Towards a Jitting VM for Prolog Execution

Carl Friedrich Bolz
cfbolz@gmx.de

Michael Leuschel
leuschel@cs.uni-duesseldorf.de

David Schneider
david.schneider@uni-duesseldorf.de

Heinrich-Heine-Universität Düsseldorf, STUPS Group, Germany

ABSTRACT

Most Prolog implementations are implemented in low-level languages such as C and are based on a variation of the WAM instruction set, which enhances their performance but makes them hard to write. We present a high-level continuation-based Prolog interpreter written in RPython, a restricted subset of Python. This interpreter is annotated with hints, so that it can be fed through the PyPy tracing JIT generator, which incorporates partial evaluation techniques. The resulting Prolog implementation is surprisingly efficient: it clearly outperforms existing implementations of Prolog in high-level languages such as Java. Moreover, on some benchmarks, our system outperforms state-of-the-art WAM-based Prolog implementations. Our paper tries to show that PyPy can indeed form the basis for implementing programming languages other than Python. Furthermore, we believe that our results showcase the great potential of the tracing JIT approach for declarative programming languages such as Prolog.[1]

Categories and Subject Descriptors

D.1.6 [**Programming Techniques**]: Logic Programming; D.3.4 [**Programming Languages**]: Processors—*code generation, interpreters, run-time environments*

Keywords

Logic Programming; JIT; Partial Evaluation; Interpreters

1. INTRODUCTION

JITs and in particular tracing JITs have started to be very successful, often outperforming traditional ahead-of-time compilers. They have been particularly successful for dynamic languages, such as Python or JavaScript. We try to show that this approach also holds great promise for a language like Prolog (which has many dynamic features). We also believe that techniques such as partial evaluation can naturally be integrated into a JIT, leading to increased applicability and performance potential [6]. In previous work [5], we have developed a tracing JIT compiler for RPython which integrates partial evaluation techniques. This JIT is part of the larger PyPy project [28]. One of the goals of PyPy was to provide an environment for writing interpreters for dynamic languages, which get then compiled at runtime using the PyPy JIT, thus combining flexibility with performance.

This paper sets out to answer the following questions:

- Can the PyPy JIT be applied to interpreters for other dynamic languages, such as Prolog?
- Can a high-level implementation of Prolog be competitive with a finely-tuned WAM-based implementation?
- Could Prolog implementations benefit from a JIT or tracing JIT?

The answers to these questions are affirmative. We will present an interpreter for Prolog written in RPython, which can be compiled to C and which is specialized at runtime using the PyPy JIT generator. The performance results are surprisingly good: we clearly outperform other Prolog implementations written in higher-level languages (e.g. Java). More importantly, however, we are faster than state-of-the-art Prolog VMs on specific benchmarks. We believe that this shows the potential of the JIT approach for Prolog systems, and that future Prolog implementations should consider integrating a JIT compiler.

In Section 2 we introduce the PyPy project and its technologies which we use as the basis of our Prolog implementation. The details of this implementation, its object model and continuation-based execution are described in Section 3. How the JIT generator developed by the PyPy project is applied to the Prolog interpreter is presented in Section 4, also we discuss the optimizations performed by the generated JIT and how they apply to Prolog. In Section 5 we present and analyze a number of benchmarks measuring performance and memory consumption and compare it to a series of other Prolog implementations. Related work is discussed in Section 6.

[1]This research is partially supported by the BMBF funded project PyJIT (nr. 01QE0913B; Eureka Eurostars).

2. BACKGROUND

2.1 Prolog Implementations

Most high-performance implementations of the Prolog language in common use today are implemented using an extension of the WAM [30]. The WAM is a great instruction set that made high-speed Prolog execution possible. However, it is also a very low-level instruction set that is predominantly useful when implementing in a low-level language operating close to the machine level. Apart from WAM-based approaches, there are a number of Prolog implementations written in object-oriented high-level languages, such as Java or the .NET VM [27, 9]. These often have flexible and extensible architectures, and integrate well with their host virtual machine, but are typically orders of magnitude slower than low-level VMs.

2.2 PyPy

The PyPy project [28, 7] is an environment where dynamic languages can be implemented in a simple and maintainable, yet efficient, way. Using PyPy, the approach is to write an interpreter for the to-be-implemented language in *RPython*, which is a subset of Python. This interpreter can then be translated into C. During the translation process, various aspects of the final VM will be introduced into the C code automatically, such as an efficient garbage collector, or optionally a just-in-time compiler (see Section 2.5).

Because of these introduced aspects, the interpreter implementation itself is free from low-level details such as memory management and can therefore focus purely on the language semantics and on high-level optimization happening on language level.

PyPy was originally started to be only a Python implementation (hence the name). However, the tools developed in the process turned out to be generally applicable, so that it is now used for the implementation of various dynamic languages, such as Squeak/Smalltalk [7], JavaScript and now Prolog (however, the Python implementation still remains the most mature language implementation in the PyPy context).

2.3 RPython

As mentioned before, the language that is used within PyPy to implement the language interpreters is called *RPython*, *Restricted Python* [1]. It is a subset of the Python language, chosen in such a way that type inference on RPython programs is possible. As its main restriction, in RPython it is not allowed to mix types at the same location in the RPython – e.g., you cannot have a function that accepts both integers and strings as its first argument. In addition, RPython forbids runtime reflection (like changing methods of classes at runtime). Despite the restrictions, RPython is still quite an expressive object-oriented high-level language, supporting garbage collection, exceptions, single inheritance, dynamic dispatch, and good builtin data-structures.

Since type inference can be performed on RPython programs, it is possible to translate them into an efficient C program. The C program can then be turned into an executable and be executed. Many aspects of the final executable are not apparent in the original interpreter. Since RPython has automatic memory management and C does not, a garbage collector needs to be inserted during the translation process.

2.4 Garbage Collection

The most effective garbage collector contained in PyPy is a generational copying collector [22]. Objects are first allocated in a nursery generation. If they survive a minor collection, they move into the mature generation, which is collected only rarely. After they survive several collections there, they are moved to an old object generation, which is collected using mark-and-sweep, to not have to copy long-living objects around all the time. References from older to younger objects are detected with the help of a write barrier.

Since the collector is using copying, allocation is extremely fast, essentially just incrementing and comparing a pointer. The GC is also very efficient at dealing with high allocation rates, as long as most objects die very quickly. The size of the nursery is chosen to be half the size of the level 1 cache of the processor, to improve cache locality.

2.5 JIT Compilers

Another aspect that can be automatically introduced by the PyPy translation toolchain into the final VM is a Just-in-Time compiler. The JIT compiler will be generated by analyzing the RPython interpreter using partial evaluation techniques [5]. This process is mostly automatic but requires a few *hints* by the interpreter's author to guide the process. Those hints are a few lines of annotations added to the interpreter and are mostly needed to identify the main interpreter loop.

Automatically generating a JIT compiler has many advantages: Writing a JIT compiler by hand is a tedious and error-prone task, particularly for complex languages. Also, many dynamic language have similar needs from a JIT compiler (e.g., type specialization, unboxing of boxed objects, dealing with changes to the program at runtime, ...), which makes it worthwhile to implement a JIT compiler generator. PyPy's JIT compiler generator is targeted at imperative object-oriented dynamic languages, and a part of the question posed by this paper is whether it can be successfully applied to a logic programming language at all. The JIT generator is still experimental and in active development, but already stable enough to give useful speedups for PyPy's Python interpreter.

The JIT that is generated by the PyPy toolchain is a *tracing JIT*. Tracing JITs have recently become a rather successful approach to writing JIT compilers for dynamic languages, since they are both easy to implement and can give sizeable performance improvements [17, 8]. As an example, the JavaScript engine of the Firefox web browser uses a trace-based JIT compiler since version 3.1[2] [15]. A tracing JIT tries to only generate assembler code for the commonly executed code paths of a program, as opposed to classical JITs, which typically operate on a per-method basis [14]. For an illustration of the various stages that a tracing JIT goes through see Figure 1.

[2] `https://wiki.mozilla.org/JavaScript:TraceMonkey`

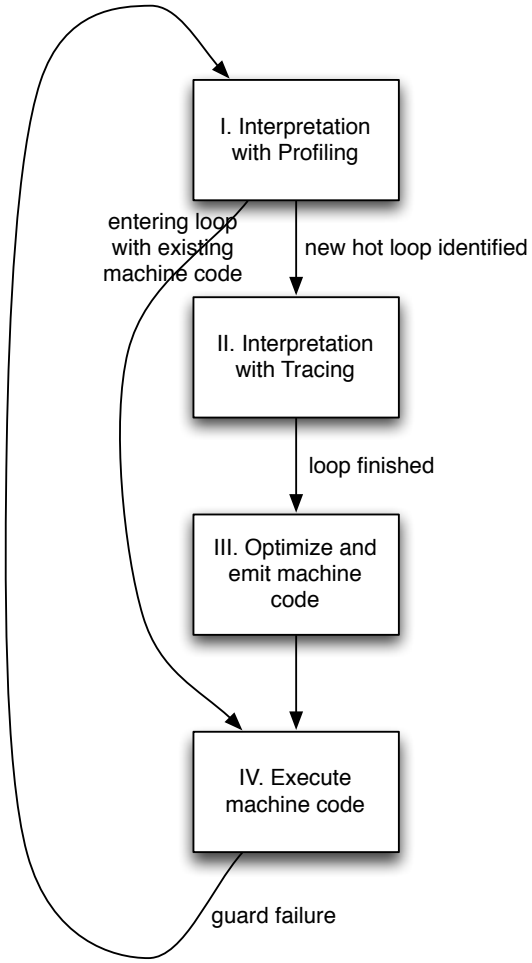

Figure 1: The stages of a tracing JIT compiler

A VM with a tracing JIT is typically a mixed-mode execution environment, containing both an interpreter and a JIT compiler. In the beginning (phase I in Figure 1), all code is executed by the interpreter, which also performs some profiling to identify hot loops of the program. If a hot loop is found, the next time the interpreter executes it, it enters a special *tracing* mode (phase II), where all operations that the interpreter performs are recorded. This history of recorded operations is called a *trace*. A trace corresponds to one possible code path through the hot loop. A trace always ends with a jump to its beginning, since it corresponds to a loop. The trace can be used to then generate efficient linear assembler code for exactly this code path (phase III). The generated assembler is cached and if the interpreter later wants to execute the same loop again, it will switch to running the compiled code instead (phase IV).

Each of these pieces of assembler code is only valid as long as the subsequent execution stays on the same code path. To check whether this is the case, *guards* are inserted into the assembler which check whether the assumptions made during the generation of the trace are still valid. This means that after creation, every piece of assembler is an infinite loop that can only be exited via a guard. The loop condition is represented as one of the guards in the trace. If one of those guards fails, execution will fall back to interpretation. If one specific guard fails often enough, another trace is generated starting from that specific point in the program and the existing assembler is patched to jump to that new trace [16].

An important property of the tracing approach is that it automatically supports function inlining by construction. During tracing, if a function is called during the execution of the hot loop, the trace will also record the operations that the called function executes, thus effectively inlining the called function.

The focus on traces means that a tracing JIT works particularly well for code that has many hot loops that do not have too many paths through them and that are not too large. It remains to be seen whether typical Prolog programs have these characteristics.

3. STRUCTURE OF THE INTERPRETER

The goal in implementing our Prolog interpreter in RPython was to have a simple, high-level object oriented implementation of Prolog. The semantics of Prolog should be mirrored closely by the structure of the interpreter. We wanted to incorporate high-level optimizations into the interpreter, but not be concerned about low-level details, which are left to the PyPy translation tools to deal with.

The resulting interpreter fulfills many of these goals. It uses a simple structure copying approach, has a straight-forward data model (Section 3.1) and uses continuation objects for the interpretation core (Section 3.2). So far it does not contain many optimizations, e.g., there is no indexing implemented yet. In addition to the interpreter core, we have implemented a number of builtins (see Section 3.3).

The interpreter is about 5000 lines of RPython code, of which 1000 lines are implementing builtins and 1700 are

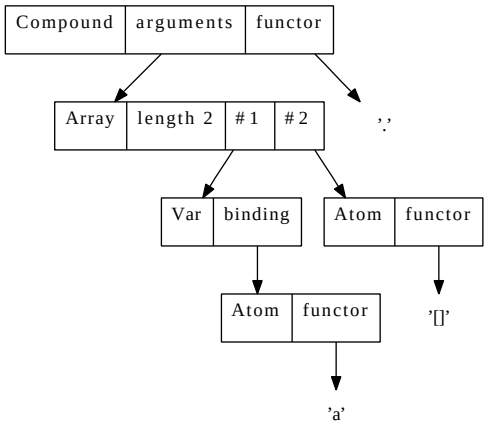

Figure 2: Representation of Prolog object `[X]` where `X` was bound to `a`

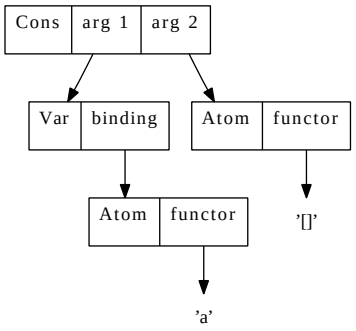

Figure 3: Representation using a specialized class for cons cells

tests. It can be translated to C using PyPy's translation toolchain and a JIT can be automatically generated for it (see Section 4). When translating to C without a JIT, the translation toolchain generates about 200'000 lines of C code, the compiled binary is 700 KB large. When also generating a JIT, about 600'000 lines of C code are generated (much of it support code for the generated JIT) resulting in a binary of 2.0 MB.

3.1 Data Model of the Interpreter

To represent Prolog terms the interpreter uses a straightforward object-oriented design of the Prolog concepts. It is the data model that would be expected from an implementation written in e.g., Java.

Prolog objects are modelled by instances of subclasses of the `PrologObject` base class. Simple non-variable terms are represented by their own class, such as `Atom`, `Number` and `Float` (which are just boxes around a string, an integer and a floating point number respectively). Logic variables are represented by instances of a class `Var`. This class has a `binding` attribute which is initialized to a `NULL` pointer to signify that the variable is not bound. If the variable gets bound, the `binding` attribute gets set to the bound value. Compound terms are represented by a class `Compound`, which has a string attribute specifying the functor and an array of more `PrologObjects`, for the arguments.

Unification is implemented in an object-oriented style: all `PrologObjects` have a `unify` method, which takes a second object as the argument, as well as a `Trail` object. The `unify` method calls itself recursively on the arguments of compound terms.

When a variable is bound, it needs to be trailed to be able to undo the binding should backtracking occur. This is done with the help of a `Trail` object. `Trail` objects are connected as a linked list, one object per choice point, each trail pointing to its next-oldest predecessor. If a variable is bound, it is stored into an array in the current, newest trail object, which holds all variables that will need to get their bindings undone when backtracking occurs.

As an optimization we implement *variable shunting* [20]. If a variable is created and immediately bound, i.e., without a choice point being created in the meantime, it does not need to be trailed. On backtracking the variable will cease to exist anyway. This is achieved my making the `Var` objects point to the trail object in which they were created. If they are bound while this trail is still being used, the variable can be replaced by its binding value.

3.1.1 Optimizing Term Classes

Most low-level Prolog engines use a tagged pointer representation [19] for commonly-used compound terms, typically for cons cells (terms with functor `./2`). The PyPy translation toolchain does not provide this level of control over low-level representation of objects. However, we still optimize terms with common functors, by representing them with their own class. This means that at least the array and the explicit reference to the functor can be saved. For an illustration of the concept, see Figure 2 for a representation of an unoptimized compound term and Figure 3 for the same term using the optimized classes.

In addition, we optimize compound terms with a small number of arguments (currently up to 10) to also use a special class to not need the array.

3.2 Continuation-Based Interpretation

The actual interpreter is based on *continuations*. Several Prolog systems have been based on continuations already [29, 26, 25]. The basic approach in our implementation is that all the state of the interpreter is encapsulated in two (possibly nested) continuation objects, a *success continuation* and a *failure continuation*. All continuations are instances of one of the subclasses of a `Continuation` class. A continuation thus contains state as well as behaviour.[3]

The success continuation contains the still to be executed "rest of the program", the failure continuation contains the code that needs to be executed if backtracking needs to happen. Calling a continuation typically consumes it, and potentially replaces the current continuations by new ones. In-

[3]Indeed, on systems that implement Prolog on Scheme or Lisp [23], a continuation is usually just represented by a closure. This was not possible for our system, because RPython supports neither closures nor does it optimize tail calls.

```
def interpret(sc, fc, trail):
    while not sc.is_done():
        try:
            sc, fc, trail = sc.activate(fc, trail)
        except UnificationFailed:
            sc, fc, trail = fc.fail(trail)
```

Figure 4: The Main Interpreter Loop

terpretation proceeds by calling the current success continuation until the computation is finished. If calling a continuation fails, the current failure continuation is called instead.

Whenever a non-deterministic choice is reached, the interpreter creates a new trail object. It then builds a failure continuation that backtracks to the previous state and then continues with the other option.

The main loop of the interpreter (in slightly simplified form) can be seen in Figure 4. The loop has three local variables: `sc` is the success continuation, `fc` the failure continuation and `trail` the current trail object. As long as there is still something to do, the `activate` method of the current continuation is called, which returns a new set of continuations. If activating the continuation fails, it will raise an `UnificationFailed` exception. If that happens, the failure continuation will get its `fail` method called.

The types of continuation used by the interpreter are (we give the continuations here as Prolog terms, in the actual implementation each sort of continuation is simply a class with the arguments of the terms as attributes):

- `call(Goal, Next)` which will call the goal, when activated
- `restore(Trail, Next, FailureContinuation)` which will backtrack the bindings done up to the point specified by `Trail`
- `apply(Rule, Goal, Next)` which applies a specific rule of a predicate to the goal
- `true` which signifies that the computation is finished

All continuations have a `Next` continuation, which will be called after the current continuation has been executed. In addition to the continuations listed, there are specific continuations used for builtins that can have more than one solution (e.g., `arg`).

For an example see Figure 5. The figure shows the continuations that are constructed when calling a predicate `f(X)` which has two rules in the database. When applying the first rule, a new trail is created and the failure continuation set to a `restore`, which can potentially undo the changes done by the first rule and continue with the second rule, if backtracking occurs later.

The overhead of constantly creating these continuation objects is kept small by the good GC support that the PyPy toolchain gives us (see Section 2.4). Since most of the continuations are very short-lived they are collected extremely efficiently by the generational GC.

Database:

```
f(a). f(b).
```

Continuations:

1	sc	call(f(X), true)
	fc	true
	trail	<trail1>
2	sc	apply(<f rule 1>, f(X), true)
	fc	restore(<trail1>, apply(<f rule 2>, f(X), true), true)
	trail	<trail2>
3	sc	true
	fc	restore(<trail1>, apply(<f rule 2>, f(X), true), true)
	trail	<trail2: X=a>

Figure 5: Continuations when calling a predicate `f(X)` with two rules

3.3 Implementing Builtins

In addition to the core Prolog execution model we also implemented a number of builtins. Most builtins are rather straightforward to implement using the continuation-based model. Builtins that always have at most one solution are trivial, builtins that can have many solutions need some more work, because they typically need a new type of continuation. There are a number of builtins that need some care, because they manipulate the current continuations in more complex ways.

The negation builtin `\+ Goal` basically performs a call to `Goal` but swaps failure and success continuation when executing the call[4]. Indeed, if `Goal` fails the whole construct succeeds, and vice versa.

The builtin `repeat` needs to introduce a special sort of failure continuation which is not consumed when it is activated, thus providing an arbitrary amount of solutions.

If-then-else `A -> B ; C` needs to remove those failure continuations that were introduced during the execution of `A`. Removing a failure continuation also means that the trail object which corresponds to the removed choice is merged with its predecessor. Similarly the cut `!` needs to remove all failure continuations that were introduced during the execution of the current predicate. It is not trivial to figure out the extend of the cut, since the continuations are not marked by which predicate they were created for. Therefore a special marker continuation is needed if a predicate that contains a cut is called. In this regard, if-then-else is a much

[4]Together with some extra code to remove the bindings done during the execution of the goal.

cleaner concept.

The `findall` builtin needs a special sort of success continuation which, when activated, collects the found solution and then forces backtracking.

The exception handling builtins work as follows. A call to `catch` will insert a special catching success continuation, which will not do anything when actually activated. When `throw` is called, it will walk the chain of success continuations until it finds a matching catching continuation and continue by calling the recovery goal of the `catch` call.

4. AUTOMATIC JIT GENERATION APPLIED TO PROLOG

In this section we will describe how the JIT generator of PyPy is applied to the Prolog interpreter. The first task in doing so is correctly placing hints in the source code of the interpreter [5]. The most important hints which are needed for the JIT generator are:

- A hint to indicate the interpreter's main loop to the JIT generator. In the Prolog case this is the `driver` loop shown in Figure 4.
- A hint to annotate those variables of the interpreter which represent a position in the program that is currently being interpreted. In a typical bytecode-based imperative-language interpreter this is the program counter. Since our interpreter is not bytecode-based, we chose to mark the currently executed Prolog rule.
- A hint to indicate the code of the interpreter that is responsible for closing a loop. Again, in an imperative language this hint is usually placed in the implementation of the bytecode which performs "backward jumps". This one is the hardest in Prolog, since there is no explicit loop construct, only tail calls. Therefore we placed the hint in the code that is responsible for applying one specific rule (see Section 3.2).
- Many classes in the interpreter are marked as *immutable*. This means that instances of these classes will not be changed after they have been initialized. Examples for such classes are all the classes implementing Prolog terms, except `Var`; the class that represents Prolog rules (but not predicates, because `assert` and `retract` can change predicate objects).

4.1 Loops in Prolog Code

Since the generated JIT is a tracing one that focuses on producing good code for loops, it is important to discuss when a loop actually occurs in Prolog. Despite Prolog not having an explicit loop construct, there are still a number of cases in which the generated JIT will detect a loop. A loop for the JIT is simply a situation where the same rule of a predicate will be applied repeatedly (potentially with other rule applications in between).

The most straightforward sort of loop is a loop with *tail calls*, like a list-append where the first argument is instantiated, or an arithmetic loop. However, it is not really necessary for the call in the loop to be in a tail position. If one takes the

```
nrev([], []).
nrev([X|Y], Z) :- nrev(Y, Z1), append(Z1, [X], Z).
```

Figure 6: Code of naive reverse

implementation of naive reverse in Figure 6, the second rule of `nrev` will repeatedly call itself (constructing a continuation that calls `append` at every iteration). Thus the JIT will also detect it as a loop. After the base case is reached, those continuations will be activated one after another, which is yet another loop (which is distinct from the loop of `append` itself).

4.2 Optimizations by the JIT

After the generated tracing JIT identified and traced a loop in the executed Prolog code, it performs a number of optimizations on the traces before they are turned into assembler code. We will describe those optimizations in this section. These optimizations are part of the JIT infrastructure of the PyPy project [5] and did not have to be written for Prolog specifically (indeed, we did not have to change any PyPy code at all).

The two most important optimizations that the JIT performs on the recorded traces are:

- Constant-folding reads out of immutable and known objects.
- Completely removing object allocations that have a limited life-time (escape analysis [18]).

In the following we will describe what effects these optimizations have when applied to the RPython Prolog interpreter.

The first optimization applies mostly to rule objects. Since a rule is immutable (even in the presence of assert and retract), all the reads out of it are constant-folded away. This applies in particular to reading the head and body of the rule. The head and body of the rule are themselves immutable terms (since they are usually not variables), thus the JIT will recursively optimize away most of those reads. This means that for the unification of a rule head with a calling term, all of the operations acting on the rule head are constant-folded away.

The second optimization will remove object allocations from the trace if the allocated object is only used locally in the trace and does not escape anywhere else, which would happen if the object is stored into another, global, object. To do this optimization, the JIT looks at an object allocation in the trace and tracks where the allocated object is used. If it is not stored anywhere outside of the trace, the object will die after its last use, thus the allocation can be removed and the object is replaced by its fields.

This process can often remove all overhead of using continuations in the interpreter. If a continuation object is created, it will often just be activated quickly afterwards and then not be used anymore. In this case the continuation object

will be fully removed by the optimizer. Only in the case when a choice point is created or the continuation actually grows, can the allocation not be removed (e.g., this is the case for naive reverse in Figure 6).

In addition to the removal of continuations, allocations of Prolog objects can be avoided by this optimization. When standardizing apart before the application of a rule a copy of the rule body is created. Some parts of the copied body will be immediately deconstructed again, thus they don't need to be allocated at all.

As an example of what the optimizations can achieve, let's look at what happens when the Prolog interpreter executes a simple arithmetic iteration (see Figure 12 for the code). At first, the interpreter will normally run the `iterate` loop, keeping count of which predicates are executed often. After a few iterations, it will identify the `iterate` predicate as a likely candidate, so it enters *tracing mode*, keeping a trace of all the execution steps that the interpreter performs. The generated trace (which is quite detailed and thus rather long, about 200 operations) will then be optimized as described above.

Most of the operations in the trace are removed by the optimization step. The resulting trace can be seen in Figure 7. This trace will then be turned into machine code by an architecture-specific assembler backend and can then be executed.

In this simple example the optimizer of the JIT was able to remove all the allocations in the trace, since the continuations that are created are immediately activated and do not escape anywhere. The same is true for the copied body of the `iterate` predicate. In addition, even the `Number` object that is used to box the integer value of the loop variable is removed, since each of these objects survives for one iteration of the loop only. Thus the generated assembler code can keep the loop index in a machine integer, which can just be kept in a CPU register. All the `int_*` operations are just simple machine instructions.

The `jump` instruction at the end of the trace jumps to the beginning again. Thus the trace by itself is an infinite loop. It can only be left via one of the guard instructions. Those guards check that the assumptions of the trace are not violated. If the machine code is executed and the iteration count reaches zero, the first guard will fail and execution will fall back to using the interpreter again.

5. EVALUATION

In this section we want to evaluate the performance and memory behaviour of our Prolog system when translated to C and then compiled to an executable, with and without generating a JIT compiler.

Unless otherwise stated, we were running the benchmarks twice in the same process, performing the time measurement during the second run. This should give the JIT a time to warm up and produce machine code without affecting the measurements. We are also looking at compilation times in Section 5.2.

```
Loop: [scont, i1, fcont, trail]

# Check whether the base case applies(X)
i2 = int_eq(i1, 0)
guard_false(i2)

# X > 0
i3 = int_gt(i1, 0)
guard_true(i3)

# X0 is X - 1
i4 = int_sub(i1, 1)

# recursive call to iterate(Y)
# Check whether assert or retract was
# used on the loop/1 function:
p2 = read_field(<address loop/1>, 'first_rule')
guard_value(p2, <address 1st rule of loop/1>)

jump(scont, i4, fcont, trail)
```

Figure 7: The intermediate code for the generated assembler code of the `loop/1` function

All benchmarks were performed on an otherwise idle Intel Core2 Duo P8400 processor with 2.26 GHz and 3072 KB of cache on a machine with 3GB RAM running Linux 2.6.31. We compare the performance of our Prolog system against that of Sicstus Prolog 4.1.1, Ciao Prolog 1.13.0-11568, SWI-Prolog 5.8.3 and tuProlog 3.0-alpha. We were running Sicstus in both its interpreter mode and its compiled mode, using `load_files/2` with the `compilation_mode(consult)` and `compilation_mode(compile)` flags respectively.

5.1 Iteration benchmarks

The first set of benchmarks we tried uses various methods to implement iteration in Prolog (see Figure 12 for the code). This is completely untypical Prolog code. We still wanted to have these benchmarks, as they are the sort of code that the tracing JIT is best at, to gauge the maximum speedup that the technology can give us.

The results of running these iteration benchmarks each with 10 million iterations can be seen in Figure 8. For two of the benchmarks, SWI-Prolog did not finish the run, as it was running out of memory. Our interpreter without a JIT seems to have about twice the speed of Sicstus in interpreted mode and is quite a bit slower than the other implementations. With the JIT we seem to be faster than all other implementations, sometimes significantly, apart from when the cut is involved (even then we are faster with the JIT than without). So far we have not been able to figure out why the generated JIT is not able to do something sensible with the cut.

In the cases of `iterate_assert` and `iterate_call` the JIT is able to show its full strength. Given the dynamic setting, the JIT can treat the asserted predicate like a static predicate and will compile new code once new clauses are added. Thus the asserted code has the same speed as the normal code here (indeed our interpreter just assumes all predicates to be

dynamic, since it would have been more work to implement otherwise). A similar thing happens in the case of `call`: The JIT can optimize the `call` builtin by optimistically assuming that the target predicate will stay the same and producing new code if this assumption proofs to be wrong later.

The `iterate_exception` benchmark is admittedly rather silly (we hope nobody actually writes code like this). However, it showcases that the JIT optimizes all features that the interpreter implements, without having to do extra work.

5.2 Classical Prolog Benchmarks

In addition to the unrealistic micro-benchmarks from the last section we also measured the various Prologs against a number of slightly larger programs, most of them well-known benchmarks. Many of these benchmarks execute so quickly that we had to run them many times to get sensible measurements. The following benchmarks were each run 500 times: `chat_parser`, `crypt`, `deriv` `qsort` sorting a list of 50 elements, `reducer`, `zebra`.

In addition we were using the following benchmarks: `boyer`, `tak`, `nrev` which uses naive reverse to reverse a list of 1700 elements, `queens` solving the queens puzzle with 11 queens, `primes` searching for all primes up to 10'000. `arithmetic` is a declarative arbitrary-precision arithmetic implementation using lists of bits to represents the numbers. The benchmark computes 14! and is derived from code from [13].

To also find out how much overhead the JIT compiler itself adds, for these benchmarks we were measuring two numbers for the JIT version. One by measuring the time of the benchmark after startup of the interpreter, which will include compile time. Then we ran the benchmark a second time in same process and measured that time to find out how fast the (now fully generated) machine code is.

The results of these benchmarks can be seen in Figure 9, a diagram showing the results in Figure 10. Our Prolog interpreter without the JIT is significantly slower for these more realistic Prolog programs. It is between 5 times slower and a bit faster than Sicstus in interpreted mode, and significantly slower than the other Prolog implementations.

If the JIT is also generated, the execution times always improve, apart from the `boyer` benchmark which actually becomes significantly slower with the JIT. We think that this is due to overspecialization by the JIT and are working on solving the problem. Apart from that benchmark, the JIT gives a speedup of up to 10 times, which makes it competitive with Sicstus in compiled mode for the benchmarks `queens` and `arithmetic`.

In most cases, even factoring in compilation time gives a speedup over interpretation, except in the cases of `boyer` and `chat_parser`, where the non-warmed up version is slower than the interpreted version. The compilation time seems to be a bit high, given the small size of the benchmarks (and given that none of the benchmarks are really useful long-running programs). We will have to improve on that in the future.

5.3 Comparison Against tuProlog

To get a impression of how other high-level Prolog implementations perform, we also tried to benchmark tuProlog [12]. The results can be seen in the last column in Figure 9. However, most benchmarks were not actually completing, either because of missing builtins or because of an out-of-memory error. For the benchmarks that actually worked, the performance is up to several orders of magnitude slower than the low-level implementations. tuProlog's performance seems to be characteristic for Java Prologs though [27]. Of course it was not the goal of tuProlog to reach good performance anyway.

5.4 Memory Footprint

To measure the overhead of having an object-oriented object model and of representing the interpreter state in continuation objects, we also measured the memory footprint of each Prolog interpreter, by running each benchmark and continuously sampling the physical memory the process used. The numbers are reported in Figure 11 and are the maximum amount of memory each benchmark used during the run.

For most benchmarks the memory footprint of our interpreter is about 2-5 times larger than that of the other interpreters. Exceptions are `nrev`, which takes 20 times more memory and `reducer`, which even uses 200 times more. The size difference that the `nrev` benchmarks exhibits shows that continuations have a large memory overhead, because `nrev` builds a chain of continuations as large as the reversed list has elements.

6. RELATED WORK

It has been the dream of partial evaluation [21] to compile programs by specialising interpreters. Unfortunately, up to now "widely used partial evaluators are nowhere to be seen" [3], even though there have been some successful applications, such as for example [2, 24, 4]. To our knowledge, these applications have in common, that they are applied to very domain-specific languages. In our work, we apply our technique on an interpreter for a general-purpose programming language, and we try and compete with industrial compilers. Furthermore, the source and object language are very different (RPython vs Prolog).

Continuations have been used in various cases as the basis for implementing Prolog system. BinProlog [29] uses a transformation to continuation-passing-style for all Prolog clauses and then uses a simplified WAM to execute those. However, it uses only a success continuation and thus doesn't make the choice points explicit. There was some more work to use this single success-continuation passing style for optimizations [11, 26].

Lindgren [25] proposes to use a continuation-passing style as an intermediate language before code generation. Contrarily to the approaches mentioned so far, he uses both a success *and* a failure continuation, thus moving all control decisions to the source level. In our implementation we don't generate continuation-passing style as a preprocessing step, but rather use continuations at runtime to represent the interpreter style.

There have been a number of attempts at writing high-level

Benchmark	SICStus-interp	SICStus	SWI	Ciao	own-interp	own-JIT warm
iterate	14930 ms	350 ms	630 ms	400 ms	4790 ms	60 ms
iterate_assert	15000 ms	15730 ms	1780 ms	7460 ms	4910 ms	60 ms
iterate_call	23400 ms	2920 ms	–	306430 ms	12950 ms	150 ms
iterate_cut	18200 ms	520 ms	850 ms	510 ms	9380 ms	4450 ms
iterate_exception	37800 ms	14370 ms	–	58710 ms	14170 ms	850 ms
iterate_failure	31210 ms	1010 ms	5380 ms	1140 ms	13510 ms	690 ms
iterate_findall	39230 ms	9780 ms	7930 ms	3470 ms	15130 ms	1190 ms
iterate_if	22770 ms	860 ms	1500 ms	1120 ms	12920 ms	150 ms

Figure 8: Benchmark times for iteration benchmarks

Benchmark	SICStus-interp	SICStus	SWI	Ciao	own-interp	own-JIT	own-JIT warm	tuprolog
arithmetic	3630 ms	490 ms	1080 ms	600 ms	3180 ms	570 ms	310 ms	80.1 **s**
boyer	490 ms	40 ms	100 ms	40 ms	1130 ms	25080 ms	20000 ms	–
chat_parser	20930 ms	5050 ms	9030 ms	5880 ms	35460 ms	44880 ms	17910 ms	–
crypt	1810 ms	70 ms	470 ms	100 ms	780 ms	710 ms	110 ms	14.9 **s**
deriv	1200 ms	190 ms	420 ms	240 ms	2770 ms	2400 ms	1040 ms	–
nrev	820 ms	60 ms	180 ms	70 ms	1220 ms	950 ms	640 ms	83.3 **s**
primes	2230 ms	190 ms	380 ms	140 ms	1680 ms	1110 ms	910 ms	–
qsort	1320 ms	160 ms	440 ms	160 ms	2080 ms	870 ms	430 ms	–
queens	8930 ms	460 ms	1880 ms	560 ms	4410 ms	600 ms	390 ms	274.3 **s**
reducer	6610 ms	930 ms	2710 ms	1430 ms	29650 ms	26350 ms	23840 ms	–
tak	720 ms	20 ms	80 ms	20 ms	570 ms	290 ms	270 ms	8.7 **s**
zebra	2480 ms	1050 ms	2400 ms	1510 ms	5990 ms	5040 ms	3410 ms	–

Figure 9: Benchmark times for classical Prolog benchmarks

object oriented Prolog interpreters. tuProlog is a Prolog running on top of a Java virtual machine which was written with good object-oriented design in mind [12]. It uses a state machine to execute Prolog programs [27], whose states can be related to the kinds of continuations of our interpreter.

Costa et al [10] have modified YAP to perform demand-driven indexing. Their technique analyses and modifies the WAM bytecode of a predicate at runtime if it looks like the predicate could benefit from indexing on other arguments than the first. Thus they avoid heuristics or costly upfront analysis to find out on which argument indexing should be performed. This is a great example of how runtime techniques can improve performance of Prolog systems.

7. CONCLUSIONS

In this paper we presented a simple Prolog interpreter written in RPython, which can be compiled into a C-level VM with the PyPy translation toolchain, optionally also generating a tracing JIT compiler in the process. The resulting VM is reasonably efficient and can be very fast in cases where the generated JIT works well. Our approach represents a success story for partial evaluation on a large language implementation. To the best of our knowledge, it is also the first Prolog implementation that defers all compilation to runtime. We argue that Prolog can greatly benefit from JIT compilation techniques, given its dynamic nature.

At the moment there are also a number of disadvantages to our approach. The memory usage of the resulting interpreter can be very bad, due to the overhead of using many objects and the lack of low-level control. In addition, the way the generated JIT works is not always very transparent, sometimes making it hard to know why certain Prolog code is compiled efficiently to assembler and other code is not. Sometimes the JIT compiler itself can take too much time to be really profitable.

We plan to investigate in much more detail why the JIT is sometimes not giving very good results; this might make it necessary to improve the JIT generator of the PyPy project itself. In addition we want to improve the interpreter itself by adding more Prolog-level optimizations such as indexing. We should also find ways to save memory, e.g., by forgoing some of the abstractions in the interpreter.

8. REFERENCES

[1] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the 2007 symposium on Dynamic languages*, pages 53–64, Montreal, Quebec, Canada, 2007. ACM.

[2] L. Augustsson. Partial evaluation in aircraft crew planning. In *PEPM*, pages 127–136, 1997.

[3] L. Augustsson. O, partial evaluator, where art thou? In J. P. Gallagher and J. Voigtländer, editors, *PEPM*, pages 1–2. ACM, 2010.

[4] S. Barker, M. Leuschel, and M. Varea. Efficient and flexible access control via logic program specialisation. In *Proceedings PEPM'04*, pages 190–199. ACM Press, 2004.

[5] C. F. Bolz, A. Cuni, M. Fijałkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*,

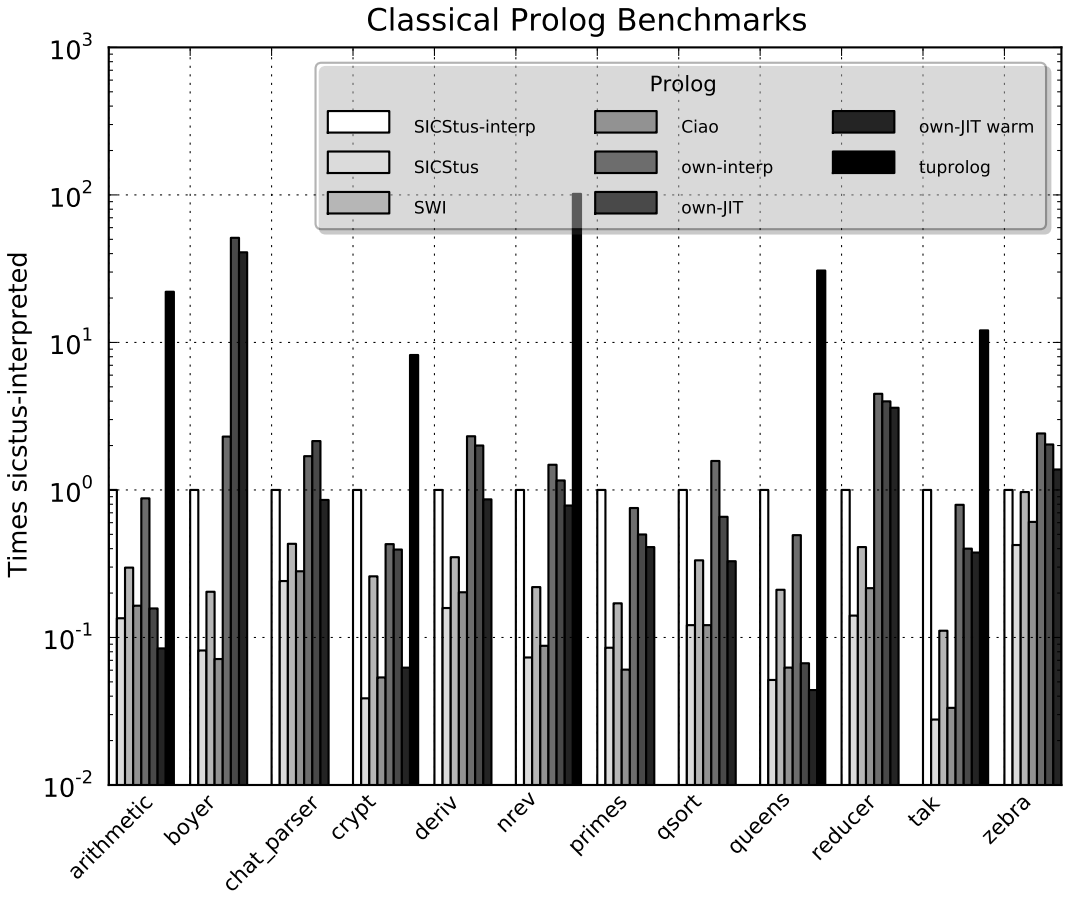

Figure 10: Results of classical Prolog Benchmarks

Benchmark	SICStus-interp	SICStus	SWI	Ciao	own-interp	tuProlog
arithmetic	3.9 MB	3.1 MB	2.8 MB	9.6 MB	5.8 MB	101.0 MB
boyer	4.5 MB	4.5 MB	3.2 MB	10.9 MB	13.9 MB	–
chat_parser	3.5 MB	3.2 MB	2.5 MB	9.5 MB	11.2 MB	–
crypt	3.2 MB	3.0 MB	2.4 MB	9.4 MB	3.8 MB	109.9 MB
deriv	3.7 MB	3.0 MB	2.5 MB	9.5 MB	7.8 MB	–
nrev	4.4 MB	4.4 MB	2.8 MB	10.9 MB	52.8 MB	155.4 MB
primes	36.1 MB	23.1 MB	28.0 MB	36.8 MB	49.2 MB	–
qsort	3.7 MB	3.0 MB	2.4 MB	9.4 MB	7.7 MB	–
queens	3.0 MB	3.0 MB	2.4 MB	9.4 MB	7.2 MB	135.7 MB
reducer	4.5 MB	4.5 MB	2.7 MB	9.9 MB	542.3 MB	–
tak	4.7 MB	3.0 MB	2.6 MB	9.4 MB	11.1 MB	106.0 MB
zebra	3.3 MB	3.0 MB	2.4 MB	9.4 MB	7.8 MB	–

Figure 11: Memory footprint for classical Prolog benchmarks

pages 18–25, Genova, Italy, 2009. ACM.

[6] C. F. Bolz, M. Leuschel, and A. Rigo. Towards Just-In-Time partial evaluation of Prolog. In *Logic-based Program Synthesis and Transformation (LOPSTR'2009)*, LNCS 6037 to appear. Springer-Verlag.

[7] C. F. Bolz and A. Rigo. How to not write a virtual machine. In *Proceedings of the 3rd Workshop on Dynamic Languages and Applications (DYLA 2007)*, 2007.

[8] M. Chang, M. Bebenita, A. Yermolovich, A. Gal, and M. Franz. Efficient Just-In-Time execution of dynamically typed languages via code specialization using precise runtime type inference. Technical Report ICS-TR-07-10, Donald Bren School of Information and Computer Science, University of California, Irvine, 2007.

[9] J. J. Cook. P#: a concurrent prolog for the .NET framework. *Softw. Pract. Exper.*, 34(9):815–845, 2004.

[10] V. S. Costa, K. Sagonas, and R. Lopes. Demand-Driven indexing of Prolog clauses. In *Logic Programming*, pages 395–409, 2007.

[11] B. Demoen. On the transformation of a Prolog program to a more efficient binary program. Technical Report 130, K.U. Leuven, Dec. 1990.

[12] E. Denti, A. Omicini, and A. Ricci. tuProlog: a light-weight Prolog for internet applications and infrastructures. In *Practical Aspects of Declarative Languages*, pages 184–198. 2001.

[13] D. P. Friedman, W. E. Byrd, and O. Kiselyov. *The Reasoned Schemer*. MIT Press, July 2005.

[14] A. Gal, M. Bebenita, and M. Franz. One method at a time is quite a waste of time. In *Proceedings of the Second Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2007)*, pages 11–16, Berlin, Germany, July 2007.

[15] A. Gal, B. Eich, M. Shaver, D. Anderson, B. Kaplan, G. Hoare, D. Mandelin, B. Zbarsky, J. Orendorff, M. Bebenita, M. Chang, M. Franz, E. Smith, R. Reitmaier, and M. Haghighat. Trace-based Just-in-Time type specialization for dynamic languages. In *PLDI*, 2009.

[16] A. Gal and M. Franz. Incremental dynamic code generation with trace trees. Technical Report ICS-TR-06-16, Donald Bren School of Information and Computer Science, University of California, Irvine, Nov. 2006.

[17] A. Gal, C. W. Probst, and M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 144–153, Ottawa, Ontario, Canada, 2006. ACM.

[18] B. Goldberg and Y. G. Park. Higher order escape analysis: optimizing stack allocation in functional program implementations. In *Proceedings of the third European symposium on programming on ESOP '90*, pages 152–160, Copenhagen, Denmark, 1990. Springer-Verlag New York, Inc.

[19] D. Gudeman. Representing type information in Dynamically-Typed languages. Technical Report TR93-27, University of Arizona at Tucson, 1993.

[20] S. L. Huitouze. A new data structure for implementing extensions to Prolog. In *Programming Language Implementation and Logic Programming*, pages 136–150. 1990.

[21] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[22] R. Jones and R. D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Wiley, Sept. 1996.

[23] K. M. Kahn and M. Carlsson. How to implement prolog on a LISP machine. In *Implementations of Prolog*, pages 117–134. 1984.

[24] M. Leuschel and D. De Schreye. Creating specialised integrity checks through partial evaluation of meta-interpreters. *The Journal of Logic Programming*, 36(2):149–193, August 1998.

[25] T. Lindgren. A Continuation-Passing style for Prolog. In *Symposium on Logic Programming*, pages 603–617, 1994.

[26] U. Neumerkel. Continuation Prolog: A new intermediary language for WAM and BinWAM code generation. *Post-ILPS'95 Workshop on Implementation of Logic Programming Languages. F16G*, 1995.

[27] G. Piancastelli, A. Benini, A. Omicini, and A. Ricci. The architecture and design of a malleable object-oriented prolog engine. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 191–197, Fortaleza, Ceara, Brazil, 2008. ACM.

[28] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 944–953, Portland, Oregon, USA, 2006. ACM.

[29] P. Tarau. BinProlog: a continuation passing style Prolog engine. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science, page 479–480. Springer, Aug. 1992. poster.

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

basic iteration:

```
iterate(0).
iterate(X) :- Y is X - 1, iterate(Y).
```

iteration with an asserted predicate:

```
:- dynamic(iterate_assert/1).
iterate_assert(a).
:- assert(iterate_assert(0)).
:- assert((iterate_assert(X) :-
            (Y is X - 1, iterate_assert(Y)))).
```

iteration with `call`:

```
iterate_call(X) :- c(X, c).
c(0, _).
c(X, Pred) :-
    Y is X - 1, C =.. [Pred, Y, Pred], call(C).
```

iteration with a cut:

```
iterate_cut(0).
iterate_cut(X) :- Y is X - 1, !, iterate_cut(Y).
iterate_cut(X) :- Y is X - 2, iterate_cut(Y).
```

iteration with exceptions:

```
e(0).
e(X) :- X > 0, X0 is X - 1, throw(continue(X0)).
iterate_exception(X) :-
    catch(e(X), continue(X0), iterate_exception(X0)).
```

iteration with a failure-driven loop:

```
g(X, Y, Out) :- Out is X - Y.
g(X, Y, Out) :- Y > 0, Y0 is Y - 1, g(X, Y0, Out).

iterate_failure(X) :- g(X, X, A), fail.
iterate_failure(_).
```

iteration using findall:

```
iterate_findall(X) :-
    findall(Out,
            (g(D, D, Out), 0 is Out mod 50),
            _).
```

iteration with if-then-else:

```
equal(0, 0). equal(X, X).
iterate_if(X) :- equal(X, 0) -> true ;
                 Y is X - 1, iterate_if(Y).
```

Figure 12: A number of iteration benchmarks