5-1-2014

Design and Implementation of a Scala Compiler Backend Targeting the Low Level Virtual Machine

Geoffrey Reedy

Geoffrey Reedy

Candidate

Computer Science

Department

This thesis is approved, and it is acceptable in quality and form for publication:

Approved by the Thesis Committee:

Darko Stefanovic , Chairperson

Jed Crandall

Matthew Lakin

Design and Implementation of a Scala Compiler Backend Targeting the Low Level Virtual Machine

by

Geoffrey Reedy

B.S., Computer Science, University of Missouri — Rolla, 2004

THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Computer Science

The University of New Mexico

Albuquerque, New Mexico

May, 2014

Design and Implementation of a Scala Compiler Backend Targeting the Low Level Virtual Machine

by

Geoffrey Reedy

B.S., Computer Science, University of Missouri — Rolla, 2004

M.S., Computer Science, University of New Mexico, 2014

Abstract

The Scala programming language successfully blends object-oriented and functional programming. The current implementation of Scala is tied to the *Java Virtual Machine (JVM)* which constrains the implementation and deployment targets. This thesis describes the implementation of a new backend for the Scala compiler that targets the *Low Level Virtual Machine (LLVM)*. Targeting LLVM allows compilation of Scala programs to optimized native executables and enables implementation techniques that are not possible on the JVM. We discuss the design and implementation of this backend and evaluate its ability to compile existing Scala programs and the performance of the generated code. We then outline the additional work needed to produce a more complete, performant and robust backend.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis we present a new backend for the Scala compiler which targets the *Low Level Virtual Machine (LLVM)*. At its core LLVM provides a type-safe, universal intermediate representation designed to be targeted by high-level compilers and code generators that translate this intermediate language to native assembly or machine code. By targeting LLVM's intermediate representation the compiler author can support several architectures with a single code generation target.

The Scala compiler currently targets the *Java Virtual Machine (JVM)*. The JVM is a managed runtime for class based object oriented programming languages. It provides runtime features such as garbage collection, method dispatch, thread-based concurrency, and rich libraries. LLVM, true to its name, does not provide these features. Instead it supplies what a language implementer needs to build exactly those facilities that are needed by the particular language. Indeed, most of the complexity in the backend we have implemented is in the implementation and support of these facilities. In exchange, these facilities can support Scala and its features in ways that the JVM does not.

Because LLVM can generate standalone native code it does not require a virtual machine on deployment targets. This could lead to Scala being a viable option for platforms where a JVM is not available including embedded systems and Apple's iOS platform.

Scala programs compiled to LLVM would also not suffer from the long start up time required for the JVM, making Scala a more attractive choice for system programming and scripting. Scala on LLVM can also provide a platform for experimentation in implementation techniques for a hybrid functional/object oriented language.

This thesis begins with brief overviews of the Scala compiler and LLVM followed by a description of the LLVM backend and how Scala constructs map to LLVM. This includes details of code generation and runtime support facilities (e.g. garbage collection). We present results on the performance of the code generated by the LLVM backend. The performance numbers are disappointing but the analysis shows that more efficient implementation of runtime features could bring significant improvements. We end with a discussion of the missing features of the backend outlining how they can be completed and an outlook towards possible future extensions of the project.

Chapter 2

Background

2.1 The Scala Compiler

Scala's compiler is built on a powerful and flexible framework that allows the compiler to be assembled from separate components each implementing different aspects of compilation.This section gives an overview of the architecture of Scala's compiler as it relates to creating a new backend. A full treatment of this architecture can be found in Odersky and Zenger's 2005 paper "Scalable component abstractions" [15].

2.1.1 Components and Phases

Each component defines a phase of the compilation pipeline and declares its dependencies. The compiler uses the dependency information to assemble the pipeline as documented in Nielsen [13].

The initial phases parse each source file into a syntax tree, enter symbols into the compiler's symbol table, and annotate the trees and symbols with type information. Intermediate phases process these trees, symbols, and types to analyze, optimize, or desugar the program. The final phases translate the syntax tree into the ICode intermediate language, optmize the ICode, and generate code for the target platform.

2.1.2 Platforms

The target platform can be considered as a meta-component within the Scala compiler architecture. Platforms specify

- platform specific phases;
- loading symbol and type information from compiled code;
- the method for comparing two objects for equality; and
- whether a value of a given type might be a boxed primitive.

For example, the JVM platform specifies a symbol loader that reads from class files in the filesystem or JAR files and that the flatten and genJVM phases should be included in the compilation pipeline.

The main addition to the Scala compiler for this project is the additional platform component for LLVM and the code generation phase it instructs the compiler to use.

Unfortunately, platform specific knowledge is not confined to the platform abstraction. There are some phases within the compiler that check which target platform is active and change their behavior accordingly. Even though the number and scope of this embedded platform knowledge is limited, any new target platform must extend these areas of the compiler to support the additional platform.

2.1.3 ICode

ICode is an intermediate language used in the Scala compiler. The icode phase of the compiler generates the ICode representation of the program from the annotated syntax trees processed in earlier compiler stages.

ICode resembles JVM bytecode in many ways. Code is organized into methods, which are in turn collected within classes. Classes, fields, and methods are each identified by their symbol within the compiler's symbol table.

The execution model is a stack based machine with a separate call stack and operand stack; each method can also define a set of local variables to use as scratch space. To facilitate analysis, optimization, and code generation, a method's code is organized into basic blocks, sequences of instructions with a single entry point (the first instruction) and a single normal exit point (the last instruction). A basic block can be exited at any time due to a thrown exception.

Each method contains a (possibly empty) list of the exception handlers for the code in the method. An exception handler covers one or more basic blocks and handles a particular exception type (or any of its subtypes). The handlers are stored in order from innermost to outermost so that iteration over the handlers that cover a given block will yield the handlers in the order that they should be evaluated. When an exception handler matches the thrown exception, control transfers to the start block specified in the handler. When no exception handlers match, execution of the method stops and the exception propagates to its caller.

ICode instructions are annotated with enough type information that the types of values consumed from and pushed onto the operand stack can be determined by inspecting the instruction alone. This makes it possible to verify that optimizations and other ICode transformations preserve type-safety. However, in ICode type parameters have been erased so the types are approximations of the types in the source program, known within the compiler as a TypeKind.

As an example the Scala source and resulting ICode for a factorial function are shown in Listing 2.1 and Listing 2.2 respectively. A detailed description of ICode can be found in Section 3.2.1 of Dragos' PhD thesis [4].

Listing 2.1: Scala source for fact

```
def fact(n: Int): Int = {
  if (n == 0) 1 else n * fact(n−1)
}
```

Listing 2.2: ICode for fact

```
def fact(n: Int (INT)): Int {
  locals: value n
  startBlock: 1
  blocks: [1,2,3,4]

  1:
    LOAD_LOCAL(value n)
    CONSTANT(0)
    CJUMP (INT)EQ ? 2 : 3

  2:
    CONSTANT(1)
    JUMP 4

  3:
    LOAD_LOCAL(value n)
    THIS(fact)
    LOAD_LOCAL(value n)
    CONSTANT(1)
    CALL_PRIMITIVE(Arithmetic(SUB,INT))
    CALL_METHOD fact.fact (dynamic)
    CALL_PRIMITIVE(Arithmetic(MUL,INT))
    JUMP 4

  4:
    RETURN(INT)

}
```

2.2 Low Level Virtual Machine

The *Low Level Virtual Machine (LLVM)* is an open source compiler framework for optimization [7], code generation and lifelong program analysis through profile guided reoptimization [8]. It began as a research project at the University of Illinois and is now being used by a number of open source and commercial projects [11]. LLVM specifies a portable and universal intermediate representation and provides tools for analysis, optimization, and native code generation (both ahead-of-time and just-in-time). It is used as the target for a number of functional and object oriented languages including Haskell [16] and Java [2].

LLVM supports features such as efficient tail calls, precise garbage collection [10], zero-cost exception handling [6], link-time optimization, and atomic memory operations. These features make LLVM an attractive target for compilers because they get all of these features, analyses and optimizations for free, the quality and quantity of which are consistently increasing.

2.2.1 Intermediate Representation

There are three equivalent representations for LLVM assembly, called LLVM Intermediate Representation (LLVM IR), a textual syntax for human inspection and authoring, a space efficient binary format known as bitcode, and an in memory representation used by the LLVM tools and libraries. LLVM includes tools for processing code in the binary format and converting between the textual and binary forms; these tools are described in more detail below. The semantics of LLVM IR are defined in the *LLVM Language Reference Manual* [9] but we give a brief overview here.

An IR file in either binary or textual form is known as a module and is the translation unit in the LLVM system. A module contains global data declarations and definitions, type definitions, named module metadata, external function declarations and function definitions.

LLVM IR is a static single assignment (SSA) [3] language with an unlimited number of registers. Function bodies are given as a list of basic blocks with a distinguished entry block in which the function's execution begins. Each block may begin with a sequence of phi instructions that merge incoming values from the block's predecessors. This is followed by any number of normal instructions (e.g., arithmetic, conversions, memory operations, function calls). The last instruction in a basic block is known as the *terminator* and determines how execution proceeds. The basic terminators are the unconditional branch, conditional branch, generalized conditional branch (switch), and function return. Control flow due to exception handling is represented explicitly in LLVM by the invoke instruction. This instruction is a function call that specifies the successor block for a normal return and the successor block for an exceptional exit from the called function. The special terminator unreachable can be used to tell the LLVM optimization passes that the end of a block will not be reached, for instance, after calling a function that does not return.

LLVM IR is distinguished from most low-level intermediate languages by the presence and use of high-level type information. This type information and the data flow information implicit in the SSA form provide opportunities for much richer optimizations and analyses than are typically possible for a low-level representation.

Continuing the previous example, a factorial function written in the textual LLVM IR is shown in Listing 2.3. This demonstrates the explicit nature of control flow and typing in LLVM IR. Every instruction contains type information for its arguments, and when necessary the result type. In this case only two types are used: i32 and i1; 32 bit and 1 bit integers respectively. In reality a third type is used in this fragment, namely the type of @factorial: i32(i32)*. This type can be inferred from the return type and argument types used in the call instruction and does not need to be given explicitly.

Listing 2.3: LLVM code for factorial

```
define i32 @factorial(i32 %n) {
entry:
  %iszero = icmp eq i32 %n, 0
  br i1 %iszero, label %return1, label %recurse
return1:
  ret i32 1
recurse:
  %nminus1 = add i32 %n, −1
  %factnminusone = call i32 @factorial(i32 %nminus1)
  %factn = mul i32 %n, %factnminusone
  ret i32 %factn
}
```

2.2.2 Extending and Embedding LLVM

One of LLVM's greatest strengths is that, rather than being a monolithic system, it is designed as a set of components that can be extended with additional functionality and used outside of the tools distributed with LLVM. This enables a compiler to emit LLVM IR, possibly annotated with metadata specific to the language, and supply passes for language-specific optimizations and analyses. A language runtime can embed the just-in-time compiler and execution engine of LLVM and use the LLVM API to expose the runtime services to the hosted program. This project uses the JIT engine in this way, whereas language-specific optimizations are deferred to future work.

Chapter 3

The LLVM Backend

Compared with the JVM backend, the LLVM backend is more complex as it must include details about garbage collection, method invocation, exception handling, layout of objects in memory, class metadata, runtime type checks, and other services that are provided by the JVM. Additionally, these features cannot be designed in isolation; they must work in concert. For example, the garbage collector needs to trace the pointers contained within objects, so object layout and class metadata must be designed to support this. This chapter discusses the design and implementation of the LLVM backend along with the consequences and rationale of the decisions made. This chapter presents examples that demonstrate various aspects of the backend. Each example uses the Scala classes, traits and objects defined in Listing 3.1 as the basis for the example.

3.1 Design Goals

Throughout the development of the LLVM backend, decisions were guided by two high level goals: minimize the changes to the existing compiler code and start with simple solutions avoiding premature cleverness.

Listing 3.1: The class definitions used in further examples.

```
trait MetaSyntactic { def name: String }
trait Contains[T] { def children: Array[T] }
class Foo(val anInt: Int) extends Object with MetaSyntactic { val name = "foo"; }
class Bar(val children: Array[Foo]) extends Foo with Contains[Foo] {
  override val name = "bar"
}
object Baz extends Foo { override val name = "baz" }
```

Minimizing the changes to the compiler is important so that the backend can continue to work as the rest of the compiler independently evolves. Unfortunately some modifications to the compiler beyond enabling the selection of the backend were required. However, these changes have been rather small and by adhering to this goal we have successfully tracked Scala's development for roughly two years.

Choosing simple solutions first allowed development of a partially functional backend in less than a month. Though some early choices later proved inadequate, this principle enabled continued rapid development.

3.2 ICode vs. Abstract Syntax Trees

The Scala compiler uses two different representations for programs: abstract syntax trees and the ICode stack-based intermediate language. The built-in backends generate their code from ICode but it is possible for a backend to generate code directly from the AST. We chose to follow the other backends and generate code from ICode.

Aligning with the other backends allowed us to use them as a template for the LLVM backend. ICode also has a clear executable semantics which makes it easier to faithfully implement Scala's semantics instead of creating a language that is slightly different.

Targeting ICode may also reduce the modifications needed as Scala evolves since new language features will either be desugared to existing ICode or accommodated by extending ICode in some way. In the first case, no additional effort is required to support the new feature. If ICode is extended, inspecting how the other backends were affected can make clear what changes are required.

Generating code from ICode does have some disadvantages. ICode assumes certain characteristics of managed runtimes for object oriented languages, the JVM in particular. This makes certain optimizations and alternative compilation strategies difficult or impossible.

3.3 Run-time Value Representation

Scala's type system has value types (subtypes of AnyVal) and reference types (subtypes of AnyRef) and unifies them under the universal type Any. In ICode, however, values and references are stratified. Reference types can be further divided into classes and arrays.

3.3.1 Values

The ICode value types are represented by scalar LLVM and C types. Table 3.1 shows each ICode value type, the corresponding LLVM type used in the generated code and the C datatype used in the runtime support code. We assume that LLVM promotes i1 to fill a full byte and that sizeof(bool) == 1.

Table 3.1: ICode value types and their corresponding LLVM and C types

ICode Type	LLVM Type	C Type
unit	void	void
bool	i1	bool
byte	i8	int8_t
short	i16	int16_t
char	i16	uint16_t
int	i32	int32_t
long	i64	int64_t
float	float	float
double	double	double

3.3.2 References

Both instance-references and array-references are represented by a pair of pointers. The first pointer points to a method table for the reference and the second points to the instance data. The method table pointer is always appropriate for the static type of the reference, which may be a supertype or an interface type implemented by the runtime type of the referenced object. For simplicity, both pointers are given a generic type (i8* in LLVM; void* in C) for the ABI and must be cast to the appropriate type when used.

3.4 Name Mangling

In ICode, classes, fields, methods, etc. are identified by symbols organized in a containment hierarchy. During code generation we need to map these symbols into a flat namespace for LLVM in a way that will not produce any collisions. We employ a name mangling scheme that encodes the symbol path, entity kind, entity name, and any further information needed to uniquely name an entity (e.g., method argument and result types).

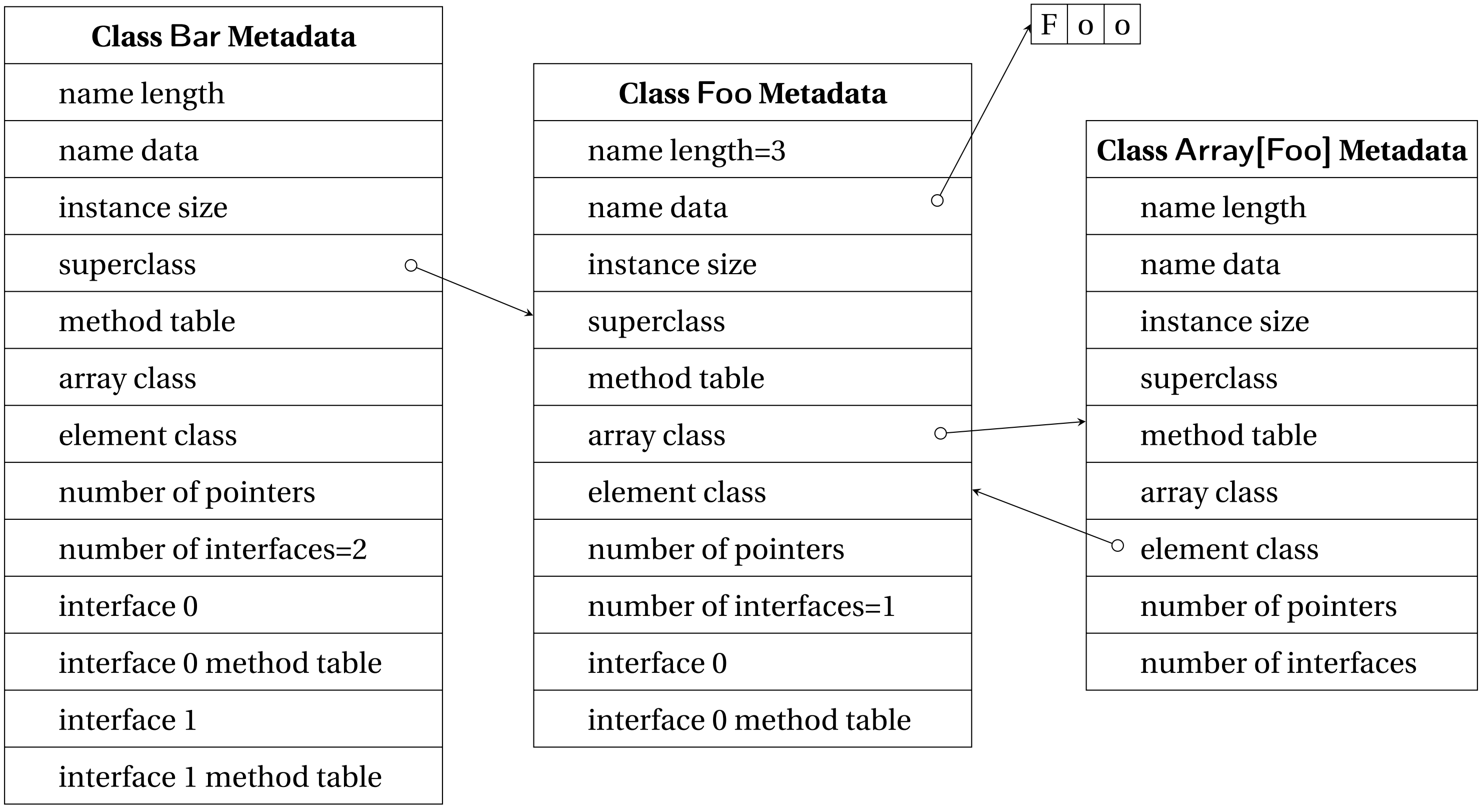


Figure 3.1: Class metadata for Bar, Foo, and Array[Foo] showing selected relationships.

3.5 Class Metadata

Information about classes needed at runtime for program execution is stored in a global metadata structure per class and associated auxiliary data. This information supports method invocation, runtime type tests, garbage collection, and debugging. The details of these structures are shown in Figure 3.1. The description of the array element and array class pointers is deferred until the discussion of the implementation of arrays. The remaining members are described below.

The superclass pointer is consulted at runtime for type tests and casts for class type targets. Likewise, the list of implemented interfaces is used for tests and casts to interface types.

The instance size member gives the total size of an instance including the superclass fields. For instances that are a subclass this gives the offset from the object's start to the subclass fields. In combination with the number of reference fields defined in the class, this information is used by the garbage collector to locate all of the object pointers within a given instance.

3.6 Instance Structures

The instance structure definition specifies the layout of instance data in memory. The structure begins with the superclass instance structure which is followed first by the reference fields defined in the class and second by the non-reference fields. An example illustrating the layout scheme is given in Figure 3.2.

The base object type is at the top of the class hierarchy and defines the fields needed to support the runtime system and basic object behavior such as garbage collection and runtime type checks. Currently, this is only a pointer to the class metadata for the

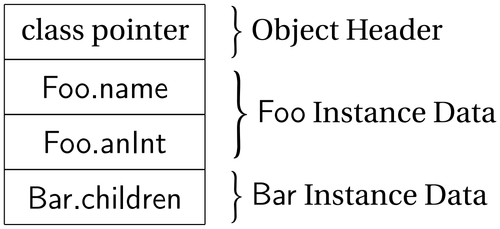


Figure 3.2: Instance structure layout of a Bar instance.

instance's dynamic type but as the system evolves more fields may become necessary. The instance layout scheme ensures that this information is at the beginning of the instance data so that it can be accessed without regard to the specific type of instance.

Including the superclass instance structure as the first member means that casts do not change pointer values, only their interpretation. The ordering of fields so that references come before non-references helps the garbage collector to find the pointers in an object.

3.7 Arrays

Array instances share the common object header followed by a length field and a variably sized block of memory that contains the data elements. The class metadata for arrays are dynamically created on demand and a pointer to the element class' metadata is stored in the array class' metadata. This is done because the element type of arrays on the JVM can be inspected at runtime. While this is not necessary to ensure type safety for cast-free programs, some Scala programs may depend on this detail so we duplicate the feature in our implementation.

3.8 Virtual Method Tables

The virtual method tables associate the methods of a class or interface with their implementations and are used during method invocation to select the proper implementation for the receiver. Virtual method tables are organized so that methods are grouped according to the class in which they are declared and methods within a group are canonically ordered. This organization of object instances and method tables allows allows an object reference to be used without conversion whenever a reference to one of its super-classes is required.

3.9 Code Generation

At a high level, the LLVM backend produces an LLVM module for each ICode class. In this section *class* refers to an ICode class unless otherwise specified. This module contains the structure definition for instances; class metadata and virtual method tables; global static data; implementation of the methods defined in the class; functions for obtaining the address of instance and static data fields; string constants; and, for classes that implement a singleton object (a module class in ICode parlance), a function that initializes the singleton instance. Declarations for functions and types from the runtime system and other classes are also included in the module. This section describes how each of these components are generated from the compiler's internal representations.

3.9.1 Method Implementations

Most of the work of the code generation phase is in translating method implementations to LLVM functions. Here we discuss the translation from method signature to function type; conversion from ICode's operand stack representation to LLVM's SSA register representation; and details of method invocation and exception handling.

In principle, LLVM supports structure types as both arguments and return types of functions but experience shows that using them can cause problems with LLVM's optimization and code generation passes (e.g. they might not be supported by all LLVM backends). This precludes the obvious mapping from method signatures to function types where the argument and results types are simply translated according to the rules in section 3.3 with the receiver (if any) passed as the first argument. Instead, this approach is modified by passing the virtual method table and object pointer of references as two separate arguments. To handle methods that return a reference value, the function accepts an extra pointer argument where the result's method table pointer will be stored; the object pointer is then returned.

Each function begins with a prologue that allocates stack space for the local variables defined in the method and other variables that are used during exception handling and method invocation. The prologue also communicates information about the function and its local variables by calling into the runtime. Each basic block of the method is translated to a basic block in the function by iterating over the ICode operations in the basic block and emitting LLVM instructions that implement the operation's semantics. An ICode operation may require only one or two LLVM instructions or perhaps tens of instructions. In some cases the operation might not produce any instructions but affect the state of the translation process.

ICode assumes several implicit value conversions that must be done explicitly when generating LLVM. In ICode the numeric types are all cross convertible and the target type is determined by the operation, so we must duplicate this behavior by emitting the LLVM instructions to do these type conversions. Another implicit conversion is from class-typed references to interface-typed references. To implement this in LLVM we must load the object's class metadata and find the method table for the target interface and create a new reference structure that holds a pointer to that method table.

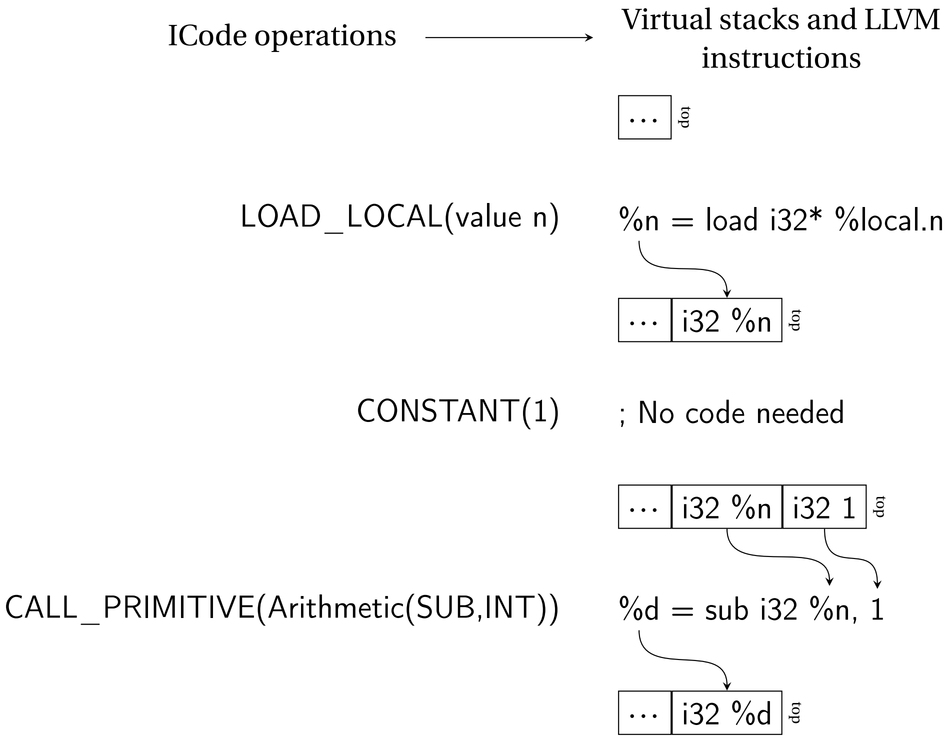


Figure 3.3: Visualization of the compile time operand stack showing how stack slots are mapped to and from LLVM values during compilation.

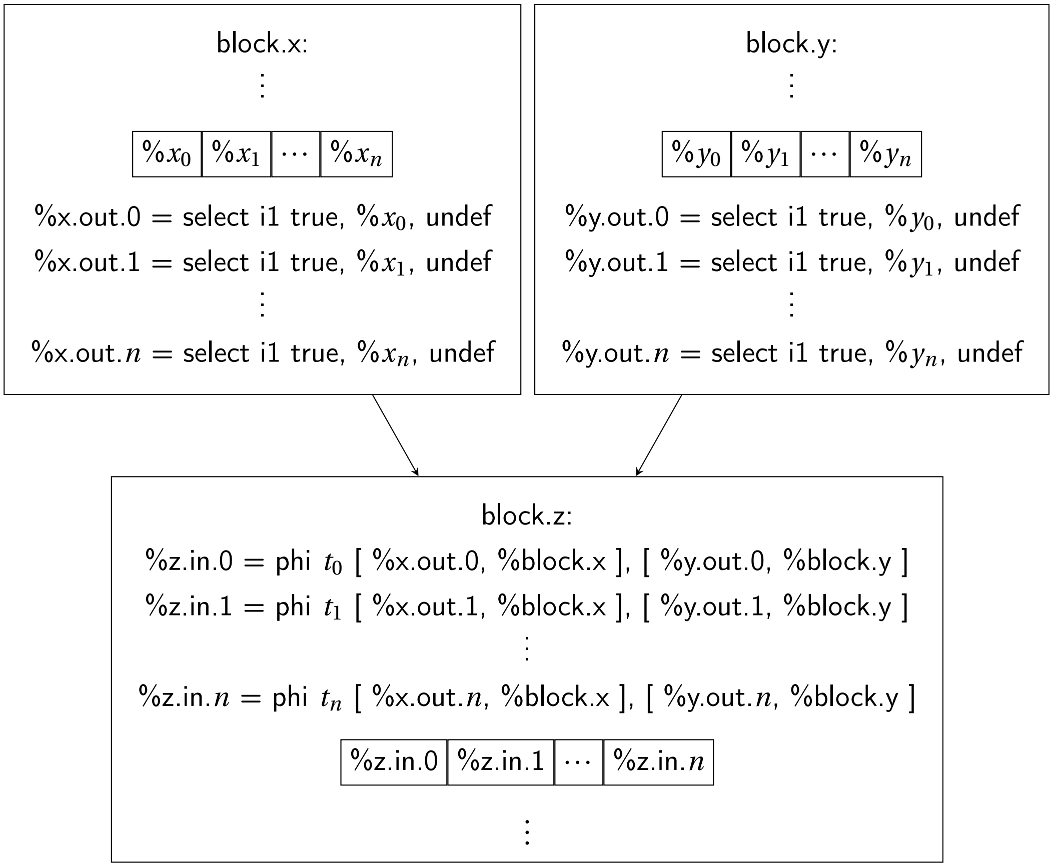
At first it seems that reconciling ICode's operand stacks and LLVM's SSA registers would be difficult, but it can actually be dealt with quite easily. The key observation is that the stack is a static single assignment structure since values on the stack are never modified: the act of pushing a value onto the stack is a static single assignment to a new stack slot. This leads to an implementation where the stack is manifest only during compilation and each stack slot holds the LLVM value that is the slot's value at runtime. When an ICode operation is translated values are popped from this stack and the result is pushed, as shown in Figure 3.3.

Conveying values between basic blocks is somewhat more complicated because LLVM requires explicit phi instructions to merge incoming values from predecessor basic blocks. The method used is similar to that described by Bebenita for converting

Java bytecodes to SSA form [1]. Each basic block is analyzed to determine how many values are required to be on the stack upon entering the block and how many values the block leaves on the stack upon exit. The outgoing stack slots of each block are given unique names based on the block identifier and the position of the slot on the stack. The use of specific names allows each basic block to be translated without any knowledge of the content of its predecessors, simplifying the compilation process. Before translating the contents of the basic block, ϕ instructions are inserted that map the outgoing stack slots of the predecessor blocks to the incoming slots of the block. Before completing translation of a basic block the values remaining on the stack are assigned to SSA registers with the appropriate distinguished names. Note that each block is responsible for selecting the stack slots it uses directly as well as those it merely passes along to its successors.

To a first approximation, method invocation is quite simple: load the appropriate function pointer from the receiver's method table and call the function with the arguments from the compile-time operand stack. However, any method may throw an exception during its execution. In order to handle exceptions the function must be called using the invoke instruction which is a terminator in LLVM. We first actually generate invalid LLVM IR by emitting the invoke in the middle of the basic block and with a non-existent destination. The code is fixed up after all basic blocks have been translated by splitting blocks at invoke instructions and rewriting the target of the invoke instruction to be the newly created successor.

This brings us to exception handling in the LLVM backend. An ICode basic block may be covered by any number of exception handlers. Exception handlers have a nested structure corresponding to the try-catch-finally expressions of the source program. Each handler handles a single exception type so a single case in a source level handler may result in several ICode handlers. We use the intrinsics that support zero-cost exception handling as documented in *Exception Handling in LLVM* [6]. An exception landing pad



block.x:

⋮

%x_0	%x_1	⋯	%x_n

%x.out.0 = select i1 true, %x_0, undef

%x.out.1 = select i1 true, %x_1, undef

⋮

%x.out.n = select i1 true, %x_n, undef

block.y:

⋮

%y_0	%y_1	⋯	%y_n

%y.out.0 = select i1 true, %y_0, undef

%y.out.1 = select i1 true, %y_1, undef

⋮

%y.out.n = select i1 true, %y_n, undef

block.z:

%z.in.0 = phi t_0 [%x.out.0, %block.x], [%y.out.0, %block.y]

%z.in.1 = phi t_1 [%x.out.1, %block.x], [%y.out.1, %block.y]

⋮

%z.in.n = phi t_n [%x.out.n, %block.x], [%y.out.n, %block.y]

%z.in.0	%z.in.1	⋯	%z.in.n

⋮

Figure 3.4: Example showing the transfer of stack values between basic blocks. Since LLVM does not have a copy operation, a select instruction with a constant condition is used to simulate it.

is emitted for each ICode basic block. The landing pad is a new basic block that calls the LLVM intrinsics to set up the exception handling machinery. All invoke instructions within the block have their exception target set to this landing pad. The landing pad extracts the exception instance from the exception information and passes control on to a set of basic blocks that check the type of the exception against the guard of each exception handler, transferring control to the first matching handler. The handlers are tested from innermost to outermost to preserve the programmer's intent. If no handlers match the exception's type the exception is rethrown.

The backend makes no special attempt to emit efficient LLVM code and in fact usually emits code with many redundancies and unnecessary register renaming. These inefficiencies are a consequence of the simplistic approach taken of transliterating ICode operations into LLVM instructions one at a time without regard for any surrounding instructions. We instead depend on LLVM optimization passes for efficiency, allowing the backend's code to remain simple and more easily maintained. However, the LLVM lacks understanding of Scala's semantics and misses some opportunities for optimizations. For example, multiple loads from a single method table may not be eliminated because the optimization passes do not realize that the method tables are immutable. Rather than change our code generation to deal with these issues we plan to annotate the code so that the default optimizations can take these opportunities or, if necessary, write our own optimization passes to address these inefficiencies. At this time we are focused on creating a functional backend; this and other optimzations are deferred as future work.

3.9.2 Native Methods

No implementation is emitted for methods marked with the `@native` annotation. Rather the method's implementation function is declared as an external functions and should be provided by code that is linked in to the final executable. This mechanism predates the foreign function interface described in section 3.11 and we expect that in most cases the FFI will be used in preference to native methods.

3.9.3 Singleton Initialization

As mentioned previously, singletons require special initialization routines as well as a global variable to hold the singleton instance. A module's initialization function is called whenever the module is accessed through a `LOAD_MODULE` operation, ensuring that the program receives a properly initialized instance. The initialization function is responsible for allocating the instance, calling its constructor, and registering the singleton as a GC root. It is not an error in Scala to have circular references among singletons so the initialization function must account for the fact that it may be invoked recursively.

3.10 Runtime

The runtime library for Scala on LLVM consists of implementations of a small portion of the Java API and support functions used by the generated LLVM code. These support functions cover dynamic creation of array classes, array bounds checks, boxing and unboxing between primitive values and objects, interfaces with the garbage collector, instance allocation and initialization, runtime type checks, interface virtual method table lookup, null pointer checks, string concatenation, and interfaces with the platform exception handling routines.

Most of the Java API classes in the runtime library are implemented purely in Scala. However the implementations of java.lang.Object, java.lang.Class and java.lang.String are written in C. The use of class names from Java is a consequence of deriving the LLVM platform component from the JVM platform component and will likely change in the future.

We intend to continue writing as much of the runtime in Scala as possible. We also plan to convert some of the existing C code to Scala. In particular, parts of the runtime library were written before the implementation of a foreign function interface and were implemented in C because of the need to interface with native libraries. The runtime should eventually implement all platform specific aspects of the standard library as specified in Chapter 12 of the Scala Language Specification [14].

The final piece necessary to run Scala programs with LLVM is the loader. The loader is written in C++ and uses the LLVM API. It performs the following steps:

1. Initializes LLVM
2. Loads the program's bitcode
3. Creates a JIT execution engine
4. Dynamically creates a function that
   (a) installs a top-level exception handler,
   (b) converts the argv C array to a Scala Array[String],
   (c) initializes the singleton containing the program's main method, and
   (d) invoke the main method with the argument array.
5. Calls the created function

We have also written an ahead-of-time compiler that emits the LLVM module along with a main function that can be compiled to native code. However the incomplete library implementation causes linking to fail for most programs due to missing symbols whereas the JIT mode can proceed as long as the undefined symbols are not accessed during execution.

3.11 Foreign Function Interface

The foreign function interface (FFI) enables access to native libraries from Scala programs. The FFI takes some inspiration from Haskell's FFI [12] and borrows some of its concepts.

3.11.1 Marshallable Types

The first borrowed concept is that of marshallable types. Marshallable types are those whose values can be transferred between Scala and foreign code. In our FFI, each of the value types and the special reference type scala.ffi.Ptr are marshallable. Table 3.2

Table 3.2: Marshallable types and their corresponding C type.

Scala Type	C Type	Argument	Result
scala.Unit	void	No	Yes
scala.Bool	bool	Yes	Yes
scala.Byte	int8_t	Yes	Yes
scala.Short	int16_t	Yes	Yes
scala.Char	uint16_t	Yes	Yes
scala.Int	int32_t	Yes	Yes
scala.Long	int64_t	Yes	Yes
scala.Float	float	Yes	Yes
scala.Double	double	Yes	Yes
scala.ffi.Ptr[T]	void *	Yes	Yes

Table 3.3: C types and the provided type alias.

C Type	Scala Type
char	scala.ffi.CChar
short	scala.ffi.CShort
int	scala.ffi.CInt
long	scala.ffi.CLong
long long	scala.ffi.CLongLong
intptr_t	scala.ffi.CIntPtr
intmax_t	scala.ffi.CIntMax
size_t	scala.ffi.CSizeT

shows these types along with the C type they are marshalled to and from. The table also indicates whether the type is valid as an argument or result type of a foreign function.

Scala's value types have a fixed size while the sizes of many basic C types are defined by the target platform. The build process for the LLVM backend probes the target platform and generates type aliases mapping basic C types to the appropriate Scala value type. The provided type aliases are listed in Table 3.3.

3.11.2 Storable Types and Pointers

Pointers are represented by scala.ffi.Ptr[T] which actually hold a scala.ffi.CIntPtr. When generating foreign calls, the compiler emits code that extracts the pointer value from the object from any pointer arguments and creates a new instance wrapping any pointer results.

For transferring values to and from raw memory the FFI defines the trait scala.ffi.Storable[T] which mirrors the Foreign.Storable type class of Haskell's FFI. Implementations of scala.ffi.Storable[T] must provide methods for storing and loading values given a scala.ffi.Ptr[T]. The FFI provides an implementation of scala.ffi.Storable[T] for each of the marshallable types.

The type parameter of scala.ffi.Ptr[T] is not used within the type (a so-called phantom type constructor) but instead provides a modicum of type safety for pointers and assists with locating the proper scala.ffi.Storable[T] instance using Scala's implicit resolution mechanism.

Valid pointers can be constructed by calls to the scala.ffi.alloc object. This object has methods to explicitly allocate and free memory as well as higher-order methods that allocate memory prior to executing their argument functions and free the memory when the argument function finishes execution, either through normal termination or by throwing an exception.

3.11.3 Importing Foreign Functions

A Scala program gains access to a foreign function by defining a method with the @foreign annotation. This annotation takes a string argument that gives the name of the foreign function. The function must have a single argument list and each argument type and the result type must be marshallable.

When a foreign method is encountered by the compiler the normal compilation of the method body is suppressed. Instead the compiler emits code that marshalls each of the method arguments, calls the foreign function with the marshalled values, unmarshalls the function's result and returns it.

3.11.4 Exporting Scala Methods to Foreign Code

Methods of singleton objects can be exported to foreign code by marking the method with a `@foreignExport` annotation. The annotation takes a string argument that specifies the name of the created foreign function. Like imported foreign functions, each argument type and the result type must be marshallable and the method must have a single argument list.

For a foreign exported method, the compiler emits a function with the specified name that marshalls the foreign arguments to Scala values, invokes the method, marshalls the result to a foreign value, and returns the marshalled result. Currently this feature is of limited usefulness because there is not any way to get a function pointer for the exported function.

3.12 Memory Management

Currently the runtime system for LLVM backend incorporates a simple mark-sweep collector. Each allocation request is serviced by the standard C allocation routines while tracking the total amount of memory allocated. The heap is given a fixed size and a collection is run when an allocation would exceed this limit.

Every allocation is extended to include a header and the returned pointer points just past the header. This header contains the information needed to run the garbage collection algorithm. This information includes the pointers used for heap walking and maintaining the marking worklist as well as the size of the allocation for heap size tracking.

3.12.1 GC Roots

Marking starts at roots. There are two types of roots: static and shadow-stack. Static roots are global objects that can be accessed at any point in a program and are always live. Shadow-stack roots are the objects stored in the local variables and in the virtual operand stack of executing methods. Static roots are stored in a global array. Stack roots are maintained in a shadow stack structure maintained during program execution. On method entry the current shadow-stack pointer is saved and method's local variables are registered on the shadow stack. The LLVM instructions emitted for each ICode operation pop any consumed references from the shadow stack and push any produced references onto the shadow stack. The shadow stack has a fixed size and the program aborts if it is exceeded. On method exit the shadow-stack pointer is restored to its saved value.

3.12.2 Marking

The marking phase builds a linked list of live objects, initially containing just the root objects. This linked list also serves as the worklist for the marking phase. During marking, objects to the left of the current work item have been scanned for pointers and objects to the right of the work item have not been scanned. The pointers within the current work item are scanned and any found objects that are not already in the work list are inserted after the current work item. When all pointers have been scanned the next item in the worklist becomes the current work item. This amounts to a depth-first traversal of the heap looking for objects reachable from the roots.

3.12.3 Sweeping

In the sweep phase of a collection cycle, the entire heap is traversed. Objects that are not marked (i.e. have a NULL worklist pointer) are freed. Marked objects have their worklist pointer cleared to prepare for the next collection cycle.

Chapter 4

Results

The LLVM backend and runtime is able to compile and execute some existing Scala programs and the fragments of Scala's standard library required for these programs. In particular we can run the *nbody* benchmark from the Programming Language Benchmark Game[1] suite and the *Scala Pro* version of the benchmark from Hundt [5]. Since we do not have implementations for formatted output routines (e.g., printf) we have modified these programs only to use unformatted output routines. We did have to modify the Scala library to avoid using unimplemented platform features. We also have a test program that is designed to exercise a wide variety of language features that also serves as a regression test for the backend. The main impediment to running a wider variety of programs is the incomplete runtime library.

[1] File `nbody.scala` as of 2013-06-01

4.1 Performance

The code generated by the LLVM backend runs significantly slower than the same program compiled for and run on the JVM.[2] As a first step to understand the performance disparity we disable the garbage collector since maintaining the shadow stack incurs significant overhead. The table below shows the execution times for a single run of each of the test programs. The time given is the sum of user and system time as reported by the time program. All measurements were taken on an otherwise idle system with an Intel Core i7-3612QM processor and 8GiB of memory and show the results for a single run of each program. We found that the running time of the programs was long enough that any variability in an individual run was negligible.

	Time (s)		
Program	LLVM JIT	LLVM JIT no GC	JVM
Test Program	6.15	4.25	0.51
nbody Benchmark ($n = 10^8$)	851.03	166.81	16.29
Havlak Benchmark	269.11	111.16	10.61

This shows that the garbage collector does indeed have a large overhead but there is still a wide performance gap. We then looked at two other runtime features with a performance cost: array bounds checking and null pointer checks. We added the capability to invidually disable these features much as was done with the garbage collector. We also considered the overhead of virtual method calls and modified the *nbody* benchmark to avoid virtual calls as much as possible. We also added, to both the original and modified versions of the program, the ability to repeat the benchmark a configurable number of times so that a linear regression could be used to factor out the fixed JIT startup times. We ran both versions of the benchmark (with $n = 10^6$) for all combinations of features

[2] java version "1.7.0_25"; OpenJDK 64-Bit Server VM (build 23.7-b01, mixed mode)

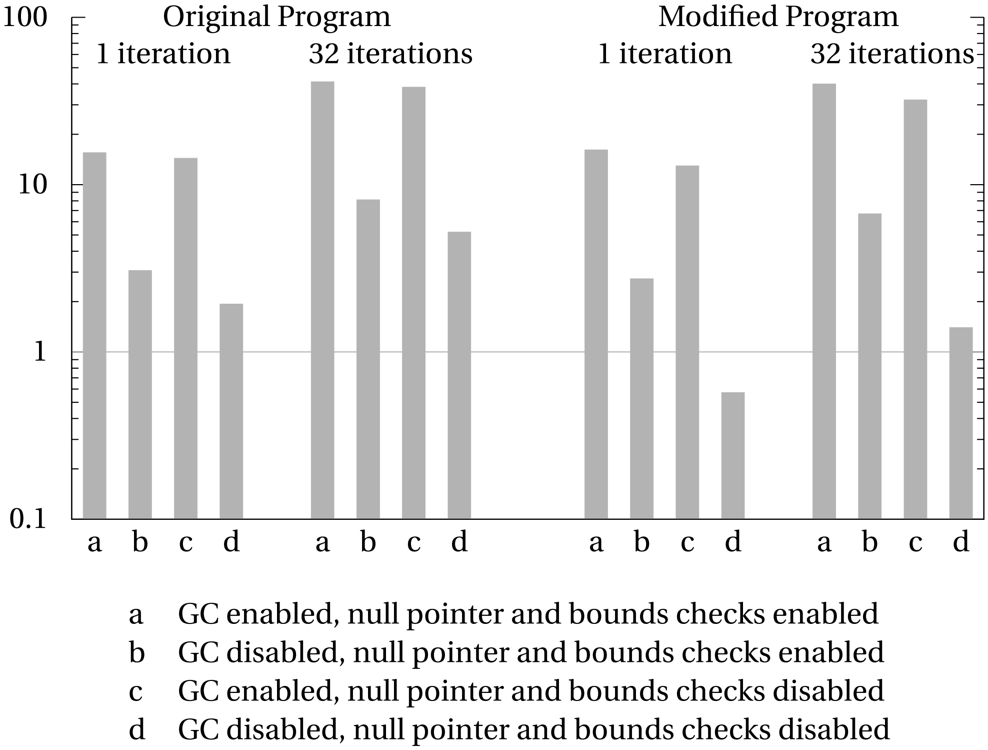


Figure 4.1: Performance of the *nbody* benchmark ($n = 10^6$) under various configurations of the LLVM backend normalized to the JVM backend.

and found that with all of the features disabled, the performance comes within a factor of two of the JVM for 32 iterations with both just-in-time and ahead-of-time compilation and actually beats the JVM for a single iteration with ahead-of-time compilation. This gives hope that, with some effort, the LLVM backend can be performance competitive with the JVM. Figure 4.1 shows a selection of these results for the ahead-of-time compiled runs.

These results indicate that more efficient implementations of these features can narrow the performance gap. Note that the JVM results for the original and modified programs are nearly identical, indicating that the HotSpot VM is able to inline the monomorphic virtual calls elimitating the virtual call overhead.

4.2 Missing Language and Runtime Features

Besides the incomplete library implementation, there are other features necessary for a complete and practical implementation of Scala that we do not provide and that will require compiler and runtime support.

Separate Compilation

We currently require that the entire source code of a program, including the source code of Scala's standard library, be submitted to the compiler. This greatly increases compilation time as the standard library code is quite large and complex. Separate compilation is not technically necessary to have a complete implementation of Scala, but it can be considered as a requirement for a practical backend.

Scala achieves separate compilation on the JVM platform by embedding pickled symbol information into the generated class files. We should be able to achieve separate compilation by generating the same pickled symbol information and outputting it within or alongside the LLVM modules.

Multithreading

The Scala standard library assumes the presence of multithreading support in the underlying platform for features such as futures and parallel collections. Since we cannot fully support Scala's standard library it cannot be considered a complete implementation.

Structural Types

The cleanup phase of the compiler replaces calls to structural types with code that uses Java's reflection facilities to locate the correct method to call at run-time. We do not override this aspect of the compiler and thus do not support structural types.

Rather than implement runtime reflection, we envision an alternative mechanism for supporting structural types. Structural types can be considered as post hoc interfaces. With this perspective, the appropriate interface method tables can be generated at link-time or run-time and the standard method lookup procedure can be used for structural types.

Chapter 5

Conclusions

This work represents an important first step towards an implementation of Scala that compiles to native code though there are still significant hurdles to overcome. Certainly, the performance problems must be addressed and more sophisticated allocation and garbage collection routines are needed. Compared to these, providing a more complete platform library is a minor concern with a straightforward resolution. If these issues can be addressed, Scala can become a viable choice for a wider variety of applications and environments.

Chapter 6

Future Work

6.1 Compiling From Trees

While generating code from the ICode representation was a prudent first step it may preclude certain compilation strategies and optimization opportunities in the future. ICode is very close to the semantics of the Java Virtual Machine and inherits many of its limitations. One of our objectives is to explore implementation techniques specifically suited for Scala and this may require generating LLVM code directly from the compiler's abstract syntax trees. Some of the ideas presented in this section might depend on this change in code generation.

6.2 Lightweight Function Objects

When compiling Scala to Java bytecode each anonymous function is implemented as a unique class because the JVM does not have first-class functions or closures. However with LLVM we could treat functions as a primitive type, much the same as is done with the JVM primitive types. Special subclasses of the Function class could be used for closures and eta expansion. The closure subclass could be represented in LLVM as a pair containing a pointer to the function and a pointer to the closure context. Likewise the eta expansion subclass could be represented as a pointer to a simple stub that resolves the method and calls it. This would likely require generating code from trees as discussed previously since ICode eliminates information needed to implement these special cases.

6.3 Elimination of Primitive Boxing

The Java platform requires that primitive values must be boxed to be used in an object context. This causes problems particularly for generic data structures. Using fields of object type within these data structures in conjunction with automatic boxing of primitives allows a single implementation to work with all types in the language but boxing can introduce significant overhead in memory usage and performance. Boxing can be eliminated by writing multiple implementations of a data structure that have fields of particular primitive types. The Scala compiler incorporates a specialization mechanism that can automatically generate these multiple implementations but this causes a combinatorial explosion in code size when multiple type parameters are specialized.

In the LLVM backend we represent references as a pair of pointers, the first pointing to the referent's method table and the second pointing to the referent's data. We could allow primitive values to be used as objects without incurring an allocation penalty by storing primitive values in the data pointer, as is done for small integers in OCaml and other systems. In OCaml, the low bit of a value is used to distinguish pointers from scalars which limits the range of unboxed integers. However, we can use the method table to determine the interpretation of the value and represent the full range of native integers unboxed. In principle this same optimization could be used for any class where the instance data fits within a pointer, such as a class representing an IPv4 address or, on platforms with 64-bit pointers, a single precision complex number.

6.4 Platform Abstraction of Scala Libraries

Scala's standard library is currently dependent on the Java API. It would be good for both the LLVM backend and the CIL backend to separate the parts of the library that are tied to the JVM from those that are not. For example, the JavaConversions class in the collection package would not have a direct analog on other platforms.

There are also parts of the library that use Java classes in their implementation for proper interaction with the underlying platform. Examples of this include I/O, exception classes, comparators, and mathematics. A potential strategy is to include the platform neutral portions directly in the class and mix in a trait from a scala.platform package. Each backend would then supply a different implementation of the traits in scala.platform.

6.5 Scala Specific LLVM Optimizations

LLVM provides facilities for writing language-specific optimization passes. These optimizations can exploit high-level information about the program communicated by the compiler by metadata attached to the LLVM instructions. There may be specific Scala idioms that would benefit from a targeted LLVM optimization. This could include whole program optimizations performed at link time. As an example, if a base class method is only invoked on instances of a certain subclass, loads of the receiver vtable could be replaced by a direct reference to the subclass vtable.

Bibliography

[1] Michael Bebenita. “Constructing SSA the Easy Way”. Jan. 25, 2011. URL: http://michael.bebenita.com/storage/ssa_class_notes.pdf (visited on 07/27/2013).

[2] Gary Benson. *Zero and Shark: a Zero-Assembly Port of OpenJDK.* May 21, 2009. URL: https://today.java.net/pub/a/today/2009/05/21/zero-and-shark-openjdk-port.html (visited on 07/27/2013).

[3] Ron Cytron et al. “Efficiently computing static single assignment form and the control dependence graph”. In: *ACM Trans. Program. Lang. Syst.* 13.4 (Oct. 1991), pp. 451–490. ISSN: 0164-0925. DOI: 10.1145/115372.115320.

[4] Iulian Dragos. “Compiling Scala for Performance”. PhD thesis. Lausanne: EPFL, 2010. DOI: 10.5075/epfl-thesis-4820.

[5] Robert Hundt. “Loop Recognition in C++/Java/Go/Scala”. Presented at the Second Scala Workshop, Stanford University, California. 2011.

[6] Jim Laskey. *Exception Handling in LLVM.* The LLVM Project. Apr. 6, 2011. URL: http://llvm.org/releases/2.9/docs/ExceptionHandling.html (visited on 07/27/2013).

[7] Chris Lattner. “LLVM: An Infrastructure for Multi-Stage Optimization”. MA thesis. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002.

[8] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization.* CGO 2004 (Palo Alto, California). Mar. 2004. DOI: `10.1109/CGO.2004.1281665`.

[9] Chris Lattner and Vikram Adve. *LLVM Language Reference Manual.* Version 2.9. The LLVM Project. Apr. 6, 2011. URL: `http://llvm.org/releases/2.9/docs/LangRef.html` (visited on 07/27/2013).

[10] Chris Lattner and Gordon Henrikson. *Accurate Garbage Collection with LLVM.* The LLVM Project. Apr. 6, 2011. URL: `http://llvm.org/releases/2.9/docs/GarbageCollection.html` (visited on 07/27/2013).

[11] *The LLVM Compiler Infrastructure: LLVM Users.* URL: `http://llvm.org/Users.html` (visited on 07/27/2013).

[12] Simon Marlow, ed. *Haskell 2010 Language Report.* July 2010.

[13] Anders Bach Nielsen. *Scala Compiler Phase and Plug-In Initialization for Scala 2.8.* Scala Improvement Document 2. May 28, 2009.

[14] Martin Odersky. *The Scala Language Specification, Version 2.9.* Programming Methods Laboratory, EPFL. Switzerland, 2011-05-24.

[15] Martin Odersky and Matthias Zenger. “Scalable component abstractions”. In: *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* OOPSLA ’05 (San Diego, CA, USA). New York, NY, USA: ACM, 2005, pp. 41–57. ISBN: 1-59593-031-0. DOI: `10.1145/1094811.1094815`.

[16] David A. Terei and Manuel M.T. Chakravarty. “An LLVM backend for GHC”. In: *Proceedings of the third ACM Haskell symposium on Haskell.* Haskell ’10 (Baltimore, Maryland, USA). New York, NY, USA: ACM, 2010, pp. 109–120. ISBN: 978-1-4503-0252-4. DOI: `10.1145/1863523.1863538`.