Delimited Continuations in Prolog: Semantics, Use, and Implementation in the WAM

Tom Schrijvers UGent
Bart Demoen KU Leuven
Benoit Desouter UGent

March 13, 2013

Abstract

An implementation of a *delimited continuations*, known in the functional programming world, is shown in the context of the WAM, and more particular in hProlog. Three new predicates become available to the user: reset/3 and shift/1 for delimiting and capturing the continuation, and call_continuation/1 for calling it. The underlying low-level built-ins and modifications to the system are described in detail. While these do not turn continuations into first-class Prolog citizens, their usefulness is shown in a series of examples. The idea behind this implementation can be adapted to other Prolog implementations. The constructs are compared with similar ones in BinProlog and Haskell. Their interaction with other parts of Prolog is discussed.

Contents

1 Introduction

As this is a technical report, we are not discussing in full a lot of related or seminal work. We limit ourselves to pointing at [15], in which the *prompt-facility* was introduced, which lead subsequently two related (but not identical) operators *shift* and *reset* in [9]. We focus here on those two operators in the context of Prolog and their implementation in the WAM. These operators lead to three new built-ins are named *reset/3, shift/1,* and *call_continuation/1.* We came to considering these operators because of our earlier work in [27], but we believe they are useful beyond that work, as was already argued by the authors of [31] in the context of BinProlog.

Overview Section 2 presents the three predicates in an informal way and through small examples. Section 3 shows the semantics based on two meta-interpreters, and Section 4 shows how a program transformation can implement the predicates. Section 5 shows interesting uses of these predicates. Section 6 dwells on the implementation details. Section 7 shows a comparison with other (Prolog) systems. Section 8 describes how our implementation deals with the cut, if-then-else, repeated activation of a continuation, backtracking and *catch&throw*. Section 9 makes a performance analysis of the new predicates. Section 10 concludes.

This work is a continuation and completion of [13]: as most technical details have been improved in the mean time, we do not refer to that document for further explanation.

We assume the reader knows the WAM [33, 3]

2 Informal Semantics

A manual could describe the three new predicates as follows:

- **reset(Goal,Cont,Term1)**: Goal must be callable; the most common case is that Goal is a simple goal (not a conjunction, disjunction or if-then-else)[1]. If the scope of the execution of the first argument calls shift(Term2), execution of the Goal ends
- **shift(Term2)**: when called in the scope of a *reset/3* goal, it unifies the delimited continuation up to the nearest call to *reset/3* goal with Cont, and Term1 with Term2; it returns control to just after this nearest call to that *reset/3* goal
- **call_continuation(Cont)**: for a continuation Cont constructed by shift/1, executes that continuation

reset/3 and shift/1 should be used together, and call_continuation/1 depends on a continuation constructed by shift/1. So we start with an example only using reset/3 and shift/1. In all following code, *w(X)* is an abbreviation for *writeln(X).*

[1]It would not be a bad idea to limit the use of reset/3 to a simple goal - see Section 8.3

```
    p :-                                    ?- p.
            reset(q,Cont,Term1),            a
            w(Term1),                       qterm
            w(Cont),                        [$cont$(785488,[])]
            w(endp).                        endp

    q :-
            w(a),
            shift(qterm),
            w(b).
```

This first example shows that shift/1 unifies the last two arguments of reset/3: both are printed out. A continuation is a particular Prolog term, in this case representing the *w(b)* goal in the context of the activation of the clause for *q/0*. Since this continuation is not activated, this goal has no effect. The example also shows that execution is continued after the *reset/3* goal.

One important thing this example shows is that it is up to the continuation of the *reset/3* to call the continuation is constructed by *shift/1*. A small adaptation of the above example shows the use of call_continuation/1:

```
    p :-                                    ?- p.
            reset(q,Cont,Term1),            a
            w(Term1),                       qterm
            call_continuation(Cont),        b
            w(endp).                        endp

    q :-
            w(a),
            shift(qterm),
            w(b).
```

The next example shows at the left a program using reset/shift and at the right a close equivalent that could be achieved by program transformation. Section 4 describes the transformation completely.

```
p0 :-
    w(before_reset),
    reset(q0,Cont,Term),
    w(after_reset),
    w(Term),
    w(Cont),
    call_continuation(Cont).

q0 :-
    w(start_q),
    r0,
    w(end_q).

r0 :-
    w(start_r),
    shift(rterm),
    w(end_r).


?- p0.
before_reset
start_q
start_r
after_reset
rterm
[$cont$(7624,[]),$cont$(7232,[])]
end_r
end_q
```

```
p1 :-
    w(before_reset),
    q1(Cont,true,Term),
    w(after_reset),
    w(Term),
    w(Cont),
    call(Cont).

q1(Cont,ContAccu,Term) :-
    w(start_q),
    r1(Cont,(w(end_q),ContAccu),Term).

r1(Cont,ContAccu,Term) :-
    w(start_r),
    Term = rterm,
    Cont = (w(end_r),ContAccu).


?- p1.
before_reset
start_q
start_r
after_reset
rterm
w(end_r) , w(end_q) , true
end_r
end_q
```

3 Meta-Interpreter Semantics

We now make the informal understanding of the semantics precise in a meta-interpreter.

3.1 Direct Style

This meta-interpreter extends the well-known vanilla meta-interpreter for Prolog with the three new operations `reset/3`, `shift/1` and `call_continuation/1`:

```
eval(G) :-
  eval(G,Signal),
  ( Signal = shift(Term,Cont) ->
      format('ERROR: Uncaught `reset(~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(call_continuation(Goal),Signal) :- !,
   eval(Goal,Signal).
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 case 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.

Finally, since the continuation is represented by a goal, `call_continuation/1` simply interprets it.

3.2 Continuation-Passing Style

The following continuation-passing style meta-interpreter is an alternative formalization of the delimited continuation semantics. It materializes the call stack as a stack of *continuation frames*. Every frame consists of a list (representing a conjunction) of goals. The evaluation of a conjunction `(G1,G2)` adds the second conjunct `G2` to the front of the list of the current frame at the top of the stack. The evaluation of `reset/3` pushes a new frame on top of the stack and `shift/1` pops the top frame from the stack.

```
eval(G) :-
  eval(G,top([])).

eval(reset(G,Cont,Term),Conts) :- !,
  eval(G,push([],Cont,Term,Conts)).
eval(shift(Term),Conts) :- !,
  eval_shift(Conts,Term).
eval(call_continuation(Cont0),Conts) :- !,
  add_conts(Cont0,Conts,NConts),
  eval_continue(NConts).
eval((G1,G2),Conts) :- !,
  add_cont(G2,Conts,NConts),
  eval(G1,NConts).
eval(true,Conts) :- !,
  eval_continue(Conts).
eval(Goal,Conts) :-
  clause(Goal,Body),
  eval(Body,Conts).

eval_continue(top([])).
eval_continue(top([G|Gs]) :-
  eval(G,top(Gs)).
eval_continue(push([],Cont,Term,Conts)) :-
  Cont = 0,
  Term = 0,
  eval_continue(Conts).
eval_continue(push([G|Gs],Cont,Term,Conts) :-
  eval(G,push(Gs,Cont,Term,Conts)).

eval_shift(top(Gs),Term) :-
  format('ERROR: Uncaught `reset(~w)\'.\n',[Term]),
  fail.
eval_shift(push(Cont,C,T,Conts),Term) :-
  C = Cont,
  T = Term,
  eval_continue(Conts).

add_cont(top(Gs),G,top([G|Gs])).
add_cont(push(Gs,Cont,Term,Conts),G,push([G|Gs],Cont,Term,Conts)).

add_conts([],Conts,Conts).
add_conts([G|Gs],Conts,NConts) :-
  add_cont(Conts,G,Conts1),
  add_conts(Gs,Conts1,NConts).
```

4 A program transformation based on the direct style interpreter

By means of the second Futamura projection we obtain a program transformation from the direct-style interpreter.

The general principle of the program transformation is that each predicate gets one extra argument (we picture it as the first argument for simplicity), which captures both the term of shift/1 and the continuation, just as the second argument *Signal* does in Section 3.1.

We need to consider the transformation of a fact, and of three types of clauses: below is at the left side the original form, and at the right the transformed form.

```
a.                               a(ok).


a :- b,                          a(X) :- b(Y),
     c.                                  (Y == ok -> c(X) ; addcont(Y,c(_),X)).


d :- reset(e,Cont,Term),         d(X) :- e(Y),
                                         (Y == ok -> Cont = 0, Term = 0
                                          ; s(Term,Cont) = Y),
     g.                                  g(X).


h :- shift(Term), k.             h(X) :- X = s(Term,k(_)).


                                 addcont(s(Term,Cont),G,s(Term,(Cont,G))).
```

In this context, it is also worth giving the implementation of call_continuation/1 and call/1. call_continuation/1 has as original definition:

```
call_continuation((G1,G2)) :- !,
           call_continuation(G1),
           call_continuation(G2).
call_continuation(G) :- call(G).
```

and the transformed version is obtained in the normal way.

For call/1, we write the transformed version *by hand*:

```
call(X,Goal) :-
           call(Goal),  % the normal call/1
           arg(1,Goal,X).
```

5 Useful Applications

There are many applications of delimited continuations, both in the literature and in the wild. While not all of these make sense in Prolog (e.g. for implementing non-determinism), there are still quite a few useful applications that are relevant.

5.1 Coroutines

Delimited continuations lend themselves well to the implementation of various forms of *coroutines*. By coroutines we do not mean the Prolog variety of coroutines, but their more general meaning: subroutines that can be suspended and resumed at certain locations. Most often a coroutine is suspended in order to communicate with another routine.

Various forms of coroutines are distinguished based on the direction of communication. Coroutines that suspend to output, or *yield*, data are called *iterators*, while those that suspend to receive data are called *iteratees*.

The main advantage of the coroutine approach is that a coroutine and its communication partner are usually loosely coupled. They are both implemented against a particular interface, which means that their communication partner can be easily replaced. This engenders a flexible and modular design that promotes reuse.

While lazy evaluation has similar advantages, there is a growing consensus [20, 25] in the lazy functional programming community that the coroutines are the better choice of the two because they are easier to understand. We believe that this argument carries over to Prolog: Prolog coroutines are to be preferred over lazy evaluation in Prolog in the style of Ciao [7].

5.1.1 Iterators

Coroutine-based iterators exist in many languages (e.g. Python). Iterators are created by generators that use the `yield` keyword to a suspend and return an intermediate value before continuing with the generation of more values. We support a similar `yield/1` operation in Prolog, which allows us to define various kinds of generators:

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

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

enumFrom(L) :-
  yield(L),
  NL is L + 1,
  enumFrom(NL).
```

Generators resemble lazy and potentially infinite streams. The `init_iterator/2` predicate packages a generator goal in an iterator structure that captures the last yielded element and the generator's continuation. The `next/3` predicate extracts this element and builds the new iterator from the continuation.

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

init_iterator(Goal,Iterator) :-
  reset(Goal,Cont,YE),
  ( YE = yield(Element) ->
      Iterator = next(Element,Cont)
  ;
      Iterator = done
  ).

next(next(Element,Cont),Element,Iterator) :-
  init_iterator(call_continuation(Cont),Iterator).
```

Consumers of iterators are independent of the particular generator:

```
sum(Iterator,Acc,Sum) :-
  ( next(Iterator,X,NIterator) ->
      NAcc is Acc + X,
      sum(NIterator,NAcc,Sum)
  ;
      Acc = Sum
  ).
```

and can be hooked up to many different ones:

```
?- init_iterator(fromList([7,2,3]),It), sum(It,0,Sum).
Sum = 12.

?- init_iterator(enumFromTo(1,5),It), sum(It,0,Sum).
Sum = 15.
```

Note that 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.

5.1.2 Iteratees

Iteratees are the opposite of iterators: they suspend to request external input. We provide the `ask/1` predicate for this purpose.

For instance, this predicate requests two numbers and adds them up:

```
sum(Sum) :-
  ask(X),
  ask(Y),
  Sum is X + Y.
```

The `ask/1` predicate generalizes Prolog's `read/1` built-in: the coroutine's context determines the source of the data.

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

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

with_list(L,Goal) :-
  reset(Goal,Cont,Term),
  ( Term = ask(X) ->
      L = [X|T],
      with_list(T,call_continuation(Cont))
  ;
      true
  ).
```

The data source can be modularly replaced:

```
?- with_list([1,2],sum(Sum)).
Sum = 3.

?- with_read(sum(Sum)).
|: 42.
|: 7.
Sum = 49.
```

5.1.3 General Coroutines

Iterator and iteratee coroutines can easily be played against each other:

```
?- play(sum(Sum),fromList([1,2])).
Sum = 3.

?- play(sum(Sum),enumFromTo(7,10)).
Sum = 15.
```

where `play/2` is defined as:

```
play(G1,G2) :-
  reset(G1,Cont1,Term1),
  ( Cont1 == 0 ->
     true
  ;
     reset(G2,Cont2,Term2),
     sync(Term1,Term2),
     play(call_continuation(Cont1),call_continuation(Cont2))
  ).

sync(ask(X),yield(X)).
sync(yield(X),ask(X)).
```

More generally, coroutines can mix `yield/1` and `ask/1` to communicate in two directions.

```
mapL([],[]).
mapL([X|Xs],[Y|Ys]) :-
  yield(X),
  ask(Y),
  mapL(Xs,Ys).

scanSum(Acc) :-
  ask(X),
  NAcc is Acc + X,
  yield(NAcc),
  scanSum(NAcc).
```

For instance:

```
?- play(mapL([1,2,3,4],L),scanSum(0)).
L = [1,3,6,10].
```

Compare this coroutine-based approach to Sterling and Kirschenbaum's approach of applying techniques to skeletons [28]. The former are much more lightweight and uniform. In contrast, the latter rely on program transformation or meta-interpretation and are more ad-hoc.

Transducers A transducer transforms an iterator of one kind into an iterator of another kind. A transducer communicates with two parties: it asks values from an underlying iterator and uses these to produce other values it yields to an iteratee.

The `transduce/2` predicate applies a transducer to an iterator.

```
transduce(IG,TG) :-
  reset(TG,ContT,TermT),
  transduce_(TermT,ContT,IG).

transduce_(0,_,_).
transduce_(yield(NValue),ContT,IG)) :-
  yield(NValue),
  transduce(IG,call_continuation(ContT)).
transduce_(ask(Value),ContT,IG) :-
  reset(IG,ContI,TermI),
  ( TermI == 0 ->
     true
  ;
     TermI = yield(Value),
     transduce(call_continuation(ContI),call_continuation(ContT))
  ).
```

The `doubler/2` predicate is an example of a transducer that doubles the values it receives.

```
doubler :-
  ask(Value),
  NValue is Value * 2,
  yield(NValue),
  doubler.
```

Here is an example:

```
?- play(sum(Sum),transduce(fromList([1,2]),doubler)).
Sum = 6.
```

5.2 Implementing `catch/3` and `throw/1`

The usual way to call catch/throw is in combination with each other, so that *catch(Goal,Ball1,Handler)* corresponds to a *throw(Ball2)*.

catch/throw has similarities with reset/shift, the important differences being:

1. throw/1 discards both the forward and backtracking continuation up to the matching call to catch/3; shift/1 only discards the forward continuation

2. throw/1 makes a copy of Ball2 (let's name it Ball2Copy) and then undoes the bindings up to the catch/3; shift/1 does not make a copy of its argument and does not undo bindings

3. if Ball1 and Ball2Copy do not unify, a matching call to catch/3 higher up in the execution is searched for; if Term1 does not unify with Term2, this results in failure of the continuation of the reset/3 goal, and backtracking occurs, potentially into Goal

4. as for catch/throw, it is as if the Handler is *true*

The following code shows how `catch/3` and `throw/1` can be implemented with reset/shift.

```
catch(Goal,_Catcher,_Handler) :-           catch1(Goal) :-
     nb_setval(thrown,nothrow),                 reset(Goal,Cont,Term),
     catch1(Goal).                              (Cont == 0 ->
catch(_Goal,Catcher,Handler) :-                     true  % no ball was thrown
     nb_getval(thrown,Term),                    ;
     Term = ball(Ball),                             !,
     nb_setval(thrown,nothrow),                     nb_setval(thrown,Term),
     (Ball = Catcher ->                             fail
         call(Handler)                          ).
     ;
         throw(Term)                       throw(Ball) :-
     ).                                         copy_term(Ball,BC),
                                                shift(ball(BC)).
```

Note that we use the fact that when there is no shift/1 inside the Goal of reset/3, Cont is unified with the integer 0.

One might wonder whether implementing reset/shift with catch/throw is possible. One showstopper is that throw/1 undoes all bindings up to the catch/3, while shift/1 does not. The other is that throw/1 copies (at least in its ISO compliant mode that SWI does not adhere to in this case) while shift/1 does not make a copy of its argument. Both are related of course.

6 Native Implementation

This section describes the nitty-gritty details of the hProlog-WAM implementation of reset/3 and shift/1. Section 8 is much easier to follow once the implementation is understood.

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. All will be described at the abstract machine level, using the hProlog WAM variant that originates in the XSB implementation [29]: the names of several abstract machine instructions reflects that. Still, the code below should be easily readable to all WAM-addicts. Note that hProlog uses a separate environment and choicepoint stack.

The predicate reset/3 as written in hProlog is below on the left side: hProlog supports a (still limited) form of the *C asm* command for generating inline WAM instructions. The generated code is at the right.

```
reset(Goal,Cont,Term) :-                  allocate 4
                                          getpvar Y2 A3
                                          getpvar Y2 A3
        call(Goal),                       call call/1  4
        after_callcc,                     builtin_after_callcc_0
        sysh:asm(getpval(Cont,1)),        getpval Y3 A1
        sysh:asm(getpval(Term,2)).        getpval Y2 A2
                                          dealloc_proceed
```

One idea is that the instruction builtin_after_callcc_0 acts like a marker in the code that can be found (from a continuation pointer) by shift/1. The other idea is that in normal circumstances (i.e. a shift/1 finds this occurrence of a reset/3) the shift/1 puts in argument registers A2 and A1 the values of Term and Cont, and that builtin_after_callcc_0 is **not** executed. This will be shown in the code of *shift/1*. builtin_after_callcc_0 is only executed if execution got there without a shift/1: how this is dealt with is shown later.

The implementation of shift/1 is given below: it relies on 5 new low-level built-ins.

- nextEP/3: given an environment pointer[2] in the first argument, it returns in argument 2 and 3 the E and CP pointers in that environment; if the first argument is the atom *first*, it returns the E and CP pointer from the current environment, as that is the start of the chain that (normally) leads up to a builtin_after_callcc_0 instruction
- get_all_cc/3 constructs in its first argument Cont, the (delimited) continuation starting from the given E,P combination, up to the first enclosing call to reset/3
- points_to_callcc/1 succeeds if and only if its argument points to the builtin_after_callcc_0 instruction
- unwind_stack_callcc/0 unwinds the environment stack in a similar way as get_all_cc traverses it: up to the first enclosing call to reset/3; unwinding means that at the end, the E,P combination becomes the WAM E and P values - with a small twist: P will point just after the builtin_after_callcc_0 instruction so that it does not get executed
- get_one_tailbody/3 constructs in argument 3 the *tail of the body* starting at P, and within the context of E; this is explained later; we name such a tail of a body a chunk of the continuation, and the whole continuation is a list of such chunks

```
shift(Term) :-
        nextEP(first,E,P),
        get_all_cc(Cont,E,P),
        sysh:asm(putpval(Cont,1)),
        sysh:asm(putpval(Term,2)),
        unwind_stack_callcc.
```

```
get_all_cc(L,E,P) :-
        (points_to_callcc(P) ->
            L = []
        ;
            L = [TB|RestCC],
            get_one_tailbody(E,P,TB),
            nextEP(E,NE,NP),
            get_all_cc(RestCC,NE,NP)
        ).
```

The *putpval* instructions in *shift/1* make sure that Cont and Term are in WAM argument registers A1 and A2, so that the two *getpval* instructions at the end of reset/3 unify them with the appropriate arguments of reset/3. In between these putpval and getpval instructions, the argument registers are not touched: only the unwind_stack_callcc is executed.

There remains to explain get_one_tailbody/3: the general idea is that the third argument TB is unified with the *clause tail starting at P in the context of E*, i.e. a chunk of the continuation.

P points into WAM-code, just after a call instruction (or one of its variants). At that code point, no argument registers are live (that is a WAM invariant), and the set of live environment variables *LEV* can be determined from the P pointer. The E pointer is

[2]It is actually a safe abstraction of an environment pointer.

relevant, because it allows to retrieve the values of the live environment variables from the correct environment.

In hProlog, just as in YAP [8] and possibly other systems, the set LEV at a continuation point is determined at compile-time, and it is one way or another linked to the continuation points. This basically follows the ideas in [6]. get_one_tailbody/3 constructs in its third argument TB a term *\$cont\$/2* whose arguments are:

- arg1: a code pointer **P**, pointing into the code for a clause, just behind a *call* instruction
- arg2: a list representing the values in **LEV**; this one can be composed because the LEV can be reconstructed from P

Dually to get_one_tailbody/3, call_tailbody/1 takes one chunk of continuation, and constructs an environment on the local stack: its size can be found in the call-instruction just before the code pointer in the first argument of *\$cont\$/2*. The appropriate variable slots can be filled in, by using the information provided by **P** (which points indirectly at the **LEV**) and the second argument of *\$cont\$/2*.

The native implementation of call_continuation/1 uses call_tailbody/1 as follows:

```
call_continuation([]).
call_continuation([TB|RestCC]) :-
        call_tailbody1(TB),
        call_continuation(RestCC).
```

6.1 An example with code

```
p :-                                      954176  allocate Y4
                                          954192  putpvar Y2 A2
                                          954208  putpvar Y3 A3
                                          954224  put_atom A1 q
        reset(q,Cont,Term),               954248  call reset/3  4
                                          954280  putpval Y2 A1
        w(Cont),                          954296  call w/1  4
                                          954328  putpval Y3 A1
        w(Term),                          954344  call w/1  4
                                          954376  putpval Y2 A1
        call_continuation(Cont).          954392  deallex call_continuation/1

q :-                                      954584  allocate Y2
        r,                                954600  call r/0  2
                                          954632  put_atom A1 endq
        w(endq).                          954656  deallex w/1

r :-                                      954680  allocate Y3
                                          954696  putpvar Y2 A1
        foo(Y),                           954712  call foo/1  3
                                          954744  put_atom A1 shiftterm
        shift(shiftterm),                 954768  call shift/1  3
                                          954800  putpval Y2 A1
        w(Y).                             954816  deallex w/1

foo(bla(_)).


              ?- p.
              [$cont$(954800,[bla(_165)]),$cont$(954632,[])]
              shiftterm
              bla(_165)
              endq
```

The flow of control should be clear to the reader by now. The above example is meant to explain in more detail the representation of a continuation. The example shows Prolog code at the left, and the WAM instructions as generated and loaded by hProlog. The address of each instruction is shown in the same abstraction as the code addresses used in the continuation.

- \$cont\$(954800,[bla(_165)]) is the first part of the continuation starting at the shift/1 call, up to the reset/3 goal; one can see that 954800 is the address of the first instruction following the shift/1 goal; moreover, the environment of r/0 has at that point one active Yvar (Y2); note that the *call* instruction just before has 3 as argument: that is the length of the environment at that point (E,CP and Y2); the value of Y2 is put in the list of variables needed by the continuation; during call_continuation, this value is put in the appropriate environment slot (the second slot) in a new environment; note that the value of Y2 is not copied with copy_term, but only the (dereferenced) reference to Y2 is copied in the list

- \$cont\$(954632,[]) is a continuation chunk whose code pointer 954632 points at the instruction after the call to r/0 in the body of q/0; since that clause has no permanent variables, the LEV is empty

Figure 1 completes the example: it shows the environments of the activations of p, q and r, at the moment that shift/1 is constructing the continuation. The reader can

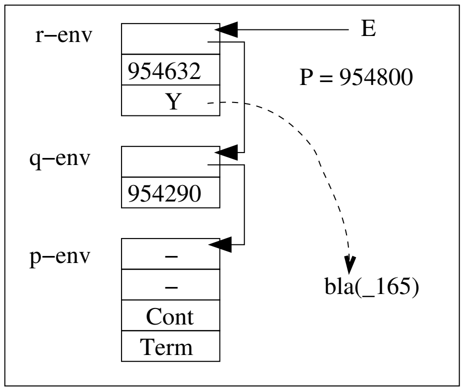

Figure 1: The local stack and the E and P pointers at the moment shift/1 is called

check the values of all code pointers in the figure. Note that the term $bla(\_165)$ resides on the heap. The E,CP values of the p-environment are not given, because they are not relevant.

6.2 The active Yvars

As mentioned earlier, hProlog basically follows the schema of [6]. There are many ways to implement this. In hProlog, a bitmap of fixed size describing which slots are live at that point in the execution of a clause, is an argument of each *call* instruction. If this fixed bitmap size is too small to represent the active Yvars, it points to a piece of memory after the code for the predicate (but still in the code zone), where there is enough space. The loader takes care of this. The bitmaps are not shown in the WAM code above.

7 Comparison

7.1 BinProlog and continuations

BinProlog [30] is based on explicit continuation passing: clauses are transformed to a binary form and carry the continuation as a first class citizen[3] in an extra argument. To be more explicit, binarization of the fact/clause/query at the left results internally in the constructs at the right:

```
a.                                a(Cont) :- call(Cont).

a :- b, c.                        a(Cont) :- b(c(Cont)).

?- q.                             ?- q(true).
```

[3]Unfortunately, being first class means it is an infinite term as soon as it is used explicitly.

While the continuation is normally invisible to the user, [31] describes how (still based on program transformation) the user can have access to the continuation, and then manipulate it. The special notation for that is by allowing multi-headed clauses of the form

```
        p(foo), bla :- body.
```

whose meaning is *if p/1 is called with first argument foo, and a continuation starting with bla, then execute body*. The above clause is binarized using the built-in BinProlog predicate strip_cont/3, which splits a continuation (a conjunction of goals, but in binarized nested form) into its first goal and the rest of the continuation. As [31] says: strip_cont/3 acts as if defined by

```
     strip_cont(f(X1,...,Xn,Cont), f(X1,...,Xn), Cont).
```

for every f/(n+1).

strip_cont/3 acts in a similar way to our get_one_tailbody/3 (see Section 6).

Based on strip_cont/3 and the implementation of catch and throw in BinProlog, we have build an implementation of reset/3 and shift/1 as follows:

```
       reset(Goal,Cont,Term) :-
               call(Goal),
               marker(Cont,Term).

       shift(Term) :-
               Marker = marker(Cs,Term),
               get_cont(Cont),
               consume_cont(Marker,(_,_,Cs),Cont,NewCont),
               call_cont(NewCont).

       consume_cont(Marker,Gs,Cont,LastCont):-
               strip_cont(Cont,Goal,NextCont),
               (Goal = Marker ->
                   LastCont = NextCont,
                   Gs = true
               ;
                   Gs=(Goal,OtherGs),
                   consume_cont(Marker,OtherGs,NextCont,LastCont)
               ).

       marker(0,0). % for catching the absence of a shift/1 goal inside Goal
```

The predicate consume_cont/4, is similar to our get_all_cc/3: it keeps peeling of the first goal of a continuation until the marker is found in the continuation. The above code works in BinProlog and we used it for the benchmarking.

7.2 BinProlog and Logic Engines

BinProlog [30, 22] 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, the logic engines approach and our 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 and return data with `return/1` that was requested by `get/2`.

7.3 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 [17, 21, 24, 10]. These attributed variables combine three useful aspects in one feature:

1. The ability to associate (updateable) data with a variable using `get_attr/3` and `put_attr/3`,
2. The ability to associate a goal, a call to the user-defined predicate `attr_unify_hook/2`, with the instantiation of the variable (the coroutine), and
3. The implicit and automatic invocation of the coroutine goal when the variable becomes instantiated or aliased to another attributed variable.

A significant difference with delimited continuations is that attributed variables allow only one way of transferring control: instantiation of a variable. In contrast, the predicates reset/shift offer explicit transfer of control.

Implementation wise and conceptually, the attributed variable coroutines are not based on continuations. Typically, either a routine is triggered only once, or the same (modified) goal is triggered over and over again. Any continuation-like behavior must be programmed explicitly.

7.4 Environments on the Heap

In [12] the authors describe an implementation of co-routining 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 the mechanism in [12]. However, the latter inferes 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.

7.5 Coroutines in Haskell

Our Prolog meta-interpreter implementation closely resembles James and Sabry's implementation in terms of the coroutine monad [18]. The Haskell code for this implementation is:[4]

```
-- The coroutine monad

data Yield t a = Return a | Yield a (Yield t a)

instance Monad (Yield t) where
  return x               = Return x
  (Return x)      >>= f  = f x
  (Yield t cont) >>= f   = Yield t (cont >>= f)

yield :: t -> Yield t ()
yield t = Yield t (return ())

-- The shift/reset implementation

shift :: t -> Yield t ()
shift t  = yield t

reset :: Yield t a -> Yield t (Either (t, Yield t a) a)
reset (Return x)      = return (Right x)
reset (Yield t cont)  = return (Left (t,cont))
```

Note that the Haskell constructors `Return` and `Yield` in this code correspond to the `Signal` parameter in our meta-interpreter. Additionally, the definition of the monadic bind `(>>=)` closely corresponds to the treatment of conjunction in the meta-interpreter.

James and Sabry are not the first to study the coroutine monad. Different variants of the coroutine monad (transformer) [5] have been studied under different names: resumption monad [26], free monad [4] and step monad [19].

7.6 The Origin of Delimited Continuations

Felleisen introduced reset and shift ("prompt applications") using the untyped lambda-calculus [15]. He defined the semantics via translation to a stack-machine, but did not provide an actual implementation. One of his examples was a yield-mechanism on a tree. Felleisen already pointed out the relation of continuations to stream-programming, although he did not distinguish yield as a separate operator. Duba *et al.* added first-class continuations to the statically typed ML language [14]. Flatt *et al.* implemented a production version in Scheme [16].

7.7 Coroutines in mainstream languages

Today, many mainstream languages like C♯, Ruby, JavaScript and Python, have some variant of a yield, although the operator is not widely described as a delimited continuation operator [2, 1, 23, 32]. Moreover expressivity greatly differs from language to language.

[4]We have omitted a few parts of James and Sabry's definition that may obfuscate the connection to our Prolog setting.

Statements vs. Expressions One distinguishing characteristic is whether the yield is an expression or a statement [18]. If yield is used as an expression, it means the iterator takes input from its calling context. This is possible in Ruby and Python 2.5. In earlier versions of Python, yield was a statement, as in Javascript 1.7 and C♯.

The continuations in our Prolog setting are essentially goals, which have a statement-like quality. However, as we have shown with iteratees, they are nevertheless able to take input from their context.

First-Class Iterators Another characteristic is whether iterators are first-class values. It is the case in *C*♯, although iterators are mostly used in combination with a foreach loop, as well as JavaScript and Python. A first-class iterator is more flexible and it is easier to work with more than one at the same time.

As we have shown, our iterators are first-class values in Prolog. They can be passed around freely.

8 Interaction with other parts of Prolog

8.1 Cut and If-then-else

One way or another, information on up to which choicepoint to cut needs to be stored in an environment. Two case are common: a cut not appearing as the first goal in a clause[5], and if-then-else with a non-simple test. Both are exemplified below:

```
                                     p :- savecp(B), p(B).
p :- a, !, body1.                    p(B) :- a, cutto(B), body1.
p :- body2.                          p(_) :- body2.

q :- b, (c -> d ; e), f.             q :- b, savecp(B), ( c, cutto(B) ; e), f.
```

At the left, the user-written code is shown, at the right the equivalent code using the (non-ISO) predicates *savecp/1* and *cutto/1*. The predicate *savecp/1* unifies in its argument the current choice point pointer B. Later *cutto/1* uses this pointer to cut up to the correct choice point.

In both cases, the variable B is a permanent variable that resides in the environment of that clause activation. The above is hProlog specific (and actually follows the XSB implementation), but a similar thing happens in many other Prolog implementations.

Since a continuation saves the active permanent variables, it is possible that the value of such a B is captured. The situation in which later the cutto(B) goal is executed in the delimited continuation must be treated carefully: the choice point B refers to might not exist any longer, and even if it does, it would be strange to cut all choicepoints upto B away, as there could be new choicepoints that are related to the continuation of the reset/3 goal.

A choice needs to be made, and we have decided that a cut in a captured continuation can only cut up to (but not included) the youngest choicepoint before calling *call_continuation*.

[5]If cut is the first goal in the clause, the relevant choicepoint can be detected in a different way.

The two examples below show this.

```
p0 :-                                     p1 :-
        reset(q0,Cont,Term),                      reset(q1,Cont,Term),
        w(Term),                                  w(Term),
        call_continuation(Cont).                  call_continuation(Cont).

q0 :- writeln(q_1), shift(fromq_1), !,    q1 :- (writeln(q_1), shift(fromq_1) ->
        writeln(endq_1).                                writeln(endq_1)
                                                ;
q0 :- writeln(q_1), shift(fromq_2),                     writeln(q_1), shift(fromq_2),
        writeln(endq_2).                                writeln(endq_2)
                                                ).
?- p0, fail.                              ?- p1, fail.
q_1                                       q_1
fromq_1                                   fromq_1
endq_1                                    endq_1
q_1                                       q_1
fromq_2                                   fromq_2
endq_2                                    endq_2
```

As for the implementation, one needs to take care that any active permanent variable whose value represents a choice point, is replaced by the appropriate choice point on executing a continuation containing that cut.

8.2 Re-activation

A continuation consists of chunks, each of which corresponds to the rest of a clause starting from a point in the clause. This point is represented by the a pointer into the WAM code of this clause, and the initial state of the execution is captured by the value of the permanent variables that have already been initialized by the WAM. The temporary variables (the argument registers) are irrelevant, as the code pointer always points just behind a *call* instruction, and argument registers do not survive a *call*. Below are two examples of reactivation of a continuation chunk.

```
calltwice1 :-                                calltwice2 :-
        reset(f1,Cont,_),                            reset(f2,Cont,_),
        writeln(Cont),                               writeln(Cont),
        call_continuation(Cont),                     call_continuation(Cont),
        call_continuation(Cont).                     call_continuation(Cont).

f1 :-                                        f2 :-
        foo(Y),
        shift(ignored),                              shift(ignored),
        writeln('Y' = Y),                            writeln('Y' = Y),
        Y = 1.                                       Y = 1.

foo(_).


?- calltwice1.                               ?- calltwice2.
[$cont$(787800,[_263])]                      [$cont$(788000,[])]
Y = _263                                     Y = _282
Y = 1                                        Y = _300
```

In both pieces of code, Y is a permanent variable. However, in f1/0, Y is initialized before the call to shift/1, so it appears in the continuation (one can see that the variable _263 occurs in the output twice), while in f2/0, Y is initialized after the call to shift/1:

so the initialization of Y in the second case happens in the continuation, every time the continuation is activated. This explains the output in both cases as well.

One could argue that the WAM optimization which initializes variables as late as possible is no good in this context, and it makes the results dependent on other optimizations (e.g. inline the call *foo(Y)* to *Y = _* and than remove that unification as it has no effect). So it seems difficult to rely on a particular sharing behavior of multiple invocations of the same continuation within the WAM. However, also in the case of BinProlog, the exact form of a continuation can depend on optimizations, or the particularities of the binarizing transformation.

8.3 The Goal in reset/3

Given the clauses for a/0 and b/0 below,

```
a :-                                                    ?- a.
     reset((b, w(inside_reset(Term))),Cont,Term),       after_reset
     w(after_reset),                                    after_shift
     call_continuation(Cont).                           inside_reset(shifted)

b :-
     shift(shifted),
     w(after_shift).
```

the result of the query *?- a.* might surprise. However, it is completely in line with the description that Cont captures the whole of the continuation after the call to shift/1 up to just *after* the reset/3 goal. Another way to see that this is the desired behavior is by considering that the clause for a/0 above is equivalent to

```
      a :-
              reset(newpred(Term),Cont,Term),
              w(after_reset),
              call_continuation(Cont).

      newpred(Term) :- b, w(inside_reset(Term)).
```

Even though the behavior of a compound goal in reset/3 is quite easy to understand, it might still be good to limit oneself to simple goals.

8.4 Reset/shift and backtracking

The following example shows that shift/1 does not cut away choicepoints:

```
c :-                                            ?- c.
        reset(d,Cont,Term),                     t(1)
        w(Term),                                aftershift(1)
        call_continuation(Cont).
                                                Yes ;
d :-                                            t(2)
        (X=1 ; X=2),                            aftershift(2)
        shift(t(X)),
        w(aftershift(X)).                       Yes
```

8.5 Possible Errors

- **there is no shift/1 inside the Goal of reset/3**: there are basically the following options
  - fail
  - throw an exception
  - unify Cont and Term with default values and proceed execution

  All three are easy to implement, but the third option seems the most useful one: we choose to unify Cont and Term with the integer 0. This is used in the code of catch/3 shown earlier, and many other places.

- **there is no reset/3 corresponding to a shift/1**: the options are
  - fail
  - throw an exception
  - let a default reset/3 goal at the toplevel handle it

  Again, all three are easy to implement. We choose the third option. It implies that in the code for get_all_cc/3 in Section 6 the test points_to_callcc(P) eventually succeeds, and ther is no risk to cross the boundaries of the environment stack.

8.6 Catch/throw and reset/shift

(ISO) Prolog has catch/throw as a scoped construct. Reset/shift is also scoped, so we must understand the interaction between both.

As long as the two constructs are properly nested, the resulting behavior is relatively easy to predict: the inner nested construct does not really interfere with the outer one.

When the two constructs are not properly nested, a little more thought is needed.

The first case is exemplified by the code below: it shows a continuation containing a throw/1. This continuation is picked up by reset/3 and is subsequently called. The call to throw/1 is no longer in the scope of the initially corresponding catch/3 goal, so the throw remains uncaught.

```
p :-
        reset(q,Cont,Term),
        writeln(Term),
        call_continuation(Cont).


q :- catch(r,Ball,writeln(Ball)).

r :- shift(rterm), throw(rball).

?- p.
rterm
Uncaught exception(rball)
```

From the language design point of view, there might be different options to explore. We are satisfied the current implementation behaves reasonably.

The other improper nesting is exemplified below:

```
             a :-
                     catch(b,Ball,writeln(Ball)).

             b :-
                     reset(c,Cont,Term),
                     writeln(Term),
                     call_continuation(Cont).

             c :-
                     throw(ballfromc),
                     shift(notseen).
             ?- a.
             ballfromc
```

Since throw/1 discards the forward continuation, it is clear that this situation does not pose any problems.

9 Performance

A comparison with functional programming implementations might be appropriate as well, but we limit ourselves here to a comparison with other Prolog systems. Not every Prolog system has delimited continuations. So we present just one small benchmark, consisting of 8 runs, grouped in two sets of 4. Below is the complete description of the first set:

```
             test(N) :-
                     reset(long0(N),Cont,_),
                     time(contlong,call_continuation(Cont)), nl,
                     fail.
             test(N) :-
                     reset(short0(N),Cont,_),
                     time(contshort,call_continuation(Cont)), nl,
                     fail.
             test(N) :-
                     time(direct,direct0(N)), nl,
                     fail.
             test(N) :-
                     G = (p,p,p,p,p,p,p),
                     time(meta,meta0(N,G)), nl,
                     fail.

long0(0) :- !,                                         direct0(0) :- !.
        time(longshift,shift(longshift)).              direct0(N) :-
long0(N) :-                                                    M is N - 1,
        M is N - 1,                                            direct0(M),
        long0(M),                                              p,p,p,p,p,p,p.
        p,p,p,p,p,p,p.
                                                       meta0(0,_) :- !.
                                                       meta0(N,G) :-
short0(0) :- !,                                                M is N - 1,
        time(shortshift,shift(shortshift)).                    meta0(M,G),
short0(N) :-                                                   call(G).
        M is N - 1,
        short0(M),                                     p07 :- p,p,p,p,p,p,p.
        p07.
```

The second set of 4 runs is obtained from the above by replacing every p/0 by *p(X,Y,Z)* and and adding the fact *p(_,_,_)*..

These runs compare the performance of constructing and calling a continuation with directly executing it, and with meta-calling it[6].

The above code can be run directly in hProlog: Table 1 refers to it as *hProlog native*.

The same code can be run in BinProlog when supplied with the definitions of reset/shift in Section 7.1: the results are in the column *BinProlog native=binary*.

The same code can be transformed using the transformation in Section 4: it is plain Prolog, so it can be executed in any system. We choose hProlog, SWI Prolog and YAP[7].

Finally, that same code can be binarized and executed with appropriate definitions of reset/shift (see the Appendix) in hProlog.[8] It is referred to as *hProlog binary* in the table.

Apart from hProlog 3.2.22-64 [11], we used YAP 6.2.2 [8], SWI-Prolog 6.2.1 [34] and BinProlog #12.00 [30].

The queries whose timings we report in Table 1 are equivalent to *?- test(2000000).* (for the code above). Garbage collection or stack expansions were excluded from the timings. Times are in msecs. The figures were obtained on an Intel Core2 Duo Processor T8100 2.10.

[6]Note however that in the binarized version later, every single goal is meta-called.

[7]It works also in BinProlog

[8]That code also works in SWI and YAP, but we did not try it yet at the time of writing - it is not clear we can learn something from it.

benchmark	hProlog native	hProlog transformed	hProlog binary	SWI transformed	YAP transformed	BinProlog native=binary
long cont 0	72+128=200	396+984=1380	1696+256=1952	10508+2523=13031	460+1188=1648	2440+770=3210
short cont 0	68+148=236	128+324=452	232+224=456	449+1331=1780	188+488=676	340+340=680
direct call 0	100	272	388	1167	380	280
meta-call 0	1124 (404)	1140 (464)	340	2461	(496)	(620)
long cont 3	80+196=274	532+980=1512	1780+392=2172	1673+2715=4388	716+1248=1964	2450+990=3440
short cont 3	68+208=276	128+388=516	232+296=528	443+1458=1901	184+792=976	350+590=940
direct call 3	172	384	616	1428	504	560
meta-call 3	1180 (444)	1248 (524)	360	3750	(516)	(850)

Table 1: Timings for 4 systems, 8 benchmarks, 4 variants

The numbers between brackets are obtained with a non-ISO conforming implementation of call/1: this is the default in YAP, and can be switched on/off in hProlog. This affects the benchmarks in which a conjunction of goals is metacalled. Also BinProlog's metacall is non-ISO conforming. Table 1 shows roughly that

- in hProlog, capturing and executing a continuation takes about twice as long as directly calling a clause, and half as long as meta-calling (in non-ISO mode) the body of a clause; the ISO meta-call takes considerably more time
- native hProlog is much faster than transformed hProlog: this shows that it is worth doing the native implementation
- transformed hProlog is a bit faster than transformed YAP and much faster than transformed SWI: this shows that YAP and SWI could also benefit a lot from a native implementation of delimited continuations
- in hProlog, both direct calling, and meta-calling is at least as fast as in YAP, and a lot faster than in SWI: this supports the conclusion that the native implementation in any system can come close to a transformed solution
- as for comparing with BinProlog: transformed hProlog is significantly better than BinProlog, and even binary hProlog is as far as the continuation implementation goes (the rows long/short cont); this seems to indicate that the binarizing approach by BinProlog does not necessarily give a performance advantage compared to another transformational approach; it is not clear whether pushing down the implementation of reset/shift deeper into the BinProlog bowels, can make it significantly faster, because in some sense, binarization itself sits in the way

We can conclude that the performance of the native implementation of reset/shift in hProlog is reasonable: one could not expect it to be as fast as direct calling, and we are satisfied it is faster than the very fast (non-ISO) hProlog meta-call.

The reason why the transformed figures are not good, is twofold: the program transformation of Section 4 introduces testing at every inference, and the size of the representation of a single continuation chunk is much bigger. In the native approach that size is linear in the number of live variables in it, while for the transformation approach, the size is linear in the number of goals of the chunk plus their arguments. So, in the case of the benchmark at hand, the difference is:

```
                  native size          transformation size
p,p,p,p,p,p,p      3 heap cells              32 heap cells

p(X,Y,Z)*7         8 heap cells              46 heap cells
```

Of course, raw performance is not the main issue of delimited continuations: their added value is in the extra programming power they allow.

10 Conclusion

Our main motivation in writing down this technical report was to show an implementation of delimited continuations (reset/shift) which is new because in the context of a non-continuation passing style Prolog implementation - and based on the WAM - and (once more) showing examples of the usefulness of these constructs. The implementation is quite independent of the rest of the system, and its interaction with other parts of the system were shown. The performance is good: even in the very fast hProlog system, performance seems not an impediment to using the extra functionality offered by delimited continuations.

Acknowledgments

We thank Henk Vandecasteele for his work on the ilProlog compiler used within hProlog, and Paul Tarau for enlightning discussions.

References

[1] *Python PEP 342 — Coroutines via Enhanced Generators*, 2005. `http://www.python.org/dev/peps/pep-0342/`.

[2] *C♯ Language Specification Version 4.0*, 2010. `http://msdn.microsoft.com/en-us/library/x53a06bb.aspx`.

[3] H. Ait-Kaci. *Warren's Abstract Machine*. The MIT Press, Cambridge, MA, 1991.

[4] S. Awodey. *Category Theory*, volume 49 of *Oxford Logic Guides*. Oxford University Press, Oxford, 2006.

[5] M. Blazevic. monad-coroutine: Coroutine monad transformer for suspending and resuming monadic computations, 2010. `http://hackage.haskell.org/package/monad-coroutine`.

[6] P. Branquart and J. Lewi. A Scheme of Storage Allocation and Garbage Collection for Algol 68. In J. E. L. Peck, editor, *ALGOL 68 Implementation*, pages 199–238. North-Holland, 1970.

[7] A. Casas, D. Cabeza, and M. V. Hermenegildo. A syntactic approach to combining functional notation, lazy evaluation, and higher-order in lp systems. In M. Hagiya and P. Wadler, editors, *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings*, volume 3945 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 2006.

[8] V. S. Costa, R. Rocha, and L. Damas. The YAP Prolog system. *TPLP*, 12(1-2):5–34, 2012.

[9] O. Danvy and A. Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 151–160, New York, NY, USA, 1990. ACM.

[10] B. Demoen. Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Dept. of Comp. Sc., KU Leuven, Belgium, Oct. 2002.

[11] B. Demoen and P.-L. Nguyen. So many WAM Variations, so little Time. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Computational Logic - CL2000, First International Conference, London, UK, July 2000, Proceedings*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1240–1254. ALP, Springer, 2000.

[12] B. Demoen and P.-L. Nguyen. Two WAM implementations of action rules. In M. Garcia de la Banda and E. Pontelli, editors, *Lecture Notes in Computer Science,*, pages 621–635. Springer, Dec. 2008.

[13] B. Demoen and T. Schrijvers. A 10' implementation of callcc in the WAM. Report CW 628, Dept. of Computer Science, KU Leuven, Belgium, Nov. 2012.

[14] B. Duba, R. Harper, and D. MacQueen. Typing first-class continuations in ML. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '91, pages 163–173, New York, NY, USA, 1991. ACM.

[15] M. Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 180–190, New York, NY, USA, 1988. ACM.

[16] M. Flatt, G. Yu, R. B. Findler, and M. Felleisen. Adding delimited and composable control to a production programming environment. *SIGPLAN Not.*, 42(9):165–176, Oct. 2007.

[17] C. Holzbaur. Meta-structures vs. Attributed Variables in the Context of Extensible Unification. In M. Bruynooghe and M. Wising, editors, *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming*, number 631, pages 260–268. Springer-Verlag, Aug. 1992.

[18] R. James and A. Sabry. Yield: Mainstream delimited continuations, 2011.

[19] M. Jaskelioff and E. Moggi. Monad transformers as monoid transformers. *Theor. Comput. Sci.*, 411(51-52):4441–4466, Dec. 2010.

[20] O. Kiselyov. Iteratees. In *FLOPS*, volume 7294 of *LNCS*, pages 166–181. Springer, 2012.

[21] S. Le Houitouze. A New Data Structure for Implementing Extensions to Prolog. In P. Deransart and J. Maluszynski, editors, *Proceedings of the Second International Symposium on Programming Language Implementation and Logic Programming*, number 456, pages 136–150. Springer-Verlag, Aug. 1990.

[22] W. D. Meuter and G.-C. Roman, editors. *Coordination Models and Languages - 13th International Conference, COORDINATION 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings*, volume 6721 of *Lecture Notes in Computer Science*, Berlin Heidelberg, 2011. Springer.

[23] Mozilla.org. *Javascript 1.7*, 2006. `https://developer.mozilla.org/en/New_in_JavaScript_1.7`.

[24] U. Neumerkel. Extensible unification by metastructures. In *Proceedings of the second workshop on Metaprogramming in Logic (META'90)*, pages 352–364, Apr. 1990.

[25] A. S. Oleg Kiselyov, Simon Peyton-Jones. Lazy vs. yield: Incremental, lazy pretty-printing. In *APLAS*, 2012.

[26] N. S. Papaspyrou. A Resumption Monad Transformer and its Applications in the Semantics of Concurrency. Technical Report CSD-SW-TR-2-01, National Technical University of Athens, 2001.

[27] T. Schrijvers, M. Triska, and B. Demoen. Tor: extensible search with hookable disjunction. In *Proceedings of the 14th symposium on Principles and Practice of Declarative Programming*, PPDP '12, pages 103–114, New York, NY, USA, 2012. ACM.

[28] L. S. Sterling and M. Kirschenbaum. Applying techniques to skeletons. In *Constructing Logic Programs*, pages 127–140. John Wiley, 1993.

[29] T. Swift and D. S. Warren. XSB: Extending Prolog with Tabled Logic Programming. *TPLP*, 12(1-2):157–187, 2012.

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

[31] P. Tarau and V. Dahl. Logic Programming and Logic Grammars with First-order Continuations. In *Proceedings of LOPSTR'94, LNCS, Springer*, Pisa, June 1994.

[32] D. Thomas and A. Hunt. *Programming Ruby: the Pragmatic Programmer's Guide*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[33] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI, 1983.

[34] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. 12(1-2):67–96, 2012.

Appendix: shift/2 and reset/4 for hProlog binary

In BinProlog parlance, the code below should be taken as if :- were actually ::-, i.e. the code is not subject to binarization.

```
reset(Goal,Cont,Term,D) :-
        add1arg(Goal,marker(Cont,Term,D),NewGoal),
        call(NewGoal).

shift(Term,Cont) :-
       Marker = marker(Before,Term,_),
       b_setval(cont,Cont),
       split_at_marker(Marker,Before,After),
       call(After).

split_at_marker(Marker,Before,After) :-
        b_getval(cont,CurrentCont),
        (
          CurrentCont = Marker ->
          Marker = marker(_,_,After),
          Before = true,
          b_setval(cont,[])
        ;
          functor(CurrentCont,_,N),
          arg(N,CurrentCont,NewCurrentCont),
          b_setval(cont,NewCurrentCont),
          CurrentCont = Before,
          setarg(N,CurrentCont,NewBefore),
          split_at_marker(Marker,NewBefore,After)
        ).


marker(0,0,Cont) :- call(Cont).

add1arg(Head,X,NewHead) :-
        Head =.. [Name|Args],
        append(Args,[X],NewArgs),
        NewHead =.. [Name|NewArgs].
```

Note how we avoid the continuation to become an infinite term by not letting it be a first class term.