Improved Compilation of Prolog to C Using Moded Types and Determinism Information

J. Morales[1], M. Carro[1], and M. Hermenegildo[1,2]

[1] C. S. School, Technical U. of Madrid,
jfran@clip.dia.fi.upm.es and mcarro@fi.upm.es
[2] Depts. of Comp. Sci. and Elec. and Comp. Eng., U. of New Mexico (UNM)
herme@fi.upm.es and herme@unm.edu

Abstract. We describe the current status of and provide performance results for a prototype compiler of Prolog to C, `ciaocc`. `ciaocc` is novel in that it is designed to accept different kinds of high-level information, typically obtained via an automatic analysis of the initial Prolog program and expressed in a standardized language of assertions. This information is used to optimize the resulting C code, which is then processed by an off-the-shelf C compiler. The basic translation process essentially mimics the unfolding of a bytecode emulator with respect to the particular bytecode corresponding to the Prolog program. This is facilitated by a flexible design of the instructions and their lower-level components. This approach allows reusing a sizable amount of the machinery of the bytecode emulator: predicates already written in C, data definitions, memory management routines and areas, etc., as well as mixing emulated bytecode with native 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.
Keywords: Prolog, C, optimizing compilation, global analysis.

1 Introduction

Several techniques for implementing Prolog have been devised since the original interpreter developed by Colmerauer and Roussel [1], many of them aimed at achieving more speed. An excellent survey of a significant part of this work can be found in [2]. 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 [3] 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), where the compiler produces relatively low level code in a special-purpose language. Most current emulators for Prolog are based on the Warren Abstract Machine (WAM) [4, 5], but other proposals exist [6, 7].

– Compilers to a lower-level language, often ("native") machine code, which require little or no additional support to be executed. One solution is for the compiler to generate machine code directly. Examples of this are Aquarius [8], versions of SICStus Prolog [9] for some architectures, BIM-Prolog [10], and Gnu Prolog [11]. Another alternative is to generate code in a (lower-level) language, such as, e.g., C-- [12] or C, for which compilers are readily available; the latter is the approach taken by `wamcc` [13].

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. This difference gets larger as more sophisticated forms of code analysis are performed as part of the compilation process. 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 [9, 14–17] offer very good performance and reduced program size which may be a crucial issue for very large programs and symbolic data sets.

Portability: Interpreters offer portability since executing the same Prolog code in different architectures boils down (in principle) to recompiling the interpreter. Emulators usually retain the portability of interpreters, by recompiling the emulator (bytecode is usually architecture-independent), unless they are written in machine code.[3] Compilers to native code require architecture-dependent back-ends which typically make porting and maintaining them a non-trivial task. Developing these back-ends can be simplified by using an intermediate RTL-level code [11], although different translations of this code are needed for different architectures.

Opportunities for optimizations: Code optimization can be applied at the Prolog level [18, 19], to WAM code [20], to lower-level code [21], and/or to native code [8, 22]. At a higher level it is typically possible to perform more global and structural optimizations, which are then implicitly carried over onto lower levels. Lower-level optimizations can be introduced as the native code level is approached; performing these low-level optimizations is one of the motivations for compiling to machine code. However, recent performance evaluations show that well-tuned emulators can beat, at least in some cases, Prolog compilers which generate machine code directly but which do not perform extensive optimization [11]. Translating to a low-level language such as C is interesting because it makes portability easier, as C compilers exist for most architectures and C is low-level enough as to express a large class of optimizations which cannot be captured 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. The emulator approach 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

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

it seems appropriate to perform optimizations at all levels and to eventually translate to machine code. The selection of a language such as C as an intermediate target can offer a good compromise between opportunity for optimization, portability for native code, and interoperability in multi-language applications.

In `ciaocc` we have taken precisely such an approach: we implemented a compilation from Prolog to native code via an intermediate translation to C which optionally uses high-level information to generate optimized C code. Our starting point is the standard version of Ciao Prolog [17], essentially an emulator-based system of competitive performance. Its abstract machine is an evolution of the &-Prolog abstract machine [23], itself a separate branch from early versions (0.5–0.7) of the SICStus Prolog abstract machine.

`ciaocc` adopts the same scheme for memory areas, data tagging, etc. as the original emulator. This facilitates mixing emulated and native code (as done also by SICStus) and has also the important practical advantage that many complex and already existing fragments of C code present in the components of the emulator (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 not to develop a prototype but a full compiler that can be put into everyday use and developing all those parts again would be unrealistic.

A practical advantage is the availability of high-quality C compilers for most architectures. `ciaocc` differs from other systems which compile Prolog to C in that that the translation includes a scheme to optionally optimize the code using higher-level information available at compile-time regarding determinacy, types, instantiation modes, etc. of the source program.

Maintainability and portability 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, which is of course very desirable. However, the associated tools do not seem to be presently mature enough as to be used for a compiler in production status within the near future, and not even to be used as base for a research prototype 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, which is assumed expressed by means of the powerful and well-defined assertion language of [24], is inferred by automatic global analysis tools. In our system we take advantage of the availability of relatively mature tools for this purpose within the Ciao environment, and, in particular the preprocessor, CiaoPP [25]. Alternatively, such assertions can also be simply provided by the programmer.

Our approach is thus different from, for example, `wamcc`, which also generated C, but which did not use extensive analysis information and used low-level tricks which in practice tied it to a particular C compiler, `gcc`. Aquarius [8] and Parma [22] used analysis information at several compilation stages, but they generated directly machine code, and it has proved difficult to port and maintain them. 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 and compiler complexity grow, 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 [26] can result in very reasonable analysis times in practice.[4] Moreover, global analysis is not mandatory in `ciaocc` and can be reserved for the phase of generating the final, "production" executable. We expect that, as the system matures, `ciaocc` itself (now in a prototype stage) will not be slower than a Prolog-to-bytecode compiler.

2 The Basic Compilation Scheme

The compilation process starts with a preprocessing phase which normalizes clauses (i.e., aliasing and structure unification is removed from the head), 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.[5] Having this analysis in the compiler (in addition to the analyses performed by the preprocessor) improves the code even if no external information is available. The compiler then translates this normalized version of Prolog to WAM-based instructions (at this point the same ones used by the Ciao emulator), and then it splits these WAM instructions into an intermediate low level code and performs 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.

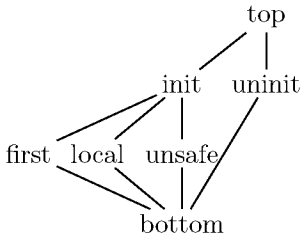

Fig. 1. Lattice of WAM types.

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). Unification instructions are represented as $\langle TypeX,\ X\rangle = \langle TypeY,\ Y\rangle$, 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. *init* and *uninit* correspond to initialized (i.e., free) and uninitialized variable cells. *First*, *local*, and *unsafe* classify the status of the variables according to where they appear in a clause.

Table 1 summarizes the aforementioned representation for some selected cases. The registers taken as arguments are the temporary registers *x(I)*, the stack variables *y(I)*, and the register for structure arguments *n(I)*. The last one

[4] See [25] and its references for reports on analysis times of CiaoPP.

[5] In general, the types used throughout the paper are *instantiation types*, i.e., they have mode information built in (see [24] for a more complete discussion of this issue). *Freeness of variables* distinguishes between free variables and the *top* type, "term", which includes any term.

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

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

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

Fig. 2. The C execution loop and blocks scheme.

can be seen as the second argument, implicit in the *unify_** WAM instructions. A number of other temporal registers are available, and used, for example, to hold intermediate results from expression evaluation. **_constant*, **_nil*, **_list* and **_structure* instructions are represented similarly. Only *x(·)* variables 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). This representation is more uniform than the traditional WAM instructions, and 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 can be generated and used in the back-end.

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.

Generation of the Intermediate Low Level Language: WAM-like control and data instructions (Table 2) are then split into simpler ones (Table 3) (of a level similar to that of the BAM [27]) which are more suitable for optimizations, and which simplify the final code generation. The *Type* argument in the unification instructions reflects the type of the their arguments: for example, in the instruction *bind*, *Type* is used to specify if the arguments contain a variable or not. For the unification of structures, write and read modes are avoided by using a two-stream scheme [2] which is implicit in the unification instructions in Table 1 and later translated into the required series of assignments and jump instructions (*jump*, *cjump*) in Table 2. 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. A more efficient indexing mechanism will be implemented in the near future.

Builtins return an exit state which is used to decide whether to backtrack or not. Determinism information, if available, is passed on through this stage and used when compiling with optimizations (see Section 3).

Choice, stack and heap management instructions	
no_choice	Mark that there is no alternative
push_choice(Arity)	Create a choicepoint
recover_choice(Arity)	Restore the state stored in a choicepoint
last_choice(Arity)	Restore state and discard latest choice point
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(Amount, Arity)	Ensure that enough heap is allocated.
Unification	
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 (safely) X to stack variable Y
globalize_to_arg(X, Y)	Copy (safely) X to structure argument Y
jump(Label)	Jump to *Label*
cjump(Cond, Label)	Jump to *Label* if *Cond* is true
not(Cond)	Negate the *Cond* condition
test(Type, X)	True if X matches *Type*
equal(X, Y)	True if X and Y are equal
Indexing	
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)	

Table 2. Control and data instructions.

Compilation to C: The final C code conceptually corresponds to an unfolding of the emulator loop with respect to the particular sequence(s) of WAM instructions corresponding to the Prolog program. Each basic block of bytecode (i.e., each sequence beginning in a label and ending in an instruction involving a possibly non-local jump) is translated to a separate C function, which receives (a pointer to) the state of the abstract machine as input argument, and returns a pointer to the continuation. This approach, chosen on purpose, does not build functions which are too large for the C compiler to handle. For example, the code corresponding to a head unification is a basic block, since it is guaranteed that the labels corresponding to the two-stream algorithm will have local scope. A failure during unification is implemented by (conditionally) jumping to a special label, *fail*, which actually implements an exit protocol similar to that generated by the general C translation. Figure 2 shows schematic versions of the execution loop and templates of the functions that code blocks are compiled into.

This scheme does not require machine-dependent options of the C compiler or extensions to ANSI C. One of the goals of our system –to study the impact of optimizations based on high-level information on the program– can be achieved with the proposed compilation scheme, and, as mentioned before, we give porta-

bility and code cleanliness a high priority. The option 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 compilation stages 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 normalized code is shown in Figure 4, and the WAM code corresponding to the recursive clause is listed in the leftmost column of Table 3, while the internal representation of this code appears in the middle column of the same table. Variables are annotated using information which can be deduced from local clause inspection.

This WAM-like representation is translated to the low-level code as shown in Figure 5 (ignore, for the moment, the framed instructions; they will be discussed in Section 3). This code 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 did not use external information, used the emulator data structures to store Prolog terms, and performed runtime checks to verify that the arguments are of the right type, even when this is not strictly necessary. Since the loop in Figure 2 is a bit more costly (by a few assembler instructions) than the WAM emulator loop, the speedup brought about by the C translation alone is, in many cases, not as relevant as one may think at first.

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

Fig. 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).
```

Fig. 4. Factorial, after normalizing.

3 Improving Code Generation

In order to improve the generated code using global information, the compiler can take into account types, modes, determinism and non-failure properties [25] coded as assertions [24] — a few such assertions can be seen in the example which appears later in this section. Automatization of the compilation process is achieved by using the CiaoPP analysis tool in connection with `ciaocc`. CiaoPP implements several powerful analysis (for modes, types, and determinacy, besides other relevant properties) which are able to generate (or check) these assertions. The program information that CiaoPP is currently able to infer automatically is actually enough for our purposes (with the single exception stated in Section 4).

The generation of low-level code using additional type information makes use of a lattice of moded types obtained by extending the *init* element in the lattice in Figure 1 with the type domain in Figure 6. `str(N/A)` corresponds to (and expands to) each of the structures whose name and arity are known at compile time. This information enriches the *Type* parameter of the low-level code. Information about the determinacy / number of solutions of each call is carried over into this stage and used to optimize the C code.

WAM code	Without Types/Modes	With Types/Modes
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.

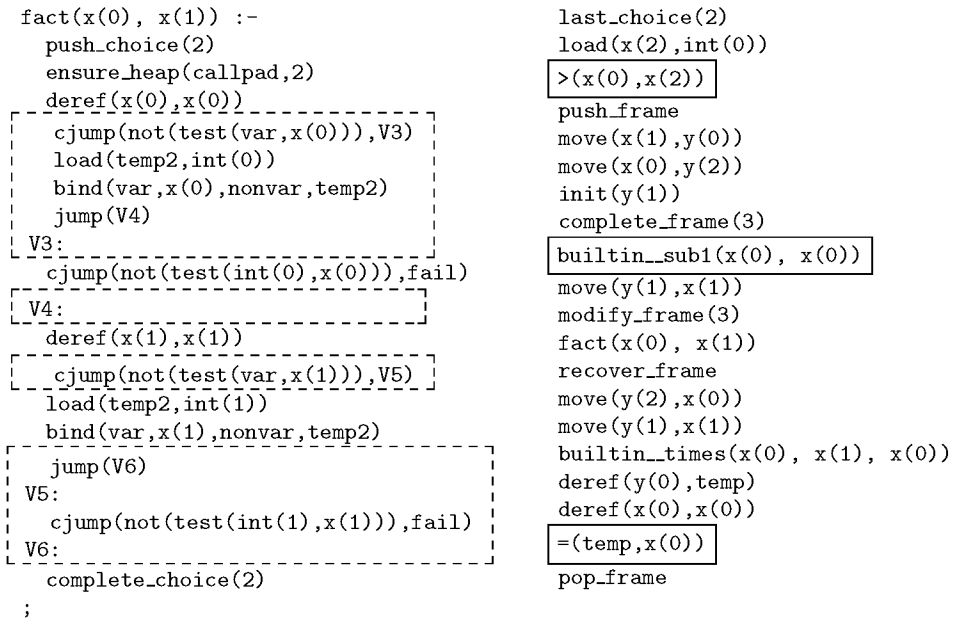

Fig. 5. Low level code for the `fact/2` example (see also Section 3).

In general, information about types and determinism makes it possible to avoid some runtime tests. The standard WAM compilation also performs some optimizations (e.g., classification of variables and indexing on the first argument),

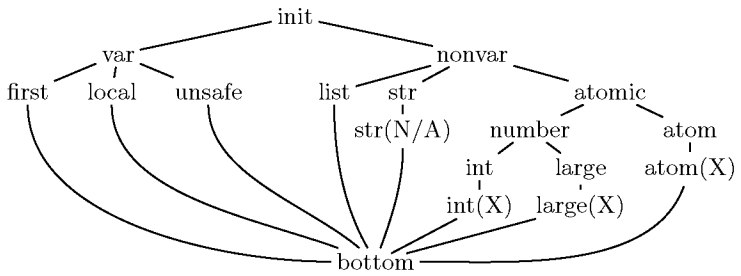

Fig. 6. Extended *init* subdomain.

but they are based on a per-clause (per-predicate, in the case of indexing) analysis, and in general it does not propagate the deduced information (e.g. from arithmetic builtins). A number of further optimizations can be done by using type, mode, and determinism information:

Unify Instructions: Calls to the general *unify* builtin are 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: Unifying a register with a structure/constant requires some tests to determine the unification mode (read or write). An additional test is required to compare the register value with the structure/constant. These tests can often be removed at compile-time if enough information is known about the variable.

Indexing: Index trees are generated by selecting literals (mostly builtins and unifications), which give type/mode information, to construct a decision tree on the types of the first argument.[6] When type information is available, the search 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 predicate versions (in the sense of low level optimizations) can be generated and selected using call patterns deduced from the type information. The current implementation does not generate specialized versions of user predicates, since this can already be done extensively by CiaoPP [18]. However it does optimize calls to internal *builtin* predicates written in C (such as, e.g., arithmetic builtins), which results in relevant speedups in many cases.

[6] This is the WAM definition, which 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 manage choicepoints 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.[7] 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`. This requires a somewhat different translation to C (which we do not have space to describe in full) and which takes into account this possibility by bypassing the emulator loop, in several senses similarly to what is presented in [28].

An Example — the `fact/2` Predicate with program information: Let us assume that it has been inferred that `fact/2` (Figure 3) is always called with its first argument instantiated to an integer and with a free variable in its second argument. This information is written in the assertion language for example as:[8]

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

which reflects the types and modes of the calls and successes of the predicate. That information is also propagated through the normalized predicate producing the annotated program shown in Figure 7, where program-point information is also shown.

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

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

Fig. 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 improved information — but note that the representation is homogeneous with respect to

[7] 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.

[8] The `true` prefix implies that this information is to be *trusted and used*, rather than to be *checked* by the compiler. Indeed, we require the stated properties to be correct, and `ciaocc` does not check them: this is a task delegated to CiaoPP. Wrong *true* assertions can, therefore, lead to incorrect compilation. However, the assertions generated by CiaoPP are guaranteed correct by the analysis process.

the "no information" case. The impact of type information in the generation of low-level code can be seen in Figure 5. Instructions inside the dashed boxes are removed when type information is available, and the (arithmetic) builtins enclosed in rectangles are replaced by calls to specialized versions which work with 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.

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 (1.46)	390 (1.59)	356 (1.74)
Average Speedup		(1.46 – 1.43)	(1.88 – 1.77)	(3.18 – 2.34)

Table 4. Bytecode emulation vs. unoptimized, optimized (types), and optimized (types and determinism) compilation to C. *Arithmetic – Geometric* means are shown.

4 Performance Measurements

We have evaluated the performance of a set of benchmarks executed by emulated bytecode, translation to C, and by other programming systems. The benchmarks, while representing interesting cases, 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. Since parts of the compiler are still in an experimental state, we have not been able to use larger benchmarks yet. All the measurements have been performed on 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:

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

A summary of the results appears in Table 4. The figures between parentheses in the first column is the number of repetitions of each benchmark. The second column contains the execution times of programs run by the Ciao bytecode emulator. The third column corresponds to programs compiled to C without

compile-time information. The fourth and fifth columns correspond, respectively, to the execution times when compiling to C with type and type+determinism information. The numbers between parentheses are the speedups relative to the bytecode version. All times are in milliseconds. Arithmetic and geometric means are also shown in order to diminish the influence of exceptional cases.

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 – 1.82

Table 5. Speed of other Prolog systems and Mercury

Table 5 shows the execution times for the same benchmarks in five 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 is not really to compare directly with them, because a different underlying technology and external information is being used, 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. In principle, comparable optimizations could be made in these systems. 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 Mercury [29] (version 0.11.0). Strictly speaking the Mercury compiler is not a Prolog compiler: the source language is substantially different from Prolog. But Mercury has enough similarities to be relevant and its performance represents an upper reference line, given that the language was restricted in several ways to allow the compiler, which generates C code with different degrees of "purity", to achieve very high performance by using extensive optimizations. Also, the language design requires the necessary information to perform these optimizations to be included by the programmer as part of the source. Instead, the approach that we use in Ciao is to infer automatically the information and not restricting the language.

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

- The simple compilation scheme generates clean, portable, "trick-free" C (some compiler dependent extensions would speed up the programs). The execution profile is very near to what the emulator would do.
- As noted in Section 2, the C compiler makes the fetch/switch loop of the emulator a bit cheaper than the C execution loop. We have identified this as a cause of the poor speedup of programs where recursive calls dominate

the execution (e.g., `factorial`). We want, of course, to improve this point in the future.
- The increment in size of the program (to be discussed later — 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 can use optimized versions where several checks have been removed. In some of these cases the functions which implement arithmetic operations are simple enough as to be inlined by the C compiler — an added benefit which comes for free from compiling to an intermediate language (C, in this case) and using tools designed for it. This is, for example, the case of `queens`, in which it is known that all the numbers involved are integers. Besides the information deduced by the analyzer, hand-written annotations stating that the integers involved fit into a machine word, and thus there is no need for infinite precision arithmetic, have been manually added.[9]

Determinism information often (but not always) improves the execution. The Takeuchi function (`tak`) is an extreme case, where savings in choicepoint generation affect execution time. While the performance obtained is still almost a factor of 2 from that of Mercury, the results are encouraging since we are dealing with a more complex source language (which preserves full unification, logical variables, cuts, `call`/1, database, etc.), we are using a portable approach (compilation to standard C), and we have not yet applied all possible optimizations.

A relevant point is to what extent a sophisticated analysis tool is useful in practical situations. The degree of optimization chosen can increase the time spent in the compilation, and this might preclude its everyday use. We have measured (informally) the speed of our tools in comparison with the standard Ciao Prolog compiler (which generates bytecode), and found that the compilation to C takes about three times more than the compilation to bytecode. A considerable amount of time is used in I/O, which is being performed directly from Prolog, and which can be optimized if necessary. Due to a well-developed machinery (which can notwithstanding be improved in a future by, e.g, compiling CiaoPP itself to C), the global analysis necessary for examples is really fast and never exceeded twice the time of the compilation to C. Thus we think that the use of global analysis to obtain the information we need for `ciaocc` is a practical option already in its current state.

Table 6 compares object size (in bytes) of the bytecode and the different schemes of compilation to C and using the same compiler options in all cases. While modern computers usually have a large amount of memory, and program size hardly matters for a single application, users stress computers more and more by having several applications running simultaneously. On the other hand, program size does impact their startup time, important for small, often-used commands. Besides, size is still very important when addressing small devices with limited resources.

[9] This is the only piece of information used in our benchmarks that cannot be currently determined by CiaoPP. It should be noted, though, that the absence of this annotation would only make the final executable less optimized, but never incorrect.

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 – 6.32)	(5.43 – 4.94)	(8.77 – 7.14)

Table 6. Compared size of object files (bytecode vs. C) including *Arithmetic - Geometric* means.

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 with respect to the bytecode can be as large as 15× when translating to C without optimizations, and the average case sits around a 7-fold increase. This increment is partially due to repeated code in the indexing mechanism, which we plan to improve in the future.[10] Note that, as our framework can mix bytecode and native code, it is possible to use both in order to achieve more speed in critical parts, and to save program space otherwise. Heuristics and translation schemes like those described in [30] can hence be applied (and implemented as a source to source transformation).

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

Size, in general, decreases when using type information, as many runtime type tests are removed, the average size being 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. Inlining was left to the C compiler; experiments show that more aggressive inlining does not necessarily result in better speedups.

It is interesting to note that some optimizations used in the compilation to C would not give comparable results when applied directly to a bytecode emulator. For example, a version of the bytecode emulator hand-coded to work 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.

[10] In all cases, the size of the bytecode emulator / runtime support (around 300Kb) has to be added, although not all the functionality it provides is always needed.

5 Conclusions and Future Work

We have reported on the scheme and performance of `ciaocc`, a Prolog-to-C compiler which uses type analysis and determinacy information to improve code generation by removing type and mode checks and by making calls to specialized versions of some builtins. We have also provided performance results. `ciaocc` is still in a prototype stage, but it 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, using the type and determinacy information inferred by CiaoPP. This code is finally translated into C by the compiler back-end. The intermediate code makes the final translation step easier and will facilitate developing new back-ends for other target languages.

We have found that optimizing a WAM bytecode emulator is more difficult and results in lower speedups, due to the larger granularity of the bytecode instructions. The same result has been reported elsewhere [2], although some recent work tries to improve WAM code by means of local analysis [20].

We expect to also be able to use the information inferred by CiaoPP (e.g., determinacy) to improve clause selection and to generate a better indexing scheme at the C level by using hashing on constants, instead of the linear search used currently. We also 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. Colmerauer, A.: The Birth of Prolog. In: Second History of Programming Languages Conference. ACM SIGPLAN Notices (1993) 37–52
2. Van Roy, P.: 1983-1993: The Wonder Years of Sequential Prolog Implementation. Journal of Logic Programming **19/20** (1994) 385–441
3. Pereira, F.: C-Prolog User's Manual, Version 1.5, University of Edinburgh. (1987)
4. Warren, D.: An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025 (1983)
5. Ait-Kaci, H.: Warren's Abstract Machine, A Tutorial Reconstruction. MIT Press (1991)
6. Taylor, A.: High-Performance Prolog Implementation. PhD thesis, Basser Department of Computer Science, Unversity of Sidney (1991)
7. Krall, A., Berger, T.: The VAM_{AI} - an abstract machine for incremental global dataflow analysis of Prolog. In de la Banda, M.G., Janssens, G., Stuckey, P., eds.: ICLP'95 Post-Conference Workshop on Abstract Interpretation of Logic Languages, Tokyo, Science University of Tokyo (1995) 80–91
8. Van Roy, P., Despain, A.: High-Performace Logic Programming with the Aquarius Prolog Compiler. IEEE Computer Magazine (1992) 54–68
9. Swedish Institute for Computer Science PO Box 1263, S-164 28 Kista, Sweden: SICStus Prolog 3.8 User's Manual. 3.8 edn. (1999) Available from `http://www.sics.se/sicstus/`.
10. Mariën, A.: Improving the Compilation of Prolog in the Framework of the Warren Abstract Machine. PhD thesis, Katholieke Universiteit Leuven (1993)

11. Diaz, D., Codognet, P.: Design and Implementation of the GNU Prolog System. Journal of Functional and Logic Programming **2001** (2001)
12. Jones, S.L.P., Ramsey, N., Reig, F.: C--: A Portable Assembly Language that Supports Garbage Collection. In Nadathur, G., ed.: International Conference on Principles and Practice of Declarative Programming. Number 1702 in Lecture Notes in Computer Science, Springer Verlag (1999) 1–28
13. Codognet, P., Diaz, D.: WAMCC: Compiling Prolog to C. In Sterling, L., ed.: International Conference on Logic Programming, MIT PRess (1995) 317–331
14. Quintus Computer Systems Inc. Mountain View CA 94041: Quintus Prolog User's Guide and Reference Manual—Version 6. (1986)
15. Santos-Costa, V., Damas, L., Reis, R., Azevedo, R.: The Yap Prolog User's Manual. (2000) Available from `http://www.ncc.up.pt/~vsc/Yap`.
16. Demoen, B., Nguyen, P.L.: So Many WAM Variations, So Little Time. In: Computational Logic 2000, Springer Verlag (2000) 1240–1254
17. Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-García, P., Puebla, G.: 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) (2002) System and on-line version of the manual available at `http://clip.dia.fi.upm.es/Software/Ciao/`.
18. Puebla, G., Hermenegildo, M.: Abstract Specialization and its Applications. In: ACM Partial Evaluation and Semantics based Program Manipulation (PEPM'03), ACM Press (2003) 29–43 Invited talk.
19. Winsborough, W.: Multiple Specialization using Minimal-Function Graph Semantics. Journal of Logic Programming **13** (1992) 259–290
20. Ferreira, M., Damas, L.: Multiple Specialization of WAM Code. In: Practical Aspects of Declarative Languages. Number 1551 in LNCS, Springer (1999)
21. Mills, J.: A high-performance low risc machine for logic programming. Journal of Logic Programming (6) (1989) 179–212
22. Taylor, A.: LIPS on a MIPS: Results from a prolog compiler for a RISC. In: 1990 International Conference on Logic Programming, MIT Press (1990) 174–189
23. Hermenegildo, M., Greene, K.: The &-Prolog System: Exploiting Independent And-Parallelism. New Generation Computing **9** (1991) 233–257
24. Puebla, G., Bueno, F., Hermenegildo, M.: An Assertion Language for Constraint Logic Programs. In Deransart, P., Hermenegildo, M., Maluszynski, J., eds.: Analysis and Visualization Tools for Constraint Programming. Number 1870 in LNCS. Springer-Verlag (2000) 23–61
25. Hermenegildo, M., Puebla, G., Bueno, F., López-García, P.: Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In: 10th International Static Analysis Symposium (SAS'03). Number 2694 in LNCS, Springer-Verlag (2003) 127–152
26. Cabeza, D., Hermenegildo, M.: A New Module System for Prolog. In: International Conference on Computational Logic, CL2000. Number 1861 in LNAI, Springer-Verlag (2000) 131–148
27. Van Roy, P.: Can Logic Programming Execute as Fast as Imperative Programming? PhD thesis, Univ. of California Berkeley (1990) Report No. UCB/CSD 90/600.
28. Henderson, F., Somogyi, Z.: Compiling Mercury to High-Level C Code. In Nigel Horspool, R., ed.: Proceedings of Compiler Construction 2002. Volume 2304 of LNCS., Springer-Verlag (2002) 197–212
29. Somogyi, Z., Henderson, F., Conway, T.: The execution algorithm of Mercury: an efficient purely declarative logic programming language. JLP **29** (1996)
30. Tarau, P., De Bosschere, K., Demoen, B.: Partial Translation: Towards a Portable and Efficient Prolog Implementation Technology. Journal of Logic Programming **29** (1996) 65–83