BRICS

Basic Research in Computer Science

From Interpreter to Logic Engine by Defunctionalization

Biernacki Dariusz
Danvy Olivier

BRICS Report Series RS-03-25
ISSN 0909-0878 June 2003

BRICS RS-03-25 Dariusz & Olivier: From Interpreter to Logic Engine by Defunctionalization

Recent BRICS Report Series Publications

RS-03-25 Biernacki Dariusz and Danvy Olivier. *From Interpreter to Logic Engine by Defunctionalization*. June 2003. 13 pp. Presented at the International Symposium on Logic Based Program Development and Transformation, LOPSTR 2003, Uppsala, Sweden, August 25–27, 2003.

RS-03-24 Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. *A Functional Correspondence between Call-by-Need Evaluators and Lazy Abstract Machines*. June 2003. 13 pp.

RS-03-23 Korovin Margarita. *Recent Advances in Σ-Definability over Continuous Data Types*. June 2003. 26 pp.

RS-03-22 Ivan B. Damgård and Mads J. Jurik. *Scalable Key-Escrow*. May 2003. 15 pp.

RS-03-21 Ulrich Kohlenbach. *Some Logical Metatheorems with Applications in Functional Analysis*. May 2003. 55 pp. Slighly revised and extended version to appear in *Transactions of the American Mathematical Society*.

RS-03-20 Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. *Fast Partial Evaluation of Pattern Matching in Strings*. May 2003. 16 pp. Final version to appear in Leuschel, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '03 Proceedings, 2003. This report supersedes the earlier BRICS report RS-03-11.

RS-03-19 Christian Kirkegaard, Anders Møller, and Michael I. Schwartzbach. *Static Analysis of XML Transformations in Java*. May 2003. 29 pp.

RS-03-18 Bartek Klin and Paweł Sobociński. *Syntactic Formats for Free: An Abstract Approach to Process Equivalence*. April 2003. 41 pp.

RS-03-17 Luca Aceto, Jens Alsted Hansen, Anna Ingólfsdóttir, Jacob Johnsen, and John Knudsen. *The Complexity of Checking Consistency of Pedigree Information and Related Problems*. March 2003. 31 pp. This paper supersedes BRICS Report RS-02-42.

RS-03-16 Ivan B. Damgård and Mads J. Jurik. *A Length-Flexible Threshold Cryptosystem with Applications*. March 2003. 31 pp.

From Interpreter to Logic Engine by Defunctionalization *

Dariusz Biernacki and Olivier Danvy

BRICS **
Department of Computer Science
University of Aarhus
Building 540, Ny Munkegade, DK-8000 Aarhus C, Denmark
E-mail: {dabi,danvy}@brics.dk
Home pages: http://www.brics.dk/~{dabi,danvy}

Abstract. Starting from a continuation-based interpreter for a simple logic programming language, propositional Prolog with cut, we derive the corresponding logic engine in the form of an abstract machine. The derivation originates in previous work (our article at PPDP 2003) where it was applied to the lambda-calculus. The key transformation here is Reynolds's defunctionalization that transforms a tail-recursive, continuation-passing interpreter into a transition system, i.e., an abstract machine. Similar denotational and operational semantics were studied by de Bruin and de Vink in previous work (their article at TAPSOFT 1989), and we compare their study with our derivation. Additionally, we present a direct-style interpreter of propositional Prolog expressed with control operators for delimited continuations.

1 Introduction

In previous work [2], we presented a derivation from interpreter to abstract machine that makes it possible to connect known λ-calculus interpreters to known abstract machines for the λ-calculus, as well as to discover new ones. The goal of this work is to test this derivation on a programming language other than the λ-calculus. Our pick here is a simple logic programming language, propositional Prolog with cut (Section 2). We present its abstract syntax, informal semantics, and computational model, which we base on success and failure continuations (Section 3). We then specify an interpreter for propositional Prolog in a generic and parameterized way that leads us to a logic engine. This logic engine is a transition system that we obtain by defunctionalizing the success and failure continuations (Section 4). We also present and analyze a direct-style interpreter for propositional Prolog (Appendix A).

* Presented at the 2003 International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2003), August 2003, Uppsala, Sweden.
** Basic Research in Computer Science (http://www.brics.dk), funded by the Danish National Research Foundation.

Prerequisites: We expect a passing familiarity with the notions of success and failure continuations as well as with Standard ML and its module language.

As for defunctionalization, it originates in Reynolds's seminal article on definitional interpreters for higher-order programming languages [21]. The point of defunctionalization is to transform a higher-order program with functional values into a first-order program. Before defunctionalization, in a higher-order program, the inhabitants of a function space are instances of anonymous lambda-abstractions. Defunctionalizing a program amounts to enumerating these lambda-abstractions in a sum type: each function introduction (i.e., lambda-abstraction) is replaced by the corresponding constructor holding the values of the free variables of this lambda-abstraction, and each function elimination (i.e., application) is replaced by a case dispatch. After defunctionalization, the inhabitants of the sum type are instances of the constructors.

For example, let us defunctionalize the factorial function when it is expressed in continuation-passing style:

```
(*  fac_c : int * (int -> 'a) -> 'a
fun fac_c (0, k)
    = k 1
  | fac_c (n, k)
    = fac_c (n - 1, fn v => k (n * v))

(*  main : int -> int  *)
fun main n
    = fac_c (n, fn v => v)
```

We consider the whole program (i.e., both `main` and `fac_c`). Therefore the polymorphic type `'a`, i.e., the domain of answers, is instantiated to `int`. The candidate function space for defunctionalization is that of the continuation, `int -> int`. Its inhabitants are instances of two lambda-abstractions: the initial continuation in `main` with no free variables, and the intermediate continuation in the induction case of `fac_c` with two free variables: `n` and `k`. The corresponding data type has therefore two constructors:

```
datatype cont = CONT0
              | CONT1 of int * cont

(*  apply_cont : cont * int -> int  *)
fun apply_cont (CONT0, v)
    = v
  | apply_cont (CONT1 (n, k), v)
    = apply_cont (k, n * v)
```

Correspondingly, the apply function associated to the data type interprets each of these constructors according to the initial continuation and the intermediate continuation. In the defunctionalized program, the continuations are replaced by the constructors, and the applications of the continuations are replaced by a call to `apply_cont`:


```
(*  fac_c : int * cont -> int  *)
fun fac_c (0, k)
    = apply_cont (k, 1)
  | fac_c (n, k)
    = fac_c (n - 1, CONT1 (n, k))

(*  main : int -> int  *)
fun main n
    = fac_c (n, CONT0)
```

The resulting program is first-order, all its calls are tail calls, and all computations in the actual parameters are elementary. It is therefore a transition system in the formal sense. Both `main` and `fac_c`, together with their actual parameters, form configurations and their ML definitions specify a transition relation, as expressed in the following table. The top transition specifies the initial state and the bottom transition specifies the terminating configurations. The machine consists of two mutually recursive transition functions; the first one operates over pairs of integers, and the second one operates over a stack of integers and an integer:

$n \Rightarrow \langle n,\ nil\rangle_{fac}$
$\langle 0,\ k\rangle_{fac} \Rightarrow \langle k,\ 1\rangle_{app}$ $\langle n,\ k\rangle_{fac} \Rightarrow \langle n-1,\ n :: k\rangle_{fac}$
$\langle n :: k,\ v\rangle_{app} \Rightarrow \langle k,\ n \times v\rangle_{app}$
$\langle nil,\ v\rangle_{app} \Rightarrow v$

Accordingly, the result of defunctionalizing a continuation-passing interpreter is also a transition system, i.e., an abstract machine. We used this property in our work on the λ-calculus [2], and we use it here for propositional Prolog.

2 Propositional Prolog

The abstract syntax of propositional Prolog reads as follows:

```
structure Source
= struct
    type ide = string
    datatype atom = IDE of ide
                  | OR of goal * goal
                  | CUT
                  | FAIL
         and goal = GOAL of atom list
    datatype clause = CLAUSE of ide * goal
    datatype program = PROGRAM of clause list
  end
```

A program consists of a list of clauses. A clause consists of an identifier (the head of the clause) and a goal (the body of the clause). A goal is a list of atoms; an empty list represents the logical value 'true' and a non-empty list of atoms represents their conjunction. Each atom is either an identifier, the disjunction of two goals, the cut operator, or the fail operator.

The intuitive semantics of the language is standard. Given a Prolog program and a goal, we try to verify whether the goal follows from the program in the sense of propositional logic, i.e., in terms of logic programming, whether the SLD-resolution algorithm for this goal and this program stops with the empty clause. If it does, then the answer is positive; if it stops with one or more subgoals still waiting resolution, then the answer is negative. Here the unification algorithm consists in looking up the clause with a specified head in the program.

An atom can be a disjunction of two goals, and therefore if a chosen body does not lead to the positive answer, the other disjunct is tried, using backtracking. Backtracking can also be used to find all possible solutions in the resolution tree, which in case of propositional Prolog amounts to counting the positive answers. Two operators provide additional control over the traversal of the resolution tree: the cut operator removes some of the potential paths and the fail operator makes the current goal unsatisfiable, which triggers backtracking.

3 A generic interpreter for propositional Prolog

To account for the backtracking necessary to implement resolution, we use success and failure continuations. A failure continuation is a parameterless function (i.e., a thunk) yielding a final answer. A success continuation maps a failure continuation to a final answer. The initial success continuation is applied if a solution has been found. The initial failure continuation is applied if no solution has been found. In addition, to account for the cut operator, we pass a cut continuation, i.e., a cached failure continuation. As usual with continuations, the domain of answers is left unspecified.

3.1 A generic notion of answers and results

We specify answers with an ML signature. An answer is a type that comes together with an initial success continuation, an initial failure continuation, and an initial cut continuation. The signature also declares a type of results and an extraction function mapping a (generic) answer to a (specific) result.

```
signature ANSWER
= sig
    type answer
    val sc_init : (unit -> answer) -> answer
    val fc_init : unit -> answer
    val cc_init : unit -> answer

    type result
    val extract : answer -> result
  end
```


3.2 Specific answers and results

We consider two kinds of answers: the first solution, if any, and the total number of solutions.

The first solution: This notion of answer is the simplest to define. Both `answer` and `result` are defined as the type of booleans and `extract` is the identity function. The initial success continuation ignores the failure continuation and yields `true`, whereas both the initial failure continuation and the initial cut continuation yield `false`.

```
structure Answer_first : ANSWER
= struct
    type answer = bool
    fun sc_init fc = true
    fun fc_init () = false
    fun cc_init () = false

    type result = bool
    fun extract a = a
  end
```

The number of solutions: This notion of answer is more delicate. One could be tempted to define `answer` as the type of integers, but the resulting implementation would no longer be tail recursive.[1] Instead, we use an extra layer of continuations: We define `answer` as the type of functions from integers to integers, `result` as the type of integers, and `extract` as a function triggering the whole resolution by applying an answer to the initial count, 0. The initial success continuation takes note of an intermediate success by incrementing the current count and activating the failure continuation. The initial failure continuation and the initial cut continuation are passed the final count and return it.

```
structure Answer_how_many : ANSWER
= struct
    type answer = int -> int
    fun sc_init fc = (fn m => fc () (m+1))
    fun fc_init () = (fn m => m)
    fun cc_init () = (fn m => m)

    type result = int
    fun extract a = a 0
  end
```

3.3 The generic interpreter, semi-compositionally

We define a generic interpreter for propositional Prolog, displayed in Figure 1, as a recursive descent over the source syntax, parameterized by a notion

[1] In "`fun sc_init fc = 1 + (fc ())`", the call to `fc` is not a tail call.

of answers. In `run_goal`, an empty list of atoms is interpreted as 'true', and accordingly, the success continuation is activated. A non-empty list of atoms is sequentially interpreted by `run_goal'` by extending the success continuation; this interpretation singles out the last atom in a properly tail-recursive manner. An identifier is interpreted either by failing if it is not the head of any clause in the program, or by resolving the corresponding goal with the cut continuation replaced with the current failure continuation. A disjunction of two goals is speculatively interpreted by extending the failure continuation. The cut operator is interpreted by replacing the failure continuation with the cut continuation. The fail operator is interpreted as 'false', and accordingly, the failure continuation is activated.

This interpreter is not compositional (in the sense of denotational semantics) because `g`, in the interpretation of identifiers, does not denote a proper subpart of the denotation of `l`. The interpreter, however, is semi-compositional in Jones's sense [15, 16], i.e., `g` denotes a proper subpart of the source program. (To make the interpreter compositional, one can follow the tradition of denotational semantics and use an environment mapping an identifier to a function that either evaluates the goal denoted by the identifier or calls the failure continuation. The environment is threaded in the interpreter instead of the program. The resulting ML interpreter represents the valuation function of a denotational semantics of propositional Prolog.)

3.4 A specific interpreter computing the first solution

A specific interpreter computing the first solution, if any, is obtained by instantiating `mkProcessor` with the corresponding notion of answers:

```
structure Prolog_first = mkProcessor (structure A = Answer_first)
```

3.5 A specific interpreter computing the number of solutions

A specific interpreter computing the number of solutions is also obtained by instantiating `mkProcessor` with the corresponding notion of answers:

```
structure Prolog_how_many = mkProcessor (structure A = Answer_how_many)
```

Appendix A contains a direct-style counterpart of the interpreter (without cut) computing the number of solutions.

4 Two abstract machines for propositional Prolog

We successively consider each of the specific Prolog interpreters of Sections 3.4 and 3.5, and we defunctionalize their continuations. As already illustrated in Section 1 with the factorial program, in each case, the result is an abstract machine. Indeed the interpreters are in continuation-passing style, and thus:


```
functor mkProcessor (structure A : ANSWER)
= struct
    fun run_goal (nil, p, sc, fc, cc)
        = sc fc
      | run_goal (a :: g, p, sc, fc, cc)
        = run_goal' (a, g, p, sc, fc, cc)
    and run_goal' (a, nil, p, sc, fc, cc)
        = run_atom (a, p, sc, fc, cc)
      | run_goal' (a, a' :: g, p, sc, fc, cc)
        = run_atom (a, p,
                    fn fc' => run_goal' (a', g, p, sc, fc', cc),
                    fc, cc)
    and run_atom (Syntax.IDE l, p, sc, fc, cc)
        = (case lookup (l, p)
             of NONE
                => fc ()
              | SOME (Syntax.GOAL g)
                => run_goal (g, p, sc, fc, fc))
      | run_atom (Syntax.OR (Syntax.GOAL g1, Syntax.GOAL g2),
                  p, sc, fc, cc)
        = run_goal (g1, p, sc,
                    fn () => run_goal (g2, p, sc, fc, cc),
                    cc)
      | run_atom (Syntax.CUT, p, sc, fc, cc)
        = sc cc
      | run_atom (Syntax.FAIL, p, sc, fc, cc)
        = fc ()

    fun execute (Syntax.GOAL g, p)
        = run_goal (g, p, A.sc_init, A.fc_init, A.cc_init)
  end
```

Fig. 1. A generic interpreter for propositional Prolog

- all their calls are tail calls, and therefore they can run iteratively; and
- all their subcomputations (i.e., the computation of their actual parameters) are elementary.

4.1 The first solution

The abstract machine is defined as the transition system shown in Figure 2. The top part specifies the initial state and the bottom part specifies the terminating configurations. The machine consists of three mutually recursive transition functions, two of which operate over a quintuple and one over a six-element tuple. The quintuple consists of the goal, the program, the (defunctionalized) success

$$
\begin{array}{rcl}
\langle g,\ p\rangle & \Rightarrow & \langle g,\ p,\ nil,\ nil,\ nil\rangle_{goal} \\
\hline
\langle nil,\ p,\ (a,\ g,\ p',\ cc') :: sc,\ fc,\ cc\rangle_{goal} & \Rightarrow & \langle a,\ g,\ p',\ sc,\ fc,\ cc'\rangle_{seq} \\
\langle a :: g,\ p,\ sc,\ fc,\ cc\rangle_{goal} & \Rightarrow & \langle a,\ g,\ p,\ sc,\ fc,\ cc\rangle_{seq} \\
\hline
\langle a,\ nil,\ p,\ sc,\ fc,\ cc\rangle_{seq} & \Rightarrow & \langle a,\ p,\ sc,\ fc,\ cc\rangle_{atom} \\
\langle a,\ a' :: g,\ p,\ sc,\ fc,\ cc\rangle_{seq} & \Rightarrow & \langle a,\ p,\ (a',\ g,\ p,\ sc,\ cc) :: sc,\ fc,\ cc\rangle_{atom} \\
\hline
\langle \mathtt{IDE}\ l,\ p,\ sc,\ fc,\ cc\rangle_{atom} & \Rightarrow & \langle g,\ p,\ sc,\ fc,\ fc\rangle_{goal} \\
 & & \text{if } lookup\,(l) \text{ succeeds with } g \\
\langle \mathtt{IDE}\ l,\ p,\ sc,\ (g,\ p,\ sc',\ cc') :: fc,\ cc\rangle_{atom} & \Rightarrow & \langle g,\ p',\ sc',\ fc,\ cc'\rangle_{goal} \\
 & & \text{if } lookup\,(l) \text{ fails} \\
\langle \mathtt{OR}\,(g_1,\ g_2),\ p,\ sc,\ fc,\ cc\rangle_{atom} & \Rightarrow & \langle g_1,\ p,\ sc,\ (g_2,\ p,\ sc,\ cc) :: fc,\ cc\rangle_{goal} \\
\langle \mathtt{CUT},\ p,\ (a,\ g,\ p',\ cc') :: sc,\ fc,\ cc\rangle_{atom} & \Rightarrow & \langle a,\ g,\ p',\ sc,\ fc,\ cc'\rangle_{seq} \\
\langle \mathtt{FAIL},\ p,\ sc,\ (g,\ p',\ sc',\ cc') :: fc,\ cc\rangle_{atom} & \Rightarrow & \langle g,\ p',\ sc',\ fc,\ cc'\rangle_{goal} \\
\hline
\langle nil,\ p,\ nil,\ fc,\ cc\rangle_{goal} & \Rightarrow & true \\
\langle \mathtt{IDE}\ l,\ p,\ sc,\ nil,\ cc\rangle_{atom} & \Rightarrow & false,\ \text{if } lookup\,(l) \text{ fails} \\
\langle \mathtt{FAIL},\ p,\ sc,\ nil,\ cc\rangle_{atom} & \Rightarrow & false \\
\langle \mathtt{CUT},\ p,\ nil,\ fc,\ cc\rangle_{atom} & \Rightarrow & true
\end{array}
$$

Fig. 2. An abstract machine computing the first solution

continuation, the (defunctionalized) failure continuation and the cut continuation (register caching a previous failure continuation). The six-element tuple additionally has the first atom of the goal as its first element.

4.2 The number of solutions

This abstract machine is displayed in Figure 3 and is similar to the previous one, but operates over a six- and seven-element tuples. The extra component is the counter.

5 Related work and conclusion

In previous work [2, 3, 8], we presented a derivation from interpreter to abstract machine, and we were curious to see it applied to something else than a functional programming language. The present paper reports its application to a logic

$$
\begin{array}{rcl}
\langle g,\ p\rangle & \Rightarrow & \langle g,\ p,\ nil,\ nil,\ nil,\ 0\rangle_{goal} \\
\hline
\langle nil,\ p,\ nil,\ (g,\ p',\ sc,\ cc') :: fc,\ cc,\ m\rangle_{goal} & \Rightarrow & \langle g,\ p',\ sc,\ fc,\ cc',\ m+1\rangle_{goal} \\
\langle nil,\ p,\ (a,\ g,\ p',\ cc') :: sc,\ fc,\ cc,\ m\rangle_{goal} & \Rightarrow & \langle a,\ g,\ p',\ sc,\ fc,\ cc',\ m\rangle_{seq} \\
\langle a :: g,\ p,\ sc,\ fc,\ cc,\ m\rangle_{goal} & \Rightarrow & \langle a,\ g,\ p,\ sc,\ fc,\ cc,\ m\rangle_{seq} \\
\hline
\langle a,\ nil,\ p,\ sc,\ fc,\ cc,\ m\rangle_{seq} & \Rightarrow & \langle a,\ p,\ sc,\ fc,\ cc,\ m\rangle_{atom} \\
\langle a,\ a' :: g,\ p,\ sc,\ fc,\ cc,\ m\rangle_{seq} & \Rightarrow & \langle a,\ p,\ (a',\ g,\ p,\ sc,\ cc) :: sc,\ fc,\ cc,\ m\rangle_{atom} \\
\hline
\langle \mathtt{IDE}\ l,\ p,\ sc,\ fc,\ cc,\ m\rangle_{atom} & \Rightarrow & \langle g,\ p,\ sc,\ fc,\ fc,\ m\rangle_{goal} \\
 & & \text{if } lookup\,(l) \text{ succeeds with } g \\
\langle \mathtt{IDE}\ l,\ p,\ sc,\ (g,\ p,\ sc',\ cc') :: fc,\ cc,\ m\rangle_{atom} & \Rightarrow & \langle g,\ p',\ sc',\ fc,\ cc',\ m\rangle_{goal} \\
 & & \text{if } lookup\,(l) \text{ fails} \\
\langle \mathtt{OR}\,(g_1,\ g_2),\ p,\ sc,\ fc,\ cc,\ m\rangle_{atom} & \Rightarrow & \langle g_1,\ p,\ sc,\ (g_2,\ p,\ sc,\ cc) :: fc,\ cc,\ m\rangle_{goal} \\
\langle \mathtt{CUT},\ p,\ (a,\ g,\ p',\ cc') :: sc,\ fc,\ cc,\ m\rangle_{atom} & \Rightarrow & \langle a,\ g,\ p',\ sc,\ fc,\ cc',\ m\rangle_{seq} \\
\langle \mathtt{CUT},\ p,\ nil,\ (g,\ p',\ sc,\ cc') :: fc,\ cc,\ m\rangle_{atom} & \Rightarrow & \langle g,\ p',\ sc,\ fc,\ cc',\ m+1\rangle_{goal} \\
\langle \mathtt{FAIL},\ p,\ sc,\ (g,\ p',\ sc',\ cc') :: fc,\ cc,\ m\rangle_{atom} & \Rightarrow & \langle g,\ p',\ sc',\ fc,\ cc',\ m\rangle_{goal} \\
\hline
\langle nil,\ p,\ nil,\ nil,\ cc,\ m\rangle_{goal} & \Rightarrow & m+1 \\
\langle \mathtt{FAIL},\ p,\ sc,\ nil,\ cc,\ m\rangle_{atom} & \Rightarrow & m \\
\langle \mathtt{CUT},\ p,\ nil,\ nil,\ cc,\ m\rangle_{atom} & \Rightarrow & m+1 \\
\langle \mathtt{IDE}\ l,\ p,\ sc,\ nil,\ cc,\ m\rangle_{atom} & \Rightarrow & m,\ \text{if } lookup\,(l) \text{ fails}
\end{array}
$$

Fig. 3. An abstract machine computing the number of solutions

programming language, propositional Prolog. In its entirety, the derivation consists of closure conversion, transformation into continuation-passing style (CPS), and defunctionalization. Closure conversion ensures that any higher-order values are made first-order.[2] The CPS transformation makes the flow of control of the interpreter manifest as a continuation. Defunctionalization materializes the flow of control as a first-order data structure. In the present case, propositional Prolog is a first-order language and the interpreter we consider is already in continuation-passing style (cf. Appendix A). Therefore the derivation reduces to defunctionalization. The result is a simple logic engine, i.e., mutually recursive and first-order transition functions. It was derived, not invented, and so, for example, its two stacks arise as defunctionalized continuations. Similarly, it is properly tail recursive since the interpreter is already properly tail recursive.

[2] Closures, for example, are used to implement higher-order logic programming [6].

Since the correctness of defunctionalization has been established [5, 20], the correctness of the logic engine is a corollary of the correctness of the original interpreter.

Our closest related work is de Bruin and de Vink's continuation semantics for Prolog with cut [11]:

- de Bruin and de Vink present a denotational semantics with success and failure continuations; their semantics is (of course) compositional, and comparable to the compositional interpreter outlined in Section 3.3. The only difference is that their success continuations expect both a failure continuation and a cut continuation, whereas our success continuations expect only a failure continuation. Analyzing the control flow of the corresponding interpreter, we have observed that the cut continuation is the same at the definition point and at the use point of a success continuation. Therefore, there is actually no need to pass cut continuations to success continuations.
- de Bruin and de Vink also present an operational semantics, and prove it equivalent to their denotational semantics. In contrast, we defunctionalized the interpreter corresponding to a denotational semantics into an interpreter corresponding to an operational semantics. We also "refunctionalized" the interpreter corresponding to de Bruin and de Vink's operational semantics, and we observed that in the resulting interpreter (which corresponds to a denotational semantics), success continuations are not passed cut continuations.

Designing abstract machines is a favorite among functional programmers [12]. Unsurprisingly, this is also the case among logic programmers, for example, with Warren's abstract machine [4]. Just as unsurprisingly, functional programmers use functional programming languages as their meta-language and logic programmers use logic programming languages as their meta-language. For example, Kursawe showed how to "invent" Prolog machines out of logic-programming considerations [18]. The goal of our work here was more modest in that we did not aim at a high-performance implementation: we simply aimed to test an interpreter-to-abstract-machine derivation that works well for the λ-calculus. The logic engine we obtain is a basic but plausible model of computation. Its chief illustrative virtue is to show that the representation of a denotational semantics can be mechanically defunctionalized into the representation of an operational semantics (and, actually, vice versa). It also shows that proper tail recursion and the two control stacks did not need to be invented—they were already present in the original interpreter.

An alternative to deriving an abstract machine from an interpreter is to factor this interpreter into a compiler and a virtual machine, using, e.g., Wand's combinator-based compiler derivation [23], Jørring and Scherlis's staging transformations [17], Hannan's pass-separation approach [14], or more generally the binding-time separation techniques of partial evaluation [16, 19]. We are currently experimenting with a such a factorization technique to stage our Prolog interpreter into a byte-code compiler and a virtual machine executing this byte code [1].

Acknowledgments: We are grateful to Mads Sig Ager, Malgorzata Biernacka, Jan Midtgaard, and the anonymous referees for their comments. This work is supported by the ESPRIT Working Group APPSEM II (`http://www.appsem.org`).

A A direct-style interpreter for Prolog

The interpreter of Section 3 is in continuation-passing style to account for the backtracking necessary to implement resolution. The failure continuation could be eliminated by transforming the interpreter into direct style [7]. The success continuation, however, would remain. Because it is used non tail-recursively in the clause for disjunctions, it is what is technically called a *delimited* continuation (in contrast to the usual unlimited continuations of denotational semantics [22]). Transforming the interpreter into direct style requires control operators for delimited continuations such as shift and reset [9, 10, 13].

Figure 4 presents such a direct-style interpreter for Propositional Prolog without cut, counting the number of solutions. CPS-transforming this interpreter

```
fun run_goal (nil, p, m)
    = m
  | run_goal (a :: g, p, m)
    = run_goal' (a, g, p, m)
and run_goal' (a, nil, p, m)
    = run_atom (a, p, m)
  | run_goal' (a, a' :: g, p, m)
    = let val m' = run_atom (a, p, m)
      in run_goal' (a', g, p, m')
      end
and run_atom (Syntax.FAIL, p, m)
    = shift (fn sc => m)
  | run_atom (Syntax.IDE i, p, m)
    = (case lookup (i, p)
         of NONE
            => shift (fn sc => m)
          | SOME (Syntax.GOAL g)
            => run_goal (g, p, m))
  | run_atom (Syntax.OR (Syntax.GOAL g1, Syntax.GOAL g2), p, m)
    = shift (fn sc => let val m' = sc (run_goal (g1, p, m))
                      in sc (run_goal (g2, p, m'))
                      end)

fun execute (Syntax.GOAL g, Syntax.PROGRAM p)
    = reset (fn () => let val m = run_goal (g, p, 0)
                      in m + 1
                      end)
```

Fig. 4. A direct-style interpreter for propositional Prolog

once makes the success continuation appear. CPS-transforming the result makes the failure continuation appear, and yields the interpreter of Section 3.5 (minus cut). Defunctionalizing this interpreter yields the abstract machine of Section 4.2 (minus cut).

The reset control operator delimits control. Any subsequent use of the shift control operator will capture a delimited continuation that can be composed; this delimited continuation is the success continuation. Conjunction, in `run_goal'`, is implemented by function composition. Failure, in `run_atom`, is implemented by capturing the current success continuation and not applying it. Disjunction, in `run_atom`, is implemented by capturing the current success continuation and applying it twice. This interpreter is properly tail recursive, which is achieved by the two functions `run_goal` and `run_goal'` that single out the last atom in a goal.

References

1. Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From interpreter to compiler and virtual machine: a functional derivation. Technical Report BRICS RS-03-14, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, March 2003.
2. Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19. ACM Press, August 2003.
3. Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. Technical Report BRICS RS-03-24, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 2003. Accepted for publication in Information Processing Letters.
4. Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* The MIT Press, 1991.
5. Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Design and correctness of program transformations based on control-flow analysis. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001*, number 2215 in Lecture Notes in Computer Science, Sendai, Japan, October 2001. Springer-Verlag.
6. Weidong Chen, Michael Kifer, and David S. Warren. Hilog: A foundation for higher-order logic programming. *The Journal of Logic Programming*, 15(3):187–230, February 1993.
7. Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.
8. Olivier Danvy. A rational deconstruction of Landin's SECD machine. Technical Report BRICS RS-03-33, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, October 2003.
9. Olivier Danvy and Andrzej Filinski. Abstracting control. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.

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. Arie de Bruin and Erik P. de Vink. Continuation semantics for Prolog with cut. In Josep Díaz and Fernando Orejas, editors, *TAPSOFT'89: Proceedings of the International Joint Conference on Theory and Practice of Software Development*, number 351 in Lecture Notes in Computer Science, pages 178–192, Barcelona, Spain, March 1989. Springer-Verlag.
12. Stephan Diehl, Pieter Hartel, and Peter Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16:739–751, 2000.
13. 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.
14. John Hannan. Operational semantics directed machine architecture. *ACM Transactions on Programming Languages and Systems*, 16(4):1215–1247, 1994.
15. Neil D. Jones. *Computability and Complexity from a Programming Perspective.* Foundations of Computing. The MIT Press, 1997.
16. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International, London, UK, 1993. Available online at http://www.dina.kvl.dk/ sestoft/pebook/.
17. Ulrik Jørring and William L. Scherlis. Compilers and staging transformations. In Mark Scott Johnson and Ravi Sethi, editors, *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 86–96, St. Petersburg, Florida, January 1986. ACM Press.
18. Peter Kursawe. How to invent a Prolog machine. *New Generation Computing*, 5(1):97–114, 1987.
19. Torben Æ. Mogensen. Separating binding times in language specifications. In Joseph E. Stoy, editor, *Proceedings of the Fourth International Conference on Functional Programming and Computer Architecture*, pages 14–25, London, England, September 1989. ACM Press.
20. Lasse R. Nielsen. A denotational investigation of defunctionalization. Technical Report BRICS RS-00-47, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 2000.
21. 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.
22. Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, 13(1/2):135–152, 2000. Reprint of the technical monograph PRG-11, Oxford University Computing Laboratory (1974), with a foreword.
23. Mitchell Wand. Semantics-directed machine architecture. In Richard DeMillo, editor, *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 234–241. ACM Press, January 1982.