BinProlog 2006 11.x Professional Edition Advanced BinProlog Programming and Extensions Guide

Paul Tarau

BinNet Corp.
WWW: http://www.binnetcorp.com

July 7, 2008

1 Introduction to BinProlog

BinProlog has been developed by Paul Tarau (http://www.cs.unt.edu/˜ tarau) 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 compact 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 also probably the first Prolog system featuring dynamic recompilation of asserted predicates (a technique similar to the one used in some object oriented languages like SELF 4.0), and a very efficient segment preserving copying heap garbage collector.

Incorporating more than 15 years of research on compilation of Prolog and logic programming based Internet development, BinProlog is a robust and complete Prolog implementation featuring both C-emulated execution and generation of standalone applications by compilation to C.

New features added in this edition of BinProlog include:

- Mobile code, multi-threading for Windows
- Built-in, easy to use, high-level Internet programming tools (part of the BinNet Internet toolkit), including CGIs and sockets
- term-unification based high-level Linda server and client operations
- packaging of BinProlog as a DLL with examples of calls from C and Java
- new, robust LogiMOO multiuser cooperative work environment ('grupware') on top of BinProlog's new Linda operations (a BinNet Internet Server demo application)

.

1.1 Contents of the BinProlog package

BinProlog distributions contain a subset of the following directories.

```
+---doc
+---winbp - only available with Windows Visual C source version
+---src - only available with source license
+---progs
+---library
+---TCL
+---c_inter
+---pl2c
+---csocks
+---BP_DLL - only available with Windows versions
+---tests
+---bin
+---lib
```

Prolog sources for the compiler and builtins are in directory `src` (only available with BinProlog source license).

BinProlog binary libraries are in directory lib.

The directory `doc` contains ASCII, Latex, PostScript and HTML versions of BinProlog documentation and a makefile to regenerate it.

The directory `pl2c` contains various project (*.pro) files and a makefile for generating a standalone applications through compilation to C.

The directory `c_inter` contains BinProlog's C-interface and examples of embedding of BinProlog in C applications.

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

The directory `BP_DLL` contains a Prolog dynamic library with an example of C application calling it (useful for owners of source license).

The directory `csocks` contains standalone socket code for embedding in C programs serving as client or server components interoperating with BinProlog.

In case you have BinProlog Full Source Edition, the directory src contains makefiles for gcc and VCC batch files to rebuild the sources after you make changes. Just go in directory src and type make all or makeall.bat.

Please read the documentation files provided separately and visit our ONLINE HELP SYSTEM at BinNet Corp.'s web site, for last minute updates, demos and information about new features and components.

On Cygnus' CYGWIN gcc, type

make all

which also works on normal Unix systems not requiring unusual flags.

On Linux PCs type

make linux

On Solaris sparcs type

make solaris

This will take care to generate the content of most of the directories.

Finally, the Full Source Edition also contains VCC 6.0 project files in directory winbp, for working conveniently with the sources on a Windows NT workstation.

Alternatively, work on sources is now also supported for Cygnus gcc as well as Linux and othe Unix platforms, through a complete set of updated makefiles.

1.2 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 your own binary programs by using

```
::-
```

instead of

```
:-
```

so that the preprocessor will not touch them[1]. 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 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 (136 million LIPS on a 2.4 GHz Pentium 4 Toshiba PC).

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 Prolog.

1.3 Platforms supported

```
Windows XP/2000/NT/95/98 (Visual C version, multi-threaded)
Windows XP/2000/NT/95/98 (Cygnus gcc version, using free cygwin b20 C compiler)
Linux-x86 gcc version
Solaris-sparc gcc version
Other 32 or 64 bit Unix platforms, upon customer request
```

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 or Itanium or G5 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.

2 Using BinProlog

2.1 Consulting/compiling files

BinProlog features a number of different compilation and consulting methods as well as dynamic recompilation of consulted (interpreted) code for fast execution.
The shorthand

[1]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.

```
   ?- [myFile].
```

defaults to the last used compilation method applied to `myFile.pro` `myFile.pl` or `myFile`. Among them, the default mcompile/1 compiling to memory and scompile/1 which uses temporary *.wam files to quickly load files which have not been changed. A good way to work with BinProlog is to make a "project" *.pro file includeing its components, as in:

```
:-[myFile1].
:-[myFile2].
..............
```

The shorthand

```
    ?- ~my_file
```

defaults to the last used *interpretation* method applied to `myFile.pro` `myFile.pl` or `myFile`. For online information on their precise behavior, do:

```
?-info(oconsult),info(consult),info(dconsult),info(sconsult).

oconsult/1:
  reconsult variant, consults and overwrites old clauses

consult/1:
  consults with possible duplication of clauses, allows
  later dynamic recompilation

consult/2:
  consult(File,DB) consults File into DB

dconsult/1: reconsult/1 variant, cleans up data areas,
  consults, allowing dynamic recompilation

sconsult/1:
  reconsult/1 variant:
    cleans up data areas, consults, makes all static
```

To control dynamic recompilation you can use `dynco/1` with `yes` and `no` as arguments or `db_ratio/1` to precisely specify the ratio between calls and updateds to predicates which will trigger moving it from interpreted to compiled representation (default=10).

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 a dozen of seconds on todays fast Pentium II/III machines.

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

An original term compression technique [31] (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 garbage collected.

For permanent information BinProlog has a, dynamically growing/shrinking 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 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.
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 has Edinburgh behavior 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. Some ISO Prolog extensions (stream I/O) are now present in BinProlog (see Appendix for a list of builtins) together with their C-style equivalents.

All the basic Prolog utilities are now supported (dynamic clauses, a metainterpreter with tracing facility, sort, setof, dynamic operators floating point operations and functions). BinProlog is one of the fastest Prolog systems around. Naive reverse makes more than 136 million LIPS on a Pentium 4 2.4GHz machine. Member/2 and for/3 are also C-based builtins for improved performance.

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

On 32 bit machines BinProlog has 30 bit integer arithmetic, *64 bit floating point operations* and functions like sin, cos, tan, log, exp, sqrt etc. (On 64 bit machines BinProlog supports 61 bit integer arithmetic). Arithmetic operations can be used either through the is/2 interface[2]:

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

```
?- 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. BinProlog's is/2 commutes with constructors and evaluates arbitrary functions (predicates returning results as their last argument), therefore something like

```
  X is [cos(3.14),2+3]++[10*10,5-2].
```

will return

```
X=[-0.999999,5,100,3]
```

Floating point 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 `backprop.pl`. This program uses also constant time arrays and is therefore unusually fast compared to its execution in other Prologs like Quintus or Sicstus.

2.2 Using multiple logic engines

A new (experimental) feature added starting with version 3.39 alows launching multiple Prolog engines having their own stack groups (heap, local stack and trail). An engine can be seen as an abstract data-type which produces a (possibly infinite) stream of solutions as needed. To create an engine use:

```
  % :-mode create_engine(+,+,+,-).
  create_engine(HeapSize,StackSize,TrailSize,Handle)
```

The Handle is a unique integer denoting the engine for further processing. To 'fuel' the engine with a goal and an expected answer variable do:

```
  % :-mode load_engine(+,+,+).
  load_engine(Handle,Goal,AnswerVariable)
```

No processing except the initialization of the engine takes place and no answer is returned with this operation. Think about it as simply puting gas in your car.

To get an answer from the engine do:

```
  % :-mode ask_engine(+,-).
  ask_engine(Handle,Answer)
```

When the stream of answers reaches its end, `ask_engine/2` will simply fail.

If for some reasons you are not interested in the engine anymore, do

```
  % :-mode destroy_engine(+).
  destroy_engine(Handle)
```

to free the data areas alocated for the engine. If you want to submit another query before using the complet stream of answers, it is more efficient to reuse an existing engine with `load_engine/3`, instead of destroying it and creating a new one.

Try out the following example (see more in files `library/engines, progs/engtest.pl`):

```
 ?-create_engine(256,64,64,E),
   load_engine(E,append(As,Bs,[A,B,B,A]),As+Bs),
   ask_engine(E,R1),write(R1),nl,
   ask_engine(E,R2),write(R2),nl,
   load_engine(E,member(X,[1,2,3]),X),
   ask_engine(E,R3),write(R3),nl,
   ask_engine(E,R4),write(R4),nl,
   destroy_engine(E).
```

As engines will be assigned to real processors in future multi-threaded implementations this reusability of a given engine for execution of multiple goals is intended to allow precise control to the programmer over the resurces of a system. Preemptive multitasking handled either with thread libraries, when available and efficient, or by BinProlog itself otherwise, is planned in the near future.

This 'super-scalar' performance improvement can already be tested with the companion (Solaris 2.x only) Linda implementation in directory `multi` (see file `multi/myprogs/mcolor.pl`).

If an engine fails due to the overflow of a given data area, it will warn you and fail. The warnings are disabled with quietness levels higher than 6 (command line switch `q6`). This allows use of engines to quietly recover from ressource errors.

For owners of BinProlog's C-source licenses, C-functions with similar names and semantics as the predicates described in this section are available to allow embedding of multiple independent Prolog engines in their C-applications.

2.3 Findall/3 with multiple engines

The file `library/engines.pl` contains some examples of how multiple engines can be used for implementing for instance all-solution predicates. Here is a re-entrant `find_all/3`.

```
find_all(X,Goal,Xs):-
  default_engine_params(H,S,T),
  find_all(H,S,T,X,Goal,Xs).

find_all(H,S,T,X,Goal,Xs):-
  create_engine(H,S,T,Handle),
  load_engine(Handle,Goal,X),
  collect_answers(Handle,Xs),
  destroy_engine(Handle).

collect_answers(Handle,[X|Xs]):-ask_engine(Handle,A),!,
  copy_term(A,X),
  collect_answers(Handle,Xs).
collect_answers(_,[]).

default_engine_params(128,32,32).
```

2.4 Standalone applications through compilation to C

BinProlog allows to separately compile user applications and just link them with the emulator library and the C-ified compiler (see directory `pl2c`). This allows creation of a fully C-ified application in a few seconds.

Just type `make PROJ=queens` in directory `pl2c`. The standalone application `queens` is ready to be executed by typing `queens`. The generated C-code can be seen in files `queens.h` and `queens.c`.

Moreover, on systems with dynamic linking like Solaris 2.x true executables of size starting at about 6K can be created starting with version 3.30 (see directory `dynpl2c`).

We refer to [28] for the details of this translation process.

If you define a predicate `main/0` then your executable will start directly from there instead of the usual interactive top-level. Calling it with a high quietness-level (i.e. command line switch `q5`) will suppress warnings and unwanted messages.

Normally BinProlog starts execution from main/1:

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

BinProlog's shell is nothing but such an application which starts with a toplevel loop i.e. something like:

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

You can override this behavior, even for the standard BinProlog system containing the predicate `main/1` (which starts the interactive top-level), by defining a predicate `main/0` in your program file. In this case, `main/0` becomes the new starting point.

You can bootstrap the compiler (if you own a source license), after modifying the `*.pl` files by:

```
?- boot.
?- make.
```

or, similarly for any other project having a top `*.pro` file:

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

or

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

if you intend to generate C-code and possibly hide non-public predicates inside a module.

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.
```

2.5 Some limitations/features of BinProlog

BinProlog is fairly close to the ISO standard both syntactically and semantically.

We have 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. These are usually unspecified by the ISO standard, therefore this does not affect our ISO compliance.

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

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

or

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

As BinProlog supports an include directive:

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

this is probaly the best way to work, in a stateless always clean workspace. Note that, following the ISO standard, multiple includes result in only one copy of the file being included.

BinProlog allows consulting multiple files subject to dynamic recompilation (use dconsult/1 to get something close to SICStus/Quintus loading convention) or use command line switch l4 to have this as default behavior for reconsult (with shorthand
`file`).

With dynamically recompiled consulted code, listing of sources and dynamic modification to any predicate is available while average performance stays close to statically compiled code (usually within a factor of 2-3).

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 `(*.wam)` files. For large projects this is the recommended technique. Creation of C-ified standalone files is also possible (see the `pl2c 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 automate this job. When everything is OK you can deliver it as a run-time-only application. Releases after 3.30 contain a basic make facility which avoids to recompile included `*.pl` files when a newer `*.wam` file exists.

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. See the file extra.pl for the implementation. Similar to asserted but update-backtrackable, is BinProlog's *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, unless overriden by multifile/1 or discontiguous/1 declarations).
- ARITY is limited to 255.

2.6 Garbage Collection

A high performance copying heap garbage collector is implemented in versions 4.x and newer (thanks to Bart Demoen and Geert Engels). `nogc/0` switches it off, `gc/0` switches it on. Gc activation messages can be made silent using quietness levels like `q3` on the command-line, (now the default). Moreover, the blackboard, dynamic code space, the string space and the hashing table ARE garbage collected before loading a new program.

2.7 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  - returns a new symbol made from its arguments
gensym/2  - forms a new name of the form name_counter
sread/2   - reads from a name a (ground) term
swrite/2  - writes a term to a name
termcat/3 - 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

            iterates over 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 to generate a customized `wam.bp` file which integrates your changes when passed as a command line parameter to the `bp` executable.

2.8 Functional style higher-order programming

BinProlog integrates a number of useful higher-order predicates working on deterministic, functional style predicates. If your program is mostly functional the best thing is of course to forget about logic programming and write it in Haskell. However, with efficient implementatations of map/3, foldl/4, foldr/4 and the call/N as builtins BinProlog reduces the amount of extra argument book-keeping programmers face. Sum/2 and prod/2, computing the sum and the product of a list (now also builtins) are good examples of this more compact and fairly efficient higher-order programming style.

```
sum(Xs,R):-foldl(+,0,Xs,R).

prod(Xs,R):-foldl(*,1,Xs,R).
```

Note that all functions (deterministic predicates returning result in their last argment) can be used from is/2 which also commutes with constructors, i.e. something like

```
   X is [5+sum([1,2,3]),sum([10,20])].
```

2.9 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.

This *generator* based programming style puts nondeterminism to work as a relatively clean and encapsulated form of iteration.

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 and preventing space consumption is always a good idea despite the fact that BinProlog 6.25's efficient heap garbage collector will get your memory back anyway.

2.10 Builtins

BinProlog has large number of builtins with semantics (intended to be) close to SICStus and QUINTUS Prolog.

We refer to the automatically generated info file in Appendix for short definitions and examples of uses of BinProlog's documented builtins.

3 Macro processing

Two forms of macro processing facilities are available in BinProlog. One is `term_expansion/2` allowing to define toplevel operators (like the DCG arrow) to be expanded to whatever the user wants to. Starting with version 5.82, `term_expansion/2` facts can be either assumed or asserted. Another is compile-time execution `##/1` which executes arbitrary goals once while compiling it. For instance, it

```
pi2(Y):- ##((X is 2*asin(1),Y is X*X)).
```

arithmetic operations will be executed once while loading the file, instead of being executed each time `pi2` is called.

Note that both operations should only call predicates which are *already* defined, before the compilation starts.

4 Inspecting some BinProlog internals

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
```

5 Compiling to C

Partial C-ification [30] is a translation framework which 'does less instead of doing more' to improve performance of emulators close to native code systems.

Starting from an emulator for a language L written in C, we translate to C a subset of its instruction set (usually frequent and fine-grained instructions which are executed in contiguous sequences) and then simply use a compiler for C to generate a unique executable program.

A translation threshold allows the programmer to empirically fine-tune the C-ification process by choosing the length of the emulator instruction sequence, starting from which, translation is enabled. The process uses a reasonable default value and can be easily controlled by the programmer

```
:-set_c_threshold(Min,Max).
```

will ensure that only emulated sequences of length between Min and Max get translated to C. This allows to handle gracefully the size/speed tradeoff.

Communication between the run-time system (still under the control of the emulator) and the C-ified chunks is handled as follows.

The emulated code representation of a given program (in particular the compiler itself) is mapped to a C data structure which allows exchange of symbol table information at link time.

To be able to call a C-routine from the emulator we have to know its address. Unfortunately, the linker is the only one that knows the eventual address of a C-routine. A simple and fully portable technique to plug the address of a C-routine into the byte code is to C-ify the byte-code of the emulator into a huge C array of records, containing the symbolic address of the C-chunks. After compilation, and linking with the emulator, the linker will automatically resolve all the missing addresses and generate warnings for the missing C-routines.

This is compiled together with the C-code of the emulator to a stand alone executable with performance in the range between pure emulators and native code implementations.

The method ensures a strong operational equivalence between emulated and translated code which share exactly the same observables in the run-time system.

An important characteristic is easy debugging of the resulting compiler, coming from the full sharing of the run-time system between emulated and compiled code and the following property we call *instruction-level compositionality*: if every translated instruction has the same observable effect on a (small) subset of the program state (registers and a few data areas) in emulated and translated mode, then arbitrary sequences of emulated and translated instructions are operationally equivalent.

Currently C-ification covers term creation on the heap and frequently used inline operations which can be processed in Binary Prolog before calling the 'real goal' in the body.

Chunks containing small built-ins that do not require a procedure call will generate 'leaf-routines' in C (which are called efficiently and do not use stack space).

On the other hand large built-ins implemented as macros in the emulator would make code size explode. Implementing them as functions to be called from the C-chunk would require code duplication and it would destroy the leaf-routine discipline which is particularly rewarding on Sparcs. We have chosen to implement them through an abstraction with a coroutining flavor: *anti-calls*. Note that calling a built-in from a C-chunk is operationally equivalent to the following sequence:

- return from the chunk,
- execute the built-in in the emulator (usually a macro),
- call a new leaf-routine to resume the work left from the previous leaf-routine.

Overall, anti-calls can be seen as form of coroutining (jumping back and forth) between native and emulated code. Anti-calls can be implemented with the direct-jump technique even more efficiently, although for portability reasons we have chosen a conventional return/call sequence, which is still fairly efficient as a return/call costs the same as a call/return. Moreover, this allows the chunks to remain leaf-routines, while delegating overflow and signal handling to the emulator. Note that excessively small chunks created as result of anti-calls are removed by an optimizing step of the compiler with the net result that such code will be completely left to the emulator. This is of course *more compact* and provable to be *not slower* than its fully C-expanded alternative.

5.1 Performance of C-ified code

The speed-up clearly depends on the amount of C-ification and on the statistical importance of C-ified code in the execution profile of a program (see figure 1). We have noticed between 10-20% speed increase for programs which take advantage of C-ified code moderately, As these programs spend only 20-30% of their time in C-ified sequences performances are expected to scale correspondingly when we extend this approach to the full BinProlog instruction set and implement low-level gcc direct jumps instead of function calls and anti-calls.

Code-sizes for C-ified BinProlog executables (dynamically linked on Sparcs with Solaris) are usually even smaller than 'compact' Sicstus code which uses classical instruction folding (a few hundreds of opcodes) to speed-up the emulator.

The following table shows some code-size/execution-speed variations with respect to the threshold for the semi-ring (`SEMI3`) benchmark. Clearly, excessively small chunks can influence adversely not only on size but also on speed. Something like threshold=20, looks like a practical optimum for this program.

Bmark/Compiler	emBP	C-BP	emSP	natSP
NREV (KLIPS)	445	455	412	882
CAL (ms)	490	310	590	310
FIBO (ms)	1730	1320	1400	800
TAK (ms)	610	470	400	180
SEMI3 (ms)	1810	1410	1810	1310
QUEENS (ms)	3170	2220	2840	1070

Figure 1: Performance of emulated (emBP) and partially C-ified BinProlog 3.22 (C-BP) compared to emulated (emSP) and native (natSP) SICStus 2.1_9 on a Sparc 10/20).

```
threshold:   0    4    8    20   30   1000 emBP emSP natSP
size:  (K)  34.5 32.2 29.9 16.3 13.1 12.9  4.8 22.0 31.9
speed: (ms) 1480 1430 1440 1450 1810 1790 1800 1810 1310
```

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 versions 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 logical global variable 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. With BinProlog and dynamic recompilation however, some applications might be faster overall by using assert and retract, depending on the update/call ratio of dynamic predicates (you can use command line switch -r to change this). Try both compiling and reconsulting the benchmark progs/assertbm.pl to have an estimate of the speed of these operations on your machine.

This is an approximation of other Prologs assert and retract predicates. It tries to be close to Sicstus and Quintus with their semantics.

If you want maximal efficiency for highly volatile data, use directly `bb_def/3, bb_set/3, bb_val/3, bb_rm` They give you access to a very fast hashing table

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

When using 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 Multiple dynamic databases

BinProlog features multiple dynamic databases. Only one is active at a time and you can switch among them with `set_db/1` or even move one over another with `db_move/2`.

Use info/1 to obtain more information on one of the following:

```
current_db/1-built_in
set_db/1-built_in
db_abolish/2-built_in
db_assert/2-built_in
db_asserta/2-built_in
db_asserted/2-built_in
db_assertz/2-built_in
db_clause/3-built_in
db_clean/0-built_in
db_clean/1-built_in
db_head/2-built_in
db_move/2-built_in
db_ratio/1-built_in
db_retract/2-built_in
db_retract1/2-built_in
db_retractall/2-built_in
```

6.5 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 expressiveness 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 behavior.

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.

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.

6.6 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 [13] 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.

6.7 An useful Prolog extension: bestof/3

`Bestof/3` is missing in other 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 *Note that use bestof requires a* `:-[library(high)]` *include command.*

```
?-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.

6.8 Blackboard based abstract data types

We will describe some simple utilizations 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.

6.8.1 Blackboard based failure surviving stacks/queues

A very useful data structure that is implemented with the blackboard is a stack/queue that survives failure but still allows the programmer to use some of the nice properties of logical variables. It is implemented through a set of C-based builtins in BinProlog and used among other things in the implementation of BinProlog's dynamic code.

```
addq/3 adds to end of persistent queue as in addq(key1,key2,33).
pushq/3 adds to beginning of a persistent queue
cpopq/3 pops (copy of) first element of the queue
cmembq/3 backtracks over (copies of) members of a queue
cdelq/4 deletes first matching element from a queue
```

See examples of use in the automatically generated Appendix.

6.8.2 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,
  for(I,0,Max), % generator for I from 0 to Max
    bb_let(Name,I,Zero),
  fail.
```

```
vector_def(_,_,_).

vector_set(Name,I,Val):-bb_set(Name,I,Val).

vector_val(Name,I,Val):-bb_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 `bb_def/3, bb_set/3, bb_val/3` as in:

```
global_array_set(I,J,Val):-bb_set(I,J,Val).
```

6.9 Blackboard based problem solving

6.9.1 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.

6.9.2 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, multi-argument hashing with BinProlog 6.25's `term_hash/3` can be used.

```
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.

Look into file dtak.pl for a Delphi-memoing version of the same program (a probabilistic automatic memoing method).

7 Continuations as first order objects

7.1 Continuation manipulation vs. intuitionistic/linear implication

Using intuitionistic implication 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/2`. 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 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/0` 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.

8.1 Standard Prolog catch and throw

BinProlog implements also Standard Prolog's catch/throw mechanism, using a similar technique (thanks to Bart Demoen!), which works as follows:

```
catch(Goal,Ball,Do):-
  execute Goal
  on throw(Term): look for closest catch
     copy Term
     undo bindings until call to catch; remove choices
     unify with Ball
        if fail: throw Ball again
        if succeed: call Do, continuation of catch goal
```

Note that our continuations and linear assumption based implementation ensures constant time execution of catch and throw, without actual stack scanning.

```
Example:

?-catch((member(X,[1,2,3]),throw(my_term(X))),my_term(Y),println(caught(Y))).
caught(1)
X=_x2352,
Y=1
```

8.2 Ancestor cut

For enhanced control BinProlog implements a form of ancestor cut,based on the following 3 built-ins:

```
get_neck_cut/1: gets the address of a choice point
                (to be use before the first not-inlined goal)

get_deep_cut/1: gets the address of a choice point
                (to be use after the first not-inlined goal)

cut_to/1:       cuts to a given choice-point

untrail_to/1:   removes bindings up to a given choice-point
```

Together with continuation manipulation and linear assumptions, the use of these built-ins allows a source-level implementation of catch and throw.

Another example of use, committing to the current resolution branch and cutting off all the alternatives up to toplevel, is written simply as:

```
commit :-
  assumed(catchmarker('$commit',Do,Choice,_)),
  cut_to(Choice), Do.
```

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 `Ith` 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[3].

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 absolute requirement.

A new `change_arg/3` builtin has been added to BinProlog to allow, efficient failure-driven iteration with persistent information. It works like `setarg/3` except that side-effects

[3]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

are permanent. Should unsafe heap objects be generated through the precess `change_arg/3` signals a run-time error. This is not the case as far the result is either a constant (which is does not need new heap allocation) or the result of moving a preexistent heap object to a new location.

For instance the (Haskell-style) `fold/4` predicate (see `library/high.pl`) uses `change_arg/3` to avoid painful iteration and slow side-effects on the dynamic database. The implementation is competitive is speed with hand-written explicitly recursive code and uses only memory proportional to the size of the answer. Alternatively, BinProlog also implements a functional style version of the same predicates, based on call/N (see help(fold)).

```
% fold,foldl based on safe failure driven destructive change_arg
foldl(Closure,Null,List,Final):-fold(Closure,Null,X^member(X,List),Final).

fold(Closure,Null,I^Generator,Final):-
  fold0(s(Null),I,Generator,Closure,Final).

fold0(Acc,I,Generator,Closure,_):-
  term_append(Closure,args(SoFar,I,O),Selector),
  Generator,
    arg(1,Acc,SoFar),
    Selector,
    change_arg(1,Acc,O),
  fail.
fold0(Acc,_,_,_,Final):-
  arg(1,Acc,Final).

?- foldl(+,0,[1,2,3],R).
?- fold(*,1,X^member(X,[1,2,3]),R).
```

10 Related work

BinProlog related papers can be found following links from

```
http://www.cs.unt.edu/~tarau
```

The reader interested in the internals of BinProlog and other issues related to binary logic programs, their transformations and performance evaluation is referred to [23, 12, 14, 6, 8, 15, 13, 11, 10, 17, 16, 2, 32, 9, 29, 18, 31, 26, 4, 3, 7, 24, 5, 1, 25, 27, 28]

Related BinProlog documentation is available at: [22, 20, 21, 19].

References

[1] K. De Bosschere, D. Perron, and P. Tarau. LogiMOO: Prolog Technology for Virtual Worlds. In *Proceedings of PAP'96*, pages 51–64, London, Apr. 1996.

[2] K. De Bosschere and P. Tarau. Blackboard-based Extensions for Parallel Programming in BinProlog. In *Proceedings of the 1993 ILPS Conference*, page 664, Vancouver, Canada, 1993. Poster Abstract.

[3] K. De Bosschere and P. Tarau. Blackboard Communication in Logic Programming. In *Proceedings of the PARCO'93 Conference*, pages 257–264, Grenoble, France, Sept. 1993.

[4] 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.

[5] K. De Bosschere and P. Tarau. Blackboard-based Extensions in Prolog. *Software — Practice and Experience*, 26(1):49–69, Jan. 1996.

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

[7] B. Demoen, G. Engels, and P. Tarau. Segment Preserving Copying Garbage Collection for WAM based Prolog. In *Proceedings of the 1996 ACM Symposium on Applied Computing*, pages 380–386, Philadelphia, Feb. 1996. ACM Press.

[8] 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.

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

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

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

[12] 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.

[13] P. Tarau. Ecological Memory Management 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.

[14] 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.

[15] 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.

[16] P. Tarau. An Efficient Specialization of the WAM for Continuation Passing Binary programs. In *Proceedings of the 1993 ILPS Conference*, Vancouver, Canada, 1993. MIT Press. poster.

[17] P. Tarau. Language Issues and Programming Techniques in BinProlog. In D. Sacca, editor, *Proceeding of the GULP'93 Conference*, Gizzeria Lido, Italy, June 1993.

[18] 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.

[19] P. Tarau. BinProlog 7.0 Professional Edition: Predicate Cross-Reference Guide . Technical report, BinNet Corp., 1998. Available from http://www.binnetcorp.com/BinProlog.

[20] P. Tarau. BinProlog 9.x Professional Edition: Advanced BinProlog Programming and Extensions Guide. Technical report, BinNet Corp., 2002. Available from http://www.binnetcorp.com/BinProlog.

[21] P. Tarau. BinProlog 9.x Professional Edition: BinProlog Interfaces Guide. Technical report, BinNet Corp., 2002. Available from http://www.binnetcorp.com/BinProlog.

[22] P. Tarau. BinProlog 9.x Professional Edition: User Guide. Technical report, BinNet Corp., 2002. Available from http://www.binnetcorp.com/BinProlog.

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

[24] P. Tarau, V. Dahl, and A. Fall. Backtrackable State with Linear Assumptions, Continuations and Hidden Accumulator Grammars. In *ILPS'95 Workshop on Visions for the Future of Logic Programming*, Portland, Oregon, Nov. 1995.

[25] P. Tarau, V. Dahl, and A. Fall. Backtrackable State with Linear Affine Implication and Assumption Grammars. In J. Jaffar and R. H. Yap, editors, *Concurrency and Parallelism, Programming, Networking, and Security*, Lecture Notes in Computer Science 1179, pages 53–64, Singapore, Dec. 1996. Springer.

[26] P. Tarau and K. De Bosschere. Memoing with Abstract Answers and Delphi Lemmas. In Y. Deville, editor, *Logic Program Synthesis and Transformation*, Springer-Verlag, pages 196–209, Louvain-la-Neuve, July 1993.

[27] P. Tarau and K. De Bosschere. Virtual World Brokerage with BinProlog and Netscape. In P. Tarau, A. Davison, K. De Bosschere, and M. Hermenegildo, editors, *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications*, JICSLP'96, Bonn, Sept. 1996. http://clement.info.umoncton.ca/ lpnet.

[28] P. Tarau, K. De Bosschere, and B. Demoen. Partial Translation: Towards a Portable and Efficient Prolog Implementation Technology. *Journal of Logic Programming*, 29(1–3):65–83, Nov. 1996.

[29] P. Tarau and B. Demoen. Language Embedding by Dual Compilation and State Mirroring. In M. Fromherz, A. Kusalik, and O. Nytro, editors, *Proceedings of 6-th Workshop on Logic Programming Environments, Santa Margherita Ligure, 1994*, pages 15–20, June 1994. also available as TR N-7034 Norges Tek. Hogscole, TRONDHEIM.

[30] P. Tarau, B. Demoen, and K. De Bosschere. The Power of Partial Translation: an Experiment with the C-ification of Binary Prolog. In K. George, J. Carrol, E. Deaton, D. Oppenheim, and J. Hightower, editors, *Proceedings of the 1995 ACM Symposium on Applied Computing*, pages 152–176, Nashville, Feb. 1995. ACM Press.

[31] P. Tarau and U. Neumerkel. A Novel Term Compression Scheme and Data Representation in the BinWAM. In M. Hermenegildo and J. Penjam, editors, *Proceedings of Programming Language Implementation and Logic Programming*, number 844 in Lecture Notes in Computer Science, pages 73–87. Springer, Sept. 1994.

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