End-of-Year Quiz

Compiler Construction, Fall 2013

Wednesday, December 18th, 2013

Last Name : ______________________________

First Name : ______________________________

Camipro Number : ______________________________

Problem	**Points**	**Rating**
1	/20	☆☆☆☆☆
2	/10	☆☆☆☆☆
Total	/30	

General Instructions for This Quiz

- You have in total **3 hours 45 minutes**.
- Have your CAMIPRO card ready on the desk.
- You are allowed to use any printed material 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 the notes with other students taking the quiz.
- Write the answer of each question on a **separate sheet**, and not on the quiz question sheets. Write your name on each sheet containing your answers.
- Return the printed question sheets back to us. Please rate each problem according to your interest (0 stars: nothing to say. 1-star: Uninteresting problem. 5 stars: Truly awesome problem)
- Use a permanent pen.
- We advise you to do the questions you know best first.
- You will perhaps discover that questions about type systems take longer to understand, but not necessarily longer to solve.

Problem 1: Type Rules for Collections (20 points)

Consider the following typed language on immutable identifiers:

$$expr ::= \textbf{val}\ ident\ \texttt{=}\ expr\textbf{;}\ expr \qquad \text{Variable binding} \tag{1}$$
$$expr ::= i \qquad \text{where i is an integer constant} \tag{2}$$
$$expr ::= (ident : T) \Rightarrow expr \qquad \text{Creates an anonymous function} \tag{3}$$
$$expr ::= expr(expr) \qquad \text{Applies a function} \tag{4}$$
$$T ::= Int \qquad \text{Int type} \tag{5}$$
$$T ::= T \Rightarrow T \qquad \text{Function type} \tag{6}$$
$$T ::= Collection[T] \qquad \text{Collection type} \tag{7}$$

For the purpose of this exercise, we add a polymorphic type $Collection[T]$ to the language, where T can be any existing type. We extend the expression syntax with existing customized function symbols:

$$expr ::= \text{EmptyCol}[\alpha] \qquad \text{Creates an empty collection of elements of type } \alpha \tag{8}$$
$$expr ::= \text{add}(expr, expr) \qquad \text{Adds an element to a collection} \tag{9}$$
$$expr ::= \text{permute}(expr) \qquad \text{Returns a function from a collection} \tag{10}$$
$$expr ::= \text{map}(expr) \qquad \text{Returns a function from a collection} \tag{11}$$
$$expr ::= \text{flatMap}(expr) \qquad \text{Returns a function from a collection} \tag{12}$$

Informally, the semantics of the extensions is the following:

- $\text{EmptyCol}[\alpha]$ (line 8) takes a type parameter and returns an empty collection of this type.
- $\text{add}(e_1, e_2)$ (line 9) takes a collection e_1 and an element e_2 and returns the collection e_1 to which the element has been added.
- $\text{permute}(expr)$ (line 10) takes a collection, and returns a function which is a permutation mapping of the elements of the collection. See example below.
- $\text{map}(expr)$ (line 11) takes a collection and returns a function. This function accepts a mapping from an element to another element and returns the mapped original collection.
- $\text{flatMap}(expr)$ (line 12) takes a collection and returns a function. This function accepts a mapping from an element to a collection of elements and returns the union of all images of elements of the original collection.

For example, if $x = \text{add}(\text{add}(\text{add}(\text{EmptyCol}[Int], 1), 2), 3)$, it could be that:

$$\text{permute}(x)(1) == 2$$

$$\text{permute}(x)(2) == 3$$

$$\text{permute}(x)(3) == 1$$

For any other integer, i, permute(x)(i) could be either 1, 2 or 3.
If collections were sets, we would also have the following (to illustrate):

$$\text{map}(x)(a \Rightarrow 1) = \{1\}$$

$$\text{flatMap}(x)(a \Rightarrow \text{add}(x, a+3)) = \{1, 2, 3, 4, 5, 6\}$$

$$\text{map}(x)(\text{permute}(x)) = \text{map}(x)((i : \mathit{Int}) \Rightarrow \text{permute}(x)(i)) = x$$

The type rules for expressions are the following:

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \quad \frac{\Gamma \vdash e : T \;\; \Gamma \vdash f : T \Rightarrow U}{\Gamma \vdash f(e) : U} \quad \frac{\Gamma \vdash e_1 : T \qquad \Gamma, (x, T) \vdash e_2 : U}{\Gamma \vdash (\text{val } x = e_1;\, e_2) : U}$$

$$\frac{\Gamma \vdash x : T \qquad T <: U}{\Gamma \vdash x : U} \quad \frac{\Gamma \vdash x : T \Rightarrow U \qquad U <: V \qquad W <: T}{\Gamma \vdash x : W \Rightarrow V}$$

We also allow standard subtyping rules, with function results covariant and function arguments contravariant.

a) **[5 pts]** Complete the following type rule templates (replace all ??? occurences) so that they are consistent with the described meaning of the extensions and sufficient to type check the extension if we exclude subtyping. You can abbreviate $\mathit{Collection}[T]$ as $C[T]$. Here is a example rule for flatMap:

$$\frac{\Gamma \vdash e : \mathit{Collection}[T]}{\Gamma \vdash \text{flatMap}(e) : (T \Rightarrow \mathit{Collection}[U]) \Rightarrow \mathit{Collection}[U]}$$

$$\frac{???}{\Gamma \vdash \text{EmptyCol}[\alpha] : ???} \quad \frac{???}{\Gamma \vdash \text{add}(e_1, e_2) : ???} \quad \frac{???}{\Gamma \vdash \text{permute}(e) : ???} \quad \frac{???}{\Gamma \vdash \text{map}(e) : ???}$$

---

solution

$$\frac{}{\Gamma \vdash \text{EmptyCol}[\alpha] : \mathit{Collection}[\alpha]} \quad \frac{\Gamma \vdash e_1 : \mathit{Collection}[T] \quad \Gamma \vdash e_2 : T}{\Gamma \vdash \text{add}(e_1, e_2) : \mathit{Collection}[T]} \quad \frac{\Gamma \vdash e_1 : \mathit{Collection}[T]}{\Gamma \vdash \text{permute}(e) : T \Rightarrow T}$$

$$\frac{\Gamma \vdash e : \mathit{Collection}[T]}{\Gamma \vdash \text{map}(e) : (T \Rightarrow U) \Rightarrow \mathit{Collection}[U]}$$

---

b) **[10 pts]** Prove that the following program type checks by writing the type derivation tree. You can write a separate tree for each right-hand side of variable definition.

```
val x = add(add(EmptyCol[Int], 1), 2)
val z = add(add(x, 3), 4)
val y = add(EmptyCol[Int => Int], permute(z))
flatMap(y)(map(x))
```

---

solution

x typechecks to $Collection[Int]$, z to $Collection[Int]$ and y to $Collection[Int \Rightarrow Int]$ Therefore map(x) typechecks to $(Int \Rightarrow Int) \Rightarrow Collection[Int]$ and flatMap(y)(map(x)) typechecks to $Collection[Int]$

Let us consider the following extended type system used to prevent at compilation time the calling of permute on empty sets. We introduce for each regular collection type $Collection[T]$ the empty-annotated types $Collection^{+}[T]$ and $Collection^{-}[T]$.

- $Collection^{+}[T]$ meaning that the collection may be empty,
- $Collection^{-}[T]$ meaning that the collection may not be empty.

It follows that $Collection^{-}[T] <: Collection^{+}[T]$. This is the only subtyping relation available between collections here.

c) [10 pts] Give sound type rules for this language. Remember, that the types in your rules have to be $Collection^{-}[T]$ or $Collection^{+}[T]$, $Collection[T]$ only appears in the code.

---

solution

$$\frac{}{\Gamma \vdash \text{EmptyCol}[\alpha] : Collection^{+}[\alpha]} \qquad \frac{\Gamma \vdash e_1 : Collection^{+}[T] \quad \Gamma \vdash e_2 : T}{\Gamma \vdash \text{add}(e_1, e_2) : Collection^{-}[T]} \qquad \frac{\Gamma \vdash e_1 : Collection^{-}[T]}{\Gamma \vdash \text{permute}(e) : T \Rightarrow T}$$

$$\frac{\Gamma \vdash e : Collection^{-}[T]}{\Gamma \vdash \text{map}(e) : (T \Rightarrow U) \Rightarrow Collection^{-}[U]} \qquad \frac{\Gamma \vdash e : Collection^{+}[T]}{\Gamma \vdash \text{map}(e) : (T \Rightarrow U) \Rightarrow Collection^{+}[U]}$$

$$\frac{\Gamma \vdash e : Collection^{+}[T]}{\Gamma \vdash \text{flatMap}(e) : (T \Rightarrow Collection^{+}[U]) \Rightarrow Collection^{+}[U]}$$

Problem 2: Code Generation for Switch (10 points)

In the following exercise we consider compilation to a stack machine that uses JVM instructions.
Suppose that we extend our language with a special additional switch-case statement on integers, with a way to reswitch on another integer if needed.

`switch(i) { (case` $(n_k \mid \_) \Rightarrow (e_k \mid$ `reswitch` $(e_k)))^*$ `}`

`switch` executes the code e_k corresponding to the expression n_k when the result of n_k is equal to the result of i, and if nothing matches it executes the default statement introduced by `case _`.

`reswitch`(e_k) is a jumping expression. It computes the value of e_k which should be an integer and re-runs the first outer switch on the resulting value. Its effectively creates a loop.

An example of a switch statement using expressions written as a, b, c, e, f, g, h, i can be the following:

```
switch(i) {
  case a => e
  case b => reswitch(f)
  case c => g
  case _ => h
}
```

a) **[10 pts]** Give the translation of the switch construct above to JVM instructions.

For this question, you can assume a number of 3 `case` statements, a `reswitch` on the second, and one default statement as above. See the available jvm bytecode on the next page. To avoid recomputing i, you might want to duplicate the value for multiple comparisons. Use square brackets $[\![\quad]\!]$ around the generic expressions like i or a to compute them.

---

solution

```
        ⟦i⟧
top     dup
        ⟦a⟧
        if_icmpne case2
        pop
        ⟦e⟧
        goto end
case2   dup
        ⟦b⟧
        if_icmpne case3
        pop
        ⟦f⟧
        goto top
case3   dup
        ⟦c⟧
        if_icmpne default
        pop
        ⟦g⟧
        goto end
default pop
        ⟦h⟧
        goto end
end
```

These are selected bytecode instructions, mostly for integers, for your quick reference.

iload_x	Loads the integer value of the local variable in slot x on the stack. $x \in \{0, 1, 2, 3\}$
iload X	Loads the value of the local variable pointed to by index X on the top of the stack.
iconst_x	Loads the integer constant x on the stack. $x \in \{0, 1, 2, 3, 4, 5\}$.
istore_x	Stores the current value on top of the stack in the local variable in slot x. $x \in \{0, 1, 2, 3\}$
istore X	Stores the current value on top of the stack in the local variable indexed by X.
ireturn	Method return statement (note that the return value has to have been put on the top of the stack beforehand.
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.
imult	Pop two (integer) values from the stack, multiply them and put the result back on the stack.
idiv	Pop two (integer) values from the stack, divide them and put the result back on the stack.
irem	Pop two (integer) values from the stack, put the result of $x_1 \% x_2$ back on the stack.
ineg	Negate the value on the stack.
iinc x, y	Increment the variable in slot x by amount y.
ior	Bitwise OR for the two integer values on the stack.
iand	Bitwise AND for the two integer values on the stack.
ixor	Bitwise XOR for the two integer values 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.
pop	Discard word currently on top of the stack.
dup	Duplicate word currently on top of the stack.
swap	Swaps the two top values on the stack.
aload_x	Loads an object reference from slot x.
aload X	Loads an object reference from local variable indexed by X.
iaload	Loads onto the stack an integer from an array. The stack must contain the array reference and the index.
iastore	Stores an integer in an array. The stack must contain the arrayreference, the index and the value, in that order.