Compiling Scala to LLVM

Geoff Reedy
University of New Mexico

Scala Days 2011
Why Scala on LLVM?

- Compiles to native code
- Fast startup
- Efficient implementations
- Leverage LLVM optimizations/analyses
- Language implementation research
- Scala as a multi-platform language
Why Scala on LLVM? – Native code

Deploy Scala where a JVM is . . .
  - not available
  - not desired
  - old and slow

For example . . .
  - Apple iOS
  - Google Native Client
Why Scala on LLVM? – Fast startup

JVM startup dominates running time of short programs
  → Scala+JVM is not so great for scripting and utilities

LLVM start up is really fast
  → Small utilities spend most time doing useful work
Why Scala on LLVM? – Efficient implementation

LLVM allows more efficient implementations of
- traits
- anonymous functions
- structural types
Why Scala on LLVM? – The rest

Language implementation research
Scala+LLVM can be a place for innovation in language implementation issues

Multi-platform language
Scala already lets the programmer choose the right paradigm
Let them pick the right platform too
## What is LLVM?

LLVM is...

- an abbreviation of Low Level Virtual Machine
- a universal assembly language
- a framework for program optimization and analysis
- an ahead of time compiler
- a just in time compiler
- a way to get fast native code without writing your own code generation
LLVM’s intermediate representation is essentially a typed assembly language with
- primitive and aggregate types
- unlimited SSA registers
- basic blocks
- tail calls
- instruction and module level metadata
Figure: Factorial Function

```c
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
}
```
# LLVM analysis and optimization

LLVM is more than just an assembler

## Analyses
- Alias Analysis
- Liveness Analysis
- Memory Dependence Analysis
- Def-Use Analysis
- and more...

## Optimizations
- Constant Propagation
- Loop Unrolling
- Function Inlining
- Dead Code Elimination
- Peephole Optimizations
- Partial Specialization
- Link-time Optimization
- and more...
LLVM is great for compiler hackers

LLVM lets you
- spit out LLVM IR
- write high-level language-specific optimizations
- leave the low-level details to the LLVM infrastructure

You get to focus on your language and make the rest of it someone else's problem.
About LLVM

LLVM is great for compiler hackers

LLVM lets you

- spit out LLVM IR
- write high-level language-specific optimizations
- leave the low-level details to the LLVM infrastructure

You get to focus on your language and make the rest of it someone else's problem
LLVM is great for compiler hackers

LLVM lets you
- spit out LLVM IR
- write high-level language-specific optimizations
- leave the low-level details to the LLVM infrastructure

You get to focus on your language and make the rest of it someone else’s problem
LLVM is great for compiler hackers

LLVM lets you
- spit out LLVM IR
- write high-level language-specific optimizations
- leave the low-level details to the LLVM infrastructure

You get to focus on your language and make the rest of it someone else’s problem
The Scala compiler is organized as a pipeline of phases.

1. Source code is parsed into syntax trees
2. Syntax trees are typed, transformed, lifted, lowered, desugared
3. ICode is generated from the syntax trees
4. LLVM is generated from ICode
The Scala compiler is organized as a pipeline of phases.

1. Source code is parsed into syntax trees
2. Syntax trees are typed, transformed, lifted, lowered, desugared
3. ICode is generated from the syntax trees
4. LLVM is generated from ICode
The Scala compiler is organized as a pipeline of phases:

1. Source code is parsed into syntax trees
2. Syntax trees are typed, transformed, lifted, lowered, desugared
3. ICode is generated from the syntax trees
4. LLVM is generated from ICode
The Scala compiler is organized as a pipeline of phases.

1. Source code is parsed into syntax trees
2. Syntax trees are typed, transformed, lifted, lowered, desugared
3. ICode is generated from the syntax trees
4. LLVM is generated from ICode
The Scala compiler is organized as a pipeline of phases.

1. Source code is parsed into syntax trees
2. Syntax trees are typed, transformed, lifted, lowered, desugared
3. ICode is generated from the syntax trees
4. LLVM is generated from ICode
ICode

ICode is the compiler’s internal intermediate representation

Like LLVM IR, it...

- is typed
- has basic blocks

Unlike LLVM IR, it is

stack based

Basically mirrors JVM bytecodes

def fact(n: Int): Int = {
    if (n == 0) 1 else n * fact(n-1)
}
ICode

ICode is the compiler’s internal intermediate representation

Like LLVM IR, it...
- is typed
- has basic blocks

Unlike LLVM IR, it is stack based

Basically mirrors JVM bytecodes
Translating ICode to LLVM

What’s the simplest thing that could work?
Translate one instruction at a time.

Problem
Because it’s a local process creates redundant, slow code.

Solution
Let LLVM optimization passes clean it up for us.
What’s the simplest thing that could work?
Translate one instruction at a time.

Problem
Because it’s a local process creates redundant, slow code

Solution
Let LLVM optimization passes clean it up for us
From ICode to LLVM

Translating ICode to LLVM

What’s the simplest thing that could work?
Translate one instruction at a time.

Problem
Because it’s a local process creates redundant, slow code

Solution
Let LLVM optimization passes clean it up for us
Translating ICode to LLVM

What’s the simplest thing that could work?
Translate one instruction at a time.

Problem
Because it’s a local process creates redundant, slow code

Solution
Let LLVM optimization passes clean it up for us
Stacks to SSA

**Problem**
ICode is stack based; LLVM IR is register based

**Solution**
Maintain a mapping from stack slots to LLVM values during translation
Stacks to SSA

ICode fragment:

\texttt{CONSTANT(1)}

\texttt{CALL\_PRIMITIVE(Arithmetic(SUB,INT))}

Stack map:

\begin{tabular}{|c|c|}
\hline
\texttt{i32} & \%n \\
\hline
\end{tabular}
Stacks to SSA

ICode fragment:

```plaintext
CONSTANT(1)
CALL_PRIMITIVE(Arithmetic(SUB, INT))
```

Stack map:

```
i32 %n  ⋮
```
From ICode to LLVM

Stacks to SSA

ICode fragment:

CONSTANT(1)

CALL_PRIMITIVE(Arithmetic(SUB, INT))

Stack map:

i32 1  i32 %n  ⋮
Stacks to SSA

ICode fragment:

CONSTANT(1)

CALL_PRIMITIVE(Arithmetic(SUB, INT))

Stack map:

\[
\begin{align*}
\text{i32 1} & \quad \text{i32 %n} & \cdots \\
\text{\%d = sub i32 \%n, 1}
\end{align*}
\]
From ICode to LLVM

Stacks to SSA

ICode fragment:

```plaintext
CONSTANT(1)
CALL_PRIMITIVE(Arithmetic(SUB, INT))
```

Stack map:

```
i32  %d  ...
```

```
%d = sub i32 %n, 1
```
Classes in LLVM

For now, we use a simple representation:

- Class types are represented by structures in LLVM.
- The first member is the super-class structure.
- Object references are simple pointers to these structures.
- The base object structure has a pointer to the class’ info as its only member.
- Class info contains virtual method tables and other important info.
### Traits

We use fat interface references: a structure containing
- an object pointer
- a vtable pointer

**Advantages:**
- Calling through interfaces is fast
- Facilitates anonymous interfaces for structure types
Method dispatch

Method dispatch is pretty simple

**Static method**  Call function directly

**Class instance method**  Lookup class vtable
Call method through vtable

**Interface method**  Call method through interface reference’s vtable
Exceptions

It’s Complicated

but it works

Ask me later if you really want to know
Exceptions

It's Complicated

but it works

Ask me later if you really want to know
Exceptions

It’s Complicated

but it works

Ask me later if you really want to know
Runtime library

Problem
We don’t have Java’s standard library as a base

Solution
Write our own

Problem
It’s a big effort.
We have some basic things implemented.
It’s a mix of C and Scala (with some @native methods).
The runtime

Loader and launcher

After compilation you get LLVM IR
Then you assemble it to LLVM bitcode
The loader runscala will

1. initialize LLVM
2. load the program’s bitcode
3. synthesize a function that
   1. installs a top-level exception handler
   2. converts argv to a Scala array
   3. invokes main
4. starts the JIT and calls the function

Ahead-of-time compilation: write bitcode and generate native executable
What works

We can compile and run a simple program that includes:
- traits; abstract classes; objects
- exceptions
- arrays
- overriding and overloading
- integer and floating point computation
What doesn’t

We don’t yet have

- separate compilation
- garbage collection
- reflection
- threads
- a complete runtime library
Future goals

Lightweight functions

- LLVM has function pointers
- We don’t need to build objects just to get something callable
- Could anonymous functions be treated as the primitives?
Foreign function interface

- We should be able to use native platform libraries!
- How about a declarative, annotation driven FFI?
- Replace @native methods with the FFI
Scala specific optimizations

- LLVM can be extended with new analyses and optimizations
- Link time devirtualization!
Platform abstraction of Scala libraries

- Much of Scala’s library is tied to the JVM
- Modularize the library
- Separate generic and implementation specific code
- Mixin platform traits
Questions?

For more information

- http://greedy.github.com/scala/
- greedy@cs.unm.edu