Rekurzija in rekurzivni tipi

Splošna oblika rekurzije

Obravnavajmo rekurzivno funkcijo `f`, ki računa faktorielo:

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

```
def telo(g, n):
    if n == 0:
        return 1
    else:
        return n * g (n - 1)

def f(n):
    return telo(f, n)
```

Samo drugi del je rekurziven. Zapišimo to še v SML. Prvotna definicija faktoriele:

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

Ko ločimo definicijo od rekurzivnega sklica:

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

fun f n = telo f n
```

Še malo drugače:

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

fun 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 `t` rekurzivne definicije in vrne pripadajočo rekurzivno funkcijo:

```
fun rek t = (fn x => t (rek t) x)
```

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

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

Poglejmo si tip funkcije `rek`. SML 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:

1. `f n = (if n = 0 then 1 else n * f (n - 1))`
2. `f = (λ n . if n = 0 then 1 else n * f (n - 1))`
3. `f = t f` kjer je `t = (λ g . if n = 0 then 1 else n * g (n - 1))`
4. `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, $h(x_0)$, $h(h(x_0))$, $h(h(h(x_0)))$, …

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 `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 = (fn (n, k) => if k = 0 then n else s (n + k, k - 1))`
3. `s = t s`, kjer je `t = (fn g => fn (n, k) => if k = 0 then n else g (n + k, k - 1))`

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`:

```
fun 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)
```

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

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

Faza 2:

```
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!

Rekurzivni tipi

Do sedaj smo spoznali podatkovne tipe:

- produkt `a * b`
- 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 SML se tako definicijo zapiše takole:

```
datatype 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:

```
datatype 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`?

Posebej pomembni so **induktivni podatkovni tipi**. 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:

```
datatype stevilo = Nic | Naslednik of stevilo
```

Ta definicija ni učinkovita. Poskusimo 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:

```
datatype stevilo = zero | shl0 of stevilo | shl1 of stevilo
```

Ni optimalno:

```
0 = shl0 0 = shl0 (shl0 0)
```

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

Primer: JSON

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

```
datatype json
  = String of string
  | Number of int
  | Object of (string * json) list
  | Array of json array (* vgrajeni array *)
  | True
  | False
  | Null
```

Primer rekurzivnega tipa, ki ni induktiven:

Rekurzivni tipi so lahko želo nenavadni:

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

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

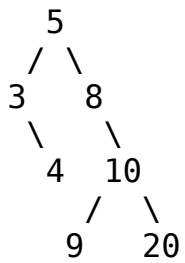
Strukturna rekurzija

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 SML se glasi:

```
datatype 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 in leno računanje

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 je **komunikacijski tok podatkov:**

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

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.