Final Exam

Foundations of Software

January 21, 2008

Last Name : _____

First Name : _____

Section : _____

Exercise	**Points**	**Achieved Points**
1	12	
2	10	
3	14	
Total	36	

Exercise 1 : Equivalences (12 points)

Several different equivalence relations can be defined on the terms of a language. The equivalence relations that we consider in this exercise are

1. structural equivalence wrt. α-renaming, denoted $\equiv$,
2. behavioral equivalence, denoted $\cong$, and
3. the smallest equivalence relation containing β-reduction, denoted $\cong_\beta$.

Since a relation is a set of pairs, the above relations are ordered as follows: $\equiv \subset \cong_\beta \subset \cong$
In each part of this exercise you are given two terms in the call-by-value lambda-calculus, unless specified otherwise. Indicate the smallest equivalence relation (that is, $\equiv$, $\cong_\beta$, or $\cong$) that relates the two terms, or indicate with "NONE" that the two terms are not related wrt. any of the above relations.
Note that to test for behavioral equivalence a term can be put into an arbitrary evaluation context. In particular if the language contains more expression forms than just pure lambda-terms, the context is not restricted to applications!

1. λ`x`. λ`y`. λ`z`. `x (y z)` and λ`f`. λ`g`. λ`x`. `f (g x)`
2. In *Scheme*, consider the following terms:

   ```
   (lambda x
     (lambda y
       (lambda z. (x (y z)))))
   ```

   and

   ```
   (lambda f
     (lambda g
       (lambda x. (f (g x)))))
   ```

   Hint: Scheme, and Lisp in general, allows to treat programs as data.
3. λ`y`. λ`x`. `y x` and λ`y`. `y`
4. In the *untyped* call-by-value lambda-calculus with numbers and arithmetic expressions (`succ` t etc.), consider the following terms:

   λ`y`. λ`x`. `y x` and λ`y`. `y`
5. (*twice* `f`) `x` and (*compose* `f f`) `x`

   where

   twice = λ`f`. λ`x`. `f (f x)`

   compose = λ`f`. λ`g`. λ`x`. `f (g x)`
6. (λ`b`. λ`f`. λ`s`. `b f s`) (λ`x`. λ`y`. `x`) and (λ`b`. λ`f`. λ`s`. `b s f`) (λ`x`. λ`y`. `y`)

0.1 Solution

$\equiv$; NONE; $\cong$; NONE; $\cong_\beta$; $\cong_\beta$.

Exercise 2 : Type Reconstruction (10 points)

Consider the language of lambda calculus with type reconstruction, described in Chapter 22 of the TAPL book. Extend the language with *sum types*, following the usual syntax for types and terms:

$$
\begin{array}{lll}
\mathsf{T} & ::= & \ldots \\
 & \mid & \mathsf{T} + \mathsf{T} \quad \text{sum type}
\end{array}
$$

$$
\begin{array}{lll}
t & ::= & \ldots \\
 & \mid & \mathsf{inl}\ t \qquad\qquad\qquad\qquad\qquad\quad \text{inject left} \\
 & \mid & \mathsf{inr}\ t \qquad\qquad\qquad\qquad\qquad\quad \text{inject right} \\
 & \mid & \mathsf{case}\ t\ \mathsf{of}\ \mathsf{inl}\ \mathsf{x} \rightarrow t_1 \mid \mathsf{inr}\ \mathsf{x} \rightarrow t_2 \quad \text{pattern match}
\end{array}
$$

Notice that the injection functions don't require a type ascription, since we rely on type inference to find a suitable type. Extend the constraint typing relation to the new terms in such a way that the language remains sound, using the typical evaluation rules (no need to prove progress and preservation). Recall that the constraint typing relation gives a type *and a set of constraints* for a term and an environment. You can find the original typing rules at page 322 in the TAPL book.

0.2 Solution

$$\text{T-Inl}\ \frac{\Gamma \vdash t_1 :\ T_1 | C}{\Gamma \vdash \texttt{inl}\ t_1 : T_1 + T_2 | C}\ T_2\ fresh$$

$$\text{T-Inr}\ \frac{\Gamma \vdash t_1 :\ T_2 | C}{\Gamma \vdash \texttt{inr}\ t_1 : T_1 + T_2 | C}\ T_1\ fresh$$

$$\text{T-Case}\ \frac{\Gamma \vdash t :\ T | C \quad \Gamma, x :\ X_1 \vdash t_1 :\ T_1 | C_1 \quad \Gamma, x :\ X_2 \vdash t_2 :\ T_2 | C_2}{\Gamma \vdash \texttt{case}\ t\ \texttt{of inl x =>}\ t_1 | \texttt{inr x =>}\ t_2 : T_1 | C \cup C_1 \cup C_2 \cup \{T_1 = T_2, T = X_1 + X_2\}}\ X_1, X_2\ fresh$$

Exercise 3 : Featherweight Java (14 points)

Consider extending Featherweight Java (FJ) with a new field extraction construct. Field extraction allows programmers to easily extract the values of an object's fields without using field accessors.
We formalize this extension by adding one new expression form to the syntax of FJ:

$$\begin{array}{rcll} t & ::= & \ldots & \\ & \mid & t_0\ \texttt{extract C}(x_1, \ldots, x_n)\ \texttt{=>}\ t_1\ \texttt{else}\ t_2 & \text{field extraction} \end{array}$$

The semantics of this construct is supposed to resemble that of the following pattern matching expression in Scala:

```
t0 match {
  case C(x1, ..., xn) => t1
  case _ => t2
}
```

For example, given the following FJ class table

```
class A extends Object {
  Object x;
  A(Object x) {
    super(); this.x = x;
  }
}
class B extends A {
  Object y;
  B(Object x, Object y) {
    super(x); this.y = y;
  }
}
class C extends Object {  }
```

evaluating the following term

```
(new B(new A(new C()), new Object())
  extract A(u) => u else new A(new Object()).x
```

should yield `new C()`.

1. Extend the operational semantics of FJ with additional computation and congruence rules that formalize the semantics of the new field extraction construct.

2. Fill in the preconditions of the typing rule for field extraction so that the above example type checks and the preservation and progress theorems of FJ still hold. Furthermore, the type `E` must be the minimal type for the expression. (You do not need to do any proofs of these properties.)

$$\text{(T-Extract)}\ \frac{\ldots}{\Gamma \vdash t_0\ \texttt{extract C}(x_1, \ldots, x_n)\ \texttt{=>}\ t_1\ \texttt{else}\ t_2\ :\ \texttt{E}}$$