Evaluating Interpreter Design in Prolog

Philipp Körner, David Schneider and Michael Leuschel

Institut für Informatik, Heinrich Heine University Düsseldorf, Germany
{p.koerner, david.schneider}@hhu.de, leuschel@cs.uni-duesseldorf.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. The data structures and language semantics are a natural match to the evaluatation of programs, in particular if those are represented as trees. 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 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 for any language implemented in Prolog. In particular, we have developed such an interpreter [4] for the full B language and wanted 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 for implementing the dispatching logic, taking advantage of available data structures and programming paradigms that might be available in higher-level languages.

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

if 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 described in section 2. For ACOL we have created several implementations described in section 3, that use different paradigms for the dispatching logic. Finally, in section 4, we present a set of benchmarks written in ACOL used to evaluate the implemented interpreters when running on SICStus and SWI Prolog.

2 A Simple Language

As a means to evaluate the different interpreter designs described in section 3, 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 is shown in fig. 1.

```
# 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: A small program

3 Interpreter Implementations

There are many ways to implement ACOL, in C as well as in Prolog. Considering several different interpreter implementation techniques, in this section we will describe possible designs of interpreters and the closely related representations of the ACOL programs. 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.

All interpreters share the same implementation of the language semantics exposed by an object-space API [5]. In order to keep the implementations

[1] ACOL is not a backronym for ACOL is a computable language

simple and compatible, they all call into the same object space. Nonetheless, the interpreters differ very much in the representation of the program and, hence, in the process of dispatching.

In order to discuss the differences, we will translate a 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.

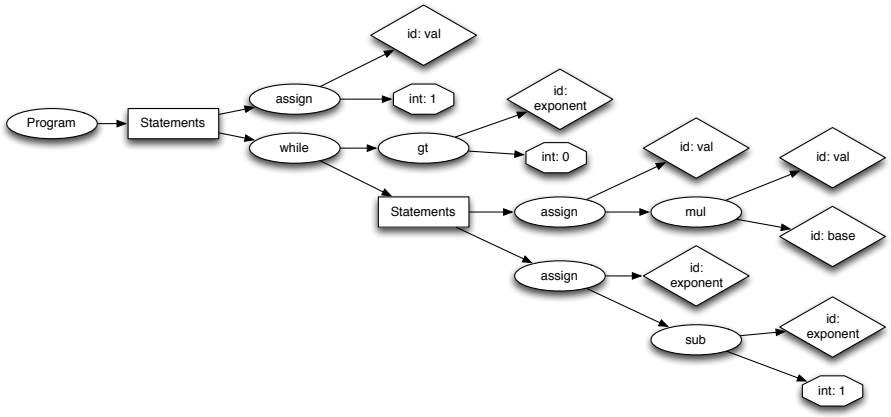

Fig. 2: AST

3.1 AST Interpreter

The most natural 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, the AST (abstract syntax tree). 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 tree. Block statements, i.e. `if` and `while`, 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 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 [7] 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.

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. An example for a bytecode based on the program above is shown in fig. 5.

#	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

Table 1: A bytecode for the described language

```
unsigned
char bc[] = {20, 1,           // push integer 1 on the stack
             45, 2, 0, 0, 0,  // store it in variable at index 2
                              // (i.e. val)
             40, 1, 0, 0, 0,  // load the variable at index 1
                              // (i.e. exponent)
             20, 0,           // push 0
             255,             // greater than
             11, 54, 0, 0, 0, // jump behind loop
                              // if condition is falsey
             40, 2, 0, 0, 0,  // load val
             40, 0, 0, 0, 0,  // load base
             198,             // mul
             45, 2, 0, 0, 0,  // store val
             40, 1, 0, 0, 0,  // load exponent
             20, 1,           // push 1
             199,             // sub
             45, 1, 0, 0, 0,  // store exponent
             10, 7, 0, 0, 0,  // jump to beginning of loop
             0}               // terminate
```

Fig. 5: Example bytecode 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:
            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. 6: 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. 7: Dispatching logic using SWI's C-Interface

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. 6. 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) 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. 7.

Then, the main loop dispatches in C, but the objects on the evaluation stack are created and the operations are executed by Prolog predicates.

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

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 $\mathcal{O}(n)$. However, 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 performing 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.

```
bytecode(0, 20, 1).
bytecode(2, 45, val).
bytecode(7, 40, exponent).
bytecode(12, 20, 0).
bytecode(14, 255, []).
bytecode(15, 11, 55).
bytecode(20, 40, val).
bytecode(25, 40, base).
bytecode(30, 198, []).
bytecode(31, 45, val).
bytecode(36, 40, exponent).
bytecode(41, 20, 1).
bytecode(43, 199, []).
bytecode(44, 45, exponent).
bytecode(49, 10, 7).
bytecode(54, 0, []).
```

Fig. 8: Bytecode as Prolog facts

Figure 8 shows a module that is generated from the bytecode. The interpreter fetches the instruction located at the current program counter, executes it and increments the program counter accordingly. This is repeated until it encounters a special zero instruction that denotes the end of the bytecode.

The dispatching mechanism is shown in fig. 9. 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 body 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

```
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. 9: Dispatching in the facts-based interpreter

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 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 10 shows an example that includes the special opcode for the while-statement.

```
[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. 10: Bytecode with sub-bytecodes

Figure 11 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.

3.3 Rational Trees

Based on [1], we have created implementations of an AST- and a bytecode-interpreter for Acol that use the idea of rational trees to represent the program

```
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. 11: Dispatching on bytecodes with sub-bytecodes

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 [1], 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.

```
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. 12: Rational tree representation

For our example program, we generate an infinite data structure for the while-loop depicted in fig. 12. 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. 13.

```
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. 13: Dispatching in a rational tree interpreter

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. 14.

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.

```
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. 14: Bytecode with rational trees

```
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. 15: Dispatching in a bytecode interpreter with rational trees

The dispatching is pretty similar to the AST interpreter that utilises rational trees, as shown in fig. 15. 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".

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, the compilation time is excluded.

The benchmarks were run on a machine that runs a linux with a 3.19.0-25-generic 64-bit kernel on an Intel i5-2400 CPU @ 3.10GHz. Two Prolog implementations were considered: SICStus Prolog 4.3.2, a commercial product, and SWI Prolog 7.2.2, a free open-source implementation. All C code was compiled by gcc 4.9.2. 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.

4.1 Benchmarks

Prime Tester The first benchmark is a naive prime tester. The program is depicted in fig. 16. 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 big numbers and that the interpreter overhead itself is rather

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

Fig. 16: Prime Tester Program

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

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

Fig. 17: Fibonacci Programs

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 $\texttt{a} \coloneqq 0$, $\texttt{b} \coloneqq 1$ and $\texttt{n} \coloneqq 400\,000$. To ensure a significant runtime for the second version, the input is modified so it calculates a longer sequence, i.e. $\texttt{n} \coloneqq 10\,000\,000$.

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, 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.

Benchmark	Interpreter	SICSTus	SWI
Prime Tester	AST	66.46 ± 0.32 (1.00)	438.05 ± 6.32 (1.00)
	Sub-Bytecodes	85.11 ± 0.45 (1.28)	565.73 ± 27.28 (1.29)
	Facts	96.77 ± 1.82 (1.46)	537.64 ± 18.03 (1.23)
	C-Interface	183.58 ± 2.85 (2.76)	82.11 ± 1.77 (0.19)
	AST w/ Rational Trees	67.21 ± 0.64 (1.01)	426.76 ± 20.79 (0.97)
	BC w/ Rational Trees	78.32 ± 0.86 (1.18)	464.41 ± 23.69 (1.06)
Fibonacci	AST	9.99 ± 0.20 (1.00)	10.49 ± 0.32 (1.00)
	Sub-Bytecodes	10.11 ± 0.05 (1.01)	11.91 ± 0.26 (1.14)
	Facts	10.47 ± 0.07 (1.05)	11.63 ± 0.35 (1.11)
	C-Interface	10.57 ± 0.07 (1.06)	3.86 ± 0.05 (0.37)
	AST w/ Rational Trees	10.16 ± 0.10 (1.02)	10.39 ± 0.27 (0.99)
	BC w/ Rational Trees	10.11 ± 0.06 (1.01)	10.72 ± 0.32 (1.02)
Fibonacci (Maxint)	AST	27.86 ± 0.52 (1.00)	191.46 ± 2.87 (1.00)
	Sub-Bytecodes	35.10 ± 0.31 (1.26)	231.86 ± 10.55 (1.21)
	Facts	41.42 ± 2.08 (1.49)	227.26 ± 9.98 (1.19)
	C-Interface	61.63 ± 1.45 (2.21)	32.19 ± 0.72 (0.17)
	AST w/ Rational Trees	28.62 ± 0.88 (1.03)	187.49 ± 7.31 (0.98)
	BC w/ Rational Trees	33.23 ± 1.01 (1.19)	200.34 ± 5.38 (1.05)
Generated	AST	16.53 ± 0.02 (1.00)	131.51 ± 3.86 (1.00)
	Sub-Bytecodes	24.89 ± 0.19 (1.51)	144.96 ± 2.86 (1.10)
	Facts	26.00 ± 0.93 (1.57)	140.42 ± 3.19 (1.07)
	C-Interface	NA	15.06 ± 0.29 (0.11)
	AST w/ Rational Trees	16.88 ± 0.04 (1.02)	129.03 ± 5.29 (0.98)
	BC w/ Rational Trees	20.41 ± 0.10 (1.23)	139.36 ± 6.72 (1.06)
Generated2	AST	24.31 ± 0.12 (1.00)	199.75 ± 5.85 (1.00)
	Sub-Bytecodes	37.26 ± 0.30 (1.53)	211.67 ± 5.09 (1.06)
	Facts	38.45 ± 1.36 (1.58)	204.97 ± 7.27 (1.03)
	C-Interface	NA	22.65 ± 0.72 (0.11)
	AST w/ Rational Trees	25.02 ± 0.06 (1.03)	193.39 ± 5.42 (0.97)
	BC w/ Rational Trees	30.20 ± 0.09 (1.24)	220.79 ± 7.53 (1.11)
Generated3	AST	15.98 ± 0.04 (1.00)	124.97 ± 3.00 (1.00)
	Sub-Bytecodes	24.14 ± 0.23 (1.51)	136.97 ± 3.92 (1.10)
	Facts	27.93 ± 0.95 (1.75)	133.31 ± 4.39 (1.07)
	C-Interface	NA	14.47 ± 0.43 (0.12)
	AST w/ Rational Trees	16.04 ± 0.03 (1.00)	121.68 ± 3.85 (0.97)
	BC w/ Rational Trees	19.43 ± 0.08 (1.22)	128.45 ± 3.90 (1.03)

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

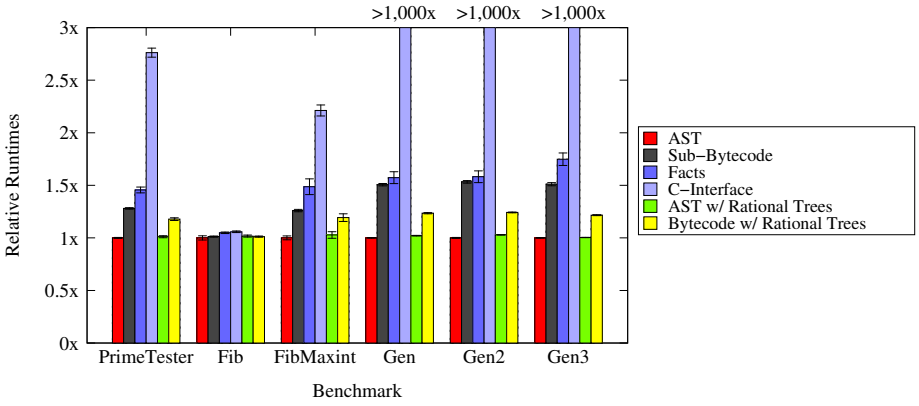

Fig. 18: Relative runtimes in SICStus, normalised to the runtime of the AST interpreter

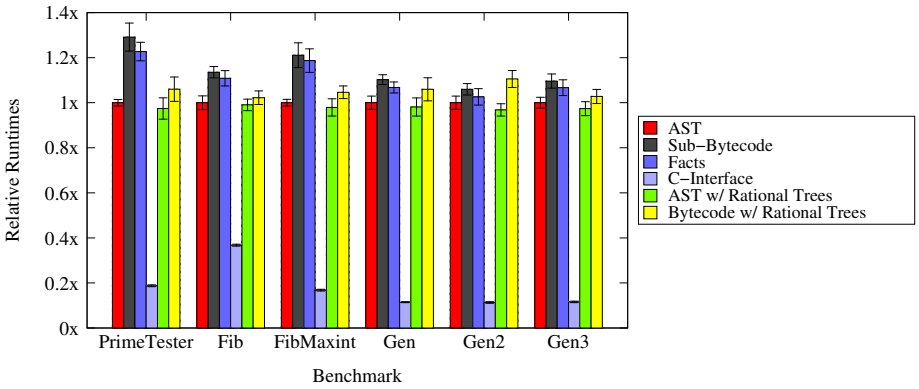

Fig. 19: Relative runtimes in SWI, normalised to the runtime of the AST interpreter

4.2 Results

The results of the benchmarks are shown in table 2. The mean value is determined by the geometric mean as proposed by [2]. For the interpreter based on SICStus' C-Interface, we cancelled the runs of the generated benchmarks after eight hours without a result.

The most important result is that using either Prolog implementation, the naive AST interpreter outperforms all of our pure Prolog implementations of bytecode interpreters.

Figure 18 shows the results specific for SICStus Prolog. Independent of the benchmark, all bytecode interpreters based on sub-bytecodes and on Prolog facts are slow in comparison. The AST interpreter utilising rational trees performs about as well as the naive AST interpreter. Surprisingly, the interpreter that dispatches in C suffers heavy performance issues. At the time of writing, we do

not understand the reasons but are in contact with the SICStus support to clarify this behaviour.

The results utilising SWI Prolog are shown in fig. 19. In comparison to any bytecode-interpreter, the AST interpreter is faster. The AST interpreter with rational trees seems to be slightly more performant. However, the difference is small enough to be included in the uncertainty of measurement. The dispatching in C, however, is very fast. Depending on the benchmark, it can achieve a speed-up by an order of magnitude.

5 Conclusion, 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 is very performant. Furthermore, it does not involve any compilation overhead as it can work on the data structure returned by the parser. Moreover, when using SWI Prolog, one can utilise C to efficiently implement the dispatching and query Prolog predicates for the domain logic.

In [6], Rossi and Sivalingam explored dispatching techniques in C based bytecode interpreters, 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 [6] could be used in combination with SWI 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 [3]. We intend to investigate the impact of offline partial evaluation when compiling a subset of the described interpreters for our benchmarks.

However, ACOL is a very simple language. Additional work is required to determine whether these findings are applicable for more complex languages. Furthermore, a richer language facilitates the creation of more benchmarks.

References

1. Manuel Carro. An Application of Rational Trees in a Logic Programming Interpreter for a Procedural Language. *CoRR*, cs.DS/0403028, March 2004.
2. 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.
3. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
4. Michael Leuschel and Michael J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.
5. The PyPy Project. The Object Space, 2015.
6. M Rossi and K Sivalingam. A survey of instruction dispatch techniques for byte-code interpreters. *Seminar on Mobile Code*, 1996.

7. David H D Warren. An Abstract Prolog Instruction Set. Technical report, Artificial Intelligence Center - SRI International, 1983.