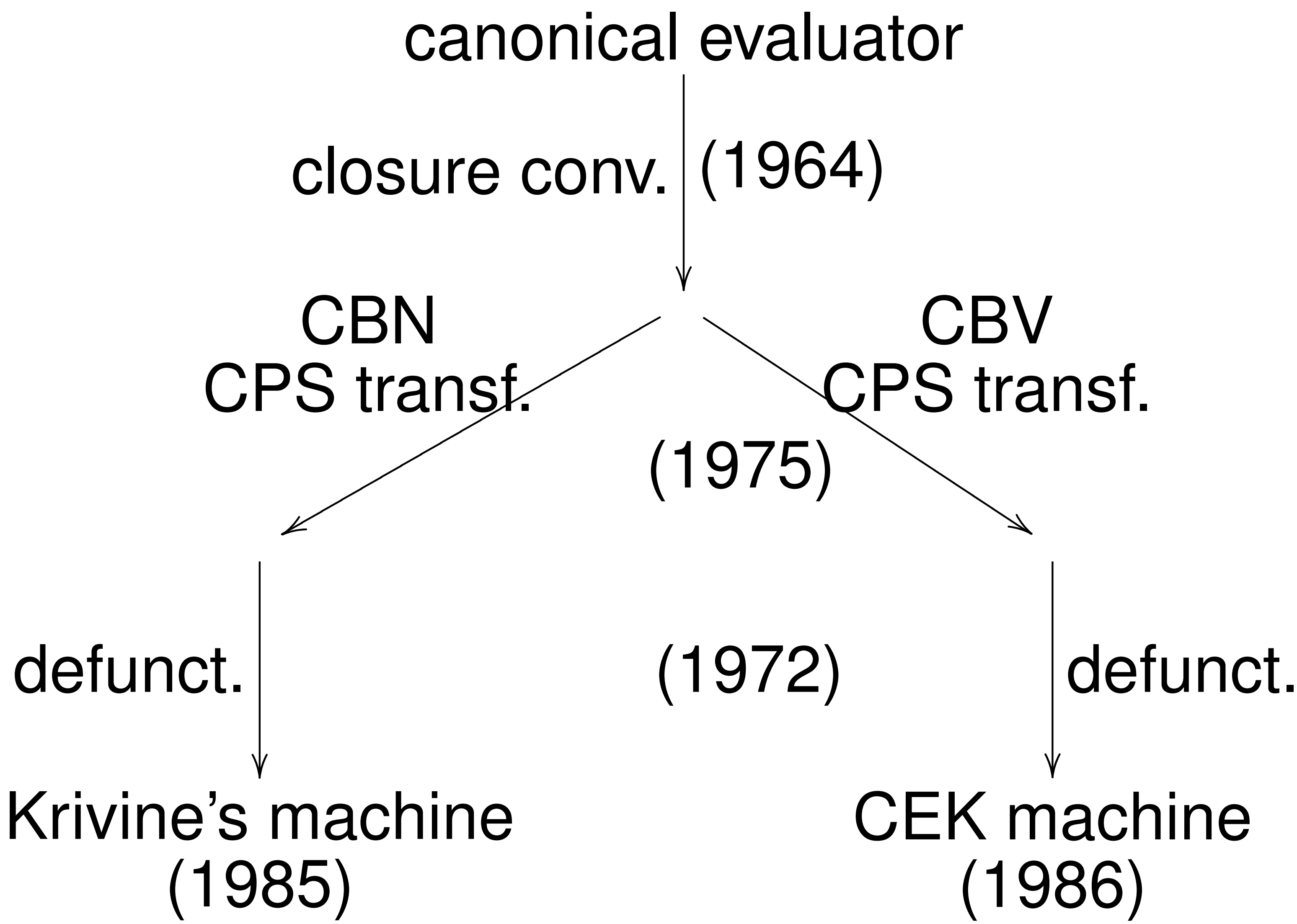
of the meta-language:

- Call by name yields call by name.
- Call by value yields call by value.


canonical evaluator
closure conv.
CBN
CPS transf.
CBV
CPS transf.
defunct.
defunct.
Krivine's machine
CEK machine

Consequence

Krivine's machine and the CEK machine are not just discovered and invented — they are two sides of the same standard coin.


canonical evaluator
closure conv.
(1964)
CBN
CPS transf.
CBV
CPS transf.
(1975)
defunct.
(1972)
defunct.
Krivine’s machine
(1985)
CEK machine
(1986)

Piet Hein's gentle reminder: T.T.T.

Put up in a place
where it's easy to see
the cryptic admonishment
T.T.T.
When you feel how depressingly
slowly you climb,
it's well to remember that
Things Take Time.

Models of abstract machines

- Eval-apply (CEK, etc.)
- Push-enter (KAM, etc.)

Models of abstract machines

- Eval-apply (CEK, etc.)
- Push-enter (KAM, etc.)

They appear naturally.

Call by need
(built-in dynamic programming)

Call by need: Call by name +

heap of updatable thunks.

Result: A host of known implementation

techniques and then some.

(see BRICS RS-04-03)