Mnemonics: Type-safe Bytecode Generation at Run Time

Johannes Rudolph
University of Freiburg, Germany
johannes.rudolph@googlemail.com

Peter Thiemann
University of Freiburg, Germany
thiemann@informatik.uni-freiburg.de

Abstract

Mnemonics is a Scala library for generating method bodies in JVM bytecode at run time. Mnemonics supports a large subset of the JVM instructions, for which the static typing of the generator guarantees the well-formedness of the generated bytecode.

The library exploits a number of advanced features of Scala's type system (type inference with bounded polymorphism, implicit parameters, and reflection) to guarantee that the compiler only accepts legal combinations of instructions at compile time. Additional instructions can be supported at the price of a check at run time of the generator. In either case, bytecode verification of generated code is guaranteed to succeed.

Categories and Subject Descriptors D.3 PROGRAMMING LANGUAGES [*D.3.4 Processors*]: Code generation; F.3 LOGICS AND MEANINGS OF PROGRAMS [*F.3.3 Studies of Program Constructs*]: Type structure

General Terms Design, Languages

1. Introduction

The Java Virtual Machine (JVM)[15] is a popular programming platform with compilers for many languages available. A compiler targeting the JVM must generate class files, which requires intimate knowledge of the class file format and of the JVM bytecode. A number of libraries alleviate this task by providing APIs to construct and change class files (e.g., [1, 3, 4]). Some of these libraries also support code generation at run time.

Before being loaded and executed by the JVM, the bytecode in a class file has to pass the bytecode verifier [25]. Roughly speaking, the bytecode verifier statically checks the type safety of a given sequence of bytecodes and rejects it if it fails to verify. Many existing bytecode generation libraries only guarantee correctly serialized class files, but the generated bytecode may still fail to pass the bytecode verifier. Some libraries have verifiers that check the generated bytecode after generation. Instead of producing ill-formed bytecode, such a library aborts the generation of bytecode.

This work introduces Mnemonics, a novel bytecode generation library written in Scala. It is geared towards the convenient generation of bytecode for methods at run time. Mnemonics encodes the constraints on the generated bytecode in the types of the generating program as much as possible. For a large subset of JVM instructions, the static typing of the generating program is sufficient to guarantee that the generated bytecode passes the verifier. Mnemonics achieves this guarantee by encoding the state of the JVM's run-time stack as a heterogeneous list where the compile-time type of the list is a fine grained abstraction of the contents of the stack at run time. Each instruction is a suitably typed function that transforms the stack and its type. To create a sequence of bytecode instructions requires combining instruction primitives with matching types. Thus, Scala's type system rejects illegal combinations at compile time by performing essential parts of the bytecode verification at the type level.

Mnemonics does not cover the entire JVM instruction set. Some complicated instructions (e.g., the switch-related instructions) are not supported. Some kinds of branch instructions are not directly available, but require the use of a type-safe pattern provided by the library. Some uses of instructions (e.g., an invocation of a method that is not accessible from the generator at compile time) require a dynamic check at generation time, but these checks are much cheaper than a full blown bytecode verification (typically just a subtype check). The set of supported instructions is chosen such that no bytecode verification is needed for a bytecode sequence returned by a Mnemonics generator.

Contributions

- We designed a Scala framework for the type-safe generation of methods in bytecode at run time.
- We implemented two instances of the framework, an interpreter and a bytecode compiler.
- We assessed the framework with a case study, the generation of formatting methods from a specification string.

Overview

Section 2 covers background on the JVM and on the constraints imposed on bytecode programs. Readers familiar with the JVM may safely skip this section. Section 3 introduces the functionality of the Mnemonics library with examples. The consideration of examples culminates in Section 4 with a case study that generates bytecode for string formatters at run time. Section 5 highlights some important details of the implementation and characterizes the set of supported instructions. Section 6 evaluates the Mnemonics library from different perspectives: correctness, usability, and performance. Section 7 discusses related work and Section 8 concludes.

The paper assumes familiarity with Scala [20, 21]. It is based on and extends Rudolph's diploma/master's thesis [26]. The thesis and the implementation are available on the web at `http://virtual-void.net/files/mnemonics.zip`.

2. Background: JVM Basics

The JVM is a stack-based abstract machine which constitutes the basis of the Java platform [15]. It executes programs written in Java bytecode, which serves as a portable target language mainly for compiling object-oriented programming languages like Java or

PEPM '10, January 18–19, 2010, Madrid, Spain
Copyright © 2010 ACM 978-1-60558-727-1/10/01...$10.00

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, [...] 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 1. Structural constraints on bytecode (excerpt) [15, §4.8.2].

Scala. Compiled code has the form of a *class file*, which contains a bytecode array for each method in a class.

The state of a thread in the JVM consists of a stack of frames, corresponding to the pending method calls. Each frame contains the data characterizing the state of a method call. It consists of an operand stack, a local variable (and parameter) array, and the program counter. The JVM includes instructions for stack manipulation as well as operations to load and store local variables by index.

The JVM is a typed machine which supports the (category 1) types integer, float, and reference data types as defined by the loaded program, as well as the (category 2) types long and double. The former types have a one-word representation, the latter require two words. There are specialized bytecode instructions for each kind of type. For example, the `areturn` instruction returns a reference value, `ireturn` an integer, `lreturn` a long, and `dreturn` a double value. Some instructions, such as `pop`, which pops a value off the stack, are applicable to operands of any category 1 type.

Bytecode has to obey structural constraints that concern the relationship between instructions. Figure 1 lists an excerpt of those constraints. The structural constraints ensure that a number of invariants hold while a program is executed, that all operands for an instruction are available, and that their types are correct. The JVM class-loading infrastructure contains the bytecode verifier, which guarantees adherence to the structural constraints. This verifier gives guarantees similar to those of a type system and can be described as such [8, 9, 24].

3. Mnemonics by Example

Let's start with generating bytecode for a function that takes a boxed integer and increments it. The bytecode for such a function first unboxes its argument by converting it to a primitive integer, pushes the constant 1 on the stack, adds the two integers on top of the stack, and boxes the result. Here it goes:

```
aload_0
invokevirtual #6 <Method int intValue()>
bipush 1
iadd
invokestatic #6 <Method java.lang.Integer valueOf(int)>
areturn
```

The corresponding generator looks quite similar to the bytecode:[1]

```
> val f = ASMCompiler.compile(
      classOf[Integer],classOf[Integer])(
      param => ret => frame => frame ~
      param.load ~
      method1((x:Integer) => x.intValue) ~
      bipush(1) ~
      iadd ~
      method1((x:Int) => Integer.valueOf(x)) ~
      ret.jmp)
f: (Integer) => Integer = <function>
```

Before delving into the explanation of the generator, let's first observe that the code generator `compile` always returns a function (in this case, an instance of an anonymous class that implements Scala's `Function1` trait[2]). Thus, the resulting function `f` can be used like any other function:

```
> f(0)
res: Integer = 1
> f(12)
res: Integer = 13
```

The `ASMCompiler.compile` function has the type signature[3]

```
  def compile[T1<:AnyRef,R<:AnyRef]
     (par1Cl:Class[T1],retCl:Class[R])
     (code: Local[T1] => Return[R] => F[Nil] => Nothing)
     : T1 => R
```

The two type parameters, `T1` and `R`, represent the argument and return type of the function to generate. The `code` argument is a function that takes a capability of type `Local[T1]` for accessing the parameter and another capability of type `Return[R]` for returning a value of type `R`. It then returns a *frame transformer* that creates the bytecode sequence. The `compile` function applies it to an initial frame of type `F[Nil]` and expects it to return `Nothing`.[4]

The frame type `F[STK]` models a stack of type `STK`. It is represented by a nested pair structure with pairing indicated by the infix type constructor `**` and the empty stack/list by the type `Nil`. Thus, in the type `F[Nil**T]`, the stack contains one entry of type `T`.

In the example, the frame transformer is an anonymous function with formal parameter `frame`, the initial frame corresponding to the prefix of a bytecode sequence. The left-associative ~ operator attaches a new instruction to the end of a bytecode sequence: it is defined as reverse function application.[5] The instruction `param.load` loads the parameter on the stack. Next,

[1] This snippet shows an interaction with Scala. The > character is Scala's prompt and the non-indented line is the response of the system.

[2] A trait is like a Java interface, but may contain code.

[3] Square brackets indicate formal and actual type parameters in Scala.

[4] `Nothing` is a special type in Scala which, when used as result type, denotes a function that never returns normally. See Sec. 5.5.

[5] Function composition would be nicer for combining frame transformers. However, Scala's type inference does not cope with this alternative setup.

method1 generates a call to a method specified by the function ((x:Integer) => x.intValue).[6] The subsequent instructions expect a primitive integer on top of the stack. Instruction bipush(1) pushes the integer 1 onto the stack resulting in two primitive integers on top of the stack. The iadd instruction consumes the two integer values, adds them, and pushes the result back on the stack.[7] The final method1 again identifies a method using a function. It converts the result back to a boxed integer. This value is returned to the caller with the return instruction generated by ret.jmp.

Scala's placeholder syntax for anonymous functions makes uses of method1 even more concise. An underline placeholder _ placed in an expression context automatically creates an abstraction. For instance, the syntax ((_:Integer).intValue) is a shorthand for ((x:Integer)=>x.intValue) and the placeholder expression (Integer.valueOf(_:Int)) abbreviates the abstraction ((x:Int)=>Integer.valueOf(x)).

3.1 Branching Code

The combination of instructions using the ~ operator only creates linear control flow. To implement loops and conditionals, branching instructions are needed along with instructions to create branching targets, as there are no explicit addresses in Mnemonics.

The following code snippets demonstrate the principle. The intended functionality of the code is to count down from 5 to 1 and print these numbers using the function

```
def printme(i:Int){ System.out.println(i) }
```

The function fBody specifies the code generator.

```
def fBody[ST<:List] = ((f : F[ST]) =>
      f ~ bipush (5) ~ withTargetHere( target =>
        _ ~ dup ~ ifne (_ ~ // short for f => f ~
                         dup ~
                         method1(printme(_:Int)) ~
                         bipush(1) ~
                         isub ~
                         target.jmp)) ~
      method1(Integer.valueOf(_:Int)))
```

A jump target is introduced by using withTargetHere. It calls the supplied closure with a jump handle that can later be called to jump back to the point designated by calling withTargetHere. The target parameter itself is typed with the frame state type at the position it was created. This ensures that it is only used from places in the method with a compatible stack as it is required by the JVM. The ifne instruction expects a integer value on top of the stack. It creates an implicit jump target for the next instruction, generates a branch instruction for it with the condition reversed, and then generates code according to its argument generator. The type of ifne (see Sec. 6.1) guarantees that the argument bytecode ends with an explicit control transfer (branch or return) and that no further instructions can follow.

As the direct handling of jump targets (in particular for forward jumps [26]) is error prone and unwieldy, Mnemonics provides shrink-wrapped operators for conditionals and loops. For example, the operator ifne2 expects a integer value on top of the stack. It takes two code sequences that map a source frame state to the same target frame state and creates a conditional out of them. As another example, the operator tailRecursive takes a command sequence and provides a self reference to enable recursive calls. Here is an example use:

[6] Method calls generated in this way are type-safe but they must be known at compile time of the generator. Sec. 5.3 explains how to invoke methods which are unavailable at compile time of the generator. Type safety of the latter cannot be guaranteed statically.

[7] See Figure 3 for the types of the instructions.

```
> ASMCompiler.compile(classOf[Integer],classOf[Integer])
 (param => ret => _ ~
   param.load ~
   method1((_:Integer).intValue) ~
   tailRecursive[Nil**Int,Nil**Int](self => _ ~
       dup ~
       method1(printme(_:Int)) ~
       bipush(1) ~ isub ~ dup ~
       ifne2(self, nop)
   ) ~
   method1(Integer.valueOf(_:Int)) ~
   ret.jmp)
```

The type parameters of tailRecursive specify the stack type at the entry and the exit frames of the loop. In this case, both entry and exit frames contain a stack with just one integer on top.

3.2 Local Variables

The JVM instructions $x$store and $x$load move values between local variables and the stack top (with x specifying the type of the value). These instructions access a local variable by index.

These instructions are not directly available in Mnemonics. Instead, there are operations to introduce typed and scoped capabilities [17], which are transparently associated to local variables in the current stack frame. The same capabilities are also used for accessing parameters. A capability for type T is a value of the abstract type Local[T]:

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

The introduction of a scoped local variable stores the current value on top of the stack in the Local and invokes a body closure with it. Thus, the lifetime of the local variable is confined to the body. Here is the signature of local variable introduction:

```
def withLocal[ST1<:List,ST2<:List,T]
    (body: Local[T] => F[ST] => F[ST2])
    :F[ST**T] => F[ST2]
```

Here is an example use of such a scoped variable:

```
bipush(5) ~
withLocal(i => _ ~
  i.store ~
  ... ~
  i.load ~ dup ~ iadd ~ i.store // i = 2*i
  // etc
)
```

3.3 High-level Composite Instructions

The full power of Scala can be used to define high-level building blocks for bytecode generation. As an example, consider a fold operation which iterates over a collection while accumulating some result. Fig. 2 defines a code generator for a fold operation on arrays. The generator is parameterized over a combining function func, which is inlined in the generated code.

The polymorphic function foldArray has three type parameters. The type ST represents the unchanged stack content. The type T is the base type of the array and U is the type of the final result. The code generated by foldArray expects an array of type T in a local variable accessed via capability array and the initialization value for the result of type U on top of the stack. After iterating through all array elements, it returns the result value of type U on the stack and leaves the array unchanged.

The parameter func is a frame transformer representing the combining function. This transformer expects the current result and the current array element on top of the stack and may access the current index into the array with the parameter of type Local[Int] which is passed to the closure. It consumes the current element and updates the result.

```
def foldArray[ST<:List,T,U]
  (array: Local[Array[T]])
  (func: Local[Int] => F[ST**U**T] => F[ST**U])
  : F[ST**U] => F[ST**U] =
    _ ~ bipush(0) ~
        withLocal{ index =>
          tailRecursive[R**U,R**U]{self =>
            _ ~ index.load ~
                array.load ~
                arraylength ~
                isub ~
                ifeq2(nop,
                  _ ~ array.load ~
                      index.load ~
                      aload ~
                      func(index) ~
                      index.load ~
                      bipush(1) ~
                      iadd ~
                      index.store ~
                      self)}}
```

Figure 2. Folding an array with an inlined combining function.

The function `foldArray` sets up a counter on the stack and iterates over the array indices using `tailRecursive`. The code in the loop first checks if the index is still within the array bounds and exits if it is not. Otherwise, the code sets up the stack according to the requirements of the `func`-generated code and inlines that code. Afterwards, the code increments the index and jumps to the beginning of the loop via `self`.

4. Case Study: ObjectFormatter

Mnemonics only generates subclasses of the Scala function type `scala.Function1` and puts all generated code in its `apply` method. Despite this limitation, Mnemonics is applicable to the implementation of DSLs for transforming values. For example, an XPath expression describes a function from a DOM instance to particular elements; a regular expression induces a function from an input string to a list of matches; and a parser is a function from an input string to an AST.

To assess Mnemonics, we developed a format specification language for object formatting. After creating an interpreter for the new language, we applied Mnemonics to create a compiler from format specifications to bytecodes and measured the resulting performance improvement. For space reasons, the paper contains an abridged version of the full account in Rudolph's thesis [26].

Given a format string and the target object type `T`, ObjectFormatter generates a function from `T` to `String`. As an example, suppose a string is to be created out of an object of type `Person`[8]:

```
> case class Person(name:String,town:String)
defined class Person
> val f = ObjectFormatter.formatter
          (classOf[Person], "#name lives in #town")
f: (Person) => String = <function>
> val str = f(Person("Tom","New York"))
str: String = Tom lives in New York
```

The factory method `ObjectFormatter.formatter` creates the formatting function from the format specification. A format specification, `"#name lives in #town"`, may contain references to properties (`#name`, `#town`) of the object, which are inserted when the formatter is applied to an object. In the end, bytecode is generated equivalent to this Scala code:

```
val format:Person => String =
  p => p.name+" lives in "+p.town
```

This example does not tap the full potential of ObjectFormatter. There are additional format specifications to insert properties, specify the format of date values, perform conditional formatting depending on the `Boolean` or `scala.Option` value of a property, and to flexibly display arrays and lists.

[8] A `case class` is a class that allows pattern matching on its instances.

ObjectFormatter works in two stages. In the first stage, it parses the format string into a typed AST. This stage checks the syntax of the format string, it resolves the names of all properties and methods, and performs a type check before generating the AST. This stage may signal errors. The second stage compiles the typed AST to bytecode. This stage never fails and always generates correct bytecode.

Here is an overview over the classes of the AST:

```
trait Exp[-T,+U]
case class MethodHandleExp[-T,+U]
  (m:Method1[T,U]) extends Exp[T,U]
case class ParentExp[T<:AnyRef,U<:AnyRef,V]
  (inner:Exp[U,V],parent:Exp[T,U]) extends Exp[T,V]

trait FormatElement[-T] extends (T => String)
case class Literal(str:String) extends FormatElement[Any]
case class ToStringConversion[T<:AnyRef]
  (exp:Exp[T,AnyRef]) extends FormatElement[T]

case class FormatElementList[T]
  (elements:Seq[FormatElement[T]]) extends (T => String)
```

A format string is parsed into a `FormatElementList [T]`, which is also a function of type `T => String`. A `FormatElement` is either a literal string or an expression, the value of which can be translated to a string. An expression is either a simple method call (`MethodHandleExp`) or a chained method invocation (`ParentExp`), resembling the common dot notation in object-oriented languages.

The result of compiling a `FormatElementList` is a function that uses an instance of `java.lang.StringBuilder` to successively append the elements of the format to a string, which is then returned. Compilation of one `FormatElement` takes place in the method `FormatCompiler.compileElement`:

```
def compileElement[R<:List,T<:AnyRef]
    (ele:FormatElement[T],value:Local[T])
    (f:F[R**StringBuilder]) : F[R**StringBuilder]
```

The function `compileElement` receives a `FormatElement` to compile and a handle to a local variable with the argument. Its generated code appends the string representation of the element to the `StringBuilder` instance on top of the stack. The body of `compileElement` pattern matches all concrete subclasses of `FormatElement`. The case for `ToStringConversion` serves as a simple example:

```
case ToStringConversion(e) =>
  f ~ value.load ~
    compileExp(e) ~
    method1((_:AnyRef).toString) ~
    method2((_:StringBuilder).append(_:String))
```

After loading the argument on top of the stack, `compileExp` compiles the expression and leaves the result on the stack. The result is converted to a string and appended to the `StringBuilder`, which is left on the stack top by `append`. Here is the code of `compileExp`:

```
def compileExp[R<:List,T<:AnyRef,Ret <% NoUnit]
            (exp:Exp[T,Ret]):F[R**T] => F[R**Ret] =
    exp match {
      case MethodHandleExp(handle) =>
        _ ~ handle
      case ParentExp(inner,parent) =>
        _ ~ compileExp(parent) ~ compileExp(inner)
    }
```

The function `compileExp` evaluates an expression that transforms a stack of type `F[R**T]` to type `F[R**Ret]`. If the expression is a simple method call (`MethodHandleExp`) with a handle, the invocation to the method is generated with `handle`. For a chained invocation (`ParentExp`), the `parent` expression evaluation code is generated first and then the `inner` one. This evaluation order is enforced by the type of `ParentExp`.

```
/* operations on integers */
def bipush[R<:List](value:Byte)              :F[R]                       => F[R**Int]
def ineg[R<:List]                             :F[R**Int]                  => F[R**Int]
def iadd[R<:List]                             :F[R**Int**Int]             => F[R**Int]
/* type cast */
def checkcast[R<:List,T<:AnyRef,U](cl:Class[U]) :F[R**T]                  => F[R**U]
/* stack manipulation */
def pop[R<:List,T]                            :F[R**T]                    => F[R]
def dup[R<:List,T]                            :F[R**T]                    => F[R**T**T]
def dup_x1[R<:List,T2,T1]                     :F[R**T2**T1]               => F[R**T1**T2**T1]
/* array access */
def baload[R<:List]                           :F[R**Array[Byte]**Int]     => F[R**Byte]
def baload[R<:List]                           :F[R**Array[Boolean]**Int]  => F[R**Boolean]
def aaload[R<:List,T<:AnyRef]                 :F[R**Array[T]**Int]        => F[R**T]
```

Figure 3. Instruction signatures.

5. Implementation

The implementation of Mnemonics consists of the definition of the instruction interface and two implementations of it, a bytecode interpreter (mainly for testing) and a bytecode compiler. The interface relies crucially on a number of features of the Scala type system and their interaction.

- Type inference. Manual type annotation of instruction sequences with intermediate frame types would be infeasible.
- Function types and Scala's concise notation for function application enable the seamless composition of functions that transform a stack type modeled as a strongly-typed heterogeneous list [12]. Function types and closures are also essential to model method calls and jump instructions.
- Variance annotations have proved crucial for implementing the structural constraint (see Sec.6.1).
- Implicit parameters facilitate the implementation of convenient method-call and branch instructions as well as help restricting instructions to category 1 types.
- The API `scala.reflect.Code` provides a way to let the Scala compiler insert an abstract syntax tree (AST) of a code block representing an expression of a particular type.

The rest of this section covers some details of the implementation of Mnemonics. It explains the strategy behind the implementation, points out limitations, discusses alternatives, and suggests future extensions.

5.1 Run-Time Stack and Local Variables

Mnemonics represents the run-time stack by a datatype for heterogeneous lists. This representation is similar to Jones's and Yelland's encoding of the JVM stack using nested pairs [11, 31]. The implementation in Mnemonics is inspired by the HList library [12] and its adaptation to Scala [19]. It defines types `List`, `Nil`, and `Cons` by[9]

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

such that an object of class `Cons[R,T]` represents a non-empty stack as a pair of a rest list of type `R` and the top element of type `T`. Any value of type `Nil` serves as an empty stack. Thus, a stack with the elements 3, "Str", 3.5f (bottom to top) is represented by an object of type

```
Cons[Cons[Cons[Nil,Int],String],Float]
```

A left-associative infix type synonym for `Cons` hones readability:

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

With this definition, the type of the example stack is

```
Nil**Int**String**Float
```

An earlier version of the library represented the local variables in a frame by a heterogeneous list, too. In that version, local variables were accessed by position and sophisticated type operations with numerals on the type level as well as a scheme involving implicit definitions[10] were needed to access and update a local variable (see the thesis [26]).

The current design with capabilities has several advantages over the list-based approach. First, the frame type gets simpler. In the previous design the frame type contained the list of the types of the local variables, too. Thus, switching to the capability-based approach simplified many type signatures in Mnemonics. Second, the type operations and the extensive use of implicit parameters in the previous design caused excessively long compile times already for simple programs. Finally, modularity of the generators improves considerably when programs no longer access local variables through explicit addresses.

5.2 Arithmetic Operations and Stack Manipulation

The instructions for performing arithmetic operations and for stack manipulation are straightforward. Each of them can be easily described by a suitable polymorphic type. The main use of polymorphism is to abstract over the elements of the run-time stack that remain unaffected by the operation.

Each arithmetic instruction applies to one primitive number type of the JVM. For instance, `iadd`, `isub`, and `imul` apply to primitive integers, `fadd` to floats, and `dadd` and `ladd` to double and long. The Java types `boolean`, `byte`, `char`, and `short` are represented as `int`s and hence usually processed by integer instructions.

As examples for integer instructions, consider the push byte constant, negation, and addition operations in the top section of Fig. 3: `Bipush` takes a `Byte` and pushes it as an `Int`. `Ineg` consumes the top stack element, negates it, and pushes the result back on the stack. `Iadd` requires the top two elements of the stack to be integers, pops and adds them, and pushes the result back on the stack. All three functions are polymorphic in the type `R`, the rest of the stack, which remains unaffected by these instructions.

The instruction `checkcast` tries to cast the top element of the stack into the supplied type, otherwise the JVM throws a `java.lang.ClassCastException`. The top element must be a reference type for `checkcast` to work correctly.

The instructions for manipulating the stack (Fig. 3) are no more complex than the arithmetic instructions. The instruction `pop` removes the top element from the stack. Its type is not needed anymore as indicated by the placeholder `_`. The `dup` instruction dupli-

[9] In the definition of `Cons`, the annotations `+R` and `+T` of the type parameters indicate their variance. Thus, `Cons[R,T]` behaves covariantly with respect to `R` and `T`.

[10] A language feature where the Scala compiler guesses a missing (implicit) function parameter by its type.

cates the top element of the stack. The instruction dup_x1 is similar to dup but stores the duplicated element two elements below the stack top.

For arrays, there are specialized access instructions tailored to each size and kind of datatype (baload and bastore, saload and sastore, iaload and iastore, and so on) that require byte (or boolean), short, or int arrays as parameters. Fig. 3 demonstrates with the baload instruction how overloading handles the typing requirements of these instructions. The other array instructions are straightforward, except the aaload and aastore instructions that load and store elements of arrays of arbitrary reference type, which is also shown in the figure (for aaload, only).

5.3 Method Call and Field Access

Section 3 already gave a glimpse of Mnemonics's treatment of method calls using the function method1:

```
method1((_:Integer).intValue)
method1(Integer.valueOf(_:Int))
```

Although the programmer writes a function as the parameter of method1, the Scala compiler does not translate it in that way. Instead, the compiler typechecks the code parameter of method1, reifies it as an *expression*, and passes the resulting abstract syntax tree for a function of type T => U, as indicated by the type of method1:

```
def method1[T,U](code:scala.reflect.Code[T => U])
  : Method1[T,U]
```

The (undocumented) use of the type scala.reflect.Code triggers this behavior of the Scala compiler. Mnemonics checks at run time of the generator that the code parameter is an abstraction of a method invocation. It only uses this mechanism as a convenience to the programmer to have the compiler transfer metadata for a correctly typed method invocation to the run-time part of Mnemonics. The facility is *not* intended or used as a general way to compile arbitrary functions to JVM code. At run time, method1 traverses the abstract syntax, checks that it consists of just a method invocation, and creates a method handle object. This approach covers invocations of static and instance methods, so that method1 covers unary static method calls and nullary instance calls.

There are a few drawbacks with this approach to method invocation. First, the parameter of method1 may have a suitable function type, but it may not represent a valid method call. The generator rejects such an ill-formed parameter at run time before generating bytecode. Second, method1 can only generate calls to known methods which are in scope of the generator.

To address the second drawback, Mnemonics supports a second way to create method handles using Java reflection, which is useful for applications, where the name of the method to be called is computed at run time of the generator (e.g., for a unary method):

```
def methodHandle[T,U](m:java.lang.reflect.Method
                     ,p1:Class[T],r:Class[U])
                     :Method1[T,U]
```

The Method parameter can be computed at run time but it carries no useful static information about the method's type. The methodHandle operation checks at generation time that method m has indeed the desired argument and return types and signals an error, otherwise.

Once a method handle is successfully created, it can be used to generate method invocations in a statically type-safe way anywhere in the program. The declaration of the Method1 type is given here:

```
trait IsUnit
trait NoUnit
implicit val unitIsUnit  : Unit      => IsUnit = null
implicit def anyrefNoUnit: AnyRef    => NoUnit = null
implicit val boolNoUnit  : Boolean   => NoUnit = null
implicit val byteNoUnit  : Byte      => NoUnit = null
/* and so on for the six remaining value types */

trait Method1[-T,+U] extends MethodHandle {
  def invoke[R<:List,T1X<:T,UX>:U <% NoUnit]()
    : F[R**T1X] => F[R**UX] = f => f.invokemethod(this)
  def invokeUnit[R<:List,T1X<:T,UX>:U <% IsUnit]()
    : F[R**T1X] => F[R] = f => f.invokemethod(this)
                            ~ Instructions.pop_unit
}
```

The method handle's invoke method is a frame transformer which generates the method invocation bytecode. Methods returning void require a special treatment. For the generator, their return type is Unit and the invocation method just discussed leaves this Unit type on top of the run-time stack type. However, the JVM does not represent void at all, so there is no corresponding entry on the real stack. The invokeUnit method is provided to correct this mismatch. After generating the method invocation, it cleans up the stack type to synchronize it with the real stack. The typing of invoke and invokeUnit ensures that either method is only invoked in the correct situation. To this end, the implicit conversions check whether the return type of the method is Unit using the types IsUnit and NoUnit.

Either of the invoke methods generates the correct bytecode sequence according to the metadata of the method in the method handle. The resulting bytecode instruction may be either invokeinterface, invokestatic, invokespecial, or invokevirtual.

As it is cumbersome to select between invoke and invokeUnit by hand, we instruct the compiler to insert it automatically by using an implicit conversion from a method handle to a frame transformer.

```
implicit def nCall1[R<:List,T,U<%NoUnit](m:Method1[T,U])
  : F[R**T]=>F[R**U] = m.invoke()
implicit def uCall1[R<:List,T](m:Method1[T,Unit])
  : F[R**T]=>F[R] = m.invokeUnit()
```

When type checking a function application f ~ method(...) of a method handle to a frame, the compiler has to convert a Method1 object to a function. If the return type of the function is Unit, then it applies uCall1 to the method handle, which results in an invokeUnit. Otherwise, the conversion happens with nCall1 and invokes plain invoke.

As a slight inconvenience to the programmer, Scala's type system requires separate definitions for each arity of method to be called.[11] Mnemonics provides the definitions method0, method1, and method2 for arities zero, one, and two, and it is simple to add further definitions covering more arities.

Mnemonics supports field access and update in a similar way as method calls. As with method calls, the restricted form of the AST is only checked at run time of the generator. Static fields may be accessed and updated by

```
putstatic(StaticVariableContainer.x = _)
getstatic(() => StaticVariableContainer.x)
```

where StaticVariableContainer is the class name for field x. Instance fields may be accessed and updated with a more pleasing syntax with the typings shown in Fig. 4:

```
putfield (_.x = _)
getfield (_.x)
```

To access fields whose names are only computed at generation time, field handles can be created analogously to method handles. At creation, a field handle is checked against the supplied run-time types and can afterwards be used statically type-safe.

[11] Unfortunately, a shortcoming in the Scala compiler prevents overloading in the presence of scala.reflect.Code, so that we cannot define a single overloaded method function.

```
def getstatic[R<:List,T](code:scala.reflect.Code[() => T])           : F[R]       => F[R**T]
def putstatic[R<:List,T](code:scala.reflect.Code[T => Unit])          : F[R**T]    => F[R]

def getfield[R<:List,T<:AnyRef,U](scala.reflect.Code[T => U])         : F[R**T]    => F[R**U]
def putfield[R<:List,T<:AnyRef,U](scala.reflect.Code[(T,U) => Unit]) : F[R**T**U] => F[R]
```

Figure 4. Typings for static and dynamic field access.

5.4 Instance Creation

Constraints 7 to 9 concern instance creation (Fig. 1):

> "[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."

These constraints pose a problem because instance creation consists of two steps: First, the `new` instruction allocates an uninitialized object. The second step calls the constructor. Parameters to the constructor are given in standard order, that is, the receiver is first on the stack followed by the parameters. Hence, there may be code for evaluating parameters between the `new` and the constructor call. The verifier has to ensure that this code does not access the uninitialized object, which has been a problem in the past [8].

To avoid such problems, Mnemonics does not provide direct access to the JVM's `new` instruction. Instead, there are only functions for generating nullary and unary constructor calls:

```
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 function `newInstance` invokes the single nullary constructor of the class. The function `newInstance1` expects the value of the constructor's single argument on the stack. It relies on `scala.reflect.Code` like `method1`. Internally, it generates code that *first* evaluates the argument, then it creates the new instance, swaps the stack entries, and employs reflection to generate the invocation of the correct (potentially overloaded) constructor. Thus it fulfills the constraints by construction of the code.

A similar construction, which evaluates the parameters before creating the instance, is possible for constructors with two and three arguments of category 1 type by emitting a suitable sequence of stack manipulation instructions after creating the instance. Higher arities seem to require more general stack manipulation instructions than are available in the JVM.

The resulting restriction of constructors to arities less than four does not appear severe because classes can always provide static factory methods as wrappers for constructors with more parameters. Alternatively, there could be a designated type constructor for uninitialized objects and operations that are illegal in their presence could rely on a predicate on the stack type to detect them. Even without such a constructor, instance creation could rely on polymorphism to guarantee that the uninitialized object is not accessed by the evaluation of the arguments. We leave the exploration of these possibilities to future work.

5.5 The `return` Instruction

Mnemonics does not directly support the `return` instructions of the JVM. Instead, it relies on a typed return capability that can be used like other control transfers anywhere in the code.

```
trait Return[U<:AnyRef]{
  def jmp[LT<:List]:F[Nil**U,LT] => Nothing
}
```

`Nothing` is a special Scala type which has no instances. Mnemonics uses it as the result type of unconditional control transfers to indicate that instructions after the control transfer are never reached.

Its use with the conditional branches is straightforward:

```
compiler.compile[Integer]
  (param => (ret:Return[String]) => _ ~
     param.load ~
     method1((_:Integer).intValue) ~ bipush(5) ~ isub ~
     ifne(_ ~ ldc("does not equal 5") ~
             ret.jmp) ~
     ldc("equals 5") ~
     ret.jmp)
```

5.6 Category 2 Types

The JVM classifies types into two different kinds, category 1 and 2, according to the number of 32 bit words required to represent them on the stack or in a local variable. A value of category 1 type has a one-word representation whereas a category 2 type requires two words. The category 2 types are `Double` and `Long`.

Mnemonics' representation of values on the stack is oblivious of the size and the layout of the types. Hence, there are no particular problems with arithmetic operations, which expect operands of category 2 types. The implementation of each instruction that is available for different types inspects the type of its operand as part of the stack type and generates the correct instruction accordingly.

Thus, for most instructions the handling of category 2 types is transparent. A few instructions are not applicable to category 2 types and the library should protect from such misuse. An example for such an instruction is `swap`, which exchanges the topmost two entries on the stack *provided they have both category 1 types*. The solution is to equip the `swap` instruction with implicit parameters that force the values on the stack top to be category 1 types.

```
trait Category1
def swap[R<:List,LT<:List,T1<%Category1,T2<%Category1]()
    :F[R**T2**T1,LT]    => F[R**T1**T2,LT]
```

The trick to distinguish category 1 types from others is to provide implicit definitions of values of type `T=>Category1` for all category 1 types `T`, but not for the category 2 types `Double` and `Long`.

```
implicit def cat1any:AnyRef=>Category1 = null
implicit def cat1int:Int=>Category1 = null
implicit def cat1boolean:Boolean=>Category1 = null
/* and so on, seven definitions in total */
```

Given these definitions, Scala's type checker correctly rejects any attempt to apply `swap` to a stack with a category 2 type on top.

Local variables and parameters present no problems. They are represented by typed capabilities that internally check their type parameter to generate the appropriate JVM instructions. In particular, the internal storage manager makes sure to reserve adjacent addresses for local variables of category 2 type.

5.7 What's Missing

The current implementation of the library does not support switch instructions, the subroutine instructions `jsr` and `ret`, exceptions, some special array instructions, and synchronization.

The instructions dealing with exceptions, one dimensional array creation and manipulation, as well as synchronization are straightforward to add as they present no new conceptual problems. Subroutines are abolished since version 1.6 of the bytecode, so there is no incentive in implementing them. Multi-dimensional array creation and the switch instructions do not seem to be feasible to support in a type-safe way.

6. Evaluation

This section evaluates several aspects of the Mnemonics bytecode generation library. First, it assesses to what extent the typed representation of instructions indeed ensures the structural constraints. Second, it evaluates its usability as a domain specific language. Third, it presents a microbenchmark comparing the performance of an interpretive and the compiling version of the ObjectFormatter from Sec. 4.

6.1 Structural Constraints

The main design goal of Mnemonics is that only valid instruction sequences should be expressible. This subsection revisits the structural constraints (numbers refer to items in Figure 1) on instruction sequences and evaluates to what extent the library prevents invalid uses of bytecode instructions.

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

For the first part, Mnemonics's typed representation of the frame and the instructions guarantees that the compile-time types are in lockstep with the corresponding frame state at execution time. When extending an instruction sequence with a new instruction, the ~ operator requires the argument type of the new instruction to match the current frame type and it returns the next frame with an appropriately updated type.

Thus, given that the type of each single instruction correctly reflects its action on the frame, an induction on the number of ~ operators extends this guarantee to sequences of instructions, which implies that the Scala compiler enforces the first part of constraint 1.

The second part of the constraint could be paraphrased as: "The frame state type at the target of each control transfer must be a supertype of the frame state types at the corresponding sources." Mnemonics's control transfer instructions implement exactly this constraint: `jmp` requires a target where the frame state type is a supertype of the frame state type at the source and similarly for the other branching operators like `ifne2`. This flexibility is facilitated by making the stack type covariant as shown in Sec. 5.1.

Constraint 2 allows the use of `int` instructions on integer types shorter than `int`. Mnemonics currently requires an explicit cast to do so. As a convenience feature, the branch instructions avoid this cast with an implicit conversion of their condition argument to a machine integer.

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

which expands to

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

For each type possibly used as integer an `implicit` conversion has to be defined which converts the type into a `JVMInt`.[12]

[12] The programmer need not supply the `implicit` parameter. After type inference, Scala fills in implicit parameters by searching for suitably typed implicit definitions in scope.

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

Constraint 4 states that a local variable must not be accessed before it has been defined. This property is trivially enforced by the API for creating capabilities, which requires an initial value.

Constraint 5: see constraint 1.

Constraints 6 to 9 are enforced by the `ASMCompiler` back end. `invokespecial` cannot be called directly, the initialization constructs have to be used instead. Mnemonics does not model stacks with uninitialized variables (see Section 5.4).

Constraint 10 and 11 refer to the compatibility of the receiver's type and parameters' types with a given method signature. This problem is addressed by specifying the called method as a Scala function value. Because a Scala function is contravariant in its 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 because the return instruction is only available through the typed return capability passed into the generator. This capability matches the return type of the generated method by construction.

Constraint 13 is enforced through the type of the array access instructions (not shown).

The typing of `ASMCompiler.compile` guarantees that every generated bytecode sequence ends with a return-typed frame. Thus, every such sequence ends with a control transfer instruction, which fulfills constraint 14.

The JVM specification mentions further structural constraints, which the current design does not consider. For example, subroutines are not supported.

Mnemonics does not support member protection levels. Because members can only be accessed through method calls and methods are represented by closures, the protection level as indicated by the closed-over environment applies. A mismatch in protection level can lead to broken and rejected bytecode. As an example, consider the following code

```
object Generator{
[private] def test(str:String):Int = //...

def generate(f:F[ST,LT]) =
  f ~ ldc("test") ~ method1(test(_:String))
}
```

The bytecode created by `generate` contains a method invocation of the `test` method. However, an attempt to run this bytecode outside of the `Generator` object leads to an error because `test` is not in scope. Analogously, in the Scala interpreter console, it is not possible to invoke a method defined in the interaction from code generated in the interaction because this method is not in scope.

6.2 Usability

The case study in Sec. 4 (cf. also the full case study [26]) demonstrates the usability of Mnemonics in a client application. The study considers the compilation of a format string into a conversion method, where the format string specifies the textual rendition of an object. The compiler designed in this study is structured in a number of phases, each of which is implemented as a composite instruction. Type annotations were only needed to bootstrap the frame state type when calling `ASMCompiler.compile` and in the definition of a composite instruction. In some cases, it is necessary to explicitly state the type parameter instantiation for an instruction.

The type error messages from the compiler are a definitive practical limitation. They are hard to decipher because the compiler always prints fully qualified types and it does not use infix notation for type operators. Another complicating aspect is the pervasive use of implicit parameters. Also the limitations of Scala's type inference are not easily foreseeable so that experimentation is sometimes required.

6.3 Performance

We developed a small benchmark suite to quantify the performance gain of a compiled format-string formatter over an interpreting one. These benchmarks deliver the expected results. For simple format strings the compiled code has a speedup of 22%–45% with respect to the interpreted one. For more complex cases of array expansion and conditional format specifications, the speedup is considerable: the generated bytecode runs 3.4–3.9 times faster than the interpreted version. More details may be found in the thesis [26].

7. Related Work

7.1 Bytecode Generation Libraries

With the broad adoption of the Java platform, several tools emerged to operate on and generate binary class-files. These tools differ in the way classes and instructions are represented and how operations on bytecodes can be defined, depending on the anticipated use of the library. All frameworks perform (de-)serialization and ensure that the static constraints hold for generated class files. Out of the many available frameworks[13], this section considers only a small, representative selection.

Objectweb ASM is a lightweight bytecode generation and transformation framework that can create and modify classes [3]. It can work in static mode as well as in dynamic mode (i.e., modify a running program). In contrast to many other bytecode frameworks, ASM has, aside from an object-tree based encoding of class and method structure, a lightweight API employing the Visitor pattern. ASM comes with a number of bytecode analyses, but there is no built-in full bytecode verifier. Mnemonics builds on ASM as the low-level library to create class files.

The Bytecode Engineering Library (BCEL) is a project of the Apache Jakarta package "to analyze, create, and manipulate (binary) Java class files (. . .)."[1]. It can analyze, change, and generate classes, adhering to the static constraints of class files. It can perform these tasks statically as well as dynamically.

Bytecode instructions are arranged in a sophisticated class hierarchy. 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. There are modules to disassemble a module into HTML or into Java source code, which itself uses BCEL to generate the code. A bytecode verifier is also available.

Javassist is the bytecode generation framework by Chiba [4]. It provides high-level abstractions for transformations on class files.

Class methods and call sites may be instrumented with code snippets at load time of the 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.

Javassist also has a bytecode level API[14] allowing direct access to and generation of bytecode instructions. These transformations work either offline on bytecode or at load time through a custom class loader. The transformations based on the template language guarantee verifiable bytecode, the bytecode-level API does not.

In summary, the existing libraries are able to generate all bytecodes, but they do not guarantee verifiable bytecode statically. Mnemonics does so, but at the price of being restricted to a (large) subset of bytecodes. We have found the supported subset adequate for a range of bytecode generation tasks.

Some libraries perform some amount of verification at/before bytecode generation time on demand of the user. Mnemonics always generated verifiable bytecode. To support additional instructions only some simple generation-time checks are required.

7.2 Foundations

Jones [11] specifies the JVM instructions as functions operating on abstract stack frames, quite similar to the approach in this paper. Jones also represents the stack with nested pairs, but his representation of local variables uses record types. The instruction types are polymorphic to abstract over those parts of the stack and the local variables that remain untouched by the instruction. A similar basis for specifying the JVM has been put forward by Yelland [31]. While Jones does not address the issue of subtyping, Yelland does so by encoding subtyping with polymorphism [7]. Our work also models the JVM instructions in the types of stack transformers. Unlike in Jones and Yelland's work, our host language Scala supports exactly the right notion of subtyping, which we reuse for the object language. Also, we are not interested in modeling the entire instruction set of the JVM, but rather in providing a convenient DSL for typed generation of verified bytecode sequences. Thus, Mnemonics models some instructions precisely, but other instructions are only indirectly available.

The idea of typing subroutines in low-level languages with types that are polymorphic over a stack is also present in work on stack-based typed assembly language (e.g., [18]).

Further work has studied type systems for JVM bytecode with the goal of finding a formal basis for bytecode verification. Examples are the work by Stata and Abadi [27] on verifying subroutines and by Freund and Mitchell [8] which focuses on object initialization. Again, these works aim at *specifying* the semantics of bytecode and deal with features not covered by Mnemonics.

7.3 Metaprogramming

Metaprogramming, like code generation at run time, has traditionally taken place in an untyped setting, that is, in languages like Lisp [2] and Smalltalk. Even there, the semantic issues are non-trivial [28]. For typed languages, Taha and Nielsen have shown how to integrate type-safe run-time code generation into a functional programming language [30]. This approach is implemented in the MetaOCaml system [29]. However, MetaOCaml relies on a specially tailored multi-level type system along with a similarly specialized run-time system to ensure type-safe execution of code generated at run time.

Other systems perform run-time code generation based on partial evaluation techniques. Examples are Fabius [13], Tempo [5], 'C [23], and DyC [10]. DynJava [22] is similar in spirit as these works. It is particularly related to our work as it performs typed, template-based run-time code generation for Java. However, it also specifies generated code by Java source templates rather than using bytecode directly. Furthermore, it relies on a specially tailored type system and compiler.

There are also relationships to typed, heterogeneous metaprogramming and embedded DSLs as touched upon by Czarnecki and coworkers [6]. It is special to our work that the source and target languages are different, but share a common type system. The work of Mainland and coworkers on Flask [16] falls in a similar category. Their metalanguage Haskell supports two object languages, NesC and RED. The former is a C-dialect, the type structure of which can be mapped into suitable Haskell types, and the latter is a subset of Haskell with a trivial mapping of the types.

[13] `http://java-source.net/open-source/bytecode-libraries` contains a comprehensive list.

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

8. Conclusion

Scala's type system facilitates the construction of a type-safe bytecode generation library, where a type correct generator guarantees the well-formedness of the generated bytecode to a large degree. The library relies heavily on advanced typing features of Scala, notably type inference, function types, implicit parameters, and reification of abstract syntax trees with `scala.reflect.Code`. While the first two features are essential, the last two contribute significantly to the ease of use of the library.

The design of Mnemonics emphasizes ease of use and safety. It cannot generate arbitrary safe bytecode sequences, but instead restricts the programmer to a subset of bytecodes and sometimes to certain usage patterns to ensure safety through Scala's typing. Thus, it is not a general cure for all bytecode generation needs, but rather a very convenient bytecode generation tool that guarantees safety through typing of the generator for a restricted subset of bytecodes.

References

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

[2] A. Bawden. Quasiquotation in Lisp. In O. Danvy, editor, *Proc. ACM Workshop Partial Evaluation and Semantics-Based Program Manipulation PEPM '99*, pages 4–12, San Antonio, Texas, USA, Jan. 1999. BRICS Notes NS-99-1.

[3] E. Bruneton. Asm 3.0 – a Java bytecode engineering library, 2007. `http://download.forge.objectweb.org/asm/asm-guide.pdf`.

[4] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In *GPCE '03: Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, pages 364–376, Erfurt, Germany, 2003. Springer-Verlag.

[5] C. Consel, J. Lawall, and R. Marlet. Tempo specializer — users' manual. `http://phoenix.labri.fr/software/tempo/doc/tempo-doc-user.html`, 1998.

[6] K. Czarnecki, J. T. O'Donnell, J. Striegnitz, and W. Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. In Lengauer et al. [14], pages 51–72.

[7] M. Fluet and R. Pucella. Phantom types and subtyping. *J. Funct. Program.*, 16(6):751–791, 2006.

[8] S. N. Freund and J. C. Mitchell. A type system for object initialization in the Java bytecode language. *ACM TOPLAS*, 21(6):1196–1250, 1999.

[9] S. N. Freund and J. C. Mitchell. A type system for the Java bytecode language and verifier. *J. Autom. Reasoning*, 30(3-4):271–321, 2003.

[10] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 2000.

[11] M. P. Jones. The functions of Java bytecode. In S. Eisenbach, editor, *Formal Underpinnings of Java*, 1998.

[12] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107, Snowbird, Utah, USA, 2004. ACM Press.

[13] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *Proc. 1996 PLDI*, pages 137–148, Philadelphia, PA, USA, May 1996. ACM Press.

[14] C. Lengauer, D. S. Batory, C. Consel, and M. Odersky, editors. *Domain-Specific Program Generation, International Seminar*, volume 3016 of *LNCS*, Dagstuhl, Germany, 2004. Springer.

[15] T. Lindholm and F. Yellin. *The Java(tm) Virtual Machine Specification*. Addison-Wesley, 2nd edition edition, 1999.

[16] G. Mainland, G. Morrisett, and M. Welsh. Flask: Staged functional programming for sensor networks. Victoria, BC, Canada, Sept. 2008. ACM Press.

[17] J. H. Morris Jr. Protection in programming languages. *Comm. ACM*, 16(1):15–21, 1973.

[18] J. G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. In *TIC '98: Proceedings of the Second International Workshop on Types in Compilation*, volume 1473 of *LNCS*, pages 28–52, Kyoto, Japan, Mar. 1998. Springer-Verlag.

[19] J. Nordenberg. MetaScala, 2008. `http://www.assembla.com/wiki/show/metascala`.

[20] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, L. Spoon, E. Stenman, and M. Zenger. An Overview of the Scala Programming Language (2. edition). Technical report, 2006.

[21] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Inc, 1st edition, Nov. 2008.

[22] Y. Oiwa, H. Masuhara, and A. Yonezawa. DynJava: Type safe dynamic code generation in java. In *3rd JSSST Workshop on Programming and Programming Languages (PPL2001)*, Mar. 2001.

[23] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM TOPLAS*, 21(2):324–369, Mar. 1999.

[24] Z. Qian. A formal specification of Java virtual machine instructions for objects, methods and subroutines. In *Formal Syntax and Semantics of Java*, pages 271–312. Springer-Verlag, 1998.

[25] E. Rose. Lightweight bytecode verification. *J. Autom. Reasoning*, 31(3-4):303–334, 2003.

[26] J. Rudolph. Mnemonics: Type-safe bytecode combination in Scala. Diploma thesis, Albert-Ludwigs-Universität Freiburg, Mar. 2009. `http://virtual-void.net/files/mnemonics.zip`.

[27] R. Stata and M. Abadi. A type system for Java bytecode subroutines. *ACM TOPLAS*, 21(1):90–137, 1999.

[28] W. Taha. A sound reduction semantics for untyped CBN multi-stage computation. In J. Lawall, editor, *Proc. ACM Workshop Partial Evaluation and Semantics-Based Program Manipulation PEPM '00*, Boston, MA, USA, Jan. 2000. ACM Press.

[29] W. Taha. A gentle introduction to multi-stage programming. In Lengauer et al. [14], pages 30–50.

[30] W. Taha and M. F. Nielsen. Environment classifiers. In G. Morrisett, editor, *Proc. 30th ACM Symp. POPL*, pages 26–37, New Orleans, LA, USA, Jan. 2003. ACM Press.

[31] P. M. Yelland. A compositional account of the Java virtual machine. In A. Aiken, editor, *Proc. 26th ACM Symp. POPL*, pages 57–69, San Antonio, Texas, USA, Jan. 1999. ACM.