Parrot: VM design gone crackers?

Jonathan Worthington
University Of Cambridge
What is Parrot?

• A virtual machine for dynamic languages.

• Started out as the Perl 6 internals project – unlike Perl 5, there was to be a clean language/runtime boundary.

• Aims to provide support for many languages and allow interoperability between them.

• Named after an April Fool’s joke which referenced a Monty Python sketch. :-)
**Dynamic Languages**

- Think Perl[56], Python, Ruby, Tcl...
- Often need their parsers available at runtime
- Classes, methods, functions etc being created at runtime is not unusual
- Much is done symbolically
- Often have language features like continuations, closures, co-routines etc.
Why a new VM?

• The JVM and the .NET CLR can handle dynamic languages, but you re-invent quite a few wheels when writing the compiler.

• Perl 6 should support the range of platforms Perl 5 does – which is a lot. Need something that ports well.

• A chance to innovate; Parrot never was to be just another JVM clone.
Parrot Architecture

• A register machine
• Contexts capturing the notion of closures, subroutines/methods and continuations
• Uses continuation passing style
• PMCs: types with a common v-table for interoperability
• Extensible at the instruction and type level
• Many HLL features supported…
Why virtual register machines?

- VMs have tended to be stack based.
  - Easy to compile to
  - Leads to compact instruction code
  - With a JIT (Just In Time) compiler, you can get very good performance
- However, register architectures have some advantages.
Why virtual register machines?

• Stack machines have heavy instruction dispatch overhead when interpreting, especially with regard to tweaking the stack pointer.

• Parrot needs to run well on lots of arcane platforms - can’t rely on having JIT.

• The cheaper instruction dispatch of register machines is a big advantage.

• Note .NET is slow to interpret – by design.
Why virtual register machines?

• Another advantage comes when JITing time – you already have register code, possibly that needs no further register allocation; even if it does, still don’t need to do stack to register mapping.

• Also, 3-address code more suited to optimization than stack code – don’t rely on JIT-time optimizations.

• But what about spilling?
Variable size register frames

• Originally had a fixed number of registers.

• Intermediate language compiler provides “virtual registers”.

  • Does register allocation

  • Spill to an array

• The register file is just a chunk of memory, so spilling just leads to wasteful memory copying => variable sized register frames.
Register Frames

- 4 types of registers: Integer, Number, String, PMC.
- Each sub annotated with the number of each that it needs.
- 2 pointers into the register frame allow access to all registers.

<table>
<thead>
<tr>
<th>I Registers</th>
<th>N Registers</th>
<th>S Registers</th>
<th>P Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bp</td>
<td></td>
<td></td>
<td>bp_sp</td>
</tr>
</tbody>
</table>
**Contexts**

- A register frame belongs to a context.
- A context is somewhat analogous to a stack frame – there’s one per invocation of a sub and a pointer to the caller’s context.
- You also have a context per closure, along with a pointer to its enclosing context.
  - Lexicals are in registers – more later.
- Continuations just a chain of contexts.
Continuation Passing Scheme

• Conceptually, before a call we take a continuation.

Parrot – VM design gone crackers?
Parrot uses Continuation Passing Scheme

- Then pass the continuation along with the arguments to the sub being called.
Parrot uses Continuation Passing Scheme

- Invoking a continuation involves replacing the current call chain with what was captured.
Parrot uses Continuation Passing Scheme

- Conveniently, this turns out to do just what a return would do (noting that a continuation captures the program counter too).
Why Continuation Passing Scheme?

• Parrot has a lot of context information to save; continuations capture all of it neatly.

• No concerns about over-flowing the stack or over-writing return addresses, so good from a security stand-point.

• Tail calls become cheap to implement – just pass on the already taken continuation.

• Doesn’t this make calling really expensive?
Return Continuation Optimization

• Don’t really copy all of the contexts.

• Give each context a “valid for re-use” flag.

• If a real continuation is taken, then walk down the contexts chain, marking each one as invalid.

• Also have a reference count on a context for how many continuation are using it, so only need to walk down as far as when the last continuation was taken.
What is a PMC?

• A PMC defines a type with a certain set of behaviours and internal representation.

• Implements some of a pre-defined set of methods that represent behaviours a type may need to customize, such as integer assignment, addition, getting the number of elements, etc.

• Method bodies written in C, but much code is generated by a build tool.
How do PMCs work?

• Each PMC has a pointer to a v-table.

• When operations are performed on PMCs, the v-table is used to call the appropriate PMC method.

• A PMC may inherit from many other PMC types.

• PMCs are eligible for garbage collection – may tell the garbage collector what other PMCs it references too.
**How do PMCs work?**

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>inc P3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How do PMCs work?

```plaintext
inc P3

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Ref</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**PMC**

<table>
<thead>
<tr>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>v-table</td>
<td>0x00C03218</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
How do PMCs work?

```
inc P3
```

---

### PMC

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>v-table</td>
<td>0x00C03218</td>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### V-table

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>inc</td>
<td>0x00A42910</td>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**How do PMCs work?**

<table>
<thead>
<tr>
<th>PMC</th>
<th>V-table</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>0x00C03218</td>
</tr>
</tbody>
</table>

V-table:

<table>
<thead>
<tr>
<th>inc</th>
<th>0x00A42910</th>
</tr>
</thead>
</table>

**Inc P3**

**Increment v-table function**
The same operation in two languages may produce very different behaviour.

Consider the increment operator (++) performed on the string “ABC”.

- In Perl, the string becomes “ABD”.
- In Python, an exception is thrown.

PerlString and PythonString PMCs can implement the “increment” method differently.
PMCs support aggregate types

- PMCs have v-table methods for keyed get and set (where the key is an integer, string or PMC).

- These provide an interface for implementing arrays and dictionary data structures (such as hash tables).

- Storage mechanism left for the PMC to implement (e.g. a BitArray PMC could be implemented that uses 1 bit per element).
PMCs enable language interoperability

• A Perl array may exhibit one set of behaviours (for example, automatically resizing) to a .NET one (which has a fixed size).

• As access to elements is through a common v-table interface, the internal representation and specific behaviours don’t matter – Perl code can access elements from a .NET array and vice versa.
And there’s more...

- PMCs provide the basis for the Parrot class and object system, with v-table methods such as `add_parent`, `add_method`, `find_method`, `isa`, `can` and more.

- Often used to provide an interface to Parrot internals and features; continuations and exceptions are represented as PMCs.

- PMCs simultaneously solve many problems through a single simple mechanism.
Type extensibility

• As well as being built into the Parrot core, PMCs may be built into dynamically loaded libraries and loaded at runtime.

• Build tools make this no harder than writing PMCs for internal use.

• Currently, most of the Parrot internals are exposed – potential to crash the VM.

• Parrot needs a way to determine whether extensions being loaded are “trusted”.
Instruction extensibility

• Can also write extra VM instructions in a dynamically loaded library.

• Again, good build tools make this easy.

• Assembler needs to load the library, so it recognizes the mnemonics and can check the types.

• Can provide specialized, language-specific instructions without bloating the core VM.
Instruction extensibility good?

• Dynamically loaded instructions can’t match the performance of core ones – for example, there is no way to JIT them.

• However, much cheaper dispatch overhead than calling a method on a PMC.

• They share the same trust issues that dynamically loaded PMCs do.

• I’m using them quite heavily with my .NET to Parrot bytecode translator.
Parrot Programs

Parrot programs are mostly represented in one of three forms (an AST format exists, too).

Higher Level

PIR = Parrot Intermediate Representation

PASM = Parrot Assembly

PBC = Parrot Bytecode

Lower Level
Parrot Intermediate Representation

Parameter access syntax

Register code looks like HLL

Virtual registers

Return conventions hidden

Simple sub declaration

Named virtual registers

Calling conventions hidden away

```
.sub factorial
    .param int n
    .local int result

    if n > 1 goto recurse
    result = 1
    goto return

.recurse:
    $I0 = n - 1
    result = factorial($I0)
    result *= n

.return:
    .return (result)
.end
```
What does PASM look like?

Looks like assembly

Opcode to get parameters

Calling conventions exposed

Opcodes for returning

factorial:
    get_params "(0)", I1
    lt 1, I1, recurse
    set I0, 1
    branch return

recurse:
    sub I2, I1, 1
@pcc_sub_call_0:
    set_args "(0)", I2
    set_p_pc P0, factorial
    get_results "(0)", I1
    invokecc P0
    mul I0, I1

return:
@pcc_sub_ret_1:
    set_returns "(0)", I0
    returncc
What does PBC look like?

• A portable binary file format.
  • Written with the endianness and word size of the machine that generated it – good for performance.
  • If running on a different type of machine translation done “on the fly” – good for portability.
• Can be executed (almost) directly by the Parrot virtual machine.
Why PIR, PASM and PBC?

• Need something that is efficient to load and directly execute – PBC
• Need something small to distribute – PBC
• Need something that is human readable and writable. – PIR or PASM
• Need a way to abstract away details (like calling conventions) from compilers – PIR
• Need low level assembly language – PASM
Looking at some HLL features

• Parrot provides support for a lot of HLL features to ensure interoperability.

• For example, it’s nice if a Perl closure can be passed to some Common LISP code.

• If closures weren’t implemented at a VM level, different compilers could do them differently.

• The final part of the talk looks at a few of the more interesting HLL features in Parrot.
Lexical variables

• The needs of various languages with regards to lexical (statically scoped) variables differ somewhat.

• Many languages need to be able to look them up by name.

• Some but not all languages know what lexical variables they’ll have at compile time.

• Then there’s nesting and closures to think about...
Lexical variables

• When lexicals are known at compile time, they can simply be stored in a register.

• PIR syntax to associate name/register.

  .lex "$x", P2

• LexInfo PMC stores these mappings in a hash table.

• Good performance – no need for “by name” lookup in common case and the mappings are frozen at compile time.
Lexical variables

- Some languages don’t know about lexical variables until runtime (e.g. Tcl).
- Can’t associate a register with it; instead always lookup and store through ops.
- LexPad PMC stores the lexical variables in a hash table, with their names being the keys.

```plaintext
store_lex "x", P0
...
P0 = find_lex "x"
```
Lexical variables - nesting

• Nesting is specified though a \texttt{:outer(...)\ modifier on a sub.}

• Take the following example Perl 6 program, which gives the result 42:

```perl
my $a = foo();
say $a();
sub foo() {
    my $x = 42;
    sub bar() { return $x; }
    return &bar; # Returns bar, doesn’t call it
}
```
Lexical variables - nesting

• Compiles to something like the following:

```plaintext
.sub foo
    .lex "$x", P0
    P0 = new Integer
    P0 = 42
    P1 = find_global "bar"
    .return(P1)
.end

.sub bar :outer(foo)
    P0 = find_lex "$x"
    .return(P0)
.end

.sub _main :main
    .lex "$a", P0
    P0 = foo()
    P1 = P0()
    print P1
    print "\n"
.end
```
Lexical variables - closures

• A closure captures its lexical environment.
  • This is formed by walking down the “outer chain” (not the call chain) and adding each lexical pad to it.
  • Optimization possible if we encounter an existing lexical environment.

• Nested subs are really creating a closure anyway – they’re just closures that can take parameters.
Namespaces

• Another places where different languages want different things, but we still want to have interoperability.

• Policy as well as technical issues.
  
  • Where does a language put its guts?
  
  • Are language’s namespaces kept apart?

• What about languages with sigils ($a, @b) sharing with those that don’t (a, b)?
Namespaces - policy

• Top level namespaces will be:
  • HLL names, in lowercase, for user defined namespaces and data (perl6)
  • HLL names, in lowercase and prefixed with an underscore, for language internals (_perl6)
• This does mean that to use classes from another language, you’d need to know what language they were written in, unlike .NET.
Namespaces - implementation

• Namespaces are hierarchical – that is, “Monkey::Abu” has the Abu namespace as a member of the Monkey namespace.

• A namespace itself is implemented as a PMC, and thus languages can provide their own specialised implementation.

• The (raw) interface is just a dictionary mapping names to PMCs (recall classes, subs, etc are PMCs).
Namespaces – implementation

- The raw interface is fine for use when the namespace “belongs” to the current HLL.
  - HLL code knows about name mangling and sigils.
- Additionally have a typed interface.
  - Hides away the quirks of a particular HLL’s naming scheme.
- Allows for HLL interoperability.
Namespaces – typed interface

• The typed interface is provided as a number of methods on a namespace PMC.
  • Add, delete and find operations.
  • Differentiates between namespaces, subs and variables.
• Only about naming – no type checking.
• Doesn’t handle the scalar/array/hash sigil distinction, but can figure that out at runtime.
Namespaces – export/import

• Exporter push rather than importer pull.

• “export_to” method on namespace PMC is provided with a list of symbols to export and the namespace to export them to.

  • Empty/null list => default exports

• Will usually use the typed interface on the destination namespace, but shortcuts and other evil are possible – just check the type of the namespace you’re exporting to.
Calling conventions

• Need a fairly rich calling convention to handle the needs of a wide variety of languages.

• This means Parrot needs to provide...
  • Both positional and named parameters
  • Optional parameters
  • “Slurpy” parameters
  • Multi-method dispatch
• PIR syntax, as shown earlier, hides away the details.

• Actually have 4 instructions: set_args, get_params, set_returns and get_results.

• Take a signature PMC and a variable number of operands specifying the register number for each argument/return value.

• Register type comes from the signature.
• In general, we can’t perform as well as VMs such as .NET and the JVM, since the calling conventions are more complex.

• However, simple cases can be JITted.

• Furthermore, simple cases of tail calling can be optimized to iteration at JIT time.

• We can run the Languages Shootout Ackerman’s function benchmark faster than optimized C (by GCC) on x86 and PPC!
Future directions

• Parrot is still missing some Important Stuff.
  • Concurrency control – the plan is to implement STM, which will be needed to support Perl 6’s more declarative style of concurrency (atomic and async blocks).
  • Completion of IO, AIO and events
  • Security subsystem (VMS inspired)
  • Parrot Grammar Engine for parsing
Conclusions

• Parrot has some notable differences to the JVM and the .NET CLR.

• Trying to support many existing and quite different languages isn’t trivial.

• There’s much left to do, but also much already done – Parrot is running notable subsets of a range of languages.

• Any questions?