Introduction to Compiler Construction

Marvin Code Generation: High-level Intermediate Representation (HIR)

Outline

1. Control Flow Graph
2. High Level Intermediate Representation (HIR)
3. Local Optimizations

Control Flow Graph

Control Flow Graph

We scan through the intermediate JVM instructions and construct a flow graph of basic blocks

Control Flow Graph

We scan through the intermediate JVM instructions and construct a flow graph of basic blocks

A basic block is a sequence of instructions with just one entry point at the start and one exit point at the end — there are no branches into or out of the instruction sequence

Control Flow Graph

We scan through the intermediate JVM instructions and construct a flow graph of basic blocks

A basic block is a sequence of instructions with just one entry point at the start and one exit point at the end — there are no branches into or out of the instruction sequence

Next, we identify basic blocks that are loop headers (LH) and loop tails (LT)

Control Flow Graph

We scan through the intermediate JVM instructions and construct a flow graph of basic blocks

A basic block is a sequence of instructions with just one entry point at the start and one exit point at the end — there are no branches into or out of the instruction sequence

Next, we identify basic blocks that are loop headers (LH) and loop tails (LT)

Next, we remove unreachable basic blocks (ie, basic blocks that are not reachable from the source block)

Control Flow Graph · Example (`Factorial.iota`)

```
// Accepts n (int) from standard input and writes n! (computed iteratively) to standard output.

// Returns n! computed iteratively.
int factorial(int n) {
    int result = 1;
    int i = 1;
    while (i <= n) {
        result = result * i;
        i = i + 1;
    }
    return result;
}

// Entry point.
void main() {
    int n = read();
    write(factorial(n));
}
```

Control Flow Graph · Example (CFGs for `Factorial.iota`)

```
>>> factorial(I)I

  B0 (pred: [], succ: [B1]):

  B1 (pred: [B0], succ: [B2]):
  0: ldc 1
  2: istore 1
  4: ldc 1
  6: istore 2

  B2 (pred: [B1, B3], succ: [B3, B4], LH):
  8: iload 2
  10: iload 0
  12: if_icmpgt 32

  B3 (pred: [B2], succ: [B2], LT):
  15: iload 1
  17: iload 2
  19: imul
  20: istore 1
  22: iload 2
  24: ldc 1
  26: iadd
  27: istore 2
  29: goto 8

  B4 (pred: [B2], succ: []):
  32: iload 1
  34: ireturn
```

Control Flow Graph · Example (CFGs for `Factorial.iota`)

```
>>> main()V

  B0 (pred: [], succ: [B1]):

  B1 (pred: [B0], succ: []):
  0: invokestatic read()I
  3: istore 0
  5: iload 0
  7: invokestatic factorial(I)I
  10: invokestatic write(I)V
  13: return
```

High Level Intermediate Representation (HIR)

High Level Intermediate Representation (HIR)

Next, we convert the tuples to HIR

High Level Intermediate Representation (HIR)

Next, we convert the tuples to HIR

HIR employs static single assignment (SSA) form, where for every variable, there is just one place in the method where that variable is assigned a value

High Level Intermediate Representation (HIR)

Next, we convert the tuples to HIR

HIR employs static single assignment (SSA) form, where for every variable, there is just one place in the method where that variable is assigned a value

For example, the following code

```
x = 3;
x = x + y;
```

is expressed in SSA form as

```
x1 = 3;
x2 = x1 + y1;
```

High Level Intermediate Representation (HIR)

High Level Intermediate Representation (HIR)

In the HIR we represent a variable's value by the instruction that computed it and we track these values in the state vector

High Level Intermediate Representation (HIR)

In the HIR we represent a variable's value by the instruction that computed it and we track these values in the state vector

The values in a state vector may change as we sequence through the block's instructions

High Level Intermediate Representation (HIR)

In the HIR we represent a variable's value by the instruction that computed it and we track these values in the state vector

The values in a state vector may change as we sequence through the block's instructions

If the next block has just one predecessor, it can copy the predecessor's state vector at its start; if there are two or more predecessors, the states must be merged

High Level Intermediate Representation (HIR)

High Level Intermediate Representation (HIR)

For example, consider the following *iota* method, where the variables are in SSA form

```
int ssa(int w1) {
    if (w1 > 5) {
        w2 = 0;
    } else if (w1 > 3) {
        w3 = 1;
    } else {
        w4 = 2;
    }
    return w?;
}
```

High Level Intermediate Representation (HIR)

For example, consider the following *iota* method, where the variables are in SSA form

```
int ssa(int w1) {
    if (w1 > 5) {
        w2 = 0;
    } else if (w1 > 3) {
        w3 = 1;
    } else {
        w4 = 2;
    }
    return w?;
}
```

In the statement

```
return w?;
```

which `w` do we return?

High Level Intermediate Representation (HIR)

For example, consider the following *iota* method, where the variables are in SSA form

```
int ssa(int w1) {
    if (w1 > 5) {
        w2 = 0;
    } else if (w1 > 3) {
        w3 = 1;
    } else {
        w4 = 2;
    }
    return w?;
}
```

In the statement

```
return w?;
```

which `w` do we return?

We solve this problem by using a Phi function, an HIR instruction that captures the possibility of a variable having one of several values

High Level Intermediate Representation (HIR)

For example, consider the following *iota* method, where the variables are in SSA form

```
int ssa(int w1) {
    if (w1 > 5) {
        w2 = 0;
    } else if (w1 > 3) {
        w3 = 1;
    } else {
        w4 = 2;
    }
    return w?;
}
```

In the statement

```
return w?;
```

which `w` do we return?

We solve this problem by using a Phi function, an HIR instruction that captures the possibility of a variable having one of several values

In our example, the final block would contain the following code

```
w5 = phi(w2, w3, w4);
return w5;
```

High Level Intermediate Representation (HIR)

Another place where Phi functions are needed are in loop headers

High Level Intermediate Representation (HIR)

Another place where Phi functions are needed are in loop headers

```
B1 (pred: [B0], succ: [B2], locals: [w1, x1]):

w1 = 1
x1 = 1

B2 (pred: [B1], succ: [B3], LH, locals: [w2, x2]):

w2 = phi(w1, w3)
x2 = phi(x1, x2)

B3 (pred: [B2], succ: [B2], LT, locals: [w3, x2]):

w3 = w2 + 1
```

High Level Intermediate Representation (HIR)

Another place where Phi functions are needed are in loop headers

```
B1 (pred: [B0], succ: [B2], locals: [w1, x1]):

w1 = 1
x1 = 1

B2 (pred: [B1], succ: [B3], LH, locals: [w2, x2]):

w2 = phi(w1, w3)
x2 = phi(x1, x2)

B3 (pred: [B2], succ: [B2], LT, locals: [w3, x2]):

w3 = w2 + 1
```

Redundant phi functions, ie, phi functions of the form `x = phi(y, x, x, ..., x)`, are replaced with `y`

High Level Intermediate Representation (HIR)

Another place where Phi functions are needed are in loop headers

```
B1 (pred: [B0], succ: [B2], locals: [w1, x1]):

w1 = 1
x1 = 1

B2 (pred: [B1], succ: [B3], LH, locals: [w2, x2]):

w2 = phi(w1, w3)
x2 = phi(x1, x2)

B3 (pred: [B2], succ: [B2], LT, locals: [w3, x2]):

w3 = w2 + 1
```

Redundant phi functions, ie, phi functions of the form `x = phi(y, x, x, ..., x)`, are replaced with `y`

In the above example, `x2` is replaced with `x1`

High Level Intermediate Representation (HIR) · Example (HIRs for `Factorial.iota`)

```
>>> factorial(I)I

  B0 (pred: [], succ: [B1], locals: [I0, ?, ?]):
  I0: ldparam 0

  B1 (pred: [B0], succ: [B2], locals: [I0, I1, I2]):
  I1: ldc 1
  I2: ldc 1

  B2 (pred: [B1, B3], succ: [B3, B4], LH, locals: [I0, I4, I5]):
  I4: phi(I1, I7)
  I5: phi(I2, I9)
  6: if I5 > I0 then B4 else B3

  B3 (pred: [B2], succ: [B2], LT, locals: [I0, I7, I9]):
  I7: I4 * I5
  I8: ldc 1
  I9: I5 + I8
  10: goto B2

  B4 (pred: [B2], succ: [], locals: [I0, I4, I5]):
  I11: ireturn I4
```

High Level Intermediate Representation (HIR) · Example (HIRs for `Factorial.iota`)

```
>>> main()V

  B0 (pred: [], succ: [B1], locals: [?]):

  B1 (pred: [B0], succ: [], locals: [I0]):
  I0: invoke read()
  I1: invoke factorial(I0)
  V2: invoke write(I1)
  3: return
```

Local Optimizations

Local Optimizations

That the HIR is in SSA form makes it amenable to several simple optimizations, which make for fewer instructions and/or faster programs

Local Optimizations

That the HIR is in SSA form makes it amenable to several simple optimizations, which make for fewer instructions and/or faster programs

Local optimizations are improvements made based on analysis of the linear sequence of instructions within a basic block

Local Optimizations

Local Optimizations

Expressions having operands that are both constants, or variables whose values are known to be constants, can be folded, that is replaced by their constant value

Local Optimizations

Expressions having operands that are both constants, or variables whose values are known to be constants, can be folded, that is replaced by their constant value

For example, consider the *iota* method

```
int f() {
    int i = 1;
    int j = 2;
    int k = i + j + 3;
    return k;
}
```

and the corresponding HIR code

```
  B0 (pred: [], succ: [B1], locals: [?, ?, ?]):

  B1 (pred: [B0], succ: [], locals: [I0, I1, I4]):
  I0: ldc 1
  I1: ldc 2
  I2: I0 + I1
  I3: ldc 3
  I4: I2 + I3
  I5: ireturn I4
```

Local Optimizations

Expressions having operands that are both constants, or variables whose values are known to be constants, can be folded, that is replaced by their constant value

For example, consider the *iota* method

```
int f() {
    int i = 1;
    int j = 2;
    int k = i + j + 3;
    return k;
}
```

and the corresponding HIR code

```
  B0 (pred: [], succ: [B1], locals: [?, ?, ?]):

  B1 (pred: [B0], succ: [], locals: [I0, I1, I4]):
  I0: ldc 1
  I1: ldc 2
  I2: I0 + I1
  I3: ldc 3
  I4: I2 + I3
  I5: ireturn I4
```

The instruction `I0 + I1` at `I2` can be replaced by the constant `3` and the instruction `I2 + I3` at `I4` can replaced by the constant `6`

Local Optimizations

Local Optimizations

Another optimization one may consider is common subexpression elimination, where we identify expressions that are re-evaluated even if their operands are unchanged

Local Optimizations

Another optimization one may consider is common subexpression elimination, where we identify expressions that are re-evaluated even if their operands are unchanged

Example

```
void f(int i) {
    int j = i * i * i;
    int k = i * i * i;
}
```

can be optimized as

```
void f(int i) {
    int j = i * i * i;
    int k = j;
}
```

Local Optimizations

Local Optimizations

Common subexpressions do arise in places one might not expect them

Local Optimizations

Common subexpressions do arise in places one might not expect them

Example

```
for (i = 0; i < 1000; i++) {
    for (j = 0; j < 1000; j++) {
        c[i][j] = a[i][j] + b[i][j];
    }
}
```

where `a`, `b`, and `c` are integer matrices

Local Optimizations

Common subexpressions do arise in places one might not expect them

Example

```
for (i = 0; i < 1000; i++) {
    for (j = 0; j < 1000; j++) {
        c[i][j] = a[i][j] + b[i][j];
    }
}
```

where `a`, `b`, and `c` are integer matrices

If `a'`, `b'`, and `c'` are their base addresses respectively, then the memory addresses of `a[i][j]`, `b[i][j]`, and `c[i][j]` are `a' + i * 4 * 1000 + j * 4`, `b' + i * 4 * 1000 + j * 4`, and `c' + i * 4 * 1000 + j * 4`; eliminating the common offsets, `i * 4 * 1000 + j * 4`, can save us a lot of computation