Università degli Studi di Udine

DIPARTIMENTO DI SCIENZE MATEMATICHE, INFORMATICHE E FISICHE
Corso di Laurea Magistrale in Informatica

Relazione del progetto
per l'insegnamento
Linguaggi e compilatori

ProgettoLC parte2b

Studente:
Filippo Callegari
callegari.filippo@spes.uniud.it
Matricola 128602

Professore dell'insegnamento:
Marco Comini

Anno Accademico 2019–2020 - Secondo Semestre

Assunzioni In quanto non presente una vera e propria regola da utilizzare all'interno del "prelude", si è ricorsi alla funzione `if'` definita sulla wiki di haskell, presso il seguente indirizzo Wiki Haskell: If-Then-Else

Esercizio 1

Le regole sono quindi descritte come segue:

```
        fact :: Integer -> Integer
R1:     fact n = if n == 0 then 1 else n * fact (n - 1)

        take :: Int -> [ a ] -> [ a ]
R2:     take _ [] = []
R3:     take n _ | n <= 0 = []
R4:     take n ( x : xs ) = x : take (n -1) xs

        map :: (a -> b) -> [a] -> [b]
R5:     map f ( x : xs ) = f x : map f xs
R6:     map _ [] = []

        if :: Bool -> a -> a -> a
R7:     if True  x _ = x
R8:     if False _ y = y
```

Si chiede di rappresentare l'esecuzione della seguente query:

Query sottoposta

```
let
    v = fact k;
    k = j - 2;
    j = 2 * 2 in
  take (5 - k)(v : map fact [j - k, v, v, error "q",v])
```

Esecuzione in notazione lineare

Per rendere univoca l'applicazione delle occorrenze dei metodi di classe, riscriviamo quindi la query come segue:

```
take
 (-#Int 5#Int κ)
 (
   ν :
   map fact [-#Int ι κ, ν, nu, error "q"#[Char], ν]
 )
```

Dove abbiamo:

- ν =v, con `v = fact` κ;
- κ = k, con `k = -#Int` ι `2#Int`;
- ι = j, con `j = *#Int 2#Int 2#Int`.

Possiamo passare alla sua esecuzione.

[Valuto R2, ma non matcha.]
[Provo ad applicare R3, devo prima valutare il valore da passare alla guardia]

`(-#Int 5#Int` κ`)`

[Dobbiamo valutare prima κ]

`k = -\#Int` ι `2\#Int`

[Dobbiamo prima valutare ι]
`j = *#Int 2#Int 2#Int` → Viene valutato `4#Int`.

[Torno a κ]
`k = -#Int 4#Int 2#Int` → Viene valutato `2#Int`.

[Torno a valutare il valore da passare alla guardia di R3]
`-#Int 5#Int 2#Int` → Viene valutato `3#Int`.

[Passo a valutare la guardia di R3]
`<=#Int 2#Int 0#Int` → Viene valutato `False`.

[Procedo con la regola R4, matcha e produce]

```
ν : take (-#Int 3#Int 1#Int)
             (map fact [-#Int 4#Int 2#Int, ν, ν, error "q"#[Char], ν])
```

[Devo ora valutare ν]

`v = fact` κ

[I passi per valutare κ sono sopra]
[Applico R1 per calcolare ν]

```
v = if (==#Int 2#Int 0#Int) then
        1#Int
        else (*#Int 2#Int (fact (-#Int 2#Int 1#Int)))
```

[Provo ad applicare R7, che mi obbliga a valutare la guardia dell'`if`]
`==#Int 2#Int 0#Int` → Viene valutato `False`.

[R7 fallisce, applico R8 per la valutazione di ν]

```
v = (*#Int 2#Int (fact (-#Int 2#Int 1#Int)))
```

[Devo valutare l'operazione per la chiamata di `fact`]
`-#Int 2#Int 1#Int` → Viene valutato `1#Int`.

[Devo valutare la seconda chiamata a `fact` applicando R1]

```
if (==#Int 1#Int 0#Int) then
    1#Int
    else (*#Int 1#Int (fact (-#Int 1#Int 1#Int)))
```

[Provo ad applicare R7, che mi obbliga a valutare la guardia dell'if]
`==#Int 1#Int 0#Int` → Viene valutato `False`.

[R7 fallisce, applico R8 per la valutazione di ν]

```
v = (*#Int 2#Int (*#Int 1#Int (fact (-#Int 1#Int 1#Int))))
```

[Devo valutare il valore da passare a `fact`]
`-#Int 1#Int 1#Int` → Viene valutato `0#Int`.

[Devo valutare la terza chiamata a `fact` applicando R1]

```
if (==#Int 0#Int 0#Int) then
    1#Int
    else (*#Int 0#Int (fact (-#Int 0#Int 1#Int)))
```

[Provo ad applicare R7, che mi obbliga a valutare la guardia dell'if]
`==#Int 0#Int 0#Int` → Viene valutato `True`.

[Applico R7 ed ottengo]

```
v = (*#Int 2#Int (*#Int 1#Int (1#Int)))
```

[Devo valutare il primo fattore dell'operazione esterna `*#Int`]
`*#Int 1#Int 1#Int` → Viene valutato `1#Int`.

[Devo valutare l'espressione da assegnare a ν]
`v = (*#Int 2#Int 1#Int)` → Viene valutato `2#Int`.

[Al termine della valutazione di ν avrò]

```
2#Int :
  take (-#Int 3#Int 1#Int)
    (map fact [-#Int 4#Int 2#Int, 2#Int, 2#Int, error "q"#[Char], 2#Int])
```

[Applico R2 e non matcha, applico R3 e devo valutare l'arg. di guardia]
`-#Int 3#Int 1#Int` → Viene valutato `2#Int`.
[Valuto quindi la guardia per R3]
`<=#Int 2#Int 0#Int` → Viene valutato `False`.
[Applico quindi R4, devo valutare quindi `map` con regola R5]

```
map fact [-#Int 4#Int 2#Int, 2#Int, 2#Int, error "q"#[Char], 2#Int]
```

[In uscita da R5 avrò]

```
(fact -#Int 4#Int 2#Int) : map fact [2#Int, 2#Int, error "q"#[Char], 2#Int]
```

[Posso quindi valutare R4]

```
2#Int :
  take (-#Int 2#Int 1#Int)
    fact (-#Int 4#Int 2#Int) :
      map fact [2#Int, 2#Int, error "q"#[Char], 2#Int]
```

[che ottengo in uscita da R4]

```
2#Int : fact (-#Int 4#Int 2#Int) :
    take (-#Int 2#Int 1#Int)
        map fact [2#Int, 2#Int, error "q"#[Char], 2#Int]
```

[Valuto il valore da passare a fact]

`k = -#Int 4#Int 2#Int` → Viene valutato `2#Int`.

[Si rimanda sopra per vedere la valutazione di `fact 2#Int`]
[All'uscita di `fact 2` avrò]

`fact 2#Int` → Viene valutato `2#Int`.

[che diventa]

```
 2#Int :2#Int :
    take (-#Int 2#Int 1#Int)
        map fact [2#Int, 2#Int, error "q"#[Char], 2#Int]
```

[A questo punto, come prima si passa a rivalutare take.]

[Applico R2 e non matcha, applico R3 e devo valutare l'arg. di guardia]

`-#Int 2#Int 1#Int` → Viene valutato `1#Int`.

[Valuto quindi la guardia per R3]

`<=#Int 1#Int 0#Int` → Viene valutato `False`.

[Applico quindi R4, devo valutare quindi `map` con regola R5]

```
map fact [2#Int, 2#Int, error "q"#[Char], 2#Int]
```

[In uscita da R5 avrò]

```
2#Int : 2#Int :
  take (-#Int 1#Int 1#Int)
    fact 2#Int :
      map fact [2#Int, error "q"#[Char], 2#Int]
```

[Ed ottengo in uscita da R4]

```
2#Int : 2#Int : fact 2#Int :
    take (-#Int 1#Int 1#Int)
        map fact [2#Int, error "q"#[Char], 2#Int]
```

[Si rimanda sopra per vedere la valutazione di `fact 2#Int`]
[All'uscita di `fact 2` avrò]

`fact 2#Int` → Viene valutato `2#Int`.

[Otterrò quindi]

```
2#Int : 2#Int : 2#Int :
    take (-#Int 1#Int 1#Int)
        map fact [2#Int, error "q"#[Char], 2#Int]
```

[Applico R2 e non matcha, applico R3 e devo valutare l'arg. di guardia]

`-#Int 1#Int 1#Int` → Viene valutato `0#Int`.

[Valuto quindi la guardia per R3]

`<=#Int 0#Int 0#Int` → Viene valutato `True`.

[Applico R3 dopo valutazione della guardia positiva]

```
2#Int : 2#Int : 2#Int : []
```

Risultato Il risultato della valutazione della query quindi è:

```
2#Int : 2#Int : 2#Int : []
```

Esercizio 2

Esercizio 1 parte 2b. Di seguito la tabella riassuntiva di calcolo del testimone più generale dell'esercizio 1 parte 2b.

mgu(Ri, Rj)	Testimone più generale
R2, R3	`take n []`
R2, R4	nessun overlap
R3, R4	`take n (x : xs)`
R5, R6	nessun overlap
R7, R8	nessun overlap

La regola `R1` non viene presa in considerazione in quanto non può avere overlap.

Esercizio 1 parte 2a. Di seguito presentiamo le regole dell'esercizio preso in considerazione:

```
fmap :: (a -> b) -> [a] -> [b]
R1:    fmap f (C x) = C $ f x
R2:    fmap f (Q ss sd is id) = Q (fmap f ss) (fmap f sd)
                                  (fmap f is) (fmap f id)

bound :: a -> a -> a -> a
R3:    bound cm cM a
         | a > cM = cM
         | a < cm = cm
         | otherwise = a

boundPicture :: Integral t => t -> t -> QT t -> QT t
R4:    boundPicture cm cM a = fmap (bound cm cM) a
```

Di seguito la tabella riassuntiva di calcolo del testimone più generale per questo esercizio.

mgu(Ri, Rj)	Testimone più generale
R1, R2	nessun overlap

Le regole `R3` ed `R4` non vengono prese in considerazione in quanto non possono avere overlap.