Llenguatges de Programació

Inferència de tipus

Jordi Petit, Albert Rubio

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Contingut

- Introducció
- Algorisme de Milner
- Exercicis
- Funcions
- Classes
- Errors
- Exercicis

Inferència de tipus

La **inferència de tipus** és la detecció automàtica dels tipus de les expressions en un llenguatge de programació.

Permet fer més fàcils moltes tasques de programació, sense comprometre la seguretat de la comprovació de tipus.

Té sentit en llenguatges fortament tipats.

És un característica habitual dels llenguatges funcionals.

Alguns LPs amb inferència de tipus:

- C++ >= 11
- Haskell
- C#
- D
- Go
- Java >= 10
- Scala
- ...

Inferència de tipus a C++

La inferència de tipus apareix a la versió 11 de l'estàndard de C++.

- `auto`: Dedueix el tipus d'una variable a través de la seva inicialització:

```
map<int, string> m;
auto x = 12;                // x és un int
auto it = m.find(x);        // x és un map<int, string>::iterator
```

- `decltype`: Obté el tipus d'una expressió.

```
int x = 12;
decltype(x + 1) y = 0;      // y és un int
```

Inferència de tipus a Haskell

- En la majoria de casos no cal definir els tipus.

- Es poden demanar els tipus inferits (que inclouen classes, si cal).

```
λ> :type 3 * 4
👉 3 * 4 :: Num a => a

λ> :type odd (3 * 4)
👉 odd (3 * 4) :: Bool
```

- Algunes situacions estranyes.
    - *Monomorphism restriction*: Sovint no es pot sobrecarregar una funció si no es dona una declaració explícita de tipus.

Inferència de tipus

Problema: Donat un programa, trobar el tipus més general de les seves expressions dins del sistema de tipus del LP.

Solució presentada: Algorisme de Milner.

Contingut

- Introducció
- Algorisme de Milner
- Exercicis
- Funcions
- Classes
- Errors
- Exercicis

Inferència de tipus

Algorisme de Milner

- Curry i Hindley havien desenvolupat idees similars independentment en el context del λ-càlcul.
- Hindley–Milner i Damas–Milner
- L'algorisme és similar a la "unificació".

Foto: Domini públic

Propietats

- Complet.
- Computa el tipus més general possible sense necessitat d'anotacions.
- Eficient: gairebé lineal (inversa de la funció d'Ackermann). L'eficiència depèn de l'algorisme d'unificació que s'apliqui.

Algorisme de Milner

Descripció general

1. Es genera l'arbre de sintaxi de l'expressió (currificant totes les aplicacions).

2. S'etiqueta cada node de l'arbre amb un tipus:

    - Si el tipus és conegut, s'etiqueta amb aquest tipus.
    - Altrament, s'etiqueta amb una nova variable de tipus.

3. Es genera un conjunt de restriccions (d'igualtat principalment) a partir de l'arbre de l'expressió i les operacions que hi intervenen:

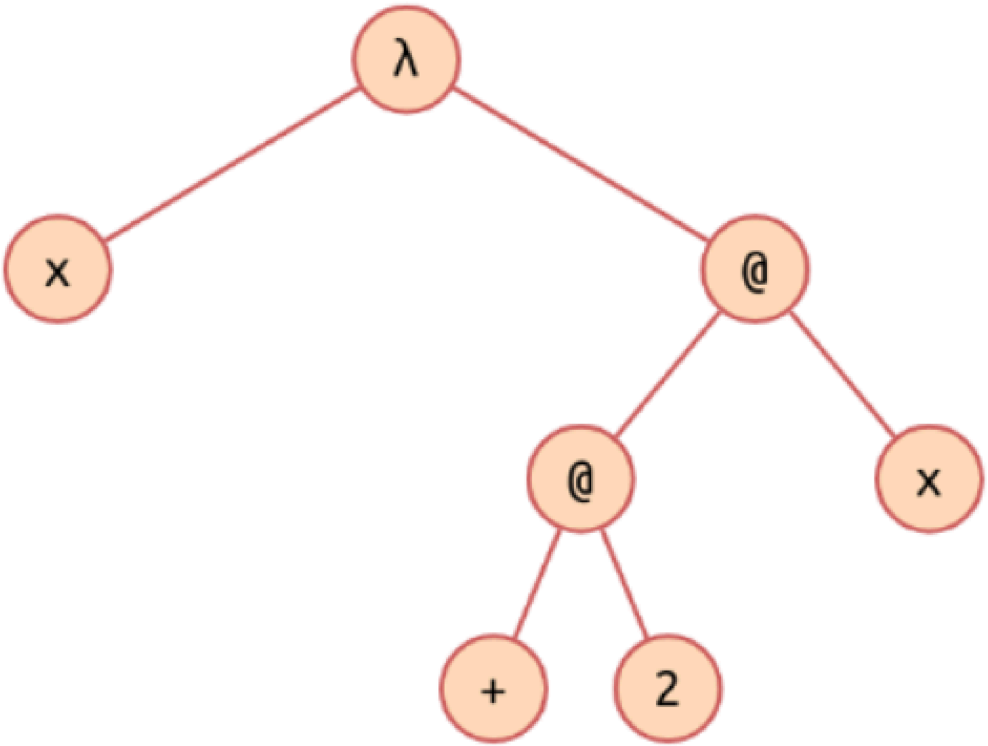
    - Aplicació,
    - Abstracció,
    - `let`, `where`,
    - `case`, guardes, patrons,
    - ...

4. Es resolen les restriccions mitjançant unificació.

Primer exemple

```
\x -> (+) 2 x
```

Arbre de l'expressió currificada:

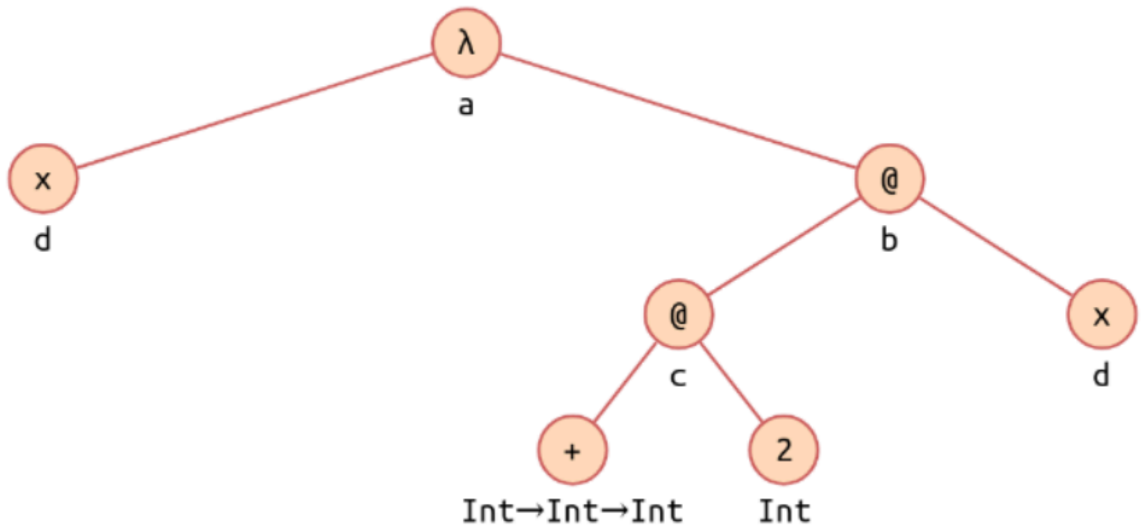


Primer exemple

```
\x -> (+) 2 x
```

Etiquetem els nodes:

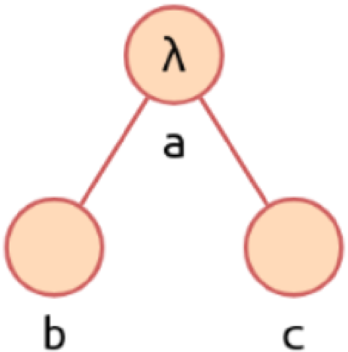
- Si el tipus és conegut, se'ls etiqueta amb el seu tipus.
- Altrament, se'ls etiqueta amb una nova variable de tipus.
- Nodes iguals han de tenir etiquetes iguals.



Algorisme de Milner

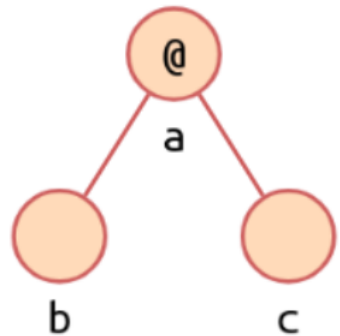
Regles per generar les equacions

- Abstracció:



- Equació: `a = b → c`

- Aplicació:



- Equació: `b = c → a`

Primer exemple

- Obtenim les equacions:

```
a = d → b
c = d → b
Int → Int → Int = Int → c
```

- Solucionem les equacions:

```
a = Int → Int
b = Int
c = Int → Int
d = Int
```

- El tipus de l'expressió és el de l'arrel (`a`):

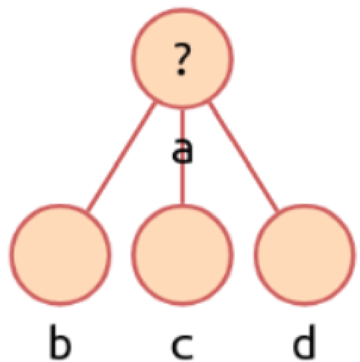
```
\x -> (+) 2 x :: Int → Int
```

- Recordeu: `->` associa per la dreta: `a → b → c = a → (b → c)`
- Recordeu: aplicació associa per l'esquerra': `f x y = (f x) y`

Algorisme de Milner

Més regles per generar les equacions

- Condicional `if-then-else`:



- Equacions:
  - `b = Bool`
  - `a = c = d`

Aquesta regla no és estrictament necessària, ja que `if-then-else` només és una funció genèrica normal de tipus `Bool → a → a → a`, però estalvia espai.

Segon exemple

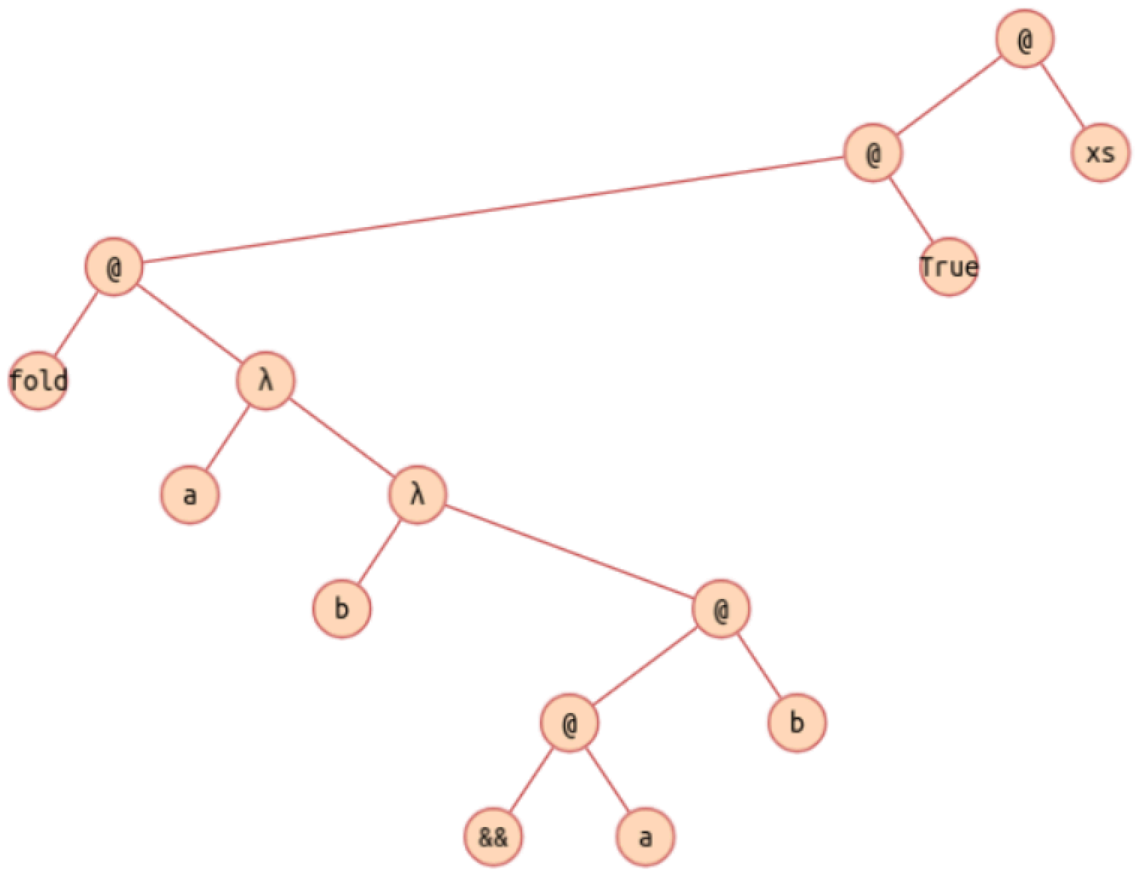
```
foldl (\a b -> a && b) True xs
```

Creeu l'arbre de l'expressió currificada...

Segon exemple

```
foldl (\a b -> a && b) True xs
```

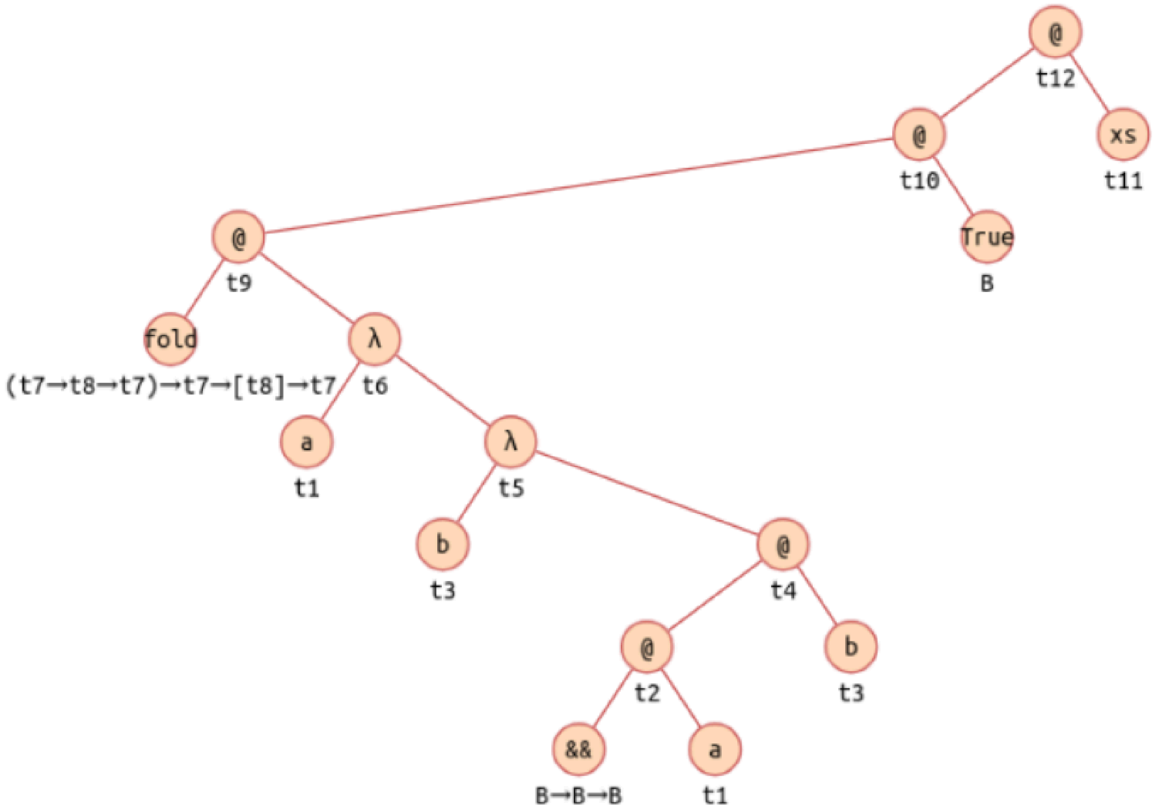
Arbre de l'expressió currificada:



Segon exemple

```haskell
foldl (\a b -> a && b) True xs
```

Arbre etiquetat amb tipus: (`B` és `Bool`)

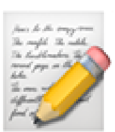


Segon exemple — Us toca!

Equacions:

- `t1 = Bool`
- `t2 = Bool → Bool`
- `t3 = Bool`
- `t4 = Bool`
- `t5 = Bool → Bool`
- `t6 = Bool → Bool → Bool`
- `(t7 → t8 → t7) → t7 → [t8] → t7 = (Bool → Bool → Bool) → t9`
- `t9 = Bool → t10`
- `t10 = t11 → t12`

Solucioneu… (volem `t12`)



Segon exemple

Equacions:

- `t1 = Bool`
- `t2 = Bool → Bool`
- `t3 = Bool`
- `t4 = Bool`
- `t5 = Bool → Bool`
- `t6 = Bool → Bool → Bool`
- `(t7 → t8 → t7) → t7 → [t8] → t7 = (Bool → Bool → Bool) → t9`
- `t9 = Bool → t10`
- `t10 = t11 → t12`

Solució:

- `t7 = Bool`
- `t8 = Bool`
- `t9 = Bool → [Bool] → Bool`
- `t10 = [Bool] → Bool`
- `t11 = [Bool]`
- `t12 = Bool` (arrel de l'expressió)

Contingut

Exercicis

- Utilitzeu l'algorisme de Milner per inferir el tipus de:

```
2 + 3 + 4
```

- Utilitzeu l'algorisme de Milner per inferir el tipus de:

```
2 + 3 <= 2 + 2
```

- Utilitzeu l'algorisme de Milner per inferir el tipus de:

```
map (* 2)
```

  (Suposeu `(*) :: Int -> Int -> Int`)

- Utilitzeu l'algorisme de Milner per inferir el tipus de:

```
foldl (flip (:)) []
```

Exercicis

- Utilitzeu l'algorisme de Milner per inferir el tipus de:

```
\f x -> f $ f x
```

- Utilitzeu l'algorisme de Milner per inferir el tipus de:

```
\f -> f . f
```

- Utilitzeu l'algorisme de Milner per inferir el tipus de:

```
\x y -> if y /= 0 then Just (x `div` y) else Nothing
```

  (Suposeu `div :: Int -> Int -> Int`)

- Utilitzeu l'algorisme de Milner per inferir el tipus de:

```
\xs ys -> zipWith (,) xs ys
```

Contingut

- Introducció
- Algorisme de Milner
- Exercicis
- Funcions
- Classes
- Errors
- Exercicis

Definició de funció

```
map f l = if null l then [] else f (head l) : map f (tail l)
```

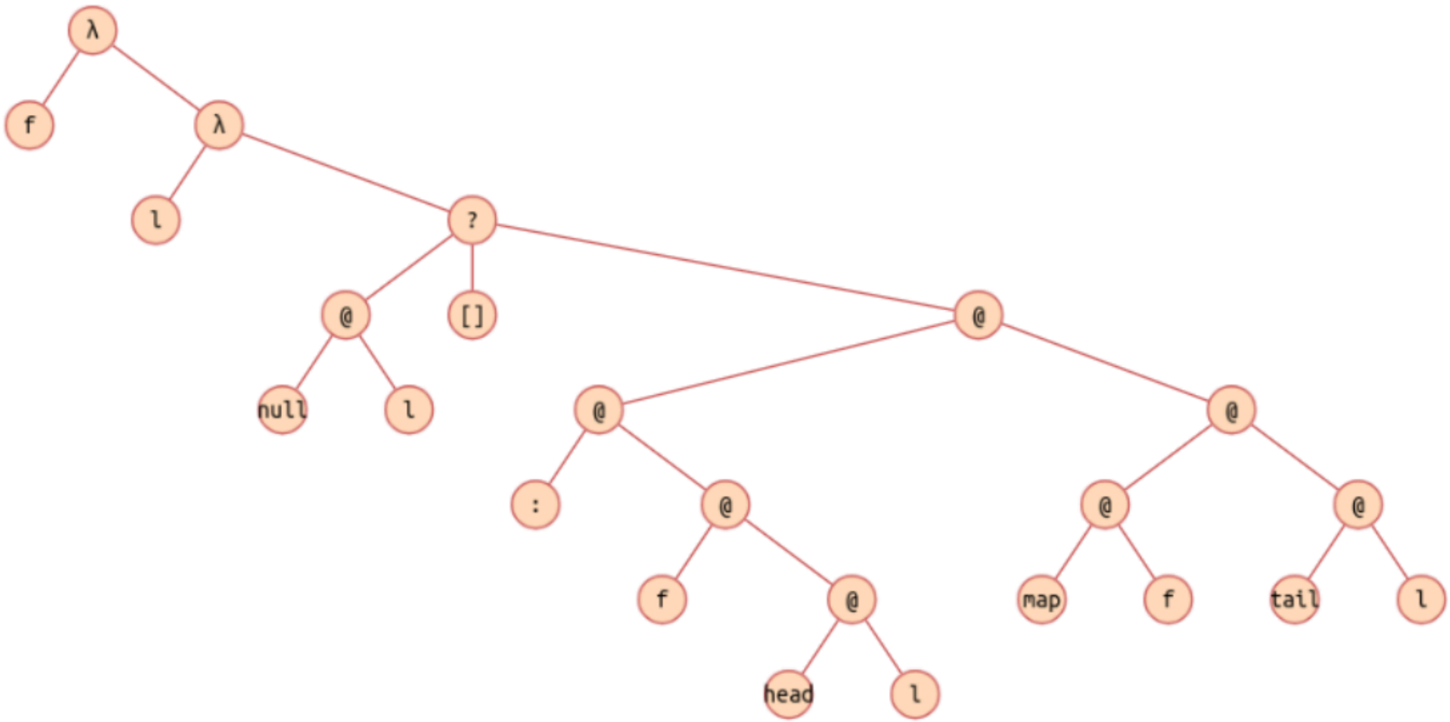
Podem entendre una definició com una funció que, aplicada als paràmetres, torna la part dreta de la definició:

```
\f -> \l -> if null l then [] else f (head l) : map f (tail l)
```

Definició de funció

```
\f -> \l -> if null l then [] else f (head l) : map f (tail l)
```

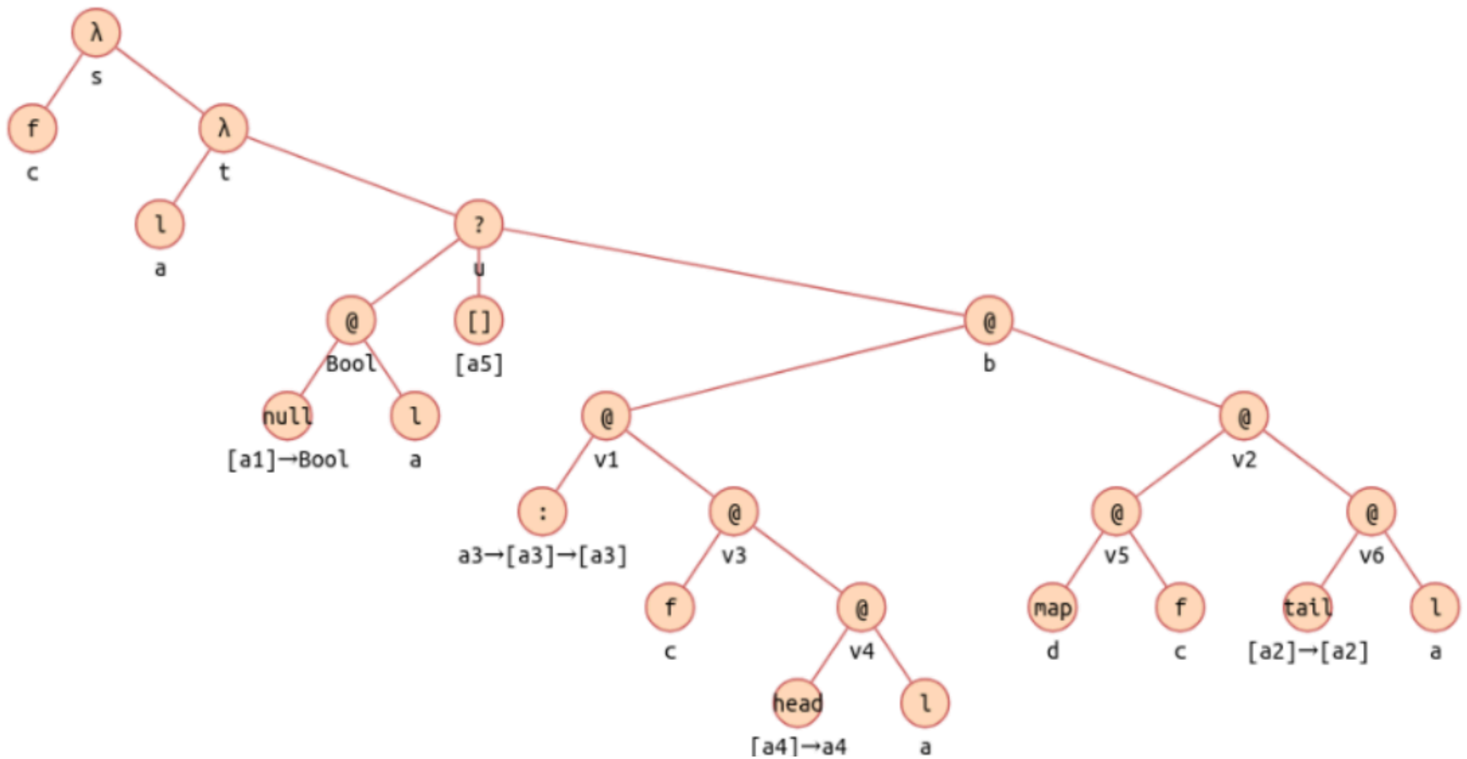
Arbre de l'expressió:



Definició de funció

```
\f -> \l -> if null l then [] else f (head l) : map f (tail l)
```

Arbre etiquetat amb tipus:



Definició de funció

```
\f -> \l -> if null l then [] else f (head l) : map f (tail l)
```

Equacions:

- `s = c → t`
- `t = a → u`
- `u = [a5]`
- `u = b`
- `[a1] → Bool = a → Bool`
- `v1 = v2 → b`
- `a3 → [a3] → [a3] = v3 → v1`
- `c = v4 → v3`
- `[a4] → a4 = a → [v4]`
- `v5 = v6 → v2`
- `d = c → v5`
- `[a2] → [a2] = a → v6`

- `s = d` (per establir que el `map` té el mateix tipus a la definició i a l'ús recursiu)

Definició de funció

```
\f -> \l -> if null l then [] else f (head l) : map f (tail l)
```

Solució:

- `a = [a1]`
- `a2 = a1`
- `a4 = a1`
- `a5 = a3`
- `b = [a3]`
- `c = a1 → a3`
- `v1 = [a3] → [a3]`
- `v2 = [a3]`
- `v3 = a3`
- `v4 = a1`
- `v5 = [a1] → [a3]`
- `v6 = [a1]`
- `d = (a1 → a3) → [a1] → [a3]`
- `s = (a1 → a3) → [a1] → [a3]` *(arrel)*

Definició de funció amb patrons

```
map f (x : xs) = f x : map f xs
```

En aquest cas la introducció de lambdes és una mica diferent, ja que tractem els patrons com si fossin variables lliures:

```
\f -> \(x : xs) -> f x : map f xs
```

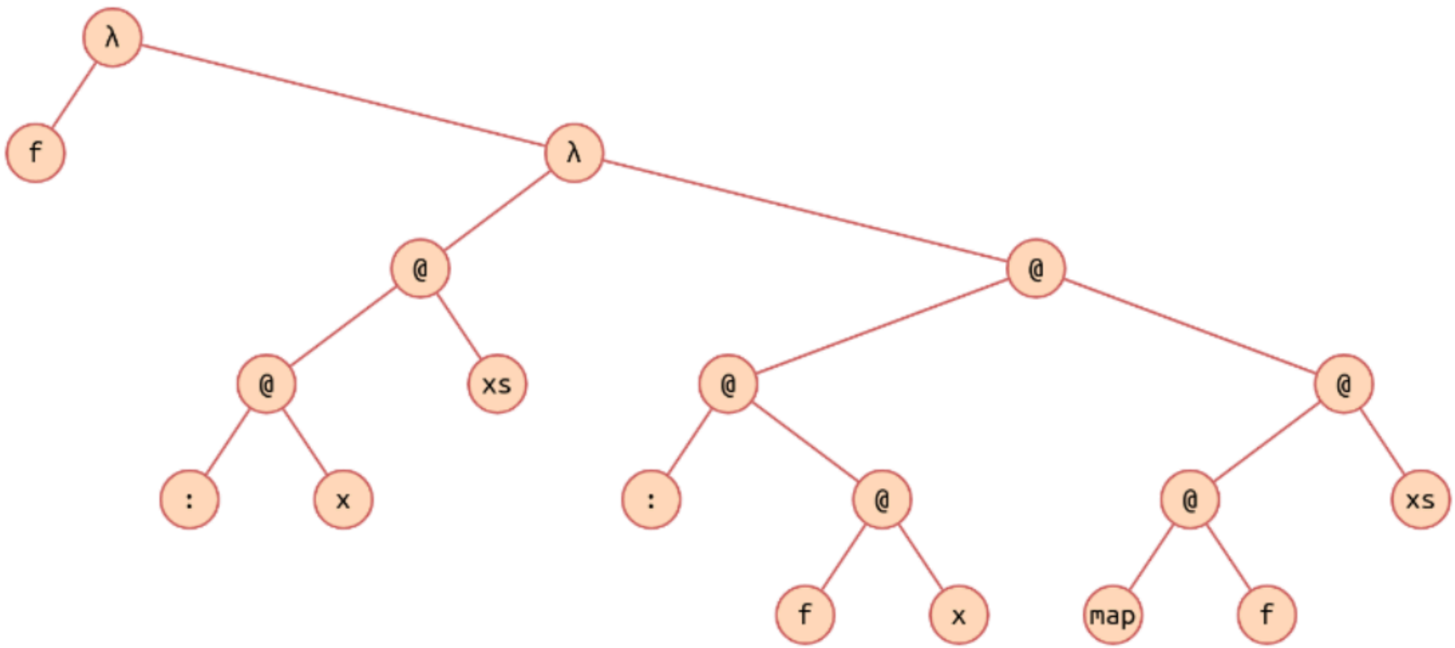
Noteu que ara hem de considerar que el primer argument de la lambda pot ser una expressió, que tractarem igual que les demés.

Totes les variables del patró queden lligades per la lambda.

Algorisme de Milner

```
\f -> \(x : xs) -> f x : map f xs
```

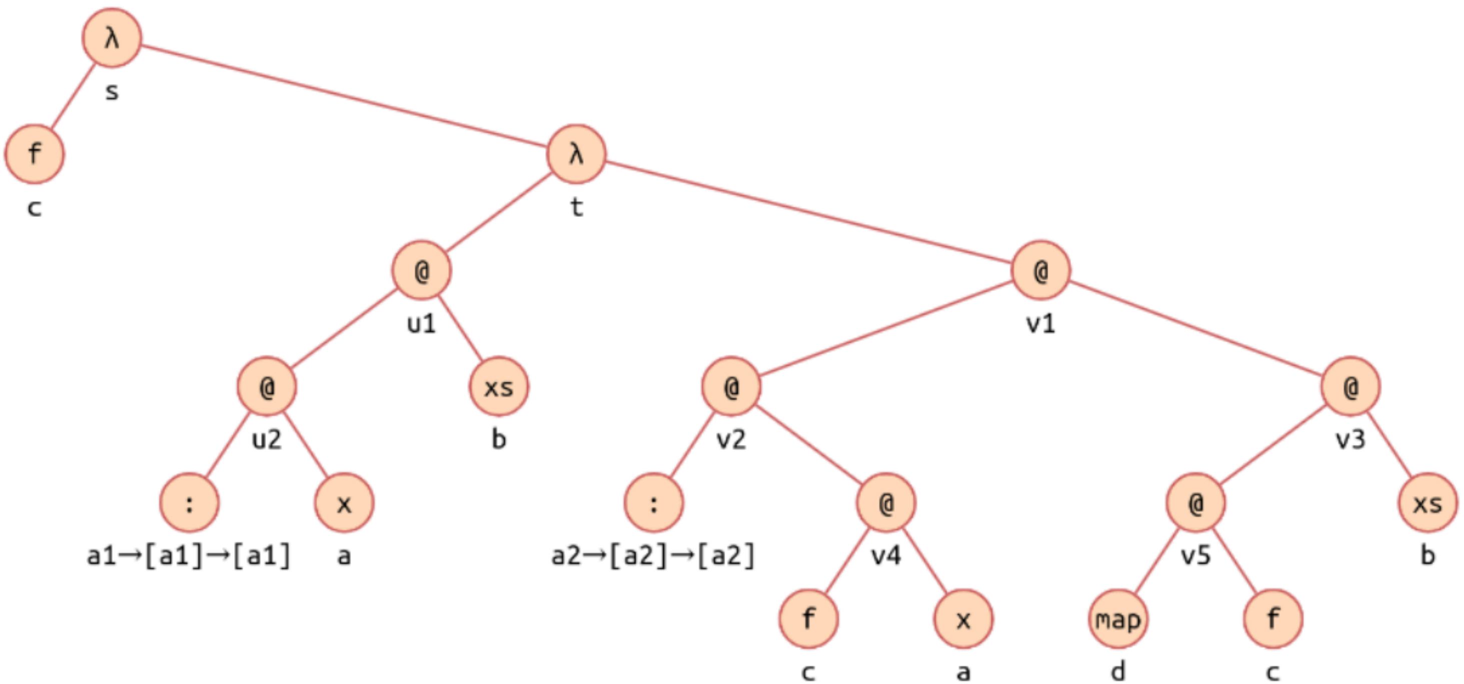
Arbre de l'expressió:



Algorisme de Milner

```
\f -> \(x : xs) -> f x : map f xs
```

Arbre etiquetat amb tipus:



Algorisme de Milner

```
\f -> \(x : xs) -> f x : map f xs
```

Equacions:

- `s = c → t`
- `t = u1 → v1`
- `u2 = b → u1`
- `a1 → [a1] → [a1] = a → u2`
- `v2 = v3 → v1`
- `a2 → [a2] → [a2] = v4 → v2`
- `c = a → v4`
- `v5 = b → v3`
- `d = c → v5`

- `s = d`

Algorisme de Milner

```
\f -> \(x : xs) -> f x : map f xs
```

Solució:

- `a1 = a`
- `b = [a]`
- `c = a → a2`
- `d = (a → a2) → [a] → [a2]`
- `s = (a → a2) → [a] → [a2]`
- `t = [a] → [a2]`
- `u1 = [a]`
- `u2 = [a] → [a]`
- `v1 = [a2]`
- `v2 = [a2] → [a2]`
- `v3 = [a2]`
- `v4 = a2`
- `v5 = [a] → [a2]`

Per tant, el tipus de l'arrel és `s = (a → a2) → [a] → [a2]`.

Funcions amb més d'una definició

```
map f [] = []
map f (x : xs) = f x : map f xs
```

Quan hi ha més d'una definició, apareix un bosc d'arbres.

Les definicions per la mateixa funció tenen el mateix tipus a l'arrel.

Analitzant una sola definició, el tipus pot ser més general que l'esperat:

```
foldr f z (x : xs) = f x (foldr f z xs)
```

`foldr :: (t1 -> t2 -> t2) -> t3 -> [t1] -> t2` ⚠️

```
foldr f z (x : xs) = f x (foldr f z xs)
foldr f z [] = z
```

`foldr :: (t1 -> t2 -> t2) -> t2 -> [t1] -> t2` 👌

Altres construccions

- Els `let` o `where` més simples es poden expressar amb abstraccions i aplicacions:

  Per exemple

  ```
  let x = y in z
  ```

  es tracta com

  ```
  (\x -> z) y
  ```

- Les guardes es tracten com un `if-then-else`.

- El `case` es tracta com una definició per patrons.

Contingut

- Introducció
- Algorisme de Milner
- Exercicis
- Funcions
- Classes
- Errors
- Exercicis

Classes

La presència de definicions com ara

```
(+) :: Num a => a -> a -> a
(>) :: Ord a => a -> a -> Bool
```

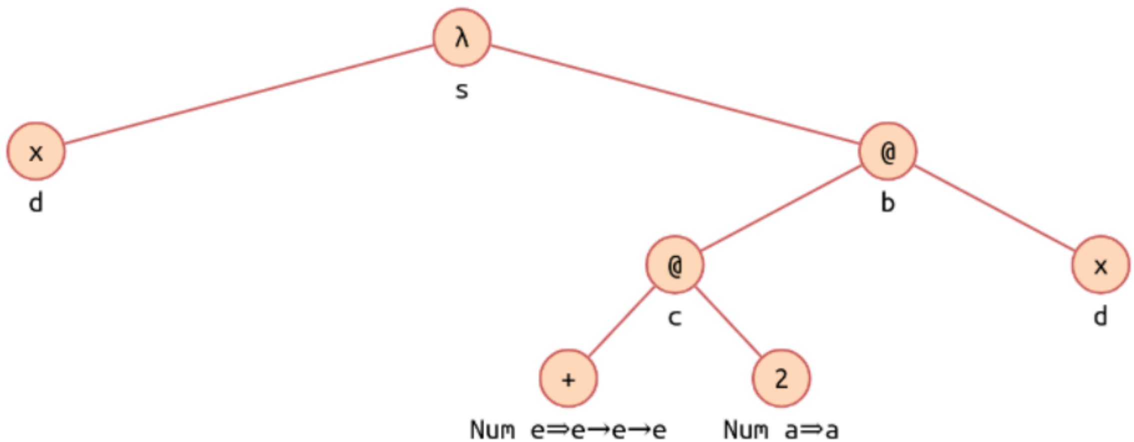
introdueix unes noves *restriccions de context*.

Per tant, les solucions també han de contenir i satisfer les condicions de classe.

Classes

```
f x = 2 + x
```

Arbre etiquetat:



Classes

Equacions:

- `s = d → b`
- `c = d → b`
- `e → e → e = a → c`

Restriccions:

- `Num a`
- `Num e`

Solució:

- `s = a → a`
- `b = a`
- `c = a → a`
- `d = a`
- `e = a`

El tipus de l'arrel (de `f`) és doncs `Num a ⇒ a → a`.

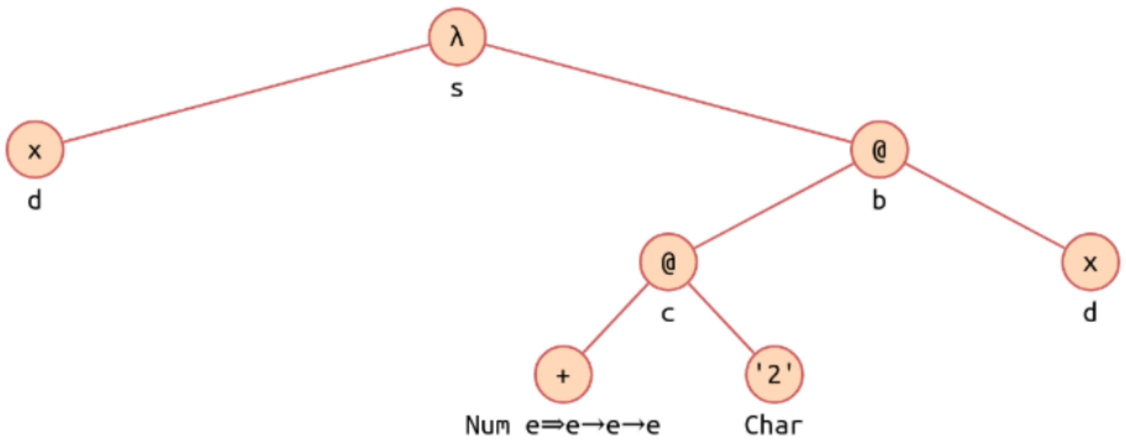
Contingut

- Introducció
- Algorisme de Milner
- Exercicis
- Funcions
- Classes
- Errors
- Exercicis

Errors

```
f x = '2' + x
```

Arbre etiquetat:



Errors

Equacions:

- `s = d → b`
- `c = d → b`
- `e → e → e = Char → c`

Restriccions:

- `Num e`

Intent de solució:

- `s = Char → Char`
- `b = Char`
- `c = Char → Char`
- `d = Char`
- `e = Char`
- `Num Char` ❌

Perquè `Char` no és instància de `Num`!

Contingut

Exercicis

- Inferiu el tipus de:

```haskell
ones = 1 : ones
```

- Inferiu el tipus de:

```haskell
even x = if rem x 2 == 0 then True else False
```

amb `rem :: Int → Int → Int`.

- Inferiu el tipus de:

```haskell
even x = rem x 2 == 0
```

- Inferiu el tipus de:

```haskell
last [x] = x
```

Recordeu que `[x]` és `x:[]`.

Exercicis

- Inferiu el tipus de:

```haskell
delete x (y:ys) =
    if x == y
    then ys
    else y : delete x ys
```

amb `(==) :: Eq a ⇒ a → a → Bool`.