Type-Safe Bytecode Generation in Scala

Johannes Rudolph Peter Thiemann

Universität Freiburg, Germany

KPS'09, Maria Taferl, Österreich, 12.10.2009

Introduction

- Generation of JVM bytecode
- . . . at run time
- . . . in a Scala program

Introduction

- Generation of JVM bytecode
- ... at run time
- ... in a Scala program
- Generator has Scala-type $\Rightarrow$

Introduction

- Generation of JVM bytecode
- ... at run time
- ... in a Scala program
- Generator has Scala-type $\Rightarrow$
- Verifier accepts generated bytecode

Introduction

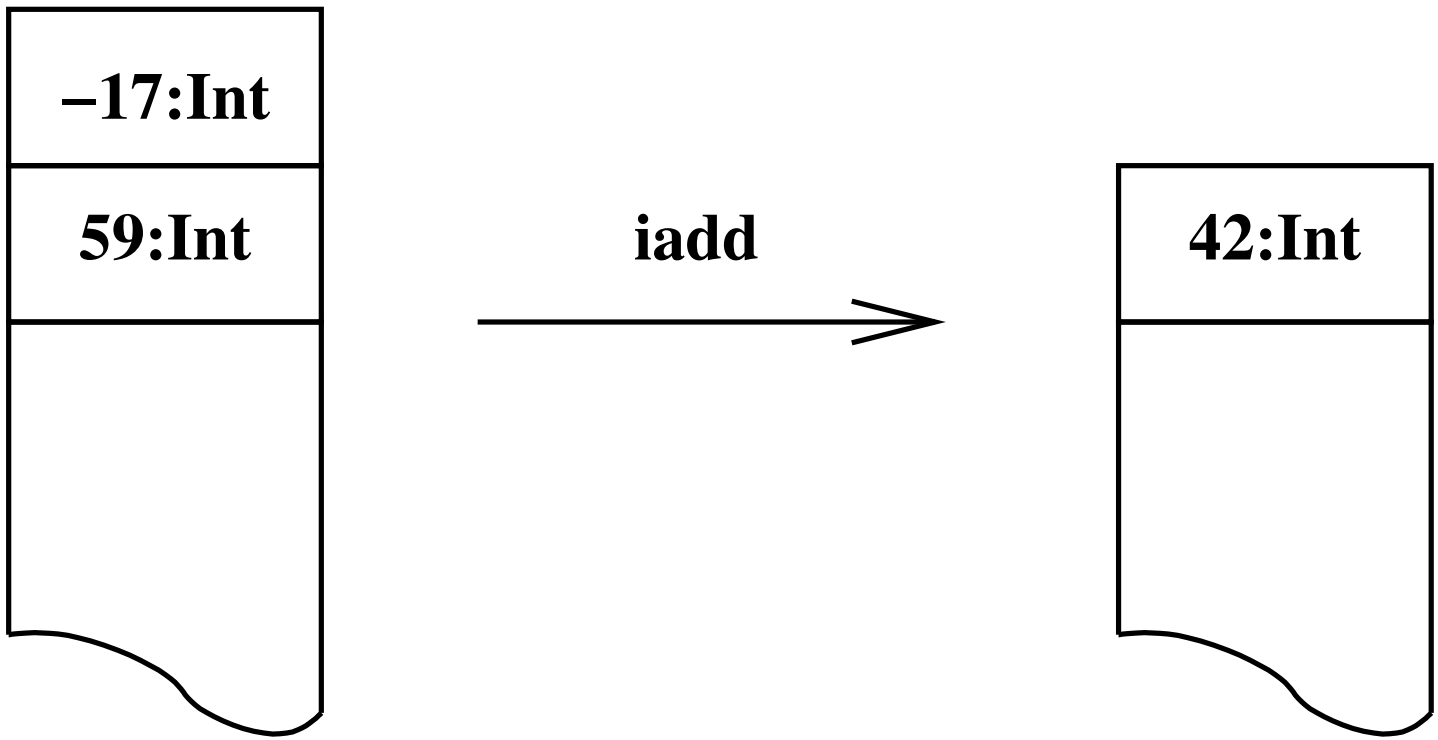
- Generation of JVM bytecode
- ... at run time
- ... in a Scala program
- Generator has Scala-type $\Rightarrow$
- Verifier accepts generated bytecode
- Implemented as a Scala library

Introduction

- Generation of JVM bytecode
- ... at run time
- ... in a Scala program
- Generator has Scala-type $\Rightarrow$
- Verifier accepts generated bytecode
- Implemented as a Scala library
- Goal: Convenience, not completeness

Basic Idea

JVM Instructions are Stack Transformers



See [Jones1998] and [Yelland1999]

Stack Transformers

Model Stack as a Heterogeneous List

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

Infix type synonym

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

Typing for example instruction

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

Stack Transformers

Model Stack as a Heterogeneous List

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

Infix type synonym

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

Typing for example instruction

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

- Most instructions polymorphic over `R<:List`, the rest of the run-time stack $\Rightarrow$ ensures that `R` is not touched

(Viz. work on STAL)

Stack Transformers

Model Stack as a Heterogeneous List

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

Infix type synonym

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

Typing for example instruction

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

- Most instructions polymorphic over `R<:List`, the rest of the run-time stack $\Rightarrow$ ensures that `R` is not touched

  (Viz. work on STAL)

- `Cons` is covariant in both parameters:
  `R**Byte**Short` is subtype of `R**Int**Int`
  $\Rightarrow$ correct handling of control transfers!

Example Generator

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

Example Generator

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

- Generator returns `(T1) => R`, that is, `Function1[T1,R]`

Example Generator

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

- Generator returns `(T1) => R`, that is, `Function1[T1,R]`
- `~` is reversed function application

Example Generator

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

- Generator returns `(T1) => R`, that is, `Function1[T1,R]`
- `~` is reversed function application
- Features
  - parameter access
  - return instruction
  - method call

Type of compile

```
def compile[T1<:AnyRef,R<:AnyRef]
   (param1Cl:Class[T1],returnCl:Class[R])
   (code: Local[T1] => Return[R] => F[Nil] => Nothing)
   : T1 => R
```

- Access to Parameters and local variables via *typed storage tokens* that map transparently to JVM local variables

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

- Return via *typed return tokens*

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

- `F[Nil]` stack frame with empty stack
- `Nothing` empty Scala type

Method Invocation

Method Handles

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

- *Typed method handle* specified with a function

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

  - For the argument function, Scala generates code that creates an AST at run time
  - Must be abstraction of a method invocation (checked at generation time)
  - Specific to unary invocations

- A Method handle can be invoked

```
trait Method1[-T,+U] extends MethodHandle {
  def invoke[R<:List]()
    : F[R**T] => F[R**U] = f => f.invokemethod(this)
}
```

Method invocation II

Implicits

- But how does the method handle become a frame transformer?

Method invocation II

Implicits

- But how does the method handle become a frame transformer?
- Implicits come to the rescue:

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

  - Scala wants to invoke method `Function1.apply` on `Method1` object
  - Scala looks for an implicit conversion `Method1[T,U] => Function1[T,U]`
  - Scala inserts the conversion automatically (if unambiguous)

Category 2 Types

Further uses of implicits

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

```
trait Category1
def swap[R<:List,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 cat1boolean:Boolean=>Category1 = null
/* and so on, seven definitions in total */
```

What's Missing

- Exceptions, synchronization
- Subroutine instructions `jsr` and `ret` (abolished from bytecode)
- Switch instructions
- Multi-dimensional array instructions

Conclusions

- Convenient, type-safe bytecode generation at run time
- Typed generator $\Rightarrow$ verifiable bytecode
- Many instructions directly available
- Type-safe patterns for method invokation, local variables, return instructions, structured control transfers, instance creation
- Vital Scala ingredients: type inference with bounded polymorphism, variance, implicit parameters, overloading, and reflection
- Try it:
  `http://virtual-void.net/files/mnemonics.zip`