Quiz 2

Compiler Construction, Fall 2014

Wednesday, December 10th, 2014

General notes about this quiz

- Have your CAMIPRO card ready on the desk.
- You are allowed to use any printed material (using standard fonts, no cursive or script fonts) that you brought yourself to the exam. You are not allowed to use any notes that were not typed-up. Also, you are not allowed to exchange notes or anything else with other students taking the quiz.
- **Use separate sheets, for each question, to write your answers**. No sheet of paper should contain answer to two or more questions at the same time.
- Make sure you write your name on each sheet of paper.
- Use a permanent pen with dark ink.
- It is advisable to do the questions you know best first.
- You have in total **2 hours 50 minutes**.

Exercise	**Points**	
Total	0	

Problem 1: Code Generation for Guarded Command Language (30 points)

Edsger Dijkstra(1930 - 2002) proposed a language named the Guarded Command Language which has two interesting constructs: a *simultaneous assignment statement* and a *non-deterministic* loop statement. The simultaneous assignment statement has the form: $x_1, x_2, \cdots, x_n := e_1, e_2, \cdots e_n$, where x_i's are variables and e_i's are expressions. The statement first evaluates the right-hand-side expressions $e_1, \cdots, e_n$ in the same order and assigns the value of e_i to the variable x_i. The non-deterministic loop statement has the form:

do
  g_1 −> s_1
  ⋮
  g_n −> s_n
od

where $g_1, \cdots, g_n$ are guards i.e, boolean valued expressions and $s_1, \cdots, s_n$ are assignment statements. The loop iterates as long as there is at least one guard that evaluates to true. In each iteration, it *non-deterministically* chooses one guard, say g_i, that evaluates to true and executes the statement corresponding to the guard, namely s_i. The loop exits if none of the guards evaluate to true. In addition to the above two statements, assume that the language has a boolean expression of the form $x > y$, and an arithmetic expression of the form $x - y$, where x and y are variables.
Figure **??** shows two programs written in the guarded command language. The program in Figure **??**(a) implements the Euclidean algorithm for computing GCD (greatest common divisor) of two positive integers, and the program in Figure **??**(b) implements a bubble sort like algorithm for sorting three number.

```
do
  x > y -> x :=  x - y
  y > x -> x,y := y,x
od
```
(a)

```
do
  x > y -> x,y,z := y,x,z
  y > z -> x,y,z := x,z,y
od
```
(b)

Figure 1: (a) a program that computes GCD of positive integers, and (b) a program that sorts three numbers

In this exercise, we will consider two deterministic implementations of the non-deterministic loop statement.
Strategy 1: choosing the first true guard
In this strategy, we fix that the loop statement always chooses the *first* guard that evaluates to true and executes the statement corresponding to it, similar to a pattern matching statement in Scala. The loop exits when none of the guards evaluate to true. Note that in every iteration the loop executes exactly one case.

- **a)** **[10 pts]** Provide a *destination passing style* translation for the loop statement that implements the strategy 1. Use the *branch* function described in the lectures in your translation. You need not show the definition of the branch function. Also use $[s_i]$ to denote the translation of a statement s_i. You need not provide a translation for the statement s_i.

$$\begin{array}{l}\textsf{[do} \\ \quad g_1 \;{-}{>}\; s_1 \\ \quad \vdots \\ \quad g_n \;{-}{>}\; s_n \\ \textsf{od] lafter}\end{array} \qquad = ???$$

where, `lafter` is the label of the statement that should be executed when the loop exits.

b) [**10 pts**] Use the translation that you designed for the previous question, and the standard translation for other statements in the language described in the lectures, to generate Java byte code for the program shown in Figure **??**(a) that computes the GCD of two numbers. It suffices to show the final byte code generated for the program. It is not necessary to show the intermediate steps.

For you reference, we have provided a list of byte code instructions that you may need for this exercise at the end of this question.

Strategy 2: Round Robin

In this strategy, in each iteration, the loop must execute every case that evaluates to true in the same order as they appear in the code. In other words, in every iteration, we first check if g_1 evaluates to true, if it does we execute s_1, then we check if g_2 evaluates to true and execute s_2 if it does, and proceed similarly.

c) [**10 pts**] Provide a *destination passing style* translation for the loop statement that realizes the strategy 2. Your translation must exit the loop *immediately* after finding that every guard evaluates to false and should not perform any redundant evaluation of guards. As before, use the *branch* function described in the lectures, and $[s_i]$ to denote the translation of the statement $[s_i]$.

Java byte code instructions

iload_#x	Loads the integer value of the local variable x on the stack.
iconst_x	Loads the integer constant x on the stack.
istore_#x	Stores the current value on top of the stack in the local variable in x
iadd	Pop two (integer) values from the stack, add them and put the result back on the stack.
isub	Pop two (integer) values from the stack, subtract them and put the result back on the stack.
ifXX L	Pop one value from the stack, compare it zero according to the operator XX. If the condition is satisfied, jump to the instruction given by label L. XX $\in$ { eq, lt, le, ne, gt, ge, null, nonnull }
if_icmpXX L	Pop two values from the stack and compare against each other. Rest as above.
goto L	Unconditional jump to instruction given by the label L.

Problem 2: Type Checking For Immutable Maps (50 points)

Consider a language that has only strings and maps. The keys of the maps are always strings but their values could be strings or other maps. The syntax of the language and its types is given by the following grammar:

$$\begin{array}{lcl} expr & \rightarrow & \text{“}\texttt{strcons}\text{”} \\ & | & \texttt{let ident} = expr \text{ in } expr \\ & | & \texttt{empty}[T] \\ & | & \texttt{put}(expr, expr, expr) \\ & | & \texttt{get}(expr, expr) \\ T & \rightarrow & \texttt{string} \mid \texttt{Map}[\texttt{string}, T] \end{array}$$

In the above grammar, `strcons` is a set of string constants, `ident` is a set of identifiers. The statement `let id` $= e_1$ in e_2 creates a new local variable `id`, initializes it to the result of e_1, and evaluates the expression e_2 that may use the local variable `id`. Note that the scope of `id` is restricted to e_2. In other words, the `let` statement is equivalent to the scala code:
`{ val id =` e_1`;` e_2 `}`. The functions `empty`, `put` and `get` are operations involving maps and are described below:

- `empty`$[T]$ creates an empty map from string to the type T.
- `put`(e_1, e_2, e_3) takes a map e_1, a string e_2, and a value e_3, and returns a new map that is same as e_1 for all keys except for the key e_2 which is mapped to e_3.
- `get`(e_1, e_2) takes a map e_1 and a string e_2, and returns the value corresponding to the key e_2. If the key does not have a mapping in the map e_1, the function raises a `KeyNotFound` exception.

The following are some type rules for the language:

$$\frac{s \in \texttt{strcons}}{\vdash \text{“}s\text{”} : \texttt{string}} \qquad \vdash \texttt{empty}[T] : \texttt{Map}[\texttt{string}, T]$$

a) **[10 pts]** Give a set of type rules for the map operations `put` and `get` that is consistent with the above description of `put` and `get` operations. For this part, you can assume that throwing an exception is an acceptable outcome of an evaluation and need not treat it as a crash or bad behaviour. You can also assume that the return type of `get` is the type of the values of the map passed as the first argument.

Consider the following expression belonging to the language. We will refer to this expression as E.

```
let m1 = put(empty[string],"x","z") in
  let m2 = put(empty[Map[string,string]],"y", m1) in
    let m3 = put(m2,"z",empty[string]) in
      get(m3,"y")
```

b) **[10 pts]** Give a type rule for the `let` statement that is consistent with its description. What is the type of the expression E under your type rules ? Will the expression type check if we change the body of the last `let` to `get(get(`$m3$, “y”), “z”) ?

Say we want to extend the type system so that expressions that type check do not throw a `KeyNotFound` exception. For this purpose, we augment the type of maps with a *set* of keys that the map must contain. Consider the following modification to the types of our language:

$$\begin{array}{rcl} T & \rightarrow & \texttt{string} \mid \text{“}\texttt{strcons}\text{”} \\ & \mid & \texttt{Map}[\texttt{string}, T]S \\ S & \rightarrow & \{\text{“}\texttt{strcons}\text{”}, \cdots, \text{“}\texttt{strcons}\text{”}\} \end{array}$$

If an expression is typed as $\texttt{Map}[\texttt{string}, T]\{\text{“}str1\text{”}, \text{“}str2\text{”}, \cdots, \text{“}strn\text{”}\}$, it implies that the result of the expression is a map from string to T and that it *must* have a mapping for the set of keys { “str1”, “str2”, $\cdots$, “strn” }. For example, the above extension allows us to type an expression $\texttt{put}(\texttt{empty}[\texttt{string}], \text{“}a\text{”}, \text{“}b\text{”})$ as $\texttt{Map}[\texttt{string}, \texttt{string}]\{\text{“}a\text{”}\}$, which conveys that the result of the expression is a map from `string` to `string` and also that it has a mapping for the key “a”.

c) **[10 pts]** Provide a sub-typing rule for the map type under this new extension.

$$\frac{???}{\texttt{Map}[\texttt{string}, T_1]S_1 <: \texttt{Map}[\texttt{string}, T_2]S_2}$$

d) **[20 pts]** Adapt the type rules of the language to the extended types that store the keys of the maps as a part of the type. You can assume that a sub-typing relation exist between the map types. Your type rules should ensure that type correct expression can never throw `KeyNotFound` Exception.

For example, in your type system, the expression $\texttt{get}(\texttt{put}(\texttt{empty}[\texttt{string}], \text{“}a\text{”}, \text{“}b\text{”}), \text{“}c\text{”})$ should not type check, whereas $\texttt{get}(\texttt{put}(\texttt{empty}[\texttt{string}], \text{“}a\text{”}, \text{“}b\text{”}), \text{“}a\text{”})$ and the expression E (shown above) should type check.