A flexible Prolog interpreter in Python

Carl Friedrich Bolz

Institut für Informatik
Heinrich-Heine-Universität Düsseldorf

24. Workshop der GI-Fachgruppe Programmiersprachen und Rechenkonzepte, 4. Mai 2007

Outline

1. What is Pyrolog?
2. The PyPy Approach to VM Construction
   - Overview
   - Motivation
   - Approach
3. The Prolog interpreter

Pyrolog

- Pyrolog is a Prolog interpreter written in RPython
- RPython is a subset of Python translatable to other languages
- translation part done with the help of the PyPy project

What is PyPy?

- started as a Python VM implementation in RPython (a well-chosen subset of Python)
- includes a translation tool-chain
- 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

It is hard to achieve:

- flexibility
- maintainability
- performance (needs dynamic compilation techniques)

Especially with limited resources (like Open Source projects, research projects)

The Python case (i)

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

- Pervasive decisions: reference counting, single global lock ...
- No dynamic compilation
- Extensions:
  - Stackless (unlimited recursion, coroutines, serializable continuations)
  - Psyco (run-time specializer)
  - Jython, IronPython

The Python case (ii)

- Extensions have problems
  - need to keep track of CPython
  - are hard to maintain
  - Psyco very hard to port to other hardware architectures
- The community wants Python to run everywhere: Jython (Java), IronPython (.NET). Lots of effort and duplication.
- At various points various incompatibilities between the implementations

The Prolog case

- problem mitigated by the fact that Prolog the language does not change
- a lot of implementations out there
- well-tuned mature C implementations (Sictsus, XSB, SWI, GNU-Prolog)
  - have sometimes incompatible extensions to core Prolog
  - interfacing with libraries is tedious
  - changing the language to experiment is hard
  - fixed implementation decisions (GC, how to generate code, etc.)
- on CLR (P#) and JVM (Prolog Café, tuProlog)
  - interfacing with libraries of the platform mostly easy
  - no extensions to core Prolog (like tabling, coroutines)
  - slow, compared to good C implementations

PyPy's Approach

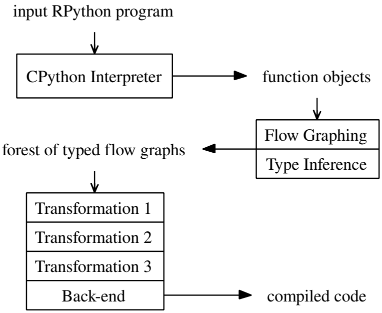
Goal: generate VMs from a single high-level description of the language, in a retargettable way.

- 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
  - Java
  - .NET
- Insert low-level aspects into the code as required by the target (Object layout, memory management)
  - object layout
  - memory management
- Optionally insert new pervasive features not expressed in the source
  - continuations
  - dynamic compilation

Translation Steps



- Generate flow graphs from the RPython program
- Peform 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

Translation Aspects (i)

Features not present in the source can be added during translation.
Example: memory management:

- Boehm garbage collector
- mark-n-sweep written in RPython, with additional features
- reference counting

Translation Aspects (ii)

- Stackless transformation: continuation capture, implemented by saving the low-level frames' local variables into the heap and back
  - allows arbitrarily deep stack usage
  - uses the C stack as long as possible
  - has the consequence of making RPython do tail call elimination
- work in progress: turning an interpreter into a just-in-time compiler is a translation aspect too

Prolog Interpreter Implementation

- naive, very simple interpreter
  - uses "structure copying"
  - interprets Prolog terms directly, no bytecode
- uses continuation passing style similar to BinProlog
- Prolog calls mapped to RPython calls
  - possible because stackless allows arbitrary deep recursion
- implements large parts of the ISO standard (some builtins missing)

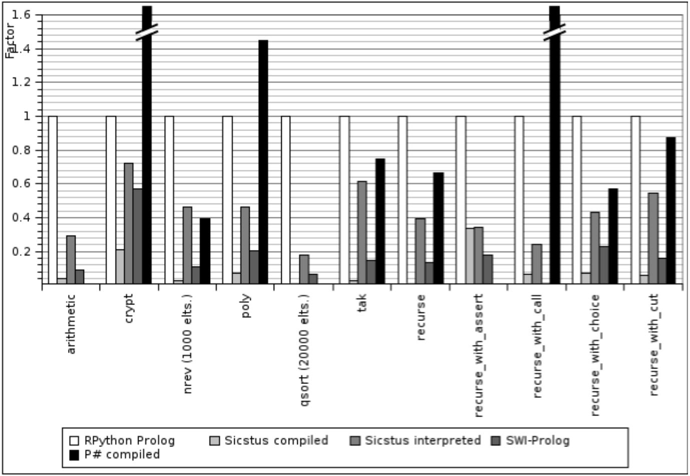
Builtins

- 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 line of C code
- part of the PyPy distribution at: http://codespeak.net/pypy

Performance (i)



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 it's hard to simulate the WAM on such a VM

Title

-

Summary

- The construction of virtual machines gets easier when using high-level languages
- XXX
- XXX

Outlook

-