Implementing dif/2 with Open Trees

Mostowski Collapse[1]

[1] XLOG Technologies AG, Switzerland

October 06, 2026

Abstract. We examine the problem of attributed variables from two perspectives. First, we deal with the WAM of a Prolog system, While the basic WAM has extensive tutorial style literature, things might get a little foggy for the more advanced control constructs. We show how catch/3, setup_call_cleanup/3 and undo/1 can be realized inside Dogelog Player without the use of host language native stack by extending the trampoline. While undo/1 can be a building block to manage the state of an attributed variable, the second part is then devoted to cyclic terms as an alternative building block for attributed variable. The report closes with a dif/2 presentation and implementation that is not only performant but can also garbage collect unused attribute variables.

Keywords: Prolog, Constraint Logic Programming, Scryer, EyeProlog

1 Introduction

The first 3 chapters are devoted to the advanced WAM control constructs and their implementation in Dogelog Player. While catch/3 needs a handler stack to eliminate the need for a host language native stack. The setup_call_cleanup/3 and setup_undo/2 implementations make use of continuation modifications.

The last 2 chapters then give an overview how neither of the control constructs are ultimately needed in a novel prototype for the implementation of dif/2. We fallback to en explicit modelling by ‘$ATTR’/2 compounds, but switch the goal list datastructure from backtrackable open lists to backtrackable open trees.

2 5 Minutes Implementation of Prolog catch/3 and throw/1

2.1 Overview

Dogelog Player is a Prolog system for the JavaScript, Python and Java target, mostly written in 100% Prolog itself. Since it runs in cooperative multitasking enviroments, it features a '$YIELD'/1. We noticed that catch/3 causes some friction and we found a simple remedy.

2.2 WAM Machines

Dogelog Player is in some sense a WAM machine, since it has the same WAM state as a WAM machine. While a WAM machine usually names its state in the form of registers mnemonic, using names such as TR, B and CP, that have the following meaning:

```
TR: Trail pointer
B : Backtrack pointer.
CP: Continuation pointer.
```

We have modelled the WAM state as global variables in class Machine:

```
public static Store.Variable trail = null; /* TR */
public static Choice redo = null;          /* B */
public static Object call = null;          /* CP */.
```

A further difference to the classic design persists. The WAM machine in classic setting wants to be also a memory manager and believes in stack style creation and destruction of Prolog terms. But this is not only challenged by environment trimming i.e. Prolog garbage collection.

It is also challenged by exception throwing or 'YIELD'/1. In exception throwing the throw ball has to be somehow copied off from this stack and then later back into this stack. While 'YIELD'/1 might hand over execution to another task, from one stack to another stack.

2.3 Host Try

Because it is doubt full, as we have already explained that a WAM is also a memory manager, Dogelog Player uses the spaghetti stack approach, and relies on the host language to provide memory objects. In as far an exception is a host object either from the host language or Prolog itself.

The old approach was simply to use the host language own try facility to deal with a throw ball. And to decide to handle it or not to handle it. A case in Dogelog Player where the throw ball is never handled, just passed over to the surrounding call, is the launch of a Prolog routine.

```
/**
 * Run a callback once, i.e. no choice point or trailing left behind.
 */
public static boolean launch() {
    Choice snap = snap_setup();
    try {
        found = solve(snap, Boolean.TRUE);
    } finally {
        snap_cleanup(snap);
    }
    return ((Boolean) found).booleanValue();
}
```

The above shows a very simple implemention of a use of a Prolog trampolin solve(). While the trampolin can be used for CALL with argument Boolean.TRUE and for REDO Boolean with argument Boolean.FALSE. The implementation is a once/1 with cleanup.

Nevertheless the above makes already use of more sophisticated mechanism of the host language. Namely the finally block is executed for both EXIT and FAIL, but also for exceptions. So that launch has actually 3 results, since it can fail or succeed, but it can also throw.

2.4 New sys_trap/3

We do not show the old sys_trap/3 here, since it is a complication of launch, that is redoable. It was also implemented in the host language. It used the same means like snap_setup() and snap_cleanup(), but created two choice points. One at the beginning of the call goal and one at the end.

The choice point at the beginning served as a cleanup barrier, in case an exception was thrown and it didn't unify with the 2nd argument of sys_trap/3. If it did unify with the 2nd argument of sys_trap/3, it pushes the 3rd argument on the continuation and left with success.

```
/* Simple sys_trap/3 */
```

```
sys_trap(G, _, _) :-
   '$MARK'(M), sys_push_handler(M),
   G,
   (sys_pop_handler; sys_push_handler(M), fail).
sys_trap(_, _, _) :-
   sys_pop_handler, fail.
```

From what we had implemented natively we extracted the above 100% Prolog code. The main idea was first only to replace the native stack, that was caused by sys_trap/3 as a FFI built-in. The new implementation is not anymore, a FFI built-in, but a 100% Prolog predicate.

2.5 Enhanced Try

Now that we got rid of the native stack, question was when and how do we match again the 2nd argument of sys_trap/3 and push the 3rd argument on the continuation. The new 100&% Prolog sys_trap/3 implementation has two unused places _ where the catcher and the recovery are supplied.

We had to find the right code location that would realizes the original logic that we implemented when we had sys_trap/3 realized via host language. They key idea is here, that we can implement a routine unwind handlers() that can natively do again the catcher unification attempts.

```
/**
 * Run a callback once, i.e. no choice point or trailing left behind.
 */
public static boolean launch() {
    Choice snap = snap_setup();
    try {
        for (; ; ) {
            try {
               found = solve(snap, Boolean.TRUE);
               break;
            } catch (Throwable err) {
               unwind_handlers(err);
            }
        }
    } finally {
        snap_cleanup(snap);
    }
    return ((Boolean) found).booleanValue();
}
```

```

```

As can be seen the idea leads to an alternative implementation of the launch() facility. The locus where exception handling is attempted is now the first Host Language to Prolog transition. The unwind mimics the old Prolog to Host Language to Prolog transitions.

2.6 Refined sys_trap/3

The handler stack is extremly lightweigh to implement, since we have anyways a spaghetti stack and rely on host language memory management. It is simply a array list in the Engine class. We might try to attempt some opimizations in the future, like reusing some fields of the Choice class.

A more user facing optimization is the detection of final deterministic goals. Goals that succeeded but didn't leave some choice points. So there is never a comming back to their execution. In this case the enter choice point and the exit choice point can be removed:

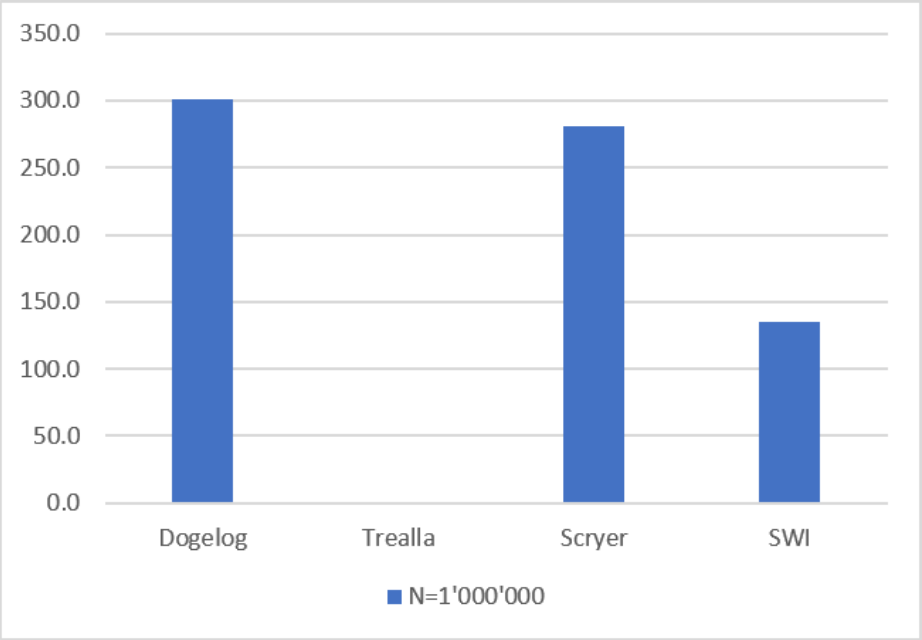
```
/* Refined sys_trap/3 */
sys_trap(G, _, _) :-
   '$MARK'(M), sys_push_handler(M),
   G,
   '$MARK'(N),
   (N == M -> !, sys_pop_handler;
    sys_pop_handler; sys_push_handler(M), fail).
sys_trap(_, _, _) :-
   sys_pop_handler, fail.
```

The later version is what we used in our benchmark. We use a nasty deep nesting of catch/3, and then a simple throw ball, immediately causing the unwinding of all handlers. Our old implementation gave up on this example when N=1'000'000 and with the default Java stack setting:

```
deep(0,G) :- !, G.
deep(N,G) :- M is N-1, catch(deep(M,G),foo,true).

test :- catch(deep(1000000,throw(bar)),_,true).
```

The time measurement in millisecond, averaging 3 runs was:



We tested Dogelog 2.3.1 for Java, Trealla 3.9.40, Scryer 0.10.0 and SWI 10.1.14. Trealla Prolog gave up on the example with an "Execution aborted, possibly because of the surrounding catch/3, and also showing 2.141s. SWI-Prolog had a hick-up in the first run, but then became speedier.

2.7 Summary

The friction was caused by a switch from Prolog, to Host Language and to Prolog so that '$YIELD'/1 had to create choice points. We eliminated both. The new catch/3 realization can easily scale to millions of nestings, since it has become independent of the native stack.

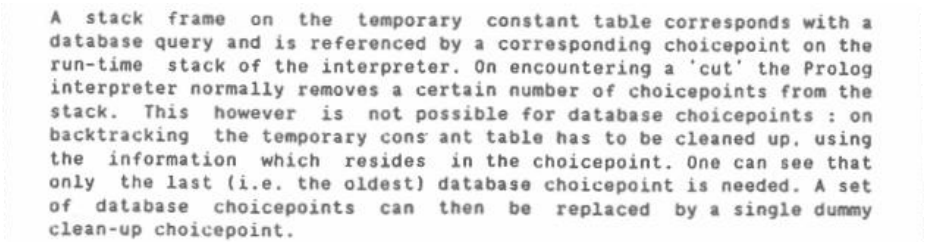
3 A rogue '$CUT'/1 for Dogelog Player

3.1 Overview

Dogelog Player is a Prolog system for the JavaScript, Python and Java target, mostly written in 100% Prolog itself. So far, we only provided once_cleanup/2 to deal with resource management. It turns out we could secure a call_cleanup/2 through a simple mechanism.

3.2 Scanning Choices

The built-in call_cleanup/2 does not originate in SICStus Prolog from 1997, as the ISO core standard effort post-N215 may insinuate. The idea is probably as old as the idea of Logic and Databases from the 1970s. But it has an explicit phrasing in a BIM Prolog document from 1984:



A stack frame on the temporary constant table corresponds with a database query and is referenced by a corresponding choicepoint on the run-time stack of the interpreter. On encountering a 'cut' the Prolog interpreter normally removes a certain number of choicepoints from the stack. This however is not possible for database choicepoints : on backtracking the temporary cons ant table has to be cleaned up, using the information which resides in the choicepoint. One can see that only the last (i.e. the oldest) database choicepoint is needed. A set of database choicepoints can then be replaced by a single dummy clean-up choicepoint.

If we would implement the matter as in formerly Jekejeke Prolog, we would allow choice points to have a cleanup callback. The cleanup callback would then be called involving the native stack. But we just came back from eliminating the native stack for error handling.

We were seeking a solution that didn't use the native stack, and got our inspiration from how error handling was recently solved. In unwinding error handlers, if an error template matches, the error handler is pushed on the continuation and execution goes back to the trampoline.

```
export function rogue(choice) {
    while (choice !== redo) {
        let peek = redo.cont;
        redo = redo.tail;
        peek = peek.args[0];
        if (is_structure(peek) &&
            "sys_cleanup" === site_name(peek.functor) &&
            peek.args.length === 1)
            return deref(peek.args[0]);
    }
```

```
    return undefined;
}
```

The above code shows a utility that is later used. The utility scans the choice point list, and looks for a well-known functor and arity pair, of the goal that caused the choice point. If such a choice point is found, it returns the first argument as the cleanup goal C, otherwise it returns undefined.

3.3 Rogue '$CUT'/1

The call_cleanup/2 semantics requires that cleanup handlers are executed for events in the continuation. One such event is the cut. Our original cut was extremely lightweight. It was not classified as a special predicate, but rather a test predicate. And did only make a pointer assignment:

```
function test_cut(args) {
   let choice = exec_build(args[0]);
   redo = choice;
   return true;
}
```

Special predicates are allowed to arbitrarily manipulate the continuation. In the case of a '$CUT'/1 call, the continuation will start with ['$CUT'(R) | L]. The core idea of the rogue '$CUT'/1 is to have two different continuation transitions depending on whether a cleanup goal is found or not:

```
function special_cut(args) {
    let choice = deref(args[0]);
    let peek = rogue(choice);
    if (peek === undefined) {
        call = call.args[1];
    } else {
        call = new Compound(".", [peek, call]);
    }
    return true;
}
```

The first if-then-else branch is the sunshine case, when there is no cleanup goal above the cut point R. In this case the continuation is transitioned to L. In the other case we

have unfinished business, and transition the list into [C, '$CUT'(R) | L], so that the cleanup goal is next invoked.

3.4 Handler Extension

Since the '$CUT'/1 is now a special predicate it did not anymore fall into the category of predicates that have a fast path in our neck optimization. To solve the problem, we added a new predicate type flag MASK_PRED_ROGUE to detect it, and use the same continuation logic as before again.

The continuation logic breaks for error events in the continuation. Although the unfinished business is the same. For an error event what should follow after a cleanup is not a continuation of '$CUT'(R), but rather further error rippling via '$RAISE'(E). We therefore added a new branching:

```
function unwind_handlers(err) {
    err = map_throwable(err);
    let found = stack_peek(engine.handlers);
    while (found !== null) {
        let peek = rogue(found);
        if (peek === undefined) {
           /* usual handling */
        } else {
            found = new Compound("os_raise", [err]);
            found = new Compound(".", [found, call]);
            found = new Compound(".", [peek, found]);
            call = found;
            return;
        }
    }
    /* usual handling */
}
```

As can be seen we could avoid any native stack calls, and solely go with a continuation manipulation. The transition is the analogue as for the rogue '$CUT'/1, only this time we transition the list into [C, '$RAISE'(E) | L] so that the exception trigger E reenters the trampoline as well.

3.5 Prolog Core

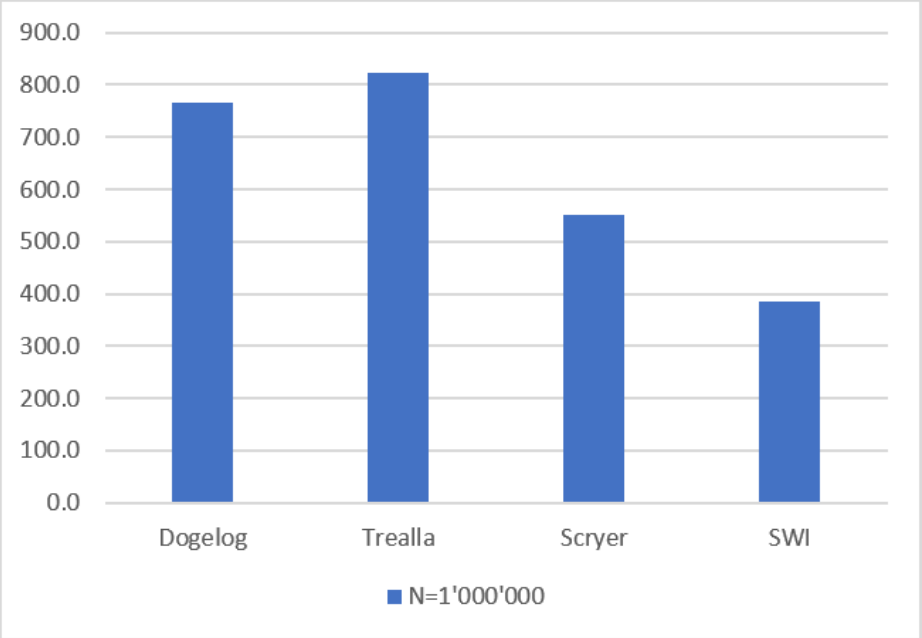
One might wonder how this all fits into the Prolog core. The cleanup choice point is created by clauses for the well-known functor and arity pair sys_cleanup/1. They just leave a choice point and already handle the case that the goal given to call_cleanup/2 fails:

```
sys_cleanup(_).
sys_cleanup(C) :- C, fail.
```

We deploy code that resembles what we did for catch/3. In particular we used a '$MARK'/1 pair and a (==)/2 check to determine when the goal given to call_cleanup/2 does not leave any choice points. The semantic of call_cleanup/2 is then, to not further delay the cleanup:

```
sys_call_cleanup(G, C) :-
   sys_cleanup(C),
   '$MARK'(X),
   G,
   '$MARK'(Y),
   (X == Y -> !; true).
```

The delay is avoided by !, which has now anyways the special_cut() realization, and can remove the sys_cleanup/1 choice point and cause the execution of the cleanup goal C. Our benchmarking, averaged over 3 runs, shows we are competitive to other Prolog systems:



Like when we tested catch/3, we did a deep nesting of call_cleanup/2, but ended it with a true. We tested Dogelog 2.3.1 for Java, Trealla 3.9.40, Scryer 0.10.0 and SWI 10.1.14. This time Trealla Prolog didn't give up. SWI-Prolog had again a hick-up in the first run, but then became speedier.

3.6 Summary

Drawing on a phrasing in a BIM Prolog document from 1984, we could implement the post-N215 semantics of call_cleanup/2. Since we had eliminated the native stack use from catch/3, we could again apply patterns of continuation manipulation for events in the continuation.

4 Getting the birds from the Byrd Box model

4.1 Overview

It turns out some things are more complicated than the mind body problem. Without seeing an analogue behind our recent rogue ‘$CUT’/1 and the requirements of an undo handler. It would have taken us another couple of years before implementing the said gadget.

4.2 Pinned Undo

The eternal duality inside a Prolog system, based on the WAM architecture, are the variable binding trail and the choice point list. The trail records the instantiated variables and allows undoing variable bindings, while the choice point records goals and their redo port.

The current implementation of Dogelog Player uses the trail also as a remembered set. So, the marking of a trail is conditional on the existence of modified Prolog logical variables, and was only done during minor garbage collection so far. This is seen here from the Java target:

```
/**
 * Minor mark the trail
 *
 * @param stop The stop.
 */
private static void mark2_trail(Store.Bind stop) {
    Store.Bind temp = trail;
    while (temp != stop) {
        if (temp instanceof Store.Variable &&
                (((Store.Variable)temp).flags                    &
Store.VAR_MASK_STATE)
                        == Store.VAR_MASK_STATE) {
            mark2_term(temp);
        } else if (temp instanceof Store.Undo) {
            mark2_term(temp.instantiated);
        }
        temp = temp.tail;
    }
}
```

The new addition is a refined class hierarchy, where a class Store.Bind forms the base class of both Store.Variable and Store.Undo. And one sees that Store.Undo

unconditionally marks its undo goal linked via the instantiated field. We also had to add a mark_trail() for the major GC.

4.3 Redo Wakeup

Such an undo made it already into formerly Jekejeke Prolog. With the same drawback that the trail is not very smart. But back then we also afforded using the native stack, to then invoke the undo goal, when the trail was undone. In Dogelog Player we tried just scanning without an action yet:

```
/**
 * Release variable bindings or find a undo goal.
 *
 * @param mark The trail mark.
 * @return Undefined if release completed, otherwise the undo goal.
 */
public static Object release(Store.Bind mark) {
    while (mark != trail) {
        Store.Bind term = trail;
        if (Store.engine.backtrail == term)
            Store.engine.backtrail = term.tail;
        trail = term.tail;
        if (term instanceof Store.Undo) {
            Object goal = term.instantiated;
            term.instantiated = Store.UNDEF_OBJ;
            term.tail = null;
            return goal;
        } else {
            term.instantiated = Store.UNDEF_OBJ;
            term.tail = null;
        }
    }
    return Store.UNDEF_OBJ;
}
```

The action then follows inside the solve() trampoline as part of going into the redo port. The idea is similar to the rogue '$CUT'/1, that places "G, '$CUT'(M)" on the continuation for a cleanup goal G. We place "G, fail", where fail assures that after undo we get back to backtracking:

```
/**
 * Solve Prolog goals.
```

```
 *
 * @param found True for call, and false for redo.
 * @return True if execution succeeds, otherwise false.
 */

/* backtracking part */
Choice choice = redo;
Object goal = release(choice.mark);
if (goal == Store.UNDEF_OBJ) {
    redo = choice.tail;
    call = choice.cont;
    found = choice.func.run(choice.data, choice.at, choice);
} else {
    found = new Store.Compound(".", new Object[]{"fail", call});
    call = new Store.Compound(".", new Object[]{goal, found});
    found = Boolean.TRUE;
}
```

The analogy goes farther, while we then implemented unwind handlers for the rogue '$CUT'/1 by placing "G, '$RAISE'(E)" for cleanup goals G. We do the same now for undo goals. To mirror SWI-Prolog behavior, we priories the undo goals over the cleanup goals for every encountered segment.

4.4 setup_arg/3 Case

The new built-in `ir_trail_undo/1` creates a Store.Undo instance. It thus tries to provide a undo/1 built-in, once shipped by SICStus Prolog, but then removed. But still shipped with SWI-Prolog. One can quickly arrive at the implementation of a set_arg/3, bootstrapped from change_arg/3:

```
/**
 * set_arg(K, X, Y):
 * Same as change_arg/3, but the change is undone upon backtracking.
 */
% set_arg(+Integer, +Callable, +Term)
set_arg(N, C, A) :-
   arg(N, C, B),
   ir_trail_undo(change_arg(N, C, B)),
   change_arg(N, C, A).
```

Because the Store.Undo instance is pinned on the trail and does mark the undo goal. The compound argument of set_arg/3 also becomes pinned, for the lifetime of the

Store.Undo instance. So, the implementation has some non-functional defects, but otherwise works fine:

4.5 Safe undo/1

A little nightmare are Prolog systems that have multi-threading or cooperative multitasking, that allow to be interrupted, and non-finally aborted, i.e. the interrupt not necessarily leading to a halt/0. In set_arg/3 we cheated our way out by using write ahead logging (WAL).

It therefore becomes necessary to address the atomicity between a setup goal call and an undo goal registry. The more luxurious variant of an undo/1 device is this setup_undo/2, which also takes care of the critical section, here using shield/1 as the mutex mechanism:

```
/**
 * setup_undo(S, U):
 * The predicate succeeds whenever S succeeds. The predicate U is
 * called for the first redo port of a goal before or equal S.
 */
% setup_undo(+Goal, +Goal)
setup_undo(S, U) :-
   current_prolog_flag(async_mode, on), !,
   shield(sys_setup_undo(S, suppress(U))).
setup_undo(S, U) :-
  sys_setup_undo(S, suppress(U)).

% sys_setup_undo(+Goal, +Goal)
sys_setup_undo(S, U) :-
   S,
   ir_trail_undo(U).
```

This adds an additional doom and gloom layer to undo/1. We also observed that undo/1 forms not only the basis of set_arg/3 in certain Prolog systems, but also for put_attr/2. We are currently researching alternative approaches that get rid of the critical sections and the pinned objects issues.

4.6 Summary

The new addition is a refined class hierarchy, where a class Store.Bind forms the base class of both Store.Variable and Store.Undo. While we can demonstrate the classical set_arg/3, a backtrackable change_arg/3, we have doubts whether we are already on the right path forward.

5 The Ouroboros Ring Road Puzzle

5.1 Overview

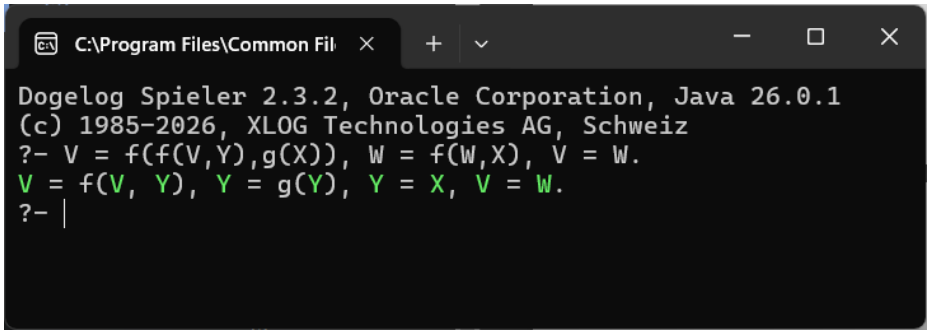
In this post, we look back at working with cyclic terms in a Prolog system, a feature that was introduced in Dogelog Player around 12 months ago. We welcome current developments like SWI-Prologs recent term_minimal/2. They serve us as templates to name our own library predicates.

5.2 The Safe WAM

In Dogelog Player there is no Prolog flag occurs_check. The occurs check is always off. If the end-user wants to avoid the creation of cyclic terms, he has to use the libary(cyclic), which provides unify_with_occurs_check/2. For other Prolog systems or other default settings one might need:

```
% Ensure cyclic terms are allowed!
:- set_prolog_flag(occurs_check, false).
```

How can a Prolog system deal with cyclic terms? Jaffar's Unification is a master piece in engineering, taking a result by Hopcroft and Karp, presented in 1971, namely an almost linear algorithm, for testing the equivalence of two DFAs, that avoids their minimization:



Jaffar's Unification uses Prolog compound head functor pointer swizzling, to build a union find data structure. This is implemented in Dogelog Player for the targets JavaScript, Python and Java. By using a log, we can undo the swizzling even in the case of a stack overflow or memory overflow.

Besides temporary functor swizzling, Jaffar's Unification doesn't do any rewriting of the given Prolog terms. The only side effect is instantiation of Prolog variables, which

is put on another log, the variable binding trail. The variable binding is then typically undone during backtracking.

5.3 Term Decomposition

Our running problem will be an adaptation of a Zebra puzzle. Dogelog Player is visiting the Black Forest, and finds a little village on a ring road named Ouroboros with 3 creatures living in 3 houses. We want to know the names, species and IT skills, but the spooky voices only tell:

```
% Clue 1: 3 creatures in 3 houses live on Ouroboros Ring Road
% Clue 2: The Baskerville Hound guards from the first house
% Clue 3: Sherlock Holmes has the Stack Cache
% Clue 4: Watson is immediately to the left of the Phosphorus Toad
% Clue 5: Cyclic Terms is to the left of the Baskerville Hound
% Clue 6: The P(doom) doomer loops back into the Deep Stack house
% Clue 7: Cthulhu Pup is immediately to the left of the Deep Stack
```

In the following we use a cyclic list to model the ring road:

```
solve(H1, H2, H3) :-
    H1 = house(_, _, _, H2),
    H2 = house(_, _, _, H3),
    H3 = house(_, _, _, H1)
```

When invoking the predicate solve/3, the Prolog system will do unification without occurs check, basically stitching together the 3 compounds in a ring structure, via Prolog variable instantiations. A recurring problem is how to display such ring structures in the top-level.

In the above we see the Scryer Prolog answer substitution display. It belongs to a new breed of Prolog systems, taking cyclic terms more seriously. This new breed doesn't simply use a depth or width limit when writing out terms, but makes efforts to give a proper serialization:

```
/* Scryer Prolog 0.10.0 */
?- solve(H1, H2, H3).
 H1 = house(_A,_B,_C,house(_D,_E,_F,house(_G,_H,_I,H1))),
 H2 = house(_D,_E,_F,house(_G,_H,_I,house(_A,_B,_C,H2))),
 H3 = house(_G,_H,_I,house(_A,_B,_C,house(_D,_E,_F,H3))).
```

```
```

Given that some mathematicians and logicians are obsessed with normal forms, it is no surprise that one can do better. Comparison to other Prolog systems shows that there is still room for improvement. For example, SWI-Prolog gives us a triangular decomposition:

```
/* SWI-Prolog 10.1.15 */
?- solve(H1, H2, H3).
H1 = house(_, _, _, house(_A, _B, _C, house(_D, _E, _F, H1))),
H2 = house(_A, _B, _C, house(_D, _E, _F, H1)),
H3 = house(_D, _E, _F, H1).
```

For Dogelog Player we followed the lead of SWI-Prolog. We had natively implemented our own take of Algorithm E from Knuth, which can give the aforementioned decomposition. For the upcoming release we added some named common expression elimination to the playbook as well:

```
/* Dogelog Player 2.3.2 */
?- solve(H1, H2, H3).
H1 = house(_, _, _, H2),
H2 = house(_, _, _, H3),
H3 = house(_, _, _, H1).
```

5.4 Coinductive Predicates

Coinductive logic programming, abbreviated as co-logic programming, is an absolute nightmare. The coinductive interpretation of the standard member/2 predicate always succeeds, since the ancestor loop checking makes it true, even when the element is not a member of the list.

A way out is the dual predicate; loop checking becomes favorable:

```
:- coinductive(notmember/2).
notmember(_, []).
notmember(X, [Y|L]) :- dif(X, Y), notmember(X, L).
```

The resolution theorem proving of Prolog combined with negation as failure, also known as SLDNF, is too weak to turn a query \+ notmember(X, L) back into its dual

member(X, L). What is lacking is so called constructive negation, that would then succeed, returning instantiations for X.

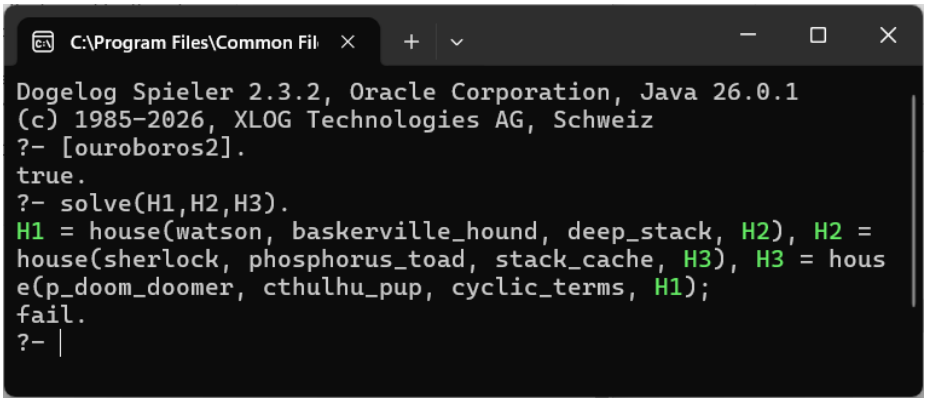
To have the Ouroboros ring road scanned via backtracking we use:

```
in_loop(Pattern, H1) :- Pattern = H1.
in_loop(Pattern, house(_, _, _, H2)) :- Pattern = H2.
in_loop(Pattern, house(_, _, _, house(_, _, _, H3))) :- Pattern = H3.
```

The riddles formalization can then be completed as:

```
H1 = house(_, baskerville_hound, _, _),
in_loop(house(sherlock, _, stack_cache, _), H1),
in_loop(house(watson, _, _, house(_, phosphorus_toad, _, _)), H1),
in_loop(house(_, _, cyclic_terms, house(_, baskerville_hound, _, _)),
H1),
in_loop(house(p_doom_doomer, _, _, house(_, _, deep_stack, _)), H1),
in_loop(house(_, cthulhu_pup, _, house(_, _, deep_stack, _)), H1).
```

And it then has this single solution:



5.5 Term Minimization

Constructive negation was cast back then by David Chan. Some technique that can be helpful is quantifier elimination for the Clark equational theory. But most likely its development was not for cyclic terms. So, for the next 100 years, there is still a lot of work for Dogelog Player.

Another activity area is minimization. We offer an adaption of Algorithm E from Knuth, by trading bisimulation equality (==)/2 for memory equality same_term/2. It is not as performant as the SWI-Prolog addition. But has already a use case in the top-level. While SWI-Prolog only shows:

```
/* SWI-Prolog 10.1.15 */
?- X = f(f(X)).
X = f(f(X)).
```

On the other hand, we show the minimized answer substitution:

```
/* Dogelog Player 2.3.2 */
?- X = f(f(X)).
X = f(X).
```

Another use case is variant equality of cyclic terms. In the case of Dogelog Player, the predicate term_hash/2 is implemented such that it can be safely used with minimized Prolog terms and the bisimulation equality (==)/2. The library(sequence) uses it for deduplication:

```
distinct(Goal) :-
   term_variables(Goal, Key),
   hash_new(Hash),
   Goal,
   term_minimal(Key, MinKey),
   \+ (numbervars(MinKey, -1, _),
       sys_distinct_make(Hash, MinKey)).

sys_distinct_make(Hash, Key) :-
   hash_current(Hash, Key, _), !.
sys_distinct_make(Hash, Key) :-
   copy_term(Key, Key2),
   hash_add(Hash, Key2, true), fail.
```

Variant ordering of cyclic terms is a further use case. We provide a predicate list_factorized/2 that does minimization and decomposition. It can be used to create sort keys. The method is currently deployed in library(aggregate) which also covers the ISO predicates bagof/3 and setof/3.

5.6 Summary

While coinductive predicates are still in its infancy for Prolog systems. We could nevertheless solve the Ouroboros Ring Road puzzle via a cyclic term modelling. The article also touches differences among Prolog systems in cyclic term display and the bagof/3 use case of ordering cyclic terms.

6 The implementation of dif/2 with Open Trees

6.1 Overview

In the following we show that P(purity) has currently almost zero probability, i.e. that a Prolog system will provide the first pillar of purity, i.e. the dif/2 constraint. Even if a Prolog system can deal with a large number of dif/2 constraints, it might still fail the reclamation test.

6.2 Attributed Variables

In the first iteration of library(edge/attrvars) we provided modelling constraints explicitly by '$ATTR'/2 compounds. The idea is not new, was already present in Ulrich Neumerkel's Extensible Unification by Metastructures, 1990. Less than a year ago we used '$ATTR'/2 for CLP(Z).

The motivation to use '$ATTR'/2 compounds was mainly the progression of our Prolog system Dogelog Player. The system had matured that it could deal with cyclic terms, and the Metastructures would be naturally cyclic for the simple case of freeze/2. Moreover, we had bind/2 for var=term events.

```
/* append another list to an open list */
% sys_attr_conc(+List, +List)
sys_attr_conc(V, R) :- var(V), !,
   V = R.
sys_attr_conc([_|L], R) :-
   sys_attr_conc(L, R).
```

The '$ATTR'/2 compounds itself used open lists to hold a multitude of suspended goals. The realization of open lists is pretty much Prolog folklore. We went with the above primitive to extend the 2nd argument of '$ATTR'/2 by a given other list, either open or closed.

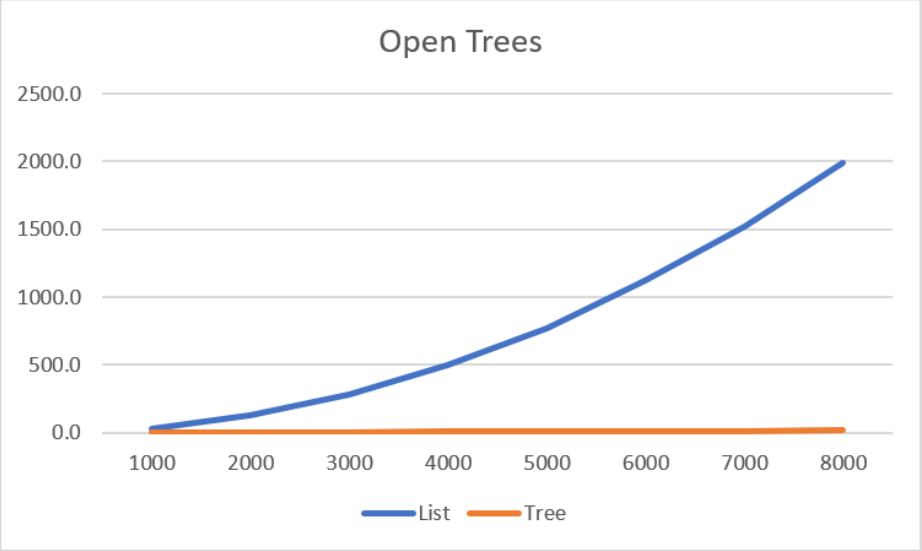
6.3 Open Trees

It recently emerged that open lists are not extremely suitable for frequent adding of constraints. The problem is the sys_attr_conc/2 has to reach the end of the list, before it can use ordinary unification to extend the list. So recently we step over open trees, with a similar API:

```
/* append another tree to an open tree */
% sys_attr_conc(+Tree, +Integer, +Tree).
sys_attr_conc(V, _, N) :- var(V), !,
```

```
   V = N.
sys_attr_conc(node(L, _, R), D, N) :- var(R),
   E is D-2, E >= 0,
   sys_attr_conc(L, E, N), !.
sys_attr_conc(node(_, _, R), D, N) :-
   E is D-1, E >= 0,
   sys_attr_conc(R, E, N).
```

The above implementation is loosely motivated by the trees of bounded balance as found in Nievergelt & Reingold from 1972. The Prolog code profits from the var(R) check as a signpost, by using asymmetric decrement we give the resulting tree a progressive shape.



We do currently no rebalancing, so if we hit the given limit the open tree is considered to be full, and a buffer overflow is thrown. Open trees obviously are backtracking data structure just like open lists. The above empirically shows O(log N) performance of trees versus the O(N) of lists.

6.4 Implementing dif/2

While Ulrich Neumerkel's sketches resemble rather a when/2 implementation of dif/2, where a when/2 uses some frontier suspension strategy, ignoring the instantiation of later variables. We think this doesn't lead to a early firing dif/2. So we went with the solved form approach.

A unification problem of the form s = t of two terms s and t can be viewed as turning the problem into a solved form v1 = t1, .., vn=tn where v1, .., vn are variables and t1, .., tn are terms not necessarily disjoint from the variables. The SWI-Prolog predicate unifiable/3 yields such a list.

To implement dif/2 we realized the same via 100% Prolog:

```
% sys_unifiable(+Term, +Term, +Map, -Map)
sys_unifiable(X, Y, P, Q) :- var(X), var(Y), X == Y, !,
   Q = P.
sys_unifiable(X, Y, P, Q) :- var(Y), !,
   Q = [Y-X|P].
sys_unifiable(X, Y, P, Q) :- var(X), !,
   Q = [X-Y|P].
sys_unifiable(X, Y, P, Q) :- compound(X), compound(Y), !,
   functor(X,F,N), functor(Y,G,M),
   F/N == G/M,
   X =.. [_|L], Y =.. [_|R],
   foldl(unifiable, L, R, P, Q).
sys_unifiable(X, Y, P, Q) :- X == Y,
   Q = P.
```

The predicate sys_unifiable threads an accumulator with the solved form. The solve form is also used inside deref/2, a 100% Prolog implementation for the usual dereferencing of Prolog logical variables, but doing a lookup in the solved form. A dif/2 reduction is then readily implemented as:

```
% reduce(+Goal, -Goal)
reduce(dif(S, T), M) :-
   unifiable(S, T, [], Q), !,
   Q \== [],
   triage(Q, L, R),
   M = dif(L,R).
reduce(dif(_, _), true).
```

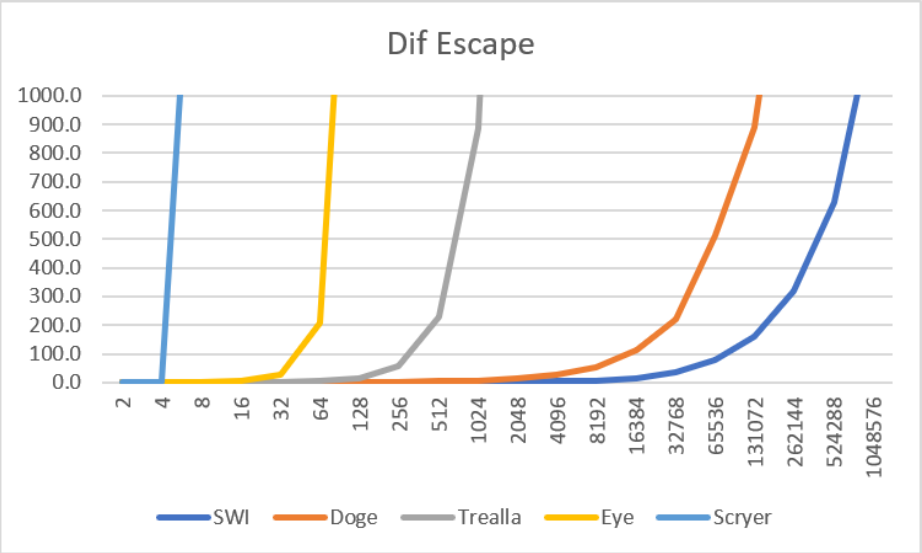
The above implementation has everything that is needed to reduce a dif/2 constraint. It has the success result returning a new goal true, if the two terms are not unifiable. It has the failure result, if they are unifiable and there is nothing to suspend. And it has the maybe result with dif(L, R).

6.5 Performance Results

To put our solution into perspective, we then went on a testing campaign. By using a similar approach like unifiable/4 we also implemented unify/2 for our new development branch of library(edge/attrvars). The later can be used where (=)/2 is suspected to involve attributed variables. We tested:

```
test(N) :-
   length(Xs, N),
   dif(V, X), dif(V, Xs),
   maplist(dif(X), Xs),
   fail.
test(_).
```

It's a test case that currently bogs down Scryer Prolog. By our survey, we could reveal that it is also enemy to Prolog systems such as Trealla Prolog and EyeProlog. Prolog systems that were unimpressed were SWI-Prolog and our Dogelog Player development branch, as this cactus plot shows:



The methodology in our testing was to use a logarithmic scale on the y-axis and progressively test N=2,4,8,etc.. At the same time measuring the milliseconds to execute test/1. We then stopped testing when 1000 ms were reached. The result gives a good overview of the current limitations.

6.6 Space Reclamation

We used the previous data to devise a memory consumption test. Prolog systems that can trim the Prolog variable trail are able to garbage collect open lists and open trees. So attributed variables that make use of such ordinary Prolog terms can reclaim constraint space.

The alternative technique to ordinary Prolog terms are value trails that are then accessed via setarg/3 or put_attr/2. That these implementations might nevertheless have space reclamation issues was already addressed by Ulrich Neumerkel's Precise Garbage Collection in Prolog paper from 2008.

Our development branch should reclaim garbage created by tidy/1:

```
tidy(N) :-
   length(Xs, N),
   dif(V, X), dif(V, Xs),
   maplist(dif(X), Xs).

loop(N) :- tidy(N), loop(N)
```

For testing we used SWI-Prolog 10.1.15 , Dogelog Player 2.3.3, Trealla Prolog 3.9.60, EyeProlog 1.6.30 and Scryer Prolog 0.10.0. We couldn't find a call_with_time_limit/2 in EyeProlog nor in Scryer Prolog. The testing was carried out on a Ryzen AI 7 350 budget AI laptop with Windows 11.

We used procgov to limit the memory to 2GB and run the loop for 60 seconds. To be able to run the Scryer Prolog test at all, we had to remove the dif(V, X), dif(V, Xs) constraint. The Length column shows the used N, and the Count column shows the reached loop/1 iterations.

	Length	Count	Pass	Comment
SWI	1040000	4	No	Stack overflow
Doge	149000	43	Yes	
Trealla	1076	7	No	Memory overflow
Eye	100	4	No	Extrem degradation
Scryer	1300	57	No	Memory overflow

A "No" shows that the Prolog system didn't pass. Either it hit the procgov limit, or run out of some of its own limits, indicating it had a reclamation problem there. Amazingly Dogelog Player was the only Prolog system that survived this additional testing.

6.7 Summary

Testing Ulrich Neumerkels maplist(dif) problem with a time limit of 1000 ms, we found Trealla Prolog, EyeProlog and Scryer Prolog only approached N=1000. While SWI-Prolog and Dogelog Player climbed above N=1000000. Further a garbage collection test was only survived by Dogelog Player.

7 Conclusions

The non-invasive character of the control constructs adds to their small memory footprint and acceptable performance. The handler stack is rather lightweight. The continuations modifications are triggered by cheap scans and have less bureaucracy than a host language native call. But since we do not rely on the control construct for the dif/2 implementation, the reader might anyways have skipped the according chapters. The new dif/2 on the other hand takes advantage of the O(log N) insert complexity of open trees, this undercuts the O(N) insert complexity of open lists. It doesn't destroy the ability to garbage collect, which is still observed.

References

1. E. Wong and K. Youssefi, Demoen, B., *Catch and Throw in WAM: Optimizations and Alternatives*, June 29, 1989
   https://lirias.kuleuven.be/retrieve/b79fb478-b288-40de-8eff-d6e1f80304e1

2. Venken, R., *BIM-Prolog*, September 1984
   https://people.cs.kuleuven.be/~Maurice.Bruynooghe/Prolog/IR06.pdf

3. Ginsberg, M.L., *The MVL Theorem Proving System*, 1991
   https://dl.acm.org/doi/pdf/10.1145/122296.122304

4. Nievergelt, J. and Reingold E.M, *Binary search trees of bounded balance*, STOC '72
   https://dl.acm.org/doi/epdf/10.1145/800152.804906

Appendix

Prolog text for library(edge/attrvars):

```
/**
 * deref(S, T):
 * The predicate succeeds in T with the value of the term S.
 * A variable is elevated to an attributed variable.
 */
% deref(+Term, -Term)
deref(S, R) :- S = '$ATTR'(V,_), var(V), !, R = S.
deref('$ATTR'(S,_), R) :- !, deref(S, R).
deref(S, S).

/**
 * freeze(T, G):
 * The predicate delays the goal G on the term T.
 */
% freeze(+Term, +Goal)
freeze(S, G) :- S = '$ATTR'(V,_), var(V), !,
   sys_attr_add(freeze(V,G), S).
freeze('$ATTR'(S,_), G) :- !, freeze(S, G).
freeze(_, G) :- G.

/*****************************************************************/
/* Unification                                                   */
/*****************************************************************/

/**
 * unify(S, T):
 * The predicate succeeds if S and T unify
 * while maintaining waken goal success.
 */
% unify(+Term, +Term)
unify(S, T) :-
   deref(S, S2),
   deref(T, T2),
   sys_unify(S2, T2).

% sys_unify(+Term, +Term)
sys_unify('$ATTR'(V,_), '$ATTR'(W,_)) :- V == W, !.
sys_unify(X, T) :- T = '$ATTR'(_,_), !, sys_attr_bind(T, X).
sys_unify(S, Y) :- S = '$ATTR'(_,_), !, sys_attr_bind(S, Y).
sys_unify(S, T) :- compound(S), compound(T), !,
   functor(S,F,N), functor(T,G,M),
```

```
   F/N == G/M,
   S =.. [_|L], T =.. [_|R],
   maplist(unify, L, R).
sys_unify(S, T) :- S == T.

% sys_attr_bind(+Term, +Term)
sys_attr_bind('$ATTR'(U,L), '$ATTR'(U,R)) :- !,
   U = '$ATTR'(_,_),
   sys_attr_list(L, A, B),
   sys_attr_list(R, B, []),
   '$SEQ'(nothing, A).
sys_attr_bind('$ATTR'(T,L), T) :-
   sys_attr_list(L, A, []),
   '$SEQ'(nothing, A).

/*****************************************************************/
/* Dif                                                           */
/*****************************************************************/

/**
 * dif(S, T):
 * The predicate posits that S and T are different.
 */
% dif(+Term, +Term)
dif(S, T) :-
   sys_unifiable(S, T, [], Q), !,
   Q \== [],
   sys_triage(Q, L, R),
   A =.. [''|L],
   B =.. [''|R],
   M = '$LATCH'(_,dif(A,B)),
   maplist(sys_attr_add(M), L),
   foldl(sys_attr_put(M), R, L, _).
dif(_,_).

% sys_triage(+Map,  -List, -List)
sys_triage([], [], []).
sys_triage([K-V|L], [K|R], [V|S]) :-
   sys_triage(L, R, S).

% sys_unifiable(+Term, +Term, +Map, -Map)
sys_unifiable(S, T, P, Q) :-
   deref(S, A), sys_attr_deref(A, P, X),
   deref(T, B), sys_attr_deref(B, P, Y),
   sys_attr_unifiable(X, Y, P, Q).
```

```
% sys_attr_deref(+Term, +Map, -Term)
sys_attr_deref('$ATTR'(V,_), P, R) :- member('$ATTR'(W,_)-S, P), V
== W, !,
   sys_attr_deref(S, P, R).
sys_attr_deref(X, _, X).

% sys_attr_unifiable(+Term, +Term, +Map, -Map)
sys_attr_unifiable('$ATTR'(X,_), '$ATTR'(Y,_), P, Q) :- X == Y, !,
   Q = P.
sys_attr_unifiable(X, T, P, Q) :- T = '$ATTR'(_,_), !,
   Q = [T-X|P].
sys_attr_unifiable(S, Y, P, Q) :- S = '$ATTR'(_,_), !,
   Q = [S-Y|P].
sys_attr_unifiable(X, Y, P, Q) :- compound(X), compound(Y), !,
   functor(X,F,N), functor(Y,G,M),
   F/N == G/M,
   X =.. [_|L], Y =.. [_|R],
   foldl(sys_unifiable, L, R, P, Q).
sys_attr_unifiable(X, Y, P, Q) :- X == Y,
   Q = P.

/*******************************************************************/
/* Open Trees                                                      */
/*******************************************************************/

% '$LATCH'(+Var, +Goal)
'$LATCH'(V, G) :- var(V), !, V = 1, G.
'$LATCH'(_, _).

% sys_attr_add(+Goal, +Attr)
sys_attr_add(G, '$ATTR'(_,L)) :-
   sys_attr_conc(L, 30, node(_,G,_)), !.
sys_attr_add(_, _) :-
   throw(error(resource_error(buffer_overflow),_)).

% sys_attr_put(+Goal, +Term, +List, -List)
sys_attr_put(_, '$ATTR'(V,_), P, P) :- member('$ATTR'(W,_), P), V ==
W, !.
sys_attr_put(G, T, P, [T|P]) :- T = '$ATTR'(_,_), !,
   sys_attr_add(G, T).
sys_attr_put(_, _, P, P).

% sys_attr_conc(+Tree, +Integer, +Tree).
sys_attr_conc(V, _, N) :- var(V), !,
```

```
   V = N.
sys_attr_conc(node(L, _, R), D, N) :- var(R),
   E is D-2, E >= 0,
   sys_attr_conc(L, E, N), !.
sys_attr_conc(node(_, _, R), D, N) :-
   E is D-1, E >= 0,
   sys_attr_conc(R, E, N).

% sys_attr_list(+Tree, -List, +List)
sys_attr_list(V) --> {var(V)}, !.
sys_attr_list(node(L,T,R)) --> [T],
   sys_attr_list(L),
   sys_attr_list(R).

/*************************************************************/
/* Portray Hook                                              */
/*************************************************************/

% term_portraying(+Term, -Term)
:- multifile(term_portraying/2).
term_portraying(T, R) :- nonvar(T), T = '$ATTR'(R,_).
```