INSTITUT FÜR INFORMATIK
Softwaretechnik und
Programmiersprachen

Universitätsstr. 1 D–40225 Düsseldorf

HEINRICH HEINE
UNIVERSITÄT
DÜSSELDORF

A Prolog Interpreter in Python

Carl Friedrich Bolz

Bachelorarbeit

Beginn der Arbeit:	17. Januar 2007
Abgabe der Arbeit:	17. April 2007
Gutachter:	Prof. Dr. Michael Leuschel
	Prof. Dr. Stefan Conrad

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 17. April 2007

Carl Friedrich Bolz

Abstract

We provide a proof-of-concept for a new approach to flexible and portable implementation of programming languages. More precisely, we describe the implementation of a Prolog interpreter in RPython, a restricted subset of the Python language intended for system programming. RPython can be translated using the PyPy compilation and specialization tool-set to various platforms, among them C/Posix and .NET. This allows us to have a flexible, single implementation running in different environments without having to maintain several implementations. We also describe how we can handle deep recursion by using PyPy's stackless transformation. We compare the performance of the interpreter after translation to C with systems such as SWI-Prolog and the commercial Sicstus Prolog as well as P#, a Prolog to C# compiler. The results show that, despite its flexibility and portability, our approach compares reasonably well with other embedded Prologs, proving that the PyPy approach to interpreter construction can scale to other programming languages than Python itself.

Contents

1 Introduction

Efficient virtual machines for the Prolog programming language (and also for other dynamic high-level languages) are usually large and intricate pieces of software implemented in a low-level language such as C that are extensively optimized over many years and are therefore difficult to change in many respects. Due to their inherent complexity it is hard to develop them further, to maintain them or to use them to experiment with language extensions or new built-ins. Early implementation decisions, such as choice of garbage collector, are extremely hard to change afterwards. In addition, it is also much harder to perform advanced optimizations or transformations (such as partial evaluation, for example) on the virtual machine itself due to the low level of the implementation language. Furthermore it is (by definition) impossible to port them to a non-C platform such as the Java Virtual Machine or the .NET framework. Therefore the numerous implementations of a Prolog virtual machine integrated into one of these environments are all new implementations which have to be maintained themselves and therefore all have different levels of maturity and slightly different sets of built-ins available.

Using a higher-level language than C addresses many of these problems. It allows the language implementor to work at a higher level of abstraction, which eases implementation, maintenance and extensions of the virtual machine and reduces the size of the code base. Furthermore, advanced optimizations and transformations can be performed. To regain usable performance the implementation needs then to be translated to a low-level language again.

The goal of the PyPy project [32] is to support the implementation of interpreters in the high-level language Python [35]. Python is an object oriented dynamic language which is becoming more and more popular for scripting use as well as for “real” applications. PyPy provides a translation toolchain that helps to translate interpreters written in Python to a low-level language such as C and the .NET Intermediate Language. To make this translation work, these interpreters have to be written in RPython, which is a proper subset of Python chosen in such a way that it is amenable to analysis. PyPy also provides a lot of support for writing interpreters, especially libraries that are useful in such a context such as I/O-support, arbitrary-length integers and parsing. PyPy also contains a fully compliant interpreter for the Python language itself.

In this thesis we present a Prolog interpreter (called “Pyrolog”) written in RPython, exploring the possibilities of implementing a well-known declarative language in a high-level language (as opposed to C). [1] One of the main goals of the implementation was to keep it as simple and as extensible as possible (even at the cost of performance). The PyPy tool suite is used to reach a reasonable level of performance and to ensure portability to various platforms. The performance of our Prolog implementation is promising when compared to other embedded Prologs, but still distinctly below that of state-of-the-art Prolog engines.

This thesis is structured as follows: Sect. 2.1 and 2.2 give some background about Prolog and Python respectively. In Sect. 2.3 we introduce the PyPy project. In Sect. 3 we describe

[1]The full source of Pyrolog is part of the PyPy subversion repository and available at http://codespeak.net/svn/pypy/dist/pypy/lang/prolog/

the implementation of our Prolog interpreter in detail, while in Sect. 4 we empirically evaluate it on a series of benchmarks. In Sect. 5 we demonstrate some of the capabilities of our embedding on a few sample applications. In Sect. 6 we compare our approach with other virtual machine implementations particularly of Prolog. In Sect. 7 we present possible future approaches and conclude.

2 Background

2.1 Prolog

Prolog is a declarative logic programming language invented by Alain Colmerauer, Philippe Roussel and Robert Kowalski in 1972 [11]. The execution of Prolog programs is based on first order predicate calculus. The Prolog language is formally defined by the ISO Prolog standard [13]. Most of today's Prolog implementations use a more or less heavily modified version of Warren's abstract machine (WAM) [36] to interpret Prolog programs. The WAM is a virtual machine designed for running Prolog programs by compiling them first to the instruction set of the WAM and then executing those in some way. Typical ways to execute programs in the WAM instruction set are to write an emulator for the WAM (for example Sicstus Prolog) or to compile the WAM code to native code or to C code (wamcc [10]).

Another approach is taken by Bin-Prolog [30]. Bin-Prolog first does a source to source transformation of the Prolog program to use only binary terms using continuation-passing-style (see also Sect.3). The interpreter itself can only interpret such terms which makes its implementation considerably simpler. For an overview over Prolog implementations see Van Roy's paper [27].

Despite its power, Prolog is seldom seen as a true general purpose language. An example for this is that for many Prolog applications, the development of the graphical user interface is carried out outside of Prolog. On the other hand, many real world applications developed in imperative/object oriented programming languages require sophisticated logical reasoning, best provided by a Prolog engine. This is especially true for agent programming (see e.g. [15]) or semantic web applications, but also for program analysis (see, e.g., [14], which uses Prolog for the dynamic analysis of Java programs).

Interfacing Prolog to another programming language can be done in various ways. One common approach is to have a separate Prolog engine which communicates with the foreign language component (e.g. via socket communication or direct procedure calls). Examples of this approach are for example the Tcl/Tk interface of Ciao and Sicstus [28], the InterProlog [7] Java to Prolog interface, or the Java to Prolog Interfaces Jasper and PrologBeans of Sicstus [28]. Another approach is to actually embed Prolog in the target language (or the byte code of the target language). Examples of this approach are jProlog, PrologCafe, JIProlog, P#, Jinni and many more. The second approach provides easier integration with simpler communication and distribution, but usually comes at the cost of a speed penalty.

In this thesis we present a new instance of the second approach, where we embed Prolog in Python, and make use of novel implementation, compilation and specialization techniques to obtain flexibility and portability and a surprisingly low speed penalty (compared to state-of-the-art Prolog engines).

2.2 Python

Python [35] is an increasingly popular high-level object-oriented imperative programming language. It is dynamically type checked and strongly typed, meaning that there is almost no automatic type conversion. Python is very readable and the language contains built-in high-level data types. Python uses garbage-collection to manage its memory. Python is commonly used for text processing, web applications, rapid prototyping, scientific applications and many more.

Python was invented by Guido van Rossum in 1991 at the Stichting Mathematisch Centrum in Amsterdam. It has been continuously extended and improved since then. The reference implementation ("CPython" [34]) is written in C and uses reference counting for garbage collection.

The Python community seems to have a latent interest in logic programming, as can be seen by the fact that there are several projects that try to bring logic (and constraint) programming capabilities to Python. Examples for these are "YieldProlog" [22] and some implementations of Prolog interpreters in Python, for example [20], [3]. These Prolog implementations in Python are mostly naive and not really intended for realistically sized Prolog programs, especially due to the fact that they do no tail call elimination and therefore overflow the limited Python stack rather quickly.

2.3 The PyPy Project

The PyPy project was started to implement a next-generation interpreter for the Python language in Python itself. In the process of doing that, a very general compiler tool suite was implemented. This tool suite is able to translate RPython to other languages, mostly low-level ones. RPython, "Restricted Python", is a less dynamic subset of the Python language, restricted in such a way that it is possible to perform type inference and code generation (general Python code is too dynamic to allow type inference [9]). The translation toolchain reuses the Python interpreter to perform abstract interpretation of RPython programs [25]. Currently backends for C/Posix, LLVM [19] and CLI/.NET are finished and backends for Java, Common Lisp, Smalltalk/Squeak and JavaScript are being developed.

PyPy's translation toolchain does not only translate RPython code to different languages but is also able to "weave in" a number of translation aspects [5] such as various garbage collection strategies or a different way to handle the stack frames into the result, which makes it possible to keep the VM implementation free of these and other low level details. This is one of the advantages of implementing an interpreter in a high-level language: These low-level details are not present in the interpreter implementation and therefore can be changed during the translation whereas many interpreter implementations in a low-level language encode a fixed choice for such details so that they cannot be changed easily. A good example is the reference counting of CPython, which makes its presence many times in every C file. Changing the choice of garbage collection strategy would require pervasive changes throughout the interpreter.

The variety of available target platforms, translation aspects and implemented optimizations make RPython especially interesting for the implementation of interpreters. It is enough to maintain one code base and obtain interpreters running on a number of different platforms, which decreases the maintenance effort and lets all the various versions benefit from new features and optimizations. Also the high level of abstraction that RPython offers makes it easier to concentrate on the core complexities of implementing a language.

Another advantage of this approach is that since RPython is a proper subset of Python, an RPython program can be fully tested and easily debugged by letting it run on a Python interpreter. The program is only translated to a different language if it is reasonably bug-free. This increases testability, eases development and decreases turnaround times.

3 Interpreter Implementation

Pyrolog is a Prolog interpreter implementing the ISO-Prolog [13] semantics including unification (with occurs check, when the `unify_with_occurs_check` built-in is used), depth first search and the cut. It supports ISO exception handling and a form of multi-argument indexing. It includes a fair number of built-ins, including `current_op` and `op` to change operator precedence or define new operators. It is written completely in RPython and is translatable to C and .NET using the PyPy translation toolchain (although there are some technical problems left when translating to .NET that we expect to resolve in the future, see Sect.3.7).

The foremost goal of the interpreter implementation was simplicity and flexibilities at all levels to allow for easy experimentation with language features. We consciously chose not to use the WAM architecture [36] and not even to use a bytecode emulator architecture to avoid the complexity of designing an instruction set and writing a compiler. See Fig. 1 for an overview of the data structures of the interpreter.

3.1 Data Model

In Prolog, a term can be either an integer, a floating point number, an atom, a variable or a compound term. We have implemented each as a subclass of an abstract base class `PrologObject`. Integers, floating point numbers and atoms are implemented as classes that contain Python integers, floats and strings respectively (which in turn are translated to their low-level equivalent by the translation toolchain). Compound terms are implemented as a class that contains the functor of the compound term (as a Python string) and an array of `PrologObject`s which are the arguments of the compound term.

Variable bindings are maintained using a union-find data structure: Variables are implemented as a class that contains an index (a Python integer) into an array of variable bindings. If a variable is not bound, the array contains `None` at the indexed position, otherwise it contains the object the variable is bound to. If a new variable is created,

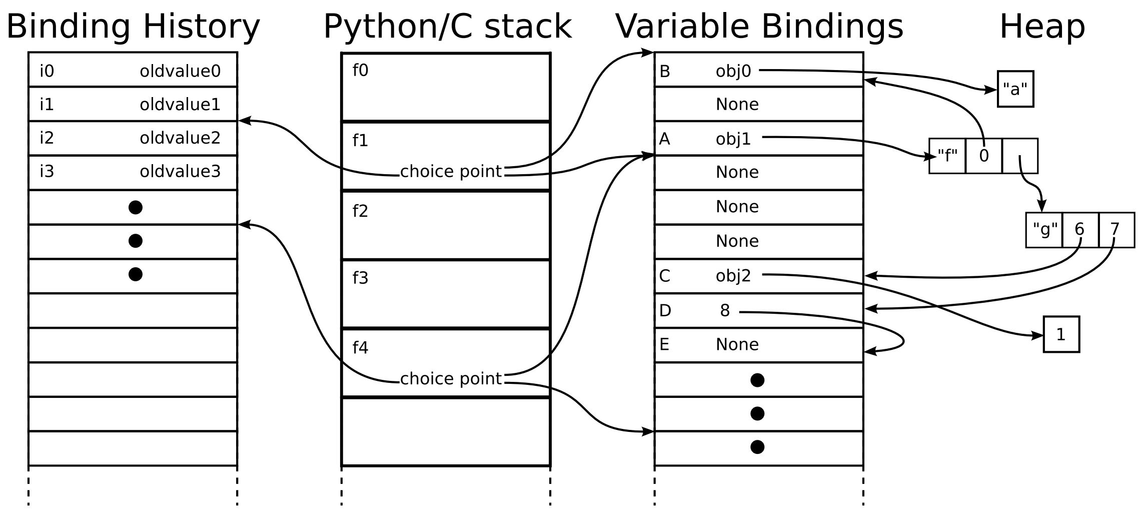


Figure 1: Memory layout of the interpreter

this array is made one element longer (internally its size is doubled, so that this is an amortized cheap operation) and the variable gets the index of this new variable. When a variable is being dereferenced, *path compression* is performed to keep the tree of variable bindings flat. The array of variable bindings is encapsulated into a frame object, which is also responsible for the trail. To be able to undo variable bindings on backtracking, the frame keeps track of the previous value if the binding of a variable is changed. This is done by a stack of tuples of the index of the changed variable and the previous value.

Since variables are very common objects it is wasteful to have to allocate memory for all the variable objects, especially as they just contain indices into an array anyway. Therefore a hint for the translation toolchain is used to use a tagged pointer for variable instances for platforms where this is supported. This hint can be given for classes which have exactly one integer attribute. Instances of these classes then become (for example when translating to C) pointers that have their lowest bit set and that contain the value of the integer attribute in the rest of the bits [6].

The "Heap" and the "Variable Bindings" part of Fig. 1 show a structure that could have been created by: `A = f(B, g(C, D)), B = a, C = 1, D = E`.

Unifications of these terms is implemented recursively as a method on the various Prolog term classes. The method takes the implicit `self` argument of Python methods, the other term, a frame that holds the variable bindings and a boolean argument which specifies whether to perform the occurs check. If the unification does not succeed, a `UnificationFailed` exception is raised. To prevent having to continuously check the flag argument while unifying two terms, a hint is given to the translation toolchain to *specialize* the unify methods by this last argument (this means that the toolchain will automatically create two versions of all the unify methods, one where the occurs check is enabled and one where it is not).

Fig. 2 shows the implementation of the unify method of the `Atom` class. The argument `occurs_check` signifies whether the occurs check should be used. The line after the

```
class Atom(Callable):
    ...

    def unify(self, other, frame, occurs_check=False):
        if isinstance(other, Var):
            return other.unify(self, frame, occurs_check)
        elif isinstance(other, Atom):
            if other.name != self.name:
                raise UnificationFailed
            return
        raise UnificationFailed
    unify._annspecialcase_ = "specialize:arg(3)"

    ...
```

Figure 2: Implementation of the unification of atoms

method definition is the above-mentioned hint that the function should be specialized by its last argument.

3.2 Interpretation

The Prolog interpreter is implemented in a recursive way which means that recursion of the user program is mapped onto recursion of the interpreter (see Sect. 3.4 for how stack overflow is prevented). Regular (user defined) functions are stored in a dictionary with their signature as keys. The values in the dictionary are an array of all rules with that signature. A call is performed by retrieving the list of applicable rules from the dictionary and trying them one by one. If the call structure unifies with the head of the rule, the body of the rule (if any) is called. To prevent variable clashes, a copy of the rule with new variables is created (standardizing apart) before unification and execution of the body. By default, unification *without* occurs check is used (as is the case for almost all Prolog systems) but it would be easy to change that default to use the occurs check.

The interpretation of Prolog programs is performed in such a way that backtracking can be done by raising an exception on the interpreter level. This unwinds the stack up to a choice point and then execution is continued. This makes it possible to use the interpreter call stack as a choice point stack too. A choice point is thus represented as an exception handler on the call stack of the interpreter and some information (stored in a local variable) that characterizes the state of the trail to be able to undo variable bindings done after the choice point. The exception handler catches `UnificationFailed` exceptions, then reverts variable bindings and then chooses another option if there is one.

To characterize the state of the trail at a given point in time the following information is needed: The number of currently needed variables in the array of variables, the size of the variable bindings history and the number of needed variables at the *last* choice point. The

```
class AndContinuation(engine.Continuation):
    def __init__(self, next_call, continuation):
        self.next_call = next_call
        self.continuation = continuation

    def call(self, engine):
        next_call = self.next_call.dereference(engine.frame)
        return engine.call(next_call, self.continuation)

def impl_and(engine, call1, call2, continuation):
    and_continuation = AndContinuation(call2, continuation)
    return engine.call(call1, and_continuation)
```

Figure 3: Implementation of *and*

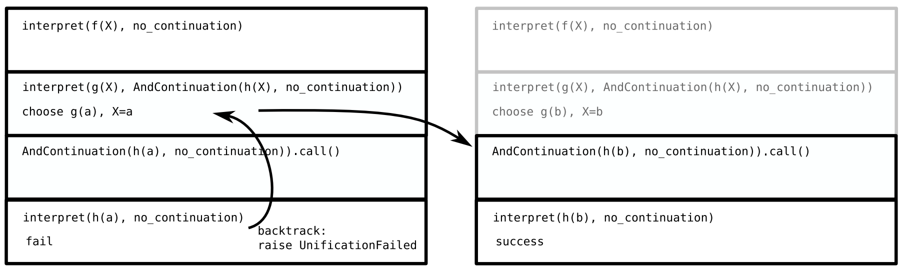
number of currently needed variables is required to shrink the array of variables when backtracking to its old size. The size of the bindings history stack is needed to undo all the bindings that were done afterwards. The number of variables of the last choice point is used for an optimization: if a variable does not exist before the last choice point, its binding history does not need to be saved, since it will be deleted when backtracking.

To be able to implement backtracking as stack unwinding it is necessary to lay out the interpretation of Prolog calls on the stack in the correct way. This means that the interpretation of Prolog calls has to be ordered on the stack in such a way that this order matches the order of backtracking. To make this possible the interpreter is implemented in a continuation passing style (CPS). In addition to the actual Prolog call the Python function interpreting the Prolog call takes a continuation argument. If the interpreted function unifies with a fact (that is, a Prolog function without body) the continuation is called, otherwise the body of the rule is being interpreted and the continuation argument passed on. New continuations are created and passed on most importantly in the case of conjunctions of two calls. The first call of the conjunction is interpreted immediately. The continuation argument for the interpretation of this first call will interpret the second call with the continuation that was passed in to the interpretation of the conjunction. See Fig. 3 for the implementation of the ',' (conjunction) built-in.

The result of how programs are interpreted using this scheme is effectively equivalent to the explicit CPS transformation described by Tarau and Boyer [31] and used for implementing BinProlog [30], except that their CPS transformation explicitly modifies the Prolog source program whereas in our case the transformation is only reflected in the architecture of the interpreter. Figure 4 shows an example of the explicit transformation. In the left column is a Prolog program, in the right column the same program transformed to CPS. Figure 5 shows the interpreter stack (omitting calls of helper functions) when interpreting the call `f(X)`.

```
g(a).                          g(a, Cont) :- call(Cont).
g(b).                          g(b, Cont) :- call(Cont).
h(b).                          h(b, Cont) :- call(Cont).
f(X) :- g(X), h(X).            f(X, Cont) :- g(X, h(x, Cont)).
```

Figure 4: Example of the explicit CPS transformation



interpret(f(X), no_continuation)
interpret(g(X), AndContinuation(h(X), no_continuation))
choose g(a), X=a
AndContinuation(h(a), no_continuation)).call()
interpret(h(a), no_continuation)
fail
backtrack:
raise UnificationFailed
interpret(f(X), no_continuation)
interpret(g(X), AndContinuation(h(X), no_continuation))
choose g(b), X=b
AndContinuation(h(b), no_continuation)).call()
interpret(h(b), no_continuation)
success

Figure 5: Interpreter stack when interpreting the call `f(X)` of Fig. 4

3.3 The Cut

The cut is implemented as another exception which unwinds the interpreter stack. The exception contains the continuation which needs to be called after the cut. This exception will be caught by an exception handler around the call to the interpreting function of the goal where the cut appears in. The exception handlers for the `UnificationFailed` exception (which is raised if a goal fails) do not catch the cut exception so that the bindings are not undone. After the cut exception is caught the contained continuation is called.

3.4 Stackless

To prevent stack overflows in the presence of deep recursion in the interpreted program (common in Prolog), a specific translation aspect of the PyPy translation toolchain is used, which is called the "stackless transformation" [5, 6]. PyPy provides the possibility to transform all flow graphs of a program in such a way that there is a check around every function call that notices when the C stack is about to overflow. If this is the case, the stack is unwound by raising a special exception. During unwinding, all the information contained in every stack frame is stored into a specific structure on the heap which contains the same information as the original proper stack frame. These structures are stored in a linked list. The unwinding stops at the last function of the stack, which is a dispatcher loop. This dispatcher continues the execution by restoring the last frame in the frame chain (which was active when the unwinding started) and then continuing execution in the corresponding function. This is the situation shown in Fig. 6. This restarted function executes normally, can call other functions in a regular way. If it returns eventually the dispatcher will restore the next function in the frame chain on the heap and so on.

The same behaviour is usually implemented using a complete transformation into con-

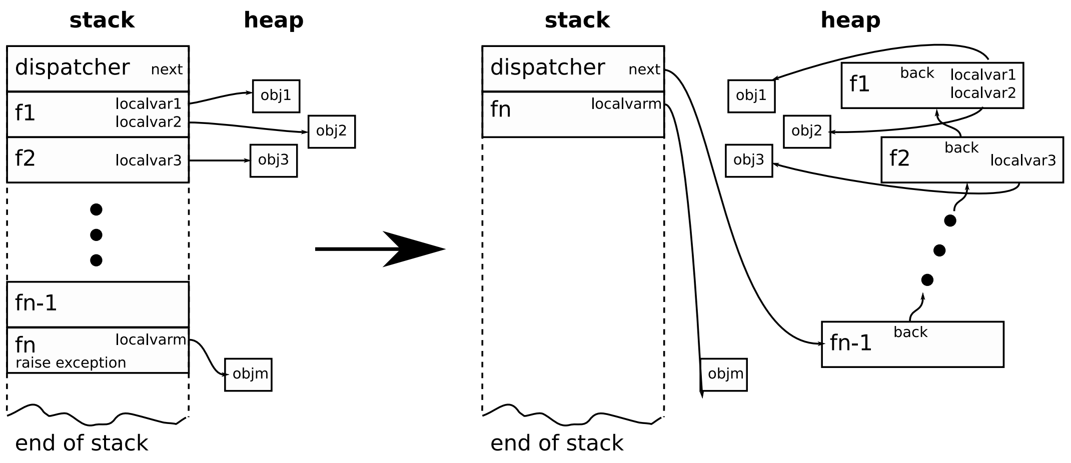


Figure 6: Storing the stack to the heap when it is about to overflow

tinuation passing style. This has the drawback that the resulting code is in a style quite different to what compilers expect, and which renders most compiler optimizations off the produced code useless. In addition, all the code has to pay the performance penalty, not only code that is deeply recursive.

The stackless transformation makes it possible to have arbitrarily deep recursion in the interpreted program (of course bounded by the amount of heap space, although the heap frames usually take a bit less space than the regular stack frames).

A peculiarity of the stack unwinding process is that for some functions no heap frame is created. This is possible if the function in question is currently executing a tail call and would directly return the result of the call. In this case it is not necessary to create a heap frame at all, because execution can just as well continue in the caller when the tail call is finished. This is equivalent to a tail call optimization for the RPython program. The tail call is not optimized immediately when the call is made but the unnecessary stack frame is not created on the heap when the stack is being unwound. This could be described as using the C stack as a cache for the chain of frames on the heap. When the cache gets full it is compacted and put on the heap. This property can be used to implement a (somewhat limited) form of tail call optimization by making sure that in the case of tail calls of the interpreted program the interpreter uses tail calls too.

The stackless transformation makes it possible to implement the interpreter in a naive recursive way and not having to explicitly spell out a more stack-friendly way of implementing recursion. Java Café [2] (a Prolog to Java compiler) and P# [12] (a Prolog to C# compiler) use an explicit dispatcher in their generated code to prevent stack overflows. Being able to use recursion directly increases readability and maintainability of the interpreter code-base. The technique also has interesting speed benefits, since most of the time regular calls are used, allowing normal compiler optimizations which expect such calls to work to their full potential.

3.5 Multi-Argument Indexing

To minimize the number of choice points created and to prevent resolution with rules that can never succeed, a limited form of indexing is implemented. The indexing is limited in the sense that it still performs a linear search over the applicable rules but is able to quickly filter out rules that would not unify with the call anyway. If no rule remains after this filtering, then the call fails immediately. If exactly one rule remains then no choice point needs to be created, otherwise a choice point is needed.

To do this filtering, every rule has a list of hashes (integers) associated. The list has as many arguments as the head of the rule has arguments. Every argument of the head of the rule is hashed in the following way: Variables are all hashed to zero. For all other terms, the lowest three bits of the hash are a tag that is unique for every type of term (number, float, atom, compound). The remaining bits are used to store a hash of the term. For number and floats a part of the bits of the real object are used. For atoms the string of the atom is hashed using Python's built-in hash function. For compound terms the signature as a string is hashed. Two terms can only be unified, if they have the same hash assigned or one of them is a variable (the hash is zero). To decide whether two compound terms that have the same functor and the same number of arguments can unify at all the lists of hashes of their arguments is examined. If all corresponding hashes are either equal or one of them zero the terms can potentially be unified.

3.6 Built-in

In addition to the core of the interpreter that interprets user programs, a decent number of Prolog built-ins have been implemented, such as arithmetic built-ins, `assert` and `retract`, type checks, `call`, `current_op` and `op` to access the operator precedence. Of the 364 test cases of the INRIA test suite [16] 281 pass, which is about 77%.

All these built-ins are implemented in RPython. Special care was taken to make the implementation of built-ins easy and pleasant to encourage the implementation of application-specific built-ins. See Fig. 7 for the implementation of the `between` built-in (a non-standard built-in implemented by SWI-Prolog). The `expose_builtin` call after the function definition registers `impl_between` as a built-in. The `unwrap_spec` argument call specifies the type of the arguments. The two `"int"` specifications mean that if the `lower` or `upper` argument to `between` are not integers, a type error will be raised automatically. If they are integers, they are automatically unboxed so that the arguments passed into `impl_between` are already RPython-level integers. The type specification of the `varorint` argument is `"obj"` which means that no special action is performed on the argument.

3.7 Translating the Interpreter

To translate the Prolog interpreter to C, the PyPy translation tool suite does type inference on the interpreter, performs the stackless transformation and then emits and compiles C code. The whole process takes roughly 2-3 minutes (see 4 for details on the hardware used). The whole interpreter source code including the built-ins is about 2500 lines of

```
def impl_between(engine, lower, upper, varorint, continuation):
    oldstate = engine.frame.branch()
    for i in range(lower, upper):
        try:
            varorint.unify(Number(i), engine.frame)
            return continuation.call(engine)
        except error.UnificationFailed:
            engine.frame.revert(oldstate)
    varorint.unify(Number(upper), engine.frame)
    return continuation.call(engine)
expose_builtin(impl_between, "between",
               unwrap_spec=["int", "int", "obj"],
               handles_continuation=True)
```

Figure 7: Implementation of the `between` built-in

Python source code (of which 1800 are for the core interpreter, the rest for built-ins), which is translated into 14000 lines of C code. The resulting executable also includes an interactive interpreter to query the Prolog system.

The translation adds a couple of translation aspects to the code during the translation process. An example for this is the used garbage collector: by default the conservative Boehm GC is used [4]. It is also possible to choose a different garbage collection strategy at compile time, such as reference counting or an exact mark-and-sweep collector.

It is also possible to translate the interpreter to the Intermediate Language (IL) of the .NET framework. This does not yet fully work and gives a Prolog interpreter which can only be used for small programs. The problem is that it is not yet possible to use the stackless transformation together with the IL backend. Therefore most realistic Prolog programs lead to a stack overflow of the underlying VM. However, we are working on porting the stackless transformation to be usable with the IL backend too.

4 Experimental Results

To evaluate the speed of our interpreter we used various benchmarks and compared Sicstus Prolog, SWI-Prolog and P# [12] (a Prolog to C# compiler) against a version of our interpreter that was translated to C. We choose Sicstus Prolog and SWI-Prolog because they are well established and mature Prolog systems and P# as a representative for the approach of embedding Prolog into the target environment.

The benchmark "arithmetic" is a self-written arbitrary precision integer benchmark using lists of bits to represent numbers and calculates 13!. The benchmarks "recurse", "recurse_with_assert", "recurse_with_call", "recurse_with_choice", "recurse_with_cut" are microbenchmarks, see Fig. 1. The actual call which is benchmarked is `f(1000000)`. All the benchmarks were run on a Intel(R) Pentium(R) M 1400 MHz with 1024 KB cache size and 512 MB RAM. The used version of the Prolog systems here: Sicstus 3.12.5 (x86-linuxglibc2.3), SWI-Prolog Version 5.2.13 and P# 1.1.3 under Mono 1.1.13.6.

Not all benchmarks ran on P#: "arithmetic" didn't run because the built-in `mod` is not supported. The compilation of the "quicksort" benchmark took more than four hours, after which we stopped it. It is possible that the reason for this is that P# has problems parsing the list of 20000 elements contained in the source file.

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

recurse_with_assert:	`f(X) :- assert(g(0)),`
	`assert((g(A):-`
	`        (B is A-1,g(B)))),`
	`g(X).`

recurse_with_call:	`f(0).`
	`f(X) :- Y is X-1,call(f(Y)).`

recurse_with_choice:	`f(0).`
	`f(X) :- Y is X-1, f(Y).`
	`f(X) :- Y is X-2, f(Y).`

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

Table 1: Microbenchmarks

The results indicate that Pyrolog has roughly the same speed as P#, depending a bit on the benchmark. Sicstus Prolog seems to be three times as fast when interpreting Prolog code and roughly twenty times as fast for compiled Prolog code. For SWI-Prolog the factor is around ten.

In addition we also did some more benchmarks for some of the embedded Prologs Prolog Café 0.9.1, tuProlog 1.3.0 and again P# 1.1.3. For P# and Prolog Café we did not only try the compiled version but also the built-in Prolog interpreter of these systems and tuProlog is a pure interpreter anyway. We used only the "nrev" benchmark with a list of 300 elements. For the results see Table 2. The results are encouraging: Apart from P#, Pyrolog is significantly faster than most of the other systems, especially the interpreting

Prolog system	Benchmark Time	Factor against Pyrolog
Pyrolog	0.316s	1.000 ×
Prolog Café		
compiled	2.536s	8.0253 ×
interpreted	71.978s	227.778 ×
P#		
compiled	0.331s	1.047 ×
interpreted	9.893s	31.307 ×
tuProlog		
interpreted	83.394s	263.905 ×

Table 2: Benchmark results for nrev(300 elements)

ones.

Benchmark	**Pyrolog**	**Sicstus compiled**	**Sicstus interpreted**	**SWI compiled**	**P# compiled**
arithmetic	22.6066s (1.0×)	0.9209s (0.0407×)	6.6169s (0.2927×)	2.0085s (0.0888×)	–
crypt	0.0087s (1.0×)	0.0019s (0.2139×)	0.0063s (0.7230×)	0.0049s (0.5698×)	0.3840s (44.2756×)
nrev (1000 elements)	2.4919s (1.0×)	0.0627s (0.0251×)	1.1622s (0.4664×)	0.2679s (0.1075×)	0.9890s (0.3969×)
poly	3.2120s (1.0×)	0.2288s (0.0712×)	1.4972s (0.4661×)	0.6653s (0.2071×)	4.6560s (1.4496×)
qsort (20000 elements)	7.5915s (1.0×)	0.1331s (0.0175×)	1.3743s (0.1810×)	0.5201s (0.0685×)	–
tak	0.5348s (1.0×)	0.0138s (0.0259×)	0.3284s (0.6140×)	0.0801s (0.1498×)	0.4000s (0.7479×)
recurse	5.3496s (1.0×)	0.0799s (0.0149×)	2.0929s (0.3912×)	0.7252s (0.1356×)	3.5490s (0.6634×)
recurse_with_assert	4.5843s (1.0×)	1.5407s (0.3361×)	1.5740s (0.3433×)	0.8333s (0.1818×)	–
recurse_with_call	10.4126s (1.0×)	0.6797s (0.0653×)	2.4998s (0.2401×)	0.1517s (0.0146×)	35.4090s (3.4006×)
recurse_with_choice	10.5789s (1.0×)	0.7916s (0.0748×)	4.5501s (0.4301×)	2.4327s (0.2300×)	6.0460s (0.5715×)
recurse_with_cut	4.9711s (1.0×)	0.2822s (0.0568×)	2.7267s (0.5485×)	0.8006s (0.1611×)	4.3480s (0.8747×)

Table 3: Benchmark results (in seconds)

5 Extending the Prolog interpreter

As proofs of concept, we implemented two small extensions to the library of Pyrolog and wrote a small demo application: we added an imperative hashmap datatype to Pyrolog and implemented a link checker in Pyrolog which uses functions of the Python standard library—exposed as special built-ins—to download and parse HTML pages. In addition we implemented a Tic-Tac-Toe game by writing the game logic in Prolog and the GUI in Python.

5.1 Adding an Imperative Hashmap Data-Type

To give the Prolog user access to the RPython dictionary data-type (essentially a hash-map) we implemented the following built-ins: `create_hashmap(H)`, `setitem_hashmap(H,Key,Value)`, `getitem_hashmap(H,Key,Value)` as well as `delitem_hashmap(H,Key)`. The hashmap objects created by these functions behave

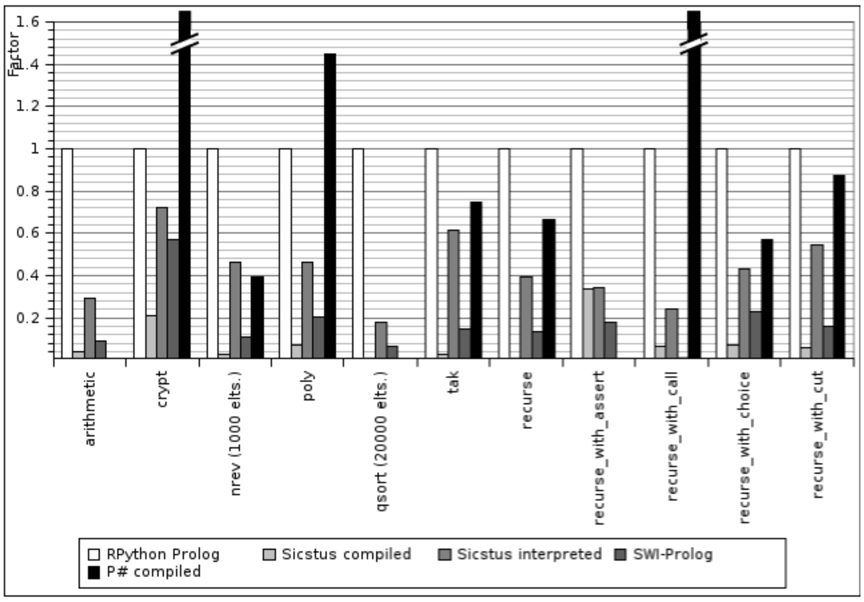


Figure 8: Graph showing relative benchmark times

essentially in an imperative manner: they have backtrackable destructive update and can thus be modified without creating a copy. If backtracking happens to go through any of the mutating functions, the mutations are undone (similar to the semantics of the `setarg` built-in provided by several Prolog implementations). See for example Fig. 9 for the implementation of `setitem_hashmap` where if the continuation fails the setting of the value is undone.

```
def impl_setitem_hashmap(engine, hashmap, key,
                         value, continuation):
    old = hashmap.setitem(key, value)
    try:
        return continuation.call(engine)
    except error.UnificationFailed:
        hashmap.setitem(key, old)
        raise
expose_builtin(impl_setitem_hashmap, "setitem_hashmap",
               unwrap_spec=["obj", "atom", "obj"],
               handles_continuation=True)
```

Figure 9: Implementing setting of an item in a hashmap

5.2 Development of a Link-checker

To access web sites and their content from within Prolog we wrote three special built-ins in Python: `fetch_url(URL, Content)` to extract the full content of a web page. The predicate fails if the web page at the given link does not exist. The built-in `extract_links(Content, Link)` uses a regular expression to extract all the links contained in the document and unifies them one after the other with Link. The built-in `url_split(URL, Transport, Server, Path)` can be used to split a URL into its parts. The link checker searches the tree of links for a dead link, asserting `already_visited(URL)` for every visited site to avoid getting into a loop. To not search the whole internet (assuming of course that the graph of all web pages is connected and reachable by the starting page), it never scans pages not on the original server and only checks whether links to them are valid. See Fig. 10 for the complete Prolog code.

```
prefix(P, S) :- append(P, _, S).

already_visited(none).

find_dead_link(URL, Loc, OnlyUnderPath, From, Info) :-
    already_visited(URL), !, fail.
find_dead_link(URL, Loc, OnlyUnderPath, From, Info) :-
    \+ fetch_url(URL, _), Info = dead(From, URL), !.
find_dead_link(URL, Loc, OnlyUnderPath, From, Info) :-
    print('checking '), print(URL), nl,
    assert(already_visited(URL)),
    url_split(URL, _, Loc, Path), prefix(OnlyUnderPath, Path),
    fetch_url(URL, Content), extract_links(Content, NewURL),
    find_dead_link(NewURL, Loc, OnlyUnderPath, URL, Info).
```

Figure 10: Link Checker

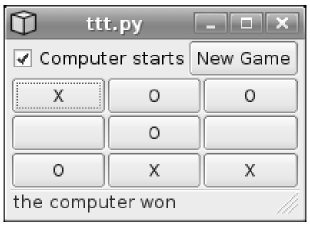


Figure 11: The game GUI

5.3 Development of a GUI

For the Tic-Tac-Toe game we implemented a simple GUI using the GTK toolkit via the PyGTK wrapper and the Gazpacho GUI builder. See Fig. 11 for a screen shot of the game. The game logic as well as the (simple) computer AI are written in Prolog. The GUI is written in Python and queries the Prolog interpreter to determine whether the game has ended, who won the game and what move the computer player should make. To give an impression of the style of the code that is involved in the Python- Prolog interaction see Figures 12 and 13. Figure 12 shows the Prolog code that is used to determine whether the game is over and who has won (the grid of boxes is numbered as a magical square, so that the sum of all rows, columns and diagonals is 15. This allows checking for the end of the game by seeing whether any one player has placed three marks whose numbers add up to 15). Figure 13 shows the Python code which calls the `end_state` function to find out whether the game has ended and who won it. Currently the interface between Prolog and Python is still somewhat rough and we hope to improve it in the future.

```
choose_moves([UserMoves, ComputerMoves], user, UserMoves).
choose_moves([UserMoves, ComputerMoves], computer, ComputerMoves).

end_state([UserMoves, ComputerMoves], draw) :-
    length(UserMoves, X), length(ComputerMoves, Y), 9 is X + Y.
end_state(State, Who) :-
    choose_moves(State, Who, Moves),
    member(A, Moves),
    member(B, Moves), A \= B,
    member(C, Moves), A \= C, B \= C, 15 is A + B + C.
```

Figure 12: Logic to decide whether the game has ended.

```
def who_won():
    global game_ended
    try:
        How = term.Var(0)
        query = wrap(("end_state", [usermoves, computermoves], How))
        engine.run(query)
        game_ended = True
        return unwrap(How, engine)
    except UnificationFailed:
        return False
```

Figure 13: Python code to determine who won the game

6 Related Work

Implementing Prolog in other programming languages dates back a long time (see Campbell's book [8] for various example implementations). In more recent times there have been plenty of efforts linking Prolog up with Java and the .NET framework, such as PrologCafe [2], JIProlog, P# [12], Jinni [33] and many more. Cook [12] compares P# with various other offerings, such as Minerva, jProlog and Jinni.

Using high-level languages to implement virtual machines has been done many times. Notable examples are Scheme48 [18], a scheme implementation written in PreScheme, a restricted subset of Scheme. PreScheme is meant to be translated to C only; Squeak [17], a Smalltalk virtual machine implemented in SLang, which is a severely restricted subset of Smalltalk which is directly translatable to C; JavaInJava [29] is a project to implement a Java VM in Java itself, although without any means for bootstrapping. This means that JavaInJava always needs an underlying Java VM and therefore remains a mostly academic exercise. More practical in this respect is the Jikes RVM [1] which implements a just-in-time compiler for Java which is able to bootstrap itself by self-application.

A Prolog VM which is implemented in an imperative extension of Prolog is described by Morales, Carro, and Hermenegildo [21]. This imperative extension is also translatable to C and is amenable to advanced optimization and transformation techniques such as partial evaluation. This project comes closest to our goals and techniques used.

The drawback of all these efforts is that the translation process to a more efficient low-level language (if it happens at all) is not meant to be re-targeted, often either C or assembler are assumed as a fixed target. None of these are able to target one of the Java or .NET platforms and the C/Posix platform at the same time.

7 Future Work

The most important step in improving Pyrolog is to complete the set of built-ins to allow more programs to be executed. Also more advanced Prolog features such as coroutining, constraint programming and a module system are used by many Prolog programs so that adding them would be a worthwhile endeavour.

Further plans would be to speed up the interpreter using classical techniques for example by implementing it as a virtual machine with an instruction set and compiling Prolog programs to that.

Another direction of research is to bring Python and Prolog closer together. Currently it is possible to call Python functions from Prolog by exposing them explicitly (which includes code that converts the arguments) and possible to call Prolog functions from Python by explicitly converting the arguments to Prolog objects. While this is comparatively simple there could still be work to make the interaction more programmer-friendly. For example there could be a general way to call Python functions from Prolog without the need to expose them first manually. Also mechanisms for calling Prolog code from Python and having the arguments be converted as closely as possible to Prolog objects where that is possible would ease the development of cross-language programs.

A translation aspect that is currently being explored in the PyPy project [26] is the automatic generation of a Just-In-Time specializing compiler out of an interpreter. A prototype for such a specializing compiler for the Python language is the Psyco project ([24], [23]), which was written manually and works as an extension module to CPython. Work is being done on generating such specializing Just-In-Time compilers with a non-trivial transformation step during the translation process given a small set of hints in the interpreter source code. So far this works very well for small interpreters and also quite well for PyPy's Python interpreter, where functions using only integer arithmetic can be speeded up by factors of up to 60. We plan to experiment with this technique and to apply it to Pyrolog in the future.

8 Conclusion

In summary we have presented a proof-of-concept of a new way to implement programming languages. Using this approach we developed a very flexible and still not too inefficient Prolog interpreter in RPython. Although the performance is below that of high-performance Prologs, compared to P# and other embedded Prologs the speed of our interpreter is competitive, especially considering the fact that our Prolog interpreter is deliberately quite simple and straightforward. Some of the performance can be attributed to PyPy's stackless transformation, which is both efficient and also frees the interpreter implementor from worrying about stack overflows in the presence of deep recursion. In the future we intend to incorporate optimizations into the Prolog VM implementation and thus drawing closer in performance to existing Prolog implementations.

We have demonstrated, by adding two small extensions to the language and by writing a program in a mixture of Python and Prolog, that it is relatively easy to interface the Prolog engine with Python and to expose new built-ins to Prolog that are written in Python. In future work we plan to simplify this interfacing work even more. We hope that our work will enable interesting new applications, combining the advantages of Python with the logical reasoning capabilities of Prolog.

Our work also provides the first example of the PyPy approach to interpreter construction scaling to interpreters for other programming languages than Python itself. Furthermore, PyPy's specializer proved useful (for compiling in occur checking, if desired). We also hope that the performance of our implementation will improve further once PyPy's just-in-time specializer will be fully available.

References

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Architecture and policy for adaptive optimization in virtual machines. Technical Report 23429, IBM Research, November 2004.

[2] M. Banbara. *Design and Implementation of Linear Logic Programming Languages*. PhD thesis, The Graduate School of Science and Technology of Kobe University, September 2002.

[3] S. Berger. Pythologic – prolog syntax in python. http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/303057.

[4] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, 1988.

[5] C. F. Bolz and A. Rigo. Memory management and threading models as translation aspects – solutions and challenges. Technical report, PyPy Consortium, 2005. http://codespeak.net/pypy/dist/pypy/doc/index-report.html.

[6] C. F. Bolz and A. Rigo. Support for massive parallelism, optimisation results, practical usages and approaches for translation aspects. Technical report, PyPy Consortium, 2006. http://codespeak.net/pypy/dist/pypy/doc/index-report.html.

[7] M. Calejo. InterProlog: Towards a declarative embedding of logic programming in Java. In J. J. Alferes and J. A. Leite, editors, *JELIA*, volume 3229 of *Lecture Notes in Computer Science*, pages 714–717. Springer, 2004.

[8] E. J. A. Campbell. *Implementations of Prolog*. Ellis Horwood/Halsted Press/Wiley, 1984.

[9] B. Cannon. Localized type inference of atomic types in python. Master's thesis, California Polytechnic State University, San Luis Obispo, June 2005.

[10] P. Codognet and D. Diaz. WAMCC: Compiling prolog to c. In *International Conference on Logic Programming*, pages 317–331, 1995.

[11] A. Colmerauer and P. Roussel. The birth of prolog. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 37–52, New York, NY, USA, 1993. ACM Press.

[12] J. Cook. P#: A concurrent Prolog for the .net framework. In *Software: Practice and Experience 34(9)*, pages 815–845. John Wiley & Sons, Ltd, 2004.

[13] P. Deransart, L. Cervoni, and A. Ed-Dbali. *Prolog: the standard: reference manual*. Springer-Verlag, London, UK, 1996.

[14] Y.-G. Guéhéneuc, R. Douence, and N. Jussien. No Java without caffeine: A tool for dynamic analysis of Java programs. In *ASE*, pages 117–. IEEE Computer Society, 2002.

[15] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. C. Meyer. Agent programming in 3apl. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[16] J. Hodgson. Test suite of conformance to the ISO Prolog standard based on the formal specification, 1999. http://pauillac.inria.fr/~deransar/prolog/inriasuite.tar.gz.

[17] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: the story of Squeak, a practical Smalltalk written in itself. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 318–326, New York, NY, USA, 1997. ACM Press.

[18] R. Kelsey. Pre-scheme: A scheme dialect for systems programming, 1997.

[19] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Mar 2004.

[20] C. Meyers. Prolog in python. http://ibiblio.org/obp/py4fun/prolog/prolog1.html.

[21] J. F. Morales, M. Carro, and M. Hermenegildo. Towards description and optimization of abstract machines in an extension of prolog. In *Pre-Proceedings of Logic Based Program Synthesis and Transformation*, pages 62–78, 2006.

[22] N.N. Yield prolog. http://yieldprolog.sourceforge.net.

[23] A. Rigo. Psyco. http://psyco.sourceforge.net.

[24] A. Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26, New York, NY, USA, 2004. ACM Press.

[25] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Proceedings of the 2006 conference on Dynamic languages symposium*, 2006.

[26] A. Rigo and S. Pedroni. DRAFT JIT compiler architecture. Technical report, PyPy Consortium, 2007. http://codespeak.net/pypy/dist/pypy/doc/index-report.html.

[27] P. V. Roy. 1983–1993: The wonder years of sequential prolog implementation. *Journal of Logic Programming*, 19,20:385–441, 1994.

[28] S. SICS, Kista. *SICStus Prolog User's Manual.* Available at `http://www.sics.se/sicstus`.

[29] A. Taivalsaari. Implementing a java virtual machine in the java programming language, 1998.

[30] P. Tarau. BinProlog: a continuation passing style Prolog engine. In M. Bruynooghe and M. Wirsing, editors, *PLILP*, volume 631 of *Lecture Notes in Computer Science*, pages 479–480. Springer, 1992.

[31] P. Tarau and M. Boyer. Elementary logic programs. In *Proceedings of Programming Language Implementation and Logic Programming, number 456 in Lecture Notes in Computer Science*, pages 159–173. Springer, August 1990.

[32] The PyPy development team. PyPy. An Implementation of Python in Python. http://codespeak.net/pypy.

[33] S. Tyagi and P. Tarau. A Most Specific Method Finding Algorithm for Reflection Based Dynamic Prolog-to-Java Interfaces. In I. Ramakrishan and G. Gupta, editors, *Proceedings of PADL'2001*, Mar. 2001. Springer-Verlag.

[34] G. van Rossum et al. Cpython. http://www.python.org/.

[35] G. van Rossum et al. Python language reference, 2006. http://docs.python.org/ref/ref.html.

[36] D. Warren. An abstract prolog instruction set. Technical Report Technical Note 309, SRI International Artificial Intelligence Center, October 1983.

List of Figures

List of Tables