Scala on LLVM: Help Wanted

Geoff Reedy

University of New Mexico

Scalathon 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

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

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

We also have a foreign function interface.

What doesn't

We don't yet have

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

Getting Started

The code is on github: `http://github.com/greedy/scala`
Check out the llvm branch
Dependencies

- icu
- llvm & clang $\geq$ 2.8

Subscribe to Google group:
`http://groups.google.com/scala-llvm`

How to help

You don't have to be a compiler hacker to help

LLVM Bitcode Reader/Writer

Right now the backend generates text-based IR

It would be nice to write LLVM bitcode directly

A reader would be nice too

ar Archive Reader/Writer

A ar archive (.a) is the natural equivalent to a JAR

The compiler should be able to consume them and possibly write them

Wrapping Libraries

Use the FFI to wrap native libraries

Need a basis for a portable runtime

For example: apr icu

Feel free to wrap your favorite library too

No compiler hacking needed

Wrapping Libraries

You'll be writing code that looks like this

```
@foreign("apr_pool_create_ex")
def _apr_pool_create_ex(newpool:Ptr[Ptr[pool_t]],
                        parent:Ptr[pool_t],
                        abortfn:abortfunc_t,
                        allocator:Ptr[allocator_t])
                : status_t = error("foreign")

lazy val rootpool = alloc.alloca { pptr: Ptr[Ptr[pool_t]] =>
  val stat = _apr_pool_create_ex(pptr,
                                 Ptr.nullPtr,
                                 Ptr.nullPtr,
                                 Ptr.nullPtr)
  if (stat != 0) throw new APRException(stat.toInt)
  else pptr.peek()
}
```

Write/find programs

Even though library coverage is low there are still some programs that should run

Bring us programs you'd want to be able to run on LLVM

Try to get shootout programs to run

Help guide runtime library implementation

How to help

But if you *are* (or want to be) a compiler hacker...

Scala Signature Reading/Writing

This is important for separate compilation

Should be relatively straightforward

Add a step to the LLVM phase that writes the signature for each class to a file

Add an LLVM classpath implementation that reads from these files

This is an in-compiler task but not deep hacking

Value types

It'd be nice to have

- Unsigned integers
- Pointers

as value types

Serious compiler hacking

Modularize Scala Library

The Scala library is large

I don't want to have to get it all working on LLVM

Split it up into isolated parts

Will help everyone not just the LLVM backend

See post by Paul Phillips on scala-internals

Optimiziations

Extend the compiler and/or LLVM to do

- Escape analysis and stack allocation
- Devirtualization
- Partial specialization

Garbage Collection

Right now there is none
There are two parts to this

1. LLVM plugin to generate stack maps and lower read/write barriers
2. The allocator and collector

Would be awesome to implement the collector in a non-allocating subset of Scala

Efficient Structural Types

A structural type is essentially an anonymous post-hoc interface

LLVM gives freedom and control to implement it this way

Thanks

Questions?

For more information

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

LLVM IR Sample

Figure: Factorial Function

```
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
}
```

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

```
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)
}
```