Llenguatges de Programació, FIB, 14 de gener de 2020

Es valorarà la concisió, claredat i brevetat a més de la completesa i l'exactitud de les respostes. No es pot consultar cap material addicional. Feu bona lletra.

Cada problema compta dos punts.

Solucioneu cada problema en un full diferent.

Temps: tres hores.

1 Inferència de tipus

Un programador novell en Haskell ha escrit aquesta funció *reves* per retornar una llista del revés:

$$\begin{aligned} &\mathit{reves}\ [] &&= [] \\ &\mathit{reves}\ (x{:}xs) &&= \mathit{reves}\ \ xs \end{aligned}$$

a) Inferiu el tipus més general de la funció *reves*. Per a fer-ho, dibuixeu l'arbre de sintàxi de les expressions, etiqueteu els nodes amb els seus tipus i genereu les restriccions de tipus. Resoleu-les per obtenir la solució i assenyaleu el resultat final amb un requadre.

b) A la vista del tipus obtingut, quina conclusió hauria de treure'n el prograd or novell?

c) Si el programador hagués especificat que *reves* $::\ [a] \to [a]$, la comprovació de tipus hauria detectat algun error?

2 Haskell

Recordeu que la classe **Functor** ve definida per

$$\begin{aligned} &\textbf{class}\ \ \textbf{Functor}\ f\ \ \textbf{where} \\ &\quad \mathit{fmap}\ ::\ \ (a\ \to b) \to (f\,a\ \to f\,b) \end{aligned}$$

i que les lleis dels functors són:

1. $\mathit{fmap}\ \textbf{id}\ \ \equiv\ \ \textbf{id}$
2. $\mathit{fmap}\ (g1\ .\ g2)\ \ \equiv\ \ \mathit{fmap}\ g1\ .\ \mathit{fmap}\ g2$

Recordeu també que la classe **Monad** ve definida per

$$\begin{aligned} &\textbf{class}\ \ \textbf{Monad}\ m\ \ \textbf{where} \\ &\quad \textbf{return}\ ::\ \ a\ \to m\,a \\ &\quad (\gg\!=) :: m\,a \to (a \to m\,b) \to m\,b \end{aligned}$$

i que les lleis de les mònades són:

1. $\textbf{return}\ x\ \gg\!= f\ \ \equiv\ \ f\ \ x$
2. $m \gg\!= \textbf{return}\ \ \equiv\ \ m$
3. $(m \gg\!= f) \gg\!= g\ \ \equiv\ \ m \gg\!= (\backslash x \to f\,x \gg\!= g)$

(Ignoreu l'operador $\gg$ i que les mònades també ha de ser aplicatius i functors.)

a) Feu que el tipus polimòrfic **Maybe** sigui instància no tribial de la classe **Functor** i demostreu que la vostra instanciació compleix les lleis dels functors.

b) Feu que el tipus polimòrfic **Maybe** sigui instància no tribial de la classe **Monad** i demostreu que la vostra instanciació compleix les lleis de les mònades.

3 Python

a) Què escriu aquest programa en Python?

```
fila = [0, 0, 0, 0]
matriu = [ fila , fila , fila , fila ]
print(matriu [0][0], matriu [0][1], matriu [1][0])
matriu [0][0] = 2
print(matriu [0][0], matriu [0][1], matriu [1][0])
```

b) Què escriu aquest altre programa en Python?

```
def foo(k): k = ["banana"]

def bar(c): c.append("rainbow")

q = ["unicorn"]
foo(q)
bar(q)
print(q)
```

c) Escriviu un generador $nombres(i)$ que generi la seqüència infinita i, $i+1$, $i+2$, ...

d) Escriviu un generador $filtre\ (s, p)$ que, donats una seqüència s i un predicat p, generi la seqüència dels elements de s que compleixen el predicat p.

e) Resumiu, en una línia, les característiques més importants del sistema de tipus de Python.

4 Herència en Java i C++

Considereu aquest esquelet de programa en Java:

```
class C {
    int x;
    void m(int y) { ··· }
}

class C1 extends C {
    int x1;
}

class C2 extends C1 {
    int x2;
}

class C3 extends C {
    void m(int y) { ··· }
}
```

Suposeu que totes les classes, mètodes i atributs són visibles.

a) Suposant que s'han declarat les variables *a*, *a1*, *a2* i *a3* com variables de tipus *C*, *C1*, *C2* i *C3* respectivament, digueu si alguna de les assignacions següents seria incorrecta:

```
a1 = a2;
a2 = a;
a3 = a;
a = a3;
```

b) Suposem ara que *després* de l'anterior seqüència d'assignacions (una vegada eliminades les assignacions incorrectes) tenim les instruccions:

```
a1.x1 = 3;
a1.x2 = 5;
a2.x2 = 4;
a2.x = 7;
a2.m(7);
a3.m(7);
a.m(7);
```

Alguna d'aquestes instruccions seria incorrecta?

c) Si les dues darreres instruccions són correctes, quin *m* es cridaria en cada cas? El de la classe *C* o el de *C3*?

[Continua a la pàgina següent.]

Considereu ara aquest esquelet de programa en C++:

```
class C {
    int x;
    void m(int y);
}

class C1: C {
    int x1;
}

class C2: C1 {
    int x2;
}

class C3: C {
    void m(int y);
}
```

Suposeu que totes les classes, mètodes i atributs són visibles.

d) Suposant que s'han declarat les variables *a*, *a1*, *a2* i *a3* com variables de tipus *C*, *C1*, *C2* i *C3* respectivament, digueu si alguna de les assignacions següents seria incorrecta:

```
a1 = a2;
a2 = a;
a3 = a;
a = a3;
```

e) Suposem ara que *després* de l'anterior seqüència d'assignacions (una vegada eliminades les assignacions incorrectes) tenim les instruccions:

```
a1.x1 = 3;
a1.x2 = 5;
a2.x2 = 4;
a2.x = 7;
a2.m(7);
a3.m(7);
a.m(7);
```

Alguna d'aquestes instruccions seria incorrecta?

f) Si les dues darreres instruccions són correctes, quin *m* es cridaria en cada cas? El de la classe *C* o el de *C3*?

5 Compilació

Dóneu (en format .g) l'especificació lèxica i sintàctica d'un compilador per a un llenguatge de gestió de fitxers i directoris. El llenguatge ha de permetre crear fitxers i directoris, copiar fitxers de diverses formes, moure fitxers i directoris, i preguntar per profunditat i amplada respecte el nombre de línies.

Per exemple, aquest és un exemple d'entrada:

```
F1 := mkfile("fitxer")
        // F1 és el fitxer amb nom "fitxer".
        // s'assumeix que l'usuari li posaria contingut.

F2 := F1 ; F1
        // F2 és un nou fitxer que conté dues còpies concatenades
        // -una darrera l'altra- del fitxer F1

dir := mkdir("directori")
        // dir és el directori amb nom "directori".

move(move(dir,F1),F2)
        // moure el fitxer F1 (primer) i el F2 (després) al director dir

F3 :=  F2 ^ 3
        // F3 es un nou fitxer que és una concatenació del fitxer F2 tres vegades

F4 := F1 <-> F2
        // F4 és un nou fitxer que surt de fusionar linies
        // (primera amb primera, segona amb segona, etc, ...) de F1 i F2

F5 := (F1;F1) <-> (F2;F1)^3
        // exemple d'expressió complexa

length(F1)
        // mostra el nombre de línies de F1

width(F1)
        // mostra la màxima amplada -linia amb nombre màxim de paraules- de F1

maxlength(dir)
        // mostra la màxima length respecte tots els fitxers que pengen de dir

maxwidth(dir)
        // mostra la màxima amplada respecte tots els fitxers que pengen de dir
```

Llenguatges de Programació, FIB, 14 de gener de 2020

Es valorarà l'eficiència, la concisió, la claredat i la brevetat a més de la completesa i l'exactitud de les respostes. No es pot consultar cap material addicional. Feu bona lletra.

Aquesta part és pels estudiants que han demanat avaluació única.

Temps: una hora.

SimpleLP

SimpleLP és un llenguatge de programació imperativa molt simple. Per exemple, aquest és un programa en SimpleLP que correspon a l'algorisme d'Euclides per calcular el màxim comú divisor de 105 i 252 (que és 21):

```
a := 105
b := 252
while a ≠ b do
    if a < b then b := b − a
    else a := a − b
end
```

En SimpleLP, l'únic tipus de dades existent són els enters, amb operacions aritmètiques de suma i resta. També hi ha operacions relacionals per a diferent-de i menor-que; aquestes retornen 0 per a fals i 1 per a cert. El llenguatge admet variables, amb un valor de zero per defecte. Les instruccions són l'assignació, la composició seqüèncial de diverses instruccions, el condicional if-then-else i la iteració while.

L'arbre de sintàxi abstracta de SimpleLP està descrit amb aquestes estructures en Haskell:

```
data Expr
    = Val Int                  -- valor
    | Var String               -- variable
    | Add Expr Expr            -- suma (+)
    | Sub Expr Expr            -- resta (–)
    | Neq Expr Expr            -- diferent-de (≠)
    | Lth Expr Expr            -- menor-que (<)

data Instr
    = Ass String Expr          -- assignació
    | Seq [Instr]              -- composició seqüèncial
    | Cond Expr Instr Instr    -- condicional
    | Loop Expr Instr          -- iteratiu
```

Per exemple, aquest és l'arbre de sintàxi abstracta que correspon al programa anterior:

```
euclides = Seq [
    (Ass "a" (Val 105)),
    (Ass "b" (Val 252)),
    (While
        (Neq (Var "a") (Var "b"))
        (Cond
            (Lth (Var "a") (Var "b"))
            (Ass "b" (Sub (Var "b") (Var "a")))
            (Ass "a" (Sub (Var "a") (Var "b")))
        )
    )
]
```

Una taula de símbols descriu l'estat de la memòria associant els identificadors de les variables amb els seus valors. La taula de símbols és de tipus *Mem* i ofereix aquestes operacions (ja implementades amb arbres AVL):

```
-- retorna una taula de símbols buida
empty :: Mem

-- insereix (o canvia si ja hi era) un identificador amb el seu valor
update :: Mem → String → Int → Mem

-- consulta el valor d'una clau en una taula de símbols
search :: Mem → String → Maybe Int

-- retorna (en ordre lexicogràfic) la llista de claus en una taula de símbols
keys :: Mem → [String]
```

La vostra feina:

a) Escriviu una funció *eval* :: *Expr* → *Mem* → **Int** que avaluï una expressió en un estat de la memòria.

b) Escriviu una funció *exec* :: *Instr* → *Mem* → *Mem* que retorni l'estat final de la memòria després d'executar una instrucció partint d'un estat inicial de memòria donat. Per exemple, *exec euclides empty* hauria de retornar una taula de símbols on "a" i "b" valen 21 ambdós.

c) Digueu en quin cas es podria donar que la vostra funció *exec* no acabés.

d) Escriviu una acció *run* :: *Instr* → **IO** () que escrigui (en ordre lexicogràfic) el valor final de cada variable després d'executar una instrucció partint d'una memòria buida. Per exemple, *run euclides* hauria d'escriure

```
a -> 21
b -> 21
```