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.
Abstract machines have been widely used for implementing and reasoning about programming languages.

- Functional languages: SECD, CEK, Krivine Machine, CLS, TIM, etc.
- 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.
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.

Olivier Danvy, EWSCS, March 3, 2005
The power function $power(n, x) = x^n$ in SML:

```ml
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.
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.
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: \( \text{int} \rightarrow \text{int} \).
- Inhabitants: \( \text{fn } x \rightarrow x \) and \( \text{fn } v \rightarrow k(x \times v) \).

\[
\text{fun loop_cps} (0, x, k) \\
\quad = k \ 1 \\
\quad | \quad \text{loop_cps} (n, x, k) \\
\quad = \text{loop_cps} (n-1, x, \text{fn } v \rightarrow k(x \times v))
\]

\[
\text{fun power_cps} (n, x) \\
\quad = \text{loop_cps} (n, x, \text{fn } x \rightarrow x)
\]
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
```
• Function space: \( \text{int} \rightarrow \text{int} \).

• Inhabitants: \( \text{fn } x =\rightarrow x \) and \( \text{fn } v =\rightarrow k \ (x \ast v) \).

• Datatype with summand for each inhabitant holding values of free variables:

\[
\text{datatype cont} = \text{STOP} \\
| \quad \text{MULT of int} \ast \text{cont}
\]

• Apply function interpreting the datatype summands:

\[
\text{fun apply\_cont} \ (\text{STOP}, \ v) \\
\quad = v \\
| \quad \text{apply\_cont} \ (\text{MULT} \ (x, k), \ v) \\
\quad = \text{apply\_cont} \ (k, x \ast v)
\]
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:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(n, x) →_{init}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>⟨n, x, STOP⟩</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>⟨0, x, k⟩ →_{loop} ⟨k, 1⟩</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>⟨n, x, k⟩ →_{loop} ⟨n − 1, x, MULT(x, k)⟩</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>⟨MULT(x, k), v⟩ →_{apply} ⟨k, x * v⟩</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>⟨STOP, v⟩ →_{final} v</td>
</tr>
</tbody>
</table>

Olivier Danvy, EWSCS, March 3, 2005

–18–
• 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.
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 $\lambda$-terms

- $\lambda$-terms:

\[ t ::= x | \lambda x.t | t \ t \]

- $\lambda$-terms represented as an inductively defined datatype in ML:

\[
\text{datatype term} = \text{VAR of string} \\
| \text{LAM of string * term} \\
| \text{APP of term * term}
\]
Call-by-value evaluation of $\lambda$-terms

• Standard call-by-value evaluator:

```latex
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
```
• 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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Rule</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t )</td>
<td>( \rightarrow_{init} )</td>
<td>( \langle t, e_{init}, \text{stop} \rangle )</td>
</tr>
<tr>
<td>( \langle x, e, k \rangle )</td>
<td>( \rightarrow_{eval} )</td>
<td>( \langle k, e(x) \rangle )</td>
</tr>
<tr>
<td>( \langle \lambda x . t, e, k \rangle )</td>
<td>( \rightarrow_{eval} )</td>
<td>( \langle k, [x, t, e] \rangle )</td>
</tr>
<tr>
<td>( \langle t_0 , t_1, e, k \rangle )</td>
<td>( \rightarrow_{eval} )</td>
<td>( \langle t_0, e, \text{arg}(t_1, e, k) \rangle )</td>
</tr>
<tr>
<td>( \langle \text{arg}(t_1, e, k), v \rangle )</td>
<td>( \rightarrow_{cont} )</td>
<td>( \langle t_1, e, \text{fun}(v, k) \rangle )</td>
</tr>
<tr>
<td>( \langle \text{fun}([x, t, e], k), v \rangle )</td>
<td>( \rightarrow_{cont} )</td>
<td>( \langle t, e[x \mapsto v], k \rangle )</td>
</tr>
<tr>
<td>( \langle \text{stop}, v \rangle )</td>
<td>( \rightarrow_{\text{final}} )</td>
<td>( v )</td>
</tr>
</tbody>
</table>
Call-by-name evaluation of $\lambda$-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
```
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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t$</td>
<td>$\rightarrow_{init}$</td>
<td>$\langle t, e_{init}, \text{stop} \rangle$</td>
</tr>
<tr>
<td>$\langle x, e, k \rangle$</td>
<td>$\rightarrow_{eval}$</td>
<td>$\langle t, e', k \rangle$, where ${t, e'} = e(x)$</td>
</tr>
<tr>
<td>$\langle \lambda x . t, e, \text{arg}(t', e', k) \rangle$</td>
<td>$\rightarrow_{eval}$</td>
<td>$\langle t, e[x \mapsto {t', e'}], k \rangle$</td>
</tr>
<tr>
<td>$\langle t_0 t_1, e, k \rangle$</td>
<td>$\rightarrow_{eval}$</td>
<td>$\langle t_0, e, \text{arg}(t_1, e, k) \rangle$</td>
</tr>
<tr>
<td>$\langle \lambda x . t, e, \text{nil} \rangle$</td>
<td>$\rightarrow_{final}$</td>
<td>$[\lambda x . t, e]$</td>
</tr>
</tbody>
</table>

Olivier Danvy, EWSCS, March 3, 2005
By introducing a heap, the evaluator for call-by-name evaluation of $\lambda$-terms can be turned into an evaluator for call-by-need evaluation of $\lambda$-terms.

Result: a lazy variant of Krivine’s machine.


Extended version: BRICS-RS-04-3
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

- Closure conversion
- CPS transformation
- Defunctionalization
- Abstract Machine

- Evaluator
- (Re)Functionalization
- Direct-style transformation
- (Re)Functionalization
• 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.
• 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.
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
For the purpose of this work we use monads purely as a factorization device for evaluators.

No category theory will be involved.
Following Wadler we specify a monad as a type constructor and two polymorphic functions \texttt{unit} (injection into the monad) and \texttt{bind} (sequencing of computations):

\begin{itemize}
  \item \texttt{datatype 'a M = ...}
  \item \texttt{unit : 'a -> 'a M}
  \item \texttt{bind : 'a M * ('a -> 'b M) -> 'b M}
\end{itemize}
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.
Identity monad:

\[
\text{type 'a M = 'a} \\
\text{fun unit a = a} \\
\text{fun bind (m, k) = k m}
\]
- Example monads: state

- State monad where state is one integer for simplicity:

```ml
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
```
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))
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)
```
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 ())
Generic monadic evaluator

• 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))
• 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.
● 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.
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
● 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

inlining (⇒ specific style)
closure conversion (⇒ first-order data)
CPS transformation (⇒ sequential evaluation)
defunctionalization (⇒ first-order control)

abstract machine

specific evaluator

instantiation
• Identity monad:
  – Inlining yields standard call-by-value evaluator,
  – closure converting, CPS transforming and defunctionalizing transforms the standard evaluator into the CEK machine.
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.
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 new variants of the CEK machine for any computational effect expressed as a monad.
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.
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.
• Reversing the correspondence:
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.
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
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.
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.
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.