Towards Description and Optimization of Abstract Machines in an Extension of Prolog

José F. Morales[1], Manuel Carro[2], and Manuel Hermenegildo[2,3]

[1] U. Complutense de Madrid (UCM)
jfmc@fdi.ucm.es
[2] T. University of Madrid (UPM)
{mcarro,herme}@fi.upm.es
[3] U. of New Mexico (UNM)
herme@unm.edu

Abstract. Competitive abstract machines for Prolog are usually large, intricate, and incorporate sophisticated optimizations. This makes them difficult to code, optimize, and, especially, maintain and extend. This is partly due to the fact that efficiency considerations make it necessary to use low-level languages in their implementation. Writing the abstract machine (and ancillary code) in a higher-level language can help harness this inherent complexity. In this paper we show how the semantics of basic components of an efficient virtual machine for Prolog can be described using (a variant of) Prolog which retains much of its semantics. These descriptions are then compiled to C and assembled to build a complete bytecode emulator. Thanks to the high level of the language used and its closeness to Prolog the abstract machine descriptions can be manipulated using standard Prolog compilation and optimization techniques with relative ease. We also show how, by applying program transformations selectively, we obtain abstract machine implementations whose performance can match and even exceed that of highly-tuned, hand-crafted emulators.

Keywords: Prolog, Abstract Machines, Compilation, Optimization, Program Transformation.

1 Introduction

Designing and implementing competitive "abstract" (or "virtual") machines is not without difficulties. In particular, the extensive code optimizations required for performance make development and, especially, maintenance and further modification non-trivial. Implementing or testing new optimizations is often involved, as decisions previously taken need to be revisited and low level and tedious recoding is often necessary to test a new idea.

Systems based on virtual machines are typically composed of a compiler from the source language ($\mathcal{L}_P$) to bytecode language ($\mathcal{L}_B$, aimed at being fast to interpret, for which an intermediate-level symbolic representation $\mathcal{L}_A$ usually exists), plus an emulator for $\mathcal{L}_B$ written in a lower-level language $\mathcal{L}_C$. In our particular case, $\mathcal{L}_P$ is Prolog, $\mathcal{L}_A$ is symbolic WAM code, and $\mathcal{L}_C$ is C.

Complexity of virtual machines and low level of $\mathcal{L}_C$ has led to several proposals in order to raise the level at which the virtual machine is written, while trying to maintain the possibility of translating it to the in principle more efficient $\mathcal{L}_C$ language. A particularly interesting possibility when $\mathcal{L}_P$ is a general-purpose language (as in our case) is to use $\mathcal{L}_P$ itself to write its virtual machine. This has been done for example in JavaInJava [1] and PyPy [2]. However, making these implementations competitive with existing hand-tuned abstract machines is undoubtedly a challenge: JavaInJava reports initial slowdowns of approximately 700 times w.r.t. then-current implementations, and PyPy started at the 2000$\times$ slowdown level.

This slowdown is largely due to the "semantic gap" existing between $\mathcal{L}_P$ and $\mathcal{L}_C$, even in the case of imperative and O.O. languages such as Java or Python. $\mathcal{L}_P$ should be precise enough to describe the algorithms underlying the basic operations of the abstract machine with, at most, a constant slowdown (i.e., with no penalty regarding computational complexity). In order to achieve this, and in addition to using improved compilation technology, we made changes to the initial $\mathcal{L}_P$ in the form of extensions which make it easier to reflect (or control) $\mathcal{L}_C$ characteristics not originally available such as, e.g., data sizes, alignments, unboxing, etc. We will refer to this extended version of $\mathcal{L}_P$ as $\mathcal{L}_I$. A similar approach has made it possible to, for example, reduce the slowdown of PyPy to $3.5\div - 11.0\div$ in more recent versions [2].

The approach of coding completely the whole abstract machine in $\mathcal{L}_P$ or $\mathcal{L}_I$ at once has the disadvantage of making it almost inevitable (as illustrated by, e.g., PyPy) to start from a large slowdown and then work slowly towards regaining performance. This makes it difficult to use the generated virtual machines as "production" software (which would therefore be routinely tested) and, especially, it makes it difficult to study how a certain optimization will carry over to a complete, optimized abstract machine.

We propose herein another possibility which is to proceed the other way around by starting from a highly optimized abstract machine, keeping some key elements coded in $\mathcal{L}_C$ and gradually replacing different pieces of code with code written in $\mathcal{L}_I$, making sure that no performance is lost at each step. In our implementation, and following this approach, we have chosen to generate the bytecode fetching and decoding loop directly in $\mathcal{L}_C$ using the emulator generator of [3]. This automates the generation of efficient emulators, makes devising and generating bytecode easy, and, notwithstanding, it makes it possible to write the definitions of the abstract machine instructions in $\mathcal{L}_I$. This is not at odds with compilation to native code and just-in-time systems, where a sizable part of the emulator machinery is still there in the form of runtime libraries.

We started with an efficient, WAM-based abstract machine for Prolog initially coded in C and we rewrote parts of it in a variant of Prolog ($\mathcal{L}_I$) which we have termed ImProlog and which both extends and restricts Prolog. ImProlog can be translated into very efficient C and at the same time its semantics is close enough to Prolog so as to be able to reuse many compilation techniques (certain analyses, specialization, etc.). This allows obtaining highly optimized and specialized emulators while avoiding obscure, redundant implementations or overuse of C macros. In addition, the combination of this approach with an emulator generator makes it possible to carry out non-trivial optimizations, such as instruction merging, automatically.

2 A Prolog Variant to Describe Virtual Machines

In this section we will describe our $\mathcal{L}_I$ language, ImProlog, and the analysis and code generation techniques used to produce highly efficient code from it.

2.1 New Features in the Language

ImProlog adds two features to Prolog that can be modeled as new language constructs (expressible, however, within standard Prolog):

Native types and operations on them: They are opaque ("hidden" types in terms of the Ciao module system and assertion language [4]), and used to reflect in $\mathcal{L}_I$ the basic data representations of $\mathcal{L}_C$ and the data types required by the abstract machine (e.g., integers, floats, tagged words, etc.).

Mutable variables (mutvars): They associate an identifier (which can be any first-order *ground* term) with an arbitrary term.

Two operations are defined over mutable variables:

Access: $@MutVar$ acts as a *function* which returns the value previously stored in $MutVar$.

Assignment: $MutVar <= Value$ assigns $Value$ to the identifier $MutVar$. The assignment is imperative and non-backtrackable. If $MutVar$ is a free variable then a new, unique identifier is allocated for it. If it is a ground term, it is used as identifier. Its behavior remains unspecified otherwise.

Figure 1 shows an example of ImProlog code which defines how to dereference a variable to reach a term. Similarly to the standard algorithm, it follows a reference chain and stops when the value pointed to is the same as the pointing term. Note the use of mutable variables and the operations on native types `tagof/2` and `tagval/2`, which check the tag of a tagged word and retrieve the value of the tagged word, respectively.

The extensions included in ImProlog can easily be defined in full Prolog, as shown in Figure 2 (we assume that `new_id/1` returns a new, unique identifier in

```
deref(Reg) :-
    ( tagof(@Reg,ref) ->
        tagval(@Reg,V), T = @V,
        ( @Reg = T -> true
        ; Reg <= T, deref(Reg) )
    ; true ).
```

Fig. 1. Dereference operation

```
tagof(tagged(Tag,Val),Tag).
tagval(tagged(Tag,Val),Val).
:- dynamic (@)/2.
Id <= V :-
 ( var(Id) -> new_id(Id) ; true ),
 retract(@(Id,_)), assertz(@(Id,V)).
```

Fig. 2. Prolog semantics of extensions

each call and that a trivial syntactic transformation makes goals `@(X, Y)` and `Y = @X` equivalent). As `@/2` can be expressed in Prolog, we would not need any additional machinery to write (and run) our virtual machine in a Prolog system and as a Prolog program, should we want to make that experiment. But that would clearly not be without an immense performance penalty (at least without complex optimizations), which is against our initial aims. By making these new constructs natively known by the compiler, and restricting their application to the cases which are useful to describe the virtual machine, we can compile them efficiently time- and memory-wise, and they become easy to map onto low-level primitive constructs of $\mathcal{L}_C$.

2.2 Conditions to Ensure Efficient Code Generation

As shown in [5,6,7] and other work (see [8] for more references), generation of highly efficient executables from logic programs heavily depends on reducing the computational overhead that supports the extended semantic capabilities of Prolog for the specific cases in which the full power of the language is not needed. This generally requires a wealth of compile-time information regarding types, modes, determinism, non-failure, and other properties of the program.

This information is generally inferred by means of static analysis.[1] When such information can be inferred, optimizations are performed, and less efficient code is generated otherwise. However, since our initial goal was to *ensure* efficiency, we will, instead of allowing the generation of suboptimal code, impose a number of constraints on the ImProlog code that can be written when describing the abstract machine: precisely those that will allow an almost direct (often one-to-one) translation to $\mathcal{L}_C$ code. The compiler will raise an (efficiency-related) error while processing the code that describes the virtual machine and abort its generation if the necessary conditions are not met. This is obviously too drastic a solution for general programs, but a good compromise in our application.

Program analysis combined with program assertions allows the compiler to identify when it is safe (or possible) to generate code based on these constraints. The conditions that must hold after analysis are that code must be deterministic (with optional support for failure continuations, as in `if-then-else` constructs,

[1] It can be also provided by program annotations written by the user, which will indeed be necessary in some cases in practice.

but not for full non-determinism), and that no garbage collection, trailing, or boxing should be required. The analyses used to ensure that those restrictions hold are listed in the next section.

2.3 Analysis

Following the order in which they are applied in the compiler, the analyses used can be divided into three main groups.

Traditional Prolog Analyses: These include analyses for types, modes, determinism, and non-failure. They are instrumental to decide the best data representation and to detect which pieces of code may require choice points or failure continuations. They are performed using the abstract interpretation-based analyzer in CiaoPP [9]. As CiaoPP was designed with extensibility in mind, knowledge about ImProlog native types and associated operations can be given to CiaoPP via (Ciao) assertions, without having to actually change the analyzer. Assertions are also used to state the types, modes, etc. of externally defined facilities and routines (so that they can be taken into account by the analyzers) and to declare properties to be met at the entry point of each abstract machine instruction, which is typically written as a predicate. This information includes implementation decisions such as the use of short or long native integers, etc.

In addition to assertions, the type of some mutable variables may be further restricted by knowledge about the location they refer to or by type-constraining program calls. For example, mutables for `X(i)` registers are always bound to elements of type 'tagged'. A typed specification of the assignment operation could be written as follows:

```
Id <= Val :- id_type(Id, Type), Type(Val),
             retract(@(Id, _)), assertz(@(Id, Val)).
```

where `id_type/2` relates an identifier with the name of its type, and `Type(Val)` is a higher-order call which states the type of `Val`. As we will see later, this knowledge helps in unboxing and analysis of mutables. Type analysis can ensure that `Type(Val)` always holds and it can therefore be harmlessly removed. This additional information makes mapping to C much easier.

Imperative State Analysis: Analysis of the value of mutable variables requires tracking their (imperative) state, which is updated using rules that reflect the actual operational semantics (i.e., sequential execution of OR-alternatives, etc.). Since $\mathcal{L}_I$ programs are limited to the deterministic case, the complexity of this analysis is reduced with respect to a more general case. The domains used are precise enough to identify an abstraction of some properties of mutable variables (e.g., whether they represent an `X` register, a `Y` register, a heap location, etc.). Strict type restrictions for some identifiers are applied here, thereby increasing the performance of the analysis. The analysis is conservative: every time a mutable may be written to (directly or by code which is externally available, and therefore difficult of impossible to access and analyze) its state is set to the *top* value of the domain lattice. Different mutable variables may be aliased (i.e., they can point to the same location), and only a limited alias analysis is performed; it takes advantage of the knowledge of the compiler regarding the memory location of the variable: e.g., a mutable variable living in `X(0)` cannot share with a mutable variable living in `Y(1)`. This simple approach was effective enough for the purpose of this work.

Analysis for Unboxing: This analysis tries to determine whether the type of some variable is known at all points where it is reachable. If so, then there is no need to reserve space for a tag to check its type at runtime. This requires a previous pass to determine the *scope* of the identifiers for mutable variables in order to establish in which program points they may be accessed. This is also needed in order to assign memory locations at compile time to the mutable variables created within the body of a predicate and which are not allocated on the heap. Since non-determinism is not allowed, and according to the compilation scheme we follow, if a variable name cannot be reached outside the scope of a predicate it can be safely mapped to a (local) C variable. A conservative approximation, which is easier to check and precise enough, is the following: the variable name can be read from, assigned to, and passed as argument to other predicates, but it cannot be assigned to anything else than other local variables.

2.4 Code Generation

The information provided by the analysis is used to optimize code generation, especially in order to partially evaluate away whole sections of code (e.g., simplifying conditionals, reducing calls to true/noop, etc.). The algorithm extends that of `ciaocc` [7] to support ImProlog and also simplifies it in view of the constraints on the code specified in Section 2.2.

Predicates that may or may not fail are mapped to C functions with boolean or void return types, respectively. Generation of code for several clauses or predicates in the same C function and jumping to C labels is also supported (e.g., to transform recursions into loops). Additionally, an interface to internal compiler modules is provided. This makes it possible to invoke instruction compilation from within the emulator generator.

Schematically, compilation distinguishes among control constructs, external C functions, and builtins. Compilation of control is as follows:

- A block `G1, G2` is translated to the code for `G1` having its success continuation pointing to `G2`, followed by the code for `G2`.
- The construct `G1 -> G2 ; G3` is compiled into an `if-then-else`, where `G1` is compiled in a context where the failure continuation points to `G3`. `G2` and `G3` are compiled in the same context where the whole construct appeared (i.e., success / failure continuations point to where `G1 -> G2 ; G3` did).

For a goal `G` which calls a C function `f()`, arguments are compiled (see later) and then `f()` is called. If the predicate is semi-deterministic, the emitted code checks the return code and, if necessary, a jump to the failure continuation is made. When `G` corresponds to a built-in, its compilation proceeds as follows:

- `true` does nothing.
- `fail` is translated to a jump to the failure continuation.
- `A <= B` is translated into assignment instructions. If `A` was not initialized it is declared.
- `A = B` is handled as follows:
  - When `A` is unbound and `B` is ground (and also for the symmetrical case), the builtin is translated into the declaration of `A` plus an assignment statement that moves the value of executing the compiled code corresponding to `B` to the memory location associated with `A`.
  - When `A` and `B` are both ground, the builtin is translated into a comparison of the values resulting from executing the compiled code of both expressions.

Note that although full unification may be assumed during program transformations, it is ultimately reduced to the two cases above. This has to be possible in order to avoid bootstrapping problems: e.g., (full) unification, also defined in ImProlog, should not be based itself on a full unification built-in.

Prolog logical variables and mutable variables are mapped to C variables (which can be global, local, or be passed as function arguments). The type of those C variables is extracted from the declarations and using type inference. Due to the determinism of ImProlog, trailing is unnecessary.

During compilation a symbol table keeps track of the type and memory location (or C variable) associated to each variable. All variables have to have an associated type in order to perform unboxing (an error is flagged otherwise), and all types are either native types or mutables whose value is of a native type. For a variable whose associated C type is `Tc`, a declaration of variable named `V`, with C type `Vt`, is emitted, and the associated memory location is set to `Mem`, as follows:

- If the variable is not mutable, `Vt` is `Tc` and `Mem` is `V`.
- If the variable is mutable:
  - if its scope is local, then `Vt` is `Tc` and `Mem` is `V`, or
  - `Vt` is `(Tc *)` and `Mem` is `*V`, otherwise.

For simplicity we assume that goal arguments have been normalized and only variables or `@` expressions appear. Compilation of arguments, assuming that the memory location for `A` is `Mem`, is as follows:

- `@A` is translated to `Mem` (and `A` must be a mutable variable in this case).
- `A` is translated to `&Mem` (if `A` is mutable), or
- `A` is translated to `Mem` otherwise.

3 Generating Emulators with ImProlog

We now sketch how WAM instructions can be described using ImProlog and how the full emulator is assembled using a generic abstract machine generator.

3.1 Defining WAM Instructions in ImProlog

The definition of every WAM instruction in ImProlog looks just like a regular predicate, and the types, modes, etc. of each of their arguments have to be declared using (Ciao) assertions. Figure 3 shows the definition of an instruction which tries to unify a term and a constant. The `pred/1` declaration states that the first argument is a mutable variable and that the second is a tagged word containing a constant. The predicates `deref/1` (from Figure 1) and `bind/2` (also a defined predicate) are used in the instruction definition.

```
:- pred u_cons(mutable, cons).
u_cons(A, Cons) :-
    T <= @A, deref(T),
    ( tagof(@T, ref) -> bind(@T,Cons) ; @T = Cons ).
```

Fig. 3. Unification with a constant

The general compilation process to C, described later, is able to unfold (if so desired) the definition of the predicates called by `u_cons/2` and to propagate information from the code inside the instruction in order to optimize the resulting piece of the emulator. After the set of transformations instruction definitions are subject to, the generated C code is of high quality.

Our approach has been to define a reduced number of instructions (50 is a ballpark figure) and let the merging and specialization process (see Section 4) generate all instructions needed to have a competitive emulator. Note that efficient emulators tend to have a large number of instructions (hundreds or even thousands) and many of them are variations (obtained through specialization, merging, etc.) on common blocks [10,11]. These common blocks are the simple instructions we aim at representing explicitly in ImProlog.

In the experiments we performed (Section 5) the emulator with a larger number of instructions had 199 different opcodes (not counting those which result from padding some other instruction with zeroes to ensure a correct alignment in memory). Starting with a simple instruction set makes it easier to maintain instruction sets and to make sure that they are consistent. Complex instructions are generated automatically in a (by construction) correct way.

3.2 Assembling the Emulator

To avoid the burden associated with the coding and $\mathcal{L}_C$-dependent details of the emulator, we chose to use here the framework previously described in [3], where instruction semantics and bytecode representation are independently handled and assembled together using an emulator compiler. Using the terminology of [3] we define the relation between $\mathcal{L}_A$ and $\mathcal{L}_B$ by means of several pieces:[2]

[2] A complete description, not included due to space constraints, would detail all expected elements for a WAM: `X` and `Y` registers, atoms, numbers, functors, etc.

$\mathcal{M}_{enc}$ which declares how bytecode encodes $\mathcal{L}_A$ instructions and data (e.g. `X(0)` is encoded as the number `0`).

$\mathcal{M}_{dec}$ which declares how bytecode should be decoded to return the initial instruction format in $\mathcal{L}_A$ (e.g., for an instruction which uses as argument an `X` register, a `0` means `X(0)`).

$\mathcal{M}_{arg}$ which expresses how $\mathcal{L}_A$ expressions are translated to $\mathcal{L}_C$, e.g., how `X(0)` goes to `x[0]` (assuming `X` registers end up in an array).

Higher-level instruction definitions in $\mathcal{L}_I$ (which abstract away bytecode representation issues) and program assertions are processed to generate:

$\mathcal{M}_{def}$ which contains the definition of each instruction in the language $\mathcal{L}_A$ in terms of $\mathcal{L}_C$ code.

$\mathcal{M}_{ins'}$ which describes the instruction set with opcode numbers and the format of each instruction, i.e., the type in $\mathcal{L}_A$ for each instruction argument.

The instruction set $\mathcal{M}_{ins'}$ is generated by reading the information for each instruction contained in the assertions, interpreting types as $\mathcal{L}_A$ elements, and assigning opcodes to each instruction, either automatically or via user annotations. The definition of $\mathcal{M}_{def}$ is based on cgen, that generates $\mathcal{L}_C$ code from $\mathcal{L}_I$ as defined in Figure 4. In this figure, *mem_storage* stands for a look-up table which relates each $\mathcal{L}_A$-level variable arg_i with its type and location in $\mathcal{L}_C$, a_i. The pseudo-instruction $failure\_ins$ takes care of causing a failure. Some $\mathcal{L}_A$ instructions are not supposed to fail (e.g., pushing a choicepoint), while others, such as performing a unification, can fail. In the former case cgen is able to discard the *else* part and simplify the *then* part; in the latter case, jumps to $failure\_ins$ are inserted in the appropriate places.

The components $\mathcal{M}_{enc}$ and $\mathcal{M}_{ins'}$ are used to generate the $\mathcal{L}_A$ to $\mathcal{L}_B$ compiler back-end. The rest of the components and $\mathcal{M}_{ins'}$ are used by the emulator compiler. The emulator has to understand $\mathcal{L}_B$ and therefore it has to agree in its format with what the compiler back-end emits. Note that the overall emulator structure is largely independent of the code of the instructions.[3] A summarized definition of the emulator compiler and how it uses the different pieces in $\mathcal{M}$ can be found in Figure 4. The scheme of the generated emulator code is somewhat similar to what the Janus compilation scheme [12] produces, although in the Janus case the continuation to every call (in the source code) is known statically. The compiler can therefore generate a direct jump to a fixed label, while in our case the continuation can in principle be any program point which comes from the bytecode program itself and is not known until the emulator is being executed.

Example 1. Code for a specialized instruction. From the instruction in Figure 3, which unifies a term living in some variable with a constant, we can derive a specialized version in which the term is supposed to live in an `X` register. The declaration:

[3] Assuming that no global transformations are done, which we are not addressing here.

```
$emucomp(\mathcal{M}) =$
  $[\,\mathbf{emu}_{\mathrm{B}}(p, prg) \equiv$
    case $get\_opcode(p, prg)$ of
      $opcode_1 : inscomp(opcode_1, \mathcal{M})$
      $\ldots$
      $opcode_n : inscomp(opcode_n, \mathcal{M})]$
    where $opcode_i \in domain(\mathcal{M}_{ins'})$
```

```
$inscomp(opcode, \mathcal{M}) =$
  $[\mathcal{M}_{def}(p', cont, name, \mathcal{M}_{args}(args));\ \ cont(p')]$
where
  $\langle name, format\rangle = \mathcal{M}_{ins'}(opcode)$
  $\langle args, p'\rangle = decode_{ins}(format, [p], [prg], \mathcal{M})$
  $cont = \lambda a.[\mathbf{emu}_{\mathrm{B}}(a, prg);\ \ return]$

$\mathcal{M}_{def}(next, cont, name, [arg1, \ldots, arg_n]) =$
  $[\![\mathsf{cgen}]\!](name(a_1, \ldots, a_n) \rightarrow true; failure\_ins)$
where $mem\_storage[a_1 : arg_1, \ldots, a_n : arg_n]$
```

Fig. 4. Emulator compiler

```
loop:
  switch(Op(short,P,0)) {
  ...
  case 97: goto ux_cons;
  ...
  }
```

```
void deref(tagged_t *a0) {
deref:
  if (tagged_tag(*a0) == REF) {
    tagged_t t0;
    t0 = *(tagged_val(*a0));
    if ((*a0) != t0) {
      *a0 = t0;
      goto deref; }}}
```

```
...
ux_cons:
  tagged t;
  t = X(Op(short,P,2));
  deref(&t);
  if (TagOf(t) == REF) {
    bind(t, Op(tagged,P,4));
  } else {
    if (t != Op(tagged,P,4))
        goto failure_ins;
  }
  P = Skip(P,8);
  goto loop;
  ...
```

Fig. 5. Schema for the code generated for a simple instruction

```
:- ins_alias(ux_cons, u_cons(xreg_mutable, any)).
```

states precisely that, assigns the (symbolic) name `ux_cons` to the new instruction, and specifies that the first argument lives in an `X` register. The declaration:

```
:- ins_entry(97, ux_cons).
```

indicates that the emulator has an entry with opcode 97 for that instruction.[4] Figure 5 shows the code generated for the instruction (right) and a fragment of the emulator generated by the emulator compiler in Figure 4.

We want to note that we deliberately stay within standard C: the use of C extensions (such as storing labels in variables, which are provided by `gcc` and used, for example, in [13,14]), is outside the scope of this paper.

[4] In fact, different assignments of instruction numbers to opcodes can have an impact on the final performance, as they dictate how the code is laid out in the emulator switch. This affects, for example, the behavior of the cache.

4 Automatic Generation of Abstract Machine Variations

Substantial work has been devoted to abstract machine generation strategies such as, e.g., [10,11], which explore different design variations with the objective of achieving highly optimized emulators. By making the semantics of the abstract machine instructions explicit in a language like ImProlog, which can be easily processed automatically, such variations can be formulated mostly as automatic transformations. Adding new transformation rules and testing them together with the existing ones becomes a relatively easy task.

We will briefly describe some of these transformations, which will be experimentally evaluated in Section 5. Each transformation is identified by a two-letter code. We make a distinction between transformations which change the instruction set (e.g., creating new instructions) and those which only affect the way code is generated.

4.1 Instruction Set Transformations

New instructions are currently synthesized from existing ones by explicitly unfolding shared pieces of code, by merging instructions (different or not), and by performing specialization for some operand values, types, or locations.

Instruction Merging [om]: Merging generates larger instructions from sequences of smaller ones, and aims at saving fetch cycles at the expense of an increased `switch` size. This technique has been used extensively in high-performance systems (e.g., Quintus Prolog, SICStus, Yap, etc.). The performance of different combinations has been studied empirically [10], but in that paper new instructions were generated by hand, although deciding which instructions had to be created was done by means of profiling. In our framework all that is needed in order to emit code for a merged instruction is a single declaration. Merging is done automatically through code unfolding based on the definitions of the component instructions. This makes it possible to define a set of optimal user rules for merging.

Instructions with a Variable Number of Operands [vo]: For some instruction families a number of instructions (e.g., `unify with void`) can be collapsed into a single instruction with a variable number of operands. Code generation emits a loop whose internal iteration code comes directly from the single instruction definition.

Instructions for Built-ins [ib]: Calling external library code or built-ins often requires ad-hoc instructions (to make the appropriate parameter conversion, etc.). A single family of instructions that call a foreign C function can be used to do that, and this is the default option. The same instruction can then be specialized for a predefined set of built-ins, thus generating a special instruction set that includes faster calls to, e.g., arithmetic operations.

4.2 Transformations of Instruction Code

Some transformations do not create new instructions, but perform instead different optimizations on already existing instructions by manipulating the code or choosing alternative translation schemes.

Unfolding Rules [ur]: Simple predicates are unfolded throughout the code before compilation. In the case of instruction merging, unfolding is used to merge the code of two or more instructions into a single piece of code. In some cases unfolding can be limited so that common pieces of instructions can be shared. This transformation enables or disables a set of predefined unfolding rules.

Different Tag Switching Schemes [ts]: Tags are used to detect dynamically the type of basic data (atom, structure, number, variable, etc.) contained in a machine word, so that different actions can be taken depending on this type. The corresponding *tag switching code* is a heavily-used operation which is worth optimizing as much as possible. This option generates either an automatic C switch (when enabled) or a set of predefined switch patterns based on tag encodings (when disabled).

Connected Continuations [jc]: Tests (or other actions) are sometimes unnecessarily repeated because they appear at the end of an operation and at the beginning of the next one. They are redundant at this point, because they are bound to fail or succeed depending on their behavior in the previous operation. For example, in the fragment `deref(T), (ref(T) -> A ; B)`, `T` is checked to test whether it is a reference just before exiting `deref/1`. Code can be generated that jumps directly to the implementation of `A` or `B` depending on the result of this test. This option enables or disables the optimization.

Read/Write Mode Specialization [rw]: WAM-based implementations sometimes use a flag to test whether heap structures (i.e., the memory representation of functors) are being read (matched against) or written (created). According to the value of this flag, several instructions adapt their behavior with an `if-then-else`. A common optimization is to partially evaluate the switch inside the emulator loop to generate two different, parallel switch structures, one for each of the read/write possibilities. We can generate instruction sets (and emulators) where this optimization has been turned on or off.

5 Experimental Evaluation

We will report here on experimental data regarding the performance which was achieved on a set of benchmarks by a collection of emulators, all of them automatically generated through different combinations of options. In particular, by using all **compatible** possibilities for the transformation and generation options given in Section 4 we generated 96 different emulators (instead of $2^7 = 128$, as not all options are independent; for example, **vo** needs **om** to be performed).

This bears a close relationship with [11], but here we are not changing the internal data structure representation (and of course our instructions are all coded in ImProlog). It is also related to the experiment reported in [10], but the tests we perform are more extensive and cover more variations. Additionally, [10] starts off by being selective about the instructions to merge; this is a point we want to address in the future by using instruction-level profiling.

Our initial point was a "bare" instruction set comprising the "common basic blocks" of a relatively efficient abstract machine (the "stock" abstract machine of Ciao 1.10, itself an independent branch off the original SICStus Prolog 0.5/0.7 emulator, and with performance currently just below modern SICStus versions). Figures 6 to 7 summarize **overall** results for the experiments, as the data gathered —96 emulators $\times$ 13 benchmarks = 1248 performance figures— is too large to be examined in detail here. In those figures we plot, for three different cases, the resulting speed of every emulator using a dot per emulator. Every benchmark was run several times on each emulator to arrive at meaningful time measures, in a Linux machine with a Pentium 4 processor and using gcc 3.4 as C compiler. Although most of the benchmarks we used are relatively well known, we include a brief description in [15].

In order to encode emulator generation options in the corresponding dots, each available option in Sections 4.1 and 4.2 is assigned a bit in a binary number (a '1' means activating the option and a '0' means deactivating it). Every value in the y axis of the figures corresponds to a combination of the three options in Section 4.1, but only 6 combinations are plotted due to dependencies among options. Options in Section 4.2, which correspond to transformations in the way code is generated, are represented with four bits which are encoded as 16 different dot shapes (shown in each figure). Every combination of emulator generation options is thus assigned a different 7-bit number and a different dot shape and location. The x coordinate represents the relative speed w.r.t. the hand-coded emulator currently in Ciao 1.10, which is assigned speedup 1.0.

Of course, different selections for the bits assigned to the y coordinate and to the dot shapes would yield a different picture. However, our selection seems intuitively appropriate, as it addresses separately two different families of transformations. Indeed, Figure 6, which uses the geometric average[5] of all benchmarks to determine the overall performance, shows a quite well defined clustering around eight centers. Although it is not immediate from the picture (it has to be "decoded"), poorer speedups come from not activating some instruction creation options (which, for the stock emulator, really means *deactivating* them, since merging and specialization was made by hand quite some time ago, and the resulting instructions are already part of the emulator).

As a side note, while this figure portrays an average behavior, there were benchmarks whose results actually tracked this average behavior quite faithfully. An example is the the doubly recursive Fibonacci, which is often disregarded as unrealistic but which, for this particular experiment, turns out to predict very well the (geometric) average behavior of all benchmarks. All in all, this picture

[5] As a means to alleviate the effect of extremely good or bad cases.

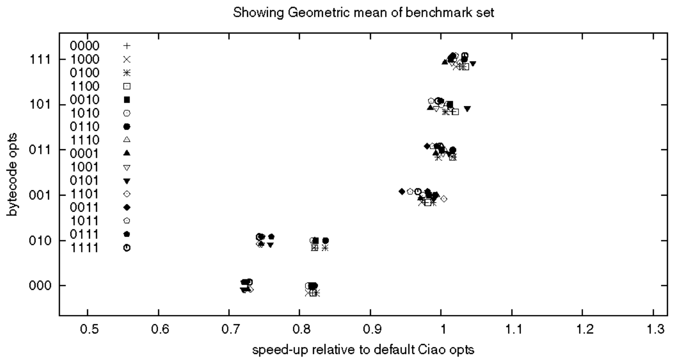


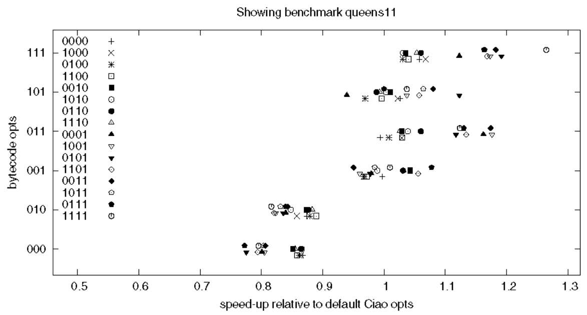
Fig. 6. Geometric average of all benchmarks (a dot per emulator)

(or, rather, the method which led to it) tries to reveal families of optimization options which give similar speed by showing dot clusters. Interestingly enough, once a set of generation options for $\mathcal{L}_B$ is fixed, the changes in the generation of $\mathcal{L}_C$ have (in general – see below) a relatively low impact. The general question *which options should be used for the "stock" emulator to be offered to general users* is answered by selecting a set of options somewhere in the topmost, rightmost cluster.

In any case, there are combinations of code generation options which achieve a speedup of 1.05, on average. While this may appear modest, consider that by starting with a simple instruction set (coded in ImProlog!) and applying systematically a set of transformation and code generation options, we have managed to match (and exceed) the time performance (memory performance was untouched) of an emulator which was hand-coded by very proficient programmers, and in which decisions were thoroughly tested along several years. Moreover, the transformation rules we have applied in our case are of course not the only ones, and we look forward to performing a more aggressive merging guided by profiling (merging is right now limited in depth to avoid a combinatorial explosion in the number of instructions). Similar work, with more emphasis on the production of languages for microprocessors is presented in [16], where a set of benchmarks is used to guide the (constrained) synthesis of such a set of instructions.

Figure 7 shows two cases of particular interest. The plot for *queens11* is a typical case which departs from the average behavior but which still resembles it. As a relevant difference, a much better speedup (around 1.25)[6] is achieved with some combinations of flags. On the other hand, the plot for *crypt* presents a completely different landscape: a plot where variations on the code generation scheme are as relevant as variations on the bytecode itself. This points to the need to find other clustering arrangements which shed some light on the interactions among different emulator code and bytecode generation schemes. Our experiments, however, lead us to think that in some cases the behavior tends to

[6] Which of course means that some benchmarks do not get any speedup.



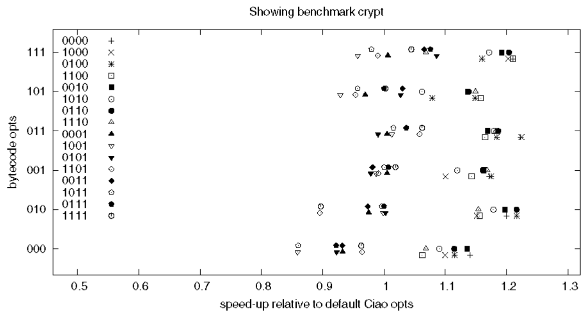


Fig. 7. Crypt: extreme case of spreading. Queens: scattered distribution.

be almost chaotic, as the lack of registers in the target architecture (i86) makes optimization a difficult task for the C compiler. This is supported by similar experiments on a PowerPC architecture, which has more general purpose registers, and in which the results are notably more stable across benchmarks. The overall conclusions for the best options and speedups remain roughly the same, only with less variance.

Table 1 tries to isolate the effects of separate options. It does so by listing, for each benchmark, including the geometric average, which options produced the best and the worst results time-wise. While there is no obvious conclusion, instruction merging is a clear winner, probably followed by having a variable number of operands, and then by specialized calls to built-ins. The first and second options save fetch cycles, while the third one saves processing time in general.

It can come as a surprise that using separate switches for read/write modes, instead of checking the mode in every instruction which needs to do so, does not seem to bring any advantage. A similar result was already observed in [11], and was attributed to modern architectures performing branch prediction and speculative work with redundant units. Therefore, short if-then-else statements might get both branches executed in parallel with the evaluation of the condition. Besides, implementing read/write modes with two switches basically doubles the size of the core of the emulator. A similar size growth happens when extensive merging is performed. In both cases a side effect is that of an increased cache miss ratio and the corresponding reduced performance.

Table 1. Options which gave best/worst performance

Benchmark	Best performance								Worst performance							
	vo	ib	om	ts	jc	ur	rw	Speed-up	vo	ib	om	ts	jc	ur	rw	Speed-down
default	x		x		x	x	x	1.00	x		x		x	x	x	1.00
all (geom.)	x	x	x		x		x	1.05					x		x	0.7
boyer	x		x		x		x	1.18							x	0.70
crypt		x	x	x				1.22				x			x	0.86
deriv	x	x	x		x			1.10		x					x	0.62
factorial	x		x				x	1.02						x	x	0.76
fib	x	x	x	x	x		x	1.02		x	x				x	0.75
knights	x	x	x				x	1.06				x	x	x	x	0.72
nreverse	x	x	x				x	1.03		x				x	x	0.57
poly	x	x	x		x		x	1.02		x					x	0.56
primes	x		x		x		x	1.10					x	x	x	0.73
qsort		x	x	x			x	1.05					x		x	0.54
queens11	x	x	x	x	x	x	x	1.26					x	x	x	0.77
query		x	x	x	x	x	x	1.06				x			x	0.71
tak	x	x	x	x				1.23				x	x	x	x	0.69

6 Conclusions

We have designed a language (ImProlog, a variation of Prolog with some imperative features) and used it to describe the semantics of instructions of a bytecode interpreter. ImProlog, with the proposed constraints, makes it possible both to perform non-trivial transformations (e.g., partial evaluation, unfolding, merging, etc.) and to generate efficient low-level code (using the cgen compiler) for each of the emulator instructions. Different transformations and code generation options can be applied, which result in different grades of optimization / specialization and different bytecode languages.

The low-level code for each instruction and the definition of the bytecode can be taken as input by a previously developed emulator generator to assemble full, high-quality emulators. Since the process of generating instruction code and bytecode format is automatic, we were able to produce and test different versions thereof to which several combinations of code generation options were applied.

We have also studied how these combinations perform with a series of benchmarks in order to find, e.g., what is the "best" average solution and how independent coding rules affect the overall speed. We have in this way as one case the regular emulator we started with (and which was decomposed to break complex instructions into basic blocks). However, we also found out that it is possible to outperform it by using some code patterns and optimizations not explored in the initial emulator, and, what is more important, starting from abstract machine definitions written in ImProlog. We intend to continue this line of exploration of improved abstract machines and incorporating them in the standard Ciao distributions.

References

1. Taivalsaari, A.: Implementing a Java Virtual Machine in the Java Programming Language. Technical report, Sun Microsystems (1998)
2. Rigo, A., Pedroni, S.: PyPy's Approach to Virtual Machine Construction. In: Dynamic Languages Symposium 2006, ACM Press (2006)
3. Morales, J., Carro, M., Puebla, G., Hermenegildo, M.: A generator of efficient abstract machine implementations and its application to emulator minimization. In Meseguer, P., Larrosa, J., eds.: International Conference on Logic Programming. LNCS, Springer Verlag (2005)
4. 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
5. Van Roy, P., Despain, A.: High-Performance Logic Programming with the Aquarius Prolog Compiler. IEEE Computer Magazine (1992) 54–68
6. Taylor, A.: High Performance Prolog Implementation through Global Analysis. Slides of the invited talk at PDK'91, Kaiserslautern (1991)
7. Morales, J., Carro, M., Hermenegildo, M.: Improving the Compilation of Prolog to C Using Moded Types and Determinism Information. In: Proceedings of the Sixth International Symposium on Practical Aspects of Declarative Languages. Number 3057 in LNCS, Heidelberg, Germany, Springer-Verlag (2004) 86–103
8. Van Roy, P.: 1983-1993: The Wonder Years of Sequential Prolog Implementation. Journal of Logic Programming **19/20** (1994) 385–441
9. 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
10. Nässén, H., Carlsson, M., Sagonas, K.: Instruction Merging and Specialization in the SICStus Prolog Virtual Machine. In: Proc. 3rd ACM SIGPLAN Int. Conf. on Principles and Practice of Declarative Programming, ACM Press (2001) 49–60
11. Demoen, B., Nguyen, P.L.: So Many WAM Variations, So Little Time. In: Computational Logic 2000, Springer Verlag (2000) 1240–1254
12. Gudeman, D., Bosschere, K.D., Debray, S.: `jc`: An efficient and portable sequential implementation of janus. In: Proc. of 1992 Joint International Conference and Symposium on Logic Programming, MIT Press (1992) 399–413
13. Henderson, F., Conway, T., Somogyi, Z.: Compiling Logic Programs to C Using GNU C as a Portable Assembler. In: ILPS 1995 Postconference Workshop on Sequential Implementation Technologies for Logic Programming. (1995) 1–15
14. Codognet, P., Diaz, D.: WAMCC: Compiling Prolog to C. In Sterling, L., ed.: International Conference on Logic Programming, MIT PRess (1995) 317–331
15. Morales, J., Carro, M., Hermenegildo, M.: Description and Optimization of Abstract Machines in an Extension of Prolog. Technical Report CLIP8/2006.0, Technical University of Madrid (UPM), School of Computer Science, UPM (2006)
16. Holmer, B.K.: Automatic Design of Computer Instruction Sets. PhD thesis, University of California at Berkeley (1993)