A flexible Prolog interpreter in Python

Carl Friedrich Bolz

Institut für Informatik
Heinrich-Heine-Universität Düsseldorf

Pyrolog is a Prolog interpreter written in Python
want to compile it to get interesting performance
translation tool-chain part of the PyPy project
Python itself is too dynamic to be translatable to other languages, need a subset
RPython ("Restricted Python") is a subset of Python translatable to other languages
RPython is designed to be significantly faster than regular Python
What is PyPy?

- started as a Python VM implementation in Python
- most important part: translation tool-chain for RPython
- is becoming a general environment for writing interpreters (JavaScript, Prolog started)
- Open Source project (MIT license)
- received EU funding for 2.5 years
VMs are still hard

Hard to reconcile:
- flexibility
- maintainability
- performance (needs dynamic compilation techniques)

Especially with limited resources (like Open Source projects, research projects)
CPython (the reference implementation) is a straightforward, portable VM.

- Pervasive decisions: reference counting, global lock . . .
- No dynamic compilation
The Python case

CPython (the reference implementation) is a straightforward, portable VM.

- Pervasive decisions: reference counting, global lock . . .
- No dynamic compilation

Extensions:

- **Stackless** (unlimited recursion, coroutines, green threads)
- **Psyco** (run-time specializing compiler)
- **Jython, IronPython**
The Prolog case (i)

- problem mitigated by the fact that Prolog the language does not change
- the core of Prolog is very simple (at least compared to Python)
- a lot of implementations out there
- well-tuned mature C implementations (Sicstus, XSB, SWI, GNU-Prolog)
- on CLR (P#) and JVM (Prolog Café, tuProlog)
The Prolog case (ii)

mature C implementations

- interfacing with libraries is tedious
- changing the language to experiment is hard
- often extensions to core Prolog, incompatible between each other

implementing on CLR and JVM

interfacing with libraries of the platform mostly easy

no extensions to core Prolog (like tabling, coroutines, constraints)

slow, compared to good C implementations
### The Prolog case (ii)

**mature C implementations**
- interfacing with libraries is tedious
- changing the language to experiment is hard
- often extensions to core Prolog, incompatible between each other

**implementations on CLR and JVM**
- interfacing with libraries of the platform mostly easy
- no extensions to core Prolog (like tabling, coroutines, constraints)
- slow, compared to good C implementations
PyPy’s Approach

Goal: generate VMs from a single high-level description of the language, in a retargettable way.

- Write an interpreter for a dynamic language (Python, Prolog, JavaScript, whatever) in a high-level language (Python)
- Leave out low-level details
- Favour simplicity and flexibility
- Define a mapping to low-level targets
- Generate VMs from the interpreter
Mapping to low-level targets

Mechanically translate the interpreter to multiple lower-level targets

- C-like
- .NET
- Java
Mapping to low-level targets

- Mechanically translate the interpreter to multiple lower-level targets
  - C-like
  - .NET
  - Java
- Insert low-level aspects into the code as required by the target
  - object layout
  - memory management
Mapping to low-level targets

- Mechanically translate the interpreter to multiple lower-level targets
  - C-like
  - .NET
  - Java
- Insert low-level aspects into the code as required by the target
  - object layout
  - memory management
- Optionally insert new pervasive features not expressed in the source
  - continuations, “micro-threads”
  - dynamic compilation
Features not present in the source can be added during translation:

- **memory management**: use different GC strategies (Boehm collector, custom mark-n-sweep)
- **Stackless transformation**: allows program to control its stack (continuations, . . .)
A JIT compiler as a translation aspect

- Transform the interpreter into a JIT compiler, using partial evaluation and specialization techniques
- Some hints in the interpreter source needed
- Current prototype applied to Python interpreter gives impressive speedups
Prolog Interpreter Implementation

- naive, very simple interpreter
  - uses "structure copying"
  - interprets Prolog terms directly, no bytecode
- uses continuation passing style inspired by BinProlog
- implements large parts of the ISO standard (some builtins missing)
Builtins

- builtins implemented in Python
- easy to add new ones to interface with libraries
- application-specific builtins
- examples:
  - functions to download and analyze webpages
  - an imperative hashmap
Interpreter Facts

- 2500 lines of Python code in total
- 700 of those are for builtins
- after translation to C: 14000 lines of C code
- part of the PyPy distribution at:  
  http://codespeak.net/pypy
Performance (i)

A flexible Prolog interpreter in Python

Carl Friedrich Bolz
Performance (ii)

- performance is quite bad compared to tuned C implementations
- performance is pretty good compared to Java and .NET implementations
- surprising, since those are often based on the WAM
- maybe the WAM model does not match these VMs very well?
Summary

- Very simple Prolog interpreter in RPython can compete with interpreters on the JVM, CLR
- Interpreter implementation eased by use of a high-level language
- Low-level details abstracted away but re-inserted later
Outlook

- complete the set of builtins
- tight language integration between Prolog and Python
- apply the dynamic compiler generator to the Prolog interpreter
http://codespeak.net/pypy
Translation Steps

- Generate flow graphs from the RPython program
- Perform global type inference on the flow graphs
- Transform flow graphs through several steps until they match the level of the target environment
- Weave in translation aspects in the process

Carl Friedrich Bolz: A flexible Prolog interpreter in Python