CS412 Homework 3 Due: 13 March 2006

1. For each of the following IC constructs, state whether it is well-typed in some typing context. If so, give the *most general* typing context in which the construct is well-typed and write the corresponding proof tree. If the construct is not well-typed in any type context, explain why.

   (a) `(new int[y])[x[2].length]`

   (b) `((a + "b") == c) || (a == (b - c))`

   (c) `if (x == a[b[x]] && y) y = b[c[x]];`

   (d) `f(y,g(x)) = g(f(x.length,null))`

2. This problem concerns the typing and the translation of for loop constructs.

   (a) Suppose we extend IC to support for loops that operate over ranges of integer values:

   $$\texttt{for } (\texttt{x from } E_1 \texttt{ to } E_2)\ \texttt{S}$$

   where `x` is an integer field or local variable (that has been declared before), and E_1, E_2 are the loop bounds. The loop body is executed for all values of `x` in the range from E_1 to E_2. Write a typing rule that ensures the safe execution of this loop.

   (b) It is difficult to reason about for loops when the execution of the loop body might change the iteration variable or the loop bounds. Describe a semantic check that would ensure this never happens.

   (c) Suppose we further augment IC with extended for loops, in a fashion similar to those in Java 1.5. An extended for loop in IC will have the following form:

   $$\texttt{for } (T\ \texttt{x}\ :\ E)\ S$$

   Here, `x` is a newly declared local variable of type T that iterates over all of the elements of the array E, and S is the loop body. Write an appropriate typing rule for this construct.

   (d) Write a syntax-directed translation for the extended for loop into three-address code.

3. Consider a C-like language that manipulates pointers. Statements and expressions have the following syntax:

   $$\begin{array}{lcl} E & ::= & n \mid \texttt{x} \mid \texttt{\&x} \mid *E \\ S & ::= & \texttt{x} = E \mid \texttt{x} = \texttt{malloc()} \mid *\texttt{x} = E \end{array}$$

where n is an integer constant, `x` is a variable, and `malloc()` allocates an integer or a pointer on the heap (according to the declared type of `x`), and then returns a pointer to that piece of data. The only types are pointers and integers, but pointers can be multi-level pointers. The syntax for types is:

$$T ::= \texttt{int} \mid T*$$

(a) Write typing rules for all of the assignment statements. Use judgments of the form $A \vdash s$ for statements, and judgments of the form $A \vdash s : T$ for expressions.

Now let's extend the types in this language with two type qualifiers `taint` and `trust`. Tainted data represents data that the program received from external, untrusted sources, such reading from the standard input or reading from a network socket. All of the other data is `trust`'ed.

We extend the set of statements with a `read()` statement that reads an untrusted integer value from an external source:

$$E \ ::= \ \ldots \mid \texttt{read()}$$

The syntax for qualified types is:

$$\begin{array}{lcl} T & ::= & QR \\ R & ::= & \texttt{int} \mid T* \\ Q & ::= & \texttt{taint} \mid \texttt{trust} \end{array}$$

For instance, `trust ((taint int) *)` represents a trusted pointer to a tainted location, and `taint ((taint int) *)` denotes a tainted pointer to a tainted location.

b) Write appropriate typing rules for expressions n, `&x`, `*x`, and `read()` for programs with qualified types.

We want to prohibit the flow of values from untrusted sources into trusted portions of the memory. However, we want to allow flows of values from trusted locations to tainted locations. We can achieve this by defining an appropriate subtyping relation $\leq$ between qualified types. Fist, we define an ordering between qualifiers: $Q \preceq Q'$ iff $Q = \texttt{trust}$ or $Q = Q'$. We then use the subtyping rule:

$$\frac{Q \preceq Q'}{QR \leq Q'R} \ [\textsc{Subtype}]$$

along with the standard assignment rule in the presence of subtyping:

$$\frac{A \vdash x : T \quad A \vdash E : T' \quad T' \leq T}{A \vdash x = E} \ [\textsc{Assign}]$$

to enforce the desired control over trusted values. For instance, these rules would make it possible to type-check this code fragment:

```
taint int x;
trust ((trust int) *) y;
y = malloc();
x = *y;
```

c) Prove that the above program type-checks by showing the proof trees for each of the two assignments.

d) Write the remaining rule for indirect assignments $*\texttt{x} = E$.

e) Consider the following, more general subtyping rules:

$$\frac{Q \preceq Q'}{Q\texttt{int} \leq Q'\texttt{int}}\ [\textsc{Subtype1}] \qquad \frac{Q \preceq Q' \quad T \leq T'}{Q(T*) \leq Q'(T'*)}\ [\textsc{Subtype2}]$$

Are these rules sound? If yes, argue why. If not, show a program fragment that type-checks, but yields a type error at run-time.