Računski učinki

Kaj so računski učinki

Pojem *računskega učinka* je zelo splošen in ga je težko natančno opredeliti. Računski učinki so vsi pojavi v programu, ki spremenjajo ali kako drugače delujejo z zunanjim okoljem programa. Pod pojmom "zunanje okolje" imamo v mislih operacijski sistem in strojno opremo, na kateri teče program.

Primeri računskih učinkov so I/O (pisanje in branje na datoteke), spreminjanje in branje pomnilnika, prekinitev izvajanja z izjemo, naključna in verjetnostna izbira, ter še mnogi drugi.

S stališča principov programskih jezikov se je smiselno vprašati, ali imajo vsi ti pojavi kaj skupnega. Jih lahko sistematično obravnavamo? Ali lahko v programski jezik vgradimo koncepte, ki nam omogočajo nadzorovano in fleksibilno delo z računskimi učinki?

Da bomo lahko odgovorili na ta vprašanja, najprej spoznajmo nekaj vrst računskih učinkov. Poskusimo ugotoviti, ali imajo kakšno skupno strukturo.

Primeri računskih učinkov

Izjeme

Izjeme pozna večina popularnih programskih jezikov. Izjema je računski učinek, ki prekine izvajanje programa in namesto rezultata vrne posebno vrednost, ki se imenuje *izjema*. Izjem je lahko več ("deljenje z nič", "neveljaven indeks", "datoteka ne obstaja" ipd.), pravzaprav jih je lahko za cel podatkovni tip `exception`.

Program, ki uporablja izjeme lahko

1. vrne rezultat, *ali*
2. sproži izjemo

Če del programa sproži izjemo, se izvajanje prekine takoj in nadaljni izračuni se *ne* izvajajo.

Vhod & izhod

Vhod/izhod (angl. input/output ali kar I/O) je računski učinek, kjer program kominicira z zunanjim okoljem tako, da bere podatke iz vhoda (tipkovnica, datoteka, komunikacijski kanal) in jih izpisuje na izhod (zaslon, datoteka, komunikacijski kanal).

Program, ki uporablja vhod in izhod

1. vrne rezultat, *in*

2. med izvajanjem *izpisuje* in *bere* znake
3. vrne rezultat

Stanje

Stanje je podatek, ki ga lahko program bere in spreminja. To je običajno pomnilniška lokacija, spremenljivka, ali element tabele. Program ima lahko kombinirano stanje iz velikega števila posameznih stanj (na primer, velika tabela ali pomnilnik), a lahko na tako stanje še vedno gledamo kot na enovito stanje, ki pa ima ogromno število možnih vrednosti.

Program, ki uporablja stanje:

1. se začne izvajati v *začetnem stanju*;
2. med izvajanjem *bere* in *nastavlja* stanje;
3. vrne rezultat *in*
4. se konča v *končnem stanju*.

Monade

Na prvi pogled učinki nimajo prav dosti skupnega, podrobnejša analiza pa pokaže, da imajo skupno strukturo, ki jo lahko izrazimo z matematičnim pojmom **monada**.

Najprej moramo ločiti programe, ki *ne* uporabljajo učinkov od tistih, ki jih uporabljajo. Pravimo, da je program, ki ne uporablja učinkov **čist** ali da je **vrednost** (angl. value). Program, ki uporablja učinke, imenujemo **učinkoven** (angl. effectful) ali **izračun** (angl. computation).

Na primer,

```
let x = 2 + 3 in
let y = x + 2 in
y * y - x
```

je čist, saj samo izračuna vrednost. Program

```
let x = 2 + 3 in
print "hello" ;
let y = x + 2 in
y * y - x
```

je izračun, ki uporablja učinek izhod (angl. output).

Veljata naslednji zelo splošni lastnosti:

1. Vsaka vrednost je tudi izračun.
2. Izračune lahko *setavljamo*.

Prva lastnost pravi, da lahko an vrednost gledamo kot na izračun, ki pač ni uporabil nobenega učinka.

Druga lastnost nam omogoča, da lahko izračun sestavimo iz bolj prepostih delov. Operacijo, ki sestavi skupaj dva izračuna, imenujemo `bind`, ker je neke vrste "vez" med izračuni. Če je `p` izračun, ki uporablja učinke in vrne rezultat, `q` pa je izračun, ki pričakuje rezultat od `p`je sestavljeni izračun

```
bind p q
```

V Haskellu namesto `bind` uporabimo operator `>>=`:

```
p >>= q
```

Mislimo si: "rezultat, ki ga izračna `p` pošljemo v izračun `q`, njune učinke pa sestavimo skupaj."

Primer: izjeme

Denimo, da lahko izračuni uporabljajo izjemo `Abort` in nobenega drugega učinka. Torej je rezultat računa bodsi izjema `Abort` bodisi rezultat `Result v`, kjer je `v` neka vrednost. Konstruktor `Result` je *pretvori* vrednost `v` v izračun.

Tedaj bi lahko `bind` definirali takole:

```
bind Abort q = Abort
bind (Result v) q = q v
```

Prva vrstica pove, da je treba izvajanje prekiniti, če prvi izračun sproži izjemo. Druga vrstica pove, da nadaljujemo z drugim izračunom, če prvi vrne rezultat. Pri tem rezultat podamo drugemu izračunu.

V Haskellu bi zapisali:

```
Abort >>= q  =  Abort
(Result v) >>= q  =  q v
```

Primer: stanje

Na izračun `p`, ki vrne rezultat tipa `τ` in hkrati uporablja stanje tipa `σ` lahko gledamo kot na funkcijo tipa

```
σ → τ × σ
```

Res, `p` sprejme začetno stanje in vrne rezultat ter končno stanje. Sam po sebi `p` ne naredi ničesar, ker je funkcija. Šele ko ga uporabimo na začetnem stanju, dobimo rezultat in končno stanje.

Če je `v` vrednost tipa `τ`, ki ne uporablja stanja, jo lahko pretvorimo v funkcijo, ki stanje sprejme in ga nespremenjenega vrne:

```
λ s . (v, s)
```

Kako pa sestavimo dva izračuna, ki uporabljata stanje?

```
p >>= q   =   λ s₁ . let (v, s₂) = p s₁ in (q v) s₂
```

Premislimo: sestavljeni izračun `p >>= q` je funkcija:

1. sprejme začetno stanje s_1
2. izvede izračun `p` v začetnem stanju s_1 in dobi rezultat `v` ter stanje s_2
3. z vrednostjo `v` nadaljuje izračun `q` v začetnem stanju s_2.

Definicija monade

Monada opisuje eno zvrst računskega učinka in sestoji iz naslednjih delov:

1. *Operacija M iz tipov v tipe.* Za vsak tip τ si mislimo, da je `M τ` tip *izačunov*, ki (morda) uporabljajo računski učinek in vrnejo rezultat tipa τ.

2. *Funkcija* $return : \tau \to M\ \tau$, ki pretvori vrednost tipa τ v izračun (ki ne uporablja nobenih učinkov).

3. *Funkcija* $bind : M\ \rho \to (\rho \to M\ \tau) \to M\ \tau$, ki sestavi izračun tipa `M ρ` in izračun tipa `M τ`, pri čemer le ta pričakuje kot vhodni podatek vrednost tipa ρ.

Veljati morajo naslednje enačbe:

1. `return` je leva enota za `bind`: `bind (return v) q ≡ q v`
2. `return` je desna enota za `bind`: `bind p return ≡ p`
3. `bind` je asociativna opreacija: `bind p (λ v . bind (q v) r) ≡ bind (bind p q) r`

V Haskellovi notaciji:

1. `(return v) >>= f ≡ f v`
2. `p >>= return ≡ p`
3. `p >>= (λ x . q x >>= q) ≡ (p >>= q) >>= r`

Premisli, ali te enačbe veljajo za izjeme in učinke, kot smo jih definirali zgoraj.

Izjeme kot monada

Poglejmo v datoteko ./exception.hs.

Blitz Haskell

Vsi programski jeziki nam omogočajo sestavljanje izračunov. V Pythonu in Javi enostavno pišemo ukaze enega za drugim in le ti se sestavijo. V SML uporabimo `let`, saj

```
let x = p in q
```

najprej požene izračun `p` (in s tem vse njegove učinke), rezultat shrani v vrednost `x`, nato pa požene še `q`, ki ima dostop do `x`.

V SML, Pythonu in Javi programer ne more nadzorovati monade za sestavljanje izračunov, ker je ta vgrajena v programski jezik. Če torej želimo spoznati programiranje z monadami, moramo uporabiti jezik, ki programerju omogoča, da nadzoruje in sam definira svoje monade – Haskell.

Na vajah boste spoznali osnove Haskella. Ker že poznamo SML bo prehod precej enostaven, seveda pa se bo treba spet navaditi na novo sintakso. SML in Haskell se razlikujeta v dveh delih:

1. SML je *neučakan* in Haskell *len* programski jezik. To pomeni, da SML vedno izračuna vsako vrednost in izračun takoj, Haskell pa šele, ko jo potrebuje.

2. SML ima eno vgrajeno monado za vse učinke, ki jih podpira (reference, izjeme, I/O, kontinuacije). Haskell je *čist*, programer pa samo definira svojo monado.

Kdor je neučakan, lahko sam bere učbenik o Haskellu in Haskell preizkusi kar v brskalniku.

Računski učinki kot monade v Haskellu

Monado v Haskellu definiramo točno tako, kot smo jo definirali zgoraj: podamo operacijo `m`, ki slika tipe v tipe ter funkciji `return` in `>>=`.

Haskell ima posebno notacijo, ki omogoča programerju bolj prijazen zapis. Namesto

```
p₁ >>= (λ x . p₂ >>= (λ y . p₃ >>= ...)
```

pišemo

```
do x <- p₁
   y <- p₂
   ...
```

Namesto

```
do x <- p₁
   _ <- p₂
   ...
```

pišemo

```
do x <- p₁
   p₂
   ...
```

V Haskellu moramo pravilno zamikati kodo, podobno kot v Pythonu.

Poglejmo si, kako bi standardne učinke implementirali v Haskellu.

Še nekateri drugi učinki

Naslednjič bomo spoznali dosti bolj zanimive učinke, ki jih lahko uporabimo za učinkovito programiranje.