Assignment 5: Tuples and More Optimizations

15-411: Compiler Design
Nathan Snyder (npsnyder@andrew) and Anand Subramanian(asubrama@andrew)

Due: Tuesday, November 2, 2010 (1:30 pm)

Reminder: Assignments are individual assignments, not done in pairs. The work must be all your own.

You may hand in a handwritten solution or a printout of a typeset solution at the beginning of lecture on Tuesday, November 2. Please read the late policy for written assignments on the course web page.

Problem 1: Tuples (30 points)

Tuples are a language feature common in functional programming languages. Suppose we wanted to give C0 programmers the ability to declare tuples using an SML-like types, assign values to them, pass them as parameters to functions, and use them as the return values. While individual tuple elements can be read, it is not permissible to assign only to a single element. Additionally, the types used to construct a tuple type must all be small types, so a tuple of an int and a struct is invalid. We will use the @ symbol to join multiple types into a tuple type, and array-like bracket syntax to reference the 0-indexed elements of a tuple. As an example, the following code would return 11 from main.

```
int@int inc_tuple (int@int x)
{
    return (x[0] + 1, x[1] + 1);
}

int main()
{
    int@int y = (0, 5);
    int@bool@bool *z = alloc(int@bool@bool);
    y = inc_tuple(y);
    *z = (5, true, true);
    return z[0] + y[1];
}
```

(a) Describe how you would handle parsing the tuple syntax given here.

(b) Describe the new typing rules that would be needed to handle tuples.

(c) Describe how you would compile tuples. Be sure you discuss all of the operations that need to be supported.

Problem 2: Alias Analysis (10 points)

For this problem, consider the program below

```
1: int main()
2: {
3:    int *x = alloc(int);
4:    int *y;
5:    int z;
6:    bool *a = alloc(bool);
7:    *a = false;
8:    if (*a)
9:    {
10:         y = x;
11:         *x = 8;
12:         z = *y;
13:         return z;
14:   }
15:   else
16:   {
17:         y = alloc(int);
18:         *y = 6;
19:         z = *y;
20:         return z;
21:   }
22: }
```

(a) Give the saturated *points* predicate, described in the lecture notes, for this program.

(b) Explain why the assignment on line 12 can't be optimized to a constant assignment using our current analysis. How could our alias analysis be modified to allow this? Just give the general idea - you don't need to define new inference rules or anything.

(c) How can line 19 be optimized? Explain how the situation here differs from the situation at line 12.

Problem 3: Inlining (20 points)

Inlining is the technique of replacing a call to some function foo by inserting the body of foo into the calling function.

(a) What are the advantages and disadvantages of inlining?

(b) Given your answer to part (a), what criteria would you use to decide which call sites to inline in a program? Your criteria should be precise enough that a compiler (as opposed to a programmer) would be able to check them.

(c) An important consideration for implementing optimizations is when to apply them in the compilation process. If you were implementing inlining, would you apply it at the AST, the IR, or the post-code-generation instructions level? Explain your decision, and also list any advantages of the other options that you considered.