A Generator of Efficient Abstract Machine Implementations and its Application to Emulator Minimization*

José F. Morales[1], Manuel Carro[1], Germán Puebla[1], and Manuel V. Hermenegildo[1,2]

[1] School of Computer Science, T. University of Madrid (UPM)
(jfran@clip.dia.fi.upm.es, {mcarro,german,herme}@fi.upm.es)
[2] Depts. of Comp. Sci. and El. and Comp. Eng., U. of New Mexico (UNM)
(herme@unm.edu)

Abstract. The implementation of abstract machines involves complex decisions regarding, e.g., data representation, opcodes, or instruction specialization levels, all of which affect the final performance of the emulator and the size of the bytecode programs in ways that are often difficult to foresee. Besides, studying alternatives by implementing abstract machine variants is a time-consuming and error-prone task because of the level of complexity and optimization of competitive implementations, which makes them generally difficult to understand, maintain, and modify. This also makes it hard to generate specific implementations for particular purposes. To ameliorate those problems, we propose a systematic approach to the automatic generation of implementations of abstract machines. Different parts of their definition (e.g., the instruction set or the internal data and bytecode representation) are kept separate and automatically assembled in the generation process. Alternative versions of the abstract machine are therefore easier to produce, and variants of their implementation can be created mechanically, with specific characteristics for a particular application if necessary. We illustrate the practicality of the approach by reporting on an implementation of a generator of production-quality WAMs which are specialized for executing a particular fixed (set of) program(s). The experimental results show that the approach is effective in reducing emulator size.

1 Introduction

The use of intermediate abstract machines as a means to compile and tune programs (specially those written in high-level languages with complex features) requires several components. In order to execute programs written in a source language $\mathcal{L}_P$, a compiler into the abstract machine language, $\mathcal{L}_A$, is needed. An emulator for $\mathcal{L}_A$, usually written in some lower-level language $\mathcal{L}_C$ for which

* Work partially funded by the European Commission IST-FET programme, IST-2001-38059 *ASAP* project, and by the Spanish Ministry of Science and Education, TIC 2002-0055 *CUBICO* project. Manuel Hermenegildo is also supported by the Prince of Asturias Chair in Information Science and Technology at UNM. J. Morales is also supported by an MCYT/MEC fellowship co-financed by the European Social Fund.

there is a compiler to native code, performs the actual execution.[3] Traditional implementations based on abstract machines start with a fixed set of abstract machine instructions and then develop the compiler and the emulator.

One important concern when implementing such emulators is that of efficiency (see [1–5]), which depends greatly on the complexity of $\mathcal{L}_P$ and, of course, on the compiler and emulator technology. As a result, emulators are very often difficult to understand, maintain, and modify. This makes the implementation of *variants* of abstract machines a hard task, since both the compiler and emulator, which are rather complex, have to be rewritten by hand for each case. Variants of emulators have been (naturally) used to evaluate different implementation options for a language [4], often manually. Automating the creation of these variants will, additionally, make it possible to tailor a general design to particular applications or environments with little effort. A particularly daunting task is to adapt existing emulators to resource-constrained tasks, such as those found in pervasive computing. While this can clearly be done by carefully rewriting existing emulators, selecting alternative data representations, and, maybe, adapting them to the type of expected applications, we deem that this task is a too difficult one, especially taking into account the amount of different small devices which are ubiquitous nowadays.

In this work we propose an approach in which, rather than being hand-written, emulators and (back-end) compilers are automatically generated from a high-level description of the abstract machine instruction set. This makes it possible to easily experiment with alternative abstract machines and to evaluate the impact of different implementation decisions, since the corresponding emulator and compiler are obtained automatically.

In order to do so, rather than considering emulators for a particular abstract machine, we formalize emulators as parametric programs, written, for purposes of improved expressiveness, in a syntactical extension of $\mathcal{L}_C$ (as explained in Example 1) that can represent directly elements of $\mathcal{L}_A$ and which receive two inputs: a *program* to be executed, written in language $\mathcal{L}_A$, and a description of the abstract machine language $\mathcal{L}_A$ in which the operational definition of each instruction of $\mathcal{L}_A$ is given in terms of $\mathcal{L}_C$. E.g., we define a generic emulator as a procedure **interpret**$(program, M)$ which takes as input a program in the abstract machine language $\mathcal{L}_A$ and a definition M of the abstract machine itself and interprets the *program* according to M.

For the sake of maintainability and ease of manipulation, $\mathcal{L}_A$ is to be as close as possible to its conceptual definition. This usually affects performance negatively, and therefore a refinement step, based on pass separation [6], a form of staging transformations [7], is taken to convert programs written in $\mathcal{L}_A$ into programs written in $\mathcal{L}_B$, a lower-level representation for which faster interpreters can be written in $\mathcal{L}_C$. By formalizing adequately the transformation from $\mathcal{L}_A$ to $\mathcal{L}_B$ it is possible to do automatically:

– The translation of programs from $\mathcal{L}_A$ into $\mathcal{L}_B$.

[3] Implementations of abstract machines are usually termed *virtual machines*. We will, however, use the term *emulator* or *bytecode interpreter* to denote a virtual machine. This is in line with the tradition used in the implementation of logic programming languages.

– The generation of efficient emulators for programs in $\mathcal{L}_B$ based on interpreters for $\mathcal{L}_A$.
– The generation of compilers from $\mathcal{L}_P$ to $\mathcal{L}_B$ based on compilers from $\mathcal{L}_P$ to $\mathcal{L}_A$.

A high-level view of the different elements we will talk about in this paper appears in Figure 1. When the abstract machine description M is available, it is possible (at least conceptually) to partially evaluate the procedure **interpret** into an emulator for a (now fixed) M. Although this approach is attractive in itself, it has the disadvantage that the existence of a partial evaluator of programs written in $\mathcal{L}_C$ is required. Depending on $\mathcal{L}_C$, this may or may not be feasible.

A well known result in partial evaluation [8] is that it is possible to partially evaluate a partial evaluator w.r.t. itself and a particular program as static data. By taking the parametric emulator as static data for the partial evaluator, we can obtain an emulator generator (*emucomp*), which will produce an efficient emulator when supplied with a description of an abstract machine. This approach, known as the second Futamura projection [9], not only requires the availability of a partial evaluator for programs in $\mathcal{L}_C$ but also needs the partial evaluator to be self-applicable. Somewhat surprisingly, the structure of emulator generators is often easy to understand. The approach we will follow is therefore to write such an emulator generator directly by hand. The emulator generator we propose has been defined in such a way that it can produce an emulator whose code is comparable to a hand-written one when provided with a description of an abstract machine.

The benefits of our approach are multifold. Writing an emulator generator is clearly much more profitable than writing a particular emulator (though more difficult to achieve for the general case) since, with no performance penalty, it will make it possible to easily experiment with multiple variations of the original abstract machine. For example, and as discussed later, it is straightforward to produce reduced emulators. As an example of the application of our technique, and taking as starting point the instruction set of an existing emulator (a production-quality implementation of a modern version of the Warren Abstract Machine for Prolog [10, 11]), we generate emulators which can be sliced with respect to the set of abstract machine instructions which a given application or sets of applications are going to actually use.

2 Automatic Generation of Emulators

In this section we will develop a compiler for emulators which takes a description of the machine and can produce emulators which are very close (and in some cases identical) to what a skilled programmer would craft.

Our initial source language is $\mathcal{L}_P$, and we assume that there is a compiler *comp* from $\mathcal{L}_P$ to an $\mathcal{L}_A$, a symbolic representation of a lower-level language intended to be interpreted by an emulator. We want *comp* to be relatively simple and independent from the low-level details of the implementation of the final emulator. The definition of $\mathcal{L}_A$ will be kept separate in M so that it can be used later (Section 2.2) in a generic emulator. Instructions in $\mathcal{L}_A$ can, in general, consult and modify a global state and change the control flow with (conditional) *jump*/*call* instructions.

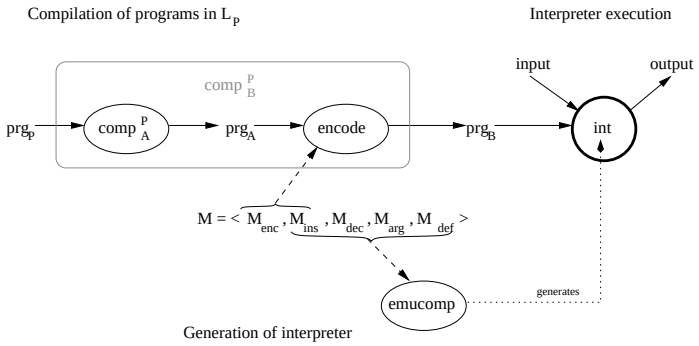

Fig. 1. "Big Picture" view of the generation of emulators

2.1 Scheme of a Basic Emulator

Emulators have usually a main loop implementing a fetch-execute cycle. Figure 2 portrays an example, where that cycle is performed by a tail-recursive procedure. The reason to choose this scheme is because it allows a shorter description of some further transformations, but note that it can be converted automatically into a proper loop. The function:

$$fetch_A : locator_A \times program_A \rightarrow \langle ins_A, locator_A \rangle$$

returns, for a given program and program point, the instruction at that point (of type ins_A, a tuple containing instruction name and arguments) and the next location in the program, in sequential order. This abstracts away program counters, which can be symbolic, and indirections through the program counter. We will reuse this function, in different contexts, in the following sections.

$\mathbf{emu_A}(p, program) \equiv$
  $\langle ins, p' \rangle = fetch_A(p, program)$
  case ins of
    $\langle \mathbf{move}, [\mathbf{r}(i), \mathbf{r}(j)] \rangle$: $reg[j] := reg[i];\ p'' := p'$
    $\langle \mathbf{jump}, [\mathbf{label}(l)] \rangle$: $p'' := l$
    $\langle \mathbf{call}, [\mathbf{label}(l)] \rangle$: $push(p');\ p'' := l$
    $\langle \mathbf{ret}, [] \rangle$: $p'' := pop()$
    $\langle \mathbf{halt}, [] \rangle$: $return$
    otherwise : **error**
  $\mathbf{emu_A}(p'', program)$

Fig. 2. An example of simple $\mathcal{L}_A$-level interpreter

$\mathbf{int_1}(p, program, M) \equiv$
  $\langle \langle name, args \rangle, p' \rangle = fetch_A(p, program)$
  if $\neg valid_A(\langle name, args \rangle, M_{ins}, M_{absexp})$ then **error**
  $cont = \lambda a \rightarrow [p'' := a]$
  $[\![M_{def}(p', cont, name, M_{args}(args))]\!]$
  $\mathbf{int_1}(p'', program, M)$

Fig. 3. Parametric interpreter for $\mathcal{L}_A$

Example 1 ($\mathcal{L}_A$ instructions and their semantics written in $\mathcal{L}_C$). The left hand side of each of the branches in the *case* expression of Figure 2 corresponds to one

instruction in $\mathcal{L}_A$. The emulator $\mathbf{emu_A}$ is written in $\mathcal{L}_C$ (syntactically extended to represent $\mathcal{L}_A$ instructions), and the semantics of each instruction is given in terms of $\mathcal{L}_C$ in the right hand side of the corresponding branch. The implementation of the memory model is implicit in the right hand side of the *case* branches; we assume that appropriate declarations for types and global variables exist. $\mathcal{L}_A$ instructions are able to move data between registers, do **jump**s and **call**s to subroutines, and stop the execution with the **halt** instruction. Alternative emulators can be crafted by changing the way $\mathcal{L}_A$ instructions are implemented. This must, of course, be done homogeneously across all the instruction definitions.

2.2 Parameterizing the Emulator

In order to make emulators parametric with respect to the abstract machine definition, we need to settle on an emulator scheme first (Figure 3) and to make the definition of the abstract machine precise. We will use a piecewise definition $M = (M_{def}, M_{arg}, M_{ins}, M_{absexp})$ of $\mathcal{L}_A$ which is passed as a parameter to the emulator scheme and which relates different parts of the abstract machine with a feasible implementation thereof. The meaning of every component of M (see also Example 2) is as follows:

M_{def} The correspondence between every instruction of $\mathcal{L}_A$ and the code to execute it in $\mathcal{L}_C$.

M_{arg} The correspondence between every argument for the instructions in $\mathcal{L}_A$ and the corresponding data in $\mathcal{L}_C$. M_{args} generalizes M_{arg} by mapping lists of arguments in $\mathcal{L}_A$ into lists of arguments in $\mathcal{L}_C$. The definitions of M_{def} and M_{arg} are highly dependent, and quite often updating one will require changes in the other.

M_{ins} The instruction set, described as the name and the *format* every instruction in $\mathcal{L}_A$ accepts, i.e., which kinds of expressions in $\mathcal{L}_A$ can be handled by the instruction. The format is given as a list of abstract expressions of $\mathcal{L}_A$, whose definition is also included in M (see next item). For example, a **jump** instruction might be able to jump to a (static) label, but not to the address contained in a register, or a **move** instruction might be able to store a number in a register but not directly in a memory location. Note that the same instruction name can be used with different formats.

M_{absexp} An abstraction function which returns the type of an instruction argument.

The interpreter in Figure 3 uses the definition of the semantics of $\mathcal{L}_A$ in terms of $\mathcal{L}_C$. For every instruction, arguments in $\mathcal{L}_A$ are translated into arguments in $\mathcal{L}_C$ by M_{args}, and M_{def} selects the right code for the instruction. Both M_{def} and M_{arg} are functions which return unevaluated pieces of code, which are meant to be executed by $\mathbf{int_1}$ — this is marked by enclosing the function call by double square brackets. The next program location is set by a function *cont* which is handed in to M_{def} as an argument. The language expressions not meant to be evaluated but passed as data are enclosed inside square brackets. The context should be enough to distinguish them from those used to access array elements or to denote lists.

$M_{def}(next, cont, name, args) =$
  case $\langle name, args\rangle$ of
    $\langle \mathbf{move}, [a, b]\rangle \to [a := b;\ cont(next)]$
    $\langle \mathbf{jump}, [a]\rangle \to [cont(a)]$
    $\langle \mathbf{call}, [a]\rangle \to [push(next);\ cont(a)]$
    $\langle \mathbf{ret}, []\rangle \to [cont(pop())]$
    $\langle \mathbf{halt}, []\rangle \to [return]$

$M_{ins} =$
  { $\langle \mathbf{move}, [\mathbf{r}, \mathbf{r}]\rangle$
    $\langle \mathbf{jump}, [\mathbf{label}]\rangle$
    $\langle \mathbf{call}, [\mathbf{label}]\rangle$
    $\langle \mathbf{ret}, []\rangle$
    $\langle \mathbf{halt}, []\rangle$ }

$M_{arg}(arg) =$
  case arg of
    $\mathbf{r}(i) \to reg[i]$
    $\mathbf{label}(l) \to l$

$M_{absexp}(arg) =$
  case arg of
    $\mathbf{r}(_) \to \mathbf{r}$
    $\mathbf{label}(_) \to \mathbf{label}$
    otherwise $\to \bot$

Fig. 4. Definition of M for our example

In order to ensure that no ill-formed instruction is executed (for example, because a wrongly computed location tries to access instructions outside the program scope), the function $valid_A$ checks that the instruction named *name* can understand the arguments *args* which it receives. It needs to traverse every argument, extract its type, which defines an argument format, and check that the instruction *name* can be used with arguments following that format.

Example 2 (Definitions for a trivial abstract machine in $\mathbf{int_1}$*).* In the definitions for M in Figure 4, the higher-order argument *cont* is used to set the program counter pointing to the instruction to be executed next. The instruction definitions do not check operator and operand types, since that has been taken care of by $valid_A$ by checking that the type of every argument which matches those accepted by the instruction at hand.

Instructions can in general admit several argument formats. For example, arithmetic instructions might accept integers and floating-point numbers. That would make M_{ins} have several entries for some instructions. This poses no problem, as long as M_{absexp} returns all abstractions for a given pattern and there is a suitable selection rule (e.g., the most concrete applicable pattern) is used to choose among different possibilities. For the sake of simplicity we will not deal with that case in this paper. Multi-format instructions are helpful when compiling weakly-typed languages, or languages with complex inheritance rules, where types of expressions might not be completely known until runtime. If this happens, compiling to a general case to be dynamically checked is the only solution.

2.3 A More Specialized Intermediate Language and its Interpreter

The symbolic nature of $\mathcal{L}_A$, which should be seen as an intermediate language, makes it convenient to express instruction definitions and to associate internally properties to them, but it is not designed be directly executed. Most emulators use a so-called *bytecode* representation, where many details have been settled: operation codes for each instruction (which capture the instruction name and argument types), size of every instruction, values of some arguments, etc. In return bytecode interpreters are quite fast, because a great deal of the work $\mathbf{int_1}$ has been statically encoded, so that many overheads may be removed. In short, the bytecode design focuses on achieving speed.

On the other hand, working right from the beginning with a low-level definition is cumbersome, because many decisions percolate through the whole language and seemingly innocent changes can force the update of a significant part of the bytecode definition (and, therefore, of its emulator). This is the main reason to keep $\mathcal{L}_A$ at a high level, with many details still to be filled in. It is however possible to translate $\mathcal{L}_A$ into a lower-level language, $\mathcal{L}_B$, closer to $\mathcal{L}_C$ and easier to represent using $\mathcal{L}_C$ data structures. That process can be instrumented so that programs written in $\mathcal{L}_A$ are translated into $\mathcal{L}_B$ and interpreters for $\mathcal{L}_A$ are transformed in interpreters for $\mathcal{L}_B$ using a similar encoding. Translating from $\mathcal{L}_A$ to $\mathcal{L}_B$ is done by a function:

$$encode : \mathcal{L}_A \to \mathcal{L}_B$$

encode accepts instructions in $\mathcal{L}_A$ (including name and arguments) and returns tokens in $\mathcal{L}_B$. The encoding function has to:

1. Assign a unique operation code (*opcode*) to each instruction in $\mathcal{L}_A$ when the precondition expressed by $valid_A$ holds (a compile-time error would be raised otherwise). This moves the overhead of checking formats from runtime to compile-time.
2. Take the arguments of instructions in $\mathcal{L}_A$ and translate them into $\mathcal{L}_B$.

encode is used to generate a compiler from $\mathcal{L}_P$ into $\mathcal{L}_B$ from a compiler from $\mathcal{L}_P$ into $\mathcal{L}_A$ (Figure 1). As *encode* gives a unique opcode to every combination of instruction name and format, it has an associated function:

$$decode : \mathcal{L}_B \to \mathcal{L}_A$$

which brings bytecode instructions back to its original form.[4] In order to capture the meaning of *encode* / *decode*, we augment and update the abstract machine definition to be $M = (M_{def}, M_{arg}, M_{ins'}, M_{absexp}, M_{enc}, M_{dec})$ (see Figure 5 and Example 3). $M_{ins'}$ is derived from M_{ins} by capturing the opcode assignment. It accepts an opcode and returns the corresponding instruction in $\mathcal{L}_A$ as a pair $\langle name, format \rangle$. Argument encoding is taken care of by a new function M_{enc}. M_{dec} is the inverse of M_{enc}.

An interpreter $\mathbf{int_2}$ for $\mathcal{L}_B$ (see Figure 6) can be derived from $\mathbf{int_1}$ with the help of bytecode decoding. $\mathbf{int_2}$ receives an (extended) definition of M and uses it to retrieve the original instruction $\langle name, format \rangle$ in $\mathcal{L}_A$ corresponding to an opcode in a bytecode program (returned by $program[p]$, where p is a program counter in $\mathcal{L}_B$). The arguments are brought from the domain of $\mathcal{L}_B$ to the domain of $\mathcal{L}_A$ by M_{dec}, and code and argument translations defined by M_{def} and M_{arg} can then be employed as in $\mathbf{int_1}$.

We want to note that in Figure 6 the recursive call has been placed inside the continuation code, which avoids the use of the intermediate variable p'' used in Figure 2 and makes it easier to apply program transformations.

Example 3 (Encoding instructions). Every combination of instruction name and format from Example 2, Figure 4, is assigned a different opcode. $M_{ins'}$ retrieves both the corresponding instruction name and format for every opcode. In Figure 8, the sample $\mathcal{L}_A$ program on the left is translated by *encode* into the program $\mathcal{L}_B$ on the right, which can be interpreted by $\mathbf{int_2}$ using the definitions for M.

[4] Both *encode* and *decode* may need to resolve symbols. As this is a standard practice in compiling (which can even be delayed until link time), we will not deal with that problem here.

$$
\begin{array}{lll}
M_{ins'}(opcode) = & M_{enc}(arg) = & M_{dec}(t, f) = \\
\quad \text{case } opcode \text{ of} & \quad \text{case } arg \text{ of} & \quad \text{case } \langle t, f\rangle \text{ of} \\
\quad\quad 0 \rightarrow \langle \mathbf{move}, [\mathbf{r}, \mathbf{r}]\rangle & \quad\quad \langle \mathbf{r}(a)\rangle \rightarrow a & \quad\quad \langle a, \mathbf{r}\rangle \rightarrow \mathbf{r}(a) \\
\quad\quad 1 \rightarrow \langle \mathbf{jump}, [\mathbf{label}]\rangle & \quad\quad \langle \mathbf{label}(l)\rangle \rightarrow symbol(l) & \quad\quad \langle l, \mathbf{label}\rangle \rightarrow \mathbf{label}(l) \\
\quad\quad 2 \rightarrow \langle \mathbf{call}, [\mathbf{label}]\rangle & & \\
\quad\quad 3 \rightarrow \langle \mathbf{ret}, []\rangle & & \\
\quad\quad 4 \rightarrow \langle \mathbf{halt}, []\rangle & &
\end{array}
$$

Fig. 5. New Parts of the Abstract Machine Definition

$$
\begin{array}{l}
\mathbf{int_2}(p, prg, M) \equiv \\
\quad opcode = prg[p] \\
\quad \langle name, format\rangle = M_{ins'}(opcode) \\
\quad \langle args, p'\rangle = decode_{ins}(format, [p], [prg], M) \\
\quad cont = \lambda a \rightarrow [\mathbf{int_2}(a, prg, M);\ return] \\
\quad [\![M_{def}(p', cont, name, M_{args}(args));\ cont(p')]\!]
\end{array}
\qquad
\begin{array}{l}
decode_{ins}(\langle f_1, \ldots, f_n\rangle, p, prg, M) = \\
\quad \langle \langle d_1, \ldots, d_n\rangle, p + 1 + n\rangle \text{ where} \\
\quad d_i = M_{dec}([prg[p + i]], f_i)
\end{array}
$$

Fig. 6. Parametric interpreter for $\mathcal{L}_B$

2.4 A Final Emulator

The interpreter $\mathbf{int_2}$ in Section 2.3 still has the overhead associated with using continuously the abstract machine definition M. However, once M is fixed, it is possible to instantiate the parts of $\mathbf{int_2}$ which depend statically on M, to give another emulator $\mathbf{int_3}$. This can be seen as a partial evaluation of $\mathbf{int_2}$ with respect to M, i.e., $\mathbf{int_3} \equiv [\![spec]\!](\mathbf{int_2}, M)$. This returns an emulator written in $\mathcal{L}_C$ without the burden of translating instructions in $\mathcal{L}_B$ to the level of $\mathcal{L}_A$ in order to access the corresponding code and argument definitions in M_{def} and M_{arg}. Finally, and although $program[p]$ is not known at compile time, we can introduce a case statement which enumerates the possible values for the opcode, thus becoming *static*. This is a common technique to make partial evaluation possible in similar cases.

Since the interpreter structure is fixed, a compiler of emulators could be generated by specializing the partial evaluator for the case of $\mathbf{int_2}$, i.e.,

$$
\begin{array}{l}
emucomp : M \rightarrow code_C \\
emucomp = [\![spec]\!](spec, \mathbf{int_2})
\end{array}
$$

which is equivalent to the emulator compiler in Figure 7. It reuses the definition of $decode_{ins}$ seen in the previous section. Note that, as stated before, this emulator compiler has a regular structure, and we have opted to craft it manually, instead of relying on a self-applicable partial evaluator for $\mathcal{L}_C$. This compiler emulator, of course, does not need to be implemented in $\mathcal{L}_C$, and, in fact, in our particular implementation it is written in Prolog and it generates emulators in C.

$$
\begin{array}{l}
emucomp(M) = \\
\quad [\,\mathbf{emu_B}(p, prg) \equiv \\
\quad\quad \text{case } get_opcode(p, prg) \text{ of} \\
\quad\quad\quad opcode_1 : inscomp(opcode_1, M) \\
\quad\quad\quad \ldots \\
\quad\quad\quad opcode_n : inscomp(opcode_n, M)] \\
\quad\quad \text{where } opcode_i \in domain(M_{ins'})
\end{array}
\qquad
\begin{array}{l}
inscomp(opcode, M) = \\
\quad [M_{def}(p', cont, name, M_{args}(args));\ cont(p')] \\
\text{where} \\
\quad \langle name, format\rangle = M_{ins'}(opcode) \\
\quad \langle args, p'\rangle = decode_{ins}(format, [p], [prg], M) \\
\quad cont = \lambda a \rightarrow [\mathbf{emu_B}(a, prg);\ return]
\end{array}
$$

Fig. 7. Emulator Compiler

$\mathcal{L}_A$ **program**

move **r**(0) **r**(2)
move **r**(1) **r**(0)
move **r**(2) **r**(1)
halt

$\mathcal{L}_B$ **program**

0 0 2	0 1 0	0 2 1	4

Fig. 8. Sample program

$\mathbf{emu_B}(p, program) \equiv$
  case $program[p]$ of
    $0 : reg[program[p+1]] := reg[program[p+2]];$
      $\mathbf{emu_B}(p+3, program); return$
    $1 : \mathbf{emu_B}(program[p+1], program); return$
    $2 : push(p+2);$
      $\mathbf{emu_B}(program[p+1], program); return$
    $3 : \mathbf{emu_B}(pop(), program); return$
    $4 : return; return$

Fig. 9. Generated emulator

Example 4 (The generated emulator). Figure 9 depicts an emulator for our working example, obtained by specializing $\mathbf{int_2}$ with respect to the machine definition in Example 3. Note the recursive call and *return*s at the end of every case branch which ensure that no other code after those statements is executed. All the recursive calls are tail recursions.

3 An Example Application: Minimal and Alternative Emulators

We will illustrate our technique with two (combined) applications: generating WAM emulators which are specialized for executing a fixed set of programs, and using different implementations of the WAM data structures. The former is very relevant in the context of applications meant for embedded devices and pervasive computing. It implements an automatic specialization scheme which starts at the Prolog level (by taking as input the programs to be executed) and, by slicing the abstract machine definition, traverses the compilation chain until the final specialized emulator for these programs is generated. The latter makes it possible to easily experiment with alternative implementation decisions.

We have already introduced how a piecewise definition of an abstract machine can allow making emulator generation automatic. In the rest of this section we will see how this technique can be used to generate such application-specific emulators, and we will report on a series of experiments performed around those ideas. We will focus, for the moment, on generating correct emulators of *minimal size*, although the technique can obviously also be applied to investigating the impact of alternative implementations on performance.

3.1 Obtaining Specialized Emulators

The objective of specializing a program with respect to some criteria is to obtain a new program that preserves the initial semantics and is smaller or requires fewer operations. The source and target language are typically the same; this is expected, since specialization which operates across different translation levels is harder. It is however highly interesting, and applicable to several cases, such as the compilation to virtual machines and JIT compilation.

Among previous experiences which cross implementation boundaries we can cite [12], where automatically specialized WAM instructions are used as an intermediate step to generate C code which outperforms compilers to native code, and the Aquarius Prolog compiler [13] which carried analysis information along the compilation process to generate efficient native code.

As mentioned before, simplifying automatically hand-coded emulators (in order to speed them up or to reduce the executable size) written in $\mathcal{L}_C$ requires a specializer for $\mathcal{L}_C$ programs able to understand the emulator structure. The task can be quite difficult for efficient, complex emulators. Even in the case that the emulator can be dealt with, there are very few information sources to use in order to perform useful optimizations: the input data is, in principle, any bytecode program.

One way to propagate bytecode properties about a particular program p down to the emulator so that the specializer can do some effective optimization is by partially evaluating the emulator w.r.t. p and specializing the resulting program. Even if the specializer is powerful enough to work with this input, this solution has some drawbacks. The resulting code lacks some interesting properties: it is not as portable as the bytecode (since it is written in $\mathcal{L}_C$) and it is presumably less compact than the combination emulator + bytecode. Portability can often be sacrificed if compactness is preserved; in exchange, the resulting program is usually self-contained and generating stand-alone applications is in principle easier. This is not a bad scenario if there are automatic tools which can do a good job on these tasks (i.e., the code explosion generated by the partial evaluator is then taken care of by the specializer). Unfortunately, this is usually not the case.

An alternative approach is to express the specialization of the emulator in terms of *slicing* [14–16]. A slicing algorithm and the properties that it focus on, ϕ, of the emulator input, such as, e.g., bytecode reachable points, output variables, etc., are defined so that only the parts of the emulator (or a conservative approximation thereof) needed to maintain those properties have to be kept by the transformation.

One problem with this approach is that the bytecode is quite low level and the emulator too complicated to be automatically manipulated. However, our emulator generation scheme makes this problem more tractable. In our case $\mathcal{L}_B$ programs are generated from a higher-level representation which can be changed quite freely (even enriched with compiler-provided information to be later discarded by *encode*) and which aims at being easily manageable rather than efficient. It seems therefore more convenient to work at the level of $\mathcal{L}_A$ to extract the slicing information, since it offers more simplification opportunities. It has to be noted that transforming the $\mathcal{L}_C$ emulator code using some $\mathcal{L}_A$ properties may be extremely difficult: to start with, suitable tools to work with $\mathcal{L}_C$ are needed, and they should be able to understand the relationship between $\mathcal{L}_B$ and $\mathcal{L}_A$ elements. It is much easier to work at the level of the definition of the abstract machine M, where $\mathcal{L}_A$ is completely captured, and where its relationship with $\mathcal{L}_B$ is contained.

We therefore formulate a slicing transformation that deals directly with M and whose result is used to generate a specialized emulator $\mathbf{emu_s}$:

$$\mathbf{emu_s} = emucomp([\![slice_M]\!](M, \phi))$$

$\mathbf{emu_s}$ can also be viewed as the result of slicing $emucomp(M)$ (i.e., $\mathbf{emu_B}$) with a particular slicing algorithm that, among other things, preserves the (loop) structure of the emulator. [5] That is, $slice_M$ deals with the instruction set or the

[5] Due to the simplicity of the interpreter scheme, this is not a hard limitation for most emulator transformations, as long as the transformation output is another emulator.

instruction code definitions, and leaves complex data and control issues, quite common in efficient emulators, untouched and under the control of *emucomp*. Slicing can change all the components of the definition of M, including M_{def}, which may cause the compiled emulator to lose or specialize instructions. Note that when M_{ins} is modified, the transformation affects the compiler, because the *encode* function uses definitions in M.

3.2 Some Examples of Opportunities for Simplification

There is a variety of simplifications at the level of M that preserve the loop structure. They can be expressed in terms of the previously presented technique.

Instruction Removal: Programs compiled into bytecode can be scanned and brought back into $\mathcal{L}_A$ using $M_{ins'}$ to find the set I of instructions used in them. M is then sliced with respect to I and a new, specialized emulator is created as in Section 2.4. The new emulator may be leaner than the initial one since it probably has to interpret fewer instructions.

Removing Format Support: If $\mathcal{L}_A$ has instructions which admit arguments of different types (e.g., arithmetical operations which admit both integers and floating point numbers), programs that only need support for some of the available types can be executed in a reduced emulator. This can be achieved, again, by slicing M with respect to the remaining instruction and argument formats.

Removing Specialized Instructions: M can define specialized instructions (for example, for special argument values) or collapsed instructions (for often-used instruction sequences). Those instructions are by definition redundant, but sometimes useful for the sake of performance. However, not all programs require or benefit from them. When the compiler to $\mathcal{L}_A$ can selectively choose whether using or not those versions, a smaller emulator can be generated.

Obtaining the optimal set of instructions (w.r.t. some metric) for a particular program is an interesting problem. It is however out of the scope of this paper.

3.3 Experimental Evaluation

We tested the feasibility of the techniques proposed herein for the particular case of the compilation of Prolog to a WAM-based bytecode. We started off with Ciao [17], a real, full-fledged logic programming system, featuring a (Ciao-)Prolog to WAM compiler, a complex bytecode interpreter (the emulator) written in C, and the machinery necessary to generate multi-platform, bytecode-based executables. We refactored the existing emulator as an abstract machine as described in the previous sections, and we implemented an emulator compiler which generates emulators written in C. We also implemented a slicer for removing unused instructions from the abstract machine definition.

Specialized emulators were built for a series of benchmark programs. For each of them, the WAM code resulting from its compilation was scanned to collect the set I of actually used instructions, and the general instruction set M_{ins} was sliced with respect to I in order to remove unused instructions. The resulting description was used to encode the WAM code into bytecode and to generate the specialized emulator. We have verified that, when no changes are applied to

the abstract machine description, the generated emulator and bytecode representation are as optimized as the original ones. Orthogonally, we defined three slightly different instruction sets and generated specialized emulators for each of these sets and each of the benchmark programs, and we measured the resulting size (Table 1).

The benchmarks feature both symbolic and numerical computation, and they are thus representative of several possible scenarios. The list of benchmarks includes some widely known programs which we will not describe here. Other programs, used less often as benchmarks, include `hw` (which prints "Hello world!"), `exp` (which computes 13^{7111} with a linear- and a logarithmic-time algorithm), `knights` (chess knight tour visiting once every board cell), `poly` (symbolically raise `1+x+y+z` to the n^{th} power), and `query` (query a database of countries, population, and area). Specially interesting are a set of signal processing programs, applied in wearable computing: `stream`, which generates 3-D stereo audio from mono audio, compass, and GPS signals to simulate the movement of a subject in a virtual world; `stream_dyn`, an improved version of `stream` which can use any number of different input signals and sampling frequencies, and `stream_opt`, an optimized version where number of signals and sampling frequency is fixed.

It has to be noted that, although most of these benchmarks are of moderate size, our aim in this section is precisely to show how to reduce automatically the footprint of an otherwise large engine for these particular cases. On the other hand, reduced size does not necessarily make them unrealistic, in the sense that they effectively perform non-trivial tasks. As an example, `stream_opt` processes audio in real time and with constant memory usage using Ciao Prolog in a 200MHz GumStix (a computer the size of a chewing gum).

The whole compilation process is fairly efficient. On a Pentium M at 1400MHz, with 512MB of RAM, and running Linux 2.6.10, the compiler compiles itself and generates a specialized emulator in 31.6 seconds: less than 0.1 seconds to generate the code of the emulator loop itself, 11.3 seconds to compile the compiler to bytecode (written in Prolog), and 20.3 seconds to compile all the C code: Prolog-accessible predicates written in C (e.g., builtins and associated glue code) and the generated emulator using `gcc` with optimization grade `-O2`. Both the Prolog compiler and emulator generator are written in (Ciao-)Prolog.

The results of the benchmarks are in Table 1, were different instruction sets were used. Columns under the ***basic*** label correspond to the instruction set of the original emulator. The ***ivect*** label presents the case for an instruction set where several compact instructions which are specialized to move register values before calls to predicates have been added to the studied emulator. Finally, columns below the label ***iblt*** shows results for the instruction set *iblt*, where specialized WAM instructions for the arithmetic builtins have been added to the emulator. In each of these set of columns, and for each benchmark, we studied the impact of specialization in the emulator size (the **loop** columns) and bytecode size (the **bytecode** columns).

The **bytecode** columns show two different figures: *full* is the bytecode size including all libraries used by the program and the initialization code (roughly constant for every program) automatically added by the standard compiler. The numbers in the *strip* column were obtained after performing dead code elimination at the Prolog level (such as removing unused Prolog library modules and

	Basic				*ivect*				*iblt*			
	loop (29331)		**bytecode**		**loop (33215)**		**bytecode**		**loop (34191)**		**bytecode**	
	full	strip	full	strip	full	strip	full	strip	full	strip	full	strip
hw	28%	71%	33116	48	29%	74%	31548	48	29%	75%	31136	48
boyer	26%	46%	40198	8594	27%	50%	38606	8542	28%	52%	38168	8512
crypt	27%	58%	33922	2318	28%	62%	32306	2242	28%	63%	31842	2186
deriv	27%	56%	33606	2002	28%	59%	32022	1958	28%	61%	31606	1950
exp	28%	59%	32102	498	29%	63%	30542	478	29%	63%	30114	458
fact	28%	69%	31756	152	29%	72%	30216	152	29%	73%	29804	148
fib	28%	70%	31758	154	29%	74%	30218	154	29%	74%	29798	142
knights	27%	54%	32306	702	28%	56%	30726	662	29%	57%	30298	642
nrev	27%	65%	31866	262	28%	69%	30322	258	28%	70%	29910	254
poly	26%	48%	34682	3078	27%	52%	33098	3034	27%	53%	32664	3008
primes	27%	56%	32082	478	28%	61%	30526	462	29%	62%	30102	446
qsort	27%	58%	32334	730	28%	61%	30778	714	28%	62%	30370	714
queens11	28%	55%	32248	644	29%	59%	30696	632	29%	60%	30220	564
query	28%	59%	32816	1212	29%	63%	31256	1192	29%	64%	30840	1184
stream_dyn	25%	42%	36060	2992	25%	45%	34420	2920	26%	45%	33890	2802
stream_opt	26%	46%	35152	2084	26%	49%	33516	2016	26%	49%	32990	1902
stream	26%	46%	34496	1428	27%	49%	32868	1368	28%	49%	32402	1314
tak	28%	67%	31886	282	29%	70%	30334	270	29%	71%	29910	254
Average	27%	56%			27%	60%			28%	61%		

Table 1. Emulator sizes for different instruction sets

predicates, producing specialized versions, etc. using information from analysis –see, e.g., [18] and its references) and then generating the bytecode. This specialization of Prolog programs at the source and module level is done by the Ciao preprocessor and is beyond the scope of this paper.

The **loop** columns contain, right below the label, the size of the main loop of the standard emulator with no specialization. For each benchmark we also show the percentage of reduction achieved with bytecode generated from full or specialized program with respect to the original, non-specialized emulator — the higher, the more savings.

Even in the case when the emulator is specialized with respect to the *full* bytecode, we get a steady savings of around 27%, including library and initialization code. We can deduce that this is a good approximation of the amount of reduction that can be expected from typical programs where no redundant code exists. Of course, programs which use all the available WAM instructions can be crafted on purpose, but this is not the general case. In our experience, not even the compiler itself uses all the abstract machine instructions: we also gen-

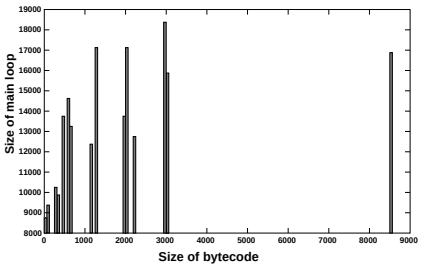

Fig. 10. Relationship between stripped bytecode size (x axis) and emulator size (y axis)

erated an abstract machine specialized for it which was simpler (although only marginally) than the original one.

The savings obtained when the emulator is generated from specialized bytecode are more interesting. Savings range from 45% to 75%, averaging 60%. This shows that substantial size reductions can be obtained with our technique. The absolute sizes do not take into account ancillary pieces, such as I/O and operating system interfaces, which would be compiled or not with the main emulator as necessary, and which are therefore subject to a similar process of selection.

It might be expected that smaller programs would result in more emulator minimization. In general terms this is so, but with a wide variation, as can be seen in Figure 10. Thus, predicting in advance which savings will be obtained from a given benchmark in a precise way is not immediate.

4 Conclusions and Further Work

We have presented the design and implementation of an emulator compiler that generates efficient code using a high-level description of the instruction set of an abstract machine and a set of rules which define how intermediate code is to be represented as bytecode. The approach allowed separating details of the low-level data and code representation from the set of instructions and their semantics. We were therefore able to perform, at the abstract machine description level, transformations which affect both the bytecode format and the generated emulator without sacrificing efficiency.

We have applied our emulator compiler to a description of the abstract machine underlying a production, high-quality, hand-written emulator. The automatically generated emulator is as efficient as the original one. By using a slicer at the level of the abstract machine definition, we were able to reduce automatically its instruction set, producing a smaller, dedicated, but otherwise completely functional, emulator. By changing the definition of the code corresponding to the instructions we were able to produce automatically emulators with substantial internal implementation differences, but still correct and efficient.

We expect to use the emulator compiler to also perform extensive experimentation with variations of abstract machine instruction sets and bytecode representations. We are already applying it in order to generate *ad-hoc* emulators for specific cases, such as those often found in pervasive computing. We are also experimenting with the combination of the emulator minimization with our automatic dead code elimination, slicing, and partial evaluation, in part at the level of the emulator and ancillary machinery and quite fully at the level of $\mathcal{L}_P$ (Ciao/Prolog, in our case) in order to generate high-quality, small executables.

There is also a strong connection with [19]: the fundamental pieces of the C code generation performed there and the code definitions for instructions in $\mathcal{L}_A$ are intimately related, and we have reached a single abstract machine definition in the Ciao system which is used both to generate bytecode emulators and to compile to C code. Also, as in [19], we are using compile-time information (such as type, mode, and determinism information), to generate better $\mathcal{L}_A$ code (e.g., generating specialized instructions or removing unnecessary instructions).

We also plan to redefine and refine the initial instruction set using information from execution profiling in order to merge frequently contiguous instructions, specialize them with respect to some frequently used argument value, etc. These

variations have been explored in [20] for a fixed set of benchmarks, but emulators were hand-coded, somewhat limiting the per-application use of this approach.

References

1. Diaz, D., Codognet, P.: Design and Implementation of the GNU Prolog System. Journal of Functional and Logic Programming **2001** (2001)
2. 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.
3. Santos-Costa, V.: Optimising Bytecode Emulation for Prolog. In: International Conference on Principles and Practice of Declarative Programming (PPDP'99). Volume 1702 of LNCS., Springer-Verlag (1999) 261–277
4. Demoen, B., Nguyen, P.L.: So Many WAM Variations, So Little Time. In: Computational Logic 2000, Springer Verlag (2000) 1240–1254
5. Van Roy, P.: 1983-1993: The Wonder Years of Sequential Prolog Implementation. Journal of Logic Programming **19/20** (1994) 385–441
6. Hannan, J.: Staging Transformations for Abstract Machines. In: Partial Evaluation and Semantics-Based Program Manipulation (PEPM), ACM SigPlan Notices (1991)
7. Jørring, U., Scherlis, W.: Compilers and staging transformations. In: Thirteenth ACM POPL. (1986) 86–96
8. Jones, N., Gomard, C., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall, New York (1993)
9. Futamura, Y.: Partial evaluation of computation process - an approach to a compiler-compiler. Systems, Computers, Controls **2** (1971) 45–50
10. Warren, D.: An Abstract Prolog Instruction Set. Technical Report 309, SRI International (1983)
11. Ait-Kaci, H.: Warren's Abstract Machine, A Tutorial Reconstruction. MIT Press (1991)
12. Ferreira, M., Damas, L.: Multiple Specialization of WAM Code. In: Practical Aspects of Declarative Languages. Number 1551 in LNCS, Springer (1999)
13. Van Roy, P., Despain, A.: High-Performance Logic Programming with the Aquarius Prolog Compiler. IEEE Computer Magazine (1992) 54–68
14. Tip, F.: A Survey of Program Slicing Techniques. Journal of Programming Languages **3** (1995) 121–189
15. Reps, T., Turnidge, T.: Program Specialization via Program Slicing. In Danvy, O., Glück, R., Thiemann, P., eds.: Partial Evaluation. Dagstuhl Castle, Germany, February 1996, Springer LNCS 1110 (1996) 409–429
16. Weiser, M., ed.: Information and Software Technology: Special Issue on Program Slicing. Volume 40. Elsevier (1999)
17. Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-García, P., Puebla, G.: The Ciao Prolog System. Reference Manual (v1.8). Technical Report CLIP4/2002.1, School of Computer Science, UPM (2002) Available at `http://clip.dia.fi.upm.es/Software/Ciao/`.
18. Puebla, G., Hermenegildo, M.: Abstract Specialization and its Applications. In: Proc. of PEPM'03, ACM Press (2003) 29–43 Invited talk.
19. Morales, J., Carro, M., Hermenegildo, M.: Improving the Compilation of Prolog to C Using Moded Types and Determinism Information. In: Intnl. Symposium on Practical Aspects of Declarative Languages. Number 3507 in LNCS, Springer (2004) 86–103
20. Nässén, H., Carlsson, M., Sagonas, K.: Instruction Merging and Specialization in the SICStus Prolog Virtual Machine. In: Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, ACM Press (2001) 49–60