Midterm Exam
CS6013

27-Sep-2013

Read the questions carefully. State all your assumption clearly. There are total 4 questions * 15 marks = 60 marks. For questions that have sub-parts, the division for the sub-parts are mentioned in square brackets.

1. **Semantic Analysis** Present a scheme to type check and generate IR for programs written in an extension of Minijava that admits type casting. If the cast fails the program aborts, which is equivalent to executing the instruction `error` in the IR.

```
class A{ int f1; }
class C { }
class  B extends A{
  boolean f1;
  void foo(){
    A a1, a2;
    B b1, b2;
    C c1;

    a1 = new B();
    b1 = a1; // type error
    b1 = (B) a1; // runtime check, succeeds

    a2 = new A();
    b2 = (B) a2 // runtime check, error

    c1 = (C) a1; // type error

    a1.f1 = ((A) new B()).f1; // type checks
    b1.f1 = ((B) a2).f1; // type checks, runtime error
  } }
```

Use your scheme to generate intermediate code for the above example (generate intermediate code by ignoring all the statements that result in type error. Use an imperative intermediate code, something like miniIR). Be liberal with your comments to help us understand your translation.

2. **Control flow analysis** Write an algorithm to construct a Depth-First-Spanning-Tree (DFST) for a control flow graph (CFG) [2].

An edge (m, n) in the input CFG is called a *retreating* edge if n is an ancestor of m in the DFST. A flow graph is reducible, if it can be transformed into a single node, by repeated application of T1/T2 transformations. An alternative definition of a flow graph being reducible is that all its retreating edges in any DFST are also back edges. Prove that a flow graph is reducible iff when we remove all the back edges, the resulting flow graph is acyclic [6.5]. Prove that both the definitions of reducibility are equivalent [6.5].

3. **Dead Code Elimination** A variable definition is *dead* if it is not used on any path from the definition point to the exit node. An instruction is *dead* if it assigns to only dead variables. Write an algorithm to eliminate dead code from the given program.

Assume: two data structures `UD` (for use-def) and `DU` (for def-use) are available for each variable. For a variable x, `UD`$[x][i]$ returns the set of definitions of x that reach the instruction i that uses x; if instruction i does not use x then `UD`$[x][i]$ returns null. For a variable x, `DU`$[x][i]$ returns the set of uses of x that use the value defined at the instruction i; if instruction i does not define x then `DU`$[x][i]$ returns null.

4. **Copy Propagation** Present an algorithm to do global *copy propagation* [10]. Given an assignment `x := y`, the copy propagation optimization, replaces each later use of `x` with `y`, as long as there is no redefinition of `x` or `y` in between. Assume a language that allows global variables (but no locals), allows function calls that take no arguments, but may return a value.
Apply your algorithm on the code given below [4].

```
int a, b, c, d, e, f, g, h, i;   bar(){
main(){                               f = a + c;
     // read a, b                     g = e;
     c = a + b;                       a = g + d;
     d = c;                           if (a < c) {
     e = d * d;                             h = g + 1;
     i = bar();                             fbar();
     j = i * i;                       }
     print (a,b,c,d,e,f,g,h,i,j);     else {
}                                       f = d - g;
                                        if (f > a)
                                          fbar();
fbar(){                                 else
   b = g * a;                             c = 2;
   if (h < f) foo();                  }
}                                     return f; }
```

As an example, copy propagation should replace `e=d*d` with `e=c*c`, and `j=i*i` with `j=f*f` among others. Discuss two uses of copy propagation [1].