TLP **16** (1): 1–58, 2016. © Cambridge University Press 2015
doi:10.1017/S1471068414000672 First published online 5 January 2015

*Description and Optimization of Abstract Machines in a Dialect of Prolog**

JOSÉ F. MORALES[1], MANUEL CARRO[1,2] and MANUEL HERMENEGILDO[1,2]

[1]*IMDEA Software Institute, Madrid, Spain*
(*e-mail:* {josef.morales,manuel.carro,manuel.hermenegildo}@imdea.org)
[2]*School of Computer Science, Technical University of Madrid (UPM), Madrid, Spain*
(*e-mail:* {mcarro,herme}@fi.upm.es)

submitted 26 January 2007; revised 8 July 2009, 20 April 2014; accepted 26 June 2014

Abstract

In order to achieve competitive performance, abstract machines for Prolog and related languages end up being large and intricate, and incorporate sophisticated optimizations, both at the design and at the implementation levels. At the same time, efficiency considerations make it necessary to use low-level languages in their implementation. This makes them laborious to code, optimize, and, especially, maintain and extend. Writing the abstract machine (and ancillary code) in a higher-level language can help tame this inherent complexity. We show how the semantics of most basic components of an efficient virtual machine for Prolog can be described using (a variant of) Prolog. 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 description 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 state-of-the-art, highly-tuned, hand-crafted emulators.

KEYWORDS: Abstract machines, compilation, optimization, program transformation, prolog, logic languages.

1 Introduction

Abstract machines have proved themselves very useful when defining theoretical models and implementations of software and hardware systems. In particular, they have been widely used to define execution models and as implementation vehicles for many languages, most frequently in functional and logic programming, and more recently also in object-oriented programming, with the Java abstract machine (Gosling *et al.* 2005) being a very popular recent example. There are also

* This is a significantly extended and revised version of the paper "Towards Description and Optimization of Abstract Machines in an Extension of Prolog" published in the proceedings of the *2006 International Symposium on Logic-based Program Synthesis and Transformation* (LOPSTR'06) (Morales *et al.* 2007).

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

early examples of the use of abstract machines in traditional procedural languages (e.g., the P-Code used as a target in early Pascal implementations (Nori *et al.* 1981)) and in other realms such as, for example, operating systems (e.g., Dis, the virtual machine for the Inferno operating system (Dorward *et al.* 1997)).

The practical applications of the indirect execution mechanism that an abstract machine represents are countless: portability, generally small executables, simpler security control through sandboxing, increased compilation speed, etc. However, unless the abstract machine implementation is highly optimized, these benefits can come at the cost of poor performance. Designing and implementing fast, resource-friendly, competitive abstract machines is a complex task. This is especially so in the case of programming languages where there is a wide gap between many of their basic constructs and features and what is available in the underlying off-the-shelf hardware: term representation versus memory words, unification versus assignment, automatic versus manual memory management, destructive versus non-destructive updates, backtracking and tabling versus Von Neumann-style control flow, etc. In addition, the extensive code optimizations required to achieve good performance make development and, especially, maintenance and further modifications non-trivial. Implementing or testing new optimizations is often involved, as decisions previously taken need to be revisited, and low-level, tedious recoding is often necessary to test a new idea.

Improved performance has been achieved by post-processing the input program (often called *bytecode*) of the abstract machine (*emulator*) and generating efficient native code — sometimes achieving performance that is very close to that of an implementation of the source program directly written in C. This is technically challenging and the overall picture is even more complex when bytecode and native code are mixed, usually by dynamic recompilation. This in principle combines the best of both worlds by deciding when and how native code (which may be large and/or costly to obtain) is generated based on runtime analysis of the program execution, while leaving the rest as bytecode. Some examples are the Java HotSpot VM (Paleczny *et al.* 2001), the Psyco (Rigo 2004) extension for Python, or for logic programming, all-argument predicate indexing in recent versions of Yap Prolog (Santos-Costa *et al.* 2007), the dynamic recompilation of asserted predicates in BinProlog (Tarau 2006), etc. Note, however, that the initial complexity of the virtual machine and all of its associated maintenance issues have not disappeared, and emulator designers and maintainers still have to struggle with thousands of lines of low-level code.

In this paper, we explore the possibility of rewriting most of the runtime and virtual machine in the high-level source language (or a close dialect of it), and then use all the available compilation machinery to obtain native code from it. Ideally, this native code should provide comparable performance to that of hand-crafted code, while keeping the size of low-level coded parts in the system to a minimum. This is the approach taken herein, where we explore writing Prolog emulators in a Prolog dialect. As we will see later, the approach is interesting not only for simplifying the task of developers but also for widening the application domain of the language to other kinds of problems which extend beyond just emulators, such as reusing analysis

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

and transformation passes, and making it easier to automate tedious optimization techniques for emulators, such as specializing emulator instructions. The advantages of using a higher-level language are rooted on one hand in the capability of hiding implementation details that a higher-level language provides, and on the other hand in its amenability to transformation and manipulation. These, as we will see, are key for our goals, as they reduce error-prone, tedious programming work, while making it possible to describe at the same time complex abstract machines in a concise and correct way.

A similar objective has been pursued elsewhere. For example, the JavaInJava (Taivalsaari 1998) and PyPy (Rigo and Pedroni 2006) projects have similar goals. The initial performance figures reported for these implementations highlight how challenging it is to make them competitive with existing hand-tuned abstract machines: JavaInJava started with an initial slowdown of approximately 700 times w.r.t. then-current implementations, and PyPy started at the 2,000 times slowdown level. Competitive execution times were only possible after changes in the source language and compilation tool chain, by restricting it or adding annotations. For example, the slowdown of PyPy was reduced to 3.5–11.0 times w.r.t. CPython (Rigo and Pedroni 2006). These results can be partially traced back to the attempt to reimplement the whole machinery at once, which has the disadvantage of making such a slowdown almost inevitable. 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, non-local optimization will carry over to a complete, optimized abstract machine.

Therefore, we chose an alternative approach: gradually porting selected key components (such as, e.g., instruction definitions) and combining this port with other emulator generation techniques (Morales *et al.* 2005). At every step we made sure experimentally[1] that the performance of the original emulator was maintained throughout the process. The final result is an emulator that is completely written in a high-level language and which, when compiled to native code, does not lose any performance w.r.t. a manually written emulator. Also, and as a very relevant byproduct, we develop a language (and a compiler for it) which is high-level enough to make several program transformation and analysis techniques applicable, while offering the possibility of being compiled into efficient native code. While this language is general-purpose and can be used to implement arbitrary programs, throughout this paper, we will focus on its use in writing abstract machines and to easily generate variations of such machines.

The rest of the paper proceeds as follows: Section 2 gives an overview of the different parts of our compilation architecture and information flow and compares it with a more traditional setup. Section 3 presents the source language with which our abstract machine is written, and justifies the design decisions (e.g., typing, destructive updates, etc.) based on the needs of applications which demand high

[1] And also with stronger means: for some core components we checked that the binary code produced from the high-level definition of the abstract machine and that coming from the hand-written one were identical.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

performance. Section 3.4 summarizes how compilation to efficient native code is done through C. Section 4 describes language extensions which were devised specifically to write abstract machines (in particular, the Warren's Abstract Machine (WAM)) and Section 5 explores how the added flexibility of the high-level language approach can be taken advantage of in order to easily generate variants of abstract machines with different core characteristics. This section also studies experimentally the performance that can be attained with these variants. Finally, Section 6 presents our conclusions.

2 Overview of our compilation architecture

The compilation architecture we present here uses several languages, language translators, and program representations which must be defined and placed in the "big picture." For generality, and since those elements are common to most bytecode-based systems, we will refer to them by more abstract names when possible or convenient, although in our initial discussion we will equate them with the actual languages in our production environment.

The starting point in this work is the Ciao system (Hermenegildo *et al.* 2012), which includes an efficient, WAM-based (Warren 1983; Ait-Kaci 1991), abstract machine coded in C (an independent evolution which forked from the SICStus 0.5/0.7 virtual machine), a compiler to bytecode with an experimental extension to emit optimized C code (Morales *et al.* 2004), and the CiaoPP preprocessor, a program analysis, specialization, and transformation framework (Bueno *et al.* 1997; Hermenegildo *et al.* 1999; Puebla *et al.* 2000b; Hermenegildo *et al.* 2005).

We will denote the source Prolog language as $\mathcal{L}_s$, the symbolic WAM code as $\mathcal{L}_a$, the byte-encoded WAM code as $\mathcal{L}_b$, and the C language as $\mathcal{L}_c$. The different languages and translation processes described in this section are typically found in most Prolog systems, and this scheme could be easily applied to other situations. We will use N:L to denote the program N written in language L. Thus, we can distinguish in the *traditional approach* (Fig. 1):

Front-end compiler: P:$\mathcal{L}_s$ is compiled to P:$\mathcal{L}_i$, where $\mathcal{L}_s$ is the source language and $\mathcal{L}_i$ is an intermediate representation. For simplicity, we assume that this phase includes any analysis, transformation, and optimization.

Bytecode back-end: P:$\mathcal{L}_i$ is translated to P:$\mathcal{L}_a$, where $\mathcal{L}_a$ is the symbolic representation of the bytecode language.

Bytecode assembler: P:$\mathcal{L}_a$ is encoded into P:$\mathcal{L}_b$, where $\mathcal{L}_b$ defines encoded bytecode streams (a sequence of bytes whose design is focused on interpretation speed).

To execute $\mathcal{L}_b$ programs, a hand-written emulator E:$\mathcal{L}_c$ (where $\mathcal{L}_c$ is a lower-level language) is required, in addition to some runtime code (written in $\mathcal{L}_c$). Alternatively, it is possible to translate intermediate code by means of a **low-level back end**, which compiles a program P:$\mathcal{L}_i$ directly into its,P:$\mathcal{L}_c$ equivalent[2]. This avoids the need for

[2] Note that in JIT systems, the low-level code is generated from the encoded bytecode representation. For simplicity we keep the figure for static code generation, which takes elements of $\mathcal{L}_i$ as the input.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

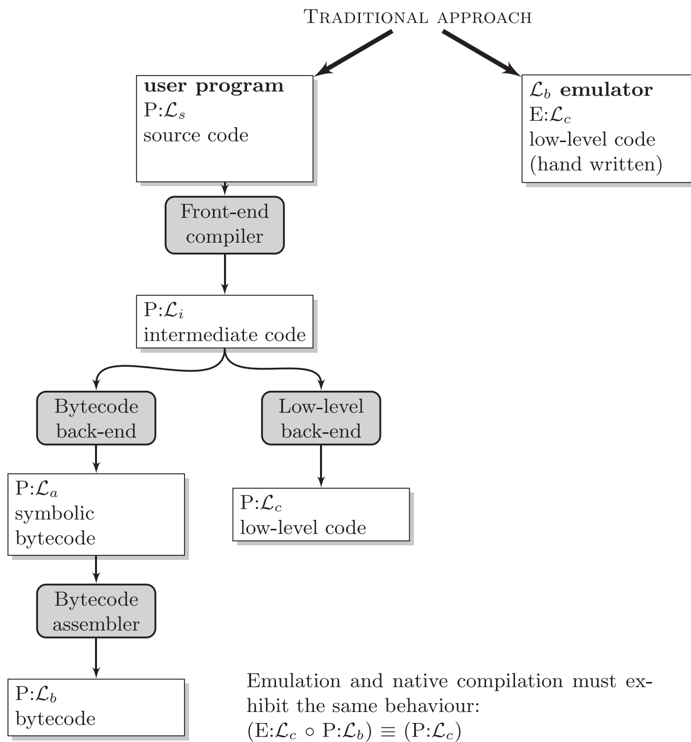

Fig. 1. Traditional compilation architecture.

any emulator if all programs are compiled to native code, but additional runtime support is usually necessary.

The initial *classical architecture* needs manual coding of a large and complex piece of code, the emulator, using a low-level language, often missing the opportunity to reuse compilation techniques that may have been developed for high-level languages, not only to improve efficiency, but also program readability, correctness, etc.

In the *extended compilation scheme* we assume that we can design a dialect from $\mathcal{L}_s$ and write the emulator for symbolic $\mathcal{L}_a$ code, instead of byte-encoded $\mathcal{L}_b$, in that language. We will call $\mathcal{L}_s^r$ the extended language and $\mathrm{E}^r{:}\mathcal{L}_s^r$ the emulator. Note that the mechanical generation of an interpreter for $\mathcal{L}_b$ from an interpreter for $\mathcal{L}_a$ was previously described and successfully tested in an *emulator generator* (Morales *et al.* 2005). Adopting it for this work was perceived as a correct decision, since it moves low-level implementation (and bytecode representation) issues to a translation phase, thus reducing the requirements on the language to preserve emulator efficiency, and in practice making the code easier to manage. The new translation pipeline, depicted in the *extended approach* path (Fig. 2) shows the following processes (the dashed lines represent the modifications that some of the elements undergo with respect to the original ones):

Extended front-end compiler: It compiles $\mathcal{L}_s^r$ programs into $\mathcal{L}_i^r$ programs (where $\mathcal{L}_i^r$ is the intermediate language with extensions required to produce efficient native code). This compiler includes the *emulator generator*. That framework makes it

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

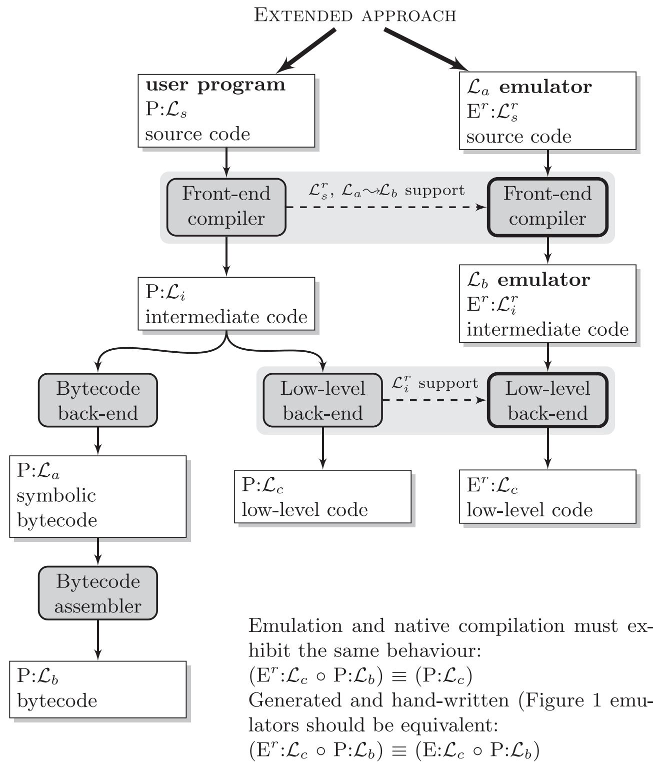

Fig. 2. Extended compilation architecture.

possible to write instruction definitions for an $\mathcal{L}_a$ emulator, plus separate bytecode encoding rules, and process them together to obtain an $\mathcal{L}_b$ emulator. That is, it translates E^r:$\mathcal{L}_s^r$ to an E^r:$\mathcal{L}_i^r$ emulator for $\mathcal{L}_b$.

Extended low-level back-end: It compiles $\mathcal{L}_i^r$ programs into $\mathcal{L}_c$ programs. The resulting E^r:$\mathcal{L}_i^r$ is finally translated to E^r:$\mathcal{L}_c$, which should be equivalent (in the sense of producing the same output) to E:$\mathcal{L}_c$.

The advantage of this scheme lies in its flexibility for sharing optimizations, analyses, and transformations at different compilation stages (e.g., the same machinery for partial evaluation, constant propagation, common subexpression elimination, etc., can be reused), which are normally reimplemented for high- and low-level languages.

3 The imProlog language

We describe in this section our $\mathcal{L}_s^r$ language, imProlog, and the analysis and code generation techniques used to compile it into highly efficient code[3]. This Prolog variant is motivated by the following problems:

[3] The name imProlog stands for *im*perative *Prolog*, because its purpose is to make typically imperative algorithms easier to express in Prolog, but minimizing and controlling the scope of impurities.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

- It is hard to guarantee that certain overheads in Prolog that are directly related with the language expressiveness (e.g., boxed data for dynamic typing, trailing for non-determinism, uninstantiated free variables, multiple-precision arithmetic, etc.) will always be removed by compile-time techniques.
- Even if that overhead could be eliminated, there is also the cost of some basic operations, such as modifying a single attribute in a custom data structure, which is not constant in the declarative subset of Prolog. For example, the cost of replacing the value of an argument in a Prolog structure is, in most straightforward implementations, linear w.r.t. the number of arguments of the structure, since a certain amount of copying of the structure spine is typically involved. In contrast, replacing an element in a C structure is a constant-time operation. Again, while this overhead can be optimized away in some cases during compilation, the objective is to define a language layer in which constant time can be ensured for these operations.

We now present the different elements that comprise the imProlog language and we will gradually introduce a number of restrictions on the kind of programs which we admit as valid. The main reason to impose these restrictions is to achieve the central goal in this paper: generating efficient emulators starting from a high-level language.

In a nutshell, the imProlog language both restricts and extends Prolog. The impure features (e.g., the dynamic database) of Prolog are not part of imProlog and only programs meeting some strict requirements about determinism, modes in the unification, and others, are allowed. In a somewhat circular fashion, the requirements we impose on these programs are those which allow us to compile them into efficient code. Therefore, implementation matters somewhat influence the design and semantics of the language. On the other hand, imProlog also extends Prolog in order to provide a single, well-defined, and amenable to analysis mechanism to implement constant-time access to data: a specific kind of mutable variables. Thanks to the restrictions aforementioned and this mechanism imProlog programs can be compiled into very efficient low-level (e.g., C) code.

Since we are starting from Prolog, which is well understood, and the restrictions on the admissible programs can be introduced painlessly (they just reduce the set of programs which are deemed valid by the compiler), we will start by showing, in the next section, how we tackle efficient data handling, which as we mentioned departs significantly, but in a controlled way, from the semantics of Prolog.

Notation. We will use lowercase math font for variables (x, y, z, ...) in the rules that describe the compilation and language semantics. Prolog variable names will be written in math capital font (X, Y, Z, ...). Keywords and identifiers in the target C language use bold text (**return**). Finally, sans serif text is used for other names and identifiers (f, g, h, ...). The typography will make it possible to easily distinguish a compilation pattern for "f(a)", where "a" may be any valid term, and "f(A)", where "A" is a Prolog variable with the concrete name "A." Similarly, the expression $f(a_1, \ldots, a_n)$ denotes any structure with functor name f and n arguments, whatever they may be. It differs from f(A_1, ..., A_n), where the functor name is fixed to f

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

and the arguments are variables). If n is 0, the previous expression is tantamount to just f.

3.1 Efficient mechanisms for data access and update

In this section, we will describe the formal semantics of *typed mutable variables*, our proposal for providing efficient (in terms of time and memory) data handling in imProlog. These variables feature backtrackable destructive assignment and are accessible and updatable in constant time through the use of a unique, associated identifier. This is relevant for us as it is required to efficiently implement a wide variety of algorithms, some of which appear in WAM-based abstract machines[4], which we want to describe in enough detail as to obtain efficient executables.

There certainly exist a number of options for implementing constant-time data access in Prolog. Dynamic predicates (assert/retract) can in some cases (maybe with the help of type information) provide constant-time operations; existing destructive update primitives (such as setarg/3) can do the same. However, it is difficult for the analyses normally used in the context of logic programming to deal with them in a precise way, in a significant part because their semantics was not devised with analysis in mind, and therefore they are difficult to optimize as much as we need herein.

Therefore, we opted for a generalization of mutable variables with typing constraints as mentioned before. In our experience, this fits in a less intrusive way with the rest of the logic framework, and at the same time allows us to generate efficient code for the purpose of the work in this paper[5]. Let us introduce some preliminary definitions before presenting the semantics of mutable variables:

Type: τ is a unary predicate that defines a set of values (e.g., regular types as in Dart and Zobel (1992), Gallagher and de Waal (1994) and Vaucheret and Bueno (2002)). If $\tau(v)$ holds, then v is said to contain values of type τ.

Mutable identifier: The identifier *id* of a mutable is a unique atomic value that uniquely identifies a mutable and does not unify with any other non-variable except itself.

Mutable store: φ is a mapping $\{id_1/(\tau_1, val_1), \ldots, id_n/(\tau_n, val_n)\}$, where id_i are mutable identifiers, val_i are terms, and τ_i type names. The expression $\varphi(id)$ stands for the pair (τ, val) associated with id in φ, while $\varphi[id/(\tau, val)]$ denotes a mapping φ' such that

$$\varphi'(id_i) = \begin{cases} (\tau, val) & \text{if } id_i = id \\ \varphi(id_i) & \text{otherwise.} \end{cases}$$

We assume the availability of a function new_id(φ) that obtains a new unique identifier not present in φ.

[4] One obvious example is the unification algorithm of logical variables, itself based on the union-find algorithm. An interesting discussion of this point is available in Schrijvers and Frühwirth (2006), where a CHR version of the union-find algorithm is implemented and its complexity studied.

[5] Note that this differs from Morales *et al.* (2007), where changes in mutable data were non-backtrackable side effects. Notwithstanding, performance is not affected in this work, since we restrict at compile-time the emulator instructions to be deterministic.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

$$\text{(M-NEW)}\ \frac{\vdash\ \tau(v) \quad id = \mathsf{new_id}(\varphi_0) \quad \varphi = \varphi_0[id/(\tau,\, v)] \quad \vdash\ m = id}{\varphi_0 \rightsquigarrow \varphi\ \vdash\ m = \mathsf{initmut}(\tau,\, v)}$$

$$\text{(M-READ)}\ \frac{\vdash\ m = id \quad (_,\, v) = \varphi(id) \quad \vdash\ x = v}{\varphi \rightsquigarrow \varphi\ \vdash\ x = m@}$$

$$\text{(M-ASSIGN)}\ \frac{\vdash\ m = id \quad (\tau,\, _) = \varphi_0(id) \quad \vdash\ \tau(v) \quad \varphi = \varphi_0[id/(\tau,\, v)]}{\varphi_0 \rightsquigarrow \varphi\ \vdash\ m \Leftarrow v}$$

$$\text{(M-TYPE)}\ \frac{\vdash\ m = id \quad (\tau,\, _) = \varphi(id)}{\varphi \rightsquigarrow \varphi\ \vdash\ \mathsf{mut}(\tau,\, m)}$$

$$\text{(M-WEAK)}\ \frac{\vdash\ a}{\varphi \rightsquigarrow \varphi\ \vdash\ a}$$

$$\text{(M-CONJ)}\ \frac{\varphi_0 \rightsquigarrow \varphi_1\ \vdash\ a \quad \varphi_1 \rightsquigarrow \varphi\ \vdash\ b}{\varphi_0 \rightsquigarrow \varphi\ \vdash\ (a \wedge b)}$$

$$\text{(M-DISJ-1)}\ \frac{\varphi_0 \rightsquigarrow \varphi\ \vdash\ a}{\varphi_0 \rightsquigarrow \varphi\ \vdash\ (a \vee b)} \qquad \text{(M-DISJ-2)}\ \frac{\varphi_0 \rightsquigarrow \varphi\ \vdash\ b}{\varphi_0 \rightsquigarrow \varphi\ \vdash\ (a \vee b)}$$

Fig. 3. Rules for the implicit mutable store (operations and logical connectives).

Mutable environment: By $\varphi_0 \rightsquigarrow \varphi \vdash g$ we denote a judgment of g in the context of a mutable environment (φ_0, φ). The pair (φ_0, φ) relates the initial and final mutable stores, and the interpretation of the g explicit.

We can now define the rules that manipulate mutable variables (Fig. 3). For the sake of simplicity, they do not impose any evaluation order. In practice, and in order to keep the computational cost low, we will use the Prolog resolution strategy, and impose limitations on the instantiation level of some particular terms; we postpone discussing this issue until Section 3.2:

Creation: The (M-NEW) rule defines the creation of new mutable placeholders. A goal $m = \mathsf{initmut}(\tau, v)$[6] checks that $\tau(v)$ holds (i.e., v has type τ), creates a new mutable identifier id that does not appear as a key in φ_0 and does not unify with any other term in the program, defines the updated φ as φ_0 where the value associated with id is v, and unifies m with id. These restrictions ensure that m is an unbound variable before the mutable is created.

Access: Reading the contents of a mutable variable is defined in the (M-READ) rule. The goal $x = m@$ holds if the variable m is bound with a mutable identifier id, for which an associated v value exists in the mutable store φ, and the variable x unifies with v[7].

Assignment: Assignment of values to mutables is described in the (M-ASSIGN) rule. The goal $m \Leftarrow v$, which assigns a value to a mutable identifier, holds iff:

- m is unified with a mutable identifier id, for which a value is stored in φ_0 with associated type τ;

[6] To improve readability, we use the functional notation of Casas *et al.* (2006) for the new initmut/3 and (@)/2 *built-in* predicates.

[7] Since the variable in the mutable store is constrained to the type, it is not necessary to check that x belongs to that type.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

- v has type τ, i.e., the value to be stored is compatible with the type associated with the mutable;
- φ_0 is the result of replacing the associated type and value for id by τ and v, respectively.

Typing: The (M-TYPE) rule allows checking that a variable contains a mutable identifier of a given type. A goal $\mathsf{mut}(\tau, m)$ is true if m is unified with a mutable identifier id that is associated with the type τ in the mutable store φ.

Note that although some of the rules above enforce typing constraints, the compiler, as we will see, is actually able to statically remove these checks and there is no dynamic typing involved in the execution of admissible imProlog programs. The rest of the rules define how the previous operations on mutable variables can be joined and combined together, and with other predicates:

Weakening: The weakening rule (M-WEAK) states that if some goal can be solved without a mutable store, then it can also be solved in the context of a mutable store that is left untouched. This rule allows the integration of the new rules with predicates (and built-ins) that do not use mutables.

Conjunction: The (M-CONJ) rule defines how to solve a goal $a \wedge b$ (written as (a, b) in code) in an environment where the input mutable store φ_0 is transformed into φ, by solving a and b and connecting the output mutable store of the former (φ_1) with the input mutable store of the latter. This conjunction is not commutative, since the updates performed by a may alter the values read in b. If none of those goals modify the mutable store, then commutativity can be ensured. If none of them access the mutable store, then it is equivalent to the classic definition of conjunction (by applying the (M-WEAK) rule).

Disjunction: The disjunction of two goals is defined in the (M-DISJ) rule, where $a \vee b$ (written as $(a ; b)$ in code) holds in a given environment if either a or b holds in such environment, with no relation between the mutable stores of both branches. That means that changes in the mutable store would be *backtrackable* (e.g., any change introduced by an attempt to solve one branch must be undone when trying another alternative). As with conjunction, if the goals in the disjunction do not update nor access the mutable store, then it is equivalent to the classic definition of disjunction (by applying the (M-WEAK) rule).

Mutable terms, conceptually similar to the mutables we present here, were introduced in SICStus Prolog as a replacement for setarg/3, and also appear in proposals for global variables in logic programming (such as Schachte (1997), Bart Demoen's implementation of (non)backtrackable global variables for hProlog/SWI-Prolog (Demoen *et al.* 1998)), or imperative assignment (Giannesini *et al.* 1985). In the latter case, there was no notion of types and the terms assigned to had to be (ground) atoms at the time of assignment.

We will consider that two types unify if their names match. Thus, typing in mutables divide the mutable store into separate, independent regions, which will facilitate program analysis. For the purpose of this work, we will treat mutable variables as a native part of the language. It would however be possible to emulate

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

the mutable store as a pair of additional arguments, threaded from left to right in the goals of the predicate bodies. A similar translation is commonly used to implement DCGs or state variables in Mercury (Somogyi *et al.* 1996).

3.2 Compilation strategy

In the previous section, the operations on mutable data were presented separately from the host language, and no commitment was made regarding their implementation other than assuming that it could be done efficiently. However, when the host language $\mathcal{L}_s^r$ has a Prolog-like semantics (featuring unification and backtracking) and even if backtrackable destructive assignment is used, the compiled code can be unaffordably inefficient for deterministic computations unless extensive analysis and optimization is performed. On the other hand, the same computations may be easy to optimize in lower-level languages.

A way to overcome those problems is to specialize the translated code for a relevant subset of the initial input data. This subset can be *abstractly* specified: let us consider a predicate bool/1 that defines truth values, and a call bool(X) where X is known to be always bound to a dereferenced atom. The information about the dereferencing state of X is a partial specification of the initial conditions, and replacing the call to the generic bool/1 predicate by a call to another predicate that implements a version of bool/1 that avoids unnecessary work (e.g., there is no need for *choice points* or *tag* testing on X) produces the same computation, but using code that is both shorter and faster. To be usable in the compilation process, it is necessary to propagate this knowledge about the program behavior as predicate-level assertions or program-point annotations, usually by means of automatic methods such as static analysis. Such techniques has been tested elsewhere (Warren 1977; Taylor 1991; Van Roy and Despain 1992; Van Roy 1994; Morales *et al.* 2004).

This approach (as most automatic optimization techniques) has obviously its own drawbacks when high performance is a requirement: (a) the analysis is not always precise enough, which makes the compiler require manual annotations or generate under-optimized programs; (b) the program is not always optimized, even if the analysis is able to infer interesting properties about it, since the compiler may not be clever enough to improve the algorithm; and (c) the final program performance is hard to predict, as we leave more optimization decisions to the compiler, of which the programmer may not be aware.

For the purposes of this paper, cases (a) and (b) do not represent a major problem. Firstly, if some of the annotations cannot be obtained automatically and need to be provided by hand, the programming style still encourages separation of optimization annotations (as hints for the compiler) and the actual algorithm, which we believe makes code easier to manage. Secondly, we adapted the language imProlog and compilation process to make the algorithms implemented in the emulator easier to represent and compile. For case (c), we took an approach different from that in other systems. Since our goal is to generate low-level code that *ensures* efficiency, we impose some constraints on the compilation output to avoid generation of code known to be suboptimal. This restricts the admissible code and the compiler informs

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

pred	::=	*head* :- *body*	(Predicates)
head	::=	$\lfloor\beta\rfloor id(var, ..., var)\lfloor\beta\rfloor$	(Heads)
body	::=	$goals \mid (goals \rightarrow goals\ ;\ body)$	(Body)
goals	::=	$\lfloor\beta\rfloor goal, ..., \lfloor\beta\rfloor goal$	(Conjunction of goals)
goal	::=	$var = var \mid var = cons \mid$	(Unifications)
		$var = var$@ $\mid var \Leftarrow var \mid$	(Mutable ops.)
		$id(var, ..., var)$	(Built-In/User call)
var	::=	uppercase name	(Variables)
id	::=	lowercase name	(Atom names)
cons	::=	atom names, integers, floats, characters, etc.	(Other constants)
β	::=	abstract substitution	(Program point anots.)

Fig. 4. Syntax of normalized programs.

the user when the constraints do not hold, by reporting efficiency errors. This is obviously too drastic a solution for general programs, but we found it a good compromise in our application.

3.2.1 *Preprocessing imProlog programs*

The compilation algorithm starts with the expansion of syntactic extensions (such as, e.g., functional notation), followed by normalization and analysis. Normalization is helpful to reduce programs to simpler building blocks for which compilation schemes are described.

The syntax of the normalized programs is shown in Figure 4, and is similar to that used in `ciaocc` (Morales *et al.* 2004). It focuses on simplifying code generation rules and making analysis information easily accessible. Additionally, operations on mutable variables are considered built-ins. Normalized predicates contain a single clause, composed of a head and a body which contains a conjunction of goals or *if-then-else*s. Every goal and head are prefixed with *program point* information which contains the abstract substitution inferred during analysis, relating every variable in the goal/head with an abstraction of its value or state. However, compilation needs that information to be also available for every temporary variable that may appear during code generation and which is not yet present in the normalized program. In order to overcome this problem, most auxiliary variables are already introduced before the analysis. The code reaches a homogeneous form by requiring that both head and goals contain only syntactical variables as arguments, and making unification and matching explicit in the body of the clauses. Each of these basic data-handling steps can therefore be annotated with the corresponding abstract state. Additionally, unifications are restricted to the *variable–variable* and *variable–constant* cases. As we will see later, this is enough for our purposes.

Program transformations. Normalization groups the bodies of the clauses of the same predicate in a disjunction, sharing common new variables in the head arguments, introducing unifications as explained before, and taking care of renaming variables local to each clause[8]. As a simplification for the purposes of this paper, we restrict

[8] This is required to make their information independent in each branch during analysis and compilation.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

ourselves to treating atomic, ground data types in the language. Structured data is created by invoking built-in or predefined predicates. Control structures such as disjunctions (_ ; _) and negation (\+ _) are only supported when they can be translated to *if-then-else*s, and a compilation error is emitted (and compilation is aborted) if they cannot. If cuts are not explicitly written in the program, mode analysis and determinism analysis help in detecting the mutually exclusive prefixes of the bodies, and delimit them with cuts. Note that the restrictions that we impose on the accepted programs make it easier to treat some *non-logical* Prolog features, such as *red* cuts, which make the language semantics more complex but are widely used in practice. We allow the use of *red* cuts (explicitly or as $(\ldots \rightarrow \ldots ; \ldots)$ constructs) as long as it is possible to insert a mutually exclusive prefix in all the alternatives of the disjunctions where they appear: $(b_1, !, b_2 ; b_3)$ is treated as equivalent to $(b_1, !, b_2 ; \backslash + b_1, !, b_3)$ if analysis (e.g., Debray *et al.* (1997)) is able to determine that b_1 does not generate multiple solutions and does not further instantiate any variables shared with the head or the rest of the alternatives.

Predicate and program point information. The information that the analysis infers from (and is annotated in) the normalized program is represented using the Ciao assertion language (Hermenegildo *et al.* 1999; Puebla *et al.* 2000a; Hermenegildo *et al.* 2005) (with suitable property definitions and mode macros for our purposes). This information can be divided into predicate-level assertions and program point assertions. Predicate-level assertions relate an abstract input state with an output state $(\lfloor\beta_0\rfloor f(a_1, \ldots, a_n)\lfloor\beta\rfloor)$, or state facts about some properties (see below) of the predicate. Given a predicate f/n, the properties needed for the compilation rules used in this work are as follows:

- $\mathsf{det}(f/n)$: The predicate f/n is deterministic (it has exactly one solution).
- $\mathsf{semidet}(f/n)$: The predicate f/n is semideterministic (it has one or zero solutions).

We assume that there is a single call pattern, or that all the possible call patterns have been aggregated into a single one, i.e., the analysis we perform does not take into account the different modes in which a single predicate can be called. Note that this does not prevent effectively supporting different separate call patterns, as a previous specialization phase can generate a different predicate version for each calling pattern.

The second kind of annotations keeps track of the abstract state of the execution at each program point. For a goal $\lfloor\beta\rfloor g$, the following judgments are defined on the abstract substitution β on variables of g:

- $\beta \vdash \mathsf{fresh}(x)$: The variable x is a fresh variable (not instantiated to any value, not sharing with any other variable).
- $\beta \vdash \mathsf{ground}(x)$: The variable x contains a ground term (it does not contain any free variable).
- $\beta \vdash x : \tau$: The values that the variable x can take at this program point are of type τ.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

3.2.2 Overview of the analysis of mutable variables

The basic operations on mutables are restricted to some instantiation state on input and have to obey to some typing rules. In order to make the analysis as parametric as possible to the concrete rules, these are stated using the following assertions on the predicates @/2, initmut/3, and ⇐/2:

```
:- pred @(+mut(T), -T).
:- pred initmut(+(^T), +T, -mut(T)).
:- pred (+mut(T)) <= (+T).
```

These state that

- Reading the value associated with a mutable identifier of type mut(T) (which must be ground) gives a value type T.
- Creating a mutable variable with type T (escaped in the assertion to indicate that it is the type name that is provided as argument, not a value of that type) takes an initial value of type T and gives a new mutable variable of type mut(T).
- Assigning a mutable identifier (which must be ground and of type mut(T)) a value requires the latter to be of type T.

Those assertions instruct the type analysis about the meaning of the built-ins, requiring no further changes w.r.t. equivalent[9] type analyses for plain Prolog. However, in our case more precision is needed. For example, given mut(int, A) and ($A \Leftarrow 3$, p(A@)) we want to infer that p/1 is called with an integer value 3 and not with any integer (as inferred using just the assertion). With no information about the built-ins, that code is equivalent to ($T_0 = 3$, $A \Leftarrow T_0$, $T_1 = A$@, p(T_1)), and no relation between T_0 and T_1 is established.

However, based on the semantics of mutables variables and their operations (Fig. 3), it is possible to define an analysis based on abstract interpretation to infer properties of the values stored in the mutable store. To natively understand the built-ins, it is necessary to abstract the mutable identifiers and the mutable store, and represent it in the abstract domain, for which different options exist.

One is to explicitly keep the relation between the abstraction of the mutable identifier and the variable containing its associated value. For every newly created mutable or mutable assignment, the associated value is changed, and the previous code would be equivalent to ($T = 3$, $A \Leftarrow T$, $T = A$@, p(T)). The analysis in this case will lose information when the value associated with the mutable is unknown. That is, given mut(int, A) and mut(int, B), it is not possible to prove that $A \Leftarrow 3$, p(B@) will not call p/1 with a value of 3.

Different abstractions of the mutable identifiers yield different precision levels in the analysis. For example, given an abstract domain for mutable identifiers that distinguishes newly created mutables, the chunk of code ($A =$ initmut(int, 1), $B =$

[9] In the sense that the behavior of the built-ins is not hard-wired into the analysis itself.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

initmut(int, 2), $A \Leftarrow 3$, p(B@)) has enough information to ensure that B is unaffected by the assignment to A. In the current state, and for the purpose of the paper, the abstraction of mutable identifiers is able to take into account newly created mutables and mutables of a particular type. When an assignment is performed on an unknown mutable, it only needs to change the values of mutables of exactly the same type, improving precision[10]. If mutable identifiers are interpreted as pointers, that problem is related to *pointer aliasing* in imperative programming (see Hind and Pioli (2000) for a tutorial overview).

3.3 Data representation and operations

Data representation in most Prolog systems often chooses a general mapping from Prolog terms and variables to C data so that full unification and backtracking can be implemented. However, for the logical and mutable variables of imProlog, we need the least expensive mapping to C types and variables possible, since anything else would bring an unacceptable overhead in critical code (such as emulator instructions). A general way to overcome this problem, which is taken in this work, is to start from a general representation and replacing it by a more specific encoding.

Let us recall the general representation in WAM-based implementations (Warren 1983; Ait-Kaci 1991). The abstract machine state is composed of a set of registers and stacks of memory cells. The values stored in those registers and memory cells are called *tagged* words. Every tagged word has a *tag* and a *value*, the tag indicating the kind of value stored in it. Possible values are constants (such as integers up to some fixed length, indexes for atoms, etc.), or pointers to larger data which does not fit in the space for the value (such as larger integers, multiple precision numbers, floating point numbers, arrays of arguments for structures)[11]. There exist a special *reference* tag that indicates that the value is a pointer to another cell. That reference tag allows a cell to point to itself (for unbound variables), or set of cells point to the same value (for unified variables). Given our assumption that mutable variables can be efficiently implemented, we want to point out that these representations can be extended for this case, using, for example, an additional *mutable* tag, to denote that the value is a pointer to another cell which contains the associated value. How terms are created and unified using a tagged cell representation is well described in the relevant literature.

When a Prolog-like language is (naively) translated to C, a large part of the overhead comes from the use of tags (including bits reserved for automatic memory management) and machine words with fixed sizes (e.g., `unsigned int`) for tagged cells. If we are able to enforce a fixed tag for every variable (which we can in principle map to a word) at compile time at every program point, those additional

[10] That was enough to specialize pieces of imProlog code implementing the unification of tagged words, which was previously optimized by hand.

[11] Those will be ignored in this paper, since all data can be described using atomic constants, mutable identifiers, and built-ins to control the emulator stacks.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

tag bits can be removed from the representation and the whole machine word can be used for the value. This makes it possible to use different C types for each kind of value (e.g., `char`, `float`, `double`, etc.). Moreover, the restrictions that we have imposed on program determinism (Section 3.2.1), variable scoping, and visibility of mutable identifiers make trailing unnecessary.

3.3.1 C types for values

The associated C type that stores the value for an imProlog type τ is defined in a two-layered approach. First, the type τ is inferred by means of Prolog type analysis. In our case, we are using the regular type analysis of Vaucheret and Bueno (2002), which can type more programs than a Hindley–Damas–Milner type analysis (Damas and Milner 1982). Then, for each variable a compatible *encoding type*, which contains all the values that the variable can take, is assigned, and the corresponding C type for that encoding type is used. Encoding types are Prolog types annotated with an assertion that indicates the associated C type. A set of heuristics is used to assign economic encodings so that memory usage and type characteristics are adjusted as tightly as possible. Consider, for example, the type flag/1 defined as

```
:- regtype flag/1 + low(int32).
flag := off | on.
```

It specifies that the values in the declared type must be represented using the `int32` C type. In this case, off will be encoded as `0` and on encoded as `1`. The set of available encoding types must be fixed at compile time, either defined by the user or provided as libraries. Although, it is not always possible to automatically provide a mapping, we believe that this is a viable alternative to more restrictive typing options, such as Hindley–Damas–Milner based typings. A more detailed description of data types in imProlog is available in (Morales *et al.* 2008).

3.3.2 Mapping imProlog variables to C variables

The logical and mutable variables of imProlog are mapped onto imperative, low-level variables which can be global, local, or passed as function arguments. Thus, pointers to the actual memory locations where the value, mutable identifier, or mutable value are stored may be necessary. However, as stated before, we need to statically determine the number of references required. The *reference mode*s of a variable will define the shape of the memory cell (or C variable), indicating how the value or mutable value is accessed

- **0v**: the cell contains the value.
- **1v**: the cell contains a pointer to the value.
- **0m**: the cell contains the mutable value.
- **1m**: the cell contains a pointer to the mutable cell, which contains the value.
- **2m**: the cell contains a pointer to another cell, which contains a pointer to the mutable cell, which contains the mutable value.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

Table 1. *Operation and translation table for different mapping modes of imProlog variables*

	refmode(x)				
	0v	**1v**	**0m**	**1m**	**2m**
Final C type	$c\tau$	$c\tau$ *	$c\tau$	$c\tau$ *	$c\tau$ * *
ref_rval$[\![x]\!]$	&x	x	-	&x	x
val_lval$[\![x]\!]$	x	*x	-	x	*x
val_rval$[\![x]\!]$	x	*x	&x	x	*x
mutval/val_lval$[\![x]\!]$			x	*x	**x
mutval/val_rval$[\![x]\!]$			x	*x	**x

For an imProlog variable x, with associated C symbol x, and given the C type for its value $c\tau$, Table 1 gives the full C type definition to be used in the variable declaration, the *r-value* (for the left part of C assignments) and *l-value* (as C expressions) for the reference (or address) to the variable, the variable value, the reference to the mutable value, and the mutable value itself. These definitions relate C and imProlog variables and will be used later in the compilation rules. Note that the translation for ref_rval and val_lval is not defined for **0m**. That indicates that it is impossible to modify the mutable identifier itself for that mutable, since it is fixed. This tight mapping to C types, avoiding when possible unnecessary indirections, allows the C compiler to apply optimizations such as using machine registers for mutable variables.

The following algorithm infers the *reference mode* (refmode(_)) of each predicate variable making use of type and mode annotations:

(1) Given the head $\lfloor\beta_0\rfloor f(a_1, \ldots, a_n)\lfloor\beta\rfloor$, the ith-argument mode argmode(f/n, i) for a predicate argument a_i, is defined as:

$$\text{argmode}(f/n, i) = \begin{cases} \text{in} & \text{if } \beta \vdash \text{ground}(a_i) \\ \text{out} & \text{if } \beta_0 \vdash \text{fresh}(a_i),\ \beta \vdash \text{ground}(a_i). \end{cases}$$

(2) For each predicate argument a_i, depending on argmode(f/n, a_i):

- If argmode(f/n, a_i) = in, then
  — if $\beta \vdash a_i$:mut(t) then refmode(a_i) = **1m**, else refmode(a_i) = **0v**.
- If argmode(f/n, a_i) = out, then
  — if $\beta \vdash a_i$:mut(t) then refmode(a_i) = **2m**, else refmode(a_i) = **1v**.

(3) For each unification $\lfloor\beta\rfloor a = b$:

- if $\beta \vdash$ fresh(a), $\beta \vdash$ ground(b), $\beta \vdash b$:mut(t), then refmode(a) = **1m**.
- Otherwise, if $\beta \vdash$ fresh(a), then refmode(a) = **0v**.

(4) For each mutable initialization $\lfloor\beta\rfloor a$ = initmut(t, b):

- if $\beta \vdash$ fresh(a), $\beta \vdash$ ground(b), $\beta \vdash b$: mut(t), then refmode(a) = **0m**.

(5) Any case not covered above is a compile-time error.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

Escape analysis of mutable identifiers. According to the compilation scheme we follow, if a mutable variable identifier cannot be reached outside the scope of a predicate, it can be safely mapped to a (local) C variable. That requires the equivalent of escape analysis for mutable identifiers. A conservative approximation to decide that mutables can be assigned to local C variables is the following: the mutable variable identifier 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. This is easy to check and has been precise enough for our purposes.

3.4 Code generation rules

Compilation processes a set of predicates, each one composed of a *head* and *body* as defined in Section 3.2.1. The body can contain control constructs, calls to user predicates, calls to built-ins, and calls to external predicates written in C. For each of these cases we will summarize the compilation as translation rules, where p stands for the predicate compilation output that stores the C functions for the compiled predicates. The compilation state for a predicate is denoted as θ, and it is composed of a set of variable declarations and a mapping from identifiers to *basic blocks*. Each basic block, identified by δ, contains a sequence of sentences and a terminal control sentence.

Basic blocks are finally translated to C code as labels, sequences of sentences, and jumps or conditional branches generated as *gotos* and *if-then-else*s. Note that the use of labels and jumps in the generated code should not make the C compiler generate suboptimal code, as simplification of control logic to basic blocks and jumps is one of the first steps performed by C compilers. It was experimentally checked that using *if-then-else* constructs (when possible) does not necessarily help mainstream C compilers in generating better code. In any case, doing so is a code generation option.

For simplicity, in the following rules we will use the syntax $\langle\theta_0\rangle\ \forall i{=}1,\ldots,n\ g\ \langle\theta_n\rangle$ to denote the evaluation of g for every value of i between 1 and n, where the intermediate states θ_j are adequately threaded to link every state with the following one.

3.4.1 Compilation of goals

The compilation of goals is described by the rule $\langle\theta_0\rangle\ \mathsf{gcomp}(goal, \eta, \delta) \Rightarrow \langle\ \theta\rangle$. η is a mapping which goes from continuation identifiers (e.g., s for the success continuation, f for the failure continuation, and possibly more identifiers for other continuations, such as those needed for exceptions) to basic blocks identifiers. Therefore, $\eta(\mathsf{s})$ and $\eta(\mathsf{f})$ denote the continuation addresses in case of success (resp. failure) of *goal*. The compilation state θ is obtained from θ_0 by *appending* the generated code for *goal* to the δ basic block, and optionally introducing more basic blocks connected by the continuations associated to them.

The rules for the compilation of control are presented in Figure 5. We will assume some predefined operations to request a new basic block identifier (bb_new) and a list of new identifiers (bb_newn), and to add a C sentence to a given basic block

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

$$(\textsc{Conj})\ \frac{\langle\theta_0\rangle\ \mathsf{bb_new} \Rightarrow \delta_b\ \langle\theta_1\rangle \quad \langle\theta_1\rangle\ \mathsf{gcomp}(a,\ \eta[\mathsf{s}{\mapsto}\delta_b],\ \delta) \Rightarrow \langle\theta_2\rangle \quad \langle\theta_2\rangle\ \mathsf{gcomp}(b,\ \eta,\ \delta_b) \Rightarrow \langle\theta\rangle}{\langle\theta_0\rangle\ \mathsf{gcomp}((a,\ b),\ \eta,\ \delta) \Rightarrow \langle\theta\rangle}$$

$$(\textsc{IfThenElse})\ \frac{\langle\theta_0\rangle\ \mathsf{bb_newn}(2) \Rightarrow [\delta_t,\ \delta_e]\ \langle\theta_1\rangle \quad \langle\theta_1\rangle\ \mathsf{gcomp}(a,\ \eta[\mathsf{s}{\mapsto}\delta_t,\ \mathsf{f}{\mapsto}\delta_e],\ \delta) \Rightarrow \langle\theta_2\rangle \quad \langle\theta_2\rangle\ \mathsf{gcomp}(then,\ \eta,\ \delta_t) \Rightarrow \langle\theta_3\rangle \quad \langle\theta_3\rangle\ \mathsf{gcomp}(else,\ \eta,\ \delta_e) \Rightarrow \langle\theta\rangle}{\langle\theta_0\rangle\ \mathsf{gcomp}((a \rightarrow then\ ;\ else),\ \eta,\ \delta) \Rightarrow \langle\theta\rangle}$$

$$(\textsc{True})\ \frac{\langle\theta_0\rangle\ \mathsf{emit}(\mathbf{goto}\ \eta(\mathsf{s}),\ \delta) \Rightarrow \langle\theta\rangle}{\langle\theta_0\rangle\ \mathsf{gcomp}(\mathsf{true},\ \eta,\ \delta) \Rightarrow \langle\theta\rangle} \qquad (\textsc{Fail})\ \frac{\langle\theta_0\rangle\ \mathsf{emit}(\mathbf{goto}\ \eta(\mathsf{f}),\ \delta) \Rightarrow \langle\theta\rangle}{\langle\theta_0\rangle\ \mathsf{gcomp}(\mathsf{fail},\ \eta,\ \delta) \Rightarrow \langle\theta\rangle}$$

Fig. 5. Control compilation rules.

(emit). The conjunction (a, b) is translated by rule (Conj) by reclaiming a new basic block identifier δ_b for the subgoal b, generating code for a in the target δ, using as success continuation δ_b, and then generating code for b in δ_b. The construct $(a \rightarrow b\ ;\ c)$ is similarly compiled by the (IfThenElse) rule. The compilation of a takes place using as success and failure continuations, the basic block identifiers where b and c are emitted, respectively. Then, the process continues by compiling both b and c using the original continuations. The goals true and fail are compiled by emitting a jump statement (**goto** _) that goes directly to the success and failure continuation (rules (True) and (Fail)).

As stated in Section 3.2.1, there is no compilation rule for disjunctions $(a\ ;\ b)$. Nevertheless, program transformations can change them into *if-then-else* structures, following the constraints on the input language. For example, $(X = 1\ ;\ X = 2)$ is accepted if X is ground on entry, since the code can be translated into the equivalent $(X = 1 \rightarrow \mathsf{true}\ ;\ X = 2 \rightarrow \mathsf{true})$. It will not be accepted if X is unbound, since the *if-then-else* code and the initial disjunction are not equivalent.

Note that since continuations are taken using C **goto** _ statements, there is a great deal of freedom in the physical ordering of the basic blocks in the program. The current implementation emits code in an order roughly corresponding to the source program, but it has internal data structures which make it easy to change this order. Note that different orderings can impact performance, by, for example, changing code locality, affecting how the processor speculative execution units perform, and changing which **goto** _ statements which jump to an immediate label can be simplified by the compiler.

3.4.2 Compilation of goal calls

External predicates explicitly defined in C and user predicates compiled to C code have both the same external interface. Thus we use the same call compilation rules for them.

Predicates that may fail are mapped to functions with *Boolean* return types (indicating success/failure), and those which cannot fail are mapped to procedures (with no return result — as explained later in Section 3.4.4). Figure 6 shows the rules to compile calls to external or user predicates. Function argpass(f/n) returns the list $[r_1, \ldots, r_n]$ of argument passing modes for predicate f/n. Depending on

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

$$\text{(CALL-S)}\ \frac{\begin{array}{l}\mathsf{semidet}(f/n)\\ [r_1, ..., r_n] = \mathsf{argpass}(f/n)\\ \forall i{=}1..n\ c_i = r_i[\![a_i]\!]\\ c_f = \mathsf{c_id}(f/n) \quad \langle\theta_0\rangle\ \mathsf{emit_s}(c_f\texttt{(}c_1\texttt{,}\ ...\texttt{,}\ c_n\texttt{)},\ \eta,\ \delta) \Rightarrow \langle\theta\rangle\end{array}}{\langle\theta_0\rangle\ \mathsf{gcomp}(f(a_1,\ ...,\ a_n),\ \eta,\ \delta) \Rightarrow \langle\theta\rangle}$$

$$\text{(EMIT-S)}\ \frac{\langle\theta_0\rangle\ \mathsf{emit}(\texttt{if (}expr\texttt{) goto}\ \eta(\mathsf{s})\texttt{; else goto}\ \eta(\mathsf{f})\texttt{;}\ ,\ \delta) \Rightarrow \langle\theta\rangle}{\langle\theta_0\rangle\ \mathsf{emit_s}(expr,\ \eta,\ \delta) \Rightarrow \langle\theta\rangle}$$

$$\text{(CALL-D)}\ \frac{\begin{array}{l}\mathsf{det}(f/n)\\ [r_1, ..., r_n] = \mathsf{argpass}(f/n)\\ \forall i{=}1..n\ c_i = r_i[\![a_i]\!]\\ c_f = \mathsf{c_id}(f/n) \quad \langle\theta_0\rangle\ \mathsf{emit_d}(c_f\texttt{(}c_1\texttt{,}\ ...\texttt{,}\ c_n\texttt{)},\ \eta,\ \delta) \Rightarrow \langle\theta\rangle\end{array}}{\langle\theta_0\rangle\ \mathsf{gcomp}(f(a_1,\ ...,\ a_n),\ \eta,\ \delta) \Rightarrow \langle\theta\rangle}$$

$$\text{(EMIT-D)}\ \frac{\begin{array}{l}\langle\theta_0\rangle\ \mathsf{emit}(stat,\ \delta) \Rightarrow \langle\theta_1\rangle\\ \langle\theta_1\rangle\ \mathsf{emit}(\texttt{goto}\ \eta(\mathsf{s}),\ \delta) \Rightarrow \langle\theta\rangle\end{array}}{\langle\theta_0\rangle\ \mathsf{emit_d}(stat,\ \eta,\ \delta) \Rightarrow \langle\theta\rangle}$$

Fig. 6. Compilation of calls.

$$\text{(UNIFY-FG)}\ \frac{\begin{array}{l}\mathsf{var}(a) \quad \mathsf{var}(b) \quad \beta \vdash \mathsf{fresh}(a) \quad \beta \vdash \mathsf{ground}(b)\\ c_a = \mathsf{val_lval}[\![a]\!] \quad c_b = \mathsf{val_rval}[\![b]\!]\\ \langle\theta_0\rangle\ \mathsf{emit_d}(c_a\texttt{=}c_b,\ \eta,\ \delta) \Rightarrow \langle\theta\rangle\end{array}}{\langle\theta_0\rangle\ \mathsf{gcomp}(\lfloor\beta\rfloor a = b,\ \eta,\ \delta) \Rightarrow \langle\theta\rangle}$$

$$\text{(UNIFY-GG)}\ \frac{\begin{array}{l}\mathsf{var}(a) \quad \mathsf{var}(b) \quad \beta \vdash \mathsf{ground}(a) \quad \beta \vdash \mathsf{ground}(b)\\ c_a = \mathsf{val_rval}[\![a]\!] \quad c_b = \mathsf{val_rval}[\![b]\!]\\ \langle\theta_0\rangle\ \mathsf{emit_s}(c_a\texttt{==}c_b,\ \eta,\ \delta) \Rightarrow \langle\theta\rangle\end{array}}{\langle\theta_0\rangle\ \mathsf{gcomp}(\lfloor\beta\rfloor a = b,\ \eta,\ \delta) \Rightarrow \langle\theta\rangle}$$

$$\text{(INSTANCE-FC)}\ \frac{\begin{array}{l}\mathsf{var}(a) \quad \mathsf{cons}(b) \quad \beta \vdash \mathsf{fresh}(a)\\ c_a = \mathsf{val_lval}[\![a]\!] \quad c_b = \mathsf{encodecons}(b,\ \mathsf{encodingtype}(a))\\ \langle\theta_0\rangle\ \mathsf{emit_d}(c_a\texttt{=}c_b,\ \eta,\ \delta) \Rightarrow \langle\theta\rangle\end{array}}{\langle\theta_0\rangle\ \mathsf{gcomp}(\lfloor\beta\rfloor a = b,\ \eta,\ \delta) \Rightarrow \langle\theta\rangle}$$

Fig. 7. Unification compilation rules.

$\mathsf{argmode}(f/n,\ i)$ (see Section 3.3.2) , r_i is val_rval for in or ref_rval for out. Using the translation in Table 1, the C expression for each variable is given as $r_i[\![a_i]\!]$. Taking the C identifier assigned to predicate ($\mathsf{c_id}(f/n)$), we have all the pieces to perform the call. If the predicate is semi-deterministic (i.e., it either fails or gives a single solution), the (CALL-S) rule emits code that checks the return value and jumps to the success or failure continuation. If the predicate is deterministic, the (CALL-D) rule emits code that continues at the success continuation. To reuse those code generation patterns, rules (EMIT-S) and (EMIT-D) are defined.

3.4.3 Compilation of built-in calls

When compiling goal calls, we distinguish the special case of built-ins, which are natively understood by the imProlog compiler and which treats them especially. The unification $a = b$ is handled as shown in Figure 7. If a is a fresh variable and b is ground (resp. for the symmetrical case), the (UNIFY-FG) rule specifies a translation that generates an assignment statement that copies the value stored in b into a (using the translation for their *r-value* and *l-value*, respectively). When a and

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

$$(\textsc{InitMut})\ \frac{\begin{array}{c}\mathsf{var}(a) \quad \beta \vdash \mathsf{fresh}(a) \quad \mathsf{refmode}(a) = \mathbf{0m} \\ \langle\theta_0\rangle\ \mathsf{gcomp}(a \Leftarrow b,\, \eta,\, \delta) \Rightarrow \langle\theta\rangle\end{array}}{\langle\theta_0\rangle\ \mathsf{gcomp}(a = \mathsf{initmut}(\tau,\, b),\, \eta,\, \delta) \Rightarrow \langle\theta\rangle}$$

$$(\textsc{AssignMut})\ \frac{\begin{array}{l}\mathsf{var}(a) \quad \beta \vdash \mathsf{ground}(a) \quad \beta \vdash a{:}\mathsf{mut}(_) \\ \mathsf{var}(b) \quad \beta \vdash \mathsf{ground}(b) \\ c_a = \mathsf{mutval/val_lval}[\![a]\!] \\ c_b = \mathsf{val_rval}[\![b]\!] \\ \langle\theta_0\rangle\ \mathsf{emit_d}(c_a\mathtt{=}c_b,\, \eta,\, \delta) \Rightarrow \langle\theta\rangle\end{array}}{\langle\theta_0\rangle\ \mathsf{gcomp}(a \Leftarrow b,\, \eta,\, \delta) \Rightarrow \langle\theta\rangle}$$

$$(\textsc{ReadMut})\ \frac{\begin{array}{l}\mathsf{var}(a) \quad \beta \vdash \mathsf{ground}(a) \quad \beta \vdash a{:}\mathsf{mut}(_) \quad \beta \vdash \mathsf{ground}(a@) \\ \mathsf{var}(b) \quad \beta \vdash \mathsf{fresh}(b) \\ c_a = \mathsf{mutval/val_rval}[\![a]\!] \\ c_b = \mathsf{val_lval}[\![b]\!] \\ \langle\theta_0\rangle\ \mathsf{emit_d}(c_b\mathtt{=}c_a,\, \eta,\, \delta) \Rightarrow \langle\theta\rangle\end{array}}{\langle\theta_0\rangle\ \mathsf{gcomp}(b = a@,\, \eta,\, \delta) \Rightarrow \langle\theta\rangle}$$

Fig. 8. Compilation rules for mutable operations.

b are both ground, the built-in is translated into a comparison of their values (rule (Unify-GG)). When a is a variable and b is a constant, the built-in is translated into an assignment statement that copies the C value encoded from b, using the encoding type required by a, into a (rule (Instance-FC)). Note that although full unification may be assumed during program transformations and analysis, it must be ultimately reduced to one of the cases above. Limiting to the simpler cases is expected, in order to avoid bootstrapping problems when defining the full unification in imProlog as part of the emulator definition.

The compilation rules for operations on mutable variables are defined in Figure 8. The initialization of a mutable $a = \mathsf{initmut}(\tau, b)$ (rule (InitMut)) is compiled as a mutable assignment, but limited to the case where the reference mode of a is $\mathbf{0m}$ (that is, it has been inferred that it will be a local mutable variable). The built-in $a \Leftarrow b$ is translated into an assignment statement (rule (AssignMut)), that copies the value of b as the mutable value of a. The (ReadMut) rule defines the translation of $b = a@$, an assignment statement that copies the value stored in the mutable value of a into b, which must be a fresh variable. Note that the case $x = a@$ where x is not fresh can be reduced to $(t = a@,\ x = t)$, with t a new variable, for which compilation rules exist.

3.4.4 Compilation of predicates

The rules in the previous sections defined how goals are compiled. In this section, we will use those rules to compile predicates as C functions. Figure 9 provides rules that distinguish between deterministic and semi-deterministic predicates. For a predicate with $name = f/n$, the lookup($name$) function returns its arguments and body. The information from analysis of encoding types and reference modes (Section 3.3) is used by argdecls and vardecls to obtain the list of argument and variable declarations for the program. On the other hand, bb_code is a predefined operation that flattens the basic blocks in its second argument θ as a C block composed of labels and statements. Finally, the emitdecl operation is responsible

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

$$\text{(Pred-D)}\quad \frac{\begin{array}{l}\mathsf{det}(name)\quad ([a_1, \ldots, a_n], body) = \mathsf{lookup}(name)\\ \theta_0 = \mathsf{bb_empty}\\ \langle\theta_3\rangle\ \mathsf{bb_newn}(2) \Rightarrow [\delta, \delta_s]\ \langle\theta_4\rangle\\ \langle\theta_4\rangle\ \mathsf{gcomp}(body, [\mathsf{s}{\mapsto}\delta_s], \delta) \Rightarrow \langle\theta_5\rangle\\ \langle\theta_5\rangle\ \mathsf{emit}(\texttt{return}, \delta_s) \Rightarrow \langle\theta\rangle\\ c_f = \mathsf{c_id}(name)\\ argdecls = \mathsf{argdecls}([a_1, \ldots, a_n])\quad vardecls = \mathsf{vardecls}(body)\quad code = \mathsf{bb_code}(\delta, \theta)\\ \langle p_0\rangle\ \mathsf{emitdecl}(\texttt{void}\ c_f\texttt{(}argdecls\texttt{) \{}\ vardecls\texttt{;}\ code\ \texttt{\}}) \Rightarrow \langle p\rangle\end{array}}{\langle p_0\rangle\ \mathsf{pcomp}(name) \Rightarrow \langle p\rangle}$$

$$\text{(Pred-S)}\quad \frac{\begin{array}{l}\mathsf{semidet}(name)\quad ([a_1, \ldots, a_n], body) = \mathsf{lookup}(name)\\ \theta_0 = \mathsf{bb_empty}\\ \langle\theta_3\rangle\ \mathsf{bb_newn}(3) \Rightarrow [\delta, \delta_s, \delta_f]\ \langle\theta_4\rangle\\ \langle\theta_4\rangle\ \mathsf{gcomp}(body, [\mathsf{s}{\mapsto}\delta_s, \mathsf{f}{\mapsto}\delta_f], \delta) \Rightarrow \langle\theta_5\rangle\\ \langle\theta_5\rangle\ \mathsf{emit}(\texttt{return TRUE}, \delta_s) \Rightarrow \langle\theta_6\rangle\\ \langle\theta_6\rangle\ \mathsf{emit}(\texttt{return FALSE}, \delta_f) \Rightarrow \langle\theta\rangle\\ c_f = \mathsf{c_id}(name)\\ argdecls = \mathsf{argdecls}([a_1, \ldots, a_n])\quad vardecls = \mathsf{vardecls}(body)\quad code = \mathsf{bb_code}(\delta, \theta)\\ \langle p_0\rangle\ \mathsf{emitdecl}(\texttt{bool}\ c_f\texttt{(}argdecls\texttt{) \{}\ vardecls\texttt{;}\ code\ \texttt{\}}) \Rightarrow \langle p\rangle\end{array}}{\langle p_0\rangle\ \mathsf{pcomp}(name) \Rightarrow \langle p\rangle}$$

Fig. 9. Predicate compilation rules.

for inserting the function declarations in the compilation output p. Those definitions are used in the (Pred-D) and (Pred-S) rules. The former compiles deterministic predicates by binding a single success to a **return** statement, and emits a C function returning no value. The latter compiles semi-deterministic predicates by binding the continuations to code that returns a *true* or *false* value depending on the success and failure status. Note that this code matches exactly the scheme needed in Section 3.4.2 to perform calls to imProlog predicates compiled as C functions.

3.4.5 A compilation example

In order to clarify how the previous rules generate code, we include here a code snippet (Fig. 10) with several types of variables accessed both from the scope of their first appearance, and from outside that frame. We show also how this code is compiled into two C functions. Note that redundant jumps and labels have been simplified. It is composed of an encoding type definition flag/1, two predicates that are compiled to C functions (p/1 semi-deterministic, swmflag/1 deterministic), and two predicates with annotations to unfold the code during preprocessing (mflag/2 and swflag/2). Note that by unfolding the mflag/2 predicate, a piece of illegal code (passing a reference to a local mutable) becomes legal. Indeed, this kind of predicate unfolding has proved to be a good, manageable replacement for the macros which usually appear in emulators written in lower-level languages and which are often a source of mistakes.

3.4.6 Related compilation schemes

Another compilation scheme which produces similar code is described in Henderson and Somogyi (2002). There are, however, significant differences, of which we will mention just a few. One of them is the source language and the constraints imposed on it. In our case, we aim at writing a WAM emulator in imProlog from which C

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

Source:

```
:- regtype flag/1 + low(int32).
flag := off | on.
:- pred p(+I) :: flag.
p(I) :-
      mflag(I, A),
      A = B,
      swmflag(B),
      A@ = on.
:- pred mflag/2 + unfold.
mflag(I, X) :-
      X = initmut(flag, I).
:- pred swmflag(+I) :: mut(flag).
swmflag(X) :-
      swflag(X@, X2),
      X <= X2.
:- pred swflag/2 + unfold.
swflag(on, off).
swflag(off, on).
```

Output:

```
1  bool p(int32 i) {
2      int32 a;
3      int32 *b;
4      int32 t;
5      b = &a;
6      swmflag(b);
7      t = a;
8      if (t == 1) goto l1; else goto l2;
9  l1: return TRUE;
10 l2: return FALSE;
11 }
12 void smwflag(int32 *x) {
13     int32 t;
14     int32 x2;
15     t = *x;
16     if (t == 1) goto l1; else goto l2;
17 l1: x2 = 0;
18     goto l3;
19 l2: x2 = 1;
20 l3: *x = x2;
21     return;
22 }
```

Fig. 10. imProlog compilation example.

code is generated with the constraint that it has to be identical (or, at least, very close) to a hand-written and hand-optimized emulator, including the implementation of the internal data structures. This has forced us to pay special attention to the compilation and placement of data, use mutable variables, and ignore for now non-deterministic control. Also, in this work we use an intermediate representation based on basic blocks, which makes it easier to interface with internal back-ends for compilers other than GCC, such as LLVM (Lattner and Adve 2004) (which enables JIT compilation from the same representation).

4 Extensions for emulator generation in imProlog

The dialect and compilation process that has been described so far is general enough to express the instructions in a typical WAM emulator, given some basic built-ins about operations on data types, memory stacks, and O.S. interface. However, combining those pieces of code together to build an *efficient* emulator requires a compact encoding of the bytecode language, and a bytecode fetching and dispatching loop that usually needs a tight control on low-level data and operations that we have not included in the imProlog language. In Morales *et al.* (2005) we showed that it is possible to automate the generation of the emulator from generic instruction definitions and annotations stating how the bytecode is encoded and decoded. Moreover, this process was found to be highly mechanizable, while making instruction code easier to manage and other optimizations (such as instruction merging) easier to perform. In this section, we show how this approach is integrated in the compilation process, by including the emulator generation as part of it.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

```
:- pred u_cons(+, +) :: mut(tagged) * constagged.
u_cons(A, Cons) :-
      deref(A@, Td),
      ( tagof(Td, ref) → bind_cons(Td, Cons), next_ins
      ; Td = Cons → next_ins
      ; fail_ins
      ).
:- pred deref/2.
deref(T, Td) :-
      ( tagof(T, ref) →
            T1 = ~tagval(T)@,
            (T = T1 → Td = T1 ; deref(T1, Td))
      ; Td = T
      ).
:- pred bind/2.
bind_cons(Var, Cons) :-
      (trail_cond(Var) → trail_push(Var) ; true),
      ~tagval(Var) ⇐ Cons.
```

Fig. 11. Unification with a constant and auxiliary definitions.

4.1 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. As an example, Figure 11 shows imProlog code corresponding to the definition of an instruction which tries to unify a term and a constant. The **pred** declaration states that the first argument is a mutable variable and that the second is a tagged word containing a constant. It includes a sample implementation of the WAM dereference operation, which follows a reference chain and stops when the value pointed to is the same as the pointing term, or when the chain cannot be followed any more. Note the use of the native type tagged/2 and the operations tagof/2 and tagval/2 which access the tag and the associated value of a tagged word, respectively. Also note that the tagval/2 of a tagged word with ref results in a mutable variable, as can be recognized in the code. Other native operations include trail_cond/1, trail_push/1, and operations to manage the emulator stacks. Note the special predicates next_ins and fail_ins. They execute the next instruction or the failure instruction, respectively. The purpose of the next instruction is to continue the emulation of the next bytecode instruction (which can be considered as a recursive call to the emulator itself, but which will be defined as a built-in). The failure instruction must take care of unwinding the stacks at the WAM level and selecting the next bytecode instruction to execute (to implement the failure in the emulator). As a usual instruction, it can be defined by calling built-ins or other imProlog code, and it should finally include a call to next_ins to continue the emulation. Since this instruction is often invoked from other instructions, a special treatment is given to share its code, which will be described later.

The compilation process is able to unfold (if so desired) the definition of the predicates called by u_cons/2 and to propagate information inside the instruction, in

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

order to optimize the resulting piece of the emulator. After the set of transformations that instruction definitions are subject to, and other optimizations on the output (such as transformation of some recursions into loops) the generated C code is of high quality (see, for example, Fig. 14, for the code corresponding to a specialization of this instruction).

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 5) 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, in the case of Quintus Prolog) and many of them are variations (obtained through specialization, merging, etc., normally done manually) on “common blocks.” These common blocks are the simple instructions we aim at representing explicitly in imProlog.

In the experiments we performed (Section 5.3) the emulator with a largest 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). A simple instruction set is easier to maintain and its consistency is easier to ensure. Complex instructions are generated automatically in a (by construction) sound way from this initial “seed.”

4.2 An emulator specification in imProlog

Although imProlog could be powerful enough to describe the emulation loop, as mentioned before we leverage on previous work (Morales *et al.* 2005) in which $\mathcal{L}_c$ emulators were automatically built from definitions of instructions written in $\mathcal{L}_a$ and their corresponding code written in $\mathcal{L}_c$. Bytecode representation, compiler back-end, and an emulator (including the emulator loop) able to understand $\mathcal{L}_b$ code can be automatically generated from those components. In our current setting, definitions for $\mathcal{L}_a$ instructions are written in $\mathcal{L}_s^r$ (recall Fig. 2) and these definitions can be automatically translated into $\mathcal{L}_c$ by the imProlog compiler. We are thus spared of making this compiler more complex than needed. More details on this process will be given in the following section.

4.3 Assembling the emulator

We will describe now the process that takes an imProlog representation of an abstract machine and obtains a full-fledged implementation of this machine. The overall process is sketched in Figure 12, and can be divided into two stages, which we have termed *mgen* and *emucomp*. The emulator definition, $\mathcal{E}$, is a set of predicates and assertions written in imProlog, and *mgen* is basically a normalization process where the source $\mathcal{E}$ is processed to obtain a machine definition $\mathcal{M}$. This definition contains components describing the instruction semantics written in imProlog and a set of *hints* about the bytecode representation (e.g., numbers for the bytecode instructions). $\mathcal{M}$ is then processed by an emulator compiler *emucomp* which generates a bytecode

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

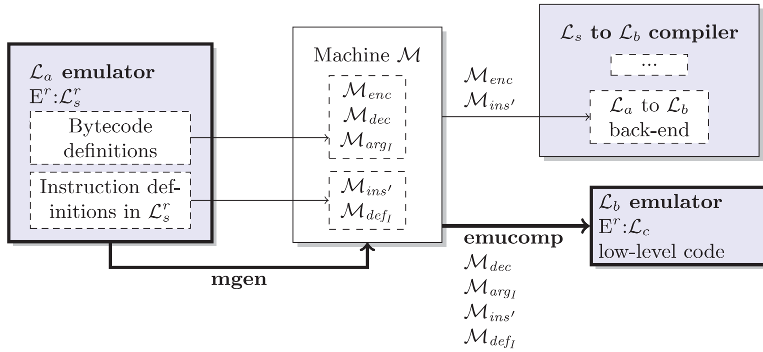

Fig. 12. From imProlog definitions to $\mathcal{L}_b$ emulator in $\mathcal{L}_c$.

emulator for the language $\mathcal{L}_b$, written in the language $\mathcal{L}_c$. The machinery to encode $\mathcal{L}_a$ programs into the bytecode representation $\mathcal{L}_b$ is also given by definitions in $\mathcal{M}$.

Using the terminology in Morales *et al.* (2005), we denote the components of $\mathcal{M}$ as follows:

$$\mathcal{M} = (\mathcal{M}_{enc}, \mathcal{M}_{dec}, \mathcal{M}_{arg_I}, \mathcal{M}_{def_I}, \mathcal{M}_{ins'}).$$

First, the relation between $\mathcal{L}_a$ and $\mathcal{L}_b$ is given by means of several components[12]:

$\underline{\mathcal{M}_{enc}}$ declares how the bytecode encodes $\mathcal{L}_a$ instructions and data: e.g., `X(0)` is encoded as the number `0` for an instruction which needs access to some `X` register.

$\underline{\mathcal{M}_{dec}}$ declares how the bytecode should be decoded to give back 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)`.

The remaining of the components of $\mathcal{M}$ capture the meaning of the (rather low-level) constituents of $\mathcal{L}_a$, providing a description of each instruction. Those components do not make bytecode representation issues explicit, as they have already been specified in $\mathcal{M}_{enc}$ and $\mathcal{M}_{dec}$. In the present work, definitions for $\mathcal{L}_a$ instructions are given in $\mathcal{L}_s^r$, instead of in $\mathcal{L}_c$ as was done in the formalization presented in (Morales *et al.* 2005). The reason for this change is that in Morales *et al.* (2005) the final implementation language ($\mathcal{L}_c$, in which emulators were generated) was also the language in which each basic instruction was assumed to be written. However, in our case, instructions are obviously written in $\mathcal{L}_s^r$ (i.e., imProlog, which is more amenable to automatic program transformations) and it makes more sense to use it directly in the definition of $\mathcal{M}$. Using $\mathcal{L}_s^r$ requires, however, extending and/or modifying the remaining parts of $\mathcal{M}$ with respect to the original definition as follows:

$\underline{\mathcal{M}_{arg_I}}$ which assigns a pair (T, mem) to every expression in $\mathcal{L}_a$, where T is the type of the expression and *mem* is the translation of the expression into $\mathcal{L}_c$. For

[12] The complete description includes all elements for a WAM: `X` and `Y` registers, atoms, numbers, functors, etc.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

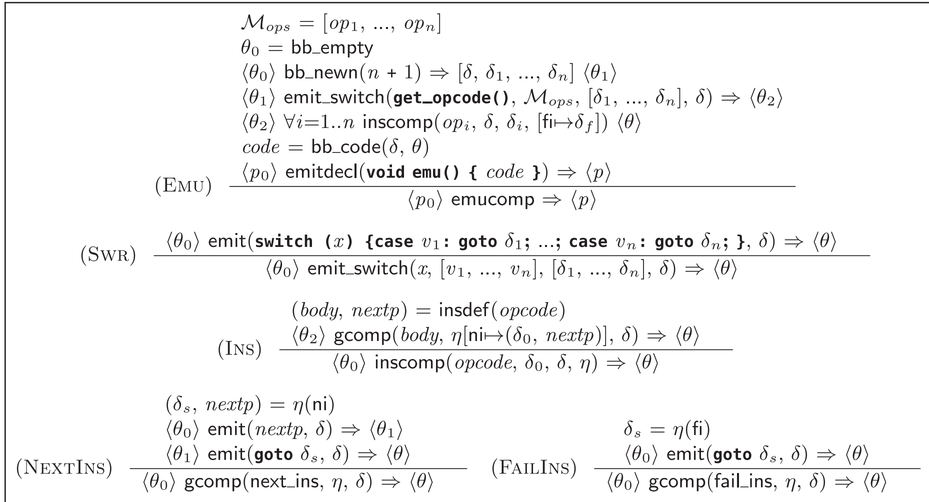

Fig. 13. Emulator compiler.

example, the type of `X(0)` is mut(tagged) and its memory location is `&(x[0])`, assuming `X` registers end up in an array[13].

$\mathcal{M}_{def_I}$ which contains the definition of each instruction in $\mathcal{L}_s^r$.

$\mathcal{M}_{ins'}$ 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: e.g., `X` registers, `Y` registers, integers, atoms, functors.

The rest of the components and $\mathcal{M}_{ins'}$ are used by the emulator compiler to generate an $\mathcal{L}_b$ emulator written in $\mathcal{L}_c$. A summarized definition of the emulator compiler and how it uses the different pieces in $\mathcal{M}$ can be found in Figure 13. The (Emu) rule defines a function that contains the emulator loop. It is similar to the (Pred-D) rule already presented, but takes parts of the source code from $\mathcal{M}$. It generates a list of basic block identifiers for each instruction, and a basic block identifier for the emulator loop entry. The (Swr) rule is used to insert a *switch* statement that implements the opcode fetching, and jumps to the code of each instruction. The (Ins) rule is used to generate the code for each instruction. To implement the built-ins next_ins and fail_ins, two special continuations ni and fi are stored in the continuation mapping. The continuation to the failure instruction is bound to the δ_f basic block identifier (assuming that the op_f opcode is that of the failure instruction). The (FailIns) rule includes a special case in gcomp that implements this call. The continuation to the next instruction is a pair of the basic block that begins the emulator switch, and a piece of C code that moves the bytecode pointer to the next instruction (that is particular to each instruction, and is returned by insdef alongside with its code). The (NextIns) rule emits code that executes that code and jumps to opcode fetching.

[13] This definition has been expanded with respect to its original $\mathcal{M}_{arg}$ definition in order to include the imProlog type in addition to the memory location.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

The abstract machine component $\mathcal{M}_{ins'}$ is used to obtain the *name* and data *format* of the instruction identified by a given *opcode*. From the format and $\mathcal{M}_{arg_I}$ definition, a type, an encoding type, and a custom *r-value* for each instruction argument are filled. In this way, the compilation process can transparently work with variables whose value is defined from the operand information in a bytecode stream (e.g., an integer, a machine register, etc.).

Relation with other compilation schemes. The scheme of the generated emulator code is somewhat similar to what the Janus compilation scheme (Gudeman *et al.* 1992) produces for general programs. In Janus, addresses for continuations are either known statically (e.g., for calls, and therefore a direct, static jump to a label can be performed) or are popped from the stack when returning. Since labels cannot be directly assigned to variables in standard C, an implementation workaround is made by assigning a number to each possible return address (and it is this number which is pushed onto / popped from the stack) and using a *switch* to relate these numbers with the code executing them. In our case, we have a similar *switch*, but it relates each opcode with its corresponding instruction code, and it is executed every time a new instruction is dispatched.

We want to note that we deliberately stay within standard C in this presentation: taking advantage of C extensions, such as storing labels in variables, which are provided by `gcc` and used, for example, in (Codognet and Diaz 1995) and (Henderson *et al.* 1995), is out of the scope of this paper. These optimizations are not difficult to add as code generation options, and therefore they should not be part of a basic scheme. Besides, that would make it difficult to use compilers other than `gcc`.

Example 4.1

As an example, from the instruction in Figure 11, which unifies a term living in some variable with a constant, we can derive a specialized version in which the term is assumed to live in an `X` register. The declaration

:– **ins_alias**(ux_cons, u_cons(xreg_mutable, constagged)).

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**(ux_cons).

indicates that the emulator has an entry for that instruction[14]. Figure 14 shows the code generated for the instruction (right) and a fragment of the emulator generated by the emulator compiler in Figure 13.

[14] We optionally allow a pre-assignment of an opcode number to each instruction entry. Different assignments of instruction numbers to opcodes can impact the final performance, as they dictate how the code is laid out in the emulator switch which affects, for example, the behavior of the cache.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

```
1  loop:
2    switch(Op(short,P,0)) {
3    ...
4    case 97: goto ux_cons;
5    ...
6    }
7    ...
8  ux_cons:
9    tagged t;
10   t = X(Op(short,P,2));
11   deref(&t);
12   if (tagged_tag(t) != REF)
13     goto ux_cons__0;
14   bind_cons(t, Op(tagged,P,4));
15   goto ux_cons__1;
16 ux_cons__0:
17   if (t != Op(tagged,P,4))
18     goto fail_ins;
19 ux_cons__1:
20   P = Skip(P,8);
21   goto loop;
22   ...
```

```
1  void deref(tagged_t *a0) {
2    tagged_t t0;
3  deref:
4    if (tagged_tag(*a0) == REF)
5      goto deref__0;
6      else goto deref__1;
7  deref__0:
8    t0 = *(tagged_val(*a0));
9    if ((*a0) != t0)
10     goto deref__2;
11     else goto deref__1;
12 deref__2:
13   *a0 = t0;
14   goto deref;
15 deref__1:
16   return;
17 }
```

Fig. 14. Code generated for a simple instruction.

5 Automatic generation of abstract machine variations

Using the techniques described in the previous section, we now address how abstract machine variations can be generated automatically. Substantial work has been devoted to abstract machine generation strategies such as, e.g., Demoen and Nguyen (2000), Nässén *et al.* (2001) and Zhou (2007), which explore different design variations with the objective of putting together highly optimized emulators. However, as shown previously, by making the semantics of the abstract machine instructions explicit in a language like imProlog, which is easily amenable to automatic processing, such variations can be formulated in a straightforward way mostly as automatic transformations. Adding new transformation rules and testing them together with the existing ones becomes then a relatively easy task.

We will briefly describe some of these transformations, which will be experimentally evaluated in Section 5.3. Each transformation is identified by a two-letter code. We make a distinction between transformations which change the instruction set (by creating new instructions) and those which only affect the way code is generated.

5.1 Instruction set transformations

Let us define an instruction set transformation as a pair (*ptrans*, *etrans*), so that *ptrans* transforms programs from two symbolic bytecode languages $\mathcal{L}_a$ and (a possibly different) $\mathcal{L}_a$'[15] and *etrans* transforms the abstract machine definition within $\mathcal{L}_s^r$. Figure 15 depicts the relation between emulator generation, program compilation, program execution, and instruction set transformations. The full emulator generation includes *etrans* as preprocessing before *mgen* is performed. The resulting emulator

[15] Those languages can be different, for example, if the transformation adds or removes some instructions.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

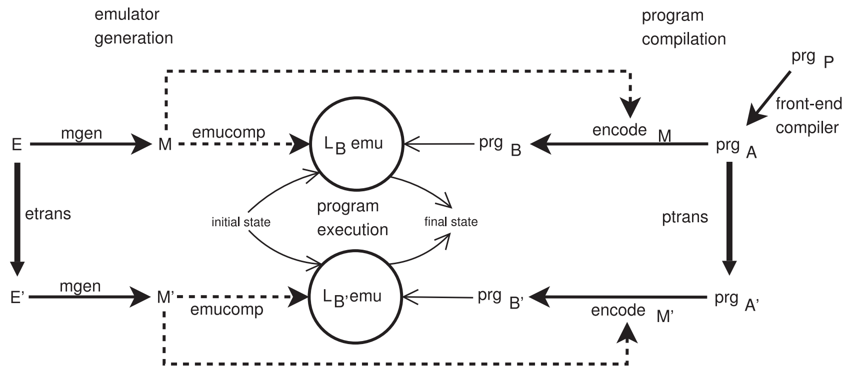

Fig. 15. Application of an instruction set transformation (*ptrans*, *etrans*).

is able to interpret transformed programs after *ptrans* is applied (before bytecode encoding), that is, the new compiler is obtained by including *ptrans* as a new compilation phase.

Note that both *ptrans* and *etrans* are working at the symbolic bytecode level. It is easier to work with a symbolic $\mathcal{L}_a$ program than with the stream of bytes that represents $\mathcal{L}_b$ code, and it is easier to transform instructions written in $\mathcal{L}_s^r$ and specified at the $\mathcal{L}_a$ level, than those already written in $\mathcal{L}_c$ code (where references to $\mathcal{L}_b$ code and implementation details obscure the actual semantics). Reasoning about the correctness of the global transformation that affects the $\mathcal{L}_a$ program and the instruction code is also easier in the $\mathcal{L}_s^r$ specification of the emulator instructions than in a low-level $\mathcal{L}_c$ emulator (assuming the correctness of the emulator generation process).

In the following sections, we will review the instruction set transformations currently available. Although more transformations can of course be applied, the current set is designed with the aim of generating, from simple imProlog definitions, an emulator which is as efficient as a hand-crafted, carefully tuned one.

5.1.1 *Instruction merging [**im**]*

Instruction Merging generates larger instructions from sequences of smaller ones, and is aimed at saving fetch cycles at the expense of a larger instruction set and, therefore, 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 (Nässén *et al.* 2001), but in that work new instructions were generated by hand, although deciding which instructions had to be created was done by means of profiling. In our framework only a single declaration is needed to emit code for a new, merged instruction. Merging is done automatically through code unfolding and based on the definitions of the component instructions. This makes it possible, in principle, to define a set of (experimentally) optimal merging rules. However, finding exactly this set of rules is actually not straightforward.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

Merging rules are specified separately from the instruction code itself, and these rules state how basic instructions have to be combined. To start with, we will need to show how instructions are defined based on their abstract versions. For example, definition

move(A, B) :– $B \Leftarrow A@$.

moves data between two locations, i.e., the contents of the a mutable into the b mutable. In order to specify precisely the source and destination of the data, it is necessary to specify the instruction *format*, with a declaration such as

:– **ins_alias**(movexy, move(xreg_mutable, yreg_mutable)).

which defines a *virtual* instruction named movexy, that corresponds to the instantiation of the code for move/2 for the case in which the first argument corresponds to an X register and the second one corresponds to a Y register. Both registers are seen from imProlog as mutable variables of type *mut(tagged)*. Then, and based on this more concrete instruction, the declaration

:– **ins_entry**(movexy + movexy).

forces the compiler to actually use during compilation an instruction composed of two virtual ones and to emit bytecode containing it (thanks to the *ptrans* transformation in Fig 15, which processes the program instruction sequence to replace occurrences of the collapsed pattern by the new instruction). Emulator code will be generated implementing an instruction which merges two movexy instructions (thanks to the *etrans* transformation). The corresponding code is equivalent to

:– **ins_entry**(movexy_movexy).
:– **pred** movexy_movexy(xreg_mutable, yreg_mutable,
xreg_mutable, yreg_mutable).
movexy_movexy(A, B, C, D) :– $B \Leftarrow A@$, $D \Leftarrow C@$.

This can later be subject to other transformations and used to generate emulator code as any other imProlog instruction.

*5.1.2 Single-instruction encoding of sequences of the same instruction [**ie**]*

In some cases, a series of similar instructions (e.g., unify_with_void) with different arguments can be collapsed into a single instruction with a series of operands which correspond to the arguments of each of the initial instructions. For example, a bytecode sequence such as

unify_with_void(x(1)), unify_with_void(x(2)), unify_with_void(x(5))

can be compiled into

unify_with_void_n([x(1), x(2), x(5)])

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

which would perform exactly as in the initial instruction series, but taking less space and needing fewer fetch cycles. Such an instruction can be created, emitted, and the corresponding emulator code generated automatically based on the definition of unify_with_void.

In order to express this *composite* instruction within imProlog using a single predicate, unify_with_void_n needs to receive a fixed number of arguments. A different predicate for each of the possible lengths of the array would have to be generated otherwise. A single argument actually suffices; hence the square brackets, which are meant to denote an array.

The imProlog code which corresponds to the newly generated instruction is, conceptually, as follows:

```
unify_with_void_n(Array) :-
    array_len(Array, L),
    unify_with_void_n_2(0, L, Array).
unify_with_void_n_2(I, L, Array) :-
    ( I = L → true
    ; elem(I, Array, E),
      unify_with_void(E),
      I1 is I + 1,
      unify_with_void_n_2(I1, L, Array)
    ).
```

It should be clear here why a fixed number of arguments is needed: a series of unify_with_void_n/1, unify_with_void_n/2, etc., would have to be generated otherwise. Note that the loop code ultimately calls unify_with_void/1, the $\mathcal{L}_s^r$ reflection of the initial instruction.

In this particular case, the compiler to $\mathcal{L}_c$ performs some optimizations not captured in the previous code. For example, instead of traversing explicitly the array with an index, this array is expanded and inlined in the bytecode and the program counter is used to retrieve the indexes of the registers by incrementing it after every access. As the length of the array is known when the bytecode is generated, it is actually explicitly encoded in the final bytecode. Therefore, all of the newly introduced operations (array_len/2, elem/3, etc.) need constant time and are compiled efficiently.

5.1.3 *Instructions for special built-Ins [**ib**]*

As mentioned before, calling external library code or internal predicates (classically termed "built-ins") requires following a protocol, to pass the arguments, to check for failure, etc. Although the protocol can be the same as for normal predicates (e.g., passing arguments as X registers), some built-ins require a special (more efficient) protocol (e.g., passing arguments as $\mathcal{L}_c$ arguments, avoiding movements in X registers). Calling those special built-ins is, by default, taken care of by a generic

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

family of instructions, one per arity. This is represented as per the instructions given below

```
:- ins_alias(bltin1d, bltin1(bltindet(tagged), xreg)).
bltin1(BltName, A) :- BltName(A@).
:- ins_alias(bltin2d, bltin2(bltindet(tagged, tagged), xreg, xreg)).
bltin2(BltName, A, B) :- BltName(A@, B@).
```

where each bltinI/i + 1 acts as a bridge to call the external code expecting *i* parameters. The *BltName* argument represents a predicate abstraction that will contain a reference to the actual code of the built-in during the execution. The type of the *BltName* argument reflects the accepted calling pattern of the predicate abstraction. When compiling those instructions to $\mathcal{L}_c$ code, that predicate abstraction is efficiently translated as an unboxed pointer to an $\mathcal{L}_c$ procedure[16]. With the definition shown above, the imProlog compiler can generate, for different arities, an instruction which calls the built-in passed as first argument.

However, by specifying at compile time a predefined set of built-ins or predicates written in $\mathcal{L}_c$ (that is, a static value for *BltName* instead of a dynamic value), the corresponding instructions can be statically specialized and an instruction set which performs direct calls to the corresponding built-ins can be generated. This saves an operand, generating slightly smaller code, and replaces an indirection by a direct call, which saves memory accesses and helps the processor pipeline, producing faster code.

5.2 Transformations of instruction code

Some transformations do not create new instructions; they perform instead a number of optimizations on already existing instructions by manipulating the code or by applying selectively alternative translation schemes.

5.2.1 Unfolding rules [**ur**]

Simple predicates can be unfolded before compilation. In the case of instruction merging, unfolding is used to merge two (or more) instructions into a single piece of code, in order to avoid fetch cycles (Section 5.1.1). However, uncontrolled unfolding is not always an advantage, because an increased emulator size can affect negatively the cache behavior. Therefore, the ***ur*** option turns on or off a predefined set of rules to control which instruction mergings are actually performed. Unfolding rules follow the scheme

$$:\!-\ \textbf{ins_entry}(Ins_1 + Ins_2 + \ldots + Ins_n,\ WhatToUnfold).$$

[16] In the bytecode, the argument that corresponds to the predicate abstraction is stored as a number that uniquely identifies the built-in. When the bytecode is actually loaded, this number is used to look up the actual address of the built-in in a table maintained at runtime. This is needed since, in general, there is no way to know which address will be assigned to the entry point of a given built-in in different program executions.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

where Ins_1 to Ins_n are the basic instructions to be merged, and *WhatToUnfold* is a rule specifying exactly which instruction(s) has to be unfolded when ***ur*** is activated. As a concrete example, the unfolding rule:

:– **ins_entry**(alloc + movexy + movexy, 1).

means that in the instruction to be generated by combining one alloc and two movexy, the code for alloc is inlined (the value of the last argument 1 refers to the *first* instruction in the sequence), and the (shared) code for movexy + movexy is invoked afterwards. A companion instruction merging rule for movexy + movexy exists

:– **ins_entry**(movexy + movexy, all).

which states that the code for both movexy has to be unfolded in a combined instruction. The instruction alloc + movexy + movexy would generate code for alloc plus a call to movexy + movexy. The compiler eventually replaces this call by an explicit jump to the location of movexy + movexy in the emulator. The program counter is updated accordingly to access the arguments correctly.

*5.2.2 Alternative tag switching schemes [**ts**]*

Tags are used to determine dynamically the type of basic data (atoms, structures, numbers, variables, etc.) contained in a (tagged) memory word. Many instructions and built-ins (like unification) take different actions depending on the type (or tag) of the input arguments. This is called *tag switching*, and it is a heavily-used operation which is therefore worth optimizing as much as possible. The tag is identified (and the corresponding action taken) using tag switching such as

$$(tagtest_1 \rightarrow tagcode_1 \ ; \ \ldots \ ; \ tagtest_n \rightarrow tagcode_n)$$

where every $tagtest_i$ has the form tagof(v, tag_i) (i.e., code that performs a different action depending on the tag value of a tagged v). The ***ts*** option chooses between either a switch control structure (when enabled) or a set of predefined test patterns based on tag encodings and assertions on the possible tags (when disabled).

Both possibilities are studied in more detail in Morales *et al.* (2008). Since the numbers that encode the tags are usually small, it is easy for a modern C compiler (e.g., gcc) to generate an indirection table and jump to the right code using it (that is, it does not require a linear search). It is difficult, however, to make the C compiler aware that checks to ensure that the tag number will actually be one of the cases in the *switch* are, by construction, unnecessary (i.e., there is no need for a *default* case). This information could be propagated to the compiler with a type system which not all low-level languages have. The alternative compilation scheme (rule (TIF)) makes explicit use of tag-checking primitives, where the sequence of $ctest_i$ and the code of each branch depends on the particular case.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

The latter approach is somewhat longer (and more complex as the number of allowed tags grows) than the former. However, in some cases there are several advantages to the latter, besides the already mentioned avoidance of boundary checks are as follows:

- Tags with higher runtime probability can be checked before, in order to select the right branch as soon as possible.
- Since the evaluation order is completely defined, tests can be specialized to determine as fast as possible which alternative holds. For example, if by initial assumption v can only be either a heap variable, a stack variable, or a structure (having a different tag for each case), then the tests can check if it is a heap variable or a stack variable and assume that it is a structure in the last branch.

Deciding on the best option has to be based on experimentation, the results of which we summarize in Section 5.3.3 and in Tables 4 and 5.

*5.2.3 Connected continuations [**cc**]*

Some actions can be repeated unnecessarily because they appear at the end of an operation and at the beginning of the next one. Often they have no effect the second time they are called (because they are, e.g., tests which do not change the tested data, or data movements). In the case of tests, for example, they are bound to fail or succeed depending on what happened in the previous invocation.

As an example, in the fragment deref(T), (tagof(T, ref) $\rightarrow$ A ; B) the test tagof(R, ref) is performed just before exiting deref/1 (see Fig. 11). Code generation for instructions which include similar patterns is able to insert a jump either to A or B from the code generated for deref/1. This option enables or disables this optimization for a series of preselected cases, by means of code annotations similar to the ones already shown.

*5.2.4 Read/write mode specialization [**rw**]*

WAM-based implementations use a flag to test whether heap structures are being read (matched against) or written (created). According to the value of this flag, which is set by code executed immediately before, several instructions adapt their behavior with an internal, local *if-then-else*.

A common optimization is to partially evaluate the *switch* statement which implements the fetch-and-execute cycle inside the emulator loop. Two different switches can be generated, with the same structure, but with heap-related instructions specialized to perform either reads or writes (Carlsson 1991). Enabling or disabling the ***rw*** optimization makes it possible to generate instruction sets (and emulators) where this transformation has been turned on or off.

This is conceptually performed by generating different versions of the code for the instructions, depending on the value of a mutable variable mode, which can only take the values read or write. Deciding whether to generate different code versions or to generate *if-then-else*s to be checked at run-time is done based on a series of heuristics which try to forecast the complexity and size of the resulting code.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

5.3 Experimental evaluation

We present in this section experimental data regarding the performance achieved on a set of benchmarks by a collection of emulators, all of which were automatically generated by selecting different combinations of the options presented in previous sections. In particular, by using all **compatible** possibilities for the transformation and generation options given in Section 5 we generated 96 different emulators (instead of $2^7 = 128$, as not all options are independent; for example, **ie** needs **im** to be performed). This bears a close relationship with (Demoen and Nguyen 2000), but here we are not changing the internal data structure representation (and of course our instructions are all initially coded in imProlog). It is also related to the experiment reported in (Nässén *et al.* 2001), but the tests we perform are more extensive and cover more variations on the kind of changes that the abstract machine is subject to. Also, (Nässén *et al.* 2001) starts off by being selective about the instructions to merge, which may seem a good idea but, given the very intricate dependencies among different optimizations, can also result in a loss of optimization opportunities. In any case, this is certainly a point we want to address in the future by using instruction-level profiling.

Although most of the benchmarks we used are relatively well known, a brief description as follows:

boyer	Simplified Boyer-Moore theorem prover kernel.
crypt	Cryptoarithmetic puzzle involving multiplication.
deriv	Symbolic derivation of polynomials.
factorial	Compute the factorial of a number.
fib	Simply recursive computation of the nth Fibonacci number.
knights	Chess knight tour, visiting only once every board cell.
nreverse	Naive reversal of a list using append.
poly	Raises symbolically the expression 1 + x + y + z to the nth power.
primes	Sieve of Eratosthenes.
qsort	Implementation of QuickSort.
queens11	N-Queens with $N = 11$.
query	Makes a natural language query to a knowledge database with information about country names, population, and area.
tak	Computation of the Takeuchi function.

Our starting point was a "bare" instruction set comprising the common basic blocks of a relatively efficient abstract machine (the "optimized" abstract machine of Ciao 1.13, in the 'optim_comp' directory of the Ciao 1.13 repository)[17]. The Ciao abstract machines have their remote roots in the emulator of SICStus Prolog 0.5/0.7 (1986–89), but have evolved over the years quite independently and been the object of many optimizations and code rewrites resulting in performance improvements and much added functionality[18]. The performance of this, our *baseline* engine matches

[17] Changes in the optimized version include tweaks to clause jumping, arithmetic operations, and built-ins and some code clean-ups that reduce the size of the emulator loop.

[18] This includes modules, attributed variables, support for higher order, multiprocessing, parallelism, tabling, modern memory management, etc.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

Table 2. *Speed comparison of baseline with other Prolog systems*

Benchmark	Yap 5.1.2	hProlog 2.7	SWI 5.6.55	Ciao-std 1.13	Ciao-opt (baseline)
boyer	1,392	1,532	11,169	2,560	1,604
crypt	3,208	2,108	36,159	6,308	3,460
deriv	3,924	3,824	12,610	6,676	3,860
exp	1,308	1,740	2,599	1,400	1,624
factorial	4,928	2,368	16,979	3,404	2,736
fft	1,020	1,652	14,351	2,236	1,548
fib	2,424	1,180	8,159	1,416	1,332
knights	2,116	1,968	11,980	3,432	2,352
nreverse	1,820	908	18,950	3,900	2,216
poly	1,328	1,104	6,850	1,896	1,160
primes	4,060	2,004	28,050	3,936	2,520
qsort	1,604	1,528	8,810	2,600	1,704
queens11	1,408	1,308	24,669	3,200	1,676
query	632	676	6,180	1,448	968
tak	3,068	1,816	27,500	5,124	2,964

that of modern Prolog implementations. Table 2 helps evaluating the speed of this baseline optimized Ciao emulator w.r.t. to the relatively unoptimized Ciao 1.13 emulator compiled by default in the Ciao distribution, some high-performance Prolog implementations (Yap 5.1.2 (Santos-Costa *et al.* 2011) and hProlog 2.7 (Demoen 2012)), and the popular SWI-Prolog system (version 5.6.55 (Wielemaker 2010)).

Figures 16 (in page 39) to 37 (in page 48) summarize graphically the results of the experiments, as the data gathered — 96 emulators $\times$ 13 benchmarks = 1,248 performance figures — is too large to be comfortably presented in regular tables.

Each figure presents the speedup obtained by different emulators for a given benchmark (or all benchmarks in the case of the summary tables). Such speedups are relative to some "default" code generation options, which we have set to be those which were active in the Ciao emulator we started with (our baseline), and which therefore receive speedup 1.0. Every point in each graph corresponds to the relative speed of a different emulator obtained with a different combination of the options presented in Sections 5.1 and 5.2.

The options are related to the points in the graphs as follows: each option is assigned a bit in a binary number, where "1" means activating the option and "0" means deactivating it. Every value in the y-axis of the figures corresponds to a combination of the three options in Section 5.1. Note that only six combinations (out of the $2^3 = 8$ possible ones) are plotted due to dependencies among options (for example, 'instruction encoding" always implies 'instruction merging"). The options in Section 5.2, which correspond to transformations in the way code is generated and which need four bits, are encoded using $2^4 = 16$ different dot shapes. Every combination of emulator generation options is thus assigned a different 7-bit number encoded as a dot shape and a y-coordinate. The x-coordinate represents the speedup as presented before (i.e., relative to the hand-coded emulator currently in Ciao 1.13).

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

Table 3. *Meaning of the bits in the plots*

Instruction Generation			Instruction Transformations			
Instruction Encoding **(ie)**	Special Builtins **(ib)**	Instruction Merging **(im)**	Tag Switching **(ts)**	Connected Conts. **(cc)**	Unfolding Rules **(ur)**	R/W Mode **(rw)**

The level of aggressiveness of the instruction set transformations used in the paper was selected to automatically generate an emulator identical to the hand-crafted one. We have experimentally confirmed that it is difficult to outperform this engine without changing other parameters, such as the overall architecture.

Different selections for the bits assigned to the *y*-coordinate and to the dot shapes would of course yield different plot configurations. However, our selection seems intuitively appropriate, as it uses two different encodings for two different families of transformations, one which affects the bytecode language itself, and another one which changes the way these bytecode operands are interpreted. Table 3 relates the bits in these two groups, using the same order as in the plots.

Every benchmark was run several times on each emulator to make timings stable. The hosts used were an x86 machine with a Pentium 4 processor running Linux and an iMac with a PowerPC 7450 running Mac OS X. Arithmetic and geometric[19] averages of all benchmarks were calculated and are shown in Figures 16, 17, 32, and 33. Their similarity seems to indicate that there are no "odd" behaviors off the average. Additionally, we are including detailed plots for every benchmark and all the engine generation variants, following the aforementioned codification, first for the x86 architecture (Figs. 19–31) and then for the PowerPC architecture (Figs 34–46), in the same order in both cases. Plots for specially relevant cases are shown first, followed by the rest of the figures sorted following an approximate (subjective) "more sparse" to "less sparse" order.

5.3.1 General analysis

The best speedup among all tried options, averaged across the exercised benchmarks and with respect to the baseline Ciao 1.13 emulator, is 1.05 times for the x86 processor (Table 4, top section, under the column *w.r.t. def.*) and 1.01 times for the PowerPC (Table 5, top section, same column heading). While this is a modest average gain, some benchmarks achieve much better speedups. An alternative interpretation of this result is that by starting with a relatively simple instruction set (coded directly in imProlog) and applying automatically and systematically a set of transformation and code generation options which can be trusted to be correct, we have managed to match (and even exceed) the time performance of an emulator which was hand-coded by very proficient programmers, and in which decisions were thoroughly

[19] The geometric average is known to be less influenced by extremely good or bad cases.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

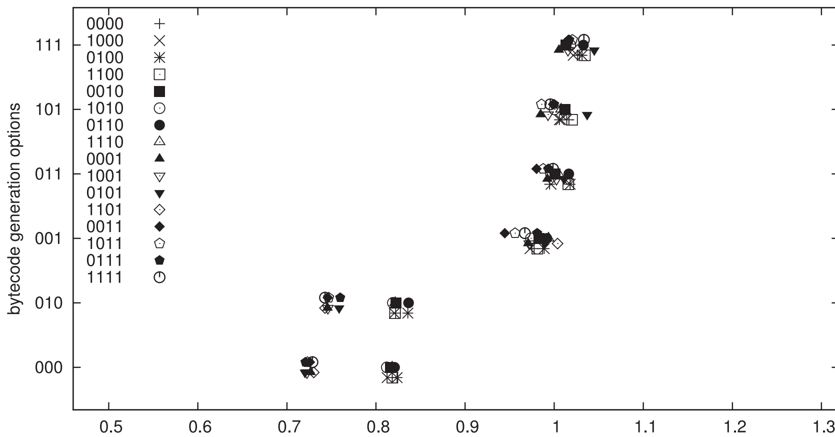

Fig. 16. Geometric average of all benchmarks (with a dot per emulator) — Intel.

tested along several years. Memory usage was unaltered. Note (in the same tables) that the speedup obtained with respect to the basic instruction set (under the column labeled *w.r.t. base*) is significantly higher.

Figure 16 depicts the geometric average of the executions of all benchmarks in an Intel platform. It aims at giving an intuitive feedback of the overall performance of the option sets, and indeed a well-defined clustering around eight centers is clear. Figure 17, which uses the arithmetic average, is very similar (but not identical — it is very slightly biased towards higher speedups), and it shows eight well-defined clusters as well.

From these pictures we can infer that bytecode transformation options can be divided into two different sets: one which is barely affected by options of the generation of code for the emulator (corresponding to the upper four clusters), and another set (the bottom four clusters) in which changes to the generation of the emulator code does have an effect in the performance.

In general, the results obtained in the PowerPC show fewer variations than those obtained in an x86 processor. We attribute this behavior to differences between these two architectures, as they greatly affect the optimization opportunities and the way the C compiler can generate code. For example, the larger number of general-purpose registers available in a PowerPC seems to make the job of the C compiler less dependent on local variations of the code (as the transformations shown in Section 5.2 produce). Additionally, internal differences between both processors (e.g., how branch prediction is performed, whether there is register renaming, shadow registers, etc.) can also contribute to the differences we observed.

As a side note, while Figures 16 and 17 portray an average behavior, there were benchmarks whose performance depiction actually match this average behavior very faithfully — e.g., the simply recursive Factorial (Fig. 19), which is often disregarded as an unrealistic benchmark but which, for this particular experiment, turns out to predict quite well the (geometric) average behavior of all benchmarks. Experiments in the PowerPC (Figs. 32 and 34) generate similar results.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

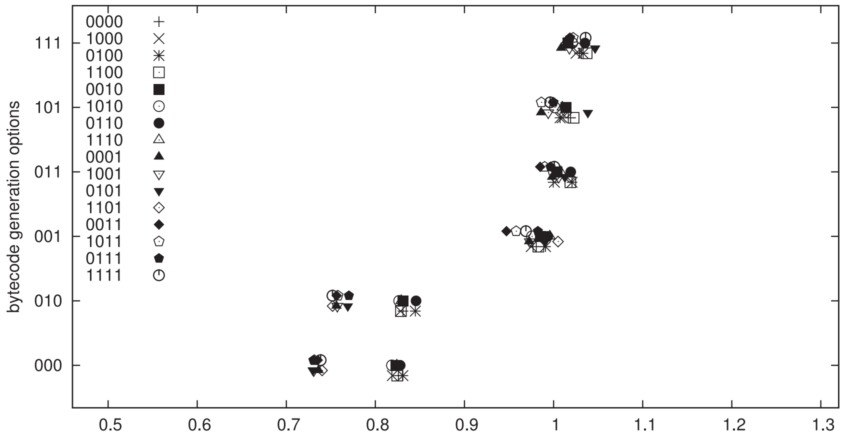

Fig. 17. Arithmetic average of all benchmarks (with a dot per emulator) — Intel.

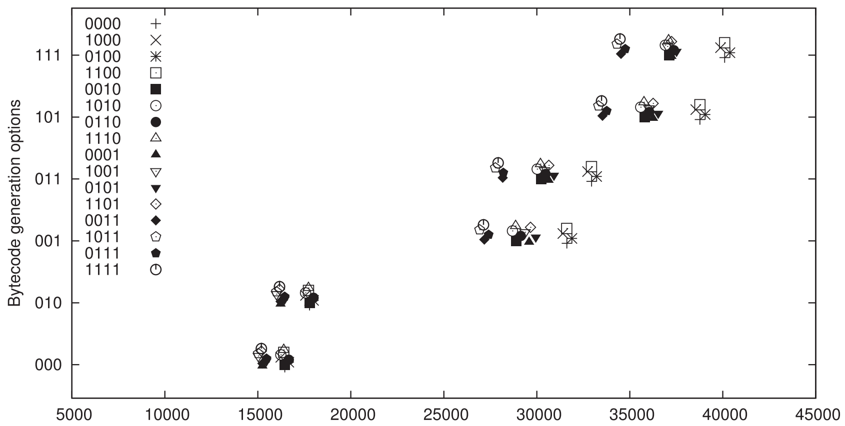

Fig. 18. Size (in bytes) of WAM emulator with respect to the generation options (i86).

Figure 18 presents the size of the WAM loop (using actual i86 object code size measured in bytes) for each bytecode and $\mathcal{L}_c$ code generation option. This size is independent from the benchmarks, and therefore only one plot is shown. It resembles notably the depictions of the speedup graphs. In fact, a detailed inspection of the distribution of low-level code generation options (where each of them corresponds to one of the 16 different dot shapes) inside each bytecode language option shows some correlation among larger and faster emulators. This is not surprising as some code generation schemes which tend to increase the size do so because they generate additional, specialized code.

As in the case for speed, $\mathcal{L}_b$ generation options are the ones which influence most heavily the code size. This is understandable because some options (for example, the ***im*** switch for instruction merging, corresponding to the leftmost bit of the "bytecode generation options") increment notably the size of the emulator loop. On the other hand, code generation options have a less significant effect, as they do not necessarily affect all the instructions.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

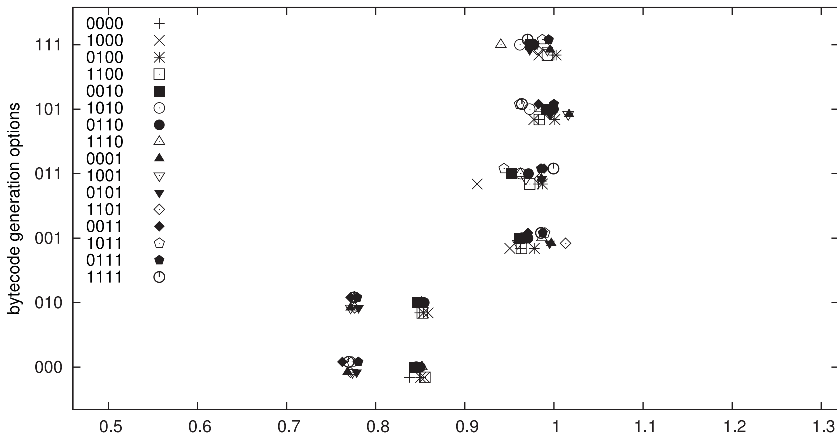

Fig. 19. Factorial involving large numbers — Intel.

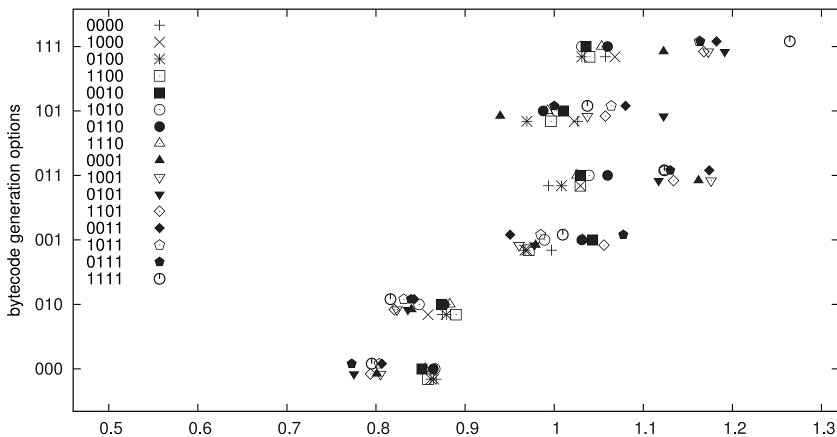

Fig. 20. Queens (with 11 queens to place) — Intel.

It is to be noted that the generation of specialized switches for the write and read modes of the WAM (the ***rw*** option) does not increase the size of the emulator. The reason is that when the ***rw*** flag is checked by all the instructions which need to do so (and many instructions need it), a large number of *if-then-else* constructions with their associated code are generated. In the case of the specialized switches, only an *if-then-else* is needed and the savings from generating less branching code make the emulator smaller.

5.3.2 *A more detailed inspection of selected cases*

Figures 20 (Queens 11) and 21 (Cryptoarithmetic puzzle) show two cases of interest. The former corresponds to results which, while departing from the average behavior, still resemble it in its structure, although there is a combination of options which achieves a speedup (around 1.25) that is significantly higher than average. Figure 21 shows a different landscape where variations on the code generation scheme appear to be as relevant as those on the bytecode itself. Both benchmarks are,

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

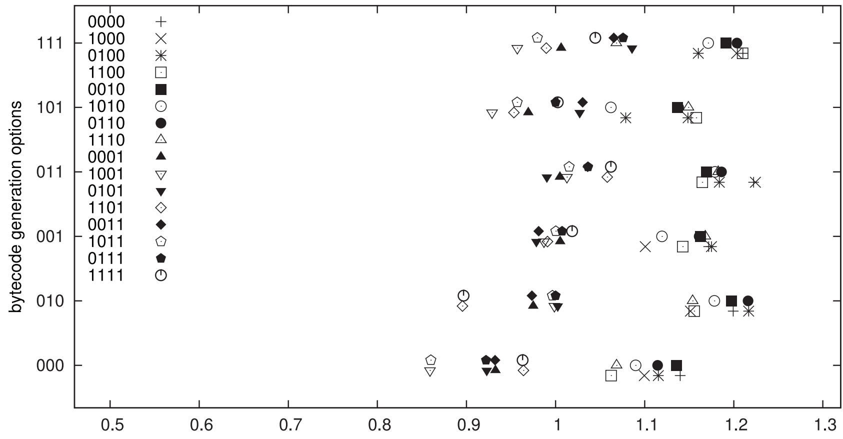

Fig. 21. Cryptoarithmetic puzzle — Intel.

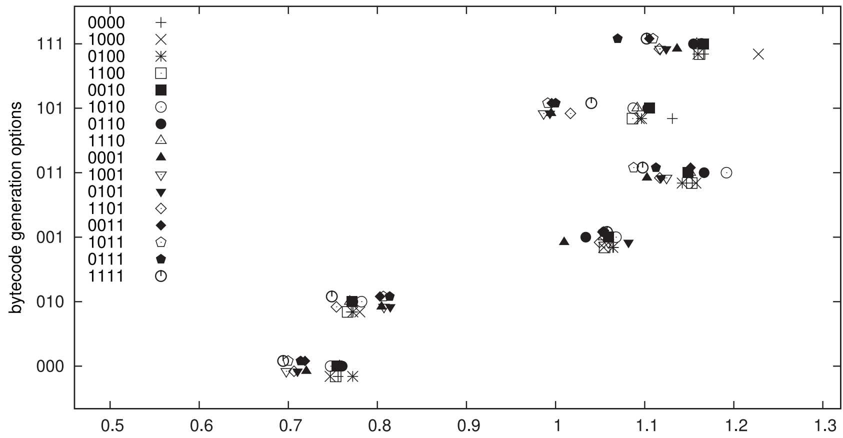

Fig. 22. Computation of the Takeuchi function — Intel.

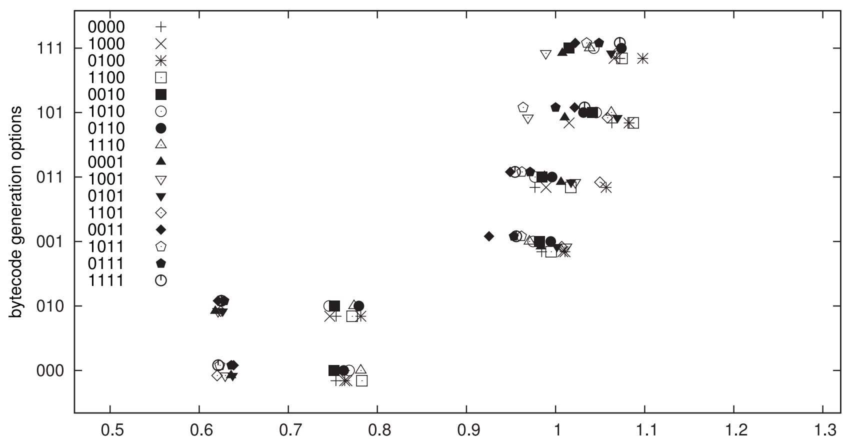

Fig. 23. Symbolic derivation of polynomials — Intel.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

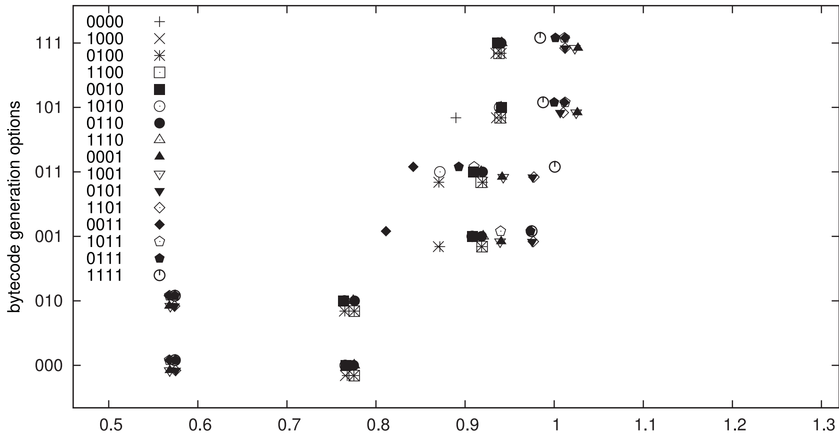

Fig. 24. Naive reverse — Intel.

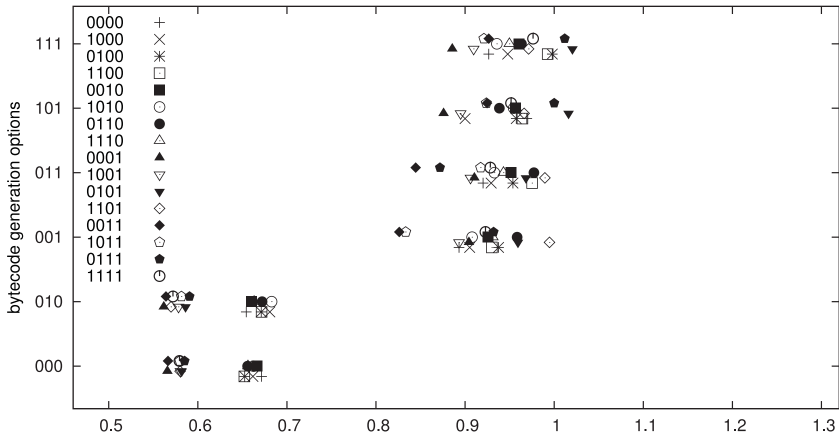

Fig. 25. Symbolic exponentiation of a polynomial — Intel.

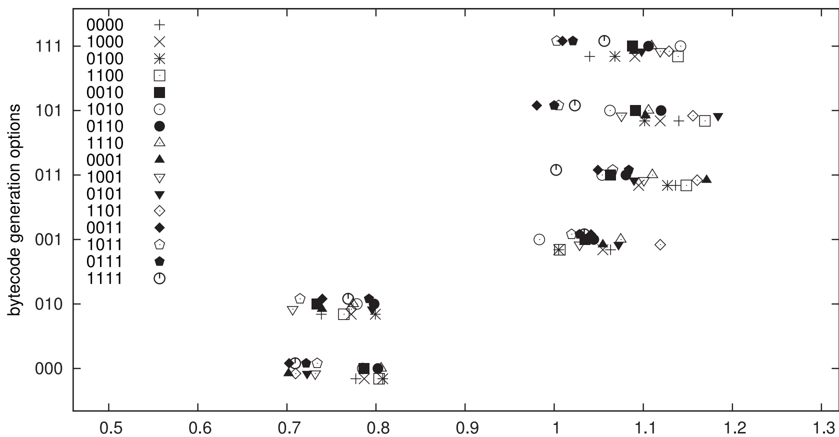

Fig. 26. Version of Boyer–Moore theorem prover — Intel.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

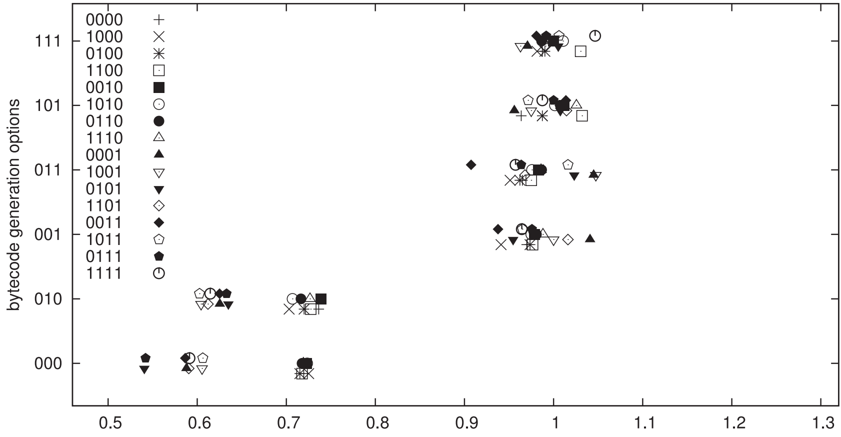

Fig. 27. QuickSort — Intel.

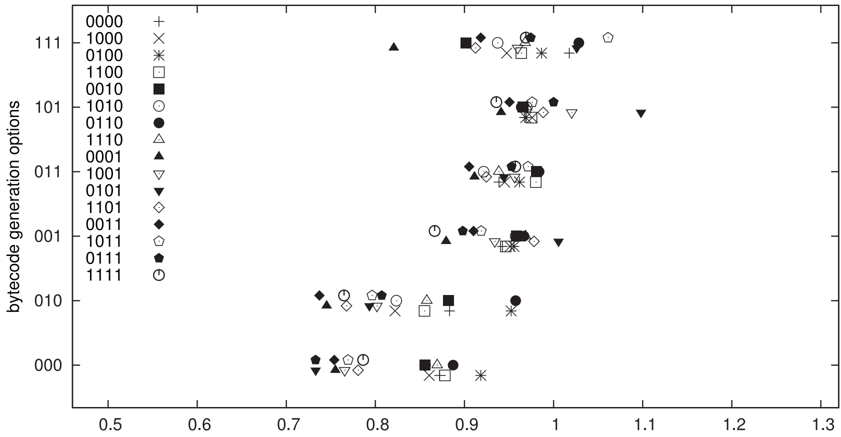

Fig. 28. Calculate primes using the sieve of Eratosthenes — Intel.

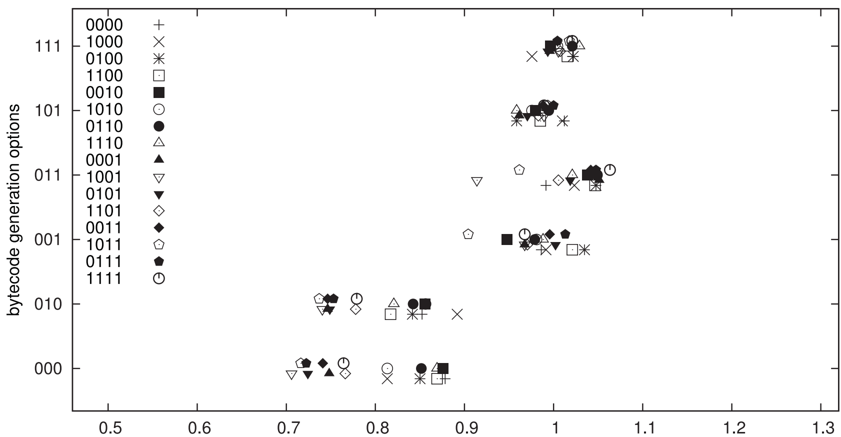

Fig. 29. Natural language query to a geographical database — Intel.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

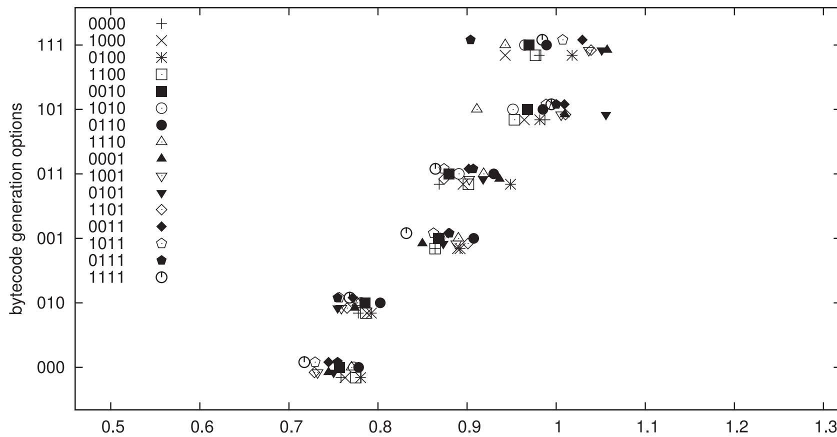

Fig. 30. Chess knights tour — Intel.

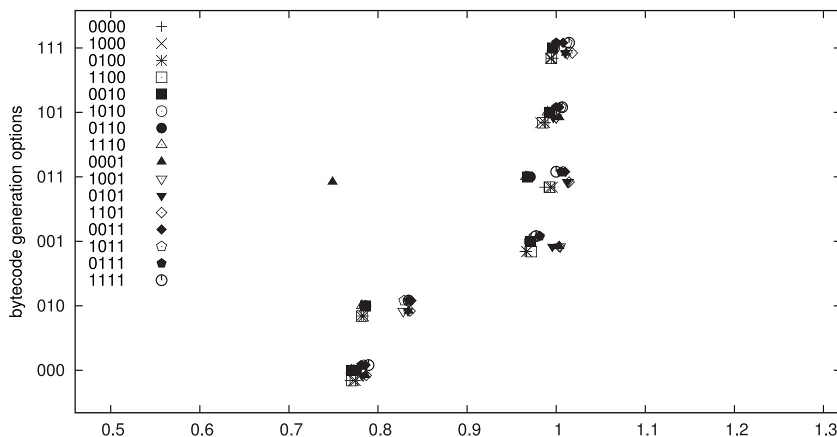

Fig. 31. Simply recursive Fibonacci — Intel.

however, search-based programs which perform mainly arithmetic operations (with the addition of some data structure management in the case of the Queens program), and could in principle be grouped in the same class of programs. This points to the need to perform a finer grain analysis to determine, instruction by instruction, how every engine/bytecode generation option affects execution time, and also how these different options affect each other.

Studying which options are active inside each cluster sheds some light about their contribution to the overall speedup. For example, the upper four clusters of Figures 16 and 17 have in common the use of the ***ib*** option, which generates specialized instructions for built-ins. These clusters have consistently better (and, in some cases, considerably better) speedups than the clusters which do not have it activated. It is, therefore, a candidate to be part of the set of "best options." A similar pattern, although less acute, appears in the results of the PowerPC experiments (Figs. 32 and 33).

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

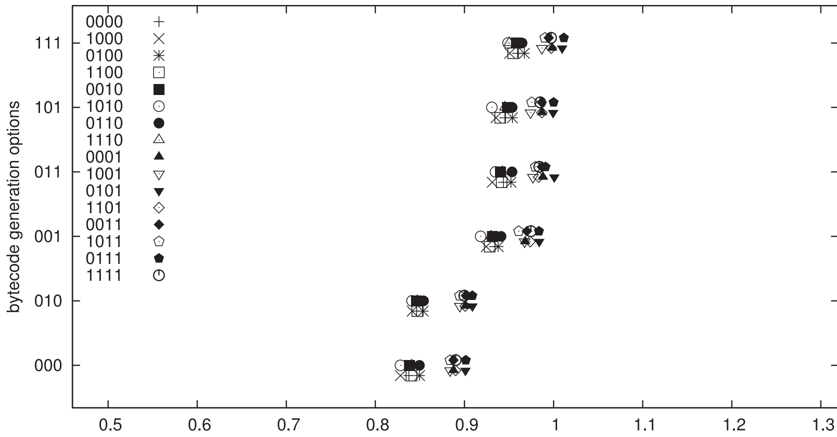

Fig. 32. Geometric average of all benchmarks (with a dot per emulator) — PowerPC.

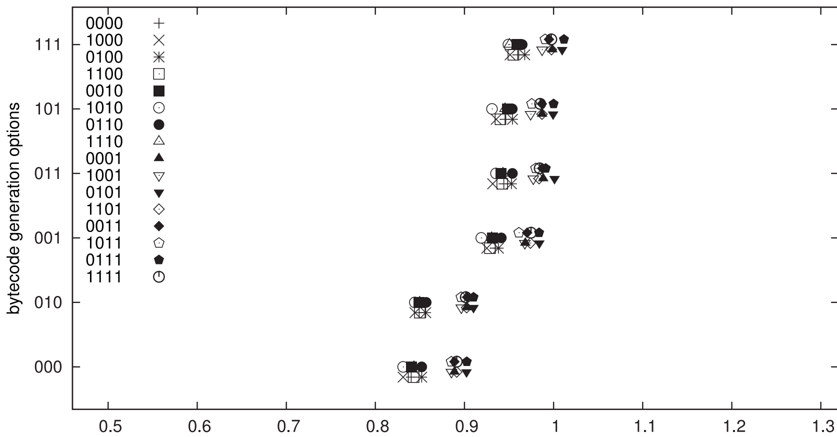

Fig. 33. Arithmetic average of all benchmarks (with a dot per emulator) — PowerPC.

The two leftmost clusters of the group of four at the bottom correspond to executions of emulators generated with the ***rw*** specialization activated, and the two clusters at their right do not have it activated. 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 in the Intel processor. Indeed, a similar result was already observed in (Demoen and Nguyen 2000), and was attributed to modern architectures performing branch prediction and speculative work with redundant units.

5.3.3 Best generation options and overall speedup

An important general question is *which options should be used for the "stock" emulator to be offered to general users*. Our experimental results show that options cannot be considered in isolation — i.e., the overall option set constructed by taking separately the best value for every option does not yield a *better set* (defined as the best options obtained by averaging speedups for every option set). As we have

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

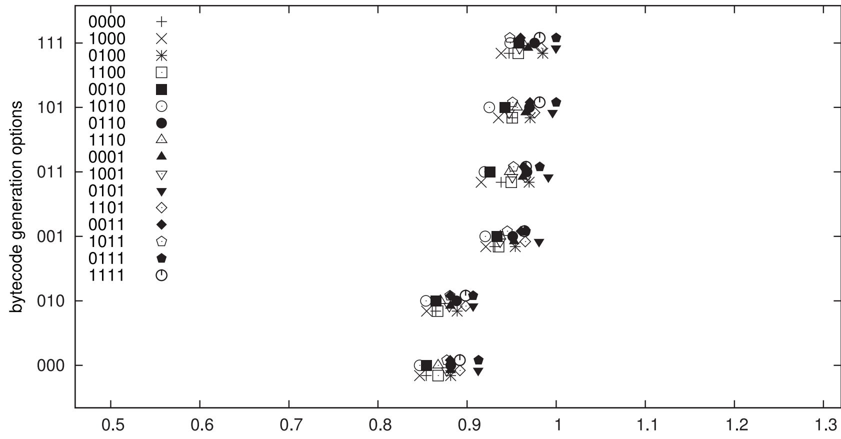

Fig. 34. Factorial involving large numbers — PowerPC.

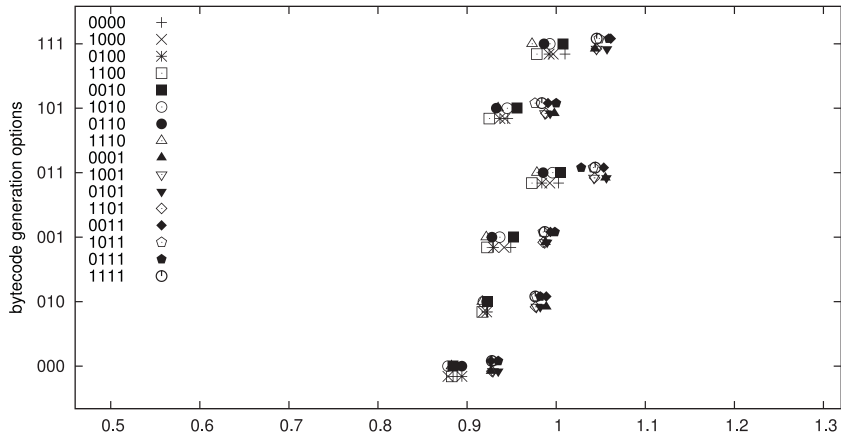

Fig. 35. Queens (with 11 queens to place) — PowerPC.

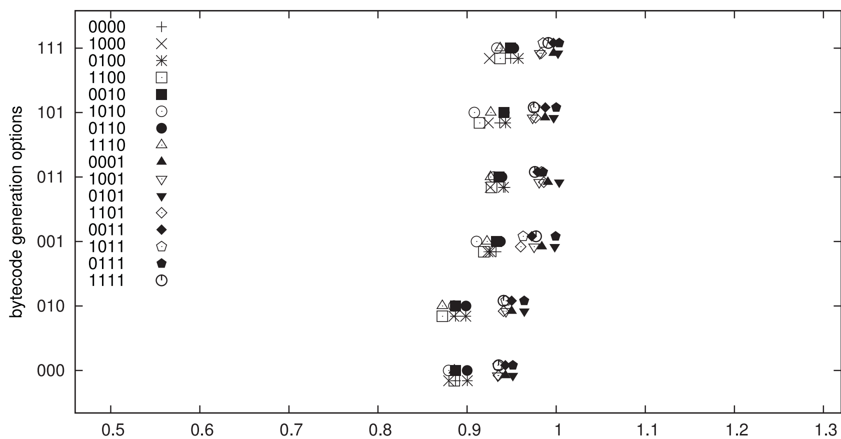

Fig. 36. Cryptoarithmetic puzzle — PowerPC.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

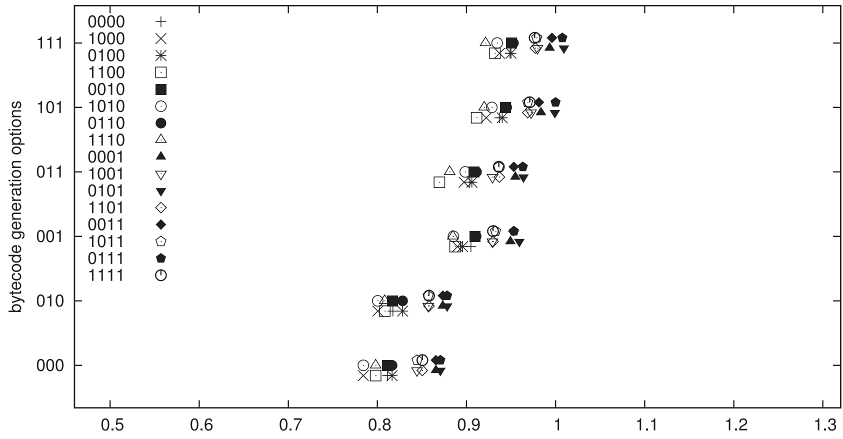

Fig. 37. Computation of the Takeuchi function — PowerPC.

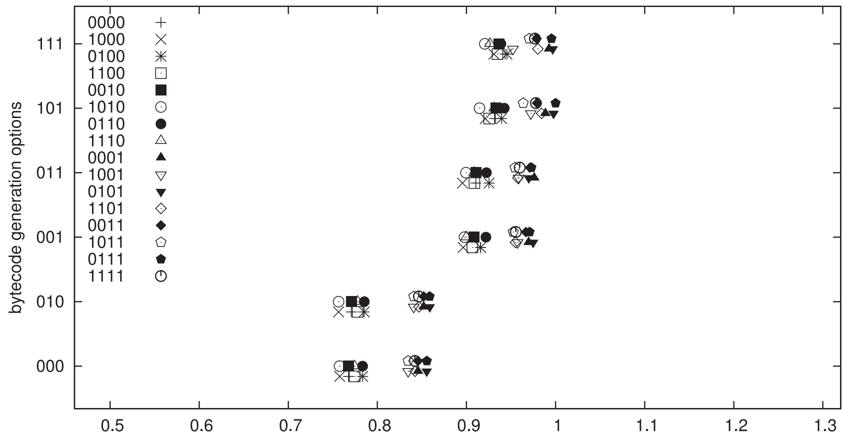

Fig. 38. Symbolic derivation of polynomials — PowerPC.

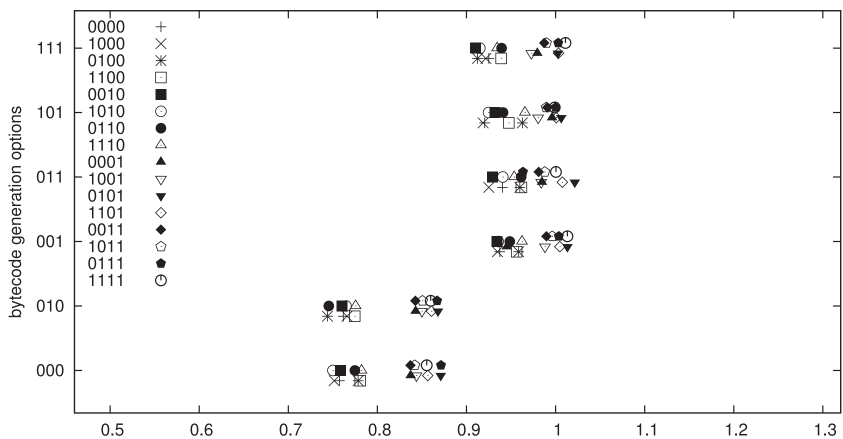

Fig. 39. Naive reverse — PowerPC.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

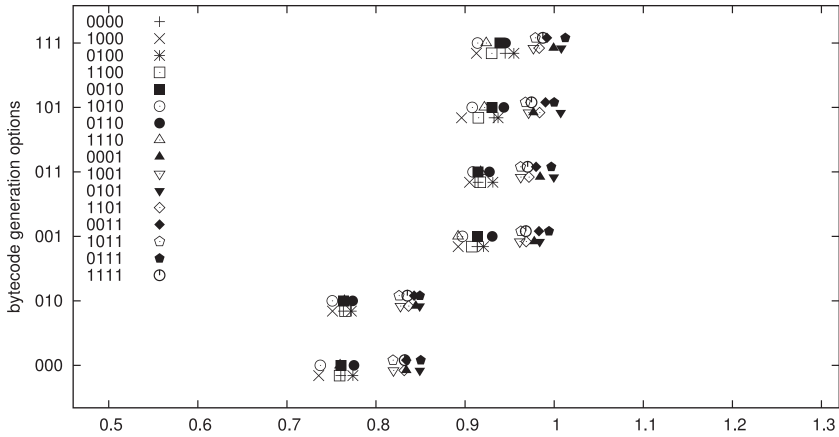

Fig. 40. Symbolic exponentiation of a polynomial — PowerPC.

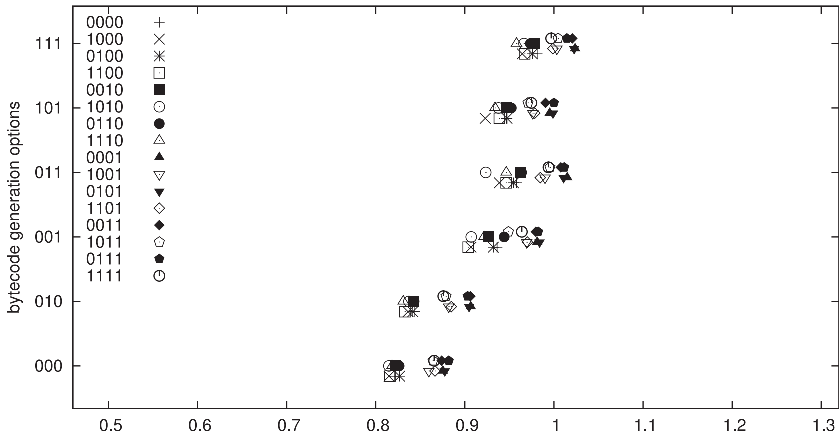

Fig. 41. Version of Boyer–Moore theorem prover — PowerPC.

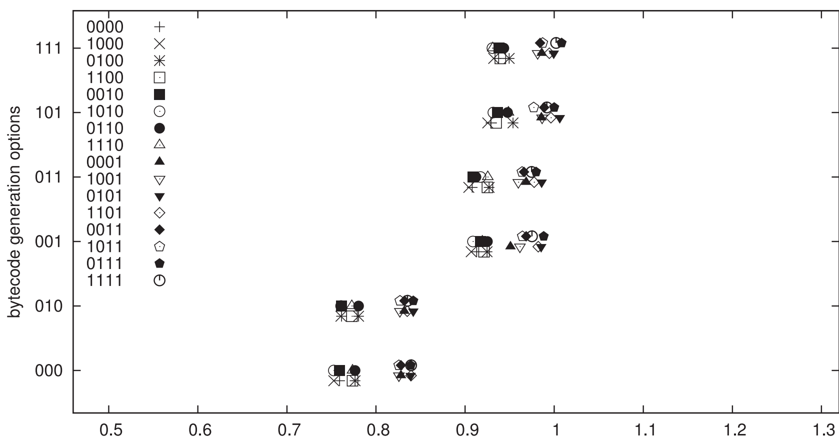

Fig. 42. QuickSort — PowerPC.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

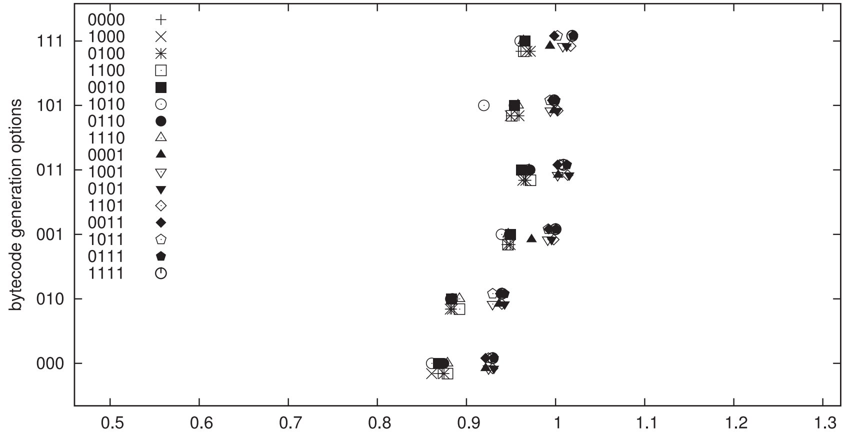

Fig. 43. Calculate primes using the sieve of Eratosthenes — PowerPC.

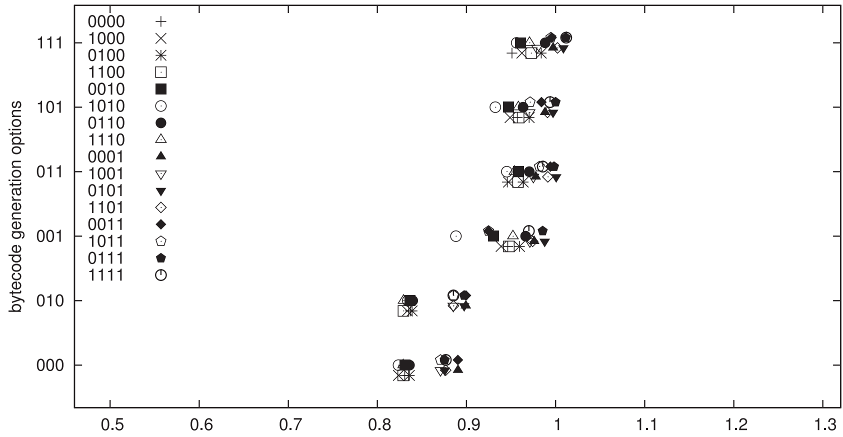

Fig. 44. Natural language query to a geographical database — PowerPC.

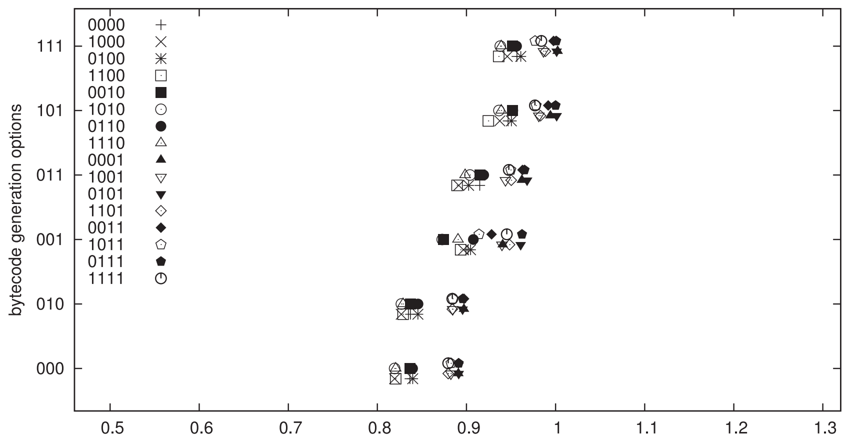

Fig. 45. Chess knights tour — PowerPC.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

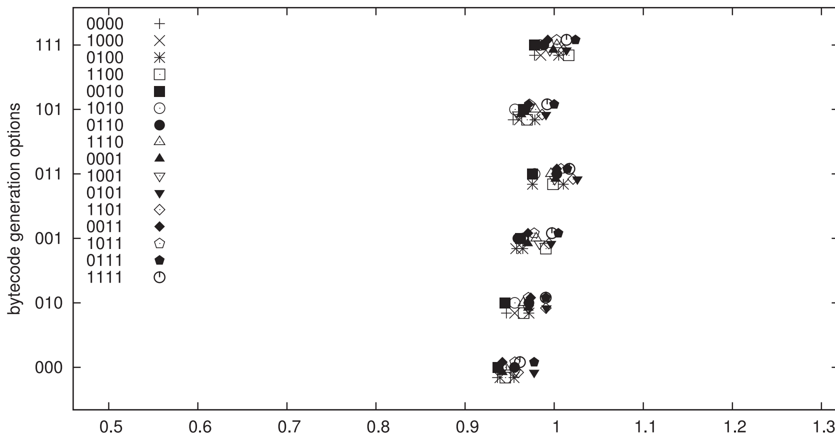

Fig. 46. Simply recursive Fibonacci — PowerPC.

seen, there is some interdependence among options. A more realistic answer is that the average best set of options should come from selecting the rightmost point in the plot corresponding to average speedups. We must however bear in mind that averages always suffer the problem that a small set of good results may bias the average and, in this case, force the selection of an option set which performs worse for a larger set of benchmarks.

In order to look more closely at the effects of individual options (without resorting to extensively listing them and the obtained performance), Tables 4 and 5 show which options produced the best and the worst results time-wise for each benchmark. We include the geometric average as an specific case and the Ciao-1.10 baseline options as reference.

It has to be noted that the best/worst set of options is not the negation of the worst/best options: there are plenty of cases where the same option was (de)activated both for the best and for the worst executions. The observed situation for the PowerPC architecture (Table 5) is more homogeneous: at least some better/worst behaviors really come from combinations which are complementary, and, in the cases where this is not so, the amount of non-complementary options goes typically from 1 to 3 — definitely less than in the x86 case.

Despite the complexity of the problem, some conclusions can be drawn: instruction merging (***im***) is a winner for the x86, probably followed by having a variable number of operands (***ie***), and then by specialized calls to built-ins (***ib***). The first and second options save fetch cycles, while the third one saves processing time in general. It is to be noted that some options appear both in the best and worst cases: this points to interdependencies among the different options.

The performance table for the PowerPC (Table 5) also reveals that instruction merging, having a variable number of operands, and generating specialized instructions for built-ins, are options which bring performance advantages. However, and unlike the x86 table, the *read/write mode* specialization is activated in all the lines of the “best performance” table, and off in the “worst performance.” A similar case

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

Table 4. *Options which gave best/worst performance (x86)*

Benchmark	Best performance							Speed-up	
	ie	ib	im	ts	cc	ur	rw	w.r.t. def.	w.r.t. base
baseline	x		x		x	x	x		
all (geom.)	x	x	x		x		x	1.05	1.28
boyer	x		x		x		x	1.18	1.52
crypt		x	x	x				1.22	1.07
deriv	x	x	x		x			1.10	1.46
factorial	x		x				x	1.02	1.21
fib	x	x	x	x	x		x	1.02	1.32
knights	x	x	x				x	1.06	1.39
nreverse	x	x	x				x	1.03	1.34
poly	x	x	x		x		x	1.02	1.52
primes	x		x		x		x	1.10	1.26
qsort		x	x	x			x	1.05	1.46
queens11	x	x	x	x	x	x	x	1.26	1.46
query		x	x	x	x	x	x	1.06	1.21
tak	x	x	x	x				1.23	1.62

Benchmark	Worst performance							Speed-up	
	ie	ib	im	ts	cc	ur	rw	w.r.t. def.	w.r.t. base
baseline	x		x		x	x	x		
all (geom.)					x		x	0.70	0.88
boyer							x	0.70	0.90
crypt				x			x	0.86	0.75
deriv		x					x	0.62	0.82
factorial						x	x	0.76	0.99
fib		x	x				x	0.75	0.91
knights				x	x	x	x	0.72	0.97
nreverse		x				x	x	0.57	0.95
poly		x					x	0.56	0.74
primes					x	x	x	0.73	0.84
qsort					x		x	0.54	0.84
queens11					x	x	x	0.77	0.75
query				x			x	0.71	0.89
tak				x	x	x	x	0.69	0.92

is that of the *tag switching schema*, in the sense that the selection seems clear in the PowerPC case.

The transformation rules we have applied in our case are of course not the only possible ones, and we look forward to enlarging this set of transformations by, for example, performing a more aggressive merging guided by profiling[20]. Similar work, with more emphasis on the production of languages for microprocessors is presented in Holmer (1993), where a set of benchmarks is used to guide the (constrained) synthesis of such a set of instructions.

[20] Merging is right now limited in depth to avoid a combinatorial explosion in the number of instructions.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

Table 5. *Options which gave best/worst performance (PowerPC)*

Benchmark	Best performance: ie	ib	im	ts	cc	ur	rw	Speed-up: w.r.t. def.	w.r.t. base
baseline	x		x		x	x	x		
all (geom.)	x	x	x		x	x	x	1.01	1.21
boyer	x	x	x				x	1.02	1.25
crypt		x	x		x		x	1.00	1.13
deriv	x		x		x	x	x	1.00	1.30
factorial	x		x		x	x	x	1.00	1.02
fib		x	x		x		x	1.03	1.17
knights	x	x	x				x	1.00	1.10
nreverse		x	x		x		x	1.02	1.20
poly	x	x	x		x	x	x	1.01	1.35
primes	x	x	x		x	x	x	1.02	1.33
qsort	x	x	x		x	x	x	1.01	1.17
queens11	x	x	x			x	x	1.06	1.33
query	x	x	x	x	x	x	x	1.01	1.20
tak	x	x	x		x		x	1.01	1.22

Benchmark	Worst performance: ie	ib	im	ts	cc	ur	rw	Speed-up: w.r.t. def.	w.r.t. base
baseline	x		x		x	x	x		
all (geom.)				x				0.82	0.99
boyer				x		x		0.81	0.99
crypt		x		x	x	x		0.87	0.98
deriv		x		x		x		0.76	0.99
factorial				x		x		0.85	0.97
fib								0.94	0.99
knights				x		x		0.82	1.00
nreverse		x			x			0.74	0.98
poly				x				0.74	0.98
primes				x		x		0.86	0.97
qsort				x				0.75	0.99
queens11				x				0.88	0.99
query				x				0.82	0.99
tak				x		x		0.78	0.99

We want to note that although exceeding the speed of a hand-crafted emulator is not the main concern in this work[21], the performance obtained by the implementation of the emulator in imProlog allows us to conclude that the imProlog approach can match the performance of lower-level languages, while making it possible to apply non-trivial program transformation techniques.

Additional experiments carried out in a very different context (that of embedded systems and digital signal processing using Ciao Prolog (Carro *et al.* 2006), which

[21] In order to do that, a better approach would probably be to start off by finding performance bottlenecks in the current emulator and redesigning/recoding it. We want to note, however, that we think that our approach can greatly help in making this redesign and recoding easier.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

pertains to a realm traditionally considered disadvantageous for symbolic languages) showed also very good performance — only 20% slower than a comparable C program — and also very good speedups (up to 7-fold compared with a bytecode-based implementation). Analysis and compilation techniques similar to those applied in this paper were used, but put to work in a program using the full Prolog language.

6 Conclusions

We have designed a language (imProlog, a variation of Prolog with some imperative features) and its compiler, and we have used this language to describe the semantics of the instructions of a bytecode interpreter (in particular, the Ciao engine). The imProlog language, with the proposed constraints and extensions, is semantically close enough to Prolog to share analysis, optimization and compilation techniques, but at the same time it is designed to make translation into very efficient C code possible. The low-level code for each instruction and the definition of the bytecode is 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.

Our main conclusion is that indeed the proposed approach can produce emulators that are as efficient as the best state-of-the-art emulators hand-coded in C, while starting from a much higher-level description. This high-level nature of the language used allows avoiding some of the typical problems that hinder the development and maintenance of highly-tuned emulators.

In our experience, in addition to greatly increased clarity and readability the port allowed replacing many duplicated code structures, redundant hand-made specializations, and a large number of C macros (which notwithstanding do help in writing less code, but they are not easily recognized in the source — leading often to programmer confusion — they are prone to subtle scoping-related errors, and they are typically not understood by automatic compilation tools), with more abstract predicates, sometimes adorned with some annotations to guide the transformations.

The proposed approach makes it possible to perform non-trivial transformations on both the emulator and the instruction level (e.g., unfolding and partial evaluation of instruction definitions, instruction merging or specialization, etc.). The different transformations and code generation options, result in different grades of optimization/specialization and different bytecode languages from a single (higher-level) abstract machine definition.

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 slightly by using some code patterns and optimizations not explored in the initial emulator, and, what is more important, starting from abstract machine definitions in imProlog.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

Performance evaluation of non-trivial variations in the emulator code showed that some results are hard to predict and that there is no absolute winner for all architectures and programs. On the other hand, it is increasingly difficult to reflect all the variations in a single source using more traditional methods like `m4` or `cpp` macros. Automatic program manipulation at the emulator level represents a very attractive approach, and although difficult, the problem becomes more tractable when the abstraction level of the language to define the virtual machine is raised and enriched with some problem-specific declarations.

As future work, it would be interesting to study the generation of efficient code for full Prolog with imProlog features. A partial application of such compilation techniques was already put to work in the real-time digital sound processing application written in Prolog mentioned before, with very successful results.

Acknowledgements

This work was funded in part by the Information Society Technologies program of the European Commission, through EU project FP7 318337 *ENTRA*, by the Spanish Ministry of Economy and Competitiveness through projects TIN2012-39391 *StrongSoft* and TIN2008-05624 *DOVES*, and by the Madrid Regional Government through project S-0505/TIC/0407 *PROMESAS*.

References

Ait-Kaci, H. 1991. *Warren's Abstract Machine, A Tutorial Reconstruction.* MIT, Cambridge, MA.

Bueno, F., Deransart, P., Drabent, W., Ferrand, G., Hermenegildo, M., Maluszynski, J. and Puebla, G. 1997. On the role of semantic approximations in validation and diagnosis of constraint logic programs. In *Proc. of 3rd International Workshop on Automated Debugging–AADEBUG'97*. University of Linköping, Sweden, 155–170.

Carlsson, M. 1991. The SICStus Emulator. SICS Technical Report T91:15, Swedish Institute of Computer Science.

Carro, M., Morales, J., Muller, H., Puebla, G. and Hermenegildo, M. 2006. High-level languages for small devices: A case study. In *Compilers, Architecture, and Synthesis for Embedded Systems*, K. Flautner and T. Kim, Eds. ACM Press/Sheridan, New York, NY, USA, 271–281.

Casas, A., Cabeza, D. and Hermenegildo, M. 2006. A syntactic approach to combining functional notation, lazy evaluation and higher-order in LP systems. In *Proc. of 8th International Symposium on Functional and Logic Programming (FLOPS'06)*. Fuji Susono, Japan, 142–162.

Codognet, P. and Diaz, D. 1995. WAMCC: Compiling Prolog to C. In *Proc. of 12th International Conference on Logic Programming*, L. Sterling, Ed. MIT, Cambridge, MA, 317–331.

Damas, L. and Milner, R. 1982. Principal type-schemes for functional programs. In *Proc. of 9th Annual Symposium on Principles of Programming Languages*, New York, NY, USA, 207–212.

Dart, P. and Zobel, J. 1992. A regular type language for logic programs. In *Types in Logic Programming*. MIT, Cambridge, MA, 157–187.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

Debray, S., López-García, P. and Hermenegildo, M. 1997. Non-failure analysis for logic programs. In *Proc. of International Conference on Logic Programming*. MIT, Cambridge, MA, 48–62.

Demoen, B. 2012. *h-Prolog*. URL: `http://people.cs.kuleuven.be/~bart.demoen/hProlog/`.

Demoen, B., de la Banda, M. G., Marriott, K., Schachte, P. and Stuckey, P. 1998. Global variables in HAL, a logic implementation. CW Reports, CW271. Department of Computer Science, K.U. Leuven, Leuven, Belgium.

Demoen, B. and Nguyen, P.-L. 2000. So many WAM variations, So little time. In *Proc. of Computational Logic*, Springer-Verlag, 1240–1254.

Dorward, S., Pike, R., Presotto, D. L., Ritchie, D., Trickey, H. and Winterbottom, P. 1997. Inferno. In *Proc. of 42nd IEEE International Computer Conference*, IEEE, Washington, DC, USA.

Gallagher, J. and de Waal, D. 1994. Fast and precise regular approximations of logic programs. In *Proc. of the 11th International Conference on Logic Programming*, P. Van Hentenryck, Ed. MIT, Cambridge, MA, 599–613.

Giannesini, F., Kanoui, H., Pasero, R. and Caneghem, M. V. 1985. *Prolog*. InterEditions, 87 Avenue du Maine, 75014, Paris. ISBN 2-7296-0076-0.

Gosling, J., Joy, B., Steele, G. and Bracha, G. 2005. *Java(TM) Language Specification, The (3rd Edition)*. Addison-Wesley Professional, Boston, MA, USA.

Gudeman, D., Bosschere, K. D. and Debray, S. 1992. `jc`: An efficient and portable sequential implementation of Janus. In *Proc. of Joint International Conference and Symposium on Logic Programming*, MIT, Cambridge, MA, 399–413.

Henderson, F., Conway, T. and Somogyi, Z. 1995. Compiling logic programs to C using GNU C as a portable assembler. In *Proc. of ILPS 1995 Postconference Workshop on Sequential Implementation Technologies for Logic Programming*, Portland, Oregon, 1–15.

Henderson, F. and Somogyi, Z. 2002. Compiling mercury to high-level C code. In *Proc. of Compiler Construction*, R. Nigel Horspool, Ed. Lecture Notes in Computer Science, vol. 2304. Springer-Verlag, Berlin Heidelberg, 197–212.

Hermenegildo, M., Puebla, G. and Bueno, F. 1999. Using global analysis, partial specifications, and an extensible assertion language for program validation and debugging. In *The Logic Programming Paradigm: a 25–Year Perspective*, K. R. Apt, V. Marek, M. Truszczynski and D. S. Warren, Eds. Springer-Verlag, Berlin Heidelberg, 161–192.

Hermenegildo, M., Puebla, G., Bueno, F. and López-García, P. 2005. Integrated program debugging, verification, and optimization using abstract interpretation (and The Ciao system preprocessor). *Science of Computer Programming 58,* 1–2, 115–140.

Hermenegildo, M. V., Bueno, F., Carro, M., López, P., Mera, E., Morales, J. and Puebla, G. 2012. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming 12,* 1–2, 219–252. URL: http://arxiv.org/abs/1102.5497.

Hind, M. and Pioli, A. 2000. Which pointer analysis should I use? In *Proc. of International Symposium on Software Testing and Analysis*, ACM, New York, NY, USA, 113–123.

Holmer, B. K. 1993. Automatic design of computer instruction sets. Ph.D. thesis, University of California, Berkeley, CA.

Lattner, C. and Adve, V. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California.

Morales, J., Carro, M. and Hermenegildo, M. 2004. Improving the compilation of prolog to C using moded types and determinism information. In *Proceedings of the 6th International Symposium on Practical Aspects of Declarative Languages*, Lecture Notes in Computer Science, vol. 3057. Springer-Verlag, Heidelberg, Germany, 86–103.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

Morales, J., Carro, M. and Hermenegildo, M. 2007. Towards description and optimization of abstract machines in an extension of prolog. In *Logic-Based Program Synthesis and Transformation (LOPSTR'06)*, G. Puebla, Ed. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, vol. 4407. 77–93.

Morales, J., Carro, M. and Hermenegildo, M. 2008. Comparing tag scheme variations using an abstract machine generator. In *10th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, ACM Press, New York, NY, USA, 32–43.

Morales, J., Carro, M., Puebla, G. and Hermenegildo, M. 2005. A generator of efficient abstract machine implementations and its application to emulator minimization. In *International Conference on Logic Programming*, M. Gabbrielli and G. Gupta, Eds. Lecture Notes in Computer Science, vol. 3668. Springer-Verlag, Berlin Heidelberg, 21–36.

Nässén, H., Carlsson, M. and Sagonas, K. 2001. Instruction merging and specialization in the SICStus prolog virtual machine. In *Proc. of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, ACM, New York, NY, USA, 49–60.

Nori, K. V., Ammann, U., Jensen, K., Nageli, H. H. and Jacobi, C. 1981. Pascal-p implementation notes. In *Pascal - The Language and its Implementation*, D. W. Barron, Ed. John Wiley, London, UK, 125–170.

Paleczny, M., Vick, C. and Click, C. 2001. The Java hotspotTM Server Compiler. Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium – Volume 1. JVM'01. USENIX Association. Berkeley, CA, USA.

Puebla, G., Bueno, F. and Hermenegildo, M. 2000a. An assertion language for constraint logic programs. In *Analysis and Visualization Tools for Constraint Programming*, P. Deransart, M. Hermenegildo and J. Maluszynski, Eds. Lecture Notes in Computer Science, vol. 1870. Springer-Verlag, Berlin Heidelberg, 23–61.

Puebla, G., Bueno, F. and Hermenegildo, M. 2000b. Combined static and dynamic assertion-based debugging of constraint logic programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*. Lecture Notes in Computer Science, vol. 1817. Springer-Verlag, 273–292.

Rigo, A. 2004. Representation-based just-in-time specialization and the psyco prototype for python. In *PEPM '04: Proc. of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ACM, New York, USA, 15–26.

Rigo, A. and Pedroni, S. 2006. PyPy's approach to virtual machine construction. In *Dynamic Languages Symposium*, ACM, New York, NY, USA.

Santos-Costa, V., Damas, L. and Rocha, R. 2011. The YAP prolog system. *Theory and Practice of Logic Programming*. URL: http://arxiv.org/abs/1102.3896v1.

Santos-Costa, V., Sagonas, K. and Lopes, R. 2007. Demand-driven indexing of prolog clauses. In *Proc. of International Conference on Logic Programming*. Lecture Notes in Computer Science, vol. 4670. Springer-Verlag, Berlin Heidelberg, 395–409.

Schachte, P. 1997. Global variables in logic programming. In *Proc. of International Conference on Logic Programming*, MIT, Cambridge, MA, 3–17.

Schrijvers, T. and Frühwirth, T. W. 2006. Optimal union-find in constraint handling rules. *TPLP 6,* 1-2, 213–224.

Somogyi, Z., Henderson, F. and Conway, T. 1996. The execution algorithm of Mercury: An efficient purely declarative logic programming language. *Journal of Logic Programming 29,* 1–3, 17–64.

Taivalsaari, A. 1998. Implementing a Java Virtual Machine in the Java Programming Language. Technical Report, Sun Microsystems. Mar. Technical report SMLI TR-98-64.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press

Tarau, P. 2006. *BinProlog 2006 version 11.x Professional Edition User Guide*. BinNet Corporation. URL: http://www.binnetcorp.com/.

Taylor, A. 1991. High performance prolog implementation through global analysis. Slides of the invited talk at PDK'91, Kaiserslautern, Germany.

Van Roy, P. 1994. 1983–1993: The wonder years of sequential Prolog implementation. *Journal of Logic Programming 19-20*, 385–441.

Van Roy, P. and Despain, A. 1992. High-performance logic programming with the aquarius prolog compiler. *IEEE Computer Magazine*, 54–68.

Vaucheret, C. and Bueno, F. 2002. More precise yet efficient type inference for logic programs. In *International Static Analysis Symposium*, Lecture Notes in Computer Science, vol. 2477. Springer-Verlag, Berlin Heidelberg, 102–116.

Warren, D. 1977. Applied logic—its use and implementation as programming tool. Ph.D. thesis, University of Edinburgh, UK. Also available as SRI Technical Note 290.

Warren, D. H. D. 1983. An abstract prolog instruction set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025.

Wielemaker, J. 2010. *The SWI-Prolog User's Manual 5.9.9*. URL: http://www.swi-prolog.org.

Zhou, N.-F. 2007. A register-free abstract prolog machine with jumbo instructions. In *Proc. of 23rd International Conference on Logic Programming*, V. Dahl and I. Niemelä, Eds. Lecture Notes in Computer Science, vol. 4670. Springer, Berlin Heidelberg, 455–457.

https://doi.org/10.1017/S1471068414000672 Published online by Cambridge University Press