BinProlog 3.00
User Guide

Paul Tarau

Departement d'Informatique
Université de Moncton
Moncton, Canada, E1A 3E9,
tarau@info.umoncton.ca

September 10, 1994

1 Installation

```
EITHER:

   Edit ‘Makefile’. Change BINDIR and comment out ARCH_AND_OS,
   for example:

   BINDIR = /usr/local/bin
   ARCH_AND_OS = sparc.sunos

   then TYPE:

      make install

OR:

  Copy or symbolically link bin/wam.<ARCH>.<OS>  to ‘bp’
  somewhere in your path, than type ‘bp’.
```

Normally the appropriate ‘wam’ (a C-ified self contained executable) or (for some old architectures) typing ‘ru wam.bp’ (possibly a small ‘bp’ shell-script) are all you need to have BinProlog 3.00 running. For DOS/Windows 386/486 PCs you will have to type something like ‘go32 ru.386 wam.bp’ (or use a small bp.bat) file.

To start BinProlog use

```
$ bp <command-line options> <wam-bytecode-file>
```

or simply

```
$ bp
```

Sizes of the blackboard, heap, stack, trail and code areas can be passed as command line parameters etc, in Kbytes:

usage: [-h heap] [-s stack] [-t trail] [-c code] [-b bboard] [-a 2âtoms] [-d 2ĥashtable entries] [-i size of IO-buffer] [-q quietness] wamfile (default: wam.bp).

2 Obtaining BinProlog

The ORIGINAL DISTRIBUTION SITE for BinProlog[1] is:

```
ftp clement.info.umoncton.ca (139.103.16.2)
```

in the directory:

```
/pub/BinProlog.
```

The ftp server clement.info.umoncton.ca is a Sparcstation ELC with SUNOS 4.1.1). Please send comments and bug reports to `binprolog@info.umoncton.ca`.

3 Introduction to BinProlog

BinProlog has been developpped by Paul Tarau mostly at the University of Moncton, Canada, and is based on his BinWAM abstract machine, a specialization of the WAM for the efficient execution of binary logic programs.

BinProlog is a fast and small Prolog compiler, based on the transformation of Prolog to binary clauses. The compilation technique is similar to the Continuation Passing Style transformation used in some ML implementations.

BinProlog is a fairly complete Prolog implementation featuring both C-emulated execution and generation of standalone applications by compilation to C. Some other features are:

- C-compatible floating point operations,
- dynamic code,
- intuitionistic and linear implication,
- efficient high-order programming builtins,
- logical global variables,
- backtrackable destructive assignment,
- circular term unification,
- extended DCGs (now built in the engine as 'invisible grammars'),
- continuation manipulation primitives,
- a garbage-collected hashing based global dictionnary for constant-time sparse arrays and graphs,
- a C-interface etc.

The `bp` command starts the BinProlog toplevel.

The directory `wam2c` contains a project (*.pro) file and a makefile for generating a standalone application through compilation to C.

The directory `TCL` contains a bidirectional pipe based BinProlog to Tcl/Tk interface.

The directory `cparser` contains Koen De Bosschere's ISO Prolog parser (written in C) and BinProlog standalones based on the parser (cbp.*).

The directory `multi` contains Koen De Bosschere's a blackboard based distributed programming extension to BinProlog and BinProlog executables which integrate them (mbp.*).

3.1 Binarization

BinProlog is a small, program-transformation based compiler. Everything is converted to Continuation Passing Binary Clauses:

A rule like

```
a(X):-b(X),c(X,Y),d(Y).
```

becomes

```
a(X,Cont):-b(X,c(X,Y,d(Y,Cont))).
```

A fact like

```
a(13).
```

becomes

```
a(13,Cont):-true(Cont).
```

A predicate using metavariables like

```
p(X,Y):-X,Y.
```

becomes

```
p(X,Y,Cont);-call(X,call(Y,Cont))).
```

with true/1 and call/2 efficient builtins in BinProlog.

You can now try out in BinProlog 3.00 your own binary programs by using `::-` instead of `:-` so that the preprocessor will not touch them[2]. Otherwise, from the outside, BinProlog looks like any other Prolog.

Binarization allows a significant simplification of the Prolog engine, which can be seen as specialization of the WAM for the execution of Continuation Passing Binary Programs.

As a consequence, a very small emulator (55K on Solaris) that often fits completely in the cache of the processor, a more efficient new data representation and some low-level optimizations make BinProlog probably the fastest freely available C-emulated Prolog at this time (812 KLIPS on a Sparcstation 20-41).

This means 3-5 times faster than C-Prolog, 2-3 times faster than SWI-Prolog, 1.5-2 times faster than (X)SB-Prolog and close to C-emulated Sicstus 2.1.

[2]Take care if you use your own binary clauses to keep always the continuation as a last argument of the last embedded continuation 'predicate'. Look at the asm/0 tracer how BinProlog itself does this.

3.2 Machines supported

This distribution contains the Prolog source of the compiler and executable emulators for:

```
- Sparc - SunOS 4.x, Solaris 2.x;
- DEC Alpha - 64 bit
- 68k NeXT - Mach; SUN3 - SunOS 4.x
- MIPS - DEC;
- IBM R6000;
- 386-486-Pentium (MsDOS+Windows - with 32bits DOS-extender
  go32 ver. 1.10).
```

As the implementation makes no assumption about machine word size it is likely to compile even on very strange machines that have a C-compiler. BinProlog's integers are inherited from the native C-system. For example on DEC ALPHA machines BinProlog has $64 - 3 = 61$ bit integers. On 32-bit systems it has $32 - 2 = 30$ bit integers. Floating point is double (64 bits) and it is guaranteed that computations in Prolog will always give the same results as in the underlying C. As a matter of fact BinProlog does not really know that it has floats but how this happens is rather long to explain here.

4 Using BinProlog

BinProlog uses R.A. O'Keefe's public domain tokeniser and parser and write utilities (see the files read.pl, write.pl), DCGs and a transformer to binary programs. It compiles itself in about 10 seconds on a Sparcstation 10-40.

The system has very fast (heap-based) `copy_term/2`, `findall/3` and `findall/4` predicates, floating point, global logical variables, but still lacks full garbage collection.

A new term compression technique (joint work with Ulrich Neumerkel) reduces heap-consumption and adds some extra speed. Ulrich's iterative `copy_term/2` algorithm further accelerates BinProlog's 'copy-once' heap-based `findall/3` and `findall/4` so that findall-intensive programs may run 2-3 times faster in BinProlog than in other (even native code) implementations.

All data areas are now user configurable, and all except the heap are garbage collected. A garbage collector for the heap will be released soon.

Although other Prolog's assert and retract primitives are emulated in BinProlog, their functionality has been decomposed in separate simpler operations that give also improved efficiency.

For permanent information BinProlog has a new, garbage-collected data area the *blackboard* where terms can be stored and accessed efficiently with a a 2-key hashing function using something like

```
?-bb_def(likes,joe,[any(beer),good(vine),vegetarian(food)]).
```

and updated with something like

```
?-bb_set(likes,joe,nothing).
```

or

```
?-bb_rm(likes,joe).
```

To get its value:

```
?-bb_val(likes,joe,What).
```

BinProlog 3.00 has also backtrackable global variables, with 2-keyed names.
Try:

```
?- Person=joe, friend#Person:=:mary, bb.
```

and then

```
?- friend # joe:=:X.
```

The blackboard can be used either directly or through an assert-retract style interface.

A small exercise: if you want backtrackable behaviour of assert and retract you can modify extra.pl and use `A#B:=:X` style global variables in their definition, instead of `bb_def/3` etc.

The blackboard also gives constant-time sparse arrays and lemmas. For example try:

```
?- for(I,1,99),bb_def(table,I,f(I,I)),fail.
?- bb.
```

BinProlog 3.00 has Edinburgh behaviour and tries to be close to Sicstus and Quintus Prolog on the semantics of builtins without being too pedantic on what's not really important.

All the basic Prolog utilities are now supported (dynamic clauses, a metainterpreter with tracing facility, sort, setof, dynamic operators floating point operations and functions).

A fast deterministic append `det_append/3 (i,i,o)` has been added. Naive reverse using `det_append/3` makes more than 3 MegaLIPS on a Sparc 20-41 (a 3-times speed-up).

Almost all the builtins are now expanded inline resulting in improved heap consumption and performance.

A few programs (an automatic Tetris player, a knight-tour, an OR-parallel simulator, Fibonacci, Tak with lemmas, a small neural-net simulator backprop.pl) illustrate some of the new features. A few well-known benchmarks have been added to help compare BinProlog with other implementations.

BinProlog has supported from start 30 bit integer arithmetic. Now it has also *floating point operations* and functions like sin, cos, tan, log, exp, pow, etc. They can be used either through the is/2 interface[3]:

```
?- X is cos(3.14)+sin(0).
```

or in relational form

```
?- cos(1,X).
```

Note that you should use something like `Y=3+4, X is 1+expr(Y)` instead of `Y=3+4, X is 1+Y` which will not work in compiled code.

Floating point works has the same precision and semantics as the type double in C. Floating point operations are close in speed to emulated Sicstus. To try them out use the toy neural-network simulator `bp.pl`. This program uses also constant time arrays and is therefore unusually fast compared to its execution in other Prologs like Quintus or Sicstus.

4.1 The interactive toplevel shell

To see the command line options:

```
$ bp -help
```

To compile and load `<file>` or `<file>.pl` or `<file>.pro`:

```
?-[<file>].
```

BinProlog searches them in the directories `.`, `./progs` and `./myprogs`.

The following still works although the new C-ification technique (see directory wam2c) can now reate true on-file standalones.

To create a small runtime application from `<file>` or `<file>.pl`:

```
?-make_appl(<file>).
```

Note: the file must contain a clause

```
main(X):-...
```

That's where execution starts. Note that BinProlog's shell (the precompiled wam.bp) file is nothing but such an application which starts with a toplevel loop i.e. something like:

```
main(X):-toplevel(X).
```

[3] Is/2 now accepts execution of any predicate of arity $n+1$ as a function of arity n.

Deliver the appropriate wam ru or ru.exe file (the Prolog engine) with your application. The user must type:

```
$ ru newappl.bp
```

to start it.

As a new feature, you can override this behaviour by simple defining a predicate main/0 which then becomes the new starting point.

You can also generate (on UNIX systems) stand-alone executables that dynamically start the emulator (thanks to Peter Reintjes for suggesting this). You can do something like:

```
?- make_executable_unix_appl('./ru','progs/hello.pl','hello').
```

Then you can run it directly from the unix prompt:

```
$ hello
```

The code of this primitive is at the end of the file `co.pl`.

Again, we recommend using the C-ification technique which can already speed up most applications and in the future will generate very fast code competitive with native code compilers.

You can bootstrap the compiler, after modifying the `*.pl` files by:

```
?- boot.
```

or

```
?-make(ProjectFile).
```

or

```
?-cmake(ProjectFile).
```

if you intend to generate C-code.

This allows to integrate your preferences and extensions written in Prolog to the BinProlog kernel.

Make sure you keep a copy the original wam.bp *in a safe place, in case things go wrong when you type* `?-boot.`

4.2 Some limitations/features of BinProlog

We passed Occam's razor on a few "features" of Prolog that are obsolete in modern programming environments and on some others that are, in our opinion, bad software practice.

Only one file at a time can be compiled in the interactive environment:

```
?-[myfile].
```

or

```
?- compile(myfile).
```

Now BinProlog supports an include directive:

```
:-[include1].
:-[include2].
....
```

This suggest to make a project `(*.pro)` file using a set of include directives each refereing to a `*.pl` file. When compiled to a file (by using the `?-make(MyProject)` command) a make-like memoing facility will avoid useless recompilation of the included `(*.pl)` files by creation of precompiled `(*.pl_wam)` files. For large projects this is the recommended technique. Creation of C-ified standalone files is also possible (see the `wam2c` directory).

Programs that work well can be added to the BinProlog kernel. This avoids repeated recompilation of working code and predicates in the kernel are protected against name clashing.

New programs can be loaded in the interactive environment. When they work well, they migrate to the kernel. You can prepare a good Makefile to do the job of `ensure_loaded` of other Prolog's. When everything is OK you can deliver it as a run-time-only application.

Programs are searched with suffixes `""`, `".pl"`, `".pro"` in the directories `.`, `./progs` and `./myprogs`.

There's no limit on the number of files you can compile to disk:

```
?- compile(wam,[file1,file2,...],'application.bp').
```

Now BinProlog does implement consult/1, reconsult/1 and listing/0 for interpreted code but use of compile/1 is highly recommended instead. See the file extra.pl for the implementation. Faster than asserted code is so called assumed code (see the next sections) i.e. intuitionistic and linear implication.

Here are some other limitations/features:

- Clauses of a predicate must be grouped.

- Clauses having undefined predicates occurring right after the head are rejected.
- ARITY is limited to 255.
- The heap garbage collector is not yet implemented, but we plan to add it soon. However, the blackboard, dynamic code space, the string space and the hashing table ARE garbage collected before loading a new program.

Mode is interactive by default (for compatibility with other Prologs) but if you use a modern, windows based environment you may want to switch it off with:

```
?- interactive(no).
```

or turn it on again with

```
?- interactive(yes).
```

4.3 Other BinProlog goodies and new predicates

A few BinProlog specific predicates are available:

- restart/0 - cleans every data area
- cwrite/1 - fast but restricted write
- symcat/3 (i,i,o) returns a new symbol of the form <first>_<second>.
- gensym/2 (i,o) forms a new name of the form <name>_<counter>.
- sread/2 (i,o) reads from a name a (ground) term, swrite(i,o) writes a term to a name.
- term termcat/3 (i,i,o) adds its second argument as last argument of its first argument and returns the new term
- term_chars/2 converts between a ground term and its string representation
- not/1 is a form of sound negation
- for/3 as for instance in ?-for(I,1,5),write(I),nl,fail generates a failure driven loop

It is a good idea to take a look at BinProlog's *.pl for other builtin-or-library predicates before implementing them yourself. The file write.pl contain various output predicates like

- write/1
- writeq/1

- portray_clause/1
- print/1
- display/1
- ttyprint/1
- ttynl/1

You can extend BinProlog by adding new predicates to the file extra.pl and then use the predicate `boot/0` defined in the file co.pl.

4.4 Efficient findall based meta-programming

BinProlog's `findall/3` is so efficient that you can afford (with some care) to use it instead of explicit (and more painful) first-order programs as in:

```
% maplist with findall
maplist(Closure,Is,Os):-
  Closure=..L1,
  det_append(L1,[I,O],L2),
  P=..L2,
  findall(O,map1(P,I,Is),Os).

map1(P,I,Is):-member(I,Is),P.
```

This can be used as follows:

```
?- maplist(+(1),[10,20,30],T).
=> T=[11,21,31]
```

Note that constructing `Closure` only once (although this may not be in any Prolog text-book!) is more efficient than doing it at each step.

The predicate `gc_call(Goal)` defined in the file `lib.pl` executes Goal in minimal space. It is explained in the Craft of Prolog by R.A. O'Keefe, MIT Press. Do not hesitate to use it. BinProlog offers a very fast, heap-oriented findall, so you can afford to use `gc_call`. In good hands, it is probably faster than using assert/retract or the usual mark-and-sweep garbage collector of other implementations.

4.5 Builtins

The following (user-level) builtins are supported, with semantics (intended to be) close to SICSTUS and QUINTUS Prolog.

```
fail/0
nl/0
var/1
nonvar/1
integer/1
atomic/1
+/3   % arithmetic offers also the usual is/2 interface
-/3
* /3
// /3
mod/3
<< /3
>> /3
/\ /3
\/ /3
# /3    % bitwise XOR
\ /3     % bitwise NOT
random/1 % returns an integer, not a float
get0/1
put/1
< /2
> /2
=< /2
>= /2
=:= /2
=\= /2
compare/3
seeing/1
seen/0
telling/1
told/0
copy_term/2
functor/3
arg/3
name/2
abort/0
is_compiled/1 % checks if a predicate is compiled
```

Other useful predicates are defined in the file lib.pl and work as expected.

```
true/0,
=/2,
.. -> .. ; ..
```

```
;/1
call/1
\+/1
repeat/0
findall/3
findall/4
for/3
numbervars/4
see/1
tell/1
system/1
statistics/0
statistics/2
atom/1
compound/1
=../2
length/2
tab/1
get/1
== /2
\== /2
A @< B
A @> B
A @=< B
A @>= B
compile/1 - see co.pl for the compiler
read/1  - see the file: read.pl
write/1 - see the file: write.pl
halt/0,
halt(ReturnCode),
listing/0, % only for interpreted code
listing(Pred,Arity),
assert/1,
asserta/1,
assertz/1,
retract/1,
retractall/1
erase/1,
dynamic/1,
instance/2,
clause/2,
clause/3,
consult/1,   % usable for debugging or dynamic code
reconsult/1, % they override compiled definitions !!!
```

```
setof/3,
bagof/3,
sort/3,
keysort/3,
setarg/3, % backtrackable update of arg I of term T with new term X

is_builtin/1 % lists the system-level builtins (written in C):
current_predicate/1 % lists/checks existence of a  predicate/arity
predicate_property/2 % lists/checks if a property (arg 2) is associated with a head (ar
```

Operators are defined and retrieved with

```
:-op/3, current_op/3.
```

Including files are supported. For `compile/1` use

```
:-compile(file).
```

or

```
:-[file]. % this calls the compiler too !!!
```

For consulting included interpreted code within an embedding reconsult/1 use:

```
:-consult(file).
```

in your (unique) top-level file. This overcomes the limitation of having only one top-level file.

5 Editing, debug/1 and trace/1

To edit a file (with emacs) and then compile it use:

```
?- e(<file>).
```

The debugger/tracer uses R.A. O'Keefe's public domain meta-interpreter. You can modify it in the file "extra.pl".

DCG-expansion is supported by the public domain file dcg.pl.

To debug a file type:

```
?- reconsult(FileName).
```

and then

```
?- trace(Goal).
```

For interactivity, both the toplevel and the debugger depend on

```
?-interactive(yes).
```

or

```
?-interactive(no).
```

My personal preference is using `interactive(no)` within a scrollable window. However, as traditionally all Prologs hassle the user after each answer BinProlog 3.00 will do the same by default.

You can use the debugger to debug compiled code with the following trick if you always debug bottom-up i.e. if you ensure that tools work before to using them. For example, on top of the compiled file `allperms.pl` you can temporarily interpret perm/2, insert/3 etc. to be able to trace them while keeping uninteresting predicates compiled. By the way, this allows, to trace/debug parts of the kernel itself in particular.

The following terminal session shows an example:

```
?- [allperms].
compiling(to(mem),progs/allperms.pl,...)
compile_time(230)
?- consult(user).
% using compile/1 is MUCH faster
reconsulting(user)
g0(N):-nats(1,N,Ns),perm(Ns,_),fail.
g0(_).
WARNING: redefining compiled predicate(g0/1)
perm([],[]).
perm([X|Xs],Zs):-
perm(Xs,Ys),
insert(X,Ys,Zs).

insert(X,Ys,[X|Ys]).
insert(X,[Y|Ys],[Y|Zs]):-
insert(X,Ys,Zs).
WARNING: redefining compiled predicate(perm/2)
WARNING: redefining compiled predicate(insert/3)
reconsulted(user,time = 20)
yes
?- interactive(no).
yes
?- trace(g0(3)).
Call: g0(3)
 Call: nats(1,3,_645)
  compiled(nats(1,3,_645))
 Exit: nats(1,3,[1,2,3])
```

```
Call: perm([1,2,3],_648)
 Call: perm([2,3],_1095)
  Call: perm([3],_1362)
   Call: perm([],_1629)
   Exit: perm([],[])
   Call: insert(3,[],_1362)
   Exit: insert(3,[],[3])
  Exit: perm([3],[3])
  Call: insert(2,[3],_1095)
  Exit: insert(2,[3],[2,3])
 Exit: perm([2,3],[2,3])
 Call: insert(1,[2,3],_648)
 Exit: insert(1,[2,3],[1,2,3])
Exit: perm([1,2,3],[1,2,3])
Call: fail
 compiled(fail)
Fail: fail
Redo: perm([1,2,3],[1,2,3])
 Redo: insert(1,[2,3],[1,2,3])
  Call: insert(1,[3],_2683)
  Exit: insert(1,[3],[1,3])
 Exit: insert(1,[2,3],[2,1,3])
Exit: perm([1,2,3],[2,1,3])
Call: fail
 compiled(fail)
Fail: fail
Redo: perm([1,2,3],[2,1,3])
 Redo: insert(1,[2,3],[2,1,3])
  Redo: insert(1,[3],[1,3])
   Call: insert(1,[],_2945)
   Exit: insert(1,[],[1])
  Exit: insert(1,[3],[3,1])
 Exit: insert(1,[2,3],[2,3,1])
Exit: perm([1,2,3],[2,3,1])
Call: fail
 compiled(fail)
Fail: fail
Redo: perm([1,2,3],[2,3,1])
 Redo: insert(1,[2,3],[2,3,1])
  Redo: insert(1,[3],[3,1])
   Redo: insert(1,[],[1])
   Fail: insert(1,[],_2945)
  Fail: insert(1,[3],_2683)
 Fail: insert(1,[2,3],_648)
```

```
 Redo: perm([2,3],[2,3])
  Redo: insert(2,[3],[2,3])
   Call: insert(2,[],_2421)
   Exit: insert(2,[],[2])
  Exit: insert(2,[3],[3,2])
 Exit: perm([2,3],[3,2])
 Call: insert(1,[3,2],_648)
 Exit: insert(1,[3,2],[1,3,2])
Exit: perm([1,2,3],[1,3,2])
Call: fail
 compiled(fail)
Fail: fail
Redo: perm([1,2,3],[1,3,2])
 Redo: insert(1,[3,2],[1,3,2])
  Call: insert(1,[2],_2945)
  Exit: insert(1,[2],[1,2])
 Exit: insert(1,[3,2],[3,1,2])
Exit: perm([1,2,3],[3,1,2])
Call: fail
 compiled(fail)
Fail: fail
Redo: perm([1,2,3],[3,1,2])
 Redo: insert(1,[3,2],[3,1,2])
  Redo: insert(1,[2],[1,2])
   Call: insert(1,[],_3207)
   Exit: insert(1,[],[1])
  Exit: insert(1,[2],[2,1])
 Exit: insert(1,[3,2],[3,2,1])
Exit: perm([1,2,3],[3,2,1])
Call: fail
 compiled(fail)
Fail: fail
Redo: perm([1,2,3],[3,2,1])
 Redo: insert(1,[3,2],[3,2,1])
  Redo: insert(1,[2],[2,1])
   Redo: insert(1,[],[1])
   Fail: insert(1,[],_3207)
  Fail: insert(1,[2],_2945)
 Fail: insert(1,[3,2],_648)
 Redo: perm([2,3],[3,2])
  Redo: insert(2,[3],[3,2])
   Redo: insert(2,[],[2])
   Fail: insert(2,[],_2421)
  Fail: insert(2,[3],_1095)
```

```
   Redo: perm([3],[3])
    Redo: insert(3,[],[3])
    Fail: insert(3,[],_1362)
    Redo: perm([],[])
    Fail: perm([],_1629)
   Fail: perm([3],_1362)
  Fail: perm([2,3],_1095)
 Fail: perm([1,2,3],_648)
 Redo: nats(1,3,[1,2,3])
 Fail: nats(1,3,_645)
Exit: g0(3)
```

Although these are very basic debugging facilities you can enhance them at your will and with some discipline in programming they may be all you really need. Anyway, future of debugging is definitely not by tracing. One thing is to have stronger static checking. In dynamic debugging the way go is to have a database of trace-events and then query it with high level tools. We plan to add some non-tracing database-oriented debugging facilities in the future.

You can generate a kind of intermediate WAM-assembler by

```
  ?- compile(asm,[file1,file2,...],'huge_file.asm').
```

A convenient way to see interactively the sequence of program transformations BinProlog is based on is:

```
?- asm.
a-->b,c,d.
^D

DEFINITE:
a(A,B) :-
        b(A,C),
        c(C,D),
        d(D,B).

BINARY:
a(A,B,C) :-
        b(A,D,c(D,E,d(E,B,C))).

WAM-ASSEMBLER:
clause_? a,3
firstarg_? _/0,6
put_structure d/3,var(4-4/11,1/2)
write_variable put,var(5-5/10,1/2)
```

```
write_value put,var(2-2/6,2/2)
write_value put,var(3-3/7,2/2)
put_structure c/3,var(3-8/14,1/2)
write_variable put,var(2-9/13,1/2)
write_value put,var(5-5/10,2/2)
write_value put,var(4-4/11,2/2)
execute_? b,3
```

6 The Blackboard

A new interface has been added to separate backtrackable and surviving uses of blackboard objects so this primitive and the def/3, set/3, rm/2 of previous version although still available should be replaced either with:

- global logical variables
- garbage-collectible permanent objects.

6.1 Global logical variables

Syntax: `A#B:=:X`, or `lval(A,B,X)`.

where X is any term on the heap. It has simply a global name `A#B` i.e. an entry in the hashing table with keys A and B. The address in the table (C-pointers are the same as logical variables in BinProlog) is trailed such that on backtracking it will be unbound (i.e. point to itself). Unification with `A#B:=:Y` is possible at any point in the program which knows the 'name' `A#B`.

Although a global logical variable cannot be changed it can be further instantiated as it happens to ordinary Prolog terms. Backtracking ensures they vanish so that no unsafe reference can be made to them.

The program lq8.pl is an efficient 8-queens program using global logical variables to simulate the chess-board.

6.2 Garbage-collectible permanent objects.

On the other hand, if `bb_def/3` or `bb_set/3` is used to name objects on the blackboard, they "survive" backtracking and can afterwards be retrieved as logical variables using `bb_val/3`.

```
bb_def/3    (i,i,i) defines a value
bb_set/3    (i,i,i) updates a value
bb_rm/2     (i,i) removes a value
bb_val/3    (i,i,o) retrieves the value
```

They are quite close to the `recorda/recordz` family of other Prologs although they offer better 2-key indexing, are simpler and can be used to do much more things efficiently.

You can look to the program `progs/knight.pl` on how to use them to implement in a convenient and efficient way programs with backtrackable global arrays.

They can be used to save information that survives backtracking in a way similar to other Prolog's `assert` and `retract` and are safe with respect to garbage collection of the blackboard.

The predicate `bb_list/1` gives the content of the blackboard as a heap object (list), while `bb/0` simply prints it out.

These predicates offer generally faster and more flexible management of dynamic state information than other Prolog's dynamic databases.

6.3 Assert and retract

For compatibility reasons BinProlog has them, implemented on top of the more efficient blackboard manipulation builtins.

This is an approximation of other Prologs assert and retract predicates. It tries to be close to Sicstus and Quintus with their semantics. For efficiency and programming style reasons I strongly suggest not to use them too much.

If you want maximal efficiency use `bb_def/3, bb_set/3, bb_val/3, bb_rm` They give you access to a very fast hashing table `<key,key>--> value`, the same that BinProlog uses internally for indexing by predicate and first argument. They are close to other Prolog's 'record' family, except that they do even less.

To use dynamic predicates it is a good idea to declare them with `dynamic/1` although asserts will now be accepted even without such a declaration. To define dynamic code in a file you compile, dynamic declarations are mandatory.

To activate an asserted predicate it is a good idea to alway call it with

```
?-metacall(Goal).
```

instead of

```
?- Goal.
```

However, this is not a strong requirement anymore, as an important number of users were unhappy with this restriction.

The dynamic predicates are:

```
assert/1
asserta/1
assertz/1

retract/1
clause/2
metacall/1
abolish/2
```

You can easily add others or improve them by looking to the sources in the file `extra.pl`.

6.4 Assumed code, intutionistic and linear implication

Intuitionistic assumei/1 adds temporarily a clause usable in later proofs. Such a clause can be used an indefinite number of times, mostly like asserted clauses, except that it vanishes on backtracking. Its scoped version `Clause=>Goal` or `[File]=>Goal` makes `Clause` or the set of clauses found in `File` available only during the proof of Goal. Both vanish on backracking. Clauses are usable an indefinite number of times in the proof, i.e. for instance `?-assumei(a(13)),a(X),a(Y)` will succed.

Linear assumel/1 adds temporarily a clause usable *at most once* in later proofs. This assumption also vanishes on backtracking. Its scoped version `Clause<>-Goal` or `[File]<>-Goal` makes `Clause` or the set of clauses found in `File` available only during the proof of Goal. Both vanish on backracking. Each clause is usable at most once in the proof, i.e. for instance `?-assumel(a(13)),a(X),a(Y)` will fail.

You can freely mix linear and intutionistic clauses and implications for the same predicate. Try out something like

```
?-a(10)<>-a(11)=>a(12)<>-a(13)=>(a(X),a(X)).
X=11;
X=13;
no
```

This shows that `a(10)` and `a(12)` are *consumed* after their first use while `a(11)` and `a(13)` are reusable indefinitely.

See the relatively straightforward implementation of these predicates in the file extra.pl.

6.5 Overriding

Assumed predicate will override similarly named dynamic predicates which in turn will override compiled ones. Note that overriding is done at *predicate*, not *clause* level. Note also that multifile compiled clauses are still forbidden. However, multifile assumed and dynamic code is now accepted.

6.6 The blackboard as an alternative to assert and retract

Designed in the early stages of Prolog, assert and retract have been overloaded with different and often conflicting requirements. They try to be both the way to implement permanent data structures for global state information, self-modifying code and tools for Prolog program management. This created not only well-known semantical but also expressivity and efficiency problems.

This unnecessary overloading is probably due to some of their intended uses in interpreted Prologs like implementing the `consult/1` and `reconsult/1` code-management predicates that can be replaced today by general purpose makefiles. As a consequence, their ability to express sophisticated data structures is very limited due mostly to unwanted copying operations (from heap to dynamic code area and back) and due to their non-backtrackable behaviour.

For example, to ensure indefinite number of uses of an asserted clause most Prologs either compile it on the fly or do some form of copying (usually twice: when asserting and when calling or retracting). This is not only a waste of resources but also forbids use of asserted clauses for dynamically evolving global objects containing logical variables, one of the most interesting and efficient data structure tricks in Prolog. Worst, variables representing global data structures have to be passed around as extra arguments, just to bore programmers and make them dream about inheritance and objects oriented languages. This also also creates error prone maintenance problems. Just think about adding a new seventh argument to a 10-parameter Prolog predicate having 10 clauses and being called 10 times.

Those are the main reasons for the re-design of these operations using BinProlog's blackboard.

Objects on the blackboard have indefinite extent. However, their names are "rigid designators" that can change their reference. Data objects do not disappear just because they have no names anymore. This is the main difference with variables in languages with destructive assignment. If the garbage collector "can prove" that an object or some part of it is will never be used again, the space will be recuperated automatically. Although objects cannot be "changed" they can be further instantiated as it happens to ordinary Prolog terms.

Efficient access to objects on the blackboard or part of them is based on an efficient 2-key hashing table, internal to BinProlog's run-time system.

6.7 The low-level blackboard operations

Six low-level[4] primitives `def/3, set/3, val/3, rm/2, copy_term/2, save_term/2` are used in BinProlog 3.00 to fully replace assert and retract while keeping distinct their overloaded *naming* and *copying* function.

6.7.1 Naming primitives

The first four are simply an interface to BinProlog's unique global hashing table working as an

```
<Atomic-key1,Atomic-key2>-->HeapOrBlackboardObject.
```

function.

- def/3: (i,i,i) defines a value (usable only once) or fails;
- set/3: (i,i,i) updates a value that has been defined or warns and fails;

[4] We suggest avoiding these operations because they are not garbage collection-safe in BinProlog 3.00.

- val/3: (i,i,o) retrieves the value or fails if absent;
- rm/3: (i,i) deletes the value or fails if absent.

We call them *naming primitives* as they are used to name arbitrary heap or blackboard objects for definition, update and access function. Naming objects on the heap is generally unsafe and should be hidden from the user except for constants which are actually copied in the hashing table instead of being pointed to. This frequent case has been given to end users as an efficient 2-keys, 1-value dictionary since the first release of BinProlog.

If blackboard operations are backtrackable heap objects can be safely named. This suggested the implementation of global logical variables in BinProlog 3.00.

Otherwise, in BinProlog's gc-safe transfers to and from the blackboard are doing with a very efficient copying algorithm.

Not that if the garbage collector is not used naming objects on the blackboard is safe as they are always at a lower address than the base of the heap. As a consequence they survive backtracking while keeping their behaviour as close as possible to the behaviour of usual heap objects. This usage is compatible with BinProlog 1.71 but not recommended in BinProlog 3.00.

6.7.2 Copying primitives

`Copy_term/2` is Prolog's usual primitive extended to copy objects from the heap and also from blackboard to the current top of the heap. We refer to [10] for the implementation and memory management aspects of these primitives.

`Save_term/2` copies an object possibly distributed over the heap and the blackboard to a new blackboard object. It also takes care not to copy parts of the object already on the blackboard.

Remark that having known modes and argument types helps in the case of partial evaluation or type inference systems. Separating Prolog's asserts two main functions (naming+copying) in lower level operations allows program transformers to go *inside* more complex blackboard operations and possibly use the typing and mode information that comes from def/3, set/3 and val/3 to infer it for other predicates.

6.8 An useful Prolog extension: bestof/3

`Bestof/3` is missing in all current Prolog implementations we know of. BinProlog's `bestof/3` works like `findall/3`, but instead of accumulating alternative solutions, it selects successively the *best* one with respect to an arbitrary *total order* relation. If the test succeeds the new answer replaces the previous one. At the end, either the query has no answers, case in which `bestof` fails, or an answer is found such that it is better than every other answer with respect to the total order. The proposed syntax is

```
?-bestof(X,TotalOrder,Goal)
```

At the end, X is instantiated to the best answer. For example, the maximum of a list of integers can be defined simply as:

```
max(Xs,X):-bestof(X,>,member(X,Xs)).
```

The following is an efficient implementation implementation using the blackboard.

```
% true if X is an answer of Generator such that
% X Rel Y for every other answer Y of Generator
bestof(X,Closure,Generator):-
  copy_term(X,Y),
  Closure=..L1,
  det_append(L1,[X,Y],L2),
  Test=..L2,
  bestof0(X,Y,Generator,Test).

bestof0(X,Y,Generator,Test):-
  inc_level(bestof,Level),
  Generator,
  update_bestof(Level,X,Y,Test),
  fail.
bestof0(X,_,_,_):-
  dec_level(bestof,Level),
  val(bestof,Level,X),
  rm(bestof,Level).

% uses Rel to compare New with so far the best answer
update_bestof(Level,New,Old,Test):-
  val(bestof,Level,Old),!,
  Test,
  bb_set(bestof,Level,New).
update_bestof(Level,New,_,_):-
  bb_let(bestof,Level,New).

% ensure correct implementation of embedded calls to bestof/3
inc_level(Obj,X1):-val(Obj,Obj,X),!,X1 is X+1,bb_set(Obj,Obj,X1).
inc_level(Obj,1):-bb_def(Obj,Obj,1).

dec_level(Obj,X):-val(Obj,Obj,X),X>0,X1 is X-1,bb_set(Obj,Obj,X1).
```

Note that precomputation of Test in bestof/3 before calling the workhorse bestof0/4 is much more efficient than using some form of apply meta-predicate inside bestof0/4.

6.9 Blackboard based abstract data types

We will describe some simple utilisations of the blackboard to implement efficiently some basic abstract data types. They all use the `saved/2` predicate instead of `save_term/2`. `Saved/2` does basically the same work but also makes a call to the blackboard garbage collector if necessary. The reader can find the code in the file `lib.pl` of the BinProlog distribution.

6.9.1 Blackboard based failure surviving stacks

A very useful data structure that can be implemented with the blackboard is a stack that survives failure but still allows the programmer to use some of the nice properties of logical variables.

The main operations are `push/3` that saves a term to the blackboard and pushes it to a named stack, `pop/3` that removes the top element from a named stack and unifies it with its third argument and `stack/3` that simply gives access to the list on the blackboard representing the stack. The only operation that uses copying is `push/3`, although if the term that is pushed to the stack has already some subterms on the blackboard, such subterms will not be copied again.

The implementation is straightforward:

```
push(Type,S,X):-
  default_val(Type,S,Xs,[]),
  saved([X|Xs],T),
  set(Type,S,T).

pop(Type,S,X):-
  default_val(Type,S,V,[]),
  V=[X|Xs],
  set(Type,S,Xs).

stack(Type,S,Xs):-val(Type,S,Xs).

default_val(X,Y,V,_):-val(X,Y,V),!.
default_val(X,Y,D,D):-def(X,Y,D).
```

6.9.2 Constant time vectors

Defining a vector is done initializing it to a given Zero element. The `vector_set/3` update operation uses `saved/2`, therefore the old content of vectors is also subject to garbage collection.

```
vector_def(Name,Dim,Zero):- Max is Dim-1,
  saved(Zero,BBVal),
  for(I,0,Max), % generator for I from 0 to Max
```

```
    let(Name,I,BBVal), % def/3 or set/3 if needed
  fail.
vector_def(_,_,_).

vector_set(Name,I,Val):-saved(Val,BBVal),set(Name,I,BBVal).

vector_val(Name,I,Val):-val(Name,I,Val).
```

Building multi-dimensional arrays on these vectors is straightforward, by defining an index-to-address translation function.

The special case of a high-performance 2-dimension (possibly sparse) global array can be handled conveniently by using def/3, set/3, val/3 and saved/2 as in:

```
global_array_set(I,J,Val):-saved(Val,S),set(I,J,S).
```

6.10 Blackboard based problem solving

6.10.1 A complete knight tour

The following is a blackboard based complete knight-tour, adapted from Evan Tick's well known benchmark program.

```
% recommended use: ?-go(5).
go(N):-
time(_),
  init(N,M),              % prepare the chess board
  knight(M,1,1),!,        % finds the first complete tour
time(T),
  write(time=T),nl,statistics,show(N). % shows the answer

% fills the blackboard with free logical variables
% representing empty cell on the chess board
init(N,_):-
  for(I,1,N),      % generates I from 1 to N nondeterministically
    for(J,1,N),    % for/3 is the same as range/3 in the Art of Prolog
      bb_def(I,J,_NewVar),  % puts a free slot in the hashing table
  fail.
init(N,M):-
  M is N*N.                  % returns the number of cells

% tries to make a complete knight tour
knight(0,_,_) :- !.
knight(K,A,B) :-
  K1 is K-1,
  val(A,B,K),  % here we mark (A,B) as the K-th cell of the tour
```

```
  move(Dx,Dy), % we try a next move nondeterministically
  step(K1,A,B,Dx,Dy).

% makes a step and then tries more
step(K1,A,B,Dx,Dy):-
    C is A + Dx,
    D is B + Dy,
    knight(K1,C,D).

% shows the final tour
show(N):-
  for(I,1,N),
    nl,
    for(J,1,N),
      val(I,J,V),
      write(' '),X is 1-V // 10, tab(X),write(V),
  fail.
show(_):-nl.

% possible moves of the knight
move( 2, 1). move( 2,-1). move(-2, 1). move(-2,-1).
move( 1, 2). move(-1, 2). move( 1,-2). move(-1,-2).
```

Constant time access in this kind of problems to cell(I,J) is essential for efficiency as it is the most frequent operation. While the blackboard based version takes 39s in BinProlog for a 5x5 squares chess board, an equivalent program representing the board with a list of lists takes 147s in BinProlog, 167s in emulated Sicstus 2.1 and 68 seconds in native Sicstus 2.1. Results are expected to improve somewhat with binary trees or functor-arg representation of the board but they will still remain worse than with the blackboard based sparse array, due to their relatively high log(N) or constant factor. Moreover, representing large size (possibly updatable!) arrays with other techniques is prohibitively expensive and can get very complicated due to arity limits or tree balancing as it can see for example in the Quintus library.

6.10.2 A lemma based TAK

The following tak/4 program uses lemmas to avoid heap explosion in the case of of a particularly AND intensive program with 4 recursive calls, a problem particularly severe in the case of the continuation passing binarization that BinProlog uses to simplify the WAM. To encode the 2 first arguments in a unique integer some bit-shifting is needed as it can be seen in `tak_encode/3`. To avoid such problems, hashing on arbitrary terms like Quintus Prolog's `term_hash/2` looks a very useful addition to BinProlog.

```
tak(X,Y,Z,A) :- X =< Y, !, Z = A.
```

```
tak(X,Y,Z,A) :-
        X1 is X - 1,
        Y1 is Y - 1,
        Z1 is Z - 1,
        ltak(X1,Y,Z,A1),
        ltak(Y1,Z,X,A2),
        ltak(Z1,X,Y,A3),
        ltak(A1,A2,A3,A).

ltak(X,Y,Z,A):-
        tak_encode(X,Y,XY),
        tak_lemma(XY,Z,tak(X,Y,Z,A),A).

tak_encode(Y,Z,Key):-Key is Y<<16 \/ Z.
tak_decode(Key,Y,Z):-Y is Key>>16, Z is Key <<17>>17 .

%optimized lemma <P,I,G> --> O (instantiated executing G)
tak_lemma(P,I,_,O):-val(P,I,X),!,X=O.
tak_lemma(P,I,G,O):-G,!,def(P,I,O).

go:-    statistics(runtime,_),
        tak(24,16,8,X),
        statistics(runtime,[_,T]),statistics,
        write([time=T,tak=X]), nl.
```

We hope that we showed the practicality of BinProlog's blackboard for basic work on data structures and problem solving.

BinProlog's blackboard primitives make a clear separation between the *copying* and the *naming* intent overloaded in Prolog's assert and retract.

Our blackboard primitives give most of the time simpler and more efficient solutions to current programming problems than assert and retract while being closer to a logical semantics and more cooperative to partial evaluation.

6.11 A Linda style interface to the blackboard

The following predicates show how to do some Linda-style operations on top of the blackboard primitives.

The current focus of the operations is managed by `object/1` and `message/1`. `Out/?` puts a tuple Mes on the blackboard, `rd/?` reads it and `in/?` removes it. `Eval/?` executes the Goal part of the clause `(Answer:-Goal)` focussed by `object/1` and `message/1`. Then it puts the result `Answer` back to the blackboard. As `Answer` itself can be of the form `(A:-G)`, a limited number of cascaded `evals` are possible.

```
% out/1, rd/1, in/1
```

```
out(Mes):-object(Obj),message(Id),out(Obj,Id,Mes).
rd(Mes):-object(Obj),message(Id),rd(Obj,Id,Mes).
in(Mes):- object(Obj),message(Id),in(Obj,Id,Mes).

% out/2, rd/2, in/2
out(Id,Mes):-object(Obj),out(Obj,Id,Mes).
rd(Id,Mes):-object(Obj),rd(Obj,Id,Mes).
in(Id,Mes):-object(Obj),in(Obj,Id,Mes).

% out/3, rd/3, in/3
out(Obj,Id,_):-val(Obj,Id,_),!,fail.
out(Obj,Id,Mes):-saved(Mes,Sent),let(Obj,Id,Sent).
rd(Obj,Id,Mes):-val(Obj,Id,Mes).
in(Obj,Id,Mes):-val(Obj,Id,Mes),rm(Obj,Id).

% eval/0, eval/1, eval/2
eval:-object(Obj),message(Id),eval(Obj,Id).
eval(Id):-object(O),eval(O,Id).
eval(Obj,Id):-val(Obj,Id,(Answer:-Goal)),Goal,!,
  saved(Answer,NewAnswer),
  set(Obj,Id,NewAnswer).

% tools
object(New):-var(New),!,val('$object','$object',New).
object(New):-atomic(New),let('$object','$object',New).

message(New):-var(New),!,object(O),val(O,'$message',New).
message(New):-atomic(New),object(O),let(O,'$message',New).
```

7 Continuations as first order objects

7.1 Continuation manipulation vs. intuitionistic/linear implication

Using intuitionistic implication (actullay it should be `<>-` but let's forget this for a moment) we can write in BinProlog:

```
    insert(X, Xs, Ys) :-
        paste(X) => ins(Xs, Ys).

    ins(Ys, [X|Ys]) :- paste(X).
    ins([Y|Ys], [Y|Zs]):- ins(Ys, Zs).
```

used to nondeterministically insert an element in a list, the unit clause `paste(X)` is available only within the scope of the derivation for `ins`. This gives:

```
?- insert(a,[1,2,3],X).
X=[a,1,2,3];

X=[1,a,2,3];

X=[1,2,a,3];

X=[1,2,3,a]
```

With respect to the corresponding Prolog program we are working with a simpler formulation in which the element to be inserted does not have to percolate as dead weight throughout each step of the computation, only to be used in the very last step. We instead clearly isolate it in a global-value manner, within a unit clause which will only be consulted when needed, and which will disappear afterwards.

Now, let us imagine we are given the ability to write part of a proof state context, i.e., to indicate in a rule's left-hand side not only the predicate which should match a goal atom to be replaced by the rule's body, but also which other goal atom(s) should surround the targeted one in order for the rule to be applicable.

Given this, we could write, using BinProlog's `@@` (which gives in its second argument the current continuation) a program for `insert/3` which strikingly resembles the intuitionistic implication based program given above:

```
insert(X,Xs,Ys):-ins(Xs,Ys),paste(X).

ins(Ys,[X|Ys]) @@ paste(X).
ins([Y|Ys],[Y|Zs]):-ins(Ys,Zs).
```

Note that the element to be inserted is not passed to the recursive clause of the predicate **ins/2** (which becomes therefore simpler), while the unit clause of the predicate **ins/2** will communicate directly with **insert/3** which will directly 'paste' the appropriate argument in the continuation.

In this formulation, the element to be inserted is first given as right-hand side context of the simpler predicate `ins/2`, and this predicate's first clause consults the context `paste(X)` only when it is time to place it within the output list, i.e. when the fact `ins(Ys,[X|Ys]),paste(X)` is reached.

Thus for this example, we can also obtain the expressive power of intuitionistic/linear implication by this kind of (much more efficient) direct manipulation of BinProlog's first order continuations.

8 Direct binary clause programming and full-blown continuations

BinProlog 3.00 supports direct manipulation of binary clauses denoted

```
Head ::- Body.
```

They give full power to the knowledgeable programmer on the future of the computation. Note that such a facility is not available in conventional WAM-based systems where continuations are not first-order objects.

We can use them to write programs like:

```
member_cont(X,Cont)::-
  strip_cont(Cont,X,NewCont,true(NewCont)).
member_cont(X,Cont)::-
  strip_cont(Cont,_,NewCont,member_cont(X,NewCont)).

test(X):-member_cont(X),a,b,c.
```

A query like

```
?-test(X).
```

will return `X=a; X=b; X=c; X=`whatever follows from the calling point of `test(X)`.

```
catch(Goal,Name,Cont)::-
   lval(catch_throw,Name,Cont,call(Goal,Cont)).

throw(Name,_)::-
   lval(catch_throw,Name,Cont,nonvar(Cont,Cont)).
```

where `lval(K1,K2,Val)` is a BinProlog primitive which unifies `Val` with a backtrackable global logical variable accessed by hashing on two (constant or variable) keys `K1,K2`.

This allows for instance to avoid execution of the infinite `loop` from inside the predicate `b/1`.

```
loop:-loop.

c(X):-b(X),loop.

b(hello):-throw(here).
b(bye).

go:-catch(c(X),here),write(X),nl.
```

Notice that due to its simple *translation semantics* this program still has a first order reading and that BinProlog's `lval/3` is not essential as it can be emulated by an extra argument passed to all predicates.

Although implementation of `catch` and `throw` requires full-blown continuations, we can see that at user level, ordinary clause notation is enough.

9 Backtrackable destructive assignment

9.1 Updatable logical arrays in Prolog: fixing the semantics of setarg/3

Let us recall that `setarg(I,Term,Value)` installs `Value` as the `I`-th argument of `Term` and takes care to put back the old value found there on backtracking.

`Setarg/3` is probably the most widelly used (at least in SICStus, Aquarius, Eclipse, ProLog-by-BIM, BinProlog), incarnation of backtrackable mutable arrays (overloaded on Prolog's universal *term* type).

Unfortunately `setarg/3` lacks a logical semantics and is implemented differently in in various systems. This is may be the reason why the standardization (see its absence from the Prolog ISO Draft) of `setarg/3` can hardly reach a consensus in the predictable future.

Ideally, `setarg/3` should work as if a new term (array) had been created identical to the old one, except for the given element. There's no reason to 'destroy' a priori the content of the updated cell or to make it subject to ugly side effects. Should this have to happen for implementation reasons, a run-time error should be produced, at least, although this is not the case in any implementation we know of.

Let us start with an inefficient but fairly clean array-update operation, `setarg/4`.

```
setarg(I,OldTerm,NewValue,NewTerm):-
  functor(OldTerm,F,N),
  functor(NewTerm,F,N),
  arg(I,NewTerm,NewValue),
  copy_args_except_I(N,I,OldTerm,NewTerm).

copy_args_except_I(0,_,_,_):-!.
copy_args_except_I(I,I,Old,New):-!,I1 is I-1,
  copy_args_except_I(I1,I,Old,New).
copy_args_except_I(N,I,Old,New):-N1 is N-1,arg(N,Old,X),arg(N,New,X),
  copy_args_except_I(N1,I,Old,New).
```

We can suppose that functor and arg are specified by a (finite) set of facts describing their behavior on the signature of the program. For a given program, we can obviously see setarg/4 as being specified similarly by a finite set of facts.

Furthermore,suppose that all uses of setarg/3 are replaced by calls to setarg/4 with the new states passed around with DCG-style chained variables.

This looks like a good definition of the intended meaning of a program using setarg/3.

We will show that actual implementations (Sicstus and BinProlog) can be made to behave accordingly to this semantics through a small, source level wrapping into a suitable ADT.

Let `'$box'/1` be a new functor not in the signature of any user program. By defining

```
safe_arg(I,Term,Val):-arg(I,Term,'$box'(Val)).

safe_setarg(I,Term,Val):-setarg(I,Term,'$box'(Val)).
```

Using `'$box'/1` in `safe_arg/3` (`safe_setarg/3`) ensures that cell `I` of the functor `Term` will be indirectly accessed (updated) even if it contains a variable which in a WAM would be created on place and therefore it would be subject of unpredictable side-effects.

The reason of the draconian warning in some Prolog manuals manual

> ...setarg is only safe if there is no further use of the old value of the replaced argument...

. will therefore disappear and a purely logical setarg (with a *translation semantics* expressible in term of `setarg/4`) can be implemented. Not only this ensures referential transparency and allows normal references to the old content of the updated cells but it also makes incompatible implementations of setarg (Sicstus, Eclipse, BinProlog) work exactly the same way[5].

To finish the job properly, something like the following ADT can be created.

```
new_array(Size,Array):-
  functor(Array,'$array',Size).

update_array(Array,I,Val):-
  safe_setarg(I,Array,Val).

access_array(Array,I,Value):-
  safe_arg(I,Array,Value).
```

We suggest to use this ADT in your program instead of basic setarg when performance is not an absoulte requirement.

10 'Invisible' grammars

By using backtrackable setarg/3 and logical global variables (lval/3) we can implement easily a superset of DCG grammars practically equivalent with Peter VanRoy's Extended DCGs. We call them *invisible grammars* as no preprocessor is involved in their implementation. It turns out that the technique has the advantage of 'metaprogramming for free' (without the expensive phrase/3), allows source level debugging and can be made more space and time efficient than the usual prprocessing based implementation. On real examples, their best use is for writting a Prolog compiler (they can contribute to the writing of compact and efficient code with very little programming effort) and for for large natural language processing systems.

Basically, they work as follows:

[5]A further ambiguity in some implementations of `setarg/3` comes from the fact that it is not clear if the location itself or its dereferenced contents should be mutated

```
% tools

begin_dcg(Name,Xs):-lval(dcg,Name,Xs-Xs).

end_dcg(Name,Xs):-lval(dcg,Name,Xs-[]).

w(Word,Name):-
  lval(dcg,Name,State),
  State=_-[Word|Xs2],
  setarg(2,State,Xs2).

begin_dcg(Xs):-begin_dcg(default,Xs).
end_dcg(Xs):-end_dcg(default,Xs).
w(Word):-w(Word,default).

% grammar
x:-ng,v.

ng:-a,n.

a:-w(the).
a:-w(a).

n:-w(cat).
n:-w(dog).

v:-w(walks).
v:-w(sleeps).

% test
go:-begin_dcg(Xs),x,end_dcg(Ys),write(Ys),nl,fail.

?- go.
[the,cat,walks]
[the,cat,sleeps]
[the,dog,walks]
[the,dog,sleeps]
[a,cat,walks]
[a,cat,sleeps]
[a,dog,walks]
[a,dog,sleeps]
```

The program can be found in progs/setarg_dcg.pl. For reasons of efficiency (i.e. to equal or beat preprocessor based DCGs in terms of both space and time) BinProlog's 'invisible grammars' have been implemented in C and are accessible through the following set of builtins:

```
dcg_connect/1  % works like 'C'/3 with 2 invisible arguments
b0(dcg_def/1   % sets the first invisible DCG argument
```

```
b0(dcg_val/1   % retrives the current state of the DCG stream
b0(dcg_tell/1  % focus on a given DCG stream (from 0 to 255)
b0(dcg_telling/1 % returns the number of the current DCGs stream

% INVISIBLE DCG connect operation: normally macro-expanded
'#'(Word):-dcg_connect(Word).

% example: ?-dcg_phrase(1,(#a,#b,#c),X).
dcg_phrase(DcgStream,Axiom,Phrase):-
  dcg_telling(X),dcg_tell(DcgStream),
    dcg_def(Phrase),
      Axiom,
    dcg_val([]),
  dcg_tell(X).
```

11 BinProlog's C-interface

To be able to extend BinProlog and possibly embedd it as logic engine in your no run-time fee application you will need a BinProlog 3.0 C-source code licence. See the files SOURCE.LICENCE and PRICING for more information. You can also interact with other languages under UNIX using the bidireactional pipe based interface starting from BinProlog's Tcl/Tk interfaced (see directory TCL).

The following sequence of quick examples shows how it works.

11.1 Calling C from BinProlog: adding new builtins

New builtins are added in header.pl and after a "make realclean; make" a new version of BinProlog containing them is generated automatically.

An example of declaration in headers.pl is:

```
  b0(+/3,arith(1),in_body). % arity=3+1 by binarization
```

Notice that arith(1) means that it is "like an arithmetic functions" which returns *one* value, i.e this should be used as in

```
  +(0,1,Result).
```

I would suggest to start with the simplest form of interaction: a call of your own C-code from BinProlog. Try modifying the file c.c (provided with the C-sources of BinProlog), function `new_builtin()` which looks as follows:

```
/*
   New builtins can be called from Prolog as:
   new_builtin(0,<INPUT_ARG>,<OUTPUT_ARG>)
```

```
   X(1) contains the integer `opcode' of your builtin
   X(2) contains your input arg
   regs[I] contains somthing that will be unified with what you return
      (the precise value of I depends on the register allocator).

   You are expected to `return' either
    - a non-null object that will be unified with <OUTPUT_ARG>, or
    - NULL to signal FAILURE

   As the returned object will be in a register this
   can be used for instance to add a garbage collector
   that moves every data area around...

*/

term new_builtin(H,regs,A,P,wam)
  register term H,regs,*A;
  register instr P;
  register stack wam;
{  BP_check_call();
   switch(BP_op)
  {
    /* for beginners ... */

    case 0:
       /* this just returns your input argument (default behavior) */
    break;

    case 1:
      BP_result=BP_integer(13); /* this example returns 13 */
    break;

    case 2:
      BP_result=BP_atom("hello"); /* this example returns 'hello' */
    break;

    /* for experts ... */

    case 3: /* iterative list construction */
    { cell middle,last,F1,F2; int i;
      BP_make_float(F1, 77.0/2);
      BP_make_float(F2, 3.14);

      BP_begin_put_list(middle);
         BP_put_list(BP_integer(33));
         BP_put_list(F1);
         BP_put_list(BP_string("hello"));
         BP_put_list(F2);
      BP_end_put_list();
```

```
  BP_begin_put_list(last);
    for(i=0; i<5; i++) {
      BP_put_list(BP_integer(i));
    }
  BP_end_put_list();

  BP_begin_put_list(BP_result);
    BP_put_list(BP_string("first"));
    BP_put_list(middle);
    BP_put_list(last);
    BP_put_list(F1);
    BP_put_list(F2);
  BP_end_put_list();
} break;

case 4: /* cons style list construction */
  BP_begin_cons();
  BP_result=
   BP_cons(
      BP_integer(1),
      BP_cons(
           BP_integer(2),
           BP_nil
      )
   );
  BP_end_cons();
break;

case 5: /* for hackers only ... */ ;
  BP_result=(cell)H;

  H[0]=g.DOT;
  H[1]=X(2);

  H[2]=g.DOT;
  H[3]=BP_integer(99);

  H[4]=g.DOT;
  H[5]=(cell)(H+5); /* new var */

  H[6]=g.DOT;
  H[7]=(cell)(H+5); /* same var as previously created */

  H[8]=g.DOT;
  H[9]=BP_atom("that's it");

  H[10]=g.NIL;
  H+=11;
```

```
break;

case 6:
  BP_fail();
break;

case 7:
  {  cell T=BP_input;
     if(BP_is_integer(T))
        {fprintf(g.tellfile,"integer: %d\n",BP_get_single_integer(T));
         BP_result=BP_integer(-1);
        }
     else
        BP_fail();
  }
break;

case 8: /* for experts: calling BinProlog from C */
  { cell L,R,Goal;

    BP_begin_put_list(L);
      BP_put_list(BP_string("one"));
      BP_put_list(BP_integer(2));
      BP_put_list(BP_string("three"));
      BP_put_list(BP_input); /* whatever comes as input */
    BP_end_put_list();

    BP_put_functor(Goal,"append",3);
    BP_put_old_var(L);
    BP_put_old_var(L);
    BP_put_new_var(R);

    BP_prolog_call(Goal); /* this will return NULL on failure !!!*/

    BP_put_functor(Goal,"write",1);
    BP_put_old_var(R);

    BP_prolog_call(Goal); /* calls write/1 */

    BP_put_functor(Goal,"nl",0);

    BP_prolog_call(Goal); /* calls nl/0 */

    BP_result=R; /* returns the appended list to Prolog */

  }
  break;
```

```
    /* EDIT AND ADD YOUR CODE HERE....*/

    default:
     return LOCAL_ERR(X(1),"call to unknown user_defined C function");
  }
  return H;
}
```

11.2 Calling Prolog from C

```
/* this can be used to call Prolog from C : see example if0 */

term bp_prolog_call(goal,regs,H,P,A,wam)
  register term goal,regs,H,*A;
  register instr P;
  register stack wam;
{
  PREP_CALL(goal);
  return bp(regs,H,P,A,wam);
}

/* simple example of prolog call */
term if0(regs,H,P,A,wam)
  register term regs,H,*A;
  register instr P;
  register stack wam;
{ term bp();
  cell goal=regs[1];

  /* in this example the input GOAL is in regs[1] */
  /* of course you can also build it directly in C */
  /* unless you want specific action on failure,
     use BP_prolog_call(goal) here */

  H=bp_prolog_call(goal,regs,H,P,A,wam);
  if(H)
     fprintf(stderr,"success: returning from New WAM\n");
  else
     fprintf(stderr,"fail: returning from New WAM\n");

  /* do not forget this !!! */
  return H; /* return NULL to signal failure */
}
```

BinProlog's `main()` should be the starting point of your program to be able to initialize all data areas. To call back from C you can follow the example if0. A sustained BinProlog-C dialog can be set up by using the 2 techniques described previously.

The C-interface (composed of files c.h c.c c.pl) is activated with `#define C_INTERFACE`.

After doing a "make" your changes will be integrated in the BinProlog ru file. If you whish you can create a standalone C-executable by using BinProlog's compilation to C (see directory wam2c).

12 Example programs

The directory `progs` contains a few BinProlog benchmarks and applications.

```
allperms.pl: permutation benchmarks with findall
bestof.pl:   an implementation  of bestof/3
bfmeta.pl:   breadth-first metainterpreter
bp.pl:       float intensive neural net learning by back-propagation
cal.pl:      calendar: computes the last 10000 fools-days
fcal.pl:     calendar: with floats
chat.pl:     CHAT parser
choice.pl:   Choice-intensive ECRC benchmark
cnrev.pl:    nrev with ^/2 as a constructor instead of ./2
cube.pl:     E. Tick's benchmark program
fibo.pl:     naive Fibonacci
ffibo.pl:    naive Fibonacci with floats
hello.pl:    example program to create stand-alone Unix application
knight.pl:   knight tour to cover a chess-board (uses the bboard)
lknight.pl:  knight tour to cover a chess-board (uses the lists)
ltak.pl:     tak program with lemmas
lfibo.pl:    fibo program with lemmas
lq8.pl :     8 queens using global logical variables
maplist.pl:  fast findall based maplist predicate
nrev.pl:     naive reverse
nrev30.pl:   small nrev benchmark to reconsult for the meta-interpreter
or.pl:       or-parallel simulator for binary programs (VT100)
other_bm*:   benchmark suite to compare Sicstus, Quintus and BinProlog
puzzle.pl:   king-prince-queen puzzle
q8.pl:       fast N-queens
qrev.pl:     quick nrev using builtin det_append/3
subset.pl:   findall+subset
tetris.pl:   tetris player (VT100)
```

13 Related work

Some BinProlog related papers can be found at clement.info.umoncton.ca in the file `papers.tar.Z`.

The reader interested in the internals of BinProlog and other issues related to binary logic programs, their transformations and performance evaluation is referred to [16, 9, 11, 4, 5, 12, 10, 8, 7, 14, 13, 1, 19, 6, 17, 15, 18, 3, 2]

References

[1] K. De Bosschere and P. Tarau. Blackboard-based logic programming. In *Proceedings of the 1993 ILPS Conference*, Vancouver, Canada, 1993. poster.

[2] K. De Bosschere and P. Tarau. Blackboard Communication in Logic Programming. In *Proceedings of the PARCO'93 Conference*, Grenoble, France, Sept. 1993.

[3] K. De Bosschere and P. Tarau. High Performance Continuation Passing Style Prolog-to-C Mapping. In E. Deaton, D. Oppenheim, J. Urban, and H. Berghel, editors, *Proceedings of the 1994 ACM Symposium on Applied Computing*, pages 383–387, Phoenix/AZ, Mar. 1994. ACM Press.

[4] B. Demoen. On the Transformation of a Prolog program to a more efficient Binary program. Technical Report 130, K.U.Leuven, Dec. 1990.

[5] B. Demoen and A. Mariën. Implementation of Prolog as binary definite Programs. In A. Voronkov, editor, *Logic Programming, RCLP Proceedings*, number 592 in Lecture Notes in Artificial Intelligence, pages 165–176, Berlin, Heidelberg, 1992. Springer-Verlag.

[6] T. Lindgren. Compiling logic programs using a binary continuation style, Dec. 1992. draft, Uppsala University.

[7] U. Neumerkel. *Specialization of Prolog Programs with Partially Static Goals and Binarization.* Phd thesis, Technische Universität Wien, 1992.

[8] U. Neumerkel. A transformation based on the equality between terms. In *Logic Program Synthesis and Transformation, LOPSTR 1993*. Springer-Verlag, 1993.

[9] P. Tarau. A Simplified Abstract Machine for the execution of Binary Metaprograms. In *Proceedings of the Logic Programming Conference'91*, pages 119–128. ICOT, Tokyo, 7 1991.

[10] P. Tarau. Ecological Memory Managment in a Continuation Passing Prolog Engine. In Y. Bekkers and J. Cohen, editors, *Memory Management International Workshop IWMM 92 Proceedings*, number 637 in Lecture Notes in Computer Science, pages 344–356. Springer, Sept. 1992.

[11] P. Tarau. Program Transformations and WAM-support for the Compilation of Definite Metaprograms. In A. Voronkov, editor, *Logic Programming, RCLP Proceedings*, number 592 in Lecture Notes in Artificial Intelligence, pages 462–473, Berlin, Heidelberg, 1992. Springer-Verlag.

[12] P. Tarau. Wam-optimizations in BinProlog: towards a realistic continuation passing prolog engine. Technical Report 92-3, Dept. d'Informatique, Université de Moncton, July 1992. available by ftp from clement.info.umoncton.ca.

[13] P. Tarau. An efficient specialization of the wam for continuation passing binary programs. In *Proceedings of the 1993 ILPS Conference*, Vancouver, Canada, 1993. poster.

[14] P. Tarau. Language issues and programming techniques in BinProlog. In D. Sacca, editor, *Proceeding of the GULP'93 Conference*, Gizzeria Lido, Italy, June 1993.

[15] P. Tarau. Low level issues in implementing a high-performance continuation passing binary prolog engine. In M.-M. Corsini, editor, *Proceedings of JFPL'94*, June 1994.

[16] P. Tarau and M. Boyer. Elementary Logic Programs. In P. Deransart and J. Małuszyński, editors, *Proceedings of Programming Language Implementation and Logic Programming*, number 456 in Lecture Notes in Computer Science, pages 159–173. Springer, Aug. 1990.

[17] P. Tarau and B. Demoen. Language embedding by dual compilation and state mirroring. In *Proceedings of 6-th Workshop on Logic Programming Environments, Santa Margherita Ligure, 1994*, June 1994.

[18] P. Tarau and U. Neumerkel. A Novel Term Compression Scheme and Data Representation in the BinWAM. In *Proceedings of Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science. Springer, Sept. 1994.

[19] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Oct. 1983.