A Functional Correspondence between Evaluators and Abstract Machines

Olivier Danvy

BRICS

Department of Computer Science

University of Aarhus

Joint work with: Mads Sig Ager, Dariusz Biernacki, and Jan Midtgaard.

Evaluators and abstract machines

- An *evaluator* is an interpreter for expressions in a programming language.
- An *abstract machine* is a deterministic transition system for executing expressions.

Why abstract machines?

- Abstract machines have been widely used for implementing and reasoning about programming languages.
  - Functional languages: SECD, CEK, Krivine Machine, CLS, TIM, etc.
  - Logic programming languages: Warren Abstract Machine and variations.
  - Object-oriented programming languages: JVM, Smalltalk, Self, etc.

How are abstract machines traditionally obtained?

- Most abstract machines are obtained in ad-hoc ways (invented).
- Some abstract machines are subsequently proven to correctly implement the specification of a programming language.
- A few abstract machines have been constructed from specifications using formal methods.
- However, most of the constructions use ad-hoc steps, are complicated, and the machines obtained do not coincide with known machines.

Goal of this work

Simple mechanical constructions

of both known and new abstract machines

from high-level programming language specifications.

Outline

1. Basic observation
2. From evaluators to abstract machines
3. From abstract machines to evaluators
4. Conclusion

Basic observation

- A recursive function over an inductively defined datatype can be mechanically transformed to a transition system by the following program-transformation steps:

1. if locally defined functions are used, *lambda lift* the program,
2. if higher-order values are used, *closure convert* them,
3. transform the program into *continuation-passing style*, and
4. *defunctionalize* the continuations.

Introduction by example: the power function

- The power function $power(n, x) = x^n$ in SML:

```
fun power (n, x)
    = let fun loop 0
                = 1
            | loop n
                = x * (loop (n-1))
      in loop n
      end
```

Lambda lifting

- Transform block structured program to a set of recursive equations.
- Remove block structure by lifting locally defined functions to the top level.
- Pass local variables accessed by the function as extra arguments at each call site.

Lambda-lifted power function

```
fun loop_ll (0, x)
    = 1
  | loop_ll (n, x)
    = x * (loop_ll (n-1, x))

fun power_ll (n, x)
    = loop_ll (n, x)
```

Continuations-passing style (CPS) transformation

- Sequentialize computation.
- Name intermediate results.
- Introduce continuations.

CPS transformed power function

```
fun loop_cps (0, x, k)
    = k 1
  | loop_cps (n, x, k)
    = loop_cps (n-1, x, fn v => k (x * v))

fun power_cps (n, x)
    = loop_cps (n, x, fn x => x)
```

Defunctionalization

- Change of representation turning a higher-order program into an equivalent first-order program.
- Replaces functional values by first-order representations.
- Introduces an `apply` function interpreting the first-order representations.

Defunctionalizing power's continuations (1/4)

- Function space: `int -> int`.
- Inhabitants: `fn x => x` and `fn v => k (x * v)`.

```
fun loop_cps (0, x, k)
    = k 1
  | loop_cps (n, x, k)
    = loop_cps (n-1, x, fn v => k (x * v))

fun power_cps (n, x)
    = loop_cps (n, x, fn x => x)
```

Defunctionalizing power's continuations (2/4)

- Function space: `int -> int`.
- Inhabitants: `fn x => x` and `fn v => k (x * v)`.
- Datatype with summand for each inhabitant holding values of free variables:

```
datatype cont = STOP
              | MULT of int * cont
```

Defunctionalizing power’s continuations (3/4)

- Function space: `int -> int`.
- Inhabitants: `fn x => x` and `fn v => k (x * v)`.
- Datatype with summand for each inhabitant holding values of free variables:

```
datatype cont = STOP
              | MULT of int * cont
```

- Apply function interpreting the datatype summands:

```
fun apply_cont (STOP, v)
    = v
  | apply_cont (MULT (x, k), v)
    = apply_cont (k, x * v)
```

Defunctionalizing power's continuations (4/4)

```
datatype cont = STOP
              | MULT of int * cont

fun loop_defun (0, x, k)
    = apply_cont (k, 1)
  | loop_defun (n, x, k)
    = loop_defun (n-1, x, MULT (x, k))
and apply_cont (STOP, v)
    = v
  | apply_cont (MULT (x, k), v)
    = apply_cont (k, x * v)

fun power_defun (n, x)
    = loop_defun (n, x, STOP)
```

The result is a transition function

- Each function name together with its arguments define state.
- Each function body performs atomic actions and moves to new state.
- Reformatting:

$$
\begin{array}{rcl}
\hline\hline
(n, x) & \rightarrow_{init} & \langle n, x, STOP \rangle \\
\hline
\langle 0, x, k \rangle & \rightarrow_{loop} & \langle k, 1 \rangle \\
\langle n, x, k \rangle & \rightarrow_{loop} & \langle n-1, x, MULT(x, k) \rangle \\
\hline
\langle MULT(x, k), v \rangle & \rightarrow_{apply} & \langle k, x * v \rangle \\
\hline
\langle STOP, v \rangle & \rightarrow_{final} & v \\
\hline\hline
\end{array}
$$

Wait a minute! We skipped one: closure conversion

- Replace higher-order values by first-order representations which are closures pairing the body of higher-order values with an environment.

- Not needed for the power function since it uses no higher-order values.

- For the purpose of this work closure conversion is just defunctionalization of the higher-order values and inlining of the apply function.

Outline

1. Basic observation
2. From evaluators to abstract machines
   - call by value
   - call by name
   - call by need
3. From abstract machines to evaluators
4. Computational effects
5. Conclusion

Abstract machines?

- If the recursive function is an evaluator for a programming language (direct implementation of its denotational semantics) then the resulting transition system is an abstract machine!

- In the rest of this talk we will explore some of the consequences of this observation.

Call-by-value evaluation of λ-terms

- λ-terms:

$$t ::= x \mid \lambda x.t \mid t\,t$$

- λ-terms represented as an inductively defined datatype in ML:

```
datatype term = VAR of string
              | LAM of string * term
              | APP of term * term
```

Call-by-value evaluation of λ-terms

- Standard call-by-value evaluator:

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

fun eval (VAR x, e)
    = Env.lookup (x, e)
  | eval (LAM (x, t), e)
    = FUN (fn v => eval (t, Env.extend (x, v, e)))
  | eval (APP (t1, t2), e)
    = let val v1 = eval (t1, e)
          val v2 = eval (t2, e)
      in (case v1
            of (FUN f) => f v2)
      end
```

Call-by-value evaluation of λ-terms

- The standard call-by-value evaluator is a recursively defined function over an inductively defined datatype.
- Closure converting (defunctionalizing the expressible values and inlining the apply function), CPS-transforming, and defunctionalizing the continuations yields the CEK machine.

The CEK machine

$$
\begin{array}{rcl}
\hline\hline
t & \rightarrow_{init} & \langle t,\ e_{init},\ \texttt{stop}\rangle \\
\hline
\langle x,\ e,\ k\rangle & \rightarrow_{eval} & \langle k,\ e(x)\rangle \\
\langle \lambda x.t,\ e,\ k\rangle & \rightarrow_{eval} & \langle k,\ [x,\ t,\ e]\rangle \\
\langle t_0\, t_1,\ e,\ k\rangle & \rightarrow_{eval} & \langle t_0,\ e,\ \texttt{arg}(t_1, e, k)\rangle \\
\hline
\langle \texttt{arg}(t_1, e, k),\ v\rangle & \rightarrow_{cont} & \langle t_1,\ e,\ \texttt{fun}(v, k)\rangle \\
\langle \texttt{fun}([x,\ t,\ e], k),\ v\rangle & \rightarrow_{cont} & \langle t,\ e[x \mapsto v],\ k\rangle \\
\hline
\langle \texttt{stop},\ v\rangle & \rightarrow_{final} & v \\
\hline\hline
\end{array}
$$

Call-by-name evaluation of λ-terms

- Standard call-by-name evaluator:

```
datatype expval = FUN of denval -> expval
     and denval = THUNK of unit -> expval

fun eval (VAR x, e)
    = let (THUNK u) = Env.lookup (x, e)
      in u ()
      end
  | eval (LAM (x, t), e)
    = FUN (fn v => eval (t, Env.extend (x, v, e)))
  | eval (APP (t1, t2), e)
    = let val (FUN f) = eval (t1, e)
      in f (THUNK (fn () => eval (t2, e)))
      end
```

Call-by-name evaluation of λ-terms

- The standard call-by-name evaluator is a recursively defined function over an inductively defined datatype.

- Closure converting (defunctionalizing the expressible and denotable values and inlining the apply functions), CPS-transforming, defunctionalizing the continuations, and inlining the apply function yields the Krivine machine.

The Krivine machine

$$
\begin{array}{rcl}
\hline\hline
t & \rightarrow_{init} & \langle t,\ e_{init},\ \texttt{stop} \rangle \\
\hline
\langle x,\ e,\ k \rangle & \rightarrow_{eval} & \langle t,\ e',\ k \rangle,\ \ where\ \{t, e'\} = e(x) \\
\langle \lambda x.t,\ e,\ \texttt{arg}(t', e', k) \rangle & \rightarrow_{eval} & \langle t,\ e[x \mapsto \{t', e'\}],\ k \rangle \\
\langle t_0\, t_1,\ e,\ k \rangle & \rightarrow_{eval} & \langle t_0,\ e,\ \texttt{arg}(t_1, e, k) \rangle \\
\hline
\langle \lambda x.t,\ e,\ nil \rangle & \rightarrow_{final} & [\lambda x.t,\ e] \\
\hline\hline
\end{array}
$$

Call-by-need evaluation of λ-terms

- By introducing a heap, the evaluator for call-by-name evaluation of λ-terms can be turned into an evaluator for call-by-need evaluation of λ-terms.
- Result: a lazy variant of Krivine's machine.
- Reference: "A Functional Correspondence between Call-by-Need Evaluators and Lazy Abstract Machines"

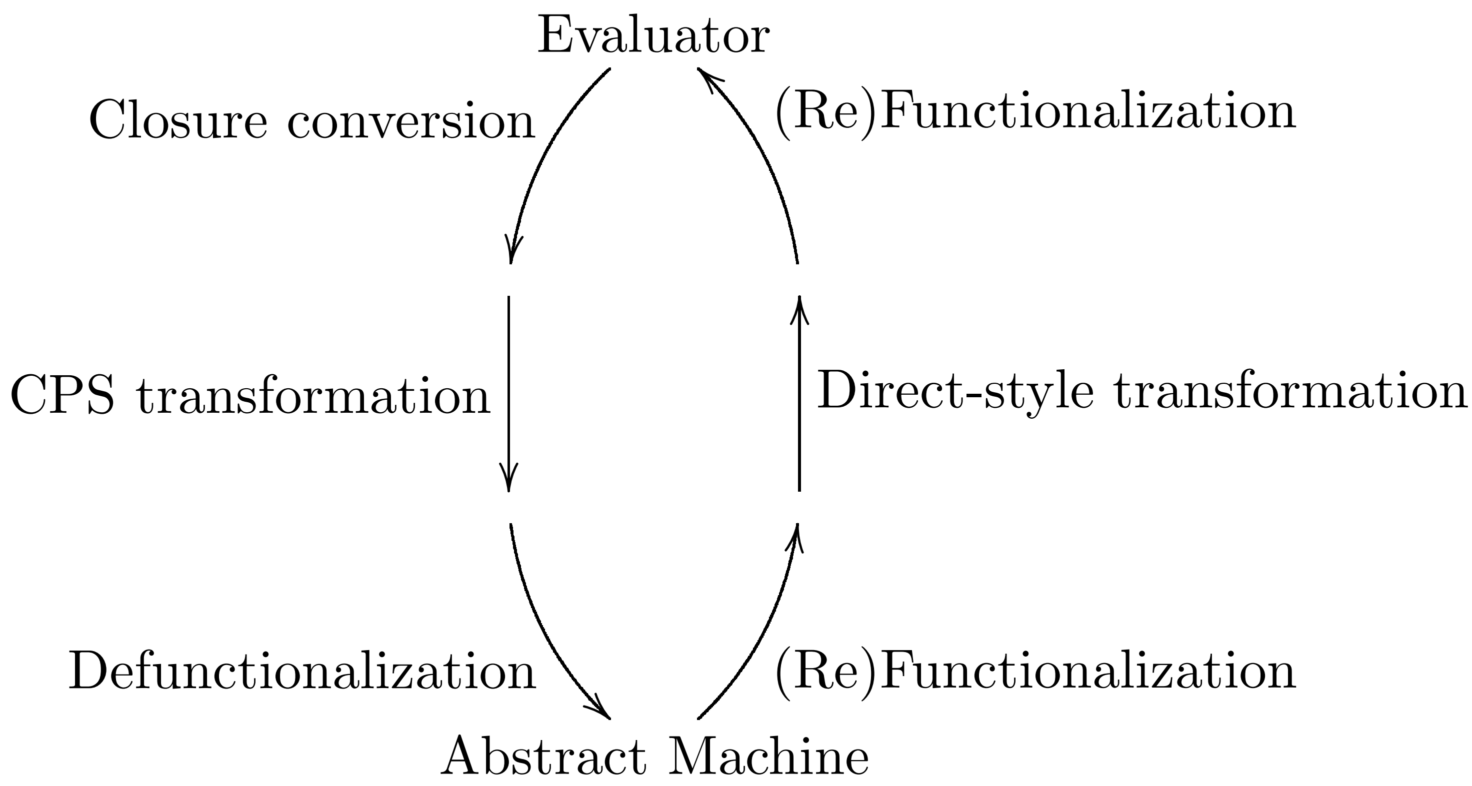
  Information Processing Letters 90(5):223-232, 2004

  Extended version: BRICS-RS-04-3

Outline

1. Basic observation
2. From evaluators to abstract machines
3. From abstract machines to evaluators
4. Computational effects
5. Conclusion

From abstract machines to evaluators



First conclusion (1/2)

- Recursive function over inductively defined datatype can be transformed to a transition system by standard well-studied program transformations.
- If the recursive function is an evaluator the resulting transition system is an abstract machine.

First conclusion (2/2)

- Closure conversion, CPS transformation and defunctionalization provides a direct correspondence between evaluators and abstract machines.
- This correspondence enables us to mechanically construct known abstract machine (CEK, Krivine) as well a new ones (call-by-need machine) from high-level specifications in the form of evaluators.

Outline

1. Basic observation

2. From evaluators to abstract machines

3. From abstract machines to evaluators

4. Computational effects

   (a) Monads as a factorization device

   (b) Deriving abstract machines for languages with computational effects

   (c) Characterizing stack inspection as a lifted state monad

5. Conclusion

Monads

- For the purpose of this work we use monads purely as a factorization device for evaluators.
- No category theory will be involved.

Monads as a factorization device

- Following Wadler we specify a monad as a type constructor and two polymorphic functions `unit` (injection into the monad) and `bind` (sequencing of computations):
  - `datatype 'a M = ...`
  - `unit : 'a -> 'a M`
  - `bind : 'a M * ('a -> 'b M) -> 'b M`

Monads as a factorization device

- The type constructor, `unit` and `bind` specifies a monad if the following monadic laws hold:
  - Left unit: `bind (unit a, k) = k a`
  - Right unit: `bind (k, unit) = k`
  - Associative:

```
bind (m, fn a => bind (k a, h))
= bind (bind (m, k), h)
```

- The monadic laws ensure that things compose as we expect them to.

Example monads: identity

- Identity monad:

```
type 'a M = 'a

fun unit a = a

fun bind (m, k) = k m
```

Example monads: state

- State monad where state is one integer for simplicity:

```
type storable = int

type 'a M = storable -> 'a * storable

fun unit a
    = fn s => (a, s)

fun bind (m, k)
    = fn s => let val (a, s') = m s
              in k a s'
              end
```

Example monads: state

- The state monad comes with operations for getting and setting the state:

```
(*  get : storable M  *)
val get = (fn s => (s, s))

(*  set : storable -> storable M  *)
fun set i = (fn s => (s, i))
```

Example monads: exceptions

- Exception monad with one kind of exception:

```
datatype 'a E = EXP of 'a | EXC

type 'a M = 'a E

fun unit a = EXP a

fun bind (m, k)
    = (case m
         of (EXP a) => k a
          | EXC => EXC)
```

Example monads: exceptions

- The exception monad comes with operations for raising and handling exceptions:

```
(*  raise_exception : ’a M  *)
val raise_exception = EXC

(*  handle_exception : ’a M * (unit -> ’a M)  *)
fun handle_exception (m, h)
    = (case m
         of (EXP a) => EXP a
          | EXC => h ())
```

- Call-by-value monadic evaluator:

```
fun eval (VAR x, e)
    = M.unit (Env.lookup (x, e))
  | eval (LAM (x, t), e)
    = M.unit (FUN (fn v => eval (t, Env.extend (x, v, e))))
  | eval (APP (t0, t1), e)
    = M.bind (eval (t0, e),
              fn v0 => M.bind (eval (t1, e),
                               fn v1 => let val (FUN f) = v0
                                        in f v1
                                        end))
```

Instantiations

- The generic evaluator together with a monad specify an evaluator for the lambda calculus extended with the effect of the monad.

- Inlining the monad, i.e., inlining the definitions of `unit` and `bind` in the evaluator gives an evaluator in a style specific to that monad.

Summary: monads as a factorization device

- Monads can be used for specifying computational effects.
- One generic evaluator can be instantiated with many different monads giving rise to evaluators for language with different effects.

Outline

1. Monads as a factorization device
2. Deriving abstract machines for languages with computational effects
3. Characterizing stack inspection as a lifted state monad
4. Conclusion

Deriving AMs for languages with computational effects

- Inlining a monad in the generic evaluator yields an evaluator in a style specific to the computational effect of the monad.
- By closure converting, CPS transforming and defunctionalizing such evaluators we obtain abstract machines for language with computational effects.

Deriving AMs for languages with computational effects

generic
monadic evaluator

computational monad

instantiation

specific evaluator

inlining ($\Rightarrow$ specific style)

closure conversion ($\Rightarrow$ first-order data)

CPS transformation ($\Rightarrow$ sequential evaluation)

defunctionalization ($\Rightarrow$ first-order control)

abstract machine

Example: identity monad

- Identity monad:
  - Inlining yields standard call-by-value evaluator,
  - closure converting, CPS transforming and defunctionalizing transforms the standard evaluator into the CEK machine.

Example: state monad

- State monad:
  - Inlining yields call-by-value evaluator in state-passing style,
  - closure converting, CPS transforming and defunctionalizing transforms the standard evaluator into a variant of the CEK machine with state.

Example: exception monad

- Exception monad:
  - Inlining yields a call-by-value evaluator in exception oriented style,
  - closure converting, CPS transforming and defunctionalizing transforms the standard evaluator into a variant of the CEK machine with exceptions.

Significance

- Each of the variants of the CEK was obtained mechanically from a monad specifying the computational effect.
- Each of the variants have previously been independently developed.
- We are now in position to derive <u>new</u> variants of the CEK machine for <u>any</u> computational effect expressed as a monad.

Outline

1. Monads as a factorization device

2. Deriving abstract machines for languages with computational effects

3. Characterizing stack inspection as a lifted state monad

4. Conclusion

Stack inspection

- Security mechanism used in the JVM and CLR.
- Allows code with different levels of trust to safely interact in the same execution environment.

Stack inspection

- All code is annotated with a set of permissions:
  - system code is annotated with all permissions, and
  - applets are only annotated with a small subset of permissions.
- Before access is granted to a resource the call stack is traversed to check that all callers are annotated with the permissions to access the resource.

Stack inspection

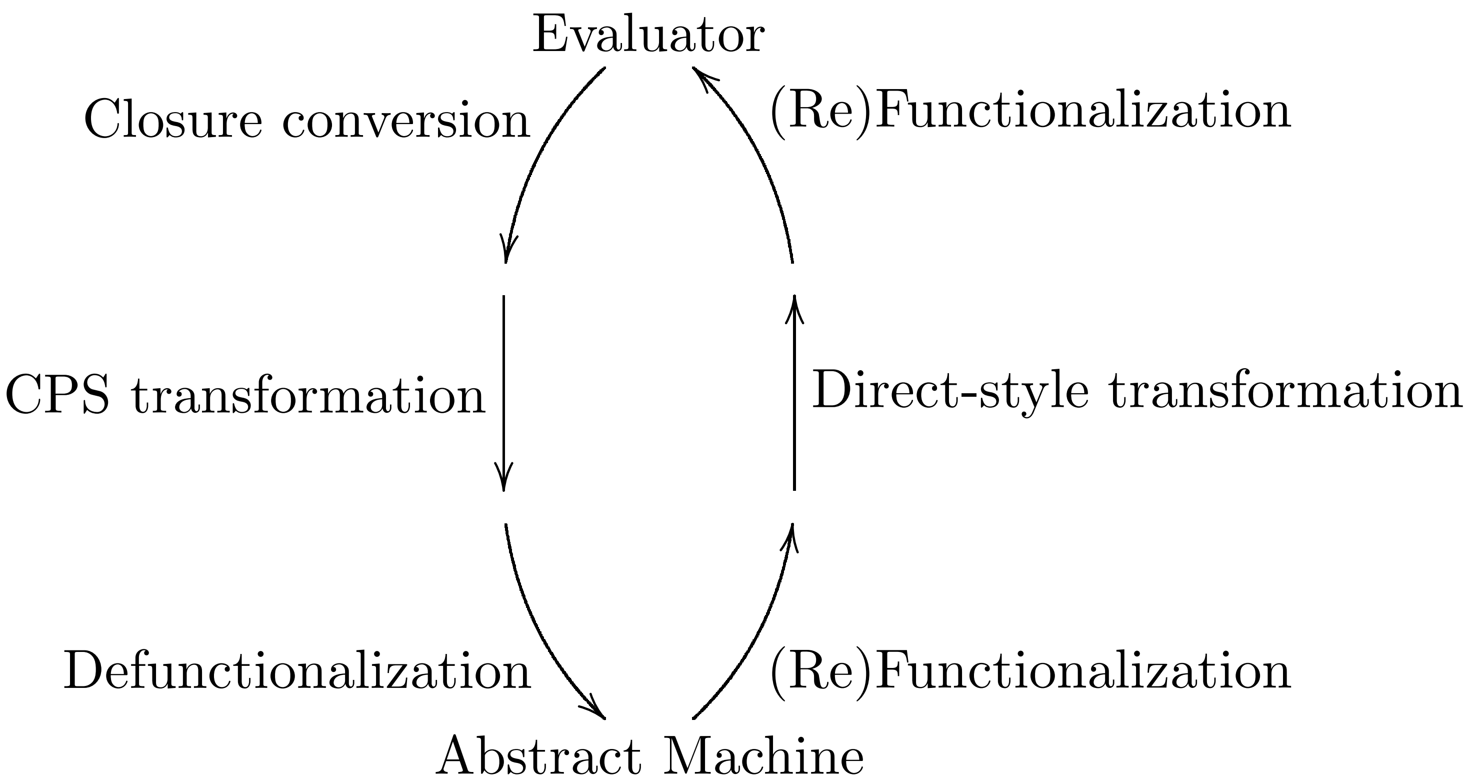
- Traversing the stack ensures that a piece of code cannot trick the system and indirectly gain access to a resource by calling trusted code.
- However, stack traversals seem to make stack inspection incompatible with tail-call optimization.
- Stack frames for all callers need to be present to ensure security.

Stack inspection as a lifted state monad

- ESOP 2003: Clements and Felleisen present a properly tail recursive semantics for stack inspection.
- Summarizes permissions in a permission table in each stack frame.
- The semantics is expressed as a variant of the CEK machine.
- They prove that the machine is properly tail recursive by bounding its space consumption.

Stack inspection as a lifted state monad

- Reversing the correspondence:



Stack inspection as a lifted state monad

- Refunctionalizing and direct-style transforming this CEK machine yields a state-passing call-by-value evaluator that can fail due to security errors.

- This evaluator can be expressed as a generic evaluator parameterized with a lifted state monad.

Stack inspection as a lifted state monad

- This characterization of stack inspection as a lifted state monad enables us to mechanically construct abstract machines for languages with properly tail-recursive stack inspection and other effects:
  - combine the lifted state monad for stack inspection with other monads to get the desired effect (pun intended),
  - inline this combined monad in the generic evaluator,
  - mechanically derive the corresponding abstract machine using the functional correspondence.

Outline

1. Monads as a factorization device
2. Deriving abstract machines for languages with computational effects
3. Characterizing stack inspection as a lifted state monad
4. Conclusion

Conclusion

- Abstract machines for languages with computational effects can be obtained by:
  - designing a monad for the desired computational effect,
  - inlining the monad in a generic evaluator,
  - closure converting, CPS-transforming and defunctionalizing this evaluator.
- In this talk we have only considered *one* monadic call-by-value evaluator. The same story can be told for other monadic evaluators.

Conclusion: recap

- A functional correspondence between evaluators and abstract machines: closure conversion, CPS transformation and defunctionalization provides a direct correspondence between evaluators and abstract machines.
- The correspondence is simple and constructive.

Conclusion: recap

- Abstract machines has been a topic of research for many years.
- Many machines have been independently invented and studied.
- Many articles have been written presenting *one* machine with a certain property.

Conclusion: recap

The functional correspondence

- provides a unifying methodology for mechanically deriving known machines from evaluators,
- puts us in position to construct a variety of new abstract machines, and
- enables us to “reverse engineer” existing machines and gain insights by considering the underlying evaluators.

Next: Implementing first-class continuations.