Explicación - Algoritmo W - HM

```haskell
data Type
  = TInt
  | TBool
  | TVar String
  | TFun [Type] Type
  deriving (Eq, Show)

type TEnv = [(String, Type)]

---------------------------------------------------
-- Sustituciones
---------------------------------------------------

type Subst = [(String, Type)]

nullSubst :: Subst
nullSubst = []

-- aplica una sustitución a un Type
class Types a where
  apply :: Subst -> a -> a
  ftv   :: a -> [String]

instance Types Type where
  apply s (TVar v) = case lookup v s of
                       Just t  -> t
                       Nothing -> TVar v
  apply s (TFun ts r) = TFun (map (apply s) ts) (apply s r)
  apply _ t = t

  ftv (TVar v)      = [v]
  ftv (TFun ts r)   = concatMap ftv ts ++ ftv r
  ftv _             = []

instance Types a => Types [a] where
  apply = map . apply
  ftv   = concatMap ftv
```

Tipos

Ambiente

sustituciones, contexto, ...

contexto vacío.

→ Sust. en una exp y return expr. con sust.

→ exp. return string: free type variables

No cambia por contexto

tipo int o bool.

contexto

return

parámetros

→ una variable sola es free type variable

param.

return

What is and isn't free can be calculated

$FV(x) = \{x\}$ [variable]

$FV(e_1\ e_2) = FV(e_1) \cup FV(e_2)$ [function application]

```haskell
-- aplicar sustitución a entorno
applyEnv :: Subst -> TEnv -> TEnv
applyEnv s env = [(v, apply s t) | (v,t) <- env]

-- composición: s1 `compose` s2 significa aplicar s1 luego s2 efectivamente
compose :: Subst -> Subst -> Subst
compose s1 s2 = [(v, apply s1 t) | (v,t) <- s2] ++ s1
```

→ aplicamos sustituciones al ambiente.

$S_1 = \{ \alpha \mapsto \gamma,\ \beta \mapsto \delta \}$
$S_2 = \{ \alpha \mapsto \beta \}$

s	$S_2(s)$	$S_1(S_2(s))$	
α	β	δ	**α ↦ δ**
β	β	δ	**β ↦ δ**

variable   tipo   sust. / contexto

```haskell
bindVar :: String -> Type -> Infer Subst
bindVar v t
  | t == TVar v          = return nullSubst
  | v `elem` ftv t       = throwError ("Ocurre-check falló para " ++ v)
  | otherwise            = return [(v,t)]
```

→ "a" (VarT "a") = [ ] → sin cambio en el contexto.

→ [("a", TInt)]

$a = \alpha \to Int$
$b = \alpha$

$S = \{ \alpha \mapsto Int \to \ldots \to Int \}$
$S(a) = S(b)$
$\quad = Int \to \ldots \to Int \to Int$

```haskell
unify (TVar v) t = bindVar v t
unify t (TVar v) = bindVar v t
unify TInt TInt   = return nullSubst
unify TBool TBool = return nullSubst
```

⇒ v es string y t es tipo ⇒ return en contexto

return [ ] en el contexto / sust.

```haskell
unify :: Type -> Type -> Infer Subst
```

param return

```haskell
unify :: Type -> Type -> Infer Subst
unify (TFun as r) (TFun bs r2)
  | length as == length bs = do
      s1 <- unifyMany as bs
      s2 <- unify (apply s1 r) (apply s1 r2)
      return (s2 `compose` s1)
  | otherwise = throwError "Aridad de función incorrecta"
```

→ si length de params es igual → proceda...

unifica los params y aplica a los return

→ return contexto

$$a = Int \to \alpha$$
$$b = \beta \to Bool$$

```haskell
data Type
  = TInt
  | TBool
  | TVar String
  | TFun [Type] Type
  deriving (Eq, Show)
```

lista de param

un solo tipo en el return

$$S = \{ \alpha \mapsto Bool, \beta \to Int \}$$
$$S(a) = S(b) = Int \to Bool$$

```haskell
unifyMany :: [Type] -> [Type] -> Infer Subst
unifyMany [] [] = return nullSubst
unifyMany (t:ts) (u:us) = do
  s1 <- unify t u
  s2 <- unifyMany (map (apply s1) ts) (map (apply s1) us)
  return (s2 `compose` s1)
unifyMany _ _ = throwError "Listas de distinto tamaño en unifyMany"
```

$$\mathcal{W}: TypEnv \times Expr \to Subst \times Type$$

```haskell
inferExpr :: TEnv -> ExprS -> Infer (Subst, Type)
```

```haskell
inferExpr env (IdS x) =
  case lookup x env of
    Just t  -> return (nullSubst, t)
    Nothing -> throwError ("Variable no definida: " ++ x)
```

return [], t

t algún tipo.

$$\mathcal{W}(\Gamma, x) = (id, \{\vec{\beta}/\vec{\alpha}\}\tau) \quad \text{where } \Gamma(x) = \forall\vec{\alpha}.\tau, \ \text{new } \vec{\beta}$$

Ambiente provisional.

```haskell
inferExpr env (LambdaS [x] body) = do
  tv <- fresh
  -- inferir el cuerpo con x : tv
  let env' = (x, tv) : env
  (s1, tBody) <- inferExpr env' body
  -- el tipo del parámetro puede haber sido refinado por s1
  let paramType = apply s1 tv
  return (s1, TFun [paramType] tBody)
```

← variable tipo $t_1, t_2, \ldots$ (genérica).

← en el amb. (x, t_0) : env.

$$\mathcal{W}(\Gamma, \lambda x.e) = \text{let } (S_1, \tau_1) = \mathcal{W}(\Gamma + x{:}\beta, e), \ \text{new } \beta \quad \text{in } (S_1, S_1\beta \to \tau_1)$$

(lambda (x) (+ x 1))

tv

env' = [("x", t0)]

inferExpr env' (+ x 1)

return ({t0 ↦ Int}, Int)

s1 tbody

apply s1 tv = Int ← paramtype.

return (s1, TFun [paramType] tBody)

ambiente nuevo parametros return.

```haskell
inferExpr env (AppS f args) = do
  -- inferir f
  (sF, tF) <- inferExpr env f
  -- inferir args secuencialmente aplicando s acumulada al entorno
  (sArgs, tArgs) <- foldM go (sF, []) args
  -- fresh para resultado
  tRes <- fresh
  -- unificar (aplicar sArgs a tF)
  sUnify <- unify (apply sArgs tF) (TFun tArgs tRes)
  let sTotal = sUnify `compose` sArgs
  return (sTotal, apply sUnify tRes)
  where
    go :: (Subst, [Type]) -> ExprS -> Infer (Subst, [Type])
    go (sAcc, tsAcc) e = do
      let env' = applyEnv sAcc env
      (sE, tE) <- inferExpr env' e
      let sNew = sE `compose` sAcc
      return (sNew, tsAcc ++ [apply sNew tE])
```

$\underbrace{((\text{lambda (f) (f 10))}}_{f} \underbrace{(\text{lambda (x) (+ x 1)))}}_{args.}$

f: Int → α

f 10 : α.

(sF, tF) <- inferExpr env f

(Int -> α) -> α   t1

sF = { t0 ↦ Int -> t1 }

tF = TFun [Int -> t1] t1

(sArgs, tArgs) <- foldM go (sF, []) args

sAcc = sF = { t0 ↦ Int -> t1 }   e = (lambda (x) (+ x 1))

tsAcc = []

env' = applyEnv sAcc env   env' = []

sE = { t2 ↦ Int }

tE = TFun [Int] Int

```
sNew = sE `compose` sAcc

sNew = { t2 ↦ Int } ∘ { t0 ↦ Int -> t1 }

sNew = { t0 ↦ Int -> t1 , t2 ↦ Int }

tsAcc ++ [apply sNew tE]

  return (sNew = { t0 ↦ Int -> t1 , t2 ↦ Int }
  , tE = TFun [Int] Int)


sArgs = { t0 ↦ Int -> t1 , t2 ↦ Int }
tArgs = [ TFun [Int] Int ]

 return
  ( { t0 ↦ Int -> Int
    , t1 ↦ Int
    , t2 ↦ Int
    , t3 ↦ Int
    }
  , Int
  )
```