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

Mads Sig Ager, Olivier Danvy, and Jan Midtgaard

BRICS[⋆], Department of Computer Science, University of Aarhus[⋆⋆]

Abstract. We extend our correspondence between evaluators and abstract machines from the pure setting of the λ-calculus to the impure setting of the computational λ-calculus. We show how to derive new abstract machines from monadic evaluators for the computational λ-calculus. Starting from (1) a generic evaluator parameterized by a monad and (2) a monad specifying a computational effect, we inline the components of the monad in the generic evaluator to obtain an evaluator written in a style that is specific to this computational effect. We then derive the corresponding abstract machine by closure-converting, CPS-transforming, and defunctionalizing this specific evaluator. We illustrate the construction with the identity monad, obtaining yet again the CEK machine, and with a lifted state monad, obtaining a variant of the CEK machine with error and state.

In addition, we characterize the tail-recursive stack inspection presented by Clements and Felleisen at ESOP 2003 as a lifted state monad. This enables us to combine the stack-inspection monad with other monads and to construct abstract machines for languages with properly tail-recursive stack inspection and other computational effects. The construction scales to other monads—including one more properly dedicated to stack inspection than the lifted state monad—and other monadic evaluators.

Keywords. Lambda-calculus, interpreters, abstract machines, closure conversion, transformation into continuation-passing style (CPS), defunctionalization, monads, effects, proper tail recursion, stack inspection.

1 Introduction

Diehl, Hartel, and Sestoft's overview of abstract machines for programming-language implementation [14] concluded on the need to develop a theory of abstract machines. In previous work [2,9], we have attempted to contribute to this theory by identifying a correspondence between interpreters (i.e., evaluation functions in the sense of denotational semantics) and abstract machines (i.e., transition functions in the sense of operational semantics). The correspondence builds on the observation that defunctionalized continuation-passing evaluators are abstract machines. It is therefore constructive: to obtain an abstract machine, we start from a compositional evaluator and

1. closure-convert its expressible and denotable values [24,32];
2. materialize its control flow into continuations by CPS transformation [10,28,31]; and
3. defunctionalize these continuations [11,30].

The correspondence originates in Reynolds's seminal article "Definitional Interpreters for Higher-Order Programming Languages" [30], where all the elements (closure conversion, CPS transformation, and defunctionalization) are presented and used. Today, these elements are classical, textbook material [15,20]. Nevertheless, this correspondence has let us derive Krivine's machine from a canonical call-by-name evaluator and Felleisen et al.'s CEK machine from a canonical call-by-value evaluator. These two machines have been independently developed and—to the best of our knowledge—have never been associated with defunctionalization, CPS transformation, and closure conversion. The correspondence has also let us reveal the evaluators underlying Landin's SECD machine, Schmidt's VEC machine, Hannan and Miller's CLS machine, and Curien et al.'s Categorical Abstract Machine [2,9].

[⋆] Basic Research in Computer Science (`www.brics.dk`),funded by the Danish National Research Foundation.

[⋆⋆] IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark. Email: `{mads,danvy,jmi}@brics.dk`

We have verified that the correspondence holds for call-by-need evaluators and lazy abstract machines [4], logic programming [6], imperative programming, and object-oriented programming, including Featherweight Java and a subset of Smalltalk. We have also constructed generalizations of Krivine's machine and of the CEK machine by starting from normalization functions [1]. The correctness of the abstract machines (resp. of the evaluators) is a corollary of the correctness of the evaluators (resp. of the abstract machines) and of the correctness of the transformations.

In this article, we take a next step by applying the methodology to evaluators and abstract machines for languages with computational effects [5, 27, 33]. We consider a generic evaluator parameterized by a monad (Sections 2 and 3). We then consider two monads: the identity monad (Section 4) and a lifted state monad (Section 5). We inline the components of these monads in the generic evaluator, obtaining two specific evaluators. We then construct the corresponding abstract machines by closure-converting, CPS-transforming, and defunctionalizing these specific evaluators.

We next turn to the security technique of 'stack inspection' [19]. Clements and Felleisen recently debunked the myth that stack inspection is incompatible with proper tail recursion [7]. To this end, they presented an abstract machine implementing stack inspection in a properly tail-recursive way. We characterize Clements and Felleisen's stack inspection as a lifted state monad (Section 6). We then present a monad that accounts for stack inspection more precisely than the lifted state monad, we review related work, and we conclude.

In a preliminary version of this article [3], we also considered a state monad, an exception monad, and the two possible monads obtained by combining these two monads. We mechanically constructed the corresponding abstract machines.

2 A call-by-value monadic evaluator in ML

As traditional [5,17,33], we specify a monad as a type constructor and two polymorphic functions. (We use Standard ML [26].)

```
signature MONAD
= sig
    type 'a monad

    val unit : 'a -> 'a monad
    val bind : 'a monad * ('a -> 'b monad) -> 'b monad
  end
```

Our source language is the untyped λ-calculus with integer literals:

```
datatype term = LIT of int
              | VAR of ide
              | LAM of ide * term
              | APP of term * term
```

where identifiers are represented with a value of type `ide`. Programs are closed terms.

The corresponding expressible values are integers and functions:

```
datatype value = NUM of int
               | FUN of value -> value M.monad
```

for a structure `M : MONAD`.

Our monadic interpreter uses an environment `Env` with the following signature:

```
signature ENV
= sig
    type 'a env

    val empty : 'a env
    val extend : ide * 'a * 'a env -> 'a env
    val lookup : ide * 'a env -> 'a
  end
```

Throughout this article e denotes environments and e_{empty} denotes the empty environment.

The evaluation function is defined by structural induction on terms:

```
(*  eval : term * value Env.env -> value M.monad  *)
fun eval (LIT i, e)
    = M.unit (NUM i)
  | 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))
```

Given a program, the main evaluation function calls `eval` with this term and the initial environment:

```
fun main t
    = eval (t, env_base)
```

In actuality, this evaluation function, `eval`, `env_base`, and `value` are defined in an ML functor parameterized with a structure `M : MONAD`.

Except for the identity monad, each monad comes with operations that need to be integrated in the source language. Rather than systematically extending the syntax of the source language with these operations, we hold some of them in the initial environment. For example, rather than having a special form for the successor function, we define it with a binding in the base environment:

```
val env_base = Env.extend ("succ", FUN (fn (NUM i) => M.unit (NUM (i + 1))), Env.empty)
```

3 On using ML as a metalanguage

ML is a Turing-complete, statically typed, call-by-value language with computational effects:

– ML programs can therefore diverge and to this end, ML comes with a 'built-in' lifting monad to account for divergence. In Section 2, we implicitly make use of this monad in the codomain of `eval`: applying `eval` to a term and an environment only yields a result if it terminates.
– Compiling the evaluator of Section 2 yields warnings to the effect that pattern matching, in the clause for `APP` and in the initial environment, is incomplete. Should we attempt to evaluate a source program that is ill-typed (e.g., because it applies the successor function to a function instead of to an integer), a run-time exception would be raised. In that sense, ML also comes with a 'built-in' error monad to account for pattern-matching errors.

In the remainder of this article, we instantiate the evaluator of Section 2 with monads. We could be pedantic and only consider monads that are layered on top of two lifting monads—one for pattern-matching errors and one for divergence. The result would be a notational overkill, and therefore we choose to use ML's built-in monads.

For the purpose of our work, we view monads as a factorization device for writing evaluators, and we symbolically simplify the monadic evaluator of Section 2 with respect to a given monad (thereby obtaining a direct-style evaluator out of the identity monad, a lifted evaluator out of the lifting monad, a state-threading evaluator out of a state monad, a continuation-passing evaluator out of the continuation monad, an exception-oriented evaluator out of an exception monad, etc.). Our symbolic simplification undoes Moggi's factorization and it is carried out by inlining the definitions of the type constructor, of `unit` and `bind`, and of the monadic operations.

4 From the identity monad to an abstract machine

We first specify the identity monad and inline its components in the monadic evaluator of Section 2, obtaining an evaluator in direct style. We then take the same steps as in our previous work [2]: closure conversion, CPS transformation, and defunctionalization. The result is Felleisen et al.'s CEK machine extended with literals [16, 18].

4.1 The identity monad

The identity monad is specified with an identity type constructor and the corresponding two polymorphic functions:

```
structure Identity_Monad : MONAD
= struct
    type 'a monad = 'a

    fun unit a = a
    fun bind (m, k) = k m
  end
```

4.2 Inlining the monad in the monadic evaluator

Inlining the components of the identity monad in the monadic evaluator of Section 2 yields a call-by-value evaluator in direct style:

```
datatype value = NUM of int
               | FUN of value -> value

val env_base = Env.extend ("succ", FUN (fn (NUM i) => (NUM (i + 1))), Env.empty)

(*  eval : term * value Env.env -> value  *)
fun eval (LIT i, e)
    = NUM i
  | 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 (t0, t1), e)
    = let val v0 = eval (t0, e)
          val v1 = eval (t1, e)
          val (FUN f) = v0
      in f v1
      end

fun main p
    = eval (p, env_base)
```

4.3 Closure conversion

We defunctionalize the function space in the data type of values. There are two function constructors:

– one in the denotation of lambda-abstractions, which we represent by a closure, pairing the code of lambda-abstractions together with their lexical environment, and
– one in the initial environment, which we represent by a specialized constructor `SUCC`.

We splice these two constructors in the data type of values:

```
datatype value = NUM of int
               | CLO of ide * term * value Env.env
               | SUCC
```

Closures are produced when interpreting lambda-abstractions, and the successor function is produced in the initial environment. Both are consumed when interpreting applications.

```
val env_base = Env.extend ("succ", SUCC, Env.empty)

(*  eval : term * value Env.env -> value  *)
fun eval (LIT i, e)
    = NUM i
  | eval (VAR x, e)
    = Env.lookup (x, e)
  | eval (LAM (x, t), e)
    = CLO (x, t, e)
  | eval (APP (t0, t1), e)
    = let val v0 = eval (t0, e)
          val v1 = eval (t1, e)
      in case v0
           of (CLO (x, t, e))
              => eval (t, Env.extend (x, v1, e))
            | SUCC
              => let val (NUM i) = v1
                 in NUM (i + 1)
                 end
      end

fun main p
    = eval (p, env_base)
```

4.4 CPS transformation

We materialize the control flow of the evaluator using continuations:

```
datatype value = NUM of int
               | CLO of ide * term * value Env.env
               | SUCC

val env_base = Env.extend ("succ", SUCC, Env.empty)

(*  eval : term * value Env.env * (value -> 'a) -> 'a  *)
fun eval (LIT i, e, k)
    = k (NUM i)
  | eval (VAR x, e, k)
    = k (Env.lookup (x, e))
  | eval (LAM (x, t), e, k)
    = k (CLO (x, t, e))
  | eval (APP (t0, t1), e, k)
    = eval (t0, e, fn v0 =>
        eval (t1, e, fn v1 =>
          (case v0
             of (CLO (x, t, e))
                => eval (t, Env.extend (x, v1, e), k)
              | SUCC
                => let val (NUM i) = v1
                   in k (NUM (i + 1))
                   end))

fun main p
    = eval (p, env_base, fn v => v)
```

The same evaluator is obtained by inlining the components of the continuation monad in the monadic evaluator of Section 2 and closure-converting the result.

4.5 Defunctionalization

We defunctionalize the function space of continuations. There are three function constructors:

- one in the initial continuation, which we represent by a constructor `STOP`, and
- two in the interpretation of applications, one with `t1`, `e`, and `k` as free variables, and one with `v0` and `k` as free variables.

We represent the function space of continuations with a data type with three constructors and an apply function interpreting these constructors. As already noted elsewhere [11, 12], the data type of defunctionalized continuations coincides with the data type of evaluation contexts for the source language [15, 16].

```
datatype value = NUM of int
               | CLO of ide * term * value Env.env
               | SUCC

datatype cont = STOP
              | ARG of term * value Env.env * cont
              | FUN of value * cont

val env_base = Env.extend ("succ", SUCC, Env.empty)

(*  eval : term * value Env.env * cont -> value  *)
fun eval (LIT i, e, k)
    = apply_cont (k, NUM i)
  | eval (VAR x, e, k)
    = apply_cont (k, Env.lookup (x, e))
  | eval (LAM (x, t), e, k)
    = apply_cont (k, CLO (x, t, e))
  | eval (APP (t0, t1), e, k)
    = eval (t0, e, ARG (t1, e, k))

(*  apply_cont : cont * value -> value  *)
and apply_cont (STOP, v)
    = v
  | apply_cont (ARG (t1, e, k), v0)
    = eval (t1, e, FUN (v0, k))
  | apply_cont (FUN (CLO (x, t, e), k), v)
    = eval (t, Env.extend (x, v, e), k)
  | apply_cont (FUN (SUCC, k), NUM i)
    = apply_cont (k, NUM (i + 1))

fun main p
    = eval (p, env_base, STOP)
```

This defunctionalized continuation-passing evaluator is an implementation of the CEK machine extended with literals [16, 18], which we present next.

4.6 The CEK machine

- Source syntax (terms):
  $t ::= i \mid x \mid \lambda x.t \mid t_0\, t_1$
- Expressible values (integers, closures, and predefined functions) and evaluation contexts (i.e., defunctionalized continuations):
  $v ::= i \mid [x, t, e] \mid \mathit{succ}$
  $k ::= \texttt{stop} \mid \texttt{fun}(v, k) \mid \texttt{arg}(t, e, k)$

– Initial transition, transition rules (two kinds), and final transition:

$$
\begin{array}{|rcl|}
\hline
t & \Rightarrow_{init} & \langle t,\, e_{init},\, \texttt{stop} \rangle \\
\hline
\langle i,\, e,\, k \rangle & \Rightarrow_{eval} & \langle k,\, i \rangle \\
\langle x,\, e,\, k \rangle & \Rightarrow_{eval} & \langle k,\, e(x) \rangle \\
\langle \lambda x.t,\, e,\, k \rangle & \Rightarrow_{eval} & \langle k,\, [x,\, t,\, e] \rangle \\
\langle t_0\, t_1,\, e,\, k \rangle & \Rightarrow_{eval} & \langle t_0,\, e,\, \texttt{arg}(t_1, e, k) \rangle \\
\hline
\langle \texttt{arg}(t_1, e, k),\, v \rangle & \Rightarrow_{cont} & \langle t_1,\, e,\, \texttt{fun}(v, k) \rangle \\
\langle \texttt{fun}([x,\, t,\, e], k),\, v \rangle & \Rightarrow_{cont} & \langle t,\, e[x \mapsto v],\, k \rangle \\
\langle \texttt{fun}(succ, k),\, i \rangle & \Rightarrow_{cont} & \langle k,\, i+1 \rangle \\
\hline
\langle \texttt{stop},\, v \rangle & \Rightarrow_{final} & v \\
\hline
\end{array}
$$

where $e_{base} = e_{empty}[\texttt{succ} \mapsto succ]$

$e_{init} = e_{base}$

4.7 Summary and conclusion

We have presented a series of evaluators and one abstract machine that correspond to a call-by-value monadic evaluator and the identity monad. The first evaluator is a traditional, Lisp-like one in direct style. The machine is the CEK machine. The correctness of the evaluators and of the abstract machine is a corollary of the correctness of the original monadic evaluator and of the transformations.

5 From a lifted state monad to an abstract machine

We specify a lifted state monad and inline its components in the monadic evaluator, obtaining an evaluator in state-passing style. Closure converting, CPS-transforming, and defunctionalizing this state-passing evaluator yields a version of the CEK machine with error and state. This monad and this machine form a stepping stone towards stack inspection.

5.1 A lifted state monad

We consider a lifted state monad where the state is, for conciseness, one integer. We equip this monad with three operations for reading and writing the state and for failing:

```
signature LIFTED_STATE_MONAD
= sig
    include MONAD
    type storable
    type state

    val get : storable monad
    val set : storable -> storable monad
    val fail : 'a monad
  end

structure Lifted_State_Monad : LIFTED_STATE_MONAD
= struct
    datatype 'a lift = LIFT of 'a | BOTTOM
    type storable = int
    type state = storable
    type 'a monad = state -> ('a * state) lift

    fun unit a = (fn s => LIFT (a, s))
    fun bind (m, k) = (fn s => case m s
                                 of (LIFT (a, s')) => k a s'
                                  | BOTTOM => BOTTOM)
    val get = (fn s => LIFT (s, s))
    fun set i = (fn s => LIFT (s, i))
    val fail = (fn s => BOTTOM)
  end
```

Proposition 1. *The type constructor above, together with the above definitions of* `unit` *and* `bind`*, satisfies the three monadic laws.*

Proof. By straightforward equational reasoning. (Alternatively, the lifted state monad is a combination of the lifting monad and of a state monad, and is known to be a monad [27].) □

We extend the base environment with three functions `get`, `set`, and `fail`:

```
val env_init
    = Env.extend ("fail", FUN (fn _ => fail),
      Env.extend ("set", FUN (fn (NUM i) => bind (set i, fn i => unit (NUM i))),
      Env.extend ("get", FUN (fn _ => bind (get, fn i => unit (NUM i))),
      env_base)))
```

Evaluation starts with an initial state `state_init : Lifted_State_Monad.state`.

5.2 A CEK machine with error and state

Inlining the components of the lifted state monad in the monadic evaluator of Section 2 and uncurrying the `eval` function and the function space in the data type of expressible values yields a call-by-value evaluator in state-passing style. As in Section 4, we closure-convert, CPS-transform, and defunctionalize this inlined evaluator. The result is a version of the CEK machine with error and state [15]. The source language and evaluation contexts are as in the CEK machine of Section 4.

– Expressible values (integers, closures, and predefined functions) and results:
$$v ::= i \mid [x,\, t,\, e] \mid \mathit{succ} \mid \mathit{get} \mid \mathit{set} \mid \mathit{fail}$$
$$r ::= \mathit{lift}(v, s) \mid \mathit{bottom}$$
– Initial transition, transition rules (two kinds), and final transition:

t	$\Rightarrow_{\mathit{init}}$	$\langle t,\, e_{\mathit{init}},\, s_{\mathit{init}},\, \texttt{stop}\rangle$
$\langle i,\, e,\, s,\, k\rangle$	$\Rightarrow_{\mathit{eval}}$	$\langle k,\, \mathit{lift}(i, s)\rangle$
$\langle x,\, e,\, s,\, k\rangle$	$\Rightarrow_{\mathit{eval}}$	$\langle k,\, \mathit{lift}(e(x), s)\rangle$
$\langle \lambda x.t,\, e,\, s,\, k\rangle$	$\Rightarrow_{\mathit{eval}}$	$\langle k,\, \mathit{lift}([x,\, t,\, e], s)\rangle$
$\langle t_0\, t_1,\, e,\, s,\, k\rangle$	$\Rightarrow_{\mathit{eval}}$	$\langle t_0,\, e,\, s,\, \texttt{arg}(t_1, e, k)\rangle$
$\langle \texttt{arg}(t_1, e, k),\, \mathit{lift}(v, s)\rangle$	$\Rightarrow_{\mathit{cont}}$	$\langle t_1,\, e,\, s,\, \texttt{fun}(v, k)\rangle$
$\langle \texttt{arg}(t_1, e, k),\, \mathit{bottom}\rangle$	$\Rightarrow_{\mathit{cont}}$	$\langle k,\, \mathit{bottom}\rangle$
$\langle \texttt{fun}([x,\, t,\, e], k),\, \mathit{lift}(v, s)\rangle$	$\Rightarrow_{\mathit{cont}}$	$\langle t,\, e[x \mapsto v],\, s,\, k\rangle$
$\langle \texttt{fun}(\mathit{succ}, k),\, \mathit{lift}(i, s)\rangle$	$\Rightarrow_{\mathit{cont}}$	$\langle k,\, \mathit{lift}(i + 1, s)\rangle$
$\langle \texttt{fun}(\mathit{get}, k),\, \mathit{lift}(v, s)\rangle$	$\Rightarrow_{\mathit{cont}}$	$\langle k,\, \mathit{lift}(s, s)\rangle$
$\langle \texttt{fun}(\mathit{set}, k),\, \mathit{lift}(i, s)\rangle$	$\Rightarrow_{\mathit{cont}}$	$\langle k,\, \mathit{lift}(s, i)\rangle$
$\langle \texttt{fun}(\mathit{fail}, k),\, \mathit{lift}(v, s)\rangle$	$\Rightarrow_{\mathit{cont}}$	$\langle k,\, \mathit{bottom}\rangle$
$\langle \texttt{fun}(v, k),\, \mathit{bottom}\rangle$	$\Rightarrow_{\mathit{cont}}$	$\langle k,\, \mathit{bottom}\rangle$
$\langle \texttt{stop},\, r\rangle$	$\Rightarrow_{\mathit{final}}$	r

where $e_{\mathit{base}} = e_{\mathit{empty}}[\texttt{succ} \mapsto \mathit{succ}]$
$e_{\mathit{init}} = e_{\mathit{base}}[\texttt{get} \mapsto \mathit{get}][\texttt{set} \mapsto \mathit{set}][\texttt{fail} \mapsto \mathit{fail}]$
and s_{init} is the initial state.

In case of failure, the machine propagates *bottom* out of the surrounding evaluation contexts and yields it as the final result. The machine could be optimized by re-classifying the *fail*-transition to be a final transition (i.e., a transition that directly yields *bottom* as the result) and by removing all the *bottom*-propagating transitions. In the corresponding CPS evaluator, this optimization hinges on the type isomorphism between the sum-accepting continuation `((value * state) lift) -> 'a` and the pair of continuations `(value * state -> 'a) * (unit -> 'a)`. This isomorphism enables the optimization from `unit -> 'a` (i.e., a propagating continuation) to `'a` (i.e., an immediate stop). We illustrate this optimization in Appendix A.

5.3 Summary and conclusion

We have presented a lifted state monad and the abstract machine that corresponds to the call-by-value monadic evaluator and this monad. The evaluator obtained by inlining the components of the lifted state monad is in state-passing style. The machine is a version of the CEK machine with state and explicit error propagation. The correctness of the evaluators and of the abstract machine is a corollary of the correctness of the original monadic evaluator and of the transformations.

6 Stack inspection as a lifted state monad

Stack inspection is a security mechanism developed to allow code with different levels of trust to interact in the same execution environment (e.g., the JVM or the CLR) [19]. Before execution, the code is annotated with a subset R of a fixed set of permissions P. For example, trusted code is annotated with all permissions and untrusted code is only annotated with a subset of permissions. Before accessing a restricted resource during execution, the call stack is inspected to test that the required access permissions are available. This test consists of traversing the entire call stack to ensure that the direct caller and all indirect callers all have the required permissions to access the resource. Traversing the entire call stack prevents untrusted code from gaining access to restricted resources by (indirectly) calling trusted code. Trusted code can prevent inspection of its callers for some permissions by explicitly granting those permissions. Trusted code can only grant permissions with which it has been annotated.

Because the entire call stack has to be inspected before accessing resources, the stack-inspection mechanism seems to be incompatible with global tail-call optimization. However, Clements and Felleisen have shown that this is not true and that stack inspection is in fact compatible with global tail-call optimization [7]. Their observation is that the security information of multiple tail calls can be summarized in a permission table. If each stack frame contains a permission table, stack frames do not need to be allocated for tail-calls—the permission table of the current stack frame can be updated instead. This tail-recursive semantics for stack inspection is similar to tail-call optimization in (dynamically scoped) Lisp [29]. It is presented in the form of a CESK machine, the CM machine, and Clements and Felleisen have proved that this machine uses as much space as Clinger's tail-call optimized CESK machine [8]. In the CM machine, the call stack is represented as CEK evaluation contexts enriched with a permission table.

The language of the CM machine is the λ-calculus extended with four constructs:

1. $R[t]$, to annotate a term t with a set of permissions R. When executed, the permissions available are restricted to the permissions in R by making the complement $\overline{R} = P \setminus R$ unavailable; t is then executed with the updated permissions.
2. `grant` R `in` t, to grant a set of permissions R during the evaluation of a term t. When executed, the permissions R are made available, and t is executed with the updated permissions.
3. `test` R `then` t_0 `else` t_1, to branch depending on whether a set of permissions R is available. When executed, the call stack is inspected using a predicate called $\mathcal{OK}$, and t_0 is executed if the permissions are available; otherwise t_1 is executed.
4. `fail`, to fail due to a security error. When executed, the evaluation is terminated with a security error (and therefore the machine is optimized as described in Appendix A).

Our starting point is a simplified version of Clements and Felleisen's CM machine. Their machine includes a heap and a garbage-collection rule to make it possible to extend Clinger's space-complexity analysis to the CM machine. For simplicity, we leave out the heap and the garbage-collection rule from the machine, and, without loss of generality (because the source language is untyped), we omit recursive functions from the source language. Clements and Felleisen's source language does not have literals; for simplicity, we do likewise and we omit literals and the successor function from the source language. Our focus is the basic stack-inspection mechanism and the features we have omitted from the CM machine do not interfere with this basic mechanism.

– Permissions and permission tables for a fixed set of permissions P:

$$\begin{aligned} &R \subseteq P \\ &m \in P \to \{grant, no\} \end{aligned}$$

– Source syntax (terms):

$$t ::= x \mid \lambda x.t \mid t_0\, t_1 \mid R[t] \mid \texttt{grant}\ R\ \texttt{in}\ t \mid \texttt{test}\ R\ \texttt{then}\ t_0\ \texttt{else}\ t_1 \mid \texttt{fail}$$

– Expressible values (closures), outcomes, and evaluation contexts:

$$v ::= [x,\, t,\, e]$$

$$o ::= v \mid \mathit{fail}$$

$$k ::= \texttt{stop}(m) \mid \texttt{arg}(t, e, k, m) \mid \texttt{fun}(v, k, m)$$

– Initial transition, transition rules (two kinds), and final transitions:

$$\begin{array}{|rl|}
\hline
t & \Rightarrow_{\mathit{init}} \langle t,\, e_{\mathit{empty}},\, \texttt{stop}(m_{\mathit{empty}})\rangle \\
\hline
\langle x,\, e,\, k\rangle & \Rightarrow_{\mathit{eval}} \langle k,\, e(x)\rangle \\
\langle \lambda x.t,\, e,\, k\rangle & \Rightarrow_{\mathit{eval}} \langle k,\, [x,\, t,\, e]\rangle \\
\langle t_0\, t_1,\, e,\, k\rangle & \Rightarrow_{\mathit{eval}} \langle t_0,\, e,\, \texttt{arg}(t_1, e, k, m_{\mathit{empty}})\rangle \\
\langle R[t],\, e,\, k\rangle & \Rightarrow_{\mathit{eval}} \langle t,\, e,\, k[\overline{R} \mapsto \mathit{no}]\rangle \\
\langle \texttt{grant}\ R\ \texttt{in}\ t,\, e,\, k\rangle & \Rightarrow_{\mathit{eval}} \langle t,\, e,\, k[R \mapsto \mathit{grant}]\rangle \\
\langle \texttt{test}\ R\ \texttt{then}\ t_0\ \texttt{else}\ t_1,\, e,\, k\rangle & \Rightarrow_{\mathit{eval}} \langle t_0,\, e,\, k\rangle\ \mathit{if}\ \mathcal{OK}[R][k] \\
\langle \texttt{test}\ R\ \texttt{then}\ t_0\ \texttt{else}\ t_1,\, e,\, k\rangle & \Rightarrow_{\mathit{eval}} \langle t_1,\, e,\, k\rangle\ \mathit{otherwise} \\
\hline
\langle \texttt{fail},\, e,\, k\rangle & \Rightarrow_{\mathit{final}} \mathit{fail} \\
\hline
\langle \texttt{arg}(t, e, k, m),\, v\rangle & \Rightarrow_{\mathit{cont}} \langle t,\, e,\, \texttt{fun}(v, k, m_{\mathit{empty}})\rangle \\
\langle \texttt{fun}([x,\, t,\, e], k, m),\, v\rangle & \Rightarrow_{\mathit{cont}} \langle t,\, e[x \mapsto v],\, k\rangle \\
\hline
\langle \texttt{stop}(m),\, v\rangle & \Rightarrow_{\mathit{final}} v \\
\hline
\end{array}$$

where m_{empty} denotes the empty permission table,

$$\begin{aligned}
\texttt{stop}(m)[R \mapsto c] &= \texttt{stop}(m[R \mapsto c]) \\
\texttt{arg}(t, e, k, m)[R \mapsto c] &= \texttt{arg}(t, e, k, m[R \mapsto c]) \\
\texttt{fun}(v, k, m)[R \mapsto c] &= \texttt{fun}(v, k, m[R \mapsto c])
\end{aligned}$$

and

$$\begin{aligned}
\mathcal{OK}[\emptyset][k] &= \text{true} \\
\mathcal{OK}[R][\texttt{stop}(m)] &= R \cap m^{-1}(\mathit{no}) = \emptyset \\
\left.\begin{array}{r}\mathcal{OK}[R][\texttt{arg}(t, e, k, m)] \\ \mathcal{OK}[R][\texttt{fun}(t, k, m)]\end{array}\right\} &= (R \cap m^{-1}(\mathit{no}) = \emptyset) \wedge \mathcal{OK}[R \setminus m^{-1}(\mathit{grant})][k]
\end{aligned}$$

In the CM machine, evaluation contexts are CEK evaluation contexts enriched with permission tables. They are therefore a zipped version of the CEK evaluation contexts and a stack of permission tables. We unzip the CM evaluation contexts into CEK evaluation contexts and a stack of permission tables. Permissions, permission tables, source syntax, expressible values, and outcomes remain the same as in the original CM machine. The $\mathcal{OK}$ predicate is changed to inspect the stack of permission tables instead of the evaluation contexts:

– Evaluation contexts:

$$k ::= \texttt{stop} \mid \texttt{arg}(t, e, k) \mid \texttt{fun}(v, k)$$

– Initial transition, transition rules (two kinds), and final transitions:

$$\begin{array}{|rl|}
\hline
t & \Rightarrow_{\mathit{init}} \langle t,\, e_{\mathit{empty}},\, m_{\mathit{empty}} :: \mathit{nil},\, \texttt{stop}\rangle \\
\hline
\langle x,\, e,\, ms,\, k\rangle & \Rightarrow_{\mathit{eval}} \langle k,\, ms,\, e(x)\rangle \\
\langle \lambda x.t,\, e,\, ms,\, k\rangle & \Rightarrow_{\mathit{eval}} \langle k,\, ms,\, [x,\, t,\, e]\rangle \\
\langle t_0\, t_1,\, e,\, ms,\, k\rangle & \Rightarrow_{\mathit{eval}} \langle t_0,\, e,\, m_{\mathit{empty}} :: ms,\, \texttt{arg}(t_1, e, k)\rangle \\
\langle R[t],\, e,\, m :: ms,\, k\rangle & \Rightarrow_{\mathit{eval}} \langle t,\, e,\, m[\overline{R} \mapsto \mathit{no}] :: ms,\, k\rangle \\
\langle \texttt{grant}\ R\ \texttt{in}\ t,\, e,\, m :: ms,\, k\rangle & \Rightarrow_{\mathit{eval}} \langle t,\, e,\, m[R \mapsto \mathit{grant}] :: ms,\, k\rangle \\
\langle \texttt{test}\ R\ \texttt{then}\ t_0\ \texttt{else}\ t_1,\, e,\, ms,\, k\rangle & \Rightarrow_{\mathit{eval}} \langle t_0,\, e,\, ms,\, k\rangle\ \mathit{if}\ \mathcal{OK}[R][ms] \\
\langle \texttt{test}\ R\ \texttt{then}\ t_0\ \texttt{else}\ t_1,\, e,\, ms,\, k\rangle & \Rightarrow_{\mathit{eval}} \langle t_1,\, e,\, ms,\, k\rangle\ \mathit{otherwise} \\
\hline
\langle \texttt{fail},\, e,\, ms,\, k\rangle & \Rightarrow_{\mathit{final}} \mathit{fail} \\
\hline
\langle \texttt{arg}(t, e, k),\, m :: ms,\, v\rangle & \Rightarrow_{\mathit{cont}} \langle t,\, e,\, m_{\mathit{empty}} :: ms,\, \texttt{fun}(v, k)\rangle \\
\langle \texttt{fun}([x,\, t,\, e], k),\, m :: ms,\, v\rangle & \Rightarrow_{\mathit{cont}} \langle t,\, e[x \mapsto v],\, ms,\, k\rangle \\
\hline
\langle \texttt{stop},\, ms,\, v\rangle & \Rightarrow_{\mathit{final}} v \\
\hline
\end{array}$$

where

$$\begin{aligned}\mathcal{OK}[\emptyset][ms] &= \text{true}\\ \mathcal{OK}[R][nil] &= \text{true}\\ \mathcal{OK}[R][m :: ms] &= (R \cap m^{-1}(no) = \emptyset) \wedge \mathcal{OK}[R \setminus m^{-1}(grant)][ms]\end{aligned}$$

As we have already observed in previous work [2,6,9,11,12], the evaluation contexts, together with the *cont* transition function, are the defunctionalized counterpart of a continuation. We can therefore "refunctionalize" this continuation and then write the evaluator in direct style. The resulting evaluator threads a state—the stack of permission tables—and evaluation can fail. The evaluator can therefore be expressed as an instance of the monadic evaluator with a lifted state monad.

In the lifted state monad for stack inspection, the storable values are permission tables, and the state is a stack of storable values. The operations on the permission tables are expressed as the monadic operations `push_empty`, `pop_top`, `mark_complement_no`, `mark_grant`, and `OK`. Furthermore, the monadic operation `fail` terminates the computation with a security error. The stack-inspection state monad is implemented as a structure with the following signature:

```
signature STACK_INSPECTION_LIFTED_STATE_MONAD
= sig
    include MONAD
    val fail : 'a monad
    val push_empty : unit monad
    val pop_top : unit monad
    val mark_complement_no : permission Set.set -> unit monad
    val mark_grant : permission Set.set -> unit monad
    val OK : permission Set.set -> bool monad
  end
```

where `permission` is a type of permissions and `Set.set` is a polymorphic type of sets.

The definitions of `unit` and `bind` are those of the lifted state monad of Section 5: `fail` implements the security error; `push_empty` pushes an empty permission table on top of the permission-table stack; `pop_top` pops the top permission table off the permission-table stack; `mark_complement_no` updates the topmost permission table by making the complement of the argument set of permissions unavailable; `mark_grant` updates the topmost permission table by making the argument set of permissions available; and `OK` inspects the permission stack to test whether the argument permissions are available.

The source language is represented as an ML datatype:

```
datatype term = VAR of ide
              | LAM of ide * term
              | APP of term * term
              | FRAME of permission Set.set * term
              | GRANT of permission Set.set * term
              | TEST of permission Set.set * term * term
              | FAIL
```

The monadic evaluator corresponding to the unzipped version of the CM machine reads as follows:

```
datatype value = FUN of value -> value monad

(*  eval : term * value Env.env -> value monad  *)
fun eval (LAM (x, t), e)
    = unit (FUN (fn v => eval (t, Env.extend (x, v, e))))
  | eval (VAR x, e)
    = unit (Env.lookup (e, x))
```

```
| eval (APP (t0, t1), e)
  = bind (push_empty, fn () =>
      bind (eval (t0, e), fn v0 =>
        bind (pop_top, fn () =>
          bind (push_empty, fn () =>
            bind (eval (t1, e), fn v1 =>
              bind (pop_top, fn () => let val (FUN f) = v0
                                      in f v1
                                      end))))))
| eval (FRAME (R, t), e)
  = bind (mark_complement_no R, fn () => eval (t, e))
| eval (GRANT (R, t), e)
  = bind (mark_grant R, fn () => eval (t, e))
| eval (TEST (R, t0, t1), e)
  = bind (OK R, fn b => if b then eval (t0, e) else eval (t1, e))
| eval (FAIL, e)
  = fail
```

The process is reversible. Starting from this lifted state monad where the state is a stack of permission tables and this monadic evaluator, it is a simple exercise to reconstruct the unzipped CM machine by inlining the monad, closure converting the expressible values, CPS-transforming the evaluator, optimizing the continuation as illustrated in Appendix A to stop immediately in case of failure, and defunctionalizing the resulting continuations. In addition, we are now in position to combine properly tail-recursive stack inspection with other effects by combining the stack-inspection monad with other monads at the monadic level. Inlining such combined monads lets us derive abstract machines with properly tail-recursive stack inspection and other computational effects [3].

To summarize, we have shown that Clements and Felleisen's properly tail-recursive stack inspection can be expressed as a lifted state monad. Constructing abstract machines for a language with stack inspection and other effects expressed as monads therefore reduces to designing the desired combination of the monads and then mechanically deriving the corresponding abstract machine. The correctness of this abstract machine is a corollary of the correctness of the original monadic evaluator and of the transformations.

7 A dedicated monad for stack inspection

We observe that the lifted state monad is overly general to characterize the computational behaviour of stack inspection:

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

This type would also fit if all permissions in the stack were updatable. However, that is not the case—only the top permission table can be modified, and the other permission tables in the stack are read-only.

Instead, we can cache the top permission table and make it both readable and writable while keeping the rest of the stack read only. The corresponding type constructor reads as follows:

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

Proposition 2. *The type constructor above, together with the following definitions of* `unit` *and* `bind`*, satisfies the three monadic laws.*

```
fun unit a = (fn (p, pl) => LIFT (a, p))
fun bind (m, k) = (fn (p, pl) => case m (p, pl)
                                   of (LIFT (a, p'))
                                      => k a (p', pl)
                                    | BOTTOM
                                      => BOTTOM)
```

Proof. By straightforward equational reasoning. □

This monad provides a more accurate characterization of stack inspection than the one in Section 6.

As an exercise, we have constructed the corresponding abstract machine which is similar to the one in Section 6.

8 Related work

Stack inspection is used as a fine-grained access control mechanism for Java [21]. It allows code with different levels of trust to safely interact in the same execution environment. Before access to a restricted resource is allowed, the entire call stack is inspected to test that the required permissions are available. Wallach, Appel, and Felten present a semantics for stack inspection based on a belief logic [34]. Their semantics is not tied to inspecting stack frames, and it is thus compatible with tail-call optimization. Their implementation, called security-passing style, allows them to implement stack inspection for Java without changing the JVM. Instead, they perform a global byte-code rewriting before loading. Fournet and Gordon develop a formal semantics and an equational theory for a λ-calculus model of stack inspection [19]. They use this equational theory to formally investigate how stack inspection affects known program transformations such as inlining and tail-call optimization. Clements and Felleisen present a properly tail-call optimized semantics for stack inspection based on Fournet and Gordon's semantics [7]. This tail-call optimized semantics is given in the form of a CESK machine, which was the starting point for our work.

Since Moggi's breakthrough [27], monads have been widely used to parameterize functional programs with effects [5, 33]. We are not aware, though, of the use of monads in connection with abstract machines for computational effects.

For several decades abstract machines have been an active area of research, ranging from Landin's classical SECD machine [24] to the modern JVM [25]. As observed by Diehl, Hartel, and Sestoft [14], research on abstract machines has chiefly focused on developing new machines and proving them correct. The thrust of our work is to explore a correspondence between interpreters and abstract machines [2, 4, 6, 9] that takes its roots in Reynolds seminal work on definitional interpreters [30].

The present work is a next step of our study of this correspondence. Essentially the same correspondence has been put to use by Graunke, Findler, Krishnamurthi, and Felleisen to transform functional programs into abstract machines for programming the web [22]. The only difference is that Graunke, Findler, Krishnamurthi, and Felleisen use lambda-lifting instead of closure conversion. They do not need closure conversion because they do not consider evaluators for higher-order programming languages, and we do not need lambda-lifting because our evaluators are already lambda-lifted [13, 23].

9 Conclusion

We have extended our correspondence between evaluators and abstract machines from the pure setting of the λ-calculus to the impure setting of the computational λ-calculus. Throughout, we have advocated that a viable alternative to designing abstract machines for languages with computational effects and then proving their correctness is to start from a monadic evaluator and a computational monad and to derive the corresponding abstract machine. We have illustrated this alternative with standard monads as well as with new monads. The identity monad leads us to the classical CEK machine. The lifted state monad leads us to a variant of the CEK machine with error and state. We have also characterized Clements and Felleisen's properly tail-recursive stack inspection as a lifted state monad which enables us to combine it with other monads and construct new abstract machines with properly tail-recursive stack inspection and other computational effects.

The contributions of this article are therefore as follows:

- a systematic construction of abstract machines for languages with computational effects from a monadic evaluator and a computational monad;
- concrete examples of the construction for the identity monad and a lifted state monad;
- a characterization of Clements and Felleisen's properly tail-recursive stack inspection as a lifted state monad and a reconstruction of the CM machine; and
- a dedicated monad for properly tail-recursive stack inspection and the construction of a corresponding abstract machine.

Constructing abstract machines for languages with effects is known to be a challenge, one that is handled on a case-by-case basis. Our correspondence between evaluators and abstract machines provides a methodology to construct abstract machines for languages with effects. In addition, our characterization of stack inspection as a monad makes it possible to combine stack inspection with computational effects at a monadic level. We are therefore in position to construct, e.g., a variant of Krivine's machine with stack inspection as well as variants of the Categorical Abstract Machine and of the SECD machine with arbitrary computational effects expressed as monads.

Acknowledgments: We are grateful to the anonymous referees for their comments. This work is partially supported by the ESPRIT Working Group APPSEM II (`http://www.appsem.org`) and by the Danish Natural Science Research Council, Grant no. 21-03-0545. A preliminary version of this article is available as the BRICS technical report RS-03-35 [3].

A Propagating vs. stopping

This appendix illustrates the optimization of propagating a final result out of surrounding evaluation contexts vs. not continuing the computation and returning the final result. We consider the traditional example of multiplying the leaves of a tree of integers:

```
datatype bt = LEAF of int
            | NODE of bt * bt
```

We want to take advantage of the fact that 0 is an absorbant element for multiplication. To this end, we lift the intermediate results of the auxiliary function that traverses the input tree:

```
datatype int_lifted = ZERO
                    | NOT_ZERO of int

(*  mult_ds : bt -> int  *) (*  visit : bt -> int_lifted  *)
fun mult_ds t
    = let fun visit (LEAF 0) = ZERO
            | visit (LEAF n) = NOT_ZERO n
            | visit (NODE (t1, t2))
              = (case visit t1
                   of ZERO => ZERO
                    | (NOT_ZERO n1) => (case visit t2
                                          of ZERO => ZERO
                                           | (NOT_ZERO n2) => NOT_ZERO (n1 * n2)))
      in case visit t
           of ZERO => 0
            | (NOT_ZERO n) => n
      end
```

If a 0 leaf is encountered during the recursive descent, `ZERO` is propagated out until the top-level case expression.

Let us write `visit` in continuation-passing style:

```
(*  mult_cps : bt -> int  *) (*  visit : bt * (int_lifted -> int) -> int  *)
fun mult_cps t
    = let fun visit (LEAF 0, k) = k ZERO
            | visit (LEAF n, k) = k (NOT_ZERO n)
            | visit (NODE (t1, t2), k)
              = visit (t1, fn ZERO => k ZERO
                            | (NOT_ZERO n1)
                              => visit (t2, fn ZERO => k ZERO
                                             | (NOT_ZERO n2)
                                               => NOT_ZERO (n1 * n2)))
      in visit (t, fn ZERO => 0
                    | (NOT_ZERO n) => n)
      end
```

The same propagation takes place. To optimize it, we use the type isomorphism between the sum-accepting continuation `int_lifted -> int` and the pair of continuations `(unit -> int) * (int -> int)`, one for propagating the final result and one to continue the computation, and we simplify the propagating continuation away:

```
(*  mult_cps_opt : bt -> int  *) (*  visit : bt * (int -> int) -> int  *)
fun mult_cps_opt t
    = let fun visit (LEAF 0, k) = 0
            | visit (LEAF n, k) = n
            | visit (NODE (t1, t2), k)
              = visit (t1, fn n1 => visit (t2, fn n2 => k (n1 * n2)))
      in visit (t, fn n => n)
      end
```

In the optimized version, the continuation is only applied to non-zero intermediate results, and as soon as a 0 leaf is encountered, the computation stops.

References

1. Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From interpreter to compiler and virtual machine: a functional derivation. Technical Report BRICS RS-03-14, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, March 2003.
2. Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19. ACM Press, August 2003.
3. Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. Technical Report BRICS RS-03-35, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2003.
4. Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. Technical Report BRICS RS-04-03, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, February 2004. Extended version of an article to appear in Information Processing Letters.
5. Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics – Advanced Lectures*, number 2395 in Lecture Notes in Computer Science, pages 42–122, Caminha, Portugal, September 2000. Springer-Verlag.
6. Dariusz Biernacki and Olivier Danvy. From interpreter to logic engine by defunctionalization. Technical Report BRICS RS-04-5, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, March 2004. To appear in the proceedings of the 2003 International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2003).
7. John Clements and Matthias Felleisen. A tail-recursive semantics for stack inspections. In Pierpaolo Degano, editor, *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003*, number 2618 in Lecture Notes in Computer Science, pages 22–37, Warsaw, Poland, April 2003. Springer-Verlag.
8. William D. Clinger. Proper tail recursion and space efficiency. In Keith D. Cooper, editor, *Proceedings of the ACM SIGPLAN'98 Conference on Programming Languages Design and Implementation*, pages 174–185, Montréal, Canada, June 1998. ACM Press.
9. Olivier Danvy. A rational deconstruction of Landin's SECD machine. Technical Report BRICS RS-03-33, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, October 2003.
10. Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.
11. Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press.
12. Olivier Danvy and Lasse R. Nielsen. Syntactic theories in practice. In Mark van den Brand and Rakesh M. Verma, editors, *Informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001)*, volume 59.4 of *Electronic Notes in Theoretical Computer Science*, Firenze, Italy, September 2001. Extended version available as the technical report BRICS RS-02-04.

13. Olivier Danvy and Ulrik P. Schultz. Lambda-lifting in quadratic time. In Zhenjiang Hu and Mario Rodriguez-Artalejo, editors, *Sixth International Symposium on Functional and Logic Programming*, number 2441 in Lecture Notes in Computer Science, pages 134–151, Aizu, Japan, September 2002. Springer-Verlag. Extended version available as the technical report BRICS-RS-03-26.
14. Stephan Diehl, Pieter Hartel, and Peter Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16:739–751, 2000.
15. Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Unpublished lecture notes. `http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html`, 1989-2003.
16. Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986.
17. Andrzej Filinski. Representing monads. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon, January 1994. ACM Press.
18. Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In David W. Wall, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 28, No 6, pages 237–247, Albuquerque, New Mexico, June 1993. ACM Press.
19. Cédric Fournet and Andrew D. Gordon. Stack inspection: Theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, May 2003.
20. Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages, second edition.* The MIT Press, 2001.
21. Li Gong and Roland Schemers. Implementing protection domains in Java Development Kit 1.2. In *Proceedings of the Internet Symposium on Network and Distributed System Security*, San Diego, California, March 1998.
22. Paul T. Graunke, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Automatically restructuring programs for the web. In Martin S. Feather and Michael Goedicke, editors, *16th IEEE International Conference on Automated Software Engineering (ASE 2001)*, pages 211–222, Coronado Island, San Diego, California, USA, November 2001. IEEE Computer Society.
23. Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 190–203, Nancy, France, September 1985. Springer-Verlag.
24. Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.
25. Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specification.* Addison-Wesley, 2nd edition, 1999.
26. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.
27. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.
28. Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
29. Christian Queinnec. *Lisp in Small Pieces.* Cambridge University Press, Cambridge, 1996.
30. John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword.
31. Guy L. Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978. Technical report AI-TR-474.
32. Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1/2):1–49, 2000.
33. Philip Wadler. The essence of functional programming (invited talk). In Andrew W. Appel, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992. ACM Press.
34. Dan S. Wallach, Andrew W. Appel, and Edward W. Felten. SAFKASI: a security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4):341–378, 2000.