Building a Multi-Language Interpreter Engine

Dan Sugalski
PythonCon 10
February 6, 2002

Or....


All your
Interpreter are
Belong to Us!
PythonCon 10

Our languages of interest

- Python
- Perl
- Ruby
- Scheme
- Tcl
- Objective C (A little)

General Interpreter Properties

What interpreters, especially for dynamic languages, provide

Resource Management

- Proper detection and destruction of dead objects
- Memory collection and management
- OS resource management (threads, files, signals, and suchlike things)

OS Independence

- The whole world isn't uniform
- Provides an abstract interface to the OS
- Allow transparent emulation of features not easily available
- Frees the programmer from having to worry about platform-specific details
- Though they can still be worried about

Rich type systems

- Interpreter's job to make complex data behave like simple data
- Easy extendibilty here requires a lot of work under the hood
- Makes non-traditional types easier for the programmer to use

Dynamic behaviour changes

- Dynamic recompilation
- Dynamic type behavior changes
- Makes classic optimizations somewhat difficult

High-level programming concept support

- Closures
- Continuations
- Curried functions
- Runtime class and method autogeneration
- Matrix operations

Safe Execution

- Resource quotas
- External access restrictions
- Paranoid runtime control flow checking
- Static checking is possible, but very restrictive

Accomodating Specificity

Everyone does the same things differently, more or less

Object Models

- Mildly different
- Object hierarchies differ
- Single/Multiple inheritance
- Per-object variables and methods

Standard Libraries

- Every language has its own
- No two are exactly alike
- Only really an issue with functions provided by C routines

Syntax

- Significant differences between languages
- Generally just a parser issue
- Most significant issue for the programmer
- Least significant (almost) issue for the interpreter

Extensions

- Extension interfaces run from horrid (perl) to very nice (Ruby)
- Usually tied tightly to the implementation of the interpreter
- Generally not considered part of the language

Semantics

- The easiest of the issues
- Semantic differences between most languages of a class are trivial
- Ultimately a matter of speed more than anything else

The Parrot Bits

How Parrot does all this stuff

Parrot's design goals

- Run perl code fast
- Portable
- Clean up all the grotty bits
- A good base for perl's language features
- Longvevity of the core design
- Multi-language capable

We assume modern hardware

- Good-sized L1 and L2 caches
- Main memory access expensive
- Unpredictable branches expensive
- A reasonable number of CPU registers
- Lots of RAM handy

Parrot's a register machine

- Reduces memory load/store
- Reduces by-name lookups of variables
- Translates well to modern hardware
- Avoids a lot of the common stack twiddling timewasters
- Can be treated as a large named temp cache for the register-phobic

Simple and complex types

- Native int, native float, string, and PMCs
- PMCs are the “everything else” class
- Supports arbitrary-precision numbers
- Interface abstract to make adding new types easy
- Simple types are basically builtin shortcuts for the optimizer

Split DOD & GC

- We check for dead objects and collect memory in separate phases
- Memory tends to get chewed up faster than objects die
- Most objects don't need to do anything when they die

Easy extendability and embeddability

- Stable binary API
- Clean interface for extenders
- Simple and small interface for embedders
- Internal details hidden
- Embedders have control over the interpreter's environment. (IO, ENV, command line args)

Portable

- Perl 5 runs (or has run) in 70+ platforms
- Support for many Unices, Win32, VMS, and Mac
- Every platform has something broken about it
- Not shooting for a lowest-common denominator

High-level I/O model

- Async I/O everywhere
- Bulk read support
- Byte, line, and record access supported where appropriate
- All I/O can be run through filters
- Finally dump C's stdio

Language-specific features are generally abstract

- We don't mandate variable types or behaviours
- Generic fallbacks are provided
- Lets us punt on parts of the design and put things off for later

Sort of OO under the hood

- OO (of sorts, it's still all C) where appropriate
- The whole world's not OO
- Neither are any CPUs to speak of
- OO support semi-abstract
- Used as an abstraction layer

Cross-language with Parrot

How to actually make it work

Variables

- Variable types know how to do things
- Variable code can be loaded on the fly
- Operator overloading is generally implemented via variable vtable functions

Opcode Libraries

- Opcode libraries may also be loaded dynamically
- Languages may define their own oplibs
- Allows maximum performance for language-specific code with interpreter flexibility

Pluggable parser

- Parser is general-purpose
- May be overridden lexically
- Has the full power of the Parrot engine to draw on
- Should be rather easier than Lex & Yacc to work with
- If we can manage perl, everything else is easy

Inter-language calling conventions

- We don't, and can't, guarantee 100% seamlessness
- Don't guarantee object hierarchies
- Provide a thunking layer for automatic type translations
- Make things work at least as well as calling unspecialized C extensions, usually better

Questions?