A Compiler... with Composition?

Bryan Chadwick

Demeter Seminar
11/1/2007

What

- Simple Expressions
- Compiled into HL Assembly
- Optional Argument/Optimization
- Stack Based JVM-like 'Machine'

Tools

- Java
- DemeterJ
- Transform/Traversal Library

First Language

Simple Arithmetic

```
E : S | C.
S = <n> Integer.
C = "(" Op *s <arg1> E *s <arg2> Option_E ")".

Op : P | M | T | D.
P = "+".
M = "-".
T = "*".
D = "/".

Option_E : NonEmpty_E | Empty_E.
NonEmpty_E = <s> E.
Empty_E = .
```

First Language

Assembly Instructions

```
Opcode: MathOp | MemOp.

MathOp: Plus | Minus | Times | Divide.
Plus = "plus".
Minus = "minus".
Times = "times".
Divide = "divide".

MemOp: Push | Pop
Push = "push" <i> Integer.
Pop = "pop".
```

First Language

The Compiler - 1

```
class FuncCompile implements Compiler{
  OpList compile(E e){
    Traversal comp = new Traversal(new Code());
    return comp.traverse(e);
  }
}

class Code extends IDfb{
  OpList single(Opcode o){ return new OpEmpty().append(o); }

  OpList combine(S s, Integer i){ return single(new Push(i)); }
  OpList combine(Option_E o, OpList l){ return l; }

  OpList combine(P p){ return single(new Plus()); }
    ...
  OpList combine(C c, OpList op, OpList left, OpList right){
    return right.append(left).append(op);
  }
}
```

First Language

The Compiler - 2

```
class PreCode extends IDf{
  E apply(C c){
    return c.arg2.hasE() ? c : c.arg1;
  }
}
```

Example 1

Expression

```
(+ 4 (- (* 2) (/ 25 5)))
```

Code

```
push 5
push 25
divide
push 2
minus
push 4
plus
```

Second Language

Simple Arithmetic

```
E : ... | V | A | L.
V = <id> Ident.
A = "[" <i> Integer "]".
L = "(let" *s <id> Ident "=" <e> E "in" <body> E ")".
```

Assembly Instructions

```
MemOp: ... | Def | Undef | Load.
Def = "def".
Undef = "undef".
Load = "load" <i> Integer.
```

Second Language

The Compiler - 1

```
// Reverse Let: (let E as (V in E))
class RevLet extends E{
  E e;
  BB bb;
  RevLet(E ee, BB bbb){ e = ee; bb = bbb; }
}

class BB{
  Ident v;
  E b;
  BB(Ident vv, E bb){ v = vv; b = bb; }
}

class FixLet extends IDf{
  E apply(L l){ return new RevLet(l.e, new BB(l.id, l.body)); }
}
```

Second Language

The Compiler - 2

```
class PreCode2 extends PreCode{
  VStack update(BB b, VStack s){ return s.push(new V(b.v)); }
  E apply(V v, VStack s){ return new A(s.lookup(v)); }
}

class Code2 extends Code{
  OpList combine(A a, Integer i){ return single(new Load(i)); }
  OpList combine(BB bb, Object id, OpList b){ return b; }
  OpList combine(RevLet l, OpList e, OpList body){
    return e.append(new Def()).append(body).append(new Undef());
  }
}
```

Example 2

Expression

```
(/ (let a = 6 in
      (let b = 5 in
         (- (* a 2) b))) 2)
```

Code

```
push 2
push 6
def
push 5
def
load 0
push 2
load 1
times
minus
undef
undef
divide
```

Third Language

Simple Arithmetic

```
E : ... | I .
I = "(if" *s <c> E *s <t> E *s <e> E ")".

Op: ... | LT | GT | EQ | AndF | OrF.
LT = "<".
GT = ">".
EQ = "=".
AndF = "and".
OrF = "or".
```

Assembly Instructions

```
Opcode: ... | ControlOp.
MathOp: ... | Less | Greater | Equal | And | Or.

ControlOp: Label | Jmp | IfZ.
Label = "label" <id> Ident.
Jmp = "jump" <id> Ident.
IfZ = "ifzero" <id> Ident.
```

Third Language

The Compiler

```
class Code3 extends Code2{
  int lnum = 0;

  OpList combine(LT l){ return single(new Less()); }
  OpList combine(GT g){ return single(new Greater()); }
  OpList combine(EQ e){ return single(new Equal()); }
  OpList combine(AndF a){ return single(new And()); }
  OpList combine(OrF o){ return single(new Or()); }

  OpList combine(I f, OpList c, OpList t, OpList e){
    Ident l1 = new Ident("else_"+lnum++),
          l2 = new Ident("done_"+lnum++);
    return c.append(new IfZ(l1)).append(t)
            .append(new Jmp(l2)).append(new Label(l1))
            .append(e).append(new Label(l2));
  }
}
```

Example 3

Expression

```
(let a = 5 in (if (< a 3) (* 5 a) (/ a 2)))
```

Code

```
push 5
def
push 3
load 0
less
ifzero else_0
load 0
push 5
times
jump done_1
label else_0
push 2
load 0
divide
label done_1
undef
```

Final Compile Function

Expression

```
class FuncCompile implements Compiler{
  public OpList compile(E e){
    Traversal pre = new Traversal(new PreCode2());
    Traversal fix = new Traversal(new FixLet());
    Traversal comp = new Traversal(new Code3());

    e = pre.traverse(fix.<E>traverse(e),new VEmpty());
    return comp.traverse(e, new VEmpty());
   }
 }
```

Can we do just one Pass?

Can we do just two Passes?