Jonathan Worthington

Virtual Machine Bytecode Translation: From The .Net CLI To Parrot

BA in Computer Science

Emmanuel College

May 16, 2006

Proforma

Name:	**Jonathan Worthington**
College:	**Emmanuel College**
Project Title:	**Virtual Machine Bytecode Translation**
Examination:	**BA in Computer Science, May 2005**
Word Count:	**11,971**
Project Originator:	**Jonathan Worthington**
Supervisor:	**Dr T. Griffin**

Original Aims of the Project

This project aimed to investigate a possible solution to virtual machine interoperability problems by implementing a translator from the .Net Common Language Infrastructure's bytecode, which runs on a stack based VM designed by Microsoft, to Parrot bytecode, which runs on a register based VM being developed by the Perl community. The translator was to cover a subset of the .Net features, including arithmetic and logical operations, branches, object-oriented features such as classes, methods and fields, managed pointers and exceptions.

Work Completed

The translation of all planned features was implemented along with a couple of extensions. By the end of the project, over 90% of the .Net instruction set and 77% of the .Net standard library could be translated. A declarative micro-language was built as part of the project to manage complexity. In addition, the pluggable design of the project allowed a number of stack to register code mapping algorithms to be compared.

Special Difficulties

None.

Declaration

I, Jonathan Worthington of Emmanuel College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

1 Introduction **1**

1.1 Virtual Machines 1

1.1.1 The .Net Common Languages Runtime 2

1.1.2 Parrot 3

1.2 Options For Interoperability 4

1.2.1 Modify The Compiler 4

1.2.2 Embed 4

1.2.3 Bytecode Translation 5

1.3 Aims Of The Project 6

2 Preparation **9**

2.1 Understanding The Core Functional Requirement 9

2.1.1 Translating Metadata 9

2.1.2 Translating Instructions 10

2.1.3 Stack To Register Mapping 10

2.1.4 Tracking The Stack Type State 11

2.2 Non-functional Requirements 11

2.2.1 Manageable Complexity 11

2.2.2 Conformance To Parrot Conventions 12

2.2.3 Robustness 12

2.2.4 Performance 12

2.2.5 Standalone 13

2.3 System Design 13

2.3.1 The Big Picture 13

2.3.2 The Metadata Translator 13

2.3.3 The Instruction Translator 15

2.3.4 PIR Compilation 17

2.4 Software Engineering 17

3 Implementation **19**
3.1 Digging Into The Metadata Translator . . . 19
3.1.1 Writing The Initial PMCs . . . 19
3.1.2 Generating Classes And Method Stubs . . . 20
3.1.3 Generating Locals And Parameters . . . 21
3.1.4 Getting Stressed Early . . . 22
3.2 Building The Instruction Translator . . . 23
3.2.1 Implementing The Translator Builder . . . 23
3.2.2 Implementing A Basic SRM . . . 24
3.3 Translating Basic Instructions . . . 25
3.3.1 Loading Locals And Parameters . . . 25
3.3.2 Storing Locals And Parameters . . . 26
3.3.3 Arithmetic And Logical Operations . . . 27
3.3.4 Branches . . . 27
3.3.5 Checked Arithmetic . . . 29
3.3.6 Conversions . . . 30
3.4 Calling . . . 31
3.4.1 Non-virtual Calls . . . 31
3.4.2 Virtual Calls . . . 32
3.4.3 Mapping Static Overloading Onto MMD . . . 32
3.4.4 Translating The Factorial Program . . . 33
3.5 Object Oriented Constructs . . . 33
3.5.1 Instance Fields . . . 33
3.5.2 Static Fields . . . 34
3.5.3 Inheritance . . . 34
3.5.4 Interfaces . . . 35
3.5.5 Abstract Classes . . . 35
3.6 Arrays . . . 35
3.6.1 Parrot's Support For Arrays . . . 36
3.6.2 Array Creation . . . 36
3.6.3 Loads And Stores . . . 36
3.6.4 Getting Array Length . . . 36
3.7 Managed Pointers . . . 37
3.8 Exceptions . . . 37
3.8.1 Contrasting .Net And Parrot . . . 37
3.8.2 From Protected Regions To Pushes, Pops And Marks . . . 38
3.8.3 Typed Handlers . . . 38
3.8.4 Finally Handlers . . . 39
3.9 Value Types . . . 40

3.9.1 Value Types Become Classes With A Property 40
3.9.2 Initialization 40
3.9.3 Copy On Load 40
3.9.4 Box and Unbox Instructions 41
3.10 More Advanced SRM Modules 41
3.10.1 Mapping The Stack Onto Registers 41
3.10.2 Adding The Lazy Moves Optimization 42

4 Evaluation **43**
4.1 Evaluating The Translator 43
4.1.1 Constructs And Instructions Translated 43
4.1.2 Translating The .Net Class Library 44
4.2 Comparing SRM Modules 46
4.2.1 Generated Code Quality 46
4.2.2 Generated Code Performance 48
4.2.3 Translation Time 49
4.3 Comparing Performance With A .Net VM 50
4.4 Software Engineering Evaluation 50

5 Conclusion **51**
5.1 Bytecode Translation Works 51
5.2 Code Less, But Smarter 52
5.3 Future Directions 52

Bibliography **53**

A Sample Regression Testing Script **55**

B Recursive Calling Regression Test **61**

C SRM Comparison Benchmark **63**

D Software Engineering **65**
D.1 Planning Good Software Engineering 65
D.1.1 Write The Documentation 65
D.1.2 Regression Testing 65
D.1.3 Backups 66
D.1.4 Version Control 66
D.1.5 Tools 67
D.2 Evaluating Software Engineering 67
D.2.1 Implementation 67

D.2.2 Documentation 68
D.2.3 Regression Testing 68

E Managed Pointers **69**
E.1 Considering Possible Parrot Safety Problems 69
E.2 A Managed Pointer PMC 70
E.3 Managed Pointers To Array Elements 70
E.4 Managed pointers to fields 70
E.5 Managed pointers to registers 70

F Project Proposal **73**

Acknowledgements

First and foremost I am very grateful to Dr. Tim Griffin, who has been a constant source of advice, inspiration and friendship throughout the project. I would also like to thank the many other people from the Computer Laboratory at Cambridge University who have given their time to listen to my ideas and/or provided encouragement. These include Dr. Neil Dodgson, Prof. Alan Mycroft, Dr. Arthur Norman, Kate Taylor and Prof. Glynn Winskel.

I probably would never have embarked on this project had it not been for my involvement with developing the Parrot virtual machine, and I owe a great deal of thanks to the Parrot development team for encouraging my interest in the field. Their help and encouragement has been valuable throughout this project. While I suspect there are names that I have missed from this list, the following people have helped in a wide range of ways, from commenting on my ideas through to fixing bugs in Parrot that would have hindered development of the translator: Leopold Toetsch, Jerry Gay, Chip Salzenberg, Will Coleda, Larry Wall and Nicholas Clark. Also, thanks to Paolo Molaro from the Mono Project.

I am also very thankful to my proof readers, who found all of the little mistakes that managed to creep in: Ian Saunders, David Cornish and Leopold Toetsch.

Last but not least, much thanks goes to the many friends who have kept me vaguely sane throughout the project, listened to my ramblings about it that probably made no sense whatsoever and even made me food to fuel the late night hacking sessions. No list of names this time, but you know who you are and you are all wonderful people.

Chapter 1

Introduction

> Love virtual machines did he,
> Shared libraries made his day.
> But libraries for VM B,
> Wouldn't work on VM A.

Compiling high level languages down to code for a virtual machine instead of native code is becoming increasingly popular. Virtual machines, such as the JVM, the .Net CLR and Parrot, abstract away the details of the underlying hardware and operating system, simplifying porting and distributing programs.

At the same time, the use of libraries of existing code remains desirable. When these libraries and the programs using them are written in native code, this can be as simple as defining some common calling convention and executing a jump instruction. With the growth of a number of virtual machines, there is a new problem: a library may run on VM A, but the program that wishes to use it is running on VM B. In this project I have explored one possible solution to this problem, involving translating the code that runs on one virtual machine to semantically equivalent code that will run on another.

1.1 Virtual Machines

The two virtual machines that were selected for use in this project are the .Net CLR (Common Language Runtime) and Parrot. Translation will be performed from the .Net CLR to Parrot.

Figure 1.1: The Mono project is named after the Spanish word for monkey.

1.1.1 The .Net Common Languages Runtime

The .Net CLR, part of the Common Languages Infrastructure, was developed by Microsoft and has been published as an ECMA[1] standard[4]. There is also an open source implementation called Mono, the name coming from the Spanish word for monkey (figure 1.1). It is a stack based virtual machine designed to support multiple languages, achieving interoperability between them by requiring that they meet the Common Language Specification. The CLS lays down a number of restrictions (for example, multiple inheritance is forbidden). It provides support for a range of high level language constructs, including:

- Classes, fields and methods
- Interfaces
- Single inheritance and multiple interface implementation
- Method calling, overriding (providing virtual methods) and overloading
- Delegates (safe function pointers)

[1] http://www.ecma-international.org/

- Arrays
- Exceptions

The runtime provides garbage collection and, through implementing certain class library methods inside the virtual machine, abstracts away the underlying operating system while providing file and network access and threading support.

1.1.2 Parrot

The Parrot virtual machine project was started by the Perl community, the name coming from an April Fool's day joke that had the Perl and Python languages being merged to create a new language called Parrot. This reflects the intention that Parrot should support many languages and allow interoperability between them. I have been involved with Parrot development for almost three years, writing code and documentation and giving talks[6][5].

Parrot is a register machine[3] with variable sized register frames. It provides for the rich runtime requirements of languages like Perl 5, Perl 6, Python and Ruby, which may require their parsers to be available, enable a lot to be done symbolically (by name) and even allow inheritance hierarchies to be changed or new methods to be added to classes.

Interoperability is achieved through PMCs (Parrot Magic Classes). A PMC defines a type that implements some subset of a fixed set of v-table methods. These methods represent common operations such as negation, increment, keyed access (e.g. for implementing an array) and so on. This way types can provide language specific behaviour (for example, incrementing the string "ABC" in Perl gives the result "ABD", but doing so in Python will throw an exception).

Parrot provides support for a wide range of language features, including:

- Subroutines
- Object orientation - classes, fields, methods, objects and multiple inheritance
- Closures
- Continuations
- Coroutines
- Exceptions

- Direct and indirect subroutine and method calling with continuation passing style and optional multi-method dispatch[2]
- Many built-in types, such as arrays and hash tables
- Namespaces
- Calls back into language compilers at runtime

Like the .Net runtime, Parrot provides garbage collection. Access to features provided by the underlying operating system are through a mixture of built-in PMCs and special instructions.

1.2 Options For Interoperability

A number of options are available for achieving interoperability between virtual machines.

1.2.1 Modify The Compiler

Libraries tend to be written in some high level language. One possible solution would be to re-implement the compiler for that language so that it emits Parrot instructions rather than .Net ones. If the compiler was available to modify, this might only involve changing the code generation phase.

A major problem with this approach is that it relies on the source code for the library being available, and even if it is the source code for other libraries that it depends on may not be. Also, this is not a good general solution to the problem, since .Net was designed to support many languages. Therefore there will be many compilers that need to be modified. In its favour, provided it sticks to the defined conventions it would produce real Parrot bytecode that should interoperate well with programs written in other languages running on the Parrot VM.

1.2.2 Embed

Virtual machines tend to have an embedding interface; Mono, the open source implementation of the .Net CLR, is no exception[7]. The Mono VM could be embedded within Parrot and "proxy objects" provided that appear to be normal

[2]I'm using MMD rather than overloading here to indicate that in Parrot, what method to call is decided at runtime; with the .Net CLR this is fixed at compile time.

Parrot objects, but actually just wrap around a .Net object and forward field accesses, method calls and other such operations to the embedded .Net VM.

The advantage of this approach is that you reach a basically usable solution in a short space of time compared to the other approaches described here. However, there are a lot of gaps to bridge between the two virtual machines. For example, consider instantiating a .Net class from Parrot code. For a truly transparent solution, that should look the same as instantiating a class that was implemented in code running on the Parrot VM. To achieve this, all .Net classes need to be registered in some way with the Parrot VM. There are other situations where similar solutions - duplicating entities in the Parrot VM that exist in the .Net VM - are required. Despite appearing to be an easy answer at first glance, once you start asking questions about whether X or Y "just works" this approach starts to fall down.

The other problem is with regards to memory usage and performance. Memory usage is going to be painful anyway since both VMs have to be loaded into memory; the duplication of state that is often required to achieve transparency only makes the situation worse. Performance is also hit hard. At startup both VMs have to initialize, leading to slower startup times. Performance across the boundary between the VMs will also be poor. Consider calling a .Net method from Parrot code; first a call would be made on the proxy object, which in turn uses the .Net embedding interface to actually make the method call. That still leaves us to consider issues relating to the differing calling conventions. When you consider that both of these virtual machines are capable of JIT compiling some calls to native code, the performance hit becomes apparent. That said, once the call into the .Net code has been made, performance should be good.

1.2.3 Bytecode Translation

This approach takes the (binary) code that would run on one virtual machine and translates it to code that will run on another. In the .Net to Parrot case, it takes a .Net EXE or DLL file and outputs a PBC (Parrot Bytecode) file. .Net instructions and constructs are mapped to semantically equivalent Parrot instructions and constructs (figure 1.2).

Translating at this level means that the translation is independent of the high level language, so compilers need not be modified. It also lacks the issues that embedding a .Net VM had - the process of running the translated code and calling into it is just the same as for any other language that compiles to Parrot. Another nice property of bytecode translation is that if each .Net instruction can be translated into one or more Parrot instructions that have equivalent seman-

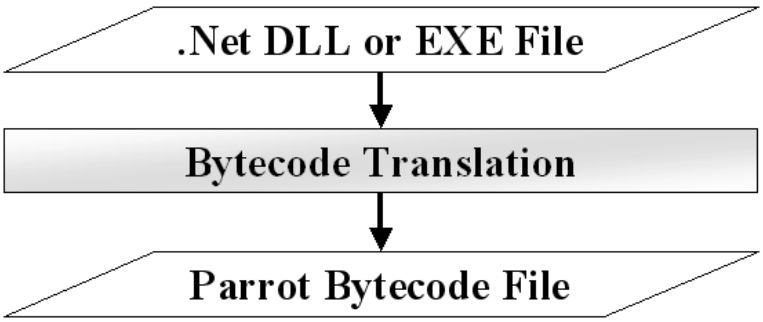

Figure 1.2: Bytecode Translation Overview

tics then, by well founded induction with translated instruction as base cases, the entire translated program should have the same semantics as the original program.

Despite these good properties, bytecode translation is not a simple solution. Stack code needs to be turned into register code and some .Net constructs require non-trivial transformations to provide equivalent semantics on Parrot. Producing good quality code may be difficult; compilation throws away information that may have helped pick more appropriate Parrot instructions. A weakness over the embedding approach is that it takes much longer before real world .Net code can be run on Parrot.

1.3 Aims Of The Project

The project aimed to explore the bytecode translation approach to virtual machine interoperability by designing and implementing a .Net CLR to Parrot translator. As the time available to complete the project would not allow for a complete translator to be implemented, I selected a subset of .Net CLR functionality to translate. This included instructions and constructs relating to:

- Arithmetic and logical operations
- Branching and comparison
- Classes and objects
- Fields and methods (both instance and static)
- Constructors, class initializers and finalizers
- Method calling, including method overriding and overloading

- Exceptions
- Type casting and coercion[3]
- Managed references
- Arrays

The translator was to be considered a success if it translated programs limited to using these features. As a more substantial test, I planned to apply the translator to the Mono implementation of the .Net Foundation Class Library, consisting of a total of over 5,800 classes spread over 40 .Net libraries.

From the outset the intention was to release the translator into the Parrot community once this project was over, making a high quality implementation that would enable other developers to contribute in the future an important secondary aim.

[3]Casting is change of type, while a coercion also changes the representation.

Chapter 2

Preparation

So a translator he conceived;
Designed so it would be,
Declarative and pluggable,
To manage complexity.

2.1 Understanding The Core Functional Requirement

The key functional requirement is translating .Net bytecode into semantically equivalent Parrot bytecode. However, there is more to bytecode translation than simply mapping one instruction to another. The main problem breaks down into four sub-problems, which will later map to concerns that are separated out in the design.

2.1.1 Translating Metadata

.Net programs are not solely described by a sequence of instructions but also by metadata. This metadata is mostly stored in a series of tables, each table describing a particular entity and often referencing rows in another table, somewhat like a relational database. However, an extent-list style approach is used to avoid fully representing the relations.

Entities described by the metadata tables include:

- Types defined in this file
- Methods that a type has

- Parameters that a method takes and their types
- Fields that a type has
- Signatures
- Other modules (.Net libraries) that this one depends on
- Types imported from other modules
- Members (methods and fields) imported from other modules

These metadata tables are referenced by some .Net instructions. For example, the `call` instruction, which calls a method, specifies a row in the `MethodDef` table (methods defined in this file) or the `MemberRef` table (methods defined in another module). Therefore, the metadata is needed during instruction translation.

2.1.2 Translating Instructions

Each row in the `MethodDef` table points to a method body. This contains a small header, followed by a sequence of .Net instructions, optionally followed by an exception handlers table. The sequence of .Net instructions defines the method's semantics.

.Net instructions have variable length. The instruction always starts with an instruction code that may be one or two bytes in length, followed by zero or more arguments. The number of arguments is usually constant; only the `switch` instruction takes a variable number of arguments (the count being specified as the first argument). Note that instructions such as `add` take no arguments, but instead take their operands from the top of the stack.

The instruction translator needs to map each .Net instruction to one or more semantically equivalent Parrot instructions.

2.1.3 Stack To Register Mapping

.Net is a stack machine whereas Parrot is a register machine. This means that items that would be on the .Net stack at runtime instead need to be placed into Parrot registers. Furthermore, when translating instructions that would pop operands from and push results onto the stack, the registers containing the operands or results now need to be explicitly provided. For example, the .Net code to add the numbers 19 and 23 together would look like this:

```
ldc.i4 19    // Load 19 onto the stack
ldc.i4 23    // Load 23 onto the stack
add          // Pop top 2 stack items, add, push result
```

Whereas in Parrot assembly, it may look like this:

```
set I1, 19      # Load 19 into integer register 1
set I2, 23      # Load 23 into integer register 2
add I0, I1, I2  # I0 becomes I1 + I2
```

2.1.4 Tracking The Stack Type State

In .Net, values of a range of types may be placed on the stack. Similarly, Parrot has several register types. Therefore, the translator needs to select the correct Parrot register type to use when placing an item that would be on the stack into a register. To do this, the types of data on the stack need to be known.

.Net instructions often convey little about the type of data they are operating on. For example, the `add` instruction in .Net could add two floats or two integers; its behaviour depends on the contents of the stack. Instead, .Net requires that the types of values on the stack, known as the stack type state, can be determined statically at any point in a program by a single pass through the instructions leading up to that point. This is achieved using a data flow analysis, made possible because the types of values being loaded onto the stack are always known and the way that each instruction transforms the stack type state is specified.

2.2 Non-functional Requirements

This section looks at some of the requirements that were placed on the design of the translator that were not related to its ability to meet the aims of the project, but that I felt were highly desirable.

2.2.1 Manageable Complexity

This project has a number of aspects that are closely coupled; while metadata translation can happen alone, instruction translation, stack to register mapping and stack type state tracking are all very intimately related. As in real life, getting too intimate leads to excessive complexity. Therefore some effort was required to keep these three aspects of the translation process clearly separated from each other and have well defined boundaries between them.

Additional complexity came from the size of the project. I was expecting to reach being able to translate between 150 and 200 instructions, so any bugs that spanned the way many instructions were translated could have been painful if you had to correct their translation code by hand.

2.2.2 Conformance To Parrot Conventions

As far as is possible, the code generated by the translator should conform to Parrot conventions. (Some of these, such as calling conventions and name space policy, are implied by the functional requirements.)

2.2.3 Robustness

The translator will not implement all .Net features and therefore will not be able to translate all code. However, it should be possible for it to recover gracefully and continue if it does run into a class that it cannot translate, perhaps because it encountered an unknown instruction or failed to track the stack type state properly. Equally, not all metadata tables will be "understood" by the translator, but that should not stop it from reading other tables, even if they follow one that is not used.

2.2.4 Performance

There are two areas that performance matters.

Translation Performance

The time taken to produce the translation is not critical since the translation is usually only performed the one time. It is better to spend more time at translation time and less at runtime if there is a trade-off to be made. However, there is a cut-off point - the translator should be able to translate some of the .Net class library and this contains some large modules. For example, the Mono Project's implementation of `mscorlib.dll` contains some 1354 types and is 1.85 MB in size. Taking a minute or two to translate this on a reasonably modern workstation is acceptable; taking over ten minutes is not.

Runtime Performance

If the translation produced runs an order of magnitude more slowly than the original code did under the .Net CLR then it may be too slow to be usable in any

real world situation. Equally, expecting the translated code running on Parrot to perform as well as the original code running under the .Net CLR is unreasonable; Parrot is at an earlier stage in its development, is less optimized and by its nature can not perform as well as the .Net CLR in some areas, since it can make fewer assumptions.

2.2.5 Standalone

Ideally the translator should be able to run anywhere that Parrot can. The easy way to achieve this is to ensure that the only dependency that the translator has is on Parrot itself. Practically this puts some limits on the languages that are chosen to implement the translator. As much of it as possible should be implemented in something that compiles down to Parrot bytecode. Parrot itself and other Parrot extensions tend to be written in C, so for anything that Parrot bytecode is not suitable for this is going to be the language of choice.

2.3 System Design

2.3.1 The Big Picture

Figure 2.1 describes the translation process, starting with an input .Net module and concluding with a Parrot bytecode file.

The metadata translator takes the .Net module, reads its metadata tables, and builds a tree of PMCs (Parrot data structures that can be manipulated from both C and Parrot programs). Control is then passed to the instruction translator, which uses the data collected into the PMCs and the .Net instruction stream to produce equivalent Parrot Intermediate Representation code. Finally, this PIR is compiled to Parrot Bytecode; the PIR compiler is a part of Parrot and invokable from Parrot programs.

2.3.2 The Metadata Translator

The input to the metadata translator is a binary file. More specifically, it is a PE[1] file that contains a short loader that runs on Windows to load the .Net runtime. These are of little interest to the translator beyond validating the file is in the expected format and pointing to where the .Net metadata tables start.

I decided that, since there were no stable Parrot-targeting languages at the start of the project that were good for handling binary files, the metadata trans-

[1]The Windows Portable Executable format by Microsoft.

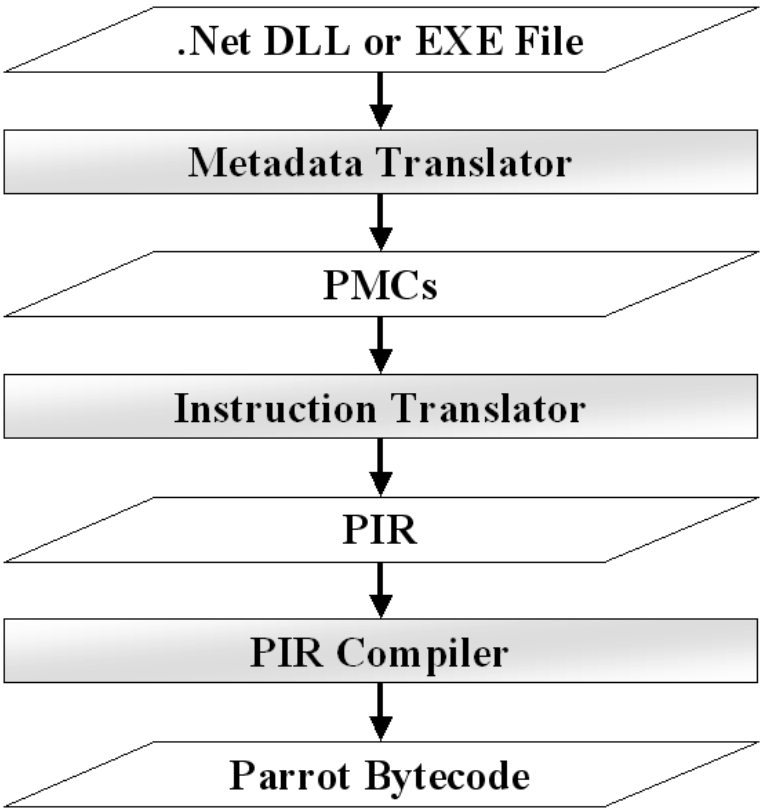

Figure 2.1: System overview

lator should be implemented in C. C is particularly strong when it comes to processing binary data quickly and, while C usually comes with the curse of memory management, the PMCs that the data was to be stored in are eligible for garbage collection by the Parrot runtime.

PMCs themselves are implemented in C. I chose to implement PMCs to represent a row from metadata table that was of interest. An existing array PMC would be used to hold all rows in a table. PMCs can point to an underlying C structure; this structure was used to store the metadata that was read and then PMC methods were implemented to access that data from Parrot bytecode, since Parrot programs can only access the data through the public interface of the PMC and not the underlying structure directly.

I could have chosen to implement the metadata translator completely in PIR, but this would have made it a much more time-consuming implementation task.

2.3.3 The Instruction Translator

The instruction translator is invoked for each method that needs to be translated and should generate PIR for that method. I decided that the instruction translator should be implemented in PIR. This is quite tedious to write, but that was not much of an issue since the instruction translator was to be...

Begotten, Not Created

To manage the complexity of the instruction translator, it is desirable to keep instruction translation, stack to register mapping and stack type state tracking apart. However, all of these are very tightly linked and translating any given instruction involves all three of them. At the same time, to achieve good performance in the instruction translator "straight line" code is preferable.

The solution I came up with has the instruction translator being generated by a script, known as the translator generator, from a file that describes how to translate each .Net instruction to a Parrot instruction and a stack to register mapping module. Since this was essentially a text-munging job, the script to handle this was written in Perl. Perl is also used by other Parrot build and testing tools, so the standalone requirement (2.2.5) is not broken.

Declarative Instruction Translation[2]

Writing lots of repetitive code is not only really boring, but often also results in a program that is difficult to maintain. The code to translate each of the 200+ .Net instructions to PIR is often very similar, making it a good candidate for boring me and a likely source of errors. Therefore, I created a small declarative "language" to describe how each instruction should be translated. While it needed to allow PIR to be hand crafted for more complex instructions, it provided a trivial and compact way to specify how to translate many of them. For example, the translation rule for the add instruction looks like this:

```
[add]
code = 58
pop = 2
push = 1
class = op
instruction = ${DEST0} = ${STACK0} + ${STACK1}
```

The name of the instruction, in this case "add", starts the declaration. The "code" field is the number of the .Net instruction, "pop" is the number of

operands that it pops off the stack and "push" is the number of results that it pushes onto the stack. The "class" field describes what type of instruction this is; this is used to mark instructions that will need to be handled in special ways, such as loads, stores, branches and calling related instructions. Finally, the "instruction" field specifies what PIR to emit for this instruction. This can optionally be replaced with a "pir" field, which provides PIR code to insert into the generated translator that generates the PIR for the instruction. An additional field, "arguments", is used to specify any arguments the instruction takes.

In the "instruction" field some meta-variables were used. Syntactically they consist of a dollar sign followed by the name of the meta-variable in curly brackets. These are substituted for by the translator generator (that is, when the instruction translator is being built). In this case, `${STACK0}` and `${STACK1}` correspond to the two top stack locations and `${DEST0}` corresponds to the stack location that the result would be pushed to.

Pluggable SRM (Stack to Register Mapping)

There is more than one way to turn stack code into register code. One easy to implement approach that has the advantage of "obvious correctness" is to use an array to emulate the stack, popping values from it into registers, then performing the instruction, then pushing any results that were also placed in registers back onto the it. However, the runtime performance of the generated code will be poor.

At the other end of the scale, we could map each stack location onto a register and try to perform on-the-fly "copy prorogation" to eliminate redundant moves. Implementing this is more subtle as control flow needs to be accounted for, but the register code that is generated looks like something you might expect a vaguely sensible compiler to produce.

The first approach has the advantage that it can be implemented quickly and that it will probably work correctly. This is great for development work, but the second approach is what is needed for production use. Since development is on-going even when production use is taking place, and also to enable new ideas for SRM to be tested, I decided that SRM should be pluggable, allowing you to choose which one you want when you build the translator.

This meant that an interface that would support many different approaches to SRM was required. It turns out that by grouping instructions into the classes discussed in the last section and providing "pre" and "post" events for each of those instruction types, you achieve an interface that is flexible enough to implement a wide range of SRM modules.

Some of the handlers are passed parameters and/or required to initialize certain meta-variables (which are mapped to real variables by the translator generator). For example, since the SRM decides which registers map to conceptual stack locations, it is responsible for setting `${STACKn}` and `${DESTn}`.

Stack Type State Tracking

As discussed in the functional requirements, the stack type state needs to be tracked. The translator generator would insert the code to do the book-keeping for this. The current stack type state would be made available through the $STYPES meta-variable.

Type information is added to the type state on a load instruction, leaves on a store instruction and is transformed by other instructions. Instructions that transform the type state need to provide PIR that implements the transformation through an additional "typeinfo" field in its translation rule. This usually makes an entry in the array $DTYPES specifying the types of the results. The transformation code and SRM modules are never meant to directly modify $STYPES.

2.3.4 PIR Compilation

As mentioned, this is delegated to the Parrot PIR compiler. However, it is worth discussing why I chose to generate PIR rather than Parrot bytecode directly. Firstly, Parrot is still undergoing development and while PIR is generally stable, the Parrot bytecode format is liable to change. Secondly, PIR also hides away the underlying details such as the calling conventions, making it easier to generate. Thirdly, textual output is far easier to debug than binary output. Finally, the PIR compiler will do register minimization, meaning that the translator did not need to implement this. These factors greatly outweigh the possible translation performance improvements.

2.4 Software Engineering

Owing to space limitations, discussion of testing methodology, backup policy, documentation requirements and choice of development tools can be found in appendix D.

Chapter 3

Implementation

For weeks he toiled, day and night,
Fuelled by chocolate and caffeine.
And wove his dreams into code:
A translator like none e'er seen!

3.1 Digging Into The Metadata Translator

In any sizeable implementation task there is more than one way to begin. The best starting point is the one that gives you something to test as early as possible, and the metadata translator fits this requirement since it is not dependent on any other part of the system and can be tested alone.

The final metadata translator would need to read and store data from many tables, but not all of this data would be needed to satisfy the needs of the initial instruction translator. Therefore a fairly minimal metadata translator was implemented at this stage, keeping in mind that much more would need to be added later.

3.1.1 Writing The Initial PMCs

The initial metadata translator required four PMCs to be implemented.

- DotNetAssembly, used to represent a .Net EXE or DLL file
- DotNetClassMetadata, storing metadata about a .Net type defined in the file
- DotNetFieldMetadata, storing metadata about a field belonging to a class defined in the file

- DotNetMethodMetadata, storing metadata about a method belonging to a class defined in the file

DotNetAssembly would have an array of DotNetClassMetadata PMCs and these in turn would have arrays of DotNetFieldMetadata and DotNetMethodMetadata PMCs.

The only PMC that was to be instantiated from PIR was DotNetAssembly. This PMC would contain a method to load a .Net assembly, read the metadata and instantiate other PMCs to represent the classes, fields and methods that were described. Further methods on DotNetAssembly objects would return the array of DotNetClass PMCs and so on. Essentially, the flat binary metadata tables were being turned into a tree of PMCs that could be walked in PIR code.

While implementing these PMCs was mostly just a matter of writing the code, there was one issue that needed some care. PMCs are eligible for garbage collection, and since this is reachability based it was important to ensure that every PMC in the tree would get marked alive. Part of this was implementing a custom `mark` method (one of a PMC's standard v-table methods) that tells the garbage collector about any other PMCs that this one references. The other part was making sure that new PMCs got placed into the tree as soon as they were created. For example, rather than creating an array PMC, populating it and then adding a reference to it from its parent, the reference should be added right after creation.

3.1.2 Generating Classes And Method Stubs

With the metadata describing classes, fields and methods stashed in PMCs, it was possible to write some simple code to iterate over the classes and generate PIR to register them and declare their attributes (the Parrot terminology for what .Net calls instance fields).

In Parrot a class is created by placing its methods into a namespace with the fully qualified name of the class and using the `newclass` instruction, supplying this namespace. This instruction registers the class and hands back a ParrotClass PMC that can then be used with the `addattribute` instruction.

```
.namespace [ "Testing.Test" ]

.sub "__onload" :load
    .local pmc type
    type = newclass "Testing.Test"
    addattribute type, "x"
```

```
    addattribute type, "y"
.end
```

Note that, unlike the .Net CLR, classes are created at runtime. To ensure this happens before a class is instantiated we mark the Parrot sub containing this initialization code with the "`:load`" modifier, meaning that it runs when Parrot bytecode file is loaded.

In addition to the class registration code, I could now also generate empty Parrot method bodies for each method in the class; later, the instruction translator would be called at this point to translate the method body to PIR. An empty method body for a method named "Add" would look like this:

```
.sub ‘‘Add’’ :method
.end
```

In Parrot a method, marked with the "`:method`" modifier, is a special case of a subroutine. Unlike a normal subroutine, it assumes the first parameter is the invocant and makes it available under the name "`self`". It also allows for virtual calls to be made on it, thus supporting overriding.

A later modification checked whether the method's static flag was set in the .Net metadata. This signifies that the method is a class method, which in Parrot maps to a normal subroutine. In this case, the "`:method`" modifier is not emitted.

3.1.3 Generating Locals And Parameters

The parameters that a method takes and its local variables are both described in the metadata. More specifically, they are described as signatures that are stored in a compressed[1] binary format in a blobs heap. The metadata tables provide an offset into this heap.

I implemented a simple Signature PMC to help with reading signatures, handling the decompression and throwing an exception if an attempt was made to read past the end of the signature. The code to parse the local and parameter signatures was implemented in PIR.

Given the following method, written in C# and compiled down to .Net bytecode:

[1]The simple compression algorithm only deals with integers, using the upper bit(s) of the first byte to specify how many bytes describe the number. For example, a 0 in the MSB means "this number is fully described by this byte", whereas the upper two bits being "10" means "this number is fully described in this byte and the next".

```
public int Add(int x, int y)
{
     int z = x + y;
     return z;
}
```

The metadata translator would now produce the following Parrot code:

```
.sub "Add" :method
    .param int arg1
    .param int arg2
    .local int local0
    .local pmc arg0
    arg0 = self
.end
```

Here, "`.param`" is PIR syntax that hides away the Parrot calling conventions; when compiled to Parrot bytecode, the names `arg1` and `arg2` will be mapped to registers. Similarly, "`.local`" declares a name for a register that will hold a local variable.

Additionally, notice that the register types have been specified, using the PIR keywords `int` and `pmc`. The .Net types have been extracted from the signatures, and another routine has been used to map the .Net types to an appropriate Parrot register type.

The final two lines are probably the most curious. Recall that when the "`:method`" modifier is used, the invocant is placed into a register and the name `self` is automatically declared as an alias for that register. For uniformity, the name `arg0` is being introduced and assigned the reference stored in `self`; since PIR performs register allocation, in the Parrot bytecode that is produced these two names would end up mapping to the same register and the assignment would be eliminated. This is done to ease code generation of ldarg (load argument) instructions.

3.1.4 Getting Stressed Early

With the first cut of the metadata translator in existence, I provided a large DLL from the .Net class library to it as an early stress test. There was no reason why it should not have been able to produce class registration code for each class along with method stubs for each of their methods. After a few small bugs were flushed out, this result was achieved.

3.2 Building The Instruction Translator

Three things were needed to build the instruction translator:

- A translation rules file providing the declarative translation rules and type state transformation code for some instructions
- A stack to register mapping module
- A build tool that uses these to generate an instruction translator

The contents of the translation rules file will be explored more in the next section, the build tool and a simple SRM module will be discussed here and more advanced SRM modules will be discussed at the end of this chapter.

3.2.1 Implementing The Translator Builder

The translator builder generates PIR code for the instruction translator. This in turn will take a .Net instruction stream and generate semantically equivalent PIR code. The translator builder is implemented in Perl and moves through a number of steps.

Rules File Parsing

This step parses the translation rules file and places its contents in a data structure that allows easy access to the data for each rule. Some validation is done here to ensure the the translation rules are sane.

Loading A Stack To Register Mapper

Next, an SRM module must be loaded. The module to be loaded is specified as a command line arguments to the script. All modules adhere to a standard interface defined by the class SRM::Base, from which they all inherit.

Emitting Setup PIR

This step outputs a hand-crafted chunk of PIR that contains the setup code for instruction translator. The SRM module's `pre_translate` method is also called, and any code that it returns is placed into the instruction translator too. This gives the stack to register mapper a chance to set up and initialize any state that it will use during the translation process. It may also concatenate data onto the meta-variable `${INS}` if it needs to place anything at the start of the translated PIR that the instruction translator produces.

Building Dispatch Code

Instruction translation is driven by a loop that translate an instruction each iteration. It is desirable to be able to jump to the translation code for a particular instruction efficiently. This step emits PIR code that ensures that this will happen in O(log n) time, where n is the number of instructions we have translation rules for. It operates very much like a binary search.

Building Rule Translation Code

This step takes a translation rule and generates code to translate the instruction in the way described by the rule. The first stage of this is similar for all rules; first a label is emitted that the dispatch code jumps to followed by some trace code to aiding debugging the translator (a command line switch turns this on and off). Following this, code is generated to read in any arguments that the instruction takes. These are made accessible to the translation code through the `${ARGn}` metavariables.

What happens next is dependent on the class of the instruction. Recall that there are several classes of instruction ("op", "branch", "load", "store" and "call"). For all classes, a "pre" SRM method will be invoked before emitting the code to translate the instruction and a "post" SRM method afterwards (for example `pre_op` and `post_op`). Beyond that, the behaviours differ. For example, instructions of class "branch" need to propagate the stack type state to all possible branch destinations, not just the next instruction.

This step is where the three aspects to instruction translated are all brought together and interwoven. It is important to understand that the "pre" and "post" calls into the SRM module are calls made by the translator builder, but no such concept as an SRM module exists in the generated instruction translator.

Emitting Trailing PIR

This step is very similar to "Emitting Setup PIR". Near enough repeating that here would probably only serve to give you a sense of deja vu, but if you like that feeling you can always go back and read it again.

3.2.2 Implementing A Basic SRM

The simplest possible SRM module to implement uses an array to emulate the .Net stack. Amongst Parrot's built-in PMCs is `ResizablePMCArray`, which implements the `push` and `pop` v-table methods for all register types. The SRM will

use these operations to place operands into registers for each instruction and to put any results back onto the stack. For example, the translated output of the "add" instruction when the top two items on the stack are integers would be:

```
$I0 = pop s          # Generated by pre_op
$I1 = pop s          # Generated by pre_op
add $I2, $I0, $I1    # From the translation rule
push s, $I2          # Generated by post_op
```

Clearly, this does not produce good code, but it does produce working code, which is all that was needed at the early stages of the implementation.

3.3 Translating Basic Instructions

Despite the heading, this section covers the translation of over half of the .Net instruction set. These instructions are basic in the sense that they require little or no extra translation machinery beyond what has been developed up to this point. They provide the basic building blocks of all programs - arithmetic, logical and branching operations along with the movement of data from locals or passed parameters onto the stack and vice versa.

3.3.1 Loading Locals And Parameters

In Parrot, locals are located in registers named `local0`, `local1`, etc. .Net has a number of instructions for loading local variables onto the stack, but they are all special cases of one of basic `ldloc` instruction.

The most trivial cases to translate are the instructions `ldloc.0` through `ldloc.3`. A translation rule for one of these instructions is written as follows:

```
[ldloc.0]
code = 06
class = load
pop = 0
push = 1
pir = ${LOADREG} = "local0"
typeinfo = ${LOADTYPE} = ${LTYPES}[0]
```

The first five lines are obvious in meaning, but the last two are less so. The line starting "pir" is not specifying a Parrot instruction to emit, but instead provides PIR code to be placed into the translator. It assigns the name of the register

holding the local variable we wish to load to the `${LOADREG}` metavariable. This line could have been replaced with:

```
instruction = ${DEST0} = local0
```

For load instructions that source their values from fields or indirectly, a construct such as this is required. However, `${LOADREG}` gives more advanced SRM modules a chance to optimize away loads where the value is already held in a register.

The final line gets the type of the local from the 0th element of `${LTYPES}`, a metavariable that maps to an array of local variable types obtained from the locals signature.

The more general variants of the ldloc instruction take a number (of 8 or 32 bits), meaning that a little more effort is required to construct the register name. Loading of arguments is also very similar to this - simply replace "local" with "arg". The following example depicts both of these changes.

```
[ldarg.s]
code = 0E
class = load
pop = 0
push = 1
arguments = uint8
pir = <<PIR
${STEMP0} = ${ARG0}
${LOADREG} = "arg"
${LOADREG} = concat ${STEMP0}
PIR
typeinfo = ${LOADTYPE} = ${PTYPES}[${ARG0}]
```

3.3.2 Storing Locals And Parameters

Translating stores is somewhat symmetric to translating loads; for most load instructions there is a corresponding store instruction. A typical store translation rule looks like this:

```
[stloc.0]
code = 0A
class = store
pop = 1
```

```
push = 0
pir = ${STOREREG} = "local0"
```

Again, the final line could be written another way:

```
instruction = local0 = ${STACK0}
```

Type information is discarded from the stack type state on a store, so a "typeinfo" declaration is not needed.

3.3.3 Arithmetic And Logical Operations

Most of the .Net arithmetic and logical instructions map directly onto Parrot instructions. This is because both virtual machines want to be able to JIT compile these instructions and thus define them to work in the way that typical hardware arithmetic and logical instructions do. As an example, the translation rule for the .Net AND (`and`) instruction is:

```
[and]
code = 5F
class = op
pop = 2
push = 1
instruction = ${DEST0} = band ${STACK1}, ${STACK0}
typeinfo = typeinfo_bin_num_op(${STYPES}, ${DTYPES})
```

The type state transformation code for many arithmetic and logical instructions is implemented in a subroutine since it is the same. The sub implements a table from the .Net standard that maps the types of the operands to the type of the result.

3.3.4 Branches

Branch instructions in .Net always have a single argument that determines the destination of the branch. It is specified as an offset from the program counter as it would be at the start of the next instruction and is a signed value to allow for backward branches. To translate this to PIR, the branch destination needs to have a label.

Before the translation of each .Net instruction, a label is emitted of the form `LABn:`, where `n` is the program counter at that point in the program. Therefore,

generating a destination label is simply a case of adding the argument of the branch instruction to the program counter for the next instruction and postfixing it to to the string "LAB". This is exactly what the following unconditional branch instruction's translation code does.

```
[br]
code = 38
class = branch
arguments = int32
pir = <<PIR
${ITEMP0} = ${NEXTPC} + ${ARG0}
${STEMP0} = ${ITEMP0}
${INS} = concat "goto LAB"
${INS} = concat ${STEMP0}
${INS} = concat "\n"
PIR
```

Meta-variables such as `${STEMP0}` and `${ITEMP0}` are temporary variables, in these cases of type string and integer. They are meta-variables so the translator builder can choose how to name them, so as to avoid naming conflicts with variables used by the translator internals. `${INS}` is the PIR that is being produced, and this translation rule just concatenates code onto it.

Conditional branches only differ in that they pop one or two values from the top of the stack, using `${STACKn}` in generating the Parrot instruction:

```
[beq.s]
code = 2E
class = branch
pop = 2
arguments = int8
pir = <<PIR
${ITEMP0} = ${NEXTPC} + ${ARG0}
${STEMP0} = ${ITEMP0}
${INS} = concat "if "
${INS} = concat ${STACK1}
${INS} = concat " == "
${INS} = concat ${STACK0}
${INS} = concat " goto LAB"
${INS} = concat ${STEMP0}
${INS} = concat "\n"
```

PIR

An additional subtlety with regard to branches is that the stack type state needs to be propagated to the destination of the branch, not just the next instruction as it would normally be. The insertion of code to handle this is provided for by the translator builder. Note that there is a constraint on backward branches - the stack must be empty when they are taken unless the destination is reachable from instructions statically preceding it. This makes single-pass translation possible by removing the need to locate the basic blocks first.

3.3.5 Checked Arithmetic

Checked arithmetic instructions throw an exception when the result of the computation would overflow. Parrot has no corresponding instructions, but its extensibility meant that I could implement them in a dynamic op library[2] that is loaded at runtime when translated .Net code is being run. Using dynamic ops is as simple as creating a file containing entries such as the following one:

```
inline op net_add_ovf(out INT, in INT, in INT) :base_core {
    if (CHECK_ADD_OVERFLOW($2, $3))
    {
        opcode_t *ret = expr NEXT();
        opcode_t *dest = dotnet_OverflowException(interpreter, ret);
        goto ADDRESS(dest);
    }
    else
    {
        $1 = $2 + $3;
        goto NEXT();
    }
}
```

The body here is essentially C but preprocessed by a Parrot build tool. For example, `$1` corresponds to the first register that is given as an operand to the instruction. The `CHECK_ADD_OVERFLOW` macro and similar were based upon ones used by the Mono project[3].

This op can then be used in the translation instruction:

[2]Think DLL file on Windows or SO file on Linux/UNIX.

[3]http://svn.myrealbox.com/viewcvs/trunk/mono/mono/interpreter/interp.c

```
[add.ovf]
code = D6
class = op
pop = 2
push = 1
instruction = net_add_ovf ${DEST0}, ${STACK0}, ${STACK1}
typeinfo = typeinfo_bin_num_op(${STYPES}, ${DTYPES})
```

The only other requirement is that a small piece of code needs to be emitted at the start of the translated output to load the dynamic op library.

```
.sub __LOAD_DOTNET_OPS :load
    loadlib $P0, "dotnet_ops"
.end
```

It would also have been possible to generate a sequence of Parrot instructions that implemented the overflow check. This would have had better performance in the presence of a JIT compiler, but worse performance on platforms where Parrot has no JIT support yet due to the instruction dispatch overhead.

3.3.6 Conversions

Conversion instructions perform coercions from one type to another. The type to coerce to depends on the instruction and the type to coerce from can be found by examining the stack type state. For example, the `conv.i2` instruction coerces whatever is on top of the stack to a 16-bit signed integer (assuming that to do so makes sense).

As Parrot only supports one integer type directly, there are no instructions of this kind. Therefore, they were implemented as dynamic ops. Here is an example of such an op:

```
inline op net_conv_i2(out INT, in INT) :base_core {
$1 = (Parrot_Int2) $2;
goto NEXT();
}
```

And here is the translation rule that uses it:

```
[conv.i2]
code = 68
class = op
```

```
pop = 1
push = 1
instruction = net_conv_i2 ${DEST0}, ${STACK0}
typeinfo = <<PIR
${PTEMP0} = new Hash
${PTEMP0}["type"] = ELEMENT_TYPE_I2
${PTEMP0}["byref"] = 0
annotate_reg_type(${PTEMP0})
${DTYPES}[0] = ${PTEMP0}
PIR
```

Here new type information needs to be introduced. A type is described by a Hash, a built in Parrot PMC implementing a hash table. The value `ELEMENT_TYPE_I2` is a constant from the .Net specification that denotes the 2-byte signed integer type. The `annotate_reg_type` routine takes a type describing hash and annotates it with some extra entries, such as the type of register that such a value should be stored in.

3.4 Calling

The .Net CLR provides a stack based calling mechanism. Arguments are pushed onto the stack left to right and then the method is called. If there is a return value, it is left on the stack. The method to call is an argument to the call instructions, specified as a row in a metadata table.

Parrot also provides standard calling conventions that attempt to cover the needs of many languages. It uses Continuation Passing Style, and under the hood passing is implemented as several variable argument register instructions. A Sub PMC (or any other PMC that implements the invoke v-table method) can be called.

3.4.1 Non-virtual Calls

The .Net `call` instruction does a non-virtual method call; the method specified by the metadata token in the instruction is the one that is called, even if the method has been overridden. This can be achieved in Parrot by looking up the method in the namespace holding the class it belongs to, which can also be determined from the meta-data token.

Assuming that registers `$P1` and `$I2` contain the parameters to be passed and `$I0` is to hold the return value, a call to the "factorial" method in the class "Test" in the namespace "Testing" will translate to the PIR:

```
$P1000000 = find_global "Testing.Test", "factorial"
$I0 = $P1000000($P1, $I2)
```

Where `$P1000000` is being used as a temporary to store a `Sub` PMC.

3.4.2 Virtual Calls

The `callvirt` instruction performs a virtual method call. That is, an object currently viewed as being a `Mammal` that is actually some subtype `Monkey` of `Mammal` may override some method `EatBanana`. Whereas `call` would call the method `EatBanana` as defined by the class `Mammal`, `callvirt` uses the runtime type of the object to decide which method to call. This can be translated directly to PIR method call syntax.

```
$I0 = $P1."EatBanana"($I2)
```

3.4.3 Mapping Static Overloading Onto MMD

Both .Net and Parrot support having methods of the same name with different signatures. However, in .Net it is up to the compiler to resolve which method to call and to emit the correct metadata reference. As Parrot supports dynamic languages, it is assumed that the method to be called can not be determined at compile time and may change. A cache is used to aid performance.[4]

Using Parrot's MMD mechanism will provide most of what is required to support .Net method overloading. However, there is a problem: unlike .Net, Parrot does not recognize different types of integers and floating point numbers as fundamental types. For efficiency it is desirable to have all types of integers stored in I registers, but at dispatch time Parrot will be unable to distinguish between the two types.

A number of options exist to solve this. Name mangling the subs and then using the signature to generate the mangled name when translating the call would work. This avoids Parrot's MMD completely, meaning it is cheaper at runtime

[4]In fact, the instruction stream can be modified at runtime to just have a call to the method that the dispatch algorithm found, so the cost of the dynamic dispatch is amortised. This technique is known as a Polymorhpic Inline Cache.

and that the intended method is always called. However, this really hurts interoperability with other languages running on Parrot.

Another option is to create a class for each of the .Net integer and floating point types that derive either from Parrot's built in Integer or Float PMCs, naming them, for example, "@@DOTNET_MMDBOX_I1" for the single byte signed integer. All methods are then annotated with `:multi(...)` modifiers, which are used to specify the arguments that a method takes that participate in multiple dispatch.[5] Instead of simply specifying `int` for all integer types, the more specific classes are used. Note that this will result in some boxing and unboxing.

Under the hood, Parrot gives methods "long names" that incorporate the signature. For non-virtual calls from within .Net, these can be generated and used when making the call, avoiding dynamic dispatch and improving performance. For calls made from other languages, dynamic dispatch should still be able to do something sensible in most cases.

3.4.4 Translating The Factorial Program

Appendix B shows a (passing) regression test featuring the recursive factorial program. As well as showing that the translation of a number of basic instructions works, it shows that calling has been translated in such a way that both translated .Net code and ordinary Parrot code can make calls to the factorial function.

3.5 Object Oriented Constructs

The object oriented paradigm is at the heart C# and a very significant part of many other .Net languages. Therefore, being able to translate instructions and constructs relating to object orientation is key to being able to translate any real world programs.

3.5.1 Instance Fields

Instance fields are already declared by the metadata translator; the missing pieces of the puzzle are instructions to load and store values to and from instance fields. This is not as straightforward as might be hoped because Parrot's object attributes can only be PMCs, meaning that integers and floats need to be boxed

[5] In some languages, such as Perl 6, it is possible that only some arguments are MMD invocants. With .Net, we must assume they all are.

and unboxed explicitly. For example, here is the translation of loading the integer field named "x" of the object in register `$P0`:

```
$P1000000 = getattribute $P0, "x"
$I1 = $P1000000
```

The second line assigns the PMC to an integer register, unboxing the integer. Storing is similar, apart from the box has to be created and the integer placed into it.

```
$P1000000 = new Integer
$P1000000 = $I0
setattribute $P1, "y", $P1000000
```

Boxing is not required for fields that have a type that maps to a PMC. This is an example of PIR generation that is dependent on the stack type state.

3.5.2 Static Fields

Static fields are essentially global variables, though with some visibility restrictions (which Parrot currently does not provide a way to enforce in the runtime; since the goal is to translate verifiable .Net code, this is not a big concern). Parrot has `find_global` and `store_global` instructions that can be used to look up and/or store a global in a given namespace. As every class has a namespace of its own, a static field translates as a global in the namespace for that class.

The same boxing and unboxing requirements exist as for instance fields. Aside from the different instruction names and the fact that there is no object to do the lookup on, the translation is very similar to that of instance fields.

3.5.3 Inheritance

.Net supports single inheritance of a class. Parrot supports multiple inheritance, which single inheritance is just a special case of. Every .Net class has a parent, except `System.Object` which is the base of all classes.

In Parrot, parents are added to a class using the `addparent` instruction. The metadata translator was extended to emit such an instruction in the initialization method for the class.

```
.sub "__onload" :load
    .local pmc type, parent
```

```
    type = newclass "Testing.Chimp"
    parent = getclass "Testing.Mammal"
    addparent type, parent
.end
```

This introduces an ordering constraint on class creation; a class must be declared before any of its children. This is not an issue for the .Net CLR, since the parent class is specified as an index into the classes metadata table. Therefore, classes need to be sorted in order of depth in the inheritance hierarchy. Any class that has a parent outside of the current module will already have its parent declared, so these classes can be placed at the start of the list. Then an iterative method is used to add all classes that have a parent in the list already, which continues until all classes are added to the list. This results in a list of classes in inheritance depth order in $O(nD)$ time, where n is the number of modules and D is the deepest inheritance hierarchy in the module.

3.5.4 Interfaces

Since Parrot currently lacks support for interfaces, they have to be "faked out" using classes and taking advantage of multiple inheritance to add them as additional parent classes. An interface is translated just as a class would be, but each method contains code to throw an exception if they are invoked stating that an interface has not been fully implemented. The constructor also throws an exception to prevent instantiation of interfaces. This provides all of the required semantics for valid .Net programs and safe failure at runtime otherwise.

3.5.5 Abstract Classes

Parrot also does not provide direct support for abstract classes (that is, classes that are not completely implemented and must be subclassed). The solution is similar to that for interfaces; the constructor and abstract methods all throw exceptions when called.

3.6 Arrays

At the runtime level, .Net only supports one-dimensional, zero-based arrays of fixed size. All elements must be of the same type (or a subtype of the same type). Anything beyond this is supported through the System.Array class.

3.6.1 Parrot's Support For Arrays

In Parrot, arrays are implemented using PMCs. PMCs have keyed v-table operations (that is, v-table functions that take a key as one of their parameters). For arrays, the key is the array index and is expected to be an integer. A number of array PMCs are built in to Parrot, however languages are free to provide their own. The v-table mechanism used to access elements means that the interface to the array is always the same, so translated .Net code will be able to handle being passed a Perl array even though their behaviours are very different (Perl arrays are resizeable, for example, where as .Net arrays are of fixed length).

3.6.2 Array Creation

In .Net arrays are created using the `newarr` instruction, which takes the element type as an argument and the length as an operand. This simply translates to creating an array PMC and assigning the length to it. To prevent boxing and unboxing, arrays that store integer or floating point types are special-cased, so a `FixedIntegerArray` or `FixedFloatArray` PMC is instantiated. All other types get stored in a `FixedPMCArray` PMC.

3.6.3 Loads And Stores

Array loads and stores can be translated trivially to the PIR keyed syntax.

```
$I1 = $P1[$I0] # load
$P1[$I0] = $I1 # store
```

Here, `$P1` is the array PMC, `$I0` is the array index and `$I1` is the the register to load a value from the array into or contains the value to store.

3.6.4 Getting Array Length

The only remaining array instruction is ldlen, which loads the element count of the array onto the stack. This maps to the elements v-table call in Parrot.

```
$I0 = elements $P0
```

Where `$P0` holds the array PMC and `$I0` will hold the number of elements.

3.7 Managed Pointers

A managed pointer holds the address of a local variable or parameter on the stack, a field from an object or an element in an array. These can be used with a number of instructions that load and store data indirectly through pointer.

Translating managed pointers needed careful consideration of VM safety issues and required me to make a couple of additions to Parrot itself as well as implementing a managed pointer PMC and some dynamic ops. There is not space for the full details here; they may be found in Appendix E.

3.8 Exceptions

3.8.1 Contrasting .Net And Parrot

The .Net exception system uses objects to represent exceptions and a per-method extent list of protected regions with associated handlers. These protected regions map to the high level language concept of a try block; if an exception is thrown from within a protected region then handlers containing that region will be searched, innermost first, to find one that can handle the exception. If there is no handler in the current method then the runtime will walk down the call stack searching the handlers of callers. .Net provides four different types of handlers, but only the two that are commonly used have been translated. One of these is the typed handler, which is invoked if an exception is thrown that is of that type. The second is the finally handler, which is run whether or not the protected region was left due to an exception.

The Parrot exception system is based around an exception stack. Handlers are created at runtime using the `push_eh` instruction, specifying a label located at the start of the handler. The last exception handler that was placed on the stack can be popped off using the `clear_eh` instruction. It is also possible to place a mark on the stack and later pop all handlers that are located above that mark together with the mark itself. When searching for an exception handler, the exception stack is checked and the top exception handler is popped off and run. If it does not wish to handle the exception, it can use the rethrow instruction to continue the unwinding of the exception stack. Exceptions themselves are instances of an Exception PMC that provides a keyed interface to store data relating to the exception.

3.8.2 From Protected Regions To Pushes, Pops And Marks

Before translating each instruction, the translator searches the protected regions table for any regions that start at the current instruction. The handlers table is in most to least nested order and therefore must be searched in reverse, since if many regions start at the same location the handler for the outermost region must be pushed first and the innermost last.

For each region that starts at the current instruction, two PIR instructions are emitted: a `push_eh` instruction to put the handler on the stack followed by a `pushmark` instruction that places a mark on the stack matching the row number in the .Net handlers table.

Branches out of a protected region are forbidden; instead the `leave` instruction must be used. This translates in the same way as an unconditional branch but is preceded by code to clear exception handlers from the stack and to run any appropriate finally blocks.

A `popmark` instruction will be inserted to clear handlers as needed. The mark is computed by scanning through the exception handlers table and locating the first protected region that occupies the location being branched to. Immediately following the `popmark`, a `pushmark` will be generated for the same mark; the intention of the popmark is to clear all handlers on the stack that belong to nested protected regions, but the mark that also gets removed is that of the region that will be branched into. If there are a sequence of protected regions within the same enclosing region, failure to restore the mark would cause problems beyond the first in the sequence.

Note that if there is no containing region, the mark 0 should be used. For this to work, a `pushmark 0` is emitted at the top of every translated method and a `popmark 0` at every return.

Figure 3.1 depicts this translation process.

3.8.3 Typed Handlers

PIR is emitted to get the exception object that was thrown and assign the .Net exception object it contains to what the translated program would consider the first stack location (since the stack is considered empty on entry to the handler). PIR will then be emitted that tests if the .Net exception object is of the required type. If it is not then the exception will be re-thrown; otherwise. The translation of the handler code follows.

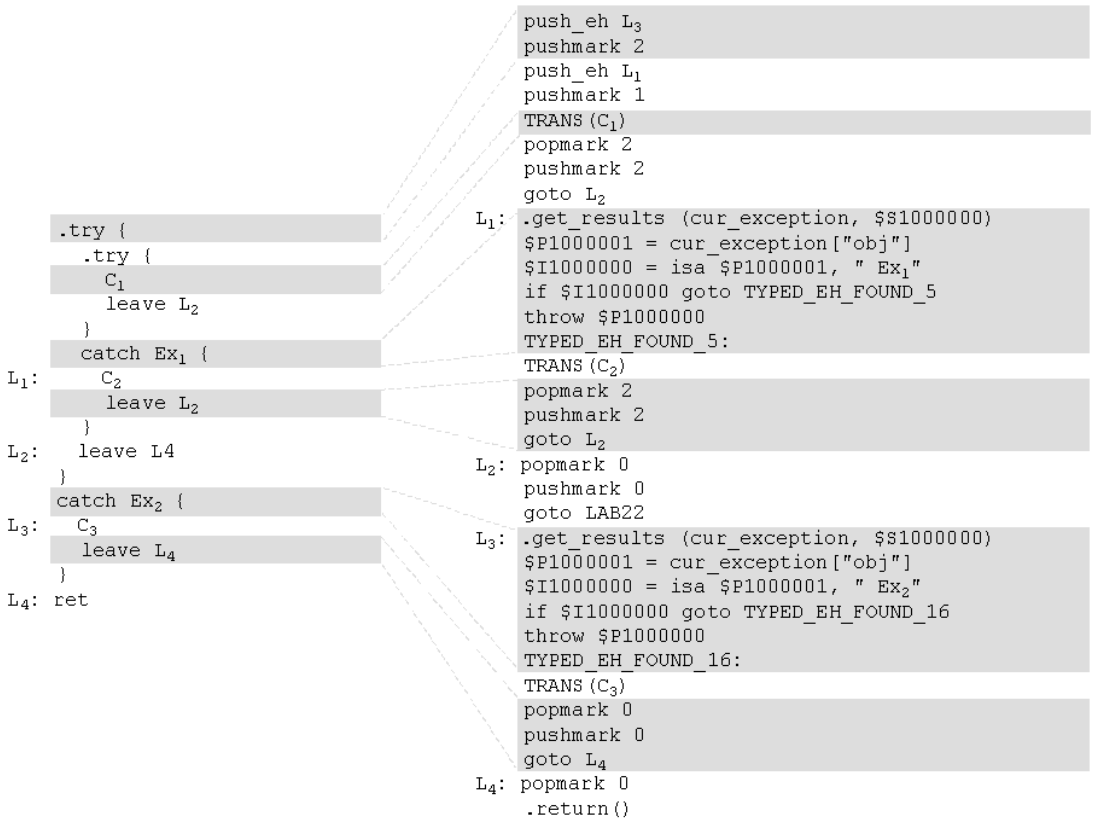

Figure 3.1: Translation of try and catch blocks

3.8.4 Finally Handlers

There are two ways to enter a finally handler. One is while un-winding the exception stack because an exception was thrown. Another is when the leave instruction is used.

The case where the finally handler is walked over is relatively simple to handle. The handler will be invoked as it is walked over and the exception object will be retrieved and stored. At the `endfinally` instruction (used to mark the end of a finally handler) the exception will be re-thrown. This is not completely trivial, since if finally handlers are nested the outermost one must remember which exception to rethrow. Therefore, an array of exceptions waiting to be thrown from finally handlers must be maintained (at runtime of the translated code, not in the translator).

The array of exceptions waiting to be thrown has a second purpose: a null entry can be used to signify that the finally block was entered from a `leave` instruction. In this case, the `endfinally` instruction should instead use the Parrot `ret` instruction, which returns from a subroutine branch made within the

current method. These subroutine branches are emitted when translating `leave` instructions and simply jump to the starts of the required finally blocks (that is, those not walked over while unwinding the stack). Determining which finally handlers to invoke involves looking at the exception handlers table and locating ones that would not have been walked over and are on the path from the current location to the destination of the leave instruction.

3.9 Value Types

Value types in .Net are simply types that exhibit value semantics (for example, they are copied when passed as a parameter or placed on the stack from a local). They are like C structures on steroids; they can have both instance and static fields and methods and may exist in both an unboxed form where they have value semantics and a boxed form where they become an object.

While Parrot provides all the primitives upon which complex value types can be built, it does not provide the level of support or optimization that .Net does.

3.9.1 Value Types Become Classes With A Property

Value semantics aside, value types are very much like objects in that they have fields and methods. Therefore, they translate to Parrot classes and objects in the same way an ordinary class would. There needs to be a way to differentiate between the boxed and unboxed forms, even though in the Parrot translation they are essentially the same thing at a data structure level. Therefore, a PMC property is used to mark an object as boxed. This is optimized for the common case (the unboxed form).

3.9.2 Initialization

Unlike objects, there is no requirement to explicitly instantiate or initialize value types. Therefore, registers holding value type locals need to be initialized at the start of a method.

3.9.3 Copy On Load

When a value type is loaded (from another register, from a field, from an array element and so on), an instruction to clone it must be emitted directly afterwards. Since these instructions are in the "load" instruction class, the translator build can inset the clone instructions automatically. This interacts badly with the

`${LOADREG}` optimization since there is no way to indicate to the SRM that it should emit a clone instruction later. Therefore, the `pre_load` and `post_load` proceed as normal, then afterwards `pre_op` and `post_op` are used to sandwich a clone. This may seem hackful, but it avoids the SRM having to know about value types. A good SRM module can produce code as compact as such cloning code could be anyway.

3.9.4 Box and Unbox Instructions

Both of these instructions need special cases for built-in raw types. For other value types, a more general mechanism is required. Boxing requires that the attributes are copied when the boxing takes place. This is be done with the Parrot clone instruction, then the "boxed" property is set. Unboxing does not require any copying of the attributes; the operation simply needs to unset the boxed property and update the stack type state. However, the .Net `unbox` instruction has an additional subtlety - it does not place the unboxed value itself onto the stack but a managed pointer to it. This is not really a problem, just some extra instructions to emit.

3.10 More Advanced SRM Modules

The SRM presented earlier in the chapter produces working PIR. Here the focus is turned to producing good PIR.

3.10.1 Mapping The Stack Onto Registers

An early paper[1] on translating Java bytecode to native code suggested that, in cases where the stack depth could be statically determined, each stack location could be mapped to a register (figure 3.2). Pushes and pops then became move instructions and a stack is no longer required. The condition that the stack depth can always be determined statically holds in .Net CLI code.

This approach means that each stack instruction becomes one register instruction, provided there is a single register instruction with equivalent semantics to the stack instruction. The array used to fake out a stack is no longer needed. The extra complication is that the SRM must track the stack height (which is trivial since the type state tracker is also doing this) and ensure that this is propagated between basic blocks properly.

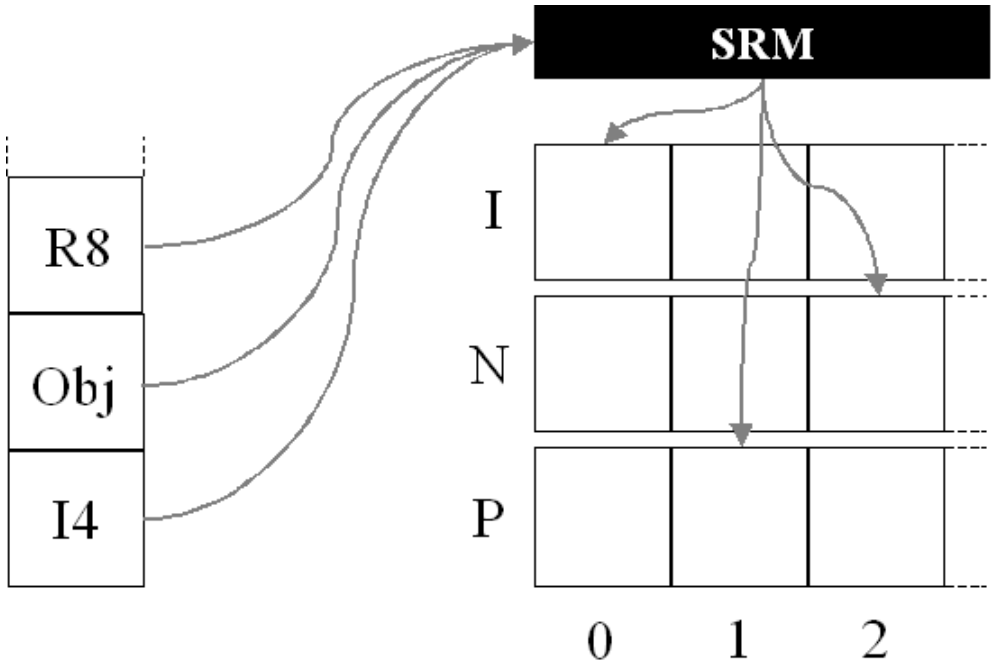

Figure 3.2: Mapping the stack onto registers

3.10.2 Adding The Lazy Moves Optimization

Consider adding two integers stored in local variables. The previous SRM scheme would generate two move instructions followed by the add instruction. This is inefficient since the add instruction could take the values directly from the registers holding the local variables. This SRM tries to avoid generating such instructions by placing loads from registers onto a list of "lazy moves". Then, when the add instruction is reached the two registers on the lazy moves list are used.

There are a lot of ways to make a mess of this; for example, around branches or branch destinations any moves that have been done lazily must be performed. However, it is a reasonably cheap optimization to implement.

Chapter 4

Evaluation

It passed all the regression tests,
Such beautiful code it made!
Some libraries were thrown at it,
And class upon class it slayed.

4.1 Evaluating The Translator

4.1.1 Constructs And Instructions Translated

The translator that I implemented was able to translate all of the constructs and relevant instructions described in the project aims. These were:

- Arithmetic and logical operations
- Branching and comparison
- Classes and objects
- Fields and methods (both instance and static)
- Constructors, class initializers and finalizers
- Method calling, including method overriding and overloading
- Exceptions
- Type casting and coercion
- Managed references
- Arrays

In addition, a number of extra instructions and constructs were translated. These were:

- Value types
- Enumerations
- Loading of required libraries
- Runtime-provided methods (infrastructure for providing methods that the .Net core library marks as implemented by the VM)

Each item presented in these lists have corresponding regression tests that pass under all implemented stack to register mapping modules. Examples of such tests are presented in appendices A and B.

Out of a total of 213 .Net instructions, 197 have translation rules and can be translated to PIR. However, 15 of these rules are marked as questionable, meaning that they may not always produce PIR that is semantically equivalent to the .Net instruction (often because a dubious assumption has been made to simplify implementation).

4.1.2 Translating The .Net Class Library

Regression tests are good for ensuring that individual constructs can be translated and verifying that the translator has met the aims of the project. However, they are less helpful for assessing whether the translator is capable of translating real world libraries. Therefore, I supplied 40 .Net DLLs from the Mono Project's implementation of the .Net Foundation Class Library to the translator. These ranged in file size from 8 KB to 1.85 MB and contained between 3 and 1354 classes.

Table 4.1 presents the number of classes from each file in the class library that were translated. In all cases, the PIR that was generated by the translator was successfully compiled to Parrot bytecode, ensuring that valid code was being generated. Overall, 4547 out of 5881 classes were translated (approximately 77%).

The dependencies that a class has are not accounted for in these results; if a class was translated successfully but a class that it inherits from or uses was not, then it has been counted as having been translated successfully. However, all methods of a class must have been translated for a class to count as having been translated successfully. Table 4.2 presents an analysis of the translation failures.

Table 4.1: Results of translating .Net Foundation Class Libraries

Name	Translated	Total	Percentage
mscorlib.dll	1054	1354	77%
System.dll	508	650	78%
Accessibility.dll	3	3	100%
I18N.CJK.dll	37	38	97%
I18N.MidEast.dll	16	23	69%
I18N.Other.dll	39	47	82%
I18N.Rare.dll	76	113	67%
I18N.West.dll	32	47	68%
I18N.dll	4	5	80%
ICSharpCode.SharpZipLib.dll	48	63	76%
Microsoft.JScript.dll	201	269	74%
Microsoft.VisualBasic.dll	35	84	41%
Microsoft.VisualC.dll	12	12	100%
Microsoft.Vsa.dll	14	15	93%
Mono.CompilerServices.SymbolWriter.dll	23	29	79%
Mono.Data.SqliteClient.dll	10	15	66%
Mono.Data.SybaseClient.dll	31	42	73%
Mono.Data.Tds.dll	33	41	80%
Mono.Data.TdsClient.dll	28	37	75%
Mono.Data.dll	7	7	100%
Mono.GetOptions.dll	19	26	73%
Mono.Http.dll	8	13	61%
Mono.Posix.dll	138	178	77%
Mono.Security.Win32.dll	12	13	92%
Mono.Security.dll	169	218	77%
System.Data.dll	248	338	73%
System.Design.dll	104	112	92%
System.DirectoryServices.dll	22	25	88%
System.Drawing.Design.dll	11	14	78%
System.Drawing.dll	168	238	70%
System.EnterpriseServices.dll	112	114	98%
System.Management.dll	48	52	92%
System.Messaging.dll	47	54	87%
System.Runtime.Remoting.dll	57	70	81%
System.Runtime.Serialization.Formatters.Soap.dll	8	12	66%
System.Security.dll	34	42	80%
System.ServiceProcess.dll	17	21	80%
System.Web.Services.dll	177	211	83%
System.Web.dll	377	549	68%
System.Xml.dll	560	687	81%
Summary	4547	5881	77%

Reason	Count	Percentage
Unimplemented instruction	710	53%
Unimplemented built-in method	260	20%
Unimplemented construct	193	14%
Translator fault	171	13%

Table 4.2: Reasons for translation failures

4.2 Comparing SRM Modules

As described in the previous chapter, three stack to register mapping modules were implemented.

- Stack, which used an array to emulate the .Net stack
- Register, which mapped each stack location to a register
- OptRegister, which attempts to not generate some of the move instructions that Register does

This section compares these three SRM modules from various angles.

4.2.1 Generated Code Quality

A very compact example clearly demonstrates the differences between the code generated by the various stack to register mapping modules. Consider the following method, implemented in C#.

```
public int add(int x, int y)
{
    return x + y;
}
```

Compiling this code and then disassembling the resultant .Net module shows the following .Net Intermediate Language code.

```
.method public hidebysig
       instance default int32 'add' (int32 x, int32 y)  cil managed
{
    // Method begins at RVA 0x20f4
    // Code size 4 (0x4)
    .maxstack 8
```

```
    IL_0000:  ldarg.1
    IL_0001:  ldarg.2
    IL_0002:  add
    IL_0003:  ret
} // end of method Test::instance default int32 'add' (int32 x, int32 y)
```

The following code was produced by the Stack SRM module. Notice that it uses a `ResizablePMCArray` to emulate the stack, meaning that a PMC must be created at every entry to the method and also every time an integer or floating point number is pushed onto or popped from the stack. In the worst case (the add instruction), a single .Net instruction becomes four Parrot instructions. This may be code that runs on a register machine, but it is hardly register machine code.

```
.sub "add" :method :multi("Testing.Test", int, int)
      .param int arg1
      .param int arg2
      .local pmc arg0
      arg0 = self
      .local pmc s
      s = new ResizablePMCArray
      pushmark 0
LAB0: push s, arg1
LAB1: push s, arg2
LAB2: $I0 = pop s
      $I1 = pop s
      $I2 = $I0 + $I1
      push s, $I2
LAB3: $I0 = pop s
      popmark 0
      .return($I0)
.end
```

The Register SRM module (generated code below) addresses these shortcomings. Since it only uses registers, the problem of instantiating a lot of PMCs has disappeared. Also, there is now just one Parrot instruction per .Net instruction.

```
.sub "add" :method :multi("Testing.Test", int, int)
      .param int arg1
      .param int arg2
```

```
      .local pmc arg0
      arg0 = self
      pushmark 0
LAB0: $I0 = arg1
LAB1: $I1 = arg2
LAB2: $I0 = $I1 + $I0
LAB3: popmark 0
      .return($I0)
.end
```

While the Register SRM is a vast improvement, it produces a number of redundant move instructions. OptRegister attempts to remove some of these, and in cases such as the above one produces a notable improvement, shown below. Here, three .Net instructions have become a single instruction.

```
.sub "add" :method :multi("Testing.Test", int, int)
      .param int arg1
      .param int arg2
      .local pmc arg0
      arg0 = self
      pushmark 0
LAB0:
LAB1:
LAB2: $I0 = arg2 + arg1
LAB3: popmark 0
      .return($I0)
.end
```

While a good optimizer would be able to perform copy prorogation to eliminate the redundant moves, picking out some of the easy cases at translation time can only help the optimizer by providing it with a smaller problem.

4.2.2 Generated Code Performance

To compare the performance of the code generated by different SRM modules, I took an implementation of the Mandelbrot program in C# from The Computer Languages Shootout[1] and modified it to create a library with a single method that summed all the values that would be used to produce a visual Mandelbrot

[1] http://shootout.alioth.debian.org/

SRM	t_1	t_2	t_3	t_4	t_5	$t_{average}$
Stack	315.4	316.1	316.6	316.4	315.2	315.9
Register	21.30	21.25	21.31	21.28	21.28	21.28
OptRegister	12.02	12.03	12.00	12.02	12.02	12.02

Table 4.3: Mandelbrot performance by SRM

SRM	t_1	t_2	t_3	t_4	t_5	$t_{average}$
Stack	267.5	267.4	267.1	267.3	267.1	267.3
Register	228.9	229.4	229.9	228.8	228.6	229.1
OptRegister	220.0	220.0	219.9	219.8	220.0	219.9

Table 4.4: Translation time by SRM

output, computing a kind of checksum. The modified C# source code can be found in Appendix C.

This benchmark does intensive numerical computation, which means that the "stack" is used heavily. This emphasises the differences between SRM modules, and it would be fair to claim that it over-emphasizes them and that real world programs tend not to be doing such intensive numerical computation. The benchmark was translated under each of the SRM modules and run five times. The durations of these five runs were measured using a Perl script and the `Time::HiRes` module. The results are contained in table 4.3.

The performance of the code generated by the Stack SRM is significantly worse than the SRM modules that completely remove the need for a stack. This is because every push and pop on the array emulating the stack causes a v-table call and because every push or pop is causing a box or unbox operation to be performed. This boxing and unboxing creates a great number of very short-lived PMCs that must be garbage collected, causing high GC overhead.

The difference made by the OptRegister SRM over the Register SRM is quite notable and shows that not generating redundant moves is a big win in a benchmark like this (in fact, a much bigger win that I was expecting).

4.2.3 Translation Time

To compare the translation times, I wrote a script that timed translating the .Net class library five times under each SRM. The translation time was the time to generate the PIR and compile it to Parrot bytecode. The results are in table 4.4.

These results are surprising at first glance, since the more elaborate the SRM scheme the quicker the translation is performed. The reasoning behind this is

VM	t_1	t_2	t_3	t_4	t_5	$t_{average}$
Mono	2.172	2.140	2.141	2.125	2.156	2.147
Parrot	12.02	12.03	12.00	12.02	12.02	12.02

Table 4.5: Mandelbrot performance by VM

that the more elaborate SRM schemes produce less code, meaning that the PIR compiler has less code to compile and a smaller register allocation problem to solve. The PIR to PBC compilation phase is far more significant in determining the total translation time than the .Net to PIR step, so reducing the amount of time the second step takes by doing more work earlier on leads to shorter overall translation times.

These times fall well within the performance criteria laid out in the preparation, where it was stated that a translation time of around 2 minutes was acceptable for translating a file such as the 1.85 MB mscorlib.dll; even using the Stack SRM this only takes just over one minute.

4.3 Comparing Performance With A .Net VM

The Mandelbrot benchmark used to compare SRM modules produces the timings in table 4.5 when run under the Mono implementation of .Net. The best Parrot result is also repeated for easy comparison.

Mono is very clearly faster by a factor of around six, but considering that an optimized build of Mono with a working JIT compiler was compared to an unoptimized non-JITing Parrot (due to a bug in the Parrot JIT compiler), the performance of the translation is quite acceptable.

Again, this program does not present a typical workload and a great deal of further benchmarking would be needed to give a better indication of how well the translated code measures up to the original code running on a .Net VM.

4.4 Software Engineering Evaluation

An evaluation of the project from a software engineering perspective can be found in appendix D.

Chapter 5

Conclusion

Love virtual machines does he,
Shared libraries make his day.
And libraries for VM B,
Now work on VM A.

5.1 Bytecode Translation Works

The project met all of the requirements laid out in the project proposal and can be considered a resounding success, verifying the claim in the introduction that bytecode translation is a good solution to virtual machine interoperability problems.

In addition to being able to translate all of the planned constructs and instructions, some of which require non-trivial transformations, I have been able to implement a number of extensions and given the translator a more "real world" test by attempting to translate the .Net class library. For the amount of time that has been put in to the project, being able to translate over 90% of the .Net instruction set and three quarters of the classes in the class library indicates that the translator could be made suitable for production use in a reasonable time frame.

The regression testing suite demonstrates that the code that is being produced is semantically equivalent to the input .Net programs, providing evidence for the correctness of the translator. Furthermore, under the best stack to register mapping algorithm that I developed, good quality register machine code is produced.

5.2 Code Less, But Smarter

During the design phase of the project I did wonder whether creating a declarative mini-language would turn out to be overkill. My conclusion is that this project would have been far less successful had I hand-coded a translator. Taking what seemed like quite a long time early on to produce a development tool paid off very well further down the road, delivering a greatly more manageable and smaller code base.

To quantify this, the generated instruction translator is a file of over 22,000 lines of PIR when using the optimising stack to register mapper. By comparison, the file of translation rules is just over 3,000 lines and highly maintainable, an SRM module is only a few hundred lines and the build tool comes to under 1,400 lines of code including a large chunk of PIR that implements translation of exception handlers.

5.3 Future Directions

The source code, documentation and regression tests that have been produced in this project are going to be contributed to the Parrot community. I intend to continue development of the translator, and others will be more than welcome to join the fun. I have also submitted a proposal for a talk about the translator at the European Perl Conference 2006.

While over 90% of .Net instructions and 77% of the class library can now be translated, I think that reaching the point where the translator is ready for production use will take about as much work as it took to get this far. The approach to perfection - where the translator always produces semantically equivalent code - is probably asymptotic, and this ignores the fact that both .Net and Parrot are moving targets. The latest version of the .Net CLR supports parametric polymorphism, which will likely take significant work to translate. Another direction might be to take the translation framework and modify it to translate JVM bytecode, since a lot of the problems that would need to be solved are the same.

Whatever the future holds, I look forward to continuing to play with virtual machines and, so long as there is a need for it, bytecode translation. Interest in VMs continues to increase, and I hope this work will be a contribution to understanding just one of the many challenges that the field presents.

Bibliography

[1] John C. Gyllenhaal Cheng-Hsueh A. Hsieh and Wen mei W. Hwu. Java bytecode to native code translation: The caffeine prototype and preliminary results. In *29th Annual Internation Symposium on Microarchitecture*, 1996.

[2] Cristina Cifuentes and Mike Van Emmerick. Uqbt: Adaptable binary translation at low cost.

[3] B. Davis, A. Beatty, K. Casey, D. Gregg, and J. Waldron. The case for virtual register machines, 2002.

[4] EMCA International. *EMCA 335 CLI Specification - Virtual Machine.*

[5] Jonathan Worthington. Parrot: VM design gone crackers? Cambridge Programming Research Group, 2006.

[6] Jonathan Worthington. Parrot: What, where and why? London Perl Workshop, 2005.

[7] Mono Project. *Embedding Mono.*

Appendix A

Sample Regression Testing Script

This appendix contains the source code for the regression test script `t/math.t`.

```
#!perl -w

use Test::More;
use DotNetTesting;
use strict;

use Test::More tests => 7;

## Testing class for this file.
die unless compile_cs("t.dll", <<'CSHARP');
namespace Testing
{
    public class Test
    {
        public int add(int x, int y)
        {
            return x + y;
        }

public int sub(int x, int y)
        {
            return x - y;
        }

public int mul(int x, int y)
```

```
        {
            return x * y;
        }

public int div(int x, int y)
        {
            return x / y;
        }

public int rem(int x, int y)
        {
            return x % y;
        }

public int neg(int x)
        {
            return -x;
        }
    }
}
CSHARP

## Attempt to translate.
ok(translate("t.dll", "t.pbc"), 'translate');

## Tests.
is (run_pir(<<'PIR'), <<'OUTPUT', 'add');
.sub main
.local pmc obj
load_bytecode "t.pbc"
obj = new "Testing.Test"
$I0 = obj.add(500,72)
print $I0
print "\n"
$I0 = obj.add(500,-72)
print $I0
print "\n"
.end
PIR
```

```
572
428
OUTPUT

is (run_pir(<<'PIR'), <<'OUTPUT', 'sub');
.sub main
.local pmc obj
load_bytecode "t.pbc"
obj = new "Testing.Test"
$I0 = obj."sub"(500,72)
print $I0
print "\n"
$I0 = obj."sub"(500,-72)
print $I0
print "\n"
.end
PIR
428
572
OUTPUT

is (run_pir(<<'PIR'), <<'OUTPUT', 'mul');
.sub main
.local pmc obj
load_bytecode "t.pbc"
obj = new "Testing.Test"
$I0 = obj.mul(50,7)
print $I0
print "\n"
$I0 = obj.mul(-7,-6)
print $I0
print "\n"
.end
PIR
350
42
OUTPUT

is (run_pir(<<'PIR'), <<'OUTPUT', 'div');
```

```
.sub main
.local pmc obj
load_bytecode "t.pbc"
obj = new "Testing.Test"
$I0 = obj.div(12,3)
print $I0
print "\n"
$I0 = obj.div(15,14)
print $I0
print "\n"
$I0 = obj.div(-121,11)
print $I0
print "\n"
.end
PIR
4
1
-11
OUTPUT

is (run_pir(<<'PIR'), <<'OUTPUT', 'rem');
.sub main
.local pmc obj
load_bytecode "t.pbc"
obj = new "Testing.Test"
$I0 = obj.rem(13,3)
print $I0
print "\n"
$I0 = obj.rem(-15,13)
print $I0
print "\n"
.end
PIR
1
-2
OUTPUT

is (run_pir(<<'PIR'), <<'OUTPUT', 'neg');
.sub main
```

```
.local pmc obj
load_bytecode "t.pbc"
obj = new "Testing.Test"
$I0 = obj.neg(100)
print $I0
print "\n"
$I0 = obj.neg(-15)
print $I0
print "\n"
.end
PIR
-100
15
OUTPUT
```

Appendix B

Recursive Calling Regression Test

This appendix contains the source code for the regression test script `t/recursion.t`.

```
#!perl -w

use Test::More;
use DotNetTesting;
use strict;

use Test::More tests => 2;

## Testing class for this file.
die unless compile_cs("t.dll", <<'CSHARP');
namespace Testing
{
    public class Test
    {
        public int factorial(int x)
        {
            if (x <= 1)
                return 1;
            else
                return x * factorial(x - 1);
        }
    }
```

```
}
CSHARP

## Attempt to translate.
ok(translate("t.dll", "t.pbc"), 'translate');

## Tests.
is (run_pir(<<'PIR'), <<'OUTPUT', 'factorial');
.sub main
.local pmc obj
load_bytecode "t.pbc"
obj = new "Testing.Test"
$I0 = obj.factorial(0)
    print $I0
    print "\n"
    $I0 = obj.factorial(1)
    print $I0
    print "\n"
    $I0 = obj.factorial(2)
    print $I0
    print "\n"
    $I0 = obj.factorial(10)
    print $I0
    print "\n"
.end
PIR
1
1
2
3628800
OUTPUT
```

Appendix C

SRM Comparison Benchmark

This is the C# class that was compiled down to .Net bytecode and translated using each SRM module to compare their performance. It was based upon code from The Computer Languages Shootout[1].

```
namespace Benchmark {
public class Mandelbrot {
   public int MandelChecksum(int width) {
      int checkSum = 0;
      int height = width, i, m = 50, bits = 0, bitnum = 0;
      bool isOverLimit = false;
      double Zr = 0.0, Zi  = 0.0, Cr, Ci, Tr, Ti, limit2 = 4.0;

      for(int y = 0; y < height; y++) {
         for(int x = 0; x < width; x++){

            Zr = 0.0; Zi = 0.0;
            Cr = 2.0*x / width - 1.5;
            Ci = 2.0*y / height - 1.0;

            i = 0;
            do {
               Tr = Zr*Zr - Zi*Zi + Cr;
               Ti = 2.0*Zr*Zi + Ci;
               Zr = Tr; Zi = Ti;
               isOverLimit = Zr*Zr + Zi*Zi > limit2;
            } while (!isOverLimit && (++i < m));
```

[1] http://shootout.alioth.debian.org/

```
            bits = bits << 1;
            if (!isOverLimit) bits++;
            bitnum++;

            if (x == width - 1) {
               bits = bits << (8 - bitnum);
               bitnum = 8;
            }

            if (bitnum == 8){
               checkSum += (byte)bits;
               bits = 0; bitnum = 0;
            }
         }
      }
      return checkSum;
   }
}
}
```

Appendix D

Software Engineering

D.1 Planning Good Software Engineering

I think the hardest thing with regards to good software engineering in an individual project can be making yourself actually do the things that deep down you know make sense in the long run, but are not particularly interesting to do at the time.

D.1.1 Write The Documentation

The end result of this project was to be a fairly large system that would be able to translate somewhere between 150 and 200 .Net instructions as well as a range of other constructs described in the metadata. That is more than I was likely to be able to keep in my head at once, and therefore ensuring I documented how things worked as I implemented them was essential for me as well as anyone who works on the translator in the future. Of course, the higher level design needed to be documented up front too.

Parrot is documented using Perl's POD (Plain Old Documentation) format - a very lightweight documentation format that can be transformed into a wide range of other formats including plain text, HTML and Latex. I chose to use this for the translator because it was familiar (to me and to any future developers who have worked on Parrot), took next to no effort to work with and was easy to transform to Latex if I wanted to include it in the final write-up.

D.1.2 Regression Testing

Automated regression testing is particularly appropriate for projects such as this one. A number of input .Net libraries can be provided to the translator and

translated into Parrot libraries. For each translated library a number of tests can then be written to ensure that the results produced by the translated library are as expected. Since this is an automated process it becomes simple to run the tests on a regular basis to check new changes have not broken things that worked previously.

I chose to use a Perl testing framework, including the modules `Test::More` and `Test::Harness`, to build my regression test suite. They make writing tests a quick job by factoring out all of the testing related machinery so you just have the code and the expected output in the test script. They also provide provide result collation, producing a summary of any failing tests, and are the same modules used for testing Parrot and many other Parrot compilers. See Appendix A for a sample test script.

Just as documentation was to be an on-going process, testing was to take place throughout the implementation rather than as a separate stage at the end. Essentially, I planned to follow the Test Driven Development paradigm.

D.1.3 Backups

Since it's all to easy to forget to make backups, I automated the process. This was achieved by a simple Perl script scheduled to run once a day. It zipped up the repository (that is, the source code along with all of the version information), named the ZIP file using the current date and then FTP'd it to the Pelican server, operated by the Cambridge University Computing Service. This meant that I could look back as far as needed rather than just having the previous day's backup to look at.

D.1.4 Version Control

Even when there is just a single developer working on a code base, version control can be of great value. I used Subversion[1] from the start of the project, keeping all code and documentation under version control. If you're not familiar with Subversion, it is basically CVS but done right; there is a global version number rather than a per-file version number, so you can easily revert a particular set of related changes that spanned multiple files. Additionally, provided you write good commit messages and check in regularly, the version control system essentially keeps a project log for you.

[1] http://subversion.tigris.org/

D.1.5 Tools

I chose to use a makefile to manage the build process. They are portable and, if written correctly, mean that you only re-build the things that need to be re-built after a change. The entire build process at the end of my project took close to a minute on my development machine; just re-building what was needed on the other hand often only took a few seconds.

I used Microsoft's Visual Studio IDE heavily while coding. I chose this over other Integrated Development Environments or just a simple programmer's text editor mostly because of its C debugger. This would not only be useful when debugging the parts of my own project that were written in C, but also for those occasions when I needed to dig into the Parrot core itself.

D.2 Evaluating Software Engineering

D.2.1 Implementation

Separating out the concerns of instruction translation, stack to register mapping and type state tracking led to a highly maintainable code base. Pulling these together with a translator generation script led to a high performance translator, which coupled with a meta-data translator implemented in C gave overall good performance. Writing the C was painless; the most common mistake was forgetting to inform the garbage collector of live PMCs, but this was easily detected and fixed. The parts of the metadata translator that were implemented in PIR were quite tedious to produce, but thankfully this did not amount to a great deal of code. The choice of Perl for the translator generator worked out well.

Declarative instruction translation provided all the benefits that I had hoped for. I sometimes wished for a slightly more powerful language, especially for cases where an "instruction" statement was only just not powerful enough and I had to drop to supplying a PIR implementation of the code to emit the translation. Branch instructions are one example of this.

The SRM interface as designed right at the start of the project proved to be sufficient for all of the SRMs that I had planned to implement. There is scope for further improvements to the optimising SRM that there was not time to implement, and they too will fit within the current framework.

Overall, the implementation phase went as well as I had hoped and there was time to dig into some extension tasks too.

D.2.2 Documentation

I actually wrote the documentation as I went along. As predicted, I did find myself referring back to it over time as I forgot how things worked or were supposed to work. Trying to put an idea into words before putting it into code also provided another chance to think it through, so the documentation process often helped clear up any minor issues with the design.

By the end of the project I had over 11,000 words of documentation, which would have come in handy had I forgotten to write a dissertation. However, it would also have made a really boring read.

D.2.3 Regression Testing

I aimed to take a Test Driven Development style approach to this project from the start. I followed this well - the first time the vast majority of new code was tested, it was through a test in the regression testing suite.

Regression testing did catch a number of bugs introduced by adding a new feature that somehow broke existing code, and without the test suite they may have slipped through the net and not have been noticed until some time later. Seeing what tests fail and how they fail can also provide valuable insight into where the problem might lie. Having a good test suite was also extremely valuable when implementing new SRM modules.

Forcing myself to be disciplined about writing tests from the start was one of the best software engineering decisions I made in this project. To my surprise I found that a couple of months into the implementation, as I started to see the benefits that having the test suite was bringing, writing the tests stopped feeling like a chore and instead just became a natural part of the development process.

Appendix E

Managed Pointers

A managed pointer holds the address of a local variable or parameter on the stack, a field from an object or an element in an array. These can be used with a number of instructions that load and store data indirectly through pointer. The "managed" part means that limitations are placed upon the pointer. Namely, in verifiable code the location the pointer references can only be set by a limited number of instructions and can not be modified by user code. This means that the pointer will always point to a valid location so the safety of the VM can not be compromised.

E.1 Considering Possible Parrot Safety Problems

Parrot does not directly support and lacked a straightforward way to implement managed pointers. One issue is that there is no official and documented way to obtain references to, or the addresses of, registers, fields of objects or array elements. Another is that post-translation, there is nothing to distinguish a register holding a reference, so verification as used in .NET does not help with regard to protecting the Parrot VM at runtime. Further complications arise with regard to references to arrays and objects, as even if undocumented knowledge of internals is used, the data a PMC holds may move at runtime, meaning the pointer may be silently invalidated at some point. Registers are somewhat easier to get the address of but also present a security issue - if the managed pointer exists beyond lifetime of the register frame, that memory location can be modified at a later time.

E.2 A Managed Pointer PMC

I decided the best way forward was to implement a custom PMC to represent a managed pointer. Beneath the PMC is a structure containing details of the pointer (conceptually, the location it points to). However, no PMC methods exist that allow this location to be modified from the outside. Instead, managed pointers can only be created through a number of special instructions, implemented as dynamic ops. This has the same net result as a .Net VM that verifies code to ensure that the pointer's address is not modified.

E.3 Managed Pointers To Array Elements

As the index of the array element being referenced is known at the time the pointer created, the pointer can be emulated by storing the index and a pointer to the array PMC. The array PMC's v-table methods are then used to get and set the value, meaning that PMC encapsulation is not broken and pointers into other language's array PMCs will work. The array PMC needs to be marked live when the garbage collector visits the managed pointer PMC.

E.4 Managed pointers to fields

Similar to the array case, the name of the field along with a pointer to the object PMC can be stored and used to implement indirect access to the field through the "getattribute" and "setattribute" v-table methods. Again, the object PMC needs to be marked live for GC purposes.

E.5 Managed pointers to registers

This covers managed pointers to local variables and parameters, which in translated programs are both stored in registers. A dynamic op can be implemented for each register type that creates a managed pointer PMC that points to the current register frame and contains the type of the register and the register number. These can then be used to modify the register indirectly. However, this leaves a big security problem, since the managed pointer may continue to exist beyond the register frame.

To solve this problem, and after discussion with other Parrot developers, I patched Parrot to have a lightweight callback mechanism such that a function can register itself to be called when a particular register frame ceases to exist.

This in turn causes the managed pointer PMC to invalidate the pointer to the register frame, and attempting to use the managed pointer PMC from that point onwards would throw an exception.

A further problem is that the local variables, when placed in registers, became eligible for register allocation and therefore two local variables may be allocated the same register when their live ranges do not overlap. This is a problem if a managed pointer is taken to a register, since when the pointer is followed the register may not contain the local variable that it did when the pointer was taken. To work around this, I modified Parrot to allow a `.local` declaration to have a `:unique_reg` modifier, which forces the register allocator to give it a register of its own.

Appendix F

Project Proposal

J. R. Worthington
Emmanuel College
jrw64

Computer Science Tripos Part II Project Proposal

Virtual Machine Bytecode Translation: From The .NET CLI To Parrot

18 October 2005

Project Originator: J. R. Worthington

Project Supervisor: Dr. T. Griffin

Signature:

Director of Studies: Dr. N. Dodgson

Signature:

Overseers: Prof. G. Winskell, Dr P. Lio

Special Resources Required

The use use of my own personal computer (AMD Athlon(tm) XP 2800+, 1.0 GB RAM, 80 GB HDD, Windows XP, Linux). I will use the Subversion version control system, which may not be available on the PWF.

Introduction

Writing high level language compilers that target a virtual machine rather than a range of hardware and operating systems is of increasing popularity. Virtual machines for high level languages usually consist of a software CPU with an instruction set, a memory management system and an API (Application Programming Interface) for performing a range of tasks that may include, for example, I/O and threading. An implementation of the virtual machine maps the instruction set of the software CPU to instructions understood by the underlying hardware and the virtual machine's API to the API provided by the operating system.

Virtual machines make cross-platform deployment much simpler: once the virtual machine has been ported to a platform, all programs that run on the virtual machine can run on that platform. Compilers only need one back end. Furthermore, the virtual machine can make a compiler's task simpler by providing support for common high level language constructs such as types, objects and exceptions. This common representation for such constructs also opens the door to high level language interoperability; it could be possible to write a class in language A, inherit and extend it in language B and instantiate the derived class in language C.

In the last decade computing power has become sufficient to offset the translation costs and researchers have produced a wide range of techniques for implementing virtual machines efficiently, making widespread use of virtual machines feasible. Sun Microsystems introduced the Java Virtual Machine (JVM) in 1995, Microsoft introduced its .NET platform at the start of the millennium and around same time work started on Parrot, an open source virtual machine project initiated by the Perl community.

Somewhat ironically, the issue of having a range of target platforms that was to be solved by implementing a virtual machine has come back in a new form: there are now a range of target virtual machines! This is not too surprising, not only because a free market works that way but also because there are good technical grounds for why one virtual machine would not be a good fit all languages. For example, a virtual machine that only needs to provide for static languages can make a lot of assumptions and optimizations that a virtual machine that was to be the target for dynamic languages could not make. Here dynamic languages refers to languages that may, for example, need their parsers available or allow new types to be created and existing types to have their structure and behaviour modified at runtime.

This project aims to investigate a way of achieving interoperability between virtual machines. Specifically, it will attempt to translate the input that would

be expected by one virtual machine into the input expected by another. The "machine code" that virtual machines execute is usually named bytecode, thus the project title. The two virtual machines that have been selected for this project are the .NET CLI and Parrot, with the aim of translating .NET CLI bytecode to Parrot bytecode.

The .NET Common Language Infrastructure

The .NET CLI was specified and first implemented by Microsoft and is an open ECMA standard. An open source implementation exists and is named Mono, the Spanish word for monkey. Mostly suited to static languages, its notable features include:

- Stack based execution model
- Garbage collection
- A range of built in value types
- Extensive support for OOP constructs including classes, single inheritance, interfaces and objects
- Built-in support for arrays, exceptions, delegates, type checking, strings, unicode, operator overloading and tail calls
- PInvoke (Platform Invoke) for dealing with platform specific libraries
- Declarative security model

Parrot

The Parrot project initially started out as the Perl 6 internals project, but shortly afterwards became a project to build a virtual machine to support a wide range of languages. The name "Parrot" came from an April fool's day joke news story that had Larry Wall and Guido van Rossum agreeing to merge the Perl and Python languages to produce a language named Parrot. With Perl and Python as just two of the languages that Parrot is to support, it needs to provide a range of features used by dynamic programming languages. Notable Parrot features include:

- Register machine based execution model with 4 register types (Integer, Number, String and PMC)

- PMC types allow for language specific behaviour on a wide range of operations through a v-table mechanism
- Garbage collection
- OOP support including classes, multiple inheritance with runtime-changeable inheritance hierarchy and objects
- Built-in support for arrays, exceptions, strings with various types of encoding, subroutines, namespaces, closures, co-routines, continuations, tail calls, lexical variables
- MMD (Multi Method Dispatch) - like method overloading but more dynamic, as new methods could appear at any time
- Dynamically loadable PMCs and instructions (the virtual machine's instruction set can be extended at runtime by loading additional instructions; yes, Parrot really is that insanely cool!)
- Interface for calling back into compilers
- NCI (Native Calling Interface) for dealing with platform specific libraries

Work that has to be done

The project has three tasks that must be completed in order before any others followed by a wide range of tasks that build upon these that have no significant dependencies. The first three tasks can be seen as building a framework for the translator. The remaining tasks then use that framework, improving it in places, to build a useful translator.

Bytecode and meta-data extraction

The first task is to implement a method of taking a .NET EXE or DLL file and extracting from it the bytecode and meta-data. The meta-data is used to describe types (classes), various signatures for locations and methods, protected regions and their related exception handlers and other details that are not encoded in the instruction stream. The bytecode is the stream of instructions that define the implementation of methods.

The plan for this is to write a number of Parrot PMCs that will be used to represent the file. This will allow full introspection of the .NET meta-data and

access to the bytecode from Parrot programs. PMCs are implemented in C. It may be possible to use code from the Mono project to parse the file, depending on whether their implementation is in C and is suitable. If not, it does not appear to be a significant undertaking to implement something from scratch.

Class hierarchy translator

In the .NET CLI the details of classes, their properties and their methods is stored in meta-data. In Parrot a series of instructions are used to set up classes. To allow the project to move forward a translator that can take the .NET meta-data and emit Parrot Intermediate Representation (PIR) code that define a class, its properties and its methods will be required. Initially, it need not support inheritance, visibility, interfaces and type checking; these can be added later.

This translator will make use of the PMCs developed as part of the first task and will also need to use the Parrot Intermediate Representation (PIR) compiler, which is invokable from PIR. Given these requirements and that it is not a large body of code, this section of the project can be written in PIR.

Note that PIR is going to be generated instead of Parrot bytecode. This is because generating Parrot bytecode directly would basically involve duplicating the Parrot assembler, which would vastly slow progress on the project for a possible small performance gain at translation time (and no gain at execution time, which is what really matters; the results of the translation can be cached).

The bytecode translator generator

This third part is the foundation for actually translating the bytecode. The bytecode translator would roughly look like a massive switch block, but with register allocation code spliced in everywhere. The best way to write a large, monolithic chunk of code is not to write it, but rather to generate it. The translator generator will be written in a high level language. It will take a rules file that maps .NET instructions to Parrot instructions and a module that knows how to generate the stack to register mapping PIR and emit a translator in PIR. This means that the various concerns (instruction mapping and register allocation) are well separated. The stack to register allocation strategy need not be a great one at first, nor does the rules file need to contain a mapping for every .NET instruction; these are to add in the later tasks.

Tasks required for a minimal translator

To build a very minimal translator that shows that .NET to Parrot bytecode translation is possible, though is of little practical use, the following things need to work on Parrot.

- Set up regression test harness
- Native integer and 32-bit integer types
- Method parameter access instructions and the return instruction
- Basic arithmetic and logical instructions
- A scheme for translation and fix-up of jump and conditional branch instructions
- Local variable related instructions

Tasks required for a successful translator

For the project to be considered a success, all of the above things need to work and in addition to those, the following list.

- Method calling (from within .NET code)
- Complete System.Object implementation
- Remaining built-in types
- Boxing and unboxing of types
- Arrays
- Strings
- Static methods
- Overloaded methods
- Object instantiation, initializers, constructors and finalizers
- Class inheritance and interfaces
- Floating point types and instructions

- Exceptions
- Type checking
- Visibility modifiers (private, family, assembly, public)
- Better register allocation
- Lots and lots of regression tests (Mono tests can be used)

Extension tasks

If the project finishes ahead of schedule, some features from the following list can be implemented.

- Delegates (the OOP equivalent to function pointers)
- PInvoke (calling into native libraries); will also need to investigate pin and unpin instructions
- Reflection
- Runtime infrastructure library
- Variable argument methods
- Nested types
- Declarative security model
- Respect "volatile"
- Operator overloading
- Tail calls and method jumps
- Typed references
- Investigate what remains to be done to make any parts of the .NET class library that currently can not be translated work - there should not be much left

Difficulties to overcome

The biggest difficulty will be ensuring the translator captures the full semantics of the .NET CLI. A number of .NET instructions will not map directly to Parrot ones, or the Parrot instruction it would appear to map to will have some slightly different semantics that need to be worked around. Also, Parrot is not so strict about typing as the .NET CLI is, so enforcing a .NET type regime will require a little effort.

Starting Point

At the time of proposing the project I have a good knowledge of most of the terminology and concepts used in the virtual machine field, although I have not greatly considered or researched the implementation of many of them in any depth. I have carried out a range of development work on Parrot, including a number of fixes with regard to building and using Parrot on the Windows platforms and implementing a Parrot bytecode file linker. I am somewhat less familiar with the .NET platform, but have just finished reading the .NET CLI ECMA Specification, Partition I. I have a good deal of experience with the Perl programming language and am familiar with the regression testing tools provided by Perl. I am familiar with the C programming language.

Work Plan

24th October - 6th November

- Document the project plans in some more detail, work out any subtle dependencies
- Set up subversion, a project wiki, backups, etc.
- Start to develop .NET EXE and DLL reading PMCs
- Start to develop meta-data translator alongside the PMCs, as this will act as an ideal way to test them

7th November - 20th November

- Complete PMCs and first version of meta-data translator
- Design translation rules engine

- Decide on initial register allocation strategy
- Do most of the work to implement the translation rules to PIR translator engine and connect it to the meta-data translator.
- Write some of the rules for basic arithmetic instructions and the return instruction
- Get some regression tests running

21st November - 4th December

- Finish first version of the instruction translation engine
- Add more arithmetic and logic instructions

5th December - 18th December

- Generate labels for jumps and add rules for jumps and conditional branch instructions
- Parameter access and local variables
- Any missing arithmetic and logical instructions

19th December - 8th January

- Christmas and new year's break
- A chance to catch up if the schedule has slipped

9th January - 22nd January

- Method calling (from within .NET code)
- Complete System.Object implementation
- Remaining built-in types
- Boxing and unboxing of types

23rd January - 5th February

- Arrays
- Strings
- Floating point types and instructions

6th February - 19th February

- Static methods
- Overloaded methods
- Object instantiation, initializers, constructors and finalizers
- Class inheritance and interfaces

20th February - 5th March

- Exceptions
- Type checking
- Visibility modifiers (private, family, assembly, public)

6th March - 19th March

- Better register allocation
- Catch up time, plus any extensions that time allows.

This takes me to the Easter break, which can be used to produce the first draft of the dissertation in time for the start of Easter term.