BRICS

Basic Research in Computer Science

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-04-28

ISSN 0909-0878 December 2004

Copyright © 2004, Mads Sig Ager & Olivier Danvy & Jan Midtgaard.
BRICS, Department of Computer Science
University of Aarhus. All rights reserved.

Reproduction of all or part of this work is permitted for educational or research use on condition that this copyright notice is included in any copy.

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/04/28/`

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‡

December 2004

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 first with the identity monad, obtaining the CEK machine, and then with a lifting monad, a state monad, and with a lifted state monad, obtaining variants of the CEK machine with error handling, state and a combination of error handling and state.

In addition, we characterize the tail-recursive stack inspection presented by Clements and Felleisen as a lifted state monad. This enables us to combine this 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.

*Extended version of an article to appear in TCS.

†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`

Contents

1 Introduction **4**
- 1.1 Example: the factorial function . . . 4
- 1.2 The functional correspondence . . . 6

2 A call-by-value monadic evaluator in ML **7**

3 On using ML as a metalanguage **9**

4 From the identity monad to an abstract machine **10**
- 4.1 The identity monad . . . 10
- 4.2 Inlining the monad in the monadic evaluator . . . 10
- 4.3 Closure conversion . . . 11
- 4.4 CPS transformation . . . 12
- 4.5 Defunctionalization . . . 12
- 4.6 The CEK machine . . . 13
- 4.7 Summary and conclusion . . . 14

5 From a lifting monad to an abstract machine **14**
- 5.1 A lifting monad . . . 14
- 5.2 A CEK machine with error handling . . . 15
- 5.3 Summary and conclusion . . . 16

6 From a state monad to an abstract machine **16**
- 6.1 A state monad . . . 16
- 6.2 A CEK machine with state . . . 17
- 6.3 Summary and conclusion . . . 18

7 From a lifted state monad to an abstract machine **18**
- 7.1 A lifted state monad . . . 18
- 7.2 A CEK machine with error handling and state . . . 20
- 7.3 Summary and conclusion . . . 20

8 Stack inspection as a lifted state monad **21**

9 A dedicated monad for stack inspection **26**

10 Related work **26**

11 Conclusion **27**

A Propagating vs. stopping **28**

B From an exception monad to an abstract machine **30**
B.1 An exception monad 30
B.2 A CEK machine with exceptions 31
B.3 An alternative implementation of exceptions 32
B.4 Summary and conclusion 32

C Combining state and exceptions **33**
C.1 From a combined state and exception monad to an abstract machine (version 1) 33
C.2 From a combined state and exception monad to an abstract machine (version 2) 35
C.3 Summary and conclusion 37

D Combining stack inspection and exceptions **37**
D.1 A combined stack-inspection and exception monad 37
D.2 An abstract machine for stack inspection and exceptions 39
D.3 Summary and conclusion 40

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 [3, 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 systems in the sense of operational semantics). The correspondence builds on the observation that *defunctionalized continuation-passing evaluators are abstract machines*. One can therefore obtain an abstract machine, i.e., a state-transition system [31], by CPS-transforming and defunctionalizing an evaluator. More generally, any recursive function that is defined over an inductive data type can be turned into a transition system by CPS transformation and defunctionalization. Let us first illustrate the correspondence with the factorial function and the corresponding transition system.

1.1 Example: the factorial function

Here is the factorial function, expressed in Standard ML [28]:

```
(*  main0 : int -> int  *)
fun main0 n
    = fac0 n
(*  fac0 : int -> int  *)
and fac0 0
    = 1
  | fac0 n
    = n * (fac0 (n - 1))
```

The definition above is in direct style. We transform it into continuation-passing style (CPS) [10, 30, 36] by naming each intermediate result, sequentializing their computation, and introducing an extra functional argument, the continuation:

```
(*  main1 : int -> int  *)
fun main1 n
    = fac1 (n, fn a => a)
(*  fac1 : int * (int -> int) -> int  *)
and fac1 (0, k)
    = k 1
  | fac1 (n, k)
    = fac1 (n - 1, fn v => k (n * v))
```

In this CPS program, as in all CPS programs, all calls are tail calls and all subcomputations are elementary (i.e., they cannot diverge).

Defunctionalizing the continuation amounts to changing its representation and replacing it by a data type and the corresponding apply function [11, 34]. We enumerate all the constructors (i.e., lambda-abstractions) that give rise to inhabitants of this function space. There are two such constructors: the initial continuation

in `main` and the continuation in the induction case of `fac`. These two constructors are consumed when the continuation is applied, which happens in both clauses of `fac`—one immediately and the other one in the continuation. The data type representing the continuation therefore has two constructors, and the corresponding apply function has two clauses:

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

(*  apply_cont : cont * int -> int  *)
fun apply_cont (C0, v)
    = v
  | apply_cont (C1 (n, k), v)
    = apply_cont (k, n * v)
```

The first constructor is nullary (i.e., constant) and the second is binary, reflecting the number of free variables in the corresponding lambda-abstractions.

In the defunctionalized factorial function, the continuation is constructed using `C0` and `C1`, and consumed using `apply_cont`:

```
(*  main2 : int -> int  *)
fun main2 n
    = fac2 (n, C0)
(*  fac2 : int * cont -> int  *)
and fac2 (0, k)
    = apply_cont (k, 1)
  | fac2 (n, k)
    = fac2 (n - 1, C1 (n, k))
```

This program is first order because it is defunctionalized. All of its calls are tail calls and all of its subcomputations are elementary because it is a (defunctionalized) CPS program. Therefore it is a state-transition system—i.e., an abstract machine—in the sense of Plotkin [31]: for each function, its actual parameters define a configuration and each of its clauses defines a transition.

For clarity, we can reformat this transition system in a more traditional way:

- Input (integer): n
- Output (integer): v
- Defunctionalized continuations: $k ::= C_0 \mid C_1(n, k)$
- Initial transition, transition rules (two kinds), and final transition:

n	$\Rightarrow_{init}$	$\langle n, C_0 \rangle$
$\langle 0, k \rangle$	$\Rightarrow_{fac}$	$\langle k, 1 \rangle$
$\langle n, k \rangle$	$\Rightarrow_{fac}$	$\langle n-1, C_1(n, k) \rangle$
$\langle C_1(n, k), v \rangle$	$\Rightarrow_{cont}$	$\langle k, n \times v \rangle$
$\langle C_0, v \rangle$	$\Rightarrow_{final}$	v

1.2 The functional correspondence

This relation between defunctionalized continuation-passing evaluators and abstract machines suggests a functional correspondence between evaluators and abstract machines [3, 9]. This correspondence is constructive: to obtain an abstract machine, we start from a compositional evaluator and

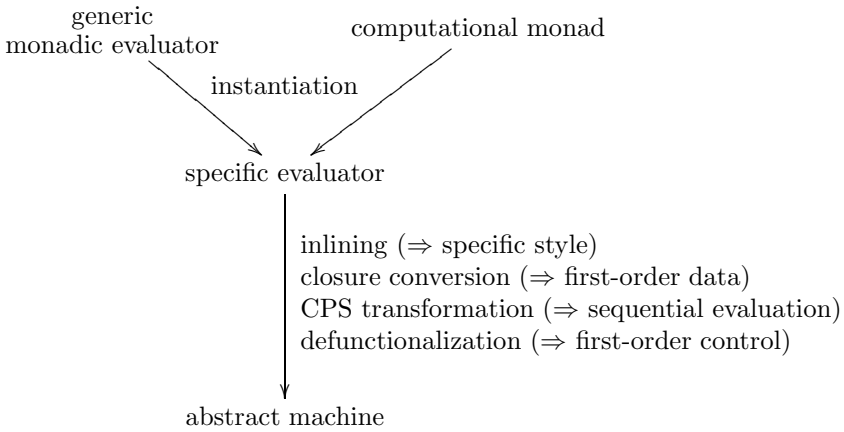
1. make it operate on first-order data by closure-converting its expressible and denotable values [25, 37];
2. sequentialize evaluation by CPS-transforming it [10, 30, 36], thereby materializing its control flow into continuations; and
3. make it operate on first-order control by defunctionalizing these continuations [11, 34].

The correspondence originates in Reynolds's seminal article "Definitional Interpreters for Higher-Order Programming Languages" [34], where all the elements (closure conversion, CPS transformation, and defunctionalization) are presented and used. Today, these elements are classical, textbook material [15, 21]. 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. To the best of our knowledge, and with the exception of Felleisen and Friedman's initial presentation of the CEK machine [16, Section 2], these two machines 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 [3, 9].

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 [2]. 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, 29, 38]. We consider a generic evaluator parameterized by a monad (Sections 2 and 3). We then successively consider several monads: the identity monad (Section 4), a lifting monad (Section 5), a state monad (Section 6), and a lifted state monad (Section 7). We inline the components of these monads in the generic evaluator, obtaining specific evaluators. The first one is in direct style, reflecting the computational effect of the identity monad. The second one is in direct style with error handling, reflecting the computational effect of the lifting monad. The third and fourth ones are in state-passing style, reflecting the computational effect of the state monad

and of the lifted state monad. 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' [20]. 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 8). 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 appendix we also consider an exception monad (Appendix B), the two possible monads obtained by combining this exception monad with the state monad (Appendix C), and a combination of the stack-inspection monad and the exception monad (Appendix D). We mechanically construct the corresponding abstract machines.

2 A call-by-value monadic evaluator in ML

As traditional [5, 17, 38], we specify a monad as a type constructor and two polymorphic functions:

```
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 as values 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, as in Wadler's tutorial [38]. 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.

Finally, we follow the functional-programming tradition initiated by Wadler [38] and we reason equationally over the definitions of `unit` and `bind` to verify that they satisfy the three monadic laws:

- Left unit: `bind (unit a, k) = k a`
- Right unit: `bind (m, unit) = m`
- Associativity: `bind (m, fn a => bind (k a, h)) = bind (bind (m, k), h)`

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 [3]: closure conversion, CPS transformation, and defunctionalization. The result is Felleisen et al.'s CEK machine [16, 19].

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
```

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

Proof: The identity monad is known to be a monad. Alternatively, the monadic laws can be verified by equational reasoning. □

4.2 Inlining the monad in the monadic evaluator

Inlining the components of the identity monad in the monadic evaluator of Section 2 yields an ordinary call-by-value evaluator in direct style where numerals are mapped to numbers, variables are mapped to their denotation, etc.:

```
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. The rest of the evaluator does not change:

```
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. The data type of values and the initial environment do not change. The evaluation function takes an extra parameter, the continuation. Values that used to be returned in the direct-style evaluator are now passed to the continuation. Intermediate values that used to be named with a let expression are now named by the parameter of a new continuation:

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

The data type of values and the initial environment do not change. Continuations that used to be constructed with a function abstraction in the continuation-passing evaluator are now constructed with `STOP`, `ARG`, or `FUN`. Continuations that used to be consumed with a function application are now consumed by the dispatching function `apply_cont`:

```
(*  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 [16,19], which we present next.

4.6 The CEK machine

- Source syntax (terms):

$$t \quad ::= \quad \ulcorner i \urcorner \mid x \mid \lambda x.t \mid t_0\, t_1$$

- Expressible values (integers, closures, and predefined functions) and evaluation contexts (i.e., defunctionalized continuations):

$$v \quad ::= \quad i \mid [x, t, e] \mid succ$$
$$k \quad ::= \quad \texttt{stop} \mid \texttt{arg}(t, e, k) \mid \texttt{fun}(v, k)$$

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

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

$$\text{where} \quad \begin{array}{rcl} e_{base} & = & e_{empty}[\mathtt{succ} \mapsto succ] \\ e_{init} & = & e_{base} \end{array}$$

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 lifting monad to an abstract machine

We specify a lifting monad and inline it in the monadic evaluator, obtaining a lifted evaluator. Closure converting, CPS-transforming, and defunctionalizing this lifted evaluator yields a CEK machine with error handling.

5.1 A lifting monad

We consider the lifting monad equipped with an operation for failing:

```
signature LIFTING_MONAD
= sig
    include MONAD

    val fail : 'a monad
  end

structure Lifting_Monad : LIFTING_MONAD
= struct
    datatype 'a lift = LIFT of 'a | BOTTOM
    type 'a monad = 'a lift

    fun unit a
        = LIFT a
```

```
  fun bind (m, k)
      = (case m
           of (LIFT a)
              => k a
            | BOTTOM
              => BOTTOM)
  val fail = BOTTOM
end
```

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

Proof: The lifting monad is known to be a monad [29]. Alternatively, the monadic laws can be verified by equational reasoning. □

We extend the base environment with the function `fail`:

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

5.2 A CEK machine with error handling

Inlining the components of the lifting monad in the monadic evaluator of Section 2 yields a call-by-value evaluator. 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 handling. The source language and evaluation contexts are as in the CEK machine of Section 4.

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

$$\begin{array}{rcl} v & ::= & i \mid [x,\, t,\, e] \mid \mathit{succ} \mid \mathit{fail} \\ r & ::= & \mathit{lift}(v) \mid \mathit{bottom} \end{array}$$

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

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

$$\text{where} \quad e_{base} = e_{empty}[\texttt{succ} \mapsto \mathit{succ}]$$
$$e_{init} = e_{base}[\texttt{fail} \mapsto \mathit{fail}]$$

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 lift -> 'a` and the pair of continuations `(value -> '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 lifting monad and an abstract machine that corresponds to the call-by-value monadic evaluator and this monad. The resulting machine is a version of the CEK machine with error handling. 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 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.

6.1 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
```

Proposition 3 *The type constructor above, together with the above definitions of* `unit` *and* `bind`*, satisfies the three monadic laws.*

Proof: The state monad is known to be a monad [29]. Alternatively, the monadic laws can be verified by equational reasoning. □

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

```
val env_init
    = 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 : State_Monad.state`.

6.2 A CEK machine with state

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. As in Section 4, we closure-convert, CPS-transform, and defunctionalize the inlined evaluator. The result is a CEK machine with 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:

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

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

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

$$\begin{array}{lcl} \text{where} \quad e_{base} & = & e_{empty}[\mathtt{succ} \mapsto succ] \\ e_{init} & = & e_{base}[\mathtt{get} \mapsto get][\mathtt{set} \mapsto set] \end{array}$$

and s_{init} is the initial state (e.g., -1).

6.3 Summary and conclusion

We have presented a state monad and an abstract machine that corresponds to a call-by-value monadic evaluator and this monad. The evaluator obtained by inlining the components of the state monad is 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.

7 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 handling and state. This monad and this machine form a stepping stone towards stack inspection.

7.1 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 4 *The type constructor above, together with the above definitions of* `unit` *and* `bind`*, satisfies the three monadic laws.*

Proof: The lifted state monad is a combination of the lifting monad and of a state monad, and is known to be a monad [29]. Alternatively, the monadic laws can be verified by equational reasoning. □

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`.

7.2 A CEK machine with error handling 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 handling 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:

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

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

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

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

As in Section 5 the machine could be optimized as illustrated in Appendix A to stop immediately in case of failure.

7.3 Summary and conclusion

We have presented a lifted state monad and an 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 error handling. 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.

8 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) [20]. 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 [32]. It is presented in the form of a CESK machine, the CM machine, and Clements and Felleisen have proved that this machine uses asymptotically 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. The simplified CM machine is as follows:

- Permissions $R \subseteq P$ and permission tables $m \in P \to \{\mathit{grant}, \mathit{no}\}$ for a fixed set of permissions P.
- 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	$\Rightarrow_{\mathit{init}}$	$\langle t,\, e_{\mathit{empty}},\, \texttt{stop}(m_{\mathit{empty}})\rangle$
$\langle x,\, e,\, k\rangle$	$\Rightarrow_{\mathit{eval}}$	$\langle k,\, e(x)\rangle$
$\langle \lambda x.t,\, e,\, k\rangle$	$\Rightarrow_{\mathit{eval}}$	$\langle k,\, [x,\, t,\, e]\rangle$
$\langle t_0\, t_1,\, e,\, k\rangle$	$\Rightarrow_{\mathit{eval}}$	$\langle t_0,\, e,\, \texttt{arg}(t_1, e, k, m_{\mathit{empty}})\rangle$
$\langle R[t],\, e,\, k\rangle$	$\Rightarrow_{\mathit{eval}}$	$\langle t,\, e,\, k[\overline{R} \mapsto \mathit{no}]\rangle$
$\langle \texttt{grant}\ R\ \texttt{in}\ t,\, e,\, k\rangle$	$\Rightarrow_{\mathit{eval}}$	$\langle t,\, e,\, k[R \mapsto \mathit{grant}]\rangle$
$\langle \texttt{test}\ R\ \texttt{then}\ t_0\ \texttt{else}\ t_1,\, e,\, k\rangle$	$\Rightarrow_{\mathit{eval}}$	$\langle t_0,\, e,\, k\rangle$ *if* $\mathcal{OK}[R][k]$
$\langle \texttt{test}\ R\ \texttt{then}\ t_0\ \texttt{else}\ t_1,\, e,\, k\rangle$	$\Rightarrow_{\mathit{eval}}$	$\langle t_1,\, e,\, k\rangle$ *if not* $\mathcal{OK}[R][k]$
$\langle \texttt{fail},\, e,\, k\rangle$	$\Rightarrow_{\mathit{final}}$	fail
$\langle \texttt{arg}(t, e, k, m),\, v\rangle$	$\Rightarrow_{\mathit{cont}}$	$\langle t,\, e,\, \texttt{fun}(v, k, m_{\mathit{empty}})\rangle$
$\langle \texttt{fun}([x,\, t,\, e], k, m),\, v\rangle$	$\Rightarrow_{\mathit{cont}}$	$\langle t,\, e[x \mapsto v],\, k\rangle$
$\langle \texttt{stop}(m),\, v\rangle$	$\Rightarrow_{\mathit{final}}$	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 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. This unzipping corresponds to separating the security mechanism from the function call mechanism. In the literature, it has been argued that security mechanisms such as stack inspection are best viewed separately from the stack. For instance, Abadi and Fournet separate the security mechanism from the stack in order to obtain a stronger security mechanism that is not tied to the behaviour of the stack [1]. Wallach, Appel, and Felten also separate the security mechanism from the stack to obtain an alternative semantics and implementation of stack inspection [39]. As for us, we separate the security mechanism from the stack in order to make the evaluation mechanism clearer: the CM machine is a variant of the CEK machine that keeps track of a stack of permission tables.

The unzipped CM machine is as follows. 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:

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

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 [3,6,9,11,12], 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 a 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`, `clear_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 clear_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 7; `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; `clear_top` clears the topmost permission table; `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 is 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 (clear_top, 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
```

This evaluator alters the state by pushing and popping permission tables when evaluating applications. One could be tempted to make these changes implicit by integrating them in the definition of `bind` and use the generic evaluator of Section 2. However, the result would not be a monad because the right-unit law would not hold. Therefore, the state changes have to appear explicitly in the monadic evaluator—a situation that has precedents, e.g., in one of Wadler's monadic evaluators [38, Section 2.5]. For these reasons the evaluator just above differs from the generic evaluator of Section 2.

The derivation 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. As an illustration we present a combination of the stack-inspection monad and the exception monad in Appendix D.

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.

9 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 is as follows:

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

Proposition 5 *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 equational reasoning. □

This monad provides a more accurate characterization of stack inspection than the one in Section 8.

As an exercise, we have constructed the corresponding abstract machine. This machine is similar to the one in Section 8.

10 Related work

Stack inspection is used as a fine-grained access control mechanism for Java [22]. 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 [39]. 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 [20]. 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 [29], monads have been widely used to parameterize functional programs with effects [5, 38]. 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 [25, 33] to the modern JVM [26]. 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 [3, 4, 6, 9] that takes its roots in Reynolds seminal work on definitional interpreters [34].

There are two forerunners to our work:

1. Reynolds's original work [34], where he CPS-transforms and defunctionalizes a call-by-value evaluator for λ-terms. We observe that the resulting first-order evaluator coincides with (and anticipates) the CEK machine.

2. Schmidt's PhD work [35], where he constructs a transition system by defunctionalizing a continuation-passing call-by-name evaluator for λ-terms. We observe that the resulting transition system coincides with (and anticipates) Krivine's machine.

The present work is a next step of our study of the correspondence between evaluators and abstract machines. 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 [23]. 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, 24].

11 Conclusion

We have extended the 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 on a case-by-case basis is deriving them from a monadic evaluator and a computational monad. As a consequence one does not need to establish the correctness of each abstract machine on a case-by-case basis since it is a corollary of the correctness of the original monadic evaluator and of the transformations. We have illustrated this alternative with several monads.

We have also characterized Clements and Felleisen's properly tail-recursive stack inspection as a lifted state monad, and we have proposed an alternative, dedicated monad for this computational effect. These two monads enable us to combine stack inspection with other computational effects at the monadic level and then systematically construct the corresponding abstract machines. 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 Dariusz Biernacki, Julia Lawall, Peter Thiemann, and Mitchell Wand for commenting a preliminary version of this article. Thanks are also due to Eugenio Moggi for his editorship and to John Clements and the anonymous referees for their feedback.

This work is partially supported by the ESPRIT Working Group APPSEM II (`http://www.appsem.org`), the SECURE project EU FET-GC IST-2001-32486, and the Danish Natural Science Research Council, Grant no. 21-03-0545.

A Propagating vs. stopping

This appendix illustrates the optimization of returning a final result directly instead of propagating it through surrounding evaluation contexts. 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  *)
fun mult_ds t
    = let (*  visit : bt -> int_lifted  *)
          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  *)
fun mult_cps t
    = let (*  visit : bt * (int_lifted -> int) -> int  *)
          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)
                                               => k (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  *)
fun mult_cps_opt t
    = let (*  visit : bt * (int -> int) -> int  *)
          fun visit (LEAF 0, k)
              = 0
            | visit (LEAF n, k)
              = 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 zero leaf is encountered, the computation stops.

B 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.

B.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

    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 (m, k)
        = (case m
             of (RES a)
                => k a
              | EXC
                => EXC)
    val raise_exception = EXC
    fun handle_exception (m, h)
        = (case m
             of (RES a)
                => RES a
              | EXC
                => h ())
  end
```

Proposition 6 *The type constructor above, together with the above definitions of* `unit` *and* `bind`*, satisfies the three monadic laws.*

Proof: The exception monad is known to be a monad [29]. Alternatively, the monadic laws can be verified by equational reasoning. □

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)
    = handle_exception (eval (t0, e), fn () => eval (t1, e))

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

B.2 A CEK machine with exceptions

Inlining this exception monad in the extended monadic evaluator yields a call-by-value evaluator in exception-oriented style. As in Section 4 we closure-convert, CPS-transform, and defunctionalize the inlined evaluator. The result is a version of the CEK machine with exceptions:

- Source syntax (terms):

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

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

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

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

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

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

B.3 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 `value E -> 'a` and a pair of continuations `(value -> 'a) * (unit -> 'a)`. 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 this double-barreled continuation-passing interpreter 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.

B.4 Summary and conclusion

We have presented an exception monad and an abstract machine that corresponds to a call-by-value monadic evaluator and this monad. The evaluator obtained by inlining the components of the exception monad is 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.

C Combining state and exceptions

As is well known, there are two ways to combine the state and exception monads, giving rise to different semantics [5]. 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

    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 Appendix B 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.

C.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
```

Proposition 7 *The type constructor above, together with the above definitions of* `unit` *and* `bind`*, satisfies the three monadic laws.*

Proof: The combined state and exception monad is known to be a monad [5]. Alternatively, the monadic laws can be verified by equational reasoning. □

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):

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

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

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

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

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

$$\begin{array}{lcl} \text{where}\quad e_{base} & = & e_{empty}[\texttt{succ} \mapsto succ] \\ e_{init} & = & e_{base}[\texttt{get} \mapsto get][\texttt{set} \mapsto set][\texttt{raise} \mapsto raise] \end{array}$$

and s_{init} is the initial state.

C.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 [18, 27]:

```
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 => (case m s
                    of (RES (a, s'))
                       => k a s'
                     | EXC
                       => EXC))
  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 => (case t0 s
                    of (RES (a, s'))
                       => RES (a, s')
                     | EXC
                       => t1 () s))
end
```

Proposition 8 *The type constructor above, together with the above definitions of* `unit` *and* `bind`*, satisfies the three monadic laws.*

Proof: The combined state and exception monad is known to be a monad [5]. Alternatively, the monadic laws can be verified by equational reasoning. □

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):

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

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

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

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

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

$$\begin{array}{lll} \text{where} \quad e_{base} & = & e_{empty}[\texttt{succ} \mapsto succ] \\ e_{init} & = & e_{base}[\texttt{get} \mapsto get][\texttt{set} \mapsto set][\texttt{raise} \mapsto raise] \end{array}$$

and s_{init} is the initial state.

C.3 Summary and conclusion

We have presented two combined monads accounting for state and exceptions and 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.

D Combining stack inspection and exceptions

We specify a combination of the stack-inspection (lifted state) monad of Section 8 and the exception monad of Appendix B. We inline this monad in the monadic evaluator corresponding to the unzipped version of the CM machine from Section 8 and obtain a version of the CM machine extended with exceptions.

D.1 A combined stack-inspection and exception monad

Adding exceptions on top of the stack-inspection monad we obtain a monad which is implemented as a structure with the following signature:

```
signature STACK_INSPECTION_AND_EXCEPTION_MONAD
= sig
    include STACK_INSPECTION_LIFTED_STATE_MONAD

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

Straightforwardly adding exceptions on top of the stack-inspection monad yields the following definitions of the monad type constructor, `unit`, and `bind`, using a type `permission_table` to represent permission tables:

```
datatype 'a lift = LIFT of 'a | BOTTOM
datatype 'a E = RES of 'a | EXC
type state = permission_table list
type 'a monad = state -> (('a * state) lift) E

fun unit a
    = fn s => RES (LIFT (a, s))
fun bind (m, k)
    = (fn s => (case m s
                  of (RES a')
                     => (case a'
                           of (LIFT (a, s1))
                              => k a s1
                            | BOTTOM
                              => RES BOTTOM)
                   | EXC
                     => EXC))
```

In this definition of the monad there are three possible outcomes of a computation. To simplify the definition of `bind`, instead of explicitly layering exceptions on top of the lifting of the stack-inspection monad we use the following isomorphic definition of the monad:

```
datatype 'a lift_E = RES of 'a | EXC | BOTTOM
type state = permission_table list
type 'a monad = state -> ('a * state) lift_E

fun unit a
    = fn s => RES (a, s)
fun bind (m, k)
    = (fn s => (case m s
                  of (RES (a, s1))
                     => k a s1
                   | EXC
                     => EXC
                   | BOTTOM
                     => BOTTOM))
```

Proposition 9 *The above type constructor, together with the above definitions of* `unit` *and* `bind`*, satisfies the three monadic laws.*

Proof: By equational reasoning. □

With the above definition of the monad the definitions of `raise_exception` and `handle_exception` are as follows:

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

Notice that a snapback semantics is used 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 when the exception is handled.

The definition of the monadic operations for manipulating permission stacks are straightforwardly extended to account for exceptions.

In Appendix C we put the raise operation in the initial environment. Here, for diversity value, we add it to the language of Section 8 as a syntactic construct:

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

The monadic evaluator is as in Section 8 with two extra clauses:

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

D.2 An abstract machine for stack inspection and exceptions

Inlining the components of the combined stack-inspection and exception monad in the monadic evaluator of Section 8 and uncurrying the `eval` function and the function space in the data type of expressible values yields an exception-oriented call-by-value evaluator in state-passing style. Closure converting the expressible values, CPS-transforming the evaluator, optimizing the continuation as illustrated in Appendix A to stop immediately in case of security failure, 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):

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

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

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

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

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

As in Appendix B, in the continuation-passing evaluator we could have further exploited the type isomorphism between a sum-expecting continuation and a pair of continuations. Doing so we obtain an abstract machine with two stacks: a regular control stack and a stack of exception handlers.

D.3 Summary and conclusion

We have presented a combined monad accounting for stack inspection and exceptions and an 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.

References

[1] Martín Abadì and Cédric Fournet. Access control based on execution history. In Virgil Gligor and Michael Reiter, editors, *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS'03)*, pages 107–121, San Diego, California, February 2003. Internet Society.

[2] 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.

[3] 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.

[4] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004. Extended version available as the technical report BRICS-RS-04-3.

[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. In Maurice Bruynooghe, editor, *Logic Based Program Synthesis and Transformation, 13th International Symposium, LOPSTR 2003*, number 3018 in Lecture Notes in Computer Science, pages 143–159, Uppsala, Sweden, August 2003. Springer-Verlag.

[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. In Clemens Grelck and Frank Huch, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, Lecture Notes in Computer Science, Lübeck, Germany, September 2004. Springer-Verlag. To appear. Extended version available as the technical report BRICS-RS-03-33.

[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. Extended version available as the technical report BRICS RS-01-23.

[12] Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Technical Report BRICS RS-04-26, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2004. A preliminary version appears in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4.

[13] Olivier Danvy and Ulrik P. Schultz. Lambda-lifting in quadratic time. *Journal of Functional and Logic Programming*, 10(1), July 2004. Available online at `http://danae.uni-muenster.de/lehre/kuchen/JFLP/`.

[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] 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.

[19] 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.

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

[21] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages, second edition*. The MIT Press, 2001.

[22] 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.

[23] 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.

[24] 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.

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

[26] Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

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

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

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

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

[31] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, September 1981.

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

[33] John D. Ramsdell. The tail-recursive SECD machine. *Journal of Automated Reasoning*, 23(1):43–62, July 1999.

[34] 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.

[35] David A. Schmidt. State transition machines for lambda calculus expressions. In Neil D. Jones, editor, *Semantics-Directed Compiler Generation*, number 94 in Lecture Notes in Computer Science, pages 415–440, Aarhus, Denmark, 1980. Springer-Verlag.

[36] 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.

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

[38] 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.

[39] 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-04-28 **Mads Sig Ager, Olivier Danvy, and Jan Midtgaard.** ***A Functional Correspondence between Monadic Evaluators and Abstract Machines for Languages with Computational Effects*****. December 2004. 44 pp. Extended version of an article to appear in *Theoretical Computer Science*.**

RS-04-27 **Gerth Stølting Brodal, Rolf Fagerberg, and Gabriel Moruz.** ***On the Adaptiveness of Quicksort*****. December 2004. 23 pp. To appear in Demetrescu and Tamassia, editors, *Seventh Workshop on Algorithm Engineering and Experiments*, ALENEX '05 Proceedings, 2005.**

RS-04-26 **Olivier Danvy and Lasse R. Nielsen.** ***Refocusing in Reduction Semantics*****. November 2004. iii+44 pp. This report supersedes BRICS report RS-02-04. A preliminary version appears in the informal proceedings of the *Second International Workshop on Rule-Based Programming*, RULE 2001, Electronic Notes in Theoretical Computer Science, Vol. 59.4.**

RS-04-25 **Mayer Goldberg.** ***On the Recursive Enumerability of Fixed-Point Combinators*****. November 2004. 7 pp.**

RS-04-24 **Luca Aceto, Willem Jan Fokkink, Anna Ingólfsdóttir, and Sumit Nain.** ***Bisimilarity is not Finitely Based over BPA with Interrupt*****. October 2004. 30 pp.**

RS-04-23 **Hans Hüttel and Jiří Srba.** ***Recursion vs. Replication in Simple Cryptographic Protocols*****. October 2004. 26 pp. To appear in Vojtas, editor, *31st Conference on Current Trends in Theory and Practice of Informatics*, SOFSEM '05 Proceedings, LNCS, 2005.**

RS-04-22 **Gian Luca Cattani and Glynn Winskel.** ***Profunctors, Open Maps and Bisimulation*****. October 2004. 64 pp. To appear in *Mathematical Structures in Computer Science*.**

RS-04-21 **Glynn Winskel and Francesco Zappa Nardelli.** ***New-HOPLA—A Higher-Order Process Language with Name Generation*****. October 2004. 38 pp.**

RS-04-20 **Mads Sig Ager.** ***From Natural Semantics to Abstract Machines*****. October 2004. 21 pp. Presented at the *International Symposium on Logic-based Program Synthesis and Transformation*, LOPSTR 2004, Verona, Italy, August 26–28, 2004.**