

# Optimization of Run-time Program Generators by Partial Evaluation

Sam Kamin

Univ.of Illinois, Urbana-Champaign

[loome.cs.uiuc.edu](mailto:loome.cs.uiuc.edu)

# Program Generation in Jumbo

Jumbo: Java-based RTPG system similar to `C, DynJava, etc.

```
public static ExpClass getExp(int n)
{
    Code r = $<1>$;
    for(int i = 0; i < n; i++)
        r = $<`r * x>$;
    String cname = "ExpClass"+n;
    Code expcl = $< public class `cname implements ExpClass {
                            public int exponent(int x) {
                                return `r;
                            }
                        } >$;
    return (ExpClass)expcl.create(cname);
}
```

# Outline of talk

- Overview
  - Approach to constructing RTPG systems using compositional compilation, and its optimization
- Mumbo: A demonstration project
  - Language
  - Optimizations
  - Results
- Progress with optimizing Jumbo
- Conclusions

# The Jumbo approach to run-time program generation I

First goal: *Write one compiler, use its back end as run-time code generator*

- Advantages
  - One compiler
  - No restriction on generated code
- Disadvantages
  - Efficiency
  - No restriction on generated code

# The Jumbo approach to run-time program generation II

Second goal: *Partially evaluate compiler back-end on partial program or fragment*

- Advantages
  - Increased efficiency
  - Binary-level operation
- Disadvantages
  - Hard to do!

# Program Generation in Jumbo

Steps in producing run-time code generator.

One: programmer writes code using quotation and anti-quotation:

```
public static ExpClass getExp(int n)
{   Code r = $<1>$;
    for(int i = 0; i < n; i++)
        r = $<`r * x>$;
    String cname = "ExpClass"+n;
    Code expcl = $< public class `cname implements ExpClass {
                                public int exponent(int x) {
                                    return `r;
                                }
                                } >$;
    return (ExpClass)expcl.create(cname);
}
```

# Program Generation in Jumbo

Two: Preprocessor removes quoted code,  
transforms to AST operations:

```
public static ExpClass getExp(int n)
{
    Code r = mkIntConst(1);
    for(int i = 0; i < n; i++)
        r = mkBinOp(MUL, r, mkVariable("x"));
    String cname = "ExpClass"+n;
    Code expcl = mkClass(true, cname, "ExpClass",
                        mkMethod(mkIntType(), "exponent", "int", "x",
                                mkReturn(r)));
    return (ExpClass)expcl.create(cname);
}
```

# Program Generation in Jumbo

Three: At run time, execution of code-generating code builds larger AST expression. For example, if  $n=3$ , we get:

```
Code expcl =
    mkClass(true, "ExpClass3", "ExpClass",
        mkMethod(mkIntType(), "exponent", "int", "x",
            mkReturn(mkBinOp(MUL, mkBinOp(MUL, mkBinOp(MUL, mkIntConst(1),
                mkVariable("x")), mkVariable("x")), mkVariable("x"))));
return (ExpClass)expcl.create(cname);
```

But we ought to be able to do better than just creating AST's and then running a compiler...

# Goal of optimizations

What the exp example “should” look like – i.e. what a programmer knowledgeable about JVM would write:

```
public static ExpClass getExp(int n) {  
    Code r = "LOADI 1";  
    for(int i = 0; i < n; i++)  
        r = "r  
            MULTI x";  
    String cname = "ExpClass"+n;  
    Code expcl = ... emit instructions to build class...  
    return (ExpClass)expcl.create(cname);  
}
```

Instead of creating entire AST, emit instructions directly.

# Getting from here to there

How do we go from:

```
for(int i = 0; i < n; i++)  
    r = mkBinOp(MUL, r, mkVariable("x));
```

to:

```
for(int i = 0; i < n; i++)  
    r = "r  
    MULTI x";
```

# Optimizing RTPG by source-level transformation

Expression built from AST operations *is* a piece of Java source code. Each AST operation is a function that compiles its particular type of expression or statement.

Crucial point: AST op's are not simply tree-constructing operations. (See "compositional compilation.")

# Compositional compilation

- Compilation of fragment is function of the compilation of sub-fragments.

$comp$ : Program Fragment  $\rightarrow$  Code

$comp(\oplus(e1, e2)) = comp_{\oplus}(comp(e1), comp(e2))$

where Code = *as close to machine language as possible*

- For example:

$comp(\text{"while (cond) stmt"}) =$

$comp_{\text{while}}(comp(\text{cond}), comp(\text{stmt}))$

# Why compositional compilation?

Compositionality implies:

- Can compile fragments separately.
- Can compile programs with holes:  
 $comp_{\square}: \text{Program-w/-hole} \rightarrow \text{Code} \rightarrow \text{Code}$   
 $comp_{\square}(P[\bullet])(c) = comp(P[A])$ , where  $A$  is  
any program such that  $comp(A) = c$ .
- Can optimize fragments and programs with holes.

# *Code* $\neq$ Machine Language

For compositionality to be possible, codomain of compilation function must be richer than machine language, e.g.

*Code* = *Environment*  $\rightarrow$  *JVM*

where *Environment* gives contextual information about variable types, etc.

Then need *codegen*: *Code*  $\rightarrow$  *JVM*

# Jumbo

$Code = Environment \rightarrow ClosedCode$

$ClosedCode = JVM \times JVM \times ConstVal$   
 $\times Type \times Environment$

implemented as function objects with *eval* operation.

To summarize: In Jumbo, every AST operation is actually a function from the Code values of its arguments to a Code value. Thus, it is a higher-order function.

# Why is optimization hard?

- Consider the definition of mkBinOp:

```
public static Code binOp(final int op, final Code c1, final Code c2) {  
    switch (op) {  
        case CMP_EQ: return binCompare(Instr.CMP_EQ, c1, c2);  
        ...  
        case MUL: return mathOp(Instr.MUL, c1, c2);  
        ... }  
}
```

# Why is optimization hard?

```
public static Code mathOp(final int op, final Code c1, final Code c2) {
    return new Code() {
        public ClosedCode eval(Environment env) {
            final ClosedCode cp = evalInAdd(env);
            if (op == Instr.ADD &&
                cp.type.isObjectType("java/lang/StringBuffer")) {
                return new ClosedCode(cp.bytecode.
                    addInvokevirtual("java/lang/StringBuffer", "toString",
                        new MethodType(Type.string_type)),
                    env, Type.string_type);
            }
            else return cp;
        }
    }
}
```

# Summary

- An abstract syntax optr is a function from  $\text{Code}^n$  to  $\text{Code}$ . Code values are function objects implementing  $\text{ClosedCode eval}(\text{Env})$ .
- Thus, a program fragment with holes – a code generator - is a function from  $\text{Code}^n$  to  $\text{Code}$ .
- Jumbo compiler consists of about 5000 lines of AST operator definitions.
- ∴ In principle, can use partial evaluation to optimize code generation. In practice...

# Outline of talk

- Overview
  - Approach to constructing RTPG systems using compositional compilation, and its optimization
- **Mumbo: A demonstration project**
  - **Language**
  - **Optimizations**
  - **Results**
- Progress with optimizing Jumbo
- Conclusions

# Mumbo

- Simplified language, modeling Jumbo
- Typed, object-oriented, and supporting RTPG
- Keeps crucial properties of Jumbo
- Rule-based implementation
- Implemented in Maude
- Compiled to LowLevel, which is
  - a 3-address code language
  - implemented in Maude
- Jumbo → JVM bytecode, Mumbo → LowLevel

# Mumbo: Optimization

- Define rules to optimize the code at source level
- Pretty standard rules
  - Constant Propagation, Copy Assignment, Loop Unfolding, Inlining, Field Value, etc.
- Using heuristics to automate optimization
- Rules work fine because of side-effect-freeness

# Mumbo: Analyses

- **INFO**: Encapsulates expressions in `info` packages to store information.
- **TAG**: Assigns unique number to names, loops and method invocations.
- **UNTAG**: Removes tags.
- **TYPE**: Propagates type information over expressions.
- **ISLOCAL**: Determines if a variable is local to the enclosing method.
- **ISFINAL**: **Determines if an accessed field is final.**
- **GEN-KILL**: Computes gen-kill sets of expressions.
- **USE-DEF**: Computes use-def chains. (can do *may* or *must* analysis)

# Mumbo: Rules

- Rules requiring analysis results
  - Inline
  - Unroll
  - Copy Assignment
  - Constant Propagation
  - Nil-Check
  - Useless Definition
  - Useless Declaration
  - Field Value
  - Useless Instantiation

# Mumbo: Rules

- Rules *not* requiring analysis results
  - $\langle \{ \{ eb \} ; eb' \} \rangle \Rightarrow \langle \{ eb ; eb' \} \rangle$
  - $\langle \text{if True then } e_1 \text{ else } e_2 \rangle \Rightarrow \langle e_1 \rangle$
  - $\langle \text{self } \rightarrow x \rangle \Rightarrow \langle x \rangle$
  - $\langle \text{set lhs} = \{ eb ; e \} \rangle \Rightarrow \langle \{ eb ; \text{set lhs} = e \} \rangle$
  - $\langle \text{send } \{ eb ; e \} x el \rangle \Rightarrow \langle \{ eb ; \text{send } e x el \} \rangle$
  - *etc.*

# Mumbo: Benchmarking

- Speedups compare the run-time generation costs before and after optimization
- First example:
  - A complete class (i.e. no *holes*) --- ideal case
    - 70% speedup in run-time generation

```
$< class Gen extends object
    field x : int

    method bar() int :
    { set x = 12 ;
      x * 2
    }
>$
```

# Mumbo: Example (A Class with 2 holes)

- A class with two holes (one hole for a field, one hole for a method body)
  - 53% speedup

```
$< class Gen extends object
    ^F(f)

    method bar() int :
        ^ (body)
>$
```

# Mumbo: Example (A Class with 2 holes)

## *Original eval method*

```
method eval(env : Env) [--- local var decls ---] ClosedCode :
{
  --- define environment
  set env = send (send env resetForNewClass(name, superName))
                addList(fieldList) ;
  set sname = superName ;
  while not(sname equals ['object'])
  { --- add fields to the environment
    set spr = send env getClassInfo(sname) ;
    set env = send env addList(send spr getFieldList()) ;
    set sname = send spr getSuperName()
  } ;
  --- defined environment

  set lowlevel = [""] ;
  set meth = methodList ;
  while (send meth hasNext())
  { --- evaluate the method
    set methVal = cast send meth value() to Code ;
    set lowlevel = lowlevel # (send (send methVal eval(env)) getCode()) ;
    set meth = send meth next()
  } ;
  new ClosedCode(lowlevel, [" "])
}
```

# Mumbo: Example (A Class with 2 holes)

## *Optimized eval method*

```
method eval( env_20 : Env) [--- local var decls ---] ClosedCode :
{ set sym858 = env_20 ;
  set sym853 = new Env(new LinkedList(NIL,NIL),
                      new LinkedList(NIL,NIL),
                      [ Gen],
                      [ object],
                      sym858 -> classInfos) ;

  --- add the field to the environment
  set tempEnv_856 = send f_846 addToEnv( sym853) ;
  set sym6443 = new Env(new LinkedList(NIL,NIL),
                      tempEnv_856 -> fieldList,
                      { set sym8968 = tempEnv_856 -> currClass ;
                        sym8968 },
                      tempEnv_856 -> superClass,
                      tempEnv_856 -> classInfos) ;

  set lowlevel_4602 = sym8968 # [" $ bar (% self" ] ;
  set lowlevel_4602 = lowlevel_4602 # [" ) { entry : " ] ;

  --- evaluate body of the method
  set bval_4600 = send body_479 eval( sym6443) ;
  set lowlevel_4602 = lowlevel_4602 # bval_4600 -> lowlevelcode
                      # [" br end ; end : return " ]
                      # bval_4600 -> name # [" ; } " ] ;

  new ClosedCode( lowlevel_4602, [" " ] )
}
```

# Mumbo: Example (Binary Operation)

- A binary expression
  - 23% speedup

```
--- original code fragment
$< `(x) * 10 >$

--- after preprocessing
new binOpCode([mul],
               x,
               new integerConstantCode(10))

--- after optimization
new binOpCode-324(x)
```

# Mumbo: Example (Binary Operation)

*Original eval method (in binOpCode):*

```
--- left, right and op are fields of the object
method eval(env : Env) [---local var decls---] ClosedCode :
{
  set sym = GENSYM ;
  --- evaluate right and left operands
  set lval = send left eval(env) ;
  set rval = send right eval(env) ;
  --- get the results of left and right, and do the binaryOp
  set lowlevel = (send lval getCode())
                  # (send rval getCode())
                  # sym # [" = "] # op # ["("]
                  # (send lval getName()) # [", "]
                  # (send rval getName()) # [")"] # [" ; "] ;
  new ClosedCode(lowlevel, sym)
}
```

# Mumbo: Example (Binary Operation)

*Optimized eval method (in binOpCode-324):*

```
--- left is a field of the object
method eval( env_5 : Env)[---local var decls---]ClosedCode :
{
  set sym_9 = GENSYM ;
  --- evaluate the left operand, which was a hole
  set lval_7 = send left eval( env_5) ;
  set sym_227 = GENSYM ;
  --- lowLevel code of the right operand completely resolved
  set lowlevel_226 = sym_227 # [" = add(0,10) ; " ] ;
  --- multiply the result of left and right operands
  set lowlevel_6 = lval_7->lowlevelcode
                  # lowlevel_226 # sym_9 # [" = mul("]
                  # lval_7->name
                  # [" , " ] # sym_227 # [" ) ; " ] ;
  --- return the resulting code
  new ClosedCode( lowlevel_6, sym_9)
}
```

# Mumbo: Benchmarking

- Classical examples
  - Exponentiation
    - various exponents, giving more than 50 % speedup
  - Loop Unrolling
    - various number of iterations, giving 47-60% speedup

Unrolling version	Num. of Iterations	Speed-up (%)
1	1	48
1	10	56
1	100	60
1	200	60
2	1	47
2	10	52
2	100	54
2	200	54
3	1	48
3	10	53
3	100	54
3	200	54

WG2.11 - Houston - March, 2005

# Mumbo: Limits of Optimization Rules

- Rules do not know what the holes are capable of doing
  - `$< method foo(x : int)`  
    `` (c) + x`  
    `>$`
  - The rules cannot reason about the scope of `x` at its occurrence in the body (This happens when `Code` objects are allowed to change the environment)

# Mumbo: Conclusion

- Side-Effect-Free style may help us to obtain substantial speedup with standard, well-known rules
- Simplified model provides fast progress
- A “testbed” for new ideas

# Outline of talk

- Overview
  - Approach to constructing RTPG systems using compositional compilation, and its optimization
- Mumbo: A demonstration project
  - Language
  - Optimizations
  - Results
- **Progress with optimizing Jumbo**
- Conclusions

# Source-level optimizations for Java

- Similar transformations:
  - Inlining
  - WhileUnroll
  - UnusedDecl
  - UnreachableCode
  - Switch
  - ... *etc.* ...

all built on a complex alias analysis

# Optimizing exponentiation

```
public static ExpClass getExp(int n)
{
    Code r = $<1>$;
    for(int i = 0; i < n; i++)
        r = $<`r * x>$;
    String cname = "ExpClass"+n;
    Code expcl = $< public class `cname implements ExpClass {
        public int exponent(int x) {
            return `r;
        }
    } >$;
    return (ExpClass)expcl.create(cname);
}
```

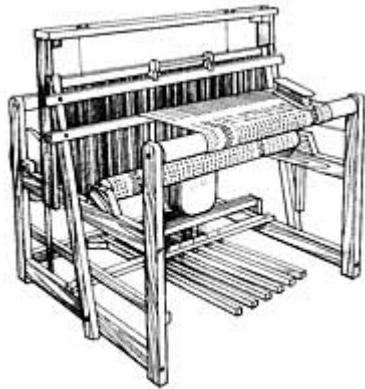
# Optimization results for Jumbo

- For exponentiation example:
  - HotSpot (with JIT compilation): 7% improvement
  - HotSpot (interpreted mode): 3% improvement
  - Kaffe (with JIT compilation): 6% improvement

# Conclusions

- The Jumbo approach to RTPG uses the back end of a static compiler – assuming the compiler is structured compositionally – to do run-time program generation
- This approach offers the opportunity for optimization of RTPG via source-level transformation of the compiler back end, based on the known parts of the code to be generated
- In practice, it is difficult to realize this optimization
- Mumbo was an experiment to demonstrate the feasibility of the idea.
- Complexity of Java is still a stumbling block, but we're working on it...

# Questions?



## LOOME

**L**anguage **T**ools for  
**M**odularity and  
**E**fficiency

- URL: [loome.cs.uiuc.edu](http://loome.cs.uiuc.edu)