Basic Research in Computer Science

BRICS RS-03-35 Ager et al.: A Functional Correspondence between Monadic Evaluators and Abstract Machines

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

Mads Sig Ager
Olivier Danvy
Jan Midtgaard

BRICS Report Series **RS-03-35**

ISSN 0909-0878 **November 2003**

**See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:**

**BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark**

**Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk**

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/03/35/`

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†

November 2003

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. Specifically, we show how to derive new abstract machines from monadic evaluators for the computational λ-calculus. Starting from a monadic evaluator and a given monad, we inline the components of the monad in the evaluator and we derive the corresponding abstract machine by closure-converting, CPS-transforming, and defunctionalizing this inlined interpreter. We illustrate the construction first with the identity monad, obtaining yet again the CEK machine, and then with a state monad, an exception monad, and a combination of both.

In addition, we characterize the tail-recursive stack inspection presented by Clements and Felleisen at ESOP 2003 as a canonical state monad. Combining this state monad with an exception monad, we construct an abstract machine for a language with exceptions and properly tail-recursive stack inspection. The construction scales to other monads—including one more properly dedicated to stack inspection than the 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.

*Basic Research in Computer Science (`www.brics.dk`),
funded by the Danish National Research Foundation.
†Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark.
Email: {`mads,danvy,jmi`}`@brics.dk`

Contents

1 Introduction

Diehl, Hartel, and Sestoft's overview of abstract machines for programming-language implementation [12] concluded on the need to develop a theory of abstract machines. In previous work [2,5,8], 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 is constructive. Starting from a compositional evaluator, we:

1. closure-convert its expressible and denotable values [20, 29];
2. materialize its control flow into continuations by CPS transformation [9, 25, 28]; and
3. defunctionalize these continuations [10, 27].

For all its simplicity, this correspondence has let us derive Krivine's machine from a call-by-name evaluator and Felleisen et al.'s CEK machine from a call-by-value evaluator [2], as well as generalizations of Krivine's machine and of the CEK machine from normalization functions [1]. It has also let us reveal the evaluator underlying Landin's SECD machine, Schmidt's VEC machine, Hannan and Miller's CLS machine, and Curien et al.'s Categorical Abstract Machine [2,8]. We have also verified that the correspondence holds for call-by-need evaluators and lazy abstract machines [3], logic programming [5], imperative programming, and object-oriented programming, including Featherweight Java and a subset of Smalltalk. 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 [4]. We consider a canonical evaluator parameterized by a monad (Section 2). We then successively consider five monads: the identity monad, the state monad, the exception monad, and the two possible combinations of the state monad and of the exception monad (Sections 3 to 6). For each of these monads, we specify it and we inline it in the monadic evaluator, obtaining an evaluator dedicated to this computational effect. We then construct the corresponding abstract machine by closure-converting, CPS-transforming, and defunctionalizing this dedicated evaluator.

We then turn to the security technique of 'stack inspection' [18]. At ESOP 2003 [6], Clements and Felleisen debunked the myth that stack inspection is incompatible with proper tail recursion. To this end, they presented a CESK abstract machine implementing stack inspection in a properly tail-recursive way. We characterize Clements and Felleisen's stack inspection as a state monad (Section 7). We also combine the stack-inspection state monad with the exception monad and construct the corresponding abstract machine, which would be nontrivial to construct from scratch (Section 8). We then present a monad that accounts for stack inspection more precisely than the canonical state monad, review related work, and conclude.

2 A call-by-value monadic evaluator

As traditional [4, 15, 30], we specify a monad as a type constructor and two polymorphic functions. (We use Standard ML [23].)

```
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`. We implicitly use ML's built-in lifting monad to account for non-termination and ML's built-in error monad to account for type mismatch.

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.

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)
```

Applying the successor function to a function instead of to an integer yields an ML pattern-matching error.

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`.

3 From the identity monad to an abstract machine

We first specify the identity monad and inline it 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 [14, 17].

3.1 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
```

3.2 Inlining the monad in the monadic evaluator

Inlining this 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)
```

3.3 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)
```

3.4 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 continuation monad in the monadic evaluator of Section 2 and closure-converting the result.

3.5 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 [10, 11], the data type of defunctionalized continuations coincides with the data type of evaluation contexts for the source language [13, 14].

```
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 [14, 17], which we present next.

3.6 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):

$$\begin{array}{lcl} v & ::= & i \mid [x,\, t,\, e] \mid \mathit{succ} \\ k & ::= & \texttt{stop} \mid \texttt{fun}(v, k) \mid \texttt{arg}(t, e, k) \end{array}$$

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

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

where
$$\begin{array}{lcl} e_{\mathit{base}} & = & e_{\mathit{empty}}[\texttt{succ} \mapsto \mathit{succ}] \\ e_{\mathit{init}} & = & e_{\mathit{base}} \end{array}$$

3.7 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 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.

4 From a state monad to an abstract machine

We specify a state monad and inline it in the monadic evaluator, obtaining an evaluator in state-passing style. Closure converting, CPS-transforming, and defunctionalizing this state-passing evaluator yields a CEK machine with state.

4.1 A state monad

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

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

    val get : storable monad
    val set : storable -> storable monad
  end

structure State_Monad : STATE_MONAD
= struct
    type storable = int
    type state = storable
    type 'a monad = state -> 'a * state

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

We extend the base environment with two functions `get` and `set`:

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

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

4.2 Inlining the monad in the monadic evaluator

Inlining this 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:

```
type storable = int

type state = storable

datatype value = NUM of int
               | FUN of value * state -> value * state

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

val env_init = Env.extend ("get",
                           FUN (fn (_, s) => (NUM s, s)),
                 Env.extend ("set",
                             FUN (fn (NUM i, s) => (NUM s, i)),
                   env_base))

(*  eval : term * value Env.env * state -> value * state  *)
fun eval (LIT i, e, s)
    = (NUM i, s)
  | eval (VAR x, e, s)
    = (Env.lookup (x, e), s)
  | eval (LAM (x, t), e, s)
    = (FUN (fn (v, s) => eval (t, Env.extend (x, v, e), s)), s)
  | eval (APP (t0, t1), e, s)
    = let val (v0, s') = eval (t0, e, s)
          val (v1, s'') = eval (t1, e, s')
          val (FUN f) = v0
      in f (v1, s'')
      end

fun main p
    = (fn state_init => eval (p, env_init, state_init))
```

4.3 A CEK machine with state

As in Section 3, we closure-convert, CPS-transform, and defunctionalize the inlined evaluator of Section 4.2. The result is a CEK machine with state [13]. The source language and evaluation contexts are as in the CEK machine of Section 3.

- Expressible values (integers, closures, and predefined functions) and results:

$$\begin{array}{lcl} v & ::= & i \mid [x, t, e] \mid \mathit{succ} \mid \mathit{get} \mid \mathit{set} \\ r & ::= & (v, s) \end{array}$$

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

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

where
$$e_{base} = e_{empty}[\mathtt{succ} \mapsto succ]$$
$$e_{init} = e_{base}[\mathtt{get} \mapsto get][\mathtt{set} \mapsto set]$$
and s_{init} is the initial state (e.g., -1).

4.4 Summary and conclusion

We have presented a series of evaluators and one abstract machine that correspond to a call-by-value monadic evaluator and a state monad. The first evaluator is a traditional one in state-passing style. The machine is a CEK machine with state. 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 an exception monad to an abstract machine

We specify an exception monad and inline it in the monadic evaluator, obtaining an exception-oriented evaluator. We closure convert, CPS-transform, and defunctionalize this exception-oriented evaluator and obtain a CEK machine with exceptions. We then consider an alternative implementation of exceptions.

5.1 An exception monad

We consider an exception monad where, for conciseness, there is only one kind of exception and it carries no values. We equip this monad with two operations for raising and handling exceptions:

```
signature EXCEPTION_MONAD
= sig
    include MONAD
    datatype 'a E = RES of 'a | EXC
```

```
    val raise_exception : 'a monad
    val handle_exception : 'a monad * (unit -> 'a monad) -> 'a monad
  end

structure Exception_Monad : EXCEPTION_MONAD
= struct
    datatype 'a E = RES of 'a | EXC
    type 'a monad = 'a E

    fun unit a
        = RES a
    fun bind (RES a, k)
        = k a
      | bind (EXC, k)
        = EXC

    val raise_exception = EXC
    fun handle_exception (RES a, h)
        = RES a
      | handle_exception (EXC, h)
        = h ()
  end
```

We extend the source language with a special form to handle an exception (and the monadic evaluator with a branch for evaluating this special form), and we extend the base environment with a function to raise an exception:

```
datatype term = ...
              | HANDLE of term * term

fun eval ...
  | eval (HANDLE (t0, t1), e)
    = Exception_Monad.handle_exception (eval (t0, e),
                                        fn () => eval (t1, e))

val env_init
    = Env.extend ("raise",
                  FUN (fn _ => Exception_Monad.raise_exception),
                  env_base)
```

5.2 Inlining the monad in the monadic evaluator

Inlining this exception monad in the extended monadic evaluator yields a call-by-value evaluator in exception-oriented style:

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

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

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

val env_init = Env.extend ("raise", FUN (fn _ => EXC), env_base)

(*  eval : term * value Env.env -> value E  *)
fun eval (LIT i, e)
    = RES (NUM i)
  | eval (VAR x, e)
    = RES (Env.lookup (x, e))
  | eval (LAM (x, t), e)
    = RES (FUN (fn v => eval (t, Env.extend (x, v, e))))
  | eval (APP (t0, t1), e)
    = (case eval (t0, e)
         of (RES v0) => (case eval (t1, e)
                           of (RES v1) => let val (FUN f) = v0
                                          in f v1
                                          end
                            | EXC => EXC)
          | EXC => EXC)
  | eval (HANDLE (t, h), e)
    = (case eval (t, e)
         of (RES a) => RES a
          | EXC => eval (h, e))

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

5.3 A CEK machine with exceptions

As in Section 3 we closure-convert, CPS-transform, and defunctionalize the inlined evaluator of Section 5.2. The result is a version of the CEK machine with exceptions:

- Source syntax (terms):

$$t \quad ::= \quad i \mid x \mid \lambda x.t \mid t_0\, t_1 \mid t_0 \,\mathtt{handle}\, t_1$$

- Expressible values (integers, closures, and predefined functions), results, and evaluation contexts:

$$\begin{array}{lcl} v & ::= & i \mid [x, t, e] \mid \mathit{succ} \mid \mathit{raise} \\ r & ::= & \mathit{res}(v) \mid \mathit{exc} \\ k & ::= & \mathtt{stop} \mid \mathtt{fun}(v, k) \mid \mathtt{arg}(t, e, k) \mid \mathtt{exc}(t, e, k) \end{array}$$

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

t	$\Rightarrow_{init}$	$\langle t,\, e_{init},\, \mathtt{stop}\rangle$
$\langle i,\, e,\, k\rangle$	$\Rightarrow_{eval}$	$\langle k,\, res(i)\rangle$
$\langle x,\, e,\, k\rangle$	$\Rightarrow_{eval}$	$\langle k,\, res(e(x))\rangle$
$\langle \lambda x.t,\, e,\, k\rangle$	$\Rightarrow_{eval}$	$\langle k,\, res([x,\, t,\, e])\rangle$
$\langle t_0\, t_1,\, e,\, k\rangle$	$\Rightarrow_{eval}$	$\langle t_0,\, e,\, \mathtt{arg}(t_1, e, k)\rangle$
$\langle t_0\, \mathtt{handle}\, t_1,\, e,\, k\rangle$	$\Rightarrow_{eval}$	$\langle t_0,\, e,\, \mathtt{exc}(t_1, e, k)\rangle$
$\langle \mathtt{arg}(t_1, e, k),\, res(v)\rangle$	$\Rightarrow_{cont}$	$\langle t_1,\, e,\, \mathtt{fun}(v, k)\rangle$
$\langle \mathtt{arg}(t_1, e, k),\, exc\rangle$	$\Rightarrow_{cont}$	$\langle k,\, exc\rangle$
$\langle \mathtt{fun}([x,\, t,\, e], k),\, res(v)\rangle$	$\Rightarrow_{cont}$	$\langle t,\, e[x \mapsto v],\, k\rangle$
$\langle \mathtt{fun}(succ, k),\, res(i)\rangle$	$\Rightarrow_{cont}$	$\langle k,\, res(i+1)\rangle$
$\langle \mathtt{fun}(raise, k),\, res(v)\rangle$	$\Rightarrow_{cont}$	$\langle k,\, exc\rangle$
$\langle \mathtt{fun}(v, k),\, exc\rangle$	$\Rightarrow_{cont}$	$\langle k,\, exc\rangle$
$\langle \mathtt{exc}(t_1, e, k),\, res(v)\rangle$	$\Rightarrow_{cont}$	$\langle k,\, res(v)\rangle$
$\langle \mathtt{exc}(t_1, e, k),\, exc\rangle$	$\Rightarrow_{cont}$	$\langle t_1,\, e,\, k\rangle$
$\langle \mathtt{stop},\, r\rangle$	$\Rightarrow_{final}$	r

where
$$\begin{aligned} e_{base} &= e_{empty}[\mathtt{succ} \mapsto succ] \\ e_{init} &= e_{base}[\mathtt{raise} \mapsto raise] \end{aligned}$$

5.4 An alternative implementation of exceptions

Alternatively, in the continuation-passing evaluator obtained after closure conversion and CPS-transformation, we can exploit the type isomorphism between a sum-expecting continuation and a pair of continuations:

```
value option -> answer  ≅  (value -> answer) * (unit -> answer)
```

The resulting interpreter is equipped with two continuations. The corresponding notion of computation, at the monadic level, also uses two continuations—a normal continuation and a handler continuation. Defunctionalizing the interpreter with double-barreled continuations yields an abstract machine with two stacks: a regular control stack and a stack of exception handlers. Architecturally, these two stacks are not a clever invention or a gratuitous variant, but the consequences of a principled derivation.

5.5 Summary and conclusion

We have presented a series of evaluators and an abstract machine that correspond to a call-by-value monadic evaluator and an exception monad. The first evaluator is a traditional one in exception-oriented style. The machine is a CEK machine with exceptions. We have also shown how to obtain an alternative implementation of exceptions with two continuations and how it leads to an abstract machine with two stacks. The correctness of the evaluators and of the abstract machines is a corollary of the correctness of the original monadic evaluator and of the transformations.

6 Combining state and exceptions

As is well known, there are two ways to combine the state and exception monads, giving rise to different semantics [4]. We consider both combinations and derive the corresponding abstract machines.

Both combinations of the state and exception monads are represented as structures with the following signature:

```
signature STATE_AND_EXCEPTION_MONAD
= sig
    include STATE_MONAD
    datatype 'a E = RES of 'a | EXC
    val raise_exception : 'a monad
    val handle_exception : 'a monad * (unit -> 'a monad) -> 'a monad
  end
```

The source language and the monadic evaluator are those of Section 5 with a special form to handle exceptions. The base environment is extended with functions `get`, `set`, and `raise` to read and write the state, and to raise an exception.

6.1 From a combined state and exception monad to an abstract machine (version 1)

We consider the combination of the state and exception monads where the state is passed both on successful termination and on exceptional termination. We equip the monad with operations to read and write the state, and to raise and handle exceptions. When an exception is handled, the state in which the exception was raised is available, and execution can be resumed in that state:

```
structure State_And_Exception_Monad : STATE_AND_EXCEPTION_MONAD
= struct
    datatype 'a E = RES of 'a | EXC
    type storable = int
    type state = storable
    type 'a monad = state -> ('a E * state)

    fun unit a
        = (fn s => (RES a, s))
    fun bind (m, k)
        = (fn s => let val (a, s') = m s
                   in case a
                        of (RES a) => k a s'
                         | EXC => (EXC, s')
                   end)

    val get = (fn s => (RES s, s))
    fun set i
        = (fn s => (RES s, i))
```

```
    val raise_exception = (fn s => (EXC, s))
    fun handle_exception (t0, t1)
        = (fn s => let val (a, s') = t0 s
                   in case a
                        of (RES a) => (RES a, s')
                         | EXC => t1 () s'
                   end)
  end
```

We inline this combined monad in the monadic evaluator and closure convert, CPS-transform, and defunctionalize the resulting evaluator to obtain the following abstract machine with state and exceptions:

- Source syntax (terms):

$$t \quad ::= \quad i \mid x \mid \lambda x.t \mid t_0\, t_1 \mid t_0\, \texttt{handle}\, t_1$$

- Expressible values (integers, closures, and predefined functions), results, and evaluation contexts:

$$\begin{array}{lcl} v & ::= & i \mid [x,\, t,\, e] \mid \mathit{succ} \mid \mathit{get} \mid \mathit{set} \mid \mathit{raise} \\ r & ::= & (\mathit{res}(v), s) \mid (\mathit{exc}, s) \\ k & ::= & \texttt{stop} \mid \texttt{fun}(v, k) \mid \texttt{arg}(t, e, k) \mid \texttt{exc}(t, e, k) \end{array}$$

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

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

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

6.2 From a combined state and exception monad to an abstract machine (version 2)

We consider the combination of the state and exception monads where the state is passed on successful termination and discarded on exceptional termination. We equip the monad with operations to read and write the state, and to raise and handle exceptions. When handling an exception, execution cannot be resumed in the state in which the exception was raised. One choice, which we take here, is to use a so-called 'snapback' or transactional semantics and resume execution in the state that was active when the handler was installed [16, 22]:

```
structure State_And_Exception_Monad' : STATE_AND_EXCEPTION_MONAD
= struct
    datatype 'a E = RES of 'a | EXC
    type storable = int
    type state = storable
    type 'a monad = state -> ('a * state) E

    fun unit a = (fn s => RES (a, s))
    fun bind (m, k)
        = (fn s => let val a = m s
                   in (case a
                         of (RES (a, s')) => k a s'
                          | EXC => EXC)
                   end)

    val get = (fn s => RES (s, s))
    fun set i
        = (fn s => RES (s, i))
    val raise_exception = (fn s => EXC)
    fun handle_exception (t0, t1)
        = (fn s => let val a = t0 s
                   in case a
                        of (RES (a, s')) => RES (a, s')
                         | EXC => t1 () s
                   end)
  end
```

We inline this combined monad in the monadic evaluator and closure convert, CPS-transform, and defunctionalize the resulting evaluator to obtain the following abstract machine with state and exceptions:

- Source syntax (terms):

$$t \;::=\; i \mid x \mid \lambda x.t \mid t_0\, t_1 \mid t_0\, \texttt{handle}\, t_1$$

- Expressible values (integers, closures, and predefined functions), results, and evaluation contexts:

$$\begin{array}{lcl} v & ::= & i \mid [x,\, t,\, e] \mid \mathit{succ} \mid \mathit{get} \mid \mathit{set} \mid \mathit{raise} \\ r & ::= & \mathit{res}(v, s) \mid \mathit{exc} \\ k & ::= & \texttt{stop} \mid \texttt{fun}(v, k) \mid \texttt{arg}(t, e, k) \mid \texttt{exc}(t, e, s, k) \end{array}$$

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

$$
\begin{array}{|rcl|}
\hline
t & \Rightarrow_{init} & \langle t,\, e_{init},\, s_{init},\, \mathtt{stop} \rangle \\
\hline
\langle i,\, e,\, s,\, k \rangle & \Rightarrow_{eval} & \langle k,\, res(i, s) \rangle \\
\langle x,\, e,\, s,\, k \rangle & \Rightarrow_{eval} & \langle k,\, res(e(x), s) \rangle \\
\langle \lambda x.t,\, e,\, s,\, k \rangle & \Rightarrow_{eval} & \langle k,\, res([x,\, t,\, e], s) \rangle \\
\langle t_0\, t_1,\, e,\, s,\, k \rangle & \Rightarrow_{eval} & \langle t_0,\, e,\, s,\, \mathtt{arg}(t_1, e, k) \rangle \\
\langle t_0\, \mathtt{handle}\, t_1,\, e,\, s,\, k \rangle & \Rightarrow_{eval} & \langle t_0,\, e,\, s,\, \mathtt{exc}(t_1, e, s, k) \rangle \\
\hline
\langle \mathtt{arg}(t_1, e, k),\, res(v, s) \rangle & \Rightarrow_{cont} & \langle t_1,\, e,\, s,\, \mathtt{fun}(v, k) \rangle \\
\langle \mathtt{arg}(t_1, e, k),\, exc \rangle & \Rightarrow_{cont} & \langle k,\, exc \rangle \\
\langle \mathtt{fun}([x,\, t,\, e], k),\, res(v, s) \rangle & \Rightarrow_{cont} & \langle t,\, e[x \mapsto v],\, s,\, k \rangle \\
\langle \mathtt{fun}(succ, k),\, res(i, s) \rangle & \Rightarrow_{cont} & \langle k,\, res(i+1, s) \rangle \\
\langle \mathtt{fun}(get, k),\, res(v, s) \rangle & \Rightarrow_{cont} & \langle k,\, res(s, s) \rangle \\
\langle \mathtt{fun}(set, k),\, res(i, s) \rangle & \Rightarrow_{cont} & \langle k,\, res(s, i) \rangle \\
\langle \mathtt{fun}(raise, k),\, res(v, s) \rangle & \Rightarrow_{cont} & \langle k,\, exc \rangle \\
\langle \mathtt{fun}(v, k),\, exc \rangle & \Rightarrow_{cont} & \langle k,\, exc \rangle \\
\langle \mathtt{exc}(t_1, e, s, k),\, res(v, s') \rangle & \Rightarrow_{cont} & \langle k,\, res(v, s') \rangle \\
\langle \mathtt{exc}(t_1, e, s, k),\, exc \rangle & \Rightarrow_{cont} & \langle t_1,\, e,\, s,\, k \rangle \\
\hline
\langle \mathtt{stop},\, r \rangle & \Rightarrow_{final} & r \\
\hline
\end{array}
$$

where
$$
\begin{array}{rcl}
e_{base} & = & e_{empty}[\mathtt{succ} \mapsto succ] \\
e_{init} & = & e_{base}[\mathtt{get} \mapsto get][\mathtt{set} \mapsto set][\mathtt{raise} \mapsto raise]
\end{array}
$$
and s_{init} is the initial state.

6.3 Summary and conclusion

We have presented two combined monads accounting for state and exceptions and the two abstract machines corresponding to a call-by-value monadic evaluator and these two monads. The design decisions of combining monads was taken at the monadic level and the corresponding abstract machines were then derived mechanically. The correctness of these abstract machines is a corollary of the correctness of the original monadic evaluator and of the transformations.

7 Stack inspection as a 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) [18]. Before execution, all 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 [6]. 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 [26]. 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 [7]. In the CM machine, the call stack is represented as evaluation contexts each containing 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.

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.

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

$$R \subseteq P$$
$$m \in P \rightarrow \{grant, no\}$$

- Source syntax (terms):

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

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

$$\begin{array}{rcl} v & ::= & [x,\, t,\, e] \\ o & ::= & v \mid \mathit{fail} \\ k & ::= & \texttt{stop}(m) \mid \texttt{arg}(t, e, k, m) \mid \texttt{fun}(v, k, m) \end{array}$$

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

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

where m_{empty} denotes the empty permission table,

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

and

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

In the CM machine, evaluation contexts contain permission tables. We unzip the CM evaluation contexts into CEK evaluation contexts and a list of permission tables that is managed last in, first out, i.e., 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:

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

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

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

where

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

As we have already observed in previous work [2, 5, 8, 10], 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 can therefore be expressed as an instance of the monadic evaluator with a state monad.

In the 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`. The stack inspection state monad is implemented as a structure with the following signature:

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

    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 state monad of Section 4: `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 -> outcome monad
   and outcome = FAILURE
               | SUCCESS of value

(*  eval : term * value Env.env -> value  *)
fun eval (VAR x, e)
    = unit (SUCCESS (Env.lookup (e, x)))
  | eval (LAM (x, t), e)
    = unit (SUCCESS (FUN (fn v => eval (t, Env.extend (x, v, e)))))
  | eval (APP (t0, t1), e)
    = bind (push_empty, fn () =>
        bind (eval (t0, e), fn (SUCCESS v0) =>
          bind (pop_top, fn () =>
            bind (push_empty, fn () =>
              bind (eval (t1, e), fn (SUCCESS 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)
    = unit FAILURE
```

The process is reversible. Starting from this 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, and defunctionalizing the resulting continuations.

8 Combining stack inspection and exceptions

As in Section 6, the (stack-inspection) state monad and the exception monad can be combined in two ways. In Section 6, the two combinations gave rise to significantly different semantics. In the case of stack inspection, the two variants do not differ much. The reason is that the state represents the permissions available during execution. When handling an exception, it is crucial for security that the permissions in effect at the time the handler was installed are reinstated and execution resumed with those permissions. The snapback semantics is therefore appropriate whether or not the state in which the exception was raised is available.

8.1 A combined stack-inspection and exception monad

We consider the combination where the state is discarded when an exception is raised. In Section 6 we put the raise operation in the initial environment. Here, for diversity value, we add it to the language of Section 7 as a syntactic construct:

```
datatype term = ...
              | RAISE
              | HANDLE of term * term
```

The combined stack inspection state monad and exception monad is implemented as a structure with the following signature:

```
signature STACK_INSPECTION_STATE_AND_EXCEPTION_MONAD
= sig
    include STACK_INSPECTION_STATE_MONAD
    datatype 'a E = RES of 'a
                  | EXC

    val raise_exception : 'a monad
    val handle_exception : 'a monad * 'a monad -> 'a monad
  end
```

The definition of the monad type constructor, `unit`, `bind`, `raise_exception`, and `handle_exception` are as follows, using a type `permission_table` to represent permission tables:

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

```
fun unit a
    = (fn s => RES (a, s))
fun bind (m, k)
    = (fn s => (case m s
                  of (RES (a, s')) => k a s'
                   | EXC => EXC))

val raise_exception = (fn s => EXC)
fun handle_exception (t0, t1)
    = (fn s => (case t0 s
                  of (RES r) => RES r
                   | EXC => t1 s))
```

The definition of the monadic operations for manipulating permission stacks are straightforwardly extended to account for exceptions. The monadic evaluator is as in Section 7 with two extra clauses:

```
  ...
| eval (RAISE, e)
  = raise_exception
| eval (HANDLE (t0, t1), e)
  = handle_exception (eval (t0, e), eval (t1, e))
```

8.2 An abstract machine for stack inspection and exceptions

Inlining the monad in the monadic evaluator, closure converting the expressible values, CPS-transforming the evaluator, and defunctionalizing the resulting continuations yields the following abstract machine with stack inspection and exceptions. Permissions, permission tables, and the definition of $\mathcal{OK}$ are as in the unzipped CM machine:

- Source syntax (terms):

$$\begin{array}{lcl} t & ::= & x \mid \lambda x.t \mid t_0\,t_1 \mid \\ & & R[t] \mid \texttt{grant}\ R\ \texttt{in}\ t \mid \texttt{test}\ R\ \texttt{then}\ t_0\ \texttt{else}\ t_1 \mid \texttt{fail} \mid \\ & & \texttt{raise} \mid t_0\,\texttt{handle}\,t_1 \end{array}$$

- Expressible values (closures), results, outcomes, and evaluation contexts:

$$\begin{array}{lcl} v & ::= & [x,\,t,\,e] \\ r & ::= & \mathit{res}(v, \mathit{ms}) \mid \mathit{exc} \\ o & ::= & r \mid \mathit{fail} \\ k & ::= & \texttt{stop} \mid \texttt{arg}(t,e,k) \mid \texttt{fun}(v,k) \mid \texttt{exc}(t,e,\mathit{ms},k) \end{array}$$

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

t	$\Rightarrow_{init}$	$\langle t,\, e_{empty},\, m_{empty} :: nil,\, \mathtt{stop}\rangle$
$\langle x,\, e,\, ms,\, k\rangle$	$\Rightarrow_{eval}$	$\langle k,\, res(e(x), ms)\rangle$
$\langle \lambda x.t,\, e,\, ms,\, k\rangle$	$\Rightarrow_{eval}$	$\langle k,\, res([x,\, t,\, e], ms)\rangle$
$\langle t_0\, t_1,\, e,\, ms,\, k\rangle$	$\Rightarrow_{eval}$	$\langle t_0,\, e,\, m_{empty} :: ms,\, \mathtt{arg}(t_1, e, k)\rangle$
$\langle R[t],\, e,\, m :: ms,\, k\rangle$	$\Rightarrow_{eval}$	$\langle t,\, e,\, m[\overline{R} \mapsto no] :: ms,\, k\rangle$
$\langle \mathtt{grant}\ R\ \mathtt{in}\ t,\, e,\, m :: ms,\, k\rangle$	$\Rightarrow_{eval}$	$\langle t,\, e,\, m[R \mapsto grant] :: ms,\, k\rangle$
$\langle \mathtt{test}\ R\ \mathtt{then}\ t_0\ \mathtt{else}\ t_1,\, e,\, ms,\, k\rangle$	$\Rightarrow_{eval}$	$\langle t_0,\, e,\, ms,\, k\rangle$ *if* $\mathcal{OK}[R][ms]$
$\langle \mathtt{test}\ R\ \mathtt{then}\ t_0\ \mathtt{else}\ t_1,\, e,\, ms,\, k\rangle$	$\Rightarrow_{eval}$	$\langle t_1,\, e,\, ms,\, k\rangle$ *otherwise*
$\langle \mathtt{raise},\, e,\, ms,\, k\rangle$	$\Rightarrow_{eval}$	$\langle k,\, exc\rangle$
$\langle t_0\, \mathtt{handle}\ t_1,\, e,\, ms,\, k\rangle$	$\Rightarrow_{eval}$	$\langle t_0,\, e,\, ms,\, \mathtt{exc}(t_1, e, ms, k)\rangle$
$\langle \mathtt{fail},\, e,\, ms,\, k\rangle$	$\Rightarrow_{final}$	*fail*
$\langle \mathtt{arg}(t, e, k),\, res(v, m :: ms)\rangle$	$\Rightarrow_{cont}$	$\langle t,\, e,\, m_{empty} :: ms,\, \mathtt{fun}(v, k)\rangle$
$\langle \mathtt{arg}(t, e, k),\, exc\rangle$	$\Rightarrow_{cont}$	$\langle k,\, exc\rangle$
$\langle \mathtt{fun}([x,\, t,\, e], k),\, res(v, m :: ms)\rangle$	$\Rightarrow_{cont}$	$\langle t,\, e[x \mapsto v],\, ms,\, k\rangle$
$\langle \mathtt{fun}([x,\, t,\, e], k),\, exc\rangle$	$\Rightarrow_{cont}$	$\langle k,\, exc\rangle$
$\langle \mathtt{exc}(t, e, ms', k),\, res(v, ms)\rangle$	$\Rightarrow_{cont}$	$\langle k,\, res(v, ms)\rangle$
$\langle \mathtt{exc}(t, e, ms, k),\, exc\rangle$	$\Rightarrow_{cont}$	$\langle t,\, e,\, ms,\, k\rangle$
$\langle \mathtt{stop},\, r\rangle$	$\Rightarrow_{final}$	r

8.3 Summary and conclusion

We have presented a combined monad accounting for stack inspection and exceptions and the abstract machine corresponding to a call-by-value monadic evaluator and this monad. The design decision of how to combine the monads is taken at the monadic level and the construction of the corresponding abstract machine is mechanical. 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.

9 A dedicated monad for stack inspection

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

```
type 'a monad = permission_table list -> permission_table list * 'a
```

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
                -> permission_table * 'a
```

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

```
fun unit a
    = (fn (p, pl) => (p, a))
fun bind (f, g)
    = (fn (p, pl) => let val (p', a) = f (p, pl)
                     in g a (p', pl)
                     end)
```

Proof: By straightforward equational reasoning. □

This monad provides a more accurate characterization of stack inspection.

As an exercise, we have constructed the corresponding abstract machine as well as a new abstract machine corresponding to this stack-inspection monad combined with the exception monad. The resulting abstract machines are similar to the ones in Sections 7 and 8.

10 Related work

Since Moggi's breakthrough [24], monads have been widely used to parameterize functional programs with effects [4]. 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 [20] to the modern JVM [21]. As observed by Diehl, Hartel, and Sestoft [12], research on abstract machines has chiefly focused on developing new machines and proving them correct. The thrust of our work is a correspondence between interpreters and abstract machines [2, 8].

Stack inspection is used as a fine-grained access control mechanism for Java [19]. 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 [31]. 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 [18]. 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 [6]. This tail-call optimized semantics is given in the form of a CESK machine, which was the starting point for our work.

11 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 state monad, the exception monad, and mixes of both lead us to variants of the CEK machine that we identify as such. We have also characterized Clements and Felleisen's properly tail-recursive stack inspection as a state monad and we have combined it with an exception monad to construct a new abstract machine with properly tail-recursive stack inspection and exceptions.

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, a state monad, an exception monad, and their combination;
- a characterization of Clements and Felleisen's properly tail-recursive stack inspection as a state monad and a reconstruction of the CM machine;
- the combination of the stack-inspection state monad with an exception monad and the construction of the corresponding abstract machine; and
- a dedicated monad for properly tail-recursive stack inspection and how to construct the 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.

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 call-by-need evaluators and lazy abstract machines. Technical Report BRICS RS-03-24, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 2003. Accepted for publication in Information Processing Letters.

[4] 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.

[5] Dariusz Biernacki and Olivier Danvy. From interpreter to logic engine by defunctionalization. Technical Report BRICS RS-03-25, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 2003. Presented at the 2003 International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2003).

[6] 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.

[7] 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.

[8] 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.

[9] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[10] 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.

[11] 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.

[12] Stephan Diehl, Pieter Hartel, and Peter Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16:739–751, 2000.

[13] 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.

[14] 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.

[15] 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.

[16] Andrzej Filinski. Representing layered monads. In Alex Aiken, editor, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 175–188, San Antonio, Texas, January 1999. ACM Press.

[17] 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.

[18] Cédric Fournet and Andrew D. Gordon. Stack inspection: Theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, May 2003.

[19] 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.

[20] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[21] Tim Lindholm and Frank Yellin, editors. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Addison-Wesley, 1999.

[22] Robert E. Milne and Christopher Strachey. *A Theory of Programming Language Semantics.* Chapman and Hall, London, and John Wiley, New York, 1976.

[23] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.

[24] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[25] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[26] Christian Queinnec. *Lisp in Small Pieces.* Cambridge University Press, Cambridge, 1996.

[27] 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.

[28] 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.

[29] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1/2):1–49, 2000.

[30] 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.

[31] 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.

Recent BRICS Report Series Publications

RS-03-35 Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. ***A Functional Correspondence between Monadic Evaluators and Abstract Machines for Languages with Computational Effects*****. November 2003. 31 pp.**

RS-03-34 Luca Aceto, Willem Jan Fokkink, Anna Ingólfsdóttir, and Bas Luttik. ***CCS with Hennessy's Merge has no Finite Equational Axiomatization*****. November 2003. 37 pp.**

RS-03-33 Olivier Danvy. ***A Rational Deconstruction of Landin's SECD Machine*****. October 2003. 32 pp. This report supersedes the earlier BRICS report RS-02-53.**

RS-03-32 Philipp Gerhardy and Ulrich Kohlenbach. ***Extracting Herbrand Disjunctions by Functional Interpretation*****. October 2003. 17 pp.**

RS-03-31 Stephen Lack and Paweł Sobociński. ***Adhesive Categories*****. October 2003. 25 pp.**

RS-03-30 Jesper Makholm Byskov, Bolette Ammitzbøll Madsen, and Bjarke Skjernaa. ***New Algorithms for Exact Satisfiability*****. October 2003. 31 pp.**

RS-03-29 Aske Simon Christensen, Christian Kirkegaard, and Anders Møller. ***A Runtime System for XML Transformations in Java*****. October 2003. 15 pp.**

RS-03-28 Zoltán Ésik and Kim G. Larsen. ***Regular Languages Definable by Lindström Quantifiers*****. August 2003. 82 pp. This report supersedes the earlier BRICS report RS-02-20.**

RS-03-27 Luca Aceto, Willem Jan Fokkink, Rob J. van Glabbeek, and Anna Ingólfsdóttir. ***Nested Semantics over Finite Trees are Equationally Hard*****. August 2003. 31 pp.**

RS-03-26 Olivier Danvy and Ulrik P. Schultz. ***Lambda-Lifting in Quadratic Time*****. August 2003. 23 pp. Extended version of a paper appearing in Hu and Rodríguez-Artalejo, editors,** ***Sixth International Symposium on Functional and Logic Programming*****, FLOPS '02 Proceedings, LNCS 2441, 2002, pages 134–151. This report supersedes the earlier BRICS report RS-02-30.**

RS-03-25 Biernacki Dariusz and Danvy Olivier. ***From Interpreter to Logic Engine: A Functional Derivation*****. June 2003.**