Jitting Prolog for Fun and Profit

David Schneider
david.schneider@uni-duesseldorf.de

Carl Friedrich Bolz
cfbolz@gmx.de

Michael Leuschel
leuschel@cs.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. In addition, many of the more dynamic features of Prolog (like `assert`), despite their popularity, are not well supported. We present a high-level continuation-based Prolog interpreter based on the PyPy project. The PyPy project makes it possible to easily and efficiently implement dynamic languages. It provides tools that automatically generate a just-in-time compiler for a given interpreter of the target language, by using partial evaluation techniques. The resulting Prolog implementation is surprisingly efficient: it clearly outperforms existing interpreters 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 aims to show that declarative languages such as Prolog can indeed benefit from having a just-in-time compiler and that PyPy can form the basis for implementing programming languages other than Python.[1]

1. INTRODUCTION

An often cited problem of dynamic languages is their relatively slow performance compared to static languages. Of course techniques for implementing such languages efficiently, such as JIT compilation, are well known and have existed since a long time [10, 14, 3]. Although writing a simple JIT compiler is not a hard task, really achieving high performance and portability is a hard and tedious undertaking.

Given these problems and the fact that different dynamic languages often have common requirements in respect to a JIT, it seems worthwhile to generalize the procedure of creating such a JIT. This is the approach the PyPy project has taken [5, 18].

Prolog is a popular logic programming language created in the 70's by Colmerauer and Roussel [7]. In addition to its logic features it is a fully dynamic language providing dynamic typing, reflection on data structures and the program itself. It is usually implemented using low level languages and a specific virtual machine designed for Prolog (called Warren's Abstract Machine, or WAM). Despite the dynamic character of the language, to our knowledge no publicly available Prolog implementations have adopted dynamic compilation techniques so far.

Our aim was to investigate the applicability of JIT techniques to Prolog and whether PyPy's JIT generator can be applied to a logic programming language. We thus present a Prolog interpreter written with the help of PyPy's tools. While writing the interpreter, we tried to follow the Prolog semantics as closely as possible. The interpreter can be compiled to C and a JIT can be generated for it by the PyPy JIT generator.

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 [20, 19]. 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 C# [17, 8]. 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 [18, 6] is an environment where interpreters for 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. RPython is restricted in such a way, that type inference is possible and therefore the interpreter can 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 2.3).

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

[1]This research is partially supported by the BMBF funded project PyJIT (nr. 01QE0913B; Eureka Eurostars). A greatly extended version of this paper has been submitted to PPDP 2010.

it is now used for the implementation of various dynamic languages, such as Squeak/Smalltalk [6], JavaScript and now Prolog.

2.3 JIT Compilers

One 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 needed to identify the main interpreter loop among other things.

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 languages 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 this process generates is a *tracing JIT*. It focuses on generating good machine code for hot loops. The detection and code generation for loops is performed by observing the interpreter executing the program [12, 11].

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 has a straight-forward data model and uses continuation objects for the interpretation core (Section 3.1). So far it does not contain many optimizations, e.g., there is no indexing implemented yet.

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

To represent Prolog terms the interpreter uses a straightforward object-oriented design of the Prolog concepts. 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

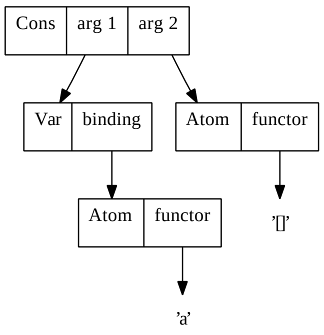

Figure 1: List representation using a specialized class for cons cells

represented by instances of a class `Var`. Unification is implemented in an object-oriented style: all `PrologObjects` have a `unify` method, which takes a second object as the argument. Whenever this process binds a variable it needs to be trailed, to make backtracking possible.

3.1 Continuation-Based Interpretation

The actual interpreter is based on *continuations*. 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.

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. Interpretation 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 failure continuation that backtracks to the previous state and then continues with the other option.

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

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 central 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.
- A hint to annotate those variables of the interpreter which represent the 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.

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

Figure 2: A simple arithmetic iteration

- 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 any call to a Prolog predicate has to be considered as the possible start of a loop.

The JIT considers Prolog code to contain a loop when the same rule of a predicate is 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.

4.1 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 machine code code.

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 [13]).

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.

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 2 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 3. 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. 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 machine code 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.

```
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 iterate/1 function:
p2 = read_field(<address iterate/1>, 'first_rule')
guard_value(p2, <address 1st rule of iterate/1>)

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

Figure 3: The intermediate code for the generated machine code code of the `iterate/1` function

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.

4.2 Evaluation

To evaluate the performance of our Prolog system we ran a number of classical Prolog benchmarks. For space reasons we cannot present the results here. To summarize the results: On micro-benchmarks such as arithmetic iterations we are around 5 times faster than Sicstus Prolog. On small to medium Prolog programs our JIT is on average (geometric mean) 6 times slower than Sicstus Prolog. Given that tuProlog, a Prolog in Java, is on average 400 times slower than Sicstus we are content with these first results.

5. RELATED WORK

It has been the dream of partial evaluation [15] to compile programs by specialising interpreters. Unfortunately, up to now "widely used partial evaluators are nowhere to be seen" [2], even though there have been some successful applications, such as [1, 16, 4]. To our knowledge, all of these applications targeted very domain-specific languages, whereas our work tries to use partial evaluation techniques on an interpreter for a general purpose language.

There have been a number of attempts at writing high-level 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 [9]. It uses a state machine to execute Prolog programs [17], whose states can be related to the kinds of continuations of our interpreter.

6. 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 transparent, sometimes making it hard to know why certain Prolog code is compiled efficiently to machine code 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 much speedup over the interpreter; 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.

7. REFERENCES

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

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

[3] J. Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003.

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

[6] 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.

[7] A. Colmerauer and P. Roussel. The birth of prolog. In *History of programming languages—II*, pages 331–367. ACM, 1996.

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

[9] 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.

[10] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302, Salt Lake City, Utah, United States, 1984. ACM.

[11] 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.

[12] 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.

[13] 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.

[14] U. Hölzle, C. Chambers, and D. Ungar. Optimizing Dynamically-Typed Object-Oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 21–38. Springer-Verlag, 1991.

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

[16] 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.

[17] 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.

[18] 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.

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

[20] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, 1983.