Mid-term Exam

Foundations of Software

November 14, 2018

Last Name : _____

First Name : _____

Section : _____

Sciper : _____

Exercise	Points	Achieved Points
Total	0	

Exercise 1 : Hacking with the untyped call-by-value lambda calculus (10 points)

In this exercise, you have to implement some operations for Church encoding of *positive rational numbers*. One approach for encoding rational numbers in lambda calculus is to use a Church pair, containing Church numerals representing a numerator and a denominator. As an example $\frac{1}{2}$ is represented as follows:

$$\texttt{pair}\ c_1\ c_2$$

where c_1 and c_2 are the standard Church number literals for 1 and 2 (see appendix). Your task is to define the following operations on rational numbers:

1. (2 points) The function ($\texttt{invr}\ x$) which returns the multiplicative inverse of the Church encoded rational number x. The result should be a Church-encoded rational number. For example, if x is $\frac{1}{2}$, the result should be 2. You do not need to handle the case where the rational number is equal to 0.

   $$\texttt{invr} = \lambda\, x.\, \texttt{pair}\ (\texttt{snd}\ x)\ (\texttt{fst}\ x)$$

2. (2 points) The function ($\texttt{multr}\ x\ y$) which returns multiplication of two Church encoded rational number x, y. The result should be a Church-encoded rational number.

   $$\texttt{multr} = \lambda\, x.\, \lambda\, y.\, \texttt{pair}\ (\texttt{times}\ (\texttt{fst}\ x)\ (\texttt{fst}\ y))\ (\texttt{times}\ (\texttt{snd}\ x)\ (\texttt{snd}\ y))$$

3. (2 points) The function ($\texttt{geqr}\ x\ y$) which returns the Church boolean as the result of the greater or equal function for Church-encoded rational numbers x and y. The result should be a Church-encoded boolean.

   $$\texttt{geqr} = \lambda\, x.\, \lambda\, x.\, \texttt{geq}\ (\texttt{times}\ (\texttt{fst}\ x)\ (\texttt{snd}\ y))\ (\texttt{times}\ ((\texttt{fst}\ y)\ (\texttt{snd}\ x))$$
   $$\texttt{geq} = \lambda\, x.\, \lambda\, y.\, (\texttt{iszero}\ (x\ \texttt{pred}\ y))$$

4. (4 points) The function ($\texttt{ceilr}\ x$) which returns the ceiling function one Church encoded rational number x. The result should be a Church-encoded number. For example, if x is $\frac{5}{2}$, the result should be 3.

   ($\texttt{ceilr}\ x$) is the smallest Church numeral y satisfying $\texttt{geq}\ (\texttt{times}\ (y\ \texttt{snd}\ x))\ (\texttt{fst}\ x)$.
   Meaning y times denominator is larger than the numerator. We are recursively searching for it starting from c_0.

   ```
   ceilr = λx. ceilh x c0
   ceilh =
     fix λme. λx. λy.
       test (geq (times y (snd x)) (fst x)))
       (λthen. y)
       (λelse.
         (me x (succ y)))
   ```

You can use the basic operations $not, succ, plus, pred, times, iszero, test, fix, pair, fst, snd$ and the constants. The complete list of predefined operations can be found in the appendix, and only those operations may be used in the exercise. In addition, you are allowed to use any helper functions you define yourself (including the solutions to the question below).

Exercise 2 : Reversal of List (10 points)

Suppose lists of natural numbers are defined inductively as follows:

```
 nat  ::=  Z    |  S  nat
list  ::=  Nil  |  Cons nat list
```

And the reversal operation *rev* is defined recursively below with the help of *snoc*:

```
          snoc Nil n  =  Cons n Nil          (S1)
snoc (Cons x ls) y  =  Cons x (snoc ls y)    (S2)

             rev Nil  =  Nil                 (R1)
rev (Cons n ls)  =  snoc (rev ls) n          (R2)
```

Prove that the following properties hold:

1. (5 points) `rev (snoc l n) = Cons n (rev l)`.
2. (5 points) If `rev l1 = rev l2`, then `l1 = l2`.

Problem 1: `rev (snoc l n) = Cons n (rev l)`.

Proof. By induction on the list l.

Case `l = Nil`.

```
    rev (snoc l n)
=   rev (snoc Nil n)
=   rev (Cons n Nil)  (S1)
=   snoc (rev Nil) n  (R2)
=   snoc Nil n        (R1)
=   Cons n Nil        (S1)
=   Cons n (rev Nil)  (R1)
=   Cons n (rev l)
```

Case `l = Cons m l'`.

```
    rev (snoc l n)
=   rev (snoc (Cons m l') n)
=   rev (Cons m (snoc l' n))  (S2)
=   snoc (rev (snoc l' n)) m  (R2)
=   snoc (Cons n (rev l')) m  (IH)
=   Cons n (snoc (rev l') m)  (S2)
=   Cons n (rev (Cons m l'))  (R2)
=   Cons n (rev l)
```

□

Problem 2: If `rev l1 = rev l2`, then `l1 = l2`.

Proof. We prove `rev (rev l) = l`, the original proof goal follows immediately from it. We do induction on the list. The case `l = Nil` is trivial. For the case `l = Cons m l'`, we have:

```
    rev (rev l)
=   rev (rev (Cons m l'))
=   rev (snoc (rev l') m)  (R2)
=   Cons m (rev (rev l'))  (P1)
=   Cons m l'              (IH)
=   l
```

□

Exercise 3 : Checked Exceptions (10 points)

In this exercise we use the Simply-Typed Lambda Calculus (STLC) extended with rules for error handling. The goal is to be able to check whether a term will throw exceptions in the type system.

In this language, terms may reduce to a normal form `error`, which is *not* a value. In addition, we add the new term form `try` t_1 `with` t_2, which allows handling errors that occur while evaluating t_1. Here is a summary of the extensions to syntax and evaluation:

$$
\begin{array}{rcll}
t & ::= & & \textbf{terms}: \\
 & | & \ldots & \\
 & | & \texttt{error} & \text{throw error} \\
 & | & \texttt{try } t \texttt{ with } t & \text{handle error}
\end{array}
$$

New evaluation rules:

$$\text{(E-AppErr1)}\ \texttt{error}\ t_2 \longrightarrow \texttt{error} \qquad \text{(E-AppErr2)}\ v_1\ \texttt{error} \longrightarrow \texttt{error}$$

$$\text{(E-TryValue)}\ \texttt{try}\ v_1\ \texttt{with}\ t_2 \longrightarrow v_1 \qquad \text{(E-TryError)}\ \texttt{try error with}\ t_2 \longrightarrow t_2$$

$$\text{(E-Try)}\ \frac{t_1 \longrightarrow t_1'}{\texttt{try}\ t_1\ \texttt{with}\ t_2 \longrightarrow \texttt{try}\ t_1'\ \texttt{with}\ t_2}$$

To tell whether a function may throw exceptions, we change the definition of types as follows:

$$
\begin{array}{rcll}
T & ::= & & \textbf{types}: \\
 & | & A & \text{base type} \\
 & | & T \xrightarrow{E} T & \text{function type} \\
E & ::= & \texttt{true} \mid \texttt{false} &
\end{array}
$$

Calling a function of the type $T \xrightarrow{\mathit{false}} T$ will never result in exceptions, while calling a function of the type $T \xrightarrow{\mathit{true}} T$ may result in exceptions.

The goal of this exercise is to define typing rules for STLC with the above extensions such that soundness holds for the system:

- (Progress) If $\emptyset$; $\texttt{false} \vdash t : T$, then either t is a value or else $t \longrightarrow t'$.
- (Preservation) If Γ ; $\texttt{E} \vdash t : T$ and $t \longrightarrow t'$, then Γ ; $\texttt{E} \vdash t' : T$.

The type system uses a typing judgment extended with a Boolean value E, written Γ ; $E \vdash t : T$ where $E \in \{\texttt{true}, \texttt{false}\}$. The flag E indicates whether evaluating the term t may result in exceptions.

Your task is to define the typing rules for this extended language such that soundness holds. *You do not need to prove soundness.* Please make sure that the following typing judgments hold in your system:

- Γ ; $\texttt{false} \vdash \lambda x{:}A.\texttt{error} : A \xrightarrow{\mathit{true}} A$.
- Γ ; $\texttt{true} \vdash \lambda x{:}A.x : A \xrightarrow{\mathit{false}} A$.
- Γ ; $\texttt{false} \vdash \lambda x{:}A.x : A \xrightarrow{\mathit{true}} A$.

- Γ ; `false` $\vdash$ `try error with` $\lambda x{:}A.x : A \xrightarrow{false} A$.
- Γ ; `false` $\vdash$ `try` $\lambda x{:}A.$`error with` $\lambda x{:}A.x : A \xrightarrow{true} A$.

Hint: Please note that it is possible to give different types to the same term.

$$\frac{x \ : \ \mathtt{T} \in \Gamma}{\Gamma \ ; \ E \vdash x \ : \ \mathtt{T}} \qquad \text{(T-Var)}$$

$$\frac{\Gamma, x : S \ ; \ E \vdash t : T}{\Gamma \ ; \ E2 \vdash \lambda x{:}S.t : S \xrightarrow{E} T} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \ ; \ E1 \vdash t_1 : S \xrightarrow{E3} T \qquad \Gamma \ ; \ E2 \vdash t_2 : S}{\Gamma \ ; \ E1 \vee E2 \vee E3 \vdash t_1 \ t_2 : T} \qquad \text{(T-App)}$$

$$\Gamma \ ; \ \mathtt{true} \vdash \mathtt{error} : T \qquad \text{(T-Error)}$$

$$\frac{\Gamma \ ; \ E2 \vdash t_1 : T \quad \Gamma \ ; \ E \vdash t_2 : T}{\Gamma \ ; \ E \vdash \mathtt{try} \ t_1 \ \mathtt{with} \ t_2 : T} \qquad \text{(T-Try)}$$

For reference: **Untyped lambda calculus**

The complete reference of the untyped lambda calculus with call-by-value semantics is:

$$
\begin{array}{rcll}
t & ::= & & \textbf{terms}: \\
& | & x & \textit{variable} \\
& | & \lambda x.\, t & \textit{abstraction} \\
& | & t\ t & \textit{application } (\textit{left assoc.}) \\
\\
v & ::= & & \textbf{values}: \\
& | & \lambda x.\, t & \textit{abstraction}
\end{array}
$$

Small-step reduction rules:

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(R-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(R-App2)}$$

$$(\lambda\ x.\ t_1)\ v_2 \longrightarrow [x \mapsto v_2]\ t_1 \quad \text{(R-AppAbs)}$$

For reference: **Predefined Lambda Terms**

Predefined lambda terms that can be used as-is in the exam

$$
\begin{array}{ll}
\texttt{unit} = & \lambda x.\ x \\
\\
\texttt{tru} = & \lambda t.\ \lambda f.\ t \\
\texttt{fls} = & \lambda t.\ \lambda f.\ f \\
\texttt{iszro} = & \lambda m.\ m\ (\lambda x.\ \texttt{fls})\ \texttt{tru} \\
\texttt{test} = & \lambda b.\ \lambda t.\ \lambda f.\ b\ t\ f\ \texttt{unit} \\
\\
\texttt{pair} = & \lambda f.\ \lambda s.\ \lambda b.\ b\ f\ s \\
\texttt{fst} = & \lambda p.\ p\ \texttt{tru} \\
\texttt{snd} = & \lambda p.\ p\ \texttt{fls} \\
\\
\texttt{c}_0 = & \lambda s.\ \lambda z.\ z \\
\texttt{c}_1 = & \lambda s.\ \lambda z.\ s\ z \\
\texttt{scc} = & \lambda n.\ \lambda s.\ \lambda z.\ s\ (n\ s\ z) \\
\texttt{plus} = & \lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ m\ s\ (n\ s\ z) \\
\texttt{times} = & \lambda m.\ \lambda n.\ m\ (\texttt{plus}\ n)\ c_0 \\
\\
\texttt{zz} = & \texttt{pair}\ c_0\ c_0 \\
\texttt{ss} = & \lambda p.\ \texttt{pair}\ (\texttt{snd}\ p)\ (\texttt{scc}\ (\texttt{snd}\ p)) \\
\texttt{prd} = & \lambda m.\ \texttt{fst}\ (m\ \texttt{ss}\ \texttt{zz})
\end{array}
$$

For reference: **Simply Typed Lambda Calculus**

The complete reference of the simply typed lambda calculus is:

$$
\begin{array}{lcll}
t & ::= & & \textbf{terms}: \\
 & | & x & \textit{variable} \\
 & | & \lambda\, x\!:\!\mathtt{T}.\ t & \textit{abstraction} \\
 & | & t\ t & \textit{application}
\end{array}
$$

$$
\begin{array}{lcll}
v & ::= & & \textbf{values}: \\
 & | & \lambda\, x\!:\!\mathtt{T}.\ t & \textit{abstraction} - \textit{value}
\end{array}
$$

$$
\begin{array}{lcll}
\mathtt{T} & ::= & & \textbf{types}: \\
 & | & \mathtt{T} \to \mathtt{T} & \textit{type of functions (right assoc.)}
\end{array}
$$

Evaluation rules:

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \qquad \text{(E-App2)}$$

$$(\lambda\ x\!:\ \mathtt{T}_1.\ t_1)\ v_2 \longrightarrow [x \to v_2]\ t_1 \qquad \text{(E-AppAbs)}$$

Typing rules:

$$\frac{x\ :\ \mathtt{T} \in \Gamma}{\Gamma \vdash x\ :\ \mathtt{T}} \qquad \text{(T-Var)}$$

$$\frac{\Gamma,\ x\ :\ \mathtt{T}_1 \vdash t_2\ :\ \mathtt{T}_2}{\Gamma \vdash (\lambda\ x\!:\ \mathtt{T}_1.\ t_2)\ :\ \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1\ :\ \mathtt{T}_1 \to \mathtt{T}_2 \quad \Gamma \vdash t_2\ :\ \mathtt{T}_1}{\Gamma \vdash t_1\ t_2\ :\ \mathtt{T}_2} \qquad \text{(T-App)}$$