Mnemonics – Type-Safe Bytecode Generation in Scala

Johannes Rudolph
CleverSoft GmbH, Munich
Peter Thiemann
Albert-Ludwigs-Universität Freiburg, Germany

Scala Days 2010, Lausanne, 15.04.2010
Existing bytecode generation libraries

- ASM
- CGLIB/BCEL
- Javassist
- Soot

Advantages:
- Proven to work
- well-engineered
- full-featured

Some disadvantages:
- Verbosity
- Complexity
- no particular support for Scala
- user often has to fight with the JVM verifier
What is Mnemonics?

A library which
  • is less verbose
  • is integrated with Scala
  • gives static guarantees
What is Mnemonics?

A library which
• is less verbose
• is integrated with Scala
• gives static guarantees

Focus on
• Runtime generation
• Essential set of bytecodes
• Generation of implementations of scala.Function1
What is Mnemonics?

A library which
• is less verbose
• is integrated with Scala
• gives static guarantees

Focus on
• Runtime generation
• Essential set of bytecodes
• Generation of implementations of scala.Function1

Mnemonics: A dynamic Scala inline bytecode assembler with static guarantees
Example

Let’s say we want to optimize formatting Strings:

```java
val c1, c2: java.util.Calendar = //...
String.format("%tm/%<te/%<tY", c1) // eg. => "04/15/2010"
String.format("%tm/%<te/%<tY", c2) // eg. => "01/23/1973"
```
Example

Let’s say we want to optimize formatting Strings:

```scala
val c1, c2: java.util.Calendar = //...
String.format("%tm/%<te/%<tY", c1) // eg. => "04/15/2010"
String.format("%tm/%<te/%<tY", c2) // eg. => "01/23/1973"
```

Instead:

```scala
val formatter = newFormatter("%tm/%<te/%<tY") // generate bytecode
formatter(c1) // eg. => "04/15/2010"
formatter(c2) // eg. => "01/23/1973"
```
Manually writing the formatter

```scala
val formatter = (cal: Calendar) => {
  new StringBuilder
    .append(cal.get(Calendar.MONTH))
    .append("/")
    .append(cal.get(Calendar.DAY_OF_MONTH))
    .append("/")
    .append(cal.get(Calendar.YEAR))
    .toString
}
```
After compiling and javap

```java
java.lang.String apply(java.util.Calendar):

new scala/StringBuilder
dup
invokespecial scala/StringBuilder."<init>":()V
aloado:
iconst: 1
invokevirtual java/util/Calendar.get:(I)I
invokevirtual scala/StringBuilder.append:(I)Lscala/StringBuilder;

ldc: String /
invokevirtual scala/StringBuilder.append:(Ljava/lang/String;)Lscala/StringBuilder;
aloado:
iconst: 5
invokevirtual java/util/Calendar.get:(I)I
invokevirtual scala/StringBuilder.append:(I)Lscala/StringBuilder;

ldc: String /
invokevirtual scala/StringBuilder.append:(Ljava/lang/String;)Lscala/StringBuilder;
aloado:
iconst: 1
invokevirtual java/util/Calendar.get:(I)I
invokevirtual scala/StringBuilder.append:(I)Lscala/StringBuilder;

invokevirtual scala/StringBuilder.toString:()Ljava/lang/String;
areturn
```

```scala
val formatter = (cal: Calendar) => {
  (new StringBuilder)
    .append(cal.get(Calendar.MONTH))
    .append("/")
    .append(cal.get(Calendar.DAY_OF_MONTH))
    .append("/")
    .append(cal.get(Calendar.YEAR))
    .toString
}
```
Java virtual machine

Runtime data model of the machine: basically stack-based
  - Stack: only the top elements can be manipulated
  - Local variables: statically-known size, random-accessible

Instruction types:
  - Stack-manipulation: \texttt{dup}, \texttt{pop}, \texttt{bipush}, ...
  - Local-variables: \texttt{xload}, \texttt{xstore}, \texttt{iinc}, ...
  - Array access: \texttt{xaload}, \texttt{xastore}
  - Method calling:
    - \texttt{invokevirtual}, \texttt{invokestatic}, \texttt{invokespecial}
  - Objects: \texttt{getfield}, \texttt{getstatic}, \texttt{putfield}, \texttt{putstatic}
  - Control-flow: \texttt{jmp}, \texttt{ifne}, \texttt{xreturn}, ...
  - ...

Appending a calendar field to the string with bytecodes

    // Convention:
    //   - Local variable 1 is the Calendar object
    //   - Before this block, StringBuilder is on top of the stack
    //   - After this block, StringBuilder is on top of the stack

    ALOAD 1
    BIPUSH calField
    INVOKEVIRTUAL Calendar.get
    INVOKEVIRTUAL StringBuilder.append
Appending a calendar field to the string with bytecodes

// Convention:
// - Local variable 1 is the Calendar object
// - Before this block, StringBuilder is on top of the stack
// - After this block, StringBuilder is on top of the stack

// Stack contents

ALOAD 1
BIPUSH calField
INVOKEVIRTUAL Calendar.get
INVOKEVIRTUAL StringBuilder.append
def appendCalField(calField: Int, mv: MethodVisitor) {
    // Convention:
    // - Local variable 1 is the Calendar object
    // - Before this block, StringBuilder is on top of the stack
    // - After this block, StringBuilder is on top of the stack

    mv.visitVarInsn(ALOAD, 1)  // ALOAD 1
    mv.visitIntInsn(BIPUSH, calField)  // BIPUSH calField
    mv.visitMethodInsn(INVOKEVIRTUAL,  // INVOKEVIRTUAL Calendar.get
           "java/util/Calendar",
           "get", "(I)I")
    mv.visitMethodInsn(INVOKEVIRTUAL,  // INVOKEVIRTUAL StringBuilder.append
           "java/lang/StringBuilder",
           "append", "(I)Ljava/lang/StringBuilder;")
}
Appending a calendar field to the string using simple methods and passing methods as closures

```java
def appendCalField(calField: Int, mv: MethodVisitor) {
    // Convention:
    //   - Local variable 1 is the Calendar object
    //   - Before this block, StringBuilder is on top of the stack
    //   - After this block, StringBuilder is on top of the stack

    mv.aload(1)
    mv.bipush(calField)
    mv.method2((_: Calendar).get(_: Int))
    mv.method2((_: StringBuilder).append(_: StringBuilder))
}
```
Basic Idea

JVM Instructions are Stack Transformers

See [Jones1998] and [Yelland1999]
Appending a calendar field to the string
the functional way

```scala
def appendCalField
  (calField: Int)
  (start: F): F = {
    // Convention:
    // - Local variable 1 is the Calendar object
    // - Before this block, StringBuilder is on top of the stack
    // - After this block, StringBuilder is on top of the stack

    start
    aload(1)
    bipush(calField)
    method2((_: Calendar).get(_: Int))
    method2((_: StringBuilder).append(_: StringBuilder))
  }
```
Appending a calendar field to the string
by applying frame transformers to a frame in sequence

```scala
def appendCalField(calField: Int)(start: F): F = {
  // Convention:
  // - Local variable 1 is the Calendar object
  // - Before this block, StringBuilder is on top of the stack
  // - After this block, StringBuilder is on top of the stack
  start ~
aload(1) ~
bipush(calField) ~
method2((_: Calendar).get(_: Int)) ~
method2((_: StringBuilder).append(_: StringBuilder))
}

trait F {
  def ~(f: F => F): F
}
```
Appending a calendar field to the string
by applying frame transformers to a frame in sequence

```scala
def appendCalField(calField: Int)(start: F): F = {
  // Convention:
  // - Local variable 1 is the Calendar object
  // - Before this block, StringBuilder is on top of the stack
  // - After this block, StringBuilder is on top of the stack

  start ~
 aload(1) ~
bipush(calField) ~
  method2((_: Calendar).get(_: Int)) ~
  method2((_: StringBuilder).append(_: StringBuilder))
}

trait F {
  def ~(f: F => F): F
}

def aload(i: Int): F => F
def bipush: F => F
```
Appending a calendar field to the string
in a typed way

```scala
def appendCalField[R<:Stack](calField: Int)(start: F[R]**StringBuilder]): F[R]**StringBuilder] = {
  // Convention:
  // - Local variable 1 is the Calendar object
  // - Before this block, StringBuilder is on top of the stack
  // - After this block, StringBuilder is on top of the stack

  start ~
 aload(1) ~
bipush(calField) ~
  method2((_: Calendar).get(_: Int)) ~
  method2((_: StringBuilder).append(_: StringBuilder))
}

trait F[+ST<:Stack] {
}

def aload[R<:Stack, X<:AnyRef](i: Int): F[R] => F[R]**X]
def bipush[R<:Stack]: F[R] => F[R]**Int]
```
Appending a calendar field to the string
explicitly passing a local variable token

```scala
def appendCalField[R<:Stack](calField: Int)(cal: Local[Calendar])(start: F[R**StringBuilder]): F[R**StringBuilder] = {
  // Convention:
  //   - Local variable 1 is the Calendar object
  //   - Before this block, StringBuilder is on top of the stack
  //   - After this block, StringBuilder is on top of the stack

  start ~
  cal.load ~
  bipush(calField) ~
  method2((_: Calendar).get(_: Int)) ~
  method2((_: StringBuilder).append(_: StringBuilder))
}

trait F[+ST<:Stack] {
}

trait Local[T] {
  def load[R<:Stack]: F[R] => F[R**T]
}
```
def appendCalField[R<:Stack]
  (calField: Int)
  (cal: Local[Calendar])
  (start: F[R**StringBuilder]): F[R**StringBuilder] = {
    start ~
    cal.load ~
    bipush(calField) ~
    method2((_: Calendar).get(_: Int)) ~
    method2((_: StringBuilder).append(_: StringBuilder))
  }

trait F[+ST<:Stack] {
}
Appending a calendar field to the string
with more types inferred

```
def appendCalField[R<:Stack]
    (calField: Int)
    (cal: Local[Calendar])
    (start: F[R**StringBuilder]): F[R**StringBuilder] = {
        start ~
        cal.load ~
        bipush(calField) ~
        method2(_.get(_)) ~
        method2(_.append(_))
    }
```

```
trait F[+ST<:Stack] {
}
```
Appending a calendar field to the string with Mnemonics

```scala
def appendCalField[R<:Stack]
  (calField: Int)
  (cal: Local[Calendar])
  (start: F[StringBuilder]): F[StringBuilder] = {
  start ~
cal.load ~
bipush(calField) ~
method2(_.get(_)) ~
method2(_.append(_))
}

def appendConstant[R<:Stack]
  (str: String)
  : F[StringBuilder] => F[StringBuilder] = { f =>
f ~
ldc(str) ~
method2(_.append(_))
}
```
Almost the implementation

```scala
def generateFormatter: Calendar => String = 
  ASMCompiler.compiler(classOf[Calendar], classOf[String]) {
    (param: Local[Calendar]) =>
    (ret : Return[String]) =>
    (f : F[Nil]) =>

    f ~
    newInstance(classOf[StringBuilder]) ~
    appendCalField(param)(Calendar.MONTH) ~
    appendConstant("/") ~
    appendCalField(param)(Calendar.DAY_OF_MONTH) ~
    appendConstant("/") ~
    appendCalField(param)(Calendar.YEAR) ~
    method1(_.toString) ~
    ret.jmp
  }
```
Summary (1)

- Bytecode instructions are represented as Scala functions which change a stack value.
- Instructions are composed using the `~` operator.
- A block of code has a type which describes its effect on the stack as `F[X] => F[Y].`
- A block of code can ignore parts of the stack when defined as a function with a polymorphic type (here: `R<:Stack`) for the rest of the stack.
- Local variables are represented by tokens of type `Local[X].`
Summary (2)

- Entry-point `ASMCompiler.compiler` takes a block of code, generates bytecode, loads and instantiates the new class and returns a function value `X => Y`
- When called, `compile` supplies to the block of code:
  1. a token to access the parameter value, `Local[X]`
  2. a token to return a value, `Return[Y]`
  3. a frame representing the empty stack `F[Nil]`
- Instructions and code blocks are treated equally: Both are functions that transform a stack value
Mnemonics

- Goal: Typed generator ⇒ verifiable bytecode
- Values on the stack must always have the correct type depending on the instruction
- Need to model
  - Types on the stack at every point in the program
  - Effect of instructions on the stack values’ types
Instruction encoding

An instruction is defined as a function from one state of the stack to another.

```python
def iadd[R<:Stack]:
    F[R**Int**Int] => F[R**Int]
```
An instruction is defined as a function from one state of the stack to another.

```python
def iadd[R<:Stack]:
    F[R**Int**Int] => F[R**Int]
```
Instruction encoding

An instruction is defined as a function from one state of the stack to another.

```
def iadd[R<:Stack]:
    F[R**Int**Int] => F[R**Int]
```

Most instructions polymorphic over R<:Stack, the rest of the run-time stack ⇒ ensures that R is not touched
The stack type

trait Stack

trait Nil extends Stack
case class Cons[R<:Stack,T] extends Stack

Infix type alias
type [x<:Stack,y] = Cons[x,y]

Examples
• Nil
• Cons[Nil,String]
• Cons[Cons[Nil,String],Int]
• Nil**String**Int

The Frame state type

trait F[ST<:Stack] {
  def ~(f: F[ST] => X): X = f(this)
}
def ldc[R<=Stack](str: String)
  : F[R] => F[R**String]

def aload[R<=Stack, T]
  : F[R**Array[T]**Int] => F[R**T]

def checkcast[R<=Stack, T, U](cl: Class[U])
  : F[R**T] => F[R**U]
Method invocation

- Type of invocation instruction depends on method called
- How to achieve type-safety?
Method invocation

- Type of invocation instruction depends on method called
- How to achieve type-safety?

Two types of method invocation:

- Method statically-known before generation
  
  ```java
  method1((x: Integer) => x.intValue)
  ```

- Method dynamically-chosen when generating
  
  ```java
  val m: java.lang.reflect.Method = ...
  methodHandle(m, classOf[Int], classOf[Int])
  ```
Method invocation

- Type of invocation instruction depends on method called
- How to achieve type-safety?

Two types of method invocation:

- Method statically-known before generation
  ```scala
def method1((x: Integer) => x.intValue)
```

- Method dynamically-chosen when generating
  ```scala
  val m: java.lang.reflect.Method = ...
  valHandle(m, classOf[Integer], classOf[Integer])
  ```
Statically-known method invocation

\[
\text{method1}((x: \text{Integer}) \Rightarrow x\text{.intValue})
\]

- Method is known and available in the compile-time namespace of the \textit{generator}
- Use first-class function to reference method

\[
\text{def method1}[T, U] \\
\phantom{def }\quad (\text{code}: T \Rightarrow U) \\
\phantom{def }\quad : \text{Method1}[T, U]
\]
Statically-known method invocation

method1((x: Integer) => x.intValue)

- Method is known and available in the compile-time namespace of the `generator`
- Use first-class function to reference method
- Use `scala.reflect.Code` to lift AST to run-time
- Must be abstraction of a method invocation (checked at generation time)

```scala
```
Statically-known method invocation

method1((x: Integer) => x.intValue)

- Method is known and available in the compile-time namespace of the generator
- Use first-class function to reference method
- Use scala.reflect.Code to lift AST to run-time
- Must be abstraction of a method invocation (checked at generation time)

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

- Convert to frame transformer with implicits
Parameter and local variable access

Access to Parameters and local variables via typed storage capabilities that map transparently to JVM local variables
Parameter and local variable access

Access to Parameters and local variables via typed storage capabilities that map transparently to JVM local variables

```scala
trait Local[T] {
}
```

Usage

```scala
bipush(5) ~
withLocal(i => f => f ~
i.load ~ dup ~ i.add ~ i.store
// i = i + i
// etc
```

Access to Parameters and local variables via typed storage capabilities that map transparently to JVM local variables

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

Introduce variable scope

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

Access to Parameters and local variables via typed storage capabilities that map transparently to JVM local variables

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

Introduce variable scope

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

Usage

```scala
bipush(5) ~
withLocal(i => f => f ~
  i.load ~ dup ~ iadd ~ i.store /* i = i + i */
  // etc
)
Category 2 Types
Further uses of implicits

- Two categories of JVM types
  - Category 1 (one word): Int, Float, Ref, ...
  - Category 2 (two words): Double, Long
- Mostly transparent on the stack
- Some instructions require special treatment

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

implicit def cat1any:AnyRef=>Category1 = null
implicit def cat1int:Int=>Category1 = null
implicit def cat1float:Float=>Category1 = null
/* and so on, seven definitions in total */
```
Composite instructions

- Construct typed high-level instructions by composing primitives
- Example: foldArray

```java
def foldArray[R<:Stack, T, U]
  (array: Local[Array[T]])
  (func : Local[Int] => F[R**U**T] => F[R**U])
  : F[R**U] => F[R**U]
```
Conclusions

- Convenient, type-safe bytecode generation at run time
- Typed generator $\Rightarrow$ verifiable bytecode
- Many instructions directly available
- Type-safe patterns for method invocation, local variables, return instructions, structured control transfers, instance creation
- Vital Scala ingredients: type inference with bounded polymorphism, variance, implicit parameters, overloading, and reflection
Current state of codebase

Unfortunately...

- Original implementation for Scala 2.7.7
- Some improvements with Scala 2.8.0
- But: Unfortunately code doesn’t work with Scala 2.8.0.Beta1, only trunk
- Tests don’t work with Scala trunk
- ⇒ No stable version right now
Further information

Thanks for listening! Questions?

At http://github.com/jrudolph/bytecode you’ll find

- Code under BSD license
- The PEPM ’10 paper and the original thesis
- A tutorial (soon)
Mnemonics has only a minimal framework:

- No support for “full” class generation
- Only generation of subclasses of `scala.Function1`
- Support for several backends: `ASMCompiler`, `interpreter`
Mnemonics has only a minimal framework:

- No support for “full” class generation
- Only generation of subclasses of scala.Function1
- Support for several backends: ASMCompiler, interpreter

```scala
class ASMCompiler {
  def compile[T1<:AnyRef, Ret<:AnyRef]
    (param1Cl: Class[T1], returnCl: Class[Ret])
    (code: Local[T1] => Return[Ret] => F[Nil] => Nothing) :
    T1 => Ret = // implementation
}
```
Applicability

- DSLs describing operations or transformations, i.e. an operation representable by a function $T \Rightarrow U$
- Candidates
  - XPath
  - Regular expressions
  - Parser combinators
  - Functional reactive programming
Branching, jumping, and returning

Low-level:
- Unconditional jump: withTargetHere and target.jmp
- Conditional jump: ifne

High-level:
- Conditional: ifne2
- Looping: tailRecursive

Return via *typed return capability*:

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