Homework 5 — assigned 29 November — due Tuesday 14 December

5.1 PCF (50pts)

Extend the language of homework 4.3 with a new syntactic form for general recursion, `fix`, and with new typing and evaluation rules for it, exactly as described in TAPL, Chapter 11.

This language is traditionally called PCF, so please prepare your work in files `PCF.lhs`, `PCF.pdf`, etc.

Thus, here is the Haskell representation of source terms:

```
data Term = ... | Fix Term
```

Concrete syntax:

```
Term --> ...  | fix_keyword  lpar Term rpar
```

As an example, a rendering of `iseven 7` is as follows:[1]

```
app (fix (abs (ie:->(Nat,Bool). abs (x:Nat. if iszero(x) then true else
if iszero (pred (x)) then false else app (ie, pred (pred (x))) fi fi))),
succ (succ (succ (succ (succ (succ (succ (0))))))))
```

Define various test terms, including at least the renderings into PCF of the following: `leq 2 3`, `equal 2 3`, `plus 2 3`, `times 2 3`, `exp 2 3`, `fact 3`, `fact (fact 3)`.

Here is the complete representation of source terms:

```
data Term  =  Var Var
           |  Abs Var Type Term
           |  App Term Term
           |  Tru
           |  Fls
           |  If Term Term Term
           |  Zero
           |  Succ Term
           |  Pred Term
           |  IsZero Term
           |  Fix Term
          deriving Eq
```

Here is the complete language to be implemented:

[1] Note that this is different from the approach taken in the textbook, p. 144–145, which uses pairs to define both odds and evens at once.

Syntax

t	$::=$		*terms:*
		x	*variable*
		$\lambda x : T.t$	*abstraction*
		$t\ t$	*application*
		true	*constant true*
		false	*constant false*
		$\mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t$	*conditional*
		$\mathsf{0}$	*constant zero*
		$\mathsf{succ}\ t$	*successor*
		$\mathsf{pred}\ t$	*predecessor*
		$\mathsf{iszero}\ t$	*zero test*
		$\mathsf{fix}\ t$	*fixed point of* t
v	$::=$		*values:*
		$\lambda x : T.t$	*abstraction value*
		true	*true value*
		false	*false value*
		nv	*numeric value*
nv	$::=$		*numeric values:*
		$\mathsf{0}$	*zero value*
		$\mathsf{succ}\ nv$	*successor value*
T	$::=$		*types:*
		$T \to T$	*type of functions*
		Bool	*type of booleans*
		Nat	*type of natural numbers*
Γ	$::=$		*typing contexts:*
		$\varnothing$	*empty context*
		$\Gamma, x : T$	*term variable binding*

Evaluation

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

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

$$(\lambda x : T_{11}.t_{12})\, v_2 \longrightarrow [x \mapsto v_2]\, t_{12} \quad \text{(E-AppAbs)}$$

$$\mathsf{if\ true\ then}\ t_2\ \mathsf{else}\ t_3 \longrightarrow t_2 \quad \text{(E-IfTrue)}$$

$$\mathsf{if\ false\ then}\ t_2\ \mathsf{else}\ t_3 \longrightarrow t_3 \quad \text{(E-IfFalse)}$$

$$\frac{t_1 \longrightarrow t_1'}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \longrightarrow \mathsf{if}\ t_1'\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3} \quad \text{(E-If)}$$

$$\frac{t_1 \longrightarrow t_1'}{\mathsf{succ}\ t_1 \longrightarrow \mathsf{succ}\ t_1'} \quad \text{(E-Succ)}$$

$$\mathsf{pred}\ 0 \longrightarrow 0 \quad \text{(E-PredZero)}$$

$$\mathsf{pred}(\mathsf{succ}\ nv_1) \longrightarrow nv_1 \quad \text{(E-PredSucc)}$$

$$\frac{t_1 \longrightarrow t_1'}{\mathsf{pred}\ t_1 \longrightarrow \mathsf{pred}\ t_1'} \quad \text{(E-Pred)}$$

$$\mathsf{iszero}\ 0 \longrightarrow \mathsf{true} \quad \text{(E-IszeroZero)}$$

$$\mathsf{iszero}(\mathsf{succ}\ nv_1) \longrightarrow \mathsf{false} \quad \text{(E-IszeroSucc)}$$

$$\frac{t_1 \longrightarrow t_1'}{\mathsf{iszero}\ t_1 \longrightarrow \mathsf{iszero}\ t_1'} \quad \text{(E-Iszero)}$$

$$\mathsf{fix}(\lambda x : T_1.t_2) \longrightarrow [x \mapsto \mathsf{fix}(\lambda x : T_1.t_2)]\, t_2 \quad \text{(E-FixBeta)}$$

$$\frac{t_1 \longrightarrow t_1'}{\mathsf{fix}\ t_1 \longrightarrow \mathsf{fix}\ t_1'} \quad \text{(E-Fix)}$$

Typing

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

$$\frac{\Gamma, x : T_1 \vdash t_2 \,:\, T2}{\Gamma \vdash \lambda x : T_1 . t_2 \,:\, T_1 \to T_2} \text{ (T-ABS)}$$

$$\frac{\Gamma \vdash t_1 \,:\, T_{11} \to T_{12} \qquad \Gamma \vdash t_2 \,:\, T_{11}}{\Gamma \vdash t_1\, t_2 \,:\, T_{12}} \text{ (T-APP)}$$

$$\Gamma \vdash \mathsf{true} \,:\, \mathsf{Bool} \text{ (T-TRUE)}$$

$$\Gamma \vdash \mathsf{false} \,:\, \mathsf{Bool} \text{ (T-FALSE)}$$

$$\frac{\Gamma \vdash t_1 \,:\, Bool \qquad \Gamma \vdash t_2 \,:\, T \qquad \Gamma \vdash t_3 \,:\, T}{\Gamma \vdash \mathsf{if}\, t_1 \,\mathsf{then}\, t_2 \,\mathsf{else}\, t_3 \,:\, T} \text{ (T-IF)}$$

$$\Gamma \vdash 0 \,:\, \mathsf{Nat} \text{ (T-ZERO)}$$

$$\frac{\Gamma \vdash t_1 \,:\, \mathsf{Nat}}{\Gamma \vdash \mathsf{succ}\, t_1 \,:\, \mathsf{Nat}} \text{ (T-SUCC)}$$

$$\frac{\Gamma \vdash t_1 \,:\, \mathsf{Nat}}{\Gamma \vdash \mathsf{pred}\, t_1 \,:\, \mathsf{Nat}} \text{ (T-PRED)}$$

$$\frac{\Gamma \vdash t_1 \,:\, \mathsf{Nat}}{\Gamma \vdash \mathsf{iszero}\, t_1 \,:\, \mathsf{Bool}} \text{ (T-ISZERO)}$$

$$\frac{\Gamma \vdash t_1 \,:\, T_1 \to T_1}{\Gamma \vdash \mathsf{fix}\, t_1 \,:\, T_1} \text{ (T-FIX)}$$

5.2 Unification (50pts)

Implement a general-purpose unification algorithm in Haskell. (By general-purpose, we mean that it can be used for tasks other than typing.) You may implement *any* unification algorithm you wish; however, it is recommended that you consult the paper by Martelli and Montanari (1982), posted on the course wiki, for a description of either the canonical nondeterministic (but inefficient) algorithm, or the efficient algorithm the authors introduced.

The following interface must be implemented: Haskell type variable `v` ranges over possible types that term variables might have (most typically `String`), whereas Haskell type variable `f` ranges over possible types that term function symbols might have (again most typically `String`).

NB. The given `EquationOutcome` tags, in particular the `NoMatch` tag, make sense in the context of a reasonable implementation of the canonical nondeterministic algorithm. Feel free to change them to fit your algorithm.

```
data Term v f =
        Fun f [Term v f]
      | Var v

type Equation v f =  (Term v f, Term v f)

type Binding v f = (v, Term v f)

type Substitution v f = [Binding v f]

data EquationOutcome = HaltWithFailure | HaltWithCycle | NoMatch | Success

unify :: (Eq v, Eq f) => [Equation v f] -> (EquationOutcome, [Equation v f])
```

In this setting, an example unification problem can be represented as follows:

```
s1 = Fun "f" [Fun "g" [Fun "a" [], Var "X"], Fun "h" [Fun "f" [Var "Y", Var "Z"]]]
s2 = Fun "g" [Var "Y", Fun "h" [Fun "f" [Var "Z", Var "U"]]]
t1 = Fun "f" [Var "U", Fun "h" [Fun "f" [Var "X", Var "X"]]]
t2 = Fun "g" [Fun "f" [Fun "h" [Var "X"], Fun "a" []],
      Fun "h" [Fun "f" [Fun "a" [], Fun "b" []]]]
problem = [(s1, t1), (s2, t2)]
solution = unify problem
```

All parts of the solution (algorithm description, implementation, testing using the given unification problem instance and others) must be combined into a single PDF file `Unification.pdf`.

5.3 Type reconstruction (100pts)

Taking advantage of the solution to the preceding problem (unification, 5.2), implement type reconstruction for PCF (from 5.1).

(Note that the use of Haskell modules is required. You will divide the code for PCF into a module `AbstractSyntax`, containing the declaration of the `Term` type (among other things), and other useful modules, such as perhaps a `Parser`, a `Scanner`, and an `OperationalSemantics` module.)

```
module AbstractSyntax where
...
type TypeVar  =  String
data Type  =  TypeArrow      Type Type
           |  TypeBool
           |  TypeNat
           |  TypeVar        TypeVar
...
type Var = String
data Term  =
              Var         Var
           |  Abs         Var Type Term
           |  App         Term Term
...

module ConstraintTyping where

import qualified AbstractSyntax as S
import qualified Unification as U

type TypeConstraint = (S.Type, S.Type)
type TypeConstraintSet = [TypeConstraint]
type TypeSubstitution = [(S.TypeVar, S.Type)]

reconstructType :: S.Term -> Maybe S.Term
reconstructType t =
  let
    constraints = deriveTypeConstraints t
    unifencoding = encode constraints
    (unifoutcome, unifsolvedequations) = U.unify unifencoding
  in
    case unifoutcome of
      U.Success ->
        let
          typesubst = decode unifsolvedequations
          t' = applyTypeSubstitutionToTerm typesubst t
```

```
        in
          Just t'
      U.HaltWithFailure -> Nothing
      U.HaltWithCycle -> Nothing
```

The bridge to generic unification is given by:

```
type TypeUnifVar = S.TypeVar
data TypeUnifFun = TypeUnifArrow | TypeUnifBool | TypeUnifNat
                   deriving (Eq, Show)

encode :: TypeConstraintSet -> [U.Equation TypeUnifVar TypeUnifFun]
encode = map (\(tau1, tau2) -> (enctype tau1, enctype tau2))
  where
    enctype :: S.Type -> U.Term TypeUnifVar TypeUnifFun
    enctype (S.TypeArrow tau1 tau2)  =  U.Fun TypeUnifArrow [enctype tau1, enctype tau2]
    enctype S.TypeBool               =  U.Fun TypeUnifBool []
    enctype S.TypeNat                =  U.Fun TypeUnifNat []
    enctype (S.TypeVar xi)           =  U.Var xi

decode :: [U.Equation TypeUnifVar TypeUnifFun] -> TypeSubstitution
decode = map f
  where
    f :: (U.Term TypeUnifVar TypeUnifFun, U.Term TypeUnifVar TypeUnifFun)
         -> (S.TypeVar, S.Type)
    f (U.Var xi, t) = (xi, g t)
    g :: U.Term TypeUnifVar TypeUnifFun -> S.Type
    g (U.Fun TypeUnifArrow [t1, t2])  =  S.TypeArrow (g t1) (g t2)
    g (U.Fun TypeUnifBool [])         =  S.TypeBool
    g (U.Fun TypeUnifNat [])          =  S.TypeNat
    g (U.Var xi)                      =  S.TypeVar xi
```

Write the function `deriveTypeConstraints` to complete the program. Test thoroughly.

Prepare a file `TypeReconstruction.pdf` containing the nicely formatted program source (all modules!) and test runs.

How to turn in

Submission instructions: use the turnin facility provided by the department; see course wiki for details.

Include the following statement with your submission, signed and dated:
I pledge my honor that in the preparation of this assignment I have complied with the University of New Mexico Board of Regents' Policy Manual.