THE POWER OF PARTIAL TRANSLATION: AN EXPERIMENT WITH THE C-IFICATION OF BINARY PROLOG

Paul Tarau

Bart Demoen

Koen De Bosschere

Université de Moncton

Katholieke Universiteit Leuven

Universiteit Gent

Abstract

We describe a new language translation framework (*partial translation*) and the implementation of one of its instances: the *C-ification* of Binary Prolog.

Partial C-ification is a translation framework which compiles sequences of emulator instructions down to native code (on top of a C compiler). In the case of logic programming languages, their complex control structure, some large instructions, and the management of the symbol table are left to the emulator while the native code chunks will deal with relatively long sequences of simple instructions.

The technique can be seen as an automatic *specialization* with respect to a given program of the traditional *instruction folding techniques* used to speed-up emulators.

When the target language of the host compiler is the same as the implementation language of the emulator (say C), the emulator, the representation of the byte code as a C data structure, some other C-ified library routines and hand-written C-code can all be compiled and linked together to a form a stand-alone application.

We give full details and performance analysis of the C-ification of a continuation passing Binary Prolog engine for which large write-mode sequences and the absence of a call-return mechanism make the framework particularly well suited.

The framework is shown as practical and easy to implement and it compares favorably with instruction folding techniques used traditionally to speedup emulators.

Keywords: *Programming language translation techniques, Compilation of binary Prolog, WAM, BinWAM, Prolog to C translation*

Introduction

Partial C-ification 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 also with pragmas for each predicate.

"Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission."

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. 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.

Partial C-ification

The main reason for translating Prolog to C are

1. gain of speed;
2. portability to multiple architectures;
3. stand alone executables.

One might expect full compilation to C to lead to code which is even better, but this not necessarily true due to

- modularity;
- code growth;
- difficult mapping between Prolog predicates and C-functions (e.g., backtracking);
- expensive (general) C-function calls.

For simpler languages like Janus [6], or KLIC [2] (a translation for KL/1 by Chikayama) full compilation has been proven quite practical. The major difference with the Prolog is that these languages do not feature backtracking, making the run-time system much simpler.

The solution we propose has the following features:

- the modularity problem is solved by the shared emulator;
- the code growth is limited by partial compilation;
- the semantic gap between Prolog and C is bridged by the emulator;
- doing less, not more: we compile relatively small sequences all inside one clause;
- cheap leaf-routines are used.

Figure 1: Basic idea of partial C-ification

Key Idea

The Figure 1 illustrates the principle of partial C-ification. By a straightforward modification of the emulator written in C we replace some contiguous sequences of interpreted WAM instructions by lightweight C-routines (*C-chunks*) which get called from a slightly differently compiled version of the emulator, linked together with them.

We can see this process as a partial evaluation of the known *instruction-folding* technique often used to speed-up emulators with respect to a known program. Instruction folding consists of creating a plethora of combinations of frequently occurring operation-code pairs or triplets with their associated C-code, all dispatched through a big switch statement and/or some other (for instance first-order label based) direct jumping techniques.

Applicability

Long PUT-sequences used to create heap-based continuations and the absence of call/return makes Binary Prolog (the subset of Prolog obtained through a transformation called binarization) particularly well suited for this framework. By this process, a Prolog clause like:

`a:-b,c,d.`

becomes

`a(Cont):-b(c(d(Cont))).`

We refer the reader to [11] for a more formal description of the binarization transformation and to [4] for its relationship to the Warren Abstract Machine (WAM) - the abstract engine used as a basis of most modern Prolog implementations.

With 'AND-continuations' reified as an extra argument of each goal, we manage to have only one call on the right side (*b/1* in the previous example). This allows the use of a simplified, yet powerful reduced instruction set WAM-like execution model (*BinWAM*), as described in [8, 9, 10, 12]. The main difference at code generation level between the WAM and the BinWAM is the call-return mechanism for the latter. The creation of continuations is well suited to be C-ified as binarized clause bodies have also the nice property, which will be illustrated in the following example, of generating *leaf-routines* i.e. small C-functions that will use no C-stack space.

Example

Let us follow the translation process on a typical example.

Prolog version

We have chosen the following clause containing some duplicate symbols and numbers:

```
treeSize(tree(Left,Right),S0,S) :-
    add(S0,1,S1),
    treeSize(Left,S1,S2),
    treeSize(Right,S2,S).
```

Binarized version

After binarization, our example becomes:

```
treeSize(tree(Left,Right),S0,S,C) :-
    add(S0,1,S1,
        treeSize(Left,S1,S2,
            treeSize(Right,S2,S,C))).
```

Hence, every clause gets an extra argument that carries the continuation, i.e., the part of the program that is still to be executed. Before `add/4` can be started, its continuation consisting of the nested `treeSize/4` terms is created on the heap.

It is precisely the creation of this continuation, and the preparation of the arguments for `add/4` that will be subject to C-ification.

Basic WAM-code

From this binary form we obtain the following WAM-code:

```
[1] clause_? treeSize 4
[2] firstarg_? tree/2 11
[3] get_structure tree/2 var(1)           % tree/2
[4] unify_variable var(5)                 % Left
[5] unify_variable var(6)                 % Right
[6] get_variable arg(2) var(1)            % S0
[7] c_chunk_begin 15 var(7)
[8] put_structure treeSize/4 var(8) % make continuation
[9] write_value var(5)                    % Left
[10] write_variable var(5)                 % S1
[11] write_variable var(9)                 % S2
[12] write_variable var(10)
[13] push_structure treeSize/4 var(10)
[14] write_value var(6)                   % Right
[15] write_value var(9)                    % S2
[16] write_value var(3)                    % S
[17] write_value var(4)                    % C
[18] put_constant arg(2) 1                 % 1
[19] put_value arg(3) var(5)               % S1
[20] put_value arg(4) var(8)               % move continuation
[21] c_chunk_end 15 var(7)                 % to its register
[22] execute_? add 4
```

Note the presence of `c_chunk_begin` and `c_chunk_end` in the WAM-code. The C-routine will be generated from the sequence of instructions between them.

C-ifying the emulator

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. The result will be a stand alone Prolog application (see Figure 2).

Figure 2: Preparing the byte code file to be linked by the C linker.

EMULATOR DATA STRUCTURE:

```
.....
{63,7,0,(void *)xx_399},
{6,0,4,"treesize"}, /* argument of xx_399 */
.....
```

All Prolog symbols are internalized by the emulator. The Prolog symbols that are needed by the C-chunk are put in the argument list of the C-call on a constant offset from the current P-pointer (here only one argument). So, the chunk can access them with a single load operation at no extra cost, and an optimizing compiler can take full advantage of this information and generate the most efficient code for a particular sequence. All this is similar to argument passing in threaded code [1].

There is no need for the C-chunk to directly access the Prolog symbol table. The final argument list of the C-call is built once by the loader of the emulator.

Generating the C-chunk

The C-chunk itself (a small static C-function `xx_399`) is generated from a sequence of C-macros.

```
c_chunk_variable(xx_399)
put_structure(0,0,8)      % treesize/4
write_value(1,5)          % Left
write_variable(2,5)       % S1
write_variable(3,9)       % S2
write_constant(4,0)       % treesize/4
write_value(5,6)          % Right
write_value(6,9)          % S2
write_value(7,3)          % S
write_value(8,4)          % C
put_integer(1,2)          % 1
move_reg(3,5)             % S1
move_reg(4,8)             % continuation
c_chunk_value(9,1)        % update heap pointer
```

Expanding the chunk to a function

The actual C function, after preprocessing by `cpp`, except for the low level `INPUT_INT` macro we have edited back for readability, is shown below.

```
term xx_399(H,regs,P)
  register term H,regs;
  register instr P;
{
```

```
regs[8] = (cell)(H+ 0); H[0] = P[0];
H[1] = regs[5];
regs[5] = H[2] = (cell)(H+2);
regs[9] = H[3] = (cell)(H+3);
H[4] = P[0];
H[5] = regs[6];
H[6] = regs[9];
H[7] = regs[3];
H[8] = regs[4];
regs[2] = INPUT_INT(1);
regs[3] = regs[5];
regs[4] = regs[8];
regs[-1] = (cell)(P+1); /*new code pointer */
return H+ 9 ;          /*new heap pointer */
}
```

Prolog symbols that are used twice by the C-chunk are only generated once as argument in the P-code section. Re-using arguments saves program code (improves locality), and speeds up the execution of the chunk by loading a common symbol only once from the argument list, and writing it as many times as needed onto the heap. The multiple assignment of C (`a=b=c`) is used to express that a particular term should be re-used.

BinProlog's tag-on-data representations makes the copying of a symbol from the code area onto the heap very efficient. Pushing a functor on the heap can be done as `H[Offset1]=P[Offset2]` with `Offset1` and `Offset2` known at compile time where `P[Offset2]` contains the full representation of the functor i.e. `<Arity,SymbolNumber,Tag>` which fits in one word.

The above code could be significantly improved by not using WAM-registers to store temporary data in the chunk, but by defining local C-variables for which the compiler might allocate hardware registers. The use of `regs[8]`, `regs[5]`, and `regs[9]` are typical examples.

Assembly code

It is interesting to take a look at the actual assembly (Sparc, Solaris 2.x, `gcc -O2`) listing, which shows clearly that our objective to have high quality code has been attained with minimal effort (H is in `%o0`, `regs` is in `%o1`, and `P` is in `%o2`).

EMULATOR DATA STRUCTURE:

```
....
.word xx_399
.byte 6
.byte 0
.half 4
.word .LLC993
...
```

C_CHUNK:

```
xx_399:
    st %o0,[%o1+32]
    ld [%o2],%g2
    st %g2,[%o0]
    ld [%o1+20],%g3
    .....
    ld [%o1+32],%g2
    st %g3,[%o1+12]
    retl
    st %g2,[%o1+16]
```

It can be seen that the mapping to a sequence of load-store instructions with precomputed offsets gives efficient code, which can be reorganized quite freely by super-scalar schedulers. Since the *c_chunks* do normally not contain calls to

other procedures or functions, they will get compiled into efficient *leaf routines* which use a limited subset of the hardware registers, so that save/restore of the caller's registers is avoided. Calls and returns are therefore jump instructions with the return address kept in a fixed register for use by the specialized `retl` instruction.

Built-ins and anti-calls

We have shown in the previous example the C-ification of PUT sequences. This has been extended to frequently used inline operations which can be processed in Binary Prolog before calling the 'real goal' in the body as described in [4]. Chunks containing small built-ins that do not require a procedure call will still generate leaf-routines.

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 corouting 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 corouting (jumping back and forth) between native and emulated code. Anti-calls can be implemented with the direct-jump technique used in WAMCC [5] 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.

Discussion

An alternative to our approach which left symbol table management to the emulator, is to compile the symbol table down to unique C-objects that are given unique addresses by the C-compiler and linker. Given the fact that these objects will have full word-wide addresses, and given the penalty to load immediates of word length for RISC architectures, this scheme does not give a speedup (4 memory accesses for both approaches) and is therefore not worth considering. Hence, it turns out that the emulator is a good tool for sharing symbol-tables between multiple C-modules, and that compiling the symbol table down to C is not a good idea. Obviously, integers are passed as immediate operands. Partial C-ification becomes very interesting in the presence of

- partial evaluation which creates long bodies, i.e. it ensures a better hit on long sequences;
- term compression [12] which reduces and simplifies the data movements in the chunks;
- a top-down compilation scheme, which takes advantage of free indirect addressing with small offsets pre-computed at compile-time;

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 3: 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).

- optimal register allocation as some pseudo-registers in the emulator become real ones in C.

Performance evaluation

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 3). 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 2.3) 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.

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

Related work

Simpler logic languages like Janus [6], KLIC [2] have been successfully compiled to C with speed which competes successfully with native code compilers.

On the other hand, most of the attempts for the C-translation of full Prolog have managed to get performance only slightly better or comparable to well-engineered emulators like Quintus or SICStus Prolog, while paying the price of very large code-size or machine-specific non-portability.

In [7] the Prolog to C compiler is built on top of a traditional WAM compiler. The WAM instructions are expanded inline or they become function calls. The call sequence of the predicates is controlled by a dispatching loop.

A smart (but machine and gcc specific) technique is used in the WAMCC system based on [5] to avoid the dispatching loop by implementing global labels and direct jumps inside C-functions, by using `asm` directives. Speed is in the middle between emulated and native code for most programs, exceptionally good on arithmetics (with data validity checking off)

although on some programs like `naive reverse` the overall performances (258 KLIPS) are significantly lower than emulated BinProlog (450KILPS).

The approach described here is significantly different from previous work. We have started with the clear objective of *partial translation* from one of the fastest existing C-emulated Prologs (BinProlog) and to obtain overall performances similar to the best full-translation schemes. Therefore, in practice, our method (which needed about 1 programmer-week of work for minimal modifications to the emulator) looks highly practical and general enough to be applied to other emulator based implementations of high level languages.

Moreover our scheme is by its nature fine-tunable in terms of the amount of translation, giving to the programmer the opportunity to configure it for a large spectrum of code-size/speed ratios.

Future Work

Modules

A module system for C-translated Prolog is proposed in [3]. This can be applied quite naturally to our translation mechanism as the mapping to C described in [3] is also based on binary prolog. Modularity is easily obtained in our case by simply making emulated code to support modular compilation. Modules are linked together in a *project* data structure accessible from the emulator at compile-time. Each module consists of an array of emulator instructions and a set of C-chunks defined as local functions (i.e. declared *static* in C).

Small self-recursive predicates

Small self-recursive predicates (like `append/3`) are a good target for full C-ification for the deterministic case while the rarely used non-deterministic case can be left to the emulator.

Applying this for the built-in `append/3` of BinProlog gave performances about 2 times faster than native SICStus Prolog 2.1.9 on the *naive reverse benchmark*. Basically, a self-recursive predicate is transformed to a while loop which advances on its known input data until non-determinism or non-applicability of the self-recursive clause is detected, when through continuation manipulation the function escapes back to the emulator.

Conclusion

We have presented a new technique called 'partial translation' and studied its use for Prolog to C translation. The technique has allowed to control the amount of translation to C for an optimal speed/code-size ratio. Although our experiments have been described in the context of Prolog to C translation, the technique itself is general purpose. The technique gives performance in a range between emulated and native code with very little implementation effort and ensures portability through C.

Acknowledgements

Paul Tarau (tarau@info.umoncton.ca) thanks for support from NSERC (grant OGP0107411), from the FESR of the Université de Moncton, and from K.U.Leuven through Fellowship F/93/36. Bart Demoen (bimbart@cs.kuleuven.ac.be) thanks the Belgian Ministry (DWTC) for support through project IT/IF/4. Koen De Bosschere (kdb@elis.rug.ac.be) is

senior research assistant with the Belgian National Fund for Scientific Research.

The code shown in this paper has been generated with BinProlog 3.00, available together with the benchmarks and some related papers by ftp from clement.info.umoncton.ca.

References

- [1] J. Bell. Threaded code. *Communications of ACM*, 16:370–372, June 1973.
- [2] T. Chikayama, T. Fujise, and D. Sekita. A portable and efficient implementation of `kl1`. In *Proceedings of Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science. Springer, Sept. 1994.
- [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 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.
- [5] D. Diaz and P. Codognet. A minimal extension of the WAM for `clp(FD)`. In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 774–790, Budapest, Hungary, 1993. The MIT Press.
- [6] D. Gudeman, K. De Bosschere, and S. Debray. `jc`: An Efficient and Portable Sequential Implementation of Janus. In K. Apt, editor, *Joint International Conference and Symposium on Logic Programming*, pages 399–413, Washington, Nov. 1992. MIT press.
- [7] R. Horspool and M. Levy. Translator-based multi-paradigm programming. *Journal of Software and Systems*. to appear.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] 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.