On the Performance of Bytecode Interpreters in Prolog

Philippp Körner, David Schneider and Michael Leuschel

Institut für Informatik, Heinrich Heine University Düsseldorf, Germany
`{p.koerner, david.schneider, leuschel}@hhu.de,`

Abstract. The semantics and the recursive execution model of Prolog make it very natural to express language interpreters in form of AST (Abstract Syntax Tree) interpreters where the execution follows the tree representation of a program. An alternative implementation technique is that of bytecode interpreters. These interpreters transform the program into a compact and linear representation before evaluating it and are generally considered to be faster and to make better use of resources.
In this paper, we discuss different ways to express the control flow of interpreters in Prolog and present several implementations of AST and bytecode interpreters. On a simple language designed for this purpose, we evaluate whether techniques best known from imperative languages are applicable in Prolog and how well they perform. Our ultimate goal is to assess which interpreter design in Prolog is the most efficient as we intend to apply these results to a more complex language. However, we believe the analysis in this paper to be of more general interest.

1 Introduction

Writing simple language interpreters in Prolog is pretty straightforward. Definite clause grammars (DCGs) enable parsing of the program, and interpretation of the resulting abstract syntax tree (AST) can be expressed in an idiomatic, recursive way: Selecting which predicate to execute in order to evaluate a part of a program is done by unifying the part of the program to be executed next with the set of rules in Prolog's database that implement the language semantics. Subsequent execution steps can be chosen by using logic variables that are bound to substructures of the matched node.

Although this approach to interpreter construction is a natural match to Prolog, the question remains if it is the most efficient way to implement the instruction dispatching logic. In particular, we have developed such an interpreter [7] for the entire B language [1] and want to evaluate the potential for improving its performance, by using alternate implementation techniques.

Interpreters implemented in imperative languages, especially low-level languages, often make use of alternative techniques to implement the dispatching logic, taking advantage of available data structures and programming paradigms.

In this article, we explore if some of these techniques can be implemented in Prolog or applied in interaction with a Prolog runtime with the goal to

```
# the initial environment (i.e. input): base = 2, exponent = 5

# the program
val = 1;
while exponent > 0 {
    val = val * base;
    exponent = exponent - 1;
}
```

Fig. 1: An Acol program implementing a power function

assess whether the instruction dispatching for language interpreters can be made faster while keeping the language semantics in Prolog. In order to examine the performance of different dispatching models in Prolog, we have defined a simple imperative language named Acol, which is briefly described in Section 2. For Acol, we have created several implementations described in Section 3, that use different paradigms for the dispatching logic. In Section 4, we evaluate our approach on a set of benchmarks written in Acol, executing the interpreters both on SICStus [3] and SWI-Prolog [12]. Finally, we give our conclusions in Section 5.

2 A Simple Language

As a means to evaluate different interpreter designs, we have defined a very simple and limited language named Acol[1].

Acol is an imperative language consisting of three kinds of statements: while-loops, if-then-else statements and variable assignments. The only supported value type is integer. Furthermore, Acol offers a few arithmetic operators (addition, subtraction, multiplication and modulo), comparisons (less than (or equal to), greater than (or equal to) and equals), as well as a boolean `not` operator.

A simple Acol program implementing a power function is shown in Fig. 1.

3 Interpreter Implementations

There are many ways to implement Acol, in C as well as in Prolog. Considering several interpreter implementation techniques, in this section, we will describe possible designs of interpreters and the closely related representations of the Acol programs in Prolog. The interpreters are based on either traversing the abstract syntax tree representation of a program or on compiling the program to bytecode first and evaluating this more compact representation instead.

We opted to implement stack-based interpreters as their design tends to be simpler. The alternative – register-based virtual machines – usually are faster [10] and allow more advanced techniques such as register allocation optimisation in

[1] Acol is *not* a backronym for Acol is a computable language.

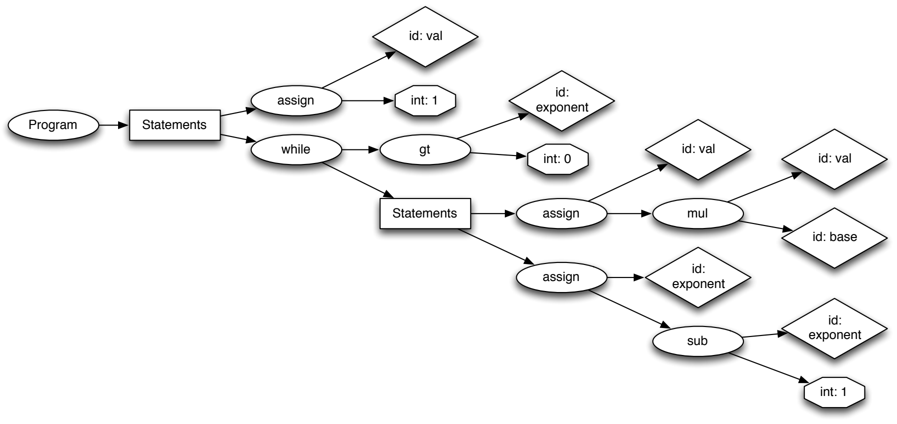

Fig. 2: AST

order to reduce the amount of load and store instructions. Yet, this endeavour would be far more involved and could be considered if this prototype already shows proper speed-ups.

All interpreters share the same implementation of the language semantics exposed by an object-space API [8]. The objects space contains the code that creates integer objects, performs arithmetic operations, compares values, and manages the environment. In order to keep the implementations simple and compatible, all interpreters that we present call into this same object space. Nonetheless, the interpreters differ very much in the representation of the program and, hence, in the process of dispatching. The full code of all interpreters, benchmark scripts and results can be found at:

https://github.com/pkoerner/prolog-interpreters

In order to discuss the differences, we will translate the small example program shown in Fig. 1 into the different representations and show an excerpt of the interpretation logic for each paradigm. In Fig. 2, the AST for the example program is depicted.

3.1 AST Interpreter

The most idiomatic way to implement an interpreter in Prolog is in form of an AST-interpreter since it synergises very well with its execution model.

The data structure used for this interpreter is the tree representation of the program as generated by the parser. In Prolog, the AST can be represented as a single term as shown in Fig. 3. The program itself is a Prolog list of statements. However, every statement is represented as its own sub-tree. Block statements, i.e. the body of `if` and `while` instructions, will contain a list of statements themselves.

```
[assign(id(val), int(1)),
 while(gt(id(exponent), int(0)),
       [assign(id(val), mul(id(val), id(base))),
        assign(id(exponent), sub(id(exponent), int(1)))])]
```

Fig. 3: Prolog representation of the AST

```
ast_int([], Env, _Objspace, Env).
ast_int([H|T], EnvIn, Objspace, EnvOut) :-
    ast_int(H, EnvIn, Objspace, Env), ast_int(T, Env, Objspace, EnvOut).
ast_int(if(Cond, Then, Else), EnvIn, Objspace, EnvOut) :-
    eval(Cond, EnvIn, Objspace, X),
    (X == true -> ast_int(Then, EnvIn, Objspace, EnvOut)
              ;  ast_int(Else, EnvIn, Objspace, EnvOut)).
ast_int(assign(id(Var), Expr), EnvIn, Objspace, EnvOut) :-
    eval(Expr, EnvIn, Objspace, Res), Objspace:store(EnvIn, Var, Res, EnvOut).
ast_int(while(Cond, Instr, _Invariant, _Variant), EnvIn, Objspace, EnvOut) :-
    ast_while(Cond, Instr, EnvIn, Objspace, EnvOut).
```

Fig. 4: Dispatching in a Prolog AST interpreter

The AST interpreter will examine the first element of the list, execute this statement and continue with the rest of the list, as can be seen in Fig. 4. Every sub-tree encountered this way is evaluated recursively.

Choosing the implementation for each node in the tree is done by unifying the current root node with the set of evaluation rules. This approach benefits from the first argument indexing [11] optimisation done by most Prolog systems.

3.2 Bytecode Interpreters

We have defined a simple set of bytecodes, described below, as a compilation target for Acol programs. Based on these instructions we will introduce a series of bytecode-interpreters that explore different implementation approaches in Prolog and C.

As many bytecode interpreters for other languages, ours are *stack-based*. Some opcodes may create or load objects and store them on the evaluation stack, e.g. `push` or `load`. Yet others may in turn consume objects from the stack and create a new one in return, e.g. `add`. Lastly, a single opcode is used to manipulate the environment, i.e. `assign`. An exhaustive list is shown in Table 1.

Imperative Bytecode Interpreter Usually, bytecode interpreters are written in imperative languages, that are rather low-level, e.g. C, that allow more control about how objects are laid out in memory and provide fine-grained control over the flow of execution.

To introduce the concept of a bytecode interpreter, we present an implementation of Acol beyond Prolog, that is purely written in C.

Table 1: A bytecode for the described language

#	Name	Arguments	Semantics
10	jump	4 bytes encoded PC	jumps to new PC
11	jump-if-false	4 bytes encoded PC	jumps to new PC if top element is falsey
12	jump-if-true	4 bytes encoded PC	jumps to new PC if top element is truthy
20	push1	1 byte encoded integer	push the argument on the stack
21	push4	4 bytes encoded integer	push the argument on the stack
40	load	4 bytes encoded variable ID	push variable on the stack
45	assign	4 bytes encoded variable ID	store top of the stack in variable
197	mod	-	pop operands, push result of operation
198	mul	-	pop operands, push result of operation
199	sub	-	pop operands, push result of operation
200	add	-	pop operands, push result of operation
240	not	-	pop operand, push negation
251	eq	-	pop operands, push result of comparison
252	le	-	pop operands, push result of comparison
253	lt	-	pop operands, push result of comparison
254	ge	-	pop operands, push result of comparison
255	gt	-	pop operands, push result of comparison

The bytecode is stored as a block of memory, that can be interpreted as an array of bytes. The index of this array that should be interpreted next is called the program counter. After that opcode is executed, the program counter is incremented by one, plus the size of its arguments. However, it may be set to an arbitrary index by opcodes implementing jumps. Integer arguments are encoded in reverse byte order.

The dispatching logic is implemented as a `switch`-statement that is contained in a loop. An excerpt of the implementation of our bytecode-interpreter in C is shown in Fig. 5. Every `case` block contains an implementation of that specific opcode. After the opcode is executed, the program counter is advanced or reset and the next iteration of the main loop is commenced.

C-Interfaces We made the digression into an interpreter written in C not only to present the concept of bytecode interpreters. Instead, we can utilise the same dispatching logic, but instead of calling an object space that is implemented in C, we can use the C interfaces provided by the Prolog runtimes we consider (SICStus and SWI-Prolog) to call arbitrary Prolog predicates. This way, we can query the aforementioned object space that contains the semantics of ACOL, but is implemented in Prolog. An excerpt when using the C interface of SWI-Prolog is shown in Fig. 6.

For the C-interface, we re-use the linear bytecode from the Prolog interpreter above. The list of bytecodes is passed to C, which allocates a C array, iterates over the list and copies the instructions into the array. Then, the main loop dispatches in C, but the objects on the evaluation stack are created and the operations are executed by Prolog predicates.

```
while (pc < bc_len) {
    unsigned char *arg = bc + pc + 1;
    switch (bc[pc]) {
        case JUMP:
            pc = decode_arg4(arg); break;
        case LOAD:
            index = decode_arg4(arg);
            push(stack, env[index]);
            pc += 5; break;
        case ASSIGN:
            env[arg] = pop(stack);
            pc += 5; break;
        case ADD:
            b = pop(stack);
            a = pop(stack);
            push(stack, add(a, b));
            pc++; break;
        // ... many further cases
    }
}
```

Fig. 5: Dispatching logic in C

```
while (pc < bc_len) {
    unsigned char *arg = bc + pc + 1;
    switch (bc[pc]) {
        case JUMP:
            pc = decode_arg4(arg); break;
        case LOAD:
            index = decode_arg4(arg)
            push(stack, env[index]);
            pc += 5; break;
        case ASSIGN:
            index = decode_arg4(arg);
            PL_put_term(env[index], pop(s));
            pc += 5; break;
        case ADD:
            arg1 = PL_new_term_refs(3);
            arg2 = arg1 + 1;
            var = arg1 + 2;
            PL_put_term(arg2, pop(s));
            PL_put_term(arg1, pop(s));
            PL_call_predicate(NULL,
                              PL_Q_NORMAL,
                              predicate_add,
                              arg1);
            push(s, var);
            pc++; break;
        // ...  many further cases
    }
}
```

Fig. 6: Dispatching logic using SWI-Prolog's C-Interface

Prolog Facts The main issue with bytecode interpreters in Prolog is the efficient implementation of jumps to other parts of the bytecode. With an interpreter in C, all we have to do is to re-assign the program counter variable. Prolog, however, does not offer arrays with constant-time indexing[2].

The idiomatic way to simulate an array would be to use a Prolog list, but on this data structure we can perform lookups only in $\mathcal{O}(n)$. Yet, there are other representations of the program that allow jumping to another position faster.

One way to express such a lookup in $\mathcal{O}(1)$ is to transform the bytecode into Prolog terms `bytecode(ProgramCounter, Instruction, Arguments)`. Those terms are written into a seperate Prolog module that is loaded afterwards. The first argument indexing optimisation then allows lookups in constant time.

In contrast to an interpreter written in C, it does not perform well to encode integer arguments into reverse byte-order arguments. Instead, we use the Prolog primitives, i.e. integers for values and atoms for variable identifiers.

Figure 7 shows a module that is generated from the bytecode. The interpreter fetches the instruction located at the current program counter, executes it and

[2] While, again, interoperability with C allows embedding of such data structures, standard library predicates usually only offer logarithmic access.

```
bytecode(0, 20, 1).         % push integer 1 on the stack
bytecode(2, 45, val).       % pop value from stack, store in val
bytecode(7, 40, exponent).  % push value of exponent
bytecode(12, 20, 0).        % push constant 0
bytecode(14, 255, []).      % greater-than comparison
bytecode(15, 11, 54).       % jump-if-false to location 55 (exit loop)
bytecode(20, 40, val).      % push value of val
bytecode(25, 40, base).     % push value of base
bytecode(30, 198, []).      % multiplication of arguments on the stack
bytecode(31, 45, val).      % store result in val
bytecode(36, 40, exponent). % load exponent
bytecode(41, 20, 1).        % push constant 1
bytecode(43, 199, []).      % subtract arguments on stack
bytecode(44, 45, exponent). % store result in exponent
bytecode(49, 10, 7).        % jump to beginning of loop
bytecode(54, 0, []).        % terminate instruction
```

Fig. 7: Bytecode as Prolog facts

```
fact_int(PC, Objspace, Env, Stack, REnv) :-
    generated:bc(PC, Instr, Args), % fetch the instruction
    fact_int(Instr, Args, PC, Stack, Env, Objspace, REnv).
fact_int(200, _Args, PC, [Y, X|Stack], Env, Objspace, REnv) :-
    Objspace:add(X, Y, Res), NewPC is PC + 1,
    fact_int(NewPC, Objspace, Env, [Res|Stack], REnv).
% fact_int also has implementations of all the other bytecodes...
```

Fig. 8: Dispatching in the facts-based interpreter

increments the program counter accordingly. This is repeated until it encounters a special zero instruction denoting the end of the bytecode – here at location 54.

The dispatching mechanism is shown in Fig. 8. Similar to an interpreter in C, every opcode has an implementation in Prolog that calls into the object space. Any rule of `fact_int` is equivalent to a `case` statement in C.

Sub-Bytecodes Another design is based on the idea that a program is executed *block-wise*, i.e. a series of instructions that is guarenteed to be executed in this specific order. This is very simple since Acol does not include a goto-statement that allows arbitrary jumps. From a programmer's point of view, blocks are the bodies of while-loops or those of if-then-else statements.

Instead of linearising the entire bytecode, only a block is linearised at once. In order to deal with blocks that are contained by another block (e.g. nested loops), two special opcodes are added. They are used to suspend the execution of the current block and look up the *sub-bytecodes* of the contained blocks that are referenced via its arguments. After those sub-bytecodes are executed, the execution of the previous bytecode is resumed.

The special if-opcode references the blocks of the corresponding then- and else-branches. After the condition is evaluated, only the required block is looked up and executed. The other special opcode for while-loops references the bytecode

```
[20, 1, 45, val, % val = 1
 2, 0, 1]        % while (condition encoded in sub-bytecode 0,
                 %        body encoded in sub-bytecode 1)

% Sub-bytecodes
sbc(0, [40, exponent, 20, 0, 255]).
sbc(1, [40, val, 40, base, 198, 45, val, 40, exponent, 20, 1, 199]).
```

Fig. 9: Bytecode with sub-bytecodes

```
bc_int([], Env, Stack, _Objspace, Env, Stack).
bc_int([H|R], Env, Stack, Objspace, REnv, RStack) :-
    bc_int2(H,R, Env, Stack, Objspace, REnv, RStack).
% special bytecodes for evaluating blocks of an if-statement
bc_int2(1, [T, E|R], Env, [Cond|Stack], Objspace, REnv, RStack) :-
    (Cond == true -> subbytecodes:sbc(T, Then),
                     h_bc_int(Then, [], Env, Objspace, TEnv)
                  ;  subbytecodes:sbc(E, Else),
                     h_bc_int(Else, [], Env, Objspace, TEnv)),!,
    bc_int(R, TEnv, Stack, Objspace, REnv, RStack).
% special bytecodes for evaluating blocks of a while-loop
bc_int2(2, [C, I|R], Env, Stack, Objspace, REnv, RStack) :-
    subbytecodes:sbc(C, Cond),
    bc_int(Cond, Env, [], Objspace, Env, [Res]),
    (Res == true -> subbytecodes:sbc(I, Instr),
                    h_bc_int(Instr, [], Env, Objspace, T),!,
                    bc_int2(2, [C, I|R], T, Stack, Objspace, REnv, RStack)
                 ;  !, bc_int(R, Env, Stack, Objspace, REnv, RStack)).

bc_int2(200, R, Env, [Y, X|Stack], Objspace, REnv, RStack) :-
    Objspace:add(X, Y, Res),!,
    bc_int(R, Env, [Res|Stack], Objspace, REnv, RStack).
% bc_int2 also has implementations of all the other bytecodes...
```

Fig. 10: Dispatching on bytecodes with sub-bytecodes

of the condition that is expected to leave true or false on the stack, as well as the body of the loop. The blocks corresponding to condition and body are evaluated in turn until the condition does not hold any more, so the execution of its parent block can continue. Similar to the facts in the interpreter above, the sub-bytecodes are asserted into their own module to allow fast lookups.

Figure 9 shows an example that includes the special opcode for the while-statement, and Fig. 10 shows an excerpt of the dispatching logic used for this interpreter. The recursion in `bc_int2` will update the bytecode-list with its tail instead of manipulating a program counter. Hence, in this implementation, the interpreter can only move forward inside of a block. If it is required to move backwards in the program, it is only possible to re-start at the beginning of a block.

```
assign(id(val), int(1),
  while(gt(id(exponent), int(0)),
    assign(id(val), mul(id(val), id(base)),
      assign(id(exponent), sub(id(exponent), int(1)),
        while(gt(id(exponent), int(0)),
         ...)))
    end))
```

Fig. 11: Rational tree representation

3.3 Rational Trees

Based on [4], we have created implementations of an AST- and a bytecode-interpreter for Acol that use the idea of rational trees to represent the program being evaluated. This technique aims to improve the performance of jumps by using recursive data structures containing references to the following instructions.

AST-Interpreter with Rational Trees Since Acol does not include a concept of arbitrary jumps as used in [4], it is not possible to achieve the speed-up described in the referenced paper. However, we can make use of the basic idea for the representation of programs: every statement has a pointer to its successor statement.

In our naive AST interpreter, a new Prolog stack frame is used for every level of nested loops and if-statements. Instead of returning from each evaluation to the predicate that dispatched to the sub-statement, we can make use of Prolog's tail-recursion optimisation and continue with the next statements directly.

For our example program, we generate an infinite data structure for the while-loop depicted in Fig. 11. The concept of rational trees allows us to have the `while`-term re-appearing in its own body, so it has not to be saved in a stack frame.

The last statement `end` is artificially added to indicate the end of the program so that the interpreter may halt.

Then, the dispatching logic is still very similar to the naive AST interpreter as shown in Fig. 12.

Bytecode-Interpreter with Rational Trees In Prolog, rational trees can also be used for bytecodes. Jumps are removed from that representation entirely. While-loops are unrolled into an infinite amount of alternated bytecodes of the condition and if-statements that contain the body of the loop in their then-branch and the next statement after the loop in their else-branch. An example is shown in Fig. 13.

At first glance, it looks weird that the opcode integers are replaced by human-readable descriptions. However, functors are limited to atoms and, then, there is not much difference between atoms that contain only a number or short readable names. We chose the latter one because they are by far more comprehensible.

```
rt_int(end, Env, _, Env) :- !.
rt_int(assign(id(Var), Expr, Next), Env, Objspace, REnv) :-
    eval(Expr, Env, Objspace, Res),
    Objspace:store(Env, Var, Res, EnvOut), !,
    rt_int(Next, EnvOut, Objspace, REnv).
rt_int(if(Cond, Then, Else), Env, Objspace, REnv) :-
    eval(Cond, Env, Objspace, V),
    (V == true -> !, rt_int(Then, Env, Objspace, REnv)
               ; !, rt_int(Else, Env, Objspace, REnv)).
rt_int(while(Cond, Instrs, Else), Env, Objspace, REnv) :-
    eval(Cond, Env, Objspace, V),
    (V == true -> !, rt_int(Instrs, Env, Objspace, REnv)
               ; !, rt_int(Else, Env, Objspace, REnv)).
```

Fig. 12: Dispatching in a rational tree interpreter

```
push(1, assign(val,                                  % code before the loop
  load(exponent, push(0, gt(                         % condition (1)
    if(load(val, load(base, mul(store(val,           % while-body (1)
         load(exponent, push(1, sub(store(exponent,  % while-body (1)
           load(exponent, push(0, gt(                % condition (2)
             if(load(val, load(base(, ....))),       % while-body (2)
                end)))))))))))                       % end of while (2)
       end))))))                                     % end of while (1)
```

Fig. 13: Bytecode with rational trees

The dispatching is pretty similar to the AST interpreter that utilises rational trees, as shown in Fig. 14. The main difference between those two interpreters is that this one uses a simulated stack to evaluate terms instead of Prolog's call stack.

4 Evaluation

To compare the performance of the different interpreters for Acol, we selected a set of different benchmarks. Because the language is very limited, it is hard to design "real-world programs". Yet, execution of any arbitrary program will give insight of the performance of the dispatching logic.

In this section, we present those benchmarks and compare their results. Each program was executed with every interpreter ten times. The runtime consists only of the time spent in the interpreter. Compilation time is excluded, as it is not implemented efficiently and, ultimately, not relevant.

The benchmarks were run on a machine that runs a linux with a 4.15.0-108-generic 64-bit kernel on an Intel i7-7700HQ CPU @ 2.80GHz. No benchmarks ran in parallel. Two Prolog implementations were considered: SICStus Prolog 4.6.0, a commercial product, and SWI-Prolog 8.2.1, a free open-source implementation. All C code was compiled by gcc 7.5.0 with the `-O3`-flag.

Since Acol does not offer complex features, we expect that the dispatching claims a bigger share of the runtime than the actual operations.

```
rt_bc_int(end, Env, Stack, _Objspace, Env, Stack).
rt_bc_int(if(Then, Else), Env, [X|Stack], Objspace, REnv, RStack) :-
    (X == true -> !, rt_bc_int(Then, Env, Stack, Objspace, REnv, RStack)
              ; !, rt_bc_int(Else, Env, Stack, Objspace, REnv, RStack)).
rt_bc_int(push(Arg, Next), Env, Stack, Objspace, REnv, RStack) :-
    Objspace:create_integer(Arg, Val),!,
    rt_bc_int(Next, Env, [Val|Stack], Objspace, REnv, RStack).
rt_bc_int(load(Arg, Next), Env, Stack, Objspace, REnv, RStack) :-
    Objspace:lookup(Arg, Env, Val), !,
    rt_bc_int(Next, Env, [Val|Stack], Objspace, REnv, RStack).
rt_bc_int(add(Next), Env, [Y, X|Stack], Objspace, REnv, RStack) :-
    Objspace:add(X, Y, Res), !,
    rt_bc_int(Next, Env, [Res|Stack], Objspace, REnv, RStack).
% rt_bc_int implements all other opcodes as well...
```

Fig. 14: Dispatching in a bytecode interpreter with rational trees

```
while (start < V) {
    if (V mod start == 0) {
        is_prime := 0;
    } else {
        is_prime := is_prime;
    }
    start := start + 1;
}
```

Fig. 15: Prime Tester Program

4.1 Benchmarks

Prime Tester The first benchmark is a naive prime tester. The program is depicted in Fig. 15. The environment was pre-initialised with `is_prime` := 1, `start` := 2, and `V` := 34 265 341.

Fibonacci Another benchmark is the calculation of the fibonacci sequence. However, we expect that most of the execution time will consist of the addition and subtraction of two large numbers and that the interpreter overhead itself is rather small. Therefore, a second version that calculates the sequence modulo 1 000 000 is included.

Again, the environment is pre-initialised, in this case with `a` := 0, `b` := 1 and `n` := 400 000. To ensure a significant runtime for the second version, the input is modified so it calculates a longer sequence, i.e. `n` := 10 000 000.

Generated ASTs Lastly, some programs were generated pseudo-randomly. Such a generated AST consists of 20 to 50 statements that are uniformly chosen from while-loops, if-statements and assignments. The body of a loop and both branches of if-statements also consist of 20 to 50 statements. However, if the nesting exceeds a certain depth, only assignments are generated for this block.

In order to guarentee termination, while-loops are always executed 20 times. An assignment is artifically inserted before the loop that resets a loop counter,

```
i := 1;                      i := 1;
while i < n {                while i < n {
    b := b + a;                  b := b + a mod 1000000;
    a := b - a;                  a := b - a mod 1000000;
    i := i + 1;                  i := i + 1;
}                            }
```

Fig. 16: Fibonacci Programs

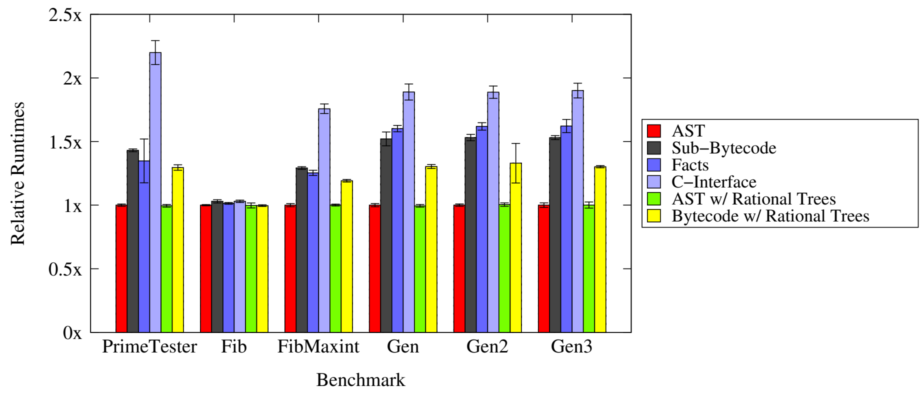

Fig. 17: Relative runtimes in SICStus, normalised to the runtime of the AST interpreter

as well as another assignment that increments this variable at the beginning of the loop.

For assignments and if-conditions, a small subtree is generated. The generator chooses uniformly between five predetermined identifiers, constants ranging from -1 to 3, as well as additions and subtractions. If-conditions have to include exactly one comparison operator.

The generator does include neither multiplications, because they caused very large integers that slowed down the Prolog execution time significantly, nor modulo operations, to avoid division by zero errors.

Three different benchmarks were generated using arbitrary seeds. Their purpose is to complement the other three handwritten benchmarks, which are rather small and might benefit from caching of the entire AST.

4.2 Results

The results of the benchmarks are shown in Table 2. The lines labelled "AST" refer to the implementation of the naive AST interpreter presented in Section 3.1, the ones with "Sub-Bytecodes", "Facts" and "C-Interface" refer to the corresponding bytecode interpreters discussed in Section 3.2. Finally, "AST-" and "BC w/ Rational Trees" are the AST and bytecode interpreters based on rational trees presented in Section 3.3. The mean value is determined by the geometric mean

Table 2: Mean runtimes in seconds including the 0.95 confidence interval. The value in parentheses describes the normalised runtime (on the basis of the AST interpreter). The fastest runtimes per benchmark and interpreter are highlighted.

Benchmark	Interpreter	SICStus	SWI-Prolog
Prime Tester	AST	54.53 ± 0.49 (1.00)	242.84 ± 3.96 (1.00)
	Sub-Bytecodes	78.03 ± 0.59 (1.43)	316.44 ± 1.89 (1.30)
	Facts	73.50 ± 9.39 (1.35)	330.89 ± 2.51 (1.36)
	C-Interface	119.94 ± 5.13 (2.20)	**54.83 ± 0.98 (0.23)**
	AST w/ Rational Trees	**54.26 ± 0.57 (1.00)**	229.65 ± 2.28 (0.95)
	BC w/ Rational Trees	70.65 ± 1.19 (1.30)	261.39 ± 5.58 (1.08)
Fibonacci	AST	9.86 ± 0.05 (1.00)	5.54 ± 0.09 (1.00)
	Sub-Bytecodes	10.16 ± 0.13 (1.03)	6.36 ± 0.15 (1.15)
	Facts	10.00 ± 0.07 (1.01)	6.49 ± 0.09 (1.17)
	C-Interface	10.16 ± 0.09 (1.03)	**2.58 ± 0.06 (0.47)**
	AST w/ Rational Trees	9.84 ± 0.19 (1.00)	5.39 ± 0.09 (0.97)
	BC w/ Rational Trees	**9.83 ± 0.06 (1.00)**	5.63 ± 0.10 (1.02)
Fibonacci (Maxint)	AST	**24.03 ± 0.30 (1.00)**	96.90 ± 1.72 (1.00)
	Sub-Bytecodes	31.03 ± 0.26 (1.29)	122.86 ± 1.54 (1.27)
	Facts	30.13 ± 0.48 (1.25)	131.38 ± 3.68 (1.36)
	C-Interface	42.23 ± 0.90 (1.76)	**22.65 ± 0.48 (0.23)**
	AST w/ Rational Trees	24.05 ± 0.17 (1.00)	94.36 ± 1.15 (0.97)
	BC w/ Rational Trees	28.63 ± 0.25 (1.19)	103.93 ± 0.70 (1.07)
Generated	AST	12.96 ± 0.16 (1.00)	60.64 ± 0.59 (1.00)
	Sub-Bytecodes	19.71 ± 0.70 (1.52)	69.26 ± 0.49 (1.14)
	Facts	20.76 ± 0.33 (1.60)	73.89 ± 0.45 (1.22)
	C-Interface	24.49 ± 0.82 (1.89)	**10.30 ± 0.18 (0.17)**
	AST w/ Rational Trees	**12.89 ± 0.14 (0.99)**	58.35 ± 1.45 (0.96)
	BC w/ Rational Trees	16.90 ± 0.20 (1.30)	61.98 ± 1.16 (1.02)
Generated2	AST	**19.01 ± 0.18 (1.00)**	83.18 ± 0.80 (1.00)
	Sub-Bytecodes	29.11 ± 0.48 (1.53)	102.17 ± 1.48 (1.23)
	Facts	30.78 ± 0.55 (1.62)	109.08 ± 2.35 (1.31)
	C-Interface	35.89 ± 0.92 (1.89)	**15.07 ± 0.24 (0.18)**
	AST w/ Rational Trees	19.10 ± 0.26 (1.00)	81.05 ± 0.39 (0.97)
	BC w/ Rational Trees	25.28 ± 2.96 (1.33)	90.44 ± 1.21 (1.09)
Generated3	AST	**12.37 ± 0.22 (1.00)**	55.52 ± 1.34 (1.00)
	Sub-Bytecodes	18.93 ± 0.20 (1.53)	66.00 ± 0.76 (1.19)
	Facts	20.06 ± 0.64 (1.62)	70.49 ± 0.45 (1.27)
	C-Interface	23.51 ± 0.71 (1.90)	**9.79 ± 0.22 (0.18)**
	AST w/ Rational Trees	12.38 ± 0.30 (1.00)	52.96 ± 1.12 (0.95)
	BC w/ Rational Trees	16.10 ± 0.11 (1.30)	60.07 ± 0.97 (1.08)

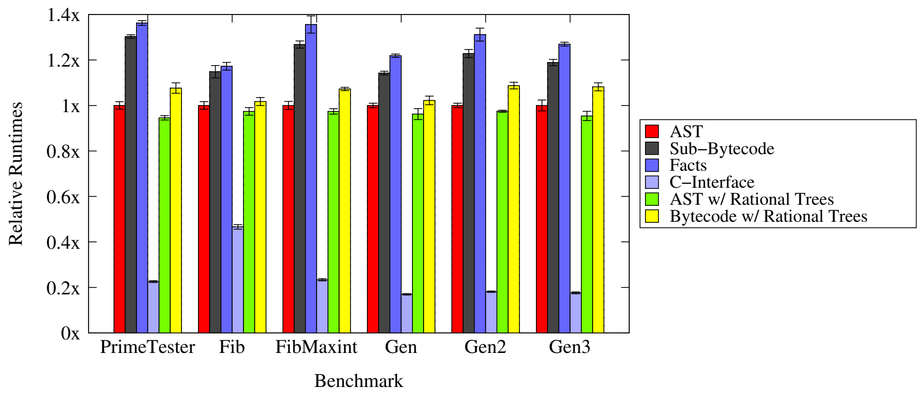

Fig. 18: Relative runtimes in SWI-Prolog, normalised to the runtime of the AST interpreter

as proposed by [5]. Though not listed, the interpreter purely written in C is 2-3 orders of magnitudes faster[3] and executes each benchmark in less than a second.

Figure 17 shows the results specific for SICStus Prolog. There is no discernible performance difference between the naive AST interpreter and the ones utilising rational trees. Independent of the benchmark, the bytecode interpreters based on sub-bytecodes and on Prolog facts are slow in comparison. One can observe a performance loss of about 25-35 % for the small handwritten programs, where we would expect caching effects to be the largest, and around 50-60% for the larger, generated programs. With our initial version of the interpreter dispatching in C, we reported an issue that was related with SICStus' FLI garbage collector. Now, it usually requires twice as much time to execute the benchmarks compared to the AST-based interpreters.

The results utilising SWI-Prolog are shown in Fig. 18. Overall, they paint a similar picture to the results for SICStus. However, the dispatching using SWI-Prolog's C-interface is very fast – compared to the AST interpreter, it can achieve more than a 5× speed-up.

5 Conclusions, Related and Future Work

In this paper, we presented the language Acol and multiple ways to implement it as AST as well as bytecode interpreters. We designed several benchmarks in order to evaluate their performance using different implementations of Prolog.

Our results suggest that if an interpreter is to be implemented in Prolog, the implementation as an AST interpreter already is very performant. It is simply not worth the hassle of writing and maintaining a bytecode compiler. Furthermore,

[3] A fair comparison is not possible since the C interpreter does not support unbounded integer values.

an AST interpreter does not involve any additional compilation overhead as it can directly work on the data structure returned by the parser.

However, SWI-Prolog's C interface performs very well. Surprisinly, even on the Fibonacci example with unlimited integers, where addition of unlimited integers is rather time-consuming, it beats the run-time of the AST interpreter by a factor of two. Additional work is required to determine whether these findings are applicable for more complex languages, that would also facilitate the creation of more sensible benchmarks.

Rossi and Sivalingam explored dispatching techniques in C based bytecode interpreters [9], with the result that a less portable approach of composing the code in memory before executing it yielded the best results. The techniques discussed in that article could be used in combination with SWI-Prolog to further improve the instruction dispatching performance in C.

An alternative for improving the execution time of a program, that was not discussed here, is partial evaluation [6]. We intend to investigate the impact of offline partial evaluation when compiling a subset of the described interpreters for our benchmarks.

In the future, it would also be interesting to evaluate the effects of different interpreter designs in other Prolog dialects, especially those that are not based on the WAM [2]. Examples include Ciao (WAM-based with powerful analysis), BinProlog (specialised version of the WAM) and Mercury (functional influences with many optimisations).

References

1. Jean-Raymond Abrial. *The B-Book*. Cambridge University Press, 1996.
2. Hassan Aït-Kaci. Warrens abstract machine - a tutorial reconstruction, 1999.
3. Mats Carlsson and Per Mildner. SICStus Prolog – the first 25 years. *Theory and Practice of Logic Programming*, 12(1-2):35–66, 2012.
4. Manuel Carro. An Application of Rational Trees in a Logic Programming Interpreter for a Procedural Language. *CoRR*, cs.DS/0403028, March 2004.
5. Philip J. Fleming and John J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, March 1986.
6. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
7. Michael Leuschel and Michael J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.
8. The PyPy Project. The Object Space. http://pypy.readthedocs.org/en/latest/objspace.html, 2015. Accessed: 2020-08-12.
9. M Rossi and K Sivalingam. A survey of instruction dispatch techniques for byte-code interpreters. *Seminar on Mobile Code*, 1996.
10. Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. Virtual machine showdown: Stack versus registers. *Transactions on Architecture and Code Optimization*, 4(4):1–36, 2008.
11. David H D Warren. An Abstract Prolog Instruction Set. Technical report, Artificial Intelligence Center - SRI International, 1983.
12. Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. Swi-prolog. *Theory and Practice of Logic Programming*, 12(1-2):6796, 2012.