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, zaradi katerih so programi neučinkoviti.)

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 neko idejo v programu izraziti tako, da jo simulira, je večja možnost napake. Poleg tega prevajalnik ne bo imel informacije o tem, kaj programer počne, in ne bo razpoznal manj napak in imel manj možnosti za optimizacijo.

Ponazorimo to načelo z idejo. Denimo, da želimo računati s seznami števil. 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. 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 ∈ A`, potem sta `ι₁(x)` in `ι₂(x)` *različna* elementa vsote `A + A`.

Operacijama `ι₁` in `ι₂` 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 `ι₁` in `ι₂`. V uniji `A ∪ B` so lahko nekateri elementi *hkrati* v `A` in v `B`. V skrajnem primeru imamo celo `A ∪ A = A`, tako da je vsak element v obeh kosih.

Če imamo element `u ∈ A + B`, potem lahko *obravnavamo dva primera**, saj je `u` bodisi oblike `ι₁(x)` za neki `x ∈ A` bodisi oblike `ι₂(x)` za neki `y ∈ 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 `ι₁(črna, 42)`, palica dolžine 7 je `ι₂(7)`, posoda s prostornino 7 pa je `ι₃(7)`. Oznake `ι₂` in `ι₃` ločita med palicami in posodami. Seveda je tak zapis s programerskega stališča nepraktičen, zato ga bomo 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`. O funkcijah tu ne bomo povedali veliko več, jih pa bomo s pridom uporabljali.

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 z

```
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 **Standardni ML**, ali krajše SML. (Lahko bi uporabili tudi OCaml, 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 SML se imena tipov piše z malo začetnico, zato bomo za imena tipov uporabljali male črke.

Zmnožek tipov

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

```
Standard ML of New Jersey v110.81 [built: Mon Oct  2 10:01:15 2017]
- (3, "banana") ;
val it = (3,"banana") : int * string
```

Zapisali smo urejeni par `(3, "banana")`. SML je ugotovil, da je tip tega urejenega para `int * string` in to izpisal. Z izpisom `val it = ...` je povedal še, da lahko zadnjo vrednost, ki smo jo izračunali, dobimo z `it`:

```
- 3 + 6 ;
val it = 9 : int
- it ;
val it = 9 : int
- 3 * 14 ;
val it = 42 : int
- it * 100 ;
val it = 4200 : int
```

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

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

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

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

Projekcije π_1, π_2, ... v SML pišemo kot #1, #2, ...:

```
- #2 (1, "banana", false, 2) ;
val it = "banana" : string
- #3 (1, "banana", false, 2) ;
val it = false : bool
- val p = (1, "banana", false, 2) ;
val p = (1,"banana",false,2) : int * string * bool * int
```

```
- #3 p ;
val it = false : bool
```

Enotski tip

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

```
- ( ) ;
val it = () : 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 ≅ 1 × 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 SML 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 komplicirane 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*):

```
- { ime = "Mojca", priimek = "Novak" } ;
val it = {ime="Mojca",priimek="Novak"} : {ime:string, priimek:string}
```

Torej je $\{x_1=e_1, \ldots, x_i=e_i\}$ kot urejena terica $(e_1, \ldots, e_i)$, le da smo poimenovali njene komponente x_1, ..., x_i. Tip zapisa pišemo $\{x_1:a_1, \ldots, x_i:a_i\}$. Če bi bil to kartezični produkt, bi ga zapisali $a_1 * \ldots * a_i$.

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

```
- { priimek = "Novak", ime = "Mojca" } ;
val it = {ime="Mojca",priimek="Novak"} : {ime:string, priimek:string}
```

V resnici so urejene večterice poseben primer zapisov, namreč takih, ki imajo polja po vrsti poimenovana 1, 2, 3, ...

```
- { 1 = 2, 2 = "banana", 3 = false, 4 = 2 } ;
val it = (2,"banana",false,2) : int * string * bool * int
```

Z zapisom lahko zapišemu tudi urejeno "enerico":

```
- {1 = "foo"};
val it = {1="foo"} : {1:string}
```

V Pythonu se to zapiše `("foo",)`.

Do komponente z imenom `foo` dostopamo s `#foo`:

```
val mati = {ime="Neza",priimek="Cankar"} : {ime:string, priimek:string}
- #ime mati ;
val it = "Neza" : string
```

Definicije tipov v SML

V SML lahko s `type a = ...` definiramo okrajšave za tipe, da jih ni treba vedno znova pisati. Na primer:

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

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

type color = { red : real, green : real, blue : real }

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 SML najprej definirati z `datatype`. Zgornji primer izdelkov v spletni trgovini, bi zapisali takole:

```
datatype izdelek
  = Cevelj of {blue:real, green:real, red:real} * int
  | Palica of int
  | Posoda of int
```

Ta definicija pravi, da je `izdelek` vsota treh tipov: prvi tip je zmnožek tipov `{blue:real, green:real, red:real}` in `int`. Drugi in tretji tip sta oba `int`. Za oznake smo izbrali `Cevelj`, `Palica` in `Posoda`. Tem oznakam v SML 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

```
Posoda 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 SML zapišemo s `case`:

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

Splošna oblika stavka `case` je

```
case e of
  p₁ => e₁
| p₂ => e₂
| p₃ => e₃
  ⋮
| pᵢ => eᵢ
```

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

```
-  case z of
     Cevelj (b, v) => if v < 35 then 15 else 25
   | Posoda y => 7 ;
= = stdIn:42.2-44.19 Warning: match nonexhaustive
          Cevelj (b,v) => ...
          Posoda y => ...

uncaught exception Match [nonexhaustive match failure]
  raised at: stdIn:44.19
```

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

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

V C je to tip `enum`. Imena konstruktorjev se lahko pišejo z malo začetnico. V SML bi lahko `bool` definirali sami, če ga še ne bi bilo:

```
datatype bool = false | true
```

V resnici SML zgornjo definicijo sprejme in z njo *prekrije* že obstoječo definicijo tipa `bool`, kar lahko pripelje do velike zmede!

Vzorci v stavku `case` 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 `case` se glasi:

```
case z of
   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 `val`:

```
- val (Posoda p) = Posoda 10 ;
val p = 10 : int

- val Cevelj (x, y) = Cevelj ({red=1.0,green=0.5,blue=0.0}, 43) ;
val x = {blue=0.0,green=0.5,red=1.0} : {blue:real, green:real, red:real}
val y = 43 : int

- val Cevelj ({red=r,green=g,blue=b},v) = Cevelj ({red=1.0,green=0.5,
 blue=0.0}, 43) ;
val b = 0.0 : real
val g = 0.5 : real
val r = 1.0 : real
val v = 43 : int
```

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 SML λ-abstrakcijo `λ x . e` zapišemo kot `fn x => e`:

```
- fn x => 2 * (x + 3) + 3 ;
val it = fn : int -> int
```

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

```
- fn x => (fn y => 2 * x - y + 3) ;
val it = fn : int -> int -> int
```

SML 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:

```
- (fn x => (fn y => 2 * x - y + 3)) 10 ;
val it = fn : int -> int
```

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

```
- (fn x => (fn y => 2 * x - y + 3)) 10 3 ;
val it = 20 : int
```

Funkcijo lahko poimenujemo:

```
- val f = fn x => x * x + 1 ;
val f = fn : int -> int
- f 10 ;
val it = 101 : int
```

Namesto `val f = fn x => ⋯` lahko pišemo tudi `fun f => ⋯`

```
- fun g x = x * x + 1 ;
val g = fn : int -> int
- g 10 ;
val it = 101 : int
```

Definicija s `fun` je lahko rekurzivna:

```
- fun fact n = (if n = 0 then 1 else n * fact (n - 1)) ;
val fact = fn : int -> int
- fact 10 ;
val it = 3628800 : int
```

Kot vidimo, SML 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:

```
- fn (x, y) => (y, x) ;
val it = fn : 'a * 'b -> 'b * 'a
```

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

```
- fn (x, y, z) => (x, y + z, x) ;
val it = fn : 'a * int * int -> 'a * int * 'a
```

Ko zapišemo funkcijo, lahko podamo tip njenih argumentov:

```
- fn (x : string) => x ;
val it = fn : string -> string
- fn x => x ; val it = fn : 'a -> 'a
```