Podtipi in objekti

Spoznali smo že *polimorfizem*, to je lastnost, da ima lahko izraz več kot en tip. Na primer, v OCamlu ima preslikava

```
fun (x, y) -> (y, x)
```

tip $\alpha \times \beta \to \beta \times \alpha$, kjer sta α in β poljubna tipa. Danes bomo spoznali **podtipe**, ki delujejo podobno: če ima izraz `e` tip `t`, ga lahko uporabljamo, kot da bi imel tudi vse nadtipe tipa `t`.

Na primer, v nekaterih programskih jezikih lahko izraz `e` tipa `int` vedno uporabimo, kot da bi ime tip `float`, saj bo jezik samo pretvoril `e`.

V Javi poznamo *podrazrede*, ki so tudi vrsta podtipov, a jih bomo obravnavali prihodnjič.

Relacija podtip $A \leq B$

Najprej se dogovorimo za zapis relacije "podtip". Za tipa `A` in `B` zapis

$$A \leq B$$

preberemo "`A` je podtip `B`" ali "`B` je nadtip `A`". Sledimo naslednjemu osnovnemu načelu:

> `A` je podtip `B`, če lahko vrednosti tipa `A` uporabljamo, kot da bi imele tip `B`.

S pravilom sklepanja to zapišemo takole:

$$\frac{e : A \qquad A \leq B}{e : B}$$

in preberemo: "Če ima `e` tipa `A` in je `A` podtip `B`, potem ima `e` tudi tip `B`"

Pozor: zmotno bi si bilo predstavljati, da je $A \leq B$ isto kot $A \subseteq B$, se pravi, da je podtip ista reč kot množica. V razdelku o podtipih zapisov bomo namreč spoznali podtipe, ki se razlikujejo od relacije podmnožica.

Kaj pričakujemo od relacije podtip? Zapišimo pravila. Vsekakor je relacija *refleksivna* in *tranzitivna*:

$$\frac{}{A \leq A}$$

$$\frac{A \leq B \qquad B \leq C}{A \leq C}$$

Morda velja antisimetričnost: če $A \leq B$ in $B \leq A$, potem $A = B$? Ne, kasneje bomo videli protiprimer.

Nadalje zapišemo še pravila, ki opredeljujejo, kako se obnašajo podtipi za razne operacije na tipih. Kartezični produkti delujejo po pričakovanjih:

$$\frac{A_1 \leq B_1 \qquad A_2 \leq B_2}{A_1 \times A_2 \leq B_1 \times B_2}$$

Pravilo za funkcijske tipe je bolj zanimivo, pozorno ga preberite:

```
B₁ ≤ A₁          A₂ ≤ B₂
------------------------
   A₁ → A₂ ≤ B₁ → B₂
```

Obrnili smo vrsti red v domeni! Pravimo, da je → **kontravarianten** (obrača smer) v prvem argumentu in **kovarianten** (ne obrača smeri) v drugem. Skupaj premislimo, zakaj pričakujemo tako pravilo.

Kaj pa osnovni tipi? Ali je na primer `int ≤ float`? To je odvisno od programskega jezika: Java samodejno pretvori celo število v realno, zato bi lahko rekli, da v javi velja `int ≤ float`. V OCamlu to ne drži, saj mora programer sam pretvoriti celoštevilsko vrednost v realno s funkcijo `float`.

Lahko se celo zgodi, da v programskem jeziku velja `int ≤ float` *in hkrati* `float ≤ int`, na primer, če jezik samodejno zaokroži realno vrednost, kadar potrebuje celoštevilsko. A takih jezikov in veliko.

Podtipi zapisov

Spoznali smo že zapise, to so nabori polj s poimenovanimi komponentami. Na primer, točke v ravnini lahko predstavimo z vrednostmi tipa zapisov

```
{ x : float; y : float }
```

Primer vrednosti tega tipa je zapis `{x = 3.14; y = 2.78}`.

V splošnem je tip zapisa

```
{ ℓ₁ : A₁; ℓ₂ : A₂; …; ℓ_n : A_n }
```

njegove vrednosti pa so oblike

```
{ ℓ₁ = e₁; ℓ₂ = e₂; …; ℓ_n = e_n }
```

kjer mora veljati `eᵢ : Aᵢ`. Zahtevamo še, da so imena polj `ℓ₁, …, ℓ_n` med seboj paroma različna.

Če je `v` zapis tipa `{ ℓ₁ : A₁; ℓ₂ : A₂; …; ℓ_n : A_n }`, potem je `v.ℓᵢ` vrednost njegove `i`-te komponente, ki ima tip `Aᵢ`.

V zapisu vrstni red polj ni pomemben, to je, `{x = 3.14; y = 2.78} = {y = 2.78; x = 3.14}`.

Zapisi so uporaben podatkovni tip, ne samo v programiranju, ampak tudi na drugih področjih računalništva, kjer želimo predstviti strukturirane podatke. Na primer, vrstica v tabeli podatkovne baze ni nič drugega kot zapis (in shema za tabelo tip zapisa).

Tudi objekti nas nekoliko spominjajo na zapise, le da vsebujejo poleg polj (atributov) še metode:

```
public class Point {
   float x;
   float y

   ...
}
```

Zapisi tipov imajo zanimivo relacijo ≤.

Obravnavajmo primer. Na primer, da imamo tipa zapisov

```
A := { x : float; y : float }
```

in

```
B := { x : float; y : float ; z : float }
```

Ali morda velja `A ≤ B` ali `B ≤ A`?

Najprej, `A ≤ B` *ne* velja. Izraz `a = {x = 3.14; y = 2.78}` ima tip `A`. Če bi ga lahko uporabljali, kot da ima tip `B`, potem bi smeli pisati `a.z`, kar seveda ne gre.

Velja pa `B ≤ A`! Izraz `b = {x = 3.14; y = 2.78; z = 1.0}` ima tip `B` in res ga lahko uporabimo, kot da bi imel tip `A`, saj smemo pisati `b.x` in `b.y`. Dejstvo, da `b` vsebuje še polje `z` nas pri tem nič ne moti.

Zapišimo pravilo za podtipe zapisov, ki izhaja iz zgornjega razmisleka:

$$\frac{\text{za vsak } j \le m \text{ obstaja } i \le n, \text{ da je } \ell_i = k_j \text{ in } A_i = B_j}{\{\ \ell_1 : A_1; \ldots; \ell\_n : A\_n\ \} \le \{\ k_1 : B_1; \ldots; k\_m : B\_m\ \}}$$

Povedano z besedami, prvi tip zapisa je podtip drugega, če se vsako polje $k_j : B_j$ iz drugega zapisa pojavi v prvem. Tej vrsti podtipov pravimo **podtip zapisa po širini**, ker je podtip "širši" (ima več polj) kot njegov podtip.

Velja torej:

```
{ z : float; x : float; y : float } ≤ { x : float; y : float }
```

Vaja: ali obstaja tip zapisa, ki je podtip vseh ostalih tipov zapisov? Kaj pa tip zapisa, ki je nadtip vseh ostalih tipov zapisov?

Poznamo še eno **podtip zapisa v globino**. Spet poglejmo primer, pri čemer predpostavimo, da velja `int ≤ float`. Zapis

```
v = { x = 3; y = 5 }
```

ima tip `{x : int; y : int}`. Ali ga lahko uporabljamo, kot da bi imel tudi tip `{x : float; y : float}`? Da, saj lahko njegovi komponenti `v.x` in `v.y` uporabljamo, kot da bi imeli tip `float`, zahvaljujoč `int ≤ float`.

Pravilo, za podtipe zapisov v globino se glasi:

$$\frac{A_1 \le B_1 \qquad A_2 \le B_2 \qquad \cdots \qquad A\_n \le B\_n}{\{\ \ell_1 : A_1; \ldots; \ell\_n : A\_n\ \} \le \{\ \ell_1 : B_1; \ldots; \ell\_n : B\_n\ \}}$$

Obe pravili, za širino in globino, lahko združimo v eno kombinirano pravilo:

$$\frac{\text{za vsak } j \le m \text{ obstaja } i \le n, \text{ da je } \ell_i = k_j \text{ in } A_i \le B_j}{\{\ \ell_1 : A_1; \ldots; \ell\_n : A\_n\ \} \le \{\ k_1 : B_1; \ldots; k\_m : B\_m\ \}}$$

Zapisi s spremenljivimi polji

Katera pravila za podtipe zapisov pridejo v poštev, je odvisno od tega, kako lahko zapise uporabljamo. Denimo, če imamo zapise s *spremenljivimi* polji, se pravi, da lahko zapisu spremenjamo vrednosti polj, potem podtipi v globino niso več veljavni. Na primer, če imamo tipa

```
A = { mutable x : int; mutable y : int}
```

in

```
B = { mutable x : float; mutable x : float}
```

potem *ne* velja `A ≤ B`. Če bi to veljalo, bi lahko v zapis `v : A` vrednost polja `x` nastavili na `3.14`. Zapišimo tip `A` še v OCamlu:

```
type a = { mutable x : int; mutable y : int}
```

Takole naredimo vrednost in ji nastavimo atribut:

```
# let p = {x = 10; y = 20} ;;
val p : a = {x = 10; y = 20}
# p.x <- 42 ;;
- : unit = ()
# p ;;
- : a = {x = 42; y = 20}
```

Problem koherentnosti

Težave nastopijo, če lahko vrednosti tipa `A` pretvorimo v nadtip `B` na več načinov, se pravi, če imamo

```
A ≤ B
A ≤ C
B ≤ D
C ≤ D
```

Zaradi tranzitivnosti sledi `A ≤ D`, kar lahko izpeljemo na *dva* načina:

1. `A ≤ B ≤ D`
2. `A ≤ C ≤ D`

To pa lahko vpliva na implicitne pretvorbe. Če imamo `e : A`, ga lahko pretvorimo v `D` preko pretvorb `A → B → D` ali preko `A → C → D`. Kako vemo, da bomo obakrat dobili isto? Temu pravimo problem koherentnosti. Pojavi se vedno, ko imamo implicitna (ali samodejne) pretvorbe vrednosti iz enega tipa v drugega.

Podtipi na nivoju struktur

Strukture ali moduli so v resnici neke vrste zapisi na višjem nivoju. Zanje veljajo pravila za podtipe, kar preizkusimo na primerih `strukture.ml`.

Objekti

Objekti so podobni zapisom, le da imajo atribute in metode. Metode se lahko sklicujejo na atribute, kakor tudi na druge metode (z uporabo `this` ali `self`).

Torej so objekti neke vrste **rekurzivni zapisi**. Rekurzivni, ker se lahko objekt skliče sam nase.

Objekti v OCamlu

Spoznajmo objekte v OCamlu. Objekt zapišemo neposredno takole:

```
object (self)
   val attr1 = ...
   val attr2 = ...

   method m1 = ...
   method m2 = ...
end
```

Več primerov bomo obdelali na vajah, glej tudi datoteko `./primeri.ml`.