Parrot 2007

A Tour of Parrot's Design, Tools, and Code

chromatic@wgz.org

09 April 2007/Portland Perl Mongers

Outline

1. Parrot Goals
2. Parrot's Design
3. Programming PIR
4. Parrot in its Tree
5. The Partridge

Outline

1. Parrot Goals
2. Parrot's Design
3. Programming PIR
4. Parrot in its Tree
5. The Partridge

A Better Virtual Machine

Perl 6 needs a powerful virtual machine.

So Do Other Languages

Why not share?

Better Tools for Building DSLs

A good, dynamic VM with good tools may *encourage* more language development.

Outline

1. Parrot Goals
2. Parrot's Design
3. Programming PIR
4. Parrot in its Tree
5. The Partridge

A Register-Based Machine

Most (many?) VMs use stacks. Parrot uses sets of registers, like a CPU.

Register Typing

Four types of registers:

- Integer: `I0 .. I`n
- Float: `N0 .. N`n
- String: `S0 .. S`n
- PMC: `P0 .. P`n

Parrot Magic Cookies

A PMC is a Parrot primitive *and* an object.

PMC: The Perl 5 Problem

Everything is an SV, except:

- A C datastructure isn't the same as a Perl data structure
- A C datastructure is *rather* static
- Perl 5 allows overloading

PMC: Vtables

PMCs have a uniform set of operations implemented as vtable methods (er, function pointers).

PMC: Uniform Access

Common operations include:

- Accessing primitive values (integer, float, string, PMC)
- Accessing keyed values
- Instantiation
- Destruction
- The standard operations you can legitimately perform on all PMCs

PMC: Language Semi-Agnostic

By design, you can write PMCs in PIR or C.

This means you can *extend* built-in PMCs or define your own without writing or compiling C code.

Common PMC Vtable Methods

- `init`: initialize the PMC
- `name`: get the PMC's name
- `clone`: make a copy of the PMC
- `get_{integer,number,bignum,bool,string}`: retrieve a typed value from the PMC
- `set_{integer,number,bignum,bool,string}`: set a typed value within the PMC
- `can`: does the PMC implement a named method?
- `does`: does the PMC perform a role?
- `subclass`: extend the PMC
- `destroy`: finalize the PMC

Lots and Lots of Ops

Like most VMs, Parrot uses opcodes... but unlike most, it uses hundreds.

Some Fun Ops You Might Like

- `newclass $P0, $S0`: creates a new class named by a string and stores it in a PMC register
- `length $I0, $S0`: stores the length of a string in an integer register
- `noop`: does nothing
- `yield`: returns from a coroutine
- `get_global $P0, $S0`: retrieves the named global into a PMC register

Bytecode, Not an Optree

Parrot executes bytecode instead of walking an optree.

The Implications of Bytecode

- Clean separation of compilation and execution
- Cross-platform bytecode format
- Cross-language compatibility through bytecode
- Easier distribution of programs
- Some platforms don't need the compiler, just the execution engine
- Better space/time characteristics (`mmap()`)

Useful Behavior by Default

Design goal: simplify the common case, while making the uncommon possible.

Lots of Customization Possibilities

Don't like what Parrot does?

- Write your own object
- Extend a core PMC
- Write your own dynpmc
- Write your own dynamic opcode
- Talk us into applying a patch

Customization Through Pervasive OO

Oh, PMCs have methods too, besides vtable methods.

Want to change the interpreter properties? Call a method....

Pervasive Multi-Dispatch

Typed registers allow some parameterized dispatch.

Parrot supports multi-dispatch on PMC types too.

True Garbage Collection

Say goodbye to buggy and time-consuming reference counting!

... mostly.

True Garbage Collection

Say goodbye to buggy and time-consuming reference counting!

... mostly.

Continuation-Passing Style

Instead of call stacks, control flow uses continuations. They're like closures for control flow.

Reflection and Metaprogramming

Parrot's OO system now has a metamodel:

- Create new classes
- Add class attributes
- Add and remove methods
- Add and remove roles
- Add and remove parents
- Work with metaclasses

Platform Abstraction for Portability

Parrot attempts to provide a useful minimal and functional abstraction of the underlying platform.

This is particularly entertaining when dealing with features such as IO or concurrency and on bizarre platforms such as Windows.

FFI and NCI

Using shared libraries with C calling semantics requires only a small translation layer to convert to and from Parrot calling conventions.

This layer can even handle structs and callbacks.

Easily Embeddable

At least, this is the goal... the Parrot executable uses the embedding interface.

Embedding Parrot should require minimal knowledge of Parrot internals.

If I can embed it in Perl 5 (and I did), it should be embeddable elsewhere – including in itself.

Outline

1. Parrot Goals
2. Parrot's Design
3. Programming PIR
4. Parrot in its Tree
5. The Partridge

Basic Syntax

Almost everything is an opcode:

```
print "Hello, world!\n"

$S1 = "Hello, world!\n"
print $S1

$I0 = 2
$I1 = 2
$I3 = $I0 + $I1
print "2 + 2 = "
print $I3
```

Watch Your Compilation Units

Opcodes don't float free.

```
.sub ’main’ :main
  print "Hello, world\n"
.end
```

Declare Your Variables

Symbolic registers help, but named variables are better:

```
.local int    counter
.local string greeting

 counter  = 0
 greeting = "Boy, howdy"
```

Make a PMC

Note: this syntax may change

```
.local pmc user_locations
user_locations = new .Hash
```

This only works for built-in PMCs.

Control Flow is... Jumpy

```
  .local string sub_name
  sub_name = argv[1]

  if sub_name goto load_sub
  sub_name = ’procedural’

load_sub:
    # do more
```

Also `if COND then LABEL`.

Subroutines Are Easy

```
.sub 'my_sub_name'
    # do stuff
    .return( some_value )
.end
```

Subroutine Parameters

Arguments come in register types, so declare them:

```
.sub ’my_sub’
    .param int    counter
    .param float  certainty
    .param string label
    .param pmc    container

    # ...
.end
```

Optional Parameters

```
.sub 'my_sub'
    .param int counter      :optional
    .param int have_counter :opt_flag

    if have_counter goto initialized
    counter = 0

  initialized:
    # ...
.end
```

Slurpy Parameters

```
.sub 'cons'
    .param pmc head
    .param pmc tail :slurpy

    unshift tail, head
    .return( tail )
.end
```

The `:flat` attribute on a variable in a call is the opposite.

Make Things Happen on Schedule

- `:init`, run this code when loading the file, if this file is the initial file
- `:onload`, run this code if this file gets loaded
- `:main`, where to start execution if this file is the initial file

"file" may be a bit of a lie; this part of Parrot is still weird to me.

Use Other Code

- `.include` *`filename`* performs a textual include
- `load_bytecode` *`filename`* loads PASM, PIR, or PBC

An OO Assembly Language

Apart from its line-oriented nature and the lack of control flow, Parrot supports a lot of modern language features–including good OO support.

Create a Class

```
.local pmc parent_class
parent_class = newclass 'Parent'

.local pmc child_class
child_class = subclass my_class, 'Child'
```

Create an Object

```
.local pmc parent_instance
parent_instance = new parent_class

.local int child_type
child_type = find_type, 'Child'

.local pmc child_instance
child_instance = new child_type
```

Add Attributes

```
addattribute parent_class, ’name’
addattribute child_class,  ’age’
```

Get and Set Attributes

```
.local pmc name
name = new .String
set name, ’Floyd’  # name = ’Floyd’

.local pmc age
age = new .Integer
set age, 60        # age  = 60

setattribute child_instance, ’name’, name
setattribute child_instance, ’age’,  age
```

Creating accessors is nice. Do I sense a metamodel?

Namespaces

```
.namespace [ 'Parent' ]

# methods here

.namespace [ 'Child' ]

# more methods here
```

Declare and Call Methods

Within the proper namespace:

```
.sub ’my_method’ :method
   .param pmc some_value

   self.’some_other_method’( some_value )
.end
```

Multi-Dispatch

```
.sub ’double_it’ :multi( int )
    .param int value
    value *= * 2
    .return( value )
.end

.sub ’double_it’ :multi( string )
    .param string value
    value .= value
    .return( value )
.end
```

Try `double_it( 10 )` or `double_it( ’meow’ )`

Importing and Exporting

```
.local pmc ns_from, ns_to
ns_from = get_namespace [ ’Other’; ’NS’ ]
ns_to   = get_namespace

.local pmc exports
exports = new .ResizableStringArray
exports = split ’ ’, ’cons head tail’
ns_from.’export_to’( ns_to, exports )
```

Metaprogramming

Classes support several methods:

- `add_method` adds a named method to the class
- `add_attribute` adds an attribute to the class
- `add_parent` adds a parent to the class
- `add_role` adds a role to the class

More Metaprogramming

You can remove attributes, methods, parents, and roles too.

Still In Progress

Check back in a week for our progress implementing this new code....

Extending Built-Ins

```
.local pmc integer_class, positive_integer
integer_class    = getclass ’Integer’
positive_integer = subclass integer_class
```

Overriding Vtable Methods

```
.sub 'set_integer_native' :vtable :method
    .param int value
    value = abs value

    self.'super'( value )
.end
```

Note: `super` is in progress and `:method` may be unnecessary soon.

Other Features

- Lexical variables
- Iterators
- Exceptions
- Closures
- Continuations
- Coroutines

Outline

1. Parrot Goals
2. Parrot's Design
3. Programming PIR
4. Parrot in its Tree
5. The Partridge

Parrot's Top-Level Directory

```
apps/
blib/
compilers/
config/
debian/
docs/
editor/
examples/
ext/
include/
languages/
lib/
runtime/
src/
t/
tmp/
tools/
```

The Main Parrot Code

```
src/
src/charset/
src/dynoplibs/
src/dynpmc/
src/encodings/
src/gc/
src/io/
src/jit/
src/ops/
src/packfile/
src/pmc/
src/stm/
include/parrot/*.h
```

IMCC

`compilers/imcc`

There's some amount of legacy here.

PMCs Live in `src/pmc/*.pmc`

A mix of C and Perl, processed by `Parrot::Pmc2c::*`:

```
  PCCMETHOD void name(STRING *name :optional,
                    int got_name :opt_flag) {
    Parrot_Role *role = PARROT_ROLE(SELF);
    STRING *ret_name  = NULL;
    if (got_name) {
        /* Set role name. */
        role->name = name;
    }
    ret_name = role->name;
    PCCRETURN(STRING *ret_name);
}
```

Opcodes are in `src/ops/*.ops`

Opcodes get processed by `tools/build/ops2c.pl`:

```
inline op newclass(out PMC, in STR) :object_classes
{
    PMC *name = pmc_new(interp, enum_class_String);
    VTABLE_set_string_native(interp, name, $2);
    $1 = pmc_new_init(interp,
             enum_class_ParrotClass, name);
    goto NEXT();
}
```

Yet Another Opcode

```
inline op newclass(out PMC, in PMC) :object_classes
{
    $1 = pmc_new_init(interp,
              enum_class_ParrotClass, $2);

    goto NEXT();
}
```

Design Documents

Parrot Design Documents are in `docs/pdds/*.pod`
Drafts are in `docs/pdds/draft/*.pod`

```
=item True Root Namespace

The true root namespace is hidden from common
usage, but it is available via the
C<get_root_namespace> opcode. For example:

$P0 = get_root_namespace
```

Useful Parrot Libraries

`runtime/parrot/libraries` contains PIR libraries to use in your programs:

- PGE
- Data::Dumper
- Parrot::HLLCompiler
- SDL bindings
- Test::More
- YAML

Various Language Implementations

`languages/*` contains various language implementations in various stages of completeness:

- Tcl
- Punie (Perl 1)
- Plumhead (PHP)
- abc (example)
- Perl 6
- Cardinal (Ruby)
- TAP (Test Anything Protocol)
- Pheme (Scheme)

Compilers and Compiler Tools

There are also compiler tools, such as Partridge:

- PGE
- TGE
- PAST
- IMCC
- PIRC
- POST

The Test Directories

Of course there are lots of tests, too:

```
t/codingstd
t/examples
t/library
t/oo
t/op
t/pmc
t/src
...
```

Outline

1 Parrot Goals

2 Parrot's Design

3 Programming PIR

4 Parrot in its Tree

5 The Partridge

Attribute Grammars

Lexx and Yacc are the old and broken way of writing new languages.

A series of transformations between trees works much better.

Start with a Grammar

```
grammar Pheme::Grammar;

rule prog { <list>+ }
rule list { ( <list_item>* ) }

# quoted_string has to come first
rule list_item { <quoted_string>
                 | <atom>
                 | <list>
                 | <empty_list> }
```

Get Back a PGE Tree

```
(write "Hello, world!\n")
<list>: (write "Hello, world!\n")
  <list_item>
    <atom>: <write @ 1>
    <quoted_string>: <"Hello, world!\n" @ 7>
      <PGE::Text::bracketed>: <"Hello, world!\n" @
```

Transform PGE to PAST, part one

Handle the appropriate element and get match data.

```
transform result (list) :language(’PIR’) {
    .local pmc result
    result = new ’PAST::Exp’

    .local pmc match
    match = node[’list_item’]
```

Transform PGE to PAST, part two

Handle any child nodes.

```
.local pmc iter
iter     = new Iterator, match

.local pmc children
children = result.’children’()

.local pmc child
child = shift iter
child = tree.get( ’result’, child, ’list_item’ )
```

Transform PGE to PAST, part three

Run any transformations on child elements.

```
.local pmc op
op    = tree.get( ’maybe_op’, child )
result.’add_child’( op )
```

Transform PGE to PAST, part four

Handle other child nodes.

```
iter_loop:
  unless iter, iter_end
  shift child, iter
  child = tree.get(’result’, child, ’list_item’)
  result.’add_child’( child )
  goto iter_loop
```

Transform PGE to PAST, part five

Handle any special elements and return the results.

```
iter_end:
  .local string child_type
  child_type = typeof op
  unless child_type == ’PAST::Op’
      goto return_result

  result = tree.get( ’handle_specials’, result )

return_result:
  .return( result )
```

Transform PAST to POST

```
<Node>
  <PAST::Sub>
    <PAST::Stmts>
      <PAST::Exp>
        <PAST::Op> => 'write',
        <PAST::Val>
           'value'   => 'Hello, world!\n',
           'valtype' => 'double_quoted'
```

Transform POST to PIR

```
<POST::Node>
  <POST::Sub>
    <POST::Ops>
      <POST::Val>
        ’value’   => ’write’,
        ’valtype’ => ’’,
      <POST::Val>
        ’value’   => ’Hello, world!\n’,
        ’valtype’ => ’double_quoted’
```

A Scheme to PIR Translator

```
.namespace [ ’Pheme’ ]

.sub __onload :anon :load
    load_bytecode ’lib/PhemeSymbols.pbc’
    main()
.end

.sub main
    $P2 = eval( ’write’, "Hello, world!\n" )
    .return( $P2 )
.end
```

... Or Transform PIR to PBC

Parrot has code to generate PBC files. It only needs an API to generate that directly from POST to skip the PIR -> IMCC -> PBC steps.

Stackable Optimization Layers

There don't have to be three tree transformation steps. There can be one, or a dozen.

TGE is Flexible on the Language

Right now you have to use PIR, but any language that Parrot can understand is fine. ... including Perl 6.

TGE is Flexible on the Language

Right now you have to use PIR, but any language that Parrot can understand is fine. ... including Perl 6.