Advanced Compiler Design
Spring Semester 2013
Assignment 2

Due date: Apr 10, 2013, 23:59

1 Introduction

The objective of this assignment is to use the SSA intermediate representation module developed in the last assignment to implement optimizations. You can use the compiler that you developed for Assignment 1 as your base compiler, or you can start with the skeleton that we provide.

Please implement three optimizations and one program analysis. In all cases, you do not need to implement these optimizations or analysis for arrays – your code must handle scalars but should of course not crash if arrays, object instances, or other constructs that go beyond scalars show up in the program.

2 Optimizations

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

2.1 Constant folding

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

```java
int a, b, c;

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

With constant folding, the compiler will discover that “a+b” can be evaluated at compile time to “3” and will generate code for the assignment “c = 3” (and continue to use this value for “c” at other appropriate places).
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.

Consider this code sequence:

```c
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:

```c
u = φ(y, y)
```

Such phi nodes would not be generated initially but can result from other optimizations. In this case, uses of the variable `u` could also be replaced with `y`. 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:

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

y = ...;
...
// x = y; (removed)
...
// w = 3; (removed)
...
// w changed to 3
```

2.3 Common subexpression evaluation

Common subexpression evaluation (CSE) replaces totally redundant expressions and thereby saves a 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-associative operators or floating point operators like $-$ where 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.cfg.SSA.

3.1 Uninitialized variables

You should generate a semantic error whenever a (potentially) uninitialized variable is used as an operand. Your analysis should be as precise as possible and avoid generating spurious errors except when there is a use of a local variable which may not yet have been initialized. Be careful not to generate errors when the only use of an uninitialized variable is as a phi-operand.