TOR: Modular Search with Hookable Disjunction

Tom Schrijvers[a,*], Bart Demoen[b], Markus Triska[c], Benoit Desouter[a]

[a] Ghent University, Krijgslaan 281 S9 WE02, 9000 Gent, Belgium
[b] KU Leuven, Celestijnenlaan 200A, 3001 Leuven, Belgium
[c] Vienna University of Technology, Karlsplatz 13, 1040 Wien, Austria

Abstract

Horn Clause Programs have a natural exhaustive depth-first procedural semantics. However, for many programs this semantics is ineffective. In order to compute useful solutions, one needs the ability to modify the search method that explores the alternative execution branches.

TOR, a well-defined hook into Prolog disjunction, provides this ability. It is light-weight thanks to its library approach and efficient because it is based on program transformation. TOR is general enough to mimic search-modifying predicates like ECLiPSe's `search/6`. Moreover, TOR supports modular composition of search methods and other hooks. The TOR library is already provided and used as an add-on to SWI-Prolog.

Keywords: Prolog, tree search, heuristics, modularity

1. Introduction

Kowalski's well-known adage [1] crisply captures the essence of programming in the equation:

$$\textsc{Algorithm} = \textsc{Logic} + \textsc{Control}$$

In Prolog, the logic part is captured in the programmer-supplied rules or clauses that have a first-order logic interpretation. The control component is supplied by the Prolog engine and essentially consists of *search*. In order

*Corresponding author

Email addresses: `Tom.Schrijvers@ugent.be` (Tom Schrijvers), `Bart.Demoen@cs.kuleuven.be` (Bart Demoen), `triska@dbai.tuwien.ac.at` (Markus Triska), `Benoit.Desouter@ugent.be` (Benoit Desouter)

to answer queries, a Prolog engine performs a backward-chaining depth-first tree search.

Prolog's default search strategy is in practice inadequate to effectively scour large search spaces. As a consequence, the programmer often has to complement Prolog's control with additional hints or heuristics in the form of extra code. This is particularly prevalent in the context of Constraint Logic Programming where it is common practice for the programmer to complement a constraint model with a search specification.

Unfortunately, it is not all that easy to cleanly separate logic and control when implementing search heuristics in Prolog. When one discovers that Prolog's control is ineffective, it is often impossible to orthogonally add one's own control without touching the existing logic. The problem is that syntactically logic and control in Prolog are tightly coupled, and adding a different control means cross-cutting existing code.

In this paper we present a novel approach to adding, in an orthogonal manner, control. Our solution features the following properties:

- It is a light-weight library-based approach that is easily portable to different Prolog systems: it is currently an SWI-Prolog library [2] available at `http://www.swi-prolog.org/pack/list?p=tor`.
- Our approach has all the benefits of modularity: search methods can be composed and the library of these heuristics is (user-)extensible.
- We demonstrate on benchmarks that its overhead is negligible compared to typical CLP applications where constraint propagation is the bottleneck. Also we demonstrate that the absolute overhead can be further eliminated through `term_expansion/2`, a feature present in most Prolog systems.

With Tor, we capture all common search methods in CLP(FD) libraries such as ECLiPSe's `search/6` [3]. This approach is indeed particularly suitable for Constraint Logic Programming, but also useful for general Prolog programs with a large search space.

2. Problem Statement

We illustrate the heart of the matter on a simple labeling predicate `label/1` written against SWI-Prolog's `clpfd` library [4] (see Fig. 1, left).

```
label([]).
label([Var|Vars]) :-
  ( var(Var) ->
      fd_inf(Var,Value),
      ( Var #= Value,
        label(Vars)
      ;
        Var #\= Value,
        label([Var|Vars])
      )
  ;
      label(Vars)
  ).
```

```
label([],_).
label([Var|Vars],D) :-
  ( var(Var) ->
      D > 0,
      ND is D - 1,
      fd_inf(Var,Value),
      ( Var #= Value,
        label(Vars,ND)
      ;
        Var #\= Value,
        label([Var|Vars],ND)
      )
  ;
      label(Vars,D)
  ).
```

Figure 1: Labeling predicate: plain (left) and with depth bound (right).

`label/1` defines a search tree where the branches are created by the disjunction.[1]

Suppose that for a certain call `label([X`$_1$`,...,X`$_n$`])` the search tree is too large to fully explore. In order to get some useful answers, certain parts of the tree can be left unexplored, effectively pruning the tree. One particular way in which this can be done is by reaping the low-hanging solutions only, and pruning the subtrees that are below a certain depth. This is achieved by imposing a *depth bound* on Prolog's depth first search. Figure 1 shows on the right a variant of `label/1` that implements this idea; the additional parameter is the depth bound.

Imposing a depth bound may or may not be a successful approach to getting useful answers. If it turns out to be unsuccessful, other pruning strategies can be tried, like imposing a node bound or a discrepancy bound. Each of these requires rewriting the `label/1` predicate to incorporate a different pruning technique. In general, an explorative process takes place whereby

[1] `fd_inf/2` returns the smallest value in a variable's finite domain.

several different variants of the labeling code are written and evaluated until an effective pruning strategy is found.

2.1. Problems with this Approach

The problems with the above approach should be apparent:

- The approach follows the well-known copy-paste-modify anti-pattern. Variants of the labeling code are copied all over the place, potentially propagating bugs and rendering maintenance into a nightmare. Working code is modified.

- The same heuristic is implemented over and over in different settings (different applications, different labeling predicates, different Prolog systems, ...). This process is error-prone, wastes precious programmer time and is bound to yield non-optimal code quality.

- The effort and expertise required to combine working labeling code with various search heuristics is non-trivial. This means that fewer combinations are explored by programmers under time pressure or unfamiliar with particular heuristics. The end result is that suboptimal solutions are obtained.

- As soon as the labeling code spans several different predicates or multiple invocations of the same predicate, the complexity of adding search heuristics increases drastically.

2.2. Current Solutions

Most of the current solutions are specific to CLP, and we are aware of one general Prolog approach.

CLP Solutions. In the context of CLP **ECLiPSe** [3] copes with this problem by providing a number of search methods in the `search/6` predicate. This predicate lets the user control through its various arguments the selection method, the choice method and the search method: the former two decide on which variable is used during labeling, and which value it is assigned first. They do not concern us here. The search method controls how the search tree is explored, e.g., depth-bounded, node-bounded or limited discrepancy search. Apart from individual search methods, only a fixed number of compositions is supported, such as changing the strategy when a depth bound is reached. In this setting users can extend the set of supported heuristics and combinations by reprogramming parts of the `search/6` predicate.

The same approach can be found in other Prolog systems' CLP(FD) libraries, albeit to a more limited extent. **SICStus** Prolog [5] allows imposing discrepancy and time limits, and **B-Prolog** [6] provides a time limit. **GNU Prolog** [7] and **Ciao**'s new `clpfd` library provide no ways to limit the search on top of depth-first.

All CLP(FD) libraries do provide one extra search method: optimization with respect to an objective value. Optimization is typically implemented as either branch-and-bound or by restarting the whole search with a new bound whenever a solution is found.

Typically these approaches only support adding search heuristics to a simple goal made up of a labeling predicate defined in the corresponding CLP library. This means that complex goals made up of a conjunction of labeling calls or custom labeling predicates are not supported. ECLiPSe is the only system that provides one search method, branch-and-bound, independent from a particular labeling predicate.

Prolog Solution. We are aware of only one other approach to modify Prolog's own search method: the breadth-first and iterative deepening program transformations in Ciao [8]. These modify annotated predicates in place and are not compositional.

All in all the available library support that Prolog systems provide is very limited indeed. As soon as users face a (constraint) problem that requires a non-trivial search method, they are forced to write all their search code from scratch, and it can be very daunting to combine different search methods.

Non-Prolog Solutions. There are a range of effective search techniques that are not based on tree search like local search, genetic algorithms, simulated annealing, ... as well as tree search techniques that are not based on depth-first search like breadth-first and A^*. However, these techniques are out of scope of this article. We only consider search methods that are compatible with Prolog's depth-first search.

Article Organization. The rest of this article is structured as follows. First, Section 3 outlines the Tor approach. Next, Section 4 reviews Tor's standard library of search methods. Then, Section 5 covers the Tor's implementation. Section 6 discusses how Tor allows to observe the search tree for (performance) debugging purposes. We illustrate the application of Tor on an example Prolog problem in Section 7. Next, Section 8 evaluates the Tor implementation. In Section 9, we present a simple automatic specializer that mitigates overhead even in applications without constraint propagation using Tor. Section 10 addresses related work and Section 11 concludes.

3. Solution Overview

3.1. User Perspective

Tor divides search code into two parts: a) the code that defines the *search tree*, and b) the code that defines the *search method*. The user defines these separately (or reuses existing definitions from a library) and combines them into a search goal.

Search Tree Code. The search tree code sets up the problem specific search tree. An example of such code is of Figure 1. To fit in the Tor framework one provision that has to be made: the code must use Tor's custom disjunction `tor/2` rather than `;/2`. For instance, `tor_label/1` is the Tor-compatible variant of `label/1`.

```
tor_label([]).
tor_label([Var|Vars]) :-
  ( var(Var) ->
      fd_inf(Var,Value),
      (    Var #= Value,
           tor_label(Vars)
      tor
           Var #\= Value,
           tor_label([Var|Vars])
      )
  ;
      tor_label(Vars)
  ).
```

Search Methods. A search method is defined as a predicate that captures the essence of that method in a declarative way, as a bare-bones search tree without any useful work (such as labeling variables). For instance, `dbs_tree/1` captures the depth-bounded search method.

```
                    Depth-bounded search
dbs_tree(Depth) :-
  Depth > 0,
  Depth1 is Depth - 1,
  ( dbs_tree(Depth1)
  tor
    dbs_tree(Depth1)
  ).
```

Just like the search tree code, the search method code must respect syntactic restrictions: it must be defined as a predicate with a single clause. This clause must contain at most one invocation of `tor/2`. Moreover, each of the two branches of that disjunction may contain at most one directly recursive invocation. Finally, there may be no indirectly recursive calls and no indirect invocations of `tor/2` outside of the recursive calls.

Combining Search Tree and Search Method. The user imposes a search method on a search tree by calling the TOR predicate `tor_merge(MGoal,TGoal)`, where `MGoal` is a call to the search method predicate and `TGoal` is a call to the search tree predicate. Conceptually, `tor_merge/2` overlays or merges the search trees of the two goals, synchronizing their `tor/2` disjunctions.

An example of `tor_merge`'s behavior is graphically depicted in Figure 2. The top left search tree is that of `dbs_tree(4)`, where all the red leaves at level 5 denote failures. The top right search tree is that of `tor_label([X,Y])`, where the blue leaves at various levels denote solutions. The bottom search tree is obtained by merging both other trees. The corresponding leaves are overlaid. When an internal node is overlaid with a leaf, the leaf wins out. If both nodes are internal nodes, the resulting node is an internal node. When both nodes are leaves, the leaf from the left tree wins out.

To facilitate reuse, we generally recommend to encapsulate the application of `tor_merge/2` to a particular search method in a separate predicate, like `dbs/2` for `dbs_tree/1`.

```
dbs(Depth,Goal) :-
  tor_merge(dbs_tree(Depth),Goal).
```

This makes for more concise calls, like `dbs(4,tor_label(Vars))`.

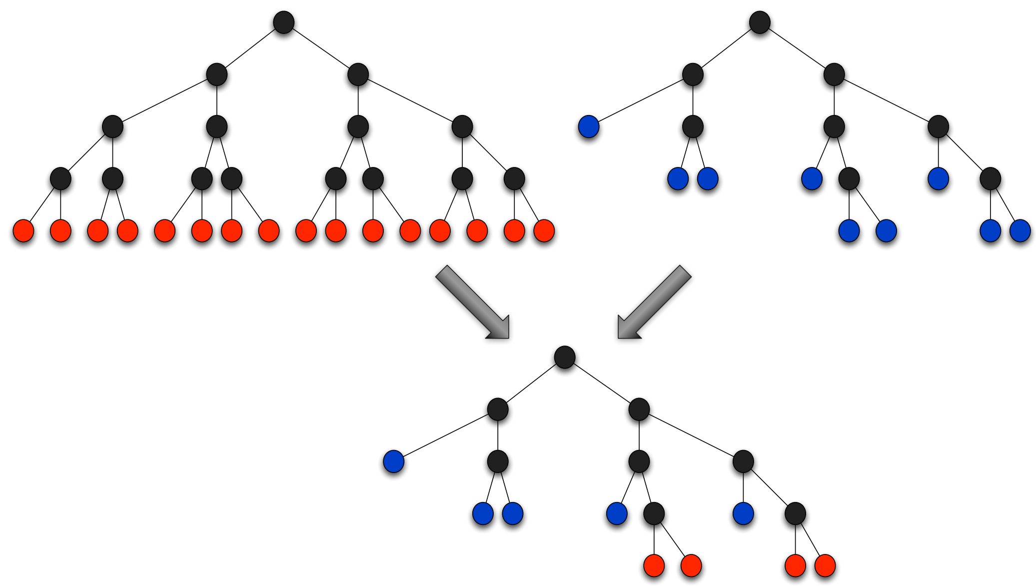

Figure 2: The search trees of `tor_merge(dbs_tree(4),tor_label([X,Y]))`.

Wrapping Up. In the final step, the TOR predicate `search(Goal)` is used to, conceptually, replace all the occurrences (merged or not) of `tor/2` by proper Prolog disjunctions.

In summary, the behavior of `label/2` of Fig. 1 is recovered as follows:

```
search(dbs(Depth,tor_label(Vars)))
                ≡
        label(Vars,Depth)
```

3.2. Modularity Aspects

The big contribution of the TOR approach is its *modularity*. Here we look in more detail at the modularity aspects of TOR that are not found in any of the existing systems.

3.2.1. Decoupling of Search Tree and Search Method

The first modularity advantage of TOR is that it decouples the code that defines the *search tree* from the code that defines the *search method*. This decoupling means that new search methods and new search tree code can be written without awareness of one another and without the modification of any existing code. This means that, once developed, new search methods and labelling code can easily be reused in many different settings. Contrast this with ECLiPSe's `search/6` predicate. It tightly couples the options for setting up the search tree (like variable and value selection strategies) with those for the search method.

Finally, we note that this decoupling does not exclude already supported forms of modularity. In particular, various problem-specific heuristics exist for deciding how to build the search tree. Well-known examples are variable and value selection strategies in CLP(FD) and these are an essential part of an effective search. There are already good solutions for modularizing variable and variable selection strategies in CLP(FD) libraries and Tor does not duplicate their effort. Nevertheless Tor is inherently compatible with these modular solutions: the strategies can easily be integrated in the search tree code. We refer to the companion code of this paper for several examples.

3.2.2. Modular Combination of Search Tree Code and Search Method

Because their implementations are decoupled, there is no inherent restriction on the combination of search tree code with search method code. To make matters more concrete, let us consider an additional search method `lds/1` (short for limited discrepancy search) and an additional search predicate `tor_member/2` (the Tor variant of the well-known `member/2`). We can now express four different search scenarios by varying both the search tree and search method code:

```
?- search(dbs(10,tor_label([X1,...,Xn]))).
?- search(dbs(10,tor_member([X1,...,Xn]))).
?- search(lds(tor_label([X1,...,Xn]))).
?- search(lds(tor_member([X1,...,Xn]))).
```

More concretely, any other search method and labeling predicate can be combined in the same way, whether they originate from the Tor library or are defined by the user. Of course, it is still up to the user to assess which composition is effective for his problem. No CLP(FD) library we are aware of provides this functionality.

3.2.3. Advanced Compositions

Beyond the basic combinations illustrated above, Tor supports the modular composition of multiple search methods and/or multiple labeling goals. None of these are readily expressible in existing CLP(FD) systems.

Composition of Labeling Goals. A user can define a complex labeling goal as the conjunction of two invocations of `tor_label/1`.[2]

```
?- search(lds((tor_label([X1,...,Xn])
              ,tor_label([Y1,...,Ym])))).
```

This example becomes more interesting when the two lists of variables are labeled with different variable and value selection strategies.

Composition of Search Methods. With nested invocation the user can compose two (or more) existing search methods into a new one. This composition denotes that both search methods are simultaneously active in every node of the search tree.

For instance, we can simultaneously apply a depth limit and perform a limited discrepancy search:

```
?- search(dbs(10,lds(tor_label([X1,...,Xn])))).
```

Contrast this with the non-modular approach where the user would face the much more complex task of writing a combined search heuristic `dbs_lds/2` from scratch.

Putting Everything Together. Finally, the compositional nature of the notation can be exploited to its fullest potential to obtain sophisticated search specifications. For instance, the goal

```
?- ...,
   search(lds((dbs(XsLimit,tor_label(Xs))
              ,dbs(YsLimit,tor_label(Ys))))).
```

applies limited discrepancy search to the whole search tree, and additionally imposes one depth-limit on the search of the `Xs` and another to that of the `Ys`.

4. Search Method Library

Following the Tor approach, it is easy to write various search methods in a modular way. While the user can write custom ones himself, Tor already provides a substantial library of search methods. We cover several of them here.

[2]Observe that this example is fundamentally distinct from the simpler goal `?- search(lds((tor_label([X1,...,Xn])))), search(lds((tor_label([Y1,...,Ym])))).`

4.1. Discrepancy-Bounded Search

The discrepancy-bounded search heuristic is a small variant of depth-bounded search: the bound is only updated in right branches.

Discrepancy-bounded search

```
dibs(Discrepancies,Goal) :-
  tor_merge(dibs_tree(Discrepancies),Goal).

dibs_tree(Discrepancies) :-
  ( dibs_tree(Discrepancies)
  tor
    Discrepancies > 0,
    NDiscrepancies is Discrepancies - 1,
    dibs_tree(NDiscrepancies)
  ).
```

4.2. Iterative Deepening

Iterative deepening emulates breadth-first search by means of increasing depth-bounds. The implementation consists of a driver `id_loop/3` that initiates an iteration with a given depth bound and, if pruning occurred, starts the next one with an incremented depth-bound.

An iteration consists of a search with a variant of the depth-bounded heuristic, `id_tree/3`; it differs from depth-bounded search in that it reports its pruning in the non-backtrackable mutable variable `PVar`.[3] This variable communicates to the driver whether a new iteration should be started or not.

Iterative deepening

```
id(Goal) :-
  new_nbvar(not_pruned,PVar),
  id_loop(Goal,0,PVar).

id_loop(Goal,Depth,PVar) :-
  nb_put(PVar,not_pruned),
  ( tor_merge(id_tree(Depth,PVar),Goal)
  ;
    nb_get(PVar,Value),
    Value == pruned,
    NDepth is Depth + 1,
```

[3] See Appendix A for the definition of mutable variables.

```
    id_loop(Goal,NDepth,PVar)
  ).

id_tree(Depth,PruneVar) :-
  ( Depth > 0 ->
      NDepth is Depth - 1,
      ( id_tree(NDepth, PruneVar)
      tor
        id_tree(NDepth, PruneVar)
      )
  ;
    nb_put(PruneVar,pruned),
    false
  ).
```

4.3. Limited Discrepancy Search and Factored Iteration

The traditional limited discrepancy search [9] is a minor variant of iterative deepening. It applies the depth-bound only in right branches. Put differently, limited discrepancy search is to discrepancy-bounded search what iterative deepening is to depth-bounded search.

With some abstraction, we can factor out the common iteration part of iterative deepening and limited discrepancy search:

```
iterate(PGoal) :-
  with_pruned(
    iterate_loop(0,PGoal)).

iterate_loop(N,PGoal) :-
  (
    call(PGoal,N)
  ;
    is_pruned,
    reset_pruned,
    M is N + 1,
    iterate_loop(M,PGoal)
  ).
```

This iteration pattern runs a goal `PGoal` that is parameterized by a natural number `N`. The goal uses this number as a bound and applies pruning when the bound is exceeded. The iteration repeatedly restarts the goal with successive values for `N` until the goal completes without pruning.

With this iteration pattern we can express iterative deepening and limited discrepancy search as follows:

```
Iterative deepening & limited discrepancy search
id(Goal)  :- iterate(flip(dbs,Goal)).
lds(Goal) :- iterate(flip(dibs,Goal)).

flip(Goal,Y,X) :- call(Goal,X,Y).
```

There is only one complicating factor: we need to communicate the pruning from the handler to the iteration. Fortunately, global variables allows us to do that.

```
prune :-
  set_pruned(true),
  fail.

reset_pruned :-
  set_pruned(false).

is_pruned :-
  get_pruned(true).

get_pruned(Flag) :-
  nb_getval(pruned,Flag).

set_pruned(Flag) :-
  nb_setval(pruned,Flag).

with_pruned(Goal) :-
  get_pruned(OldFlag),
  ( reset_pruned,
    call(Goal)
  ;
    set_pruned(OldFlag),
    fail
  ).
```

With the imperative ugliness hidden in the above definitions, the following new definition of `dbs_tree` handler subsumes both `id_tree/2` and the previous `dbs_tree/1` definitions.

```
dbs_tree(Depth) :-
   ( Depth > 0 ->
       Depth1 is Depth - 1,
       ( dbs_tree(Depth1)
       tor
         dbs_tree(Depth1)
       )
   ;
       prune
   ).
```

4.3.1. Node-Bounded Search

A node-bounded search is much like a depth-bounded search, except that the decrements of the limit are not backtracked. Hence, as an optimization we abort the whole search at once by throwing an exception rather than gradually failing out of the search tree.

Node-bounded search

```
nbs(Nodes,Goal) :-
  new_nbvar(Nodes,NodesVar),
  catch(
    tor_merge(nbs_tree(NodesVar),Goal),
    out_of_nodes(NodesVar),
    fail
  ).

nbs_tree(Var) :-
  nb_get(Var,N),
  ( N > 0 ->
    N1 is N - 1,
    nb_put(Var, N1),
    ( nbs_tree(Var)
    tor
      nbs_tree(Var)
    )
  ;
```

```
    throw(out_of_nodes(Var))
  ).
```

4.4. Branch-and-Bound Optimization

This well-known optimization approach posts constraints in the intermediate nodes of the search tree to find increasingly better solutions. Our implementation uses TOR to access those intermediate nodes and generate increasingly larger values of the `Objective` variable. It uses two variables, `BestVar` and `Current`. The former keeps track of the overall best solution so far, while the latter is the solution that the current node tries to improve upon.

Both the overall and current best solution are initialized to a value smaller than the infimum of the objective variable's domain. Whenever a solution is found, the overall best solution is updated. Whenever we backtrack into a TOR choice point, the heuristic synchronizes the current best solution with the overall best solution. If the current best solution was out of sync, the handler also imposes a new lower bound on the objective variable. Note that `inf` denotes negative infinity.

Branch-and-Bound

```
bab(Objective,Goal) :-
  fd_inf(Objective,Inf),
  LowerBound is Inf - 1,
  new_nbvar(LowerBound,BestVar),
  Current = LowerBound,
  tor_merge(bab_tree(Objective,BestVar,Current),Goal),
  nb_put(BestVar,Objective).

bab_tree(Objective,BestVar,Current) :-
  nb_get(BestVar,Best),
  ( Best \= inf , (Current == inf ; Best > Current ) ->
      Objective #> Best,
      NCurrent = Best
  ;
      NCurrent = Current
  ),
  ( bab_tree(Objective,BestVar,NCurrent)
  tor
```

```
    bab_tree(Objective,BestVar,NCurrent)
  ).
```

4.5. More Search Methods

We have implemented many other orthogonal search methods with Tor, including all those offered by ECLiPSe's `search/6` predicate. These can be found in the companion code.

5. Tor Infrastructure Implementation

5.1. Hookable Disjunction

Tor is built around one core predicate, `tor/2`, which replaces the regular Prolog disjunction in search tree code. The predicate is defined as:

```
G1 tor G2 :-
  ( b_getval(left,Left),
    call(Left,G1)   % conceptually: Left(G1)
  ;
    b_getval(right,Right),
    call(Right,G2)  % conceptually: Right(G2)
  ).
```

This definition provides two hooks into the disjunction by means of global variables `left` and `right`.[4] In these hooks the programmer installs *handlers* for the left and right branches to control the search. These handlers are *higher-order* predicates that take a goal and execute it in a (possibly) modified manner.

We obtain standard Prolog disjunction, if we use `call/1` as handler:

```
?- findall(X, ( X in 1..10
              , b_setval(left,call)
              , b_setval(right,call)
              , tor_label([X])
              ), Values).
Values = [1,2,3,4,5,6,7,8,9,10].
```

[4]Note that `b_getval/2` and `b_putval/2` are SWI-Prolog builtins for reading and writing global mutable variables, whose names are atoms. Their non-backtrackable counterparts are `nb_getval/2` and `nb_putval/2`.

The point of TOR is of course to install more interesting handlers.

5.2. From Search Methods to Handlers

More interesting handlers originate from the search method. The `tor_merge/2` predicate transforms their high-level definitions into pairs of low-level handlers, before it installs those handlers. This transformation proceeds in two phases. First the search method definition is normalized, and then the handlers are extracted.

5.2.1. Search Method Normalization

In the first phase, the `rewrite/2` predicate rewrites the search method definition into a normal form

```
sm(X1,...,Xn) :-
  ( Left
  tor
    Right
  ).
```

If `tor/2` is defined as the usual disjunction, both arguments of `rewrite/2` have (on success) the same logical interpretation.

```
rewrite((Head :- Body),(Head :- Left tor Right)) :-
  split(Body,Left,Right).

split(tor(GL,GR),GL,GR) :- !.
split((G1,G2),(GL1,GL2),(GR1,GR2)) :- !,
  split(G1,GL1,GR1),
  split(G2,GL2,GR2).
split((Test -> G1 ; G2),
        (Test -> GL1 ; GL2),(Test -> GR1 ; GR2)) :- !,
  split(G1,GL1,GR1),
  split(G2,GL2,GR2).
split((G1;G2),(GL1;GL2),(GR1;GR2)) :- !,
  split(G1,GL1,GR1),
  split(G2,GL2,GR2).
split(G,G,G).
```

5.2.2. Handler Extraction

The left and right handlers are derived from the `Left` and `Right` branches of the search method's normal form:

```
sm_left(X1,...,Xn,Goal) :-
  NLeft.
sm_right(X1,...,Xn,Goal) :-
  NRight.
```

where `NLeft` and `NRight` are derived from `Left` and `Right` by replacing any recursive calls with `call(Goal)`. Moreover, if any of the recursive calls features parameters that are not the same as in the head, that parameter is wrapped in a mutable variable. For instance, the `Depth` parameter of `dbs_tree/1` changes to `Depth1` in the recursive calls. Hence, the following handlers are derived:

```
dbs_tree_left(MDepth,Goal) :-
  b_get(MDepth,Depth),
  Depth > 0,
  Depth1 is Depth -1,
  b_put(MDepth,Depth1),
  call(Goal).
dbs_tree_right(MDepth,Goal) :-
  ... % identical
```

Finally, a `tor_merge(sm(T1,...,Tn),Goal)` goal is rewritten into the appropriate invocation of `tor_handlers/3`:

```
tor_handlers(Goal, sm_left(T1,...,Tn), sm_right(T1,...,Tn))
```

In case any of the parameters need to be wrapped in a mutable variable, `tor_merge/2` also takes care of that. For instance,

```
?- tor_merge(dbs_tree(4),tor_label(Xs)).
```

becomes

```
?- new_bvar(4,MVar),
   tor_handlers(tor_label(Xs),dbs_tree_left(MVar)
                             ,dbs_tree_right(MVar)).
```

5.3. Handler Infrastructure

5.3.1. Default Handler

The predicate `search/1` sets up the default handler for both hooks: `call/1`.[5]

```
search(Goal) :-
  b_setval(left,call),
  b_setval(right,call),
  call(Goal).
```

With this default handler, `tor/2` corresponds simply to plain disjunction `(;)/2`.[6] For instance, with `search/1` we recover the behavior of `label/1` of Fig. 1 from the TOR-variant:

$$\texttt{search(tor\_label(Vars))} \equiv \texttt{label(Vars)}$$

5.3.2. Extending Installed Handlers

In order to facilitate installing new handlers, TOR provides a convenient predicate: `tor_handlers/3`.

```
tor_handlers(Goal,Left,Right) :-
  b_getval(left,LeftHandler),
  b_getval(right,RightHandler),
  b_setval(left,compose(LeftHandler,Left)),
  b_setval(right,compose(RightHandler,Right)),
    call(Goal),
  b_setval(left,LeftHandler),
  b_setval(right,RightHandler).

compose(G1,G2,Goal) :- call(G1,call(G2,Goal)).
  % conceptually: G1(G2(Goal))
```

This predicate assumes that there are already handlers installed, either by `search/1` or a previous invocation of `tor_handlers/2`. It does not replace

[5] By storing the previous values of the handlers and restoring them after the search, we can easily support nested scopes with entirely different search methods.

[6] Apart from the scope of any cuts in the alternative branches

the installed handlers by the new ones, but composes them with `compose/3`.[7] This accounts for the ability to compose search methods, discussed in Section 3.2.3.

Finally, `tor_handlers/2` also scopes the effect of the new handlers: they are only active in the provided goal. After execution of the goal, the old handlers are reset.

5.4. Custom Low-Level Handlers

In addition to writing high-level search methods, expert users can also exploit TOR's low-level infrastructure and write custom low-level handlers that don't fit the search method pattern. Here we show two such cases.

5.4.1. Higher-Order Search Methods

ECLiPSe's `search/6` provides several *higher-order search methods*. These are search methods that are parameterized by other search methods.

An example of this is the following `dbs/3` variant on depth-bounded search. When it reaches the depth bound, it does not prune the remaining subtree, but activates the search method `Method`. A typical example is to limit the discrepancy once we reach a certain level in the search tree. This is achieved with `dbs(Level,lds(Discrepancies),Goal)`.

```
dbs(Level, Method, Goal) :-
  new_bvar(yes(Level),Var),
  tor_handlers(Goal,dbs_handler(Var,Method)
                   ,dbs_handler(Var,Method)).

dbs_handler(Var,Method,Goal) :-
  b_get(Var,MDepth),
  dbs_handler_(MDepth,Var,Method,Goal).

dbs_handler_(yes(Depth),Var,Method,Goal) :-
  ( Depth > 1 ->
      NDepth is Depth - 1,
      b_put(Var,yes(NDepth)),
      call(Goal)
  ;
```

[7] While `compose` is a ternary predicate, recall that it has to be used in partially applied form in `left` and `right`.

```
      b_put(Var,no),
      call(Method,Goal)
  ).
dbs_handler_(no,_,_,Goal) :-
  call(Goal).
```

The original first-order search method `dbs/2` can be defined as `dbs(Level,prune,Goal)` where:

```
prune(Goal) :- prune.
```

In ECLiPSe, only a fixed number of parameters can be supplied to these higher-order search methods, and `search/6` explicitly caters for each separate combination in its implementation. Not so with Tor. There is no restriction on the possible combinations; the higher-order search methods are truly parametric.

5.4.2. Parallel Search

It turns out that the comparatively simple interface of Tor is even general enough to express at least a naive implementation of parallel search. The query `?- search(parallel(tor_label(Vars),5))` uses 5 processes to explore parts of the search tree in parallel. It is based on the definition of `parallel/2` below.

Parallel

```
parallel(Goal,N) :-
  set_available_processes(N),
  tor_handlers(Goal, tor_fork, call).

tor_fork(Goal) :-
  ( i_am_a_child ->
      call(Goal)
  ;
      wait_for_available_process,
      fork(PID),
      PID == child,
      call(Goal)
  ).
```

The code uses the `fork/1` predicate to duplicate the current Prolog process,[8] yielding a so-called *parent* process and a concurrent *child* process. The child process (determined via `i_am_a_child/0`) explores the goal. Since the goal `PID == child` fails in the parent process, this parent process backtracks and considers the alternative which is delegated by the installed handler to the built-in `call/1` predicate, and whose left tor-branches are again subject to `tor_fork/1`.

We have used three more predicates that need explanation:

- `set_available_processes/1` initializes the number of available (sub-) processes,
- `i_am_a_child/0` succeeds if and only if the current process is not the main Prolog process, and
- `wait_for_available_process/0` waits until a process is available and then succeeds: any time a process is forked, the number of available processes goes down by one, and when a process finishes, the number of available processes goes up by one.

All three predicates can be implemented in an ad-hoc way in SWI-Prolog.

To illustrate the parallel exploration of two independent branches in a simple and self-contained example, consider the query:

```
?- search(parallel(repeat tor X = 2,1)).
```

which yields `X = 2` on the toplevel (a shared resource among all created processes), whereas this specific solution cannot be obtained with regular Prolog disjunction because it is hidden by an infinite branch due to the goal `repeat`.

Clearly, the possibilities of search parallelism based on the Tor framework are worth exploring further, in particular regarding communication between processes, and using threads instead of processes for portability and efficiency.

[8] `fork/1` is available in SWI-Prolog on Unix platforms

6. Search Tree Observation

The original purpose of TOR was to allow the manipulation of search tree traversal by various search heuristics. It turns out that TOR also enables various ways to observe the search tree, so that one can gain insight in the search process itself, e.g., for (performance) debugging purposes. We illustrate in the next sections plain statistics and visualization.

6.1. Statistics

Similar to SWI-Prolog's `profile/1`, `time/1` and `statistics/0` predicates, we can provide different components that monitor various metrics of the search tree and provide us with a convenient summary. In the following example, we constrain 4 finite domain variables to the domain $1, \ldots, 4$ via the library's `ins/2` constraint[9] and emit all solutions found by labeling, including accompanying statistics:

```
?- length(Xs,4), Xs ins 1..4,
   search(tor_statistics((tor_label(Xs),writeln(Xs)))),
   false.
[1,1,1,1]
% Number of solutions: ......... 1
% Number of nodes: ............. 4
% Number of failures: .......... 0
...
[4,4,4,4]
% Number of solutions: ....... 256
% Number of nodes: ........... 510
% Number of failures: .......... 0
```

The code for `tor_statistics/1` is in the TOR library.

To support users who want to check whether they have successfully replaced all regular disjunctions with TOR, we also provide a tool that uses SWI-Prolog's choice point inspection primitive `prolog_current_choice/1` to verify this.

[9] This constraint restricts the domains of the variables in the given list to the given range.

6.2. Visualization

In addition to summarized data of the search tree, we can also visualize the actual search tree itself with Tor. For that purpose, we provide a predicate that emits a textual representation, a log, of the search tree:

```
log(Goal) :-
  tor_merge(log_tree,Goal),
  writeln(solution).

log_tree :-
  ( ( writeln(left)
    tor
      writeln(right)
    ),
    log_tree
  ;
    writeln(false),
    false
  ).
```

A complimentary tool that turns this log into a PDF image is also available from our public code repository. Due to our concise decision to transform the textual logs to scalable vector graphics in PDF format, there is no inherent limit on the sizes of search trees that users of Tor can visualize with this tool.

Fig. 3 shows the complete search tree for labeling 3 variables with domains of size 3 that are not involved in any constraints. The symbol $\top$ denotes that a solution is found at this node, while l and r denote internal nodes generated by left and right branches of `tor/2` respectively.

Fig. 4 shows two search trees for the 8-queens puzzle: The left one was created with depth limit (search strategy `dbs`) 4 and contains no solutions. The right one was created with depth limit 7 and stopped the search after finding the first solution. Hence, only the right-most leaf is a solution. The symbol $\perp$ denotes pruning due to constraint propagation, and ! denotes a node that is not explored because the depth limit is exceeded at this level of the search tree.

It would be interesting to further integrate the logging output with the more powerful CP visualization tool CP-Viz [10].

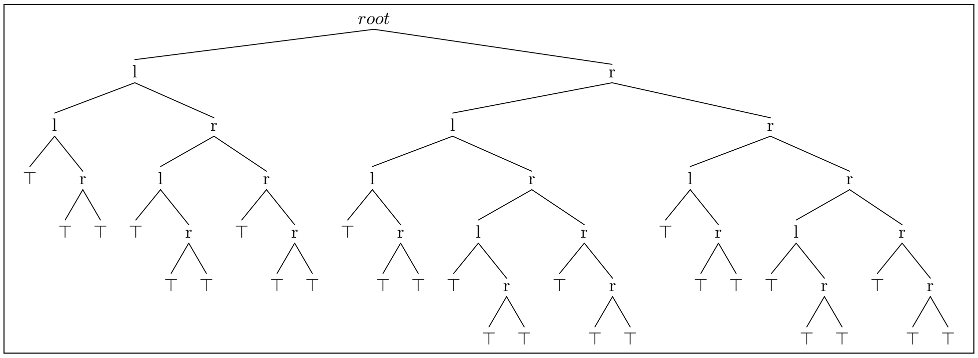

Figure 3: Search tree of `Xs = [_,_,_], Xs ins 1..3, search(log(tor_label(Xs)))`

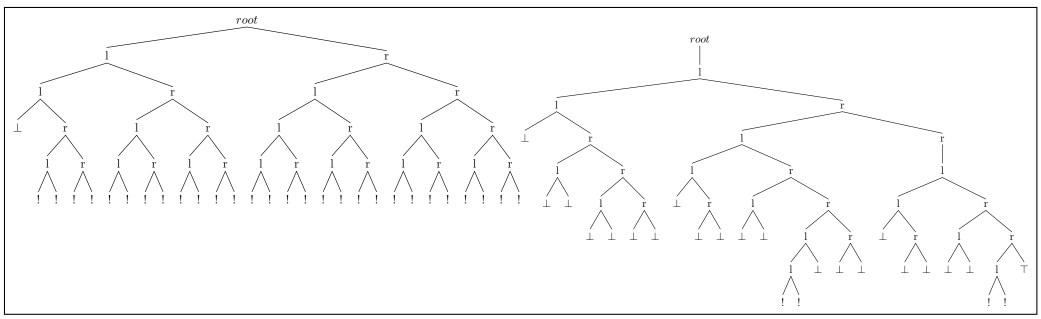

Figure 4: Search trees of 8-queens with depth bound 4 and 7

7. Plain Prolog Example

While the application of Tor to CLP problems is obvious, we wish to emphasize that Tor is not limited to CLP.

For that reason we illustrate the use of Tor on the well-known problem of the wolf, the goat and the cabbage. The following code, adapted from Sterling and Shapiro [11], implements this decision problem in plain Prolog (without constraints). Naive depth-first execution of this code loops infinitely.

```
wgc :-
  initial_state(State),
  wgc(State).

wgc(State) :-
  final_state(State), !.
```

```
wgc(State) :-
  move(State,Move),
  update(State,Move,State1),
  legal(State1),
  wgc(State1).

initial_state(wgc(left, [wolf, goat, cabbage], [])).

final_state(wgc(right, [], [wolf, goat, cabbage])).

move(wgc(Bank, Left, Right),Move) :-
  ( Bank == left,
    tor_member(Move, Left)
  tor
    Bank == right,
    tor_member(Move, Right )
  tor
    Move = alone
  ).

:- tor tor_member/2.
tor_member(X,[X|_]).
tor_member(X,[_|Xs]) :- tor_member(X,Xs).

update(wgc(B,L,R), Cargo, wgc(B1, L1, R1)) :-
  update_boat(B, B1),
  update_banks(Cargo, B, L, R, L1, R1).

update_boat(left, right).
update_boat(right, left).

update_banks(alone, _B, L, R, L, R) :- !.
update_banks(Cargo, left, L, R, L1, R1) :- !,
  select(Cargo, L, L1),
  insert(Cargo, R, R1).
update_banks(Cargo, right, L, R, L1, R1) :-
  select(Cargo, R, R1),
  insert(Cargo, L, L1).

insert(X,[Y|Ys], [X,Y|Ys]) :-
```

```
  precedes(X,Y), !.
insert(X, [Y|Ys], [Y|Zs]) :-
  precedes(Y,X), !,
  insert(X,Ys,Zs).
insert(X, [], [X]).

precedes(wolf, _X).
precedes(_X, cabbage).

legal(wgc(left, _L, R)) :- \+ illegal(R).
legal(wgc(right, L, _R)) :- \+ illegal(L).

illegal(Bank) :- memberchk(wolf, Bank),
                 memberchk(goat, Bank).
illegal(Bank) :- memberchk(goat, Bank),
                 memberchk(cabbage, Bank).
```

The nondeterministic enumeration in this code is situated in the `move/2` and `tor_member/2` predicates.[10] In order to use Tor, we have replaced ordinary Prolog disjunction with `tor/2`.

To avoid the non-termination, we can apply a depth-bound and discover in finite time that the problem has a solution.

```
?- search(dbs(17,wgc)).
true.
```

Of course this is not the only search method that solves the problem. Thanks to Tor, it is convenient to explore many others and to determine the most effective one for the problem at hand.

8. Evaluation

To study Tor's overhead, we have performed a number of benchmarks on a MacBook Pro, with a 2.4 GHz CPU and 4 GB RAM, running Mac OS X 10.6.7. We compare two Prolog systems with different performance

[10] The `tor/1` declaration implicitly adds Tor-disjunctions between the clauses of a predicate.

characteristics. On the one hand we consider SWI-Prolog 5.11.7, a feature-rich, but relatively slow Prolog system with a CLP(FD) solver written in Prolog. On the other hand, we consider B-Prolog 7.5#3, one of the fastest Prolog systems with a highly optimized CLP(FD) implementation.

8.1. Pure Search

Figure 5 considers the extreme situation where the search is pure enumeration of *unconstrained* constraint variables: `length(N,Vars), Vars ins 1..D`. Hence, no constraint propagators are activated due to choices. Values are simply enumerated.

The first column denotes the problem size, expressed in the number of variables `N` and their domain size `D`. The other three pairs of columns denote different implementations of labeling: 1) `label/1` as listed in this paper, 2) `label/1` from SWI-Prolog's `clpfd` library and the corresponding `labeling/1` provided by B-Prolog, and 3) `search/6` ported from ECLiPSe to SWI-Prolog and B-Prolog with minimal changes. For each of these, we show the absolute runtime of the standard/manual version (man) and the relative runtime of the Tor version (tor).

In both SWI-Prolog and B-Prolog the impact of Tor is pretty consistent across the problem sizes, but depends on the labeling implementation. In SWI-Prolog, the overhead is most prominent (140-180 %) in our bare-bones `label/1`, while it is less so (50-60 %) in `clpfd`'s `label/1`. The latter delegates to `labeling/2`, which involves more generic option processing. Finally, in `search/6` Tor compensates its overhead further (to 30-40 %) by not collecting search statistics when these are not demanded. In ECLiPSe's implementation, these statistics are collected regardless of demand.

In B-Prolog, the performance characteristics of the labeling predicates are markedly different. Firstly, the cost of the inequality `(#\=)/2` in our `label/1` is relatively high, which keeps the overhead of Tor low (60%). In contrast, the two other labeling predicates rely on B-Prolog's `domain_inst_next/3` for enumeration, which compiles down to a single abstract machine instruction. As a result the overhead of Tor is much higher, more so in the tight `labeling/1` (170%-230%) than the more bloated `search/6` (120%).

In summary, in these propagation-free benchmarks, the overhead of Tor goes up to about a factor three for tight labeling loops, but is lower for option-rich labeling predicates. Moreover, Tor is better behaved in SWI-Prolog than in B-Prolog. All in all, we find that this is a very reasonable

	our `label/1`		`clpfd`'s `label/1` B-Prolog's `labeling/1`		`search/6`	
	man	Tor	man	Tor	man	Tor
SWI-Prolog						
`N=6,D= 8`	1.80 s	240 %	2.08 s	151 %	2.55 s	132 %
`N=6,D= 9`	3.63 s	249 %	4.20 s	153 %	5.09 s	135 %
`N=6,D=10`	6.82 s	269 %	7.87 s	155 %	9.53 s	137 %
`N=7,D= 8`	14.44 s	244 %	16.63 s	153 %	20.40 s	134 %
`N=7,D= 9`	32.80 s	269 %	37.80 s	155 %	46.04 s	136 %
`N=7,D=10`	68.27 s	278 %	78.63 s	157 %	94.30 s	139 %
B-Prolog						
`N=6,D= 8`	0.49 s	156 %	0.09 s	276 %	0.12 s	223 %
`N=6,D= 9`	0.99 s	157 %	0.18 s	283 %	0.23 s	221 %
`N=6,D=10`	1.87 s	160 %	0.32 s	291 %	0.44 s	219 %
`N=7,D= 8`	4.56 s	144 %	0.71 s	306 %	0.94 s	220 %
`N=7,D= 9`	8.90 s	163 %	1.59 s	301 %	2.06 s	225 %
`N=7,D=10`	18.64 s	163 %	3.25 s	332 %	4.37 s	220 %

Figure 5: Labeling benchmarks without propagation.

	our `ff_label/1`		`labeling/2`		`search/6`	
	man	Tor	man	Tor	man	Tor
SWI-Prolog						
`allinterval`	4.03 s	101 %	4.02 s	101 %	4.01 s	101 %
`golf`	3.93 s	99 %	3.92 s	100 %	3.96 s	99 %
`mhex`	18.59 s	102 %	18.61 s	101 %	18.46 s	101 %
`n_queens`	2.03 s	103 %	2.05 s	102 %	2.09 s	102 %
`sudoku`	2.14 s	101 %	2.15 s	101 %	3.40 s	100 %
B-Prolog						
`allinterval`	1.14 s	100 %	0.81 s	112 %	0.89 s	109 %
`knapsack`	3.94 s	125 %	2.11 s	175 %	2.17 s	172 %
`knight`	0.67 s	101 %	0.71 s	100 %	0.91 s	100 %
`mhex`	0.23 s	106 %	0.19 s	107 %	0.23 s	104 %
`n_queens`	1.01 s	107 %	0.89 s	107 %	1.03 s	106 %

Figure 6: Labeling benchmarks with propagation. (Note that the problem sizes of the benchmarks are not the same for SWI-Prolog and for B-Prolog.)

price to pay for the extra flexibility that Tor provides. Still, invoking Tor's specializer (see the next section) can get rid of all overhead.

8.2. Search vs. Propagation

While the performance penalty of Tor is limited in the previous benchmarks, the performance-wary user may not be willing to accept the overhead. However, the previous benchmarks are not representative of realistic CLP problems, that spend a lot of time on constraint propagation in every node of the search tree. All this extra work easily dwarfs the overhead of Tor. Figure 6 illustrates this observation on a number of typical CLP benchmarks.

For added realism, the benchmarks use the first-fail variable selection strategy, with hand-written labeling code `ff_label/1`, the two library predicates `labeling/2` (SWI-Prolog) and `labeling_ff/1` (B-Prolog), and the

	plain	lds	dibs-1	dibs-2	credit/bbs
`N= 95`	2.11 s	0.66 s	0.45 s	**0.28 s**	0.33 s
`N= 96`	**0.65 s**	4.98 s	4.89 s	1.13 s	1,04 s
`N= 97`	T/O	3.68 s	**3.56 s**	22.66 s	4,08 s
`N= 98`	T/O	15.67 s	† 5.71 s	10.16 s	**2.50 s**
`N= 99`	T/O	2.42 s	**2.22 s**	9.85 s	2.57 s

† no solution

Figure 7: N-Queens benchmarks with various search methods

ported `search/6`. Because B-Prolog's CLP(FD) solver is orders of magnitude faster than SWI-Prolog's, it makes little sense to use exactly the same benchmarks for the two platforms. Instead, we resorted to different problem sizes or different benchmarks altogether.

In the case of SWI-Prolog, we see that Tor introduces no (significant) overhead; its runtime is marginal compared to that of constraint propagation. In the case of B-Prolog, the overhead of Tor is more noticeable, in the order of 10% for most benchmarks. Only in the case of the knapsack problem does it go up to 75% for the tightest labeling loop.

In summary, we see no performance reason to avoid the use of Tor for most CLP problems. Especially in SWI-Prolog there is no runtime price to pay. In the setting of B-Prolog, an extra 10% runtime is a low price for the extra flexibility that Tor provides. Moreover, in the next section we will see how we can eliminate Tor's overhead to the extent that we don't pay for it if we don't use the capabilities it provides.

8.3. Search Methods

Finally, Figure 7 illustrates once more why we want to use different search methods: they can significantly reduce the runtime while still leading to useful solutions. The figure shows the runtime for finding the first solution of the **n_queens** benchmark in SWI-Prolog for 5 different problem sizes and 5 different search methods: (plain) plain depth-first search, (lds) limited discrepancy search, (dibs-1/-2) discrepancy bounds of 1 and 2, and (credit/bbs) credit-based search with 10,000 credits that switches to a bounded backtracking (1 backtrack) search when the credits are exhausted.

9. Automatic Specialization

Tor encourages writing fairly abstract and generic code. This style clearly incurs some overhead (notably due to meta-calling) compared to specialized search code. Fortunately, in the case of CLP applications, this overhead is very modest compared to the cost of constraint propagation. However, in the case of applications without constraint propagation, we do observe an overhead that is significant. In order to mitigate that overhead, we exploit Prolog's homoiconic nature to provide a simple but effective automatic specializer.

Even though there is a large body of work on automatic program specialization for Prolog, notably involving partial evaluation, we decided to write our own program specializer. Its main tasks are 1) to perform *constant propagation* on the global variables `left` and `right`, 2) to replace instantiated meta-calls by direct calls and 3) to inline the handler code into the main search loop. For control we follow a light-weight approach based on declarations of what predicates to inline and specialize.

Example 1 Our specializer yields `label/1` for the generic composition `search(tor_label(Vars))`. Similarly, we recover SWI-Prolog's `labeling/2` by specializing its Tor variant. Hence, we do not pay if we do not modify the search.

Example 2 The specialized form of the goal `search(dbs(N, tor_label(Vars)))` is `new_bvar(N,DVar), label21(Vars, DVar)`, with:

```
label21([], _).
label21([Var|Vars], DVar) :-
  ( var(Var) ->
      fd_inf(Var, Val),
      ( b_get(DVar, Depth),
        Depth>0,
        NDepth is Depth+ -1,
        b_put(DVar, NDepth),
        Var#=Val,
        label21(Vars, DVar)
      ;
        b_get(DVar, G),
        G>0,
        NDepth is G+ -1,
        b_put(DVar, NDepth),
```

```
            Var#\=Val,
            label21([Var|Vars], DVar)
          )
        ;
          label21(Vars, DVar)
        ).
```

This code is slightly less efficient than that of `label/2`. Firstly, the overhead of mutable variables is not entirely eliminated here, as `DVar` is still present. Secondly, the two branches have some code in common that could be shared. However, there are no more meta-calls and all code is inlined in the recursive loop of `label21/2`.

In future work, we intend to get rid of the remaining inefficiencies by implementing additional transformations, including Peter Schachte's approach [12] for eliminating mutable variables adapted to our setting.

10. Related Work

We have already covered the most closely related work, existing approaches to search heuristics in Prolog, in Section 2.2. Here we cover other important related topics.

Combinators. Tor is related to earlier work on *Monadic Constraint Programming* (MCP) [13] in the context of Haskell, and *Search Combinators* [14] in the context of C++ and the Gecode library[11]. In contrast to those works, Tor is tailored towards Prolog's built-in depth-first search and, as a consequence, consists of a much simpler and more elegant design.

Comet. The imperative Comet language [15] features fully programmable search by means of *search controllers* [16]. There are two main differences between Tor and Comet's search controllers. Firstly, search controllers trade simplicity for flexibility, providing more hooks and first-class continuations to manipulate the search. Secondly, search controllers are not intended to be composed, in contrast to Tor's handlers that are explicitly designed to support composition.

[11] `http://www.gecode.org`

Gecode. Gecode [17] is a C++ library for constraint programming that provides two complimentary means to control the search: *search engines* and *branchers*. A valid search consists of a combination of one search engine and one or more branchers. The search engine determines how to navigate the search tree (e.g., depth first search, depth-first search with iterative deepening, ...) and the branchers define the search tree. A typical brancher is defined, like typical CLP(FD) labeling predicates, in terms of a set of variables, and a variable and value selection strategy. Multiple branchers denote a conjunction. Unlike Tor search engines cannot be composed, and all branchers are subject to one and the same search engine.

Aspect-Oriented Programming. The Tor approach is closely related to Aspect-Oriented Programming (AOP) [18, 19]. AOP provides a generic approach for modularly *cross-cutting* existing code with new code, so-called *advice*. This advice is injected in arbitrary *join points* (i.e., program points) based on a *pointcut* predicate.

Obviously Tor is more limited in scope, as only `tor/2` disjunctions are cross-cut and only at the positions of the two hooks. However, we believe that these "limitations" are actually Tor's strength: its simplicity makes it easy to express all common search methods and its discipline favors compositionality.

11. Conclusion and Future Work

We have presented Tor, a light-weight library-based approach for modifying Prolog's depth-first search with reusable and compositional search methods. While the notion of hookable disjunction has enabled a surprisingly large number of possibilities for modifying Prolog search, we still see a few areas that could be improved in future work:

Increased Expressivity. Simplicity has been a guiding principle in the design of Tor. In order to minimize the threshold for users, we keep the effort and complexity of defining and using search methods low. We pay for this simplicity with a somewhat restricted expressivity. An example of a search method that cannot be expressed with Tor is swapping the order of branches in a disjunction. In order to overcome this limitation we would have to add extra complexity to the `tor/2` built-in in the form of an additional hook. However, we choose simplicity over additional expressivity. Nevertheless, TOR is remarkably expressive as it is, covering all of the commonly found search methods in CLP(FD) libraries.

On a more drastic account, we will investigate ways to replace the underlying depth-first queuing strategy. The stack freezing functionality of tabling systems like XSB [20] and YAP [21] provides interesting perspectives for this purpose.

Multiway Disjunctions. TOR currently only supports binary disjunctions; multiway disjunctions have to be decomposed into binary ones. For some applications, this decomposition can be somewhat unnatural. For instance, when enumerating all the values `V` of a constraint variable `X`, one might expect that all alternative assignments `X #= V` sit at the same level in the search tree. This is of course generally not the case in a binary decomposition. For that reason we are considering backward compatible ways to generalize the handler approach.

Declarative State Management. We have hidden the operational aspects of TOR from the programmer with the use of the high-level programming interface for heuristics. Even though the underlying implementation relies on mutable variables, the interface provides a declarative view on state management.

Unfortunately, non-backtrackable state is not covered by the high-level interface; the programmer has to manage it explicitly in an imperative style. The problem is that non-backtrackable state updates are often followed immediately by failure. There is no idiomatic declarative alternative for this technique. However, we could turn to pure deterministic encodings of failure with non-backtrackable state, like Haskell's `ListT (State s)` monad [22] and use Filinski's reification/reflection technique [23] to translate to and from Prolog's native effects.

Acknowledgements. We are grateful to Rémy Haemmerlé, Jose Morales Caballero and David S. Warren for our discussions on TOR, and to Neng-Fa Zhou for revealing B-Prolog's `labeling/1` code to us.

References

[1] R. Kowalski, Logic for Problem Solving, North-Holland, 1979.

[2] J. Wielemaker, T. Schrijvers, M. Triska, T. Lager, SWI-Prolog, Theory and Practice of Logic Programming 12 (2012) 67–96.

[3] J. Schimpf, K. Shen, ECLiPSe From LP to CLP, Theory and Practice of Logic Programming 12 (2012) 127–156.

[4] M. Triska, The finite domain constraint solver of SWI-Prolog, in: Proceedings of the 11th International Symposium on Functional and Logic Programming (FLOPS 2012), volume 7294 of *Lecture Notes in Computer Science*, 2012, pp. 307–316.

[5] M. Carlsson, P. Mildner, SICStus Prolog - The first 25 years, Theory and Practice of Logic Programming 12 (2012) 35–66.

[6] N.-F. Zhou, The language features and architecture of B-Prolog, Theory and Practice of Logic Programming 12 (2012) 189–218.

[7] D. Diaz, S. Abreu, P. Codognet, On the implementation of GNU-Prolog, Theory and Practice of Logic Programming 12 (2012) 253–282.

[8] M. V. Hermenegildo, F. Bueno, M. Carro, P. López-García, E. Mera, J. F. Morales, G. Puebla, An overview of Ciao and its design philosophy, Theory and Practice of Logic Programming 12 (2012) 219–252.

[9] W. D. Harvey, M. L. Ginsberg, Limited discrepancy search, in: Proceedings of the 15th International Joint Conferences on Artificial Intelligence (IJCAI 1995), 1995, pp. 607–613.

[10] H. Simonis, P. Davern, J. Feldman, D. Mehta, L. Quesada, M. Carlsson, A generic visualization platform for cp, in: Proceedings of Principles and Practice of Constraint Programming - CP 2010, volume 6308 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 460–474.

[11] L. Sterling, E. Shapiro, The Art of Prolog: Advanced Programming Techniques, 2. ed., MIT Press, Cambridge, MA, 1994.

[12] P. Schachte, Global variables in logic programming, in: Proceedings of the International Conference on Logic Programming (ICLP 1997), 1997, pp. 3–17.

[13] T. Schrijvers, P. J. Stuckey, P. Wadler, Monadic constraint programming, Journal of Functional Programming 19 (2009) 663–697.

[14] T. Schrijvers, G. Tack, P. Wuille, H. Samulowitz, P. Stuckey, Search Combinators, in: Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP 2011), volume 6876 of *Lectures Notes in Computer Science*, Springer, 2011, pp. 774–788.

[15] P. Van Hentenryck, L. Michel, Constraint-Based Local Search, MIT Press, 2005.

[16] P. Van Hentenryck, L. Michel, Nondeterministic control for hybrid search, Constraints 11 (2006) 353–373.

[17] C. Schulte, et al., Gecode, the generic constraint development environment, 2013. `http://www.gecode.org/`, accessed March 2013.

[18] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 1997), volume 1241 of *Lecture Notes in Computer Science*, 1997, pp. 220–242.

[19] W. Lohmann, G. Riedewald, G. Wachsmuth, Aspect-Orientation in Prolog, in: Proceedings of the 16th International Symposium on Logic-based Program Synthesis and Transformation, 2006.

[20] T. Swift, D. S. Warren, XSB: Extending Prolog with Tabled Logic Programming, Theory and Practice of Logic Programming 12 (2012) 157–187.

[21] V. Santos Costa, R. Rocha, L. Damas, The YAP Prolog system, Theory and Practice of Logic Programming 12 (2012) 5–34.

[22] M. P. Jones, L. Duponcheel, Composing monads, Research Report YALEU/DCS/RR-1004, Yale University, Department of Computer Science, New Haven, Connecticut, 1993.

[23] A. Filinski, Monads in action, in: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010), ACM, 2010, pp. 483–494.

[24] A. Aggoun, N. Beldiceanu, Time stamps techniques for the trailed data in constraint logic programming systems, in: SPLT'90, 8ème Séminaire Programmation en Logique, 16-18 mai 1990, Trégastel, France, 1990, pp. 487–510.

Appendix A. Mutable Variables

TOR's mutable variables (also known as *reference cells*) are implemented by means of mutable terms, as proposed by Aggoun and Beldiceanu [24]. Our implementation for creating, reading and writing such variables comes in a backtrackable and a non-backtrackable version, and is as follows:

```
new_bvar(InitialValue,Var) :-
  var(Var),
  Var = bvar(InitialValue).

b_put(Var,Value) :-
  Var = bvar(_),
  setarg(1,Var,Value).

b_get(bvar(Value),Value).
```

```
new_nbvar(InitialValue,Var) :-
  var(Var),
  Var = nbvar(InitialValue).

nb_put(Var,Value) :-
  Var = nbvar(_),
  nb_setarg(1,Var,Value).

nb_get(nbvar(Value),Value).
```

The non-backtrackable variant of reference cells is useful in case handler information must persist across backtracking.

The mutable variables are available as a separate library at `http://www.swi-prolog.org/pack/list?p=mutable_variables`.