Perl 6 and the Parrot Virtual Machine

Fabian Fagerholm

April 1, 2005

Abstract

Parrot is a virtual machine designed to be a target for languages that have traditionally been run using an interpreter. The virtual machine and its assembly language include many high-level features such as objects, thread synchronization support, and garbage collection. It is set apart from other virtual machines by its register-machine structure, while many other contemporary VMs use a stack-based model.

This paper provides an overview of the design of the Parrot virtual machine, the surrounding tools, the current development status and the relation to the Perl 6 language in particular, and dynamic languages in general.

1 Introduction

The Parrot virtual machine (VM) [2] came into existence as part of an ongoing effort to implement the sixth version of the Perl language (Perl6). Perl has grown from a scripting language whose purpose was to replace and augment the Unix tools sed and awk, into a full-blown programming language with support on many platforms.

However, Parrot is not Perl6—rather, it is a register-based VM designed to run dynamically typed languages efficiently. In dynamically typed languages, type casting or type conversion is carried out by the interpreter or compiler. The runtime environment will enforce certain conversion rules and may halt execution or cause an exception if an impossible type conversion is attempted.

Parrot is an ongoing development project as is far from complete. However, the current implementation, with cruical parts written in C, forms a working system that can already be used for research projects on many platforms.

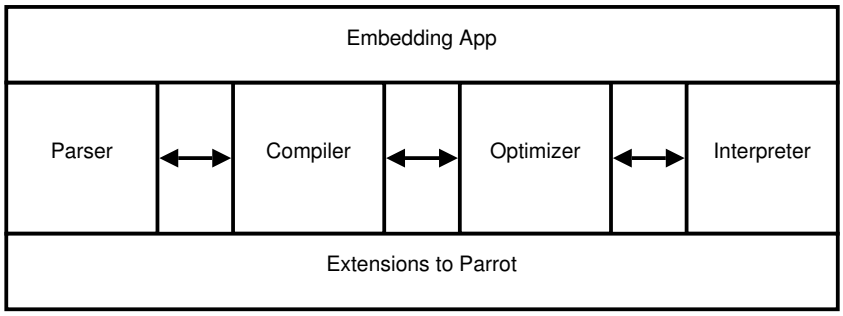

Figure 1: High-level overview of Parrot system. Adapted from A High-Level Overview of the Parrot System by Dan Sugalski [4].

2 A High-Level Overview of Parrot

In the current Parrot design, the system consists of a number of parts: a parser, a compiler, an optimizer and an interpreter. Each is typically arranged in a compiler pipeline, where a module gets a data structure to carry out its operations on from the previous module in the chain. The overall structure is shown in figure 1.

The Parrot parser is interchangeable per language. For example, one parser may parse Perl6 while others may parse Python or Ruby. The parser constructs an abstract syntax tree (AST) which is fed to the compiler. The compiler operates on the AST and produces a straightforward, non-optimized Parrot bytecode stream, which is then passed on to the optimizer. The optimizer operates on the bytecode, but may also aquire the original AST to produce a better optimization. Finally, the optimized bytecode is fed to the interpreter for running. The optimization phase is optional, and the interpreter may operate directly on the unoptimized bytecode produced by the compiler.

Traditionally, languages like Perl, Python and Ruby have been distributed as source code. In fact, there has been no distinction from a programmer or user perspective between a source and distributable (runnable) form, because the langauge has been interpreted from the source code. Languages like C and C++ are compiled directly to machine code, while languages like Java and C# are compiled to an intermediate form (bytecode), which is then executed in a bytecode interpreter. Parrot modifies these concepts to fit into the world of dynamic, source-interpreted languages.

Parrot can accept the source code of a program and perform parsing and compiling to bytecode at run-time. Thus, execution will begin by compiling the code to bytecode, and continue by executing the bytecode. However,

the Parrot interpreter can be exchanged with an interpreter that does not execute the bytecode, but instead saves it into a file. Thus, the precompiled bytecode can be run separately in the VM, bypassing the parsing, compilation and optimization stages.

It is also possible to extend the Parrot system to emit bytecode for another VM. The Parrot interpreter could perform an additional compilation cycle, and translate the Parrot bytecode into another bytecode, such as Java bytecode or the CIL bytecode of the .NET platform [4]. In fact, such functionality has been implemented for experimentation.

3 Low-level features of the Parrot VM

The Parrot VM or "Software CPU" has a number of low-level features that resemble the features of today's computer hardware. The VM is a register-machine, unlike many other VMs such as Java and .NET. The Parrot designers cite performance in dynamically typed languages as the main reason for choosing a register-based approach. Their reasoning says that compilation from bytecode to machine code is easier and produces more efficients results if the underlying model is the same [3]. Section 7.1 discusses the efficiency issues in greater detail.

The registers of the Parrot VM are stored in register frames. These frames can be pushed and popped onto a register stack, thus implementing visibility and scope. The register stack also reflects the subroutine call structure.

The registers are able to store different data types. Among the basic types are integers, floating-point values and strings. A more advanced type that facilitates many aspects of high-level languages are Parrot Magic Cookies, which are explained in the next section.

The current Parrot VM design specifies four groups of 32 registers. Each group is assigned a specific data type among the basic types.

4 High-level features of the Parrot VM

The Parrot VM contains support for low-level datatypes such as integers and floating-point values. However, it also handles two higher-level datatypes that low-level languages such as C usually leave to the programmer or an external utility library to handle.

4.1 Integers and floating-point values

Parrot divides integers into two subtypes: platform-native integers and arbitrary precision integers ("big integers"). The former depends on a compile-time choice and is dependent of the C compiler and platform used to compile Parrot. The latter has not yet been finally specified, but the general idea is to be able to represent values that are as large as the available memory or as long as as a C UINTVAL (unsigned long), whichever is shorter.

Floating point values follow the integer division into platform-native and arbitrary precision ("big number") floats. A compile-time choice determines the size of the former, and the latter has not yet been finally specified. The current drafts outline an implementation similar to the big integer specification, with the stated goal that conversion between the two should be as fast as possible.

4.2 Strings

Parrot strings hide all the tedious work required to maintain a dynamic string representation with multiple encoding schemes behind a standard interface. Internally, the Parrot string consists of a memory buffer, a pointer to the starting point of the string in the buffer, the amount of bytes used from the buffer, and information on the type and encoding of the string stored.

The string also includes a special property: the language of the string. This is used when sorting strings, since not all languages sort in the same way. This property is a good example of the fact that Parrot is designed to support dynamic languages in the VM, instead of putting such functionality in an external utility library.

4.3 Parrot Magic Cookies

A Parrot Magic Cookie (PMC) is a special datatype that is specifically designed to support dynamic languages. It is the mechanism by which Parrot can be made to support many different languages. A PMC is a container for other types—even the basic types such as integers may be encapsulated in a PMC. PMCs also allow for a certain kind of polymorphism in the objects. They contain a list of function pointers to functions that implement operations such as addition, subtraction or copying for the particular kind of object stored in the PMC [5].

The PMC contains some features that have not yet been finalized. For example, there is support for a metadata structure. There are many potential uses for this, one example is an image object that stores EXIF metadata

1	Control flow
2	Data manipulation
3	Transcendental operations
4	Register and stack ops
5	Names, pads, and globals
6	Exceptions
7	Object ops
8	Module handling
9	I/O operations
10	Threading ops
11	Interpreter ops
12	Garbage collection
13	Key operations
14	Properties
15	Symbolic support for High Level Languages
16	Foreign library access
17	Runtime compilation

Table 1: Opcodes of the Parrot Assembly Language. Adapted from Parrot Assembly Language by Dan Sugalski [6].

about date, time, exposure information, etc. in the metadata structure while the actual image data is stored in the data section of the object.

5 The Parrot Assembly Language

The Parrot authors describe the Parrot VM as a "virtual super CISC machine". They have opted to create an assembly language, Parrot Assembly Language (PASM) that maps directly to the VM bytecode. Given the complex capabilities of the VM, the assembly language is also quite powerful. The opcodes can be divided into 17 categories, shown in table 1.

The four register types of the Parrot VM are visible in the assembly language by a naming convention that says PMC registers are prefixed with a P, string registers with an S, integer registers with an I and number (floating-point) registers with an N. The assembly language has support for namespaces and subroutines.

As shown in table 1, the assembly language and thus the VM supports a wide range of high- and low-level features. We will look at a few of these in closer detail.

if tx, ix	Check register tx. If true, branch by X.
unless tx, ix	Check register tx. If false, branch by X.
jump tx	Jump to the address held in register x (Px, Sx, or Ix).
branch tx	Branch forward or backward by the amount in register x. (X may be either Ix, Nx, or Px) Branch offset may also be an integer constant.
jsr tx	Jump to the location specified by register X. Push the current location onto the call stack for later returning.
bsr ix	Branch to the location specified by X (either register or label). Push the current location onto the call stack for later returning.
ret	Pop the location off the top of the stack and go there.

Table 2: Opcodes for control flow. Adapted from Parrot Assembly Language by Dan Sugalski [6].

5.1 Control flow and data manipulation

The most basic opcodes deal with program control flow and data manipulation. This will provide a basic idea of how the assembly language is constructed.

There are seven opcodes for control flow and 17 for data manipulation. The opcodes takes a variable amount of parameters. For instance, the `if` opcode takes two parameters, while the `ret` opcode takes none. Table 2 shows all the opcodes for control flow.

The data manipulation opcodes and outlined in table 3. Notice the reference to the in-VM polymorphism by the use of the vtable structure in the PMCs.

5.2 Objects, exceptions and garbage collection

The assembly language contains opcodes for dealing with the PMC registers as objects. For example, the language can make the variable in a PMC to be an object of a specified type (`make_object`). It can also find a specific method and return it as a PMC in a specified register (`find_method`). The language contains opcodes for introspection; finding out if an object can perform a

new Px, iy	Create a new PMC of class y stored in PMC register x.
destroy Px	Destroy the PMC in register X, leaving it undef.
set tx, ty	Copies y into x. Obeys reference semantics, so may actually copy an object reference.
exchange tx, ty	Exchange the contents of registers X and Y, which must be of the same type. (Generally cheaper than using the stack as an intermediary when setting up registers for function calls.)
assign Px, ty	Takes the contents of Y and assigns them into the existing PMC in X. X's assign vtable method is invoked and it does whatever is appropriate.
clone Px, Py clone Sx, xy	Performs a "deeper" copy of y into x, using the vtable appropriate to the class of Py if cloning a PMC.
tostring Sx, ty, Iz	Take the value in register y and convert it to a string of type z, storing the result in string register x.
add tx, ty, tz *	Add registers y and z and store the result in register x. (x = y + z) The registers must all be the same type, PMC, integer, or number.
sub tx, ty, tz *	Subtract register z from register y and store the result in register x. (x = y - z) The registers must all be the same type, PMC, integer, or number.
mul tx, ty, tz *	Multiply register y by register z and store the results in register x. The registers must be the same type.
div tx, ty, tz *	Divide register y by register z, and store the result in register x.
inc tx, nn *	Increment register x by nn. nn is an integer constant. If nn is omitted, increment is 1.
dec tx, nn *	Decrement register x by nn. nn is an integer constant. If nn is omitted, decrement by 1.
length Ix, Sy	Put the length of string y into integer register x.
concat Sx, Sy	Add string y to the end of string x.
repeat Sx, Sy, iz	Copies string y z times into string x.

Table 3: Opcodes for control flow. Instructions tagged with * implement polymorphism and will call a vtable function if used on PMC registers. Adapted from Parrot Assembly Language by Dan Sugalski [6].

specified method (`can`), if it implements a specified interface (`does`) or if it is an instance of a specified class (`isa`).

Exceptions are supported in the VM. These low-level exceptions work by registering an exception handler that is active while in scope. Exceptions are thrown by the `throw` opcode, and the exceptions themselves are objects encapsulated by PMCs.

To go with objects and exceptions, the assembly language includes garbage collection functions. Parrot uses a Dead Object Detection Garbage Collection scheme (DOD/GC), which means the garbage collection proceeds in two steps:

1. Start from the root set and recursively mark all reachable objects as being alive (DOD).

2. Sweep through the memory space and deallocate unmarked objects—these are considered "dead" (GC).

The garbage collector can be paused and resumed, and sweeps for dead objects and actual garbage collection can be triggered at will.

5.3 Threading

The VM includes some features that facilitate threading. There is no concept of thread creation, starting or stopping, but the thread-supporting opcodes implement three locking operations that can be used to synchronize threads: `lock`, `unlock` and `pushunlock`. The first two can be used to lock and unlock access to a certain PMC, which the last is used to push an unlock request to the stack.

It is not clear from the current Parrot documentation whether the designers anticipate an in-VM cooperative threading model. It seems that the system embedding Parrot (see figure 1) is free to start multiple interpreters for the same code, and the interpreters can synchronize concurrent activity using the locking operations. The details of embedding are not yet available, though.

6 The Parrot Intermediate Code

In addition to the Parrot Assembly Language, the Parrot system includes an intermediate language called IMC. Compared to PASM, IMC is a medium-level language whose purpose is to be compiled either directly to Parrot

bytecode, or to PASM. It is the preferred language for compilers for the Virtual Machine.

The difficulty of writing a VM for a dynamic language is that dynamic languages need to support runtime evaluation and compilation of code. Languages like Java are static in this sense, since they can be precompiled to bytecode without problems [7]. Being a medium-level language, IMC can retain some of the dynamic features needed, while still allowing to "compile down" the high-level code into a format that allows more efficient runtime translation into Parrot bytecode.

7 Development status

Although Parrot is already usable for some things, it still appears to be far from complete. It can still be considered a project in its experimental phase, where the authors and designers are collecting experiences while working towards an initial release. There are some things that can already be said about Parrot, however. While other contemporary VMs are typically implemented using a stack-based model, Parrot has chosen a register-based model. This has led to a discussion regarding the efficiency of the two approaches. Meanwhile, Parrot suggests a direction that can be seen with other compilers: using one compiler to compile several languages, separately, but also potentially bridging the gap between different langauges. These aspects will be briefly discussed.

7.1 Efficiency of register-based VMs

The Parrot designers argue that a register-based model results in a VM, a bytecode and an assembly language that can be more efficient on today's register-based hardware, since the model is the same. However, the two most popular VM designs today, the Java VM and the .NET platform's CIL, use a stack-based model. There have been lengthy discussions in Internet forums regarding the merits of the different approaches.

Davis et al. [1] have compared the two approaches, and found, perhaps not surprisingly, that the efficiency is better in some areas and worse in some. In particular, they conclude that the amount of instructions executed in their register-based tests were smaller than in their stack-based tests. However, the number of bytecode loads increased. They divide the cost of executing a virtual machine instruction into three components:

1. instruction dispatch,

2. operand access, and

3. performing computation.

For the first component, they found that a register machine can often perform a given computation with a smaller amount of instructions than a stack-machine, even though the actual dispatch or a single instruction is not faster in either. The major drawback for the register machine was found in the second component, where a register machine must perform expensive operations to find the locations of the operands. The actual computation did not differ in efficiency.

To summarize, the main difference in efficiency between a stack-based and a register-based VM approach is in the overhead of instruction dispatch and operand access. The register-based approach can lead to fewer instructions and thus faster dispatch times, but operand access is slower. Finally, the tests performed by Davis et al. used the Java language, and it is not clear how the results translate to the context of dynamic languages.

7.2 Relation to Perl6 and other dynamic languages

While Parrot is a product of the Perl community, one of its specific goals is to allow compilation and running of many other languages. Parrot already contains functionality to compile Perl6, Python, Ruby, Tcl, Scheme, Forth and a number of other esoteric languages. Compilation of C# to the dotGNU implementation of .NET is possible, but available separately for licensing reasons. Some implementations are incomplete, but a quick look at what is available suggests that Parrot is indeed capable of accommodating the needs of many languages.

It may be interesting to note that the name Parrot was chosen as a pun on an April fool's joke that stated that the Perl and Python langauges were to merge. While the languages as such may not merge, there is now an emerging option to run them using the same VM.

8 Conclusions

Parrot is a register-based virtual machine for dynamic languages that have traditionally been run from source code by language-specific interpreters. Parrot consists of a compiler framework to compile different languages into

Parrot Assembly Language, which can be directly translated into a corresponding Parrot bytecode, or to a Parrot Intermediate Code that can be translated into either of those. The bytecode is then run on the Parrot VM. Translation can happen either as precompilation or at runtime.

Parrot is a work in progress and its primary reason for existence is the upcoming Perl6 language. It is not yet mature enough for wide use, but it can already be used to some extent, particularly for research and education.

References

[1] Davis, Brian, Beatty, Andrew, Casey, Kevin, Gregg, David, Waldron, John, The Case for Virtual Register Machines, in IVME '03: Proceedings of the 2003 workshop on Interpreters, Virtual Machines and Emulators, pages 41–49, ACM Press, New York 2003.

[2] The Perl Foundation, www.parrotcode.org website, April 1, 2004, URL http://www.parrotcode.org/

[3] The Perl Foundation, Parrot FAQ, March 31, 2005, URL http://www.parrotcode.org/faq/

[4] Sugalski, Dan, A High-Level Overview of the Parrot System, August 12, 2003, URL http://www.parrotcode.org/docs/pdd/pdd01_overview.html

[5] Sugalski, Dan, Parrot's internal datatypes, February 20, 2004, URL http://www.parrotcode.org/docs/pdd/pdd04_datatypes.html

[6] Sugalski, Dan, Parrot Assembly Language, December 2, 2002, URL http://www.parrotcode.org/docs/pdd/pdd06_pasm.html

[7] Smith, Melvin, IMCC and Parrot Programming for Compiler Developers — FAQ, December 3, 2001, URL http://www.parrotcode.org/docs/imcc/imcfaq.html