CS322 Winter'14 Lab 6 Handout
Generating Better IR Code

Overview

In this lab, students will learn and practice a few optimization techniques that enables an IR-gen to produce better IR code. The focus area are Boolean expressions.

Learning Objectives

Upon successful completion, students will be able to:

- implement short-circuit semantics for Boolean expressions
- implement simple optimizations for If and While statements
- understand control-flow representation for Boolean expressions

1. Short-Circuit Semantics

Most programming languages, including Java, use "short-circuit" semantics for Boolean expressions. Under this semantics, the evaluation of a Boolean expression stops as soon as its value can be determined. For example, for the expression

```
true || (a < b)
```

the evaluation will stop after the first operand is evaluated, since the expression's value will be true, regardless of what's remaining in the expression. However, for the expression

```
true && false && (a < b)
```

the evaluation will go on after the first operand is evaluated, since the expression's value can' be determined yet. After the second operand is evaluated, the evaluation can stop, since now, the expression's value is certain to be false. A quick observation from these examples is that when to stop the evaluation process is dependent on the type of operator.

IR0Gen's Approach

IR0Gen's template for logical operations is shown below:

```
// Ast0.Binop ---
// Ast0.BOP op;
// Ast0.Exp e1,e2;
//
// Template for arith and logical op:
//   newTemp: t
//   code: e1.c + e2.c
//         + "t = e1.v op e2.v"
//
```

Both sub-expressions `e1` and `e2` are always evaluated. This implementation is not compatible with short-circuit semantics.

Implementing Short-Circuit Semantics

We'd like to re-implement IR-gen for logical operations to support short-circuit semantics. We'll devise the new IR templates first. As noted above, it's necessary to separate `||` and `&&`. The template for `||` is shown below:

```
// Template for "e1 || e2":
//   newTemp: t
//   newLabel: L
//   code:
//     "t = true"
//     + e1.c
//     + "if e1.v == true goto L"
//     + e2.c
//     + "if e2.v == true goto L"
//     + "t = false"
//     + "L:"
```

The idea is that if `e1` evaluates to `true`, then `e2` will not be evaluated.

Exercise: Try to come up with a parallel template for `e1 && e2`.

Question: To support short-circuit semantics, do we need to change IR-gen code for any other operators?

Exercise: The second step of this practice is to translate the new templates into real IR-gen code. Copy `IR0Gen.java` to `IR0Gen2.java`. Edit the new program to implement the new templates. (Don't forget to rename all `IR0Gen` references to `IR0gen2`.)

2. Better IR Code for If and While

Here are IR0Gen's templates for the If and While statements:

```
// Ast0.If ---
// Ast0.Exp cond;
// Ast0.Stmt s1, s2;
//
// Template:
//   newLabel: L1[,L2]
//   code: cond.c
//         + "if cond.v==false goto L1"
//         + s1.c
//         [+ "goto L2"]
//         + "L1:"
//         [+ s2.c]
//         [+ "L2:"]
```

```
// Ast0.While ---
// Ast0.Exp cond;
// Ast0.Stmt s;
//
// Template:
//   newLabel: L1,L2
//   code: "L1:"
//         + cond.c
//         + "if cond.v==false goto L2"
//         + s.c
//         + "goto L1"
//         + "L2:"
//
```

In both cases, the IR code evaluates the `cond` expression first; it then compares the value against `false` to decide whether to jump or to fall through. Look at one actual example:

```
# Ast0:
If (Binop > n 0)
 Assign x 1
```

```
# IR0 Program
 t1 = true
 if n > 0 goto L1
 t1 = false
L1:
 if t1 == false goto L0
 x = 1
L0:
```

For a simple AST statement, the IR code generated by IR0Gen looks quite complicated: it contains one temp, two labels, and two jumps. In one execution scenario, five instructions will be executed; in another, two jumps will happened. In comparison, the following IR code is a better version for the statement:

```
if n <= 0 goto L0
 x = 1
L0:
```

It has fewer labels and jumps and uses no temp. In any execution scenario, only one or two instructions will be executed.

This example is not an isolated case. It is very common to have a relational expression such as `i < 0` or `a != b` to appear as the `cond` expression in an If or While statement.

An Ad Hoc Optimization

We can make a small change in the IR0Gen routines for If and While to generate the better version of IR code shown above. The idea is to perform a simple look-ahead on the `cond` expression, *before* calling the gen routine on it. If `cond` happens to be a relational expression, then we take advantage and embedded it directly in a `CJUMP`.

Exercise: Implement this change in `IR0Gen2.java`.

3. Introduction to Control-Flow Representation

The ad hoc technique described above does not generalize. Consider the following example:

```
# Ast0:
If (Binop || (Binop > a 0) (Binop > b 0))
 Assign x 1
Else
 Assign x 2
```

IR0Gen (with short-circuit semantics and the above ad hoc optimization) generates the following code:

```
# IR0 Program                     L2:
 t3 = true                         if t2 == true goto L3
 t1 = true                         t3 = false
 if a > 0 goto L1                 L3:
 t1 = false                        if t3 == false goto L0
L1:                                x = 1
 if t1 == true goto L3             goto L4
 t2 = true                        L0:
 if b > 0 goto L2                  x = 2
 t2 = false                       L4:
```

It uses three temps and has lots of jumps. The two relational operations in the AST program are berried deep in the Boolean expression, and the ad hoc optimization has no effect on them.

Can we generate better IR code for this AST program? The answer is yes. Using an alternative approach, called *control-flow representation*, we can generate the following code:

```
# IR0 Program                     L1:
 if a > 0 goto L1                  x = 1
 goto L0                           goto L2
L0:                               L3:
 if b > 0 goto L1                  x = 2
 goto L3                          L2:
```

In this version, there is not a single temp! The Boolean expression is implemented with only labels and jumps. When a sub-expression is evaluated, the result is realized by jumping to the right target.

How Does It Work?

Let's put the AST and IR code side-by-side, in the order of the IR instructions:

```
AST                 Desired IR
-----------------------------------------------------
(Binop > a 0)       if a > 0 goto L1
                    goto L0

(Binop ||           L0:                 Boolean expr

(Binop > b 0)       if b > 0 goto L1
                    goto L3
-----------------------------------------------------
If.1                L1:
Assign x 1          x = 1
If.2                goto L2
IF.3                L3:                 Other code
Assign x 2          x = 2
If.4                L2
-----------------------------------------------------
```

We can make the following observations:

- The IR code for the two sub-expressions `(Binop > a 0)` and `(Binop > b 0)` have the same structure — the code evaluates the expression and jumps to two separate targets depending on the expression's value.
- The target labels for the jumps in the Boolean expressions are all pointing to future instructions. This means when generating IR code for these expressions, the actual label values are not available. The jumps' targets are left as blank "holes". Later on, when more information becomes available, these holes are patched with actual labels.
- The IR code for the `||` operation consists of just a label. There is no other instructions. What is not explicitly shown is that the IR code for `||` actually serves to connect the two sub-expressions' code, honoring short-circuit semantics.

To understand the control-flow representation approach, we present the templates for the If and While statements and Boolean operations below.

First, we introduce two attributes to Boolean expression `E`:

`E.t` — position to jump to when `E` evaluates to true;
`E.f` — position to jump to when `E` evaluates to false.

```
// Ast0.If ---                           // Ast0.While ---
// Ast0.Exp cond;                        // Ast0.Exp cond;
// Ast0.Stmt s1, s2;                     // Ast0.Stmt s;
//                                       //
// Template:                             // Template:
//   newLabel: L1,L2[,L3]                //   newLabel: L1,L2,L3
//   code: cond.c                        //   code: "L1:"
//         + "L1:"                       //         + cond.c
//   -- cond.t = L1 -- back patching     //         + "L2:"
//         + s1.c                        //   -- cond.t = L2 -- back patching
//         [+ "goto L3"]                 //         + s.c
//         + "L2:"                       //         + "goto L1"
//   -- cond.f = L2 -- back patching     //         + "L3:"
//         [+ s2.c]                      //   -- cond.f = L3 -- back patching
//         [+ "L3:"]                     //
```

Analysis: After the IR code for the `cond` expression is generated, the code will contain two sets of label holes in it. One set for going to a "true" target, the other for going to a "false" target. In the If statement case, the true target is realized by the label `L1` (for pointing to the then-clause), and the false target is realized by the label `L2` (for pointing to the else-clause). The two sets of label holes hence can be patched with `L1` and `L2`, respectively. The case for the While statement is similar.

```
// e -> e1 op e2
//
// Template for relop:
//   code: e1.c + e2.c
//         + "if e1.v relop e2.v goto e.t"  -- e.t needs to be patched
//         + "goto e.f"                     -- e.f needs to be patched
//
```

Analysis: The IR code for a relation operation contains two label holes, both need to be patched later.

```
// Template for "e1 || e2":
//   newLabel: L
//   code: e1.c
//         + "L:"
//   -- e1.t = e.t -- adding patching points to parent's list
//   -- e1.f = L   -- back patching
//         + e2.c
//   -- e2.t = e.t -- adding patching points to parent's list
//   -- e2.f = e.f -- adding patching points to parent's list
```

Analysis: According to short-circuit semantics, if `e1` evaluates to false, `e2` has to be evaluated. This means that the starting label for `e2`, `L`, should be used to patch those label holes in `e1` that are for going to a false target. On the other hand, the other set of label holes in `e1` can not be resolved yet; they are merged to the parent's set for later patching. The same is true for `e2`'s label holes.

```
// Template for "e1 && e2":
//   newLabel: L
//   code: e1.c
//         + "L:"
//   -- e1.t = L   -- back patching
//   -- e1.f = e.f -- adding patching points to parent's list
//         + e2.c
//   -- e2.t = e.t -- adding patching points to parent's list
//   -- e2.f = e.f -- adding patching points to parent's list
```

Analysis: Similar to the above case.

Exercise: Use the templates to manually generate IR code for the AST example program:

```
# Ast0:
If (Binop || (Binop > a 0) (Binop > b 0))
 Assign x 1
Else
 Assign x 2
```