CS 496: Homework Assignment 4
Due: 3 May 2020, 11:59pm

1 Assignment Policies

Collaboration Policy. It is acceptable for students to collaborate in understanding the material but not in solving the problems or programming. Use of the Internet is allowed, but should not include searching for existing solutions.

Under absolutely no circumstances code can be exchanged between students from different groups. Excerpts of code presented in class can be used.

2 Assignment

In this assignment you are asked to extend the type-checker for the language REC with references, pairs, lists and trees. The interpreter is provided for you. This assignment is organized in four parts:

1. Part 1. Add type-checking for references.
2. Part 2. Add type-checking for pairs.
3. Part 3. Add type-checking for lists.
4. Part 4. Add type-checking for trees.

The new language including all these constructs will be called CHECKED_PLUS.

3 Type-Checking References

One new grammar production, which is added to the parser for CHECKED_PLUS, is needed for the concrete syntax of expressions.

<Expression> ::= ()

The concrete syntax of types must be extended so as to allow the typing rules described below to be implemented. Two new productions are added, the one for `unit` and for `ref`:

```
<Type>  ::=  int
<Type>  ::=  bool
<Type>  ::=  unit
<Type>  ::=  ref(<Type>)
<Type>  ::=  (<Type> -> <Type>)
```

The typing rules are:

$$\frac{\Gamma \vdash \texttt{e} :: \texttt{t}}{\Gamma \vdash \texttt{newref(e)} :: \texttt{ref(t)}}$$

$$\frac{\Gamma \vdash \texttt{e} :: \texttt{ref(t)}}{\Gamma \vdash \texttt{deref(e)} :: \texttt{t}}$$

$$\frac{\Gamma \vdash \texttt{e1} :: \texttt{ref(t)} \quad \Gamma \vdash \texttt{e2} :: \texttt{t}}{\Gamma \vdash \texttt{setref(e1,e2)} :: \texttt{unit}}$$

$$\frac{}{\Gamma \vdash \texttt{()} :: \texttt{unit}}$$

Note the use of the type `unit`, to indicate that the return result of the assignment operation is not important.

3.1 Task

You have to update the code for the type checker by adapting the file `checker.ml` of the CHECKED language to EXPLICIT-REFS so that `type_of_expr` can handle :

```
let rec type_of_expr : expr -> texpr tea_result = function
  ...
  | Unit -> return UnitType
  ...
  | NewRef(e) -> error "Implement me!"
  | DeRef(e) -> error "Implement me!"
  | SetRef(e1, e2) -> error "Implement me!"
```

Here are some examples:

```
# chk "let x = newref(0) in deref(x)";;
- : texpr Checked.ReM.result = Ok IntType
# chk "let x = newref(0) in x";;
- : texpr Checked.ReM.result =Ok (RefType IntType)
# chk "let x = newref(0) in setref(x,4)";;
- : texpr Checked.ReM.result = Ok UnitType
# chk "newref(newref(zero?(0)))";;
- : texpr Checked.ReM.result =Ok (RefType (RefType BoolType))
# chk "let x = 0 in setref(x,4)";;
- : texpr Checked.ReM.result = Error "setref: Expected a reference type"
```

4 Pairs

4.1 Concrete syntax

Two new productions are added to the grammar of CHECKED_PLUS:

<Expression> ::= pair(<Expression>,<Expression>)
<Expression> ::= unpair (<Identifier>,<Identifier>)=<Expression> in <Expression>

Some examples of programs using pairs follow:

```
pair(3,4)

pair(pair(3,4),5)

pair(zero?(0),3)

pair(proc (x:int) { x - 1 }, 4)

proc (z:<int*int>) { unpair (x,y)=z in x }

proc (z:<int*bool>) { unpair (x,y)=z in pair(y,x) }

let f = proc (z:<int*bool>) { unpair (x,y)=z in pair(y,x) }
in (f pair(1, zero?(0)))
```

Note that the concrete syntax of the types is also extended with a new production:

<Type> ::= int
<Type> ::= bool
<Type> ::= unit
<Type> ::= ref(<Type>)
<Type> ::= (<Type> -> <Type>)
<Type> ::= <<Type> * <Type>>

Regarding the behavior of these expressions they are clear. For example, the expression `(proc (z:<int*int>) unpair (x,y)=z in x pair(2, 3))` is a function that given a pair of integers, projects the first component of the pair.

4.2 Task

Add two new cases to the definition of the function `type_of_expr` in the file `checker.ml`:

```
let rec type_of_expr : expr -> texpr tea_result = function
  ...
  | Pair(e1,e2) -> error "Implement me!"
  | Unpair(id1,id2,e1,e2) -> error "Implement me!"
```

You first need to devise the appropriate typing rules!
Here are some examples:

```
#  chk "pair(2, 3)";;
- : texpr Checked.ReM.result = Ok (PairType (IntType, IntType))
# chk "(proc (z:<int*int>) {unpair (x,y)=z in x} pair(2, 3))";;
- : texpr Checked.ReM.result = Ok IntType
#  chk "(proc (z:int) {unpair (x,y)=z in x} pair(2, 3))";;
- : texpr Checked.ReM.result = Error "unpair: Expected a pair type"
```

5 Lists

5.1 Concrete syntax

The concrete syntax for expressions should be extended with the following productions:

<Expression>	::=	emptylist <Type>
<Expression>	::=	cons (<Expression>, <Expression>)
<Expression>	::=	null? (<Expression>)
<Expression>	::=	hd (<Expression>)
<Expression>	::=	tl (<Expression>)

The concrete syntax for types includes one new production (the last one listed below):

<Type>	::=	`int`
<Type>	::=	`bool`
<Type>	::=	`unit`
<Type>	::=	`ref(`<Type>`)`
<Type>	::=	`(`<Type> `->` <Type>`)`
<Type>	::=	`<<`<Type> * <Type>`>>`
<Type>	::=	`list(`<Type>`)`

The new type constructor is for typing lists. For example,

1. `list(int)` is the type of lists of integers
2. `(int -> list(int))` is the type of functions that given an integer produce a list of integers.

Here are some sample expressions in the extended language. They are supplied in order to help you understand how each constructor works.

```
# chk "emptylist int";;
- : texpr Checked.ReM.result = Ok (ListType IntType)

# chk "cons(1, emptylist int)";;
- : texpr Checked.ReM.result = Ok (ListType IntType)

# chk "hd(cons(1, emptylist int))";;
- : texpr Checked.ReM.result = Ok IntType

# chk "tl(cons(1, emptylist int))";;
- : texpr Checked.ReM.result = Ok (ListType IntType)

# chk "cons(null?(emptylist int), emptylist int)";;
- : texpr Checked.ReM.result =
Error "cons: type of head and tail do not match"

# chk "proc(x:int) { proc (l:list(int)) { cons(x,l) } }";;
- : texpr Checked.ReM.result =
Ok (FuncType (IntType, FuncType (ListType IntType, ListType IntType)))
```

Here are the typing rules:

5.2 Typing rules

$$\frac{}{\Gamma \vdash \texttt{emptylist t} :: \texttt{list(t)}} \qquad \frac{\Gamma \vdash \texttt{e1} :: \texttt{t} \quad \Gamma \vdash \texttt{e2} :: \texttt{list(t)}}{\Gamma \vdash \texttt{cons(e1,e2)} :: \texttt{list(t)}}$$

$$\frac{\Gamma \vdash \texttt{e} :: \texttt{list(t)}}{\Gamma \vdash \texttt{tl(e)} :: \texttt{list(t)}} \qquad \frac{\Gamma \vdash \texttt{e} :: \texttt{list(t)}}{\Gamma \vdash \texttt{hd(e)} :: \texttt{t}}$$

$$\frac{\Gamma \vdash \texttt{e} :: \texttt{list(t)}}{\Gamma \vdash \texttt{null?(e)} :: \texttt{bool}}$$

5.3 Task

Extend the type-checker to del with the new constructs:

```
let rec type_of_expr : expr -> texpr tea_result = function
  ...
  | EmptyList(t) -> error "Implement me!"
  | Cons(he, te) -> error "Implement me!"
  | Null(e) -> error "Implement me!"
  | Hd(e) -> error "Implement me!"
  | Tl(e) -> error "Implement me!"
```

6 Trees

6.1 Concrete syntax

The concrete syntax for expressions should be extended with the following productions:

<Expression>	::=	emptytree <Type>
<Expression>	::=	node (<Expression>, <Expression>, <Expression>)
<Expression>	::=	nullT? (<Expression>)
<Expression>	::=	getData (<Expression>)
<Expression>	::=	getLST (<Expression>)
<Expression>	::=	getRST (<Expression>)

The concrete syntax for types includes one new production (the last one listed below):

<Type>	::=	`int`
<Type>	::=	`bool`
<Type>	::=	`unit`
<Type>	::=	`ref`(<Type>)
<Type>	::=	`(`<Type> `->` <Type>`)`
<Type>	::=	`<<`<Type> `*` <Type>`>>`
<Type>	::=	`list`(<Type>)
<Type>	::=	`tree` (<Type>)

Here is a sample program that assumes you have implemented the `append` operation on lists. Note that you will not be able to run this program unless you have extended the

interpreter to deal with lists and trees. This assignment only asks you to write the type-checker, not the interpreter. You are, of course, encouraged to write the interpreter too!

Here is another example. It should type-check with result `Ok (ListType IntType)` and evaluate to `Ok (ListVal [NumVal 1; NumVal 2; NumVal 3])`:

```
letrec append(xs:list(int)): list(int) -> list(int)  =
       proc (ys:list(int)) {
          if null?(xs)
            then ys
            else cons(hd(xs),((append tl(xs)) ys))
       }
in
letrec inorder(x:tree(int)):list(int)  =
  if nullT?(x)
    then emptylist int
    else ((append (inorder getLST(x))) cons(getData(x),
     (inorder getRST(x))))
in
(inorder node(2,
              node(1,emptytree int,emptytree int),
              node(3,emptytree int,emptytree int)))
```

6.2 Typing rules

$$\frac{}{\Gamma \vdash \texttt{emptytree t} :: \texttt{tree(t)}} \qquad \frac{\Gamma \vdash \texttt{e1} :: \texttt{t} \quad \Gamma \vdash \texttt{e2} :: \texttt{tree(t)} \quad \Gamma \vdash \texttt{e3} :: \texttt{tree(t)}}{\Gamma \vdash \texttt{node(e1,e2,e3)} :: \texttt{tree(t)}}$$

$$\frac{\Gamma \vdash \texttt{e} :: \texttt{tree(t)}}{\Gamma \vdash \texttt{getData(e)} :: \texttt{t}} \qquad \frac{\Gamma \vdash \texttt{e} :: \texttt{tree(t)}}{\Gamma \vdash \texttt{nullT?(e)} :: \texttt{bool}}$$

$$\frac{\Gamma \vdash \texttt{e} :: \texttt{tree(t)}}{\Gamma \vdash \texttt{getLST(e)} :: \texttt{tree(t)}} \qquad \frac{\Gamma \vdash \texttt{e} :: \texttt{tree(t)}}{\Gamma \vdash \texttt{getRST(e)} :: \texttt{tree(t)}}$$

6.3 Task

Extend the type-checker to del with the new constructs:

```
let rec type_of_expr : expr -> texpr tea_result = function
  ...
  | EmptyTree(t) -> error "Implement me!"
  | Node(de, le, re) -> error "Implement me!"
  | NullT(t) -> error "Implement me!"
  | GetData(t) -> error "Implement me!"
  | GetLST(t) -> error "Implement me!"
  | GetRST(t) -> error "Implement me!"
```

Here are some sample expressions in the extended language. They are supplied in order you to help you understand how each construct works.

```
 # chk "emptytree int";;
- : texpr Checked.ReM.result = Ok (TreeType IntType)

# chk "nullT?(node(1, node(2, emptytree int, emptytree int), emptytree int))";;
- : texpr Checked.ReM.result = Ok BoolType
```

```
# chk "getData(node(1, node(2, emptytree int, emptytree int), emptytree int))";;
- : texpr Checked.ReM.result = Ok IntType

# chk "getLST(node(1, node(2, emptytree int, emptytree int), emptytree int))";;
- : texpr Checked.ReM.result = Ok (TreeType IntType)
```

7 Submission instructions

Submit a file named `HW3_<SURNAME>.zip` through Canvas which includes all the source files required to run the interpreter and type-checker. Your grade will be determined as follows, for a total of 100 points:

Section	Grade
Unit	5
Reference	15
Pair	20
List	30
Tree	30