Mnemonics: Type-safe Bytecode Generation at Run Time

Johannes Rudolph (johannes.rudolph@googlemail.com) and
Peter Thiemann (thiemann@acm.org)
University of Freiburg, Germany

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 are 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.

Keywords: run-time code generation, java bytecode, scala, domain-specific language

CR Subject Classification: 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

1. Introduction

The Java Virtual Machine (JVM) [19] 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., [2, 3, 5]). Some of these libraries also support code generation at run time.

Each generated bytecode sequence has to pass the bytecode verifier before being loaded and executed by the JVM [33]. Roughly speaking, after checking the format of the bytecode, the verifier attempts to statically prove the safety of a given sequence of bytecodes. It rejects a bytecode sequence if it fails to verify. Many existing bytecode generation libraries only guarantee that generated class files are correctly formatted, but the generated bytecode may still fail to pass the verifier. Some libraries offer a verification pass that checks the generated bytecode after generation. If turned on, this verification pass aborts the generation of bytecode instead of having the library emit bytecode, on which the bytecode verifier would fail. In both cases, however, the problem is discovered at a time when it is hard to connect it to its origin in the code of the generator: either after running the generator or when attempting to load the generated code. It would be much better, if bytecode generation would fail when generating the offending instruction or —better yet— when compiling the generator of the offending instruction.

This work introduces Mnemonics, a novel bytecode generation library written in Scala that spots most bytecode verification errors at compile time of the generator. 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. This encoding is facilitated by Scala's type system because it incorporates Java's (and thus the JVM's) type system. For most use cases, 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 abstracts the contents of the stack at run time. Mnemonics maintains this abstraction by modeling the generator of an instruction with a suitably typed function that transforms the compile-time stack and its type. To create a sequence of bytecode instructions requires combining instruction-generating 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.

The concepts underlying Mnemonics extend to all JVM instructions. Most kinds of instruction are directly supported by a corresponding Mnemonics function. The instructions for branches, switches, and object instantiation are not directly accessible but require the use of a type-safe pattern provided by the library. The implementation has been developed in a demand driven way. It implements all patterns explained in this paper and many of the directly supportable instructions. Adding instructions of the latter kind is straightforward.

A type correct Mnemonics generator always returns a sequence of bytecode instructions such that no bytecode verification is needed before running it. However, if the generator is to output an invocation of a method that is not accessible at compile time of the generator, then the generator must perform a generation-time subtype check, which may throw an exception. As this rare case requires using a special Mnemonics function, the programmer is aware of this possibility. In any case, the check is much cheaper than a full blown bytecode verification.

Each run of a Mnemonics generator creates a class that implements one of Scala's function traits (i.e., `Function1`, `Function2`, ...) and puts all generated code in its distinguished `apply` method. Although

Mnemonics cannot generate arbitrary classes with arbitrary methods, it is nevertheless applicable to the implementation of any DSL (domain specific language) 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 abstract syntax tree. Further potential uses are compiled evaluation of expressions or commands entered at run time, object serialization, and compiled XML validation.

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.
- We identified reusable design patterns for embedding typed DSLs in Scala.
- We defined a formal model for a subset of Mnemonics and proved its correctness.

Overview

Sections 2 and 3 cover background on the JVM and on the constraints imposed on bytecode programs as well as on the Scala language [25, 24]. Readers familiar with the JVM and/or Scala may safely skip one of these sections. Section 4 introduces the functionality of the Mnemonics library with examples. The consideration of examples culminates in Section 5 with a case study that generates bytecode for string formatters at run time. Section 6 highlights some important details of the implementation and characterizes the set of supported instructions. The subsequent Section 7 extracts design patterns used in the construction of Mnemonics that may be useful in a more general context. The correctness of Mnemonics is investigated in Section 8, which presents a formal model and gives a correctness proof for a small subset. Section 9 evaluates the Mnemonics library from different perspectives: correctness, usability, and performance. Section 10 discusses related work and Section 11 concludes.

The paper is based on and extends Rudolph's diploma thesis [34]. The thesis and the implementation are available on the web at `http://virtual-void.net/files/mnemonics.zip`.

This paper extends our previous paper [35] on the topic in several respects. The extended approach supports all instructions. The case study is further elaborated. We developed a new, more general scheme for implementing instance creation and an implementation of the switch instructions. We added a section discussing the design patterns used in Mnemonics. We extended the performance evaluation.

2. Background: JVM Basics

The JVM is a stack-based abstract machine which constitutes the basis of the Java platform [19]. It executes programs written in Java bytecode, which serves as a portable target language mainly for compiling object-oriented programming languages like Java or Scala. Compiled code has the form of a *class file*, which contains a bytecode array for each method in a class.

Execution of bytecode in the JVM is organized in threads. The state of each thread 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 category 1 types have a one-word representation (32 bits), the category 2 types 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.

A typed machine statically associates metadata (the type) with each bit vector (word or doubleword) that it manipulates. This metadata fixes the meaning of the bit vectors associated to it once and for all. In contrast, an untyped machine (like a CPU) dynamically reinterprets a bit vector as dictated by each instruction.

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 of an instruction are available on the stack, and that their types correspond to the expectations of the instruction. The JVM class-loading infrastructure contains the bytecode verifier, which guarantees adherence to the structural con-

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) [19].

straints. This verifier gives guarantees similar to those of a type system and can be described as such [30, 10, 9].

3. Background: The Scala Programming Language

Scala is based on the JVM and blends object oriented programming (classes, traits, singletons, inheritance, subtyping) with functional programming (functions are objects, anonymous functions, higher-order functions, type classes). This section presents an overview of those aspects of Scala, on which our work relies. It is not a complete introduction to the language.

3.1. Types, Classes, Objects, and Traits

Scala's type hierarchy is a superset of Java's type hierarchy. It integrates the reference type hierarchy (subtypes of `scala.AnyRef` corresponding to `java.lang.Object`) and value types (subtypes of `scala.AnyVal`) into one unified hierarchy with type `scala.Any` at the top. There is also a bottom type `Nothing` which is a subtype of all other types and which has no values. Type `Nothing` is used as a return type for methods which are never supposed to return normally, for example, a method that always throws an exception or that never halts.

In Scala the primitive types from Java are defined with capitalized names as `Int`, `Float`, `Byte`, etc. inside the `scala` package.

A class is defined with the `class` keyword. Constructor parameters follow directly in parentheses after the class name. The body of the class declaration serves at the same time as a container for member declarations as well as for the constructor code. This example creates a class with a member `name`, which is initialized in the constructor:

```
class Person(_name: String) {
  val name = _name
}

// Usage
new Person("Bob")
```

Scala has no notion of static members. Instead it has syntax for directly declaring singleton objects by using the `object` keyword instead of `class`.

Aside from classes and singleton objects, Scala has `trait`s as a basic means of code reuse. A trait is similar to a Java interface because it can contain abstract members and a class can inherit from several traits. However, unlike an interface a trait can also contain concrete members.

```
trait Addition {
```

```
trait Exp
case class Literal(value: Int) extends Exp
case class Plus(left: Exp, right: Exp) extends Exp

object Evaluator {
  def eval(e: Exp): Int = e match { // pattern match
    case Literal(v) => v
    case Plus(l, r) => eval(l) + eval(r)
  }

  val exp = Plus(Literal(5), Literal(12))
  println(eval(exp)) // == 17
}
```

Figure 2. Case class hierarchy and usage

```
  def add(a: Int, b: Int): Int = a + b
}
trait Multiplication {
  def mult(a: Int, b: Int): Int = a * b
}
class Arithmetics extends Addition with Multiplication

val a = new Arithmetics
a.add(12, 5)
a.mult(4, 6)
```

3.2. Case Classes

Scala supports algebraic datatypes and pattern matching through a convenience syntax for classes, introduced by the keyword `case class`. In a case class all constructor parameters are public class members. A case class can be instantiated without using the `new` keyword and pattern matching can be used to access members by position.

Figure 2 defines a data structure to represent arithmetic expressions. `Exp` is the common marker type for all expression types. There are two types of expressions: literal integer values and addition. The object `Evaluator` defines a method `eval`, which calculates the value of an expression using pattern matching.

3.3. Type Parameters

Classes, traits, and methods may have type parameters. Type parameters are enclosed in square brackets and can be annotated with upper and lower bounds. Here is an example of a simple container type:

```
class Box[T](value: T) {
  def get: T
}

def objectBox[T <: AnyRef](t: T): Box[T] = new Box(t)
```

```
val a = objectBox("Test") // ‘a‘ is of type Box[String]
// objectBox(5) would fail: Int is no subtype of AnyRef
```

Upper bounds are indicated with `<:`, lower bounds with `>:`. Type applications are also written with square brackets. In practice, they are rarely needed because Scala infers most type parameters.

In trait or class definitions, type parameters may be annotated with variance modifiers. By default, type parameters are invariant as in Java. A `+`-sign designates a covariant type parameter, a `-`-sign a contravariant one:

```
val a = new Box("Test")  // type: Box[String]

//val b: Box[AnyRef] = a // fails Box[String] is no Box[AnyRef]
                         // because of invariance

class CoBox[+T](t: T) {
  def get: T
}
val c = new CoBox("Test")  // type: CoBox[String]
val d: CoBox[AnyRef] = d   // works: CoBox[String] is a subtype of
                           // CoBox[AnyRef] because of covariance
```

3.4. Functions

An object of trait-type `scala.Function1` represents a first-class function of one parameter. This trait has one abstract method `apply` containing the body of the function in concrete subclasses. The notation `T => R` abbreviates the type `Function1[T, R]`. Aside from the (uncommon) possibility to simply extend and implement `Function1` there are several other ways to create function objects. Here are three possibilities to create anonymous function objects:

```
val f = new Function1[Int, String] { def apply(i: Int) = i.toString }
val g = (i: Int) => i.toString // same as f
val h = (_: Int).toString // same as f and g
f(5) // == "5"
```

In the first line, `Function1` is implemented like one would have done in Java. In the next line the short form is shown, where the parameter list is separated from the implementation by an arrow. For simple functions there is an even shorter version where no parameter list is given but instead the single occurrence of the parameter in the function body is replaced by the placeholder character `_`. In many situations the type annotations for the function parameters can be omitted if the type can be inferred from the context.

For other arities, there are traits named `Function0`, `Function2`, and so on analogous to `Function1` and with similar syntactic support.

3.5. Implicits

Implicits are a powerful feature of the Scala type system where function calls and values from the scope can be automatically inserted into the program at certain points depending on the static type of the context. Using implicits enables a range of features from type coercion (implicit conversion) to ad-hoc polymorphism (implicit parameters).

The basic principle of implicits is that the developer may omit information at certain points in a program and this information is implicitly provided by the compiler by inferring it from the scope. The compiler searches for an implicit value in the following situations:

– The type of a value is incompatible with the expected type.

– A member value or method that does not exist is selected from a value.

– When invoking a method, a parameter list is missing which is annotated with the modifier `implicit`.

Only specific implicit definitions are eligible for implicit resolution. Here is an example that makes a string convertible to a `Person` object:

```
implicit def stringMaybePerson(name: String): Person =
  new Person(name)
```

The scope for implicit resolution is the normal scope for resolving other names in Scala[1]. Therefore, the resolution of implicits can be controlled by imports, shadowing, or inheritance.

3.6. Arguments and Reflection

3.6.1. *Variable Length Argument Lists*

Scala supports variable length argument lists for methods and constructors. A variable length argument is declared by appending `*` to the type of the last argument of an argument list:

```
def sum(integers: Int*) = // ...

sum(2,3,5,2,6,2)
```

Inside the method body the `integers` argument has type `Seq[Int]`.

3.6.2. *The `Class` Type*

The type `Class` represents the type `java.lang.Class` from Java reflecting over a class in the JVM. It can be acquired by calling `getClass`

[1] Actually, the scope is slightly extended by including companion objects of types associated with the needed type for the implicit value.

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

Figure 3. Sample bytecode for a boxed increment function.

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

Figure 4. Generator for increment. The code snippet shows an interaction with the Scala toplevel. The `>` character is Scala's prompt and the last, non-indented line contains the response of the system.

on any reference at runtime or by using `classOf[X]` at compile time (corresponding to `X.class` in Java). To check whether a value `x` is of type `X` the syntax `x.isInstanceOf[X]` is used.

4. 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. The inconvenience of unboxing and reboxing is sometimes needed in the bytecode because the JVM differentiates between the primitive type `int` and the boxed reference type `java.lang.Integer` (or just `Integer`), an instance of which wraps a value of type `int`. In Java, this distinction is blurred because of autoboxing, which automates the introduction of the boxing and unboxing operations. Scala also distinguishes boxed and primitive integers, but it calls the primitive type `Int`. Figure 3 contains the bytecode that we want to generate.

The corresponding generator in Figure 4 looks quite similar to the bytecode. Before delving into the explanation of the generator, it is important to realize that the `ASM.compile`[2] function wraps the bytecode

[2] ASM is the global object that contains the implementation of Mnemonics. It contains the `compile` function.

generated by the Mnemonics function into an anonymous class implementing Scala's `Function1` trait, loads the resulting class, creates an instance from it, and returns a reference to this instance. The resulting value `f` has type `Function1[Integer,Integer]`, which means it can be used like a function in Scala:

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

The `ASM.compile` function has the type signature

```
  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. Both must be reference types as indicated by the subtype constraint `<:AnyRef`. Not supporting primitive types here is an unfortunate consequence of the goal to return a subclass of `Function1`. Alas, `Function1` is a generic class, and after erasure the most general type for parameter and return values of `apply` is `AnyRef`, which is no supertype of the primitive types. Therefore, it is not possible to support the general interface of `Function1` for primitive types. However, primitive types can be supported by manually boxing values as needed.[3] The arguments `par1Cl` and `retCl` are the corresponding class objects that contain descriptions of the types `T1` and `R`, respectively.

The `code` argument relies on a number of types provided by Mnemonics. A value of type `Local[T1]` is a handle (capability) for accessing the parameter (of type `T1`) of the generated function and a value of type `Return[R]` is a capability for returning a value of type `R`. Capabilities are further discussed in Section 4.2. After receiving these two parameters, the `code` function must return a *frame transformer* that creates the bytecode sequence. The `compile` function applies the transformer to an initial frame of type `F[Nil]`, where `F` is an abstract frame type provided by Mnemonics, and expects it to return `Nothing`.

A frame of type `F[ST]` encapsulates a JVM stack of type `ST`. The stack is represented by a nested pair structure with pairing indicated by the infix type constructor `**` and the empty stack by the type `Nil`, which are both subtypes of a type `Stack` (see Section 6.1). For instance, the frame type `F[Nil**T]` indicates a frame where the stack contains exactly one entry of type `T`.

In the example in Figure 4, the frame transformer is an anonymous function with formal parameter `frame`, the initial frame corresponding

[3] We considered auto-boxing but decided to be explicit about what we can support natively.

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

Figure 5. Generator for increment with shorthand notations. The first `_ ~ ...` abbreviates `frame => frame ~ ...` The remaining underlines are explained in the text.

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.[4] The instruction `param.load` loads the parameter on the stack. Next, `method` generates a call to a method specified by the function `((x:Integer) => x.intValue)`.[5] 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.[6] The final `method` 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 can make this example even more concise. A placeholder, `_`, used in an expression context automatically creates an abstraction. For instance, the syntax `((_:Integer).intValue)` is short for `((x:Integer)=>x.intValue)` and the placeholder expression `(Integer.valueOf(_:Int))` abbreviates the abstraction `((x:Int)=>Integer.valueOf(x))`. Figure 5 contains the same example, but this time maximally exploiting shorthand notations.

4.1. Branching Code

The combination of instructions using the ~ operator only creates linear control flow. To implement loops and conditionals, branching instruc-

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

[5] Method calls generated in this way are type-safe but they must be known at compile time of the generator. Section 6.4 explains how to invoke methods which are unavailable at compile time of the generator. Type safety of the latter cannot be guaranteed statically.

[6] See Figure 9 for the types of the instructions.

```
def fCountdown[ST<:Stack] = ((f : F[ST]) =>
      f ~ bipush (5) ~ withTargetHere( target =>
        _ ~ dup ~ ifne (_ ~
                        dup ~
                        method(printme(_:Int)) ~
                        bipush(1) ~
                        isub ~
                        target.jmp)) ~
      method(Integer.valueOf(_:Int)))
```

Figure 6. Generator for a countdown function.

tions are needed along with instructions to create branching targets, as there are no explicit addresses in Mnemonics.

Figure 6 demonstrates this principle with the function `fCountdown`. The intended functionality of the code is to count down from 5 to 1 and print these numbers using the function `printme` defined by

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

The function `fCountdown` introduces a jump target by using `withTargetHere`. The latter function invokes the supplied closure with a jump handle `target` that designates the position of `withTargetHere`. The type of `target` is parameterized with the frame state type of the position where it was created. This typing ensures that the jump target is only used from places in the method with a compatible stack type as required by the JVM. The scoping of the closure furthermore ensures that the jump target is not used outside of the defined method.

The function `withTargetHere` cannot be used to generate irreducible control flow. Nested uses give rise to nested, reducible loops according to the nesting of scopes.

The `ifne` instruction expects an 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 Section 9.1) 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 [34]) is error prone and unwieldy, Mnemonics provides predefined operators that implement common patterns for conditionals and loops. For example, the operator `ifne2` expects an integer value on top of the stack. It takes two code sequences of identical type (they both map the same source frame state to the same target frame state) and creates a conditional that executes one of the two sequences depending on the condition. As another example, the operator `iterate` takes a command sequence and provides a self reference to enable starting the next iteration if desired. Here is an example use:

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

The type parameters of `iterate` 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.

4.2. Local Variables

The JVM instructions x`store` and x`load` move values between local variables and the top of the stack (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 [21], 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]`, which is further structured as a subtype of a capability `LocalR[T]` for just reading the local variable:

```
trait LocalR[+T] {
  def load[ST<:Stack, T2 >: T] :F[ST]    => F[ST**T2]
}
trait Local[T] extends LocalR[T] {
  def store[ST<:Stack]         :F[ST**T] => F[ST]
}
```

The introduction of a scoped local variable stores the current value on top of the stack in the new variable and passes the corresponding `Local` capability to the `body` closure. The lifetime of the local variable is confined to the extent of this closure. Here is the signature of local variable introduction:

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

The supertype `LocalR[T]` for read-only access to a variable enables additional possibilities for subtyping. The variance annotation `+T` makes the `T` parameter covariant and the supertype constraint `T2 >: T` enables the type of the object pushed on the stack to be a supertype of the stored type. The latter constraint is needed because type synonyms like `**` are invariant, but the stack behaves covariantly in its element parameters. Furthermore, this choice enables less restrictive uses of abstractions. For example, consider a generator that generates code for performing an iteration with a signature like the following:[7]

```
def foldIterable[R <: Stack, T <: AnyRef, U: Category1]
      (iterable: LocalR[java.lang.Iterable[T]])
      (func: LocalR[java.util.Iterator[T]] => F[R**U**T] => F[R**U])
    : F[R**U] => F[R**U]
```

Thanks to covariance, this function accepts an `iterable` argument of type `Local[java.util.List[T]]`: It has `LocalR[java.util.List[T]]` as a supertype and thus a subtype of `LocalR[java.lang.Iterable[T]]`, as desired.

Safe use of capabilities requires that their extent coincides with their static scope: they should not escape from it. Consequently, the generator should neither store them in global data structures nor return them explicitly or as part of an exception. Unfortunately, Scala provides no linguistic tools to ensure this encapsulation. The programmer of the generator is responsible to ensure that capabilities do not escape from their defining scope.

The Haskell community has developed a method to deal with the escaping capabilities problem using rank-2 polymorphism [16, 14]. Implementing this idea requires indexing the frame type and the capability type with an extra type parameter. Introducing a new capability creates a new universally quantified index type that links a frame type with its capability. Unfortunately, it is cumbersome to scale this approach to multiple nested scopes of capabilities, so that its applicability to the problem at hand appears limited.

4.3. High-level Composite Instructions

As already suggested at the end of the previous section, 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

[7] See Section 6.3 for an explanation of the type `Category1`.

```
def foldArray[ST<:Stack,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 =>
          iterate[ST**U,ST**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 7. Folding an array with an inlined combining function.

over an array while accumulating some result. Figure 7 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.

The function `foldArray` sets up a counter on the stack and iterates over the array indices using `iterate`. 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`.

5. Case Study: ObjectFormatter

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 directly to bytecode and measured the resulting performance improvement. This compiler is called ObjectFormatter. For space reasons, the paper contains an abridged version of the full account in Rudolph's thesis [34].

Given a format string and the type `T` of the objects to be formatted, ObjectFormatter generates a function of type `T => String`. As an example, suppose a string is to be created out of an object of type `Person`:

```
> 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 fields (as in `#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 display values of arbitrary type, specify the format of date values, perform conditional formatting depending on the `Boolean` or `scala.Option` value of an instance variable, and to flexibly display arrays and lists.

ObjectFormatter works in two stages, both of which happen at run time of the program. In the first stage, it parses the format string into a typed AST. This stage checks the syntax of the format string, resolves the names of all fields and methods, and performs a type check before generating the AST. This stage may signal run-time errors, if the format string is malformed or asks for non-existing fields or methods. The second stage compiles the typed AST to bytecode. This stage never fails and always generates type-correct bytecode.

Figure 8 contains an overview of the classes comprising the AST. A format string is a sequence of format elements. Accordingly, a format string is parsed into an AST element `FormatElementList[-T]` con-

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

trait FormatElement[-T]
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:List[FormatElement[T]])
```

Figure 8. AST classes.

taining a list of parsed format elements of type `FormatElement[T]`. The type parameter `T` is marked as contravariant (`-T`) because given a `FormatElement[T1]` which can format an object of type `T`, it can always format an object of `T2 <: T1`, as well. Therefore, `FormatElement[T1]` has to be a subtype of `FormatElement[T2]` and this is expressed by declaring contravariance of `T`. In this respect `FormatElement[T]` behaves like a function of type `T => String`, which is also contravariant in `T`.

Two subtypes of `FormatElement` are shown here: First, `Literal` which represents literal string parts of the format string. `Literal` extends `FormatElement[Any]` because it does not depend on the type of the object to be formatted. Second, `ToStringConversion` for formatting the result of the evaluation of an expression by calling the `toString` method on the result.

An expression is represented by an AST element of type `Exp[T,U]`. Each expression is a unary function from a value of type `T` to a value of type `U` represented by a sequence of method calls. For illustration, we consider two subtypes of `Exp`. `MethodHandleExp` represents a method call on the current object (like `#town` in the example). The method to call is represented by an instance of `Method1`, a typed method reflection handle (cf. Section 6.4). The other subtype of `Exp` is `ParentExp`, which represents the composition of two expressions. For example, the formatting instruction `#town.length` yields the number of characters of the `town` field, which would be translated to `p.town.length`. The class `ParentExp` has two fields, `parent` represents the expression before the dot (`town` in the example), and `inner` represents the expression which uses the result of `parent` to select a component value (`length` in the example). There are further AST elements not shown in this excerpt for formatting expressions of collection types, dates, as well as boolean and `Option` values.

The FormatCompiler transforms the typed AST into a code generator. The generated code uses an instance of java.lang.StringBuilder to successively append the elements of the format, evaluating and converting expressions as they arise. The method compileElement is responsible for compiling one FormatElement:

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

It receives a FormatElement to compile and a handle to a local variable that contains the argument of the formatting function. Its generated code appends the string representation of the element to the StringBuilder instance on top of the stack. The body of the function compileElement performs pattern matching against all concrete subclasses of FormatElement. As an example, consider the case for ToStringConversion:

```
case ToStringConversion(e) =>
  f ~ value.load ~
    compileExp(e) ~
    method((_:AnyRef).toString) ~
    method((_: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:[8]

```
def compileExp[R<:Stack,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)
    }
```

It evaluates an expression that transforms a stack of type F[R**T] to type F[R**Ret]. If the expression is a 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 the fields of ParentExp, because a value of type T on the stack must be transformed to a value of type U via an intermediate value of type V.

[8] See Section 6.4 for an explanation of NoUnit.

6. 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 [15]. Function types and closures are also essential to model method calls and jump instructions.
- Variance annotations have proved crucial for implementing the structural constraints (see Section 9.1).
- 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.

6.1. 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 [12, 40]. The implementation in Mnemonics is inspired by the HList library [15] and its adaptation to Scala [23]. It defines the types `Stack`, `Nil`, and `Cons` by

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

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 ** [R<:Stack,T] = Cons[R,T]
```

With this definition, the type of the example stack is

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

The introduction of the type synonym favorably interacts with Scala's typing algorithm, which switches between inference mode and checking mode depending on the context [26]. As the type synonym does not indicate the variance of its parameters, the inference mode of Scala's typing algorithm treats its parameters as invariant. However, in a type checking context, Scala expands the type synonym to the underlying `Cons` type and performs the type check with the variance annotations in force. This arrangement enables Scala to infer more types than without the type synonym (or with a variance-annotated version of the type synonym) while retaining the needed flexibility of matching stack types at join points in the control flow.

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 were needed to access and update a local variable (see the thesis [34]).

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.

6.2. Arithmetic Operations and Stack Manipulation

The instructions for performing arithmetic operations and for stack manipulation are straightforward. Each of them can be described by a suitable polymorphic type. The polymorphic type of the rest of the stack ensures that the operation cannot touch anything on the run-time stack beyond the elements explicitly stated in the type [31].

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 Figure 9: `Bipush` takes a `Byte` and pushes it as an `Int`. `Ineg` consumes the top

```
/* operations on integers */
def bipush[R<:Stack]
    (value:Byte)            :F[R]                    => F[R**Int]
def ineg[R<:Stack]          :F[R**Int]               => F[R**Int]
def iadd[R<:Stack]          :F[R**Int**Int]          => F[R**Int]
/* type cast */
def checkcast
    [R<:Stack,T<:AnyRef,U]
    (cl:Class[U])           :F[R**T]                 => F[R**U]
/* array access */
def aload[R<:Stack,T<:AnyRef] :F[R**Array[T]**Int]   => F[R**T]
def aload[R<:Stack]         :F[R**Array[Byte]**Int] => F[R**Byte]
/* and so on */
```

Figure 9. Instruction signatures.

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` attempts to cast the top element of the stack into the supplied type. If this cast succeeds, the element remains unchanged but with the new type on top of the stack. Otherwise, the JVM throws a `java.lang. ClassCastException` . The top element must be a reference type for `checkcast` to work correctly.

Scala's type system incorporates arrays in exactly the same way as Java. Hence, the array instructions obtain types in the same way as other basic JVM instructions. In particular, there are specialized array 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. Mnemonics provides a more convenient interface through a suitably overloaded `aload` instruction (see Figure 9), which is internally resolved to the appropriate JVM instruction.

The instructions for manipulating the stack are in principle no more complicated than the arithmetic instructions. However, they require distinguishing between different operand sizes as explained in the next subsection.

6.3. 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 types `Double` and `Long` are the only category 2 types.

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 that 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 words* on the stack. The problem is that each of the words must be an encoding of a category 1 type value to ensure that `swap` works as expected. The solution is to equip the `swap` instruction with *implicit conversions* (requested by the constraint `T1<%Category1`) that force the values on top of the stack to be category 1 types.

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

An implicit conversion is applied by the compiler in case of a type mismatch. The compiler searches for a suitable implicit definition in scope and inserts it automatically. With this machinery, 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 rejects any attempt to apply `swap` to a stack with a category 2 type entry on top or just below the top. These definitions are only present to make the `<%` constraints satisfiable, they are never actually executed.

Figure 10 contains the signatures of some further stack manipulating instructions affected by different sizes. The instructions `pop` and `pop2` remove the top element from the stack, for a category 1 type and for a category 2 type, respectively. The `pop2` instruction may also be used to remove two elements of category 1 type from the top of the stack. The `dup` instruction duplicates the top element of the stack. The typing of the remaining instructions for reorganizing the stack (`dup_x1` / `dup_x2` and their 2-word variants `dup2_x1` / `dup2_x2`) follows the same pattern and is thus omitted.

Local variables and parameters present no problems. They are represented by typed capabilities that internally check their type parameter

```
def pop[R<:Stack,T]     :F[R**T]           => F[R]
def pop2[R<:Stack,T<%Category2]
                        :F[R**T]           => F[R]
def pop2[R<:Stack,T1<%Category1,T2<%Category1]
                        :F[R**T1**T2]      => F[R]
def dup[R<:Stack,T]     :F[R**T]           => F[R**T**T]
def dup2[R<:Stack,T<%Category2]
                        :F[R**T]           => F[R**T**T]
def dup2[R<:Stack,T1<%Category1,T2<%Category1]
                        :F[R**T1**T2]      => F[R**T1**T2**T1**T2]
```

Figure 10. Signatures for stack manipulation.

to generate the appropriate JVM instructions. In particular, the internal storage manager reserves adjacent addresses for local variables of category 2 type.

One might be tempted to solve this problem by introducing a new supertype of `AnyRef`, `Int`, `Boolean`, and so on. However, like most other languages, Scala does not permit the retroactive introduction of supertypes.

6.4. Method Call and Field Access

Section 4 already gave a glimpse of Mnemonics's treatment of method calls using the function `method`:

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

Although the programmer writes a function as the parameter of `method`, the Scala compiler does not translate it in that way. Instead, the compiler type checks the `code` parameter of `method`, 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 `method`:

```
def method[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, `method` 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 `method` covers unary static method calls and nullary instance calls.

There are a few drawbacks with this approach to method invocation. First, the parameter of `method` 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, `method` 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. There are specialized method handle types depending on the numbers of values the method call consumes from the stack (we have implemented arities zero, one, and two, other ones can be added analogously). Likewise, there are overloads of the method handle factories, `method` and `methodHandle` as shown above, for all the arities supported.

Type `Method1` is the type of a method handle for a method call that consumes one value from the stack, this can be either a static method call with one argument or a parameterless method call of an instance method where the receiver is the only value taken from the stack.

The declaration 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<:Stack,T1X<:T,UX>:U <% NoUnit]()
    : F[R**T1X] => F[R**UX] = f => f.invokemethod(this)
  def invokeUnit[R<:Stack,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 with the internal function `Instructions.pop_unit` 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<:Stack,T,U<%NoUnit](m:Method1[T,U])
  : F[R**T]=>F[R**U] = m.invoke()
implicit def uCall1[R<:Stack,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`.

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`. Fields of an instance may be accessed and updated with a more pleasing syntax with the typings shown in Figure 11:

```
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.

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

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

Figure 11. Typings for static and dynamic field access.

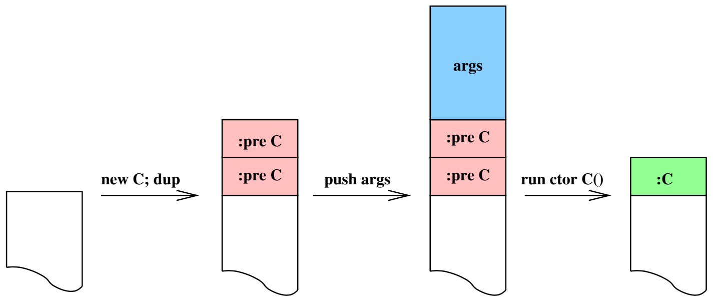

Figure 12. Instance creation in the JVM.

6.5. Instance Creation

Instance creation has some unique problems because it temporarily deposits uninitialized objects on the stack which must not be leaked to the rest of the program. The JVM bytecode verification constraints 7 to 9 specify the rules for instance creation (Figure 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 a two-step process illustrated in Figure 12: First, the `new` instruction allocates an uninitialized object indicated as `:pre C` in the figure. Next, this object is duplicated to serve as the receiver of the constructor call and then the constructor's parameters are pushed on the stack. Finally, the code invokes the constructor. As a constructor method does not return a result, it initializes the instance as a side effect, which is why there is a duplicate reference on the stack, initially. The result is an initialized object on top of the stack indicated by `:C`.

The crucial observation for this process is that there is usually code for evaluating parameters between the `new` and the constructor call. The

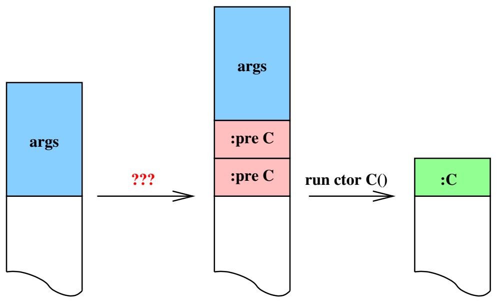

Figure 13. Problem with instance creation.

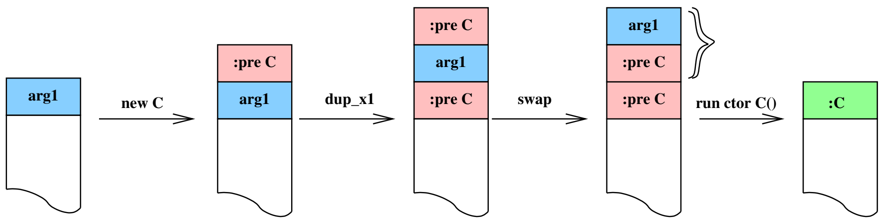

Figure 14. Invoking a one argument constructor.

verifier has to ensure that this code does not access the uninitialized object, which has been a problem in the past [9]. The solution is, of course, not to provide direct access to the JVM's `new` instruction but instead to create wrapper methods that safely encapsulate instance creation. However, the design of these methods is not straightforward. In particular, Mnemonics could not call constructors analogously to other methods, as indicated in Figure 13: If the uninitialized instance is pushed first, it is a problem to guarantee that the instance is not accessed. On the other hand, if the parameters are pushed on the stack before the `new` instruction executes, then the two references to the uninitialized instance must be inserted below the constructor arguments as indicated with the code fragment `???` in Figure 13.

For constructors with zero, one, and two arguments, there are indeed code sequences to insert the two references to the uninitialized instance at the proper place in the stack. Thus, it is straightforward to provide the following functions:

```
def newInstance[ST<:Stack,T](cl:Class[T])
    :F[ST] => F[ST**T]
def newInstance1[ST<:Stack,T,U<%Category1]
    (code:scala.reflect.Code[U=>T])
    :F[ST**U] => F[ST**T]
def newInstance2[ST<:Stack,T,U<%Category1,V<%Category1]
    (code:scala.reflect.Code[(U,V)=>T])
    :F[ST**U**V] => F[ST**T]
```

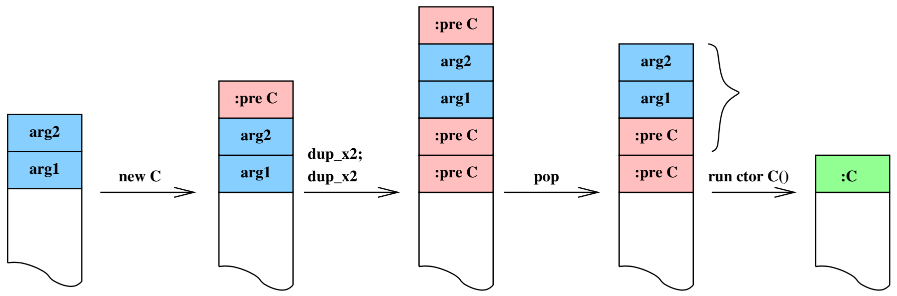

Figure 15. Invoking a two argument constructor.

The function `newInstance` invokes the single nullary constructor of the class. The functions `newInstance1` and `newInstance2` expect the values of the constructor's arguments on the stack, but require that they are restricted to category 1 types to ensure correct operation of the stack rearrangement. Both rely on `scala.reflect.Code` like `method`. The code generated by either function expects that the top of the stack already contains the evaluated arguments. Then the code creates the new instance, swaps the stack entries (cf. Figure 14 and 15), and employs reflection to generate the invocation of the correct (potentially overloaded) constructor. Thus, the code fulfills the constraints by construction.

However, this technique is limited because it cannot handle constructors with more than two arguments (respectively, more than one argument of category 2 type). Looking for a more general arrangement, it seems feasible to define a function that takes for each constructor argument a code generator which is polymorphic with respect to the rest of the stack as a parameter. The polymorphic type of the stack guarantees that none of them inspects the stack below, so that these code generators are safe to invoke even with uninitialized objects on the stack. As another advantage, this approach is completely oblivious with respect to category 1 and category 2 types.

Here is a code snippet with suitable definitions for a one-argument constructor call:

```
def ctor1[R<:Stack,T,U] (cl:Class[U], arg: CtorArg[T])
  : F[R] => F[R**U]

trait CtorArg[T] {
  def arg[Rest<:Stack] : F[Rest] => F[Rest**T]
}
```

A major concern in the design of Mnemonics is ease and convenience of use. For example, we would want to write:

```
  ...
~ ctor1 (classOf[Integer],
        new CtorArg () { val arg = _ ~ ldc("747") })
```

```
~ ...
```

This idiom for defining the polymorphic `arg` method is not accepted by Scala's type inference algorithm because it requires rank-2 polymorphism. While there is a type inference algorithm for rank-2 types [13], it is not implemented in any production compiler. All implemented systems require type annotations for a rank-2 polymorphic function like `arg`. This approach works in Scala, too:

```
new CtorArg[String] {
  def arg[Rest<:Stack]
    : F[Rest] => F[Rest**String]
    = (f:F[Rest]) => f ~ ldc("747")
}
```

Thus, for the general case, we propose a range of constructor functions that accept the arguments on the stack but store them in local variables to clear the stack for creating the new instance. For two arguments, such a function looks as follows:

```
case class Ctor2[T1, T2, U]
  (constructor: java.lang.reflect.Constructor[_]) {
  def apply[R <: Stack](): F[R**T1**T2] => F[R**U] =
    withLocal { arg2 => _ ~
      withLocal { arg1 => _ ~
        newUninitialized(constructor.getDeclaringClass) ~
        dup ~
        arg1.load ~ arg2.load ~
        invokeconstructor(this)
    }}
}

def ctor2[T1, T2, U]
  (c: scala.reflect.Code[(T1, T2) => U]): Ctor2[T1, T2, U] =
  Ctor2[T1, T2, U](CodeTools.constructorFromTree(c.tree))
```

This implementation is structured analogously to method invocation. The generator uses `ctor2` with a suitable closure as its argument. The resulting `Ctor2` instance is then applied to `()` to expose the frame transformer that generates the actual code. This frame transformer first stores the arguments in new local variables, then allocates a new, uninitialized instance (using an unsafe backend method `newUninitialized`, which is not available to the user of the library), creates the stack arrangement for invoking the constructor, and finally performs the invocation (again with an unsafe backend method `invokeconstructor`).

The code generated by this function is completely general. It does not violate any restrictions imposed by the JVM and it is also oblivious of category 1 and 2 types. A small drawback seems to be that it squanders local variable space, but in our experiments we found that the argument values mostly end up in their target registers during JIT compilation. So, there is not even a performance penalty.

```
(bytes:Local[Array[Byte]]) =>
  ... ~
  bytes.load ~
  ldc("UTF-8") ~
  method(Charset.forName(_: String)) ~
  ctor2(new java.lang.String(_: Array[Byte], _: Charset)) () ~
  ...
```

Figure 16. Calling a constructor.

Figure 16 contains an example use of the constructor interface that invokes the `String` constructor that converts a `byte` array to a `String` using a given character set encoding.

6.6. 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[R<:Stack,U<:AnyRef]{
  def jmp:F[R**U] => Nothing
}
```

The typing indicates that the `jmp` method expects a stack with one element of type `U` on top and does not return as indicated by the type `Nothing`. Mnemonics uses it as the result type of unconditional control transfers to indicate that instructions after the control transfer are never reached. The remaining contents of the stack, indicated by `R`, are discarded by the JVM.

Its use with the conditional branches is straightforward:

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

Both the function passed to the `compile` method and the function passed to the conditional branch instruction `ifne` return `Nothing`, which fits perfectly with the type of a control transfer.

6.7. The Switch Instruction

The JVM contains instructions for jumping to different targets based on the integer value on top of the stack to facilitate the `switch` instruction of Java. Mnemonics defines an encoding for the lookup-switch instruction. A lookup-switch is defined by a mapping from integer keys to branch targets. Scala has a syntactic element similar to Java's `switch` instruction, pattern matching. Pattern matching facilitates specifying a generator for the switch instruction with a fairly intuitive syntax. The following signature specifies a suitable method:

```
def lookupSwitch[R <: Stack, ST2 <: Stack]
  (candidates: Int*)
  (mapping: F[R] => Option[Int] => F[ST2])
  : F[R**Int] => F[ST2]
```

Method `lookupSwitch` is a frame transformer which expects an integer value on top of the stack. For each branch a stack transformer has to be specified. An obvious way to specify the mapping would be as a list of key-stack-transformer tuples. However, to exploit Scala's pattern matching facilities, it is necessary to specify the mapping with a value of type `F[R] => Option[Int] => F[ST2]`. With this type, the `lookupSwitch` generator can be used as in the following example code:

```
lookupSwitch(1, 5)(f => key => key match {
  case Some(1) => f ~ ldc("one")
  case Some(5) => f ~ ldc("five")
  case None => f ~ ldc("unknown number")
})
```

The mapping is established by a function which transforms the frame given as the first parameter depending on the key given as another parameter. By convention, for an argument `Some(key)` the generator for the branch for `key` must be returned and for the argument `None` the generator for the default branch must be returned. The backend calls this generator function once for each key in the list `candidates` and once with `None` to generate the default branch.

Scala has a shorthand notation for unary functions which consists of only a pattern match on the argument which allows further shortening the use of this instruction:

```
lookupSwitch(1, 5)(f => {
  case Some(1) => f ~ ldc("one")
  case Some(5) => f ~ ldc("five")
  case None => f ~ ldc("unknown number")
})
```

Besides the lookup switch, the JVM defines another type of switching instruction which is currently unsupported by Mnemonics, the table switch. As with the lookup switch instruction, a mapping from keys to branch targets has to be established. However, when using a table switch instruction, a table with the targets has to be specified for a closed range of integer keys, as well as the range itself by defining the least and the greatest key from the table. The dispatch is resolved by calculating an index into the table of targets. Mnemonics could use a similar pattern to encode table switch instructions, as well.

6.8. A Brief Look at Subroutines

The current implementation of Mnemonics does not support subroutines. However, a facility similar to `CtorArg`, as suggested in Section 6.5 for instance creation, could be put to use. The important constraints are the following. A subroutine must be callable from different places in the bytecode. Hence, its type must be polymorphic in the stack fragment on which it is called. This observation dictates the definition of the subroutine trait, which again relies on rank-2 polymorphism (so that type inference is out of reach).

```
trait Subroutine {
  def jsr[ST<:Stack] : F[ST] => F[ST]
}
```

To make use of a subroutine, the library would provide another capability factory.

```
def withSubroutine[ST1<:Stack,ST2<:Stack]
  (sub : Subroutine) (code : Subroutine => F[ST1] => F[ST2]) :
  F[ST1] => F[ST2]
```

A typical use would look like this:

```
withSubroutine(
  new Subroutine {
    def jsr[ST<:Stack] : F[ST] => F[ST]
    = (f:F[ST]) => f ~ body_of_subroutine
  })
  ( (sr) => _ ~ some_code ~ sr.jsr
                  ~ more_code ~ sr.jsr
                  ~ yet_more_code)
```

The implementation of `withSubroutine` would be able to choose in which way to implement the subroutine. It could either create a new label and generate a true subroutine from its `jsr` method, with implicit management of the return address and the `ret` instruction, or just duplicate the code, as done by recent Java compilers. In the first implementation, the code argument gets passed another `Subroutine` instance, where the `jsr` method generates a `jsr` instruction that invokes the subroutine created from the `sub` argument. The duplicating implementation just passes its `sub` argument unchanged to `code`.

7. Design Patterns

Developing a library like Mnemonics which makes heavy use of Scala's type system to prove properties of the user's code is always a compromise between the expressiveness of the host language, the power of the type system, the expectations of the library writer, and the predictability of the actual implementation of the compiler. During the development of Mnemonics we identified a set of recurring patterns which work well with the current language and compiler of Scala (Scala 2.8.1). This section presents an overview of the patterns we came across. For a language as expressive as Scala, this list is, of course, by no means comprehensive. It is notable, however, that in hindsight complex type manipulation and meta-programming as originally reported in Rudolph's thesis [34] (cf. Section 6.1) were both less predictable and had considerably longer compile times than the simpler capability-based constructs presented in this work.

7.1. Capabilities

Capabilities provide an interface for safely accessing resources. A capability is an opaque object which is issued by an authority after checking that access to the requested resource is valid and possible. Mnemonics provides capabilities for local variable slots (`Local[T]`), jump targets (`Target[T]`), returning from methods (`Return[T]`), method invocation (`Method`n), and constructor invocation (`Ctor`n).

There are two different patterns for issuing capabilities in Mnemonics. First, calling a method `withX` and passing a function object `X[ST] => Y`. This function may use the capability object of type `X[ST]` in its scope but must not store it for outside use (cf. local variable access, Section 4.2). In this pattern, the type of the capability usually depends on the inferred stack type at that point. While this design suggests that the scope of the capability is limited to the extent of the closure, this restriction is not enforced at compile time. A programmer may subvert the pattern by storing the capability in a global reference and by using it outside the closure's extent.[9] Currently, Mnemonics can only signal a run-time error in such a case.

The second pattern is used with type-safe method handles. Here, scope is unimportant because at run time, a method is either available all the time or it is not. Hence, a method handle provides a way to assume the signature of an object at compile time where the library can check its validity only at run time. Consider the signature of the method which creates a method handle out of a run-time method reflection instance:

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

[9] There is an experimental extension of Scala's type system to check ownership constraints. Such an extension may be sufficient to guarantee the correct use of capabilities.

At compile time, it is impossible to tell if the `Method` instance has a parameter of type `T` and returns an instance of `U`. However, the method dynamically checks the types at run time (of the generator) and returns a manifestly typed capability of type `Method1[T, U]`, which can safely be used afterwards. This organization separates run-time type-checking from type-safe use when generating method invocations with the handle. For example, it is possible to collect all calls to `methodHandle` (and similar methods) into a central place and completely separate code that may fail at run time and code that is guaranteed to succeed. This separation simplifies error handling.

7.2. Read-only Covariant Interfaces

When passing capabilities (like `Local` for local variable access) it is often desirable for `Local` to be covariant in its type parameter if the local variable is only accessed for reading. This demand is met by separating the read-only part of trait `Local` into the trait `LocalR` as follows:

```
trait LocalR[+T] {
  def load[ST<:Stack, T2 >: T]:F[ST] => F[ST**T2]
}
trait Local[T] extends LocalR[T] {
  def store[ST<:Stack]:F[ST**T] => F[ST]
}
```

By requiring a parameter of type `LocalR[T]`, a method indicates that it does not write to a local variable and, because of covariance, the user of the method can pass local variable capabilities of type `Local[T0]` where `T0` is a subtype of `T`.

Using this technique, in Mnemonics, improves the flexibility of the signature of `foldIterable` (implemented analogously to `foldArray` in Section 4.3).

```
def foldIterable[R <: Stack, T <: AnyRef]
  (iterable: LocalR[java.lang.Iterable[T]]) // was Local[...]
  // [more parameters]
```

Because `LocalR` is covariant another code generator can pass a local variable of a subtype of `Iterable[T]` like `java.util.List[T]` to `foldIterable` which is impossible (and unsafe) with the invariant `Local` capability. Using the same reasoning, it is possible to extract a contravariant interface containing only the write access (although we had no need for it so far).

7.3. Design with Type Inference in Mind

Without type-inference a library like Mnemonics is not feasible at all. The type of the stack can be very complex at certain points in the program, so that a design that forces the programmer to state this type explicitly would not be usable. Like other languages with advanced type systems (like OCaml and Haskell), Scala's designers have traded completeness of type inference for expressiveness. This choice enables them to support the wealth of features offered: subtyping and general bounded polymorphism (with upper and lower bounds) as well as existential types, which are not exploited in this work. Instead of being able to infer types globally for a complete program, Scala can only do so locally [26]. The compiler tries to infer the types along the control flow by looking only at the immediate context of an expression when trying to find its type and then following the control flow.

Mnemonics exploits type inference by chaining stack-transformers in a way that the input stack type is always sufficiently known to the compiler to infer the output types.

At some points it is necessary to specify types explicitly. In particular, when referring to methods via method handles, type annotations are needed to guide the compiler. Removing one layer of indirection at method invocation instructions (cf. `invoke` and `invokeUnit`, see Section 6.4) would enable the compiler to infer those types correctly but, on the other hand, we consider method handles as basic building blocks for programs where an explicit type annotation serves to explicitly express the intentions of the programmer.

Another detail of the Scala compiler is the inference of type parameters when calling methods. The call site either has to specify all the type parameters explicitly or none to enable their inference. As a pattern to enable explicit passing of only some of the type parameters it is possible to split the method call into two successive calls. In the first step, type parameters are required to be passed explicitly and an instance of a helper trait is created which is parameterized with the types. In the second step a method of the helper trait is called without type parameters leaving them for the compiler to infer.

8. Formal Model

The formal model presented in this section illustrates the working of the Mnemonics library and enables a correctness proof. The correctness proof establishes a correspondence between a typed code generator and the execution of a generated program on a typed stack machine. The basis for this correspondence is that the typing of the generator for each instruction models the stack manipulation of that instruction on the typed stack machine.

For simplicity, the typed stack machine (TSM) of this model has only three instructions, which are borrowed from the JVM:

$$\mathit{Ins} ::= \texttt{Bipush}\ i \mid \texttt{Iadd} \mid \texttt{I2f}$$

The instructions are chosen such that simple operations can be modeled and that different types are involved. The `Bipush` instruction is parameterized with an integer i and pushes the integer on the stack, `Iadd` adds the top two integers, and `I2f` the integer on top of the stack to a floating point number.

Towards a formal definition of the semantics of TSM, we define the type of values and stacks manipulated by the TSM as follows:

$$\begin{array}{lcl} \mathit{Value} & ::= & \mathit{Int}\ i \mid \mathit{Float}\ f \\ \mathit{Stack} & ::= & [] \mid \mathit{Stack} :: \mathit{Value} \end{array}$$

That is, a value is either an integer i tagged with *Int* or a floating point number f tagged with *Float*. The stack is a list of such values with [] denoting the empty stack and :: denoting the infix cons (push) operator.

The semantics of a single instruction is a function exec_i of type $\mathit{Ins} \rightarrow \mathit{Stack} \rightarrow \mathit{Stack}$ which is defined analogously to the JVM definition:

$$\begin{array}{lllcl} \mathit{exec}_i & (\texttt{Bipush}\ i) & (\ \mathit{stk} &) = & \mathit{stk} :: \mathit{Int}\ i \\ \mathit{exec}_i & \texttt{Iadd} & (\ \mathit{stk} :: \mathit{Int}\ i :: \mathit{Int}\ j &) = & \mathit{stk} :: \mathit{Int}\ (i+j) \\ \mathit{exec}_i & \texttt{I2f} & (\ \mathit{stk} :: \mathit{Int}\ i &) = & \mathit{stk} :: \mathit{Float}\ (i * 1.0) \end{array}$$

Executing a list of instructions means to execute one instruction after the other:

$$\begin{array}{lllcl} \mathit{exec}_p & [] & \mathit{stk} & = & \mathit{stk} \\ \mathit{exec}_p & (i :: i^*) & \mathit{stk} & = & \mathit{exec}_p\ i^*\ (\mathit{exec}_i\ i\ \mathit{stk}) \end{array}$$

A typing discipline for the stack is defined by the judgment $\vdash_s \mathit{stk} : t$, where types are defined inductively by

$$s, t, u ::= \texttt{Nil} \mid t\ \texttt{**}\ \texttt{Int} \mid t\ \texttt{**}\ \texttt{Float}$$

The (three) inference rules for the judgment are as follows.

$$\vdash_s [] : \texttt{Nil}$$

$$\frac{\vdash_s \mathit{stk} : t}{\vdash_s (\mathit{Int}\ i :: \mathit{stk}) : t\ \texttt{**}\ \texttt{Int} \qquad \vdash_s (\mathit{Float}\ f :: \mathit{stk}) : t\ \texttt{**}\ \texttt{Float}}$$

Next, we define the generating functions. As in Mnemonics, the type of a generator is $F[t_1] \rightarrow F[t_2]$ where t_1 is the type of the input stack and t_2 the type of the output stack. The intuition is that the generated code transforms an input frame to an output frame. The typing of each generator is directly read off the semantics definition of the corresponding TSM instruction.

$$\begin{array}{lll}
\texttt{bipush}\ i & : F[t] & \to F[t\ \texttt{**}\ \texttt{Int}] \\
\texttt{iadd} & : F[t\ \texttt{**}\ \texttt{Int}\ \texttt{**}\ \texttt{Int}] & \to F[t\ \texttt{**}\ \texttt{Int}] \\
\texttt{i2f} & : F[t\ \texttt{**}\ \texttt{Int}] & \to F[t\ \texttt{**}\ \texttt{Float}]
\end{array}$$

In addition to the instruction generators, there is an operator `+++` to concatenate two generated pieces of code and an operator `nil` to create an empty piece of code. The concatenation operator is related to the `~` operator in Mnemonics, which is more restricted because it only attaches one instruction to the end of a code sequence.

$$\begin{array}{ll}
\texttt{+++} & : (F[s] \to F[t]) \to (F[t] \to F[u]) \to (F[s] \to F[u]) \\
\texttt{nil} & : F[s] \to F[s]
\end{array}$$

The internal representation of the frame type is a list of instructions so that the code type actually transforms an instruction sequence. Hence, the generators for single instructions prepend the instruction to the sequence, the composition operator `+++` is function composition, and the `nil` operator is the identity function.

$$\begin{array}{lll}
\texttt{bipush}\ i & = & \lambda i^*.\ \texttt{Bipush}\ i :: i^* \\
\texttt{iadd} & = & \lambda i^*.\ \texttt{Iadd} :: i^* \\
\texttt{i2f} & = & \lambda i^*.\ \texttt{I2f} :: i^* \\
c\ \texttt{+++}\ d & = & \lambda i^*.\ c\ (d\ i^*) \\
\texttt{nil} & = & \lambda i^*.\ i^*
\end{array}$$

The correctness property of the code generator boils down to proving the following theorem, where we call a term of type $F[s] \to F[t]$ composed of the above operators and function application a *typed generator.*

THEOREM 1. *Suppose* $\vdash_s stk_{in} : s$ *and let* $c : F[s] \to F[t]$ *be a typed generator. Then* $stk_{out} = exec_p\ (c\ [])\ stk_{in}$ *is defined and* $\vdash_s stk_{out} : t$.

We actually prove a more general proposition by a straightforward induction on the typing derivation of c using the above typing of the operators and the standard typing rule for function application.

LEMMA 1. *Let* stk_{in} *be a stack such that* $\vdash_s stk_{in} : s$, $c : F[s] \to F[t]$ *be a typed generator, and* i^* *be any instruction sequence. Then* $exec_p\ (c\ i^*)\ stk_{in} = exec_p\ i^*\ stk_{out}$ *for some defined stack* stk_{out} *with* $\vdash_s stk_{out} : t$.

This model and proof is intended to give an idea how a correctness proof for the full Mnemonics implementation could be constructed. As it stands, it ignores several issues, like subtyping and the language embedding: in a more ambitious setting, c could be any lambda term of type $F[s] \to F[t]$.

9. 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 micro benchmark comparing the performance of an interpretive and the compiling version of the ObjectFormatter from Section 5.

9.1. 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, if the type of each single instruction correctly reflects its action on the frame, then 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.

Currently, users of Mnemonics have to trust the developers of the library that they specify the correct type of each instruction. This connection is not formally established. To do so would require to formalize the untyped core of Mnemonics and the underlying untyped bytecode generation library.

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 Section 6.1.

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

which expands to

```
def ifne[R<:Stack,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`.

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 `ASM` backend. `invokespecial` cannot be called directly, the initialization constructs have to be used instead. Mnemonics does not provide a special type for uninitialized instances (see Section 6.5).

Constraints 10 and 11 refer to the compatibility of the receiver's type and the parameters' types with a given method signature. This problem is addressed by specifying the called method as a Scala function value. Because the Scala function type 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 (see Figure 9).

The typing of `ASM.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.

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") ~ method(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.

9.2. Usability

Our experience in the construction of the case study from Section 5 (cf. also the full case study [34]) is that abstracting instruction sequences into composite instructions is a good thing. In particular, each phase of the compiler designed in that study is implemented as a composite instruction. Type annotations were only needed to bootstrap the frame state type when calling `ASM.compile` and in the definition of a composite instruction. In some cases, it was 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 they sometimes involve stack types of substantial size. Moreover, to avoid ambiguities, 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.[10] Finally, the limitations of Scala's type inference are not easily foreseeable so that experimentation is sometimes required.

[10] When an implicit parameter cannot be found the compiler usually emits a generic error message. Later versions of Scala support a mechanism to improve those error messages. An annotation `@implicitNotFound` with the custom error message is put at the definition site of a class which is used as an implicit parameter

Format	Interpreter	Compiler	Speedup int./comp.
Literal string	30.0 ± 5	20.7 ± 2.7	1.45
Simple method	86.0 ± 9	70.2 ± 2.0	1.22
Array expansion	608.0 ± 2.3	155.7 ± 0.8	3.90
Conditional/Option	85.4 ± 1.5	25.0 ± 1.6	3.41

Figure 17. Run times in ms (average ± standard deviation) for 100000 invocations of `formatter.format`, averaged over 20 runs.

9.3. 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. Figure 17 shows the result, running the benchmark on an Intel Core 2 Duo 1.666 GHz using the OpenJDK 1.6.0_0-b12 Server VM. 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 warm up. 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 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. Further details may be found in the thesis [34].

10. Related Work

10.1. 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

type. This error message is then used instead of the generic one. However, in its current implementation this mechanism is quite limited because in many cases a failed search for an implicit value in the compiler does not directly fail with an error but often triggers other, again generic, typing errors. This defeats the original purpose of the annotation of providing domain specific, customized error reporting.

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[11], 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 [2]. 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 [5]. 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 bytecode instruction sequences when replacing or inserting them. This template compiler guarantees that the transformed bytecode is verifiable.

Javassist also has a bytecode level API[12] which facilitates direct access to bytecode instructions including their generation and transformation. These manipulations work either offline on bytecode or at

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

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

load time through a custom class loader. Using the bytecode-level API entails no verification guarantees.

In summary, the existing libraries are able to generate all bytecodes, but they do not statically guarantee verifiable bytecode. Mnemonics does so, but at the price of requiring the use of type-safe patterns for some bytecode instructions. 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 generates verifiable bytecode. To support additional instructions only some simple generation-time checks are required.

10.2. Foundations

Jones [12] 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 [40]. While Jones does not address the issue of subtyping, Yelland does so by encoding subtyping with polymorphism [8]. 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 JVM instruction set verbatim, but rather in providing a convenient DSL for the typed generation of verified bytecode sequences. Thus, Mnemonics models some instructions precisely, but other instructions are only indirectly available. Some further components of bytecode generation, like full control of classes and methods, are not in the scope of Mnemonics.

The idea of typing functions 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., [22]). More generally, the idea of using polymorphism to ignore the presence or absence of parts of a data structure can be traced back to the use of bounded polymorphism with record types [4].

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 [36] on verifying subroutines and by Freund and Mitchell [9] who focus on object initialization. Again, these works aim at *specifying* the semantics of bytecode and deal with features not covered by Mnemonics.

Oury and Swierstra [28] argue for using a dependently typed language to construct DSLs which capture many invariants of the domain. While it is certainly possible to construct a DSL describing code generation for a stack machine with their methods, the integration of subtyping in a dependently typed programming language is still a research problem. However, subtyping is essential for generating JVM bytecode.

10.3. Metaprogramming

Metaprogramming, like code generation at run time, has traditionally taken place in an untyped setting, that is, in languages like Lisp [1] and Smalltalk. Even there, the semantic issues are non-trivial [37]. For typed languages, Taha and Nielsen have shown how to integrate type-safe run-time code generation with the possibility of running the generated code into a functional programming language [39]. This approach is implemented in the MetaOCaml system [38]. However, MetaOCaml relies on a specially tailored multi-level type 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 [17], Tempo [6], 'C [29], and DyC [11]. DynJava [27] is similar in spirit to 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.

Lightweight modular staging [32] is an approach to run-time code generation in Scala. While it builds on the idea that Scala's type system is well suited to express domain-specific constraints, its main emphasis is on the clever implementation of staging and in particular on choosing the right representation for generated code. Both aspects are non-issues in Mnemonics: there is no staged code, rather there are dedicated operators for generating code; also the representation of the generated code is fixed to JVM bytecode.

There are also relationships to typed, heterogeneous metaprogramming and embedded DSLs as touched upon by Czarnecki and coworkers [7]. 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 [20] 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.

11. 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 usability of the library.

The design of Mnemonics emphasizes ease of use and safety. Although it cannot generate arbitrary bytecode sequences, it is sufficiently expressive to accommodate all bytecode generation needs for functional DSLs. In most cases, the bytecode instructions are directly accessible in a type-safe way. In a few special cases, the library supports type-safe patterns that ensure safety of the generated code.

The same ideas are applicable to other code generation tasks with a stack machine as the target machine, for example, CIL. The only restriction is that the target machine's type system must be embeddable in Scala's type system.

It is interesting to speculate about further uses of the Mnemonics approach. For example, most bytecode frameworks support the implementation of bytecode transformations, but Mnemonics currently does not. It might be possible to extend the type-based approach such that it guarantees that an instruction or a sequence of instructions is only transformed to an instruction sequence of the same type, for example: replace method body, replace field access, replace method call, insert code at beginning or end of method (monitoring or logging), inline method. However, there are many transformations that are geared towards transforming the types as well (for example: introduce interface, insert field, insert method, merge classes). It is unclear how they could benefit from the Mnemonics approach.

References

1. Bawden, A.: 1999, 'Quasiquotation in Lisp'. In: O. Danvy (ed.): *Proceedings ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM '99*. San Antonio, Texas, USA, pp. 4–12. BRICS Notes NS-99-1.

2. bcel: 2011, ‘Homepage of BCEL’. `http://jakarta.apache.org/bcel/index.html`.
3. Bruneton, E.: 2007, ‘ASM 3.0 – A Java Bytecode Engineering Library’. `http://download.forge.objectweb.org/asm/asm-guide.pdf`.
4. Cardelli, L. and P. Wegner: 1985, ‘On Understanding Types, Data Abstraction, and Polymorphism’. *Computing Surveys* **17**, 471–522.
5. Chiba, S. and M. Nishizawa: 2003, ‘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.* Erfurt, Germany, pp. 364–376.
6. Consel, C., J. Lawall, and R. Marlet: 1998, ‘Tempo Specializer — Users’ manual’. `http://phoenix.labri.fr/software/tempo/doc/tempo-doc-user.html`.
7. Czarnecki, K., J. T. O’Donnell, J. Striegnitz, and W. Taha: 2003, ‘DSL Implementation in MetaOCaml, Template Haskell, and C++’. in [18], pp. 51–72.
8. Fluet, M. and R. Pucella: 2006, ‘Phantom Types and Subtyping’. *Journal of Functional Programming* **16**(6), 751–791.
9. Freund, S. N. and J. C. Mitchell: 1999, ‘A Type System for Object Initialization in the Java Bytecode Language’. *ACM Transactions on Programming Languages and Systems* **21**(6), 1196–1250.
10. Freund, S. N. and J. C. Mitchell: 2003, ‘A Type System for the Java Bytecode Language and Verifier’. *Journal of Automated Reasoning* **30**(3-4), 271–321.
11. Grant, B., M. Mock, M. Philipose, C. Chambers, and S. J. Eggers: 2000, ‘DyC: An Expressive Annotation-Directed Dynamic Compiler for C’. *Theoretical Computer Science* **248**(1-2), 147–199.
12. Jones, M. P.: 1998, ‘The Functions of Java Bytecode’. In: S. Eisenbach (ed.): *Formal Underpinnings of Java.*
13. Kfoury, A. J. and J. B. Wells: 1994, ‘A Direct Algorithm for Type Inference in the Rank-2 Fragment of the Second-Order lambda-Calculus’. In: *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming.* Orlando, FL, USA, pp. 196–207.
14. Kiselyov, O. and C. chieh Shan: 2007, ‘Lightweight Static Capabilities’. *Electronic Notes in Theoretical Computer Science* **174**(7), 79–104.
15. Kiselyov, O., R. Lämmel, and K. Schupke: 2004, ‘Strongly Typed Heterogeneous Collections’. In: *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell.* Snowbird, Utah, USA, pp. 96–107.
16. Launchbury, J. and S. L. Peyton Jones: 1995, ‘State in Haskell’. *Lisp and Symbolic Computation* **8**(4), 293–341.
17. Lee, P. and M. Leone: 1996, ‘Optimizing ML with Run-Time Code Generation’. In: *Proceedings of the 1996 Conference on Programming Language Design and Implementation.* Philadelphia, PA, USA, pp. 137–148.
18. Lengauer, C., D. S. Batory, C. Consel, and M. Odersky (eds.): 2004, ‘Domain-Specific Program Generation, International Seminar’, Vol. 3016 of *Lecture Notes in Computer Science.* Dagstuhl, Germany:, Springer-Verlag.
19. Lindholm, T. and F. Yellin: 1999, *The Java(tm) Virtual Machine Specification.* Addison-Wesley, 2nd edition edition.
20. Mainland, G., G. Morrisett, and M. Welsh: 2008, ‘Flask: Staged Functional Programming for Sensor Networks’. In: P. Thiemann (ed.): *Proceedings International Conference on Functional Programming 2008.* Victoria, BC, Canada, pp. 335–346.

21. Morris Jr., J. H.: 1973, 'Protection in Programming Languages'. *Communications of the ACM* **16**(1), 15–21.
22. Morrisett, J. G., K. Crary, N. Glew, and D. Walker: 1998, 'Stack-Based Typed Assembly Language'. In: *TIC '98: Proceedings of the Second International Workshop on Types in Compilation*, Vol. 1473 of *Lecture Notes in Computer Science*. Kyoto, Japan, pp. 28–52.
23. Nordenberg, J.: 2008, 'MetaScala'. `http://www.assembla.com/wiki/show/metascala`.
24. Odersky, M., 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: 2006, 'An Overview of the Scala Programming Language (2. Edition)'. Technical report, EPFL Lausanne.
25. Odersky, M., L. Spoon, and B. Venners: 2008, *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Inc, 1st edition.
26. Odersky, M., C. Zenger, and M. Zenger: 2001, 'Colored Local Type Inference'. In: H. R. Nielson (ed.): *Proceedings 28th Annual ACM Symposium on Principles of Programming Languages*. London, England, pp. 41–53.
27. Oiwa, Y., H. Masuhara, and A. Yonezawa: 2001, 'DynJava: Type Safe Dynamic Code Generation in Java'. In: *3rd JSSST Workshop on Programming and Programming Languages (PPL2001)*.
28. Oury, N. and W. Swierstra: 2008, 'The Power of Pi'. In: J. Hook and P. Thiemann (eds.): *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. Victoria, BC, Canada, pp. 39–50.
29. Poletto, M., W. C. Hsieh, D. R. Engler, and M. F. Kaashoek: 1999, ''C and tcc: A language and compiler for dynamic code generation'. *ACM Transactions on Programming Languages and Systems* **21**(2), 324–369.
30. Qian, Z.: 1998, 'A Formal Specification of Java Virtual Machine Instructions for Objects, Methods and Subroutines'. In: *Formal Syntax and Semantics of Java*. pp. 271–312.
31. Reynolds, J. C.: 1983, 'Types, Abstraction, and Parametric Polymorphism'. In: R. E. A. Mason (ed.): *Information Processing '83*. pp. 513–523.
32. Rompf, T. and M. Odersky: 2010, 'Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs'. In: E. Visser and J. Järvi (eds.): *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010*. Eindhoven, The Netherlands, pp. 127–136.
33. Rose, E.: 2003, 'Lightweight Bytecode Verification'. *Journal of Automated Reasoning* **31**(3-4), 303–334.
34. Rudolph, J.: 2009, 'Mnemonics: Type-safe Bytecode Combination in Scala'. Diploma thesis, Albert-Ludwigs-Universität Freiburg. `http://virtual-void.net/files/mnemonics.zip`.
35. Rudolph, J. and P. Thiemann: 2010, 'Mnemonics: Type-Safe Bytecode Generation at Run Time'. In: J. P. Gallagher and J. Voigtländer (eds.): *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2010*. Madrid, Spain, pp. 15–24.
36. Stata, R. and M. Abadi: 1999, 'A Type System for Java Bytecode Subroutines'. *ACM Transactions on Programming Languages and Systems* **21**(1), 90–137.
37. Taha, W.: 2000, 'A Sound Reduction Semantics for Untyped CBN Multi-Stage Computation'. In: J. Lawall (ed.): *Proceedings ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM '00*. Boston, MA, USA.

38. Taha, W.: 2004, ‘A Gentle Introduction to Multi-stage Programming’. in [18], pp. 30–50.
39. Taha, W. and M. F. Nielsen: 2003, ‘Environment Classifiers’. In: G. Morrisett (ed.): *Proceedings 30th Annual ACM Symposium on Principles of Programming Languages*. New Orleans, LA, USA, pp. 26–37.
40. Yelland, P. M.: 1999, ‘A Compositional Account of the Java Virtual Machine’. In: A. Aiken (ed.): *Proceedings 26th Annual ACM Symposium on Principles of Programming Languages*. San Antonio, Texas, USA, pp. 57–69.