The heavyweight parts of lightweight languages

LL1 Workshop

November 17, 2001

The generic bits

What the heavy lifting is

Resource Management

- Proper destruction of dead objects
- Memory collection and management
- OS ‘object’ management (threads, files, signals, and suchlike things)

OS independence

- The whole world isn't uniform
- Provides an abstract interface to OS
- Allows transparent emulation of features not easily available
- Frees the programmer from having to worry about the grotty details

Rich type systems

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

Dynamic behaviour changes

- Dynamic recompilation
- Dynamic type behaviour 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 possible, but very restrictive

The Parrot bits

How we're doing the heavy lifting for Perl 6. (And Python, Ruby, and Scheme, though they don't know it yet…)

Parrot's design goals

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

We assume modern hardware

- Good-sized L1 & 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/stores
- Reduces by-name lookups of variables
- Translates well to modern hardware
- Avoids a lot of the common stack twiddling time wasters
- Can be treated as a large named temp cache for the register-phobic

Simple and complex types native

- Native int, native float, strings, and PMCs
- PMCs are ‘everything else’
- Support for arbitrary-precision numbers
- Interface abstract to make adding new types easy
- Simple types 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 (I/O & %ENV)

Portable

- Perl 5 runs (or has run) on 70+ platforms
- Support for many Unices, Win32, VMS, and Mac
- Everyone's got something broken about them
- Not shooting for 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
- We don’t mandate method dispatch
- Generic fallbacks provided
- Lets us punt on the design and put it off for later

Sort of OO under the hood

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