ProjectFortress:

Bytecode Optimizer

or

Bytecode as Intermediate Representation

Christine H. Flood
Oracle Labs

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

Mathematical Syntax

Run Your White Board!

$$y = \sum_{k \leftarrow 1:n} a_k \, x^k$$

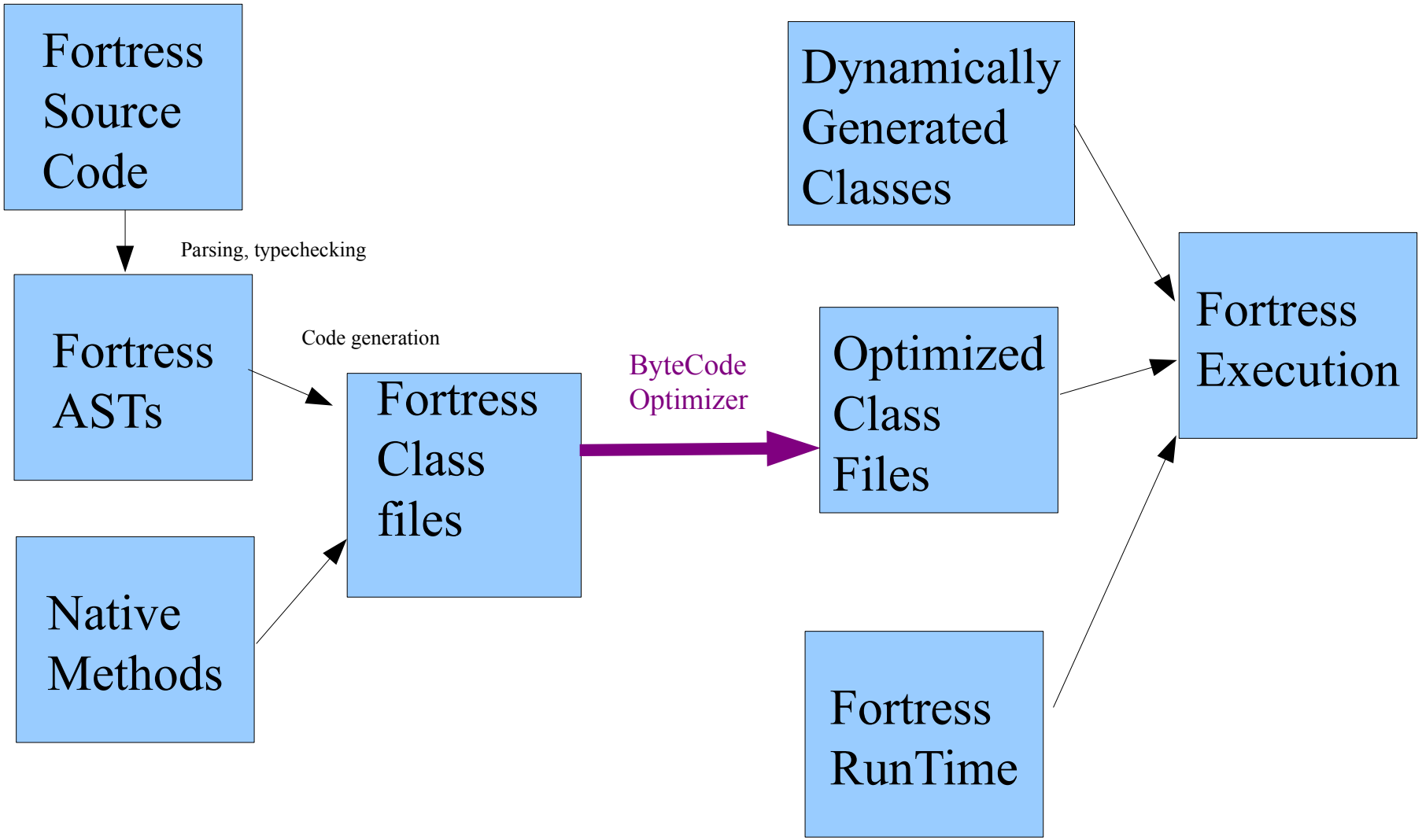
Or in Ascii ...

SUM[k<-1:n] a[k] x^k

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.

Current Compilation Strategy



Code Generation: ASM

- “ASM is an all purpose Java bytecode manipulation and analysis framework. It can be used to modify existing classes or dynamically generate classes, directly in binary form. Provided common transformations and analysis algorithms allow to easily assemble custom complex transformations and code analysis tools.”
- We use it to generate bytecode, to transform generated bytecode into optimized bytecode, and to dynamically generate classes for run time type method dispatch.

Bytecode as Intermediate Representation

- Why?
  - Convenience
  - We can execute both our optimized and unoptimized code.
  - ASM makes walking/optimizing bytecode easy.

ASM Instructions broken into groups

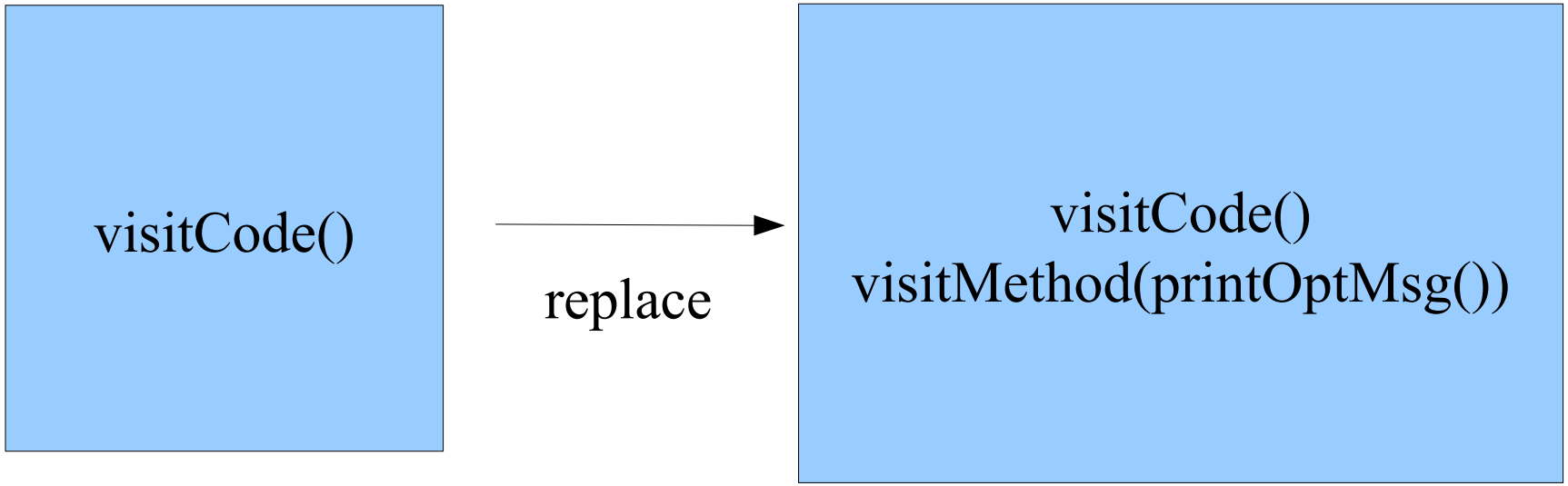
- FieldInsn
- IincInsn
- IntInsn
- JumpInsn
- LabelInsn
- LdcInsn
- LocalVariableInsn
- LookupSwitchInsn
- MethodInsn
- SingleInsn
- TableSwitchInsn
- TypeInsn
- VarInsn
- VisitLineNumberInsn

We mirror these groups with our own Insn super class.

- Index
- Expansion Instructions
- Parent Instruction
- IsDef, IsUse
- IsBoxing, isUnboxing
- Stack, Locals
- toAsm
- Match

Substitution

main([java/lang/String;)V



Substitution example

```
package com.sun.fortress.compiler.asmbytecodeoptimizer;

import java.util.ArrayList;

public class AddString {
   public static void optimize(ByteCodeVisitor bcv) {

      ByteCodeMethodVisitor bcmv = (ByteCodeMethodVisitor) bcv.methodVisitors.get("main([Ljava/lang/String;)V");
      if (bcmv != null) {
         ArrayList<Insn> matches = new ArrayList<Insn>();
         matches.add(new VisitCode());
         ArrayList<Insn> replacements = new ArrayList<Insn>();
         replacements.add(new VisitCode());
         replacements.add(new FieldInsn("GETSTATIC", Opcodes.GETSTATIC, "java/lang/System", "out", "Ljava/io/PrintStream;",
                                 "AddOptimizeString0"));
         replacements.add(new LdcInsn("LdcInsn", "Running Optimized Version", "AddOptimizeString1"));
         replacements.add(new MethodInsn("INVOKEVIRTUAL", Opcodes.INVOKEVIRTUAL, "java/io/PrintStream", "println",
                                 "(Ljava/lang/String;)V", "AddOptimizeString2"));
         Substitution s = new Substitution(matches, replacements);
         s.makeSubstitution(bcmv);
      }
   }
}
```

Substitution: Removing Literals

```
public static Substitution removeIntLiterals(ByteCodeMethodVisitor bcmv) {
        String intLiteral = "com/sun/fortress/compiler/runtimeValues/FIntLiteral";
        String ZZ32 = "com/sun/fortress/compiler/runtimeValues/FZZ32";

        ArrayList<Insn> matches = new ArrayList<Insn>();
        matches.add(new MethodInsn("INVOKESTATIC", Opcodes.INVOKESTATIC,
                                   intLiteral,
                                   "make",
                                "(I)L" + intLiteral + ";", "targetedForRemoval"));
        matches.add(new LabelInsn("LabelInsn", new Label(), "targetedForRemoval"));
        matches.add(new VisitLineNumberInsn("visitlinenumber", 0, new Label(),
"targetedForRemoval"));
        matches.add(new MethodInsn("INVOKESTATIC", Opcodes.INVOKESTATIC, "
“fortress/CompilerBuiltin",
                                   "coerce_ZZ32",
                                "(Lfortress/CompilerBuiltin$IntLiteral;)L" + ZZ32 + ";",
"targetedForRemoval"));
        ArrayList<Insn> replacements = new ArrayList<Insn>();
        replacements.add(new MethodInsn("INVOKESTATIC", Opcodes.INVOKESTATIC,
                                   ZZ32,
                                   "make",
                                "(I)L" + ZZ32 + ";", "ReplacementInsn"));
        return new Substitution(matches, replacements);
    }
```

Inlining

- Why do we do our own inlining?
  - Enables more aggressive unboxing
  - Eliding Fortress Dispatch for base types.
- What to inline?
- How to inline it?
  - Variable renaming
  - Label renaming
  - Stack management
  - End management

Unboxing

- Why?
  - We were spending the bulk of our time allocating/deallocating boxed values.
- How?
  - Built upon bytecode verification.

Unboxing:Abstract Interpretation

- Walk the bytecodes figuring out the values of the stack and locals at the boundaries of each basic block. (Bytecode verification) (revived 1998 verifier)
- Keep track of the definition and use of each value, noting if it was boxed or unboxed.
- If the only use of a value was unboxed, remove all boxing/unboxing instructions.
- If the majority of the uses are unboxed, move the boxing instructions to where the boxed values are needed.

Where's the Beef?

Numeric Integration benchmark

Total time before:	345.497
Total time after:	58.309

Which gave us a factor of 6 speedup.

Caveat: We are still a factor of 4 slower than Java.

Future Work

- Generate boxed and unboxed versions of methods.
- Move parts of the optimizer into the runtime so we can optimize generated classes.

Wishlist: I had to add this, it doesn't look much different from last year.

- Tail call optimization
  - Especially important for mostly functional languages like Fortress.
- Interface Injection
  - Because recursive types could potentially require an infinite number of methods the entire type hierarchy can't be generated at compile time, and some classes must be generated on demand at run time. Interface injection would save us a whole lot of complicated dispatch code.

The End