BRICS RS-07-8 Danvy & Millikin: A Simple Application of Lightweight Fusion to Proving the Equivalence of Abstract Machines

Basic Research in Computer Science

A Simple Application of Lightweight Fusion to Proving the Equivalence of Abstract Machines

Olivier Danvy
Kevin Millikin

BRICS Report Series **RS-07-8**

ISSN 0909-0878 **March 2007**

Copyright © 2007, Olivier Danvy & Kevin Millikin.
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
IT-parken, Aabogade 34
DK–8200 Aarhus N
Denmark
Telephone: +45 8942 9300
Telefax: +45 8942 5601
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/07/8/

A Simple Application of Lightweight Fusion to Proving the Equivalence of Abstract Machines

Olivier Danvy and Kevin Millikin
BRICS
Department of Computer Science
University of Aarhus*

March 21, 2007

Abstract

We show how Ohori and Sasano's recent lightweight fusion by fixed-point promotion provides a simple way to prove the equivalence of the two standard styles of specification of abstract machines: (1) as a transition function together with a 'driver loop' implementing the iteration of this transition function; and (2) as a function directly iterating upon a configuration until reaching a final state, if ever. The equivalence hinges on the fact that the latter style of specification is a fused version of the former one. The need for such a simple proof is motivated by our recent work on syntactic correspondences between reduction semantics and abstract machines, using refocusing.

*IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark.
Email: <danvy@brics.dk>, <kmillikin@brics.dk>

Contents

1 Introduction **1**
1.1 Implementation #1 1
1.2 Implementation #2 2
1.3 Question 2
1.4 Our answer 3

2 From Implementation #1 to Implementation #2 **3**

3 From Reduction Semantics to Abstract Machine **4**

1 Introduction

Abstract machines are either

- defined as a transition function together with a ‘driver loop’ implementing the iteration of this transition function; or
- defined directly as a function iterating upon a configuration until reaching a final state, if ever.

For example, consider a recognizer for Dyck words. Dyck words are well-balanced strings of left and right parentheses, which we represent in ML as follows:

```
datatype parenthesis = L | R

type word = parenthesis list
```

For example, the list `[L, L, R, L, R, R]` forms a Dyck word whereas the list `[R, L]` does not.

Dyck words are classically recognized with an abstract machine implementing a push-down automaton. This state-transition system operates iteratively over a given list and a counter reflecting the number of open parentheses seen in the list so far:

```
datatype nat = ZERO | SUCC of nat
```

The machine starts with a given word and a zero counter. At each iteration, one of the following transitions takes place:

- if the list of parentheses is empty and the counter is zero, a final, accepting state is reached;
- if the list of parentheses is empty and the counter is positive, a final, non-accepting state is reached;
- if the first parenthesis is a left one, the tail of the list is taken and the counter is incremented;
- if the first parenthesis is a right one and if the counter is zero, a final, non-accepting state is reached;
- if the first parenthesis is a right one and if the counter is positive, the tail of the list is taken and the counter is decremented.

1.1 Implementation #1

The following ML program implements the transition system above. We define states with a data type, the corresponding transition function as a total function from states to states, and a driver loop as a function iterating the transition function until a final state is reached, if ever:

```
datatype state = FINAL of accepted
               | INTERMEDIATE of configuration
withtype accepted = bool
     and configuration = word * nat
```

```
(*  move : configuration -> state  *)
fun move (nil, ZERO)
    = FINAL true
  | move (nil, SUCC c)
    = FINAL false
  | move (L :: ps, c)
    = INTERMEDIATE (ps, SUCC c)
  | move (R :: ps, ZERO)
    = FINAL false
  | move (R :: ps, SUCC c)
    = INTERMEDIATE (ps, c)

(*  drive : state -> accepted  *)
fun drive (FINAL a)
    = a
  | drive (INTERMEDIATE (ps, c))
    = drive (move (ps, c))

(*  recognize : word -> accepted  *)
fun recognize ps
    = drive (INTERMEDIATE (ps, ZERO))
```

This style of specification is standard in algorithmics. For example, pushdown automata are classically presented in this fashion [9, Chapter 7].

1.2 Implementation #2

The following ML program also implements the transition system above. We directly define an iterative function over the configurations:

```
(*  iterate : configuration -> accepted  *)
fun iterate (nil, ZERO)
    = true
  | iterate (nil, SUCC c)
    = false
  | iterate (L :: ps, c)
    = iterate (ps, SUCC c)
  | iterate (R :: ps, ZERO)
    = false
  | iterate (R :: ps, SUCC c)
    = iterate (ps, c)

(*  recognize : word -> accepted  *)
fun recognize ps
    = iterate (ps, ZERO)
```

This style of specification is standard in semantics. For example, the CEK machine and the Krivine machine are classically defined in this fashion [7, 8]. The Dragon book also presents finite automata in this fashion [1, Figure 3.2.2, page 116].

1.3 Question

How do we know that these two specifications are equivalent?

1.4 Our answer

These two specifications are equivalent because the latter is a 'fused' version of the former, based on Ohori and Sasano's recent work on lightweight fusion [10]. Ohori and Sasano proved the full correctness of the derivation method we apply in the following section.

2 From Implementation #1 to Implementation #2

We consider the following composition:

```
val c0 = fn (ps, c) => drive (move (ps, c))
```

Step 1: Inline the definition of `move` in the composition.

```
val c1 = fn (ps, c) => drive (case (ps, c)
                                of (nil, ZERO)
                                   => FINAL true
                                 | (nil, SUCC c)
                                   => FINAL false
                                 |  (L :: ps, c)
                                   => INTERMEDIATE (ps, SUCC c)
                                 | (R :: ps, ZERO)
                                   => FINAL false
                                 | (R :: ps, SUCC c)
                                   => INTERMEDIATE (ps, c))
```

Step 2: Distribute `drive` to the conditional branches.

```
val c2 = fn (ps, c) => (case (ps, c)
                          of (nil, ZERO)
                             => drive (FINAL true)
                           | (nil, SUCC c)
                             => drive (FINAL false)
                           |  (L :: ps, c)
                             => drive (INTERMEDIATE (ps, SUCC c))
                           | (R :: ps, ZERO)
                             => drive (FINAL false)
                           | (R :: ps, SUCC c)
                             => drive (INTERMEDIATE (ps, c)))
```

Step 3: Simplify by inlining applications of `drive` to known arguments.

```
val c3 = fn (ps, c) => (case s
                          of (nil, ZERO)
                             => true
                           | (nil, SUCC c)
                             => false
                           |  (L :: ps, c)
                             => drive (move (ps, SUCC c))
                           | (R :: ps, ZERO)
                             => false
                           | (R :: ps, SUCC c)
                             => drive (move (ps, c)))
```

Step 4: Use `c3` to define a new (recursive, or more precisely, tail-recursive, i.e., iterative) function `drive_move` equal to `drive o move`.

```
fun drive_move (nil, ZERO)
    = true
  | drive_move (nil, SUCC c)
    = false
  | drive_move (L :: ps, c)
    = drive_move (ps, SUCC c)
  | drive_move (R :: ps, ZERO)
    = false
  | drive_move (R :: ps, SUCC c)
    = drive_move (ps, c)

fun recognize ps
    = drive_move (ps, ZERO)
```

The fused version coincides with the second implementation.

Ohori and Sasano have proved that this fixed-point promotion is correct if `drive` is strict, which it is here. (This kind of condition occurs frequently for fixed points of composite functions [11, Exercise 10.3, page 165].) Their proof is based on a denotational semantics and shows that the denotation before and after fixed-point promotion are equal. Implementation #1 and Implementation #2 are therefore equivalent.

3 From Reduction Semantics to Abstract Machine

The simple proof presented here is directly applicable in our current work on syntactic correspondences between reduction semantics and abstract machines, using refocusing. The idea is as follows.

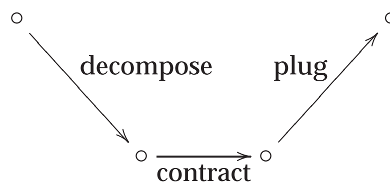
In a reduction semantics, a one-step reduction function is defined as the composition of

1. a total 'decomposition' function from non-value terms to potential redexes and reduction contexts;

2. a partial 'contraction' function[1] mapping actual redexes and their reduction context to a contractum and a reduction context (possibly another one, to account for control effects [6]); and

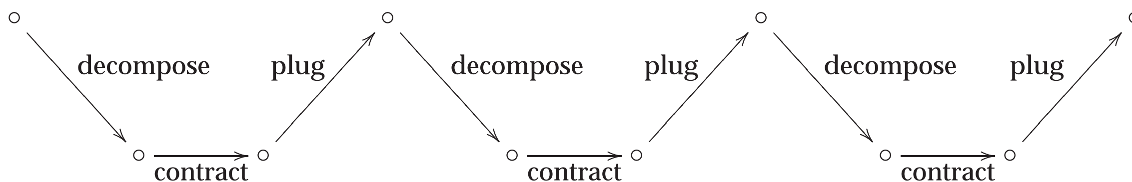
3. a total 'plug' function filling the reduction context with the contractum.

Graphically:

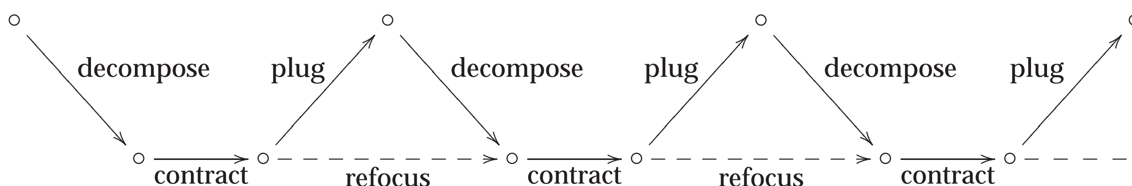


[1]The contraction function is partial because stuck redexes are not contracted.

Evaluation is defined as iterated reduction:



Danvy and Nielsen [5] pointed out that the composition of the two total functions decompose and plug could be fused into a 'refocus' function that continues the decomposition from the current reduction context to the next one, if there is one:



The resulting refocused evaluation function takes the form of Implementation #1:

```
datatype term = ...
datatype context = EMPTY_CONTEXT | ...
datatype value = ...
datatype potential_redex = ...
datatype value_or_decomposition = VAL of value
                                | DEC of potential_redex * context

(*  contract : potential_redex * context -> (term * context) option  *)
(*   refocus :            term * context -> value_or_decomposition   *)

(*  iterate : value_or_decomposition -> value option  *)
fun iterate (VAL v)
    = SOME v
  | iterate (DEC (pr, c))
    = (case contract (pr, c)
         of NONE
            => NONE
          | (SOME (t, c'))
            => iterate (refocus (t, c')))

(*  evaluate : term -> value option  *)
fun evaluate t
    = iterate (refocus (t, EMPTY_CONTEXT))
```

The transition function is `refocus` and the driver loop is `iterate`.

Lightweight fusion of `iterate o refocus` yields a tail-recursive evaluation function in the form of Implementation #2, i.e., an abstract machine such as the CEK machine or the Krivine machine [2–4].

Acknowledgments: Thanks are due to Małgorzata Biernacka and Kristian Støvring for comments. This work is partly supported by the Danish Natural Science Research Council, Grant no. 21-03-0545.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. World Student Series. Addison-Wesley, Reading, Massachusetts, 1986.

[2] Małgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic*, 2006. To appear. Available as the technical report BRICS RS-06-3.

[3] Małgorzata Biernacka and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. Technical Report BRICS RS-06-18, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 2006. To appear in Theoretical Computer Science (extended version).

[4] Olivier Danvy and Kevin Millikin. A rational deconstruction of Landin's J operator. In Andrew Butterfield, Clemens Grelck, and Frank Huch, editors, *Implementation and Application of Functional Languages, 17th International Workshop, IFL'05*, number 4015 in Lecture Notes in Computer Science, pages 55–73, Dublin, Ireland, September 2005. Springer-Verlag. Extended version available as the technical report BRICS RS-06-17 (December 2006).

[5] Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Research 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.

[6] Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, August 1987.

[7] 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.

[8] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 2007. To appear. Available online at `<http://www.pps.jussieu.fr/~krivine/>`.

[9] John C. Martin. *Introduction to Languages and the Theory of Computation*. Programming Language Series. McGraw-Hill International Editions, second edition, 1997.

[10] Atsushi Ohori and Isao Sasano. Lightweight fusion by fixed point promotion. In Matthias Felleisen, editor, *Proceedings of the Thirty-Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 143–154, New York, NY, USA, January 2007. ACM Press.

[11] Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundation of Computing Series. The MIT Press, 1993.

Recent BRICS Report Series Publications

RS-07-8 **Olivier Danvy and Kevin Millikin.** ***A Simple Application of Lightweight Fusion to Proving the Equivalence of Abstract Machines*****. March 2007. ii+6 pp.**

RS-07-7 **Olivier Danvy and Kevin Millikin.** ***Refunctionalization at Work*****. March 2007. ii+16 pp. Invited talk at the 8th International Conference on Mathematics of Program Construction, MPC '06.**

RS-07-6 **Olivier Danvy, Kevin Millikin, and Lasse R. Nielsen.** ***On One-Pass CPS Transformations*****. March 2007. ii+19 pp. Theoretical Pearl to appear in the** ***Journal of Functional Programming*****. Revised version of BRICS RS-02-3.**

RS-07-5 **Luca Aceto, Silvio Capobianco, and Anna Ingólfsdóttir.** ***On the Existence of a Finite Base for Complete Trace Equivalence over BPA with Interrupt*****. February 2007. 26 pp.**

RS-07-4 **Kristian Støvring and Søren B. Lassen.** ***A Complete, Co-Inductive Syntactic Theory of Sequential Control and State*****. February 2007. 36 pp. Appears in the proceedings of POPL 2007, p. 161–172.**

RS-07-3 **Luca Aceto, Willem Jan Fokkink, and Anna Ingólfsdóttir.** ***Ready To Preorder: Get Your BCCSP Axiomatization for Free!*** **February 2007. 37 pp.**

RS-07-2 **Luca Aceto and Anna Ingólfsdóttir.** ***Characteristic Formulae: From Automata to Logic*****. January 2007. 18 pp.**

RS-07-1 **Daniel Andersson.** ***HIROIMONO is NP-complete*****. January 2007. 8 pp.**

RS-06-19 **Michael David Pedersen.** ***Logics for The Applied π Calculus*****. December 2006. viii+111 pp.**

RS-06-18 **Małgorzata Biernacka and Olivier Danvy.** ***A Syntactic Correspondence between Context-Sensitive Calculi and Abstract Machines*****. dec 2006. iii+39 pp. Extended version of an article to appear in TCS. Revised version of BRICS RS-05-22.**

RS-06-17 **Olivier Danvy and Kevin Millikin.** ***A Rational Deconstruction of Landin's J Operator*****. December 2006. ii+37 pp. Revised version of BRICS RS-06-4. A preliminary version appears in the proceedings of IFL 2005, LNCS 4015:55–73.**