Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ciencias Exactas y Naturales
Departamento de Ciencia de la Computación
Análisis de Lenguajes de Programación

Nombre y Apellido: Legajo:

Examen Parcial 2

1. Dada la representación de listas en Sistema F:

$$\texttt{List}\ X = \forall\, R.\, (X \to R \to R) \to R \to R$$

Definir las siguientes funciones en Sistema F, suponiendo que se tienen definidas las funciones `nil`, `cons`, `map` y append.

a) $\texttt{singleton} : \forall\, X.\, X \to \texttt{List}\ X$, que construye una lista unitaria a partir de un elemento.

b) $\texttt{inits} : \forall\, X.\, \texttt{List}\ X \to \texttt{List}\ (\texttt{List}\ X)$, que dada una lista construye una lista con los comienzos de ésta. (En Haskell, por ejemplo $\texttt{inits}\ [1,2,3] = [[1],[1,2],[1,2,3]]$)

2. A continuación se presenta la sintaxis de un lenguaje de expresiones booleanas y números una semántica de paso chico.

Sintaxis:

$$\begin{array}{l} e ::= \texttt{T} \mid \texttt{F} \mid n \mid x \mid \texttt{and}\ e\ e \mid \texttt{not}\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \\ v ::= \texttt{T} \mid \texttt{F} \mid n \\ \texttt{Type} ::= \texttt{Bool} \mid \texttt{Nat} \end{array}$$

Semántica:

$$\frac{e \to e_1}{\texttt{not}\ e \to \texttt{not}\ e_1}\ \text{(E-Not1)} \qquad \texttt{not T} \to \texttt{F}\ \text{(E-Not2)} \qquad \texttt{not F} \to \texttt{T}\ \text{(E-Not3)}$$

$$\texttt{let}\ x = v\ \texttt{in}\ e \to e\,[v\,/\,x]\ \text{(E-Let1)} \qquad \frac{e \to e_2}{\texttt{let}\ x = e\ \texttt{in}\ e_1 \to \texttt{let}\ x = e_2\ \texttt{in}\ e_1}\ \text{(E-Let2)}$$

$$\texttt{and T}\ e \to e\ \text{(E-And)} \qquad \texttt{and F}\ e \to \texttt{F}\ \text{(E-And1)} \qquad \frac{e_1 \to e_2}{\texttt{and}\ e\ e_1 \to \texttt{and}\ e_2\ e}\ \text{(E-And2)}$$

a) Enunciar las propiedades de progreso y preservación.

b) Suponiendo que el lenguaje dado tiene un sistema de tipos usual, el lenguaje no cumple con la propiedad de progreso. Demostrar con un contraejemplo por qué no se cumple la propiedad.

CamScanner

3. El siguiente tipo es utilizado para definir cómputos monádicos que leen información de un entorno:

```
type Env = [(String, Bool)]
newtype Reader a = R {runR :: Env → a}
```

a) Dar la instancia de Monad para Reader.

b) Definir las operaciones de ésta mónada:

   i. ask :: Reader Env, obtiene el entorno.

   ii. local :: (Env → Env) → Reader a → Reader a, permite modificar el contenido del entorno.

c) El siguiente tipo de datos permite combinar los efectos de Reader con los efectos de la mónada Write. La cual utilizaremos para definir un evaluador de expresiones booleanas que lleva una traza de evaluación junto con la cantidad de pasos que fueron necesarios para evaluar la expresión.

```
type Traza = String
newtype ReaderWriter a = RW {runRW :: Env → (a, Traza, Int)}
```

Dar la instancia de Monad para ReaderWriter.

Para ésta mónada se definieron las siguientes operaciones:

```
addTrace :: Traza → ReaderWriter ()
addTrace s = RW (\_ → ((), s, 0))
addStep :: ReaderWriter ()
addStep = RW (\_ → ((), "", 1))
```

y se modificaron las operaciones ask y local, que ahora tienen éste tipo:

- local :: (Env → Env) → ReaderWriter a → ReaderWriter a
- ask :: ReaderWriter Env

d) El siguiente tipo de datos representa expresiones booleanas:

```
data BoolExp = T | F                          -- constantes
             | Var String                     -- variable boolena
             | Let String BoolExp BoolExp     -- define una variable local nueva
             | And BoolExp BoolExp
             | Not BoolExp
  deriving (Eq, Show)
```

Definir una función eval :: BoolExp → ReaderWriter Bool que evalúe una expresión generando una traza de evaluación para las operaciones: let , not y and y cuente la cantidad de pasos de evaluación.

e) Definir una función mostrar :: BoolExp → IO (), que evalúe una expresión en el entorno vacío y muestre en pantalla el resultado, la traza y el número de operaciones.

Por ejemplo mostrar (Let "x" T (And (Var "x") (And (Not (Var "x")) F))) debe motrar:

```
El resultado de la evaluación es False

Pasos de evaluación:
Not (Var "x") --> False
And (Not (Var "x")) F --> False
And (Var "x") (And (Not (Var "x")) F) --> False
Let "x" T (And (Var "x") (And (Not (Var "x")) F)) --> False

La expresión se evaluó en 4 pasos.
```


CamScanner

Examen Parcial 3

1. Un *functor aplicativo* es un functor que soporta la aplicación de funciones dentro de una estructura functorial. Está definido por la siguiente clase:

```
class (Functor f) ⇒ Applicative f where
  pure :: a → f a
  (<*>) :: f (a → b) → f a → f b
```

y cumple las leyes:

```
pure id <*> v = v                            — Identity
pure f <*> pure x = pure (f x)               — Homomorphism
pure (∘) <*> u <*> v <*> w = u <*> (v <*> w) — Composition
```

a) Dar la instancia de Functor para *Aplicative*:

```
instance Applicative f ⇒ Functor f where
  fmap ...
```

b) Probar que un functor aplicativo satisface las leyes de functor.

2. Un valor de tipo Parser i o es una función que toma una entrada de tipo i (que representa el resto de la entrada en un momento dado) y devuelve *Nothing* si el parseo falla, o una tupla con el valor parseado y el resto de la entrada.

```
newtype Parser i o = P {runP :: i → Maybe (o, i)}
```

a) Dar la instancia de Monad para Parser i.

b) Definir los siguientes combinadores de parser:

i. item :: Parser String Char, consume el primer caracter o falla si su entrada es vacía.

ii. failure :: Parser i o, parser que falla siempre.

iii. word :: String → Parser String String, parsea una cadena dada.

c) Definir una función parse :: Show o ⇒ Parser String o → String → IO () que dado un parser y un nombre de archivo aplique el parser al contenido del archivo e imprima el resultado en pantalla si el parseo consumió toda la entrada (es decir, si la función dentro del parser devuelve un valor de tipo o junto con []) o un mensaje de error si hubo un error de parseo o no se consumió toda la entrada.

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ciencias Exactas y Naturales
Departamento de Ciencia de la Computación
Análisis de Lenguajes de Programación

Nombre y Apellido: SEBASTIAN MORALES Legajo: M-6307/3

Examen Parcial recuperatorio 3

1. Un *functor contravariante* es un constructor de tipo para el cual se puede definir una función contramap :: $(a \to b) \to f\ b \to f\ a$ que satisfaga las siguientes leyes:

```
contramap id = id                                  -- functor contravariante.1
contramap f ∘ contramap g = contramap (g ∘ f)      -- functor contravariante.2
```

En Haskell, se definen mediante la siguiente clase:

```
class Contravariant f where
  contramap :: (a → b) → f b → f a
```

Dado el siguiente tipo de datos:

```
data B a = B (a → Bool)
```

a) Dar la instancia de Contravariant para B.

b) Probar que la instancia dada satisface las leyes de functor contravariante.

2. El siguiente tipo de datos es utilizado para combinar los efectos de una mónada m con los de la mónada Reader, la cual es utilizada para leer valores de un entorno:

```
newtype ReaderM m e a = R {runR :: (e → m a)}
```

a) Dar la instancia de Monad para ReaderM m e (No hace falta probar que es una mónada)

b) Definir las siguientes funciones para manejar el entorno:

i. ask :: Monad m ⇒ ReaderM m e e, devuelve el entorno.

ii. local :: Monad m ⇒ $e \to$ ReaderM m e $a \to$ ReaderM m e a, dado un entorno *env* y un cómputo monádico r ejecuta r en el entorno *env*.

c) El siguiente tipo de datos define la sintaxis de un lenguaje de proposiciones:

```
data Prop = Var String | Cons Bool | And Prop Prop | Not Prop | Def String Prop Prop
```

donde Def "x" p q representa la proposición q donde la variable "x" hace referencia a la proposición p. Las variables pertenecen a un entorno representado mediante el tipo de datos:

```
type Env = [(String, Bool)]
```

Se desea implementar un evaluador monádico para dicho lenguaje, donde el resultado de la evaluación está representado por el siguiente tipo de datos:

```
data Result a = UndefVar String | Res a
```

i. Dar la instancia de Monad para Result.

ii. Definir una función undef :: String → ReaderM Result Env a, que dada una variable devuelva un cómputo monádico que represente el error de que dicha variable está indefinida.

iii. Definir una función eval :: Prop → ReaderM Result Env Bool, que defina un evaluador para el lenguaje.

iv. Definir una función app :: Prop → Env → Result Bool, que dada una proposición y un entorno evalúe la proposición en el entorno dado.

Asamb

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ciencias Exactas y Naturales
Departamento de Ciencia de la Computación
Análisis de Lenguajes de Programación

Nombre y Apellido:

Examen Parcial 2

1. A continuación se presenta un λ-cálculo tipado extendido con naturales y listas de naturales.

Sintaxis:

$$
\begin{array}{rl}
t ::= & x \mid \lambda x : T.\ t \mid t\ t \mid 0 \mid \texttt{suc}\ t \\
& \mid \texttt{nil} \mid \texttt{cons}\ t\ t \mid \texttt{head}\ t \mid \texttt{R}\ t\ t\ t \\
v & ::= \lambda x : T.\ t \mid nv \mid lv \\
nv & ::= 0 \mid \texttt{suc}\ nv \\
lv & ::= \texttt{nil} \mid \texttt{cons}\ nv\ lv \\
T & ::= T \to T \mid \mathbb{N} \mid \mathbb{N}^*
\end{array}
$$

Sistema de tipos:

$$\Gamma \vdash 0 : \mathbb{N} \qquad \text{(T-Zero)}$$

$$\frac{\Gamma \vdash t : \mathbb{N}}{\Gamma \vdash \texttt{suc}\ t : \mathbb{N}} \qquad \text{(T-Suc)}$$

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

$$\frac{\Gamma, x : T \vdash t : T'}{\Gamma \vdash \lambda x : T.\ t : T \to T'} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : T \to T' \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1\ t_2 : T'} \qquad \text{(T-App)}$$

$$\Gamma \vdash \texttt{nil} : \mathbb{N}^* \qquad \text{(T-Nil)}$$

$$\frac{\Gamma \vdash t_1 : \mathbb{N} \quad \Gamma \vdash t_2 : \mathbb{N}^*}{\Gamma \vdash \texttt{cons}\ t_1\ t_2 : \mathbb{N}^*} \qquad \text{(T-Cons)}$$

$$\frac{t : \mathbb{N}^*}{\texttt{head}\ t : \mathbb{N}} \qquad \text{(T-Head)}$$

$$\frac{t_1 : T \quad t_2 : \mathbb{N} \to \mathbb{N}^* \to T \to T \quad t_3 : \mathbb{N}^*}{\texttt{R}\ t_1\ t_2\ t_3 : T} \qquad \text{(T-R)}$$

Semántica:

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

$$\frac{t_2 \to t_2'}{v\ t_2 \to v\ t_2'} \qquad \text{(E-App2)}$$

$$(\lambda x : T.\ t_1)\ v \to t_1\ [v\ /\ x] \qquad \text{(E-Abs)}$$

$$\frac{t_1 \to t_2}{\texttt{cons}\ t_1\ t \to \texttt{cons}\ t_2\ t} \qquad \text{(E-Cons1)}$$

$$\frac{t_1 \to t_2}{\texttt{cons}\ v\ t_1 \to \texttt{cons}\ v\ t_2} \qquad \text{(E-Cons2)}$$

$$\frac{t_1 \to t_2}{\texttt{suc}\ t_1 \to \texttt{suc}\ t_2} \qquad \text{(E-Suc)}$$

$$\frac{t \to t'}{\texttt{head}\ t \to \texttt{head}\ t'} \qquad \text{(E-Head1)}$$

$$\texttt{head}\ (\texttt{cons}\ v\ t) \to v \qquad \text{(E-Head2)}$$

$$\frac{t_3 \to t_3'}{\texttt{R}\ t_1\ t_2\ t_3 \to \texttt{R}\ t_1\ t_2\ t_3'} \qquad \text{(E-R1)}$$

$$\texttt{R}\ t_1\ t_2\ \texttt{nil} \to t_1 \qquad \text{(E-R2)}$$

$$\texttt{R}\ t_1\ t_2\ (\texttt{cons}\ v\ t) \to t_2\ v\ t\ (\texttt{R}\ t_1\ t_2\ t) \qquad \text{(E-R3)}$$

a) Enunciar las propiedades de progreso y preservación.

b) Determinar si el lenguaje dado cumple las propiedades de progreso y preservación dando la prueba correspondiente si la propiedad se cumple o dando un contraejemplo en caso contrario.

c) Definir en el lenguaje las siguientes funciones:

   i) La función *tail* que devuelve la cola de una lista.

   ii) La función *map* que aplica una función a todos los elementos de una lista.

d) Se desea extender el lenguaje con un término *iterate* $n\ x$ que construya una lista de n elementos x.
   Extender la gramática, la semántica y el sistema de tipos para contemplar esta nueva construcción.

2. Una forma de representar enteros en λ-calculus consiste en usar un par formado por dos numerales de Church, los cuales representan un valor positivo y un valor negativo, de manera que el valor entero es la diferencia de estos dos números. Por ejemplo, el entero $-x$ podría representarse como $(0, x)$ o como $(y, x + y)$, siendo y cualquier natural.
Usando la representación de pares, booleanos y numerales de Church dadas en clase definir las siguientes funciones en λ-cálculo (suponiendo definidas las funciones *pair*, *fst*, *snd*, *ifthenelse*, *true*, *false*, *succ*, *zero*, *isZero* y *pred*):

1. La función *isPos* determina si un entero es positivo o negativo.
2. La función *neg* que represente la operación negación unaria de un entero.
3. La función *abs* que calcula el valor absoluto de un entero.

3. Dada la representación de naturales y listas en Sistema F:

$$Nat = \forall X.\ (X \to X) \to X \to X$$
$$List\ X = \forall Y.\ (X \to Y \to Y) \to Y \to Y$$

Definir la función $sum : List\ Nat \to Nat$, que suma los elementos de una lista de naturales. Suponer que se tiene definida la función $suma : Nat \to Nat \to Nat$, que suma dos naturales.

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ciencias Exactas y Naturales
Departamento de Ciencia de la Computación
Análisis de Lenguajes de Programación

Nombre y Apellido:

Examen Parcial 3

1. La siguiente mónada es utilizada para cómputos que llevan información adicional, aparte de su valor de cómputo.

```
newtype Writer w a = Writer {runWriter :: (a, [w])}
instance Monad (Writer w) where
  return a = Writer (a, [])
  (Writer (a, w)) >>= f = let (a', w') = runWriter (f a)
                          in Writer (a', w ++ w')
```

a) Probar que Writer w es una mónada. Puede asumir que $+\!\!+$ es asociativa sin demostrarlo.

b) Para combinar los efectos de la mónada Writer con los efectos de Maybe se definió el siguiente tipo de datos:

```
newtype WriterMaybe w a = WM {runWM :: (Maybe a, [w])}
```

Dar la instancia de Monad para WriterMaybe w.

c) Definir las siguientes operaciones sobre la mónada WriterMaybe:

i. tell :: $[w] \to$ WriterMaybe w (), agrega la información dada.

ii. fail :: WriterMaybe e a, que representa una falla.

d) Se desea utilizar la mónada WriterMaybe para definir un firewall simple que filtre paquetes según reglas. Además de filtrar paquetes el firewall producirá mensajes de logs sobre su actividad.

Dadas las siguientes definiciones que representan los tipos para reglas de filtrado, paquetes y resultados de la aplicación de una regla:

```
type Rule = String
type Packet = String
data Result = Accepted | Rejected
  deriving (Show, Eq)
```

Suponinedo definida una función match :: [Rule] → Packet → [(Rule, Result)], que dada una lista de reglas y un paquete, devuelva una lista con las reglas que matchearon el paquete y el resultado de la aplicación de éstas sobre el paquete; definir una función filterPacket :: [Rule] → Packet → WriterMaybe Char Packet, que determine si un paquete es aceptado por las reglas y agregue mensajes del resultado como logs.

Si un paquete P no machea ninguna regla debe mandar un mensaje de log "UNMATCHED PACKET P" y fallar, sino debe agregar un mensaje "MATCHED P WITH RULE R" por cada regla macheada junto con un mensaje "RULE Accepted" o "RULE Rejected" (dependiendo si la regla acepta o no el paquete). Si todas las reglas aceptan el paquete lo retorna y sino falla.

Ayuda: Puede usar la función mapM :: $(a \to m\ b) \to [a] \to m\ [b]$ definida en práctica.

2. Probar que T es un functor.

```
data T a = Uno a | Dos (a, a) | More (a, a) (T a)
```

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ciencias Exactas y Naturales
Departamento de Ciencia de la Computación
Análisis de Lenguajes de Programación

Nombre y Apellido: FERNANDO FIORI Legajo: F-3239/5

Examen Parcial 1

1. Un lenguaje, según lo visto en la materia, puede ser definido por su sintáxis, su semántica y un sistema de tipos. A continuación se presenta un lenguaje, definiendo estos componentes.

Sintaxis:

$$
\begin{array}{rl}
t & ::= \texttt{true} \mid \texttt{false} \mid \texttt{not}\ t \mid 0 \mid \texttt{suc}\ t \mid \texttt{error} \mid \texttt{catch}\ t\ t \\
v & ::= \texttt{true} \mid \texttt{false} \mid \texttt{error} \mid nv \\
nv & ::= 0 \mid \texttt{suc}\ nv \\
T & ::= \mathbb{N} \mid \mathbb{B}
\end{array}
$$

Sistema de tipos:

$$\texttt{true} : \mathbb{B} \quad \text{(T-True)}$$

$$\texttt{false} : \mathbb{B} \quad \text{(T-False)}$$

$$\texttt{error} : T \quad \text{(T-Error)}$$

$$0 : \mathbb{N} \quad \text{(T-Zero)}$$

$$\frac{t : \mathbb{N}}{\texttt{suc}\ t : \mathbb{N}} \quad \text{(T-Suc)}$$

$$\frac{t : \mathbb{B}}{\texttt{not}\ t : \mathbb{B}} \quad \text{(T-Not)}$$

$$\frac{t_1 : T \quad t_2 : T}{\texttt{catch}\ t_1\ t_2 : T} \quad \text{(T-Catch)}$$

Semántica:

$$\frac{t_1 \to t_1'}{\texttt{catch}\ t_1\ t_2 \to \texttt{catch}\ t_1'\ t_2} \quad \text{(E-Catch0)}$$

$$\frac{v \neq \texttt{error}}{\texttt{catch}\ v\ t_2 \to v} \quad \text{(E-Catch1)}$$

$$\texttt{catch error}\ t_2 \to t_2 \quad \text{(E-Catch2)}$$

$$\frac{t \to t'}{\texttt{suc}\ t \to \texttt{suc}\ t'} \quad \text{(E-Suc)}$$

$$\frac{t \to t'}{\texttt{not}\ t \to \texttt{not}\ t'} \quad \text{(E-Not0)}$$

$$\texttt{not true} \to \texttt{false} \quad \text{(E-Not1)}$$

$$\texttt{not false} \to \texttt{true} \quad \text{(E-Not2)}$$

a) Calcular, según la semántica operacional, a qué valor evalúan los términos:

i. `catch true (not true)`

ii. `suc (catch error (suc 0))`

b) Determinar si los siguientes términos están bien tipados. Si lo están, dar el árbol de derivación correspondiente.

i. `catch (catch (suc 0) 0) (suc (suc 0))`

ii. `not (catch error true)`

c) Explique brevemente cuáles son los teoremas principales (y clásicos) para demostrar la seguridad de un lenguaje según lo visto en clase.

d) Analice si los teoremas valen para el lenguaje presentado: ejemplos de ésto. Para cada teorema justifique informalmente su validez o dé un contrajemplo y proponga modificaciones para solucionar el problema.

2. La representación de los números naturales en el cálculo lambda no tipado mediante la codificación de Church tiene el problema que algunas operaciones (como el predecesor) pueden ser engorrosas. En el Sistema T de Gödel no existe este problema porque el recursor R permite el acceso al predecesor directamente.

$$
\begin{array}{l}
R\ 0 \qquad f\ z = z \\
R\ (n+1)\ f\ z = f\ (R\ n\ f\ z)\ n
\end{array}
$$

a) Obtener una nueva codificación de los naturales que esté basada en R en lugar de *foldn*. Es decir, encontrar λ-expresiones para cero, sucesor y R, tal que cumplan con la especificación de más arriba.

b) Definir la función *pred* para esta codificación.

c) Dadas las siguientes expresiones, reducirlas a su forma β-normal de ser posible, o probar que no poseen forma β-normal.

   i. *pred* (*suc* ω), donde $\omega = (\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$.

   ii. *suc* (*pred cero*).

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ciencias Exactas y Naturales
Departamento de Ciencia de la Computación
Análisis de Lenguajes de Programación

Nombre y Apellido: Legajo:

Examen Parcial 2

1. Se representarán los tipos de un lenguaje con el siguiente tipo de datos:

```
data Type v = TVar v                    -- Variable de tipo
            | TInt                      -- Tipo Integer
            | Fun (Type v) (Type v)     -- Tipo de función
```

a) Dar una instancia de *Functor* para *Type*.

b) Para modelar la sustitución de variables se definió la siguiente instancia:

```
instance Monad Type where
  return v       = TVar v
  TVar v >>= f   = f v
  TInt   >>= f   = TInt
  Fun d r >>= f  = Fun (d >>= f) (r >>= f)
```

Es decir, que *result* corresponde a la sustitución nula (no hay sustitución de variables), y $t \gg\!= s$ corresponde al resultado de aplicar la sustitución s al término t, reemplazando cada ocurrencia de una variable v en t con el término $s\ v$.

Probar que *Type* es una mónada.

c) Definir las siguientes funciones:

i. $apply :: (a \to Type\ b) \to Type\ a \to Type\ b$, que aplique una sustitución a un tipo.

ii. $comp :: (a \to Type\ b) \to (c \to Type\ a) \to (c \to Type\ b)$, que permita componer sustituciones.

iii. $(\ggg) :: Eq\ v \Rightarrow v \to Type\ v \to (v \to Type\ v)$, tal que $v \ggg t$ sea una función que mapee la variable v con el término t y deje las demás variables sin modificar.

iv. $varBind :: Eq\ v \Rightarrow v \to Type\ v \to Maybe\ (v \to Type\ v)$, que dada una variable v y un término t, falle si v es una variable de t, o devuelva la sustitución $v \ggg t$ si v no está en t.

d) La mónada sustitución puede utilizarse para implementar de manera concisa un *algoritmo de unificación* de tipos para valores de tipo *Type* v.

Dados dos tipos $t_1, t_2 :: Type\ v$, la *unificación* de t_1 con t_2 es una sustitución de variables $\sigma :: v \to Type\ v$ que al aplicarse sobre t_1 y t_2 los hace iguales. Es decir, la unificación de t_1 y t_2 es un σ tal que

$$apply\ \sigma\ t_1 = apply\ \sigma\ t_2$$

Notar que algunos términos pueden unificarse, pero otros no. Por ejemplo, considere la siguiente tabla con los resultados de una implementación de unificación sobre ciertos tipos:

$unify\ x$	y	$= [x \mapsto y]$
$unify\ (Int \to Int)$	$(x \to Int)$	$= [x \mapsto Int]$
$unify\ (x \to y)$	$(a \to b)$	$= [x \mapsto a, y \mapsto b]$
$unify\ (x \to (y \to z))$	$((a \to b) \to c)$	$= [x \mapsto (a \to b), c \mapsto (y \mapsto z)]$
$unify\ (x \to y)$	Int	= No existe substitución que unifique
$unify\ (x \to x)$	y	$= [y \mapsto (x \to x)]$
$unify\ x$	$(x \to y)$	= No existe substitución que unifique

Notar que en el último ejemplo la unificación falla ya que, si se unificase una variable con un término que contiene dicha variable, se obtendría un tipo infinito.

Definir una función $unify :: Type\ v \to Type\ v \to Maybe\ (v \to Type\ v)$, que implemente la unificación de tipos. Utilizar la función *varBind* para el caso de unificación de una variable con un término.

Ayuda: Las funciones definidas en el apartado anterior pueden ser de utilidad. Si así lo desea, puede usar la instancia de *Monad* para *Maybe* para manejar errores:

```
instance Monad Maybe where
  return a        = Just a
  Nothing >>= f   = Nothing
  Just x  >>= f   = f x
```

2. Sean D y E dominios, $f : D \to E$ y $g : E \to D$ funciones continuas, f estricta. Probar que:

$$\bigsqcup_{i=0}^{\infty} (g \cdot f)^n \perp = fix\,(g \cdot f) = g\,(fix\,(f \cdot g)) = g \cdot \bigsqcup_{i=0}^{\infty} (f \cdot g)^n \perp$$

(1)

(1) $\bigsqcup_{i=0}^{\infty} (g \cdot f)\perp = \perp \sqsubseteq (g \cdot f)\perp \sqsubseteq (g \cdot f)((g \cdot f)\perp) \sqsubseteq \ldots$

$= \perp \sqsubseteq g(f\perp) \sqsubseteq (g \cdot f)(g(f\perp)) \sqsubseteq \ldots$

(f estricta) $= \perp \sqsubseteq g\perp \sqsubseteq (g \cdot f)(g\perp) \sqsubseteq \ldots$ ($(g \cdot f)(g\perp) = g((f \cdot g)\perp)$)

(2) $g\left(\bigsqcup_{i=0}^{\infty} (f \cdot g)\perp\right) = g \cdot (\perp \sqsubseteq f g \perp \sqsubseteq \ldots)$

(g continua) $= g\perp \sqsubseteq g((f g)\perp) \sqsubseteq \ldots$

($\perp$ es el mínimo) $= \perp \sqsubseteq g\perp \sqsubseteq g((f g)\perp) \sqsubseteq \ldots$

$g \bigsqcup_n = \bigsqcup_n g (f g)^n \perp = \bigsqcup_n g (f \cdot g)^n (f\perp)$

$= \bigsqcup_n (g f)^{n+1} \perp$