A comparison of some schemes for translating logic to C.

Bart Demoen Greet Maris
Department of Computer Science, K.U.Leuven
Celestijnenlaan 200 A
B-3001 Leuven
Belgium
{bimbart, greet}@cs.kuleuven.ac.be

Abstract

The general improvement of C compilers, and some new non standard features of gcc have made it more attractive to compile (logic) to C: it is no longer unthinkable that the speed of a native code optimizer can be matched and even beaten by a scheme that compiles to C and lets most of the hard work be done by the C compiler. The new features, especially gcc's treatment of labels as first class types, are a clear invitation to abandon native code generators. Also, the possibility to assign hardware registers to global variables looks attractive at first sight. Three schemes for exploiting a C compiler in the compilation of different languages, are examined and their effectiveness is measured. The conclusion is that it is probably better not to make use of all the features gcc offers. We also show that if C compilers were a bit smarter about compiling the C switch statement, compliance to standard C would become even more attractive. The measurements were done on different risc platforms and the conclusions seem to hold generally.

1. Introduction

Writing a native code generator is a big task, which requires intimate knowledge of the target machine. It is also a moving target since machines succeed each other quickly and even within the same line of machines or processors, the characteristics can change dramatically. On the other hand, parts of a native code compiler are just tedious work, like the register allocator, the peep hole optimizer etc. It is appealing to have all this work done by someone else and reuse that work. Compiling to C instead of to native code, is such an attempt to reuse the effort by others. This approach is worth considering for any language, as C leaves lots of possibilities for abusing the type system and because it has in general the flavor of a low level language. The advantages that compiling to C offers, are otherwise clear: portability, less knowledge about low level machine required and linking with other C programs becomes quite easily. The same argument could apply to other languages instead of C (be it ADA or ALGOL), if efficient implementations were widely available.

Although compiling to C is attractive, a straightforward mapping of a logic language to C will give bad performance. This is due the differences in control and data characteristics between logic and C. A logic program consists mainly of small recursive predicates because recursion is the only way to express iteration in logic. Moreover, global data is used to represent bindings, terms and pass arguments to procedures.

In the recent past, three languages have been implemented by compiling them to C first: Janus [3], Erlang [2] and KL1 [4]. They all three belong to the class of logic languages -

although Erlang could as well be thought of as a functional language, but it does share some characteristics with logic.

These three languages use a different basic compilation scheme and each is based on different characteristics of the C compiler. We will not go into the details of these languages, but since all three compilation schemes can be applied to (deterministic) Prolog easily, we will use Prolog as a vehicle to explain the compilation schemes and to describe the benchmarks that were used for testing and comparing these schemes.

Since 1983, Prolog has often been implemented starting from the WAM [10] for which either an emulator, threaded code interpreter or a native code generator has been written. Even implementations that do not adhere directly to the WAM instruction set, should often be classified as belonging to the WAM family, like [11, 12] and many implementations of languages that differ substantially from Prolog, are based on WAM [1]. So, the choice of Prolog as the red thread through the paper is by no means an unfortunate accident or a serious limitation, as the implementation of Prolog has modelled the implementation of many other languages.

In gcc [6] there are two features that are of particular interest to compiler writers, but which have the disadvantage (at the moment) of being non-standard [7]: in gcc, there is a label type and a goto statement can refer to a variable of the label type; secondly, a register declaration can be given for global variables, i.e. variables that are visible over function boundaries. For the compiler writer to C, both look attractive. And both have been exploited by the Erlang system. On the other hand, the Janus and KL1 scheme for translating to C, does not necessarily use non-standard features. Our measurements show that a certain amount of non-standardness is good for efficiency, but too much is bad. As a matter of fact, it turns out that the compilation scheme adopted in Erlang performs worse than the Janus schema; moreover, it restricts the possibilities of the C compiler, so that it seems counterproductive to use global labels.

We started this comparison because we were at the point of implementing AKL [5] using the compilation to C technique: since the three schemes presented here existed already, we wanted to make a conscious choice. We have also deliberately chosen goal stacking, because we believe that this offers more implementation freedom when implementing a parallel system. Still, we have also done a test with environment stacking as in WAM and found that the same conclusions hold.

In the next section, the three compilation schemes are presented in their pure and extreme form and exemplified on the Prolog predicate naive reverse: this predicate has been used a lot in the past for assessing the speed of Prolog systems. However, as for Prolog, naive reverse is a bad benchmark and we will make some slight changes to it, in order to show the point more clearly.

Section 2 describes the three compilation schemes. Section 3 describes the benchmarks at the Prolog level. Section 4 discusses the results of the measurements and the conclusions to be drawn from them. Section 5 goes into some detail about the compilation of the C switch statement. Section 6 reviews shortly other compilation to C schemes for Prolog. General conclusions are given in section 7. The appendices contain the results of some additional measurements and the full C code for a representative sample of the benchmark set.

2. The three compilation schemes

We introduce the three compilation schemes mainly by an example: C code for the clause a :- b,c,d . is given and if necessary also for the fact b. The full C code for some of the benchmarks as used for the timings, can be found in the appendices. The complete benchmark

set is obtainable from the authors.

We will not go into the details of the languages Janus [3], Erlang [2] and KL1 [4]: the reader is referred to the appropriate literature.

2. 1. Janus

The basic approach in Janus [3] is to compile a complete program to one C function which is essentially one switch statement of which the case labels represent entries of predicates and to which jump labels are added so that the call to the first goal in a body, doesn't have to pass by the switch: the switch is used when the program is activated through its top query, and by the proceed instruction which pops a case label from the goal stack and goes to the top of the switch statement. Then the one function for Janus is essentially:

```
jump_switch:
        switch(*--goalstacktop)
        { case a :
          jump_a :
                        *goalstacktop++ = d ;
                        *goalstacktop++ = c ;
                        goto jump_b ;

          case b :
          jump_b:
                        goto  jump_switch;
        ...
        }
```

The above code is ANSI C. One can use the gcc jump label variables so that jump labels are pushed on the goal stack. The resulting code is then:

```
          jump_a :
                        *goalstacktop++ = &&jump_d ;
                        *goalstacktop++ = &&jump_c ;
                        goto jump_b ;

          jump_b :      goto  *(*(--goalstacktop));
        ...
```

The switch statement has disappeared. The two versions of this scheme - without and with jump labels on the goal stack - will be referred to as *jc* and *jclab*. Note however that the use of jump labels in *jclab*, concerns **local** labels.

2. 2. Erlang

Erlang [2] uses both global labels and global registers. The basic scheme consists in translating each module to one C function, which has a prologue that fills in a global jump table (gjt) with the jump label needed for calling a predicate:

```
void *f_a()
{/*prologue*/
  gjt[a] = &&jump_a ;
  return ;
jump_a:
 *goalstacktop++ = gjt[d] ;
 *goalstacktop++ = gjt[c] ;
  goto *gjt[b] ;
} /*f_a*/
```

The idea is that when the system is initialized, every function corresponding to a module is called in the usual way, so that the global jump table (gjt) is filled in. An Erlang predicate is called by directly jumping into the C function. The aim is to avoid subroutine calls: as such, a subroutine call is not more expensive than a goto on most modern architectures, but one must avoid register window overflows which are rather costly. (see also section 6) We have two versions of this scheme: one which uses only the global labels and one which also uses global registers for the top of the goal stack, the argument stack and the begin address of the global jump table. We refer to these two versions as *erl* and *erlreg*. Neither is ANSI C.

2. 3. KLIC

The C implementation of KL1 [4] is called KLIC: a single module is compiled to a C function exactly like in the Janus scheme. Switching between modules requires the help of a driver which dispatches between modules. If there is only one module, the schema reduces to the Janus schema, so we have taken the extreme position that every predicate belongs to a different module so that the effect of the driver becomes more clear. Again for the same example, we obtain:

```
void *f_a()
{*goalstacktop++ = f_d ;
  *goalstacktop++ = f_c ;
  *goalstacktop++ = f_b ;
  return ;
} /*f_a*/
```

and the driver essentially becomes:

```
void *driver()
{ while (1)
        (*--goalstacktop)() ;
} /*driver*/
```

The KLIC scheme is ANSI C compliant. We have used 3 variants in the benchmarks: one which uses local registers to keep the goal stack top, argument stack top and the cached goal stack top contents (see 2.4 for more explanation on this); this version is still ANSI C compliant and is named *kliclocreg*; the same items can be assigned to global registers: *klicreg*; or these three items can be kept in plain global variables: *klic*.

2. 4. Some remarks about the final C code

Since we wanted to squeeze as much out of each compilation scheme as reasonable, we have performed a number of optimizations that are sometimes scheme specific:

- for KLIC, it pays to keep the contents of the goal stack top in a separate variable (be it a register or not): we refer to this as the cached goal stack top contents
- for KLIC and Erlang: if the first call in the clause is to the same predicate (like in a :- a , b .) then a local jump is performed instead of through the driver or the global jump table (note that Janus can always perform such a local jump)
- in Janus, it makes no sense to assign registers to some of the data structures, as they are local to the one function and the compiler assigns them to registers anyway
- we wanted to make sure that the switch statement was compiled to a computed goto, instead of to a cascade of if-then-else: this was achieved by adding enough meaningless cases to the switches whenever necessary

3. The benchmarks measured

Naive reverse has often been used to measure the performance of Prolog systems:

```
nrev([],[]) .
nrev([X|R],O) :- nrev(R,A) , append(A,[X],O) .

append([],L,L) .
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3) .
```

Although it is a rather perverted benchmark, it was the first program we used as a benchmark for the 3 compilation schemes. Since we were interested in measuring the effect on the flow of control only (as the data representation is an orthogonal issue), the actually used version of nrev, where N is assumed to be greater or equal to zero, was

```
nrev(N) :- N = 0 -> true ; nrev(N-1) , append(N-1) .
append(N) :- N = 0 -> true ; append(N-1) .
```

and the initial query was ?- nrev(1000) . and the arithmetic expressions as arguments of body goals should be understood as being evaluated. In this way, the same number of logical inferences (or successful predicate calls) is achieved.

Since the nrev benchmark is dominated completely by append, we decided that a goal should be added in the body of append: a call to a predicate that - in terms of the WAM - performs only a proceed:

```
append(N) :- N = 0 -> true ; append(N-1) , dummy .
dummy.
```

Since we used goal stacking for the implementation of the continuation, the different compilation schemes benefit to different extents from this left recursion; in decreasing order of benefit:

- in KLIC, it makes returning to the driver unnecessary

• in Erlang, it allows for a direct (local) goto, instead of an indirect one
• in Janus, only the (code) cache comes into play (as they do with the other schemes!)

So, we thought it was only fair to measure both a left recursive and a tail recursive version of append, i.e., one with the dummy call added as above, and one as in:

append(N) :- N = 0 -> true ; dummy , append(N-1) .

We will refer to the benchmarks by LEFT and RIGHT - depending on whether the dummy was added to the left or the right of the append goal.

4. The timings and the conclusions.

program	LEFT	RIGHT
klic	2130	1120
kliclocreg	2060	890
klicreg	990	500
erl	1650	910
erlreg	840	480
jc	980	690
jclab	720	460

Table 1: timings in ms on SPARC1

Table I shows the timings for the left and right versions of the benchmarks on a SPARC station 1. All programs were compiled with gcc -O2 (version 2.5.7). The time was measured with getrusage. The horizontal difference between left and right is not relevant; vertically, they show the same pattern: *jc* is faster than *erlang* and *erlang* is faster than *klic*. Moreover, being less standard within a scheme, improves efficiency. There is however a small surprise in the table: since *erlreg* uses global labels and global registers, whereas *jclab* uses only local labels in variables, *erlreg* is less standard than *jcla*b. Still, *jclab* performs better than *erlreg*. It shows that if one is willing to leave ANSI C, *jclab* is the best choice. *jclab* has two more advantages over *erlreg* which are not shown in our benchmarks, but it is clear that:

• the use of non local labels, prevents the safe use of local variables in the functions: this means that in the Erlang schema, other items (like heap pointer, intermediate variables for construction of arguments ...) must be global; most probably, there will be more such items than available registers, so that Erlang ends up with some data structures that cannot be kept in registers, while in Janus, they can be local to a case entry and be in registers
• the assignment of some items to global registers, removes these registers from the pool of available registers to assign to other variables; this is potentially harmful for runtime procedures as well (e.g. general unify)

In the appendix, similar tables for other architectures are given: they lead to the same conclusions.

As mentioned in the introduction, we have also done tests on environment stacking. The main difference in execution speed between goal stacking and environment stacking depends on the amount of module switching. When two consecutive goals in the same clause body, belong to different modules, like in:

```
module_0:a :- module_1:b , module_2:c , module_0:d .
```

then for environment stacking four returns to the driver are needed: to module_1 for b, to module_0 for the continuation, to module_2 for c, to module_0 for the continuation. For goal stacking only three returns to the driver are executed. And if module_1 = module_2 goal stacking only needs two returns, where environment stacking still needs four. Even when there is only one module (i.e. one C function), environment stacking is slightly slower, for two reasons: argument manipulation for permanent variables and the fact that a proceed cannot directly go to the next predicate, since it has to pass by the environment of its parent goal.

5. Improving on the compilation of the switch statement

Both gcc and the SUN C compiler could do better in the compilation of a switch statement that is compiled to an indirect jump with a switch table.

- a register can be assigned to the starting address of the switch table;
  gain : 2 instructions
- range checking is most often unnecessary; gain: 2 instructions (*)
- shifting the switch variable can be avoided: 1 instruction (**)

(*) range checking is clearly unnecessary in code like:

```
switch (0x3 & i)
{ case 0 : ... ; break ;
  case 1 : ... ; break ;
  case 2 : ... ; break ;
  case 3 : ... ; break ;
}
```

but C compilers don’t see this. Moreover, it still costs the masking operation to make clear that range checking could be ommitted safely; instead, it would be nice to have a range check omission option.

(**) The switch value cannot be used directly as an index in the switch table because the size of a switch table entry is 4 bytes. This is unnecessary if the values of the labels are all multiples of 4 and starting with 0. Of course, this can only work if the range check is provably (or declared) redundant.

To summarize: the usual code generated for a switch statement is:

```
cmp %o1,7               ! %o1 contains the switch value in the range [0,7]
bgu L2                  ! L2 = default label
sethi %hi(L11),%o0      ! %o0 will contain the starting address
or %o0,%lo(L11),%o0     ! of the switch table
```

```
sll %o1,2,%o1
ld [%o1+%o0],%o0
jmp %o0
```

while the following is achievable:

```
ld [%o1+%o0],%o0
jmp %o0
```

The above optimizations were performed by hand on the code generated for the *jc* tests. On average, *jc* with optimized switch, gives the same result as *erlreg*. Since C compilers will certainly become better, the optimizations might be included in the future, so that the preference for the (ANSI C) Janus compilation scheme, becomes stronger.

	LEFT	RIGHT
jc ordinary switch	980/190	690/160
jc hand optimized switch	810/165	500/135
jclab	720/140	460/120

Table 2: comparing jc with and without optimized switch (Sparc1/Sparc10)

6. Other schemes

There have been other attempts to compile Prolog to C; we mention only the recent [8] which differs mainly from *jc* in that it starts from a binarized program (see [11]).

In [9], the Prolog to C compiler is built on top of a traditional WAM compiler. The WAM instructions are expanded in-line or they become function calls. The call sequence of the predicates is controlled by a dispatching loop.

One of the main decisions one has to make when compiling Prolog to C, is how to map the execution of Prolog to the runtime stack of C: it is tempting to map e.g. the WAM environment stack to the C stack. This is however a mistake for several reasons:

- the WAM environment stack is not a proper stack as soon as backtracking comes into the picture: this imposes some painful and inefficient work arounds
- it is difficult for a garbage collector to find active environment variables in older register windows
- most important however: the penalty of window overflows is very high; in [12], it is reported that Aquarius is faster for the tak benchmark than C; this is due to the fact that in Aquarius no register window overflows occur during execution of the benchmark: Aquarius does indeed not map the environments to the C stack

So without provisions to avoid register window overflows, mapping the environment stack to the C stack is a bad idea.

Also the choicepoint stack could be mapped to the C stack: this will pose problems for implementing a garbage collector and cut.

7. General conclusions

As C compilers become always better, compiling a new language to C seems a fast and portable way to arrive at an implementation with a high efficiency. One must however choose carefully the translation scheme, not only from the point of view of efficiency, but also taking portability into account. Thereby, one might be inclined to give up some portability (after all, gcc compliance is almost as good as portable) for speed, but it turns out that staying ANSI compliant gives almost optimal performance and that for sure, some of the newest features of gcc are damaging performance. From the above measurements, we can conclude that the Janus scheme, is to be preferred over any other scheme, whether one wants to remain ANSI compliant or not. The disadvantage of Janus is that large programs, when translated to a single C function, cannot be compiled in reasonable time by an optimizing C compiler: a program can be divided arbitrarily into pieces similar to the modules of KLIC. Program analysis - in fact a call dependency analysis seems enough - can decide how to divide the program in an optimal way. So, as a conclusion, the KLIC approach seems to be the optimal one when the Janus function is made as large as possible.

Acknowledgments

Greet Maris thanks SICS for hospitality during a one month stay there, and in particular Sverker Janson, Ralph Haygood, Per Brand and Johan Montelius. She also thanks Bogdan Hausman of Ellemtel Telecommunications Systems Laboratories, Sweden for explanations on the implementation of Erlang. This work was sponsored by DPWB contract IT/4 and Esprit contract ACCLAIM. We also thank Patrick Weemeeuw and Philippe Bekaert for help with some of the gcc compilers.

References

[1] S. Janson and S. Haridi, ‘Programming Paradigms of the Andorra Kernel Language’, in *Proceedings of The International Logic Programming Symposium*, ed. V. Saraswat, The MIT Press, Cambridge, Massachusetts, 1991.

[2] B. Hausman. ‘Turbo Erlang: approaching the Speed of C’, in *Implementations of Logic Programming Systems*, ed. Evan Tick, Kluwer 1994 (to appear)

[3] D. Gudeman, K. De Bosschere and S. Debray, ‘jc: An Efficient and Portable Sequential Implementation of Janus’, in *Proceedings of the Joint International Conference on and Symposium on Logic Programming 1992*, pp 399 - 413, MIT Press, 1992.

[4] T. Chikayama, T. Fujise and D. Sekita, ‘A Portable and Efficient Implementation of KL1’, in *ICOT/NSF Workshop on Parallell Logic Programming and its Programming Environments*, CIS-TR-94-04, Department of Computer and Information Science, University of Oregon, March 1994.

[5] R. Moolenaar and B. Demoen, ‘A parallell implementation for AKL’, in *Programming Language Implementation and Logic Programming: PLILP ’93*, Tallinn, Estonia, 1993.

[6] R. M. Stallman, Using and Porting GNU CC, manual.

[7] B.W. Kernighan and D.M. Ritchie, 'The C Programming Language -2nd Edition', Prentice Hall, 1988, ISBN 0-13-110632-8.

[8] K. De Bosschere and P. Tarau, 'A continuation based Prolog-to-C mapping', in *Proceedings of the ACM Symposium on Applied Computing (SAC '94)*, Phoenix, Arizona, March, 1994

[9] M.R. Levy and R.N. Horspool, 'Translating Prolog to C: a WAM-based approach', in *Proceedings of the Second Compulog Network Area Meeting on Programming Languages, and the workshop on Logic Languages* in Pisa, Italy, May 1993.

[10] D. H. D. Warren, 'An abstract Prolog instruction set', Technical Report no 309, p.30, SRI International, Menlo Park, 1983.

[11] P. Tarau and M. Boyer, 'Elementary Logic Programs', in *Proceedings of PLILP'90*, pp. 159-173, (eds) P. Deransart & J. Maluszynski, Springer-Verlag 1990

[12] P. Van Roy, 'Can Logic Programming Execute as Fast as Imperative Programming ?' Report No. UCB/CSD 90/600 Dec 1990, Berkeley, California 94720

Appendix A: more RISC architectures

	SPARC 10	SPARC 10	MIPS 3000	MIPS 3000	DEC Alpha	DEC Alpha	HP	HP
	left	right	left	right	left	right	left	right
klic	410	230	632	367	256	148	1070	560
kllocreg	420	200	621	316	232	122	1010	510
klicreg	260	130	343	207	148	76	820	380
erl	260	160	437	285	155	94	410	230
erlreg	190	120	210	164	116	67	220	100
jc	190	160	281	242	117	79	190	150
jclab	140	120	203	167	85	62	170	110

Table 3: some other RISC architectures

The above figures show clearly that the conclusion holds on a wide range of risc processors. It should be stressed that only comparisons within one column are intended to be meaningfull: comparison of processors should not be based on the above table.

Appendix B: klic.c: left

```
void *rev()
{ int i ;

  i = *--argp ;
label_rev:

  if (i--) { *argp++ = i ; *goalstacktop++ = ap ;
             goto label_rev ;
            }
  else cachedgoal = *--goalstacktop ;
} /*rev*/

void *dummy()
{
  cachedgoal = *--goalstacktop ;
} /*dummy*/

void *ap()
{ int i ;

  i = *--argp ;
label_ap:

  if (i--) { *argp++ = i ; *goalstacktop++ = ap ;
             cachedgoal = dummy ;
             return ;
            }
  else cachedgoal = *--goalstacktop ;
} /*ap*/

void *ex()
{       exit(0) ; }

main()
{
        goalstacktop = goalstack ;
        argp = args ;
        *goalstacktop++ = ex ;
        cachedgoal = rev ;
        *argp++ = 1000 ;
label:
        (cachedgoal)() ;
        goto label ;
} /*main*/
```

Appendix C: jclab: left

```
static one_funct()
{ int i ;

 void * goalstack[100000] ;
 void * *goalstacktop ;

 int args[100000] ;
 int *argp ;

        goalstacktop = goalstack ;
        argp = args ;
        *argp++ = 1000 ;
        *goalstacktop++ = &&label_ex ;
        *goalstacktop++ = &&label_rev ;

label_pred :

        goto *(*(--goalstacktop)) ;
          label_ap :
                i = *--argp ;
                label_ap1 :
                if (i--) { *argp++ = i ; *goalstacktop++ = &&label_ap ;
                           goto label_dum ;
                         }
                goto label_pred ;

          label_rev :
                i = *--argp ;
                label_rev1 :
                if (i--) { *argp++ = i ; *goalstacktop++ = &&label_ap ;
                           goto label_rev1 ;
                         }
                goto label_pred ;
          label_dum:
                goto label_pred ;
          label_ex :exit(0) ;

} /*one_funct*/
```

Appendix D: erlreg.c: left

```
typedef void * function;
static function goalstack[100000] ;
register function *goalstacktop asm("%g3") ;
static function jct[1000] ;
register function *jctreg asm("%g5") ;
static int args[100000] ;
register int *argp asm("%g4") ;

void *rev()
{
 jctreg[revi] = &&L_rev ;
 return ;
L_rev:
 {int i ;
 i = *--argp ;
label_rev:
 if (i--) { *argp++ = i ;
             *goalstacktop++ = jctreg[api] ;
             goto label_rev ;
            }
 else { goto *(*(--goalstacktop)) ; }
 }
} /*rev*/

void *ap()
{
 jctreg[api] = &&L_ap ;
 return ;
L_ap:
 {int i ;
 i = *--argp ;
label_ap:
 if (i--) { *argp++ = i ;
             *goalstacktop++ = jctreg[api] ;
             goto *(jctreg[dumi]) ;
            }
 else { goto *(*(--goalstacktop)) ; }
 }
} /*ap*/

void *dummy()
{
 jctreg[dumi] = &&L_dummy ;
 return ;
L_dummy:
        goto *(*(--goalstacktop)) ;
}

void *ex()
{
 jctreg[exi] = &&L_ex ;
 return ;

L_ex:

 exit(0) ;
} /*ex*/

main()
{
        jctreg = jct ;
        ex();
        rev();
        dummy();
        ap();

        goalstacktop = goalstack ;
        argp = args ;
        *goalstacktop++ = jctreg[exi] ;
        *goalstacktop++ = jctreg[revi] ;
        *argp++ = 1000 ;

        goto *(*(--goalstacktop)) ;
} /*main*/
```