A Pizza Compiler For .NET

Morten Sylvest Olsen

IMM-THESIS-2002-20

IMM

Trykt af IMM, DTU

Foreword

This report is the result of a masters project titled "A Pizza Compiler for .NET", with work being done from October 2001 through March 2002 under the supervision of Associate Professor Jørgen Steensgaard-Madsen at the section of Computer Science and Engineering (CSE), part of the department of Informatics and Mathematical Modelling (IMM) at the Technical University of Denmark (DTU).

I would like to thank my supervisor, Jørgen Steensgaard-Madsen, for his help in guiding me in the right direction. I would also like to thank my parents for support and encouragement.

April 2nd 2002

Morten Sylvest Olsen

Abstract

The notion of abstract virtual machines is introduced. Overviews of the Microsoft .NET Common Language Runtime, and the Pizza language, are given. The design and implementation of a new back-end for the Pizza compiler that emits code for the Microsoft .NET runtime is shown. Tests that compare code size and performance between the Java Virtual Machine and the .NET Common Language Runtime are performed. Some further possible work on the Pizza compiler is laid out, and the suitability of using the .NET runtime, as target for Pizza, is discussed.

Keywords

portability, virtual machines, Pizza, Java, JVM, .NET, Common Language Runtime, code generation, compiler bootstrap

Contents

List of Figures

List of Tables

Chapter 1

Preface

1.1 Executive summary

In this report I document the design and construction of a new back-end for the Pizza compiler. The new back-end generates code for the Microsoft .NET *Common Language Runtime* (CLR).

I start by introducing the notion of portability and abstract virtual machines. Since the .NET runtime is new, and probably unfamiliar to most readers, I have devoted a chapter to a short description. I then introduce the Pizza language extensions to Java.

Then the design of the new back-end is specified; this involves mapping the dynamic semantics of the Java Virtual Machine onto the Common Language Runtime. The actual implementation is described. The correctness of the compiler is established, and some tests are performed.

From the experience gained with the new back-end I compare the JVM with the CLR, and conclude that the CLR is well suited for supporting the Pizza (and by extension, the Java) language, even surpassing the JVM in some respects.

1.2 Prerequisites

In this report I presume knowledge of the Java language, and some knowledge of the underlying Java Virtual Machine. Although the Pizza compiler extends the Java language, these extensions should all be familiar from functional programming. Some knowledge of compiler construction will also be assumed. No prior knowledge of .NET is assumed.

1.3 Typographical conventions

Italic is used for class names.
`Verbatim` is used for virtual machine instructions and identifiers.
Boldface is used for Java keywords.

1.4 Terminology

Throughout this report I will refer to the modified Pizza compiler with the new back-end as *pizzacil*.

Unfortunately this report contains many acronyms, these are the most important.

CIL Common Intermediate Language.
CLR Common Language Runtime. The .NET virtual machine.
JVM Java Virtual Machine.
JLS Java Language Specification.
JIT Just-In-Time compiler.
IL Intermediate Language.

1.5 Organization

- Chapter 1 introduces portability, Pizza, .NET and abstract virtual machines.
- Chapter 2 gives an overview of the .NET CLR.

- Chapter 3 briefly explains the Pizza extensions to Java.
- Chapter 4 explains the design of the new back-end. This involves mapping Java/JVM operations onto the .NET CLR.
- Chapter 5 shows the structure of the implementation of the back-end.
- Chapter 6 shows how correctness of the back-end can be validated, and gives some test results.
- Chapter 7 summarizes the current status and possible future work.
- Chapter 8 contains the conclusion.

The report has the following appendices:

- Appendix A contains the project description.
- Appendix B includes a short manual on how to get, install and use the pizzacil compiler.
- Appendix C has a *very* condensed description and incomplete grammar of the syntax for CIL assembler, hopefully enough to understand this report.
- Appendix D lists an example of code generated by the new back-end.
- Appendix E is a list of bugs found in the Pizza compiler unrelated to the back-end.

Chapter 2

Introduction

This chapter contains a short description of Pizza and .NET. It also defines the notion of portability, and abstract virtual machines.

2.1 The purpose of the project

Portability of programs is an important issue. In recent years Java, and the Java Virtual Machine, has shown that cross-platform binary compatibility is feasible. While the Java language has changed over the years, the JVM has not. Presumably the .NET Common Language Runtime has been build on the experiences learned from the JVM, and should have improved on some areas of it.

The purpose of the project was to compare the properties of the Java Virtual Machine to the .NET CLR. A practical goal was to map the dynamic semantics of the Pizza language onto the CLR, and construct a new back-end for the Pizza compiler to make it possible to run Pizza programs unmodified and transparently on the .NET runtime.

2.2 The Pizza compiler

The Pizza compiler was written by Martin Odersky and Philip Wadler, as part of a project to research extensions to the Java language [OW97],

[ORW98].

It was then released as open source, and is now maintained by Nick Fortescue et al. as a SourceForge project. [1]

Pizza extends Java with concepts from functional programming languages, namely:

- Generics.
- Algebraic types.
- First-class functions.
- Tail recursion.

Pizza was chosen as the basis for this project because it is freely available and of high quality.

2.3 .NET Common Language Runtime

.NET is a recent product from Microsoft, that overlaps somewhat with the goals of Java. Probably for marketing reasons, many of their products has now been renamed to include the term ".NET", but in the scope of this project the essential parts are:

- A new language named C#. C# is a modern statically typed object-oriented language, that shares much syntax and semantics with Java. It also contains ideas from Visual Basic, C++ and Delphi. Like Java, C# does not support multiple inheritance, but unlike Java it allows the programmer to step out of the restricted environment and use unsafe operations like pointer arithmetic.
- The definition of a virtual machine architecture, the Common Language Runtime, or *CLR*.
- A comprehensive class library.

Parts of .NET has been submitted for standardization by ECMA. I will refer to the standards documentation throughout this report. A short introduction to the runtime itself can be found in [MG01].

[1] `http://pizzacompiler.sourceforge.net/`

2.4 Related work

Microsoft has released a .NET version of their J++ compiler, called *J#*. This is a Java compiler targeting the .NET Common Language Runtime. A beta version has been made available. I have used their re-implementation of the Java class library in my project.

2.5 Portability

With the release of the Java language, and the Java Virtual Machine in 1994, much attention was directed at the notion of using a *virtual machine* to achieve portability over a range of architectures and platforms. This idea was not new though, but stems back from the 60's, if not before.

2.5.1 Portability

One definition of portability is given in [Moo97]:

> A software unit is portable (exhibits portability) across a class of environments, to the degree that the cost to transport and adapt it to the new environment in the class, is less than the cost of redevelopment

Many factors can impede portability. Low-level hardware architecture differences are the basic differences between platforms:

- Instruction set.
- Basic word size.
- Endiannes.
- Alignment requirements for code and data.
- Memory model
- Data representation.

Then there are differences in operating system and other external interfaces.

Portability is usually something that is given as a desired quality of a software project, but rarely is much work done on specifying the requirements for it. *Source portability* depends on compilers existing on all the target platforms, and that external interfaces are compatible. In under-specified

languages, like C, achieving source portability is an art of deep magic. Operating system dependencies needs to be abstracted, and compiler differences needs to be taken into account, this usually leads to massive amounts of **#ifdefs** and incomprehensible code.

This report does not treat the area of source portability. Instead the focus is on *binary portability.*

2.6 Portability through virtual machines

The idea of using a virtual machine stems from the problem of writing portable compilers. If one desires to create compilers for a number of source languages, N, and want to emit code for a number of hardware architectures, M, then $N * M$ compilers needs to be created.

If instead the problem is re-factored into a frontend that parses the source language, and emits an intermediate language (IL), and a back-end that consumes the IL and generates code for a specific platform, the problem has been reduced, so that only $N + M$ compilers are needed.

This was first formulated in [SWT+58] where the idea of an *UNCOL (UNiversal Computer Oriented Language)* was presented. The purpose was to create a universal IL that could be used for every possible source language and every possible target architecture. This has been one of the "holy grails" of compiler technology since, but no IL has been able to gain universal acceptance. In theory any Turing-complete language would suffice, but the problem is that it should be easy and efficient to translate. It should be easy to translate to, from the source language, and it should be efficient to translate to the target architecture.

One of the first intermediate languages to be widely used was *P-Code*, which was invented as the IL for the Pascal-P compiler by Niclaus Wirth at ETH. It was then noted, that not only could it be used as input to a back-end, but an interpreter for the *abstract virtual machine* defined by the semantics of P-Code could be used to run the program.

It is easier to build a portable interpreter than a code generator, which is one of the reasons Pascal-P became popular over a range of platforms.

2.6.1 Virtual machine architectures

Many different intermediate languages and virtual machines has been defined through the years. Some are higher level, because the try to "bridge the semantic gap" between a very abstract source language and the real hardware. An example is the Warren Abstract Machine (WAM) used in logic programming systems. Others are more general abstractions of existing hardware architectures. An important feature of an IL is, that it should be placed at a level of abstraction where it is both easy to generate in the front-end, and where translation to optimized native code in the back-end is possible.

Register Transfer Lists

RTL, or Register Transfer Lists, is the IL used in the GNU Compiler Collection (GCC). During configuration of the compiler for a specific architecture, hardware specific instructions for the desired target architecture are included into the RTL, from a machine description table. When compiling, the IL nodes contain both symbolic information needed for optimization, and textual representations of the machine instructions to emit.

GCC only compiles a basic block at a time into RTL, a full representation of the source program never exists. This intermediate representation originally stems from the Very Portable Optimizer (VPO) in [DF80].

ANDF

Architecture Neutral Distribution Format (ANDF), is a largely failed attempt from the Open Software Foundation (OSF)[2] in 1989, to create a standard format usable for portable software distribution, between the multitude of Unix platforms that existed at the time. ANDF is basically a binary encoding of the abstract syntax tree of the source program. The idea was that when the software was installed, it would be compiled to native code. Interpretation was not considered at all. The ability to generate code from ANDF, that was as optimized as that generated by a target dependent optimizing C compiler, was a major design parameter.

[2]Now The Open Group

Today ANDF is only used very few places, one as the IL for the Ada compiler at DDCI [Bun95]. ANDF failed for several reasons. It was designed with the C language in mind, and one goal was to be able to install on everything from a plain 32-bit RISC CPU to something completely different like the iAPX-432, which does not use a linear memory model [DER95]. This means that *no* assumptions on the size of any structure, or the offsets of fields within it, can be made at compilation time, which makes code generation for ANDF complicated, because all references have to be made using "token pointers". Whether it makes sense to worry about support for very "weird" architectures still is also questionable.

Stack-based

Many intermediate languages have been based on a stack architecture. One that was already mentioned, is P-Code. But the most recent, and most commercially used one, is the Java Virtual Machine. The most important features that separate the JVM from earlier attempts are:

- An object-oriented memory model, including garbage collection.
- All references to objects, fields and methods are made symbolically. This means that portability can be achieved, without the problems that makes ANDF unwieldy.
- The IL contain enough type information, so that a program can be verified for memory safety before execution.
- The virtual machine strictly defines the environment separate from the underlying operating system. This includes thread support, memory management and I/O.

Tree-based

There is no reason why an IL should necessarily resemble real hardware architectures. Actually those linear representations are not optimal, because they destroy information about the program structure. *Slim Binaries* [KF99] is a IL based on a tree structure, used in the *Juice* language, that grew out of a dissertation on *Semantic Dictionary Encoding* and its use as an IL in an Oberon compiler [Fra94].

They argue that these *Slim Binaries* are both much more compact than Java Byte Code, and that better native code can be generated from them,

because control and data flow of the original source are preserved in the IL. One of the major problems of JIT compilers for the JVM is reconstruction of this information, to be able to perform many optimizations.

2.6.2 Summary

Stack architectures in hardware, with the notable exception of the Sun picoJava, has gone the way of the dinosaur, but as virtual machines they seem to have been successful.

Register-based architectures reign in hardware, but they are not well-suited for an IL:

- Usually they will fit badly with the target hardware. For example the three-address, register-architecture virtual machine MMIX [Knu] has 256 general purpose registers. Most real architectures have less, and even if the target architecture is a three-address architecture, like the PowerPC or MIPS, it would probably not be a one-to-one match. A very common architecture, the Intel x86 would be a very bad fit indeed. In most cases this would mean that a JIT compiler would need to recompute the register assignment. Also instruction scheduling and cache issues means that there is little or no benefit compared to starting from a stack-machine.
- A register IL is not well suited for interpretation. Whereas in hardware the register decoding comes "for free", in an interpreter this is costly [Ert96].

Stack architectures on the other hand have several nice properties:

- It is relatively easy to generate code for a stack machine from most imperative languages.
- Since argument addressing is implicit, efficient interpreters can be easily build.

Lately the emphasis has not been on interpretation. Interpreters execute at least an order of magnitude slower than compiled code. To achieve the portability of using virtual machines, without the overhead, much research has gone into *Just In Time (JIT)* compilers. Instead of interpretation, the IL is translated to native code on the architecture, at execution time. This is usually complicated by a need to support mixed-mode execution, where

some functions are executed natively, while others, in the same program, are interpreted.

For fast optimized code generation, stack based IL has some drawbacks. Much code- and data-flow information from the original source is lost. This makes the task of the JIT compiler much harder, or at least very different, from traditional optimizing compilers.

Both the JVM and the .NET CLR are stack-based virtual machines. In the next chapter I will describe the CLR in more detail.

Chapter 3

The .NET Common Language Runtime

This chapter gives an overview of the .NET Common Language Runtime, or CLR, necessary to understand the following chapters. Readers familiar with the .NET virtual machine should be able to skip this chapter.

3.1 Motivation

The CLR is touted more for its capabilities as an inter-language platform, than its use for portability. More weight has been placed at the ability to compile from more source languages, than the possibility to run on more platforms. Although there are many compilers that target the Java Virtual Machine, most of them do not integrate with each other at all [pro]. It is not possible to extend a class, defined in language A translated into Java Byte Code, in another language B.

One of the stated goals of the CLR is this ability. For example to write a class in C# and use it in a Visual Basic program, without having to marshal calls through a foreign function interface, or COM/Corba.

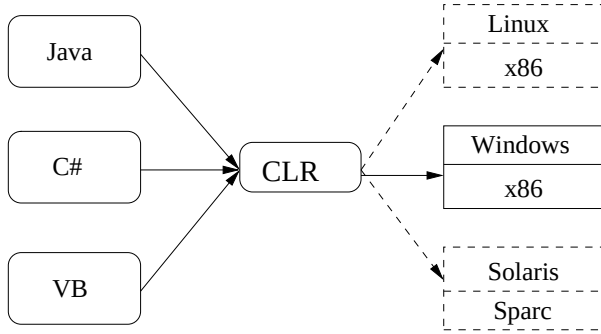

Figure 3.1: The Common Language Runtime focus

3.2 Overview

The architecture of the CLR is a stack machine with locals. In many ways it is similar to the JVM, but there are many differences as well.

To be able to support a larger class of source languages, the CLR supports many things that has no equivalents in the JVM, including pointer arithmetic. This makes it possible to translate, for example, ANSI C to the CLR without having to resort to inefficient hacks. A subset of the CLR can be statically verified to be type-safe, like the JVM.

The language of the CLR is known as the Common Intermediate Language or CIL. Figure 3.2 shows the layers of CIL. A program *can* be type-safe, even though it is not verifiable. The type-unsafe valid CIL can be as powerful as real native machine language. For mobile or Internet applications only the verifiable subset will usually be acceptable.

3.3 Types

When a constant is loaded on the stack, or an array element is referenced, the type of the value need to be specified. For each possible type, a different instruction is used. For example to load a one byte constant on the stack the `ldc.i1` instruction shall be used. Truncating conversion instructions exists to convert between the types.

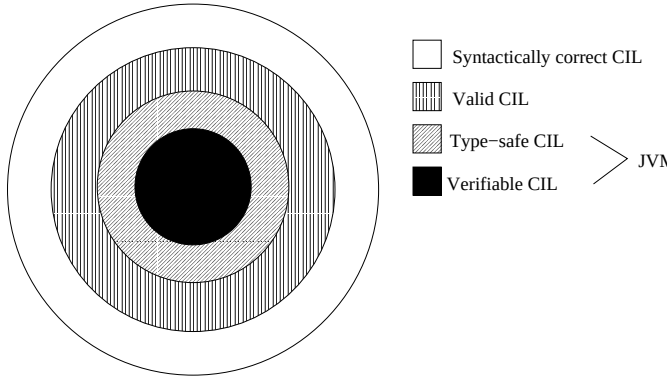

Figure 3.2: Layers of CIL

There are two levels of the basic types:

1. The types that the runtime operates on.
2. The types that can be specified in CIL

The types that are can be used when generating code are listed in table 3.1. The first column shows the name of the type, the second the mnemonic used in the instruction set to specify type. The last column is the corresponding type in pseudo-Java.[1]

As the table shows, some CIL types are just aliases. The boolean type is equal to int8 and char to uint16. The unsigned types are also in reality aliases to their signed counterparts. No distinction is made between stack locations, instead special instructions exists for unsigned arithmetic.[2]

The runtime actually only operates on these types:

- int32, int64
- Hardware dependent floating-point, F
- Hardware dependent integer, natural int
- Object reference

Smaller integer values are sign-extended to int32 when loaded on the stack.

[1] Because Java does not have unsigned types
[2] For the operations where it makes a difference!

Name	Opcode type T	Type name
int8	i1	byte
int16	i2	short
int32	i4	int
int64	i8	long
uint8	u1	“unsigned” byte
uint16	u2	“unsigned” short
uint32	u4	“unsigned” int
uint64	u8	“unsigned” long
natural int	i	Hardware dependent
object	ref	Reference
float32	r4	float
float64	r8	double
char	u2	char (Unicode)
boolean	i1	boolean

Table 3.1: CLR types

The CLR allows floating point calculations to be made with machine dependent precision. This is probably due to the fact that the Intel x86 uses a 80-bit internal representation. Strict IEEE754 compliance, where all calculations needs to be rounded to 64-bit precision, can only be achieved on the x86 by doing a register to memory transfer after each operation.

3.4 Execution environment

At method invocation a new activation record is allocated. It contains the following elements:

- The return handle, used to restore the callers activation record.
- A local variable array. A zero-based array of locals that can be referenced within the current method.
- An argument array. A zero-based array of incoming arguments. If the method is non-static, the first argument will be a self reference.
- An evaluation stack. This is used when performing computations.

Method definitions cannot be nested, and methods cannot reference local variables in other methods, so there is no static link in the activation record.

3.4.1 Return handle

The return address is saved outside the evaluation stack by the runtime. To return from a method the `ret` instruction is issued. If the return type of the method is non-void, a value will be popped from the evaluation stack.

3.4.2 Local variables

A method can have up to 65535 local variables. They are numbered from 0 to 65534. Instructions for accessing locals are:

ldloc(.s) ***indx*** Loads the local variable to the top of stack.
ldloc.*indx* Compact encoding for $0 <= indx < 4$.
stloc(.s) ***indx*** Stores to top of stack in local variable.
stloc.*indx* As for ldloc.

In the JVM each local variable slot can contain one value of 32-bit size. This means that values of larger sizes (eg. a double) takes up two slots.

In the CLR each value takes up only one slot. This is regardless of whether it is an int, long or double. It can even be structured data of arbitrary size, a value-type.

3.4.3 Incoming arguments

The arguments are placed separately from the local variables. This is unlike the JVM, which places incoming arguments in the first local variable slots. The argument array is abstract in the same way as the local variables. Instructions for manipulation of this array are:

ldarg ***indx*** Load argument number *indx* to the top of stack.
ldarg.*indx* Compact encoding for $0 <= indx < 4$.
starg ***indx*** Store top of stack in argument slot *indx*.

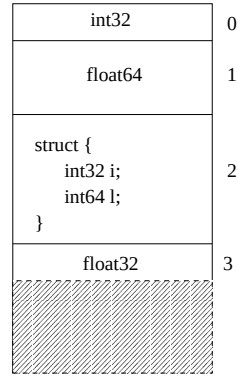

Figure 3.3: Local variables (and evaluation stack) in the CLR

3.4.4 Evaluation stack

Like the two arrays, argument and locals, the evaluation stack is also abstracted. This means that, regardless of type, a value takes up one stack element. When an element is loaded onto the stack, the instruction specifies the type of the element. Subsequent operations (for example arithmetic or logical instructions) does not have to specify their argument types. Of course the two operands to for example an `add` instruction needs to be of the *same* type. The verifier will be able to check this.

When invoking a method the arguments are placed on the evaluation stack. If the method called is not static, the first argument shall be an object reference to an object of the correct type. The rest of the arguments, if any, are placed in the order of their formals.

3.5 Instruction set

This is a short overview of the instruction set of the CLR. I have not shown stack transitions, or all arguments. For the full specification see [gT01c].

Some opcodes have special short versions, specified by a `.s` suffix. Some instructions need to specify the type of the element that they operate on, this is signified with a T, which can be any one from table 3.1.

Instructions that reference classes, methods, fields and other symbolic information take a *token* argument, **¡T¿**. This is a 32-bit value that indexes into the metadata tables. These token values will not be consecutive integers, as they are divided into different classes. Method references start with hexadecimal value 0x06000000, class references with 0x02000000 and so on. See [gT01b].

The instruction set can be divided into two parts:

- The basic set which is powerful enough that any language could be implemented. The instructions include function calls, control flow, arithmetic, pointers and so on.
- The object model instructions. These are specially tailored to implement certain languages that follow the Common Language Subset. This subset defines a class based, object oriented language with single inheritance, multiple implementation, exception handling and garbage collection very much like C# or Java.

3.5.1 Basic opcodes

These instructions, together with the load and store operations shown earlier, make up the basic part of the instruction set. Some arithmetic instructions have special versions for operations on unsigned types The instructions where special unsigned versions exists are marked with a `.un` in parenthesis.

add,sub,mul,div(.un),rem(.un),neg Polymorphic arithmetic instructions. Works on both floating-point and integer types. Both arguments (for binary operations) need to be of the same type.

add.ovf(.un),sub.ovf(.un),mul.ovf(.un) Integer arithmetic with overflow detection.

and,or,xor,not,shl,shr(.un) Logical operations. Can be used on values of type int32, int64 and natural int.

conv.T**, conv.ovf.**T Convert the top-of-stack to the given type, with or without overflow detection.

chkfinite Check whether a floating point value is finite.

ldc.T**,ldc.i4.**X Load a constant on the stack. The instruction need to specify the type of the constant, T. The constant is stored "inline" in the instruction stream following the instruction. A special compact

encoding exists in the case where the constant is an integer in the range: $-1 <= X <= 8$.

dup,pop Duplicate or drop top-of-stack element.

ceq,cgt(.un),clt(.un) Compare two values.

volatile. This instruction can be prefixed load or store instructions. Specifies that the value should be accessed with volatile semantics.

3.5.2 Control flow

All the following branch instructions come in two flavors. The ones shown below which accepts a 32-bit (signed) offset, and a short version which only takes an 8-bit offset.

br Branch unconditionally.

ble(.un),blt(.un),bge(.un),bgt(.un),bne.un,beq Compare two topmost values, and branch accordingly.

brtrue,brfalse Compare top-of-stack and branch.

switch Branch according to offsets following instruction, chosen by the top-of-stack value.

3.5.3 Function calls

These are the basic instructions for implementing functions. They are independent of the object model. For each method there will

call ¡T¿ Invoke method referenced by token.

ret Return from method.

tail. A prefix instruction. If placed before a `call` specify that a tail-call is desired.

Actual arguments to a function shall be loaded on the evaluation stack before issuing `call`. They are then moved from the stack to the argument array of the callee by the runtime.

3.5.4 Objects

These instructions are part of the extended instruction set that specifies an object model for the CLR.

ldnull Load a null reference on the stack.
ldstr ¡T¿ Load a literal string.
ldtoken ¡T¿ Load a runtime handle representing type **¡T¿** on the stack.
newobj ¡T¿ Create and initialize object. Token should be a reference to an object constructor.
castclass ¡T¿ Attempt to cast object on stack to class given by token.
ldfld ¡T¿,stfld ¡T Load or store field.
ldsfld ¡T¿¿,stsfld ¡T¿ Load or store static field.
callvirt ¡T¿ Call a virtual method. Which method to call is determined at runtime from the type of the object reference on the stack.
box ¡T¿,unbox ¡T¿ Box/unbox a simple- (integer, long,...) or value-type inside an object of reference type.

3.5.5 Arrays

Only single-dimensional arrays are directly supported by the virtual machine. When storing or loading elements in an array, it is necessary to specify the type of the element. T can be any one of the types in table 3.1.

newarr ¡T¿ Allocates a zero-based, one-dimensional array.
ldlen Load length of array on stack.
ldelem.T, stelem.T Load or store an element of type T in array.

3.5.6 Exception handling

Exception handlers specify a range of instructions that are to be protected, and a range of instructions that contains the given handler.

Instructions that deal with exceptions are:

throw Throw an exception.
rethrow Within an exception handler, rethrow the exception.
leave(.s) Leave protected block.
endfinally Mark end of a finally exception handler.
endfilter Mark end of an exception filter block.

3.5.7 Pointers

The CLR operates with three different kinds of pointers. Pointer operations can operate on any of those three. A subset of the pointer operations are, somewhat surprisingly, verifiable. Full use of pointer arithmetic is, of course, not.

transient A transient pointer points to local variables and arguments. These are only valid in the scope of the current method.

managed Managed pointers can point to data on the garbage collected heap. There are some restrictions on them, for example they cannot be *null*. If the garbage collector moves objects around when compacting the heap, these pointers are updated.

unmanaged An unmanaged pointer is equivalent to an integer of hardware dependent size, as in ANSI C. There are no restrictions on operation on these pointers.

ldloca(.s), ldarga(.s) Load address of local or argument.

ldflda ¡T¿, ldsflda ¡T¿ Load address of object field, instance or static.

ldelema Load address of array element.

ldind.T**, stind.**T Load/store value indirect through address obtained earlier.

ldftn ¡T¿,ldvirtftn ¡T¿ Load address of (virtual) method.

calli ¡T¿ Call method indirect through pointer.

3.5.8 Unsafe instructions

Some instructions are inherently unsafe. They are necessary to implement traditional languages, like C.

localloc Allocate space on local stack, similar to C alloca.

cpblk,initblk Copy or initialize an arbitrary block of memory.

arglist Used to implement C like varargs.

jmp ¡T¿ Jump to method specified by token.

3.6 CIL assembler

In this report, and in the back-end, I use the assembler format defined for CIL in the ECMA standards. The complete syntax in the form of a YACC

grammer is specified in [gT01d]. The semantics of the single instructions are defined in [gT01c], and the meaning of attributes related to class and method definitions are defined in [gT01b].

A short overview of the syntax is given in appendix C.

Chapter 4

The Pizza Compiler

In this chapter I give an overview of the Pizza compiler, and the extensions that it implements over Java.

4.1 Pizza

The Pizza compiler extends the pure Java language with some concepts known from functional programming. It supports Java version 1.3, compliant with the most recent Java Language Specification.

Consisting of about 30000 lines of code, the compiler is itself written in Pizza. Apart from constant folding and dead code elimination, it does not implement any optimizations.

The Pizza extensions are:

- Parametric polymorphism or *generics*.
- First-class functions.
- Algebraic types.
- Tail recursion.

The Pizza compiler can emit both (pure) Java source code, or compile straight to JVM byte-code, no extensions to the JVM are used to implement the Pizza constructions.

The *generics* part of Pizza, with some simplifications, has been selected to become part of the official Sun Java standard, known as Generic Java [Jav].

4.2 The Pizza extensions

Using examples, I will briefly introduce the Pizza concepts. As the implementation of the compiler uses these, it is necessary to know them, to be able to understand the source. Although the transformations from Pizza into straight Java are very interesting, I will not go into very deep details about them. They are performed on the abstract syntax tree before code generation, and so are not directly relevant for the back-end. If interested please refer to [ORW98].

4.2.1 Generics

In Pizza generic classes can be instantiated with both reference and basic types.

An example of a generic class is the Pair class from the Pizza source.

```
public class Pair<A, B> {

    public A fst;
    public B snd;

    public Pair(A fst, B snd) {
        this.fst = fst;
        this.snd = snd;
    }
}
```

This defines a class that contains two objects of generic type. Instances of this class are instantiated by specifying the types of A and B.

```
Pair<int,String> p = new Pair();
```

Notice that the type parameters are only specified on the type, not the constructor.

The Pizza distribution contains generic classes for hashtables, sets, vectors and more. In the implementation of the compiler generic classes are used, among other things, for the symbol table and for environments used during attribution and code generation.

The parametric types are translated into Java by *type erasure*. In [ORW98] they name this their *homogenous* translation, in opposition to their *heterogeneous* translation which works by specialization of the classes at runtime, using a modified class loader. The current Pizza compiler only implements the homogeneous translation, which has some drawbacks. Because type information is not preserved in the resulting code, using Reflection[1] on Pizza classes does not give meaningful results. Also simple types has to be encapsulated within a corresponding reference type, which incurs some overhead.[2]

4.2.2 First-class functions

In Pizza functions can be used as values. A function type can be declared as:

(argtype , ..., argtype) throws exception , ..., exception −> resulttype

It is also possible to create anonymous functions. The following example shows a higher-order map function, which demonstrates all of the above.

```
// this method takes a function as argument
public Object[] map ((Object) -> Object f, Object[] a) {
  Object newa = new Object[a.length]
  for (int i = 0; i<a.length; i++) {
    newa[i] = f(a[i]);
    }
  }
...
...
Object [] list = new Object[10];

// This is a variable of function type
```

[1] *Reflection* is the ability at runtime to retrieve names and types of fields and methods of a class.

[2] This is usually known as *boxing*

```
(Object) -> Object f;

// Which is now bound to a anoymous
// function
f = fun (Object o) -> Object {
  if (o==null)
    return new Boolean(0);
  else
    return new Boolean(1);
}

// Which is then used in the call to map
map (f, list );
```

Anonymous functions are used in the compiler to implement lazy loading of class information, and to emit code for return statements.

4.2.3 Algebraic datatypes

Algebraic datatypes should be well known from functional programming, the Pizza syntax being the only difference. In ML one could write:

```
datatype AST = Package of AST
             | DoLoop of AST * AST
             | NewArray of AST * AST list
             ...
```

In Pizza this would be declared in the following way:

```
public class AST {
  public case Package(AST qualid);
  public case DoLoop(AST cond, AST body);
  public case NewArray(AST elemtype, AST [] dims);
   ...
 }
```

Pattern matching on values of algebraic type is then done using a (modified) **switch** statement. This is used many places in the Pizza compiler for operations on the abstract syntax tree (from where the above example is taken).

```
 ...
static void genStat(AST tree, ...) {
  switch(tree) {
    case Package(_):
    break;
    case DoLoop(AST cond, AST body):
      // emit loop
      // locals "cond" and "body" are bound in
      // this scope
    break;
     ....
}
```

Note that it is necessary to specify the types in the pattern even though this could in theory be inferred from the selector.

4.2.4 Tail recursion

Tail recursion is not used anywhere in the compiler, and is included in Pizza mostly for reasons of completeness. It is implemented using the "tiny interpreter" transformation from [Jon92], and therefore not very efficient. The special case, where the function is self-recursive, could in theory be optimized and implemented using a jump with the `goto` bytecode, but this is not done in Pizza.

A tail recursive method is declared with the **continue** modifier. The recursive call is the specified using a special **return goto** expression. The following piece of Pizza demonstrates this with the classic iterative implementation of the factorial function:

```
continue int itfac(int n, int m) {
    if (n == 0)
      return m;
    else return goto itfac(n−1, n∗m);
```

Chapter 5

Design

In this chapter I will look at what code should be emitted for the CLR. The goal is to make the change as transparently as possible, to make sure that a program running on the CLR would behave identical to the program running on the JVM.

5.1 Overview

The style of the discussion in this chapter is somewhat informal. The Java language specification is large and complicated, but the required dynamic semantics are captured in the JVM. Therefore in the attempt to figure out how to do the translation, I have looked at how it is translated for the JVM, and how a corresponding translation can be made for the CLR. Some parts warrant more discussion than others. Also on some exotic parts of Java I will lay out a possible solution, even though its implementation will be deferred.

I this chapter I do not care about the Pizza extensions, but only looks at the core Java language. This is sufficient, since the Pizza extensions are reduced to this. The JVM has no defined assembler syntax, but when necessary, I will use the "de-facto standard" as defined by the `jasmin` assembler and its corresponding book [MD97].

5.2 CLR assembly files

An *assembly* is a file containing an executable for the CLR. It corresponds to one or more JVM class files, because one assembly can contain definitions of more than one class. The actual file format of assemblies are a lot more complicated than class files, so I will not attempt to generate them directly in the first iteration of the back-end. Instead the back-end shall generate assemblies in textual form, and then let the details of the file format be handled by the `ilasm` assembler included in the .NET SDK.

It would probably be beneficial to look the example given in appendix D to get an overview of how the assembler files look, and how fields, methods and classes are defined and referenced. All the following discussions will use this syntax.

5.2.1 Name resolution

Java classes are normally resolved from their fully qualified name which consists of the package they belong to, and the classname. For example the class *System* in the package java.lang are refered to as *java.lang.System* in the source. For symbolic references in the class files, this is then converted into "java/lang/System". When the classloader needs to load this class, it looks for this class in the filesystem using the path `java/lang/`.

In the CLR class names are not resolved this way. For references to symbols that are not defined in the same assembly, it is necessary to know the name of the assembly containing the class. So if the *java.lang.System* class is defined in an assembly named "BJLIB.dll", references to it needs to include this assembly reference as:

```
[BJLIB]java.lang.System
```

This maps poorly from the Java method of resolution based on the fully qualified class name relative to a classpath.

An intermediate solution during the bootstrap phase of the compiler is to simply force all external references to either **BJLIB**, the J# clss library.

5.2.2 Scoping

The fully qualified name of a Java class includes the package name, if any. As an example the fully qualified name of the following class:

```
package tests;

public class testbranch {
 ...
}
```

is *tests.testbranch*. This will be translated using the **.namespace** keyword. So the above class would be defined in CIL as follows.

```
.namespace tests
{
  .class  testbranch  ...
  {
    ..
  }
}
```

5.2.3 Symbolic references

In the JVM all references to constants, class and method references and so on, are encoded as an integer offset into the *constant pool* of the class. In the CLR the references are not directly offsets into the metadata, but *Tokens*, as explained in chapter 3.

As long as the back-end does not create assemblies directly, I need not worry too much about this, since the generation of correct tokens are taken care of by the assembler. Instead all references need to be written out in their canonical text form.

5.3 Basic types

The CLR supports a wider range of types than does the JVM. Since the Java language only uses signed integer types, the JVM only operates on

signed values, except for Unicode characters. How the JVM types are mapped to the CLR are shown in table 5.1.

JVM Type	CLR Type
byte	int8
boolean	bool (int8)
short	int16
char	char (uint16)
int	int32
long	int64
float	float32
double	float64

Table 5.1: Mapping of basic types from JVM to CLR.

There are really no surprises here. The CLR types are sufficient to support Java without any problems or conversions necessary.

5.4 Reference types

Java programs expects that *java.lang.Object* to be top of the class hierarchy. This means that any other type, excluding basic types, is a subclass of it, and values of any type can be assigned to locations of type *Object*.

In works similarly in the CLR but it has *System.Object* at the top of the hierarchy. The solution depends on:

- Whether we want Java programs to be able to directly access CLR types.
- Or whether they are just to be kept within their Java "solitary confinement".

If we really want Java programs to integrate properly, then we need to convert all instances of *java.lang.Object* to *System.Object*. If not, then we could in theory just ignore it, and define *java.lang.Object* as just another class. This would work, because all other Java classes that we could define in a Java program would inherit from it. A small part of the class library would look like this (in CIL assembler):

```
.namespace java.lang {
   .class  Object extends System.Object {
     .field  ...
     .method hashCode () { .. }
    }
}
```

But there is a problem with this approach. Some classes get special treatment by the runtime. When compiling for the JVM we expect that:

- Arrays can be treated as subclasses of *Object*.
- Literal strings are instances of *String*, which is a subclass of *Object*.
- All exceptions inherits from *java.lang.Throwable*.

The problems with strings and exceptions will be treated in later sections.

For *Object* the best solution would be to transparently substitute it with *System.Object*. When doing this we would just need to make sure that the public interface is preserved.

Static methods do not necessarily present a problem. If they cannot be directly mapped onto the CLR type, then methods in an extra hidden class could be created to perform their function. Virtual methods on the other hand need some consideration, because they might be overridden in a subclass. But in the case of *java.lang.Object* this is not a problem, since it only has five virtual methods, and they can all be mapped 1:1 on *System.Object*, as shown in table 5.2.

java.lang.Object	System.Object
int hashCode()	int32 GetHashCode()
boolean equals(Object)	bool Equals(Object)
String toString()	string ToString()
Object clone()	object MemberWiseClone()
void finalize()	void Finalize()

Table 5.2: Mapping from *java.lang.Object* to *System.Object*

5.4.1 The current solution

To be able to use the class library from J# I have been forced to use their solution to this problem, which is the first solution from above. What this means for exceptions and strings is explained later. The major problem is arrays. We would like to be able to do:

```
Object o;
o = new int[10]   // but an array is subtype of Object
```

To do this we define a method `getJavaObjectFromSystemObject` as:

```
method public hidebysig static class [BJLIB]java.lang.Object
        getJavaObjectFromSystemObject(object o) cil managed
{
  .maxstack 1
  IL_0000:  ldarg.0   // load object on stack
  IL_0001:  ret       // and return it as different type
}
```

Using this unsafe method, we are able to cheat the verifier into believing that its all-right to do the assignment. It is now possible to compile the above Java code as:

```
   .local ( [BJLIB]java.lang.Object 'o')

   ldc.s     10
   newarr    int32  // this is a System.Object
   call      [BJLIB]java.lang.Object getJavaObjectFromSystemObject
             ([mscorlib] System.Object) // but now its a java.lang.Object!
   stloc     'o'    // and it will work.
```

Because the method that performs the unsafe type "conversion" is placed in a local library, it is not part of the verification process, and the runtime will believe that the code is now type-safe. This is of course an illusion, and it only works because the class library always checks the runtime-type of arguments, to determine the real type, and acts accordingly.

5.5 Arrays

5.5.1 Creation

The JVM has three instructions for creating arrays: `newarray`, `multianewarray` and `anewarray`.

These three instructions are used to handle three different cases correspondingly:

1. Single-dimensional array of simple values (int, long, float, double): `long[] l = new long[10]`.
2. Single-dimensional array of reference type. This could either be: `Object[] oa = new Object[10]` or implicitly as in `long[][] la = new long[10][]`.
3. Multi-dimensional arrays: `int[][] = new int[10][20]`.

The CLR instruction set only deals with single dimensional arrays (called *vectors* in the documentation!) Creating a one-dimensional array (of any type, both value and reference) is done with `newarr`. The first case is easily handled:

```
.locals ( int64 [] 'l' )
...
ldc.s   10
newarr  int64
stloc   'l'
```

Arrays of reference types in the second case, does not need special treatment, whether the type is some class, or another array.

```
.locals ( class [BJLIB]java.lang.Object [] 'oa',
          int64  [][] ' la ' )
...
ldc.s   10
newarr  [BJLIB]java.lang.Object
stloc   'oa'
ldc.s   10
newarr  int64 []
stloc   'la'
```

Multidimensional rectangular arrays in the CLR are supported through the library by the *System.Array* class. It has methods to create and access array elements, although these would probably be inlined by an optimizing JIT compiler.

In Java multidimensional arrays are really arrays of arrays, and I need to implement this so I cannot use the library support. A Java program would do:

```
int e1,e2 = ...
int [][] a = new int[e1][e2];
```

Which actually means:

```
int e1,e2 = ...
int [][] a = new int[e1][];
for (int k=0;k<e1; k++) a[k] = new int[e2];
```

and the back-end needs to generate code to do this inline. This is complicated by the fact that the dimension sizes need not be compile-time constants. The code generator needs to handle arrays of up to 255 dimension. These initializers could either be generated as ASTs first or emitted directly. The latter approach will be used in the back-end.

5.5.2 Array covariance

Array covariance presents a particular problem in the chosen design of trying to pretend *java.lang.Object* is still at the top of the class hierarchy. If T_1 is a subtype of T, then a location of type $T_1[]$ is considered to be assignment compatible with another of type $T[]$:

```
Object[] o = new Object[10];
String [] s = new String[10];

o = s // legal : String extends Object
s = o // illegal
```

Because of this, the runtime need to check the dynamic type of the element at each array access, and throw an exception if they are incompatible. This

is the case for both the JVM and the CLR. This means, that we cannot just use the previous solution to cheat the runtime to *think* that the types are compatible. A sneaky Java program might try to do something like the following, because arrays are supposed to be subclasses of *java.lang.Object*.

```
// We think we can put anything in this array
Object[] o = new Object[10];
// including arrays ...
o[1] = new int[20];
```

Which would then be compiled into:

```
// This does not work
.locals ( class [BJLIB]java.lang.Object[] 'o')
 ...
ldloc.s       'o'
ldc.s         1
ldc.s         20
newarr        int32
stelem.ref     // exception, not compatible
```

This would fail with an *ArrayTypeMisMatch* exception at runtime, because the runtime performs the check of the dynamic type at array accesses.

In theory there are two solutions to this problem. A solution would be to box all CLR arrays inside a Java type. This solution is undesirable because it would incur overhead at every array access.

The solution is to substitute the element type. So when the Java program allocates an array of *Object*, what it will really get is an array of *System.Object*.

```
  // This works
.locals ( class [BJLIB]System.Object[] 'o')
ldc.s         10
newarr        [mscorlib]System.Object
stloc.s       'o'
 ...
ldloc.s       'o'
ldc.s         1
ldc.s         10
newarr        int32
```

```
stelem.ref        // OK, int32[] subtype of System.Object
```

Now things will work as expected, because the element type really is the top of the class hierarchy.

5.6 Classes

When defining a class in an assembly, there is a number of attributes. In CIL assembler syntax, it looks like the following:

```
.class <modifiers> <type> auto beforefieldinit <name>
  extends <classname>
  implements <interfacename>{,<interfacename>}
{
  // methods and fields
}
```

The *modifiers* are explained in a later section. The *type* of a class can only be either empty or:

interface This is an interface. It will only contain public abstract virtual methods and static fields.

If the **interface** type is not present, it is a real class. The **auto** keyword signifies that layout of instances of this class can be determined by the runtime. This is the case for all classes when translating Java. The **beforefieldinit** keyword will be explained later. A class can extend another class, and implement a number of interfaces.

5.6.1 Inner classes

Inner classes was introduced with Java version 1.1, after the specification of the JVM was already set in stone. The implementation of inner classes was therefore done without requiring any extra runtime functionality. There is three kinds of inner classes: “nested top-level”, “member” and “local”.

Since the JVM does not directly support any of these, they have been implemented by creation of auxiliary classes with mangled names, and in

the case of local and member classes, an extra hidden field in the class used to reference the enclosing class.

The CLR directly supports the definition of "nested top-level" classes and interfaces. The definition of a class can be nested within another:

```
.class private Test extends java.lang.Object {
  .field int32 i
  .class nested assembly NestedClass extends java.lang.Object {
    .field float f
    .method .....
  }
}
```

But there is no support for the other two types of inner classes that Java needs. [1]

Since not all of the necessary infrastructure to directly support all types of inner classes is present, it does not seem beneficial to make any changes in the current scheme of "flattening" inner classes.

5.6.2 Object creation

Object creation in the JVM is handled in two steps. First the space for the object is created by issuing the `new` opcode. Then one of the class constructors are called using `invokespecial`. This partition of the object creation process has several nasty implications for the verification of legal JVM code. The following must always be true:

- An object must not be used before it has been initialized
- Only a constructor from the class itself is allowed to initialize the class. (Specifically, a constructor from a superclass cannot be used)
- An object must only be initialized once.
- If exceptions are thrown by the constructor, the object must not be used, since it might not be initialized.

This is explained very briefly in [LY97], and researched thoroughly in [DD00] and [FM99]. There seems to be no good reason for the design

[1] C# does not support inner classes, instead "delegates" are supposed to implement the same functionality

of splitting up object creation up in this way. The CLR avoids this complexity by doing it in one step. The opcode `newobj` takes a token argument that specifies both the desired class of the new object, and the constructor that must be called. The arguments, if any, must have been pushed on the stack prior to this.

Since it is illegal to use or depend on the object before the constructor has been called, this change does not matter for the semantics of the translated code.

The Java code:

```
StringBuffer b = new StringBuffer(10)
```

will then be translated into CIL assembler as:

```
ldc.s   10
newobj java.lang.StringBuffer ::. ctor(int32)
stloc    "b"
 ...
```

where `.ctor` is the name reserved for constructors in the CLR, corresponding to `<init>` in Java. Luckily Java identifiers cannot contain `.` so there are no problems with name clashes due to the different reserved names. Currently there is a problem, because the compiler reads class files, where constructors are named `<init>` and this is used during the type check. The conversion of the constructor names need to be deferred until the last moment.

5.6.3 Methods

When defining methods, they have a number of attributes. Shown in CIL assembler syntax, a method header has the following components:

```
.method <modifiers> <method-type> hidebysig <return-type> <name>
        ( {<arguments>} ) cil managed
{
  // method body
}
```

The different *modifiers* will be dealt with later.

There are four *method-type* modifiers that are relevant when compiling Java programs.

static The method is static.
instance The method is an instance method. This is only the case for constructors in Java.
virtual The method is virtual. All methods that are not either static, or an instance method, shall be virtual.
abstract The method is abstract. The body must be empty.

The **hidebysig** modifier is ignored by the runtime, but is a directive to tools, that overriding of methods in subclasses is determined by the combination of method name and types of arguments. The **cil managed** modifiers at the end of the header signify, that this method is not a native method. How to implement Java native methods will not be considered.

Unlike the JVM which has 4 different byte-codes to invoke methods, depending on whether the method is from an interface, is a constructor or a virtual method, the CLR only has two different instructions for method invocation.

call Can be used to call any type of method. If the method is a virtual method, which method to call is determined by the static type of method reference, not the dynamic type of the object. This instruction shall be used where `invokestatic` and `invokespecial` is used for the JVM.
callvirt Call a virtual method, regardless of whether it is a class or interface method. This is used where `invokevirtual` and `invokeinterface` would be used when compiling for the JVM.

Before emitting the call instruction the arguments should have been placed on the evaluation stack. If the method is virtual, the first argument should be a this reference, the rest of the arguments should be placed in order of appearance of the formals.

5.7 Constants

5.7.1 Numeric

In the JVM all constants are placed in the constant pool, and each class has its own pool. With the exception of those frequently used small numbers that have special instructions to load, all constants have to be fetched from the constant pool.

To load an integer, float or String the `ldc` bytecode is used. The `ldc_w` takes as argument a 16-bit constant pool offset. Constants that occupy two words of storage are fetched using the `ldc2_w` bytecode.

In the CLR numerical constants are treated more like in normal hardware architectures, so constant operands follow the opcode in the instruction sequence. To load a constant on the stack one of the `ldc.`T instructions are used, depending on the type T.

For integer constants in the range from -128 to 127, a special short instruction is available: `ldc.i4.s`. If a 64-bit long value can be represented in 32-bit, it should be loaded as:

```
ldc.i4    <32-bit value>
conv.i8
```

5.7.2 Strings

Literal strings are placed in the metadata part of an assembly. To load a string the `ldstr` instruction, with a token argument pointing into the metadata, is used.

A Java program expects literal strings to behave as subclasses of *Object*. One would, for example, want to call the method `equals` (inherited from *Object*) on a string:

```
String s = "World Hello";

if ("Hello World".equals(s)) {
  // do stuff
}
```

There is two solutions to the problem:

1. All literal strings are converted from *System.String* to *java.lang.String* when they are loaded onto the stack.
2. Or *java.lang.String* is transparently changed to *System.String* in the same way as *java.lang.Object* could be.

Although it would be possible to map the methods of the Java string type to the CLR string type, it will be easier just to use the first method. Since literal strings are only encountered as result of a `ldstr`, it is simply a matter of issuing a call to a method to convert it after each.

If we desire to ease inter-operability with other languages, then it would be necessary to use the second approach so that, for example, a C# class would be able to call Java methods using the native string type. But this issue has not been considered.

5.8 Arithmetic instructions

The existing JVM back-end needs to keep track of types on the evaluation stack, to be able to issue the right instruction. CLR arithmetic instructions does not care about the types of its operands, but we still need to keep track of the types, because both operands need to match.

In table 5.3 is a list of Java operators, and the corresponding CIL operation. There are no major surprises in the choices.

Floating point operations in the CLR are specified to follow IEEE754 conventions. Integer division and remainder are defined identically for both the JVM and the CLR, with "round towards zero" semantics.

Strict floating-point

Java since version 1.3 supports the **strictfp** modifier for classes, interfaces and methods. When present, it indicates that all floating-point operations shall have the strict IEEE754 semantics. This was default behavior earlier, but was reversed due to performance considerations on Intel.

The CLR does not directly support this. Using explicit insertion of the conversion instructions (`conv.r4` and `conv.r8`) it would be possible to emulate it. Conversions need to be placed after each operation.

Java operator	CIL instruction	Description
+	add	addition
-	sub	subtraction
	mul	multiplication
/	div	division
%	rem	remainder
&	and	bitwise and
—	or	bitwise or
ˆ	xor	bitwise xor
˜	not	bitwise complement
>>	shr	shift right with sign extension
<<	shl	shift left
>>>	shr.un	shift right with zero extension

Table 5.3: Arithmetic instructions

Compound operators

The JVM has many instructions for stack manipulation, some of which are:

dup2 Duplicates the two top stack elements.
dup_x2 Duplicates top of stack, and stores it beneath the third element.
swap Swaps the two top elements

These are rarely used, but can be very convenient when implementing compound or post-increment or decrement operators (where it is important that the operand is only evaluated once). For example:

```
int i = 2, j;
j = i++;
```

can be compiled into the following JVM code: (presuming i resides in local variable slot 1 and j in slot 2)

```
iload_1      // stack : ...2
dup          // stack : ...2 2
iconst_1     // stack : ...2 2 1
iadd         // stack : ...2 3
istore_1     // stack : ...2
```

```
istore_2        // stack : ...
```

Equivalent code could be generated for the CLR, since it also has `dup` and `pop` instructions. But the problem arises when operating on array elements, because the element reference takes up two stack elements, one element for the array reference, and one for the index. The following code gives an example:

```
int a [] = {2,0};
a[1] = a[0]++;
```

This will translate nicely using stack manipulation into:

```
aload 0         // stack : ... a
iconst_1        // stack : ... a 1
aload 0         // stack : ... a 1 a
iconst_0        // stack : ... a 1 a 0
dup2            // stack : ... a 1 a 0 a 0
iaload          // stack : ... a 1 a 0 2
dup_x2          // stack : ... a 1 2 a 0 2
iconst_1        // stack : ... a 1 2 a 0 2 1
iadd            // stack : ... a 1 2 a 0 3
iastore         // stack : ... a 1 2
iastore         // stack : ...
```

The CLR does not have instructions equivalent to the `dup2` and `dup_x2` necessary to do this. Instead it is necessary to use temporary variables to store intermediates, which makes the code rather unwieldy.

```
ldloc .0        // ... a
ldc . i4 .1     // ... a 1
ldloc .0        // ... a 1 a
ldc . i4 .0     // ... a 1 a 0
stloc .s 5      // ... a 1 a
stloc .s 4      // ... a 1
ldloc .s 4      // ... a 1 a
ldloc .s 5      // ... a 1 a 0
ldloc .s 4      // ... a 1 a 0 a
ldloc .s 5      // ... a 1 a 0 a 0
ldelem.i4       // ... a 1 a 0 2
dup             // ... a 1 a 0 2 2
```

```
stloc.s 6      // ... a 1 a 0 2
ldc.i4.1       // ... a 1 a 0 2 1
add            // ... a 1 a 0 3
stelem.i4      // ... a 1
ldloc.s 6      // ... a 1 2
stelem.i4      // ...
```

In [Gou02] a solution for the CLR using *transient pointers* is shown. A transient pointer to the array element can be obtained, and then the element reference only uses one stack slot, and the first idiom shown above can be used again, without using any temporaries.

5.9 Local variables

The CLR does not have the irregular design of the JVM where all variable slots are 32 bit in size, so that wide types like `long` and `double` occupies two slots.

Slots can be reused, but only for storing values of the same type. This is because the CLR depends on the declared types of the locals to track the types on the stack for verification and code generation. In the following example, the variable j goes out of scope before k is allocated.

```
{
    int i=1;
     ...
    {
      int j=2;
       ...
    }
    int k=3;
}
```

When generating code from the above Java, it is possible to use only two local variables.

```
    .locals (int32 V_0,
             int32 V_1)
    ldc.i4.1
```

```
stloc.0
 ...
ldc.i4.2
stloc.1
 ...
ldc.i4.3
stloc.1
```

Because locals in the CLR has a declared type, full type information have to be carried around during code generation all the way to the final assembly generation.

5.10 Exception handling

The CLR supports structured exception handling. An entry in the exception handler table for a method is defined by the following:

- The offset of the first protected instruction in the protected block.
- The offset *after* the last protected instruction in the block.
- The offset of the first instruction in the exception handler.
- The offset *after* the last instruction in the handler.
- A token that specify the class of the exception that is handled.

The *protected block* corresponds to the block enclosed in the **try**, and the *handler block* can be either a **catch** or **finally** block.

The following Java program:

```
try {
  throw new Exception
}
catch (Exception) { // catch it }
```

shall be translated into CIL as:

```
.try
 {
  lab1: newobj [BJLIB]java.lang.Exception
        throw
  lab2:
```

```
} catch [BJLIB]java.lang.Exception
  {
   // catch it
  }
```

This syntax is accepted by `ilasm`, but the equivalent, and easier for the compiler to generate, is:

```
lab1:  newobj [BJLIB]java.lang.Exception
       throw
lab2:
       //catch it
lab3:
.try lab1 to lab2 catch [BJLIB]java.lang.Exception handler lab2 to lab3
```

The CLR runtime, unlike the JVM, does not have any restrictions on the classes of objects that can be thrown. So any class could in theory be thrown by the `throw` opcode. But it is highly discouraged to make exceptions that are not subclasses of *System.Exception*, if for example interoperability with C# is desired.

There are some restrictions placed on flow of control in and out of protected and catch-handler blocks in the CLR. For the code to be verifiable, these must be obeyed:

- A protected block can only be entered at the first instruction, no branches must transfer control into the middle of it.
- Once inside a protected block, it must only be left in one of these three ways:
  - By falling through the bottom of the block.
  - By throwing an exception.
  - Or by transferring control outside the block using the `leave` instruction.

This has consequences for code generation for **break**, **continue** and **return** statements, which will be handled later.

The CLR treats finally-handlers in the same way as catch-handlers. When a finally handler has been defined, it will be executed by the runtime at the exit of a block, whether the exit is normal or abnormal through an exception. This is what we require for Java.

Each handler only handles one exception, or a finally, so several protected blocks must be defined if more than one handler is present. For example:

```
try {
  // something
}
catch (IOException e) { // catch 1}
catch (Exception e) { // catch 2}
finally { // finally }
```

This shall result in three protected blocks being defined, with the range of protected instructions being the same for the two catch handlers, but with the ranges of the handlers themselves being disjunct. The finally handler shall span the catch blocks also:

```
lab1:
   // something
lab2:
   // catch 1
lab3:
   // catch 2
lab4:
   //finally
   endfinally   // this ins signifies the end of the handler
lab5:

.try lab1 to lab2 catch [BJLIB]java.lang.IOException handler lab2 to lab3
.try lab1 to lab2 catch [BJLIB]java.lang.Exception handler lab3 to lab4
.try lab1 to lab3 finally handler lab4 to lab5
```

Virtual machine exceptions

For exceptions that are thrown by explicit **throw** statements, it does not matter how class library support has been implemented. A problem arises only in exceptions thrown by the system, as the following example shows:

```
int mightBeZero = 0;
try {
    int i = 1/mightBeZero;
}
```

```
catch (ArithmeticException e) { ... }
```

The *ArithmeticException* would be thrown by the virtual machine itself, as response to a hardware condition. When running on the CLR it would throw the *System.DivideByZeroException*, from the .NET class library, but the generated code would expect to catch *ArithmeticException*. For this to work as expected there are two solutions:

- A "filter" block before the exception handler.
- Mapping of JVM system exceptions onto the CLR ditto.

The CLR allows the compiler to insert a *filter* handler. This is a block of arbitrary code that, for example, could convert the CLR exception into an equivalent JVM one, before transferring control to the desired catch handler. This is the way it is done in the J# class library.

A better solution would be to map from Java to CLR exceptions. So when the Java program tries to catch *java.lang.ClassCastException*, the back-end would emit code that tried to catch *System.InvalidCastException* instead.

This mapping is not as straightforward as is the case with *Object* unfortunately. There is no direct equivalent in the CLR type to the `fillInStackTrace` method of *Throwable*, and a workaround would need to be devised.

It is not possible to conserve exception timing fully. Exceptions that relate to loading of classes, ie. TypeLoadException may typically, according to [gT01c], be thrown at initial load time when the IL is converted into native code. Whereas in the JVM these exceptions would be thrown by the classloader at any time during execution, when a reference to a class is resolved. This is a very minor issue, though.

break/continue

Java does not have **goto** statements, but allows **break** and **continue** statements with or without labels. These are used to either terminate the enclosing statement, or to skip an iteration, in the case of **continue**. When generating code for these statements, a branch outside the enclosing statement will be issued. In code as the following:

```
for (int i = 0; ; i++) {
  // if some condition c1
```

```
    break;
    // rest of loop
  }
```

This would be translated as:

```
top :...
    <c1>
    br   end
     ...          // rest of loop
    br   top
end:
```

While this would normally work, the problem arises when exceptions are involved. It is not allowed to transfer control out of protected block, with anything else than the `leave` instruction. So in the following case we have a problem, because the branch transfers control outside the protected block.

```
  for (int i = 0; ; i++) {
    try {
        // if some condition
        break;
      }
    catch (Exception e) { .. }
  }
```

Fortunately the solution is simple. Instead of the `br` instruction a `leave` should be emitted. According to [gT01c] `leave` can be used outside protected blocks also, in which case it just functions as a normal branch. So the back-end can use `leave` when translating any break or continue statements, whether inside a protected block or not.

Return

For return statements a similar problem arises when encountered within a protected or handler block. For example:

```
int test () {
    ...
   try {
```

```
        if (something)
            return 1;
        else
            return 2;
    }
    catch (Exception e) { ... }
}
```

Because it is not allowed to return from within a protected block, the return statements needs to be "lifted" outside of it. The same is the case with **return** statements within **catch** clauses or **finally** blocks.

This mean that when generating code for return statements, it is necessary to determine whether the statement is placed within a protected block. If not, a `ret` instruction can simply be emitted at the current position. If we are within a protected block, then it is deferred, and instead a `leave` instruction is emitted. If a value need to be returned, then a local variable is first allocated, and the value stored in this.

When the end of the protected block is reached, then any pending return clauses are emitted, and the `leave` instructions within the block have their target adjusted so that they jump to the correct clause.

This is complicated by the fact that protected blocks might be nested, so the return statements need to be lifted all the way out of a number of environments. In the example shown above, code like this would be created:

```
.try {
  // if something
   ...
  ldc.i4.1        // store return value
  stloc.0         // in local slot 0
  leave   lab1
  ldc.i4.2        // store return value
  stloc.1         // in slot 1
  leave   lab2
}
catch Exception {
  // catch block
}
lab1: ldloc.0     // retrieve return value
      ret         // from correct slot, and return
lab2: ldloc.1     // do
```

ret

5.11 Modifiers

Modifiers are attributes on classes, methods and fields, that restrict access to, or specify certain properties.

Accessibility

Methods and fields in Java can have four different accessibility modifiers, here listed in order of wider access.

private The member can only be accessed from instances of the specific class.

default (package protected) This is the default modifier, if no other modifier is present. The member can be accessed from classes in the same package.

protected The member is accessible from subclasses, and from other classes in the same package.

public The member can be accessed by all classes.

Classes in Java can have one of two different visibility modifiers:

public A public class is visible in all packages.

default A class without modifier has default visibility. The class is only visible within the same package.

In the CLR the granularity of access modifiers are at the assembly level, which fits poorly with Java, where it is at packages. Because pizzacil will compile several packages into one assembly, it is not possible to obtain exactly the same meaning as in the original Java source.

Classes in the CLR can only have either private or public visibility. The CLR does not allow interfaces or abstract classes to be declared private, so for those it is necessary to widen the default visibility into public. To keep things consistent, normal classes with default visibility will also be declared public.

	Java modifier	CIL modifier
Classes	public	public
	default	public
Members	public	public
	private	private
	default	assembly
	protected	famorassem

Table 5.4: Accessibility modifiers

The default modifier is translated into assembly accessibility. This will most likely be too wide, because more than one Java package has been translated into one assembly. The famorassem CLR modifier allows access to classes in the same assembly, or subclasses. Again this is wider than desirable, but it is the closest possible to protected.

Other modifiers

The rest of the Java modifiers that can be used are listed in table 5.5. Java uses the final modifier on fields, methods and classes with different meaning. A final field (that is not a compile time constant) is one that can only be assigned a value in the constructor. A final method is a virtual method that cannot be overridden in a subclass and a final class is one that cannot be subclassed. The CLR has different modifiers with identical semantics for these three cases.

Volatile variables are a special problem, which will be dealt with later. Native methods will not be considered currently.

5.12 Synchronization

In Java thread critical sections comes in two flavors, both declared using the **synchronized** keyword. It is possible to define an entire method to be a critical section by adding the **synchronized** modifier. The other possibility is to lock a given block of code:

	Java modifier	CIL modifier
Methods	synchronized	synchronized
	native	n/a
	final	final
Fields	volatile	n/a
	transient	notserialized
	final	initonly
Classes	final	sealed

Table 5.5: Other modifiers

```
Object o = new Object();

synchronized (o) {
  // Critical section
}
```

5.12.1 Synchronized methods

In the JVM synchronized methods are simply compiled into regular methods, with an extra modifier. Mutual exclusion is then ensured by the runtime. The CLR has a similar modifier, and it is defined analogous.

5.12.2 Syncronized blocks

Synchronized blocks are a bit more complicated to implement. When compiling the above example, the code in the critical section is encapsulated within a try-finally block. It is necessary to do this, to make sure that the monitor is exited, if the code in the critical section throws an exception. No explicit instructions exists in the CLR to implement synchronization, instead methods in the CLR class library can be used for the same purpose. Before entering the critical section, a call to the method `System.Threading.Monitor.Enter (Object o)` is issued where a `monitorenter` instruction would be placed for the JVM. In the finally

block, where a `monitorexit` instruction would be issued, we instead invoke `System.Threading.Monitor.Exit (Object o)` for the CLR.

5.13 Entry-point

Java defines the entry-point of a class to be a method with name `main`, being defined as **public static** and taking an array of strings as argument. A collection of classes being compiled can have any number of entry-points defined.

When compiling an executable for the CLR one, and only one, specific method *must* be designated as the entry-point. Furthermore the entry-point that the CLR expects is a method taking an array of *System.String* elements, while the Java entrypoint of course expects the elements to be of type *java.lang.String*.

To accomplish this, a synthetic method designated as the entrypoint should be added. This method should convert the incoming arguments, and invoke the real Java main method. Because Java programs may have any number of main methods, it might be necessary for the programmer to specify to the compiler, which one should be the chosen entrypoint.

5.14 Class library

Without the support of the libraries in the *java.* hierarchy, the compiled code cannot really do anything. Since most of the class library is normally written in Java, it would seem the best way to do it. To make this work, support for native calls and a way to call the CLR libraries from a Java program would be necessary. A starting point for the class library would be to base it on the Classpath project [cla].

In the scope of this project, it has been necessary to use the class library from the Microsoft J# compiler. Included with the compiler is a re-implementation of parts of the Java class libraries, unfortunately only JDK1.02.[2]

[2]The beta 1 version, a newer beta supporting JDK1.1.4 was released just before deadline.

5.15 Bootstrap

Using the class library discussed in the previous section, the goal is to be able to bootstrap the compiler on the CLR. Keeping track of the different part of a bootstrap can be a bit confusing. I will use the notation of T-diagrams from [ES70] to illustrate. An example is shown in figure 5.1. In this case it is a compiler from the source language **SL**, to the target language **TL**. The compiler is created in the auxiliary language **XL**. A "machine" running programs in some language **M** is represented by a triangle.

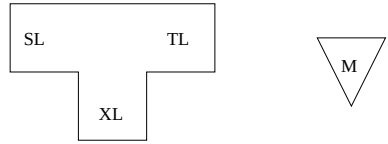

Figure 5.1: T-diagrams example

In the case of pizzacil, the auxiliary language is equal to the source language, namely Pizza. Luckily a pre-compiled binary (for the JVM) is included in the Pizza distribution, which can be used for the first compilation. The process of bootstrapping pizzacil is shown in figure 5.2.

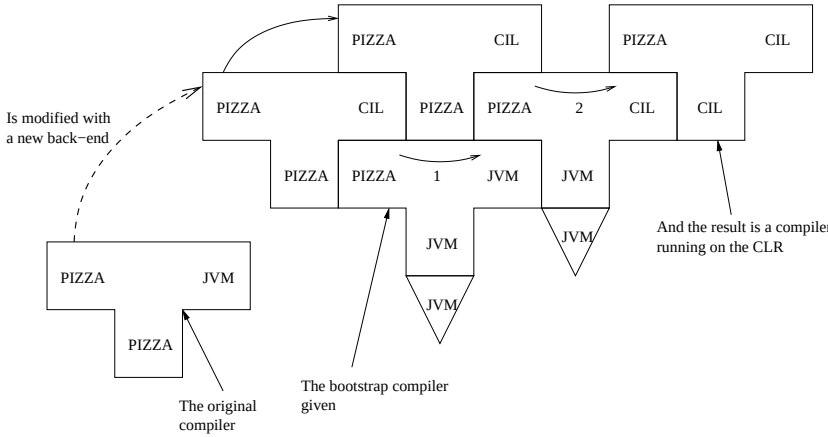

Figure 5.2: Bootstrap of pizzacil on the CLR

The signatures of methods and fields are read from Java class files, and used during the type check. An assembly for the CLR is then created. When the runtime executes this, the library code that gets executed is contained in the J# DLL file. Because these are not synchronized, inconsistencies are possible. The process is shown in figure 5.3.

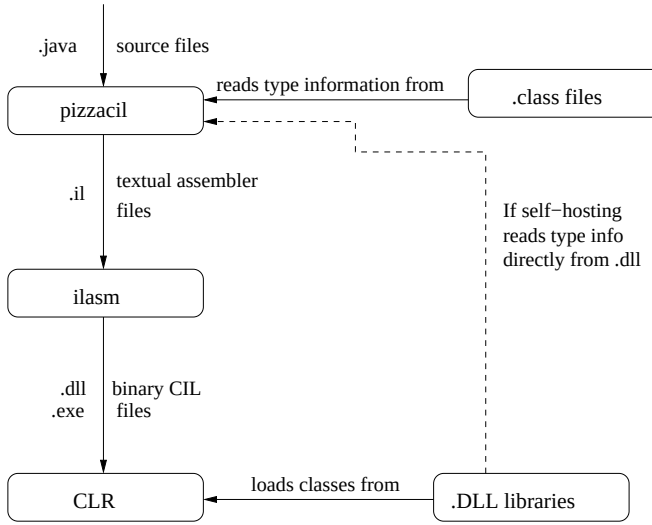

Figure 5.3: Loading classes in Pizza

5.15.1 Self hosting on the CLR

The compiler still needs Java class files for the type check. Apart from a way to actually read the assemblies, some extensions to them are also necessary, if self hosting is desired.

Custom attributes

There is no way in the CLR to specify what exceptions a method might throw. All exceptions are “unchecked”. If the compiler are to become independent of class files, there need to be a way to specify exceptions

of methods in assembly files. This can be implemented using *custom attributes*. These are defined as classes that inherit from the *System.Attribute* class, and can be attached to other CIL declarations. So we could defined a class *ExceptionAttribute*[3], to contain information about the exceptions that a method throws.

```
class ExceptionAttribute : System.Attribute {
  private string exc;

  // A "constructor" that takes a string of
  // exception names seperated by commas,
  public ExceptionAttribute(string s ) { ... }
}
```

Objects of this class can then be attached to method declarations, to specify exceptions.

```
.method public static test () cil managed {
  .custom instance void ExceptionAttribute::.ctor(string)
       = ( 42 23 03 ... ) // The string "IOException" in hex binary
...
}
```

This way pizzacil could add exception declarations to methods in CIL. The attributes will be ignored by the CLR, but can be read by pizzacil from the assembly files and used to check exception declarations. Since self-hosting is not going to be attempted, attributes will not be implemented.

5.16 Static initializers

Static initializers or *class constructors* are implicitly created when static variables have initial values assigned.

```
class teststaticinit {
        static int f = 1;
        static double d1 = Math.cos(f);
```

[3]By convention, attribute classes end in Attribute

```
        static {
                System.out.println(
                    "In static initializer ");
        }
}
```

In the above source, the statement in the **static {}** block, and the initializers for the fields **f** and **d1** together, are the static initializer for this class. In the CLR static initializers shall have the name `.cctor`.

The JVM spec defines the semantics of static initializers in section 2.17.4. The static initializer for a class should be called in the following circumstances:

- An instance of the class is created.
- A static method of the class is invoked.
- A static (non-final) field is referenced.

Before an initializer is executed, the superclass must be initialized. The semantics for class initializers in the CLR are specified in [gT01a] p.50. In the CLR it is, for performance reasons, possible per class to relax the timing constraints on when the initializer should run. The strict semantics, using the **beforefieldinit** modifier on a class, will Unlike the JVM, the CLR does *not* automatically invoke static initializers for superclasses.

There are two ways to solve this:

- Adding a synthetic field to the base class and refer to this in the subclass, thereby triggering execution of its initializer.
- Invoke the System.Runtime.CompilerServices.RunClassConstructor method from the .NET core library on the base class.

Of the two, the second is probably the better solution, but it is complicated to use without proper support in pizzacil for loading of CLR assemblies. I will ignore the problem of automatic base class initialization for the time being. It does not seem likely that many programs depend on it.

5.17 Unsolved issues

These are some more exotic parts of the Java language. I will describe the problem, and possible solutions, but actual implementation of these will be deferred, and I will argue that it should not matter for most programs.

5.17.1 Finalizers

In Java a *finalizer* for an object can be defined. A finalizer is like a *destructor* in C++, but because there is no explicit freeing of memory in Java, the exact timing of the invocation of finalizers cannot be depended on.

A finalizer is defined by overriding the virtual method, `void finalize ()`, which is defined in *java.lang.Object* Finalization in the CLR is defined in [gT01a] section 7.9.6.7.

Informally the JVM and CLR behaviors are identically defined as far as the timing of the invocation of finalizers, and the resurrection of objects containing finalizers are concerned. In the proper design of the class library, finalizers would be handled by simply mapping them to `Finalize`. But currently they have had to be ignored.

5.17.2 Volatile variables

Variables in the JVM can be declared *volatile*. This means that certain restrictions are set on how they behave, with regards to coherency between threads. Unfortunately the original specification of the Java memory model was apparently not very well thought out, and is currently being revised [Pug99b].

The current coherency semantics of the JVM `volatile` modifier is *not* equal to the CLR. Also the JVM spec section 8.4 guarantees that 64-bit volatile variables will be accessed atomically, even on architectures with smaller native word size. Volatile accesses in the CLR does not guarantee this.

I will therefore simply ignore volatile modifiers. This should not be a major problem though: According to [LP00] all existing JVM implementations break most of the specification. Only the Solaris JVM supported atomic accesses for 64-bit values, most of them failed to avoid optimization of redundant loads, and none of them ensured sequential consistency. Since Java programs cannot depend on existing JVM implementations, it should not be a problem to ignore it on the CLR.

Chapter 6

Implementation

In this chapter I will give a short overview of the implementation of the back-end. Major parts of the source-code can be found in the addendum to this report.

6.1 General

I decided to modify Pizza "destructively", so as not to complicate the implementation by trying to have both the old and new back-ends coexists in the same source. I did try to isolate the changes enough, that it should be relatively easy to re-factor the implementation, and integrate it together with the old back-end.

The core architecture of both the CLR and the JVM are quite similar, most of all they are both based on a stack architecture, so laziness dictated that I based my implementation on the old back-end, instead of reinventing the wheel.

I will try here to give an overview of the classes that make up the back-end. Instructions on how to build and use the compiler can be found in appendix B. A full elaboration on how to translate the individual AST nodes is not included, as this would become too repetitive, and code generation for stack machines is well known.

6.2 Structure of the Pizza compiler

A full overview of the source tree is given in appendix B. The main parts of the compiler resides in the `net/sf/pizzacompiler/compiler` directory. The auxiliary classes for the Pizza to Java translation is located in `pizza/support`.

The front-end of the compiler is the *Main* class. It parses commandline parameters and drives the rest of the compiler. This happens in the `process` method. It performs the following steps:

1. Parse each source file given on the commandline into an abstract syntax tree, of type *AST*. Each file is represented by a separate tree, anchored at a *TopLevel* node.
2. Add all classes referenced in **import** statements, and all defined classes to the symbol table.
3. Perform attribution on the tree and fold constants.
4. If the source files use any of the Pizza extension, transform them into Java. First any algebraic datatypes, then closures and finally any generic classes.
5. Either output the resulting AST as Java source, or:
6. Call the back-end on each *TopLevel* node. The main entrypoint for the back-end is the method `generate` in *CILCodeGen*.

6.3 The back-end

The new back-end consists of these classes:

- AssemblyWriter
- CILBasic
- CILCode
- CILCodeGen
- CILConstants
- CILGen
- CILItem
- Metadata
- Modifiers

I will just give a short explanation of what each class does. There are no deep architectural issues or smart class hierarchies that need to be explained.

6.3.1 AssemblyWriter

This class contains methods for writing the generated code into a file. The code generator emits binary code into an array, which is then disassembled back into CIL assembler in text format by this class. The idea was, that this class should later be replaced by one that generates executables directly.

6.3.2 CILBasic

This class contains tables of the instructions, their mnemonics, how they influence the stack and how bytes of arguments follows each instruction. Some of this have been extracted from the file `opcode.def`, which is included in the .NET SDK.

6.3.3 CILCode

This class is an abstraction of the body for one method. It contains the following information:

- The maximal stack height.
- Whether this method is the program entrypoint.
- The exception table of the method.
- The number, and types, of all local variables.
- The method body as an array of bytes.

6.3.4 CILCodeGen

This is the main part of the back-end, and contains methods that traverse the AST, emitting CIL instructions.

6.3.5 CILGen

This class contains auxiliary methods that are used by the code generator. It contain methods for:

- Local variable allocation and reuse.
- Type coercion.
- Back-patching of branch offsets.
- Definite assignment analysis.

6.3.6 CILItem

A *CILItem* is a generalized placeholder for the result of code generated for any expression. This way the difference between, for example, a value on the evaluation stack, a local variable or a static field is abstracted, during code generation.

6.3.7 MetaData

This class contain a hashtable of the symbols that need to go into the metadata part of an assembly. Every time a symbol is used during code generation, it is inserted into the metadata, and a *token* value is returned as reference.

6.3.8 Other

To integrate the new back-end into the compiler, some of the other parts had to be modified, a short summary is given here:

ClassReader When reading classes assign them to an assembly.

ConstantFolder Folding of constants rely on the arithmetic opcodes, and therefore needed to be changed to support the operators of the CLR.

Main Some commandline arguments have been added.

Symbol Additional information about assembly membership need to be carried in the symbols for classes.

Symtab Symbols representing operators in the AST contain the opcode itself.

6.4 Bootstrap

After the initial implementation I proceeded with the attempt to successfully bootstrap the compiler using the class library from J#. Modifications had to be made to accommodate the limitations of the library.

During code generation floating point constants are converted into a byte stream using methods from *java.lang.Double*. In the J# class library only the `doubleToLongBits` and `floatToIntBits` methods are available. These methods collapse different NaN values into the same bit pattern, but unfortunately the accurate methods `doubleToRawLongBits` and `floatToRawIntBits`, were not available in JDK1.02.

Some classes in the compiler used *java.util.Map* and *java.util.Set* and others used methods on *StringBuffer* and *Vector* that were all introduced recently. Workarounds were made to remove these dependencies on JDK 1.2.

The J# libraries has no support for jar or zip files. To support class loading *ClassReader* was modified to not depend on the two classes *ZippedFile* and *ZipFileDirectory* that had to be removed. As a result the jar file with the class library needs to be un-archived first for pizzacil to work.

Chapter 7

Tests

In this chapter I will argue the correctness of the back-end. I will then show the results of some benchmarks.

7.1 Correctness

It is necessary to somehow validate that the generated code, running on the CLR, implements the correct Pizza semantics. I have taken three steps in doing this:

1. Verify that the back-end generates valid and verifiable CIL.
2. Show that the compiler passes the bootstrap test.
3. Perform Java compatibility tests.

7.1.1 Verifiable CIL

As shown in figure 3.2 it is possible to create sequences of CIL, that although allowed by the syntax, are invalid. It is also possible to generate code that is unverifiable, even though it is type-safe. The CLR *can* accept code that is unverifiable, subject to user settable security settings. When translating Pizza, it *should* be possible to only generate code that is both type-safe and verifiable, because no unsafe operations are allowed by the

source language. To verify the generated code, the `peverify` tool from the .NET SDK is very useful. During development verification errors was a great help in pinpointing bugs in the back-end.

7.1.2 Verification of pizzacil

The generated code is valid, but not verifiable. The problem is a combination of two: Exception handling and the choice of not mapping *Object* to a CLR type.

Exceptions

Because the class library has defined *java.lang.Exception* as a subclass of the CLR type *System.Exception*, it is no longer a subclass of *java.lang.Object*, as a Java program would expect, and unverifiable code is the result. The following example is taken from `readClassFile` in class *ClassReader*:

```
try {
   ..
   // Something that might throw an exception
   }
catch (RuntimeException ex) {
 throw new IOException("bad class file (" + ex + ")");
}
```

Constructing the *IOException* implicitly calls the static mathod `valueOf` on the exception object *ex* to convert it into a string.

Looking at a fragment of the generated CIL for the exception handler, the problem shows up.

```
      .locals ( ...
                class [BJLIB]java.lang.Exception V_5
                ... )
        ...
IL_146: ldstr     "bad class file ("
...
IL_155: ldloc.s 'V_5'
IL_157: call      class [BJLIB]java.lang.String
                        [BJLIB]java.lang.String::valueOf
```

(class [BJLIB]java.lang.Object)

At hex-offset 157 is the call to the `valueOf` method, which takes an argument of type *java.lang.Object*. The object on the stack is not compatible with the method argument, so this is not verifiable to be type-safe.

In praxis it works correctly. That is because the implementation of `valueOf` in the class library checks the runtime type of its argument, and treats instances of *System.Exception* as a special case.

7.2 Bootstrap test

As shown in figure 5.2 (and repeated in figure 7.1), the compiler can be bootstrapped using the compiler included with the original Pizza source. This is the stage 1 compiler. We now have a compiler generating CIL, running on the JVM. In stage 2 this compiler is used to compile itself, yielding a CIL generating compiler, in CIL.

This can be taken a step further as shown in figure 7.1. The stage 2 compiler running on the CLR is, in the 3rd stage, used to compile itself once again.

The compilers generated in stage 2 and 3 are then compared. This is known as the bootstrap, or "strong compiler" test from [Wir77]. It can be shown that if a compiler is correct and deterministic, then the bootstrap test will succeed, ie. the two compilers generated in stage 2 and 3 will be identical [Goe99].

In practice the test is performed simply by comparing the generated assembler code for stage 2 and 3 using the `diff` tool. The value of being able to use tools like `peverify` to statically type-check the generated code showed its value. When the compiler passed the verifier, modulo the exception shown in the previous section, it was also able to pass the bootstrap test.

That the compiler passes the bootstrap test only proves that it generates correct code in the test case of the compiler. This is a very practical "sanity" test, and since the compiler itself is a large program, also quite covering.

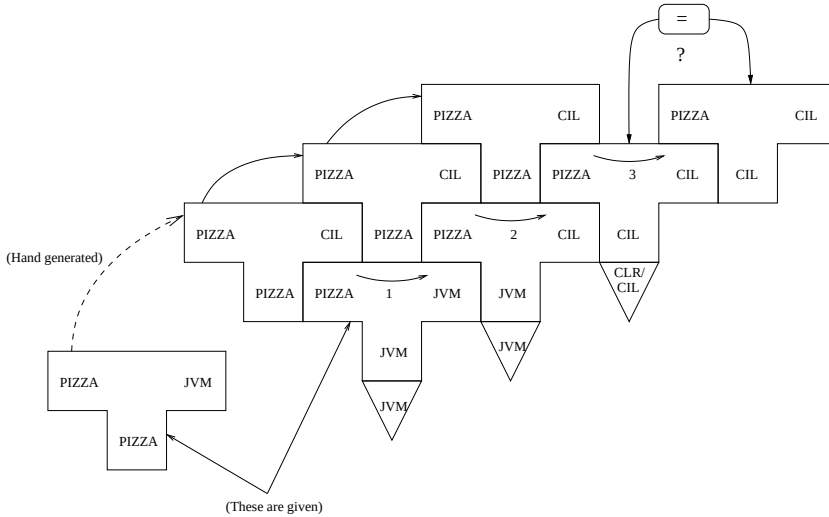

Figure 7.1: Performing the bootstrap test

7.3 Test suite

Apart from the compiler I have been able to successfully compile and run applets from the JDK1.0 distribution, many benchmarks (Linpack, JavaWorld, JavaGrande) and other older programs that did not depend on newer class libraries, the Java VNC viewer among others [vnc].

I wrote some small tests during development of the back-end, to figure out specific problems in code-generation. This meant they had to be small enough that I was able to manually check the generated code, but this also means that they are not very exhaustive. They can be found in the `tests/` directory in the source package.

A comprehensive test suite would be needed to properly test every corner of the back-end, and especially library, because there are still many things not covered. Remote Method Invocation, sockets and Reflection to mention some. Sun has a full test suite that they use to certify Java implementations, the Java Compatability Kit. Unfortunately it is not available for free, and so could not be used to test the compiler in this project.

A free alternative exists in Mauve [mau]. This test suite has a number of

tests for the Java class libraries. Most of the tests are written for JDK1.1, and so only very few of them could be used, none of which tested anything particular interesting. Some more test cases should be written to better test the back-end.

7.4 Performance tests

The Java tests has been run using the Sun Java Development Kit version 1.3.1_01 for Linux. All performance tests were run with the "-server" VM for maximal speed. The CLR tests were run using the .NET beta 2 SDK running on Windows 2000 Professional, using the Java libraries from J# beta 1. The tests were performed on a Pentium III 850MHz PC.

I have made performance tests to evaluate the compiler. Several factors, which all influence the results, come into play:

- The generated code
- The runtime
- The libraries

As the .NET runtime and libraries used for the tests were beta versions, no hard conclusions should be made from the results.

7.4.1 Floating point performance

For floating point performace measurement I used the Linpack benchmark from `http://www.netlib.org/benchmark/linpackjava/`. The results are shown in table 7.1.

Description	mflops/sec 1st run	mflops/sec avg.
javac / JVM	34.5	37.2
javac -o / JVM	34.7	37.9
pizza / JVM	32.8	35.0
pizzacil / CLR	33.5	35.9

Table 7.1: Linpack benchmark

The first two rows shows the benchmark compiled using the Sun compiler, with and without optimization, running on the JVM .There is not much difference between the results from Javac with or without optimization turned on. The third row shows the Pizza compiled benchmark running on JVM. Comparing the benchmark between Javac and Pizza, the code generated by Pizza is slightly slower when run on the JVM. This is due to the difference in optimization.

The difference between the first run, and the averaged results, are a result of the Sun "HotSpot" virtual machine that performs optimization depending on which parts of a program is executed often. After running the benchmark a couple of times, all of the methods involved will have been compiled.

What is perhaps more surprising is that the CLR actually outperforms the JVM slightly on the code generated by pizzacil.

7.4.2 Bootstrap

The above micro benchmark only shows floating-point performance. Timing the compiler bootstrap is a good overall benchmark. Table 7.2 shows average timings for compiling the compiler itself.

Description	JVM	CLR	Difference
Avg. compile time	45.5s	64.2s	41%
Avg. max footprint	27172k	24820k	-8 %

Table 7.2: Bootstrap benchmark

The compilation on the CLR uses about 40% more time for the bootstrap, which is not too bad considering that both the runtime and the class libraries are products still in beta, and that the back-end generates sub-optimal code for **switch** statements.

7.5 Code size

Code size is important. One of the major applications, touted by both Java and .NET, is dynamic downloading of programs across networks. For

mobile low bandwidth applications, or for embedded devices, code size can be an important factor.

In table 7.3 I have collected data on size for a few test cases: The compiler itself, the JavaWorld benchmark [jwb] and the VNC viewer.

There are 4 relevant sizes to look at:

jar This is the standard distribution format for JVM code. Each file is compressed individually before being archived together.

exe The standard file format for CLR files in *Portable Executable* (PE) format.

j0r.gz The jar file is created with -0 (only archiving) before being compressed with GNU zip. This is the smallest possible size of the JVM executable, if special schemes for class compression are not considered.

exe.gz The CLR executable file compressed with GNU zip.

		pizzacil	JWBench	VNC
Executable	jar	447k	48k	24k
	exe	502k	66k	35k
Compressed	j0r.gz	360k	28k	21k
	exe.gz	179k	12k	14k

Table 7.3: Code sizes of resulting executables in kilobytes.

Looking at the sizes of the files, comparing the first two rows, the JVM jar files are slightly smaller than the CLR executables. Actually the CLR has a slight disadvantage, because I have not implemented use of short branches, and the code generated for **switch** statements takes up more space than necessary.

The compiled code of the compiler contains roughly 10000 branches, and most of them would only needs a short offset, which would save 3 bytes per branch, for a total of about 30 kilobytes. This would bring the size down on par with the jar file. Although relevant in the context of disk space, the comparison is skewed because the jar archives are compressed, while the PE executables are not.

In [Pug99a] it is shown that the actual byte-code only takes up around 20% of a class file. The rest is used by the constant pool, with most of

it being string entries. The problem with JVM class files are, that all external references are symbolic, so that every class file constant pool will contain many strings representing class names and methods. Most class files will contain references to all the basic Java classes, but references to other classes in the same package, also results in symbolic string references being created. The metadata in the PE files on the other hand, only has to contain these external references once, because all internal references are made using tokens. This results in large savings in the size of the executable.

The last two rows compare the sizes for compressed executables, which is relevant in regards to transmission at least. Even when the jar files are only archived and then compressed, which results in about 20% reduction in size, the .NET CLR executables are still less than half the size of the JVM jar files.

7.6 Local variable optimization

In the first prototype implementation of the back-end, I had not implemented reuse of local variable slots. Later this was done, and I was curious to see whether this had any impact on performance or memory usage.

Altogether the compiler consist of 2290 methods. Compiled only 1004 have any local variables. Counting the locals in these methods yields:

	Maximum # of locals	Average # locals	Executable size
No reuse	158	5.1	502272 bytes
Reuse	116	4.4	500736 bytes

Table 7.4: Reuse of local variable slots.

Obviously the reuse logic did result in less locals being allocated. It only reduced the size of the executable about 2 kilobytes though, a small fraction.

Performance wise it made no difference. The footprint of the compiler, when compiling itself, in both cases was about 25 megabytes, and no difference in speed could be detected. This confirms that the CLR does a good job on the liveness analysis and register allocation for locals.

Chapter 8

Status

This chapter contains an evaluation of the CLR as target for Pizza. Then an overview of possible future work on the compiler, and ideas for some related projects.

8.1 Evaluation of the CLR

Designing and implementing the new back-end has shown some interesting properties of the .NET Common Language Runtime, compared to the Java Virtual Machine. Some has meant more work is placed on the compiler writer, but others has made simplifications possible.

The CLR lacks some of the high-level instructions of the JVM. No doubt, this is due to a design decision of not supporting (or at least, not worrying about) interpretation. While high-level instructions are good for interpretation, because less instructions means less wasted time on decoding them, they can be a major source of problems when efficient translation to native code is desired. On the other hand, lower-level instructions mean more work for the source compiler, but arguably this can be defended because performance is usually not as big an issue there.

Switch statements

The lack of an instruction to directly implement a lookup table, like the JVM has in `lookupswitch`, complicates things if sparse ranges are present in **switch** instructions.

Multidimensional arrays

The JVM directly supports both single- and multidimensional arrays, while the CLR only supports single-dimensional arrays. To implement the multi-dimensional Java arrays in the CLR, the back-end has to emit code inline for the allocation, which is made trickier by the fact that array sizes are not compile time constants.

Stack manipulation

Operations for esoteric stack manipulation, like the JVM `dup2`, `dup_x2` and `swap`, do not have counterparts in the CLR. As shown in section 5.8, the stack operations can be used to translate some constructs elegantly. The problem is that, although not a problem for an interpreter, stack manipulation impede the data-flow analysis in a JIT compiler, and can lead to generation of poor native code. The IBM JVM have to use a heuristic of commonly occurring stack manipulation idioms to avoid inefficient code [S+00].

Conditional branches

The CLR conditional branch instructions work on all types, which simplified generation of conditional expressions, because there is no need to distinguish between the case of boolean and integer comparisons, and comparisons on other types.

Polymorphic instructions

Because the arithmetic instructions are polymorphic, there is no need to emit different instructions depending on the type of the operands.

Unified type sizes

One of the most beneficial improvements is the removal of the distinction between 32- and 64-bit basic types. The JVM back-end have to jump through hoops when referencing values on the stack, in local variables or in the constant pool because 64-bit types always takes up two slots. Avoiding this saved much grief in the new back-end.

Exceptions

In the CLR try-finally handler blocks are treated in the same way as try-catch handlers. Although the restrictions on entering and leaving protected- and handler-blocks necessitate some extra work, it is better than the JVM way of having to make the compiler emit subroutine calls to the finally handler, at all points where the try-block is exited. These subroutine calls, using the `jsr` byte-code in the JVM, also makes verification harder.

8.1.1 Portability of the CLR

Unlike the JVM which has been implemented on almost any possible platform, the CLR currently only exists on one: Windows/x86. There are projects working on implementing the ECMA specifications on other platforms, but it remains to be seen how portable the CLR really is. During an early part of the project I looked at translation of CIL to ANDF. This brought up some issues:

- Explicit layout of classes. The CLR allows the exact memory layout of fields in a class to be specified using `.pack` and `.size` directives. This can be used, for example, when translating ANSI C structs and unions. This is not portable, for example some hardware platforms restricts alignment of data.
- It seems more emphasis has been placed on being able to generate efficient code for the Intel architecture, than on being portable. Both with floating-point, but also other examples like integer division, which is defined to throw an extra exception on x86.
- In the CLR, stepping outside the sand-box environment, and invoking functions in the underlying operating system, can be done directly using the *PInvoke* construct. When using native methods in Java,

one first has to create a dynamic library with "glue" stubs. The easy access, outside the environment of the virtual machine, could tempt people to create unportable programs.
- Not all parts of the class library has been submitted for ECMA standardization, for example the graphical user interface libraries are not part of the standardized set.

8.2 Further work

Some things are missing from the implementation. There are also areas where the back-end can be improved upon, or better optimized.

Exceptions

Proper support for exceptions thrown by the runtime is not fully implemented. The J# library uses *filter* handlers to translate CLR exceptions to JVM equivalents, and I did not complete code generation for this. This means, that currently a program will not be able to successfully catch exceptions generated internally by the runtime[1]

Branches

The back-end does not perform compaction of branches, it only uses the long form. Since the generated code is stored in a buffer before being written to a file, this should not be too hard to solve.

Debug information

Support for symbolic debugging is missing. The ECMA standards does not define how to include debug information, so one would have to rely on how the Microsoft .NET SDK implements it.

[1] Like **ArithmeticException** or **NullPointerException**

Switch translation

The new back-end emits naive code when translating switch statements with sparse ranges. This should be improved using the algorithm from [Gou02].

Array allocation

The way code generation for array allocation is implemented, the backward branch constraint[2] does no longer hold at all times, but the generated code is still verifiable.

Strictfp

An implementation of the **strictfp** modifier, using explicit use of truncating conversion instructions is missing.

Native methods

Support for native methods should be implemented. This will be necessary when implementing the Java class libraries, for methods that need to co-operate with the runtime, like implementation of reflection and threading.

8.3 Pizza language related

Some areas related to implementation of the Pizza extensions have not been fully explored. Some of the extra features the CLR supports, compared to the JVM, could perhaps be used for easier, or more efficient translation.

8.3.1 Tail-calls

Transformation of tail-calls could be replaced with an implementation using the CIL `tail.` prefix instruction, which turns a normal method invocation

[2] [gT01c] 1.8.2: The stack must be empty at a backward branch

using `call`, into a tail-call. Tail-calls are represented as *Goto* nodes in the AST before being transformed. An better implementation of tail-calls would need to:

- Remove the transformation done in the *TransClosures* class.
- Implement code generation for the *Goto* node in the back-end.

8.3.2 Boxing of basic types

The current method of implementing instantiation of generic classes with basic types involves much overhead, because of boxing within reference types. It would be interesting to see whether using the CLR primitives, `box` and `unbox`, could result in any noticeable difference.

8.3.3 Generic CLR

An extended version of the CLR runtime implementing generics has been created by Andrew Kennedy and Don Syme [KS01]. Compared to Pizza they have had the advantage of being able to change the behavior of the runtime, while Pizza pays a premium for backwards compatibility. Their implementation uses both specialization and sharing, which means:

- That instantiation with basic types is efficient, the generic class is specialized with the required type, no boxing operations are made.
- That instantiation with all reference types will share the implementation, meaning no code bloat.

Informally comparing what can be read from the released article, with the demands of Pizza, it seems it would be possible to map Pizza directly onto the Generic CLR.

Targeting it, the parts of pizzacil that performs the type erasure should be bypassed.

8.3.4 ILX

The current version of the CLR does not, apart from tail-calls, directly support any of the other Pizza concepts.

A project within Microsoft has created an extended CIL, called ILX, which contains direct support for higher-order functions, algebraic datatypes and generics [Sym01]. The actual implementation transforms ILX into regular CIL, much in the same way as Pizza does, just from a lower level. A design of how to use the type unsafe CIL instructions `ldftn` and `calli` to implement first-class functions efficiently is sketched. Trying this out for Pizza could be attempted.

8.4 Further projects

8.4.1 Assembly toolkit

A library for CIL assembly file manipulation is needed. The code generator outputs binary CIL code already, which the current implementation then spends a lot of unnecessary time on converting into text for input to `ilasm`. The .NET *Reflection* classes could perhaps be used as basis for this.

8.4.2 Java class library re-implementation

A full free re-implementation of the Java2 class library is needed. Most of the library should be implemented in Java, with use of classes from the .NET class library to implement the necessary interface to the virtual machine.

Chapter 9

Conclusion

9.1 Status

I have designed and implemented a new back-end, that targets the .NET Common Language Runtime for the Pizza compiler.

- Enough of the dynamic semantics of the Java language has been successfully mapped to the CLR that the compiler was able to bootstrap and pass the bootstrap test.
- A number of smaller benchmark programs and applets was also successfully translated, but attempts to translate some larger programs stranded, due to the lack of library support more recent than JDK1.02.
- Support for exceptions thrown by the runtime itself, and some other parts of the Java Language Specification that were not essential, are still missing. Some solutions have been presented, but not implemented yet.
- The generated code is not type safe, due to design choices constrained by the class library, over which I had no control.
- Performance tests showed that the generated code does not perform significantly worse when running on the CLR, and that on floating-point arithmetic the CLR actually outperformed the tested JVM slightly.
- The compiler itself, with the new back-end, runs an order of magnitude slower than the original, and improvements in the generated

code can be made in some places.
- I started out work on the back-end by modification of the existing. This did save some grunt work, since the translation of some AST nodes only needed smaller modifications. In retrospect it might have been better to start from scratch, instead of trying to fit within the old framework.

It is my conclusion that the .NET Common Language Runtime works well as target for Pizza/Java. Apart from some workarounds necessary to preserve the class hierarchy, and some operations that need library support, the CLR can support the dynamic semantics of Java in a fairly straightforward manner.

The back-end need to do more work for some things to generate code, because of the missing high-level instructions, but if this makes better JIT compilation possible, it seems like a good tradeoff.

The choice made by the developers of the J# library to rely on unsafe tricks seems unnecessary and unwise. Even more so because the methods that implement it and can be accessed by any .NET program, not just Java programs. This opens up a major security hole.

The CLR implements a superset of the Java Virtual Machine, which *should* make it a better candidate to be used as an "UNCOL" also for languages that are less like C# than Pizza is. But its limits have not really been stressed in this project.

9.2 The project

The course of my project has been a bit disastrous. When I defined it originally, I did not have a good enough notion of the eventual goal. I did a pretty poor job of project management, and did not early enough figure out the essentials, and decide what to concentrate on. I also did not do a good job of soliciting help from my supervisor in the beginning.

At one point I got side-tracked, spending time looking at ANDF, and trying to figure out whether it could be used as a code generator for CIL. The time spent on this is unfortunately not quite reflected in this document, although it did help me familiarize myself with the CIL instruction set. It was only after scrapping this, that I started work on the Pizza back-end. I also feel a more formal view might have been appropriate on some issues.

9.3 The future

I have been in touch with the maintainers of the Pizza compiler, and they showed enthusiasm for the possibility of getting a new back-end targeting .NET. I hope to work with them to merge this project back into the regular Pizza compiler, so that others may benefit from it, and hopefully improve it.

In spite of the problems I have had during the project, I have learned a lot about code generation for .NET, the Pizza language and Java Virtual Machine. I have also gained a good understanding of compiler construction in general, and the concepts, and implementation of, generics.

Bibliography

[Bun95] Jørgen Bundgaard. An andf based ada 95 compiler system. Technical report, DDC-I A/S, 1995.

[cla] The classpath project. `http://www.gnu.org/software/classpath/classpath.html`.

[DD00] S. Doyon and M. Debbabi. On object initialization in the java bytecode. *Computer Communications*, 23:1594–1605, 2000.

[DER95] DERA. *A Guide to the TDF Specification*, June 1995.

[DF80] J.W. Davidson and C. W. Fraser. "the design and application of a retargetable peephole optimizer". *Transactions on Programming Languages and Systems*, 2(2):191–202, 1980.

[Ert96] M. Anton Ertl. *Implementation of Stack-Based Languages on Register Machines*. PhD thesis, Technische Universität Wien, Austria, 1996.

[ES70] Jay Earley and Howard E. Sturgis. A formalism for translator interactions. *Communications of the ACM*, 13(10):607–617, 1970.

[FM99] Stephen N. Freund and John C. Mitchell. The type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, 1999.

[Fra94] Michael Franz. *Code-Generation On-the-Fly: A Key to Portable Software*. PhD thesis, Swiss Federal Institute of Technology, Zürich, 1994.

[GJ+00] James Gosling, Bill Joy, et al. *The Java (tm) Language Specification, Second Edition*. Addison-Wesley, 2000.

[Goe99] Wolfgang Goerigk. On Trojan Horses in Compiler Implementations. In F. Saglietti and W. Goerigk, editors, *Proc. des Workshops* Sicherheit und Zuverlässigkeit softwarebasierter Systeme,

ISTec Report ISTec-A-367, ISBN 3-00-004872-3, Garching, 1999.

[Gou02] John Gough. *Compiling for the .NET Common Language Runtime (CLR)*. Prentice Hall PTR, 2002.

[gT01a] "ECMA Working group TC39/TG3". "common language infrastructure (cli) partition i: Concepts and architecture". `http://msdn.microsoft.com/net/ecma/`, 2001.

[gT01b] "ECMA Working group TC39/TG3". "common language infrastructure (cli) partition ii: Metadata". `http://msdn.microsoft.com/net/ecma/`, 2001.

[gT01c] "ECMA Working group TC39/TG3". "common language infrastructure (cli) partition iii: Cil instruction set". `http://msdn.microsoft.com/net/ecma/`, 2001.

[gT01d] "ECMA Working group TC39/TG3". "common language infrastructure (cli) partition v: Annexes". `http://msdn.microsoft.com/net/ecma/`, 2001.

[Jav] Generic Java. `http://www.research.avayalabs.com/user/wadler/pizza/gj/`.

[Jon92] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine. *Journal of Functional Programming*, pages 127–202, 1992.

[jwb] Javaworld benchmark. `http://www.javaworld.com/javaworld/jw-09-1998/jw-09-speed.html`.

[KF99] Thomas Kistler and Michael Franz. A tree-based alternative to java byte-codes. *International Journal of Parallel Programming*, 27(1):21–33, 1999.

[Knu] Donald Knuth. Mmix. `http://www-cs-faculty.stanford.edu/~knuth/mmix.html`.

[KS01] Andrew Kennedy and Don Syme. Design and implementation of generics for the .net common language runtime, 2001.

[LP00] Doug Lea and William Pugh. Correct and efficient synchronization of java technology-based threads. Slides, 2000.

[LY97] Tim Lindholm and Frank Yellin. *The Java(tm) Virtual Machine Specification*. Addison-Wesley, 1997.

[mau] The mauve project. `http://sources.redhat.com/mauve/`.

[MD97] Jon Meyer and Troy Downing. *Java Virtual Machine*. O'Reilly Associates, 1997.

[MG01] Erik Meijer and John Gough. Technical overview of he common language runtime. 2001.

[Moo97] James D. Mooney. Bringing portability to the software process. Technical Report TR 97-1, West Virginia University, Dept. of Statistics and Comp.Sci., 1997.

[ORW98] Martin Odersky, Enno Runne, and Philip Wadler. Two ways to bake your pizza - translating parameterised types into java. In *Generic Programming*, pages 114–132, 1998.

[OW97] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, New York (NY), USA, 1997.

[pro] Programming languages for the jvm. `http://flp.cs.tu-berlin.de/~tolk/vmlanguages.html`.

[Pug99a] William Pugh. Compressing java class files. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 247–258, 1999.

[Pug99b] William Pugh. Fixing the java memory model. In *Java Grande*, pages 89–98, 1999.

[S+00] T. Suganuma et al. Overview of the ibm java just-in-time compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[SWT+58] J. Strong, J. Wegstein, A. Tritter, J. Olsztyn, O. Mock, and T. Steel. The problem of programming communication with changing machines: a proposed solution. *Communications of the ACM*, 1(8):12–18, 1958.

[Sym01] Don Syme. Ilx: Extending the .net common il for functional language interoperability, 2001.

[vnc] The vnc java viewer. `http://www.uk.research.att.com/vnc/`.

[Wir77] Niklaus Wirth. *Compilerbau, eine Einführung*. B.G. Teubner, 1977.

Appendix A

Project description

English title: Portability Through Virtual Machines
Advisor: Jørgen Steensgaard-Madsen
Period: 31/10-1/4-2002
Credit: 30 ECTS points
Participant: Morten Sylvest Olsen

English project desciption:
Having to write software that is easily portable to, or runnable on, multiple platforms, is an engineering challenge.

One way of achieving this goal is to use abstract machine representations in lieu of the actual hardware.

.NET is a new Microsoft product which attempts to promote language and platform interoperability, in the same realm as Java. A central part of .NET is the specification of a Common Language Runtime (CLR) and Common Intermediate Language (CIL).

While Java byte-code was originally meant to be used only for the execution of Java programs, CIL is aimed to include support for a larger class, including functional languages and languages that support "unsafe" operations, for example pointer arithmetic.

Another intermediate language is the Register Transfer Language (RTL) which is used internally in the GNU Compiler Collection (GCC) to facilitate

translation from a large number of source languages, to a very large number of target platforms.

The project goal is to compare the characteristics of these intermediate representations with emphasis on platform and language issues.

A possibly unachievable goal will be to find a mapping between some of the languages.

Appendix B

User manual

B.1 Getting the code

The full distribution of the source code can be found at:

`http://www.student.dtu.dk/~c958496/pizzacil.zip`

The code is released under the Artistic License (the Perl license).

The archive only contains the files necessary to build and run the pizzacil compiler, some things has been removed from the original distribution since they were irrelevant.[1]

B.2 Necessary prerequisites

Apart from the Pizza source code, the following tools are necessary to build the compiler.

Microsoft .NET SDK The compiler has only been tested under the Beta 2 version of the SDK. This is no longer available, but the released SDK *should* work.

[1] Some tests and files for the horrible Ant build tool

Microsoft J# SDK Beta 1 To be able to install this, it is necessary to have Visual Studio(.NET) installed. During development I have used Beta 1, a newer version was only just released, and have not been tested.

Sun Java Development Kit Java is necessary for the bootstrap of Pizza. I have only tested with Sun JDK1.3.

Cygwin (make, diff) The makefile has been created using Unix paths. Needs cygwin to run under Windows.

B.3 Structure

The toplevel directory is **pizza/**.

bootstrap/ This directory contains the pizzaself.jar file necessary to bootstrap the compiler.

misc/ The manifest file for building a new jar file.

main/src Toplevel directory for the source. Contains the make file.

- **pizza/support** Support classes used to implement the Pizza language.
- **net/sf/pizzacompiler/**
  - **classlib** Glue classes to interface with the CLR.
  - **compiler** The compiler itself.
  - **contrib** Some contributed generic datatypes.
  - **lang** Generic classes used by the compiler. List, Pair and more.
  - **pizzadoc** A JavaDoc-like preprocessor that understands Pizza.
  - **tests** Tests.
  - **util** Auxiliary generic classes used by the compiler. Set¡A¿, Vector¡A¿ and more.

B.4 Installing Pizza

1. Unpack the zip archive
2. Edit the "prefix" variable in the makefile in **main/src/**
3. Unpack the Java class libraries somewhere, since the pizzacil compiler has no builtin zip support.
4. Make sure that the variables JAR, PIZZA, PIZZACIL, PIZZANET are set correctly. The default values should work in most cases.

B.5 Bootstrapping the compiler

1. First execute "make jar". This should create a new jar file in **main/src/pizza.jar**. This is the CIL generating compiler (stage 1 compiler).
2. Execute "make pizzacil". This should generate a "pizzacil.il" file in main/src using the compiler generated in the previous step. (This is the stage 2 compiler)
3. Run `ilasm` on it. This creates `pizzacil.exe`.
4. Execute "make bootstrap" to let the generated compiler compile itself, and hopefully it will be able to pass the bootstrap test.

B.6 Using the compiler

For a full list of switches, run the compiler without arguments. These are the most important when compiling for .NET:

-o ¡name¿ The desired name of the executable file. Default is "out.il". The name should not include the suffix.

-m ¡classname¿ The name of the class that contains a function of signature `static void main(String[] args)` which should be the entrypoint. If this argument is omitted, the compiler assumes that the first file on the commandline contains the entrypoint.

B.7 Miscellaneous

- When running programs remember that the necessary J# DLL's (bjlib.dll, bjcor.dll, ...) must be either in the current directory, or somewhere else in the CLR search path.
- Pizza does not automatically search for dependencies. Also when compiling for .NET, all necessary classes need to be compiled into one executable. Therefore all source files must be compiled in one go.

Appendix C

Assembler format

This appendix contains a short overview of the assembler syntax for CIL assembler. To describe it I will show fragments of syntax in EBNF. I have simplified the syntax, and left out details that are not essential. For the full syntax, see the standard documents.

An assembly file specifies a number of classes and interfaces. Each of these can contain field, method and nested class definitions. Unlike a Java class file, a CLR assembly can contain more than one class definition.

ASSEMBLY → PROLOGUE {CLASSDECL}

The prologue contains declarations that specify what other assemblies are referenced from within this. It can also contain some version magic, and public-keys used for the security mechanisms. These will not be used in pizzacil though.

C.0.1 Class definitions

The syntax of a class definition is:

```
CLASSDECL → '.class' CLASSATTR ID ['extends' TYPEREF]
                {'implements' TYPEREF} '{' CLASSBODY '}'
CLASSATTR → 'public' | 'interface' |'auto' | ...
```

An interface is declared by using the `interface` keyword. Most of the other class attributes will not be important and can just be ignored as "magic" incantations.

An example class definition could be:

```
.class public auto ansi  beforefieldinit  ErrorMessage
                         extends [BJLIB]java.lang.Object
                         implements [BJLIB]java.lang.Cloneable
{
// class body
}
```

In our case, all classes will extend something, usually *Object*, but not all classes will implement any interfaces.

The class body contains declarations of nested classes, methods and fields. Nested classes will not be used, so we ignore those.

```
CLASSBODY → {FIELDDECL}
          | {METHODDECL}
```

C.0.2 Field definitions

Both classes and interfaces can have fields. A field can either be a static or an instance field.

```
FIELDDECL → '.field' {FIELDATTR} TYPEDECL ID
FIELDATTR → 'private' | 'public' | 'static' | ...
```

C.0.3 Method definitions

```
METHODDECL → '.method' {METHODATTR} RETURNTYPE ID
               '(' PARAMS ')'  {IMPLATTR} '{' METHODBODY '}'
IMPLATTR   → 'cil' | 'managed' | 'synchronized' | ...
METHODATTR → 'public' | 'abstract' |'virtual' | ...
PARAMS     → TYPEDECL ',' PARAMS | TYPEDECL
```

The method body can be empty, if the method is declared abstract.

```
.method public virtual hidebysig instance class
        [BJLIB]java.lang.String getSource () cil managed
{
  // method body
  // follows
}
```

If it is not empty, it needs to declare the maximal size of the evaluation stack used, and the types of the local variables used. Each local variable *can* be named, and the local can be referenced either by its number, or its name. Following these declarations are the instructions that make up the method.

```
METHODBODY → '.maxstack' INTEGER '.locals' '(' LOCALDEFS ')' BODY
LOCALDEFS  → TYPEDECL ID ',' LOCALDEFS | TYPEDECL ID
```

The body of a method consists of lines of instructions. Each instruction can be prefixed with an optional label, and some have an argument.

```
BODY   → [LABEL ':'] INSN
INSN   → 'add' | 'sub' | 'ldc.i4' | ...
```

An example method body:

```
.maxstack 2
.locals (
      class net.sf.pizzacompiler.compiler.ErrorMessage V_0)
IL_0: ldarg.1
IL_1: ldarg.0
IL_2: bne.un    IL_9
IL_7: ldc.i4.1
IL_8: ret
IL_9: ldarg.1
 ...
IL_53: ldc.i4.0
IL_54: ret
```

For an example of an entire assembly with class, field and method definitions, please refer to the CIL example in appendix D.

Appendix D

Codeexample

The following is an example of the compilation of a Java program. The class contains a method to calculate fibonacci numbers.

```
/* A simple class to demonstrate how the code
   looks translated into CIL */
package tests;

public class fibonacci {

        // This is a constant string
        private static String s = "The result is: ";

        // The instance constructor
        public fibonacci () {
                System.out.println("In constructor");
        }

        // An instance method
        private long fib(long n) {
                if (n==0)
                        return 0;
                else if (n==1)
                        return 1;
                else
                        return fib(n-2)+fib(n-1);
```

```
        }

        // The main entry point
        public static void main(String [] args) {
                fibonacci  f = new fibonacci ();

                System.out.println(s+f.fib (10));
        }
}
```

The above source, compiled by pizzacil:

```
// Pizza Java .NET Compiler Version 1.0
// Read artistic.html
.assembly 'fibonacci' {}
.assembly extern BJLIB
    {.publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A )
    .ver 1:0:3227:0}
.assembly extern JAVALIB{}
.subsystem 3
.file alignment 512
.corflags 0x00000001

.namespace tests
{
.class public auto ansi beforefieldinit fibonacci
                    extends [BJLIB]java.lang.Object
{
    .field private static class [BJLIB]java.lang.String s
    .method public specialname rtspecialname
                            hidebysig instance void .ctor ( ) cil managed
    {
        .maxstack 2
        IL_0: ldarg.0
        IL_1: call instance void [BJLIB]java.lang.Object ::.ctor ( )
        IL_6: ldsfld      class [BJLIB]java.io.PrintStream
                                                [BJLIB]java.lang.System ::'out'
        IL_b: ldstr "In constructor"
        IL_10: call class [BJLIB]java.lang.String
                            [JAVALIB]cil.compat.Misc::toJavaString
                            ( class [mscorlib]System.String )
        IL_15: callvirt instance void
```

```
                                [BJLIB]java.io.PrintStream::println
                                ( class [BJLIB]java.lang.String )
    IL_1a: ret

} // end of method

.method private virtual hidebysig instance int64 fib ( int64 ) cil managed
{
    .maxstack 4
    IL_0: ldarg.1
    IL_1: ldc.i4    0
    IL_6: conv.i8
    IL_7: bne.un    IL_13
    IL_c: ldc.i4    0
    IL_11: conv.i8
    IL_12: ret
    IL_13: ldarg.1
    IL_14: ldc.i4   1
    IL_19: conv.i8
    IL_1a: bne.un   IL_26
    IL_1f: ldc.i4   1
    IL_24: conv.i8
    IL_25: ret
    IL_26: ldarg.0
    IL_27: ldarg.1
    IL_28: ldc.i4   2
    IL_2d: conv.i8
    IL_2e: sub
    IL_2f: call instance int64 tests.fibonacci :: fib ( int64 )
    IL_34: ldarg.0
    IL_35: ldarg.1
    IL_36: ldc.i4   1
    IL_3b: conv.i8
    IL_3c: sub
    IL_3d: call instance int64 tests.fibonacci :: fib ( int64 )
    IL_42: add
    IL_43: ret

} // end of method

.method public static hidebysig void
                main ( class [BJLIB]java.lang.String [] ) cil managed
{
```

```
    .maxstack 4
    .locals (
        class tests.fibonacci V_0)
    IL_0: newobj instance void tests.fibonacci ::.ctor ( )
    IL_5: stloc.0
    IL_6: ldsfld class [BJLIB]java.io.PrintStream
                 [BJLIB]java.lang.System::'out'
    IL_b: ldsfld class [BJLIB]java.lang.String tests.fibonacci::s
    IL_10: call class [BJLIB]java.lang.String
                 [BJLIB]java.lang.String::valueOf
                 ( class [BJLIB]java.lang.Object )
    IL_15: ldloc.0
    IL_16: ldc.i4 10
    IL_1b: conv.i8
    IL_1c: call instance int64 tests.fibonacci::fib ( int64 )
    IL_21: call class [BJLIB]java.lang.String
                 [BJLIB]java.lang.String::valueOf ( int64 )
    IL_26: callvirt instance class [BJLIB]java.lang.String
                                   [BJLIB]java.lang.String::concat
                                   ( class [BJLIB]java.lang.String )
    IL_2b: callvirt instance void
                 [BJLIB]java.io.PrintStream::println
                 ( class [BJLIB]java.lang.String )
    IL_30: ret

} // end of method

.method assembly static hidebysig void .cctor ( ) cil managed
{
    .maxstack 1
    IL_0: ldstr "The result is: "
    IL_5: call class [BJLIB]java.lang.String
                 [JAVALIB]cil.compat.Misc::toJavaString
                 ( class [mscorlib]System.String )
    IL_a: stsfld class [BJLIB]java.lang.String tests.fibonacci::s
    IL_f: ret

} // end of method

.method public static hidebysig void
                $main ( class [mscorlib]System.String [] ) cil managed
{
    .entrypoint
```

```
        .maxstack 1
        IL_0: ldarg.0
        IL_1: call    class [BJLIB]java.lang.String []
                            [JAVALIB]cil.compat.Misc::toJavaStringA
                            ( class [mscorlib]System.String [] )
        IL_6: call    void tests.fibonacci::main
                           ( class [BJLIB]java.lang.String [] )
        IL_b: ret

    } // end of method

} // end of class fibonacci
} // end of namespace
```

Appendix E

Bugs in Pizza

During the work on the pizzacil back-end, I discovered several bugs in the other parts of the compiler. None of these bugs are major, although some of them makes Pizza violate the Java Language Specification [GJ+00]. Two of them have later been corrected after I submitted bug reports, but they still exists in the code I have based my work on.

Back-end

I discovered two minor bugs in the original JVM bytecode generator. Method `incr` in class *LocalItem* is called to increment/decrement a local variable. If the local variable is not of integer type invalid code is generated because no coercion is done before emitting an `iadd` instruction, which expects integers as arguments.

The compiler cannot bootstrap itself with debugging turned on. At least one class, TransPatterns, is miscompiled. The problem lies in the generation of LocalVariable debug attributes. I did not bother to research this further, since it was possible to work around.

Parsing

The parser rejects code that includes class literals. For example the code `int[].class`, which is a class literal, of type *Class*, for an array of integers

is not accepted. The code is legal according to the JLS section 15.8.2.

Flow analysis

According to the JLS section 14.20, a java compiler should carry out flow analysis and make sure that all statements are reachable. If two catch clauses following a protected region catches the same exception, only the first will be reachable. Pizza does not reject such code, which is clearly wrong.

Type check

Arrays should implement both the Cloneable and Serializable interfaces, but in Pizza they do not. This is a bug in the symbol table initialization.