Тема 14

Изследване на свойствата на програми на Haskell

Едно от най-големите предимства на програмирането във функционален стил е възможността строго да се доказват свойства на функционалните програми.

Тук ще покажем един прост подход за доказване на това, че определено множество от функции е дефинирано така, че дадено свойство, във формулировката на което участват тези функции, е вярно за всички (крайни) списъци.

Принцип на структурната индукция при работата със списъци

В случай, че трябва да се докаже, че дадено свойство P(xs) е вярно за всички крайни списъци xs, доказателството може да се извърши на две стъпки:

- ***Базов случай.*** Доказване, че е вярно P([ ]).
- ***Индуктивна стъпка.*** Доказване, че е вярно P(x:xs) при предположение, че е вярно P(xs).

Твърдението P(xs) във втората стъпка се нарича ***индуктивна хипотеза,*** тъй като за него се предполага, че е вярно в процеса на доказателството на P(x:xs).

Пример 1. Нека sum и doubleAll са функции, дефинирани както следва:

```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs

doubleAll :: [Int] -> [Int]
doubleAll []     = []
doubleAll (z:zs) = 2*z : doubleAll zs
```

Ще покажем, че за всеки списък xs е в сила
sum (doubleAll xs) = 2*sum xs.

Доказателството ще извършим на две стъпки:

1. Базов случай: трябва да покажем, че
   sum (doubleAll [ ]) = 2*sum [ ].
2. Индуктивна стъпка: ще покажем, че е в сила
   sum (doubleAll (x:xs)) = 2*sum (x:xs),
   използвайки за целта индуктивната хипотеза
   sum (doubleAll xs) = 2*sum xs.

Базов случай

sum (doubleAll [ ]) = sum [ ] = 0;
2 * sum [ ] = 2*0 = 0.

Следователно, sum (doubleAll [ ]) = 2 * sum [ ].

Индуктивна стъпка

sum (doubleAll (x:xs)) = sum (2*x : doubleAll xs)
= 2*x + sum (doubleAll xs);
2 * sum (x:xs) = 2 * (x + sum xs) = 2*x + 2*sum xs.

Според индуктивното предположение е в сила
sum (doubleAll xs) = 2*sum xs, следователно

sum (doubleAll (x:xs)) = 2*sum (x:xs).

Пример 2. Връзка между length и ++.

Нека length и ++ са функциите за намиране на дължината на списък и конкатенация на списъци, дефинирани както следва:

```
length :: [a] -> Int
length []     = 0
length (z:zs) = 1 + length zs

(++) :: [a] -> [a] -> [a]
[] ++ ys     = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

Ще покажем, че за всеки два списъка xs и ys е в сила length (xs ++ ys) = length xs + length ys.

Доказателство

Базов случай

Ще покажем, че length ([ ] ++ ys) = length [ ] + length ys.

length ([ ] ++ ys) = length ys;
length [ ] + length ys = 0 + length ys = length ys.

Следователно length ([ ] ++ ys) = length [ ] + length ys.

Индуктивна стъпка

Ще покажем, че е вярно
length ((x:xs) ++ ys) = length (x:xs) + length ys
при условие, че съгласно индуктивното предположение е вярно
length (xs ++ ys) = length xs + length ys.

length ((x:xs) ++ ys) = length (x:(xs ++ ys))
= 1 + length (xs ++ ys) = 1 + length xs + length ys;

length (x:xs) + length ys = 1 + length xs + length ys.

Следователно
length ((x:xs) ++ ys) = length (x:xs) + length ys.

Пример 3. Връзка между reverse и ++.

Нека reverse е функцията, която обръща реда на елементите на даден списък, дефинирана по следния начин:

```
reverse :: [a] -> [a]
reverse []     = []
reverse (z:zs) = reverse zs ++ [z]
```

Ще покажем, че
reverse (xs ++ ys) = reverse ys ++ reverse xs.

Доказателство

Базов случай

reverse ([ ] ++ ys) = reverse ys;

reverse ys ++ reverse [ ] = reverse ys ++ [ ] = reverse ys.

Следователно

reverse ([ ] ++ ys) = reverse ys ++ reverse [ ].

Забележка. Горното доказателство ще е коректно, ако докажем, че добавянето чрез конкатенация (++) на празен списък към даден друг списък е операция – идентитет, т.е.

xs ++ [ ] = xs за всеки списък xs.

Доказателство на това твърдение ще покажем по-нататък в тази лекция.

Индуктивна стъпка

reverse ((x:xs) ++ ys) = reverse (x:(xs ++ ys))
= reverse (xs ++ ys) ++ [x].

Според индуктивното предположение е вярно
reverse (xs ++ ys) = reverse ys ++ reverse xs.

Следователно
reverse ((x:xs) ++ ys) = (reverse ys ++ reverse xs) ++ [x].

От друга страна
reverse ys ++ reverse (x:xs) = reverse ys ++ (reverse xs ++ [x]).

Тъй като операцията конкатенация на списъци (++) е асоциативна, т.е. за всеки три списъка xs, ys и zs е изпълнено
xs ++ (ys ++ zs) = (xs ++ ys) ++ zs, то следователно
reverse ((x:xs) ++ ys) = reverse ys ++ reverse (x:xs).

Забележки

1. Асоциативността на операцията ‘++’ подлежи на строго доказателство (такова е направено в края на тази лекция).
2. Принципът на структурната индукция е подходящ за доказване на свойства на функции, дефинирани с помощта на примитивна рекурсия.

Доказателство на твърдението, че добавянето чрез конкатенация (++) на празен списък към даден друг списък е операция – идентитет, т.е. xs ++ [ ] = xs за всеки списък xs

Базов случай

По дефиниция [ ] ++ [ ] = [ ].

Индуктивна стъпка

По дефиниция (x:xs) ++ [ ] = x:(xs ++ [ ]). Според индуктивното предположение xs ++ [ ] = xs, откъдето следва, че
(x:xs) ++ [ ] = x:(xs ++ [ ]) = x:xs.

Доказателство на твърдението, че операцията конкатенация на списъци (++) е асоциативна, т.е. за всеки три списъка xs, ys и zs е изпълнено xs ++ (ys ++ zs) = (xs ++ ys) ++ zs

Базов случай

[ ] ++ (ys ++ zs) = ys ++ zs;
([ ] ++ ys) ++ zs = ys ++ zs.

Следователно
[ ] ++ (ys ++ zs) = ([ ] ++ ys) ++ zs.

Индуктивна стъпка

По дефиниция (x:xs) ++ (ys ++ zs) = x:(xs ++ (ys ++ zs)).

Според индуктивното предположение е вярно
xs ++ (ys ++ zs) = (xs ++ ys) ++ zs.

Следователно
(x:xs) ++ (ys ++ zs) = x:((xs ++ ys) ++ zs) = (x:(xs ++ ys)) ++ zs = ((x:xs) ++ ys) ++ zs.