Disjunctive Delimited Control

Alexander Vandenbroucke[1] and Tom Schrijvers[2(✉)]

[1] Standard Chartered, London, UK
alexander.vandenbroucke@sc.com
[2] KU Leuven, Leuven, Belgium
tom.schrijvers@kuleuven.be

Abstract. Delimited control is a powerful mechanism for programming language extension which has been recently proposed for Prolog (and implemented in SWI-Prolog). By manipulating the control flow of a program from inside the language, it enables the implementation of powerful features, such as tabling, without modifying the internals of the Prolog engine. However, its current formulation is inadequate: it does not capture Prolog's unique non-deterministic nature which allows multiple ways to satisfy a goal.

This paper fully embraces Prolog's non-determinism with a novel interface for *disjunctive* delimited control, which gives the programmer not only control over the sequential (conjunctive) control flow, but also over the non-deterministic control flow. We provide a meta-interpreter that conservatively extends Prolog with delimited control and show that it enables a range of typical Prolog features and extensions, now at the library level: findall, cut, branch-and-bound optimisation, probabilistic programming, ...

Keywords: Delimited control · Disjunctions · Prolog · Meta-interpreter · Branch-and-bound

1 Introduction

Delimited control is a powerful programming language mechanism for control flow manipulation that was developed in the late '80s in the context of functional programming [2,5]. Schrijvers et al. [12] have recently ported this mechanism to Prolog.

Compared to both low-level abstract machine extensions and high-level global program transformations, delimited control is much more light-weight and robust for implementing new control-flow and dataflow features. Indeed, the Prolog port has enabled powerful applications in Prolog, such as high-level implementations of both tabling [3] and algebraic effects & handlers [8]. Yet, at the same time, there is much untapped potential, as the port fails to recognise the unique nature of Prolog when compared to functional and imperative languages that have previously adopted delimited control.

E. De Angelis and W. Vanhoof (Eds.): LOPSTR 2021, LNCS 13290, pp. 75–91, 2022.
https://doi.org/10.1007/978-3-030-98869-2_5

Indeed, computations in other languages have only one *continuation*, i.e., one way to proceed from the current point to a result. In contrast, at any point in a Prolog continuation, there may be multiple ways to proceed and obtain a result. More specifically, we can distinguish 1) the success or *conjunctive* continuation which proceeds with the current state of the continuation; and 2) the failure or *disjunctive* continuation which bundles the alternative ways to proceed, e.g., if the conjunctive continuation fails.

The original delimited control only accounts for one continuation, which Schrijvers et al. have unified with Prolog's conjunctive continuation. More specifically, for a given subcomputation, they allow to wrest the current conjunctive continuation from its track, and to resume it at leisure, however many times as desired. Yet, this entirely ignores the disjunctive continuation, which remains as and where it is.

In this work, we adapt delimited control to embrace the whole of Prolog and capture both the conjunctive and the disjunctive continuations. This makes it possible to manipulate Prolog's built-in search for custom search strategies and enables clean implementations of, e.g., `findall/3` and branch-and-bound. This new version of delimited control has an executable specification in the form of a meta-interpreter (Sect. 3), that can run both the above examples, amongst others. Appendices to this paper are available in the extended version [18].

2 Overview and Motivation

2.1 Background: Conjunctive Delimited Control

In earlier work, Schrijvers et al. [12] have introduced a Prolog-compatible interface for delimited control that consists of two predicates: `reset/3` and `shift/1`.

Motivation. While library developers and advanced users typically do not build in new language features in Prolog, they have traditionally been able to add various language extensions by means of Prolog's rich meta-programming and program transformation facilities. Examples are definite clause grammars (DCGs), extended DCGs [17], Ciao Prolog's structured state threading [7] and logical loops [11]. However, there are several important disadvantages to non-local program transformations for defining new language features: A transformation that combines features can be quite complex and is fragile under language evolution. Moreover, existing code bases typically need pervasive changes to, e.g., include DCGs.

Delimited continuations enable new language features at the program level rather than as program transformations. This makes features based on delimited continuations more light-weight and more robust with respect to changes, and it does not require pervasive changes to existing code.

Behavior. The precicate `reset(Goal,ShiftTerm,Cont)` executes `Goal`, and, 1. if `Goal` fails, `reset/3` also fails; 2. if `Goal` succeeds, then `reset/3` also succeeds and unifies `Cont` and `ShiftTerm` with `0`; 3. if `Goal` calls `shift(Term)`, then

the execution of `Goal` is suspended and `reset/3` succeeds immediately, unifying `ShiftTerm` with `Term` and `Cont` with the remainder of `Goal`.

The `shift/reset` pair resembles the more familiar `catch/throw` predicates, with the following differences: `shift/1` does not copy its argument (i.e., it does not refresh the variables), it does not delete choice points, and also communicates the remainder of `Goal` to `reset/3`.

Example 1. Consider Definite Clause Grammars (DCGs), a language extension to sequentially access the elements of an implicit list. It is conventionally defined by a program transformation that requires special syntax to mark DCG clauses `H --> B` and to mark non-DCG goals {`G`}. The delimited control approach requires neither. It introduces two new predicates: `c(E)` consumes the next element `E` in the implicit list, and `phrase(G,Lin,Lout)` runs goal `G` with implicit list `Lin` and returns unconsumed remainder `Lout`. 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.
```

The two DCG primitives are implemented as follows in terms of `shift/1` and `reset/3`.

```
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)
  ).
```

In words, `phrase/3` executes the given goal within a `reset/3` and analyzes the possible outcomes. If `Cont == 0`, this means the goal succeeds without consuming any input. Then the remainder `Lout` is equal to the input list `Lin`. Alternatively, the execution of the goal has been suspended midway by the invocation of a `shift/1` because it wants to consume an element from the implicit list with `c/1`. In that case, `Term` has been instantiated with a request `c(E)` for an element `E`. This request is satisfied by instantiating `E` with the first element of `Lin`. Finally, the remainder of the suspended goal, `Cont` (the continuation), is resumed with the remainder of the list `Lmid`.

Other examples of language features implemented in terms of delimited control are co-routines, algebraic effects [8] and tabling [3].

Obliviousness to Disjunctions. This form of delimited control only captures the conjunctive continuation. For instance `reset((shift(a),G1),Term,Cont)` captures in `Cont` goal `G1` that appears in conjunction to `shift(a)`. In a low-level operational sense this corresponds to delimited control in other (imperative and functional) languages where the only possible continuation to capture is the computation that comes sequentially after the shift. Thus this approach is very useful for enabling conventional applications of delimited control in Prolog.

In functional and imperative languages delimited control can also be characterised at a more conceptual level as capturing the entire remainder of a computation. Indeed, in those languages the sequential continuation coincides with the entire remainder of a computation. Yet, the existing Prolog approach fails to capture the entire remainder of a goal, as it only captures the conjunctive continuation and ignores any disjunctions. This can be illustrated by the `reset((shift(a),G1;G2),Term,Cont)` which only captures the conjunctive continuation `G1` in `Cont` and not the disjunctive continuation `G2`. In other words, only the conjunctive part of the goal's remainder is captured.

This is a pity because disjunctions are a key feature of Prolog and many advanced manipulations of Prolog's control flow involve manipulating those disjunctions in one way or another.

2.2 Delimited Continuations with Disjunction

This paper presents an approach to delimited control for Prolog that is in line with the conceptual view that the whole remainder of a goal should be captured, including in particular the disjunctive continuation.

For this purpose we modify the `reset/3` interface, where depending on `Goal`, `reset(Pattern,Goal,Result)` has three possible outcomes:

1. If `Goal` fails, then the `reset` succeeds and unifies `Result` with `failure`. For instance,

```
?- reset(_,fail,Result).
Result = failure.
```

2. If `Goal` succeeds, then `Result` is unified with `success(PatternCopy, DisjCont)` and the `reset` succeeds. Here `DisjCont` is a goal that represents the disjunctive remainder of `Goal`. For instance,

```
?- reset(X,(X = a; X = b),Result).
X = a, Result = success(Y,Y = b).
```

   Observe that, similar to `findall/3`, the logical variables in `DisjCont` have been renamed apart to avoid interference between the branches of the computation. To be able to identify any variables of interest after renaming, we provide `PatternCopy` as a likewise renamed-apart copy of `Pattern`.

3. If `Goal` calls `shift(Term)`, then the `reset` succeeds and `Result` is unified with `shift(Term,ConjCont,PatternCopy,DisjCont)`. This contains in addition to the disjunctive continuation also the conjunctive continuation. The latter is not renamed apart and can share variables with `Pattern` and `Term`. For instance,

```
?- reset(X,(shift(t),X = a; X = b),Result).
Result = shift(t,X = a, Y, Y = b).
```

Note that `reset(P,G,R)` always succeeds if `R` is unbound and never leaves choicepoints.

Encoding. `findall/3` Sect. 4 presents a few larger applications, but our encoding of `findall/3` with disjunctive delimited control already gives some idea of the expressive power:

```
findall(Pattern,Goal,List) :-
  reset(Pattern,Goal,Result),
  findall_result(Result,Pattern,List).

findall_result(failure,_,[]).
findall_result(success(PatternCopy,DisjCont),Pattern,List) :-
  List = [Pattern|Tail],
  findall(PatternCopy,DisjCont,Tail).
```

This encoding is structured around a `reset/3` call of the given `Goal` followed by a case analysis of the result. Here we assume that `shift/1` is not called in `Goal`, which is a reasonable assumption for plain `findall/3`.

Encoding. `!/0` Our encoding of the `!/0` operator illustrates the use of `shift/1`:

```
cut :- shift(cut).

scope(Goal) :-
  copy_term(Goal,Copy),
  reset(Copy,Copy,Result),
  scope_result(Result,Goal,Copy).

scope_result(failure,_,_) :-
  fail.
scope_result(success(DisjCopy,DisjGoal),Goal,Copy) :-
  Goal = Copy.
scope_result(success(DisjCopy,DisjGoal),Goal,Copy) :-
  DisjCopy = Goal,
  scope(DisjGoal).
scope_result(shift(cut,ConjGoal,DisjCopy,DisjGoal),Goal,Copy) :-
  Copy = Goal,
  scope(ConjGoal).
```

The encoding provides `cut/0` as a substitute for `!/0`. Where the scope of regular cut is determined lexically, we use `scope/1` here to define it dynamically. For instance, we encode

```
p(X,Y) :- q(X), !, r(Y).
p(4,2).
```

as

```
p(X,Y) :- scope(p_aux(X,Y)).
p_aux(X,Y) :- q(X), cut, r(Y).
p_aux(4,2).
```

The logic of cut is captured in the definition of `scope/1`; all the `cut/0` predicate does is request the execution of a cut with `shift/1`.

In `scope/1`, the `Goal` is copied to avoid instantiation by any of the branches. The copied goal is executed inside a `reset/3` with the copied goal itself as the pattern. The `scope_result/3` predicate handles the result:

- `failure` propagates with `fail`;
- `success` creates a disjunction to either unify the initial goal with the now instantiated copy to propagate bindings, or to invoke the disjunctive continuation;
- `shift(cut)` discards the disjunctive continuation and proceeds with the conjunctive continuation only.

3 Meta-interpreter Semantics

We provide an accessible definition of disjunctive delimited control in the form of a meta-interpreter. Broadly speaking, it consists of two parts: the core interpreter, and a top level predicate to initialise the core and interpret the results.

3.1 Core Interpreter

Figure 1 defines the interpreter's core predicate, `eval(Conj, PatIn, Disj, PatOut, Result)`. It captures the behaviour of `reset(Pattern,Goal,Result)` where the goal is given in the form of a list of goals, `Conj`, together with the alternative branches, `Disj`. The latter is renamed apart from `Conj` to avoid conflicting instantiations.

The pattern that identifies the variables of interest (similar to `findall/3`) is present in three forms. Firstly, `PatIn` is an input argument that shares the variables of interest with `Conj` (but not with `Disj`). Secondly, `PatOut` outputs the instantiated pattern when the goal succeeds or suspends on a `shift/1`. Thirdly, the alternative branches `Disj` are of the form `alt(BranchPatIn,BranchGoal)` with their own copy of the pattern.

When the conjunction is empty (1–4), the output pattern is unified with the input pattern, and `success/2` is populated with the information from the alternative branches.

When the first conjunct is `true/0` (5–6), it is dropped and the meta-interpreter proceeds with the remainder of the conjunction. When it is a composite conjunction `(G1,G2)` (7–8), the individual components are added separately to the list of conjunctions.

When the first conjunct is `fail/0` (9–10), the meta-interpreter backtracks explicitly by means of auxiliary predicate `backtrack/3`.

```
backtrack(Disj,PatOut,Result) :-
    ( empty_alt(Disj) ->
       Result = failure
    ; Disj = alt(BranchPatIn,BranchGoal) ->
       empty_alt(EmptyDisj),
       eval([BranchGoal],BranchPatIn,EmptyDisj,PatOut,Result)
    ).

empty_alt(alt(_,fail)).
```

If there is no alternative branch, it sets the `Result` to `failure`. Otherwise, it resumes with the alternative branch. Note that by managing its own backtracking, `eval/5` is entirely deterministic with respect to the meta-level Prolog system.

```
1  eval([],PatIn,Disj,PatOut,Result) :- !,
2      PatOut = PatIn,
3      Disj   = alt(BranchPatIn,BranchGoal),
4      Result = success(BranchPatIn,BranchGoal).
5  eval([true|Conj],PatIn,Disj,PatOut,Result) :- !,
6      eval(Conj,PatIn,Disj,PatOut,Result).
7  eval([(G1,G2)|Conj],PatIn,Disj,PatOut,Result) :- !,
8      eval([G1,G2|Conj],PatIn,Disj,PatOut,Result).
9  eval([fail|_Conj],_,Disj,PatOut,Result) :- !,
10     backtrack(Disj,PatOut,Result).
11 eval([(G1;G2)|Conj],PatIn,Disj,PatOut,Result) :- !,
12     copy_term(alt(PatIn,conj([G2|Conj])),Branch),
13     disjoin(Branch,Disj,NewDisj),
14     eval([G1|Conj],PatIn,NewDisj,PatOut,Result).
15 eval([conj(Cs)|Conj],PatIn,Disj,PatOut,Result) :- !,
16     append(Cs,Conj,NewConj),
17     eval(NewConj,PatIn,Disj,PatOut,Result).
18 eval([shift(Term)|Conj],PatIn,Disj,PatOut,Result) :- !,
19     PatOut = PatIn,
20     Disj   = alt(BranchPatIn,Branch),
21     Result = shift(Term,conj(Conj),BranchPatIn,Branch).
22 eval([reset(RPattern,RGoal,RResult)|Conj],PatIn,Disj,PatOut,Result):- !,
23     copy_term(RPattern-RGoal,RPatIn-RGoalCopy),
24     empty_alt(RDisj),
25     eval([RGoalCopy],RPatIn,RDisj,RPatOut,RResultFresh),
26     eval([RPattern=RPatOut,RResult=RResultFresh|Conj]
27          ,PatIn,Disj,PatOut,Result).
28 eval([Call|Conj],PatIn,Disj,PatOut,Result) :- !,
29     findall(Call-Body,clause(Call,Body), Clauses),
30     ( Clauses = [] -> backtrack(Disj,PatOut,Result)
31     ; disjoin_clauses(Call,Clauses,ClausesDisj),
32       eval([ClausesDisj|Conj],PatIn,Disj,PatOut,Result)
33     ).
```

Fig. 1. Meta-interpreter core

When the first conjunct is a disjunction `(G1;G2)` (11–14), the meta-interpreter adds (a renamed apart copy of) `(G2,Conj)` to the alternative branches with `disjoin/3` and proceeds with `[G1|Conj]`.

```
disjoin(alt(_,fail),Disjunction,Disjunction) :- !.
disjoin(Disjunction,alt(_,fail),Disjunction) :- !.
disjoin(alt(P1,G1),alt(P2,G2),Disjunction) :-
    Disjunction = alt(P3, (P1 = P3, G1 ; P2 = P3, G2)).
```

Note that we have introduced a custom built-in `conj(Conj)` that turns a list of goals into an actual conjunction. It is handled (15–17) by prepending the goals to the current list of conjuncts, and never actually builds the explicit conjunction.

When the first goal is `shift(Term)` (18–21), this is handled similarly to an empty conjunction, except that the result is a `shift/4` term which contains `Term` and the remainder of the conjunction in addition the branch information.

When the first goal is a `reset(RPattern,RGoal,RResult)` (22–27), the meta-interpreter sets up an isolated call to `eval/5` for this goal. When the call returns, the meta-interpreter passes on the results and resumes the current conjunction `Conj`. Notice that we are careful that this does not result in meta-level failure by meta-interpreting the unification.

Finally, when the first goal is a call to a user-defined predicate (28–33), the meta-interpreter collects the bodies of the predicate's clauses whose head unifies with the call. If there are none, it backtracks explicitly. Otherwise, it builds an explicit disjunction with `disjoin_clauses`, which it pushes on the conjunction stack.

```
disjoin_clauses(_G,[],fail) :- !.
disjoin_clauses(G,[GC-Clause],(G=GC,Clause)) :- !.
disjoin_clauses(G,[GC-Clause|Clauses], ((G=GC,Clause) ; Disj)) :-
    disjoin_clauses(G,Clauses,Disj).
```

An example execution trace of the interpreter can be found in [18, Appendix C].

Toplevel. The `toplevel(Goal)`-predicate initialises the core interpreter with a conjunction containing only the given goal, the pattern and pattern copy set to (distinct) copies of the goal, and an empty disjunction. It interprets the result by non-deterministically producing all the answers to `Goal` and signalling an error for any unhandled `shift/1`.

```
toplevel(Goal) :-
    copy_term(Goal,GoalCopy),
    PatIn = GoalCopy,
    empty_alt(Disj),
    eval([GoalCopy],PatIn,Disj,PatOut,Result),
    ( Result = success(BranchPatIn,Branch) ->
        ( Goal = PatOut ; Goal = BranchPatIn, toplevel(Branch))
```

```
    ; Result = shift(_,_,_,_) ->
        write('toplevel: uncaught shift/1.\n'), fail
    ; Result = failure ->
        fail
    ).
```

4 Case Studies

To illustrate the usefulness and practicality of our approach, we present two case studies that use the new `reset/3` and `shift/1`.

4.1 Branch-and-Bound: Nearest Neighbour Search

Branch-and-bound is a well-known general optimisation strategy, where the solutions in certain areas or branches of the search space are known to be bounded. Such branches can be pruned, when their bound does not improve upon a previously found solution, eliminating large swaths of the search space in a single stroke.

We provide an implementation of branch-and-bound (see Fig. 2) that is generic, i.e., it is not specialised for any application. In particular it is not specific to nearest neighbour search, the problem on which we demonstrate the branch-and-bound approach here.

```
bound(V) :- shift(V).

bb(Value,Data,Goal,Min) :-
    reset(Data,Goal,Result),
    bb_result(Result,Value,Data,Min).

bb_result(success(BranchCopy,Branch),Value,Data,Min) :-
  ( Data @< Value -> bb(Data,BranchCopy,Branch,Min)
  ; bb(Value,BranchCopy,Branch,Min)
  ).
bb_result(shift(ShiftTerm,Cont,BranchCopy,Branch),Value,Data,Min) :-
  (  ShiftTerm @< Value ->
     bb(Value,Data,(Cont ; (BranchCopy = Data,Branch)),Min)
  ;  bb(Value,BranchCopy,Branch,Min)
  ).
bb_result(failure,Value,_Data,Min) :- Value = Min.
```

Fig. 2. Branch-and-Bound effect handler.

```
nn((X,Y),BSP,D-(NX,NY)) :-
    ( BSP = xsplit((SX,SY),Left,Right) ->
        DX is X - SX,
        branch((X,Y), (SX,SY), Left, Right, DX, D-(NX,NY))
    ; BSP = ysplit((SX,SY),Up,Down) ->
        DY is Y - SY,
        branch((X,Y), (SX,SY), Up, Down, DY, D-(NX,NY))
    ).
branch((X,Y), (SX,SY), BSP1, BSP2, D, Dist-(NX,NY)) :-
    ( D < 0 -> % Find out which partition contains (X,Y).
        TargetPart = BSP1, OtherPart = BSP2, BoundaryDistance is -D
    ;
        TargetPart = BSP2, OtherPart = BSP1, BoundaryDistance is D
    ),
    ( nn((X,Y), TargetPart, Dist-(NX,NY))
    ; Dist is (X - SX) * (X - SX) + (Y - SY) * (Y - SY),
      (NX,NY) = (SX,SY)
    ; bound(BoundaryDistance-nil),
      nn((X,Y), OtherPart,Dist-(NX,NY))
    ).
run_nn((X0,Y0),BSP,(NX,NY)) :-
    toplevel(bb(10-nil,D-(X,Y),nn((X0,Y0),BSP,D-(X,Y)),_-(NX,NY))).
```

Fig. 3. 2D nearest neighbour search with branch-and-bound.

The framework requires minimal instrumentation: it suffices to begin every prunable branch with `bound(V)`, where `V` is a lower bound on the values in the branch.[1]

1. If the `Goal` succeeds normally (i.e., `Result` is `success`), then `Data` contains a new solution, which is only accepted if it is an improvement over the existing `Value`. The handler then tries the next `Branch`.
2. If the `Goal` calls `bound(V)`, `V` is compared to the current best `Value`:
   - if it is less than the current value, then `Cont` could produce a solution that improves upon the current value, and thus must be explored. The alternative `Branch` is disjoined to `Cont`, and `DataCopy` is restored to `Data` (ensuring that a future `reset/3` copies the right variables);
   - if it is larger than or equal to the current value, then `Cont` can be safely discarded.
3. Finally, if the goal fails entirely, `Min` is the current minimum `Value`.

Nearest Neighbour Search. The code in Fig. 3 shows how the branch and bound framework efficiently solves the problem of finding the point (in a given set) that is nearest to a given target point on the Euclidean plane.

[1] The framework searches for a minimal solution.

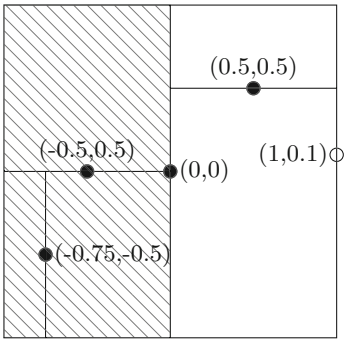

Fig. 4. Nearest-neighbour search using a BSP-tree

The `run_nn/3` predicate takes a point `(X,Y)`, a Binary Space Partitioning (BSP)-tree[2] that represents the set of points, and returns the point, nearest to `(X,Y)`. The algorithm implemented by `nn/3` recursively descends the BSP-tree. At each node it first tries the partition to which the target point belongs, then the point in the node, and finally the other partition. For this final step we can give an easy lower bound: any point in the other partition must be at least as far away as the (perpendicular) distance from the given point to the partition boundary.

As an example, we search for the point nearest to $(1, 0.1)$ in the set $\{(0.5, 0.5), (0, 0), (-0.5, 0), (-0.75, -0.5)\}$. Figure 4 shows a BSP-tree containing these points, the solid lines demarcate the partitions. The algorithm visits the points $(0.5, 0.5)$ and $(0, 0)$, in that order. The shaded area is never visited, since the distance from (1,0.1) to the vertical boundary through $(0, 0)$ is greater than the distance to $(0.5, 0.5)$ (1 and about 0.64). The corresponding call to `run_nn/3` is:

```
?- BSP = xsplit((0,0),
          ysplit((-0.5,0),leaf,xsplit((-0.75,-0.5),leaf,leaf)),
          ysplit((0.5,0.5),leaf,leaf)),
   run_nn((1,0.1),BSP,(NX,NY)).
NX = NY, NY = 0.5.
```

4.2 Probabilistic Programming

Probabilistic programming languages (PPLs) are programming languages designed for probabilistic modelling. In a probabilistic model, components behave in a variety of ways—just like in a non-deterministic model—but do so with a certain probability.

[2] A BSP-tree is a tree that recursively partitions a set of points on the Euclidean plane, by picking points and alternately splitting the plane along the x- or y-coordinate of those point. Splitting along the x-coordinate produces an `xsplit/3` node, along the y-coordinate produces a `ysplit/3` node.

Instead of a single deterministic value, the execution of a probabilistic program results in a probability distribution of a set of values. This result is produced by probabilistic *inference* [6,19], for which there are many strategies and algorithms, the discussion of which is out of scope here. Here, we focus on one concrete probabilistic *logic* programming languages: PRISM [10].

A PRISM program consists of Horn clauses, and in fact, looks just like a regular Prolog program. However, we distinguish two special predicates:

- `values_x(Switch,Values,Probabilities)` This predicate defines a probabilistic switch `Switch`, that can assume a value from `Values` with the probability that is given at the corresponding position in `Probabilities` (the contents of `Probabilities` should sum to one).
- `msw(Switch,Value)` This predicate samples a value `Value` from a switch `Switch`. For instance, if the program contains a switch declared as `values_x(coin, [h,t], [0.4,0.6])`, then `msw(coin,V)` assigns `h` (for heads) to `V` with probability 0.4, and `t` (for tails) with probability 0.6. Remark that each distinct call to `msw` leads to a different sample from that switch. For instance, in the query `msw(coin,X),msw(coin,Y)`, the outcome could be either `(h,h)`,`(t,t)`, `(h,t)` or `(t,h)`.

Consider the following PRISM program, the running example for this section:

```
values_x(coin1,[h,t],[0.5,0.5]).
values_x(coin2,[h,t],[0.4,0.6]).
twoheads :- msw(coin1,h),msw(coin2,h).
onehead :- msw(coin1,V), (V = t, msw(coin2,h) ; V = h).
```

This example defines two predicates: `twoheads` which is true if both coins are heads, and `onehead` which is true if either coin is heads. However, note the special structure of `onehead`: PRISM requires the *exclusiveness condition*, that is, branches of a disjunction cannot be both satisfied at the same time. The simpler goal `msw(coin1,heads) ; msw(coin2, heads)` violates this assumption.

The code in Fig. 5 interprets this program. Line 1 defines `msw/2` as a simple shift. Lines 6–9 install a `reset/3` call over the goal, and analyse the result. The result is analysed in the remaining lines: A *failure* never succeeds, and thus has success probability 0.0 (line 9). Conversely, a successful computation has a success probability of 1.0 (line 10). Finally, the probability of a switch (lines 11–15) is the sum of the probability of the remainder of the program given each possible value of the switch multiplied with the probability of that value, and summed with the probability of the alternative branch.

The predicate `msw_prob` finds the joint probability of all choices. It iterates over the list of values, and sums the probability of their continuations.

```
msw_prob(_,_,[],[],Acc,Acc).
msw_prob(V,C,[Value|Values],[Prob|Probs],Acc,ProbOfMsw) :-
  prob((V = Value,C),ProbOut),
  msw_prob(V,C,Values,Probs,Prob*ProbOut + Acc,ProbOfMsw).
```

```
msw(Key,Value) :- shift(msw(Key,Value)).
prob(Goal) :-
    prob(Goal,ProbOut),
    write(Goal), write(': '), write(ProbOut), write('\n').
prob(Goal,ProbOut) :-
    copy_term(Goal,GoalCopy),
    reset(GoalCopy,GoalCopy,Result),
    analyze_prob(GoalCopy,Result,ProbOut).
analyze_prob(_,failure,0.0).
analyze_prob(_,success(_,_),1.0).
analyze_prob(_,shift(msw(K,V),C,_,Branch),ProbOut) :-
    values_x(K,Values,Probabilities),
    msw_prob(V,C,Values,Probabilities,0.0,ProbOfMsw),
    prob(Branch,BranchProb),
    ProbOut is ProbOfMsw + BranchProb.
```

Fig. 5. An implementation of probabilistic programming with delimited control.

Now, we can compute the probabilities of the two predicates above:

```
?- toplevel(prob(twoheads)).
twoheads: 0.25
?- toplevel(prob(onehead)).
onehead: 0.75
```

In [18, Appendix B.3] we implement the semantics of a definite, non-looping fragment of ProbLog [6], another logic PPL, on top of the code in this section.

5 Properties of the Meta-interpreter

In this section we establish two important correctness properties of our meta-interpreter with respect to standard SLD resolution. Together these establish that disjunctive delimited control is a conservative extension. This means that programs that do not use the feature behave the same as before.

The proofs of these properties are in [18, Appendix A]. The first theorem establishes the soundness of the meta-interpreter, i.e., if a program (not containing `shift/1` or `reset/3`) evaluates to success, then an SLD-derivation of the same answer must exist.

Theorem 1 (Soundness). *For all lists of goals* $[A_1, \ldots, A_n]$*, terms* $\alpha, \beta, \gamma, \nu$*, variables* P, R *conjunctions* $B_1, \ldots, B_m$*;* $C_1, \ldots, C_k$ *and substitutions* θ*, if*

$$?- \mathit{eval}([A_1, \ldots, A_n], \alpha, \mathit{alt}(\beta, (B_1, \ldots, B_m)), P, R).$$
$$P = \nu, R = \mathit{success}(\gamma, C_1, \ldots, C_k).$$

and the program contains neither `shift/1` *nor* `reset/3`*, then SLD-resolution*[3] *finds the following derivation:*

$$\leftarrow (A_1, \ldots, A_n, \mathit{true}); (\alpha = \beta, B_1, \ldots, B_m)$$
$$\vdots$$
$$\square$$
$$\textit{(with solution } \theta \textit{ s.t. } \alpha\theta = \nu\textit{)}$$

Conversely, we want to argue that the meta-interpreter is complete, i.e., if SLD-derivation finds a refutation, then meta-interpretation—provided that it terminates—must find the same answer eventually. The theorem is complicated somewhat by the fact that the first answer that the meta-interpreter arrives at might not be the desired one due to the order of the clauses in the program. To deal with this problem, we use the operator ?-$_{\mathrm{p}}$, which is like ?-, but allows a different permutation of the program in every step.

Theorem 2 (Completeness). *For any goal* $\leftarrow A_1, \ldots, A_n$*, if it has solution* θ*, then*

$$?\text{-}_{\mathrm{p}}\ \mathit{eval}([A_1, \ldots, A_n], \alpha, \mathit{alt}(\beta, (B_1, \ldots, B_m)), P, R).$$
$$P = \mathit{success}(\gamma, (C_1, \ldots, C_k)), R = \alpha\theta.$$

Together, these two theorems show that our meta-interpreter is a conservative extension of the conventional Prolog semantics.

6 Related Work

Conjunctive Delimited Control. Disjunctive delimited control is the culmination of a line of research on mechanisms to modify Prolog's control flow and search, which started with the hook-based approach of Tor [13] and was followed by the development of conjunctive delimited control for Prolog [12,14].

The listing below shows that disjunctive delimited control entirely subsumes conjunctive delimited control. The latter behaviour is recovered by disjoining the captured disjunctive branch. We believe that Tor is similarly superseded.

```
nd_reset(Goal,Ball,Cont) :-
  copy_term(Goal,GoalCopy),
  reset(GoalCopy,GoalCopy,R),
  ( R = failure -> fail
  ; R = success(BranchPattern,Branch) ->
    ( Goal = GoalCopy, Cont = 0
    ; Goal = BranchPattern, nd_reset(Branch,Ball,Cont))
  ; R = shift(X,C,BranchPattern,Branch) ->
    ( Goal = GoalCopy, Ball = X, Cont = C
    ; Goal = BranchPattern, nd_reset(Branch,Ball,Cont))
  ).
```

[3] Standard SLD-resolution, augmented with disjunctions and `conj/1` goals.

```
get(Interactor,Answer) :-
  get_engine(Interactor,Engine),        % get engine state
  run_engine(Engine,NewEngine,Answer),  % run up to the next answer
  update_engine(Interactor,NewEngine).  % store the new engine state
return(X) :- shift(return(X)).
run_engine(engine(Pattern,Goal),NewEngine,Answer) :-
  reset(Pattern,Goal,Result),
  run_engine_result(Pattern,NewEngine,Answer,Result).
run_engine_result(Pattern,NewEngine,Answer,failure) :-
  NewEngine = engine(Pattern,fail),
  Answer    = no.
run_engine_result(Pattern,NewEngine,Answer,success(BPattern,B)) :-
  NewEngine = engine(BPattern,B),
  Answer    = the(Pattern).
run_engine_result(Pattern,NewEngine,Answer,S) :-
  S = shift(return(X),C,BPattern,B)
  BPattern  = Pattern,
  NewEngine = engine(Pattern,(C;B)),
  Answer    = the(X).
```

Fig. 6. Interoperable Engines in terms of delimited control.

Abdallah [1] presents a higher-level interface for (conjunctive) delimited control on top of that of Schrijvers et al. [12]. In particular, it features *prompts*, first conceived in a Haskell implementation by Dyvbig et al. [4], which allow shifts to dynamically specify up to what reset to capture the continuation. We believe that it is not difficult to add a similar prompt mechanism on top of our disjunctive version of delimited control.

Interoperable Engines. Tarau and Majumdar's Interoperable Engines [16] propose *engines* as a means for co-operative coroutines in Prolog. An engine is an independent instance of a Prolog interpreter that provides answers to the main interpreter on request.

The predicate `new_engine(Pattern,Goal,Interactor)` creates a new engine with answer pattern `Pattern` that will execute `Goal` and is identified by `Interactor`. The predicate `get(Interactor,Answer)` has an engine execute its goal until it produces an answer (either by proving the `Goal`, or explicitly with `return/1`). After this predicate returns, more answers can be requested, by calling `get/2` again with the same engine identifier. The full interface also allows bi-directional communication between engines, but that is out of scope here.

Figure 6 shows that we can implement the `get/2` engine interface in terms of delimited control (the full code is available in [18, Appendix B.2]). The opposite, implementing disjunctive delimited control with engines, seems impossible as engines do not provide explicit control over the disjunctive continuation. Indeed, `get/2` can only follow Prolog's natural left-to-right control flow and thus we can-

not, e.g., run the disjunctive continuation before the conjunctive continuation, which is trivial with disjunctive delimited control.

Tabling without Non-bactrackable Variables. Tabling [9,15] is a well-known technique that eliminates the sensitivity of SLD-resolution to clause and goal ordering, allowing a larger class of programs to terminate. As a bonus, it may improve the run-time performance (at the expense of increased memory consumption).

One way to implement tabling—with minimal engineering impact to the Prolog engine—is the tabling-as-a-library approach proposed by Desouter et al. [3]. This approach requires (global) mutable variables that are not erased by backtracking to store their data structures in a persistent manner. With the new `reset/3` predicate, this is no longer needed, as (non-backtracking) state can be implemented in directly with disjunctive delimited control.

7 Conclusion and Future Work

We have presented *disjunctive delimited control*, an extension to delimited control that takes Prolog's non-deterministic nature into account. This is a conservative extension that enables implementing disjunction-related language features and extensions as a library.

In future work, we plan to explore a WAM-level implementation of disjunctive delimited control, inspired by the stack freezing functionality of tabling engines, to gain access to the disjunctive continuations efficiently. Similarily, the use of `copy_term/2` necessitated by the current API has a detrimental impact on performance, which might be overcome by a sharing or shallow copying scheme.

Inspired by the impact of conjunctive delimited control, which has brought tabling to SWI-Prolog, we believe that further development of disjunctive delimited control is worthwhile. Indeed, it has the potential of bringing powerful disjunctive control abstractions like branch-and-bound search to a wider range of Prolog systems.

Acknowledgment. We are grateful to Paul Tarau and the anonymous LOPSTR 2021 reviewers for their helpful feedback. Part of this work was funded by FWO grant G0D1419N and by KU Leuven grant C14/20/079.

References

1. Abdallah, S.: More declarative tabling in Prolog using multi-prompt delimited control. CoRR abs/1708.07081 (2017)
2. Danvy, O., Filinski, A.: Abstracting control. In: Proceedings of the LFP 1990, pp. 151–160 (1990)
3. Desouter, B., van Dooren, M., Schrijvers, T.: Tabling as a library with delimited control. TPLP **15**(4–5), 419–433 (2015)
4. Dyvbig, R.K., Jones, S.P., Sabry, A.: A monadic framework for delimited continuations. Technical report 615, Computer Science Department Indiana University (2005)

5. Felleisen, M.: The theory and practice of first-class prompts. In: Proceedings of the POPL 1988, pp. 180–190 (1988)
6. Fierens, D., et al.: Inference and learning in probabilistic logic programs using weighted Boolean formulas. TPLP **15**(3), 358–401 (2015)
7. Ivanovic, D., Morales Caballero, J.F., Carro, M., Hermenegildo, M.: Towards structured state threading in Prolog. In: CICLOPS 2009 (2009)
8. Saleh, A.H., Schrijvers, T.: Efficient algebraic effect handlers for Prolog. TPLP **16**(5–6), 884–898 (2016)
9. Santos Costa, V., Rocha, R., Damas, L.: The YAP Prolog system. TPLP **12**(1–2), 5–34 (2012)
10. Sato, T.: Generative modeling by PRISM. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 24–35. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02846-5_4
11. Schimpf, J.: Logical loops. In: Stuckey, P.J. (ed.) ICLP 2002. LNCS, vol. 2401, pp. 224–238. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45619-8_16
12. Schrijvers, T., Demoen, B., Desouter, B., Wielemaker, J.: Delimited continuations for Prolog. TPLP **13**(4–5), 533–546 (2013)
13. Schrijvers, T., Demoen, B., Triska, M., Desouter, B.: Tor: modular search with hookable disjunction. Sci. Comput. Program. **84**, 101–120 (2014)
14. Schrijvers, T., Wu, N., Desouter, B., Demoen, B.: Heuristics entwined with handlers combined: from functional specification to logic programming implementation. In: Proceedings of PPDP 2014, pp. 259–270. ACM (2014)
15. Swift, T., Warren, D.S.: XSB: extending Prolog with tabled logic programming. TPLP **12**(1–2), 157–187 (2012)
16. Tarau, P., Majumdar, A.: Interoperating logic engines. In: Gill, A., Swift, T. (eds.) PADL 2009. LNCS, vol. 5418, pp. 137–151. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-92995-6_10
17. Van Roy, P.: A useful extension to Prolog's definite clause grammar notation **24**(11), 132–134 (1989)
18. Vandenbroucke, A., Schrijvers, T.: Disjunctive delimited control. CoRR abs/2108.02972 (2021). https://arxiv.org/abs/2108.02972
19. Wood, F.D., van de Meent, J., Mansinghka, V.: A new approach to probabilistic programming inference. In: AISTATS. JMLR Workshop and Conference Proceedings, vol. 33, pp. 1024–1032. JMLR.org (2014)