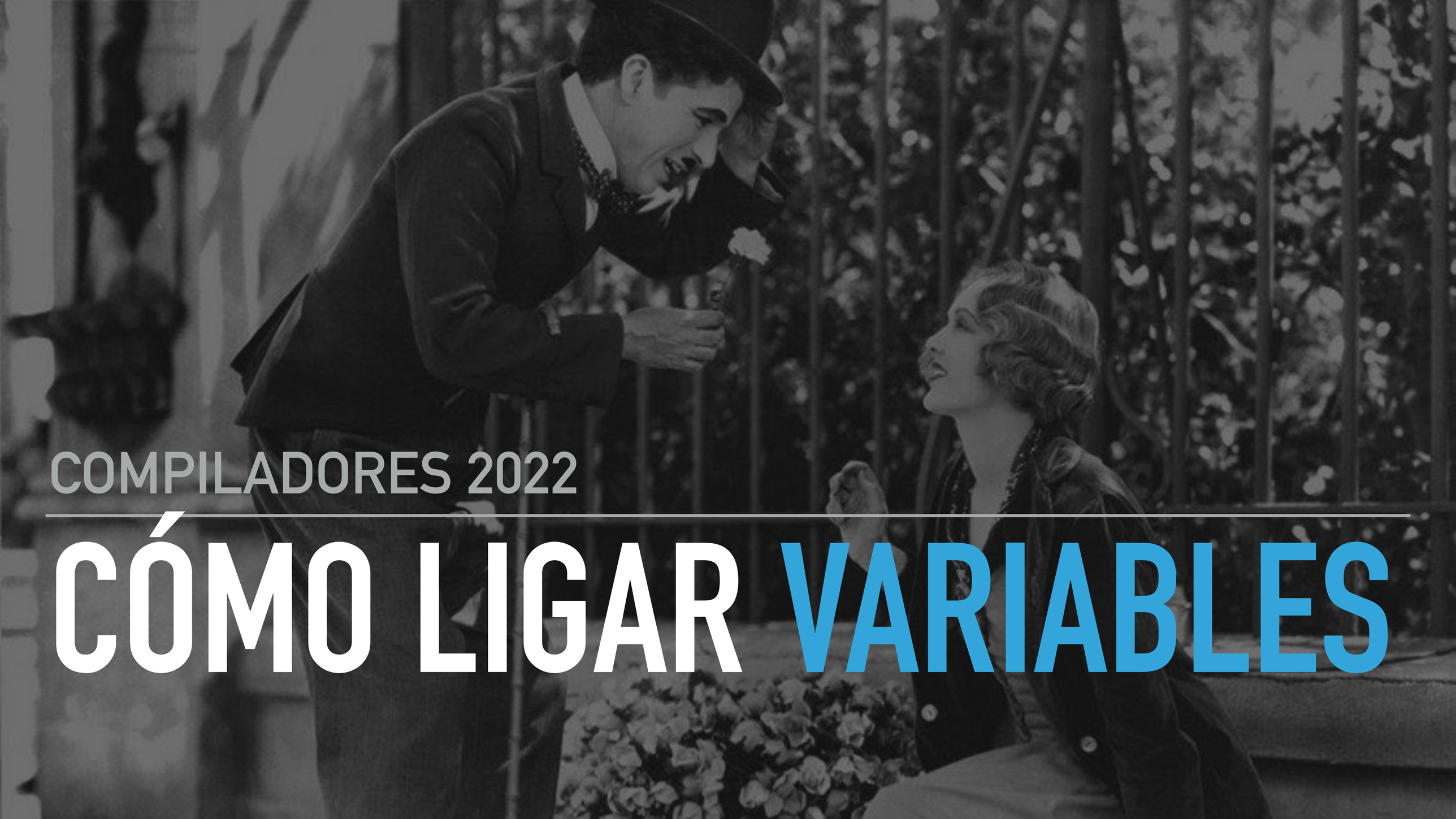


COMPILADORES 2022

CÓMO LIGAR VARIABLES

REPRESENTACIÓN DE VARIABLES LIGADAS

- La mayoría de los lenguajes tienen alguna forma de binder.
- Una buena representación es importante para que el software sea correcto.
- En papel, es fácil. En forma automática, es complicado.

VARIABLES CON NOMBRES

- La forma más obvia de representar términos:

```
type Name = String

data Term = Lam Name Term
          | App Term Term
          | Var Name

idTerm :: Term
idTerm = Lam "x" (Var "x")
```

- Fácil de escribir términos 👍

PROBLEMA

- La representación no es canónica:
  dos términos α-equivalentes no son iguales:

```
idTerm :: Term
idTerm = Lam "x" (Var "x")

idTerm' :: Term
idTerm' = Lam "y" (Var "y")
```

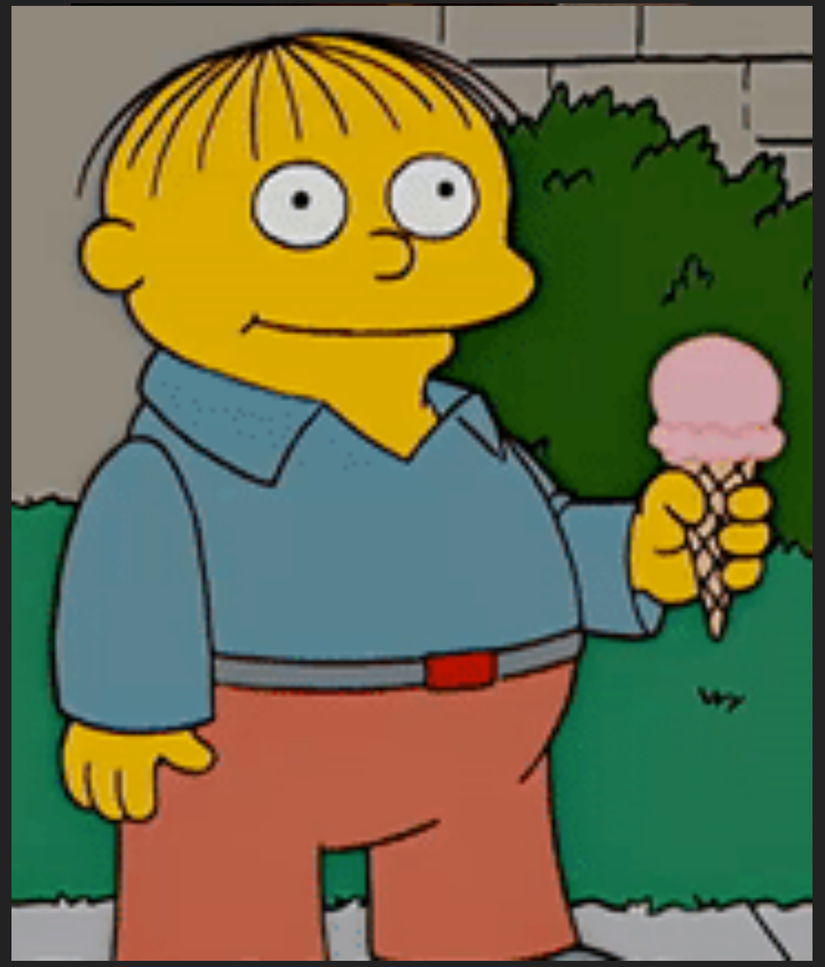
- Los términos idTerm y idTerm’ no son iguales a pesar de ser α-equivalentes 👎

MÁS PROBLEMAS

- La substitución es complicada ya que debemos:
  - analizar las variables libres para no capturarlas,
  - para saber las variables libres debemos recorrer todo el término,
  - hacer renombre de variables ligadas,
  - para renombrar necesitamos nombres frescos.

REPRESENTACIÓN CON NOMBRES

- Es la representación más simple e intuitiva
- Pero es muy fácil meter la pata



ALTERNATIVA: NO USAR NOMBRES

- De Bruijn introdujo dos formas de representar variables ligadas sin usar nombres:
  - Índices
  - Niveles
- Ambas tienen representación canónica de los términos

ÍNDICES DE DE BRUIJN

- Los binders no llevan nombre
- Las variables ligadas son naturales que cuentan cuántos λ saltar hasta llegar a su binder
- 0 se refiere al λ más cercano, 1 al λ siguiente:

$$\lambda x\,.\,\lambda y\,.\,x \quad \mapsto \quad \lambda\lambda 1$$

$$\lambda x\,.\,\lambda y\,.\,x\ (\lambda z\,.\,x\ z) \quad \mapsto \quad \lambda\lambda 1(\lambda 2\ 0)$$

- No es una forma intuitiva de escribir términos 👎

ÍNDICES DE DE BRUIJN

- Los podemos implementar:

```haskell
data Term = Lam Term
          | App Term Term
          | Var Int

idTerm :: Term
idTerm = Lam (Var 0)
```

- La representación es canónica 👍

ÍNDICES DE DE BRUIJN

- Para la substitución necesitamos desplazar índices

```
shift :: Int -> Term -> Term
shift d t = go 0 t where
        go c (Var k) | k < c      = Var k
                     | otherwise = Var (k+d)
        go c (App t u) = App (go c t) (go c u)
        go c (Lam t)   = Lam (go (c+1) t)
```

- `shift d t` desplaza en `d` posiciones los índices "libres" en el término `t`

ÍNDICES DE DE BRUIJN

- La substitución queda:

```
subst :: Int -> Term -> Term -> Term
subst j s (Var k)   | j==k       = s
                    | otherwise = Var k
subst j s (App t u) = App (subst j s t) (subst j s u)
subst j s (Lam t)   = Lam (subst (j+1) (shift 1 s) t)
```

- No hace falta buscar nombres frescos

ÍNDICES DE DE BRUIJN

- Al transformar árboles hay que tener cuidado con los índices:

```
f :: Term -> Code
f (Var i) = …
f (App t u) = … (f t) … (f u)
f (Lam t) = … (f (shift 1 t)) …
```

¿LO MEJOR DE DOS MUNDOS?

- Usar nombre es intuitivo, pero es peligroso
- Usar índices nos da canonicidad y seguridad, pero hay que lidiar con índices
- La representación locally nameless combina estas dos ventajas, minimizando las desventajas
- Las variables ligadas con índice, las libres con nombre.

```
data Term = Free Name
          | Bound Int
          | Lam Scope
          | App Term Term

newtype Scope = S Term
```

TÉRMINOS LOCALMENTE CERRADOS Y SCOPES

- Evitamos lidiar con índices trabajando con términos localmente cerrados
- Un término se dice localmente cerrado (LC) cuando sus índices no escapan a sus binders.
  - $\lambda\lambda 1$ es LC ✓
  - $\lambda\lambda 2$ no es LC ✗
- Si tenemos variables que escapan a sus binders éstas deben ser libres, y por lo tanto, tener nombre.
- Usamos los Scope para encapsular un término que no es LC y por lo tanto debe ser dado un nombre antes de poder ser operado.

ABRIENDO TÉRMINOS

- Si λt es LC, el t es un Scope y debe ser abierto para obtener un término.
- Obtenemos un término LC si abrimos t dándole un nombre a esa variable que escapa.

```
open :: Name -> Scope -> Term
open n (S t) = go 0 t where
   go j (Free n)             = Free n
   go j (Bound i) | j == i = Free n
                  | i < j  = Bound i
   go j (App t u)            = App (go j t) (go j u)
   go j (Lam (S t))          = Lam (S (go (j+1) t))
```

- open x (λλ2) = λλx
- Para evitar capturas, un t debe ser abierto con un nombre que no esté libre en t.
  - No hace falta recorrer t, alcanza con recordar los nombres usados al abrir.

CERRANDO TÉRMINOS

- Se puede **cerrar** una variable **x** en un término t y obtener un Scope reemplazando **x** por el índice correspondiente:

```
close :: Name -> Term -> Scope
close x t = S (go 0 t) where
   go j (Bound i)             = Bound i
   go j (Free n) | x == n     = Bound j
                 | otherwise = Free n
   go j (App t u)             = App (go j t) (go j u)
   go j (Lam (S t))           = Lam (S (go (j+1) t))
```

- Es la operación **inversa** a open:
  - close z (λλz) = λλ2

SUBSTITUCIÓN

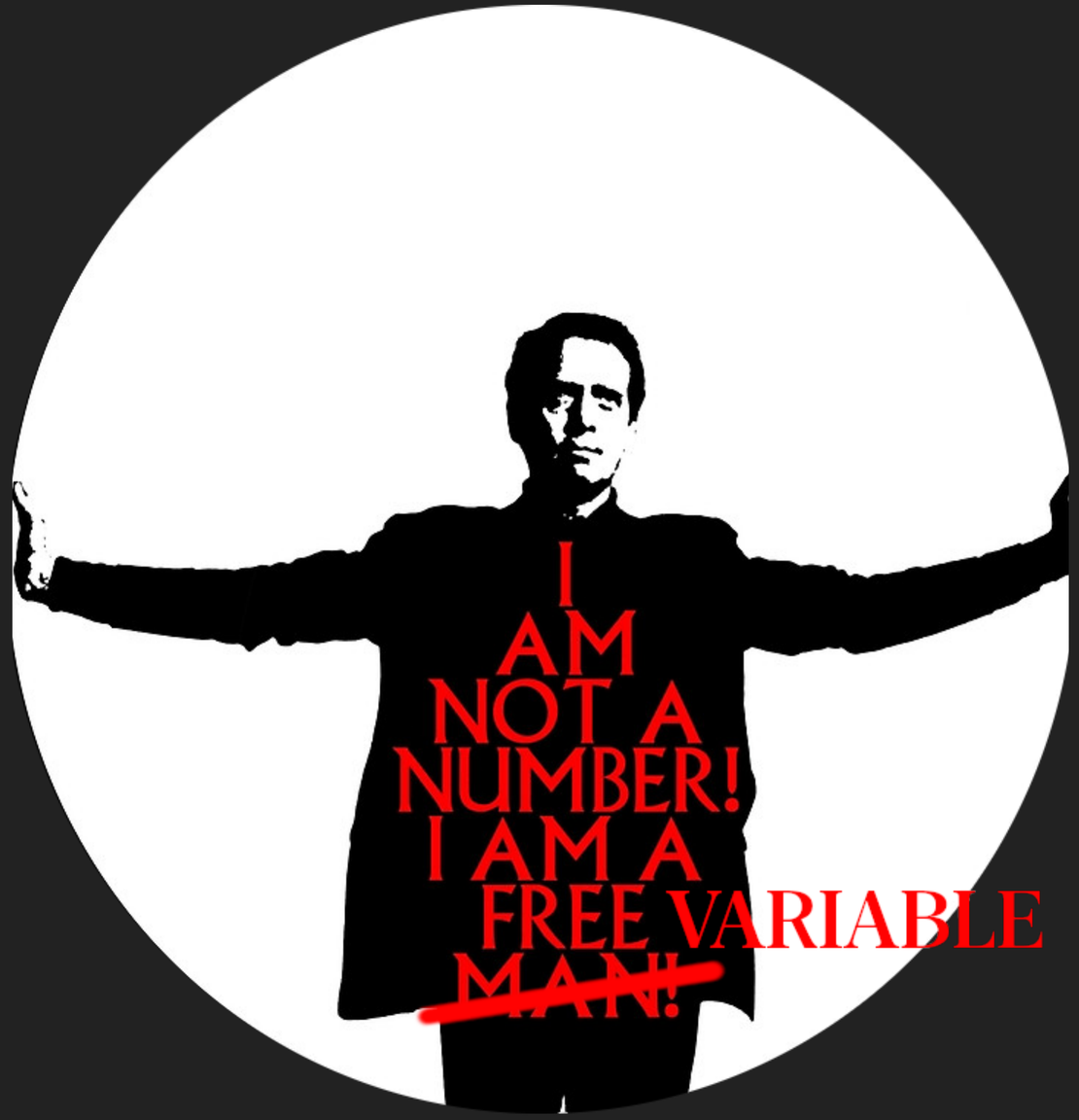
- La substitución es análoga a abrir un Scope con una variable fresca x y luego reemplazar esa variable x por un término u (no hay posibilidad de captura!)
- Haciendo esto en un solo paso obtenemos:

```
subst :: Term -> Scope -> Term
subst u (S t) = go 0 t where
   go j (Free n)           = Free n
   go j (Bound i) | j == i = u
                  | i < j  = Bound i
   go j (App t u)          = App (go j t) (go j u)
   go j (Lam (S t))        = Lam (S (go (j+1) t))
```

- Comparar con open

REPRESENTACIÓN LOCALMENTE SIN NOMBRES

- Nunca es necesario encontrarse con un índice
- Les podemos dar nombres a las variables
- Las operaciones open y close manejan los índices por nosotros



HIGHER ORDER ABSTRACT SYNTAX (HOAS)

- En lenguajes con alto orden se puede usar la ligadura del lenguaje anfitrión

```
data Term = Lam (Term -> Term)
          | App Term Term

idTerm :: Term
idTerm = Lam (\x -> x)
```

- Notar que el tipo Term es de alto orden.
- No hay variables. Las variables las provee el lenguaje anfitrión.
- Contiene elementos foráneos (por ejemplo una función que nos dice si su argumento es una abstracción). 👎

HIGHER ORDER ABSTRACT SYNTAX (HOAS)

- La evaluación de un término es fácil 👍

```
eval :: Term -> Term
eval (Lam t) = Lam t
eval (App f x) = eval f `app` eval x
 where app (Lam f) x = f x
```

- Inspeccionar un término es difícil 👎

```
countLam :: Term -> Int
countLam (App f x) = countLam f + countLam x
countLam (Lam f) = 1 + …
```

PARAMETRIC HIGHER ORDER ABSTRACT SYNTAX (PHOAS)

- Parametrizamos el tipo por el tipo de variable

```haskell
data Term a = Lam (a -> Term a)
            | App (Term a) (Term a)
            | Var a

idTerm :: Term a
idTerm = Lam (\a -> Var a)

type ClosedTerm = forall a. Term a
```

- No tiene elementos foráneos 👍

PARAMETRIC HIGHER ORDER ABSTRACT SYNTAX (PHOAS)

- Permite la inspección de términos 👍

```
countLam :: Term Int -> Int
countLam (Var n) = n
countLam (App t u) = countLam t + countLam u
countLam (Lam t) = 1 + countLam (t 0)
```

- No es tan fácil de manipular como un árbol de primer orden 👎

REPRESENTANDO VARIABLES LIGADAS

- Primer Orden
  - Con nombres
  - Sin nombres
  - Localmente sin nombres
- Alto Orden
  - HOAS
  - PHOAS