Deklarativno programiranje

Z λ-računom smo spoznali uporabno vrednost funkcij in dejstvo, da lahko z njimi programiramo na nove in zanimive načine. A kot programski jezik λ-račun ni primeren, saj je zelo neučinkovit, poleg tega pa se programer večino časa ukvarja s kodiranjem podatkov s pomočjo funkcij. Da ne omenjamo grozne sintakse in neučinkovitosti.

Obdržimo, kar ima λ-račun koristnega, a ga nato nadgradimo z manjkajočimi koncepti. Pomembna spoznanja so:

1. **Funkcije so podatki.** V programskem jeziku lahko funkcije obravnavamo enakovredno vsem ostalim podatkom. To pomeni, da lahko funkcije sprejmejo druge funkcije kot argumente, ali jih vrnejo kot rezultat, da lahko tvorimo podatkovne strukture, ki vsebujejo funkcije ipd.

2. **Program ni nujno zaporedje ukazov.** V λ-računu program *ni* navodilo, ki pove, kako naj se izvede zaporedje ukazov. Kot smo videli, je vrstni red računanja nedoločen, saj je v splošnem možno izraz v λ-računu poenostaviti na več načinov (ki pa vsi vodijo do istega odgovora).

Kakšne vrste programiranje pa potemtakem je λ-račun, če ni ukazno? Nekateri uporabljajo izraz **funkcijsko programiranje**, mi pa bomo raje rekli **deklarativno programiranje**. S tem izrazom želimo poudariti, da s programom izrazimo (najavimo, deklariramo) strukturo podatka, ki ga želimi imeti, ne pa nujno kako se izračuna. Postopek, s katerim pridemo do rezultata je nato v večji ali manjši meri prepuščen programskemu jeziku.

Podatki

V λ-računu moramo vse podatke predstaviti, ali *kodirati*, s funkcijami. Tako opravilo je zamudno in podvrženo napakam, ker krši načelo:

> **Programski jezik naj programerju omogoči neposredno izražanje idej.**

Če mora programer neki koncept v programu izraziti tako, da jo simulira s pomočjo drugih konceptov, je večja možnost napake. Poleg tega prevajalnik ne bo imel informacije o tem, kaj programer počne, zato bo prepoznal manj napak in imel manj možnosti za optimizacijo.

Ponazorimo to načelo z idejo. Denimo, da želimo računati s seznami. Potem od programskega jezika pričakujemo *neposredno* podporo za sezname: sezname lahko preprosto naredimo, jih analiziramo, podajamo kot argumente. Ali programski jeziki, ki jih že poznamo, podpirajo sezname? Poglejmo:

- **C**: sezname moramo simulirati s pomočjo struktur (`struct`) in kazalcev
- **Java**: sezname moramo simulirati z objekti
- **Python**: seznami so vgrajeni, z njimi lahko delamo neposredno

Python težavo torej reši tako, da ima sezname kar vgrajene v jezik. To je prikladna rešitev, vendar pa ne moremo pričakovati, da bomo lahko z vgrajenimi podatkovnimi strukturami zadovoljili vse potrebe. V vsakem primeru moramo progamerju omogočiti, da definira *nove* strukture in *nove* načine organiziranja idej, ki jih načrtovalec jezika ni vnaprej predvidel. Različni programski jeziki to omogočajo na različne načine:

- **C**: definiramo lahko strukture (`struct`), unije (`union`), uporabljamo kazalce, itd.
- **Java**: definiramo razrede in podatke organiziramo kot objekte
- **Python**: definiramo razrede in podatke organiziramo kot objekte

Zdi se, da se novejši jeziki vsi zanašajo na objekte. A to še zdaleč ni edina rešitev za predstavitev podatkov – in tudi ne najboljša. Spoznajmo *neporedne* konstrukcije podatkovnih tipov, ki *niso* simulacije s pomočjo kazalcev ali objektov. Navdih bomo vzeli iz matematike, kjer poznamo operacije, s katerimi gradimo množice.

Konstrukcije množic

V matematiki gradimo nove množice z nekaterimi osnovnimi operacijami, ki jih večinoma že poznamo, a jih vseeno ponovimo.

Zmnožek ali kartezični produkt

Zmnožek ali **kartezični produkt** množic `A` in `B` je množica, katere elementi se imenujejo *urejeni pari*:

- za vsak $x \in A$ in $y \in B$ lahko tvorimo urejeni par $(x, y) \in A \times B$

Če imamo element $p \in A \times B$, lahko dobimo njegovo **prvo komponento** $\pi_1(p) \in A$ in **drugo komponento** $\pi_2(p) \in B$. Pri tem velja:

$$\pi_1(x, y) = x$$
$$\pi_2(x, y) = y$$

Operacijama π_1 in π_2 pravimo **projekciji**.

Tvorimo lahko tudi zmnožek več množic, na primer $A \times B \times C \times D$, v tem primeru imamo urejene četvertice (x, y, z, t) in štiri projekcije, π_1, π_2, π_3 in π_4.

Vsota ali disjunktna unija

Vsota množic $A + B$ je množica, ki vsebuje dve vrsti elementov:

- za vsak $x \in A$ lahko tvorimo element $\iota_1(x) \in A + B$
- za vsak $y \in A$ lahko tvorimo element $\iota_2(x) \in A + B$

Predstavljamo si, da je vsota $A + B$ sestavljena iz dveh ločenih kosov `A` in `B` Simbola ι_1 in ι_2 sta *oznaki*, ki povesta, iz katerega kosa je element. To je pomembno, kadar tvorimo vsoto $A + A$. Če je $x \in A$, potem sta $\iota_1(x)$ in $\iota_2(x)$ *različna* elementa vsote $A + A$.

Operacijama ι_1 in ι_2 pravimo **injekciji**.

Vsoti pravimo tudi **disjunktna unija**. Ločiti jo moramo od običajne unije. V vsoti $A + B$ se `A` in `B` nikoli ne prekrivata, ker elemente označujemo z ι_1 in ι_2. V uniji $A \cup B$ so lahko nekateri elementi *hkrati* v `A` in v `B`. V skrajnem primeru imamo celo $A \cup A = A$, tako da je vsak element v obeh kosih.

Če imamo element $u \in A + B$, potem lahko *obravnavamo dva primera**, saj je `u` bodisi oblike $\iota_1(x)$ za neki $x \in A$ bodisi oblike $\iota_2(x)$ za neki $y \in B$. Matematiki ne poznajo prikladnega zapisa za obravnavanje primerov. Nasploh matematiki vsoto množic slabo poznajo in jo neradi uporabljajo (kdo bi vedel, zakaj). V programiranju so vsote izjemno koristne, a na žalost jih programski jeziki bodisi ne podpirajo bodisi implementirajo narobe.

Poglejmo si primer uporabe vsot v programiranju. Na primer, da v spletni trgovini prodajamo čevlje, palice in posode. Čevelj ima barvo in velikost, palica velikost in posoda prostornino. Če je `B` množica vseh barv in `N` množica naravnih števil, lahko izdelek predstavimo kot element množice

```
(B × N) + N + N
```

Res: črn čevelj velikosti 42 je element ι_1`(črna, 42)`, palica dolžine `7` je ι_2`(7)`, posoda s prostornino `7` pa je ι_3`(7)`. Oznake ι_2 in ι_3 ločita med palicami in posodami. Seveda je tak zapis s programerskega stališča nepraktičen, zato ga bomo v programskem jeziku izboljšali.

Eksponent ali množica funkcij

Eksponent B^A, ki ga pišemo tudi `A → B`, je množica vseh funkcij iz `A` v `B`. Če je `f ∈` B^A, pravimo, da je `A` **domena** in `B` **kodomena** funkcije `f`.

Dogovorimo se, da je → desno asociativna operacija, se pravi

```
A → B → C  =  A → (B → C)
```

Primeri:

1. `R → R` je množica realnih funkcij ene spremenljivke, na primer `sin`, `cos`, `exp`.
2. `R × R → R` je množica realnih funkcija dveh spremenljivk, na primer `+`, `×`, $(x, y) \mapsto \sqrt[3]{x^2 + y^3}$.
3. `R → R → R` je množica funkcij, ki sprejmejo eno realno število in vrnejo funkcijo, ki sprejme še eno realno število in vrne realno število, na primer `x ↦ (y ↦ x² + y³)`.
4. `(R → R) → R` je množica funkcij, ki sprejmejo realno funkcijo in vrnejo realno število, na primer `f ↦` $\int_0^1$ `f(x) dx` (določeni integral od `0` do `1`).

Več bomo o eksponentih povedali kasneje, ko jih bomo obravnavali kot podatkovne tipe v programskem jeziku.

Podatkovni tipi

V programskem jeziku ne govorimo o množicah, ampak o **tipih**, ki so podobni množicam, a so bolj splošni in imajo širšo uporabno vrednost. Tipi so zelo splošen in uporaben koncept, ki presega meje programiranja in celo računalništva. Tipi se uporabljajo tudi v logiki in drugih vejah matematike.

Programski jeziki lahko podpirajo tipe v večji ali manjši meri. V λ-računu ni nobenih tipov, C jih ima, prav tako Java. Če je `e` izraz tipa `T`, bomo to zapisali

```
e : T
```

Ta zapis nas spominja na zapis `e ∈ T` iz teorije množic. Vendar pa je bolje, da na tip `T` ne gledamo kot na "zbirko elementov", ampak kot na informacijo o tem, kakšen podatek je `e` in kaj lahko z njim počnemo.

Konstrukcije množic, ki smo jih spoznali, bomo predelali v konstrukcije tipov. V ta namen potrebujemo primer programskega jezika, ki neposredno podpira te konstrukcije. Izbrali bomo **OCaml**. (Lahko bi uporabili tudi SML, Haskell, Idris, ali Elm. Jeziki kot so C/C++, Java, Python in Javascript ne podpirajo konstrukcij, ki jih bomo obravnavali, lahko jih le bolj ali manj uspešno simuliramo.)

V OCaml se imena tipov piše z malo začetnico, zato bomo za imena tipov uporabljali male črke.

Spletni viri za OCaml

- Uradna spletna stran za OCaml
- Eksperimentalna verzija REPL za OCaml na repl.it (kaj pomeni REPL?)

Zmnožek tipov

Zmnožek tipov ali **kartezični produkt** `a * b` tipov `a` in `b` vsebuje urejene pare, ki jih v OCaml zapišemo enako, kot v matematiki:

```
        OCaml version 4.06.1

# (3, "banana") ;;
- : int * string = (3, "banana")
```

Zapisali smo urejeni par `(3, "banana")`. OCaml je ugotovil, da je tip tega urejenega para `int * string` in to izpisal.

Če želimo vpeljati definicijo, to naredimo z `let x = ...`:

```
# let i = 10 + 3 ;;
val i : int = 13
# let j = 100 + i * i ;;
val j : int = 269
```

Tvorimo lahko tudi urejene `n`-terice, za poljuben `n`:

```
# (1, "banana", false, 2) ;;
- : int * string * bool * int = (1, "banana", false, 2)
```

Projekciji π_1 in π_2 v OCamlu zapišemo `fst` in `snd` (okrajšavi za "first" in "second"):

```
# fst (1, "banana") ;;
- : int = 1
# snd (1, "banana") ;;
- : string = "banana"
```

Ostalih projekcij (tretja, četrta, peta, ...) ni, ker v praksi malokdaj uporabljamo urejene `n`-terice za `n > 2`. Namesto projekcij uporabimo *vzorce*:

```
# let (a, b, c) = (1, "banana", false) ;;
val a : int = 1
val b : string = "banana"
val c : bool = false
```

S podčrtajem lahko označimo komponente, ki jih ignoriramo:

```
# let (_, _, z) = (1, "banana", false) ;;
val z : bool = false
```

Takole bi si definirali tretjo projekcijo:

```
# let thd (_, _, z) = z ;;
val thd : 'a * 'b * 'c -> 'c = <fun>
# thd (1, "banana", false) ;;
- : bool = false
```

Enotski tip

Če smemo pisati urejene pare, trojice, četverice, …, ali smemo zapisati tudi "urejeno ničterico"? Seveda!

```
# () ;;
- : unit = ()
```

Dobili smo **enotski tip** `unit`. To je tip, ki ima en sam element, namreč urejeno ničterico `()`, ki ji pravimo **enota**. Zakaj se mu reče "enotski"? Ker je množica z enim elementom "enota za množenje". Matematiki namesto `unit` pišejo kar `1 = {*}`:

$A \cong 1 \times A$

Morda se zdi enotski tip neuporaben, a to ni res. V C in Java so ta tip poimenovali `void` ("prazen") in se ga uporablja za funkcije, ki ne vračajo rezultata. Tip `void` sploh ni prazen, ampak ima en sam element, ki pa ga programer nikoli ne vidi (in ga tudi ne more). Če namreč funkcija vrača v naprej predpisan element, potem vemo, kaj bo vrnila, in tega ni treba razlagati.

Zapomnimo si torej, da funkcija, ki "ne vrne ničesar" v resnici vrne `()`. V OCaml se to dejansko vidi, v Javi in C pa ne.

Kaj pa funkcija, ki "ne sprejme ničesar"? Če funkcija sprejme argumente `x`, `y` in `z`, potem sprejme urejeno trojico. Če ne sprejme ničesar, potem v resnici sprejme urejeno ničterico `()`, torej spet enoto.

Pa še to: morda ste si kdaj želeli, da bi lahko v C ali Java brez velikih muk napisali funkcijo, ki vrne dva rezultata? Jezik, ki ima zmnožke, to omogoča sam od sebe: preprosto vrnete urejeni par!

Zapis

Urejeni pari včasih niso prikladni, ker si moramo zapomniti vrstni red komponent. Na primer, polno ime osebe bi lahko predstavili z urejenim parom `("Mojca", "Novak")`, a potem moramo vedno paziti, da ne zapišemo pomotoma `("Novak", "Mojca")`. Težava nastopi tudi, ko imamo kompliciranе podatke. Na primer, podatke o trenutnem času bi lahko predstavili z naborom

```
(leto, mesec, dan, ura, minuta, sekunda, milisekunda)
```

Kdo si bo zapomnil, da so minute peto polje in milisekunde sedmo?

Težavo razrešimo tako, da komponent ne štejemo po vrsti, ampak jih poimenujemo. Dobimo tako imenovani tip *zapis* (angl. *record*). Najprej ga definiramo z deklaracijo `type`:

```
type oseba = { ime : string; priimek : string; }
```

S tem smo uvedli nov tip `oseba`, ki je zapis z dvema poljema. Sedaj lahko namesto urejenega para tvorimo zapis:

```
# { ime = "Mojca"; priimek = "Pokraculja" } ;;
- : oseba = {ime = "Mojca"; priimek = "Pokraculja"}
```

Torej je $\{\ell_1 = e_1; \ldots; \ell_i = e_i\}$ kot nabor $(e_1, \ldots, e_i)$, le da smo poimenovali njene komponente $\ell_1, \ldots, \ell_i$.

Težave z vrstnim redom izginejo, ker je v zapisu pomembno ime komponente in ne vrstni red:

```
# { priimek = "Pokraculja"; ime = "Pokraculja" } ;;
- : oseba = {ime = "Pokraculja"; priimek = "Pokraculja"}
```

Z zapisom lahko zapišemu tudi urejeno "enerico":

```
# type zajec = { masa : int } ;;
type zajec = { masa : int; }
```

Sedaj lahko tvorimo zapis z enim samim poljem:

```
# { masa = 42 } ;;
- : zajec = {masa = 42}
```

V Pythonu se to zapiše `("42",)`.

Do komponente z imenom `foo` v zapisu `s` dostopamo s `s.foo`:

```
# let mati = { ime = "Neza"; priimek = "Cankar" } ;;
val mati : oseba = {ime = "Neza"; priimek = "Cankar"}
# mati.ime ;;
- : string = "Neza"
# mati.priimek ;;
- : string = "Cankar"
```

Definicije tipov v OCaml

Videli smo že, da lahko s `type a = ...` definiramo zapise:

```
type complex = { re : float; im : float }

type datetime = { year : int;
                  month : int;
                  hour : int;
                  minute : int;
                  second : int;
                  millisecond : int }

type color = { red : float; green : float; blue : float }
```

Definiramo lahko tudi okrajšave za tipe, na primer:

```
type krneki = int * bool * string
```

Vsota tipov

Elemente vsote množic `A + B` smo označevali z ι_1 in ι_2. Izbor oznak je z matematičnega stališča nepomemben, namesto ι_1 in ι_2 bi lahko pisali tudi kaj drugega. V programiranju bomo to seveda izkoristili: tako kot smo uvedli zapise, ki so pravzaprav zmnožki s poimenovanimi komponentami, bomo uvedli vsote tipov, pri katerih si oznake izbere programer.

Če želimo imeti vsoto, jo moramo v OCaml najprej definirati s `type`, tako kot zapise. Zgornji primer izdelkov v spletni trgovini, bi zapisali takole:

```
type barva = { blue : float; green : float;  red : float }

type izdelek =
  | Cevelj of barva * int
  | Palica of int
  | Posoda of int
```

Ta definicija pravi, da je `izdelek` vsota treh tipov: prvi tip je zmnožek tipov `barva` in `int`. Drugi in tretji tip sta oba `int`. Za oznake smo izbrali `Cevelj`, `Palica` in `Posoda`. Tem oznakam v Ocaml pravimo **konstruktorji** (angl. constructor).

Črn čevelj velikosti `42` zapišemo

```
Cevelj ({blue=0.0; green=0.0; red=0.0}, 42)
```

palico velikosti `7`

```
Palica 7
```

in posodo s prostornino `7`

Razločevanje primerov

Kot smo omenili, potrebujemo zapis za *razločevanje primerov*. Nadaljujmo s primerom. Denimo, da je cena izdelka `z` določena takole:

- čevelj stane 15 evrov, če je številka manjša od 25, sicer stane 20 evrov
- palica dolžine `x` stane `1 + 2 * x` evrov
- posoda stane 7 evrov ne glede na prostornino

To v Ocaml zapišemo z `match`:

```
match z with
| Cevelj (b, v) -> if v < 25 then 15 else 25
| Palica x -> 1 + 2 * x
| Posoda y -> 7
```

Splošna oblika stavka `match` je

```
match e with
| p1 -> e1
| p2 -> e2
| p3 -> e3
  ⋮
| pi -> ei
```

Tu so p_1, ..., p_i **vzorci**. Vrednost izraza `match ...` je pvi e_j, za katerega `e` zadošča vzorcu p_j. V OCaml je `match` dosti bolj uporaben kot `switch` v C in Javi ali `if … elif … elif …` v Puthnu, ker OCaml izračuna, ali smo pozabili obravnavati kakšno možnost. Primer:

```
# match (Palica 7) with
          | Cevelj (b, v) -> if v < 35 then 15 else 25
          | Posoda y -> 7 ;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Palica _
Exception: Match_failure ("//toplevel//", 29, -65).
```

Včasih želimo uvesti tip, ki sestoji iz končnega števila konstant. To lahko naredimo z vsoto takole:

```
type t = Foo | Bar | Baz | Qux
```

V C je to tip `enum`, podobno v Javi. Imena konstruktorjev se morajo pisati z veliko začetnico. V OCaml bi lahko `bool` definirali sami, če ga še ne bi bilo:

```
type bool = False | True
```

Vzorci v stavku `match` so lahko poljubno gnezdeni. Denimo, da bi želeli ceno izračunati takole:

- čevelj stane 15 evrov, če je številka manjša od 25, sicer stane 20 evrov
- palica dolžine 42 sane 1000 evrov
- palica dolžine `x ≠ 42` stane `1 + 2 * x` evrov
- posoda stane 7 evrov ne glede na prostornino

Pripadajoči stavek `match` se glasi:

```
match z with
 | Cevelj (b, v) -> if v < 35 then 15 else 25
```

```
  | Palica 42 -> 1000
  | Palica x -> 1 + 2 * x
  | Posoda y -> 7
```

Vzorce lahko uporabljamo tudi v definicijah vrednosti `let`:

```
# let (Posoda p) = Posoda 10 ;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Palica _
val p : int = 10


# let Cevelj (x, y) = Cevelj ({red=1.0; green=0.5; blue=0.0}, 43) ;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
(Palica _|Posoda _)
val x : barva = {blue = 0.; green = 0.5; red = 1.}
val y : int = 43


# let Cevelj ({red=r;green=_;blue=b},v) = Cevelj ({red=1.0; green=0.5; blue=0.0}, 43) ;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
(Palica _|Posoda _)
val b : float = 0.
val r : float = 1.
val v : int = 43
```

Vzorcem se bomo bolj podrobno še posvetili.

Funkcijski tip

Funkcijski tip `a -> b` je tip funkcij, ki sprejmejo argument tipa `a` in vrnejo rezultat tipa `b`. V OCaml λ-abstrakcijo `λ x . e` zapišemo kot `fun x -> e`:

```
# fun x -> 2 * (x + 3) + 3 ;;
- : int -> int = <fun>
```

Veljajo podobna pravila kot v λ-računu. Na primer funkcije lahko gnezdimo:

```
# fun x -> (fun y -> 2 * x - y + 3) ;;
- : int -> int -> int = <fun>
```

OCaml je izračunal tip funkcije `int -> int -> int`. Operator `->` je *desno asociativen* , torej je

```
a → b → c  ≡  a → (b → c)
```

Tip `int -> int -> int` torej opisuje funkcije, ki sprejmejo `int` in vrnejo `int -> int`. Funkcije lahko tudi uporabljamo:

```
# (fun x -> (fun y -> 2 * x - y + 3)) 10 ;;
- : int -> int = <fun>
```

Ste razumeli, kaj naredi zgornji primer? Kaj pa tale:

```
# (fun x -> (fun y -> 2 * x - y + 3)) 10 3 ;;
- : int = 20
```

Funkcijo lahko poimenujemo:

```
# let f = fun x -> x * x + 1 ;;
val f : int -> int = <fun>
# f 10 ;;
- : int = 101
```

Namesto `let f = fun x ->` ⋯ lahko pišemo tudi `let f x =` ⋯

```
# let g x = x * x + 1 ;;
val g : int -> int = <fun>
# g 10 ;;
- : int = 101
```

Definicija funkcije je rekurzivna, če to naznanimo `let rec`:

```
# let rec fact n = (if n = 0 then 1 else n * fact (n - 1)) ;;
val fact : int -> int = <fun>
# fact 10 ;;
- : int = 3628800
```

Kot vidimo, OCaml sam izračuna tip funkcije. Pravzaprav vedno sam izračuna vse tipe. Pravimo, da tipe *izpelje* in s tem se bomo še posebej ukvarjali. Včasih kak tip ostane nedoločen, na primer:

```
# fun (x, y) -> (y, x) ;;
- : 'a * 'b -> 'b * 'a = <fun>
```

Tip `x` je poljuben, prav tako tip `y`. Ocaml ju zapiše z `'a` in `'b`. Znak apostrof označuje dejstvo, da sta to *poljubna* tipa, ali *parametra*. Še en primer:

```
# fun (x, y, z) -> (x, y + z, x) ;;
- : 'a * int * int -> 'a * int * 'a = <fun>
```

Ko zapišemo funkcijo, lahko podamo tip njenih argumentov:

```
# fun (x : string) -> x ;;
- : string -> string = <fun>
```

Brez oznake tipa OCaml izpelje najbolj splošen tip:

```
# fun x -> x ;;
- : 'a -> 'a = <fun>
```