

Implementing Backjumping by Means of Exception Handling*

WŁODZIMIERZ DRABENT

Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland
(e-mail: drabent@ipipan.waw.pl)

submitted 4 June 2023; revised 25 June 2023; accepted 27 June 2023

Abstract

We discuss how to implement backjumping (or intelligent backtracking) in Prolog by using the built-ins `throw/1` and `catch/3`. We show that it is impossible in a general case, contrary to a claim that “backjumping is exception handling.” We provide two solutions. One works for binary programs; in a general case it imposes a restriction on where backjumping may originate. The other restricts the class of backjump targets. We also discuss implementing backjumping by using backtracking and the Prolog database. Additionally, we explain the semantics of Prolog exception handling in the presence of coroutines.

KEYWORDS: Prolog, intelligent backtracking, backjumping, exception handling

1 Introduction

In this note we first explain the incompatibility between backjumping (or intelligent backtracking) and the exception handling of Prolog. We show that in general it is impossible to implement the former by the latter. Then show how to do this for some restricted cases (Section 3). We present two approaches. The first one is applicable to a restricted but broad class of cases, including binary programs with arbitrary backjumping. For non-binary programs the restriction constrains from where the backjumping may originate. In the second approach the class of available backjump targets is restricted, so the resulting backjumping may only be an approximation of that intended. Section 4 presents an example of each approach. The next section discusses implementing backjumping by means of backtracking and the Prolog database. The report is concluded by a brief discussion of the related work and conclusions (Section 6). In an [appendix](#) we explain the semantics of Prolog exception handling in a presence of coroutines (also known as delays). The main motivation for this work were opinions that exception handling was an ideal tool to implement backjumping ([Robbins et al. 2021](#)).

Preliminaries. This paper employs the standard terminology and basic well-known notions of logic programming ([Apt 1997](#)). So “atom” means atomic formula, and the nodes

* Thanks are due to the anonymous reviewers for their stimulating comments. Jan Wielemaker, David Gelebus and Ed Robbins commented on the content of the [Appendix](#).

of SLD-trees are queries, that is sequences of atoms. Unless stated otherwise, we consider LD-resolution, that is SLD-resolution under the Prolog selection rule. So we do not deal with coroutining/delays. By a *p-atom* we mean an atom with the predicate symbol *p*. *Procedure p* of a program *P* is the set of the clauses of *P* beginning with *p*. By an *answer* (resp. *computed answer*) of a program *P* with a query *Q* we mean $Q\theta$ where θ is a correct (computed) answer substitution for *P* and *Q*.

Now we formalize some concepts often seen as obvious. The rest of this section may be skipped at the first reading, as it is only needed for formal aspects of few fragments of this paper.

Popular ways of explaining Prolog, like the Byrd box model (Byrd 1980), treat the atom selected in a query as a procedure call. The issue of what is the execution of such atom is often supposed to be obvious, and left unexplained. For instance the Prolog standard (Deransart et al. 1996) uses a notion “the execution of” but does not seem to define it (however some hint is given in Fig. 4.5). Here we provide such a definition in terms of LD-trees, following the idea of Drabent and Małuszyński (1988, Def. 5.1, 5.2).

Consider a (possibly infinite) LD-derivation $\mathcal{D} = Q_0, Q_1, \dots$ and a query $Q_i = A, B$ from \mathcal{D} (where *A, B* are sequences of atoms, and *A* is nonempty). The *execution* in \mathcal{D} of (the occurrence of) *A* in Q_i is the part \mathcal{D}' of \mathcal{D} consisting of those queries Q_k ($k \geq i$) which are of the form $Q_k = A', B\theta$, and for $i \leq j < k$, each query Q_j is not an instance of *B*.

In other words, the execution \mathcal{D}' of *A* consists of the queries of \mathcal{D} starting from $Q_i = A, B$ up to the first query of the form $B\theta$, if such query exists. (So otherwise \mathcal{D}' contains Q_i and all the following queries.) If such query exists, the execution $\mathcal{D}' = (A, B), \dots, (B\theta)$ is called *successful*. We may assume that θ is the composition of the mgu's used in the LD-derivation \mathcal{D}' . Then $A\theta$ is the computed answer for *A*.

Now consider a query $Q_l = B', A, B$ (with nonempty B' and *A*) in \mathcal{D} , and assume that \mathcal{D} contains a query Q_j , $l < j$, which is an instance of *A, B*. Let $Q_i = (A, B)\theta$ be the first such query. Then by the *execution* in \mathcal{D} of (the occurrence of) *A* in Q_l we mean the execution in \mathcal{D} of (the occurrence of) $A\theta$ in Q_i .

Let $Q = B', A, B$ (with nonempty *A*) be a node in an LD-tree \mathcal{T} . Consider the derivations that are the branches of \mathcal{T} containing *Q*. The *execution* of the occurrence *A* in *Q* in \mathcal{T} (briefly, the execution of *A*) is the subgraph of \mathcal{T} consisting of the executions of *A* in *Q* in these derivations. Note that if B' is empty then the execution is a tree, otherwise it is a forest.

2 Backjumping and Prolog exception handling

In this section we present and compare backjumping and the Prolog exception handling.

2.1 Backjumping

A Prolog computation can be seen as a depth-first left-to-right traversal of an SLD-tree. Each node with *i* children is visited *i + 1* times (it is first entered from its parent and then from each of its children). Moving from a node to its parent is called *backtracking*. By *backjumping* we mean skipping a part of the traversal, by moving immediately from a node to one of its non-immediate ancestors (called the backjumping *target*). *Intelligent backtracking* (Bruynooghe and Pereira 1984) is backjumping in which it is known that there are no successes in the omitted part of the SLD-tree.

Let us have a more detailed look. Consider a node *Q* with *k* children Q_1, \dots, Q_k (Figure 1). We may say that *k* backtrack points correspond to *Q*; backtracking from Q_i means arriving to *Q* at its *i*-th backtrack point. Obviously, this is followed by visiting Q_{i+1} when $i < k$, and by backtracking to the parent of *Q* when $i = k$. Similarly, backjumping

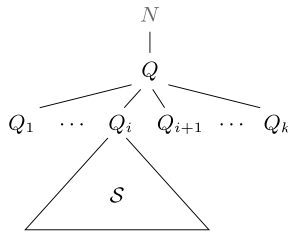


Fig. 1. Backjumping to Q from the subtree S rooted in Q_i (when $i < k$) is followed by visiting Q_{i+1} . For $i = k$ the backjumping is followed by visiting the parent of Q .

Consider exception handling (implemented by making N to be $catch(Q, s, H)$). Exception s raised in S (and not caught in S) results in skipping Q_{i+1}, \dots, Q_k and all their descendants. Additionally, the LD-tree is extended by a subtree corresponding to executing the exception handler H .

to Q from a node in the subtree rooted in Q_i should arrive at the i -th backtrack point. Thus such backjumping is followed by visiting Q_{i+1} when $i < k$, and the parent of Q when $i = k$.

One may consider a generalization of backjumping, where not only a part of the subtree rooted in Q_i is skipped, but also the subtrees rooted in Q_{i+1}, \dots, Q_j (for some $j \in \{i+1, \dots, k\}$). We do not discuss such a generalization here.

2.2 Exception handling

Prolog provides an exception handling mechanism, consisting of built-in predicates $throw/1$ and $catch/3$. Let us follow the Prolog standard and explain them in terms of LD-trees. Assume that a $catch$ -atom $A_c = catch(Q, s, Handler)$ is selected in a node N of an LD-tree. This means the node is of the form $N = A_c, N'$. It is required that Q and $Handler$ are queries. Informally, the execution of A_c means executing Q . More precisely, node A_c, N' has a single child Q, N' . A second child may however be created as a result of exception handling.

An exception t is raised by invoking $throw(t)$. Formally, the tree has a node $N_t = throw(t), N'_t$; such node has no children. This is sometimes called *throwing a ball t* . Visiting N_t starts a search along the path from N_t to the root. The search is for a node $N_c = catch(Q', s', Handler')$, N'_c with one child such that

- (a) a freshly renamed copy t' of the ball t is unifiable with s' , with an mgu θ , and
- (b) “the ball is thrown during the execution of” Q' (Deransart *et al.* 1996).

It is useful to provide a more formal wording for the latter (cf. Preliminaries, Section 1):

- (b) no node between N_c and N_t (including N_t) is an instance of N'_c (this means that each of these nodes is of the form $M, N'_c\sigma$, with a nonempty M).

The first (closest to N_t) such node N_c on the path is chosen, and a new child ($Handler', N'_c$) θ of N_c is added to the tree. The new child becomes the next visited node of the tree. (It is an error if such node N_c is not found.)

2.3 Implementing backjumping

Prolog does not provide any way to directly implement backjumping. It may seem that exception handling is a suitable tool for this task. There is however an important difference.

Backjumping to a query Q cannot be implemented by $catch(Q, s, H)$, as this results in arriving at the last backtrack point corresponding to Q (cf. Figure 1). All the unexplored descendants of Q are omitted. (The same happens if we instead use $catch(Q', s, H), Q''$, where $Q = Q', Q''$). Also, the omitted part of the tree cannot be explored by reconstructing it by the exception handler H , at least in the general case. Roughly speaking, the handler may re-execute Q , but it does not have access to information making it possible to skip the already visited children of Q .

This shows that backjumping to a node Q cannot be implemented by augmenting Q with $catch$. In the next section we show how to obtain such backjumping by adding $catch$ (with $fail$ as the handler) to each child of Q .

The scope of $catch/3$ is the reason of the main limitation; $catch(Q, \dots)$ cannot intercept an exception raised outside the execution of Q . Consider a query A, B and assume that backjumping is to be performed from the execution of B to the execution of A . To implement it by means of exception handling, $catch$ has to appear in a clause used in the execution of A . But then it cannot catch an exception raised in the execution of B .

This discussion shows that backjumping cannot be, in general, directly implemented by means of Prolog exception handling.

We should also mention differences not related to implementing backjumping. In exception handling, after an exception is caught, the exception handler is activated. So the search space is augmented. In backjumping there is nothing similar to an exception handler; the search space is not modified. Also, in contrast to backjumping, exception handling makes it possible to pass information (an arbitrary term) from the point where the exception is raised to the one where it is caught. This is done by means of the argument of $throw/1$.

3 Employing exception handling

This section introduces two restricted approaches to implement backjumping. Examples are given in the next section. We consider Prolog without coroutining/delays.

3.1 Approach 1

Assume that we deal with a definite clause program P , which we want to execute with backjumping. The target of backjumping is to be identified by a term id . So backjumping is initiated by $throw(id)$.

Assume that the target of backjumping is a node A, Q of the LD-tree, where A is an atom. Assume that A, Q has k children, Q_1, \dots, Q_k . Let p be the predicate symbol of A and

$$\begin{aligned} p(\vec{t}_1) &\leftarrow B_1. \\ \dots & \\ p(\vec{t}_n) &\leftarrow B_n. \end{aligned} \tag{1}$$

where $k \leq n$, be the procedure p of program P (i.e. the clauses of P beginning with p).

Consider backjumping initiated by $throw(id)$ in the subtree rooted in Q_i . The subtree should be abandoned, but the descendants of Q_{i+1}, \dots, Q_k should not. Thus we need to restrict the exception handling to this subtree. A way to do this is to replace each B_j by $catch(B_j, id, fail)$. Then performing $throw(id)$ while executing B_j results in failure of

the clause body and backtracking to the next child of A, Q , as required. Assume that a query $btid(\vec{t}, Id)$ (backjump target identifier) produces out of the arguments \vec{t} of p the unique identifier of A, Q as the backjump target. Now, the backjumping is implemented by a transformed procedure consisting of clauses

$$p(\vec{t}_j) \leftarrow btid(\vec{t}_j, Id), catch(B_j, Id, fail). \quad \text{for } j = 1, \dots, n \quad (2)$$

(where Id is a variable).

Transforming a program in this way correctly implements backjumping, however with an important limitation. Backjumping to a node $p(\vec{t}), Q$ must occur during the execution of $p(\vec{t})$. Otherwise the exception is not caught and the whole computation terminates abnormally.

An important class of programs which satisfy this limitation are binary logic programs (i.e. programs with at most one body atom in a clause). The approach presented here works for such programs and arbitrary backjumping. It also works when each backjump target is a node consisting of a single atom.

Sometimes (like in Example 2 below) it may be determined in advance that, for some j , no exception will be caught by the $catch/3$ in (2). So in practice some clauses of (1) may remain unchanged (or a choice between B_j and $catch(B_j, Id, fail)$ may be made dynamically, for example by modifying the body of (2) into $btid(\vec{t}_j, Id) \rightarrow catch(B_j, Id, fail); B_j$).

Approach 1a. Here we present a variant of Approach 1. Roughly speaking, in the former approach control is transferred to the next clause due to failure of a clause body. So catching an exception causes an explicit failure. Here control is transferred to the next clause by means of an exception (hence standard backtracking is to be implemented by means of exceptions). To simplify the presentation we assume that in (1) all the clause heads are the same, $\vec{t}_1 = \dots = \vec{t}_n = \vec{t}$.

Assume first that $n = 2$. Backjumping equivalent to that of Approach 1 can be implemented by

$$p(\vec{t}) \leftarrow btid(\vec{t}, Id), catch((B_1; throw(Id)), Id, catch(B_2, Id, fail)). \quad (3)$$

Invocation of B_2 is placed in the exception handler, so we additionally raise an exception when B_1 (the first clause body) fails. For arbitrary n , the transformed procedure (1) is:

$$p(\vec{t}) \leftarrow btid(\vec{t}, Id), catch((B_1; throw(Id)), Id, catch((B_2; throw(Id)), Id, \dots catch((B_{n-1}; throw(Id)), Id, catch(B_n, Id, fail)) \dots)). \quad (4)$$

Generalizing this transformation to clauses with different heads is rather obvious. Note that in this approach it is possible to augment backjumping by passing information (from

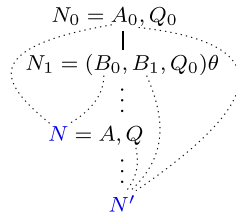


Fig. 2. Dotted lines connect an atom sequence with a query from its execution. N, N' are of the form $N = A, \dots, (B_1, Q_0)\theta\rho$, $N' = throw(id), \dots, Q_0\theta\rho\varphi$.

the place where the backjump originates to the backjump target).¹ Such augmenting is impossible in Approach 1 and Approach 2 below.

3.2 Approach 2, approximate backjumping

We have shown how to implement backjumping to an LD-tree node $N = A, Q$ (with atomic A) from within the execution of A . It remains to discuss backjumping originating in the execution of Q .

Assume that the initial query is atomic; dealing with arbitrary initial queries is similar. In such case, the program contains a clause $H \leftarrow B_0, B_1$ (where B_0, B_1 are nonempty), such that, speaking informally, the backjumping is from within the execution of B_1 , and its target N is within the execution of B_0 .

Let us prove that such clause exists. Assume that the backjumping target is $N = A, Q$ (A is an atom), and that the backjumping originates in a node N' in the execution of Q in N (Figure 2). Let \mathcal{D} be a branch of the LD-tree containing N and N' .

Obviously N occurs in the execution in \mathcal{D} of A in N , and N' does not. As the root of the tree is atomic, there exist in \mathcal{D} a node N_0 and its child N_1 such that N and N' (i) occur in the execution in \mathcal{D} of a single atom A_0 in N_0 and (ii) do not occur in the execution in \mathcal{D} of a single atom in N_1 . Note that A_0 is the first atom of N_0 (otherwise (ii) does not hold). The two nodes are of the form $N_0 = A_0, Q_0$ and $N_1 = (Q', Q_0)\theta$, and a clause $H \leftarrow Q'$ was used to obtain N_1 .

Now Q' can be split as $Q' = B_0, B_1$ (hence $N_1 = (B_0, B_1, Q)\theta$), so that N occurs in the execution of $B_0\theta$ in N_1 , and N' occurs in the execution of $B_1\theta$ in N_1 . So $H \leftarrow B_0, B_1$ has the required property.

We are going to implement backjumping from N' to the last node N'' of the execution of $B_0\theta$ in N_1 . To make N'' as close as possible to N , one should choose B_0, B_1 so that N is in the execution of the last atom of $B_0\theta$.

As discussed in Section 2.3, such backjumping (from within the execution of B_1 to a target in the execution of B_0) cannot be implemented by means of *throw/1* and *catch/3*. What can be done is to force B_1 to fail when an exception is thrown. This means, speaking informally, backjumping to the success of B_0 , instead of the original target N . This in a sense approximates backjumping to N . In some cases such shorter backjumping may still be useful. It may exclude from the search space a major part of what would be excluded by backjumping to N .

¹ To pass a term t , one may choose the backjump target identifier to be $f(X_i)$ for clause i . Then performing *throw(f(t))* while executing B_i results in binding X_i to t when the exception is caught. This makes t available in those bodies B_{i+1}, \dots, B_n that contain X_i . For example for $n = 2$ instead of the body of (3) we obtain *catch((B₁; throw(f(nobj))), f(X₁), catch(B₂, f(X₂), fail))*; constant *nobj* (for “no backjumping”) is passed when standard backtracking takes place.

The success of B_0 (formally of $B_0\theta$) is the topmost descendant of N of the form $N'' = (B_1, Q_0)\theta\rho\psi$ (here $\theta, \rho, Q_0, N', N''$ are as in Figure 2 and the proof above). Node N'' appears in the tree branch between N and node N' that originates the backjump. To catch the exception at N'' we modify the program, so that the instance $B_1\theta\rho\psi$ of B_1 in node N'' is replaced by $\text{catch}(B_1\theta\rho\psi, \text{id}, \text{fail})$. (The resulting backjumping may be understood as arriving to the last backtrack point of N'' , or – due to the handler *fail* – to the parent of N'' .) To obtain this, the clause

$$H \leftarrow B_0, B_1 \quad \text{is transformed to} \quad H \leftarrow B_0, \text{btid}(\dots, \text{Id}), \text{catch}(B_1, \text{Id}, \text{fail})$$

where *btid*, as previously, is used to obtain the unique identifier for the backjump target.

4 Examples

We apply the approaches introduced above to a simple program, a naive SAT solver. It uses the representation of clauses proposed by Howe and King (2012). (Note that we deal here with two kinds of clauses – those of the program, and the propositional clauses of a SAT problem.) A conjunction of clauses is represented as a list of (the representations of) clauses. A clause is represented as a list of (the representations of) literals. A positive literal is represented as a pair **true**- X and a negative one as **false**- X , where the Prolog variable represents a propositional variable. For instance a formula $(x \vee \neg y \vee z) \wedge (\neg x \vee v)$ is represented as $[[\text{true-X}, \text{false-Y}, \text{true-Z}], [\text{false-X}, \text{true-V}]]$. In what follows we do not distinguish literals, clauses, etc from their representations.

Thus solving a SAT problem for a conjunction of clauses *sat* means instantiating the variables of *sat* in such way that each of the lists contains an element of the form *t-t*. This can be done by a program P_1 :

```
sat_cl( [Pol-Pol|Pairs] ).
sat_cl( [H|Pairs] ) :- sat_cl( Pairs ).
sat_cnf( [] ).
sat_cnf( [Clause|Clauses] ) :-
    sat_cl( Clause ), sat_cnf( Clauses ).
```

and a query *sat_cnf(sat)*. See the paper by Drabent (2018, Section 3) for further discussion and a formal treatment of the program.

The examples below add backjumping to program P_1 . They are not intended to provide a correct SAT solver, their role is only to illustrate Approaches 2 and 1. In the examples, the backjumping is performed after a failure of *sat_cl(cl)* (where *cl* is the representation of a partly instantiated clause). The intended backjumping target is the last point where a variable from clause *cl* was assigned a value.

Note that such backjumping does not correctly implement intelligent backtracking, some answers are lost. For example for $(x \vee y) \wedge (\neg z \vee z) \wedge (\neg x \vee \neg y) \wedge (\neg x \vee y \vee z)$ no solution with z being **true** is found. An explanation is that, speaking informally, backjumping from the last clause (with x, y, z instantiated to **true**, **false**, **false** or to **true**, **false**, **false**) arrives to the previous one (where y or x was set), this causes backjumping to the first clause.

Example 1

Here we employ Approach 2 to program P_1 . Speaking informally, the required backjumping originates from within *sat_cnf(Clauses)* in the last clause of the program,

and its target is in *sat_cl(Clause)*. We approximate this backjumping by a failure of *sat_cnf*. (Note that in this case the approximation is good, the intended target is a node of the form *sat_cl([v-V|t])*, *sat_cnf(t')* and we implement backjumping to its child *sat_cnf(t'{V/v})*.)

We augment the values of variables; the value of a variable is going to be of the form (l, v) , where l is a number (the *level* of the variable) and v a logical value **true** or **false**. The level shows at which recursion depth of *sat_cnf* the value was assigned. The levels will be used as identifiers for backjump targets. In such setting, a substitution θ assigning values to variables makes a SAT problem *sat* satisfied when each member of list *sat θ* contains a pair of the form $v-(l, v)$. This leads to transforming the first clause of P_1 to *sat_cl([Pol-(_, Pol)|Pairs])*.

We transform P_1 into a program P_2 which takes levels into account. We add the current level as the second argument of *sat_cnf* and of *sat_cl*. A third argument is added to *sat_cl*; it is used in finding the highest variable level in a clause. The declarative semantics of the new program is similar to that of P_1 ; the answers of P_2 are as follows. If the first argument of *sat_cl* is a list then it has a member of the form $t-(t', t)$. Also, this condition is satisfied by each element of the list that is the first argument of *sat_cnf*.

Operationally, an invariant will be maintained that, whenever *sat_cl(cl, l, hl)* is selected in LD-resolution, *cl* is a list and l and hl are numbers, $l > hl$ and l is greater than any number occurring in *cl*. List *cl* is the not yet processed fragment of a clause cl_0 (possibly instantiated), l is the current level, and hl is the highest level of those variables that occur in the already processed part of cl_0 and have been bound to some values at previous levels; $hl = -1$ when there is no such variable. In case of failure of *sat_cl(cl, l, hl)*, an exception will be raised with the ball being the maximum of hl and the levels of the variables occurring in *cl* (provided the maximum is ≥ 0).

Checking the value already assigned to a variable must be treated differently from assigning a value to an unbound variable (as its level is set only in the latter case). This leads to two clauses playing the role of the first clause of P_1 . So procedure *sat_cl* of P_1 is transformed into the following procedure of P_2 :

```
sat_cl( [Pol-V|_Pairs], _L, _HL ) :-
    nonvar(V), V=(_,Pol). (5)
```

```
sat_cl( [Pol-V|_Pairs], L, _HL ) :-
    var(V), V=(L,Pol). (6)
```

```
sat_cl( [_-V|Pairs], L, HL ) :-
    new_highest( V, HL, HLnew ),
    sat_cl( Pairs, L, HLnew ). (7)
```

Predicate *new_highest* takes care of updating the highest level of the variables from the already processed part of the clause.

```
% new_highest(var, h, hnew) - if var is a Prolog variable then h = hnew
%                               otherwise var = (l, v) and hnew = max(h, l)
new_highest( V, H, H ) :- var( V ). (8)
```

```
new_highest( V, H, H ) :- nonvar( V ), V=(L,_Value), H>=L. (9)
```

```
new_highest( V, H, L ) :- nonvar( V ), V=(L,_Value), H<L. (10)
```


Procedure *sat_cnf* is transformed into

```
sat_cnf( [], _L ).
```

(11)

```
sat_cnf( [Clause|Clauses], L ) :-
    sat_cl( Clause, L, -1 ),
    Lnew is L+1,
    sat_cnf( Clauses, Lnew ).
```

(12)

Program P_2 consists of clauses (5)–(12). An initial query *sat_cnf(sat,0)* results in checking the satisfiability of a conjunction of clauses *sat*.

Now we add backjumping to P_2 . The backjumping has to be triggered instead of a failure of *sat_cl*. The latter happens when the first argument of *sat_cl* is []. The new program P_3 contains the procedure *sat_cl* of P_2 , and additionally a clause

```
sat_cl( [], _, HL ) :- HL >= 0, throw( HL ).
```

(13)

triggering a backjump. When $HL < 0$ then there is no target for backjumping, and standard backtracking is performed.

The procedure *sat_cnf* of the new program P_3 , is constructed out of that of P_2 by transforming clause (12) as described in Approach 2:

```
sat_cnf( [Clause|Clauses], L ) :-
    sat_cl( Clause, L, -1 ),
    Lnew is L+1,
    catch( sat_cnf( Clauses, Lnew ),
          L,
          fail
        ).
```

(14)

So backjumping related to the variable with level l , implemented as *throw(l)*, arrives to an instance of clause (14) where L is l . The whole *catch(...)* fails, and the control backtracks to the invocation of *sat_cl* that assigned the variable. (An additional predicate *btid* was not needed, as L is the unique identifier.)

Now program P_3 consists of clauses (5)–(11) and (13)–(14). To avoid leaving unnecessary backtrack points in some Prolog systems, each group of clauses with *var/1* and *nonvar/1* (clause (5) with (6), and (8) with (9) and (10)) may be replaced by a single clause employing (*var(V) → ...; ...*) and, in the second case additionally ($H < L \rightarrow \dots; \dots$). To simplify a bit the initial queries, a top level predicate may be added, defined by a clause `sat(Clauses) :- sat_cnf(Clauses,0)`.

Example 2

Here we transform P_1 from Example 1 to a binary program and apply Approach 1. The binary program P_b is

```
sat_b( [] ).
sat_b( [[Pol-Pol|_] | Clauses] ) :- sat_b( Clauses ).
sat_b( [_ | Pairs] | Clauses ) :- sat_b( [Pairs | Clauses] ).
```

Note that in Example 1 the unprocessed part of the current clause was an argument of *sat_cl*, now it is the head of the argument of *sat_b*. In what follows we do not explain some details which are as in the previous example.

As previously we introduce levels, and represent a value of a variable by (l, v) , where l is a level and v a logical value. As previously, we first transform P_b into P_{b2} dealing with levels, and then add backjumping to P_{b2} . We add two arguments to *sat_b*, they are the same as the arguments added to *sat_cl* in Example 1. The declarative semantics is similar, the first argument of *sat_b* (in an answer of P_{b2}) is as the first argument of *sat_cnf* in P_2 . An invariant similar to that of Example 1 will be maintained by the operational semantics. Whenever *sat_b*(*cls*, l , hl) is selected, l and hl are numbers, $l > hl$ and l is greater than any number occurring in *cls*. List *cls* is a conjunction of clauses (possibly instantiated), and its head, say *cl*, is the not yet processed fragment of the current clause, say cl_0 ; number l is the current level, and hl is the highest level of variables from the already processed part of cl_0 . Now program P_{b2} is:

$$\text{sat_b}([], _L, _HL). \tag{15}$$

$$\begin{aligned} \text{sat_b}([[\text{Pol-V}|_] \mid \text{Clauses}], L, _HL) &:- \text{nonvar}(V), \\ &V=(_ , \text{Pol}), L_{\text{new}} \text{ is } L+1, \\ &\text{sat_b}(\text{Clauses}, L_{\text{new}}, -1). \end{aligned} \tag{16}$$

$$\begin{aligned} \text{sat_b}([[\text{Pol-V}|_] \mid \text{Clauses}], L, _HL) &:- \text{var}(V), \\ &V=(L, \text{Pol}), L_{\text{new}} \text{ is } L+1, \\ &\text{sat_b}(\text{Clauses}, L_{\text{new}}, -1). \end{aligned} \tag{17}$$

$$\begin{aligned} \text{sat_b}([[_V|\text{Pairs}] \mid \text{Clauses}], L, HL) &:- \\ &L_{\text{new}} \text{ is } L+1, \\ &\text{new_highest}(V, HL, HL_{\text{new}}), \\ &\text{sat_b}([\text{Pairs} \mid \text{Clauses}], L_{\text{new}}, HL_{\text{new}}). \end{aligned} \tag{18}$$

Procedure *new_highest/3* is the same as in the previous example. Program P_{b2} with a query *sat_b*(*sat*, 0, -1) checks satisfiability of the conjunction of clauses *sat*.

Now we add backjumping to P_{b2} . Approach 1 is applicable, as each backjump target of interest consists of a single atom *sat_b*(...). As previously, backjumping originates when an empty clause is encountered:

$$\text{sat_b}([[] \mid _Clauses], _L, HL) :- HL \geq 0, \text{throw}(HL). \tag{19}$$

Let us discuss backjump targets. Speaking informally, we backjump to a place where a variable obtained a value; this happens only in clause (17). So only this clause has to be modified to implement backjump target. For a proof, assume that the nodes of an LD-tree satisfy the invariant. Consider the descendants of a node $N = \text{sat_b}(cls, l, hl)$ obtained by first resolving N with clause (16) or (18). A ball thrown from such a descendant N_2 is not l .² So (16), (18) transformed to the form (2) will never catch a ball, and transforming them is unnecessary.

² Consider a node $N = \text{sat_b}(cls, l, hl)$ and its closest descendant N' of the form *sat_b*(...). So $N' = \text{sat_b}(\dots, l+1, \dots)$. If a number i occurs in a node between N and N' , or in N' , then $i = l+1$ or i occurs in N . By induction, if i occurs in a descendant of N then i occurs in N or $i > l$. Additionally, if N' was obtained by first resolving N with (16) or (18), then N' does not contain l . Thus no descendant of N' contains l .

Out of (17) we obtain:

```

sat_b( [[Pol-V|_] | Clauses], L, _HL ) :-
    catch( ( var(V), V=(L,Pol), Lnew is L+1,
            sat_b(Clauses, Lnew, -1)
            ),
           L,
           fail
        ).

```

(20)

The final program P_{b3} consists of clauses (15), (16), (20), (18), (19), and (8)–(10).

In (20), the first atom $var(V)$ from the body of (17) can be moved outside of $catch$, transforming the body of (20) to $var(V), catch(\dots)$. (This is because $var(V)$ is deterministic and not involved in backjumping.) Now, similarly as in the previous example, some backtrack points may be avoided by replacing clauses (20) and (16) by a single clause with the body $(var(V) \rightarrow V=(L,Pol); V=(_,Pol)), Lnew \text{ is } L+1, sat_b(Clauses, Lnew, -1)$.

Let us also mention that applying Approach 1a to P_{b2} results in adding clause (19) and replacing clauses (16)–(18) by

```

sat_b( [[Pol-V|Pairs] | Clauses], L, HL ) :-
    catch( (nonvar(V), V=(_,Pol), Lnew is L+1, sat_b(Clauses, Lnew, -1)
           ; throw(L)
           ),
           L,
           catch( (var(V), V=(L,Pol), Lnew is L+1, sat_b(Clauses, Lnew, -1)
                  ; throw(L)
                  ),
                  L,
                  catch( (Lnew is L+1, new_highest(V, HL, HLnew),
                          sat_b([Pairs|Clauses], Lnew, HLnew)
                          ; throw(L)
                          ),
                          L,
                          fail
                      )
                )
        ).

```

5 Simulating backjumping by backtracking

This section complements the current paper by presenting an approach not based on exception handling. We discuss simulating backjumping by means of Prolog backtracking, as suggested by Bruynooghe (2004). A backjump is initiated by a failure preceded by depositing in the Prolog database an identifier of the backjump target. At each backtracking step, the database is queried to check if backjumping is being performed and if its target is reached; further backtracking is caused if necessary. This is done by some extra code placed at the beginning of the body of each clause involved in backjumping. (In the presented example (Bruynooghe 2004), there is only one such clause.)

The paper by Bruynooghe is focused on a single example, here we make it explicit how to implement the idea in a general case. Let us introduce a predicate $catch/1$, to deal with backtracking that simulates backjumping. The role of $catch(t)$ is to succeed immediately, unless during backjumping. In the latter case $catch(t)$ fails if t is not unifiable with (the identifier of) the backjumping target. In this way the backjumping is continued.

Otherwise it removes the target from the database and succeeds (instantiating t in the obvious way). This means completing the backjump.

We can use `assert(target(t')), fail` to cause a backjump, and define

$$\begin{aligned} \text{catch}(Id) \leftarrow \text{target}(_) \rightarrow \text{retract}(\text{target}(Id)) & \quad \% \text{ within backjumping} \\ & \quad ; \text{ true} \quad \% \text{ no backjumping} \end{aligned}$$

Query `retract(target(t))` fails when t is not unifiable with the recorded target t' . Otherwise it succeeds and removes the database item, in this way indicating the end of the backjump.

To maintain such simulated backjumping, we convert each clause $p(\vec{t}) \leftarrow B$ of the program into

$$p(\vec{t}) \leftarrow \text{btid}(\vec{t}, Id), \text{catch}(Id), B. \tag{21}$$

where `btid/2` is as in Section 3. Let us present an informal explanation. Note first that if the database is empty then the behaviour of (21) does not differ from that of the original clause. Assume now that backjumping has been initiated, so `target(t')` is asserted and backtracking has started. At each backtrack point, a clause of the form (21) is involved. If `catch` finds that the backjumping target is reached, B is executed. Otherwise `catch` fails, and the simulated backjump continues.

Often for many clauses of the program it is known that `catch(Id)` in (21) will not catch any backjump. In such case the clause can be simplified to

$$p(\vec{t}) \leftarrow \text{!}+(\text{target}(_)), B. \tag{22}$$

Note that there are no restrictions in this approach on the origin or target of backjumping, in contrast to those discussed in Section 3.

6 Final comments

Related work. For the approach of Bruynooghe (2004), see the previous section. Robbins *et al.* (2021) do not present any general approach, but they show a sophisticated example of using Prolog exception handling to implement backjumping. (The main example is preceded by a simple introductory one.) The program is a SAT solver with conflict-driven clause learning. A learned clause determines the target of a backjump. Note that the issue dealt with is not exactly backjumping, understood as in Section 2. In the SAT solver, not only a fragment of the SLD-tree is to be skipped, but additionally the tree has to be extended, as a new clause is added to the SAT problem.

The program employs Prolog coroutining in a fundamental way. It uses exception handling also to deal with plain backtracking. It keeps the learned clauses in the Prolog database, to preserve them during backjumping. The program is rather complicated; it seems impossible to view it as some initial program with added backjumping. To understand it one has to reason about the details of the operational semantics. This is not easy, due to sophisticated interplay of coroutining and exception handling. (The involved semantic issues are discussed here in Appendix A.)

That paper does not propose any general way of adding backjumping to logic programs. The difference between backjumping and Prolog exception handling discussed here in

Section 2 is not noticed.³ We cannot agree with the claims “backjumping is exception handling” and that “`catch` and `throw` [provide] exactly what is required for programming backjumping” (Robbins *et al.* 2021, the title, and pp. 142–143).

Conclusions. The subject of this paper is adding backjumping to logic programs. We discussed the differences between backjumping and Prolog exception handling, and showed that implementing the former by the latter is impossible in a general case (Section 2.3). We proposed two approaches to such implementation. The first approach imposes certain restrictions on where backjumping can be started. The second one – on the target of backjumping. The restrictions seem not severe. The first approach is applicable, among others, to binary programs with arbitrary backjumping. For the second approach, the presented example shows that sometimes the difference between the required and the actual target may be unimportant. As every program can be transformed to a binary one (Maher 1988; Tarau and Boyer 1990), the first approach is indirectly applicable to all cases.

Conflicts of interest

The author declares none.

References

- APT, K. R. 1997. *From Logic Programming to Prolog*. International Series in Computer Science. Prentice-Hall.
- BRUYNNOGHE, M. 2004. Enhancing a search algorithm to perform intelligent backtracking. *Theory and Practice of Logic Programming* 4, 3, 371–380.
- BRUYNNOGHE, M. AND PEREIRA, L. M. 1984. Deduction revision by intelligent backtracking. In *Implementations of Prolog*, J. A. Campbell, Ed. Ellis Horwood/Halsted Press/Wiley, 194–215.
- BYRD, L. 1980. Understanding control flow of Prolog programs. In *Logic Programming Workshop*, S.-Å. Tärnlund, Ed.
- DERANSART, P., ED-DBALI, A. AND CERVONI, L. 1996. *Prolog - the Standard: Reference Manual*. Springer.
- DRABENT, W. 2018. Logic + control: On program construction and verification. *Theory and Practice of Logic Programming* 18, 1, 1–29.
- DRABENT, W. AND MALUSZYŃSKI, J. 1988. Inductive assertion method for logic programs. *Theoretical Computer Science* 59, 133–155.
- HOWE, J. M. AND KING, A. 2012. A pearl on SAT and SMT solving in Prolog. *Theoretical Computer Science* 435, 43–55.
- MAHER, M. J. 1988. Equivalences of logic programs. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan Kaufmann, 627–658.
- ROBBINS, E., KING, A. AND HOWE, J. M. 2021. Backjumping is exception handling. *Theory and Practice of Logic Programming* 21, 2, 125–144.
- TARAU, P. AND BOYER, M. 1990. Elementary logic programs. In *Programming Language Implementation and Logic Programming, PLILP'90*, P. Deransart and J. Maluszynski, Eds. Lecture Notes in Computer Science, vol. 456. Springer, 159–173.

³ This may be due to not facing the limitations pointed out here. Backjumping in the program of Robbins *et al.* (2021) is similar to that of (3) (Approach 1a for $n = 2$), with `throw(Id)` dropped (as there is no standard backtracking), and `catch(B2, Id, fail)` replaced by `B2` (as there is no backjumping from `B2` with the current `Id`).

Appendix A Exception handling in the presence of coroutinging

In this paper we considered Prolog without coroutinging. Coroutinging, also known as delays, is a particular way of modifying the Prolog selection rule. The first atom of the query may not be selected (we say that it is delayed, or blocked). The Prolog built-ins to deal with delays are `when/2`, `freeze/2` (and `block` declarations of SICStus).

The behaviour of exception handling combined with delays seems far from obvious. Programs using both these features, like that of Robbins *et al.* (2021), may be difficult to understand. Explanations are difficult to find. Delays are outside of the scope of the Prolog standard. So it should be useful to provide a formal description. For this we first describe the Prolog coroutinging, in terms of SLD-resolution.

Semantics of coroutinging. This description is restricted to the built-in `when/2`. The other constructs modifying the selection rule can be expressed in terms of `when/2`.

A query $when(C, Q)$ blocks the query Q until the condition C is true. An example condition is $nonvar(X)$, it blocks Q until X is bound to a non-variable term. For the possible form of the condition, we refer the reader to Prolog manuals.

The nodes of SLD-trees are queries; queries are, as usual, sequences of atoms. However in addition to atoms of the underlying logical language, atoms of the form $when(C, Q)$ can be used. (Note that this is a recursive definition, as Q is a query.) At the beginning of a query there may appear some *when*-atoms that are known to be blocked. We separate them by symbol $\&$ from the rest of the query, and we call them the *blocked part* (or delayed part) of the query. We may skip $\&$ when the blocked part is empty. The rest of the query is its *active part*. An invariant will be maintained that in a query $when(C_1, Q_1), \dots, when(C_n, Q_n) \& Q$ each condition C_i ($1 \leq i \leq n$) is not satisfied.

The selection rule of SLD-resolution selects in each query the first atom after $\&$, i.e. the first atom of the active part of a query. The atom will be called the *selected atom* of the query. Now we are ready to describe a resolution step.

Definition 1

A *successor* Q' of a query $Q = D \& A, Q$, where A is the selected atom, is obtained by an extension of an SLD-resolution step as follows.

1. If A is $when(C, Q')$ then
 - (a) if condition C is satisfied then Q' is $D \& Q', Q$,
 - (b) otherwise Q' is $D, A \& Q$,
2. If A is not a *when*-atom then
 - (a) a standard SLD-resolution step is performed (unification of A with a clause head, replacing A with the clause body B , applying the mgu to the resulting query). This produces $Q_1 = (D \& B, Q)\theta$.
 - (b) Let D_{unbl} be those *when*-atoms from $D\theta$ whose conditions are true, and D' be those whose conditions are false. Now Q' is $D' \& D_{unbl}, B\theta, Q\theta$.

In this way we defined the notions of SLD-derivation and SLD-tree for Prolog in the presence of coroutinging. Note that the order of elements of D_{unbl} is not specified. So we do not specify the order of unblocking when several queries are unblocked in the same

resolution step. Such details are not described by the documentation of Prolog systems. (SWI-Prolog seems to preserve in D' and in D_{unbl} the order of atoms from D .)

Let us define (similarly as in Section 1) the *execution* of a query $Q = D \& A, Q$ (in an SLD-derivation \mathcal{D}) to be the part \mathcal{D}' of \mathcal{D} consisting of Q and the following queries of the form $D' \& Q', Q\theta$, up to the first query of the form $D' \& Q\theta$, if such query exists. If the last query of \mathcal{D}' is $D' \& Q\theta$ (i.e. \mathcal{D}' is *successful*) and θ is the composition of the mgu's used in \mathcal{D}' , then $A\theta$ will be called a *pseudo-answer* for A (in \mathcal{D}'). The Prolog debugger displays such pseudo-answer at the Exit port corresponding to the Call port for A . If D, Q are empty and $Q = A$ is the initial query, then Prolog displays (in the standard encoded form) a pseudo-answer $A\theta$, augmented with D' .

Semantics of exception handling. Now we can specify the semantics of Prolog exception handling in the presence of delays, as described in Def. 1. The following modifications to the description of Section 2.2 are sufficient.

- We consider $A_c = \text{catch}(Q, s, \text{Handler})$ selected in a node $D \& A_c, N'$; its child is $D \& Q, N'$.
- The node raising the exception is $N_t = D_t \& \text{throw}(t), N'_t$.
- The search is for a node $N_c = D' \& \text{catch}(Q', s', \text{Handler}'), N'_c$.
- Condition (b) is modified to
 - (b) no node between N_c and N_t (including N_t) is of the form $D'' \& N'_c\theta$.