ORACLE®

ProjectFortress: Overview and Implementation Status

Christine H. Flood
Sun Labs, Oracle

Project Fortress Overview

Project Fortress is a programming language with the following tenets:

- Growable (small fixed core, rich libraries)
- Mathematical Notation
- Implicit Parallelism
- Strong Static Typing

Project Fortress Timeline

- Started in 2004 as part of the DARPA HPCS program
- Open source code base since January 2007
- Complete interpreter implementation of the Fortress 1.0 specification as of April 2008 running on top of the JVM.
- Since then we've been working on a compiler which generates Java Bytecodes.

Motivating Example: BirdCount

(Broad Institute Perl Code for Identifying Chicken Mutations)

- Perl Code
  - Sequential
  - 776 lines
- Fortress Code
  - Parallel
  - 177 lines

This is our first big compiler challenge.

Motivating Example from BirdCount

- This is a small simple piece of the birdcount code used as an illustrative example.
- DNA is represented in two ways:
    - The Reference Chicken DNA is fully mapped in ATCG format. The letters ATCG each stand for a different molecule called a base:
        - Adenine (A)
        - Thymine (T)
        - Cytosine (C)
        - Guanine (G)
    - The DNA samples from the sample chickens are produced in color format. Machines use chemical reactions to produce colors which represent pairs of molecules.

Transition to Color Space

	A	C	G	T
A	0	1	2	3
C	1	0	3	2
G	2	3	0	1
T	3	2	1	0

A	A	G	C	T	C	G	A	C	T
	0	2	3	2	2	3	2	1	2

How would we write this in Java?

- Sequential Iteration: Walk the list converting each character and its neighbor into color format.
- Parallel with 8 threads:
  - Partition the string into 8 chunks.
  - Over Partition the string into 32 chunks.
  - Recursive subdivision of the string with dynamic load balancing (JSR166y).

How would we write this in Fortress?

- $$\big\Vert_{p \leftarrow \mathrm{Pairs}(input)} \big(convert(p)\big)$$
  - $p \leftarrow \mathrm{Pairs}(input)$ is a parallel generator.
  - $\Vert$ is concatenate, a parallel reducer.

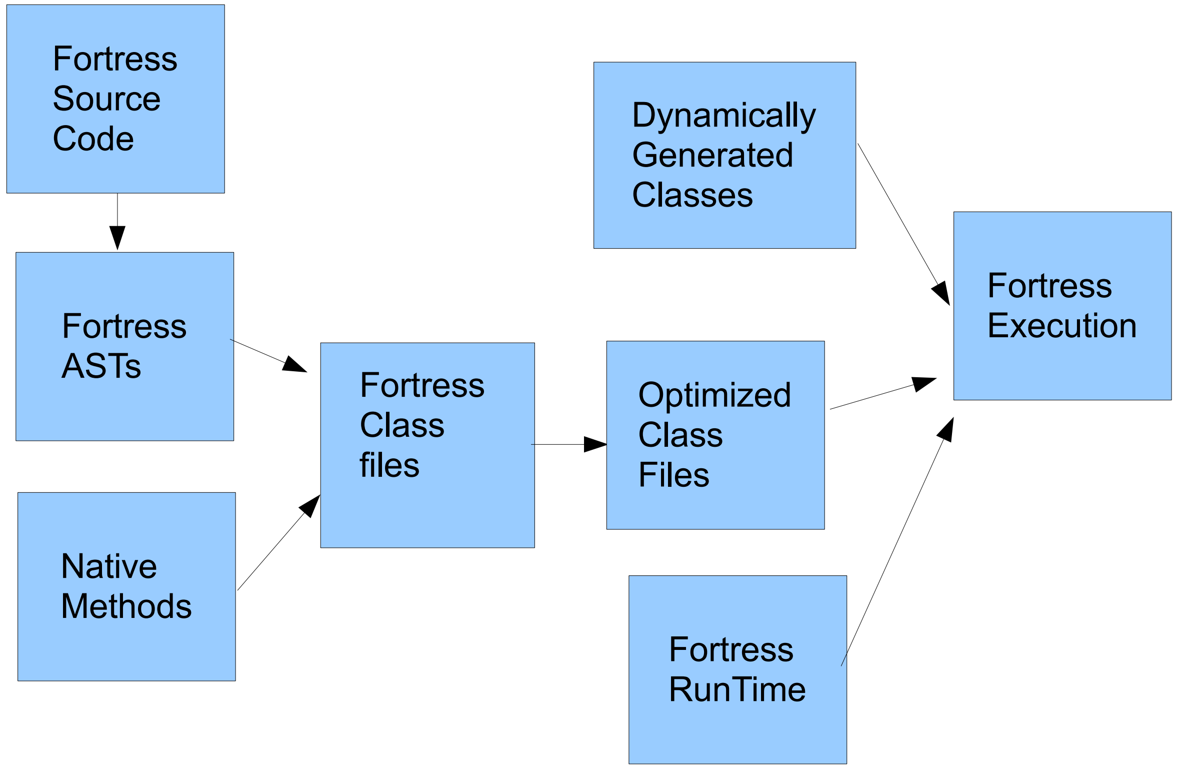
Project Fortress Compiler Status Outline

- Implementation Status

Parser	Done, shared with interpreter.
Type Checker	In Progress
Native Interface	In Progress
Code Generation	In Progress
Byte Code Optimization	In Progress
Runtime	In Progress

- This talk will focus on the items in blue (Native Interface, Code Generation, ByteCode Optimization and Runtime).

Fortress Compilation Process



Native Interface

- Our first compiler target was Hello World
- We needed a way to access System.out.println from Fortress code in a way that the typechecker would understand.

Native Interface

- Fortress side

```
component Compiled0
import java com.sun.fortress.nativeHelpers.{simplePrintln.nativePrintln ⇒ jPrintln}
export Executable
run(): () = jPrintln("Hello, World!")

end
```

- Java side

```
package com.sun.fortress.nativeHelpers
public class simplePrintln {
    public static void nativePrintln(String s) {
        System.out.println(s);
   }
    public static void nativePrintln(int i) {
        System.out.println(String.valueOf(i));
   }
}
```

Native Interface: Transformer

- Generated Code injected into class simplePrintln

```
...
public static FVoid nativePrintln(FString s) {
      simplePrintln.nativePrintln(s.getValue());
      return FVoid.make();
}
...
```

- This injection works great as long as we aren't modifying classes on the bootclasspath. Then we are required to generate sibling classes for wrapping/unwrapping.
- We hope to get back this performance cost by aggressive inlining with the BytecodeOptimizer.

Traits, Objects, and Functional Methods

```
trait List comprises {Cons, Empty}
    cons(x: Object) = Cons(x, self)
end
object Cons(first: Object, rest: List)
    extends List
end
object Empty extends List
end

trait BigNum extends Number
...
end
opr −(Number, BigNum) : BigNum
```

Traits, Objects, and Functional Methods

- Traits are rewritten into interfaces. (multiple inheritance)
- Objects are rewritten into classes.
- Fields are largely eliminated by getter and setter methods.
- Functional methods are translated into top level function definitions that trampoline into corresponding dotted method definitions.

Method Dispatch

- Java uses single dispatch, based on the runtime type of the method's owner.
- Fortress uses symmetric multiple dispatch based on the runtime type of the method's owner and the runtime types of the arguments.

- How do we implement our method dispatch on top of Java's?

Overloading

$which(x\colon \mathrm{Object}, y\colon \mathrm{Object})\colon \mathrm{String} = \text{``}\texttt{neither}\text{''}$ `(* 1 *)`
$which(x\colon \mathrm{Object}, y\colon \mathrm{String})\colon \mathrm{String} = \text{``}\texttt{second}\text{''}$ `(* 2 *)`
$which(x\colon \mathrm{String}, y\colon \mathrm{Object})\colon \mathrm{String} = \text{``}\texttt{first}\text{''}$ `(* 3 *)`
$which(x\colon \mathrm{String}, y\colon \mathrm{String})\colon \mathrm{String} = \text{``}\texttt{both}\text{''}$ `(* 4 *)`

Generates:

```
$which\$1\$entry(x\colon \mathrm{Object}, \colon \mathrm{Object})\colon \mathrm{String} =$
    $which\$1\$dispatch\big(x, x.ilk(), y, y.ilk()\big)$

$which\$1\$dispatch(x\colon \mathrm{Object}, \mathrm{T1}\colon \mathrm{Type}, y\colon \mathrm{Object}, \mathrm{T2}\colon \mathrm{Type})\colon \mathrm{String} =$
    if $which\$4\$applicable(\mathrm{T1}, \mathrm{T2})$ then $which\$4\$dispatch(x, \mathrm{T1}, y, \mathrm{T2})$
    elif $which\$2\$applicable(\mathrm{T1}, \mathrm{T2})$ then $which\$2\$dispatch(x, \mathrm{T1}, y, \mathrm{T2})$
    elif $which\$3\$applicable(\mathrm{T1}, \mathrm{T2})$ then $which\$3\$dispatch(x, \mathrm{T1}, y, \mathrm{T2})$
    else $which\$1(x, y)$
end
```

Or after inlining:

```
public static which(CompilerBuiltin$Object x , CompilerBuiltin$Object y) {
   if (x instanceof String) then
      if (y instanceof String) then
         which((String) x, (String) y)
...
```

Parametric Polymorphism

Potentially infinite possible instantiations (We can't do the above template expansion at compile time), so we use a lookup table.

```
private static BAlongTree sampleLookupTable = new BAlongTree();
pulic Object sampleLookup(long hashcode, String signature) {
   Object o = sampleLookupTable.get(hashcode);
   if (o == null)
      o = InstantiatingClassloader.findGenericMethodClosure(hashcode,
                                                            sampleLookupTable
                                                            "template_class_name"
                                                            signature);
      return o;
}
```

Run Time type checking

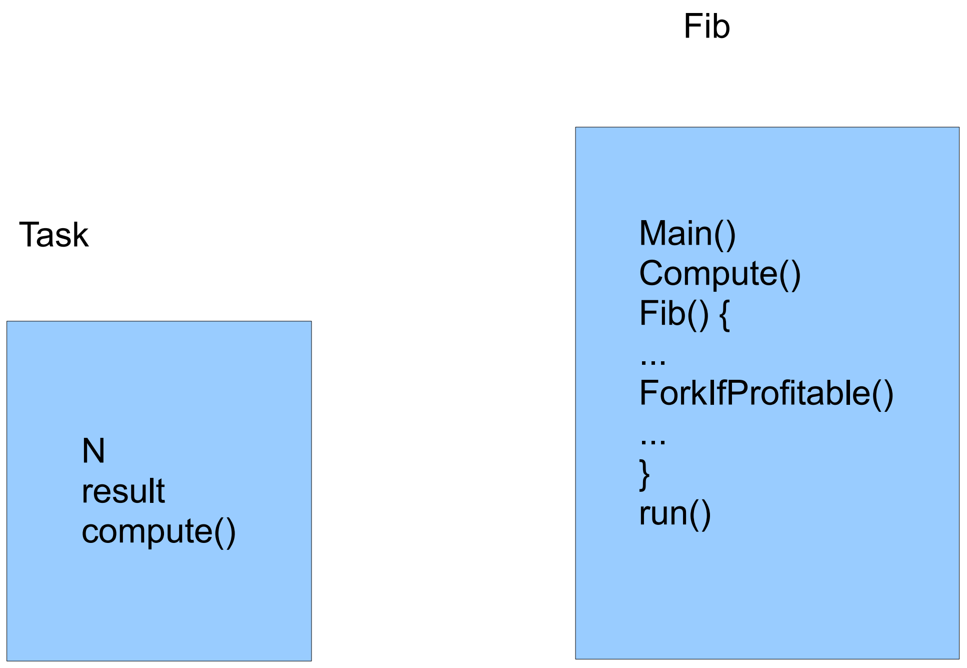
- The object we get back from the lookup must be assignable to the appropriate ArrowType.
- Integer extends Number which extends Object however F(Object) extends F(Number) which extends F(Integer). There is no good way to graft this upside down hierarchy onto Java's type system, so we do the checks ourselves.

Parallelism Example: Fib

```
export Executable
$\mathit{fib}(n\colon \mathbb{Z}32)\colon \mathbb{Z}32 =$
if $n \leq 2$ then $1$ else $\mathit{fib}(n-1) + \mathit{fib}(n-2)$
end
$\mathit{run}()\colon () = \mathit{println}(\mathit{fib}\,20)$
end
```

ORACLE

Implicit Parallelism in Fib



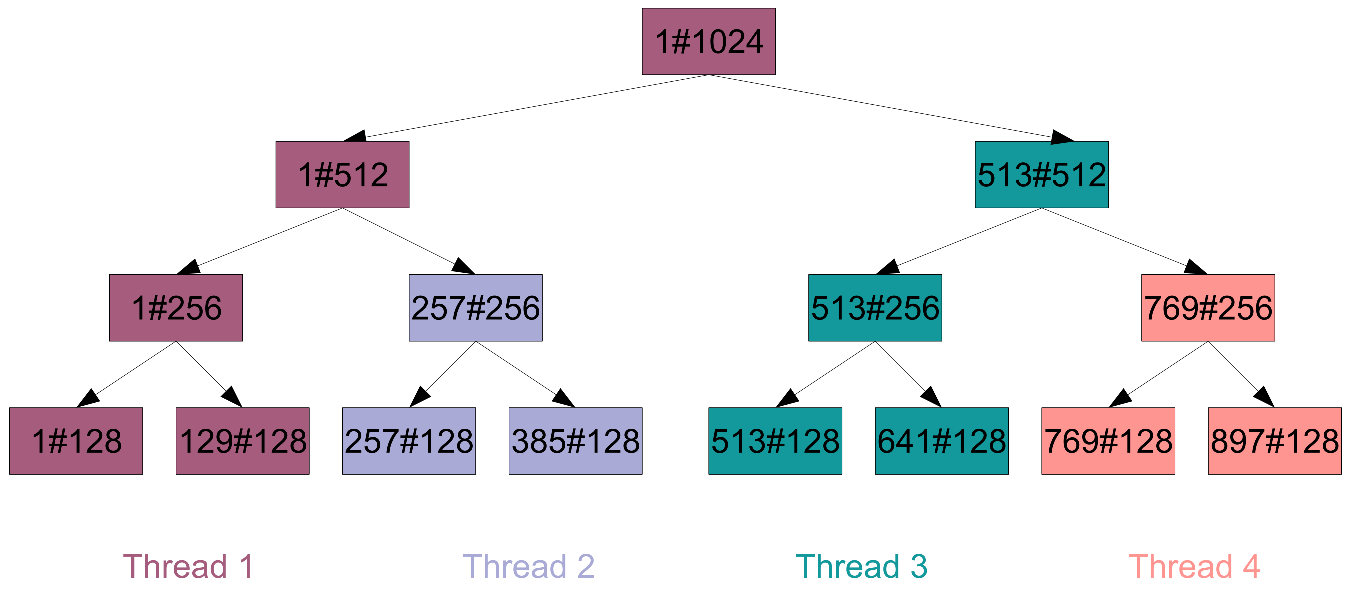
```
N
result
compute()
```

```
Main()
Compute()
Fib() {
...
ForkIfProfitable()
...
}
run()
```

Fork If Profitable

- Parallelism Analyzer
  - Does it contain a function call?
  - Does it contain a loop?
  - Does it contain a sufficient number of bytecodes?
- Runtime System
  - How Busy is the machine?

Work Stealing



ByteCode Optimization

- Why not just generate better byte codes to begin with?
- Using Bytecodes as our intermediate representation.
- What optimizations can we do that the JVM can't?
    - Inlining across classes
    - Unboxing of Fortress types (raw integer performance)
    - Linking

Bytecode Optimization Substitutions

If we see:

```
invokestatic "FFloatLiteral" "make" "(D)LFFloatLiteral;"
invokestatic "CompilerBuiltin" "coerce_RR64" "(LCompilerBuiltin$FloatLiteral;)LRR64;"
```

We know we can replace it with:

```
invokestatic "RR64" "make" "(D)LRR64;"
```

Bytecode Optimization

- Abstract Interpretation gives us the static types of stacks and locals
- We can incorporate knowledge about Fortress types
- $foo() \colon \mathbb{Z}32 = 7 + 5$
- would generate code to create two intliterals, convert them to ZZ32s, and use Fortress methods to add them.
- The bytecode optimizer knows that ZZ32 can never be extended and that add can never be overridden, so we
- can replace the generated code with a call to make ZZ32(7 + 5)
- assuming that none of the intliterals overflow 32 bits.

Tools we Use

- ASM
  - Generate bytecodes during compilation
  - Insert marshalling and unmarshalling for Fortress Types into Java Classes
  - Modify bytecodes during optimization
  - Generate new classes during runtime for generics.
- Scala for typechecking
- JSR166y for work stealing
- Rats! for packrat parsing
- Ant for building

ORACLE

Wishlist

- Tail Call Optimization (We heard it was coming, but we want it now.)
- Interface Injection (We need this for simplifying our generics)
- Closures (What (I think) you really want for work stealing)
- It sure would be nice to have better tools for folks that generate bytecodes.
  - Debugger that single stepped bytecodes.
  - Pretty viewer of bytecodes.