Recent BRICS Report Series Publications


From Interpreter to Logic Engine by Defunctionalization

Biernacki Dariusz
Danvy Olivier
From Interpreter to Logic Engine by Defunctionalization

Dariusz Bieracki and Olivier Danvy
BRICS
Department of Computer Science
University of Aarhus
Building 540, Ny Munkegade, DK-8000 Aarhus C, Denmark
E-mail: [dab1, danvy]@brics.dk
Home pages: http://www.brics.dk/~[dab1, 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 Bruijn and de Vink in previous work (their article at TAPSOFT 1999), 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).

** Basic Research in Computer Science (http://www.brics.dk), funded by the Danish National Research Foundation.
2 Propositional Prolog

The abstract syntax of propositional Prolog, reads as follows:

```
<table>
<thead>
<tr>
<th>Source</th>
<th>Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>type_id</td>
<td>struct</td>
</tr>
<tr>
<td>data_type</td>
<td>string</td>
</tr>
<tr>
<td>data_type</td>
<td>int</td>
</tr>
<tr>
<td>data_type</td>
<td>rule</td>
</tr>
<tr>
<td>data_type</td>
<td>clause</td>
</tr>
</tbody>
</table>
```

Accordingly, the result of defunctionalizing a continuation-passing interpreter is also a translation system, in an abstract machine. We used this property in our work on the Lambda calculus. And we use it here for propositional Prolog.

**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 definition-rewriting, it originates in Reynolds's seminal article on definition-rewriting [8].

**Definition-Rewriting:**

We consider the following continuation-passing abstraction function: 

\[ \text{defab}(x, f) = \lambda k. f(x, k) \]

This function takes a function \( f \) and a continuation \( k \) and applies \( f \) to \( x \) and the result to \( k \) as a continuation.

**Prolog Expressions:**

We consider the following Prolog expression:

\[ \text{factorial}(n) \]

This expression computes the factorial of \( n \).

**Compiled Abstract Machine:**

We consider the following compiled abstract machine:

\[ \text{apply} \cdot \text{cost} \cdot \text{int} \cdot \text{int} \rightarrow \text{int} \]

This machine applies a continuation to an integer and returns the result.

**Compiled Machine:**

We consider the following compiled machine:

\[ \text{apply} \cdot \text{cost} \cdot \text{int} \cdot \text{int} \rightarrow \text{int} \]

This machine applies a continuation to an integer and returns the result.

**Compiled Machine:**

We consider the following compiled machine:

\[ \text{apply} \cdot \text{cost} \cdot \text{int} \cdot \text{int} \rightarrow \text{int} \]

This machine applies a continuation to an integer and returns the result.

**Compiled Machine:**

We consider the following compiled machine:

\[ \text{apply} \cdot \text{cost} \cdot \text{int} \cdot \text{int} \rightarrow \text{int} \]

This machine applies a continuation to an integer and returns the result.

**Compiled Machine:**

We consider the following compiled machine:

\[ \text{apply} \cdot \text{cost} \cdot \text{int} \cdot \text{int} \rightarrow \text{int} \]

This machine applies a continuation to an integer and returns the result.

**Compiled Machine:**

We consider the following compiled machine:

\[ \text{apply} \cdot \text{cost} \cdot \text{int} \cdot \text{int} \rightarrow \text{int} \]

This machine applies a continuation to an integer and returns the result.

**Compiled Machine:**

We consider the following compiled machine:

\[ \text{apply} \cdot \text{cost} \cdot \text{int} \cdot \text{int} \rightarrow \text{int} \]

This machine applies a continuation to an integer and returns the result.

**Compiled Machine:**

We consider the following compiled machine:

\[ \text{apply} \cdot \text{cost} \cdot \text{int} \cdot \text{int} \rightarrow \text{int} \]

This machine applies a continuation to an integer and returns the result.

**Compiled Machine:**

We consider the following compiled machine:

\[ \text{apply} \cdot \text{cost} \cdot \text{int} \cdot \text{int} \rightarrow \text{int} \]

This machine applies a continuation to an integer and returns the result.
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.

```plaintext
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.

```plaintext
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. 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.

```plaintext
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 successively 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 !. 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 for propositional Prolog.)

3.4 A specific interpreter computing the first solution

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

    structure Prolog_first = mProcessor (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 mProcessor with the corresponding notion of answers:

    structure Prolog_how_many = mProcessor (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 mProcessor (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, sc, fc', cc)
        | run_goal (a, p, fn fc' => run_goal (a', g, p, sc, fc', cc), cc, fc, cc)
        and run_atom (Syntax.IDE 1, p, sc, fc, cc) = (case lookup (1, p) of SOME -> fc ()
          | SOME (Syntax.GOAL g) -> run_goal (g, p, sc, fc, cc)
          | run_atom (Syntax.GOAL (g1, g), Syntax.GOAL g2), p, sc, fc, cc) = run_goal (g1, p, sc, fc, 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
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 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. 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 are 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 Bruijn and de Vink's continuation semantics
for Prolog with cut [11]:
- de Bruijn 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 continua-
tion and a cut continuation, whereas our success continuations expect only a
failure continuation. Analyzing the control flow of the corresponding in-
terpreter, 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 Bruijn and de Vink also present an operational semantics, and prove it
equivalent to their denotational semantics. In contrast, we defunctionalized
the interpreter corresponding to de Bruijn 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 pro-
grammers use logic programming languages as their meta-language. For example,
Kursawe showed how to "invert" 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 fac-
tor this interpreter into a compiler and a virtual machine, using, e.g., Wand's
combinator-based compiler derivation [23], Jorring and Scherlis's staging trans-
fomations [17], Hazan's pass-separation approach [14], or more generally the
binding-time separation techniques of partial evaluation [16, 19]. We are cur-
rently 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
Middgaard, and the anonymous referees for their comments. This work is sup-
ported 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 del-
imited continuations such as shift and reset [9, 10, 13].

Figure 4 presents such a direct-style interpreter for Propositional Prolog with-
out 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 a' = 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.IBE 1, p, m)
  = (case lookup (1, p) of
     SOME => shift (fn sc => m)
     | SOME (Syntax.GOAL g)
      => run_goal (g, p, m))

| run_atom (Syntax.GOAL g, Syntax.GOAL g1, Syntax.GOAL g2), p, m)
  = shift (fn sc => let val a' = 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
References


