Delimited Continuations for Prolog

TOM SCHRIJVERS
Ghent University, Belgium
(*e-mail:* `tom.schrijvers@ugent.be`)

BART DEMOEN
KU Leuven, Belgium
(*e-mail:* `bart.demoen@cs.kuleuven.be`)

BENOIT DESOUTER
Ghent University, Belgium
(*e-mail:* `benoit.desouter@ugent.be`)

JAN WIELEMAKER
University of Amsterdam, The Netherlands
(*e-mail:* `jan@swi-prolog.org`)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Delimited continuations are a famous control primitive that originates in the functional programming world. It allows the programmer to suspend and capture the remaining part of a computation in order to resume it later. We put a new Prolog-compatible face on this primitive and specify its semantics by means of a meta-interpreter. Moreover, we establish the power of delimited continuations in Prolog with several example definitions of high-level language features. Finally, we show how to easily and effectively add delimited continuations support to the WAM.

KEYWORDS: delimited continuations, Prolog

1 Introduction

As a programming language Prolog is very lean. Essentially it consists of Horn clauses extended with mostly simple built-in predicates. While this minimality has several advantages, the lack of infrastructure to capture and facilitate common programming patterns can be quite frustrating. Fortunately, programmers can mitigate the tedious drudgery of encoding frequent programming patterns by automating them by means of Prolog's rich meta-programming and program transformation facilities. Well-known examples of these are definite clause grammars (DCGs), extended DCGs (Roy 1989), Ciao Prolog's structured state threading (Ivanovic et al. 2009) and logical loops (Schimpf 2002).

However, non-local program transformations are not ideal for defining new language features for several reasons. Firstly, the effort of defining a transformation is proportional to the number of features in the language – the more features are added, the harder it becomes. Secondly, program transformations are fragile with respect to language evolution: they require amendments when other features are added to the language. Thirdly, when the new feature is introduced in existing code, the whole system may have to be transformed. For instance, consider introducing DCGs into an existing code base to pass information from a top-level predicate, through several layers of predicate calls, to a particular target predicate.

All of the above issues stifle the initial development and further adoption of new programming language features that are defined by means of non-local program transformations.

We remedy these problems by bringing for the first time a well-known control primitive to Prolog: *delimited continuations* (Felleisen 1988; Danvy and Filinski 1990). Delimited continuations enable the definition of new high-level language features at the program level (e.g., in libraries) rather than at the meta-level as program transformations. As a consequence, feature extensions based on delimited continuations are more light-weight, more robust with respect to changes and do not require pervasive changes to existing code bases.

Our specific contributions are:

1. We put a new Prolog-compatible face on the originally functional delimited continuations feature and specify its semantics by means of a meta-interpreter.
2. We establish the power of delimited continuations in Prolog by defining several high-level language features, including implicit state and DCGs, on top of them.
3. We show how to easily and effectively add delimited continuations support to the WAM.

2 Delimited Continuations by Example

2.1 Informal Definition

Our design of delimited continuations consists of two interacting predicates, *reset/3* and *shift/1*. The meta-predicate *reset(Goal,Cont,Term1)* executes *Goal*, and,

- if *Goal* fails, then *reset/3* also fails.
- if *Goal* succeeds, then *reset/3* also succeeds and binds *Cont* and *Term1* to *0*.
- if at some point *Goal* calls *shift(Term2)*, then its further execution is suspended and *reset/3* succeeds immediately, binding *Term1* to *Term2* and *Cont* to the remainder of *Goal*.

The pair *reset/3-shift/1* is similar to *catch/3-throw/1*, with the following differences: (1) *shift/1* does not copy its argument, (2) *shift/1* does not delete choice points, (3) apart from its argument, *shift/1* also communicates the delimited continuation to the enclosing *reset/3*.
The following applications provide more insight in the possibilities of delimited continuations.

2.2 Effect Handlers

Plotkin and Pretnar (2009) have recently formulated a particularly insightful class of applications: *effect handlers*. Effect handlers are an elegant way to add many kinds of side-effectful operations to a language and far less intrusive than monads (Moggi 1991).

Implicit State Passing Figure 1 (left) defines an effect handler for an implicit state passing feature. The feature provides two primitive operations: *get/1* for reading the implicit state, and *put/1* for writing it. For instance, the predicate *inc/0* uses these primitives to increment the state.

```
        inc :- get(S), S1 is S + 1, put(S1).
```

```
-- State
get(S) :- shift(get(S)).
put(S) :- shift(put(S)).

run_state(Goal,Sin,Sout) :-
  reset(Goal,Cont,Command),
  ( Cont == 0 ->
      Sout = Sin
  ; Command = get(S) ->
      S = Sin,
      run_state(Cont,Sin,Sout)
  ; Command = put(S) ->
      run_state(Cont,S,Sout)
  ).
```

```
-- DCG

c(E) :- shift(c(E)).

phrase(Goal,Lin,Lout) :-
  reset(Goal,Cont,Term),
  ( Cont == 0 ->
      Lin = Lout
  ; Term = c(E) ->
      Lin = [E|Lmid],
      phrase(Cont,Lmid,Lout)
  ).
```

Fig. 1. Effect handler expressing the State monad (left) and effect handler for DCGs (right).

The effect handler decouples the syntax of the new operations from their semantics. The *put/1* and *get/1* predicates are all syntax and no semantics; they simply *shift* their own term representation. The semantics is supplied by the handler predicate *run_state/3*. This handler predicate runs a goal and interprets the two primitive operations whenever they are shifted. For the interpretation, *run_state* recursively threads a state. Hence, a minimal example that uses implicit state is:

```
        ?- run_state(inc,0,S).
        S = 1.
```

Definite clause grammars Figure 1 (right) shows a light-weight effect handler for definite clause grammars (DCGs). DCGs are a well-known Prolog extension to sequentially access the elements of an implicit list. They are conventionally defined by program transformation, for which they require special syntax to mark DCG clauses H --> B and to mark non-DCG goals $\{G\}$. Our effect handler requires neither. It only introduces one new primitive operation *c(E)* to access the current element E in the implicit list. For instance, the following predicate implements the grammar $(ab)^n$ and returns n.

```
      ab(0).
      ab(N) :- c(a), c(b), ab(M), N is M + 1.

      ?- phrase(ab(N),[a,b,a,b],[]).
      N = 2.
```

Composing Effect Handlers Effect handlers can easily be made compositional. All it takes is for a handler to propagate unknown operations to the next one in line. For example we can mix the DCG and state features this way.

```
      phrase(Goal,Lin,Lout) :-
        reset(Goal,Cont,Term),
        ( Cont == 0   -> ...
        ; Term = c(E) -> ...
        ;
            shift(Term),
            phrase(Cont,Lin,Lout)
        ).
```

```
ab.
ab :- c(a), c(b), inc, ab.

?- run_state(phrase(ab,[a,b,a,b],[]),0,S).
S = 2.
```

Iterators

```
yield(Term) :-
  shift(yield(Term)).

with_write(Goal) :-
  reset(Goal,Cont,Term),
  ( Term = yield(X) ->
      write(X),
      with_write(Cont)
  ;
      true
  ).
```

```
from_list([]).
from_list([X|Xs]) :-
  yield(X),
  from_list(Xs).
```

```
enum_from_to(L,U) :-
  ( L < U ->
      yield(L),
       NL is L + 1,
      enum_from_to(NL,U)
  ;
      true
  ).
```

Iteratees

```
ask(X) :-
  shift(ask(X)).

with_read(Goal) :-
  reset(Goal,Cont,Term),
  ( Goal = ask(X) ->
     read(X),
     with_read(Cont)
  ;
     true
  ).
```

```
sum(Sum) :-
  sum(0,Sum).

sum(Sum0,Sum) :-
  ask(X),
  ( X == eof ->
     Sum = Sum0
  ;
     Sum1 is Sum0 + X,
     sum(Sum1,Sum)
  ).
```

```
play(C,P) :-
  reset(C,C1,Term1),
  ( C1 == 0 ->
     true
  ; Term1 = ask(X) ->
     reset(P,P1,Term2),
     ( Term2 == 0 ->
         X = eof,
         call(C1)
     ; Term2 = yield(X) ->
         play(C1,P1)
     )
  ).
```

Fig. 2. Iterators and iteratees.

2.3 Coroutines

Delimited continuations also lend themselves well to the implementation of various *coroutines*, i.e., subroutines that can be suspended and resumed at certain locations to communicate with another routine.

Coroutines that suspend to output data are called *iterators*. They are created by generators that use the *yield* keyword to suspend and return an intermediate value before continuing with the generation of more values. Figure 2 (top) shows two straightforward examples, which we will use later. In a sense *yield/1* generalizes Prolog's *write/1* built-in: the coroutine runs in a context that consumes its output in a *user-defined* way.

Iteratees are the opposite of iterators: they suspend to request external input, using *ask/1*, which generalizes Prolog's *read/1* built-in. See Figure 2 (bottom) for an example. We can now play iterator and iteratee coroutines against each other, using *play/2* (see Figure 2, bottom right):

```
?- play(sum(Sum),from_list([1,2,3])).            ?- play(sum(Sum),enum_from_to(7,10)).
Sum = 6.                                          Sum = 34.
```

Note how loosely coupled the communication partners are. This engenders a flexible and modular design that promotes reuse. There is growing consensus that coroutines are easier to understand than lazy evaluation, which has similar advantages (Kiselyov 2012; Kiselyov et al. 2012).

There are many more variations of coroutines: coroutines that mix *yield/1* and *ask/1* to communicate in two directions, coroutines captured in data structures that have the look and feel of Java iterators, transducers that transform iterators of one kind of elements into iterators of another kind, . . . We refer interested readers to our companion technical report (Demoen et al. 2013) for several such examples.

3 Meta-Interpreter Semantics

This section formalizes the delimited continuations feature by extending the vanilla meta-interpreter for Prolog with the *reset/3* and *shift/1* predicates. See Section 4.3 for the fine print.

```
eval(G) :-
  eval(G,Signal),
  ( Signal = shift(Term,Cont) ->
      format(’ERROR: Uncaught `shift(˜w)\’.\n’,[Term]),
      fail
  ;
      true
  ).

eval(shift(Term),Signal) :- !,
  Signal = shift(Term,true).
eval(reset(G,Cont,Term),Signal) :- !,
  eval(G,Signal1),
  ( Signal1 = ok ->
     Cont = 0,
     Term = 0
  ;
     Signal1 = shift(Term,Cont)
  ),
  Signal = ok.
eval((G1,G2),Signal) :- !,
  eval(G1,Signal1),
  ( Signal1 = ok ->
      eval(G2,Signal)
  ;
      Signal1 = shift(Term,Cont),
      Signal = shift(Term,(Cont,G2))
  ).
eval(Goal,Signal) :-
  built_in_predicate(Goal), !,
  call(Goal),
  Signal = ok.
eval(Goal,Signal) :-
  clause(Goal,Body),
  eval(Body,Signal).
```

The meta-interpreter extends every goal with an extra output parameter *Signal*. It is instantiated to *ok* when the goal succeeds normally. The base case for this behavior is the *eval/2* clause for built-in predicates.

When a goal's evaluation is abruptly terminated by a call to *shift(Term)* before its continuation *Cont* can be executed, *Signal* is instantiated to *shift(Term,Cont)*. The base case for this behavior is the *eval/2* clause for *shift(Term)*, where the empty continuation is represented by the goal *true*.

The clause for conjunction *(G1,G2)* evaluates the first goal. If it succeeds normally, the conjunction clause proceeds with *G2*. If *G1* is aborted by *shift/1*, then the whole conjunction case is aborted too and *G2* is added to the returned continuation.

The clause for *reset(G,Cont,Term)* evaluates *G* and binds *Cont* and *Term* to *0* when *G* terminates normally; otherwise, they are bound to the returned values.

4 Implementation

This section presents an implementation *reset/3* and *shift/1* in the WAM (Aït-Kaci 1991; Warren 1983), more specifically in the context of hProlog (Demoen and Nguyen 2000).

4.1 The hProlog Implementation

There are three main issues in the implementation: (1) the representation of a (delimited) continuation, (2) the change of control involved in *shift/1*, and (3) how to pass the continuation and the argument of *shift/1* to *reset/3*. They are described at the abstract machine level, using the hProlog WAM variant that originates in the XSB implementation (Swift and Warren 2012): the name of several abstract machine instructions reflects that. Still, the code below should be easily readable to anyone acquainted with the WAM. Note that hProlog uses a separate environment and choice point stack, that WAM argument registers are numbered starting at 1, and that a free variable (a self-reference) never occurs in an environment.

Reset The hProlog code of *reset/3* is shown below on the left; *sysh:asm/1* is a variant of the *C asm* command for generating inline WAM instructions. The corresponding WAM code is on the right: for each instruction, it shows the code address.

```
reset(Goal,Cont,Term) :-                        000 allocate 4
                                                016 getpvar Y2 A3
                                                032 getpvar Y3 A2
        call(Goal),                             048 call call/1  4
        reset_marker,                           080 builtin_reset_marker_0
        sysh:asm(getpval(Cont,1)),              088 getpval Y3 A1
        sysh:asm(getpval(Term,2)).              104 getpval Y2 A2
                                                120 dealloc_proceed
```

The *builtin_reset_marker_0* serves two roles:

- If no *shift* is executed inside a succeeding *Goal* execution returns to the *reset_marker*. It is then responsible for putting the default value 0 in the WAM argument registers 1 and 2. The *getpval* instructions subsequently unify these registers with the appropriate arguments of *reset*.
- If a *shift* is executed inside *Goal*, the code for *shift* puts the correct values of *Cont* and *Term* in argument registers 1 and 2. The *reset_marker* then acts as a marker in the stack for *shift*, which returns to the *getpval* right **behind** the *marker* so that the *reset_marker* itself is not executed.

Shift The implementation of *shift/1* (together with that of a helper predicate *get_chunks/3*) is listed below. It performs two tasks: 1) to capture the continuation up to the nearest enclosing *reset/3* in a heap term *Cont*, and 2) to unwind the local stack to pass control back to that *reset/3*.

```
shift(Term) :-                              get_chunks(E,P,L) :-
        % 1/ capture continuation                   ( points_to_reset_marker(P) ->
        nextEP(first,E,P),                              L = []
        get_chunks(E,P,L),                          ;
        Cont = call_continuation(L),                    get_chunk(E,P,TB),
        % 2/ pass control                               L = [TB|Rest],
        sysh:asm(putpval(Cont,1)),                      nextEP(E,NextE,NextP),
        sysh:asm(putpval(Term,2)),                      get_chunks(NextE,NextP,Rest)
        unwind_environments.                        ).
```

1. The first task is handled by the auxiliary predicate *get_chunks(E,P,L)*. It captures the (delimited) continuation as a list *L* of *continuation chunks* by traversing the local stack and constructing with *get_chunk/3* a continuation chunk for each environment on the way.
The traversal is made possible by the new *nextEP(E,NextE,NextP)* primitive that retrieves the next environment pointer *NextE* and next continuation pointer *NextP* stored in the given

environment *E*. The traversal starts at the current environment, aliased by the atom *first*, and ends at the environment of the *reset/3* call, which is reached when the continuation pointer points to the *reset_marker* (identified by the new primitive *points_to_reset_marker/1*).
Finally, *shift* wraps the resulting list in the functor of the *call_continuation/1* predicate (shown later) so that it can be directly meta-called.
2. For the second task, *shift* first sets up *Cont* and *Term* in the first and second WAM argument registers (with *putpval*) where the two *getpval* instructions at the end of *reset/3* can find them. Then it passes control to *reset/3* with the new primitive *unwind_environments/0*. This primitive unwinds the environment stack up to the environment of the first enclosing *reset/3* call. Then it sets the WAM E register to point to this environment, and the WAM P register to point to just **after** the *builtin_reset_marker_0* instruction so that it does not get executed. Note that *unwind_environments/0* is careful not to upset the WAM argument registers set up by *shift/1*. Also note that *unwind_environments/0* leaves the choice points unchanged, so that later backtracking could bring the execution back in the scope of the *reset/3* goal. This is compatible with the meta-interpreter semantics in Section 3.

Continuation Chunks The predicate *get_chunk(E,P,TB)* builds a continuation chunk in its TB argument. Such a chunk captures in a heap term all the necessary information to resume the unexecuted remainder (the tail) of a predicate body. This information consists of 1) the code to execute, and 2) the data to execute with.

The first part is easy: the code to execute starts at P. The second part is more involved: the code may refer to data in both argument registers and environment variables. Fortunately, it is a WAM invariant that no argument registers are live at a code point, like P, right after a call. Hence, we only need to capture the set of live environment variables (LEV) in the environment E. In hProlog, just as in YAP (Costa et al. 2012) and possibly other systems, the LEV set at a continuation point P is determined at compile-time, and can at runtime be retrieved from P. This basically follows the ideas of Branquart and Lewi (1970). Hence, in summary, the term built by *get_chunk/3* in argument TB is term *$cont$(P,LEV)*.

The dual of *get_chunk/3* is *call_chunk($cont$(P,LEV))*: it builds a new environment on the local stack from the continuation chunk. The size of the new environment can be found in the call-instruction right before *P*, and the live variables LEV can be filled in the appropriate slots of the environment by using the position information provided by *P*.

The predicate *call_continuation/1* extends *call_chunk/1* to a list of continuation chunks.

```
call_continuation([]).
call_continuation([TB|Rest]) :-
        call_chunk(TB),
        call_continuation(Rest).
```

4.1.1 Example

The example of Fig. 3 provides more insight in the representation of a continuation. The example shows Prolog code on the left, and the corresponding hProlog WAM instructions on the right. The continuation captured in the example consists of two chunks:

1. *$cont$(800,[bla(_165)])* mentions: 1) the address (800) of the first instruction following the *shift/1* goal, and 2) the one active Yvar (Y2) of *r/0* at that point. The existence of the latter is derived from the preceding *call 3* instruction (768): 3 is the length of the

```
p :-                                        176  allocate 4
                                            192  putpvar Y2 A2
                                            208  putpvar Y3 A3
                                            224  put_atom A1 q
        reset(q,Cont,Term),                 248  call reset/3  4
                                            280  putpval Y2 A1
        w(Cont),                            296  call w/1  4
                                            328  putpval Y3 A1
        w(Term),                            344  call w/1  4
                                            376  putpval Y2 A1
        call(Cont).                         392  deallex call/1

q :-                                        584  allocate 2
        r,                                  600  call r/0  2
                                            632  put_atom A1 endq
        w(endq).                            656  deallex w/1

r :-                                        680  allocate 3
                                            696  putpvar Y2 A1
        foo(Y),                             712  call foo/1  3
                                            744  put_atom A1 shiftterm
        shift(shiftterm),                   768  call shift/1  3
                                            800  putpval Y2 A1
        w(Y).                               816  deallex w/1

foo(bla(_)).


      ?- p.
      call_continuation([$cont$(800,[bla(_165)]),$cont$(632,[])])
      shiftterm
      bla(_165)
      endq
```

Fig. 3. Example Prolog and WAM code

environment at that point (E,CP and Y2). Hence, the (dereferenced) reference to Y2 is copied in the list; the term is not copied with *copy_term/1*. During *call_continuation/1*, this reference is put in the appropriate environment slot in a new environment.

2. $cont$(632,[]) points to the instruction 632 right after the call to *r/0* in the body of q/0. Since that clause has no permanent variables, the LEV is empty.

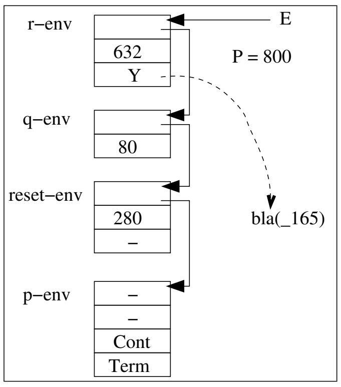

Fig. 4. The local stack and the E and P pointers at the moment *shift/1* is called

Figure 4 completes the example. It shows the environments of the activations of *p*, *reset*, *q* and *r*, at the moment that *shift/1* is constructing the continuation. The reader can check the values of all code pointers in the figure. Note that the term *bla(_165)* resides on the heap. Some environment entries are not shown as they are not relevant here.

4.2 Alternative Implementations

As a proof of concept, we have implemented delimited continuations also in SWI-Prolog (Wielemaker et al. 2012): it is based on the simpler ZIP (Bowen et al. 1983). Since an SWI-*frame* (similar to a WAM environment) records which predicate it belongs to, the reset_marker is not needed for finding the corresponding reset activation. SWI-Prolog uses code scanning (Wielemaker and Neumerkel 2008) to determine the live frame variables. Code scanning is performed once while constructing the delimited continuation, and avoided while reconstructing the frame with its correct slots. The SWI-Prolog analogue of the hProlog *$cont$/2* term is a *$cont$/3* term with as first argument a clause reference for keeping the clause corresponding to the chunk alive long enough. The second argument is a program counter similar to the first argument of the hProlog $cont$/2. The third argument is a list of *Offset-Term* pairs identifying the frame slot and the value of the live frame variables.

The interaction between the two implementations lead to the exploration of some alternatives.

Reinstalling the whole continuation in one go at call_continuation This can lead to repeatedly scanning (a copy of) the same continuation, and sometimes leads to changing the runtime complexity of the program from linear to quadratic.

Leaving continuations on the local stack instead of copying them to the heap. The obvious advantages are: a) creating the continuation in *shift/1* is cheap, and b) no data is created on the heap that must be garbage collected. After implementing this alternative, we made the following observations: (1) it can lead to the same complexity increase as installing the whole continuation in one go, unless one introduces an extra WAM-register (or global scoped variable) that remembers the environment with the CP that points to the marker: in this way, no scanning of the stack is needed to find the limits of the continuation. (2) The recursive reactivation of a delimited continuation is no longer possible. (3) To make the approach work, one needs to implement a local stack garbage collector. We have refrained from doing so. (4) Heap garbage collection needs some modification to scan the live continuations. (5) In case the delimited continuation is small (a few frames) the approach yields no performance advantage. (6) In SWI-Prolog, there are two extra advantages: a) the clause reference in the $cont/3$ term is no longer needed. b) It does no longer interfere with SWI-Prolog's meta-calling of control structures by means of a special temporary clause on the local stack.

In conclusion, keeping continuations on the stack is feasible, but whether it is desirable depends on the design of the virtual machine.

Non-selective Environment Saving We have also tried saving all environment slots in the *$cont$/2* structure, rather than selectively saving only the live variables. As a consequence, heap garbage collection becomes slightly more complicated but can still be accurate. Whether this approach saves time and space depends theoretically on the ratio between the sizes of the environment and the LEV, but in practice, it turns out to be only slightly more efficient, and we have abandoned it.

	Native		Transformed		Binarization
Depth	hProlog	SWI-Prolog	hProlog	SWI-Prolog	BinProlog
shift					
5,000	64	1,965	164	505	1,120
10,000	128	3,950	328	1,028	2,230
20,000	268	8,388	664	2,037	4,450
exec					
5,000	248 (398)	1,951 (1,137)	480 (398)	1,415 (1,137)	260 (270)
10,000	492 (796)	3,886 (2,283)	964 (796)	2,847 (2,283)	530 (550)
20,000	992 (1,586)	7,780 (4,390)	1,932 (1,586)	5,688 (4,390)	1,040 (1,100)

Table 1. *Benchmark results for shifting (top) and calling (bottom) continuations.*

4.3 Semantic Fine Print

Now that the general implementation approach is clear, we draw attention to a few semantic intricacies.

Cut and If-then-else WAM-based implementations usually store information (e.g., in a permanent variable in hProlog) for how far to cut in the environment. This distance may no longer be appropriate when the cut is captured in and executed as part of a delimited continuation. Special care must be taken to not inadvertently cut unrelated choice points. hProlog solves this problem by restricting the scope of a cut in a captured continuation to the most enclosing call to *call_continuation*.

Re-activation A continuation can be called more than once, so the question arises: what variables do those different activations share? In our approach, this depends on which optimizations are performed.

```
        a :- shift(x), nonvar(X), X = 1.
        a :- X = X, shift(x), nonvar(X), X = 1.
```

E.g. in the first clause the variable X is not live at the moment *shift/1* is called. Hence, the variable X is not shared between different invocations of the continuation. However, in the second clause sharing depends on whether *X=X* is optimized away or not. The meta-interpreter does not have this problem, and this is the only point where the low-level implementation differs from the meta-interpreter.

Shift-less resets and reset-less shifts In hProlog, we have chosen to unify the Cont and Term arguments of *reset/3* with zero in the absence of shift, and to have the toplevel catch shifts outside of a reset. Alternative semantics are easy to implement as a variation on the basic schema.

4.4 Evaluation

To assess the quality of our *native* implementation approach, we compare it to two other approaches for implementing delimited continuations:[1]

- The *transformation*-based approach can be thought of as partially evaluating the meta-interpreter of Section 3 for a given program. It adds a signal parameter to every predicate that is checked at every conjunction.
- The *binarization* approach uses the internal continuation-passing representation of Prolog clauses in BinProlog (Tarau 2012) and implements delimited continuations using the BinProlog built-ins.

The native and transformation approaches were implemented in hProlog and SWI-Prolog. It only makes sense to use the binarization approach in BinProlog.

We compare the three implementation approaches on two artificial benchmarks: (1) **shift** shifts a delimited continuation, and (2) **exec** calls a previously shifted continuation. We use three different sizes of continuations: 5,000, 10,000 and 20,000 chunks.

Table 1 shows the timing results (in milliseconds) obtained on an Intel Core2 Duo Processor T8100 2.10. Garbage collection times (only in SWI-Prolog) were not included, and the timings of *empty* loops were subtracted.

The shift benchmark in the upper half of the table shows that the native hProlog implementation is about 2.5 times faster than the transformed hProlog implementation. This shows that in hProlog the native implementation effort payed off. This is not the case in SWI-Prolog, partly because the native implementation is more involved in the ZIP and also because of other implementation choices made in SWI-Prolog. BinProlog's binarization does not exhibit an advantage compared to hProlog's transformation-based and native implementations. It is even outperformed by transformation in SWI-Prolog. Overall, we see that all implementations scale roughly linear with the size of the continuation, as expected.

The lower half of the table shows the exec benchmark which measures the time to execute the delimited continuation. This is contrasted (in brackets) with the time to meta-call a conjunction of equivalent goals. The same pattern shows here: SWI-Prolog performs better in transformed mode, while hProlog performs better in native mode. hProlog even executes its native continuations faster than meta-calling equivalent conjunctions, but note that in hProlog and SWI-Prolog, meta-call suffers a performance penalty because of the ISO Prolog semantics. Calling continuations in BinProlog is almost as fast as in hProlog and on par with BinProlog's meta-call.

In summary, a native implementation of delimited continuations in the WAM is worthwhile. This does not seem true in the ZIP, or at least not within the overall design of SWI-Prolog.

5 Related Work

Delimited continuations have their roots in functional programming, and their use in programs that explicitly pass continuations (CPS) is folklore. In the late 1980's, Felleisen (Felleisen 1988) and Danvy & Filinski (Danvy and Filinski 1990) independently proposed operators for delimited continuations in direct style programs. The latter is the *reset/shift* approach adopted in the article, which has a simple static interpretation in terms of continuations.

[1] See also our technical report (Demoen et al. 2013) for more details on these approaches.

Masuko and Asai (2009) give a good account of implementing delimited continuations in the context of the functional language MinCaml. Our implementation has - almost by necessity - similarities with theirs.

While we are not aware of any prior implementation of delimited continuations in Prolog, there are several noteworthy related works.

BinProlog Continuations The implementation of BinProlog (Tarau 2012) is based on explicit continuation passing: clauses are transformed to a binary form and carry the continuation as a first class citizen in an extra argument. While this continuation is normally invisible to the user, Tarau and Dahl (1994) describe how (still based on program transformation) the user can access and manipulate it. Based on this functionality it is possible to implement reset and shift (see (Demoen et al. 2013) for details).

BinProlog Logic Engines BinProlog (Tarau 2012; De Meuter and Roman 2011) also provides a coroutine-like feature: *logic engines*. A logic engine is essentially an independent Prolog environment that can be queried for successive answers to a goal.

In spirit, logic engines and coroutines are quite similar: to consider concurrency decoupled from multi-threading. However, our coroutines are more lightweight as they live in the same engine and, e.g., share the same heap and choice point stack. Moreover, in our approach the interfaces are more symmetric: coroutines receive data with *ask/1* that was sent by another coroutine with *yield/1* and vice versa. Logic engines receive data with *from_engine/1* that was sent by *to_engine/2* and return data with *return/1* that was requested by *get/2*.

Conventional Prolog Coroutines Various coroutine-like features have been proposed in the context of Prolog for implementing alternative execution mechanisms such as constraint logic programming: *freeze/2*, *block/1* declarations, . . . Nowadays most of these are based on a single primitive concept: attributed variables (Holzbaur 1992; Le Houitouze 1990; Neumerkel 1990; Demoen 2002). However, apart from the common name “coroutine” these attributed variable coroutines share very little with coroutines based on delimited continuations.

Environments on the Heap Demoen and Nguyen (2008) describe an implementation of coroutining in which environments of certain (declared) predicates are put on the heap instead of on the local stack. The programming interface proposed in that paper can be easily implemented with the constructs of the current paper. Without going in too much detail, it is fairly clear that our reset/shift are more general, and therefore not so efficient as their mechanism. However, the latter interferes more with other parts of the implementation (stack management, garbage collection ...) and is therefore perhaps not so attractive. Future work on the implementation might lead to a unified implementation which uses the best of both approaches.

6 Conclusion

This article has introduced a design of delimited continuations for Prolog that enables many useful applications. Alongside this design, it has described a complimentary implementation of the reset and shift operators in the WAM. The implementation is lightweight, because it is independent of most of the rest of the system, and its performance accommodates the applications.

Acknowledgments We thank Paul Tarau and Olivier Danvy for enlightening discussions.

References

AÏT-KACI, H. 1991. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press.

BOWEN, D., BYRD, L., AND CLOCKSIN, W. 1983. A portable Prolog compiler. In *Proceedings of the Logic Programming Workshop*. 74–83.

BRANQUART, P. AND LEWI, J. 1970. A Scheme of Storage Allocation and Garbage Collection for Algol 68. In *ALGOL 68 Implementation*. North-Holland, 199–238.

COSTA, V. S., ROCHA, R., AND DAMAS, L. 2012. The YAP Prolog system. *TPLP 12*, 5–34.

DANVY, O. AND FILINSKI, A. 1990. Abstracting control. LFP '90. 151–160.

DE MEUTER, W. AND ROMAN, G.-C., Eds. 2011. *Coordination Models and Languages*. LNCS, vol. 6721.

DEMOEN, B. 2002. Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Dept. of Comp. Sc., KU Leuven, Belgium.

DEMOEN, B. AND NGUYEN, P.-L. 2000. So many WAM Variations, so little Time. LNAI, vol. 1861. 1240–1254.

DEMOEN, B. AND NGUYEN, P.-L. 2008. Two WAM implementations of action rules. LNCS, vol. 5366. 621–635.

DEMOEN, B., SCHRIJVERS, T., AND DESOUTER, B. 2013. Delimited continuations in Prolog: semantics, use and implementation in the WAM. Report CW 631, Dept. of Computer Science, KU Leuven, Belgium.

FELLEISEN, M. 1988. The theory and practice of first-class prompts. POPL '88. 180–190.

HOLZBAUR, C. 1992. Meta-structures vs. Attributed Variables in the Context of Extensible Unification. LNCS, vol. 631. 260–268.

IVANOVIC, D., MORALES CABALLERO, J. F., CARRO, M., AND HERMENEGILDO, M. 2009. Towards structured state threading in Prolog. In *CICLOPS 2009*.

KISELYOV, O. 2012. Iteratees. LNCS, vol. 7294. 166–181.

KISELYOV, O., PEYTON-JONES, S., AND SABRY, A. 2012. Lazy vs. yield: Incremental, lazy pretty-printing. In *APLAS*.

LE HOUITOUZE, S. 1990. A New Data Structure for Implementing Extensions to Prolog. LNCS, vol. 456. 136–150.

MASUKO, M. AND ASAI, K. 2009. Direct implementation of shift and reset in the MinCaml compiler. ML'09. 49–60.

MOGGI, E. 1991. Notions of computation and monads. *Inf. Comput. 93,* 1.

NEUMERKEL, U. 1990. Extensible unification by metastructures. In *META'90*. 352–364.

PLOTKIN, G. AND PRETNAR, M. 2009. Handlers of algebraic effects. In *ESOP '09*.

ROY, P. V. 1989. A useful extension to prolog's definite clause grammar notation. *24,* 11, 132–134.

SCHIMPF, J. 2002. Logical loops. LNCS, vol. 2401. 224–238.

SWIFT, T. AND WARREN, D. S. 2012. XSB: Extending Prolog with Tabled Logic Programming. *TPLP 12,* 1-2, 157–187.

TARAU, P. 2012. The BinProlog experience: Architecture and implementation choices for continuation passing Prolog and first-class logic engines. *TPLP 12,* 1-2, 97–126.

TARAU, P. AND DAHL, V. 1994. Logic Programming and Logic Grammars with First-order Continuations. In *LOPSTR '94*. Vol. 883.

WARREN, D. H. D. 1983. An Abstract Prolog Instruction Set. Tech. Rep. 309, SRI.

WIELEMAKER, J. AND NEUMERKEL, U. 2008. Precise garbage collection in Prolog. In *CICLOPS '08*. 1–15.

WIELEMAKER, J., SCHRIJVERS, T., TRISKA, M., AND LAGER, T. 2012. SWI-Prolog. *TPLP 12,* 1-2, 67–96.