
Mid-term Exam

Foundations of Software

November 14, 2018

Last Name : _____

First Name : _____

Section : _____

Sciper : _____

Exercise	Points	Achieved Points
1	10	
2	10	
3	10	
Total	30	

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:

`pair c1 c2`

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 (`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.
2. (2 points) The function (`mult x y`) which returns multiplication of two Church encoded rational number x, y . The result should be a Church-encoded rational number.
3. (2 points) The function (`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.
4. (4 points) The function (`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.

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:

$$\begin{array}{lcl} \text{nat} & ::= & \text{Z} \quad | \quad \text{S } \text{nat} \\ \text{list} & ::= & \text{Nil} \quad | \quad \text{Cons nat list} \end{array}$$

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

$$\text{snoc Nil } n = \text{Cons } n \text{ Nil} \quad (\text{S1})$$

$$\text{snoc (Cons } x \text{ ls) } y = \text{Cons } x \text{ (snoc ls } y) \quad (\text{S2})$$

$$\text{rev Nil} = \text{Nil} \quad (\text{R1})$$

$$\text{rev (Cons } n \text{ ls)} = \text{snoc (rev ls) } n \quad (\text{R2})$$

Prove that the following properties hold:

1. (5 points) $\text{rev (snoc l } n) = \text{Cons } n \text{ (rev l)}$.
2. (5 points) If $\text{rev l1} = \text{rev l2}$, then $\text{l1} = \text{l2}$.

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:

$t ::=$	terms :
...	
error	throw error
try t with t	handle error

New evaluation rules:

(E-APPERR1) error $t_2 \longrightarrow \mathbf{error}$	(E-APPERR2) $v_1 \mathbf{error} \longrightarrow \mathbf{error}$
(E-TRYVALUE) try v_1 with $t_2 \longrightarrow v_1$	(E-TRYERROR) try error with $t_2 \longrightarrow t_2$
(E-TRY) $\frac{t_1 \longrightarrow t'_1}{\mathbf{try} \ t_1 \ \mathbf{with} \ t_2 \longrightarrow \mathbf{try} \ t'_1 \ \mathbf{with} \ t_2}$	

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

$T ::=$	types :
A	base type
$T \xrightarrow{E} T$	function type
$E ::= \mathbf{true} \mid \mathbf{false}$	

Calling a function of the type $T \xrightarrow{\mathbf{false}} T$ will never result in exceptions, while calling a function of the type $T \xrightarrow{\mathbf{true}} T$ 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 $\emptyset ; \mathbf{false} \vdash t : T$, then either t is a value or else $t \longrightarrow t'$.
- (Preservation) If $\Gamma ; E \vdash t : T$ and $t \longrightarrow t'$, then $\Gamma ; E \vdash t' : T$.

The type system uses a typing judgment extended with a Boolean value E , written $\Gamma ; E \vdash t : T$ where $E \in \{\mathbf{true}, \mathbf{false}\}$. 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:

- $\Gamma ; \mathbf{false} \vdash \lambda x:A. \mathbf{error} : A \xrightarrow{\mathbf{true}} A$.
- $\Gamma ; \mathbf{true} \vdash \lambda x:A. x : A \xrightarrow{\mathbf{false}} A$.
- $\Gamma ; \mathbf{false} \vdash \lambda x:A. x : A \xrightarrow{\mathbf{true}} A$.

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

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

For reference: Untyped lambda calculus

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

$t ::=$	terms :
x	<i>variable</i>
$\lambda x. t$	<i>abstraction</i>
$t t$	<i>application (left assoc.)</i>
$v ::=$	values :
$\lambda x. t$	<i>abstraction</i>

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

```
unit =     $\lambda x. x$ 

tru =     $\lambda t. \lambda f. t$ 
fls =     $\lambda t. \lambda f. f$ 
iszro =   $\lambda m. m (\lambda x. \text{fls}) \text{tru}$ 
test =    $\lambda b. \lambda t. \lambda f. b t f \text{unit}$ 

pair =    $\lambda f. \lambda s. \lambda b. b f s$ 
fst =     $\lambda p. p \text{tru}$ 
snd =     $\lambda p. p \text{fls}$ 

c0 =     $\lambda s. \lambda z. z$ 
c1 =     $\lambda s. \lambda z. s z$ 
scc =     $\lambda n. \lambda s. \lambda z. s (n s z)$ 
plus =    $\lambda m. \lambda n. \lambda s. \lambda z. m s (n s z)$ 
times =   $\lambda m. \lambda n. m (\text{plus } n) c_0$ 

zz =      $\text{pair } c_0 c_0$ 
ss =      $\lambda p. \text{pair } (\text{snd } p) (\text{scc } (\text{snd } p))$ 
prd =     $\lambda m. \text{fst } (m \text{ ss } \text{zz})$ 
```

For reference: **Simply Typed Lambda Calculus**

The complete reference of the simply typed lambda calculus is:

$t ::=$		terms :
x		<i>variable</i>
$\lambda x:\mathbf{T}. t$		<i>abstraction</i>
$t t$		<i>application</i>
$v ::=$		values :
$\lambda x:\mathbf{T}. t$		<i>abstraction – value</i>
$\mathbf{T} ::=$		types :
$\mathbf{T} \rightarrow \mathbf{T}$	<i>type of functions (right assoc.)</i>	

Evaluation rules:

$$\frac{t_1 \longrightarrow t'_1}{t_1 t_2 \longrightarrow t'_1 t_2} \quad (\text{E-APP1})$$

$$\frac{t_2 \longrightarrow t'_2}{v_1 t_2 \longrightarrow v_1 t'_2} \quad (\text{E-APP2})$$

$$(\lambda x:\mathbf{T}_1. t_1) v_2 \longrightarrow [x \rightarrow v_2] t_1 \quad (\text{E-APPABS})$$

Typing rules:

$$\frac{x : \mathbf{T} \in \Gamma}{\Gamma \vdash x : \mathbf{T}} \quad (\text{T-VAR})$$

$$\frac{\Gamma, x : \mathbf{T}_1 \vdash t_2 : \mathbf{T}_2}{\Gamma \vdash (\lambda x:\mathbf{T}_1. t_2) : \mathbf{T}_1 \rightarrow \mathbf{T}_2} \quad (\text{T-ABS})$$

$$\frac{\Gamma \vdash t_1 : \mathbf{T}_1 \rightarrow \mathbf{T}_2 \quad \Gamma \vdash t_2 : \mathbf{T}_1}{\Gamma \vdash t_1 t_2 : \mathbf{T}_2} \quad (\text{T-APP})$$