Llenguatges de Programació, FIB, 17 de gener de 2022

Possibles solucions

1 Pragmàtica dels LPs

Sense els parèntesis hi podria haver ambigüitats. Per exemple: **if** x ++ y; voldria dir **if** $(x{+}{+})\ y$; o bé **if** $(x)\ {+}{+}y$; ?

2 λ-càlcul **(1 punt)**

Veremos que $AND \equiv \lambda a\ b.a\ b\ (\lambda x\ y.y)$, evaluando $(AND\ u\ v)$ considerando los posibles casos de u y v:

$$(AND\ u\ v) \equiv (\lambda a\ b.a\ b\ (\lambda x\ y.y))u\ v \rightarrow_\beta\ u\ v\ (\lambda x\ y.y)$$

Si u es *False*, es decir $u \equiv \lambda x\ y.y$:

$$u\ v\ (\lambda x\ y.y) \equiv (\lambda x\ y.y)\ v\ (\lambda x\ y.y) \rightarrow_\beta \lambda x\ y.y \equiv False$$

Es decir $AND\ False\ v = False$. Y si u es *True*, es decir $u \equiv \lambda x\ y.x$:

$$u\ v\ (\lambda x\ y.y) \equiv (\lambda x\ y.x)\ v\ (\lambda x\ y.y) \rightarrow_\beta v$$

Por tanto, si $v \equiv False$, entonces $AND\ True\ False = False$, y si $v \equiv True$, entonces $AND\ True\ True = True$.

3 Subtipus

- En principio, los punteros podrían considerarse covariantes ya que las operaciones de acceso y de modificación del contenido de la dirección apuntada por el puntero preservan la relación de subtipo. Sin embargo, la covarianza de los punteros puede dar problemas de inconsistencia de tipos. Veamos un ejemplo. Supongamos que tenemos las clases `Empleado`, `Vendedor` y `Administrativo`, con `Vendedor` $\leq$ `Empleado` y `Administrativo` $\leq$ `Empleado` y definimos la función

  ```
  void P(Empleado* p, Empleado e){
     *p = e; \\
  }
  ```

  El problema es que si los punteros son covariantes entonces, podemos escribir:

  ```
  Vendedor v;
  Administrativo* p1;
  ...
  P(p1, v);
  ```

  lo que produciría que `p1`, de tipo puntero a Administrativo apuntara a un objeto de tipo `Vendedor`.

- Los punteros no se deben de considerar contravariantes. Por ejemplo, si lo fueran, podrámos escribir;

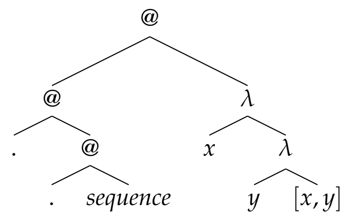
```
Vendedor v;
Administrativo a;
Empleado* p1;
Vendedor* p2;
...
p1 = new Empleado a;
p2 = p1;
v = *p2;
```

  lo que produciría que a `v` se le asignara un objeto de tipo `Administrativo`.

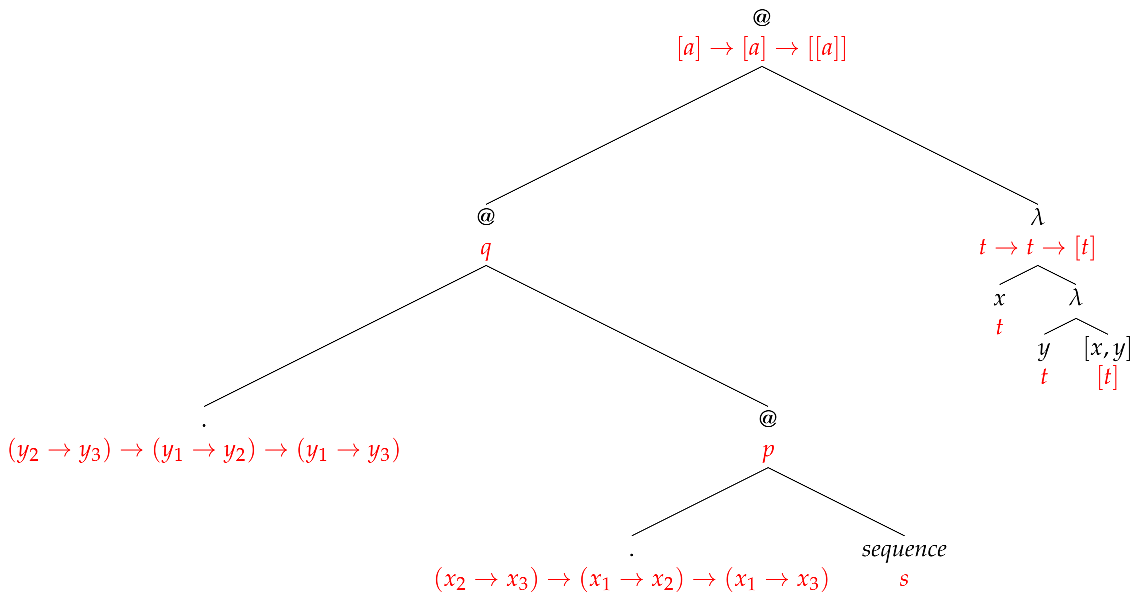
- Desde este punto de vista, lo más razonable sería considerar que los punteros deberían de ser invariantes.

4 Inferència de tipus

1. Aquest és l'arbre de sintàxi abstracta de *cartesian* :



2. Aquesta és la seva anotació de tipus. Per no generar tantes restriccions, ja s'ha inferit tribialment el tipus del fill dret de l'arrel.



3. Les restriccions que apareixen són les següents:

$$\begin{aligned}(x_2 \to x_3) \to (x_1 \to x_2) \to (x_1 \to x_3) &= s \to p \\ (y_2 \to y_3) \to (y_1 \to y_2) \to (y_1 \to y_3) &= p \to q \\ (t \to t \to [t]) \to ([a] \to [a] \to [[a]]) &= q\end{aligned}$$

4. Resolem les restriccions per trobar el tipus s:

$$\begin{aligned}
s &= x_2 \to x_3 \\
p &= (x_1 \to x_2) \to (x_1 \to x_3) \\
p &= y_2 \to y_3 \\
y_2 &= x_1 \to x_2 \\
y_3 &= x_1 \to x_3 \\
q &= (y_1 \to y_2) \to (y_1 \to y_3)) \\
y_1 \to y_2 &= t \to t \to [t] \\
y_1 \to y_3 &= [a] \to [a] \to [[a]] \\
y_1 &= t = [a] \\
y_2 &= t \to [t] \\
y_3 &= [a] \to [[a]] \\
x_1 &= t \to [a] \\
x_2 &= [t] = [[a]] \\
x_3 &= [[a]] \\
s &= [[a]] \to [[a]].
\end{aligned}$$

   Per tant, $\mathit{sequence} :: [[a]] \to [[a]]$.

   [En realitat, $\mathit{sequence} :: (\textbf{Traversable}\ t, \ \textbf{Monad}\ m) \Rightarrow t\ (m\ a) \to m\ (t\ a)$ però això no es podia inferir de l'arbre, calia més context.]

5 Python

```
def make_italic(func):     return lambda: "<i>" + func() + "</i>"
def make_bold(func):       return lambda: "<b>" + func() + "</b>"
def make_underline(func):  return lambda: "<u>" + func() + "</u>"
```

I refactoritzar mai és dolent:

```
def add_tag(func, tag):
    return "<" + tag + ">" + func() + "</" + tag + ">"

def make_italic(func):     return add_tag(func, "i")
def make_bold(func):       return add_tag(func, "b")
def make_underline(func):  return add_tag(func, "u")
```

6 Haskell

```
instance Functor Tree where
    fmap f (Leaf x)     = Leaf (f x)
    fmap f (Node e1 e2) = Node (fmap f e1) (fmap f e2)

instance Monad Tree where
    return = Leaf
    (Leaf x) >>= f = f x
    (Node e1 e2) >>= f = Node (e1 >>= f) (e2 >>= f)
```

$$
\begin{aligned}
&\mathit{replace}\ \ \mathit{dic}\ \ e \ = \ e \gg\!= f \\
&\quad \textbf{where}\ f\ x \ = \ \mathit{Leaf}\ \ (\textbf{lookup}\ x\ \mathit{dic})
\end{aligned}
$$

Primera llei dels functors:

- $\mathit{fmap}\ \textbf{id}\ \ (L\ x) \ = \ L\ (\textbf{id}\ x) \ = \ L\ x \ = \ \textbf{id}\ \ (L\ x)$ ✓
- $\mathit{fmap}\ \textbf{id}\ \ (N\ e1\ e2) \ = \ N\ (\mathit{fmap}\ \textbf{id}\ e1)\ \ (\mathit{fmap}\ \textbf{id}\ \ e2) \ = \ N\ e1\ e2 \ = \ \textbf{id}\ \ (N\ e1\ e2)$ ✓

Segona llei dels functors:

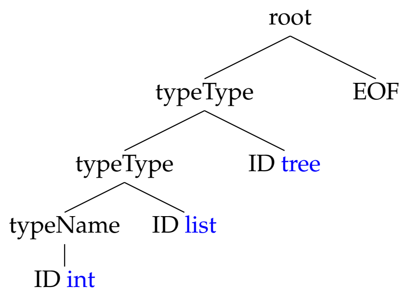
- $\mathit{fmap}\ (f\ \ .\ \ g)\ \ (L\ x) \ = \ L\ \ ((f\ \ .\ \ g)\ \ x) \ = \ L\ f\ \ (g\ \ x) =$
  $\mathit{fmap}\ f\ \ (L\ \ (g\ \ x)) \ = \ \mathit{fmap}\ f\ \ ((\mathit{fmap}\ g)\ \ (L\ x)) \ = \ ((\mathit{fmap}\ f).(\mathit{fmap}\ g))\ \ (L\ x)$ ✓
- FALTA FER!!! ✓

7 Compilació

1. `visitRoot`, `visitTypeParen`, `visitTypeVar`, `visitTypePair`, `visitTypeApp`, `visitTypeRecord`, `visitTypeName`, `visitTypeType`, `visitTypeArrow` i `visitRecord`.

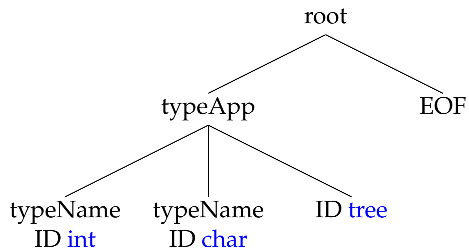
2. (a) no ho és perquè el subtratllat no és a cap token.

   (b)

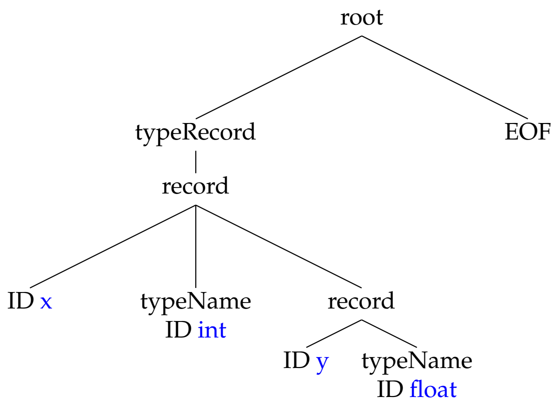


   (c) no ho és perquè el símbol de suma no és a cap token.

   (d)



   (e)



(parèntesis, dos punts i comes no es mostren per concisió)

(f) no ho és perquè la gramàtica impedeix que dos `TVAR`s vagin seguits.

3. Els nodes dels arbres es visiten en preordre:

   (b) `visitRoot`, `visitTypeType`, `visitTypeType`, `visitTypeName`.

   (d) `visitRoot`, `visitTypeApp`, `visitTypeName`, `visitTypeName`.

   (e) `visitRoot`, `visitTypeRecord`, `visitRecord`, `visitTypeName`, `visitRecord`, `visitTypeName`.

4. Aquesta gramàtica no es pot parsejar amb un parser LL(1) perquè la regla `type` té recursivitat per l'esquerra i prefixos comuns.