Mnemonics: Type-safe Bytecode Combination in Scala

Diploma Thesis
of

Johannes Rudolph

1. März 2009

Prof. Dr. Peter Thiemann
Arbeitsbereich Programmiersprachen
Institut für Informatik
Fakultät für Angewandte Wissenschaften
Albert-Ludwigs-Universität Freiburg

Revision history

- 27. Februar 2009: Submitted revision

- 1. März 2009: Restored accidentally deleted text from section 3.4

ERKLÄRUNG

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Ort, Datum	Unterschrift

Abstract

Writing a compiler targetting the Java Virtual Machine (JVM) requires the generation of Java bytecode. Java bytecode is the instruction set of the JVM, that is used in the binary representation of a Java program, the class-files. Many libraries for generating class-files exist today. They often provide an API to construct and change class-files on-the-fly. To ensure the well-behaviour of so generated code, the JVM imposes several constraints upon class-files. When a class-file is loaded, these constraints are checked by the JVM verifier. A bytecode generation library ensures by definition, that class-files are serialized correctly. However, invalid sequences of bytecode instructions can still be generated. Class-files containing illegal bytecode sequences are rejected by the JVM verifier.

We propose *Mnemonics*, a novel bytecode generation library, written in and for Scala, that allows only representations of valid bytecode sequences to be expressed in terms of the library, invalid sequences are rejected by the Scala compiler. For this purpose, instructions are modelled in Scala's type-system as functions over the types of values on the stack and in local variables. A sequence of bytecode instructions can be created by combining matching instruction primitives, while combinations of incompatible instructions are ruled out by the type-system of Scala.

Zusammenfassung

Ein Compiler, der für die Java Virtual Machine (JVM) übersetzt, muss Java Bytecode generieren können. Der Java Bytecode ist der Befehlssatz der JVM, der in den Klassendateien verwendet wird, der binären Form eines Java Programmes. Es gibt heutzutage einige Bibliotheken, um Klassendateien zu erzeugen. Diese haben häufig eine API, mit der Klassendateien zur Laufzeit angelegt und verändert werden können. Um sicherzustellen, dass sich so erzeugte Klassendateien wohl verhalten, fordert die JVM, dass Klassendateien diverse Beschränkungen einhalten. Wenn eine Klasse geladen wird, werden diese Beschränkungen vom Java verifier überprüft. Eine Bibliothek zur Erzeugung von Klassendateien sollte per Definition korrekt serialisierte Klassendateien erzeugen. Davon abgesehen, kann eine Folge von Bytecode-Anweisungen dennoch ungültig sein. Eine solche Klassendatei mit einer ungültigen Folge von Anweisungen wird vom JVM verifier zurückgewiesen.

Wir präsentieren hier deshalb *Mnemonics*, eine neuartige Bibliothek zur Generierung von Java Bytecodes, die in Scala und für Scala programmiert wurde. In ihr können überhaupt nur gültige Anweisungsfolgen ausgedrückt werden. Ungültige Folgen hingegen, werden schon vom Scala Compiler zurückgewiesen. Um dies zu erreichen, werden Anweisungen als Funktionen über die Typen der Werte auf dem Stack und in lokalen Variablen beschrieben. Eine Folge von Anweisungen kann erzeugt werden, indem passende Anweisungen kombiniert werden. Ungültige Kombinationen hingegen können durch das Typsystem von Scala von vornherein ausgeschlossen werden.

Contents

Listings

List of Figures

Chapter 1

Introduction

Since large programs grow from small ones, it is crucial that we develop an arsenal of standard program structures of whose correctness we have become sure – we call them idioms – and learn to combine them into larger structures using organizational techniques of proven value.
Alan J. Perlis in his foreword to SICP

Programming languages are often classified by the means of abstraction they offer to the programmer into high- or low-level languages.

So are object-oriented programming languages higher rated than structured languages because of the higher level of encapsulation achievable; so are functional languages deemed more expressive for their support of function values, their high-level control abstractions, higher-order functions, and even higher-kinded types; so is a domain specific language prefered over a general-purpose language when it comes to its specific domain; so is assembler favoured over machine code because cryptic mnemonics still seem better than the plain bits and bytes, that the CPU may comprehend in the first place.

Higher-level constructs come at a price, though. Abstracting means hiding basic building blocks behind an interface. The reduction of complexity achieved often comes along with added layers of indirection. Hence, programs are needed which transform high-level code into low-level code: Compilers.

A popular platform, today's compilers target, is the Java Virtual Machine (JVM)[25]. Targetting the JVM usually means generating binary class-files; they are the currency of code in the JVM. The instruction set of the JVM is called Java bytecode. Generating

Java bytecode is a common business in the Java universe of today. There are plenty of libraries which output class-files[6, 7, 1], and though, generating Java bytecode remains a tenacious task prone to errors, always creating both, fear of rejection by the Java verifier and hope for its approval.

We developed yet another library to aid compiler writers targetting the JVM in generating Java bytecode. It is called *Mnemonics* (MNEMO) and written in Scala. Like other bytecode generation libraries, it provides the programmer with means to specify the contents of a class-file and especially bytecode instructions in terms of a higher-level language. While using a generation library always produces correctly serialized class-files, there would still remain the danger of semantic errors caused by instructions linked together in a nonsensical way. Nonsensical, because bytecode instructions are tried to be applied to missing data or data of the wrong type. These errors are then catched by the Java verifier when loading a class.

Would it not be desirable to prevent such errors first of all? What, if a bytecode generation library would it not even allow to express illegal sequences of bytecodes? It is that, what MNEMO tries to achieve with the support of the Scala type system. The following steps are proposed in order to achieve this goal:

- The relevant elements concerning the well-formedness of the sequence of instructions, the *types* of values on the stack and in the local variable array, are encoded at the type-level of the generator. The type configuration at each point in the to-be-generated program is so encoded in a *frame state type*.
- Instructions, accordingly, are then defined as transformations of the frame state. Both, the expected input state type, such as the fact, that two integers must be on the stack to execute an integer addition instruction, as well as, the output state, such as an integer addition instruction leaving one integer on stack, is expressed explicitly in the type of each instruction.
- Several instruction primitives may subsequently be combined to form a bytecode sequence if and only if the output frame state type of the previous instruction is compatible with the expected input frame state type of the next instruction.

The challenge is therefore two-fold: First, an appropriate encoding for a *list of types* of arbitrary length has to be found. Then, instructions have to be defined with regard to this type. Whereas the above cited integer addition instruction has a fairly static type, and so, is easy to define, several other instructions, that are parametric in their input and output types or that depend on other information, such as the ones for method invocations, local variable access, or branching, are much more difficult to specify.

We developed the above sketched framework and implemented prototypes of instructions of all kinds. The implementation uses many features of the Scala type system, some of them in the latest version of the Scala compiler still experimental, to express the types of these instructions. These features are:

- The type inferer; having to annotate types throughout the code would not be feasible at all.
- An implementation of a heterogeneous list type in Scala as suggested in [13, 19, 17].
- Using type-based numerals to access type list elements by index[18].
- `scala.reflect.Code`, which defines a way to let the compiler insert an abstract syntax tree (AST) of a code block representing an expression of a particular type.

Finally, the so created means of specifying instruction sequences are embedded in a minimal framework, that only allows the on-the-fly generation of classes with a single method. To test the feasibility of even this minimal framework for real-world applications, we present the implementation of a small compiler for a custom domain specific language (DSL).

Chapter 2

Background

2.1 Java Virtual Machine

The Java Virtual Machine (JVM) is an abstract machine model which constitutes the base of the Java platform [25]. It is stack-based and has a custom instruction set to operate on the data. The instructions are called *bytecode* instructions. Any program written to run on the JVM has to be compiled into bytecode instructions and serialized into a binary format, the class file.

In the past, programs for the JVM were primarily Java programs, but today, there are many programming languages, designed to run on the JVM or ported to it. Since the JVM is an abstract machine, programs targetting the JVM are not bound to any particular processor architecture. Instead, programs are executedthe JVM by interpreting or just-in-time compiling them. An interpreter or a just-in-time compiler accompanied by an implementation of the class library is a *JVM runtime*. The portability of Java programs is achieved by only porting the JVM runtime to the target architecture. Since all parts of the JVM are standardized, a program should run unchanged on any JVM runtime implementation.

Java itself is an object oriented languages and the design of the JVM, as well, is centered around objects. All code is compiled into binary class files. Each class file contains information about a single class. The important components of the class-file are (1) meta-data, such as flags and modifiers, describing the class and its signature, (2) a constant pool which is a shared storage area for constants used throughout code and meta-data, and (3) the class's methods themselves and their code and meta-data.

The binary class-file format is standardized and evolved throughout the years[3, 4, 5, 12, 25]. This was possible, because the class-file format is defined in an extensible manner. Classes, methods, fields, and code are described by attributes which basically are key-value pairs defining a particular aspect of an element. For example, the code of a method is serialized as an attribute of the method.

The constant pool is the central data store for static data in a class-file. Every constant used in one of the methods, aside from very small ones, which may be encoded directly with the bytecode instruction, is serialized in the constant pool, as well as signatures of called methods. Bytecode instructions refer to elements of the constant pool by index.

In contrast to modern mainstream CPUs which are register-machines, the JVM is stack-based. That simplifies the design of compilers targetting the JVM, since no register allocation has to be done. All data are either allocated on a heap or it exists in a frame of an execution thread. A frame contains the data stored during a method call. It consists of an operand stack, a local variable array, and the program counter. Whereas only the top elements of the stack can be accessed directly, all values in the local variable array can be accessed by index. All calculation is done on values of the stack. Parameters are passed to the invoked method on the stack. Methods receive parameters in the local variable array. A static method receives its n parameters in local variables 0 to n-1. In contrast, an instance method receives its instance in local variable 0 and its n parameters in local variables 1 to n. After a method returns its frame is discarded. The maximum size of stack and local variable array is determined per method and fixed to this value at compile-time.

The Java virtual machine is type-aware down to the bytecode-level. The primitive types integer, float, and double are supported by specialized bytecode instructions. Integer types smaller than `int`, such as boolean, byte, short and char values, are represented as integers. Other instructions can be used exclusively with reference data types. The data type an instruction can handle is indicated by the prefix character of the instruction's mnemonic. So is `ARETURN` the instruction to return a reference value, `IRETURN` the instruction returning an integer value, and `DRETURN` the instruction returning a double value. Some instructions, such as `POP`, which throws away the top value of the stack, are applicable to operands of any type.

Bytecode instructions can be classified into several groups:

- Arithmetic instructions for doing calculations on numbers. Several instructions do conversions from one number type into another.
- Load and store instructions are used to access constant values or values on the stack.
- Object manipulation instructions operate on references of heap objects. Instantiating objects, accessing fields, and casting are primary use cases.

- The code array must not be empty.
- The code array must not exceed the size of 65536.
- The first instruction has to start at index 0.
- Only instances of valid opcodes are allowed, reserved opcodes may not occur in the code array.
- There must not be any gaps in the code array.
- Each operand of an instruction must fit to its operator.

Figure 2.1: Examples of static constraints on the code array

- Array access instructions are, like arithmetic instructions, specialized to the component type of the array they should operate on. Instructions create arrays, load and store values of arrays, or retrieve the length of an array.
- Stack manipulation instructions move, duplicate, or throw away values on the stack.
- Branching instructions, such as conditional and unconditional jumps and switch instructions, change the control flow of the program.
- Method invocation instructions are distinguished by the sort of method, that should be invoked. Static methods, interface methods, constructors and normal methods are all invoked by different instructions. Methods have to return control flow to the calling method using a return instruction.
- `ATHROW` is used to throw exceptions.

Certain constraints are imposed upon the binary structure of a class-file to ensure its well-formedness and integrity. *Static constraints* concern the serialization of bytecodes in the code array of a method[25, §4.8.1]. Some of them are listed in figure 2.1. These constraints are necessary to allow a fast processing of the bytecodes once they have to be interpreted (or just-in-time compiled). Therefore, the constraints are checked by verification when the class-file is accessed for the first time. If the class-file is accepted once by the JVM verifier, these constraints are assumed to be satisfied and no further checks are necessary or done while running the program.

In contrast, *structural constraints* concern the relationship between bytecode instructions. Figure 2.2 lists only the structural constraints, that MNEMO is aware of. Others will be briefly discussed in section 5.1. Structural constraints ensure that expected invariants hold when a program is executed, that types are correct, and the needed

1. Each instruction must only be executed with the appropriate type and number of arguments in the operand stack and local variable array, regardless of the execution path that leads to its invocation.

2. An instruction operating on values of type int is also permitted to operate on values of type boolean, byte, char, and short. [...]

3. If an instruction can be executed along several different execution paths, the operand stack must have the same depth [...] prior to the execution of the instruction, regardless of the path taken.

4. No local variable [...] can be accessed before it is assigned a value.

5. At no point during execution can more values be popped from the operand stack than it contains.

6. Each invokespecial instruction must name an instance initialization method [...], a method in the current class, or a method in a superclass of the current class.

7. When the instance initialization method [...] is invoked, an uninitialized class instance must be in an appropriate position on the operand stack. An instance initialization method must never be invoked on an initialized class instance.

8. When any instance method is invoked or when any instance variable is accessed, the class instance that contains the instance method or instance variable must already be initialized.

9. There must never be an uninitialized class instance on the operand stack or in a local variable when any backwards branch is taken. [...]

10. The arguments to each method invocation must be method invocation compatible [...] with the method descriptor [...].

11. The type of every class instance that is the target of a method invocation instruction must be assignment compatible [...] with the class or interface type specified in the instruction.

12. Each return instruction must match its method's return type. [...] If the method returns a reference type, it must do so using an areturn instruction, and the type of the returned value must be assignment compatible [...] with the return descriptor [...] of the method.

13. The type of every value stored into an array of type reference by an aastore instruction must be assignment compatible [...] with the component type of the array.

14. Execution never falls off the bottom of the code array.

Figure 2.2: Selected structural constraints on the code array [25, §4.8.2]

number of operands for an instruction is available. Adherence of structural constraints is as vital to the well-behaviour of the virtual machine as the static constraints if not even more so. A wrongly serialized class-file (broken static constraint) can probably not be loaded at all; this may stop loading of the class and even cancel the execution of a program if not catched, but at least no rogue code is tried to be executed. Now consider a program where an integer is on the stack where a reference is expected (broken structural constraint). If not prevented, a possibly arbitrary or in a worse case rogue integer value would be dereferenced, which in the worst case would allow the program to access the raw memory of the process.

Therefore, each class-file (aside from the ones in the boot classpath) is carefully checked to comply to the structural constraints. At load-time of each class, the verifier of the virtual machine analyzes each class for breaches of these constraints. Verification is expensive, since it requires the checking of type conformance at every point in the program. Particularly difficult is the check of program points where different branches of control flow are joined. The JVM requires that the types of operands of instructions match regardless which control path is taken to reach the instruction. At the targets of control flow instructions, it must be checked that the depth of the stack is the same always, and types of values on the stack and local variables have to be unified from all paths of control flow joining there. With Java 6 this unification of the types is therefore already done at compile-time and saved with each method into the *stack map table*. The new Java 6 verifier only has to check if all the types comply with the ones in the table[12].

In the past several attempts were made to prove the unsoundness of the Java verifier by devising a formal type system of the JVM[23, 11], against which the verifier can be validated. Especially complexities with regard to object initialization and subroutines were noticed[10, 24].

2.2 Bytecode generation libraries

With the broad adoption of the Java platform, several tools emerged to operate on and generate binary class-files. There are several use cases for such tools:

- Backends of compilers which have to generate complete classes.
- Aspect oriented programming which primarily is a structured means of injecting code into existing class-files to add some functionality in hindsight (or because the added functionality is seen as a "crosscutting concern" better implemented aside from the actual implementation), e.g. additional logging or providing callbacks.
- Generation of proxy classes to generically implement interfaces or (abstract) classes, e.g. to create facade interfaces to remote implementations.

- Optimization by just-in-time or ahead-of-time compiling data-structures into bytecodes, e.g. by replacing reflected calls with bytecodes[1] and for improving performance of domain specific languages.
- Doing static code analyzation or optimization on the bytecode level.

These tools differ in the way classes and instructions are represented and how operations on bytecodes can be defined, depending on the focus of the library and on the anticipated use case.

Since all the frameworks, are responsible for (de-)serializing, they have to ensure that the static constraints for class-files hold. Using a bytecode generation library, the compiler writer can focus on the higher-level issues and on maintaining structural constraints.

2.2.1 ASM

Objectweb ASM is a lightweight bytecode generation and transformation framework[6]. In contrast to many other bytecode frameworks, ASM has, aside from an object-tree based encoding of class and method structure, a light-weight API employing the Visitor pattern. Fields, Methods, and instructions are generated by calling methods of the visitors.

See this (slightly changed) example from [6]:

```
ClassWriter cv = new ClassWriter(0);

// create class, annotations, fields, etc.
cv.visit(...);

// create method m1
MethodVisitor mv1 = cv.visitMethod(..., "m1", ...);
mv1.visitCode();
mv1.visitInsn(...);
...
mv1.visitMaxs(...);
mv1.visitEnd();

// create method m2
MethodVisitor mv2 = cv.visitMethod(..., "m2", ...);
mv2.visitCode();
mv2.visitInsn(...);
...
```

[1]Java itself does so in the Sun JDK: After `java.lang.reflect.Method.invoke` is called several times, the implementation generates trampoline classes on the fly to call the method. These classes occur in stacktraces sometimes as `GeneratedMethodAccessorNNN`. The Java library uses a custom bytecode generator.

```
mv2.visitMaxs(...);
mv2.visitEnd();

cv.visitEnd();

// get byte array containing the complete binary class-file data
byte []bs = cv.toByteArray();
```

A `ClassWriter` is a particular `ClassVisitor`. While visited it creates the binary representation of the class-file. A `ClassVisitor` can also represent a transformation of a class-file. This `ClassVisitor` is chained between a `ClassReader` and a `ClassWriter`. One can combine several transformations so that all of them are applied.

ASM has some tools available which aid in generating class-files[2]. They are implementations of `ClassVisitor`. There is a disassembler (`TraceClassVisitor`) and a tool checking that methods of `ClassVisitor` are properly used. The `ASMifier` is of special importance: Given an example class-file, it prints the code needed to generate the class-file with ASM. Therefore, one can bootstrap a generic generator by using `ASMifier` to create a stub from a concrete example and then edit this code to suit one's needs.

ASM can generate the stack map tables required by the new type checking verifier in Java 6 automatically or manually.

In the development of MNEMO, ASM was used as the low-level library to create class-files.

2.2.2 BCEL

The Bytecode Engineering Library (BCEL) is a project of the Apache Jakarta package "intended to give users a convenient possibility to analyze, create, and manipulate (binary) Java class-files (those ending with .class)."[1]. It has a complete object model, not only of class and method structure, but as well for bytecode instructions. It can analyze, change, and generate classes, hereby adhering to the previously described static constraints of class-files. Bytecode instructions are arranged in a sophisticated class hierarchy. Generating code for a method looks like this (quoting from the manual on the webpage and adding the appending of the `DUP` instruction):

```
InstructionFactory f  = new InstructionFactory(class_gen);
InstructionList    il = new InstructionList();
...
il.append(InstructionConstants.DUP);
il.append(new PUSH(cp, "Hello, world"));
il.append(new PUSH(cp, 4711));
...
```

[2]http://asm.objectweb.org/asm31/javadoc/user/

```
il.append(f.createPrintln("Hello World"));
...
il.append(f.createReturn(type));
```

Primitive instructions (like `DUP`) are accessible as constants. More complex instructions can be created by instantiating an instruction class value or by using an `InstructionFactory`.

BCEL contains a library of tools surrounding class generation. So there are modules to disassemble a module into HTML (`Class2HTML`) or into Java source code, which itself uses BCEL to generate the code (`BCELifier`); a bytecode verifier (`Verifier`) is available as well.

The Code Generation Library (cglib)[3] is another open-source library built on top of BCEL. It contains high-level functions to intercept methods and create proxies. The Java Object-Relational-Mapping (ORM) tool Hibernate used cglib in the past to intercept calls to getters and setters and implement lazy loading of entities from the database.

2.2.3 Javassist

Javassist is the bytecode generation framework by Shigeru Chiba[7]. It supports a high-level abstraction to transformations upon class-files.

You may instrument class methods and call-sites with code snippets at load-time of these classes. The code snippets can be stated in a Java-like template language. Javassist employs a custom Java-compiler to compile these templates into appropriate instruction sequences when replacing or inserting them.

Additionally, Javassist has a bytecode level api[4] allowing direct access to and generation of bytecode instructions. Instructions of existing class-files can be analyzed using a `CodeIterator`. To create instructions, the class `Bytecode` can be used, which represents a bytecode array. It has methods to add particular bytecodes like `Bytecode.addIconst` similar to ASM's `MethodVisitor`.

2.2.4 FJBG

FJBG (Fast Java Bytecode Generator?) is the library used by the JVM backend of the Scala compiler. Its source code is freely accessible[5] but there is no documentation avail-

[3]http://cglib.sourceforge.net/

[4]http://www.csg.is.titech.ac.jp/~chiba/javassist/tutorial/tutorial3.html#intro

[5]https://lampsvn.epfl.ch/trac/scala/browser/fjbg

able. It is written in Java. At the time it was written, there seemed no bytecode generation framework available to match the performance needs of the Scala compiler[6]. There were recent efforts to port FJBG to Scala[7].

The API to generate bytecode instructions is similar to the one of Javassist.

2.3 Scala

Scala is a statically typed programming language targetting the JVM[21, 20]. It mixes concepts from object oriented languages with concepts from functional programming languages.

Whereas Java only has classes and interfaces, Scala has **`trait`**`s`, **`class`**`es` and singleton **`object`**`s`. A **`trait`** is comparable to a Java interface, though, it is possible to specify default implementations for methods. A Scala **`class`** is similar to a Java class. It may only have one superclass but may derive from multiple **`trait`**`s`. **`object`**`s` are singleton objects, which cannot be instantiated explicitly but instead, are available once per runtime. **`case class`**`ses` are special classes that are concisely defined and may be instantiated without the **`new`** keyword.

Scala has support built-in support for function values. The **`trait`** `FunctionX` represents a function with X parameters. Scala has two different syntaxes to define literal function values: First, a function value might be declared by a parameter declaration followed by an arrow and the function body, as seen here:

```
(a,b,c) => {/*...*/}
```

Otherwise, a function value may be defined by using the wildcard character _:

```
Integer.valueOf(_)
```

is a short form of

```
i => Integer.valueOf(i)
```

The Scala compiler has a powerful type-inferer (at least when compared to Java)[22]. Only types of method parameters must be declared. Other types can be infered from expected result types or declared argument types by local type inference.

[6]`http://www.nabble.com/-scala---Re%3A-FJBG-port-to-Scala-p17861145.html`
[7]`http://www.drmaciver.com/repos/fjbg/`

Scala has the concept of **`implicit`** method parameters. These are parameters of methods that need not be passed explicitly but instead, are filled in by the compiler. The compiler may choose, therefore, from a list of **`implicit`** defined values or methods currently in scope. Special **`implicit`** methods may be used to extend classes with functionality. If a member of an object is not found, Scala looks for an **`implicit`** method which may convert the object into another object which has the required member available. **`implicit`** defined methods may have **`implicit`** parameters themselves and may even be applied recursively. The Scala compiler ensures that multiple recursive applications of **`implicit`**s converge. An important fact to note is, that **`implicit`** parameters (and other parameters from additional parameter lists of a method) never take part in type infering, since the type inferer always runs before the phase where **`implicit`**s are resolved.

2.4 Heterogeneous lists

Before Java 5 collections were completely untyped in the Java language. In a Java 1.4 collection, it is possible to insert a reference to any object into a collection, i.e. the methods of the collection are typed to work on any `java.lang.Object` value. This allows using the collection types with arbitrary element types, aside from primitive values. The downside is the lack of (static type) information about the element types. At every access to an element of the collection, the element has to be casted from `Object` into the element type. Even if a collection can only hold elements of a specific type, this knowledge is only known to the programmer or written down in comments, but it is not stated explicitly in types the compiler can check.

```
interface List{
  void add(Object element);
  Object get(int index);
}

List x = new List();
x.add("a string");
x.add(new Integer(5));

String str = (String) x.get(0);
Integer i = (Integer) x.get(1);
```

You may put different objects of different type into a Java 1.4 `List`, but when accessing an element it has to be casted into the right type, whether the type is the same for all elements of the list or not.

Java 5 introduced generics. Generics allow to parameterize classes over types. In the case of collections, type parameters enable the programmer to share code between collections of different element types, while maintaining a specific and thus, type-safe, interface to access the collection.

```
interface List<E>{
  void add(E element);
  E get(int index);
}

List<String> x = new List<String>();
x.add("a string");
x.add("another string");

for(String str:x)
  System.out.println(str);
```

Here, the list is parameterized over the element type `E`. Into a list of type `List<String>`, only elements of type `String` (or a subtype thereof) can be inserted. When accessing an element, no explicit casts are necessary, because the return type of `List<String>.get` is `String`.

Going one step further, sometimes lists are needed, whose element types may differ, but that fact should be explicitly represented in the type of the list. From a different point of view, they can be seen as a typed n-tuple like the Scala classes `TupleX`, but of arbitrary arity. Kiselyov et al. call these *typeful heterogeneous lists*[13]. The authors present a Haskell data structure which resembles the structure of ordinary list values, i.e. built up from `cons` and `nil`, but doing so in types. A heterogeneous list, accordingly, is either an empty list of type `HNil` or a list is recursively constructed (`HCons e l`) out of a new element of a particular type `e` and a rest list of type `l`. The type of the elements is, thus, captured in nested `HCons` types analogous to the values themselves being nested within `cons` cells.

Afterwards, several operations are defined which may operate on these lists. Of particular interest are numeral-based access operations. To access elements of a heterogeneous list by position while retaining type information, natural numbers are defined in terms of types as well. *Type-level natural numbers* are defined isomorphic to the list type and, therefore, as suggested by the peano axioms, by a type `HZero`, representing zero, and a successor type `HSucc p` of another numberal type `p`.

A similar encoding of natural numbers in the lambda calculus are *Church Numerals*[8], in which a natural number n is represented by a higher-order function which maps a function to its n-fold combination. A type-level version of this idea can be found in [16, 15]. A Scala implementation of this principle is recently being discussed on the internet[9, 14].

Chapter 3

Bytecode combination with *Mnemonics*

As shown, many libraries are available to generate Java bytecodes. They support the bytecode generation task by varying degrees and on different levels of abstraction. They handle the serialization into binary class-files and manage constant-pool entries. Aside from possible bugs, every class-file emitted by such tools is a valid class-file on the binary level. Though, it still remains possible to generate classes being rejected by the verifier. It may be caused by using bytecode instructions on values of wrong types or accessing unset local variables. These are breaches of the structural constraints, as explained before.

Bytecode generation libraries may check at generation time that the generated sequence of instructions is valid, and in fact the ASM tool has such a plugable verifier. However, the situation could be improved further if invalid bytecode sequences could not even be expressed in terms of the generation library.

It is this, what the developed bytecode combination library strives to achieve. The types to occur in the *generated* program are encoded as type parameters in the *generating* program. Scala's type-system makes it possible to constrain the combination of bytecodes instructions to a sensible way, and this without making to much type declaration necessary, but, instead, by infering types, whenever possible.

The work on this library was two-fold: First, the type-safe API to combine bytecode instructions had to be developed, and then the defined interfaces had to be implemented and embedded in a framework to make it accessible. The sequence of the next sections follows this structure. After an introductory example, first, the API is described and how to achieve type-safety; second, the actual implementation is outlined as well as the minimal framework to generate classes from combined bytecode instructions.

3.1 A small example

MNEMO defines a minimal framework to specify and compile bytecode blocks through `ASMCompiler`:

```
object ASMCompiler{
  def compile[T<:AnyRef,U<:AnyRef](cl: Class[T])
                               (code: F[Nil**T,Nil] => F[Nil**U,_]): T => U
}
```

A call to `compile` expects an input class and an instruction sequence by means of a function which transforms a frame state denoted by the type `F[Nil**T,Nil]` into another frame state `F[Nil**U,_]`. `compile` dissects the bytecode sequence and compiles it into a Scala function value instance by generating a custom class implementing Scala's `Function1` trait and instantiates it.

Here is a short example which does some math:

```
> val func = ASMCompiler.compile(classOf[java.lang.Integer])(
    _ ~
      invokemethod1(_.intValue) ~
      bipush(1) ~
      iadd ~
      invokemethod1(Integer.valueOf(_))
  )
func: (java.lang.Integer) => java.lang.Integer = <function>
```

This example generates a function which takes a `java.lang.Integer` as input parameter as specified by the first parameter to `compile`. The second parameter defines the actual bytecode instructions to transform. It can be expected that a value of the former given input type is on the stack. A bytecode instruction sequence is built by combining single instructions with the ~ operator. `_ ~` is a short form of `f => f ~` and can be read as, "starting with a frame f, apply the now following bytecode instruction(s)". `invokemethod1(_.intValue)` represents the instruction to call a method; the method to call is given as a closure. At length the parameter could be written as

```
(i:java.lang.Integer) =>  i.intValue
```

which is a function value, which, when called, extracts the primitive integer value from a boxed `java.lang.Integer` object by invoking the instance method `java.lang.Integer.intValue`. Note that in the short form, the parameter type is infered automatically. So, with this instruction, the input value is converted from a boxed integer value into a primitive one. One can now use bytecode instructions for arithmetics to operate on this value. `bipush(1)` pushes a constant integer `1` onto the stack. Now, there is the input parameter as a primitive value on the stack, and on top of this, `1`. Next, `iadd` consumes the two integer values on the stack and adds them, and pushes the result back on stack.

The instruction `invokemethod1(Integer.valueOf(_))` converts the result back into a boxed integer. The semantics of `compile` define, that this last value on the stack is returned to the caller.

The result of the compilation is `func`, a Scala function value of type `java.lang.Integer => java.lang.Integer`. This function value can further be used like an otherwise defined function.

```
> func(0)
res: java.lang.Integer = 1

> func(12)
res: java.lang.Integer = 13
```

An integer can be passed to the function, which then is operated on as defined before: It is increased by 1 and returned.[1]

So, it is possible to create a function out of a sequence of bytecode instructions on the fly. An invalid sequence of bytecodes, however, should be rejected by the Scala compiler. Consider this code which wrongly tries to apply a method of `String` onto an integer parameter on the stack:

```
> val func = ASMCompiler.compile(classOf[java.lang.Integer])(
    _ ~
      invokemethod1(_.intValue) ~
      bipush(1) ~
      iadd ~
      invokemethod1((str:String) => str.length) ~
      invokemethod1(Integer.valueOf(_))
  )
```

The Scala compiler answers this failed attempt with an error message:

```
<console>:15: error: type mismatch;
 found   : scala.reflect.Code[(String) => Int]
 required: scala.reflect.Code[(Int) => ?]
            invokemethod1((str:String) => str.length) ~
                                        ^
```

3.2 Type-safe bytecode combination

As explained in the last chapter, the main concern of structural integrity of bytecode methods is the concern that an instruction may only appear at a position where the

[1]It may look as if the input value would be passed as a primitive integer into the function, but, actually, it is automatically boxed by Scala into a `java.lang.Integer`.

data it operates on has the correct type. Thus, an integer addition instruction is only valid when two integers are on top of the stack, or a method invocation instruction can only be called when values of the expected argument type are on the stack in the correct order.

During method execution values are stored on the stack and in local variables, together called a *frame*. A frame at a particular point of execution in a method is called *frame state*.

MNEMO specifies an encoding for the *types* of frame states and defines instructions as transformations of frame states. The types of operands are tracked as type parameters of a class representing the frame state. An instruction defined as a transformation of frame states is encoded as a Scala function which itself is parameterized and can only be applied to correctly parameterized frame states.

To construct a code block or a method out of instructions one combines instructions by combining the functions they are represented by. This enables building higher-level abstractions by combining simple instructions.

The data structures encoding the type of frame states and how instructions can be combined is described next, followed by the actual encoding of bytecode instructions.

3.2.1 Encoding of frame states

For each method invocation the JVM allocates a frame. All bytecode instructions operate exclusively on data being in the current frame. The frame consists of the operand stack and the array of local variables.

Bytecode instructions manipulate the contents of stack and local variables. All the data is typed and many instructions only work on data having the correct type. `iadd`, `fadd` and `dadd` are examples of specialized bytecode instructions adding two numbers for integer, float and double values.

In MNEMO the types of the data on the operand stack and in the local variables at a particular point in the execution are encoded as type variables of a Scala **`trait`**:

```
trait F[ST<:List,LT<:List]
```

The type of a frame at a point is parameterized by the types of the stack and the local variables. Since the type of stack and local variables are solely defined by the types of the *elements* of the stack or the local variables at this specific point we can just list the types of the values on the stack and in the local variables in the type of the Frame. The stack type is given in the first parameter of the frame state type while the local variables type is given in the second parameter.

Since the size of elements on the stack may be arbitrary, the `List` type cannot just have some type parameter tuple defining the types of the elements. Instead, a different encoding is needed.

Similar to the heterogeneous lists, as presented before, a list of types is constructed recursively with the given definition of `Nil` and `Cons`.

```
trait List

trait Nil extends List
case class Cons[R<:List,T](rest:R,top:T) extends List
```

A list may be the empty list in which case `Nil` is the type of the list or may be constructed of a rest and a top element. This resembles the classical definition of a cons list but in this case it may not only hold elements of one type, but values of different types as well. The types of elements are tracked by the type parameters of the `Cons` type.

An empty stack has the type `Nil` while a stack with the elements 3,"Str",3.5f in this order on top would have a type of `Cons[Cons[Cons[Nil,Int],String,Float]]`[2]. To aid readability a type shortcut was introduced like this:

```
type ** [x<:List,y] = Cons[x,y]
```

It can be used in infix notation and is left-associative. With this definition the type of the example above simply reads as `Nil**Int**String**Float` with the top of the stack on the right side of the type expression.

Stacks being a FIFO data structure are matched well by the given definition of `List`. For local variables a random access data structure would be better suited but since a random access data structure which tracks an arbitrary amount of types is difficult to achieve with the Scala type system, the `List` data type is used to track the types of local variables as well. Local variable types are modelled from outer to inner, so the outermost (or in infix notation the rightmost) type is local variable 0. To access variables with indices greater than 0 the type must be 'unfolded'. This fact and the complications it introduces are discussed later in detail in 3.2.2.

[2]Primitive integers are noted as `Int` in Scala.

3.2.2 Instructions

Most bytecode instructions interact with the stack or local variables: Arithmetic operators fetch their operands from the stack and put the result back on top of the stack, load and store instructions move data from a local variable onto the stack or vice versa, and method invocations receive parameters from the top of the stack and return values onto the stack. Some instructions like the unconditional branch do not change the stack but expect the stack to be of a specific type.

As enforced by the JVM verifier and as outlined before, many instructions may only operate on data of a certain type (figure 2.2). They are constrained regarding the types of input and output data. More precisely instructions can be seen as functions from input frame states to output frame states (which may or may not have side effects on objects on the heap as well).

Instructions are therefore modelled as functions over frame states. The type of an instruction that changes the stack value types from `ST` to `ST2` and the local variable types from `LT` to `LT2` has the type `F[ST,LT] => F[ST2,LT2]`.

An instruction and so the function it is defined by only operates on the top elements of the stack or one local variable, so the remaining types have to be transparently passed from input to output. This is accomplished by encoding each instruction as a polymorphic method. The definition explicitly states types which constitute pre- or post-condition of an instruction and the method is polymorphic in the types of unchanged elements of the frame state.

Instructions may applied to frames with the special ~ operator, which is defined on the frame state type `F`.

A table with an example sequence of instructions and the corresponding frame state types is given in this figure:

Instruction	Frame state type
	`F[Nil,Nil]`
`bipush(4) ~`	`F[Nil**Int,Nil]`
`dup ~`	`F[Nil**Int**Int,Nil]`
`invokemethod1(Integer.toString(_)) ~`	`F[Nil**Int**String,Nil]`
`local[_0,String].store()`	`F[Nil**Int,Nil**String]`

`bipush` pushes a constant integer onto the stack. The stack type, denoted by the first type parameter of `F`, therefore has the `Int` type appended. `dup` just duplicates the top value on the stack regardless which. Next, a method with an integer parameter is called, which returns a `String`. The last instruction stores this `String` in the local variable `0`.

The `String` type moved into the second type parameter of `F` denoting the local variable types.

In the following sections, different instructions will be examined, their types and their encoding.

Implemented instruction set

Several bytecode instructions have not been considered at all. Partly to constrain the extent of this thesis into a feasible set of instructions, partly because it was sufficient to present an example of a class of instructions to show the interesting bits.

- Loading constants: Only `LDC` and `BIPUSH` were implemented. In addition, `LDC` can only load Strings from the constant pool.
- Arithmetics: Only few integer arithmetic instructions were implemented. Float or double instructions could be handled analogously. No bit-level instructions were implemented. No conversion between number types are supported.
- Objects: Object creation is supported only through a nullary or unary constructor call. Casting (`CHECKCAST`) is supported. `INSTANCEOF`, the constant null(`ACONST_NULL`), and object comparison (`IF_ACMPEQ`, `IF_ACMPNE`, `IFNULL`, and `IFNONNULL`) instructions were not implemented, but implementation would be straightforward.
- Stack manipulation: Only a subset was implemented.
- Branching: Only unconditional branches (`GOTO`) were considered, as well as `IFNE` as an example of conditional branching. Others could be added in the same manners. Switch instructions (`TABLESWITCH`, `LOOKUPSWITCH`) were not considered.
- Subroutines (`JSR` and `RET`) were not considered because of their de facto deprecation. They were formerly used to implement the behaviour of **`finally`** blocks in the Java language. Complexities they introduce w.r.t. typing were researched in the past[24]. The new Java 6 type-checking bytecode verifier allows no subroutines any more[2]. Instead, subroutines can be replaced by inlining.
- Exceptions (`ATHROW` and the exception table) are not supported. An implementation of exception instructions and handling would be useful and probably interesting, but was left out for the scope of the thesis.
- Arrays: Access is supported through `aload` and `arraylength`. Array creation was not considered but implementation would be straightforward.
- Synchronization (`MONITORENTER` and `MONITOREXIT`) was not implemented.

- Static or instance field access (GETFIELD, PUTFIELD, GETSTATIC, PUTSTATIC) was not implemented, but a scheme similar to the presented method invocation scheme would be feasible.
- Return instructions (ARETURN etc.) are generated implicitly but are not available for explicit use.

Arithmetic instructions

Arithmetic instructions perform simple calculations on the data on the stack. Each instruction, e.g. IADD, ISUB, IMUL etc., is only valid for one type of numbers on the stack. The bytecode instruction set contains variants, like DADD for doubles, FADD for floats and LADD for longs, for each of the primitive number types of the JVM. The data types boolean, byte, char and short are represented as ints and there are only specialized versions of the constant push instructions (BIPUSH and SIPUSH) and the array access instructions (BALOAD and BASTORE, SALOAD and SASTORE).

An example for both a unary and a binary instruction are increment and addition of integers:

```
def iadd[R<:List,LT<:List]:F[R**Int**Int,LT] => F[R**Int,LT]
def iinc[R<:List,LT<:List]:F[R**Int,LT] => F[R**Int,LT]
```

iadd consumes the top two elements of the stack, adds them and pushes the result back on the stack. iinc consumes the top stack element increments it by one and pushes it back on the stack.

R, abbreviated for '**R**est of the stack', and LT, abbreviated for '**L**ocal variables **T**ype', are type parameters enclosed in square brackets making the method polymorphic i.e. applicable where a return value of F[R**...,LT] => F[R**...,LT] is expected and R and LT match their declared bound List. The bounds are required since ** and its longer form Cons require the first element to be of type List. Type parameters are replaced by their matching instances by the type inferer in the compiler.[3]

Noting the missing parameter list of the function, one might wonder why the method is not just declared as a constant function value (in Scala terms: a **val**). It is because Scala does not support polymorphic values because they interfere with the type inferencing scheme.

So these definitions are straightforward applications of the aforementioned principle: Types the instruction operates on are explicitly stated in the frame state type while

[3]This is one of the causes this scheme breaks down in Java: While a definition of the List type is possible in Java and Java supports polymorphic methods, Java's type inferer could not infer the result types of such functions in the tested version of the Sun JDK compiler.

not-changing parts of the types are declared as type parameters of the polymorphic method.

Stack manipulation instructions

Stack manipulation instruction definitions are no more complex than arithmetic instructions:

```
def pop[R<:List,LT<:List]          :F[R**_,LT]      => F[R,LT]
def dup[R<:List,LT<:List,T]        :F[R**T,LT]      => F[R**T**T,LT]
def dup_x1[R<:List,LT<:List,T2,T1]:F[R**T2**T1,LT] => F[R**T1**T2**T1,LT]
def swap[R<:List,LT<:List,T2,T1]  :F[R**T2**T1,LT] => F[R**T1**T2,LT]
def checkcast[T,U,R<:List,LT<:List](cl:Class[U])
                                   :F[R**T,LT]      => F[R**U,LT]
```

pop just throws away the top element of the stack. Its type is not needed anymore; this is indicated by the placeholder _. The dup instruction duplicates the top element of the stack. As the new top element is of the same type as the old one another type parameter T is added which then is used to specify this fact. dup_x1 is similar to dup but stores the duplicated element behind the element behind the former top element. swap is self-explanatory. checkcast tries to cast the top element of the stack into the supplied type, otherwise the JVM throws a java.lang.ClassCastException.

Method invocation

Several different bytecode instructions deal with method invocation. invokeinterface invokes interface methods, invokestatic invokes static methods, invokespecial invokes special methods like constructor and static or instance class initialization methods, finally invokevirtual invokes all other methods.

The different invocation types are completely hidden by MNEMO and transparently resolved when generating bytecode. This is possible because you can call only methods which are already declared. So the bytecode generator can use java reflection to decide which instruction to use.

The type of the invocation instruction affects how method calls are dispatched at the runtime of the *generated code*. This aside, two possible cases of method invocation can be identified regarding the dispatch of methods when *generating code*:

- It is a case of **static method invocation** when you call a method which is already known and in the classpath when *compiling* the generator.

- It is a case of **dynamic method invocation** when you call a method which is only known when *running* the generator.

Static methods invocation In the first case, when a method is already known when implementing the generator, the easiest way to specify a method call is to create a function object which represents the method call. Here is a short example how this may look like:

```
bipush(10) ~
invokemethod1(java.lang.Integer.toString(_))
```

This represents a snippet of bytecode which just pushes a constant integer on top of the stack and then converts it into a string using a standard JRE method. The expression `java.lang.Integer.toString(_)` is just an abbreviation of

```
(i:Int) => java.lang.Integer.toString(i)
```

which is a function literal of type `Int => String`. If abstracted over the types of this function, this leads to a (preliminary) definition of `invokemethod1` which specifies the invocation of a method with one parameter:

```
def invokemethod1[R<:List,LT<:List,T,U](code:T => U)
  :F[R**T,LT] => F[R**U,LT]
```

`T` and `U` are type parameters representing parameter and return type of the to-be-called method. The return type of `invokemethod1` itself indicates the effect of the bytecode on the stack: A value of type `T` is consumed and another value of type `U` is pushed back on the stack.

With this definition, it is now possible to infer the type of the frame state change, given a function literal. This is sufficient to type-check the generator but lacks information needed to actually emit the method invocation instruction. When the generator runs and calls `invokemethod1`, `java.lang.Integer.toString(_)` is just an instance of an anonymous function object, which is callable but contains no accessible reference to the called method.

Instead, the experimental `scala.reflect.Code` facility can be used to retrieve the meta-data needed. When defined like this

```
def invokemethod1[R<:List,LT<:List,T,U](code:scala.reflect.Code[T => U])
  :F[R**T,LT] => F[R**U,LT]
```

a call to `invokemethod1` may remain unchanged. Now, the Scala compiler will not create an anonymous function class but, instead, it generates code, which will use components from `scala.reflect` to build up an abstract syntax tree of the expression and pass it into `invokemethod1`. The implementation of `invokemethod1` parses this information and extracts the name and the class of the method to call.

This way of specifying method invocations is a very short and concise one but it has the disadvantage that one could pass *any* anonymous function into `invokemethod1` which has a valid type. In our example one could substitute `java.lang.Integer.toString(_)` by an arbitrary function value of the particular type, such as `(i:Int) => "constant"`, which is a perfectly valid function of type `Int => String`, so it is accepted when compiling the generator. It does, however, not represent a valid method call. In cases like this, no wrong bytecode is emitted but the code reponsible for parsing the `scala.reflect` AST throws an exception.

In this manner, one could define `invokemethod` for any arity of possible methods. Since this is not feasible, we defined only `invokemethod0`, `invokemethod1` and `invokemethod2` for invocations of methods with zero, one or two parameters, which covers most cases. A more complex definition which uses currying to allows invocations of arbitrary parameter count could be used instead.

Dynamic invocation Often, the actual method to call is only known at the runtime of the generator. Then, one cannot rely on static type information when compiling the generator. MNEMO provides another means of method invocation for this case which allows to explicitly state the types involved but postpones the type checks to the generation-time when the method is known.

A method in Java may be referenced by an instance of `java.lang.reflect.Method`. One can obtain it from a `java.lang.Class` instance by calling the methods `getMethod` or `getMethods`. An instance of `java.lang.reflect.Method` contains a description of the method, such as types of the parameters or the return value, and can be used to invoke the method dynamically.

Since no type information for the result type is available when compiling the generator, not only the method reference, but also the expected result type of the method has to be passed to `invokemethod1Dyn` to determine the resulting frame stack type:

```
def invokemethod1Dyn[T,U,R<:List,LT<:List]
  (method:java.lang.reflect.Method,resT:Class[U]):F[R**T,LT] => F[R**U,LT]
```

In contrast, there is no need to specify the parameter types of the reflected `method` since they are known (1) at compile-time by looking at the stack type (2) when generating because the stack keeps track of the types of its elements. At generation-time the implementation of `invokemethod1Dyn` checks if `method` actually complies to the expected types.

A special case is the invocation of a method with a `void` result type, i.e. a method returning no value. In Scala this situation is identified by the `Unit` return type. Calling such a method with the shown invocation instructions, leaves behind a `Unit` type on the type stack, where on the real runtime stack would be nothing. But since it is impossible to prevent `Unit` returning functions to be called through the static or dynamic invocation instructions, and it is not feasable to create special Unit-invocation instructions for each arity, the most pragmatic approach was the introduction of a "virtual" instruction, `pop_unit`. Its purpose is solely to dump the top stack *type* `Unit` (without changing the runtime stack at all). Its signature is

```
def pop_unit[R<:List,LT<:List]:F[R**Unit,LT]=>F[R,LT]
```

Creating new instances

Object instantiation has its particular pitfalls. Constraints 7 to 9 (see figure 2.2 as well) require that

> "[t]here must never be an uninitialized class instance on the operand stack or in a local variable when any backwards branch is taken. There must never be an uninitialized class instance in a local variable in code protected by an exception handler. However, an uninitialized class instance may be on the operand stack in code protected by an exception handler. When an exception is thrown, the contents of the operand stack are discarded."

The problem arises since the object instantiation process has two steps: In the first step, an object is allocated using the `NEW` instruction, leaving it in uninitialized state; in the second step the object's constructor has to be called. Parameters to the constructor have to be in the normal method calling order, i.e. the receiver has to be first on the stack followed by the parameters. Parameters must potentially be evaluated first, so that between allocation of the new object and its constructor call, there might be some code. The verifier has to ensure that in the meantime no accesses to the uninitialized object (which might probably be filled with random data) can take place. In the past there were breaches in the Sun Java verifier of the time, which allowed uninitialized objects to be accessed[10].

To circumvent such problems, MNEMO models object instantiation as a single step. Since the problems with parameter order for the construction call remains unsolved, MNEMO does not support arbitrary constructor calls, but only nullary or unary constructors:

```
def newInstance[ST<:List,LT<:List,T](cl:Class[T])
                                    :F[ST,LT] => F[ST**T,LT]
```

```
def newInstance1[ST<:List,LT<:List,T,U](code:scala.reflect.Code[U=>T])
                                    :F[ST**U,LT] => F[ST**T,LT]
```

The bytecode generator ensures that the necessary `NEW` and constructor invocations are correct. A nullary constructor call is simple, since there can only be one constructor of this signature, and no overload resolution has to be done. It is therefore sufficient to pass in the class of the new object.

A unary constructor call is done like a normal method call with one parameter. The implementation of `newInstance1`, like `invokemethod1` for normal methods invocations, uses the Java reflection API to lookup the correct constructor. Behind the scenes the parameter order of constructor argument and newly allocated uninitialized object has to be swapped on the stack.

Local variable access

On the bytecode level there are the instructions `xstore` and `xload` for loading and storing values to or from local variables from or to the stack, each specialized for a primitive or reference type (by replacing *x* with the character denoting the appropriate type). Both instructions are used to access a local variable *by index*.

In MNEMO, types of local variables are tracked as the second type parameter of the frame state type `F`. Like the stack type list, the list of local variables is a FIFO list. The convention in MNEMO is, that the outermost type (i.e. the rightmost in infix notation) is the type of local variable 0.

Thus, the type

```
F[Nil,Nil**Int**Float**String]
```

denotes a frame state where nothing is on the stack and parameters of type `String`, `Float`, and `Int` in local variables 0, 1, and 2.

So, although use cases and access-modes of stack and local variables differ fundamentally, with the stack on the one hand being a short-term storage accessed in a FIFO way, and local variables on the other hand being a random-access and, therefore, longer-term storage, the encoding in types in MNEMO is the same. To extract types out of the presented nested `List` structure *by index*, special attention is necessary.

The compiler must be able to reason about the type of the n-th local variable. A representation for the natural numbered indices is needed, which is available on the Scala compiler level. As the other mentioned implementations of heterogeneous lists suggest, type-based numerals are such a way to represent numbers.

The following implementation of type numerals (and operations upon them) is based on the recent work of Jesper Nordenberg on his experimental MetaScala package (in particular, the natural number definition employing the Visitor pattern, and the idea of using **`implicit`** parameters for compile-time processing is taken from there)[19].

Numerals are, accordingly, defined by a zero type and a successor type of another type, both subclasses of a common superclass `Nat`:

```
trait Nat
final class _0 extends Nat
final abstract class Succ[Pre<:Nat] extends Nat
```

A natural number n is represented by the n-fold type application of `Succ` to the zero element `_0`.

```
type _1 = Succ[_0]
type _2 = Succ[_1]
type _3 = Succ[_2]
type _4 = Succ[_3]
```

Numerals are then used to define operations to access local variable types. A load or store operation is dependent on many types: The type of the stack values, the type of the local variables, the type of the specific local variable to be accessed, and the numeral (type) of the index. The type of the index, on one hand, has to be specified *explicitly* to define which local variable should be accessed; the other types, on the other hand, can be infered from the context. Therefore, the previously common scheme of defining instructions as functions over the frame state cannot be applied here, because either all type parameters have to be stated or no one, in which case they are infered. The solution is to use a helper trait `LocalAccess` with explicitly stated type parameters which has polymorphic methods which can be used to let the compiler infer the remaining types. A local variable access instruction is represented by a method of `LocalAccess`. For example, to load a string from local variable 1, one would use `LocalAccess.load`:

```
local[_1,String].load()
```

`local` just creates a `LocalAccess` object:

```
def local[N<:Nat,T]:LocalAccess[N,T]
```

`local` takes the index of the parameter into the local variable array, `N`, and the type of the local variable to get, `T`, as type parameters. These parameters have to be given explicitly. The fact, that `T` must be given, is to simplify the definition of the `load` method below.

`LocalAccess` itself defines the two access methods `load` and `store`:

```
trait LocalAccess[N<:Nat,T]{
  def load[ST<:List,LT<:List]()(implicit fn:CheckNTh[N,LT,T],depth:Depth[N])
    :F[ST,LT] => F[ST**T,LT]
  def store[ST<:List,LT<:List]()(implicit fn:Depth[N])
    :F[ST**T,LT] => F[ST,ReplaceNTh[N,LT,T]]
}
```

For now focussing on the return types and leaving aside the **implicit** declared parameter, load's type is as expected: Given some stack ST and local variable types LT, it returns a value of type T on top of the old stack ; the actual static check if there is a local variable with the desired type at the position is checked with the **implicit** parameter. store's return type is more complicated: Given a stack ST with a value of type T on top, the instruction has to consume this value and replace it at the right position in the local variable list. ReplaceNTh[N,LT,T] is the type which replaces the N-th type of a LT<:List with the type T.

To be able to declare the type ReplaceNTh an extension to the natural number type Nat is needed:

```
trait NatVisitor{
  type ResultType
  type Visit0 <: ResultType
  type VisitSucc[P<:Nat] <: ResultType
}

trait Nat{
  type Accept[V<:NatVisitor] <: V#ResultType
}
final class _0 extends Nat{
  type Accept[V<:NatVisitor] = V#Visit0
}
final class Succ[Pre<:Nat] extends Nat{
  type Accept[V<:NatVisitor] = V#VisitSucc[Pre]
}
```

A visitor pattern is used on type-level to be able to define a type by induction over numerals. For this purpose, an abstract type member Accept is added to Nat. Accept has a type parameter itself, a NatVisitor which contains the induction logic.

ReplaceNTh is defined in terms of a NatVisitor:

```
final class ReplaceNThVisitor[R<:List,T] extends NatVisitor{
  type ResultType = List
  type Visit0 = Cons[R#Rest,T]
  type VisitSucc[P<:Nat] = Cons[P#Accept[ReplaceNThVisitor[R#Rest,T]],R#Top]
}
type ReplaceNTh[N<:Nat,R<:List,T] = N#Accept[ReplaceNThVisitor[R,T]]
```

This definition resembles the definition of a replace function over an actual list of values closely. VisitSucc is applied recursively while decrementing the numeral and destructuring the list in sync. The final case Visit0 throws away the current top value of the list type (R#Top) and, instead, appends type T. Winding up the "call-chain", all the remaining old front element types are appended back to the result. See section C for a examplary step-wise expansion of ReplaceNTh.

Finally, going back to the **implicit** parameters of load and store. They serve a dual purpose: (1) They act as a meta-programming facility to calculate values from types at compile-time, and (2) they represent constraints which have to be fulfilled for the method to be applicable.[4]

A value for an **implicit** parameter for a function call is found at compile-time by going through the list of **implicit** defined functions and values and trying to find one with matching types. An **implicit** function may have an **implicit** parameter itself. In this case the same rules to find a value are applied recursively. Implicits are resolved after typing, therefore, they have no impact on the types of the function at all (aside from provoking errors when a matching **implicit** value is absent).

It is possible, thus, to define **implicit** functions which recursively create behaviour at compile-time based on an expected type. The above referenced Depth class is an implementation of this scheme:

```
case class Depth[N<:Nat](depth:Int)

implicit def depth_0:Depth[_0]
  = Depth[_0](0)
implicit def depthSucc[P<:Nat](implicit next:Depth[P]):Depth[Succ[P]]
  = Depth[Succ[P]](next.depth + 1)
```

It is used to calculate the integer value of a numeral. load and store both require an **implicit** parameter of type Depth[N]. There are only two possibly matching **implicit** definitions (the return types of the **implicit** functions are determining): depth_0 or depthSucc. If N is _0 then the depth is zero. If N is the successor of any numeral P, use an **implicit** parameter to require the depth of P and return it increased by one.

Additionally, in the case of load, it has to be checked if the local variable at the position N is of the correct type. This constraint is checked with CheckNTh:

```
case class CheckNTh[N<:Nat,L<:List,T]

implicit def nth_0[R<:List,T,U<:T]
               :CheckNTh[_0,R**U,T] = null
```

[4]This **implicit** scheme can be understood as a constraint system: An **implicit** parameter is a constraint and its type is the predicate to be checked. An **implicit** value or a function without an **implicit** parameter is a fact, and an **implicit** function requiring an **implicit** value itself is an implication. See http://gist.github.com/66925/ for an example how a translation of a Prolog program to Scala might look like.

```
implicit def nthSucc[P<:Nat,R<:List,T,U](implicit next:CheckNTh[P,R,T])
              :CheckNTh[Succ[P],R**U,T] = null
```

CheckNTh can be understood as the predicate stating that the list type L has type T at the position N. The rules, which are used to prove this, are gained, again, by inductively decomposing into the two cases of the numeral type. nth_0 assumes the base case: If the element on top of the type list, U, is a subtype of the searched-for type T, then CheckNTh for index _0 is valid. nthSucc states: The fact, that list R has a type T at index P, implicates, that a list created by putting any type U on top of list R must have T at position P $+1$. Therefore, the Scala compiler can prove, that a local variable of said type is available by recursively trying to apply one of these rules. If a local variable with such a type is absent, this fails and the compiler complains about a missing **implicit** value.

Jumping and branching

There are two levels of branching instructions in MNEMO. There is a direct, imperative, low-level translation of the basic bytecode instructions, such as GOTO and IFNE, and there is a more functional, high-level conditional construct and a special support for loops, which needs no explicit jumps at all.

So far, instructions would only be combined using the ~ operator. This created a linear control flow. Branching instructions, however, allow several code paths to be taken based on a condition. An encoding is therefore needed, how to specify an alternative code block. In MNEMO, code blocks are represented by a function from one frame state to another. This principle applies here as well: An alternative code block is passed as a function value F[ST1,LT1] => F[ST2,LT2] – built by combining primitive instructions – into the branching instruction.

Low-level instructions are jmp and ifne and functions to define jump targets. There are two main applications of jumps: Do something conditionally and then merge control flows, or loop. The constraint imposed on jump instructions is the fact, that the frame type of the jump target has to be compatible with the frame type of the jump site. I.e. the stack must have the same depth, and each type of values on the stack or in local variables must be assignable to the types at the jump target. To make jumps type-safe, in MNEMO, a target marker specifying the expected types is necessary:

```
trait Target[ST<:List,LT<:List]
trait BackwardTarget[ST<:List,LT<:List] extends F[ST,LT] with Target[ST,LT]
trait ForwardTarget[ST<:List,LT<:List] extends Target[ST,LT]
```

There are two types of targets: A ForwardTarget is declared and can be used before its actual position is known. It has to be placed with targetHere. In contrast, a `BackwardTarget` is placed before it is used.

The instructions used with targets are

```
def forwardTarget[ST<:List,LT<:List]:ForwardTarget[ST,LT]
def targetHere[ST<:List,LT<:List](t:ForwardTarget[ST,LT]):F[ST,LT] => F[ST,LT]

def target[ST<:List,LT<:List]:F[ST,LT] => BackwardTarget[ST,LT]

def jmp[ST<:List,LT<:List](t:Target[ST,LT]):F[ST,LT] => Nothing
```

To create a forward-declared target, `forwardTarget` is called. The type parameters have to be specified explicitly, since they cannot be known or infered in advance. To actually place a forward-declared target, `targetHere` is used. It returns a frame which is of the same type but has the target defined. It should be noted that it is impossible to prevent jumps to forward-declared targets, which never were placed, at compile-time. Instead, a runtime exception will be thrown in such a case.

For a `BackwardTarget` no such restriction applies since its definition places the target at the current position. A backward target is a frame state at the same time, so one can append the following instructions to the target as it will be seen in the example below.

Given a frame of the same type, the `jmp` instruction uses a previously defined target to generate a jump bytecode (which actually is named `GOTO`) to the target. The result type of `jmp` is `Nothing`. That might seem odd, since a `jmp` instruction does change the frame state to the frame state where the control flows to after the jump. Since we cannot reason about what that might be, and since appending further bytecodes to a bytecode sequence, ending with an unconditional jump instruction, would only create potentially unreachable code, the only meaningful result type is `Nothing` which prohibits any further instructions.

An implication of `jmp` returning `Nothing` is that it cannot be used standalone. Instead, a jump instruction has to be always paired with a conditional jump instruction. As an archetype of a conditional jump instruction, `ifne` was selected and implemented:

```
def ifne[R<:List,LT<:List,T<%JVMInt](thenBlock:F[R,LT]=>Nothing):F[R**T,LT] => F[R,LT]
```

The `ifne` instruction consumes an integer value from the top of the stack. The declaration `T<%JVMInt` takes care that short integer data types, such as boolean and short, which are represented as integers in the JVM, are accepted as well. The semantics of `ifne` are: If the integer on top of the stack is non-zero, execute the `thenBlock`, if it is zero, do nothing. `thenBlock` has to return `Nothing` so it has to end with an unconditional jump.

When used with a `BackwardTarget`, `ifne` may implement a loop. The following bytecode snippet pushes the integer 5 onto the stack and saves a target after this instruction.

Following the target it duplicates the integer on the stack and checks if it is non-zero. If so, it duplicates it again and calls a method with this integer. The result is discarded. Afterwards the value is decreased by one and then the program jumps back to the target. If the value is zero, it is boxed finally, and returned.

```
> def doSomethingWithAnInt(i:Int) = {System.out.println(i);i}
doSomethingWithAnInt: (Int)Int

val jmpTarget =
f ~
  bipush(5) ~
  target

jmpTarget ~
  dup ~
  ifne(
    _ ~
      dup ~
      invokemethod1(doSomethingWithAnInt(_)) ~
      pop ~
      bipush(1) ~
      isub ~
      jmp(target)
  ) ~
  invokemethod1(Integer.valueOf(_))
```

Otherwise, to branch conditionally, one can use `ifne` in connection with a forward target:

```
> val func = ASMCompiler.compile(classOf[java.lang.Integer]){ f =>
    val jmpTarget:ForwardTarget[Nil**Int,Nil] = f.forwardTarget

    f ~ // <- we expect a java.lang.Integer on the stack
      invokemethod1(_.intValue) ~
      dup ~
      bipush(5) ~
      isub ~
      ifne(
      _ ~
        bipush(1) ~
        iadd ~
        jmp(jmpTarget)
      ) ~
      bipush(12) ~
      isub ~
      targetHere(jmpTarget) ~
      invokemethod1(Integer.toString(_))
  }
func: (java.lang.Integer) => java.lang.String = <function>

> func(1)
res1:  java.lang.String = 2

> func(2)
```

```
res2:  java.lang.String = 3

> func(5)
res3:  java.lang.String = -7
```

In this example, it is checked if a value of 5 is on the stack. If it is not, 1 is added, otherwise, it is decreased by 12 and finally, the result is converted into a string.

In the first example, the way the target is defined, may be perceived as misleading, because it looks like the target would represent the whole block starting with `bipush(5)` and thus, a jump would occur to the start of the block. Instead, the target is placed at the position where the `target` method is called. This, as well as the definition of `jmp` as returning `Nothing`, interferes with the aspiration to define bytecode snippets as declarative transformations of frame state. Instead, the `target` method imperatively places jump targets when called. This inversion is necessary to avoid type annotations where possible and instead let the type inferer of Scala's compiler capture the type of the frame state.

To solve this problems, MNEMO contains another, higher-level set of instructions for branching avoiding jumps altogether. `ifne2` provides another means to branch conditionally:

```
def ifne2[R<:List,LT<:List,ST2<:List,LT2<:List,T<\%JVMInt]
        (thenB:F[R,LT]=>F[ST2,LT2],
         elseB:F[R,LT]=>F[ST2,LT2])
           :F[R**T,LT]=>F[ST2,LT2]
```

`ifne2` uses two blocks of instructions, one for each of the possible outcomes of a condition. Both blocks have to return an identically typed frame state, which becomes the result type of the whole expression. In listing 3.1 the difference is shown between the original code using `ifne` and the code obtained by reimplementing the former example with `ifne2`. There is no target declaration needed and both branches are treated equally in syntactical terms, too.

Furthermore, to get rid of the remaining explicit jumps, an alternative option is needed to create loops. There are programming languages without any looping constructs at all. Tail-recursive functions are then used to define loops. We employed a similar approach in MNEMO and created a facility to define tail-recursive bytecode blocks where the tail-recursive self-call is later replaced by an unconditional jump instruction.

To call itself, the recursive bytecode block must be able to reference itself to initiate the tail-call. We decided to allow the creation of *anonymous recursive functions*, where no names have to be introduced for the recursive function but, instead, a parameter is passed to the bytecode block which represents the code itself and can be used to call itself. So, `tailRecursive` is a higher-level instruction which, when applied, ex-

```
> val func = ASMCompiler.compile
    (classOf[java.lang.Integer]){ f =>

    f ~
      invokemethod1(_.intValue) ~
      dup ~
      bipush(5) ~
      isub ~
      ifne2(
      _ ~
        bipush(1) ~
        iadd,
      _ ~
        bipush(12) ~
        isub
      ) ~

      invokemethod1(Integer.toString(_))
  }
```

Listing 3.1: `ifne2`

Figure 3.1: Reimplementing the example using `ifne2`

ecutes an inner code block. In doing so, the inner code block may call itself through a function value which is passed to the inner code block. The resulting signature of `tailRecursive` is shown here:

```
def tailRecursive[ST<:List,LT<:List,ST2<:List,LT2<:List]
      (inner: (F[ST,LT] => F[ST2,LT2]) => (F[ST,LT] => F[ST2,LT2]))
      (f:F[ST,LT])
      :F[ST2,LT2]
```

It requires a parameter `inner` which, given an function on frame states, returns a function of the same type. The result of `tailRecursive` itself is a function of, again, the same type.

Its usage is shown in this example, equivalent to the example above:

```
> val func = ASMCompiler.compile(classOf[java.lang.Integer])(
    _ ~
      invokemethod1(_.intValue) ~
      tailRecursive[Nil**Int,Nil,Nil**Int,Nil]( self =>
        _ ~
          invokemethod1(doSomethingWithAnInt(_)) ~
          bipush(1) ~
          isub ~
          dup ~
          ifne2(
            self,
            f=>f) // identity = nop
      ) ~
      invokemethod1(Integer.valueOf(_))
  )
```

`tailRecursive` accepts a function, which represents some bytecode block that may call itself by using the supplied self-reference, `self`. As the result type of the code block is not known in advance, and because of the recursive character, it is necessary to specify the input and output types explicitly when calling `tailRecursive`. Just as `ifne` and `jmp` have to be paired, `tailRecursive` and `ifne2` have to be paired to avoid an infinite loop.

High-level composite instructions

Using the bytecode primitives as shown up to now, it is possible to compose high-level instructions to achieve more complex functionality.

A common use case is to iterate over a collection while collecting data about its elements: a fold operation. The fold operation has to be defined differently for folds over `java.lang.Iterable` collections and native Java arrays. MNEMO defines a fold operation over arrays:

```
def foldArray[R<:List,LT<:List,T,U,X]
        (func:F[R**Int**U**T,LT**Array[T]]
           => F[R**Int**U,LT**Array[T]])
             :F[R**Array[T]**U,LT**X] => F[R**U,LT**Array[T]] =
  _ ~
    swap ~
    local[_0,Array[T]].store() ~
    bipush(0) ~
    tailRecursive[R**U**Int,LT**Array[T],R**U,LT**Array[T]]{self =>
      _ ~
        dup ~
        local[_0,Array[T]].load() ~
        arraylength ~
        isub ~
        ifeq2(pop,
              _ ~
                dup_x1 ~
                local[_0,Array[T]].load() ~
                swap ~
                aload ~
                func ~
                swap ~
                bipush(1) ~
                iadd ~
                self
        )
    }
```

foldArray is defined as a polymorphic method. The type parameters are: Some types R and LT representing unchanging parts of the stack, the element type of the array T, the type of the current result, U, and some type X which defines the type of local 0 before the invocation. foldArray expects an array of type T on the stack and the initialization value of the result value of type U. After iterating through all the elements, it returns the result value of type U on the stack, as well as the array, which is still in local 0 (that is an unwanted but tolerated artifact).

The function given to foldArray, the loop body, is called with the current index, the current result, and the current array element. The function has to consume the current element and update the result. The array itself and the current index, may be read, but should not be changed.

foldArray is defined like that: The initialization stores the array in local 0 and initializes the index as 0. The loop is implemented by means of tailRecursive. In the loop body, the array is loaded and it is checked, if the index already reached the length of the index. If it is reached, the index is discarded, and thus, only the result value remains on the stack, foldArray is finished at this point. If the length is not reached yet, the stack is prepared for the call to the passed function, func: The index is duplicated deeper into the stack, the array is loaded from local 0, and the element at the index position is loaded with aload. Now func may be called. Afterwards, the index is increased by 1 and the recursive call to the start of the loop is executed.

Other instructions

Several other instructions, which are defined in a straightforward way, should be mentioned briefly.

There are instructions that deal with arrays:

```
def aload[R<:List,LT<:List,T]:F[R**Array[T]**Int,LT] => F[R**T,LT]
def astore[R<:List,LT<:List,T]:F[R**Array[T]**Int**T,LT] => F[R,LT]
def arraylength[R<:List,LT<:List,T]:F[R**Array[T],LT] => F[R**Int,LT]
```

aload consumes an array of type Array[T] and an integer index as input on an otherwise unknown stack of type R and retrieves the element of type T from the position denoted by the index and pushes it on the unknown rest stack of type R. The local variables and their type LT remain unchanged and are handled equally. In contrast to that, astore is exactly the inverse operation. arraylength can be used to retrieve the length of an array. Note that the actual bytecode instructions for array accesses are called AALOAD, IALOAD, etc., depending on the (primitive) type they operate on. MNEMO unifies the different instructions into the ones shown. This is possible, because MNEMO tracks the current types on the stack, and therefore, can emit the correct bytecode instructions automatically.

3.3 Implementation

The overall design of MNEMO allows for several backends behind the presented interface. Two such backends were created, an interpreter, and the already briefly mentioned `ASMCompiler` which uses the ASM library to generate Java classes on the fly.

All backends have to implement the **trait** `ByteletCompiler`:

```
trait ByteletCompiler{
  def compile[T<:AnyRef,U<:AnyRef](cl:Class[T])
                                  (code: F[Nil**T,Nil] => F[Nil**U,_])
                                  : T => U
}
```

Listing 3.2: `ByteletCompiler`: The frontend interface of MNEMO

Its signature equals the previously given signature of `ASMCompiler` (see 3.1). Given an input type `cl` and a function `code`, representing a bytecode sequence as a function between input and output frame states, `ByteletCompiler.compile` has to return a Scala function object which executes this instructions.

`code` is a sequence of bytecode instructions combined by the ~ operator as described above. The bytecode instructions are themselves not defined as instance methods, but instead are methods of a Scala **object**, similar to Java's static methods, so they can be easily imported into the client using a Scala **import** statement.

```
object Instructions {
  // ...

  def iadd[R<:List,LT<:List]: F[R**Int**Int,LT] => F[R**Int,LT] = // ...
  def imul[R<:List,LT<:List]: F[R**Int**Int,LT] => F[R**Int,LT] = // ...

  def swap[R<:List,LT<:List,T2,T1]:F[R**T2**T1,LT] => F[R**T1**T2,LT] = // ...
}
```

The question arises, how to support different backends with only one set of instructions (this is desirable because it allows to use different backends without needing to recompile). The pragmatic solution is to delegate each call to an instruction to a method of the frame state type `F`. For example, `swap` is implemented in an internal method `F.swap_int`:

```
object Instructions {
  // ...
```

```
    def swap[R<:List,LT<:List,T2,T1]:F[R**T2**T1,LT] => F[R**T1**T2,LT] =
      f => f.swap_int(f.stack.rest.rest,f.stack.rest.top,f.stack.top)
  }

  trait F[ST<:List,LT<:List]{
    def stack:ST
    def locals:LT
    def swap_int[R<:List,T1,T2](rest:R,t2:T2,t1:T1):F[R**T1**T2,LT]

    // ...
  }
```

Since the actual types of the stack elements are only known in the delegate, `Instructions.swap`, and not in `F.swap_int`, relevant values are extracted in the delegate and passed to the actual implementation, `swap_int`.

So, a backend can be implemented by

1. Implementing a custom frame state type by subclassing `F`.
2. Implementing `ByteletCompiler.compile` by passing an instance of the custom frame state type into the instruction sequence, passed as the `code` parameter.

Basically, it is an implementation of the Visitor pattern. The built-up instruction sequence visits the backend's internal instruction implementations.

Two backends were implemented, a compiler (`ASMCompiler`) and an interpreter. The interpreter is used to compare results with that of the compiler backend in the test suite. Not all instructions could be implemented in the interpreter, especially some of the branching instructions. Aside from these cases, the implementation of the interpreter is straightforward (see listing A.2).

In the following, the overall structure of the compiler is explained and representative examples of instruction compilation.

3.3.1 `ASMCompiler`

`ASMCompiler` is an instance of `ByteletCompiler`. Its `compile` method uses the `code` parameter to create a class implementing the Scala function interface, loads this class directly from its binary representation in memory and returns an new instance.

A unary function in Scala is of type `scala.Function1` (the type `T => U` is just a shortcut for `scala.Function1[T,U]`). `scala.Function1` is a **`trait`**. Scala's **`trait`**s correspond to Java's interfaces, but, in contrast to interfaces, may contain implementation.

Since the JVM does not support implemented methods in interfaces and Scala supports multiple inheritance of **`trait`**s, the Scala compiler inlines the bytecode of non-abstract **`trait`** methods into all, concrete and abstract, subclasses. Therefore, `ASMCompiler` does not implement `scala.Function1` directly, but an abstract subclass, `AbstractFunction1`.

```
object ASMCompiler extends ByteletCompiler {
  // ...

  var i = 0
  def compile[T<:AnyRef,U<:AnyRef](cl:Class[T])
                              (code:F[Nil**T,Nil]=>F[Nil**U,_])
                              : T => U = {
    i+=1
    val className = "Compiled" + i

    val cw = new ClassWriter(ClassWriter.COMPUTE_MAXS)
    cw.visit(V1_5,ACC_PUBLIC + ACC_SUPER,className,null
            ,"net/virtualvoid/bytecode/AbstractFunction1", null)

    { // constructor:
      // call superclass constructor
      // ...
    }

    { // apply
      val mv = cw.visitMethod(ACC_PUBLIC, "apply"
                              ,"(Ljava/lang/Object;)Ljava/lang/Object;"
                              , null, null);

      mv.visitCode()

      // put the parameter on the stack
      mv.visitVarInsn(ALOAD, 1);
      mv.visitTypeInsn(CHECKCAST, Type.getInternalName(cl));

      code(new ASMFrame[Nil**T,Nil](mv,EmptyClassStack ** cl,EmptyClassStack))

      mv.visitInsn(ARETURN)
      mv.visitMaxs(1, 2)
      mv.visitEnd
    }
    cw.visitEnd
    classFromBytes(className,cw.toByteArray).newInstance.asInstanceOf[T=>U]
  }
}
```

Listing 3.3: `ASMCompiler.compile`: generating an implementation of `scala.Function1` on the fly

Listing 3.3 shows the implementation of the `compile` method (see A.3 in the appendix for the full source code). It uses a `ClassWriter` from ASM to generate the function class. Two methods have to be generated, the constructor and the `apply` method. The constructor just calls its superclass constructor and is therefore omitted from the listing. apply is the method which represents the function and, thus, should contain the bytecode sequence.

The JVM passes parameters to methods as local variables, while in MNEMO the parameter is expected on the stack. The move necessary is done by ALOAD. scala.Function1 has type parameters, but Scala, like Java, erases type parameters when compiling: The parameter of apply (as can be seen in apply's signature in the visitMethod statement) is java.lang.Object. Since the real parameter type cl is known, CHECKCAST can be called safely to cast the parameter value into the expected type.

Next, an instance of the custom ASMFrame type is created and passed into code. code visits the instructions and ASMFrame uses the ASM MethodVisitor mv to emit the according bytecode instructions. An object is expected on the stack of the resulting bytecode sequence, which is then returned.

The binary data of the generated class is available through ClassWriter.toByteArray. A helper function classFromBytes creates a custom ClassLoader to load the class and return a runtime instance of this class, which is used to instantiate a function object.

3.3.2 ASMFrame

ASMFrame is defined as

```
class ASMFrame[ST<:List,LT<:List](mv:MethodVisitor
                                 ,stackClass:ClassStack
                                 ,localsClass:ClassStack) extends F[ST,LT]
```

It accepts the MethodVisitor to append the instructions to and two ClassStacks to keep track of the current types on the stack and in local variables. These are needed to emit bytecode instructions specialized to the type of their operands (iload to load an integer from local variables, aload to load a reference value, etc.). ClassStack is a custom list implementation analogous to Cons on the type level. The nil-element is called EmptyClassStack. ClassStacks have an infix constructor ** used to append a new element to a list of types.

3.3.3 Simple instructions

As explained before, bytecode instructions (from Bytecode.Instructions) delegate to the (**abstract**) internal instruction methods declared in **trait** F. These methods have to be implemented in each backend.

As an example how `ASMFrame` implements instructions, see `ASMFrame.swap_int`, the implementation of the `swap` instruction:

```
class ASMFrame/*...*/ extends F{
 // ...

 def swap_int[R<:List,T1,T2](rest:R,t2:T2,t1:T1):F[R**T1**T2,LT] = {
   mv.visitInsn(SWAP)
   new ASMFrame[R**T1**T2,LT](mv
                  ,stackClass.rest.rest**stackClass.top
                     **stackClass.rest.top
                  ,localsClass)
 }

 // ...
}
```

When called, `swap_int` emits the bytecode for the `SWAP` instruction using the `MethodVisitor`. Afterwards, it returns a new `ASMFrame` reflecting the change of stack types by swapping the top two types on the stack. The parameters passed to `swap_int` are ignored, they are used only in the interpreter.

Most instructions are implemented following this simple scheme.

3.3.4 Method invocation

```
object Instructions{
  // ...
  def invokemethod1[T,U,R<:List,LT<:List](code:scala.reflect.Code[T=>U])
    :F[R**T,LT] => F[R**U,LT] =
      f => f.method1_int(f.stack.rest,f.stack.top,code)

  def invokemethod1Dyn[T,U,R<:List,LT<:List](method:java.lang.reflect.Method
                                            ,resT:Class[U])
    :F[R**T,LT] => F[R**U,LT] =
      f => f.method1Dyn_int(f.stack.rest,f.stack.top,method,resT)
}

class ASMFrame/*...*/ {
  // ...
  def method1_int[R<:List,T,U](rest:R,top:T,code:scala.reflect.Code[T=>U])
    :F[R**U,LT] =
      invokeMethod(methodFromTree(code.tree))

  def method1Dyn_int[R<:List,T,U](rest:R,top:T
                                 ,method:java.lang.reflect.Method,resCl:Class[U])
    :F[R**U,LT] =
      invokeMethod(method)

  def invokeMethod[R<:List,U](m:java.lang.reflect.Method) =
    invokeMethodX[R,U](stackClass.rest,m)
26
27   def invokeMethodX[R<:List,U](rest:ClassStack,m:java.lang.reflect.Method) = {
28     val cl = m.getDeclaringClass
29     mv.visitMethodInsn(getInvokeInsn(m)
30                       ,Type.getInternalName(cl)
31                       ,m.getName
32                       ,Type.getMethodDescriptor(m))
33     new ASMFrame[R**U,LT](mv,rest ** m.getReturnType,localsClass)
34   }
35
36   def getInvokeInsn(m:java.lang.reflect.Method) =
37     if ((m.getModifiers & java.lang.reflect.Modifier.STATIC) > 0)
38       INVOKESTATIC
39     else if (m.getDeclaringClass.isInterface)
40       INVOKEINTERFACE
41     else
42       INVOKEVIRTUAL
43 }
```

Listing 3.4: Compilation of method instructions

Listing 3.4 shows the compilation of method invocations (with one parameter). Both `invokemethod1` and `invokemethod1Dyn` delegate (lines 4 vs 9) to different methods in `F`. The actual generation of the bytecode is the same (`invokeMethodX`), once it is known which `java.lang.reflect.Method` to call. In the case of a dynamic invocation, the method is passed as a parameter (line 21) and needs no further processing. In the case of the method being passed as an AST, i.e. `scala.reflect.Code`, a helper method, `methodFromTree`, is called which extracts the proper method from the AST (line 16).

The invocation is generated in `invokeMethodX`. According to the type of method, static, interface, or normal, an invocation instruction is selected and generated.

3.3.5 Local variable access

Listing 3.5 shows the compilation of the local variable access instructions. In 3.2.2 the usage of the `Depth` type was explained. It calculates the index of the local variable which is accessed. The index is passed to delegate methods `F.loadI` and `F.storeI`.

The implementation of `loadI` first looks up the type of the local variable at the given index from the list of local variable types, `localsClass`. The type is needed to choose the correct bytecode instruction for the primitive or reference type (`opcode(toLoad,ILOAD)`). The instruction is emitted and a new `ASMFrame` is returned.

`storeI` looks up the type from the top of the stack type list, `stackClass` and then emits the matching bytecode instruction. Again, a new `ASMFrame` is returned reflecting the move of the type from stack to local variable `i`.

```
object Instructions {
  // ...
  def local[N<:Nat,T]:LocalAccess[N,T] = new LocalAccess[N,T]{
    def load[ST<:List,LT<:List]()(implicit check:CheckNTh[N,LT,T]
                                 ,depth:Depth[N])
                                 :F[ST,LT] => F[ST**T,LT] =
      f => f.loadI(depth.depth)
    def store[ST<:List,LT<:List]()(implicit depth:Depth[N])
                                  :F[ST**T,LT] => F[ST,ReplaceNTh[N,LT,T]] =
      f => f.storeI(f.stack.rest,f.stack.top,depth.depth)
}
class ASMFrame /*...*/ {
  // ...
  def opcode(cl:Class[_],opcode:Int) =
    Type.getType(cleanClass(cl.getName)).getOpcode(opcode)

  def loadI[T](i:Int):F[ST**T,LT] = {
    val toLoad = localsClass.get(i)
    mv.visitVarInsn(opcode(toLoad,ILOAD), i);
    new ASMFrame[ST**T,LT](mv,stackClass**toLoad,localsClass)
  }
  def storeI[R<:List,T,NewLT<:List](rest:R,top:T,i:Int):F[R,NewLT] = {
    mv.visitVarInsn(opcode(stackClass.top,ISTORE), i);
    new ASMFrame[R,NewLT](mv
                         ,stackClass.rest
                         ,localsClass.set(i,stackClass.top))
  }
}
```

Listing 3.5: Compilation of `local.load` and `local.store`

3.3.6 Targets & Branching

Targets are implemented as subclasses of `ASMFrame` with an additional paramater saving the current position in the bytecode sequence.

```
class ASMFrame/*...*/{
  // ...
  case class ASMBackwardTarget[ST<:List,LT<:List](mv:MethodVisitor
                                                 ,stackClass:ClassStack
                                                 ,localsClass:ClassStack
                                                 ,label:Label)
    extends ASMFrame[ST,LT](mv,stackClass,localsClass)
    with BackwardTarget[ST,LT]
    with ASMTarget

  def target:BackwardTarget[ST,LT] = {
    val label = new Label
    mv.visitLabel(label)
    ASMBackwardTarget(mv,stackClass,localsClass,label)
  }

  case class ASMForwardTarget[ST<:List,LT<:List](label:Label)
```

```
    extends ForwardTarget[ST,LT] with ASMTarget

  def forwardTarget[ST<:List,LT<:List] = {
    val label = new Label
    ASMForwardTarget(label)
  }
  def targetHere(t:ForwardTarget[ST,LT]):F[ST,LT] = {
    mv.visitLabel(t.asInstanceOf[ASMForwardTarget[ST,LT]].label)
    this
  }

  object JmpException extends RuntimeException
  def jmp(t:Target[ST,LT]):Nothing = {
    mv.visitJumpInsn(GOTO,t.asInstanceOf[ASMTarget].label)
    throw JmpException
  }
}
```

Listing 3.6: Compilation of targets and unconditional jumps

An `ASMBackwardTarget` (listing 3.6) is a frame state itself, so it derives from `ASMFrame`. In addition to `ASMFrame`'s parameters, a `ASMBackwardTarget` has another field of type `Label`. A label is an ASM data structure denoting a position in the code of a method. `target` places a newly created `Label` at the current position and returns an `ASMBackwardTarget`.

In constrast, an `ASMForwardTarget` is declared before its use, so its label is not placed immediately, but only later when `targetHere` is called.

`jmp` generates a `GOTO` instruction to the target of the given label. It does not return normal but throws a `JmpException` to ensure that no unreachable code may be expressed.

`ifne` To implement `ifne`, the instruction `IFEQ` is used actually: The `inner` code block should only be executed if the value on the stack *does not equal* zero, so `ifne_int` jumps over this `inner` block if *it is* zero. In pseudo-bytecode the the code for `ifne` is aligned like this:

```
IFEQ label
  // inner-block

  // by convention, inner clause has to end with jmp
label:
```

Listing 3.7 shows the implementation of `ifne_int`. First, the label is created, and the conditional jump `IFEQ` is emitted. Afterwards, the inner code block is evaluated. When generating the instructions for the `inner` clause, it catches the `JmpException` thrown from a `jmp` instruction which has to be thrown from an inner clause of an `ifne` instruction by convention (the inner clause has to be of type `F[R,LT] => Nothing`). Finally, the label is visited and a new `ASMFrame` reflecting the consumption of the condition value is returned.

```scala
object Instructions{
  // ...
  def ifne[R<:List,LT<:List,T<%JVMInt](inner:F[R,LT]=>Nothing)
    :F[R**T,LT] => F[R,LT] =
      f => f.ifne_int(f.stack.rest,f.stack.top,inner)
}
class ASMFrame/*...*/{
  // ...
  def ifne_int[R<:List](rest:R,top:JVMInt,inner:F[R,LT] => Nothing)
    :F[R,LT] = {

    val l = new Label
    mv.visitJumpInsn(IFEQ,l)

    try{
      inner(self)
    }
    catch{
      case JmpException =>
    }

    mv.visitLabel(l)
    new ASMFrame[R,LT](mv,stackClass.rest,localsClass)
  }
}
```

Listing 3.7: Compilation of `ifne`

`ifne2` In contrast, the higher-level `ifne2` requires a more sophisticated implementation. `ifne2` accepts two code blocks, one for each possible condition. The strategy to layout the code is:

```
IFNE thenLabel
  // else-block
  GOTO endLabel

thenLabel:
  // then-block

endLabel:
```

If the condition is not met (i.e. on the stack is an integer 0), control falls through to the else-block and then jumps over the then-block to the `endLabel`. If the condition is met, control jumps over the else-block to the `thenLabel` and then falls through to `endLabel`. Listing 3.8 shows how `ifne2_int` follows this outline. The check in the end ensures that no `InvalidFrame` is returned, which may occur if used with `tailRecursive` and one of the branches ends with a tail-call.

```scala
object Instructions{
  // ...
  def ifne2[R<:List,LT<:List,ST2<:List,LT2<:List,T<\%JVMInt]
      (then: F[R,LT] => F[ST2,LT2]
      ,elseB:F[R,LT] => F[ST2,LT2])
         :F[R**T,LT] => F[ST2,LT2] =
      f => f.ifne2_int[R,ST2,LT2](f.stack.rest,f.stack.top,then,elseB)
}
class ASMFrame/*...*/ {
  // ...
  def ifne2_int[R<:List,ST2<:List,LT2<:List](rest:R
                                            ,top:JVMInt
                                            ,then: F[R,LT] => F[ST2,LT2]
                                            ,elseB:F[R,LT] => F[ST2,LT2])
                                                             :F[ST2,LT2] = {
    val thenLabel = new Label
    val endLabel = new Label

    val frameAfterCheck = new ASMFrame[R,LT](mv,stackClass.rest,localsClass)

    mv.visitJumpInsn(IFNE,thenLabel)

    val afterElseFrame = elseB(frameAfterCheck)

    mv.visitJumpInsn(GOTO,endLabel)

    mv.visitLabel(thenLabel)

    val afterThenFrame = then(frameAfterCheck)

    mv.visitLabel(endLabel)

    if (afterElseFrame.isInstanceOf[InvalidFrame])
      if (afterThenFrame.isInstanceOf[InvalidFrame])
        throw new java.lang.Error("One execution path of ifeq2" +
                                 " must have inferable types as output")
      else
        afterThenFrame
    else
      afterElseFrame
  }
}
```

Listing 3.8: Compilation of ifne2

`tailRecursive` As explained before, `tailRecursive` supplies a self-reference to the inner code block, so that the inner code block can call itself to loop. When compiling, a call to self should be converted into a simple jump. The compilation of `tailRecursive` is shown in listing 3.9. First, a label is created at the start of the code block. Then, the code block is generated by calling the function it is represented by. The self-reference passed to the inner block is here a function which emits a `GOTO` instruction to the start of the block.[5] After jumping an `InvalidFrame` is returned. A meaningful use of `tailRecursive` always calls the self-reference inside of a branching instruction like `ifne2`, otherwise it would not loop at all or loop infinitely. Therefore, the call to `ifne2` is usually the last instruction of the inner block of `tailRecursive` and thus, determines the result value of the inner block. `ifne2` throws away the invalid result of the tail-calling branch (i.e. the `InvalidFrame`) and returns the other one instead as seen before. This result value becomes the result of `tailRecursive`.

```scala
object Instructions{
  // ...
  def tailRecursive[ST<:List,LT<:List,ST2<:List,LT2<:List]
    (func: (F[ST,LT] => F[ST2,LT2]) => (F[ST,LT]=>F[ST2,LT2]))
    (fr:F[ST,LT]):F[ST2,LT2] =
      fr.tailRecursive_int(func)(fr)
}
class ASMFrame /*...*/ {
  // ...
  class InvalidFrame extends ASMFrame[Nothing,Nothing](null,null,null)
  def tailRecursive_int[ST2<:List,LT2<:List]
       (func: (F[ST,LT] => F[ST2,LT2]) => (F[ST,LT]=>F[ST2,LT2]))
       (fr:F[ST,LT])
       :F[ST2,LT2] = {
    val start = new Label
    mv.visitLabel(start)
    func {f =>
      mv.visitJumpInsn(GOTO,start)
      (new InvalidFrame).asInstanceOf[F[ST,LT]]
    }(this)
  }
}
```

Listing 3.9: Compilation of tailRecursive

3.4 Conclusion

In chapter 2, structural constraints were presented and their importance for the well-behaviour of the JVM. Consequently, a sequence of bytecode instructions is only valid, if the types of operands and instruction match at each point in the program. Operands of bytecode instructions may be on the stack or in local variables.

In this chapter, we presented *Mnemonics*, a Scala library that allows the specification of sequences of bytecode instructions in such a way, that all bytecode sequences expressible with it should conform to the structural constraints, or otherwise, should be rejected by the Scala compiler.

[5] In the interpreter, `tailRecursive_int` does actually not tail-call but, instead, is implemented as `func(tailRecursive_int(func)_)(fr)`, which is an implementation of the classical call-by-value Y-combinator.

To accomplish this, first, an encoding in Scala for the types of values on the stack and in local variables was given by introducing the frame state type `F`, which represents the types of values on the stack and local variables at a point in the execution of the program. Next, bytecode instructions were defined as functions from an input frame state type to an output frame state type. A selected set of bytecode instructions was encoded following this design and it was shown how meaningful programs or high-level composite instructions can be defined by combining these primitives.

Finally, we went through a prototype runtime implementation of this type-level interface, which uses the ASM bytecode generation library for the actual serialization of class-files.

Chapter 4

A case study: Objectformatter

MNEMO with its minimal framework is not intented to be a general purpose bytecode generation framework. While it certainly could be extended to such a scale by adding facilities to specify complete classes with fields and methods, it was designed to meet the needs to optimize the performance of some piece of code by specifying its bytecode translation in a concise and integrated way and then compiling it on the fly into bytecodes. Domain specific languages (DSLs) which describe some manipulation, transformation or filtering of data which should be applied to some amount of data are a main target for such optimizations.

A DSL is – in contrast to a general purpose language like Java – designed only to express and describe data or operations of a certain domain. By tight-fitting it to the realm of a problem one tries to make descriptions of the problem, data or operation at hand as concise and efficient as possible.

DSLs come in two flavors: internal and externals DSLs. An internal or embedded DSL has no syntax of its own, but it is defined completely in terms of the host language it integrates with. Thus, an implementation of a parser is not needed and all the host language's tools can be used. This is especially convenient since IDE facilities, such as code completion, work out of the box with the DSL. MNEMO is an instance of an internal DSL and as shown makes heavy use of syntax constructs and the type system of its host language Scala.

To the contrary, an external DSL defines its own syntax. Therefore, the DSL is not constrained to the syntax and semantics of the host language but is totally flexible in its design. The added cost of external DSLs compared with internal DSLs arises from the need to implement a parser and an interpreter or compiler and sometimes a whole toolchain to operate on the DSLs. A prime example of a special purpose language like this is SQL, the Structured Query Language, to access data from databases. Others include BNF, HTML or LaTeXwhich each serve a specific need.

There are some ubiquitous external DSLs in programming languages viz. regular expression and format strings. Both center around string processing and both strive to define a concise syntax for the task at hand by mixing literal string fragments with special chars. MNEMO could be used to create a compiler for each of them. The Java package `java.util.regex` implements regular expressions by compiling a regular expression string into an abstract syntax tree. When matching, the abstract syntax tree is evaluated for the target string. A further optimization could use a bytecode generation library to compile the evaluation instructions encoded in the abstract syntax tree to bytecodes.

We chose another domain and developed a small sample DSL to test the applicability of MNEMO. It is a special format string compiler centered around objects. Given a format string, the compiler will create a function which creates a textual representation of an object as specified by the format string. It resembles the notion of *variable interpolation* in several programming languages like Perl, Ruby or Groovy but it is implemented as a library feature. Variable interpolation allows in a string literal to directly reference variables of the enclosing scope. Using Ruby and having a variable of `cat_name` in scope, you could use

```
"This is the cat's name: #{cat_name}."
```

instead of the expression

```
"This is the cat's name: " + cat_name + "."
```

The transformation from the inline syntax will be done by the compiler or the interpreter.

4.1 Formatter API

The API to access Objectformatter is given through this two traits:

```
trait IObjectFormatter[T<:AnyRef] {
  def format(o:T):String
}
trait IObjectFormatterFactory {
  def formatter[T<:AnyRef](clazz:Class[T],format:String)
    :IObjectFormatter[T]

  def format[T<:AnyRef](format:String,o:T):String =
     formatter(o.getClass.asInstanceOf[Class[T]],format).format(o)
}
```

First, the definition of an object formatter (`IObjectFormatter`): An object formatter is always specialized to a particular type `T` which has to be a reference type. It has one method `format` which takes an object of type `T` and creates a string out of it.

An object formatter can be created using an `IObjectFormatterFactory`: Given a class object and a format specification it creates a formatter for instances of the type specified by `clazz`. There are two implementations of `IObjectFormatterFactory`, an interpreter, `ObjectFormatter`, that parses the format into an AST when `formatter` is called and wraps an `IObjectFormatter` around it which will interpret the AST when its `format` method is called; and the other implementation, `FormatCompiler`, which after parsing uses MNEMO to compile the format string into a function of type `T => String` and returns it wrapped in an `IObjectFormatter`.

`IObjectFormatterFactory` has a method `format` which can be used to apply a format directly to an object. It shortcuts formatter generation and formatting and throws away the formatter after use.

4.2 Syntax and Semantics

```
// literals
char        ::= regexp:'[^\[\]|#]'
              | '##'
              | '[['
              | ']]'
              | '||'
literal     ::= char char*

// expressions
idChar      ::= regexp:'\w'
idPart      ::= idChar idChar*
id          ::= 'this'
              | idPart ['.' id]
exp         ::= '#' (id | '{' id '}')

// expansions
separator   ::= regexp:'[^}]*'
expansion   ::= exp ['[' format ']'] ['{' separator '}'] '*'

// conversions
conditional ::= exp '?[' format '|' format ']'
dateConv    ::= exp '->date[' regexp:'[^\]]*'  ']'

// format string
formatElement ::= expansion
                | conditional
                | dateConv
                | exp
                | literal
```

```
format        ::= formatElement*
```

Listing 4.1: Grammar for Objectformatter format strings

The grammar of an Objectformatter format string is given in listing 4.1. Terminal symbols enclosed in square brackets (like in `['.' id]` of the `id` production rule are optional. Terminal symbols prefixed with `regexp:` represent the strings matched by the quoted regular expression.

The simplest format is just an ordinary `String` literal, however, the special characters '#', '[', ']', '|' may not be used as such but have to be doubled if they are meant literally:

```
> val o = new Object
o: java.lang.Object = java.lang.Object@1f6f3dc

> ObjectFormatter.format("just a string",o)
res: String = just a string

> ObjectFormatter.format("just a string with ##, ||, [[ and ]]",o)
res2: String = just a string with #, |, [ and ]
```

4.2.1 Property access

A property value of the object can be inserted into the string by using `#<property_path>` in the format string. To create a string containing information about the class of an object, you could use

```
> ObjectFormatter.format("The class of the object is: '#class'",o)
res3: String = The class of the object is: 'class java.lang.Object'
```

which is equivalent to the expression

```
> "The class of the object is: '" + o.getClass.toString + "'"
res4: String = The class of the object is: 'class java.lang.Object'
```

The property path syntax is similar to Java field access syntax or the JSF EL property access. We roughly follow the Java Bean convention in that a property is defined by its *getter*. But in addition to looking for a getter, if none is found, Objectformatter also looks for another parameterless instance method with just the given property name, so that this works as well:

```
> ObjectFormatter.format("The class of the object is: '#getClass'",o)
res5: String = The class of the object is: 'class java.lang.Object'
```

Dot notation may be used to access subproperties:

```
> ObjectFormatter.format(
    "The class of the object is: '#class.simpleName'",o)
res6: String = The class of the object is: 'Object'
```

The object itself can be accessed by `#this`. Thus, `#class.simpleName` is just short for `#this.class.simpleName`. The property path may be enclosed in curly brackets to disambiguate syntactical cases where a possible control character is meant literally after a property access.

4.2.2 Conversion

There are some options to convert a non-string property value into a string. For this thesis only formatting of dates and formatting of `booleans/scala.Options` were implemented.

Date formatting is specified with `->date[<date_format>]`:

```
> val aDate = new java.util.GregorianCalendar(2008,1,1)
> ObjectFormatter.format("Date: #this->date[dd.MM.yyyy]",aDate)
res6: String = Date: 01.02.2008
```

The conversion is implemented using `java.text.SimpleDateFormat`, so its format pattern rules apply.

`Boolean` and `scala.Option` values formatting can be specified using a special conditional expression. The format pattern is `?[<true format>|<false format>]`. In the case of a boolean property, `true_format` or `false_format` are used depending on the value.

```
> case class Result(outcome:Boolean)
defined class Result

> val fo = ObjectFormatter.formatter(classOf[Result],
                       "The guess was #outcome?[right|wrong]")
fo: net.virtualvoid.string.IObjectFormatter[Result] = //...

> fo.format(Result(true))
res7: String = The guess was right

> fo.format(Result(false))
res8: String = The guess was wrong
```

`scala.Option` values are handled similar: If the value is `None`, `false_format` is used as the format; otherwise, if the value is `Some(x)`, `true_format` is used with the value of `this` being x.

```
// define the outcome of a lottery game
// prize may be Some(dollars) or None
> case class LotteryOutcome(prize:Option[Int])
defined class LotteryOutcome

> val lofo = ObjectFormatter.formatter(classOf[LotteryOutcome],
              "You played lottery and... you" +
              "#prize?['ve won #this Euros!|drew a blank.]")
lofo: net.virtualvoid.string.IObjectFormatter[LotteryOutcome] = //...

> lofo.format(LotteryOutcome(Some(2000000)))
res9: String = You played lottery and... you've won 2000000 Euros!

> lofo.format(LotteryOutcome(None))
res10: String = You played lottery and... you drew a blank.
```

The example defines a case class `LotteryOutcome` which may hold the result of a lottery game. There are two possible types of outcomes: First, the case in which you win something, this is encoded by `Some(amountOfMoney)` with amountOfMoney being an integer value. Second, losing is encoded by `None`. Then, a formatter is defined which will calculate a textual representation of a `LotteryOutcome`; the format states two different representations for each case, win and loss. Applied to a positive outcome, the first format is selected and the contents of `Some(2000000)` is inserted at the place denoted by `#this`, while applied to the negative outcome, the second format is inserted.

4.2.3 List and array expansion

Objectformatter allows formatting of containers by iterating over its elements, applying a format and joining all the elements' string representations into the result string. Right now every collection type implementing `java.lang.Iterable` is supported as well as native Java arrays.

As evident from the grammar, an expansion format consists of the property access expression, an optional format applied to each element, an optional separator string, and the asterisk denoting the expansion. The default element format, if no format is given, is `#this`, which will call the `toString` method of each element. The default separator string is the empty string.

```
> val list = List("This","is","a","list")
list: List[java.lang.String] = List(This, is, a, list)

// Example A
> ObjectFormatter.format("#this*",list)
res: String = Thisisalist

// Example B
> ObjectFormatter.format("#this{, }*",list)
res: String = This, is, a, list
```

```

// Example C
> ObjectFormatter.format("#this[String '#this' has length: #length]{, }*",list)
res: String = String 'This' has length: 4, String 'is' has length: 2, //...

// Example D
> ObjectFormatter.format("#this[\"#this\": #toCharArray['#this']{, }*]{\n}*",list)
res14: String =
"This": 'T', 'h', 'i', 's'
"is": 'i', 's'
"a": 'a'
"list": 'l', 'i', 's', 't'
```

Listing 4.2: Collection formatting examples

Figure listing 4.2 shows examples of collection formats. In example A the string elements are just appended one after the other. Example B shows the use of a separator string. In example C a subformat is used to list the length of each of the string elements. Finally, example D shows nesting of expansions.

4.3 Parser and AST

The format strings are parsed into an abstract syntax tree (AST). Objectformatter uses the scala parser combinators library, which turned out well suited for the task. Parser combinators enable us to state the format grammar in a form similar to BNF, but in Scala source code. A further explanation of the parser code is out of scope of this thesis. Instead, only the AST is explained, since this is the input for the compiler.

```
case class FormatElementList(toks:Seq[FormatElement]){
  def format(o:AnyRef):String = toks.map(_.format(o)) mkString ""
}

trait FormatElement {
  def format(o:AnyRef):String
}
case class Literal(str:String) extends FormatElement{
  def format(o:AnyRef):String = str
}
case class ToStringConversion(exp:Exp) extends FormatElement{
  def format(o:AnyRef):String = exp.eval(o).toString
}
case class DateConversion(exp:Exp,format:String) extends FormatElement{
  val df = new java.text.SimpleDateFormat(format)
  def format(o:AnyRef) = df.format(exp.eval(o) match {
    case cal:java.util.Calendar => cal.getTime
    case date:java.util.Date => date
  })
}
case class Conditional(condition:Exp
                      ,thenToks:FormatElementList
                      ,elseToks:FormatElementList) extends FormatElement{
  def format(o:AnyRef) = condition.eval(o) match {
    case java.lang.Boolean.TRUE => thenToks.format(o)
    case java.lang.Boolean.FALSE => elseToks.format(o)
    case x:Option[AnyRef] => x.map(thenToks.format)
                              .getOrElse(elseToks.format(o))
  }
}
case class Expand(exp:Exp,sep:String,inner:FormatElementList)
     extends FormatElement{
  def realEval(l:Iterable[AnyRef]):String = l.map(inner.format(_))
                                                       mkString sep
  import Java.it2it
  def format(o:AnyRef) = exp.eval(o) match{
    // array or collection or similar
    case l : java.lang.Iterable[AnyRef] => realEval(l)
    case l : Seq[AnyRef] => realEval(l)
  }
}

case class Exp(identifier:String){
  import java.lang.reflect.{Method}
  def method(cl:Class[_]):Method = // stripped
  def returnType(callingCl:Class[_]):Class[_] = method(callingCl)
                                                         .getReturnType
  def genericReturnType(callingCl:Class[_]):java.lang.reflect.Type =
       method(callingCl).getGenericReturnType
  def eval(o:AnyRef) = method(o.getClass).invoke(o,null)
}
case object ThisExp extends Exp(""){
  override def eval(o:AnyRef) = o
  override def returnType(callingCl:Class[_]):Class[_] = callingCl
  override def genericReturnType(callingCl:Class[_]):java.lang.reflect.Type

}
case class ParentExp(inner:Exp,parent:String) extends Exp(parent){
  override def eval(o:AnyRef) = inner.eval(super.eval(o))
}
```

Listing 4.3: AST and interpreter of Objectformatter

As shown in listing 4.3, most of the nonterminal symbols of the grammar, i.e. every one that is not representable as a string, is encoded by a Scala **`class`**. More precisely, every nonterminal symbol on the left-hand side of the grammar, which is defined through alternatives on the right-hand side, is encoded as an (abstract) class, while every alternative is encoded as a subclass of this class.

At the top of the hierarchy is `FormatElementList` which corresponds to the grammar element `format`. It contains a list of `FormatElements`. `FormatElement` is a **`trait`**, i.e. an abstract Scala class, which is the base class for all possible format elements (see `formatElement` in the grammar). Like `FormatElementList`, it declares a method `format` whose implementation actually defines the semantics of the element in the interpreter.

First, there is the `Literal` which takes the literal string as an argument. Its `format` method just returns this literal string.

Second, there is `ToStringConversion` which corresponds to the simplest form of inserting a property value into a format: When evaluated it converts the calculated value of the expression argument `exp` into a string by calling its `toString` method.

Third, there is `DateConversion`, which accepts an expression as argument that should evaluate to a date (`java.util.Date` or `java.util.Calendar`), and a date format string as described before.

Fourth, there is `Conditional`, which has the test expression as argument, which should evaluate into a `Boolean` or `Option` value. For both possible outcomes there is an argument, `thenToks` and `elseToks`, which specifies the format to be used in each case.

Fifth, there is `Expand`, which has the collection- or array-valued expression to be expanded as argument, as well as the separator string and the `FormatElementList` to be used to format the elements.[1]

Finally, there are **`case class`**`es` for expressions. Each expression class has to define a method `eval`, which, given the receiver object, evaluates the expression. The base case is `Exp` which accepts one property identifier as argument. As one can see in listing 4.3, its `eval` method is implemented in terms of a helper method `method` (which here is stripped for space reasons), that, given the class of an object and the identifier, finds a suitable method of the class to call. `eval` then invokes that method with the passed receiver object `o`.

`ThisExp` is the equivalent AST element of `#`**`this`**. Its `eval` method just returns its argument.

Finally, there is `ParentExp`, which encodes the case of property accesses with the dot notation. Aside from the `parent` argument, the parent identifier, which corresponds to one property path element on the left side of the dot and which is directly passed to the `Exp` constructor, it has another argument, `inner`, which corresponds to the expression on the right side of the dot. The implementation of `eval` is straightforward in that it first delegates the calculation of its parent expression to its super class `Exp` and then passes the result into the `inner` expression.

As you can see in listing 4.4, `"#`**`class`**`.simpleName.length"` parses into nested instances of `ParentExps`. The nesting is done as if the dot operator was right-associative.

[1]One may have noticed, that except for `Literal` every `FormatElement` specifies an expression and a conversion, so one may wonder, why the AST does not reflect this. Actually, it is owed to the evolution of the Objectformatter and may be changed in future but was not done yet due to time constraints.

This may seem not very intuitive, since the normal evaluation order is from left to right in dot notation and normally in nested expressions from inner to outer, but in the case of `ParentExp` the order is inverted. This is due to a constraint of the backtracking nature of the Scala parser combinators library. It requires that a grammar avoids left-recursive productions. [21, p.645] It is possible to workaround those restrictions, but the chosen way seemed the simplest.

```
> val parser = net.virtualvoid.string.EnhancedStringFormatParser

> parser.parse("a literal string")
res = FormatElementList(List(Literal(a literal string)))

> parser.parse("a property value: #class")
res = FormatElementList(List(Literal(a property value: ),
                        ToStringConversion(Exp(class))))

> parser.parse("#class.simpleName.length")
res = FormatElementList(List(ToStringConversion(
                           ParentExp(ParentExp(Exp(length),simpleName),class))))
```

Listing 4.4: Example AST output of the Objectformatter parser

4.4 Compilation

The `FormatCompiler` is an instance of `IObjectFormatterFactory`. Its `formatter` method uses MNEMO to compile the AST into an anonymous function, which then is returned, wrapped into an instance of `IObjectFormatter`:

```
object FormatCompiler extends IObjectFormatterFactory{
  def formatter[T<:AnyRef](clazz:Class[T],fmt:String) =
    new IObjectFormatter[T]{
      val compiler = Compiler.compile[T](fmt,clazz)
      def format(o:T):String = compiler(o)
    }
}
```

The basic structure of the compiled function is the following (The items in brackets are implicitly done by MNEMO):

1. (Put the current receiver from local 1 onto the stack.)
2. Put the current receiver object into local 0.
3. Instantiate a new `java.lang.StringBuilder`.
4. Evaluate each format element and append the result to the `StringBuilder` instance.

5. Call `StringBuilder.toString` to prepare the final result string.
6. (Return the top element of the stack.)

This is implemented in the `compile` method:

```
def compile[T<:AnyRef](format:String,cl:Class[T]): T => java.lang.String = {
  val elements:FormatElementList = EnhancedStringFormatParser.parse(format)
  ASMCompiler.compile(cl)(
    _
    ~ local[_0,T].store()
    ~ newInstance(classOf[StringBuilder])
    ~ compileFormatElementList(elements,cl)
    ~ invokemethod1(_.toString)
  )
}
```

Method `compile` is polymorphic in the receiver type `T` and takes two arguments: The format string and the runtime class of the object `Class[T]`. The runtime class is needed to dispatch the property accesses and generate proper method invocations. `compile` returns a function which, given an object of type `T`, returns the string representation. The method's implementation first parses the format into the AST and then calls MNEMO to generate and instantiate the function class.

The `ASMCompiler.compile` is called with a sequence of bytecode instructions. Each of them corresponds to one of the steps 2-5 from above. The bytecodes representing the evaluation of the `FormatElementList` are created by the method `compileFormatElementList`.

Given some `FormatElement` and the class of the receiver object `cl`, `compileFormatElementList` has to accept a frame with a `StringBuilder` on top of the stack and a value of type T in local variable 0. It should append the evaluated values of the `FormatElements` to the `StringBuilder` and return the stack in exactly the same form. Thus, the signature of the `compileFormatElementList` method is given as:

```
def compileFormatElementList[R<:List,LR<:List,T<:java.lang.Object]
      (elements:FormatElementList,cl:Class[T])
      (f:F[R**StringBuilder,LR**T])
      :F[R**StringBuilder,LR**T]
```

As said before, it is `compileFormatElementList`'s task to compile each element into its bytecode form. Another function is needed to compile *each* of the elements: `compileElement`. Considering its task, it must have the same signature as `compileFormatElementList`, just with the `elements` argument replaced by an argument of type `FormatElement`.

```
def compileElement[R<:List,LR<:List,T<:java.lang.Object]
                  (ele:FormatElement,cl:Class[T])
                  (f:F[R**StringBuilder,LR**T])
                  :F[R**StringBuilder,LR**T]
```

Given this function, compileFormatElementList can be implemented by chaining compileElement invocations, in which each element of the FormatElementList is passed into compileElement. This can concisely be written as a fold operation:

```
def compileFormatElementList[R<:List,LR<:List,T<:java.lang.Object]
    (elements:FormatElementList,cl:Class[T])
    (f:F[R**StringBuilder,LR**T])
    :F[R**StringBuilder,LR**T] =
  elements.elements.foldLeft(f){(frame,element) =>
                                compileElement(element,cl)(frame)}
```

which paraphrased reads: Starting from a frame state f, call compileElement for each of the elements passing in the result of the particular last call.

compileElement itself is implemented by matching over the type of format elements. On top of the stack is the StringBuilder instance and in local 0 is the object of type T.

In the following paragraphs we will give details about the compilation of particular FormatElements.

4.4.1 Literal

The simple case of Literal is shown here:

```
case Literal(str) =>
    f ~ ldc(str) ~ invokemethod2(_.append(_))
```

The literal string is just loaded from the constant pool (the constant pool is managed by MNEMO and its backend) and appended to the StringBuilder instance. StringBuilder.append returns its receiver, so it is on the stack again after the call.

In contrast to the Literal format element, the conversion format elements, ToStringConversion, DateConversion, etc., all need to generate bytecodes to evaluate expressions. This is accomplished by a function compileGetExp with following signature (its implementation will be shown later):

```
def compileGetExp[R<:List,LR<:List,T,Ret](exp:Exp
                                          ,cl:Class[T]
                                          ,retType:Class[Ret])
                                          (f:F[R**T,LR])
                                          :F[R**Ret,LR]
```

Given an expression, exp, the class of the receiver, cl, and the expected return type of the expression, compileGetExp generates bytecodes to evaluate the expression with the receiver of type T on top of the stack into the expression's value of type Ret which is on top of the stack afterwards.

With compileGetExp at hand, one is able to define the compilation **case** of ToStringConversion:

```
case ToStringConversion(e) =>
  f ~ local[_0,T].load() ~
      compileGetExp(e,cl,classOf[AnyRef]) ~
      invokemethod1(_.toString) ~
      invokemethod2(_.append(_))
```

First, the receiver object is loaded from local 0 onto the top of the stack. Then, compileGetExp is called to insert the bytecodes to actually evaluate the expression. Since the actual return type of the expression is neither known nor needed, classOf[AnyRef] is passed as return type. AnyRef is Scala's equivalent to java.lang.Object, so it is possible to call Object.toString method to convert the expression value into a string. Finally, this string is appended to the StringBuilder.

4.4.2 DateConversion

The next format element to compile is DateConversion. For the sake of simplicity we just translate the model from the interpreter to the compiler and, therefore, use java.text.SimpleDateFormat to convert the date.

```
case DateConversion(exp,format) => {
  val retType = exp.returnType(cl)

  f ~ newInstance(classOf[java.text.SimpleDateFormat]) ~
      dup ~
      ldc(format) ~
      invokemethod2(_.applyPattern(_)) ~ pop_unit ~
      local[_0,T].load() ~
      (f =>
        if (classOf[java.util.Date].isAssignableFrom(retType))
          f ~ compileGetExp(exp,cl,classOf[java.util.Date])
        else if (classOf[java.util.Calendar].isAssignableFrom(retType))
          f ~ compileGetExp(exp,cl,classOf[java.util.Calendar])
            ~ invokemethod1(_.getTime)
        else
          throw new java.lang.Error("Expected date- or calendar-typed property, " +
            cl.toString + " is unsupported for date conversion.")
      ) ~
      invokemethod2(_.format(_)) ~
      invokemethod2(_.append(_))
}
```

The first four bytecode instructions initialize the SimpleDateFormat for its use. Then, the object is loaded from local 0. As explained before, DateConversion can convert java.util.Dates as well as java.util.Calendars. The actual type of the expression (retType) is known when compiling the format, since both are known, the type of the receiver object passed into the formatter and each of the property's or method's return types. Therefore, we can emit just the bytecodes needed to convert the expression's particular type by using the **if** statement to generate the needed bytecodes. In particular, this means that if the type is java.util.Date, just the expression is evaluated, otherwise, if a java.util.Calendar is expected, it has to be converted into a java.util.Date by calling Calendar.getTime, because SimpleDateFormat only supports the formatting of java.util.Dates.

Note that, since it is currently not possible to save any state in the generated function object, we recreate the SimpleDateFormat object each time. As SimpleDateFormat pre-compiles the pattern into an internal format when a pattern is applied[2], a better solution would be to initialize a private field of the function object in the function's constructor and use this value later on. Because this functionality is not currently not available in MNEMO, we settled on this simple implementation.

4.4.3 Conditional

The compilation of Conditional is split up into the two cases of expressions allowed in the conditional, Boolean and Option[_] valued expressions.

A Boolean conditional is compiled by (1) loading the receiver, (2) evaluating the condition expression, and (3) branching to the code responsible for formatting the *then* or *else* branches of the Conditional depending on the value of condition expression:

```
case Conditional(inner,thens,elses) => {
  val retType = inner.returnType(cl)

  if (retType == java.lang.Boolean.TYPE ||
       classOf[java.lang.Boolean].isAssignableFrom(retType)){
    f ~
      local[_0,T].load() ~
      (if (retType == java.lang.Boolean.TYPE)
        compileGetExp(inner,cl,classOf[Boolean])
       else
        compileGetExp(inner,cl,classOf[java.lang.Boolean]) _
           ~ invokemethod1(_.booleanValue)
      ) ~
      ifne2(
        compileFormatElementList(thens,cl),
        compileFormatElementList(elses,cl))
  }
  else if // ...
}
```

[2]See the compile method of http://hg.openjdk.java.net/jdk7/jdk7/jdk/file/2113813eda62/src/share/classes/java/text/SimpleDateFormat.java

There is an inner **if** expression needed to convert a possible boxed java.lang.Boolean into a primitive value. The branching is done through the high-level ifne2 instruction of MNEMO which takes care of all the jump instructions needed.

In constrast, a Option[_] valued expression has to be treated specially, because of the change of the #**this** value in the case of its value being Some(x):

```
case Conditional(inner,thens,elses) => {
  val retType = inner.returnType(cl)

  if (retType == java.lang.Boolean.TYPE ||
        classOf[java.lang.Boolean].isAssignableFrom(retType)){
    // ...
  else if (classOf[Option[AnyRef]].isAssignableFrom(retType)){
    val eleType = elementType(inner.genericReturnType(cl)
                        ,classOf[Option[_]]).asInstanceOf[Class[AnyRef]]
      f ~
        local[_0,T].load() ~
        compileGetExp(inner,cl,classOf[Option[AnyRef]]) ~
        dup ~
        invokemethod1(_.isDefined) ~
        ifne2(
          _ ~
            checkcast(classOf[Some[AnyRef]]) ~
            invokemethod1(_.get) ~
            local[_0,T].load() ~
            swap ~
            local[_0,AnyRef].store() ~
            swap ~
            compileFormatElementList(thens,eleType) ~
            swap ~
            local[_0,T].store[R**StringBuilder,LR**AnyRef](),
          _ ~ pop ~ compileFormatElementList(elses,cl))
  }
```

The first instructions load the current receiver, evaluate the expression, duplicate the Option value on the stack, and then call Option.isDefined to decide which branch to take.

In the case of None, the duplicated Option on the stack is discarded with pop and the string defined by the elses format elements is appended.

In the case of Some(x), x has to be loaded and then the thens format elements have to be appended with the #**this** parameter in local 0. To achieve this, several steps are necessary. First, we can safely cast to Some, because of Option.isDefined being **true**. Now it is possible to extract the Option's value using the get method. The following load, swap, and store instructions save the current receiver on the stack and put x into local 0, where the subsequent invocation of compileFormatElementList expects it. After the call the saved receiver object is put back into local 0.

The compiler has to know the element type of `Option` to correctly resolve property accesses in the `Some(x)` branch of the generation. This turns out as quite a problem, since the element type of an `Option` is defined as its type parameter and generic types are erased during compilation. So, given an `Option` value, it is impossible at runtime to figure out which element type it has. Instead, the generic type information of method signatures is available through `java.lang.reflect.Method.getGenericReturnType`. Since every expression aside from `ThisExp` represents a method call, it is possible to extract this information.

4.4.4 Expand

Expansion is possible for `java.lang.Iterable`s and native Java arrays. We show here only the compilation of arrays. The compilation of `java.lang.Iterable`s is similar.

In Scala pseudo-code an expansion could look like this:

```
val elements:Array[T]
val format:T=>String
val separator:String

var sb = new StringBuilder
for(ele <- elements){
  if (sb.length != 0)
    sb = sb.append(separator)
  sb = sb.append(format(ele))
}
```

Given an array of elements of type `T` and a format function of type `T=>String`, one can start with a new `StringBuilder` and iterate over all elements of the array, appending each formatted element to the `StringBuilder`. If the `StringBuilder` is not empty, the separator string is appended before the formatted element.

This scheme can also be represented as a fold operation:

```
elements.foldLeft(new StringBuilder)(
   (sb,element) => (if (sb.length==0) sb else sb.append(separator))
                     .append(format(element)))
```

When compiling the expansion we make use of this representation and use the high-level MNEMO bytecode function `foldArray`:

```
case Expand(exp,sep,inner) => {
  val retType = exp.returnType(cl)
  if (retType.isArray){
    val eleType:Class[AnyRef] = retType.getComponentType
                                       .asInstanceOf[Class[AnyRef]]
```

```
      f ~  //sb | o
        local[_0,T].load() ~ // sb,o
        dup ~ //sb,o,o
        compileGetExp(exp,cl,retType.asInstanceOf[Class[Array[AnyRef]]]) ~
        newInstance(classOf[StringBuilder]) ~
        foldArray( // sb,o,index,sb,ele | array
            // add separator if nothing was added yet
          _ ~
            swap ~
            dup ~
            invokemethod1(_.length) ~
            ifne2(
              _ ~ ldc(sep) ~ invokemethod2(_.append(_)),
              id
            ) ~
            swap ~ // sb,o,index,sb,ele | array

            // format element
            swapTopWithLocal0 ~ //sb,o,index,sb,array | ele
            swap ~
            compileFormatElementList(inner,eleType) ~
            swap ~
            local[_0,Array[AnyRef]].store() ~ id
        ) ~ // sb,o,sb | array
        swap ~ // sb,sb,o | array
        local[_0,T].store() ~
        invokemethod2(_.append(_))
    }
  }
```

First, the receiver is loaded from local variable 0, duplicated on the stack to restore it later and the expression is evaluated. Then a new instance of `java.lang.StringBuilder` is created. Now it is possible to call `foldArray` which expects the array and the start value, in our case the `StringBuilder`, on the stack. `foldArray` handles the iteration and calls the inner bytecode fragment (the loop body) with a frame, which has the current iteration result and the current element on top of the stack and the array in local 0 (The expected state of the frame is written as comments in the snippet).

In the loop body it must first be decided if the separator string must be added. Thus, `StringBuilder.length` is called to determine its current content and append the separator string if needed. Afterwards the element is formatted and appended. Some swapping instructions safe the array on the stack during formatting the element, which expects the element in local 0. `swapTopWithLocal0` is a method here omitted which issues a short bytecode sequence to swap the top element of the stack with the contents of local 0.

After executing the fold, the receiver object is restored from the stack and the auxiliary `StringBuilder` is appended to the main `StringBuilder`.

Note that here, too, is no support for arrays of primitive values, since formatting of primitive values is currently not implemented.

4.4.5 Expression compilation

The last missing part is the compilation of the expression evaluation. It is implemented in `compileGetExp`:

```
def compileGetExp[R<:List,LR<:List,T,Ret](exp:Exp
                                        ,cl:Class[T]
                                        ,retType:Class[Ret])
                                    (f:F[R**T,LR])
                                    :F[R**Ret,LR] =
  exp match{
    case p@ParentExp(inner,parent) => {
      val parentMethod = p.method(cl)
      f ~ invokemethod1Dyn(parentMethod,classOf[AnyRef]) ~
      compileGetExp(inner
                   ,parentMethod.getReturnType
                                .asInstanceOf[Class[Object]]
                   ,retType)
    }
    case ThisExp =>
      f ~ checkcast(retType)
    case e:Exp => {
      f ~ invokemethod1Dyn(e.method(cl),retType)
    }
  }
```

The signature describes what the method does: Given an expression, the class of the receiver and a return type, the method generates a piece of bytecode which uses the receiver of type `T` on top of the stack to evaluate the expression, ensuring that the result is of type `Ret`.

The basic one is `Exp` (it has to be last in the list of cases, because `Exp` would match the other cases as well, since they are subclasses of `Exp`). It uses `invokemethod1Dyn` to call the method represented by the expression. The method is retrieved by calling the aforementioned `Exp.method` which looks it up in class `cl` using the rules as explained before. `invokemethod1Dyn` ensures at runtime of the generator, that parameter and return types fit and throws an exception otherwise.

`ThisExp` is compiled by casting the value into the correct type.

The `ParentExp` matching code first compiles the outer expression analogously to the `Exp` expression and afterwards calls `compileGetExp` recursively to evaluate the rest of the expression.

4.5 Conclusion

We introduced a small DSL to format objects into strings. A custom format syntax makes it possible to access, operate on, and insert properties of an object of a particular type into a string. Various format elements were presented along with their internal representation. It was shown how a bytecode compiler can be built by dissecting the AST of the format string with pattern matching and specifying each format element's bytecode representation in a type-safe manner. Bytecode generation can be decomposed by introducing methods for each of the available cases in a type-safe way and recomposed using the natural MNEMO operator.

Chapter 5

Evaluation

In this chapter we evaluate the presented bytecode generation library. Several aspects are examined: In the first part, it is checked to what extent the presented type-safe instruction representation indeed ensures structural constraints. In the second part, the usability of the domain specific language is evaluated. In the third part, we present a microbenchmark comparing the performance of interpreter and compiler of Objectformatter .

5.1 Structural constraints

It was one design goal of MNEMO that only valid instruction sequences should be expressable in MNEMO. In this chapter, we revisit the structural constraints (numbers refer to figure 2.2) that are imposed over instruction sequences and evaluate if and to what extent the library can prevent invalid uses of bytecode instructions.

Constraint 1 sums up the guiding principle for the other constraints. It can be split in two parts: First, an instruction must only be executed with operands having an appropriate type, and second, this has to be valid for all execution paths leading to an instruction.

For the first part, MNEMO uses the type encoding of types of stack and local variables to track relevant types throughout a bytecode sequence. An invalid instruction can not be applied (in general, the caveats and holes of the system are discussed later), since the ~ operator enforces matching types of frame state and instruction.

In this example, an integer operation is attempted with only strings on the stack:

```
val f : F[Nil**String**String,Nil] = null
f ~ iadd
```

This example fails to compile with this error (full type-qualification was removed and infix type notation restored):

```
error: polymorphic expression cannot be instantiated to expected type;
found   : [R <: List
          ,LT <: List]
          (F[R**Int**Int,LT]) => F[R**Int,LT]
required: (F[Nil**String**String,Nil]) => ?

      f ~ iadd
          ^
```

In the example, the found type refers to the type of the `iadd` instruction, the required type is the expected parameter type of `F.~`. The type inferer of Scala cannot infer the polymorphic types of the instruction, so that it would match the type of the given frame state and therefore rejects the code.

Another example, where a local variable is to be loaded from a position where no local variable is known, is shown here:

```
val f: F[Nil,Nil**Int] = null // only local 0 defined
f ~ local[_1,Int].load()      // try to load local 1 of typ Int
```

It fails with the following error:

```
error: no implicit argument matching parameter type
       CheckNTh[_1,Nil**Int,Int] was found.
      f ~ local[_1,Int].load()
                          ^
```

Here, the previously explained concept of using **`implicit`** parameters to constrain the applicability of an instruction prevents an invalid application of `load` on a local variable array.

Given the type encoding of the instructions in MNEMO is valid, the first part of constraint 1 can be fulfilled and ensured by the Scala compiler.

The second part of the constraint could be paraphrased as: "At the targets of jump instructions, frame state types of all incoming control flow paths have to be unified. Starting from this common super-frame-state-type subsequent instructions must match." In MNEMO, jump instructions are even constrained stricter than required: `jmp` requires a target with exactly the same frame state type as the current type, so that type unification is not needed at all. In the same way, the higher-level `ifne2` requires both possible control flow paths to result in exactly the same frame state type.

Constraint 2 (which is widening more than restricting) refers to the fact that integer types smaller than `int` are treated as `ints` internally and instructions working on `ints` may operate on smaller integer types as well. This fact is only exemplarily implemented in the conditional branching instructions (because `boolean` had to be supported) through the implicits and Scala views:

```
def ifne[R<:List,LT<:List,T<%JVMInt](inner:F[R,LT]=>Nothing)
                                    :F[R**T,LT] => F[R,LT]
```

which is expanded into

```
def ifne[R<:List,LT<:List,T](inner:F[R,LT]=>Nothing)
                            (implicit converter:T=>JVMInt)
                            :F[R**T,LT] => F[R,LT]
```

For each type possibly used as integer an **`implicit`** conversion has to be defined which converts the type into a `JVMInt`.

Constraint 3 concerning the depth of the stack is fulfilled with the strict handling of frame state types in branching instructions as described above.

Constraint 4 was explained above.

Constraint 5 is enforced through frame state types as explained above.

Constraints 6 to 9 are enforced by the `ASMCompiler` backend. `invokespecial` cannot be explicitly called, instead, the initialization constructs have to be used. The concept of uninitialized variables is not used in MNEMO.

Constraint 10 and 11 refer to the compability of the receiver's type and parameters' types with a given method signature. This is enforced by specifying methods to call as Scala function values. Because the Scala function values are contravariant in their input value types, it is possible to call methods that require values of a supertype of the values on the stack.

Constraint 12 is enforced since only reference typed return values are allowed on the stack in the code defining parameter of `ByteletCompiler.compile`. Explicit return instructions are not available.

Constraint 13 is enforced through the type of the array access instructions.

Constraint 14 is fulfilled, since the last instruction `ASMCompiler.compile` generates is always the return instruction.

Other structural constraints are mentioned in the JVM specificaton which were not yet treated.

So are `long` and `double` values handled specially in the JVM. They need two slots in the stack or in the local variables. This was not handled at all in MNEMO.

There is no support for member protection levels. Since member accesses are only possible through method calls, and methods are represented through closures, the protection level as indicated by the closed-over environment applies. This can lead to broken and rejected bytecode. Hence, when in the Scala interpreter console, it is not possible to invoke a method defined on the top level.

Field accesses, exceptions and subroutines are not supported, so constraints regarding only these cannot be broken.

MNEMO contains a test suite, which checks that expected constraints are adhered to on a syntactic level, as well as it checks that the runtime behaviour of generated classes is as expected.

5.2 Usability

The usability of the presented bytecode generation library may be examined from several perspectives. First, how difficult is it to extend MNEMO with additional bytecode instructions? Second and more important, how easy is it to use MNEMO, especially how much type annotations are needed, given the sophisticated type model?

Adding a new bytecode instruction to the DSL consists of

- figuring out, how to encode the type requirements of the new instructions in terms of the frame state type `F`,
- adding the instruction to `Instructions` and a new delegate to the frame state type `F`,
- adding an implementation to the compiler (and to the interpreter).

This simple scheme may only work for instructions similar to the prototype ones already implemented. Higher-level instructions may be created by composing the given set of primitive instructions into a composite instruction. Since most instructions should operate on general frame state types, the defining function must always be polymorphic in unused types of the stack or local variables. That forces the method definitions that specify an instruction to be extensive.

The usability of MNEMO in a client application was presented in the case study above. It was shown how a format string, which on a high-level specifies, how an object should

Type of format string	Interpreter	Compiler	Speed-up comp. vs interpreted
Literal String	30 ± 5	20.7 ± 2.7	1.45
Simple method access	86 ± 9	70.2 ± 2.0	1.22
Array expansion	608.0 ± 2.3	155.70 ± 0.84	3.90
Conditional/Option	85.43 ± 1.54	25.0 ± 1.6	3.41

Figure 5.1: Runtimes in ms for 100000 invocations of formatter.format, averaged over 20 runs. OpenJDK 1.6.0_0-b12 Server VM

be converted into a string, was compiled into one method that does the conversion. It was possible to decompose the generation into smaller steps. Each of the steps was implemented as a composite instruction. Type annotations were used when (1) calling `ASMCompiler.compile` to bootstrap the frame state type and (2) when defining composite instructions. Sometimes the Scala type inferer cannot correctly infer types (code blocks ending in `local.store()` instructions), then explicitly stating the type parameters of an instruction was necessary.

5.3 Performance

We developed a small benchmark suite to quantify the performance gain of a compiled format-string formatter over an interpreting one. Figure 5.1 shows the result, running the benchmark on an Intel Core 2 Duo 1.666 GHz. To mitigate effects of the Hotspot compiler starting to just-in-time compile code at the time a benchmark is executed, each test was run once with 1000000 iterations to "warmup". The actions of the just-in-time compiler and the garbage collector were monitored using the Java command-line switches `-XX:+PrintCompilation` and `-verbose:gc` as suggested in `http://wikis.sun.com/display/HotSpotInternals/MicroBenchmarks`.

For simple format-strings the compiled code seems only little faster than the interpreted one. For the more complex cases of array expansion and conditional format-strings, the speed-up is notable.

Chapter 6

Discussion

6.1 Heterogeneous lists vs. local variable encoding

The basic idea, which makes MNEMO feasible at all, is the concept of heterogenous lists, here implemented in the `List` classes. For the stack types this concept worked flawlessly and so it was readily extended to track types of local variables as well. The advantage is having only one concept for types of both, stack and local variables.

The requirement to access types from the local variable array by index in a general way, is not easily met. Especially the `store` instruction is difficult, because a type must potentially be saved deep in the type stack. Draft versions of MNEMO, first used another scheme to access locals. Accesses had to be enclosed in n-fold applications of extractor and packaging functions (e.g. `_.l.l.l.store.e.e.e` to access local 2). The disadvantage of this approach is its verbosity and missing possiblity to transform a instruction storing into the i-th local variable into an instruction storing into the $i+1$-th variable. Another attempt was using "type closures" to build up a list of types that were to be appended, after the `store` instruction placed the type at the correct position of the list of local variables types. This attempt was discarded because inner types could not be applied recursively, the very same issue, the current approach depends on as well.

After discovering the experimental compiler flag `-Yrecursive`, which solves this issue for now, we settled on the current approach. It has the advantage of being flexible because it uses the general numeral type. Using the visitor pattern to match over numeral types[19] proved to be an expressive way to reason about type lists particularly in MNEMO, but as well in general, because it is a prototype how to write type functions.

Aside from relying on an experimental compiler flag, the disadvantage of the approach was the need to add the `Top` and `Rest` type members to the `List` class. This did not work together with covariance annotations on `List`'s type parameters and in consequence that of `F` as well. The covariance annotations would have made a type `F[Nil**String,Nil]` assignable to `F[Nil**Object,Nil]` which, instead, would be more correct (e.g. see the definition of the partial order $\sqsubseteq$ on program point types in [23, p.9]). The effect of the change from covariance back to invariance could have been bigger, if method invocations, then, would have only been allowed to methods with the exact types of parameters currently on the stack. Fortunately this is avoided because we represent methods in MNEMO by Scala function value and these are declared contravariant in their input type parameters.

The current encoding of types of local variables as another type parameter next to the stack types, finally, has another effect: Although only little instructions actually operate on local variables (and hence, their types), every instruction has to state its ignorance towards them explicitly by creating a type parameter to channel it from input frame state type to output frame state type.

6.2 Method calls

Method invocations rely on another experimental (and undocumented), but, as seen, useful feature of the Scala compiler: `scala.reflect.Code`. The possibilities created by having access to the abstract syntax tree of an expression of a particular type is in this thesis only little explored. With this feature we can rely on the Scala compiler to infer receiver and parameter types from the frame state type, so that only little more code is necessary to generate a method invocation in comparison to an actual call, (`invokemethod1(_.append(_)` vs. `sb.append(str)`) and resolve the overloads of the method to call.

Several issues are worth noting here: First, as shown before, it is, of course, possible to create closures and, therefore, instances of `scala.reflect.Code` that do not represent a valid method call. This cannot be ruled out in any way but only rejected at compile-time of the generator. In future one could swap parameters on the stack automatically if a closure is passed which indicates this (e.g. `(a:String,b:StringBuilder) => b.append(a)`) or allow even more expressions to be expressed and generated in this way.

Another issue may arise because of the possible namespace mismatch (mentioned in 5.1 w.r.t. member protection levels). A closure and the resulting AST is calculated in the namespace they are defined in. But this is the namespace of the generator, and not all elements may be available in the classloader it is later loaded with.

Method invocation instructions are currently only available up to an arity of 2. Although an extension to 4 or 5 would suffice for most cases, Java allows methods with up to 255 parameters. Therefore, it would be nice if a general way could be found to support method invocations of arbitrary arity. One possibility would be to use the `curry` method of `scala.FunctionX`. Afterwards, the stack type list has to be unrolled (upon another type stack) up to the first parameter for the method call. In the final step, stack values could be consumed one-by-one in sync with the curried function representing the method.

6.3 Other limitations

MNEMO has several limitations that could not be addressed in the scope of this thesis. So called "category 2 types" are not supported. These are long and double values which need two slots in the stack or in the local variable array. It must be ensured that no slot of these two slots is treated separately from the other. This is difficult to achieve and was not even attempted.

Support for instantiation of objects is limited since only nullary and unary constructors may be called. To support constructors with more parameters, the current implementation to support allocation and constructor only as atomic unit could no longer be sustained. Instead, similar to the type model suggested in [23], an explicit type for uninitialized values would have to be created.

Creating special types, though, is difficult. They have to be created separately from the normal type hierarchy or their special status must be flagged in some way or another. Instead of using the normal type hierarchy as it is now, one could ultimately create a separate hierarchy by nesting each incoming type into a custom type.

Here, a trade-off had to be made between the characteristic simplicity of the current approach to use types from the normal Scala type hierarchy whenever possible, and a more sophisticated approach which would allow further constraints to be checked statically.

A practical limitation is the sometimes cryptic seeming output of the compiler once an error occurs. This stems partly from the fact that types in error messages are always fully qualified and type operators are (of course) not shown in infix notation. A simple tool could be written which simplifies error messages regarding MNEMO by stripping of qualification and restoring infix notation.

Chapter 7

Conclusion

In this thesis, it was shown how a type-safe bytecode generation library can be implemented in Scala. The result is a library of typed bytecode instruction primitives which can be combined into larger sequences of instructions in a type-safe and yet concise way.

Based on a heterogeneous type list, a frame state type was derived. It was shown how bytecode instructions can then be defined as transformations of frame state types.

For many simple instructions a straightforward translation of their type requirements could be found. For other ones, a more sophisticated encoding was necessary. Especially to define the types for local variable access, the complex type-system of Scala was hard to cope with in the beginning, but a satisfactory solution could be found later.

The restriction on the minimal framework might seem limiting but had the effect that the focus was on the encoding of instructions, instead of on generating a more complete surrounding class. The case study has shown that simple yet useful compilers can be built even with this minimal framework. Any data structure which represents only a single operation or transformation is a valid use case for compilation with MNEMO. That might be: A regular expression engine, evaluation of an XPath expression, compilation of parsers created with parser combinators, compilation of chains of higher-order function application (Scala's `Projections`).

In the end, the question remains if possible performance gains (in our cases between 3-fold and 4-fold increase of performance) are worthwhile the efforts of writing a bytecode compiler. With the JVM Hotspot compiler becoming better with every version, the speed-ups gained by compiling to bytecodes dwindle. However, some performance gain will probably always be possible, and with a tool at hand which simplifies bytecode generation and prevents mistakes the balance might still be on the side of bytecode generation.

7.1 Future work

The implementation of local variables using a type stack turned out as sometimes difficult to handle and prone to inducing compiler bugs. Sometimes type inference did not work in a reasonable manner making it necessary to annotate types where one would gladly do without them.

One idea is, to abandon the explicit concept of local variables in MNEMO altogether and replace it by typed and scoped storage tokens which are issued as needed. A storage token is just a Cell:

```
trait Cell[T]{
  def load[R]:F[R] => F[R**T]
  def store[R]:F[R**T] => F[R]
}
```

To use such a `Cell` one could think of a method like this:

```
def withLocal[ST,ST2,T](func: (Cell[T],F[ST]) => F[ST2]):F[ST**T] => F[ST2]
def withLocal[ST,ST2,T](tpe:Class[T],func: (Cell[T],F[ST]) => F[ST2]):F[ST] => F[ST2]
```

The first alternative would store the current value on top of the stack in the `Cell` and call the closure with it. The second alternative could be used to create an uninitialized cell of a given type `tpe`, which would allow uninitialized reads of `Cells` so it would be discouraged.

It could be used like that:

```
f ~
  bipush(5) ~
  withLocal((cell,f) =>
    f ~
      cell.load ~
      dup ~
      iadd ~
      cell.store
      // etc
  )
```

Since no local types would have to be tracked in a type stack anymore, one could completely do away with the local variables type in the frame type, which would simplify the declaration of any bytecode operation severely.

Bibliography

[1] *Homepage of BCEL*. `http://jakarta.apache.org/bcel/index.html`

[2] *New Java SE 6 Feature: Type Checking Verifier*. `https://jdk.dev.java.net/verifier.html`

[3] *JSR 45: Debugging Support for Other Languages*. `http://jcp.org/en/jsr/detail?id=45`. Version: November 2003

[4] *JSR 924: JavaTM Virtual Machine Specification*. `http://jcp.org/en/jsr/detail?id=924`. Version: July 2004

[5] *Clarifications and Amendments to the Java Virtual Machine Specification*. `http://java.sun.com/docs/books/jvms/second_edition/jvms-clarify.html`. Version: June 2005

[6] BRUNETON, Eric: *ASM 3.0 – A Java bytecode engineering library*. `http://download.forge.objectweb.org/asm/asm-guide.pdf`. Version: 2007

[7] CHIBA, Shigeru ; NISHIZAWA, Muga: An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. In: *In 2nd International coference on Generative Programming and Component Engineering (GPCE '03), volume 2830 of Springer Lecture Notes in Computer Science*, Springer-Verlag, 2003, S. 364–376

[8] CHURCH, A: A formulation of the simple theory of types. In: *The Journal of Symbolic Logic* (1940), Nr. 5, S. 55–68

[9] DÜRIG, Michael: *Meta-Programming with Scala Part I: Addition*. `http://michid.wordpress.com/2008/04/18/meta-programming-with-scala-part-i-addition/`. Version: April 2008

[10] FREUND, Stephen N. ; MITCHELL, John C.: A type system for object initialization in the java bytecode language. In: *ACM Transactions on Programming Languages and Systems*, 1998, S. 310–328

[11] FREUND, Stephen N. ; MITCHELL, John C. ; COLLEGE, Williams: *© 2003 Kluwer Academic Publishers. Printed in the Netherlands. A Type System for the Java Bytecode Language and Verifier*

[12] *JSR 202: JavaTM Class File Specification Update*. `http://jcp.org/aboutJava/communityprocess/final/jsr202/index.html`. Version: September 2006

[13] KISELYOV, Oleg ; LÄMMEL, Ralf ; SCHUPKE, Keean: Strongly typed heterogeneous collections / CWI, Amsterdam. 2004 (SEN-E0420). – Forschungsbericht

[14] MCBEATH, Jim: *Practical Church Numerals in Scala*. `http://jim-mcbeath.blogspot.com/2008/11/practical-church-numerals-in-scala.html`. Version: November 2008

[15] MOORS, Adriaan ; PIESSENS, Frank ; ODERSKY, Martin: Towards equal rights for higher-kinded types. In: *6th International Workshop on Multiparadigm Programming with Languages at the European Conference on Object-Oriented Programming (ECOOP*, 2007

[16] NEUBAUER, Matthias ; THIEMANN, Peter ; GASBICHLER, Martin ; SPERBER, Michael: A functional notation for functional dependencies. In: *Proceedings of the 2001 Haskell Workshop*, 2001, S. 101–120

[17] NORDENBERG, Jesper: *HList in Scala*. `http://jnordenberg.blogspot.com/2008/08/hlist-in-scala.html`. Version: August 2008

[18] NORDENBERG, Jesper: *HList in Scala revisited*. `http://jnordenberg.blogspot.com/2008/09/hlist-in-scala-revisited-or-scala.html`. Version: September 2008

[19] NORDENBERG, Jesper: *MetaScala*. `http://www.assembla.com/wiki/show/metascala`. Version: 2008

[20] ODERSKY, Martin ; ALTHERR, Philippe ; CREMET, Vincent ; DRAGOS, Iulian ; DUBOCHET, Gilles ; EMIR, Burak ; MCDIRMID, Sean ; MICHELOUD, Stéphane ; MIHAYLOV, Nikolay ; SCHINZ, Michel ; SPOON, Lex ; STENMAN, Erik ; ZENGER, Matthias: An Overview of the Scala Programming Language (2. Edition). 2006. – Forschungsbericht

[21] ODERSKY, Martin ; SPOON, Lex ; VENNERS, Bill: *Programming in Scala: A Comprehensive Step-by-step Guide*. 1st. Artima Inc, 2008. – ISBN 9780981531601

[22] ODERSKY, Martin ; ZENGER, Christoph ; ZENGER, Matthias: Colored local type inference. In: *ACM SIGPLAN Notices*, ACM Press, 2001, S. 41–53

[23] QIAN, Zhenyu: A formal specification of Java Virtual Machine instructions for objects, methods and subroutines. In: *Formal Syntax and Semantics of Java*, Springer-Verlag, 1998, S. 271–312

[24] STATA, Raymie ; ABADI, Martn: A type system for Java bytecode subroutines. In: *ACM Transactions on Programming Languages and Systems*, ACM Press, 1998, S. 149–160

[25] YELLIN, Tim Lindholm & F.: *The Java(tm) Virtual Machine Specification*. 2nd Edition. Addison-Wesley, 1999

Appendix A

Source code of MNEMO

The complete source code for this thesis is available under `http://virtual-void.net/files/mnemonics.zip`.

```
package net.virtualvoid.bytecode

object Bytecode{
  import java.lang.{String => jString,
                    Boolean => jBoolean
  }

  trait NatVisitor{
    type ResultType
    type Visit0 <: ResultType
    type VisitSucc[P<:Nat] <: ResultType
  }

  /* Peano-like natural numbers types */
  trait Nat{
    type Accept[V<:NatVisitor] <: V#ResultType
  }
  final class _0 extends Nat{
    type Accept[V<:NatVisitor] = V#Visit0
  }
  final class Succ[Pre<:Nat] extends Nat{
    type Accept[V<:NatVisitor] = V#VisitSucc[Pre]
  }

  type _1 = Succ[_0]
  type _2 = Succ[_1]
  type _3 = Succ[_2]
  type _4 = Succ[_3]

  val _0 = new _0
  val _1 = new _1
  val _2 = new _2
  val _3 = new _3
```

```

trait List{
  type Rest <: List
  type Top
}
trait Nil extends List{
  type Rest = Nil
  type Top = Nothing
}
object N extends Nil

case class Cons[R<:List,T](rest:R,top:T) extends List{
  type Rest = R
  type Top = T
  def l = rest
  /*def rest2:Rest = rest
  def top2:Top = top*/
}
// define an infix operator shortcut for the cons type
type ** [x<:List,y] = Cons[x,y]

// define the same for values
trait Consable[T<:List]{
  def **[U](next:U): T**U
}
implicit def conser[T<:List](t:T) = new Consable[T]{
  def **[U](next:U): T**U = Cons(t,next)
}

trait Target[ST<:List,LT<:List]
trait BackwardTarget[ST<:List,LT<:List] extends F[ST,LT] with Target[ST,LT]
trait ForwardTarget[ST<:List,LT<:List] extends Target[ST,LT]

case class JVMInt(v:Int){
  override def equals(o:Any) = v.equals(o)
}

trait F[ST<:List,LT<:List]{
  def depth = -1
  def frame = this

  def stack:ST
  def locals:LT

  def bipush(i1:Int):F[ST**Int,LT]
  def ldc(str:jString):F[ST**jString,LT]
  def target:BackwardTarget[ST,LT]
  def jmp(t:Target[ST,LT]):Nothing

  // support for forward declaring targets
  def forwardTarget[ST<:List,LT<:List]:ForwardTarget[ST,LT]
  def targetHere(t:ForwardTarget[ST,LT]):F[ST,LT]

  def ~[X](f:F[ST,LT]=>X):X = f(this)

  def iadd_int[R<:List](rest:R,i1:Int,i2:Int):F[R**Int,LT]
  def isub_int[R<:List](rest:R,i1:Int,i2:Int):F[R**Int,LT]
  def imul_int[R<:List](rest:R,i1:Int,i2:Int):F[R**Int,LT]
  def pop_int[R<:List](rest:R):F[R,LT]
  def dup_int[R<:List,T](rest:R,top:T):F[R**T**T,LT]
  def swap_int[R<:List,T1,T2](rest:R,t2:T2,t1:T1):F[R**T1**T2,LT]
  def dup_x1_int[R<:List,T1,T2](rest:R,t2:T2,t1:T1):F[R**T1**T2**T1,LT]
  def method1_int[R<:List,T,U](rest:R
                              ,top:T
                              ,code:scala.reflect.Code[T=>U])
                              :F[R**U,LT]
  def method1Dyn_int[R<:List,T,U](rest:R
                                 ,top:T
                                 ,method:java.lang.reflect.Method
                                 ,resCl:Class[U])
                                 :F[R**U,LT]
  def method2_int[R<:List,T2,T1,U](rest:R
                                  ,top2:T2
                                  ,top1:T1
                                  ,code:scala.reflect.Code[(T2,T1)=>U])
                                  :F[R**U,LT]
  def checkcast_int[R<:List,T,U](rest:R,top:T)(cl:Class[U]):F[R**U,LT]
  def ifne_int[R<:List](rest:R,top:JVMInt,inner:F[R,LT] => Nothing):F[R,LT]
  def ifne2_int[R<:List,ST2<:List,LT2<:List](rest:R
                                            ,top:JVMInt
                                            ,then:F[R,LT]=>F[ST2,LT2]
                                            ,elseB:F[R,LT]=>F[ST2,LT2])
  :F[ST2,LT2]
  def aload_int[R<:List,T](rest:R,array:AnyRef/*Array[T]*/,i:Int):F[R**T,LT]
  def astore_int[R<:List,T](rest:R,array:AnyRef,index:Int,t:T):F[R,LT]
  def arraylength_int[R<:List](rest:R,array:AnyRef):F[R**Int,LT]

  def tailRecursive_int[ST2<:List,LT2<:List]
      (func: (F[ST,LT] => F[ST2,LT2]) => (F[ST,LT]=>F[ST2,LT2]))
        (fr:F[ST,LT]):F[ST2,LT2]

  def pop_unit_int[R<:List](rest:R):F[R,LT]

  def newInstance[T](cl:Class[T]):F[ST**T,LT]

  def loadI[T](i:Int):F[ST**T,LT]
  def storeI[R<:List,T,NewLT<:List](rest:R,top:T,i:Int):F[R,NewLT]
}

case class CheckNTh[N<:Nat,L<:List,T]
implicit def nth_0[R<:List,T,U<:T]:CheckNTh[_0,R**U,T] = null
implicit def nthSucc[P<:Nat,R<:List,T,U](implicit next:CheckNTh[P,R,T])
  :CheckNTh[Succ[P],R**U,T] = null

case class Depth[P<:Nat](depth:Int)
implicit def depth_0:Depth[_0] = Depth[_0](0)
implicit def depthSucc[P<:Nat](implicit next:Depth[P]):Depth[Succ[P]] =
  Depth[Succ[P]](next.depth + 1)

/* it would be nice if we could abandon the () in declaration and application of
 * load/store altogether but that doesn't seems to work since then
 * type and implicit infering won't work any more
 */
trait LocalAccess[N<:Nat,T]{
  def load[ST<:List,LT<:List]()(implicit fn:CheckNTh[N,LT,T]
                                        ,depth:Depth[N])
                                        :F[ST,LT] => F[ST**T,LT]
  def store[ST<:List,LT<:List]()(implicit fn:Depth[N])
    :F[ST**T,LT] => F[ST,ReplaceNTh[N,LT,T]]
}

final class ReplaceNThVisitor[R<:List,T] extends NatVisitor{
  type ResultType = List
  type Visit0 = Cons[R#Rest,T]
  type VisitSucc[P<:Nat] = Cons[P#Accept[ReplaceNThVisitor[R#Rest,T]],R#Top]
}
type ReplaceNTh[N<:Nat,R<:List,T] = N#Accept[ReplaceNThVisitor[R,T]]

object Instructions {
  def local[N<:Nat,T]:LocalAccess[N,T] = new LocalAccess[N,T]{
    def load[ST<:List,LT<:List]()(implicit check:CheckNTh[N,LT,T]
                                          ,depth:Depth[N])
                                          :F[ST,LT] => F[ST**T,LT] =
      f => f.loadI(depth.depth)
    def store[ST<:List,LT<:List]()(implicit depth:Depth[N])
      :F[ST**T,LT] => F[ST,ReplaceNTh[N,LT,T]] =
      f => f.storeI(f.stack.rest,f.stack.top,depth.depth)
  }

  def iop[R<:List,LT<:List](func:(F[R**Int**Int,LT],R,Int,Int)=>F[R**Int,LT])
    :F[R**Int**Int,LT] => F[R**Int,LT] = f =>
      func(f,f.stack.rest.rest,f.stack.rest.top,f.stack.top)

  def iadd[R<:List,LT<:List] =
    iop[R,LT](_.iadd_int(_,_,_))
  def imul[R<:List,LT<:List] =
    iop[R,LT](_.imul_int(_,_,_))
  def isub[R<:List,LT<:List] =
    iop[R,LT](_.isub_int(_,_,_))

  def invokemethod1[T,U,R<:List,LT<:List](code:scala.reflect.Code[T=>U])
    :F[R**T,LT] => F[R**U,LT] =
    f => f.method1_int(f.stack.rest,f.stack.top,code)
  def invokemethod2[T1,T2,U,R<:List,LT<:List](code:scala.reflect.Code[(T1,T2)=>U])
    :F[R**T1**T2,LT] => F[R**U,LT] =
      f => f.method2_int(f.stack.rest.rest,f.stack.rest.top,f.stack.top,code)
  def invokemethod1Dyn[T,U,R<:List,LT<:List](method:java.lang.reflect.Method
                                            ,resT:Class[U])
                                            :F[R**T,LT] => F[R**U,LT] =
      f => f.method1Dyn_int(f.stack.rest,f.stack.top,method,resT)

  def pop_unit[R<:List,LT<:List]:F[R**Unit,LT] => F[R,LT] =
    f => f.pop_unit_int(f.stack.rest)

  def pop[R<:List,LT<:List,T]:F[R**T,LT]=>F[R,LT] = f=>f.pop_int(f.stack.rest)
  def dup[R<:List,LT<:List,T]:F[R**T,LT]=>F[R**T**T,LT] =
    f => f.dup_int(f.stack.rest,f.stack.top)
  def dup_x1[R<:List,LT<:List,T2,T1]:F[R**T2**T1,LT] => F[R**T1**T2**T1,LT] =
    f => f.dup_x1_int(f.stack.rest.rest,f.stack.rest.top,f.stack.top)
  def swap[R<:List,LT<:List,T2,T1]:F[R**T2**T1,LT] => F[R**T1**T2,LT] =
    f => f.swap_int(f.stack.rest.rest,f.stack.rest.top,f.stack.top)

  def checkcast[T,U,R<:List,LT<:List](cl:Class[U]):F[R**T,LT]=>F[R**U,LT] =
    f => f.checkcast_int(f.stack.rest,f.stack.top)(cl)

  def bipush[R<:List,LT<:List](i:Int):F[R,LT]=>F[R**Int,LT] = _.bipush(i)
  def ldc[R<:List,LT<:List](str:String):F[R,LT]=>F[R**String,LT] = _.ldc(str)

  def aload[R<:List,LT<:List,T]:F[R**Array[T]**Int,LT] => F[R**T,LT] =
    f => f.aload_int(f.stack.rest.rest,f.stack.rest.top,f.stack.top)
  def astore[R<:List,LT<:List,T]:F[R**Array[T]**Int**T,LT] => F[R,LT] =
    f => f.astore_int(f.stack.rest.rest.rest
                     ,f.stack.rest.rest.top
                     ,f.stack.rest.top
                     ,f.stack.top)
  def arraylength[R<:List,LT<:List,T]:F[R**Array[T],LT] => F[R**Int,LT] =
    f => f.arraylength_int(f.stack.rest,f.stack.top)

  implicit def int2JVMInt(i:Int) = JVMInt(i)
  implicit def bool2JVMInt(b:Boolean) = JVMInt(if (b) 1 else 0)

  def ifne[R<:List,LT<:List,T<%JVMInt](inner:F[R,LT]=>Nothing)
    :F[R**T,LT] => F[R,LT] =
      f=>f.ifne_int(f.stack.rest,f.stack.top,inner)
  def target[ST<:List,LT<:List] = (f:F[ST,LT]) => f.target
  def targetHere[ST<:List,LT<:List](t:ForwardTarget[ST,LT]) =
    (f:F[ST,LT]) => f.targetHere(t)
  def jmp[ST<:List,LT<:List](t:Target[ST,LT]) = (f:F[ST,LT]) => f.jmp(t)

  def newInstance[ST<:List,LT<:List,T](cl:Class[T]) =
    (f:F[ST,LT]) => f.newInstance(cl)

  def after[ST<:List,LT<:List](f:F[_,_]=>F[ST,LT]):F[ST,LT]=>F[ST,LT] = f => f

  def ifne2[R<:List,LT<:List,ST2<:List,LT2<:List,T<%JVMInt]
          (then:F[R,LT]=>F[ST2,LT2]
          ,elseB:F[R,LT]=>F[ST2,LT2]):F[R**T,LT]=>F[ST2,LT2] =
    f => f.ifne2_int[R,ST2,LT2](f.stack.rest,f.stack.top,then,elseB)
  def ifeq2[R<:List,LT<:List,ST2<:List,LT2<:List,T<%JVMInt]
          (then:F[R,LT]=>F[ST2,LT2],elseB:F[R,LT]=>F[ST2,LT2])
            :F[R**T,LT]=>F[ST2,LT2] =
    ifne2(elseB,then) // implemented in terms of ifne2

  def tailRecursive[ST<:List,LT<:List,ST2<:List,LT2<:List]
    (func: (F[ST,LT] => F[ST2,LT2]) => (F[ST,LT]=>F[ST2,LT2]))
      (fr:F[ST,LT]):F[ST2,LT2] =
      fr.tailRecursive_int(func)(fr)
}

object Implicits{
  implicit def richFunc[ST1<:List,ST2<:List,LT1<:List,LT2<:List]
                       (func:F[ST1,LT1] => F[ST2,LT2]):RichFunc[ST1,LT1,ST2,LT2] =
                         new RichFunc[ST1,LT1,ST2,LT2]{
    def apply(f:F[ST1,LT1]):F[ST2,LT2] = func(f)
  }
}
object RichOperations{
  import Instructions._
  import Implicits._
   /* def foldArray(array,func,start)
    *  let f(i,u) =
    *        if (i<array.length)
    *     f(i+1,func(u,ar[i]))
    *        else
    *           u
    *    f(0,start)
    */
    import Bytecode.Implicits._
  def foldArray[R<:List,LT<:List,T,U,X]
              (func:F[R**Int**U**T,LT**Array[T]]=>F[R**Int**U,LT**Array[T]])
              :F[R**Array[T]**U,LT**X] => F[R**U,LT**Array[T]] =
    _ ~
    swap ~
      local[_0,Array[T]].store() ~
    bipush(0) ~
    tailRecursive[R**U**Int,LT**Array[T],R**U,LT**Array[T]]{self =>
      _ ~
      dup ~
      local[_0,Array[T]].load() ~
      arraylength ~
      isub ~
      ifeq2(pop,
            _ ~
          dup_x1 ~
          local[_0,Array[T]].load() ~
          swap ~
          aload ~
          func ~
          swap ~
          bipush(1) ~
          iadd ~
          self
      )
    }
}

trait ByteletCompiler{
  def compile[T<:AnyRef,U<:AnyRef](cl:Class[T])(
                      code: F[Nil**T,Nil]
                       => F[Nil**U,_]
  ): T => U
}

trait RichFunc[ST1<:List,LT1<:List,ST2<:List,LT2<:List]
    extends (F[ST1,LT1] => F[ST2,LT2]){ first =>
  def ~[ST3<:List,LT3<:List](second:F[ST2,LT2]=>F[ST3,LT3])
    :RichFunc[ST1,LT1,ST3,LT3] = new RichFunc[ST1,LT1,ST3,LT3]{
    def apply(f:F[ST1,LT1]):F[ST3,LT3] = second(first(f))
  }
}
```

```scala
}

abstract class AbstractFunction1[T,U] extends Function1[T,U]
```

Listing A.1: Bytecode.scala: interface declaration

```scala
package net.virtualvoid.bytecode

import Bytecode._

import java.lang.{String=>jString}

object Interpreter extends ByteletCompiler{
    case class IF[ST<:List,LT<:List](stack:ST,locals:LT) extends F[ST,LT]{
      import CodeTools._

      def notImplemented(what:String) =
        new java.lang.Error(what + " not implemented in Interpreter")

      def bipush(i1:Int):F[ST**Int,LT] = IF(stack ** i1,locals)
      def ldc(str:jString):F[ST**jString,LT] = IF(stack ** str,locals)
      def target:BackwardTarget[ST,LT] = throw notImplemented("target")
      def jmp(t:Target[ST,LT]):Nothing = throw notImplemented("jmp")

      def forwardTarget[ST<:List,LT<:List]:ForwardTarget[ST,LT] =
        throw notImplemented("forwardTarget")
      def targetHere(t:ForwardTarget[ST,LT]):F[ST,LT] =
        throw notImplemented("targetHere")

      def iadd_int[R<:List](rest:R,i1:Int,i2:Int):F[R**Int,LT] = IF(rest ** (i1+i2),locals)
      def isub_int[R<:List](rest:R,i1:Int,i2:Int):F[R**Int,LT] = IF(rest ** (i1-i2),locals)
      def imul_int[R<:List](rest:R,i1:Int,i2:Int):F[R**Int,LT] = IF(rest ** (i1*i2),locals)
      def pop_int[R<:List](rest:R):F[R,LT] = IF(rest,locals)
      def dup_int[R<:List,T](rest:R,top:T):F[R**T**T,LT] =
        IF(rest**top**top,locals)
      def swap_int[R<:List,T1,T2](rest:R,t2:T2,t1:T1):F[R**T1**T2,LT] =
        IF(rest**t1**t2,locals)
      def dup_x1_int[R<:List,T1,T2](rest:R,t2:T2,t1:T1):F[R**T1**T2**T1,LT] =
        IF(rest**t1**t2**t1,locals)
      def method1_int[R<:List,T,U](rest:R,top:T,code:scala.reflect.Code[T=>U]):F[R**U,LT] =
        IF(rest ** invokeMethod(methodFromTree(code.tree),top).asInstanceOf[U],locals)
      def method1Dyn_int[R<:List,T,U](rest:R
                                      ,top:T
                                      ,method:java.lang.reflect.Method
                                      ,resCl:Class[U])
        :F[R**U,LT] = IF(rest ** method.invoke(top).asInstanceOf[U],locals)
      def method2_int[R<:List,T2,T1,U](rest:R
                                      ,top2:T2
                                      ,top1:T1
                                      ,code:scala.reflect.Code[(T2,T1)=>U]):F[R**U,LT] =
        IF(rest ** invokeMethod(methodFromCode(code),top2,top1).asInstanceOf[U],locals)
      def checkcast_int[R<:List,T,U](rest:R,top:T)(cl:Class[U]):F[R**U,LT] =
        IF(rest**top.asInstanceOf[U],locals)
      def ifne_int[R<:List](rest:R,top:JVMInt,inner:F[R,LT] => Nothing):F[R,LT] =
        throw notImplemented("ifeq_int")
      def ifne2_int[R<:List,ST2<:List,LT2<:List](rest:R
                                                 ,top:JVMInt
                                                 ,then:F[R,LT]=>F[ST2,LT2]
                                                 ,elseB:F[R,LT]=>F[ST2,LT2]):F[ST2,LT2] =
        if (top != 0) then(IF(rest,locals)) else elseB(IF(rest,locals))

      import java.lang.reflect.{Array => jArray}
      def aload_int[R<:List,T](rest:R,array:AnyRef,i:Int):F[R**T,LT] = {
        IF(rest**jArray.get(array,i).asInstanceOf[T],locals)
      }
      def astore_int[R<:List,T](rest:R,array:AnyRef,index:Int,t:T):F[R,LT] = {
        jArray.set(array,index,t)
        IF(rest,locals)
      }
      def arraylength_int[R<:List](rest:R,array:AnyRef):F[R**Int,LT] =
        IF(rest**jArray.getLength(array),locals)

      def pop_unit_int[R<:List](rest:R):F[R,LT] = IF(rest,locals)

      def get[T](i:Int,l:List):T = l match{
        case N => throw new Error("not possible")
        case Cons(r,t:T) => if (i == 0) t else get(i-1,r)
      }
      def store[T](i:Int,l:List,t:T):List = l match {
        case N => if (i == 0) Cons(N,t) else Cons(store(i-1,N,t),N)
        case Cons(r,old:T) => if (i == 0) Cons(r,t) else Cons(store(i-1,r,t),old)
      }

      def loadI[T](i:Int):F[ST**T,LT] = IF(stack**get(i,locals),locals)
      def storeI[R<:List,T,NewLT<:List](rest:R,top:T,i:Int):F[R,NewLT] =
        IF(rest,store(i,locals,top).asInstanceOf[NewLT])

      def newInstance[T](cl:Class[T]):F[ST**T,LT] =
        IF(stack**cl.newInstance,locals)

      def tailRecursive_int[ST2<:List,LT2<:List]
        (func: (F[ST,LT] => F[ST2,LT2]) => (F[ST,LT]=>F[ST2,LT2]))
          (fr:F[ST,LT]):F[ST2,LT2] =
          // classical y combinator in strict languages
          func(tailRecursive_int(func)_)(fr)
    }

    def compile[T,U](cl:Class[T])(code: F[Nil**T,Nil]=>F[Nil**U,_]): T => U =
      t => code(IF((N:Nil)**t,N)).stack.top
  }
```

Listing A.2: Interpreter.scala: The Bytecode interpreter

```
package net.virtualvoid.bytecode

import Bytecode._
import java.lang.{String=>jString}

object ASMCompiler extends ByteletCompiler{
    import _root_.org.objectweb.asm._
    import Opcodes._

```

```
  case class ClassStack(mrest:ClassStack,mtop:Class[_])
    extends Cons[ClassStack,Class[_]](mrest,mtop){
    def **(cl:Class[_]) = ClassStack(this,cl)

    def get(i:Int):Class[_] = if (i>0) rest.get(i-1) else top
    def set(i:Int,cl:Class[_]):ClassStack =
      if (i>0) ClassStack(rest.set(i-1,cl),top) else ClassStack(rest,cl)
  }
  class UnsetClassStack(lrest:ClassStack) extends ClassStack(lrest,null){
    override def get(i:Int):Class[_] =
      if (i==0)
        throw new Error("tried to get local which never was saved")
      else super.get(i)
    override def set(i:Int,cl:Class[_]):ClassStack =
      if (i>0) new UnsetClassStack(set(i-1,cl)) else ClassStack(this,cl)
  }
  case object EmptyClassStack extends UnsetClassStack(null)

  object JmpException extends RuntimeException

  class ASMFrame[ST<:List,LT<:List](mv:MethodVisitor
                                  ,stackClass:ClassStack
                                  ,localsClass:ClassStack)
                                    extends F[ST,LT]{
    def self[T]:T = this.asInstanceOf[T]

    val loopingList = new Cons(null.asInstanceOf[List],null){
      override val rest = this
      override val top = null
    }

    def stack = loopingList.asInstanceOf[ST]
    def locals = loopingList.asInstanceOf[LT]

    def newStacked[T](cl:Class[T]) =
      new ASMFrame[ST**T,LT](mv,stackClass**cl,localsClass)

    def bipush(i1:Int):F[ST**Int,LT] = {
      mv.visitIntInsn(BIPUSH, i1)
      newStacked(classOf[Int])
    }
    def ldc(str:jString):F[ST**jString,LT] = {
      mv.visitLdcInsn(str)
      newStacked(classOf[jString])
    }

    trait ASMTarget{
      def label:Label
    }

    case class ASMBackwardTarget[ST<:List,LT<:List](mv:MethodVisitor
                                                    ,stackClass:ClassStack
                                                    ,localsClass:ClassStack
                                                    ,label:Label)
        extends ASMFrame[ST,LT](mv,stackClass,localsClass)
          with BackwardTarget[ST,LT] with ASMTarget
    def target:BackwardTarget[ST,LT] = {
      val label = new Label
      mv.visitLabel(label)
      ASMBackwardTarget(mv,stackClass,localsClass,label)
    }

    case class ASMForwardTarget[ST<:List,LT<:List](label:Label)
      extends ForwardTarget[ST,LT] with ASMTarget
    def forwardTarget[ST<:List,LT<:List] = {
      val label = new Label
      ASMForwardTarget(label)
    }
    def targetHere(t:ForwardTarget[ST,LT]):F[ST,LT] = {
      mv.visitLabel(t.asInstanceOf[ASMForwardTarget[ST,LT]].label)
      this
    }

    def jmp(t:Target[ST,LT]):Nothing = {
      mv.visitJumpInsn(GOTO,t.asInstanceOf[ASMTarget].label)
      throw JmpException
    }

    def iadd_int[R<:List](rest:R,i1:Int,i2:Int):F[R**Int,LT] = {
      mv.visitInsn(IADD)
      new ASMFrame[R**Int,LT](mv,stackClass.rest,localsClass)
    }
    def isub_int[R<:List](rest:R,i1:Int,i2:Int):F[R**Int,LT] = {
      mv.visitInsn(ISUB)
      new ASMFrame[R**Int,LT](mv,stackClass.rest,localsClass)
    }
    def imul_int[R<:List](rest:R,i1:Int,i2:Int):F[R**Int,LT] = {
      mv.visitInsn(IMUL)
      new ASMFrame[R**Int,LT](mv,stackClass.rest,localsClass)
    }
    def pop_int[R<:List](rest:R):F[R,LT] = {
      mv.visitInsn(POP)
      new ASMFrame[R,LT](mv,stackClass.rest,localsClass)
    }
    def dup_int[R<:List,T](rest:R,top:T):F[R**T**T,LT] = {
      mv.visitInsn(DUP)
      new ASMFrame[R**T**T,LT](mv,stackClass**stackClass.top,localsClass)
    }
    def swap_int[R<:List,T1,T2](rest:R,t2:T2,t1:T1):F[R**T1**T2,LT] = {
      mv.visitInsn(SWAP)
      new ASMFrame[R**T1**T2,LT](mv
        ,stackClass.rest.rest**stackClass.top**stackClass.rest.top,localsClass)
    }
    def dup_x1_int[R<:List,T1,T2](rest:R,t2:T2,t1:T1):F[R**T1**T2**T1,LT] = {
      mv.visitInsn(DUP_X1)
      new ASMFrame[R**T1**T2**T1,LT](mv
        ,stackClass.rest.rest**stackClass.top**
          stackClass.rest.top**stackClass.top,localsClass)
    }
    def checkcast_int[R<:List,T,U](rest:R,top:T)(cl:Class[U]):F[R**U,LT] = {
      mv.visitTypeInsn(CHECKCAST, Type.getInternalName(cl));
      new ASMFrame[R**U,LT](mv,stackClass.rest**cl,localsClass)
    }
    def ifne_int[R<:List](rest:R,top:JVMInt,inner:F[R,LT] => Nothing):F[R,LT] = {
      val l = new Label
      mv.visitJumpInsn(IFEQ,l)

      try{
        inner(self)
      }
      catch{
        case JmpException =>
      }

      mv.visitLabel(l)
      new ASMFrame[R,LT](mv,stackClass.rest,localsClass)
    }
    import CodeTools._
    def aload_int[R<:List,T](rest:R,array:AnyRef,i:Int):F[R**T,LT] = {
      val elType = stackClass.rest.top.getComponentType

      mv.visitInsn(opcode(elType,IALOAD))

      new ASMFrame[R**T,LT](mv,stackClass.rest.rest ** elType,localsClass)
    }
    def astore_int[R<:List,T](rest:R,array:AnyRef,index:Int,t:T):F[R,LT] = {
      val elType = stackClass.rest.rest.top.getComponentType

      mv.visitInsn(opcode(elType,IASTORE))

      new ASMFrame[R,LT](mv,stackClass.rest.rest.rest,localsClass)
    }
    def arraylength_int[R<:List](rest:R,array:AnyRef):F[R**Int,LT] = {
      mv.visitInsn(ARRAYLENGTH)

      new ASMFrame[R**Int,LT](mv,stackClass.rest ** classOf[Int],localsClass)
    }
    def opcode(cl:Class[_],opcode:Int) =
      Type.getType(cleanClass(cl.getName)).getOpcode(opcode)

    def loadI[T](i:Int):F[ST**T,LT] = {
      val toLoad = localsClass.get(i)
      mv.visitVarInsn(opcode(toLoad,ILOAD), i);
      new ASMFrame[ST**T,LT](mv,stackClass**toLoad,localsClass)
    }
    def storeI[R<:List,T,NewLT<:List](rest:R,top:T,i:Int):F[R,NewLT] = {
      mv.visitVarInsn(opcode(stackClass.top,ISTORE), i);
      new ASMFrame[R,NewLT](mv,stackClass.rest,localsClass.set(i,stackClass.top))
    }

    def newInstance[T](cl:Class[T]):F[ST**T,LT] = {
      val cons = cl.getConstructor()
      mv.visitTypeInsn(NEW,Type.getInternalName(cl))
      mv.visitInsn(DUP)
      mv.visitMethodInsn(INVOKESPECIAL,Type.getInternalName(cl),"<init>"
                         ,Type.getConstructorDescriptor(cons))
      new ASMFrame[ST**T,LT](mv,stackClass**cl,localsClass)
    }

    def getInvokeInsn(m:java.lang.reflect.Method) =
      if ((m.getModifiers & java.lang.reflect.Modifier.STATIC) > 0)
        INVOKESTATIC
      else if (m.getDeclaringClass.isInterface)
        INVOKEINTERFACE
      else
        INVOKEVIRTUAL

    def method2_int[R<:List,T2,T1,U](rest:R,top2:T2,top1:T1
                                    ,code:scala.reflect.Code[(T2,T1)=>U]):F[R**U,LT] =
      invokeMethod2(methodFromCode(code))
    def method1Dyn_int[R<:List,T,U](rest:R
                                   ,top:T
                                   ,method:java.lang.reflect.Method
                                   ,resCl:Class[U]):F[R**U,LT] =
      invokeMethod(method)

    def invokeMethodX[R<:List,U](rest:ClassStack,m:java.lang.reflect.Method) = {
      val cl = m.getDeclaringClass
      mv.visitMethodInsn(getInvokeInsn(m),Type.getInternalName(cl)
                        ,m.getName
                        ,Type.getMethodDescriptor(m))
      new ASMFrame[R**U,LT](mv,rest ** m.getReturnType,localsClass)
    }
    def invokeMethod[R<:List,U](m:java.lang.reflect.Method) =
      invokeMethodX[R,U](stackClass.rest,m)
    def invokeMethod2[R<:List,U](m:java.lang.reflect.Method) =
      invokeMethodX[R,U](stackClass.rest.rest,m)

    def method1_int[R<:List,T,U](rest:R,top:T,code:scala.reflect.Code[T=>U]):F[R**U,LT] =
      invokeMethod(methodFromTree(code.tree))

    def pop_unit_int[R<:List](rest:R):F[R,LT] =
      new ASMFrame[R,LT](mv,stackClass.rest,localsClass)

    def ifne2_int[R<:List,ST2<:List,LT2<:List]
    (rest:R,top:JVMInt,then:F[R,LT]=>F[ST2,LT2],elseB:F[R,LT]=>F[ST2,LT2])
    :F[ST2,LT2] = {
      /*
       * ifeq thenLabel
       *   elseB
       *   jmp endLabel
       * thenLabel:
       *   elseB
       * endLabel:
       */

      val thenLabel = new Label
      val endLabel = new Label

      val frameAfterCheck = new ASMFrame[R,LT](mv,stackClass.rest,localsClass)

      mv.visitJumpInsn(IFNE,thenLabel)

      val afterElseFrame = elseB(frameAfterCheck)

      mv.visitJumpInsn(GOTO,endLabel)

      mv.visitLabel(thenLabel)

      val afterThenFrame = then(frameAfterCheck)

      mv.visitLabel(endLabel)

      if (afterElseFrame.isInstanceOf[InvalidFrame])
        if (afterThenFrame.isInstanceOf[InvalidFrame])
          throw new java.lang.Error("One execution path of ifeq2 must"+
                                    " have inferable types as output")
        else
          afterThenFrame
      else
        afterElseFrame
    }
    class InvalidFrame extends ASMFrame[Nothing,Nothing](null,null,null){
      override val toString = "invalid frame"
    }
    def invalidFrame[ST<:List,LT<:List]:F[ST,LT] =
      (new InvalidFrame).asInstanceOf[F[ST,LT]]
    def tailRecursive_int[ST2<:List,LT2<:List]
      (func: (F[ST,LT] => F[ST2,LT2]) => (F[ST,LT]=>F[ST2,LT2]))
      (fr:F[ST,LT]):F[ST2,LT2] = {
        val start = new Label
        mv.visitLabel(start)
        func {f =>
          mv.visitJumpInsn(GOTO,start)
          invalidFrame
        }(this)
    }
  }
  def classFromBytes(className:String,bytes:Array[Byte]):Class[_] = {
    new java.lang.ClassLoader{
      override def findClass(name:String):java.lang.Class[_] = {
        val fos = new java.io.FileOutputStream(name+".class")
        fos.write(bytes)
        fos.close
        defineClass(className,bytes,0,bytes.length);
      }
    }.loadClass(className)
  }
  var i = 0
  def compile[T<:AnyRef,U<:AnyRef](cl:Class[T])
  (code: F[Nil**T,Nil]=>F[Nil**U,_]): T => U = {
    i+=1
    val className = "Compiled" + i

    val cw = new ClassWriter(ClassWriter.COMPUTE_MAXS)
    cw.visit(V1_5,ACC_PUBLIC + ACC_SUPER,className,null
             ,"net/virtualvoid/bytecode/AbstractFunction1", null)

    { // constructor
      val mv = cw.visitMethod(ACC_PUBLIC, "<init>", "()V", null, null);
      mv.visitCode();
      mv.visitVarInsn(ALOAD, 0);
      mv.visitMethodInsn(INVOKESPECIAL
                         , "net/virtualvoid/bytecode/AbstractFunction1"
                         , "<init>"
                         , "()V");
      mv.visitInsn(RETURN);
      mv.visitMaxs(1, 1);
      mv.visitEnd();
    }

    { // apply
      val mv = cw.visitMethod(ACC_PUBLIC
                              , "apply"
                              , "(Ljava/lang/Object;)Ljava/lang/Object;"
                              , null
                              , null);
      mv.visitCode()
      // put the parameter on the stackClass
      mv.visitVarInsn(ALOAD, 1);
      mv.visitTypeInsn(CHECKCAST, Type.getInternalName(cl));

      code(new ASMFrame[Nil**T,Nil](mv,EmptyClassStack ** cl,EmptyClassStack))

      mv.visitInsn(ARETURN)
      mv.visitMaxs(1, 2)
      mv.visitEnd
    }
    cw.visitEnd
    classFromBytes(className,cw.toByteArray).newInstance.asInstanceOf[T=>U]
  }
}
```

Listing A.3: Compiler.scala: The Bytecode compiler

Appendix B

Source code of the format-string compiler

```
package net.virtualvoid.string

import java.lang.{StringBuilder,String=>jString}

object Compiler{
  import net.virtualvoid.bytecode.Bytecode
  import net.virtualvoid.bytecode.ASMCompiler
  import Bytecode._
  import Bytecode.Instructions._
  import Bytecode.Implicits._

  val parser = EnhancedStringFormatParser
  import AST._

  def elementType(it:java.lang.reflect.Type,of:Class[_])
    :Class[_ <: AnyRef] = {
    TypeHelper.genericInstanceType(it,of,Array()) match{
      case Some(cl:java.lang.Class[AnyRef]) => cl
      case _ => throw new java.lang.Error("Can't get element type of "+it)
    }
  }

  def compileGetExp[R<:List,LR<:List,T,Ret](exp:Exp
                                            ,cl:Class[T]
                                            ,retType:Class[Ret])
                                           (f:F[R**T,LR]):F[R**Ret,LR] =
  exp match {
    case p@ParentExp(inner,parent) =>{
      val m = p.method(cl)
      f ~ invokemethod1Dyn(m,classOf[AnyRef]) ~
       compileGetExp(inner,m.getReturnType.asInstanceOf[Class[Object]],retType)
    }
    case ThisExp =>
```

```
      f ~ checkcast(retType) // TODO: don't know why we need this, examine it
    case e:Exp => {
      f ~ invokemethod1Dyn(e.method(cl),retType)
    }
  }

  def compileFormatElementList[R<:List,LR<:List,T<:java.lang.Object]
                (elements:FormatElementList,cl:Class[T])
                (f:F[R**StringBuilder,LR**T]):F[R**StringBuilder,LR**T] =
    elements.elements.foldLeft(f){(frame,element) =>
      compileElement(element,cl)(frame)}

  def id[X]:X=>X = x=>x

  def compileElement[R<:List,LR<:List,T<:java.lang.Object]
                    (ele:FormatElement,cl:Class[T])
                    (f:F[R**StringBuilder,LR**T])
                    :F[R**StringBuilder,LR**T]
    = ele match {
      case Literal(str) =>
        f ~ ldc(str) ~ invokemethod2(_.append(_))
      case ToStringConversion(e) =>
        f ~ local[_0,T].load() ~
          compileGetExp(e,cl,classOf[AnyRef]) ~
          invokemethod1(_.toString) ~
          invokemethod2(_.append(_))
      case Expand(exp,sep,inner) => {
        val retType = exp.returnType(cl)

        if (classOf[java.lang.Iterable[_]].isAssignableFrom(retType)){
          val eleType:Class[AnyRef] = elementType(exp.genericReturnType(cl)
                                                  ,classOf[java.lang.Iterable[_]])
                                                  .asInstanceOf[Class[AnyRef]]
          val jmpTarget =
            f ~
             local[_0,T].load() ~
             swap ~ // save one instance of T for later
             local[_0,T].load() ~
             compileGetExp(exp,cl,classOf[java.lang.Iterable[AnyRef]]) ~
             invokemethod1(_.iterator) ~
             local[_0,java.util.Iterator[AnyRef]].store() ~
             target

          jmpTarget ~
             local[_0,java.util.Iterator[AnyRef]].load() ~
             invokemethod1(_.hasNext) ~
             ifne(f =>
               f ~
                local[_0,java.util.Iterator[AnyRef]].load() ~
                swap ~
                local[_0,java.util.Iterator[AnyRef]].load() ~
                invokemethod1(_.next) ~
                checkcast(eleType) ~
                local[_0,AnyRef].store() ~
                compileFormatElementList(inner,eleType) ~
                swap ~
                dup ~
                local[_0,java.util.Iterator[AnyRef]].store() ~
                invokemethod1(_.hasNext) ~
                ifne(f =>
                   f~ldc(sep:jString) ~
                    invokemethod2(_.append(_)) ~
                    jmp(jmpTarget)) ~ //todo: introduce ifeq(thenCode,elseTarget)
                jmp(jmpTarget)) ~
             swap ~
             local[_0,T].store[R**StringBuilder,LR**java.util.Iterator[AnyRef]]()
        }
        else if (retType.isArray){
          val eleType:Class[AnyRef] = retType.getComponentType.asInstanceOf[Class[AnyRef]]

          if (eleType.isPrimitive)
            throw new java.lang.Error("can't handle primitive arrays right now");

          import Bytecode.RichOperations.foldArray

          def swapTopWithLocal0[S<:List,L<:List,ST,LT]:F[S**ST,L**LT] => F[S**LT,L**ST] =
            _ ~
            local[_0,LT].load() ~
            swap ~
            local[_0,ST].store() ~ id

          f ~  //sb | o
            local[_0,T].load() ~ // sb,o
            dup ~ //sb,o,o
            compileGetExp(exp,cl,retType.asInstanceOf[Class[Array[AnyRef]]]) ~
            newInstance(classOf[StringBuilder]) ~
            foldArray( // sb,o,index,sb,ele | array
              // add separator if nothing was added yet
              _ ~
                swap ~
                dup ~
                invokemethod1(_.length) ~
                ifeq2(
                  f=>f,
                  _ ~ ldc(sep) ~ invokemethod2(_.append(_))
                ) ~
                swap ~ // sb,o,index,sb,ele | array

                // format element
                swapTopWithLocal0 ~ //sb,o,index,sb,array | ele
                swap ~
                compileFormatElementList(inner,eleType) ~
                swap ~
                local[_0,Array[AnyRef]].store() ~ id// sb,o,index,sb | array
            ) ~ // sb,o,sb | array
            swap ~ // sb,sb,o | array
            local[_0,T].store() ~
            invokemethod2(_.append(_))
        }
        else
          throw new java.lang.Error("can only iterate over "+
                                      "iterables and arrays right now")
```

```
      }
      case Conditional(inner,thens,elses) => {
        val retType = inner.returnType(cl)

        if (retType == java.lang.Boolean.TYPE ||
              classOf[java.lang.Boolean].isAssignableFrom(retType)){
          f ~
            local[_0,T].load() ~
            (if (retType == java.lang.Boolean.TYPE)
               compileGetExp(inner,cl,classOf[Boolean])
             else
               compileGetExp(inner,cl,classOf[java.lang.Boolean]) _
             ~ invokemethod1(_.booleanValue)
            ) ~
            ifeq2(
              compileFormatElementList(elses,cl),
              compileFormatElementList(thens,cl))
        }
        else if (classOf[Option[AnyRef]].isAssignableFrom(retType)){
          val eleType = elementType(inner.genericReturnType(cl)
                                    ,classOf[Option[_]])
            .asInstanceOf[Class[AnyRef]]
          f ~
            local[_0,T].load() ~
            compileGetExp(inner,cl,classOf[Option[AnyRef]]) ~
            dup ~
            invokemethod1(_.isDefined) ~
            ifeq2(
              _ ~ pop ~ compileFormatElementList(elses,cl),
              _ ~
                checkcast(classOf[Some[AnyRef]]) ~
                invokemethod1(_.get) ~
                local[_0,T].load() ~
                swap ~
                local[_0,AnyRef].store() ~
                swap ~
                compileFormatElementList(thens,eleType) ~
                swap ~
                local[_0,T].store[R**StringBuilder,LR**AnyRef]())
        }
        else
          throw new Error("can't use "+retType+" in a conditional")
      }
      case DateConversion(exp,format) => {
        val retType = exp.returnType(cl)

        val DateClass:Class[java.util.Date] = classOf[java.util.Date]
        val CalendarClass:Class[java.util.Calendar] = classOf[java.util.Calendar]

        f ~ newInstance(classOf[java.text.SimpleDateFormat]) ~
          dup ~
          ldc(format) ~
          invokemethod2(_.applyPattern(_)) ~ pop_unit ~
          local[_0,T].load() ~
          (f =>
            retType match {
              case x if DateClass.isAssignableFrom(x)     =>
                f ~ compileGetExp(exp,cl,DateClass)
              case x if CalendarClass.isAssignableFrom(x) =>
                f ~
                  compileGetExp(exp,cl,CalendarClass) ~
                  invokemethod1(_.getTime)
              case _ => throw new java.lang.Error(
                "Expected date- or calendar- typed property. "+
                cl+" can't be converted.")
            }
          ) ~
          invokemethod2(_.format(_)) ~
          invokemethod2(_.append(_))
      }
    }
  def compile[T<:AnyRef](format:String,cl:Class[T]):T=>jString = {
    val elements:FormatElementList = parser.parse(format)
    ASMCompiler.compile(cl)(
      _
      ~ local[_0,T].store()
      ~ newInstance(classOf[StringBuilder])
      ~ compileFormatElementList(elements,cl)
      ~ invokemethod1(_.toString)
     )
  }
}

object FormatCompiler extends IObjectFormatterFactory{
  def formatter[T<:AnyRef](clazz:Class[T],fmt:String) =
    new IObjectFormatter[T]{
      val compiler = Compiler.compile[T](fmt,clazz)
      def format(o:T):String = compiler(o)
      }
}
```

Listing B.1: FormatCompiler.scala: The format-string compiler

Appendix C

Step-wise expansion of ReplaceNTh

This is the step-wise expansion of `ReplaceNTh[_1,Nil**Float**Int**String,Double]`, which replaces the type at position 1 of the list with `Double`:

```
ReplaceNTh[_1,Nil**Float**Int**String,Double]

  // Definition of ReplaceNTh
  =  _1#Accept[ReplaceNThVisitor[Nil**Float**Int**String,Double]]

  // Definition of Succ
  =  Succ[_0]#Accept[ReplaceNThVisitor[Nil**Float**Int**String,Double]]

  // Definition of Succ.Accept
  =  ReplaceNThVisitor[Nil**Float**Int**String,Double]#VisitSucc[_0]

  // Definition of ReplaceNThVisitor.VisitSucc
  =  Cons[_0#Accept[ReplaceNThVisitor[Nil**Float**Int,Double]],String]

  // Definition of _0.Accept
  =  Cons[ReplaceNThVisitor[Nil**Float**Int,Double]#Visit0,String]

  // Definition of ReplaceNThVisitor.Visit0
  =  Cons[Cons[Nil**Float,Double],String]

  // Using infix notation
  =  Nil**Float**Double**String
```

In the current version of Scala, 2.7.3, the given type is not valid and rejected with a compiler error

```
illegal cyclic reference involving type Nat#Accept
```

This is due to a limitation in the type checking algorithm which allows no recursive application of inner types. Since Scala version 2.7.3 there is an experimental compiler flag `-Yrecursion=X` available which allows such type recursions up to a depth `X`. See `http://lampsvn.epfl.ch/trac/scala/ticket/1291` for more info [17, 18].