Rekurzija in rekurzivni tipi

Rekurzija je eden od osnovnih konceptov v računalništvu, ki se pojavlja na različnih nivojih.

Splošna oblika rekurzije

Obravnavajmo rekurzivno funkcijo `f`, ki računa faktorielo. V Javi bi jo zapisali takole:

```
public static int f(int n) {
    if (n == 0) {
        return 1 ;
    } else {
        return n * f(n - 1)
    }
}
```

Ekvivalentna definicija v Pythonu:

```
def f(n):
  if n == 0:
      return 1
  else:
      return n * f(n -1)
```

Ekvivalentna definicija v OCamlu:

```
let rec f n =
    if n = 0 then 1 else n * f (n - 1)
```

Definicijo razstavimo na dva dela: na *telo* rekurzije, ki samo po sebi ni rekurzivno, in na *rekurzivni sklic* funkcije `f` same nase:

```
let telo g n =
    if n = 0 then 1 else n * g (n - 1)

let rec f n = telo f n
```

Samo drugi del je rekurziven.

Še malo drugače:

```
let telo g = fun n -> if n = 0 then 1 else n * g (n - 1)

let rec f n = telo f n
```

Vsako rekurzivno funkcijo lahko razstavimo na ta način in drugi del je vedno enak. Definirajmo si funkcijo `rek` (v angleščini običajno `rec` ali `fix`), ki sprejme telo rekurzivne definicije `t` in vrne pripadajočo rekurzivno funkcijo:

```
let rec rek t = fun x -> t (rek t) x
```

Vsa rekurzija je shranjena v `rek`, od tu naprej je ne potrebujemo več:

```
let f = rek (fun self -> (fun n -> if n = 0 then 1 else n * self (n - 1)))
```

Poglejmo si tip funkcije `rek`. OCaml je izpeljal njen tip:

```
(('a -> 'b) -> 'a -> 'b) -> 'a -> 'b
```

Tipa `'a` in `'b` sta *parametra*, ki označujeta poljubna tipa. Zapišimo z α in β:

```
((α → β) → (α → β)) → (α → β)
```

Preberemo: "`rek` je funkcija, ki sprejme funkcijo `t` tipa `(α → β) → (α → β)` in vrne funkcijo tipa `α → β`."

Poglejmo si postopek še enkrat, tokrat zapisan z λ-računom:

1. Prvotna definicija `f` se glasi: `f n = if n = 0 then 1 else n * f (n - 1)`
2. Zapišemo s pomočjo λ-abstrackije: `f = λ n . if n = 0 then 1 else n * f (n - 1)`
3. Ločimo rekurzijo in telo funkcije: `f = t f` kjer je `t = (λ g n . if n = 0 then 1 else n * g (n - 1))`
4. S pomočjo `rek` definiramo `f`: `f = rek t`

Kadar imamo preslikavo `h` in točko `x`, ki zadošča enačbi `x = h(x)`, pravimo, da je `x` **negibna točka** preslikave `h`. V numeričnih metodah je eden od osnovih postopkov reševanja enačb ta, da enačbo zapišemo v obliki `x = h (x)` in nato iščemo njeno rešitev kot zaporedje približkov

$x_0,\ \ h(x_0),\ \ h(h(x_0)),\ \ h(h(h(x_0))),\ \ldots$

Negibne točke so pomembne tudi na drugih področjih matematike in o njih matematiki veliko vedo.

Opomba: če imam enačbo `l(x) = d(x)`, jo lahko prepišemo v obliki `x = d(x) - l(x) + x` in definiramo `h(x) = d(x) - l(x) + x`, da dobimo `x = h(x)`.

Ugotovili smo, da je tudi rekurzivna definicija funkcije `f` pravzaprav enačba, ki ima oblike negibne točke:

```
f = t f
```

Pomnimo:

> Rekurzivno definirana funkcija je negibna točka.

Rekurzivna funkcija več argumentov

Ali to deluje tudi za rekurzivne funkcije dveh argumentov? Seveda!

1. `s (n, k) = (if k = 0 then n else s (n + k, k - 1))`
2. `s = λ (n, k) . if k = 0 then n else s (n + k, k - 1)`
3. `s = t s`, kjer je `t = λ g . λ (n, k) . if k = 0 then n else g (n + k, k - 1)`
4. `s = rek t`

Hkratna rekurzivna definicija

Kaj pa definicija rekurzivnih funkcij `f` in `g`, ki kličeta druga drugo? Primer: funkcija `f` kliče `f` in `g`, funkcija `g` pa kliče `f`:

```
let rec f x = if x = 0 then 1 + f (x - 1) else 2 + g (x - 1)
    and g y = if y = 0 then 1 else 3 * f (y - 1)
```

Če obravnavamo `f` in `g` skupaj kot urejeni par `(f, g)` dobimo

```
(f, g) = ((λ x . if x = 0 then 1 + f (x - 1) else 2 + g (x - 1)),
          (λ y . if y = 0 then 1 else 3 * f (y - 1)))
```

To je *rekurzivna definicija urejenega para (funkcij)*.

kar prepišemo v

```
(f, g) = t (f, g)
```

kjer je

```
t = λ (f', g') . ((λ x . if x = 0 then 1 + f' (x - 1) else 2 + g' (x - 1)),
                  (λ y . if y = 0 then 1 else 3 * f' (y - 1)))
```

Torej tudi za hkratne rekuzrivne definicije velja, da so to **negibne točke**.

Iteracija je poseben primer rekurzije

V proceduralnem programiranju poznamo zanke, na primer zanko `while`. Ali je tudi ta negibna točka? Če upoštevamo ekvivalenco

```
while b do c done
```

in

```
if b then (c ; while b do c done) else skip
```

vidimo, da je zanka `while b do c done` negibna točka. Če pišemo `W` za našo zanko:

```
W ≡ (if b then (c ; W) else skip)
```

Torej je `W` in s tem zanka `while b do c done` negibna točka funkcije:

```
t = (λ W . if b then (c ; W) else skip)
```

```
(while b do c done) = t (while b do c done)
```

Tudi **iteracija** je negibna točka!

Opomba: zanko `while` lahko na zgornji način "odvijamo v nedolged":

Faza 0:

```
while b do c done
```

Faza 1:

```
if b
then
  (c ; while b do c done)
else skip
```

Faza 2:

```
if b
then
  (c ;
   if b
   then
     (c ; while b do c done)
   else skip
  )
```

```
else skip
```

Faza 3:

```
if b
then
  (c ;
   if b
   then
     (c ;
      if b
      then
        (c ; while b do c done)
      else skip
     )
   else skip
  )
else skip
```

In tako naprej. Če bi lahko imeli neskončno programsko kodo, ne bi potrebovali zank!

Rekurzivne podatkovne strukture

Rekurzivno lahko definiramo tudi ostale strukture, ne samo funkcije. Na primer, neskončni seznam

```
ℓ = [1; 2; 1; 2; 1; 2; ...]
```

lahko definiramo kot

```
ℓ = 1 :: 2 :: ℓ
```

OCaml dopušča take definicije v omejeni meri (in tudi niso uporabne, ker je OCaml neučakan jezik). Haskell omogoča splošne rekurzivne definicije podatkov.

Rekurzivni tipi

Do sedaj smo spoznali podatkovne tipe:

- produkt `a * b` in zapisi
- vsota `a + b`
- eksponent `a → b`

S temi konstrukcijami ne moremo dobro predstaviti bolj naprednih podatkovnih tipov, kot so seznami in drevesa. Poglejmo si na primer, kako se tvori sezname celih števil:

- prazen seznam: `[]` je seznam
- sestavljen seznam: če je `x` celo število in `ℓ` seznam, je tudi `x :: ℓ` seznam

Zapis `[1; 2; 3]` je okrajšava za `1 :: (2 :: (3 :: []))`.

Seznami so **rekurzivni podatkovni tip**, saj gradimo sezname iz seznamov. Brez uporabe posebnih oznak `[]` in `::` bi zgornjo definicijo zapisali takole (oznaki `Nil` in `Cons` izhajata iz programskega jezika LISP, kjer pišemo `nil` in `(cons x ℓ)`):

- prazen seznam: `Nil` je seznam
- sestavljen seznam: če je `x` celo število in `ℓ` seznam, je tudi `Cons (x, ℓ)` seznam

Seznam `[1; 2; 3]` je okrajšava za `Cons (1, Cons (2, Cons (3, Nil)))`.

V OCamlu se tako definicijo zapiše takole:

```
type seznam =
  | Nil
  | Cons of int * seznam
```

Spet imamo opravka z rekuzijo. Tipi, ki se sklicujejo sami nase v svoji definiciji, se imenujejo **rekurzivni tipi**.

In spet vidimo, da je rekuzija negibna točka. Podatkovni tipi `seznam` je negibna točka za preslikavo `T`, ki slika tipe v tipe:

```
seznam = T (seznam)
```

kjer je `T` definiran kot

```
T (a) = (Nil | Cons of int * a)
```

Z besedami: `T` je funkcija, ki sprejme poljuben tip `a` in vrne vsoto tipov `Nil | Cons of int * a`.

Induktivni tipi

Izhajamo iz definicije seznama:

```
type seznam = Nil | Cons of int * seznam
```

Vprašajmo se: ali ta definicija zajema neskončne sezname? Na primer:

```
Cons (1, Cons (2, Cons (3, Cons (4, Cons (5, ⋯)))))
```

Ali se mora to kdaj zaključiti z `Nil`? Možna sta dva odgovora. Če zahtevamo, da morajo biti elementi rekurzivnega tipa končni, govorimo o *induktivnih* tipih. Če pa dovolimo neskončne elemente, govorimo o *koinduktivnih* tipih.

Poglejmo si najprej **induktivne podatkovne tipe**. To so rekurzivni tipi, v katerih vrednosti sestavljamo začenši z osnovnimi s pomočjo konstruktorjev in neskončne vrednosti niso dovoljene. Primeri:

1. naravna števila
2. končni seznami
3. končna drevesa
4. abstraktna sintaksa jezika:
    - programski jeziki
    - jeziki za označevanje podatkov
5. hierarhija elementov v uporabniškem vmesniku

Primer: naravna števila

Definicija naravnega števila:

- `0` je naravno število
- če je `n` naravno število, je tudi `n⁺` naravno število (ki mu rečemo "naslednik `n`")

Deficija podatkovnega tipa:

```
type stevilo = Nic | Naslednik of stevilo
```

Ta definicija ni učinkovita, ker predstavi naravna števila z naslednikom, torej v "eniškem" sistemu. Naravna števila bi lahko definirali tudi takole:

- `0` je naravno število
- če je `n` naravno število, je tudi `shl0 n` naravno število
- če je `n` naravno število, je tudi `shl1 n` naravno število

Ideja: z `shl0 n` predstavimo število `2 * n + 0` in z `shl1 n` predstavimo število `2 * n + 1`. Primer: število

```
shl0 (shl1 (shl0 (shl1 0)))
```

je število `10`. Kot podatkovni tip:

```
type stevilo = Zero | Shl0 of stevilo | Shl1 of stevilo
```

Vendar to še vedno ni optimalna rešitev, ker lahko število nič predstavimo na neskončno načinov:

```
0 = Shl0 0 = Shl0 (Shl0 0) = Shl0 (Shl0 (Shl0 0)) = ⋯
```

Vaja: poišči predstavitev dvojiških števil z induktivnimi tipi (lahko jih je več), da bo imelo vsako nenegativno celo število natanko enega predstavnika.

Primer: JSON

Poglejmo si definicijo standarda JSON in iz nje izluščimo podatkovni tip:

```
type json =
  | String of string
  | Number of int
  | Object of (string * json) list
  | Array of json array (* Ocaml ima vgrajen array *)
  | True
  | False
  | Null
```

Primer rekurzivnega tipa, ki ni induktiven:

Rekurzivni tipi so lahko zelo nenavadni:

```
type d = Foo of (d -> bool)
```

Poskusite si predstavljati, kaj so vrednosti tega tipa...

Strukturna rekurzija

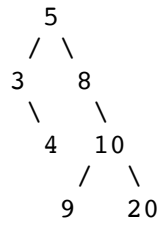
Ker so induktivni podatkovni tipi definirani rekurzivno, jih običajno obdelujemo z rekurzivnimi funkcijami. Kot primer si oglejmo, kako bi implementirali iskalna drevesa.

Obravnavajmo preprosta **iskalna drevesa**, v katerih hranimo cela števila. Iskalno drevo je

- bodisi **prazno**
- bodisi **sestavljeno** iz korena, ki je označen s številom `x`, ter dveh poddreves `l` in `r` pri čemer velja:
  - vsa števila v vozliščih `l` so manjša od `x`,

- vsa števila v vozliščih `r` so večja od `x`

Primer:



Podatkovni tip v OCaml se glasi:

```
type searchtree = Empty | Node of int * searchtree * searchtree
```

V tipu *nismo* shranili informacije o tem, da je iskalno drevo urejeno! Če bo programer ustvaril iskalno drevo, ki ni pravilno urejeno, prevajalnik tega ne bo zaznal.

Vaja: Sestavi funkcije za iskanje, vstavljanje in brisanje elementov v iskalnem drevesu.

Koinduktivni tipi

Ponovimo osnovno o koinduktivnih tipih

Poznamo še en pomembno vrsto rekurzivnih tipov, to so **koinduktivni tipi**. Pojavljajo se v računskih postopkih, ki so po svoji naravi lahko neskončni.

Tipičen primer koinduktivnega tipa je **komunikacijski tok podatkov:**

- bodisi je tok podatkov prazen (komunikacije je konec)
- bodisi je na voljo sporočilo `x` in preostanek toka

Primere take komunikacije najdemo povsod, kjer program komunicira z okoljem ali z drugim programom: odjemalec s strežnikom, dva programa med seboj, program z uporabnikom ipd.

Če preberemo zgornjo definicijo kot induktivni tip, se ne razlikuje od definicije seznamov. To bi pomenilo, da bi moral biti komunikacijski tok vedno končen, kar je nespametna predpostavka. V praksi seveda komunikacija ni *dejansko* neksnončna, a je *potencialno* neskončna, kar pomeni, da lahko dva procesa komunicirata v nedogled in brez vnaprej postavljene omejitve.

Koinduktivni tipi so rekurzivni tipi, ki dovoljujejo tudi neskončne vrednosti. Vendar pozor, kadar imamo opravka z neskončno velikimi seznami, drevesi itd., moramo paziti, kako z njimi računamo. Izogniti se moramo temu, da bi neskončno veliko drevo ali komunikacijski tok poskušali izračunati v celoti do konca.

Haskell ima koinduktivne podatkovne tipe.

Tokovi

Poglejmo si različico tokov, ki so vedno neskončni, ker pri njih koinduktivna narava pride še bolj do izraza. Tok je:

- sestavljen iz sporočila in preostanka toka

Če to definicijo preberemo induktivno, dobimo *prazen* tip, saj ne moremo začeti. Res, če zapišemo v OCaml

```
type 'a stream = Cons of 'a * 'a stream
```

dobimo podatkovni tip, ki nima nobene končne vrednosti. Vrednost bi bila nujno neskončna, na primer:

```
Cons (1, Cons (2, Cons (3, Cons (4, …))))

Cons (1, Cons (1, Cons (1, Cons (1, …))))
```

OCaml sicer dopušča *nekatere* take vrednosti z definicijo `let rec`:

```
# let rec s = Cons (1, Cons (2, s)) ;;
val s : int stream = Cons (1, Cons (2, <cycle>))
```

A te vrednosti le pokvarijo induktivno naravo tipov, hkrati pa ne dovoljujejo poljubnih neskončnih vrednosti. Če bi imele v praksi uporabno vrednost, bi jih morda tolerirali, ker pa jih redkokdaj uporabimo, smo lahko nekoliko razočarani nad odločitvijo snovalcev OCamla, da jih dovolijo.

Tokovi v Haskellu

Ista definicija v Haskellu deluje, ker ima Haskell koinduktivne tipe.

V Haskellu podatkovne tipe pišemo nekoliko drugače:

- imena tipov se piše z velikimi začetnicami: `Bool`, `Integer`, ...
- produkt tipov `a` in `b` zapišemo `(a,b)`, se pravi tako kot urejene pare. Na primer, elmenti tipa `(Bool, Int)` so `(False, 0)`, `(False, 42)`, `(True, 23)` itd.
- enotski tip pišemo `()`, torej tako kot njegovo edino vrednost.
- zapis `e :: t` pomeni "`e` ima tip `t`", zapis `e : t` pa seznam z glavo `e` in repom `t` (ravno obratno, kot v OCamlu)
- podatkovni tip uvedemo z določilom `data` in parameter pišemo za ime tipa z malimi črkami. Torej namesto `'a stream` zapišemo `Stream a`.

A to so le podrobnosti konkretne sintakse.

Poglejmo si definicijo tokov:

```
data Stream a = Cons (a, Stream a)

-- tok iz samih enic
enice :: Stream Integer
enice = Cons (1, enice)

-- prvih n elementov toka daj v seznam
to_list :: Integer -> Stream a -> [a]
to_list 0 _ = []
to_list n (Cons (x, s)) = x : (to_list (n-1) s)

-- tok števil od k naprej
from :: Integer -> Stream Integer
from k = Cons (k, from (k + 1))
```

Tokovi v OCaml

V OCaml lahko *simuliramo* tokove z uporabo tehnike *zavlačevanja* (angl. "thunk").

Denimo da imamo izraz `e` tipa `t`, ki ga zaenkrat še ne želimo izračunati. Tedaj ga lahko predelamo v funkcijo `fun () -> e` (po angleško se imenuje taka funkcija *thunk*), ki je tipa `unit -> t`. Ker je `e` znotraj telesa funkcije, se bo izračunal šele, ko funkcijo uporabimo na `()`. Od tod dobimo idejo, kako bi predstavili t.i. lene vrednosti v OCaml:

```
type 'a stream = Cons of 'a * (unit -> 'a stream)
```

Primeri uporabe so v stream.ml.

Vhod/izhod kot koinduktivni tip

Še en primer koinduktivnega tipa je vhod/izhod (input/output). Tokrat s koinduktivnim tipom izrazimo strukturo programa, ki izvaja operaciji `Read` in `Write`, glej io.hs.