Improving the Compilation of Prolog to C Using Type and Determinism Information: Preliminary Results*

J. Morales† jfran@clip.dia.fi.upm.es

M. Carro† mcarro@fi.upm.es

M. Hermenegildo† ‡ herme@fi.upm.es

Abstract

We describe the current status of and provide preliminary performance results for a compiler of Prolog to C. The compiler is novel in that it is designed to accept different kinds of high-level information (typically obtained via an analysis of the initial Prolog program and expressed in a standardized language of assertions) and use this information to optimize the resulting C code, which is then further processed by an off-the-shelf C compiler. The basic translation process used essentially mimics an unfolding of a C-coded bytecode emulator with respect to the particular bytecode corresponding to the Prolog program. Optimizations are then applied to this unfolded program. This is facilitated by a more flexible design of the bytecode instructions and their lower-level components. This approach allows reusing a sizable amount of the machinery of the bytecode emulator: ancillary pieces of C code, data definitions, memory management routines and areas, etc., as well as mixing bytecode emulated code with natively compiled code in a relatively straightforward way. We report on the performance of programs compiled by the current version of the system, both with and without analysis information.

1 Introduction

Several techniques for implementing Prolog have been devised since the original interpreter developed by Colmerauer and Roussel [5], many of them aimed at achieving more speed. An excellent survey of a significant part of this work can be found in [26]. The following is a rough classification of implementation techniques for Prolog (which is, in fact, extensible to many other languages):

- Interpreters (such as C-Prolog [16] and others), where a slight preprocessing or translation might be done before program execution, but the bulk of the work is done at runtime by the interpreter.
- Compilers to *bytecode* and their interpreters (often called emulators). The compiler produces relatively low level code in a special-purpose language. An interpreter of such low-level code is still required. Most emulators are currently based on the Warren Abstract Machine (WAM) [28, 1], but other proposals exist [24, 13].
- Compilers to a lower-level language, typically ("native") machine code. In this case the output requires little or no additional support to be executed. One solution is for the compiler to generate machine code directly. Examples of this are the Aquarius system [27], versions of the SICStus Prolog [22] compiler for some architectures, the BIM-Prolog compiler [14], and the Gnu Prolog compiler [7]. Another alternative is to generate code in a (lower-level) language, such as, e.g., C-- [12] or C, for which a machine code compiler is readily available; the latter is the approach taken by `wamcc` [4].

*This work is partially supported by Spanish MCYT Project TIC 2002-0055 *CUBICO*, and EU IST Projects 2001-34717 *Amos* and IST-2001-38059 *ASAP*, and by the Prince of Asturias Chair in Information Science and Technology at the University of New Mexico. J. Morales is also supported by an MCYT fellowship co-financed by the European Social Fund.

†Technical University of Madrid, Boadilla del Monte, E-28660, Spain.

‡Departments of Computer Science and Electrical and Computer Engineering, U. of New Mexico (UNM).

Each solution has its advantages and disadvantages:

Executable performance vs. executable size and compilation speed: Compilation to lower-level code can achieve faster programs by eliminating interpretation overhead and performing lower-level optimizations. In general, performing as much work as possible at compile time in order to avoid run-time overhead brings faster execution speed at the expense of using more resources during the compilation phase and possibly producing larger executables. In general, compilers are much more complex and take longer to preprocess programs for execution than interpreters. This difference gets larger as more sophisticated forms of code analysis are performed as part of the compilation process. This can impact development time, although sophisticated analyses can be turned off during development and applied when only generating production code. Interpreters in turn have potentially smaller load/compilation times and are often a good solution due to their simplicity when speed is not a priority. Emulators occupy an intermediate point in complexity and cost. Highly optimized emulators [19, 20, 6, 22, 2] offer very good performance and reduced program size (since single bytecode instructions correspond to several machine code instructions), which may be a crucial issue for very large programs and symbolic data sets.

Portability: Interpreters offer portability in a straightforward way since executing the same Prolog code in different architectures boils down (in principle) to simply recompiling the interpreter. Emulators usually retain the portability of interpreters, since only the emulator has to be recompiled for every target architecture (bytecode is usually architecture-independent), unless of course they are written in machine code.[1] Compilers to native code require architecture-dependent back-ends, i.e., a new translation into machine code has to be developed for each architecture. This typically makes porting and maintaining these compilers a non-trivial task. The task of developing these back-ends can be simplified by using an intermediate RTL-level code [7], although still different translations of this code are needed for different architectures.

Opportunities for optimizations: Code optimization can applied at all levels: to the Prolog level itself [18, 29], to WAM code [8], to lower-level code [15], and/or to native code [27, 23]. At a higher language level it is typically possible to perform more global and structural optimizations, which are then implicitly carried over onto lower levels. On the other hand, additional, lower-level optimizations can be introduced as we approach the native code level. These optimizations require exposing a level of detail in the operations that is not normally visible at higher levels. One of the most important motivations for compiling to machine code is precisely to be able to perform these low-level optimizations. In fact, recent performance evaluations show that well-tuned emulator-based Prolog systems can beat, at least in some cases, Prolog compilers which generate machine code directly but do not perform extensive optimization [7]. The approach of translating to a low-level language such as C is interesting because it makes portability straightforward, as C compilers exist for most architectures, and, on the other hand, C is low-level enough that it allows expressing in it a large class of low-level optimizations which will make into the final executable code in a form known beforehand, and which go beyond what can be expressed solely by means of Prolog-to-Prolog transformations.

Given all the considerations above, it is safe to say that different approaches are useful in different situations and perhaps even for different parts of the same program. In particular, the emulator approach with its competitive compilation times, program size, and performance, can be very useful during development, and in any case for non-performance bound portions of large symbolic data sets and programs. On the other hand, in order to generate the highest performance code it seems appropriate to perform optimizations at all levels and to eventually translate to machine code so that even the lowest-level optimizations can be performed, at least for parts of the program. The selection of a low-level language such as C as an intermediate target can offer a good compromise between opportunity for optimization and portability for native code.

In our compiler we have taken precisely such an approach: we use translation to C during which we apply several optimizations, making use of high-level information. Our starting point is the standard version of the Ciao Prolog system [2]. This is essentially an emulator-based system of quite competitive performance. Its abstract machine is an evolution of the &-Prolog abstract machine [10], itself a separate evolution branch from early versions (0.5–0.7) of the SICStus abstract machine. We have developed a compiler from Prolog to native code, via an intermediate translation to C, where the translation scheme adopts the same scheme for memory areas, data tagging, etc. as the emulator. This facilitates mixing emulated and native code (as done also by SICStus) and also has the important practical advantage that many complex and already

[1] This is the case for the Quintus emulator although it is coded in a generic RTL language ("PROGOL") to simplify ports.

existing fragments of C code present in the components of the emulator (such as builtins, low-level file and stream management, memory management and garbage collection routines, etc.) can be reused by the new compiler. This is important because our intention is to develop not a prototype but a full compiler that can be put into everyday use and it would be an unrealistic amount of work to develop all those parts again. Also, compilation to C allows us to translate Prolog modules into C files that can be compiled as source files in multi-language applications.

As mentioned before, the selection of C as target low-level language allows performing a large class of low-level optimizations while easing portability. A practical advantage in this sense is the availability of C compilers for most architectures, such as `gcc`, which generate very efficient executable code. The difference with other systems which compile to C comes from the fact that the translation that we propose includes a scheme for optimizing the resulting code using higher-level information available at compile-time regarding determinacy, types, instantiation modes, etc. of the source program. The objective is better run-time performance, including a reduction of the size of executables. We also strive to preserve portability and maintainability by avoiding the use of non-standard C code as much as possible.

This line of reasoning lead us also not to adopt other approaches such as compiling to C--. The goal of C-- is to achieve portable high performance without relinquishing control over low-level details. However, the associated tools do not seem to be presently mature enough as to be used for a compiler in production status within a near future, and not even to be used as base for a research protoype in their present stage. Future portability will also depend on the existence of back-ends for a range of architectures. We, however, are quite confident that the backend which now generates C code could be adapted to generate C-- (or other low-level languages) without too many problems.

The high-level information is expressed by means of a powerful and well-defined assertion language [17], and inferred by automatic global analysis tools which code the results of analysis as assertions or simply provided by the user. In our system we take advantage of the availability of relatively mature tools for this purpose within the Ciao system, and, in particular the preprocessor, CiaoPP [11].

Our approach is thus different from, for example, `wamcc` (the forerunner of the current Gnu Prolog), which also generated C, but did not use extensive analysis information (and it used low-level, clever tricks which in practice tied it to a particular C compiler, `gcc`). Aquarius [27] (and [23]) used analysis information at several compilation stages, but they generated directly machine code, and it has proved difficult to port and maintain these systems. Also, program analysis technology was not as mature at the time as it is now. Notwithstanding, they were landmark contributions that proved the power of using global information in a Prolog compiler.

A drawback of putting more burden on the compiler is that compile times grow, and compiler complexity increases, specially in the global analysis phase. While this can turn out to be a problem in extreme cases, incremental analysis in combination with a suitable module system [3] can result in very reasonable analysis times in practice. Moreover, global analysis (or even the compilation to C) are not mandatory in our proposal and can be reserved for the phase of generating the final, “production” executable. We expect that, as the system matures, the Prolog-to-C compiler itself (now in a prototype stage) will not be slower than a Prolog-to-bytecode compiler.

2 The Basic Compilation Scheme

We now present the basic compilation strategy. The optimizing compilation using global program information will be described in Section 3.

The compilation process starts with a preprocessing phase which canonizes clauses (removing aliasing and structure unification from the head, also known as “normalization”), and expands disjunctions, negations and if-then-else constructs. It also unfolds calls to `is/2` when possible into calls to simpler arithmetic predicates, replaces the cut by calls to the lower-level predicates `metachoice/1` (which stores in its argument the address of the current choicepoint) and `metacut/1` (which performs a cut to the choicepoint whose address is passed in its argument), and performs a simple, local analysis which gathers information about the type and freeness state of variables.[2] Having this analysis in the compiler (in addition to the analyses

[2] In general, the types used throughout the paper are *instantiation types*, i.e., they have mode information built in (see [17] for

put_variable(I,J)	⟨uninit,I⟩ = ⟨uninit,J⟩
put_value(I,J)	⟨init,I⟩ = ⟨uninit,J⟩
get_variable(I,J)	⟨uninit,I⟩ = ⟨init,J⟩
get_value(I,J)	⟨init,I⟩ = ⟨init,J⟩

unify_variable(I[, J])	`if` (initialized(J)) `then`
	⟨uninit,I⟩ = ⟨init,J⟩
	`else`
	⟨uninit,I⟩ = ⟨uninit,J⟩

unify_value(I[, J])	`if` (initialized(J)) `then`
	⟨init,I⟩ = ⟨init,J⟩
	`else`
	⟨init,I⟩ = ⟨uninit,J⟩

Table 1: Representation of some WAM unification instructions with types.

performed by the preprocessor) allows improving the code even in the case that no external information is available from previous stages or the user. The following steps of the compiler include the translation from this normalized version of Prolog to WAM-based instructions (at this point the same ones used by the Ciao emulator), and then splitting these WAM instructions into an intermediate low level code and performing the final translation to C.

Typing WAM Instructions: WAM instructions dealing with data are handled internally using an enriched representation which encodes the possible instantiation state of their arguments.

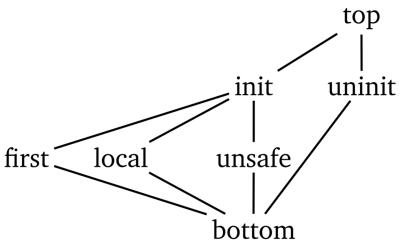

Figure 1: Lattice of WAM types.

This allows using original type information, and also generating and propagating lower-level information regarding the type (i.e., from the point of view of the tags of the abstract machine) and instantiation/initialization state of the variables (which is not seen at a higher level). Each unification instruction is represented as ⟨TypeX, X⟩ = ⟨TypeY, Y⟩, where *TypeX* and *TypeY* refer to the classification of WAM-level types (see Figure 1), and *X* and *Y* refer to variables, which may be later stored as WAM X or Y registers or directly passed on as C function arguments.

Table 1 summarizes the aforementioned representation for some selected cases. The registers taken as arguments are the temporary registers *x(I)*, the stack registers *y(I)* and the register for structure arguments *n(I)*. The last one can be seen as the second argument which is implicit in the unify_* WAM instructions. A number of other special-purpose registers (`ok`, `temp`, ...) are available, and used, for example, to hold intermediate results from expression evaluation and to record whether a builtin failed. *_constant, *_nil, *_list and *_structure WAM instructions are represented similarly. Only register variables *x(·)* are created in an uninitialized state, and they are initialized on demand (in particular, when calling another predicate which may overwrite the registers, and in the points where garbage collection can start). Stack and structure (heap) variables are created initialized.

One of the advantages of this representation is that it is more uniform than the traditional WAM instructions. In particular, as more information is known about the variables, the associated (low level) types can be refined and more specific code generated. Using a richer lattice and initial information (Section 3), a more descriptive intermediate code is generated and used in the back-end.

Generation of the Intermediate Low Level Language: WAM instructions are then split into simpler ones, which are more suitable for optimizations. This also allows simplifying the generation of the final C code (and probably also the generation of code in other languages of similar abstraction level). The degree of complexity of the low-level code is similar to the one proposed in the BAM [25]. Table 2 summarizes the instructions. The *Type* argument which appears in several of them is intended to reflect the type of the instruction arguments: for example, in the instruction *bind*, *Type* used to specify if the arguments contain a

a more complete discussion of this issue). *Freeness of variables* distinguishes between free variables and the *top* type, "term", which includes any term.

Choice, stack and heap management instructions	
no_choice	Mark that there is no alternative
first_choice(Arity, Alt)	Create a choicepoint
middle_choice(Arity, Alt)	Change the alternative
last_choice(Arity)	Remove the alternative
complete_choice(Arity)	Complete the choice point
cut_choice(Chp)	Cut to a given choice point
push_frame	Allocate a frame on top of the stack
complete_frame(FrameSize)	Complete the stack frame
modify_frame(NewSize)	Change the size of the frame
pop_frame	Deallocate the last frame
recover_frame	Recover after returning from a call
ensure_heap(CS, Amount, Arity)	Ensure that enough heap is allocated. (CS indicates completion status of the choice point)
Data	
load(X, Type)	Load *X* with a term
trail_if_conditional(A)	Trail if *A* is a conditional variable
bind(TypeX, X, TypeY, Y)	Bind *X* and *Y*
read(Type, X)	Begin read of the structure arguments of *X*
deref(X, Y)	Dereference *X* into *Y*
move(X, Y)	Copy *X* to *Y*
globalize_if_unsafe(X, Y)	Copy *X* to *Y* ensuring safety
globalize_to_arg(X, Y)	Copy *X* to argument register *Y* ensuring safety
call(CallerImp, CalledImp, Pred, In, Out, FailCont)	Call a builtin or a user predicate. *CallerImp* and *CalleeImp* mark how caller and callee are compiled.
Control	
ijump(X)	Jump to the address stored in *X*
jump(Label)	Jump to *Label*
cjump(Cond, Label)	Jump to *Label* if *Cond* is true
switch_on_type(X, Var, Str, List, Cons)	Jump to the label that matches the type of *X*
switch_on_functor(X, Table, Else)	
switch_on_cons(X, Table, Else)	
Conditions	
not(Cond)	Negate the *Cond* condition
test(Type, X)	True if *X* matches *Type*
equal(X, Y)	True if *X* and *Y* are equal
erroneous(X)	True if *X* has an erroneous value

Table 2: Control and data instructions.

variable (and, if this is known, whether it lives in the heap, in the stack, etc.) or not. For the unification of structures, the use of write and read modes is avoided using a two-stream scheme (see [26] for an explanation and references) which is encoded using with the unification instructions in Table 1 and later translated into the required series of assignments. This scheme requires explicit control instructions, hence the existence of jump instructions (*jump*, *cjump*, and *ijump*). Jumps are performed to labels, marked as *global* (when they have to be stored in global data structures, such as the next alternative in a choicepoint) or *local*. For efficiency in indexing, the WAM instructions *switch_on_term*, *switch_on_cons* and *switch_on_functor* are also included, although the C back-end does not exploit them fully at the moment, resorting to a linear search in some cases. Failing is done by jumping to the special label *fail*. Builtins return an exit state in one argument, which is used to decide whether to backtrack or not. Determinism information, when available, is passed on through this stage and used when compiling with optimizations (see Section 3).

Compilation to C: This stage in turn produces the output C code. This C code conceptually corresponds to an unfolding of the initial bytecode emulator loop with respect to the particular sequence(s) of bytecode corresponding to the program. In the points where the emulated program counter changes continuations are passed using pointers to functions. Each block of bytecode (i.e., each sequence beginning in a label and ending in an instruction involving a possible jump) is translated to an individual C function. The state of the

```
        while (code != NULL)
           code = ((Continuation (*)(State *))code)(state);
```

```
Continuation foo(State *state) {
   ...
   state->cont = &foo_cont;
   return &foo2;
}
```

```
Continuation foo_cont(State *state) {
  ...
  return state->cont;
}
```

Figure 2: The C execution loop and blocks scheme.

abstract machine is the input argument to the function and the next continuation is the output argument. This approach avoids building functions that are too large and would create problems for the C compiler. Figure 2 shows schematic versions of the execution loop and of the functions that code blocks are compiled into. The translation also incorporates an optimization which reduces the function calling overhead: `goto` statements are used for jumps to local labels which are located in the same code block.

This scheme does not require using machine-dependent options of the C compiler or extensions to the ANSI C language (although machine-dependent optimizations can of course be given to the C compiler). Other systems, as [20] or [21], take advantage of machine-dependent and non-portable constructs to increase performance. However, one of the goals of our system –to study the impact of optimizations based on available information on the program– can be achieved with the proposed compilation scheme, and, as mentioned before, we give portability and code cleanliness a high priority. The possibility of producing more efficient but non-portable code can always be added at a later stage.

An Example — the `fact/2` Predicate: We will illustrate briefly the different stages of compilation using the well-known factorial program (Figure 3). We have chosen it due to its simplicity, even if the performance gain is not very high in this case. The code after the first canonizing and rewriting stage is shown in Figure 4. The WAM code corresponding to the recursive clause is listed in the leftmost column of Table 3, and the internal representation of this code appears in the same table, in the middle column. Note how variables are annotated using information which can be deduced from local inspection of the clause.

This WAM-like representation is translated to the low-level code shown in Figure 5 (ignore, for the moment, the shadowed and framed regions; they will be further discussed in Section 3). This code, which is quite low-level now, is what is finally translated to C.

For reference, executing `fact(100, N)` 20000 times took 0.65 seconds running emulated bytecode, and 0.63 seconds running the code compiled to C (a speedup of 1.03). This was all done without using any external, global type information. In the next section we will see how this performance can be improved with the use of type information.

```
fact(0, 1).
fact(X, Y) :-
    X > 0,
    X0 is X - 1,
    fact(X0, Y0),
    Y is X * Y0.
```

Figure 3: Factorial, initial code.

```
fact(A, B) :-
    0 = A,
    1 = B.
```

```
fact(A, B) :-
    A > 0,
    builtin__sub1_1(A, C),
    fact(C, D),
    builtin__times_2(A, D, B).
```

Figure 4: Factorial, after preprocessing.

3 Improving Code Generation

As mentioned in Section 1, our objective is to improve the code generation process using information regarding global properties of predicates which is coded as assertions [17] –a few such assertions can be seen in the example of Section 3. In the current version of the compiler optimization is performed using information on instantiation types (i.e., moded types) as well as number of solutions (determinacy).

WAM code	Without Types	With Types
put_constant(0,2)	0 = ⟨uninit,x(2)⟩	0 = ⟨uninit,x(2)⟩
builtin_2(37,0,2)	⟨init,x(0)⟩ > ⟨int(0),x(2)⟩	⟨int,x(0)⟩ > ⟨int(0),x(2)⟩
allocate	builtin__push_frame	builtin__push_frame
get_y_variable(0,1)	⟨uninit,y(0)⟩ = ⟨init,x(1)⟩	⟨uninit,y(0)⟩ = ⟨var,x(1)⟩
get_y_variable(2,0)	⟨uninit,y(2)⟩ = ⟨init,x(0)⟩	⟨uninit,y(2)⟩ = ⟨int,x(0)⟩
init([1])	⟨uninit,y(1)⟩ = ⟨uninit,y(1)⟩	⟨uninit,y(1)⟩ = ⟨uninit,y(1)⟩
true(3)	builtin__complete_frame(3)	builtin__complete_frame(3)
function_1(2,0,0)	builtin__sub1_1(⟨init,x(0)⟩, ⟨uninit,x(0)⟩)	builtin__sub1_1(⟨int,x(0)⟩, ⟨uninit,x(0)⟩)
put_y_value(1,1)	⟨init,y(1)⟩ = ⟨uninit,x(1)⟩	⟨var,y(1)⟩ = ⟨uninit,x(1)⟩

call(fac/2,3)	builtin__modify_frame(3)
	fact(⟨init,x(0)⟩, ⟨init,x(1)⟩)
		builtin__modify_frame(3)
		fact(⟨init,x(0)⟩, ⟨var,x(1)⟩)

put_y_value(2,0)	⟨init,y(2)⟩ = ⟨uninit,x(0)⟩	⟨int,y(2)⟩ = ⟨uninit,x(0)⟩
put_y_value(2,1)	⟨init,y(1)⟩ = ⟨uninit,x(1)⟩	⟨number,y(1)⟩ = ⟨uninit,x(1)⟩
function_2(9,0,0,1)	builtin__times_2(⟨init,x(0)⟩, ⟨init,x(1)⟩,⟨uninit,x(0)⟩)	builtin__times_2(⟨int,x(0)⟩, ⟨number,x(1)⟩, ⟨uninit,x(0)⟩)
get_y_value(0,0)	⟨init,y(0)⟩ = ⟨init,x(0)⟩	⟨var,y(0)⟩ = ⟨init,x(0)⟩
deallocate	builtin__pop_frame	builtin__pop_frame
execute(true/0)	builtin__proceed	builtin__proceed

Table 3: WAM code and internal representation without and with external types information. Underlined instruction changed due to additional information.

The generation of low-level code using additional type information makes use of an extended type lattice obtained by replacing the *init* element in the lattice in Figure 1 with the type domain in Figure 6. This information enriches the *Type* parameter of the low-level code. Additionally, as mentioned before, any information about the determinacy / number of solutions of each call is carried over into this stage and used in it to optimize the generated C code.

In general, information about the types of variables and determinism of predicates allows avoiding introducing unnecessary tests during the compilation to low level code. The standard WAM compilation performs

```
global(fact/2):
  first_choice(2,V1)
  ensure_heap(incompleted_choice,callpad,2)
  deref(x(0),x(0))
  cjump(not(test(var,x(0))),local(V3))
  load(temp2,int(0))
  bind(var,x(0),nonvar,temp2)
  jump(local(V4))
local(V3):
  cjump(not(test(int(0),x(0))),fail)
local(V4):
  deref(x(1),x(1))
  cjump(not(test(var,x(1))),local(V5))
  load(temp2,int(1))
  jump(local(V6))
local(V5):
  cjump(not(test(int(1),x(1))),fail)
local(V6):
  complete_choice(2)
  ijump(continuation)
global(V1):
  last_choice(2)
  load(x(2),int(0))
  builtin(numgt_2,[x(0),x(2)],ok)
```

```
  cjump(not(ok),fail)
  push_frame
  move(x(1),y(0))
  move(x(0),y(2))
  load(y(1),var(stack))
  complete_frame(3)
  function(sub1_1,[x(0)],x(0),0,1)
  cjump(erroneous(x(0)),fail)
  move(y(1),x(1))
  modify_frame(3)
  load(continuation,global(V0))
  jump(global(fact/2))
global(V0):
  recover_frame
  move(y(2),x(0))
  move(y(1),x(1))
  function(times_2,[x(0),x(1)],x(0),0,2)
  cjump(erroneous(x(0)),fail)
  deref(y(0),temp)
  deref(x(0),x(0))
  builtin(unify,[temp,x(0)],ok)
  cjump(not(ok),fail)
  pop_frame
  ijump(continuation)
```

Figure 5: Low level code for the fact/2 example (see also Section 3).

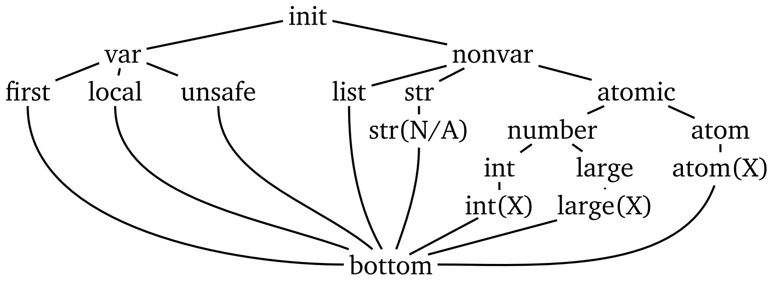

Figure 6: Extended *init* subdomain.

also some optimizations (e.g., classification of variables and indexing on the first argument), but it is based on a per-clause (per-predicate, in the case of indexing) based analysis, and in general it does not propagate, e.g., information deduced from arithmetical builtins. A number of further optimizations can be done by using richer type, mode, and determinism information:

Unify instructions: A call to the general *unify* builtin is replaced by the more specialized *bind* instruction if one or both arguments are known to store variables. When arguments are known to be constants, a simple comparison instruction is emitted instead.

Two-Stream Unification: The unification of a register with a structure or constant requires some tests for determining the unification mode (read or write). Also, in read mode, an additional test is required to compare the register value with the constant or the structure functor. These tests can often be reduced to true or false at compile-time if enough information is known about the variable.

Index Tree Generation: Type information is also used to optimize the generation of index trees, which are used as part of the clause selection process. An index tree is generated by selecting some literals from the beginning of the clause, mostly builtins and unifications, which give some amount of type/mode information. This is used to construct a decision tree on the types of the first argument.[3] When type information is available, the indexing tree can be optimized by removing some of the tests in the nodes.

Avoiding Unnecessary Variable Safety Tests: Another optimization performed in the low level code using type information is the replacement of globalizing instructions for unsafe variables by explicit dereferences. When the type of a variable is nonvar, its globalization is equivalent to a dereference, which is faster.

Uninitialized Output Arguments: When possible, letting the called predicate fill in the contents of output arguments in pre-established registers avoids allocation, initialization and binding of free variables, which is slower.

Selecting Optimized Predicate Versions: Calls to predicates can also be optimized in the presence of type information. Specialized versions (in the sense of low level optimizations) can exist and be selected using the call patterns deduced from the type information. The current implementation does not support automatic versions of user predicates (since this is done automatically by the preprocessor[18]), but it does optimize natively internal *builtin* predicates written in C (such as, e.g., arithmetic builtins) which results in relevant speedups in many cases.

[3] This can of course be extended to other arguments.

Determinism: These optimizations are based on two types of analysis. The first one uses information regarding the number of solutions for a predicate call to deduce, for each such call, if there is a known and fixed fail continuation. Then, instructions to create choicepoints and to restore previous choicepoint states are inserted. The resulting code is then re-analyzed to remove these instructions when possible or to replace them by simpler ones (e.g., to restore a choice point state without untrailing, if it is known at compile time that the execution will not trail any value since the choice point was created). The latter can take advantage of additional information regarding register, heap, and trail usage of each predicate.[4] In addition, the C back-end can generate different argument passing schemes based on determinism information: predicates with zero or one solution can be translated to a function returning a boolean, and predicates with exactly one solution to a function returning `void`.

An Example — the `fact/2` Predicate with program information: Let us assume that it is known that `fact/2` (Figure 3) is always called with its first argument instantiated to a small integer (an integer which fits into a tagged word of the internal representation) and its second argument is a free variable. This information can be written in the assertion language as follows:[5]

```
:- true pred fact(X, Y) :  int * var => int * number.
```

which reflects the types and modes of the calls and successes of the predicate. The propagation of that information through the canonized predicate gives the annotated program shown in Figure 7.

```
fact(A, B) :-                          fact(A, B) :-
    true(int(A)),                          true(int(A)),
    0 = A,                                 A > 0,
    true(var(B)),                          true(int(A)), true(var(C)),
    1 = B.                                 builtin__sub1_1(A, C),
                                           true(any(C)), true(var(D)),
                                           fact(C, D),
                                           true(int(A)), true(number(D)),
                                           true(var(B)),
                                           builtin__times_2(A, D, B).
```

Figure 7: Annotated factorial (using type information).

The WAM code generated for this example is shown in the rightmost column of Table 3. Underlined instructions were made more specific due to the initial information — note, however, that the representation is homogeneous with respect to the "no information" case. The impact of type information in the generation of low-level code can be seen in Figure 5. Instructions in the shaded regions are **removed** when type information is available, and the (arithmetic) builtins enclosed in rectangles are replaced by calls to versions specialized to work with small integers and which do not perform type/mode testing. The optimized `fact/2` program took 0.54 seconds with the same call as in Section 2: a 20% speedup with respect to the bytecode version and a 16% speedup over the compilation to C without type information.

4 Performance Measurements

We have evaluated the performance behavior of the executables generated with our compiler using translation to C with respect to that of the emulated bytecode on a set of standard benchmarks. The benchmarks are not real-life programs, and some of them have been executed up to 10.000 times in order to obtain reasonable and stable execution times. All the measurements have been made in a Pentium 4 Xeon @ 2.0GHz with 1Gb of RAM, running Linux with a 2.4 kernel and using `gcc` 3.2 as C compiler. A short description of the benchmarks follows:

[4] This is currently known only for internal predicates written in C, and which are available by default in the system, but the scheme is general and can be extended to Prolog predicates.

[5] The `true` prefix implies that this information is to be *used*, rather than to be *checked* by the compiler.

Program	Bytecode (Std. Ciao)	Non opt. C	Opt1. C	Opt2. C
queens11 (1)	691	391 (1.76)	208 (3.32)	166 (4.16)
crypt (1000)	1525	976 (1.56)	598 (2.55)	597 (2.55)
primes (10000)	896	697 (1.28)	403 (2.22)	402 (2.22)
tak (1000)	9836	5625 (1.74)	5285 (1.86)	771 (12.75)
deriv (10000)	125	83 (1.50)	82 (1.52)	72 (1.74)
poly (100)	439	251 (1.74)	199 (2.20)	177 (2.48)
qsort (10000)	521	319 (1.63)	378 (1.37)	259 (2.01)
exp (10)	494	508 (0.97)	469 (1.05)	459 (1.07)
fib (1000)	263	245 (1.07)	234 (1.12)	250 (1.05)
knights (1)	621	441 (1.40)	390 (1.59)	356 (1.74)
Average Speedup		(1.47)	(1.88)	(3.18)

Table 4: Bytecode emulation vs. unoptimized, optimized (types), and optimized (types and determinism) compilation to C.

Program	GProlog	WAMCC	SICStus	SWI	Yap	Mercury	Opt2. C / Mercury
queens11 (1)	809	378	572	5869	362	106	1.57
crypt (1000)	1258	966	1517	8740	1252	160	3.73
primes (10000)	1102	730	797	7259	1233	336	1.20
tak (1000)	11955	7362	6869	74750	8135	482	1.60
deriv (10000)	108	126	121	339	100	72	1.00
poly (100)	440	448	420	1999	424	84	2.11
qsort (10000)	618	522	523	2619	354	129	2.01
exp (10)	—	—	415	—	340	—	—
fib (1000)	—	—	285	—	454	—	—
knights (1)	911	545	631	2800	596	135	2.63
						Average	1.98

Table 5: Speed of other Prolog systems and Mercury

crypt: Cryptoarithmetic puzzle involving multiplication.
primes: Sieve of Erathostenes (with N = 98).
tak: Computation of the Takeuchi function with arguments `tak(18, 12, 6, X)`.
deriv: Symbolic derivation of polynomials.
poly: Symbolically raise `1+x+y+z` to the 10th power.
qsort: QuickSort of a list of 50 elements.
exp: Computation of 13^{7111} using both a linear- and a logarithmic-time algorithm.
fib: Computation of F_{1000} using a simply recursive predicate.
knight: Chess knight tour (visit only once all the board cells) in a 5×5 board.

A summary of the results appears in Table 4. The number between parentheses in first column is the number of iterations of each benchmark (used to obtain an execution long enough). The second column contains the execution times of programs compiled to bytecode (this represents the speed of the standard Ciao bytecode emulator). The third column corresponds to programs compiled to C without compile-time information (which corresponds, basically, to mimicking the bytecode execution). The fourth and fifth columns correspond, respectively, to the execution times when compiling to C using type information and type+determinism information to optimize the resulting code. The numbers between parentheses are the speedups relative to the bytecode version. All times are in milliseconds.

In order to know how these numbers compare with the performance of other Prolog systems, Table 5 shows the execution times (also in milliseconds) for the same benchmarks in four well-known Prolog compilers: GNU Prolog 1.2.16, wamcc 2.23, SICStus 3.8.6, SWI-Prolog 5.2.7, and Yap 4.5.0. The aim here is not really to compare directly with them, because a different technology is being used (compilation to C and use of external information which they cannot take advantage of), but rather to establish that our baseline, the speed of the bytecode system (Ciao), is similar (and quite close, in particular, to that of SICStus and Yap). Thus, in principle, comparable optimizations could be made in these systems. YAP was itself compiled with multi-precision arithmetic, which makes its execution a little bit slower than without it (just some milliseconds, not enough as to make a significant difference). The cells marked with "—" correspond to cases where the benchmark could not be executed (in GNU Prolog, wamcc, and SWI, due to lack of multi-precision arithmetic).

We also include the performance results for the compiler for the Mercury language [21] (version 0.11.0). Strictly speaking the Mercury compiler is not a Prolog compiler, since the source language is substantially different from Prolog. On the other hand the Mercury language does have enough similarities to be relevant and its performance is interesting as an upper reference line given that the language was designed precisely to allow the compiler, which directly generates machine code, to achieve very high performance by using extensive low-level optimization compilation techniques. Also, the language design requires that the programs necessarily contain as part of the source the necessary information to perform these optimizations.

Going back to Table 4, while some performance gains are obtained in the *naive* translation to C, such gains are not very significant, and there is even one program which shows a slowdown. We have tracked this down to be due to several factors:

- The simple compilation scheme generates C code that is as clean and portable as possible, avoiding tricks which would speed up the programs. The execution profile is also very near to what the emulator would do.
- The C execution loop (Figure 2) is slightly more costly (by a few assembler instructions) than the fetch/switch loop of the emulator. We have identified this as a cause for the poor speedup of programs where recursive calls dominate the execution time. We want, of course, to improve this point in the future.
- The increment in size of the program (see Table 6) may also cause more cache misses. We also want to investigate this point in more detail.

As expected, the performance obtained when using compile-time information is much better. The best speedups are obtained in benchmarks using arithmetic builtins, for which the compiler produces optimized versions where several groundness and type checks have been removed. This is, for example, the case of `queens`, in which it is known that all the numbers involved are small integers (i.e., there is no need for infinite precision arithmetic). Besides avoiding checks, the functions which implement the arithmetic operations for small integers are simple enough as to be inlined by the C compiler. This is an example of an added benefit which comes for free from compiling to an intermediate language (C, in this case) and using tools designed for it. When determinism information is used, the execution is often (but not always) improved. The Takeuchi function (`tak`) is an extreme case, where determinism information saves choicepoint generation and execution time. While the performance obtained is still far (a factor of 2 on average) from that of Mercury, the results are encouraging given that we are dealing with a more complex source language (which preserves fully unification, logical variables, etc.), we are using a portable approach (compilation to standard C), and we have not applied yet all possible optimizations.

Table 6 compares object size of the bytecode and the different schemes of compilation to C. Unit is bytes. As mentioned in Section 1, due to the different granularity of instructions, larger object files and executables are expected when compiling to C. The ratio depends heavily on the program and the optimizations applied. Size increase can be as large as 15× when translating to C without optimizations, and the average case sits around a 7-fold increase. This is also partially due to the indexing mechanism, which repeats some code. We plan to improve this in the future. It must also be pointed out that executing the bytecode requires always the presence of the bytecode emulator, which is around 300Kb (depending on the architecture and the optimizations applied) and which should be added to the figures in Table 6 for the emulator. On the other hand this can be shared among different executables as a library. The executables obtained through C do not need the emulation loop, and only the code for the GC routines and the definition of predicates internally written in C has to be linked at runtime.

The size of the object code produced by `wamcc` is roughly comparable to that generated by our compiler, although `wamcc` produces smaller object code files. However the final executable / process size depends also

Program	Bytecode	Non opt. C	Opt1. C	Opt2. C
queens11	7167	36096 (5.03)	29428 (4.10)	42824 (5.97)
crypt	12205	186700 (15.30)	107384 (8.80)	161256 (13.21)
primes	6428	50628 (7.87)	19336 (3.00)	31208 (4.85)
tak	5445	18928 (3.47)	18700 (3.43)	25476 (4.67)
deriv	9606	46900 (4.88)	46644 (4.85)	97888 (10.19)
poly	13541	163236 (12.05)	112704 (8.32)	344604 (25.44)
qsort	6982	90796 (13.00)	67060 (9.60)	76560 (10.96)
exp	6463	28668 (4.43)	28284 (4.37)	25560 (3.95)
fib	5281	15004 (2.84)	14824 (2.80)	18016 (3.41)
knights	7811	39496 (5.05)	39016 (4.99)	39260 (5.03)
Average Increase		(7.39)	(5.43)	(8.77)

Table 6: Compared size of object files (bytecode vs. C).

on which libraries are linked statically and/or dynamically. The Mercury system is somewhat incomparable in this regard: it certainly produces relatively small component files but then relatively large final executables (over 1.5 MByte).

The size, in general, decreases when using type information, as many dynamic type tests are removed. The average size is now around five times the bytecode size. Adding determinism information increases the code size because of the additional inlining performed by the C compiler and the more complex parameter passing code. The options passed on to the C compiler were the same in all the programs, and the decision of whether to inline or not was left to it. Some experiments showed that asking the C compiler to do more aggressive inlining did not help to achieve better speedups.

It is interesting that some of the optimizations used in the compilation to C would not give comparable results when applied directly to a bytecode emulator. A version of the bytecode emulator hand-coded to work only with small integers (which can be boxed into a tagged word) performed worse than that obtained doing the same with compilation to C. That suggests that when the overhead of calling builtins is reduced, as is the case in the compilation to C, some optimizations which only produce minor improvements for emulated systems acquire greater importance.

5 Conclusions and Future Work

We have reported on the scheme and performance of a Prolog-to-C compiler which uses type analysis and determinacy information to improve the final code by removing type and mode checks and by making calls to specialized versions of some builtins. We have also provided preliminary performance results for this compiler. Our compiler is still in a preliminary stage, but already shows promising results.

The compilation uses internally a simplified and more homogeneous representation for WAM code, which is then translated to a lower-level intermediate code. This step uses the type and determinacy information available at compile time. This code is finally translated into C by the compiler back-end. The intermediate code, as in other similar compilers, makes the final translation step easier and will make it easier to develop new back-ends for other target languages.

We have found using the same information to optimize a WAM bytecode emulator to be more difficult and to result in lower speedups, due to the greater granularity of the bytecode instructions (which aims at reducing the cost of fetching them). The same result has been reported elsewhere [26], although some recent work tries to improve WAM code by means of local analysis [9, 8].

We expect to be able to use information (e.g., determinacy) to improve also clause selection, as well as to generate a better indexing scheme at the C level by using hashing on constants, instead of the linear search performed now. Also, we want to study which other optimizations can be added to the generation of C code without breaking its portability, and how the intermediate representation can be used to generate code for other back-ends (for example, GCC RTL, CIL, Java bytecode, etc.).

References

[1] H. Ait-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction*. MIT Press, 1991.

[2] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference Manual (v1.8). The Ciao System Documentation Series–TR CLIP4/2002.1, School of Computer Science, Technical University of Madrid (UPM), May 2002. System and on-line version of the manual available at `http://clip.dia.fi.upm.es/Software/Ciao/`.

[3] D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 131–148. Springer-Verlag, July 2000.

[4] P. Codognet and D. Diaz. WAMCC: Compiling Prolog to C. In L. Sterling, editor, *International Conference on Logic Programming*, pages 317–331. MIT PRess, June 1995.

[5] A. Colmerauer. The Birth of Prolog. In *Second History of Programming Languages Conference*, ACM SIGPLAN Notices, pages 37–52, March 1993.

[6] B. Demoen and P.-L. Nguyen. So Many WAM Variations, So Little Time. In *Computational Logic 2000*, pages 1240–1254. Springer Verlag, July 2000.

[7] D. Diaz and P. Codognet. Design and Implementation of the GNU Prolog System. *Journal of Functional and Logic Programming*, 2001(6), October 2001.

[8] M. Ferreira and L. Damas. Multiple Specialization of WAM Code. In *Practical Aspects of Declarative Languages*, number 1551 in LNCS. Springer, January 1999.

[9] M. Ferreira and L. Damas. Wam local analysis. In B. Demoen, editor, *Proceedings of CICLOPS 2002*, pages 13–25, Copenhagen, Denmark, June 2002. Department of Computer Science, Katholieke Universiteit Leuven.

[10] M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[11] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *10th International Static Analysis Symposium (SAS'03)*, number 2694 in LNCS, pages 127–152. Springer-Verlag, June 2003.

[12] S. L. P. Jones, N. Ramsey, and F. Reig. C--: A Portable Assembly Language that Supports Garbage Collection. In G. Nadathur, editor, *International Conference on Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 1–28. Springer Verlag, September 1999.

[13] A. Krall and T. Berger. The VAM_{AI} - an abstract machine for incremental global dataflow analysis of Prolog. In M. G. de la Banda, G. Janssens, and P. Stuckey, editors, *ICLP'95 Post-Conference Workshop on Abstract Interpretation of Logic Languages*, pages 80–91, Tokyo, 1995. Science University of Tokyo.

[14] A. Mariën. *Improving the Compilation of Prolog in the Framework of the Warren Abstract Machine*. PhD thesis, Katholieke Universiteit Leuven, September 1993.

[15] J. Mills. A high-performance low risc machine for logic programming. *Journal of Logic Programming (6)*, pages 179–212, 1989.

[16] F. Pereira. *C-Prolog User's Manual, Version 1.5*. University of Edinburgh, 1987.

[17] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[18] G. Puebla and M. Hermenegildo. Abstract Specialization and its Applications. In *ACM Partial Evaluation and Semantics based Program Manipulation (PEPM'03)*, pages 29–43. ACM Press, June 2003. Invited talk.

[19] *Quintus Prolog User's Guide and Reference Manual—Version 6*, April 1986.

[20] V. Santos-Costa, L. Damas, R. Reis, and R. Azevedo. *The Yap Prolog User's Manual*, 2000. Available from `http://www.ncc.up.pt/~vsc/Yap`.

[21] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *JLP*, 29(1–3), October 1996.

[22] Swedish Institute for Computer Science, PO Box 1263, S-164 28 Kista, Sweden. *SICStus Prolog 3.8 User's Manual*, 3.8 edition, Oct. 1999. Available from `http://www.sics.se/sicstus/`.

[23] A. Taylor. LIPS on a MIPS: Results from a prolog compiler for a RISC. In *1990 International Conference on Logic Programming*, pages 174–189. MIT Press, June 1990.

[24] A. Taylor. *High-Performance Prolog Implementation*. PhD thesis, Basser Department of Computer Science, Unversity of Sidney, June 1991.

[25] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, Univ. of California Berkeley, 1990. Report No. UCB/CSD 90/600.

[26] P. Van Roy. 1983-1993: The Wonder Years of Sequential Prolog Implementation. *Journal of Logic Programming*, 19/20:385–441, 1994.

[27] P. Van Roy and A. Despain. High-Performace Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer Magazine*, pages 54–68, January 1992.

[28] D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.

[29] W. Winsborough. Multiple Specialization using Minimal-Function Graph Semantics. *Journal of Logic Programming*, 13(2 and 3):259–290, July 1992.