Advanced Compiler Design

Spring Semester 2016

Homework 2

Due date: Apr 15, 2016, 23:59

1 Introduction

The objective of this homework is to use the SSA intermediate representation developed in the last homework to implement optimizations. You can use the compiler that you developed for Homework 1 as your base compiler, or you can start with the skeleton that we provide (highly recommended).

You have to implement three optimizations and one program analysis. For all tasks, you **do not** need to implement these optimizations or the analysis for array items, and fields. These operations may be left unoptimized.

While not mandatory, we encourage you to think about how far you can go with the optimizations. For example, you can perform some strength reductions during constant folding. But above all, the optimizations must not change the result of a program (including the exit code for dynamic errors, e.g., division by zero).

Note that the testing framework creates an `*.opt.ref` file from the reference solution to compare against your `*.opt` file. While it shows the number of operations that our reference executed for the test case, you may have different numbers, since you applied optimizations differently. In that case, just add your own `.opt.ref` file with your solution to make the test pass.

2 Optimizations

Optimizations should be implemented in the class `cd.transform.Optimizer`. You also want to perform these optimizations repeatedly until the program reaches a steady state in order to fully capture all optimization opportunities.

2.1 Constant folding

Constant folding attempts to evaluate expressions at compile time. Consider this code sequence:

```
int a, b, c, d;

a = 1;
b = 2;
c = 1 + 2;
d = a + b;
```

With constant folding, the compiler will discover that `1 + 2` can be evaluated at compile time to `3` and will generate code for the assignment `c = 3`.

Constant folding alone does not allow the compiler to discover that `d = a + b` can evaluate to `3` at compile time. To fully exploit all the optimization opportunities in a program, the compiler can combine constant folding with other optimizations such as copy propagation.

2.2 Copy propagation

This optimization attempts to eliminate variables whose definition is a copy of another variable or a constant. This optimization often sets the ground for successful identification (and elimination) of common subexpressions and for constant folding.

Consider this code sequence:

```
int w, x, y, z;

y = ...;
x = y;
w = 3;
z = x;
write(w);
```

All uses of `x` can be replaced with uses of `y`, and similarly all uses of `w` can be replaced with the constant 3. Another source of useless copy operations can be a phi node where all operands have the same value, such as:

$$\mathtt{u}_2 = \phi(\mathtt{y}_0, \mathtt{y}_0)$$

Such phi nodes would not be generated initially, but can result from other optimizations. In this case, uses of the variable u_2 could also be replaced with y_0. Once all uses of a variable have been replaced, it is also safe to eliminate the assignments to that variable entirely. (Note that you can only apply copy propagation to local variables whose full set of uses can be statically enumerated within one method.)

The end result of applying copy propagation to the above code fragment would be something like the following:

```
int w, x, y, z;

y = ...;
// x = y; (unnecessary)
// w = 3; (unnecessary)
z = y; // x changed to y
write(3); // w changed to 3
```

2.3 Common subexpression elimination

Common subexpression elimination (CSE) replaces redundant recalculation of expressions with caching and thereby prevents re-evaluation of these expressions. Consider this code sequence

```
int a, b, c, d;

a = c+d;
...
b = c+d;
```

This code would be transformed (provided that there is no intervening assignment to "c" or "d") as follows:

```
int a, b, c, d, t1;

t1 = c+d;
a = t1;
...
b = t1;
```

It is not necessary to handle all possible common subexpressions, but only those with the same AST tree "shape". For example, two expressions `a+b` and `b+a` should be detected. However, common subexpressions like `(a+b)+c` and `a+(b+c)` do not need to be detected, because their AST structure is different. Be careful with non-commutative operators and associativity in general. For example, the operator − for which the order of the operands is significant (i.e., `a-b` and `b-a` are not equivalent).

3 Analysis

We suggest performing the uninitialized variable analysis after SSA generation but before optimization. A `ToDoException` has been added to `cd.transform.SSA`.

3.1 Uninitialized variables

You should generate a semantic error whenever a (potentially) uninitialized local variable is used as an operand. Your analysis should be as precise as possible and avoid generating spurious errors. For example, be careful not to generate errors when the only use of an uninitialized variable is as an operand of a (dead) phi node.