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 neskonči.

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.

Tokovi

Poglejmo si različico tokov, ki so 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 SML

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

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

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

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

Tokovi v Haskellu

Ista definicija v Haskellu deluje, ker ima Haskell koinduktivne tipe. Poglejmo si to na primeru.

Tokovi v SML

V SML lahko *simuliramo* tokove z uporabo tehnike *zavlačevanja* (angl. "thunk"). Imamo težavo, da hoče SML takoj izračunati preostanek toka. V splošnem lahko "zavlačujemo" z računanjem izraza `e` tako, da ga zapakiramo v funkcijo `fn () => e` in dobimo "thunk". Kasneje ga lahko "aktiviramo" tako, da ga uporabimo na `()`.

Izpeljava tipov

Kako programski jeziki uporabljajo tipe

Skoraj vsi programski jeziki imajo tipe, razlikujejo pa se po tem, kako se le-ti uporabljajo.

Kako striktni so tipi

Tipi so lahko bolj ali manj **striktni**. Če so popolnoma striktni, ima vsak izraz v veljavnem programu tip (SML, OCaml, Haskell, Java, C++). Lahko se zgodi, da veljavni program nima tipa, ali vsaj ne takega, ki bi dobro opisal njegovo delovanje (Javascript, Python).

Primer: nabori v Pythonu imajo zelo ohlapen tip `tuple`, ki ne pove nič več kot to, da gre za urejeno večterico:

```
>>> type((1, 'foo', False))
<type 'tuple'>
```

V SML so tipi striktni. Tip urejene trojice je bolj informativen:

```
- (1, "foo", false) ;
val it = (1,"foo",false) : int * string * bool
```

Dinamični in statični tipi

Poznamo delitev glede na *fazo*, v kateri se uporabijo tipi:

- Programski jezik ima **statične tipe**, če preveri ali izpelje tipe v *statični fazi*, se pravi ob prevajanju ali nalaganju kode, preden se koda požene. Primeri: C, C++, Java, C#, SML, OCaml, Haskell, Swift, Scala.

- Programski jezik ima **dinamične tipe**, če preverja tipe v *dinamični fazi*, se pravi, ko se koda izvaja. Primeri: Scheme, Racket, Javascript, Python.

Preverjanje in izpeljevanje tipov

Programski jezik lahko tipe **preverja** ali **izpeljuje**:

- **preverja** jih, če programer v večji meri zapiše tipe spremenljivk, funkcij in atributov, programski jezik pa preveri, da so pravino uporabljeni. Primeri: C, C++, Java, C#.

- **izpeljuje** jih, če programerju ni treba podajati tipov spremenljivk, funkcij in atributov (lahko pa jih, če to želi), programski jezik pa sam ugotovi, kakšnega tipa so. Primeri: SML, OCaml, Haskell.

Monomorfni in polimorfni tipi

Tipi so lahko:

- **monomorfni**, če ima vsak izraz največ en tip
- **polimorfni**, če ima lahko izraz hkrati več različnih tipov

Poznamo več vrst polimorfizma, danes bomo obravnavali **parametrični polimorfizem**.

Izpeljava tipov

Programski jeziki kot so SML, OCaml in Haskell imajo polimorfne tipe, ki jih izpeljejo z algoritmom, ki sta ga razvila Hindley in Milner.

Kakšen tip ima funkcija `λ x . x`, oziroma v SML `fn x => x`? Možnih je veliko odgovorov:

- `int → int`
- `bool → bool`
- `int * int → int * int`
- `α list -> α list` za poljuben `α`
- `β → β` za poljuben `β`.

Od vseh je zadnji najbolj splošen, ker lahko vse ostale dobimo tako, da **parameter** `β` zamenjamo s kakim drugim tipom. Pravimo, da je `β → β` *glavni* tip funkcije `fn x => x`.

Definicija: Tip izraza je **glavni**, če lahko vse njegove tipe dobimo tako, da v glavnem tipu parametre zamenjamo s tipi (ki lahko vsebujejo nadaljne parameter).

SML je načrtovan tako, da ima vsak veljaven izraz glavni tip, ki ga SML sam izpelje.

Postopek izpeljave glavnega tipa

Glavni tip izraza `e` izpeljemo v dveh fazah:

1. Izračunamo kandidata za tip `e`, ki vsebuje neznanke, in enačbe, ki jim morajo neznanke zadostovati
2. Rešimo enačbe s postopkom *združevanja*.

Druga faza se lahko zalomi, če se izkaže, da enačbe nimajo rešitve.

Prva faza

V prvi fazi izračunamo kandidata za tip in nabiramo enačbe, ki morajo veljati:

- `true` ima tip `bool`, brez enačb
- `false` ima tip `bool`, brez enačb
- celoštevilska konstanta `0`, `1`, `2`, … ima tip `int`, brez enačb
- spremenljivka ima svoj dani tip (tipe spremenljivk sproti beležimo v *kontekstu*)
- aritmetični izraz $e_1 + e_2$:
    - izračunamo tip τ_1 izraza e_1 in dobimo še enačbe E_1
    - izračunamo tip τ_2 izraza e_2 in dobimo še enačbe E_2

  Tip izraza $e_1 + e_2$ je `int`, z enačbami E_1, E_2 in $\tau_1 = \mathtt{int}$, $\tau_2 = \mathtt{int}$
  Podobno obravnavamo ostale aritmetične izraze $e_1 * e_2$, $e_1 - e_2$, …
- boolov izraz e_1 `and` e_2: obravnavamo podobno kot aritmetični izraz, le da uporabimo pričakovani `bool` namesto `int`.
- primerjava celih števil $e_1 < e_2$:
    - izračunamo tip τ_1 izraza e_1 in dobimo še enačbe E_1
    - izračunamo tip τ_2 izraza e_2 in dobimo še enačbe E_2

  Tip izraza $e_1 < e_2$ je `bool`, z enačbami E_1, E_2 in $\tau_1 = \mathtt{int}$, $\tau_2 = \mathtt{int}$
- pogojni stavek `if` e_1 `then` e_2 `else` e_3:
    - izračunamo tip τ_1 izraza e_1 in dobimo še enačbe E_1
    - izračunamo tip τ_2 izraza e_2 in dobimo še enačbe E_2
    - izračunamo tip τ_3 izraza e_3 in dobimo še enačbe E_3

  Tip izraza `if` e_1 `then` e_2 `else` e_3 je τ_2, z enačbami E_1, E_2, E_3, $\tau_1 = \mathtt{bool}$, $\tau_2 = \tau_3$

- urejeni par `(e₁, e₂)`:

  - izračunamo tip τ_1 izraza e_1 in dobimo še enačbe E_1
  - izračunamo tip τ_2 izraza e_2 in dobimo še enačbe E_2

  Tipi izraza (e_1, e_2) je $\tau_1 \times \tau_2$, z enačbami E_1, E_2.

- prva projekcija `fst e`:

  - izračunamo tip τ izraza `e` in dobimo še enačbe `E`

  Uvedemo nova parametra α in β (se ne pojavljata v `E`). Tip izraza `fst e` je α, z enačbami E_1, $\tau = \alpha \times \beta$.

- druga projekcija `snd e`:

  - izračunamo tip τ izraza `e` in dobimo še enačbe `E`

  Uvedemo nova parametra α in β. Tip izraza `snd e` je β, z enačbami E_1, $\tau = \alpha \times \beta$.

- funkcija `fn x => e`: uvedemo nov parameter α in zabeležimo, da ima `x` tip α, ter

  - izračunamo tip τ izraza `e` (pri predpostavki, da ima `x` tip α) in dobimo še enačbe `E`

  Tip funkcije `fn x => e` je $\alpha \to \tau$ z enačbami `E`

- aplikacija $e_1\ e_2$:

  - izračunamo tip τ_1 izraza e_1 in dobimo še enačbe E_1
  - izračunamo tip τ_2 izraza e_2 in dobimo še enačbe E_2

  Uvedemo nov parameter α. Tip izraza $e_1\ e_2$ je α, z enačbami E_1, E_2, $\tau_1 = \tau_2 \to \alpha$

- rekurzivna definicija `x = e` (kjer se `x` pojavi v `e`): uvedemo nov parameter α, zabeležimo, da ima `x` tip α, ter

  - izračunamo tip τ izraza `e` (pri predpostavki, da ima `x` tip α) in dobimo še enačbe `E`

  Tip izraza `x` je τ, z enačbami `E`, $\alpha = \tau$. Opomba: običajno na ta način definiramo rekurzivne funkcije, torej bo `x` v resnici funkcija.

Druga faza: združevanje

Imamo množico enačb `E`

$$
\begin{aligned}
l_1 &= d_1 \\
l_2 &= d_2 \\
l_3 &= d_3
\end{aligned}
$$

```
...
lᵢ = dᵢ
```

v neznankah α, β, γ, δ, … Rešujemo z naslednjim postopkom:

1. Imamo seznam rešitev `r`, ki je na začetku prazen.
2. Če je `E` prazna množica, vrnemo rešitev `r`
3. Sicer iz `E` odstranimo katerokoli enačbo `l = d` in jo obravnvamo:
    - če sta leva in desna stran povsem enaki, enačbo zvržemo ter gremo na korak 2
    - če je enačba oblike `α = d`, kjer je `α` neznanka:
        - če se `α` pojavi v `d`, postopek prekinemo, ker *ni rešitve*
        - sicer smo našli rešitev za `α`, namreč `α ↦ d`. Povsod v `r` in `E` zamenjamo `α` z `d` in v `r` dodamo rešitev `α ↦ d`
    - če je enačba oblike `l = α`, kjer je `α` neznanka, imamo primer, ki je simetričen prejšnjemu
    - če je enačba oblike $(l_1 \to l_2) = (d_1 \to d_2)$, v `E` dodamo enačbi $l_1 = d_1$ in $l_2 = d_2$ in gremo na korak 2
    - če je enačba oblike $(l_1 \times l_2) = (d_1 \times d_2)$, v `E` dodamo enačbi $l_1 = d_1$ in $l_2 = d_2$ in gremo na korak 2
    - če je enačba katerekoli druge oblike, na primer $(l_1 \to l_2) = (d_1 \times d_2)$, postopek prekinemo, ker *ni rešitve*.

Kako to deluje, si poglejmo na primerih.

Primer 1

Izpelji glavni tip funkcije

`fn x => x + 3`

Odgovor:

Primer 2

Izpelji glavni tip funkcije

`fn f x => f (x, x)`

Odgovor:

Primer 3

Izpelji glavni tip izraza

```
if 3 < 5 then (fn x => x) else (fn y => y + 3)
```

Odgovor:

Primer 4

Izpelji glavni tip izraza

```
if 3 < 5 then (fn x => x) else (fn y => (y, y))
```

Odgovor:

Primer 5

Izpelji glavni tip rekurzivne funkcije

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

Odgovor:

Churchovi numerali

Kakšen je tip števila 3?

```
0 = (λ f x . x)
1 = (λ f x . f x)
2 = (λ f x . f (f x))
3 = (λ f x . f (f (f x)))
```

To naj izračuna SML:

```
val zero  = (fn f => fn x => x) ;
val one   = (fn f => fn x => f x) ;
val two   = (fn f => fn x => f (f x)) ;
val three = (fn f => fn x => f (f (f x))) ;
```

Churchovi-Scottovi numerali

Kakšen je tip števila 3?

```
0 = (λ f x . x)
1 = (λ f x . f 0 x)
2 = (λ f x . f 1 (f 0 x))
3 = (λ f x . f 2 (f 1 (f 0 x)))
```

To naj izračuna SML:

```
val zero  = (fn f => fn x => x) ;
val one   = (fn f => fn x => f zero x) ;
val two   = (fn f => fn x => f one (f zero x)) ;
val three = (fn f => fn x => f two (f one (f zero x))) ;
```