UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO

Finančna matematika – 1. stopnja

Ana Golob

Prestrezniki in algebrajski učinki na primerih iz finančne matematike

Delo diplomskega seminarja

Mentor: prof. dr. Andrej Bauer

Ljubljana, 2018

Kazalo

Prestrezniki in algebrajski učinki na primerih iz finančne matematike

Povzetek

Diplomsko delo na preprost način predstavlja idejo in uporabo prestreznikov in algebrajskih učinkov. S primeri iz področja finančne matematike je razloženo njihovo delovanje in sintaksa (programski jezik Eff).

Opisana je logika nedeterminističnega programiranja pri katerem z uporabo prestreznikov in splošnim/abstraktnim načinom pisanja kode na identični kodi izvajamo več različnih izračunov.

Kot alternativo prestreznikom pri nedeterminističnem programiranju so opisani tudi delovanje in uporaba konceptov modul, funktor in monada. Narejena je primerjava med zgornjimi koncepti in prestrezniki s poudarki na prednostih, ki jih slednji prinašajo.

Handlers and Algebraic Effects on examples from Financial Mathematics

Abstract

This final thesis strives to present a general idea and use of handlers and algebraic effects. Their logic and syntax (in Eff programming language) is presented and explained on numerous examples from the field of Financial mathematics.

The logic and principles of nondeterministic programming are also presented, with the help of which (by using handlers) we can write code very generally, meaning it is possible to use the same code to acquire different computations.

As an alternative to handlers at nondeterministic programming, concepts like modules, functors and monades are also presented. A comparison between those concepts and the use of handlers is made with emphasis on the advantages of handlers.

Math. Subj. Class. (2010): 68-04, 68Qxx

Ključne besede: algebrajski učinki, prestrezniki, funkcijsko programiranje, finančna matematika

Keywords: algebraic effect, handler, functional programming, financial mathematics

1 Uvod

Prestrezniki za algebrajske učinke so nov koncept v programiranju, ki odpira nove, zanimive, elegantne in dinamične načine programiranja. Omogočajo nam pisanje nedeterminističnih programov, s katerimi lahko na identični kodi kasneje izvajamo različne izračune. Na primer tako, da na enaki kodi z različnimi prestrezniki računamo en sam možen izračun, optimalen izračun, množico vseh možnih rešitev ali pa npr. verjetnostno porazdelitev vseh izračunov.

Diplomska naloga želi na preprost način predstaviti idejo prestreznikov in učinkov, ter preko primerov pokazati njihovo delovanje in sintakso. Predstavljeni so razmisleki, ki so potrebni za razumevanje logike nedeterminističnega programiranja s prestrezniki in uporabnost takega načina programiranja na primerih, ki jih večinoma črpamo iz verjetnosti in finančne matematike.

V začetnih poglavjih je (brez zapletenega teoretičnega ozadja v algebri) na idejni ravni predstavljeno, kaj so algebrajski učinki in kaj prestrezniki. Predstavljen je programski jezik Eff, v katerem je napisan večji del kode, ki se nahaja v tem diplomskem delu. V tretjem poglavju na primerih iz finančne matematike predstavljamo sintakso in delovanje računskih učinkov in prestreznikov. V četrtem poglavju kot alternativo prestreznikom, predstavljamo koncepte modul, funktor in monada. To poglavje se zaključi s primerjavo in predstavitvijo prednosti, ki jih prestrezniki prinašajo. V šestem poglavjih na razširjenem primeru simulacije finančnega trga predstavljamo nedeterministični način programiranja. Nato v sedmem poglavju kot drugi razširjeni primer dodajamo implementacijo Brownovega gibanja, pri čemer bomo spoznali kako deluje prestreznik za spreminjanje stanj.

2 Učinki in ideja prestreznikov

Računalniški programi običajno niso čisti v smislu, da ne delujejo tako kot matematične funkcije. Pogosto izvajajo določene izračune, ob tem pa vsebujejo še množico operacij, ki ustvarjajo interakcije z zunanjim svetom. Pravimo, da so v njih prisotni različni učinki.

Učinki so torej vse kar nastane kot interakcija z zunanjim svetom (ukazi, ki nekaj izpisujejo, preberejo), ukazi, ki delajo spremembe v pomnilniku, nadalje so učinki tudi generatorji naključnih števil (kadar niso psevdo-random generatorji), izjeme (npr. division by zero) itd.

Gordon Plotkin in John Power sta ugotovila, da številne računske učinke lahko popišemo z algebrajskimi teorijami [1]. Ko se program izvaja namreč nečisto vedenje nastaja zaradi niza operacij, kot so `get` & `set`, `read` & `print`, `input` & `output` in `raise` za izjeme [9]. Te operacije pa lahko popišemo matematično kot algebraične operacije, od koder sledi tudi ime algebrajski učinki.

Izračune, ki jih koda izvaja torej lahko delimo na čiste izračune in učinke. Za prve velja, da ne glede na to, kdaj ali kolikokrat jih izvedemo vedno dobimo enak rezultat. To so običajno funkcije, ki nekaj računajo in nato vrnejo rezultat,[1] medtem

[1] Čiste operacije lahko popišemo z lambda računom.

ko učinki izvajajo določeno operacijo [1].

Prestrezniki so tesno povezani z učinki in jih lahko v grobem razumemo kot nekakšno programersko „orodje“ za delo z učinki. S prestrezniki predpisujemo delovanje učinka, s čimer lahko povsem preusmerimo njegovo delovanje in na ta način spreminjamo delovanje celotne kode. Tudi prestrezniki imajo svojo podlago v algebrajskih teorijah [1], pravzaprav je zanimivo (da so, na način na katerega jih bomo spoznavali tukaj), najprej obstajali kot matematična teorija in bili šele nato kot takšni podlaga za ustvarjanje programskega jezika Eff (opisan v nadaljevanju).

V programskih jezikih se prestrezniki že uporabljajo za:

1. *Izjeme* (ang. exeption hendlers): Te vrste prestrezniki so se pojavili najprej. Izvedejo se, ko pride do izjeme.

2. *Dogodke* (ang. event hendlers): Za sprejemanje dogodkov in signalov iz okolja. Lahko si jih predstavljamo kot del kode, ki se izvede, ko pride do nekega dogodka (npr. uporabnik nekaj klikne z miško).

3. *Pomnilnik* (ang. memory hendler): izvaja določene naloge v pomnilniku.

V večini programskih jezikov sami zaenkrat še ne moremo definirati novih prestreznikov[2] (tako kot npr. lahko napišemo funkcijo). Zaradi tega bomo uporabljali programski jezik Eff, ki je bil napisan posebej za testiranje prestreznikov in algebrajskih učinkov in nam omogoča, da napišemo in testiramo prestreznike za poljubne računske učinke.

3 Programski jezik Eff

Eff je funkcijski programski jezik, ki temelji na uporabi prestreznikov za algebrajske učinke. To pomeni, da se v njem prestrezniki ne uporabljajo le za izjeme, temveč za poljubne računske učinke. Omogoča nam, da lahko sami definiramo ne le svoje prestreznike, temveč tudi predpisujemo nove računske učinke [2].

Eff statično preverja tipe, kar pomeni, da se ti preverjajo že pred samim zaganjanjem programa. Poleg tega vsebuje večino osnovnih programerskih orodij (aritmetične operacije, funkcije, rekurzivne funkcije, ...) na način, ki je značilen za funkcijske programske jezike. Sintaksa Eff-a je podobna sintaksi funkcijskega programskega jezika OCaml, z izjemo prestreznikov za algebrajske učinke, ki se v OCamlu (na takšen način) ne uporabljajo [2].

V delu predpostavljamo, da je bralec seznanjen z osnovno sintakso OCaml-a [6], podrobneje pa bomo prestavili konstrukte, ki so za Eff specifični.

Poleg standardnih tipov imamo v Eff-u še tip prestreznik (handler). Prav tako imamo možnost definiranja operacij, ki bodo znotraj kode čakale, da jim prestrezniki povedo, kako naj se izvedejo, nato pa se bo z rezultatom, ki ga bodo vrnile nadaljevalo računanje v preostanku kode.

Nekatere „nečiste“ operacije, kot so generatorji naključnih števil ter operaciji `print` in `read`, so v Eff že vgrajene. Kadar izvedemo takšne operacije, pred njimi pišemo `perform`. Spodaj navajamo prej omenjene primere:

[2]To velja z izjemo multicore verzije OCamla, kjer se nekatere vrste prestreznikov že uporabljajo.

```
perform (Read ()) ;;
perform (Print "sporocilo") ;;
perform (Random_float 1.) ;;
```

Računski učinki so v Eff implementirani kot operacije, ki imajo podlago v algebraičnih teorijah [2, 9].

Preostale posebnosti učinkov in prestreznikov bomo na primerih predstavili v nadaljevanju.

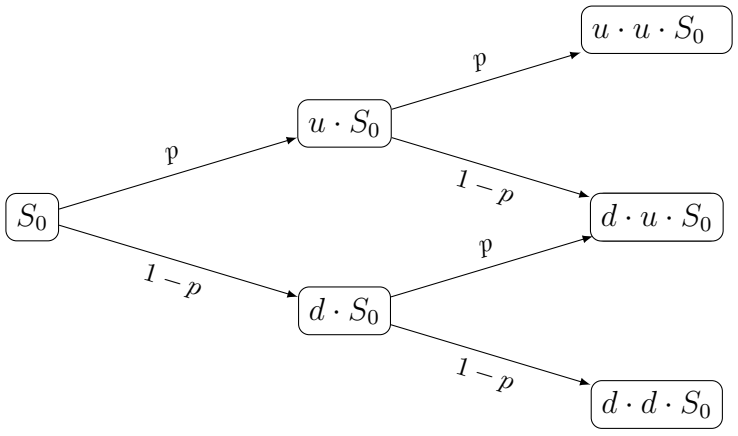
4 Prikaz uporabe računskih učinkov in prestreznikov

Uporabo prestreznikov bomo predstavili na primerih iz področja finančne matematike, zato predpostavljamo, da je bralec seznanjen z osnovnimi vsebinami na področju verjetnosti in pozna osnove finančne matematike [3]. Na enostavnih primerih bosta predstavljena sintaksa in delovanje, unikatne prednosti in zmožnosti prestreznikov pa bodo na kompleksnejših primerih predstavljene v zadnjih poglavjih.

V primerih bomo prestreznike večinoma uporabljali za namene nedeterminističnega programiranja. Od že implementiranih učinkov se bodo večinoma pojavljali generatorji naključnih števil, vpeljali pa bomo tudi lastne računske učinke, ki jih bomo definirali na mestih, kjer bomo želeli nedeterministično vedenje. Delovanje računskih učinkov bodo določali ustrezni prestrezniki. Pri tem velja poudariti, da za en sam učinek običajno predpišemo več različnih prestreznikov.

4.1 Preprosta simulacija trga in model Cox-Ross-Rubinstein

Poglejmo si Cox-Ross-Rubinstein-ov model trga za n obdobij (tudi binomski model). V vsakem diskretnem času $t = 0, 1, 2, \ldots, n$ se ekonomija lahko razvije v enega izmed dveh možnih stanj. Naj bo eno izmed stanj dobro, z obrestno mero u, drugo stanje pa slabo, z obrestno mero d. Iz povedanega očitno sledi neenakost $u > d$. Ceno vrednostnega papirja v poljubnem diskretnem času t označujemo z S_t. Če vzamemo $n = 2$, potem cena vrednostnega papirja sledi binomskemu procesu, ki je predstavljen na spodnjem drevesu:



V vsakem času imamo možnost izbirati med dvema različnima razvojema, kar implicira nedeterministično vedenje. Če preidemo na teorijo algebrajskih učinkov za nedeterminizem, potem v osnovi velja, da jih lahko predstavimo z binarno operacijo `izberi` (decide) [9]. To je operacija, ki sprejme vrednost tipa `unit` in nedeterministično vrne logično vrednost. Za naš primer je to predstavljeno s spodnjo kodo:

```
let obresti  =
        let b = perform (Izberi ()) in
        if b then u else d
;;
```

Zgornja funkcija določi obrestno mero tveganega vrednostnega papirja v vsakem naslednjem stanju. Tvegani vrednostni papir bomo v našem modelu imenovali delnica. V skladu z zgoraj opisanim binomskim modelom, lahko izbiramo med obrestnima merama u za dobro in d za slabo stanje ekonomije. Nedeterminizem smo zajeli v operaciji `Izberi`. Tipično bi v binomskem modelu ta funkcija delovala verjetnostno. Z neko vnaprej dano verjetnostjo p, bi izbrala vrednost `true` in z verjetnostjo $1 - p$ vrednost `false`. V našem primeru bomo dovolili tudi drugačne definicije.

Prehod trga iz stanja v diskretnem času $t - 1$, v novo stanje v času t, določajo obrestne mere vrednostnih papirjev. Zato bomo obresti definirali kot spremenljivo operacijo `Obresti`. Spodaj je predstavljena sintaksa za takšno definicijo:

```
type vrednosti_delnic = float list
type obresti = float list
effect Obresti:  vrednosti_delnic -> obresti;;
```

Najprej smo za večjo berljivost kode definirali tipa `vrednosti_delnic` in `obresti`. Oba sta določena s seznamom realnih števil. Pri prvem tipu vsako realno število predstavlja vrednost posamezne delnice, pri drugem pa obrestno mero. Število različnih tveganih vrednostnih papirjev na trgu je določeno z dolžino seznama.

V zadnji vrstici zgornje kode nam beseda `effect` pove, da definiramo operacijo, katere delovanje bo predpisoval prestreznik. Za dvopičjem je, kot je značilno za funkcijske programske jezike, naveden tip te operacije. Definirali smo torej novo operacijo, ki vzame stanje vrednosti delnic in nam nedeterministično vrne obrestne mere s katerimi se obrestujejo delnice na prehodu v novo stanje. Definirani operacijo lahko sedaj uporabimo v kodi in z njegovo pomočjo simuliramo preprost primer finančnega trga z vhodnima parametroma:

- `s0` – vektor začetnih vrednosti delnic v času 0
- `n` – število obdobij

Naša funkcija ne bo popisovala izključno modela Cox-Ross-Rubinstein. Delovanje trga želimo definirati čim bolj splošno, da bomo kasneje na njem lahko izvajali različne izračune. Seveda bo osnovno delovanje enako kot pri binomskem modelu, le da bo omogočalo tudi realizacije drugih podobnih modelov. Spodaj podajamo predpis funkcije:

```
let obrestuj obresti delnice =
  map2 ( *. ) obresti delnice
;;
```

```
let simulacija_trga s0 n =
 (*En korak simulacije*)
  let rec korak k s =
    if k >= n
      then s
    else
      korak (k + 1)  (obrestuj (perform (Obresti s)) s) in
  korak 0 s0
;;
```

Simulirani trg vrednostnih papirjev se v vsakem izmed n obdobij obrestuje. V naši kodi smo že lahko uporabljali zgoraj definirano operacijo `Obresti`, čeprav še ne vemo kako deluje. Statični tipi v Effu nam zagotavljajo, da se bodo tipi kasneje, ko bomo delovanje operacije `Obresti` tudi konkretno določili, ujemali.

Kakšne bodo obrestne mere delnic bo določal prestreznik, ki ga bomo definirali v nadaljevanju. Prestreznik je tisti del kode, ki pove, kako naj se operacija izvede.

Zaradi enostavnosti v začetku predpostavimo, da imamo en sam vrednostni papir torej, da so vrednosti delnic in obrestne mere predstavljene s seznami dolžine 1. Spodaj definiramo prvi primer prestreznika z imenom `deterministicne_obresti`. Deluje tako, da na vsakem prehodu v novo stanje deterministično izbere višjo izmed obrestnih mer u in z njo obrestuje vrednost delnice. Spodaj podajamo kodo:

```
let u = [1.05]
let d = [0.9]

let  deterministicne_obresti = handler
        | v -> v
        | effect (Obresti s) k ->
                continue k  u
;;
```

Izraz `handler` v zgornji kodi pove, da smo definirali prestreznik. S tem implicira tip definirane vrednosti in hkrati pove, da ta sprejme parameter[3] (tipa učinek).

Prvi predpis `v -> v` pove da prestreznik, kadar dobi „čisto“ vrednost (ki ni učinek) vrne kar enako vrednost. Ta predpis je trivialen in ga ni potrebno pisati, zato ga bomo v nadaljevanju izpuščali. Pisali ga bomo le v primerih, ko bomo želeli, da prestreznik spremeni tudi „čiste“ vrednosti.

V drugem primeru predpišemo, da naj prestreznik, kadar v kodi zasledi operacijo `Obresti`, njegovo delovanje izvede tako, da vrne obrestno mero u in na ta način izvrši operacijo.

Parameter `k` v zgornjem prestrezniku stoji za kontinuacijo. Zaradi njega se bo izvedel tudi preostanek kode za prestreznikom in bo prestreznik zajel tudi vsak nadaljnji operaciji `Obresti` v naši kodi. Sledi da bo v zgornjem primeru v vsakem obdobju prišlo do obrestovanja z obrestno mero u. Več o tem kako deluje parameter

[3]Bralec, ki je domač s programiranjem v OCamlu, lahko opazi podobnost med definiranjem funkcije:
let funkcija = function
| x -> ...
Kjer zapis function implicira, da funkcija sprejme nek parameter.

`k` in kakšen je njegov pomen bomo povedali v nadaljevanju. Najprej si poglejmo kako prestreznik uporabimo na zgoraj definirani simulaciji trga:

```
with deterministicne_obresti  handle
        simulacija_trga [20.]  5
;;
```

Izračunali smo, koliko po petih obdobjih znaša vrednost vhodnega vrednostnega papirja, ki je imel v času 0 ceno 20. Pri tem je prestreznik `deterministicne_obresti`, s katerim smo prestregli celotno funkcijo, določil obrestne mere za obrestovanje v vseh obdobjih. Kadar se je znotraj simulacije trga izvajala npr. operacija `k + 1` in ostale, ki niso povezane z delovanjem operacije `Obresti`, prestreznik nanje ni imel vpliva. Določa le vsako izmed petih obrestovanj v simulaciji.

Če bi v definiciji prestreznika izpustili parameter za kontinuacijo `k`, bi se vrednostni papir kljub petim obdobjem obrestoval samo enkrat, potem pa bi se nadaljevanje prekinilo. Ob klicu zgornje kode bi v tem primeru dobili vrnjeno vrednost $21 = 20 \cdot 1,05$. V našem primeru je torej kontinuacija iz vsebinskih razlogov zaželena, vendar pa to v splošnem ne drži vedno.

Izjeme, kot eden izmed najpomembnejših učinkov za katere se prestrezniki uporabljajo, po drugi strani nimajo kontinuacije [9]. Ko se v kodi sproži izjema, se izvajanje kode prekine,[4] kar je zaradi napake oz. razloga, ki je izjemo sprožil, zaželeno.

Funkcijo `simulacija_trga` lahko sedaj preusmerimo z novimi prestrezniki, ki bodo obrestovanje predpisali na drugačen način in s tem realizirali različne modele trga. Poglejmo si še primer prestreznika za obrestovanje, kot ga poznamo iz zgoraj opisanega binomskega modela:

```
let p = 0.6 (*Verjetnost dobrega razvoja ekonomije*)

let binomske_obresti = handler
  | effect (Obresti s) k->
  let izberi = perform (Random_float 1.) < p in
  continue k (if izberi then  u else  d)
;;
```

Lokalna spremenljivka `izberi` v zgornji funkciji z dano verjetnostjo p pokaže `true` in z verjetnostjo $1 - p$ logično vrednost `false`. V pogojnem stavku se nato kasneje na podlagi vrednosti funkcije `izberi` odločimo, ali se trg obrestuje z obrestno mero u ali d. Če simulacijo trga prestrežemo s tem prestreznikom, dobimo eno realizacijo Cox-Ross-Rubinstein modela. Za vhodno začetno ceno delnice 20 in za 5 obdobij, izračun izvedemo z naslednjo kodo:

```
with binomske_obresti handle
        simulacija_trga [20.]  5
;;
```

[4]Primer izjeme je deljenje z nič. Ko pride do deljenja z 0, se sproži izjema, izvajanje kode pa se prekine, saj rezultata ob deljenju z nič ne moremo dobiti, brez rezultata pa ne moramo nadaljevati računanja.

4.2 Prestreznik za verjetnostno porazdelitev

V tem podpoglavju bomo z novim prestreznikom spremenili simulacijo trga, ki smo jo definirali zgoraj. Naš cilj je definirati prestreznik, ki ne bo ustrezal zgolj eni realizaciji binomskega modela, temveč bo zajel vsa možna končna stanja ekonomije v tem modelu. Ker je odločitev o tem, v katero stanje bomo prešli, sprejeta na podlagi verjetnosti, ne bomo želeli zgolj množico stanj, temveč njihovo porazdelitev. Tip porazdelitve definiramo kot

```
type porazdelitev = (float * float) list
```

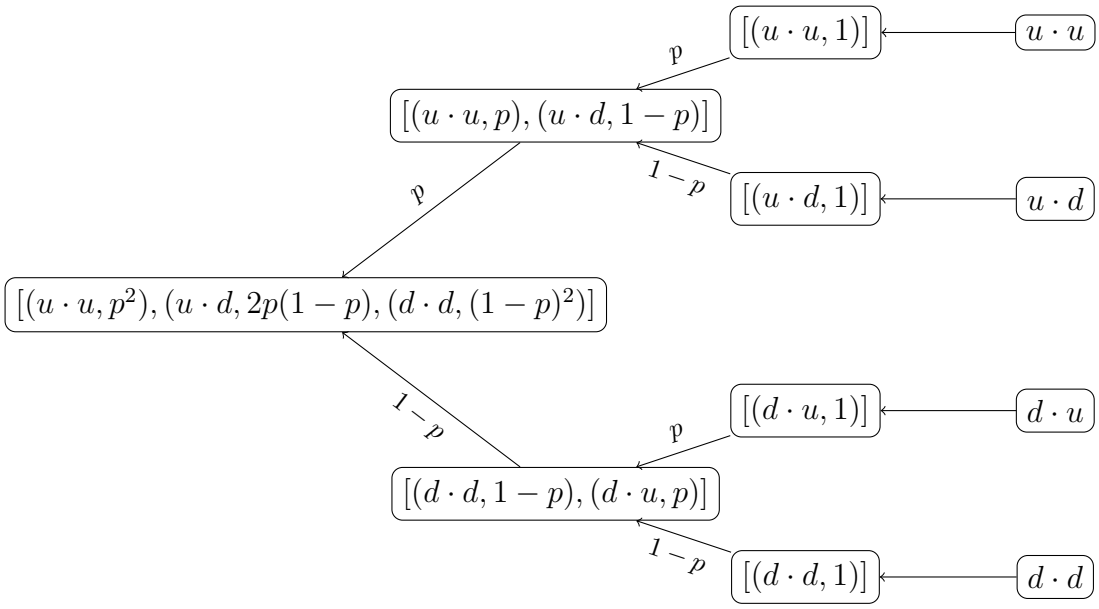
torej seznam parov, v katerem bo prvi element para predstavljal vrednost delnice, drugi pa njegovo verjetnost.

V prestrezniku za porazdelitev želimo ujeti informacije za vse možne razvoje ekonomije skupaj z njihovo verjetnostjo. Spodaj je predstavljena definicija prestreznika za računanje porazdelitve, v kateri je potrebno na smiseln način definirati še funkcijo `combine`:

```
let u = [1.05]
let d = [0.9]

let porazdelitev = handler
  | v -> [(v, 1.0)]
  | effect (Obresti s) k ->
  combine p (continue k u)
                      (continue k d)
;;
```

V drugem delu predpišemo, da naj prestreznik operacijo `Obresti` izvede tako, da obrestuje na oba možna načina. V kodi zgoraj opazimo predpis `continue k u` kot tudi predpis `continue k d`. Lahko si predstavljamo, da se na vsakem prehodu iz enega v drugo obdobje računanje razveji v dve veji, tako kot v binomskem drevesu. Dvojna kontinuacija pomeni, da se bo obrestovanje za naslednja obdobja nadaljevalo na obeh vejah. Ko pridemo v zadnje obdobje na koncu vsake veje dobimo vrednost. Vrednosti nato želimo spremeniti v porazdelitve, te pa smiselno združiti med seboj. Potrebujemo torej še predpis, ki bo drevo smiselno združil v eno samo porazdelitev. Želeni proces je za obdobje $n = 2$ in binomski model trga predstavljen s spodnjim drevesom:



Prvi vzorec v prestrezniku, ki „čiste“ vrednosti `v` priredi porazdelitev $[(v, 1.0)]$, naredi prvi potreben korak iz zgornjega drevesa in končne vrednosti v listih drevesa pretvori v porazdelitve. Preostane nam samo še definicija funkcije `combine` [8], ki bo ustrezno združila dve porazdelitvi, tako kot na zgornji sliki. Poglejmo si predpis ustrezne funkcije:

```
let combine p dist1 dist2 =
  let scale p dist = map (fun (x, q) -> (x, p *. q)) dist
  in
  let rec add (x, p) = function
    | [] -> [(x, p)]
    | (y, q) :: dist ->
        if x = y then
        (x, p +. q) :: dist else (y, q) :: add (x, p) dist
    in
    let dist1 = scale p dist1 in
    let dist2 = scale (1.0 -. p) dist2 in
  fold_right add dist1 dist2
;;
```

S tako definirano funkcijo `combine` zgornji prestreznik smiselno deluje in ga lahko uporabimo na simulaciji trga. Kot rezultat dobimo porazdelitev vrednostnega papirja v končnem obdobju:

```
with porazdelitev handle
    simulacija_trga [20.] 5;;
```

5 Moduli, funktorji in monade

Alternativa prestreznikom, ki se za podoben način programiranja v funkcijskih programskih jezikih že uporablja, so monade in moduli ter njihova razširitev v funktorje. Uporaba teh je v primerjavi s prestrezniki precej bolj zapletena. Na kratko jih bomo predstavili v nadaljevanju poglavja.

Pogledali si bomo, kako lahko v programskem jeziku OCaml brez uporabe prestreznikov napišemo kodo na nedeterminističen način. Novemu pristopu bo s programiranjem v Eff-u (predstavljeno v prejšnjih poglavjih) skupno to, da bomo kodo napisali abstraktno oz. splošno (simulacija trga bo ostala nespremenjena), na ta način pa bo delovala za več različnih izračunov.

V prvih dveh podpoglavjih si bomo pogledali, kaj so in kako delujejo funktorji in monade, nato pa bomo z njihovo pomočjo izvedli simulacijo binomskega modela. Pri tem si bomo pomagali z identičnimi primeri kot v prejšnjem poglavju, ko smo predstavljali prestreznike v Eff-u. Zaradi enostavnosti bomo ohranjali predpostavko, da imamo na trgu le eno delnico, kljub temu pa bomo računali s seznamom dolžine ena, ki nam bo kasneje omogočil lažjo razširitev v trg z več vrednostnimi papirji.

5.1 Moduli

Če v OCamlu želimo daljšo kodo razdeliti na manjše dele (predvsem zapisane v različnih datotekah), jo moramo zapisati v modulih. O modulih lahko razmišljamo kot o zbirki definicij, ki so shranjene v manjših povezanih enotah. Tudi že napisane knjižnice v OCamlu uvozimo v datoteko kot module.

Definirali bomo tip modula z imenom `TRG`, ki bo vseboval eno samo funkcijo. To bo funkcija `obresti`, ki bo določala obrestne mere in s tem evolucijo trga v vsako naslednje obdobje. Najprej imamo torej definicijo signature oz. tipa modula, kot je prikazano spodaj:

```
module type TRG =
sig
  val obresti : vrednosti_delnic -> obresti
end
```

Tip modula `TRG` opisuje modul, ki vsebuje funkcijo `obresti` predpisanega tipa. V modulu tega tipa se lahko nahajajo tudi funkcije, ki jih v signaturi nismo zapisali, vendar ob prevajanju modula, ko se izpiše njegov tip, te funkcije ne bodo vidne. Na ta način bi lahko zunanjemu uporabniku onemogočili uporabo nekaterih funkcij, ki jih modul vsebuje in jih v signaturi ne navedemo, zaradi česar ostajajo skrite. Kasneje lahko zapišemo več različnih modulov istega tipa. Primerov implementacije tipa modulov `TRG` je modul:

```
module Deterministicne_obresti : TRG =
struct
  let u = [1.05]
  let d = [0.9]
  let obresti s = u
;;
end
```

V njem, kot je bilo predpisano, najdemo funkcijo `obresti`. V tem primeru deluje tako, da vsakega od vrednostnih papirjev obrestuje z višjo izmed obrestnih mer u. Da gre za modul tipa `TRG`, povemo v prvi vrstici kode za dvopičjem.

Naredimo še en primer modula istega tipa. V njem se obrestne mere izbirajo slučajno, na način, ki je značilen za binomski model trga. Spodaj navajamo opisani modul:

```
module Binomske_obresti : TRG =
struct
  let u = [1.05]
  let d = [0.9]
  let p = 0.6 (*Verjetnost dobrega razvoja ekonomije*)

  let obresti s =
    let izberi = (Random.float 1.) < p in
      if izberi then u else d
;;
end
```

Z večimi različnimi moduli torej lahko popišemo različno vedenje nedeterministične operacije `obresti`. Opazimo, da se v modulu nahajajo predpisi, ki smo jih v prejšnjem poglavju zapisali v prestreznikih, medtem ko bi signaturo modula lahko primerjali z definicijo nove operacije.

5.2 Funktorji

Funktor je modul, ki je parametriziran z drugimi moduli podobo kot je funkcija vrednost, ki je parametrizirana z drugimi vrednostmi (argumenti funkcije) [6]. V njem se bo nahajal del kode, ki bo ostajal nespremenjen. Ker sledimo primeru iz prejšnjega poglavja, bo funktor vseboval funkcijo, ki simulira trg. Spremenljivo operacijo `obresti` pa bomo definirali izven funktorja, v vhodnem modulu. V spodnji kodi vidimo predpis funktorja, ki kot parameter sprejme modul tipa `TRG`:

```
module Simulacija_trga (T : TRG) =
struct
  open T

  let simulacija_trga s0 n=
    let rec korak k s =
      if k >=  n then s
      else
        let s' =
        List.map2 (fun x y -> x *. y) (obresti s) s in
        korak (k + 1) s'
    in
    korak 0 s0
  ;;

end
```

Statični tipi v OCamlu nam omogočajo, da lahko v simulaciji sprožimo operacijo `obresti`, še preden je ta definirana. Program pri preverjanju tipov namreč ve, da bo funktor predpis za to funkcijo dobil v modulu, ki ga bo sprejel kot parameter, saj modul tipa `TRG` po definiciji vsebuje funkcijo `obresti`, katere tip je prav tako znan v naprej in se bo torej preverjeno ujemal z vsemi nadaljnjimi operacijami v funktorju. Sedaj lahko vidimo uporabnost predhodne definicije tipa modula. V njem moramo predpisati vse funkcije, ki jih iz modula želimo prenesti v funktor in jih tam uporabljati.

Kljub temu, da simulacija trga v funktorju ves čas ostaja nespremenjena, lahko z njo delamo različne izračune, odvisno od tega, kako je v vhodnem modulu definirana funkcija `obresti`. Spodaj dodajamo še kodo s katero na zgornjem primeru naredimo izračuna za oba možna predpisa obrestnih mer:

```
module Primer1 =
       Simulacija_trga (Deterministicne_obresti);;
module Primer2 = Simulacija_trga (Binomske_obresti);;

Primer1.simulacija_trga [20.] 5;;
Primer2.simulacija_trga [20.] 5;;
```

5.3 Monade

Pri programiranju z moduli in funktorji smo še vedno zelo omejeni, saj funkcija `obresti`, ki se nahaja v modulu, lahko vrača le seznam obrestnih mer, ne more pa vračati npr. njihove porazdelitve. Če bi v različnih modulih definirali funkcijo `obresti` z različnim tipom, bi namreč prekršili sledeči predpis iz tipa modula `obresti : vrednosti_delnic -> obresti`. Če pa bi v tipu modula funkcijo `obresti` napisali bolj abstraktno (`obresti : vrednosti_delnic -> 'a`), tako da bi dovoljevala različne realizacije tipov, bi imeli težave v funktorju. Tam namreč nadaljujemo računanje z obrestnimi merami, ki jih vrne funkcija iz modula. Pri teh računih so uporabljene operacije za sezname realnih števil, ki za drugačne tipe ne bi delovale.

V nadaljevanju bi kodo želeli razširiti tako, da bo funkcija `obresti` lahko vračala tako porazdelitev obrestnih mer, kot tudi eno samo realizacijo, odvisno od tega kaj bomo želeli računati.

V ta namen potrebujemo monade. Monade v OCaml niso neposredno vgrajene, pozna pa jih funkcijski programski jezik Haskell. Napisali bomo modul, ki bo opravljal enako vlogo, kot jo v Haskellu opravljajo monade. Monade nam omogočajo, da za vrednosti nekaterih posebnih podatkovnih tipov, kot so podatkovni tipi s kontekstom (z dodano vrednostjo) uporabljamo funkcije za normalne vrednosti in vseeno ohranjamo pomen kontekstov [5]. Kaj natančneje to pomeni in kakšna je sintaksa za definiranje monade, bomo preko primerov spoznali v nadaljevanju.

Monada, ki jo bomo napisali za porazdelitev, je v začetku nekoliko zahtevnejša za razumevanje, zato si koncept monad najprej poglejmo na enostavnejšem primeru.

Vrednosti tipa Maybe

Preprost primer monad v Haskell-u so vrednosti tipa `Maybe a` [5]. Gre za vrednosti, ki predstavljajo vrednosti tipa `a`, z možnostjo, da pride do napake in ne dobimo vrednosti tipa `a`. Zato ima tip `Maybe` enega izmed dveh možnih predpisov, kot je prikazano v spodnji kodi:

```
type Maybe a = Just a | Nothing
```

V tem primeru `Nothing` stoji za odsotnost vrednosti tipa `a`. Če bi kot rezultat izračuna dobili `Nothing` bi pomenilo, da izračun ni uspel. Predpona `Just` pa pomeni, da vrednost tipa `a` imamo. Če bi s pomočjo tipa `Maybe` definirali funkcijo deljenje, bi to izgledalo nekako takole:

```
let deli x y =
   match y with
        | 0. -> Nothing
        | y -> Just (x /. y)
;;
```

V primeru ko pride do deljenja z nič, bi dobili vrednost `Nothing`, v preostalih primerih pa bi se deljenje izvedlo in bi dobili vrnjen rezultat izračuna s predpono `Just`.

Ker monade v programskem jeziku OCaml niso implementirane jih definiramo preko modula. Najprej definiramo njegov tip, v katerem so naštete vse vrednosti, ki jih monada mora vsebovati:

```
module type MONADA =
sig
  type m 'a
  val return : 'a -> m 'a
  val (>>=) : m 'a -> ('a -> m 'b) -> m 'b
end
```

Tukaj `'a` predstavlja parameter tip, to pomeni, da stoji za poljuben tip, ki pa mora na vseh mestih kjer `'a` nastopa, ostajati nespremenjen. Kot vidimo, mora monada vsebovati tri stvari, poglejmo si, kako vsako izmed njih definiramo na primeru monade `Maybe`.

Prva stvar, ki jo mora modul tipa `MONADA` vsebovati, je konstruktor `m`. To je operacija, ki poljubni tip `'a` pretvori v novi tip `'a m`. Primer konstruktorja je `Maybe`.

Kot smo že povedali, monada potrebuje vrednost podatkovnega tipa v kontekstu oz. z neko dodano vrednostjo, kar predstavlja črka `m` pred parametrom tipa `'a`. Za naš primer je ta dodana vrednost tip `Maybe`, zato bomo v modulu za prvi predpis napisali:

```
type m 'a  = Maybe 'a
```

Pri definiciji funkcije `return` se sprašujemo, kako vrednost tipa `'a` smiselno pretvoriti v vrednost tipa `m 'a` torej `'a` s kontekstom. V našem primeru iz pomena tipa `Maybe` sledi naslednji predpis:

```
let  return x = Just x
```

Če imamo vrednost, potem ji dodamo predpono `Just`, ki stoji ravno za prisotnost vrednosti tipa `'a`.

Tretja stvar, ki jo še moramo definirati je, kako deluje operacija `(>>=)`, ki ji pravimo `bind`. V tem delu se sprašujemo, kako ravnati, če imamo vrednost tipa `a` s kontekstom `'a t` in jo želimo smiselno vstaviti v funkcijo, ki sprejme normalno vrednost tipa `'a` in vrne vrednost s kontekstom [5]. Poglejmo si, kakšen je predpis v našem primeru:

```
let (>>=) (Just x) f = f x
let (>>=) Nothing f = Nothing
```

Zanimalo nas je kako vrednost tipa `Maybe 'a` smiselno vstavimo v funkcijo tipa `('a -> Maybe 'b)`. Ker imamo pri vrednostih tipa `Maybe` dva različna možna predpisa, tudi funkcijo `bind` definiramo v dveh delih. V primeru, ko imamo vrednost `Just x`, funkcijo f uporabimo na x-u, sicer pa vrnemo vrednost `Nothing`. Če sedaj

vse naše predpise združimo v skupni modul, dobimo modul tipa `MONADA`, ki definira monado `Maybe`, kot je prikazano v spodnji kodi:

```
module Mybe_monada : MONADA =
sig
  type m 'a  = Maybe 'a

  let  return x = Just x

  let (>>=) (Just x) f = f x
  let (>>=) Nothing f = Nothing
end
```

Porazdelitev

V nadaljevanju želimo definirati modul tipa `MONADA`, ki bo obravnaval končne, diskretne porazdelitve. Tip porazdelitve je predpisan z naslednjo kodo:

```
type finite_dist 'a = ('a * float) list
```

Parameter `'a` zopet stoji za poljuben tip. V primeru, ko bi imeli porazdelitev vrednosti delnic, bi bil `'a` enak tipu `float list`, saj so vrednosti delnic z njim predstavljene. Končne porazdelitve tipa `'a` predstavljajo naš tip z dodano vrednostjo za katerega bomo napisali monado. Prvo definicijo, ki je v modulu `MONADA` potrebna, torej že imamo:

```
type m 'a  = finite_dist 'a
```

Nadalje moramo smiselno napisati predpis funkcije `return`, ki vrednosti tipa `'a` priredi vrednost tipa `finit_dist 'a`. To bomo storili tako, da bomo vrednost vložili v porazdelitev, v kateri bo kot edina možna izbira izbrana z verjetnostjo 1, to predpisuje naslednja koda:

```
return x = [(x, 1.)]
```

Preostane nam še definicija tretjega obveznega predpisa v monadi, operacije `>>=`. Sprašujemo se, kako vrednost tipa `finit_dist 'a` smiselno vstavimo v funkcijo, ki sprejme normalno vrednost tipa `'a` in vrne porezdelitev tipa `finit_dist 'b`.

Če so a_i vrednosti tipa `'a` in $p_i, i = 1, \cdots, n$ njihove verjetnost, imamo na začetku neko konkretno končno in diskretno porazdelitev:

$$\begin{pmatrix} a_1 & a_2 & \dots & a_n \\ p_1 & p_2 & \dots & p_n \end{pmatrix}$$

Imamo tudi funkcijo f, ki sprejme vrednost tipa `'a` in vrne porazdelitev. f lahko kot parameter sprejme katero koli od vrednosti a_i. Omenjena preslikava izgleda na sledeči način:

$$f(a_i) = \begin{pmatrix} b_1 & b_2 & \dots & b_m \\ p_i \cdot q_{i1} & p_i \cdot q_{i2} & \dots & p_i \cdot q_{im} \end{pmatrix}$$

Pri čemer je $f(a_i) = b_1$ z verjetnostjo q_{i1}, to verjetnost pa pomnožimo s p_i, ki predstavlja verjetnost, da f kot parameter izmed vseh a-jev sprejme ravno a_i.

Vrnimo se k definiciji funkcije `(>>=)`. Zanima nas, kako bi lahko funkcijo f smiselno uporabili na zgornji porazdelitvi a_i-jev. To bomo storili tako, da bomo f

uporabili na vsakem izmed elementov a_i iz porazdelitve. Pri tem bomo za vsak parameter a_i, ki ga bomo vstavili v f, kot izhodno vrednost funkcije dobili porazdelitev, kot smo pokazali v zgornjem razmisleku.

Na koncu bo potrebno še smiselno združiti vse dobljene porazdelitve. S tem problemom smo se že srečali v poglavju s prestrezniki, kjer smo definirali funkcijo `combine`. Identično definirano funkcijo uporabimo tudi v tem primeru.

Razmisleku, ki smo ga ravnokar opisali, sledi spodnji predpis funkcije `(>>=)`:

```
let (>>=) x f =
    combine (List.flatten (List.map (fun (a, p) ->
     List.map (fun (b, q) -> (b, p *. q)) (f a)) x))
```

Če strnemo vse potrebne definicije v skupni modul, dobimo monado za porazdelitev.

5.4 Simulacija trga z moduli, funktorji in monadami

Sedaj, ko smo spoznali vsa potrebna orodja, bomo zgoraj napisano kodo združili in preoblikovali tako, da bo tako kot pri prestreznikih, naša simulacija predstavljala model Cox-Ross-Rubinstein. Pri tem bo predpis za simulacijo trga v funktorju ves čas ostajal nespremenjen. Odvisno od definicije vhodnega modula, pa bomo enkrat računali zgolj posamezne realizacije, drugič pa končno porazdelitev modela.

Modul tipa `TRG`, ki smo ga definirali zgoraj, razširimo z monado. Operacija `obresti` pa ostaja nespremenjena. S tem dobimo sledečo signaturo:

```
module type TRG' =
sig
  include MONADA
  val obresti : 'a  -> m 'a
end
```

Nekoliko moramo spremeniti tudi zgornjo simulacijo trga, ki se nahaja v funktorju. Videli smo, da brez uporabe monad ne moremo spreminjati tipov s katerimi računamo v funktorju, zato je morala funkcija `obresti` vedno vračati seznam obrestnih mer. Nismo mogli poljubno prehajati na porazdelitve. Spodaj bomo pokazali, kako to omejitev odpravljajo monade. Sedaj namreč funkcije, v katerih smo prej računali z obrestmi, opremimo z operacijo `bind` iz monade. Ta bo povedala, kako lahko funkcijo, ki kot parameter sprejme navadne obresti, sedaj uporabimo tudi na porazdelitvah vseh možnih obrestnih mer na trgu. Porazdelitve namreč predstavljajo dodatni kontekst k tipu obrestnih mer v monadi. Spodaj najdemo popravljeni predpis v funktorju, v katerem ključno vlogo igra funkcija `<<=`:

```
module Simulacija_trga' (T : TRG') =
struct
  open T

  let simulacija_trga s0 n =
    let rec korak k s =
      if k >=  n then
        return s
      else
        (obresti s) >>= fun stanje ->
```

```
        let s_novi = (List.map2 ( *. ) stanje s) in
        korak (k + 1) s_novi
    in
    korak 0 s0;;

end
```

V prvi realizaciji modula želimo računati porazdelitev modela, zato bomo funkcije iz monade predpisali tako, kot v monadi za porazdelitve. Monadi za porazdelitev smo dodali le funkcijo `obresti`, ki vrne celotno porazdelitev obrestnih mer v prihodnjem obdobju in s tem spodaj dobimo prvo realizacijo modula tipa `TRG'`:

```
module Obrestuj_porazdelitev : TRG' =
struct
  type m  'a = finite_dist 'a

  let return x = [(x, 1.)]

  let combine lst =
    let rec add_to dist (a, p) =
      match dist with
      | [] -> [(a,p)]
      | (b,q) :: lst ->
        if a = b
        then (b, p +. q) :: lst
        else (b, q) :: (add_to lst (a, p))
    in
    List.fold_left add_to [] lst

  let (>>=) x f =
        combine (List.flatten (List.map (fun (a, p) ->
          List.map (fun (b, q) -> (b, p *. q)) (f a)) x))

  let u = [1.05]
  let d = [0.9]
  let obresti  s = [(u, 0.6), (d, 0.4)]

end
```

Če sedaj to monado uporabimo na funktorju, dobimo izračun končne porazdelitve binomskega modela, ki vse vrednostne papirje v dobrem stanju obrestuje z obrestno mero `u`, v slabem stanju pa z `d`. Spodaj dodajamo še kodo s katero izvedemo izračun:

```
module Primer3 = Simulacija_trga (Obrestuj_porazdelitev);;
Primer3.simulacija_trga [20.] 5;;
```

Za primerjavo na drugi strani definiramo še modul za eno samo realizacijo binomskega modela. Tako bomo računali s seznamom obrestnih mer, zaradi česar bo tipa `'a` s kontekstom kar enak normalnemu tipu `'a`. Tudi definiciji funkcij `return` in `bind` bosta trivialni, saj bi ta primer lahko definirali že brez uporabe monade. Spodaj kljub temu dodajamo primer kode:

```
module Obrestuj_realizacija =
```

```
struct
  type 'a t = 'a

  let return x = x

  let (>>=) x f = f x

  let u = 1.05
  let d = 0.9
  let p = 0.6 (*Verjetnost dobrega razvoja ekonomije*)

  let obresti s =
    let izberi = (Random.float 1.) < p in
    List.map (fun x ->  if izberi then u else d) s
end
```

5.5 Primerjava s prestrezniki

Identični primer simulacije trga smo zgoraj napisali na z uporabo različnih konceptov na dva različna načina, pri čemer smo sledili želji, da bi funkcija s simulacijo trga ostajala nespremenjena, spreminjale naj bi se le obrestne mere delnic in s tem evolucija simuliranega trga. Želeli smo, da nam oba načina omogočata tudi spremembo tipa izračuna iz seznama realnih števil v končno porazdelitev seznamov realnih števil (zaradi česar smo morali v drugem načinu vpeljati monade).

V prvem načinu smo uporabili prestreznike in računske učinke, v drugem načinu pa smo potrebovali koncepte modul, funktor in monada, katerih uporaba zahteva nekoliko bolj kompleksen in daljši zapis kode (za isti primer smo napisali dvakrat toliko vrstic kode kot pri uporabi prestreznikov).

Prednost, ki jo prinašajo prestrezniki je morda še v tem, da je večje število prestreznikov za različne računske učinke med seboj zelo preprosto gnezditi in kombinirati, s čimer se bomo srečali še v nadaljevanju. Videli smo tudi, da prestrezniki lahko sami spremenijo tip izračunov, brez potrebe po definiciji operacije `bind`.[5]

Prestrezniki so imeli za svoj nastanek med drugim podlago tudi v monadah, ki so splošnejši koncept in zato lahko pokrivajo več različnih primerov.

6 Nedeterministično programiranje in razširitev simulacije trga

Iz zgornjih primerov že lahko povzamemo nekaj značilnosti, ki jih prinaša nedeterminističen način programiranja in vlogo, ki jo prestrezniki v njem igrajo. Del kode s simulacijo trga smo definirali zgolj enkrat, kasneje je ves čas ostajal nespremenjen. Prestrezniki so nam omogočili, da kljub temu z njim lahko delamo različne izračune.

V našem primeru je šlo za preproste in krajše kode. Kljub temu si lahko predstavljamo uporabnost takšnega načina programiranja, kjer bi večji del kode lahko

[5]Razlaga skupaj s primerjavo, kako ujemanje tipov zagotavljajo prestrezniki bo poudarjena še v poglavju 6.2 Prestrezniki za operacijo Obrestuj

ostajal ne spremenjen, z njim pa bi lahko delali različne izračune. Da je to mogoče mora biti jedro kode, ki bo ostalo nespremenjeno, napisano čim bolj splošno, saj bo tako omogočalo več različnih in bolj spremenljivih realizacij trga. Nedeterministične operacije, katerih delovanje želimo spreminjati definiramo kot operacije, ki bodo čakale na predpis iz prestreznika.

V nadaljevanju bo simulacija trga vsebovala večje število spremenljivih operacij, zaradi česar si bomo lahko pogledali, kako enostavno je gnezdenje in kombiniranje različnih prestreznikov, ki bodo predpisovali njihovo delovanje.

Delovanje finančnega trga želimo razširiti tako, da bo poleg tveganih vrednostnih papirjev (delnic) vseboval tudi banko, ki predstavlja ne tvegan, napovedljiv proces. Na finančni trg bomo vključili tudi uporabnika, ki v vsakem obdobju lahko na njem trguje. Delnice so sedaj predstavljene s seznamom parov, kjer za vsako delnico najprej navedemo njeno vrednost, nato pa število teh delnic na trgu. Za razširjeni trg najprej definiramo osnovne tipe:

```
type banka = float
type delnice = (float * int) list
type portfelij = int list
type zgodovina = float list
```

Portfelij predstavlja seznam celih števil, ki pove, koliko posameznih delnic iz trga imama uporabnik v posesti. Poleg tega bomo v seznam realnih števil včasih morali beležiti zgodovino dogajanja na trgu, da bomo lahko določili želeno trgovalno strategijo. Nadalje definiramo tri nove operacije:

```
effect Trguj : zgodovina * float * delnice -> portfelij;;
effect Obrestuj : delnice -> delnice;;
effect Shrani_zgodovino : zgodovina -> zgodovina;;
```

Namesto operacije `Obresti` imamo sedaj operacijo `Obrestuj`, ki ne vrača več obrestnih mer, temveč že obrestovane vrednosti delnic. Poleg tega imamo na trgu dodatni nedeterministični operaciji `Trguj` in `Shrani_zgodovino`, ki sta definirani kot spremenljivi operaciji. Prva izmed njiju določi trgovalno strategijo oziroma portfelij, ki ga uporabnik oblikuje na trgu. Drugi operacija bo v zgodovino shranil podatke o preteklih stanjih finančnega trga, ki bodo potrebni za oblikovanje trgovalne strategije.

Razširjena simulacija trga sprejme sledeče vhodne parametre:

- `n` – število korakov simulacije
- `b_p` – obrestna mera v banki za pozitivno stanje
- `b_n` – obrestna mera v banki za negativno stanje
- `b0` – začetno stanje na banki
- `s0` – začetna vrednost trga (vrednosti in števila delnic na trgu)
- `p0` – začetni portfelij.

```
let simulacija n b_p b_n b0 s0 p0 =
  (* EN KORAK SIMULACIJE *)
  let rec korak k b s zgo p =
```

```
  if k >= n then
     let vrednost_p' = vrednost_portfelija p s in
     (b , vrednost_p')
  else
    (* TRGUJEMO *)
    let staraVrednost_p = vrednost_portfelija p s in
    let premozenje = b +. sestej staraVrednost_p in
    let p_novi =
       perform (Trguj (zgo, premozenje, s)) in
    let p_novi = omeji s  p_novi in
    let novaVrednost_p = vrednost_portfelija p_novi s in
    let b_novo = premozenje -. (sestej novaVrednost_p) in
    (* SHRANI ZGODOVINO *)
    let zgo_nova =
       perform (Shrani_zgodovino (zgo, s)) in
    (* EVOLUCIJA TRGA *)
    let s_obrestovane = perform (Obrestuj s) in
    let b_obrestovano =
       b_novo *. (if b_novo < 0.0 then b_n else b_p) in

    korak (k+1) b_obrestovano s_obrestovane
          zgo_nova p_novi
in
korak 0 b0 s0 (cene_delnic s0) p0
;;
```

Zgoraj napisana funkcija določa splošno delovanje trga. Vedno najprej trguje uporabnik. Izračuna se njegovo premoženje, potem prestreznik pove, kakšen portfelij si oblikuje. Trg portfelij omeji, če je uporabnik želel kupiti več delnic, kot jih trg ponuja. Izračuna se trenutna cena portfelija in se za njeno vrednost zmanjša uporabnikovo stanje na bančnem računu.

Druga operacija nato pove, kateri podatki o trenutnem stanju trga naj se shranijo v zgodovino in prenesejo v naslednje obdobje. Nazadnje pride še do evolucije trga. Tu se obrestujejo vrednosti delnic, kar določa operacija `Obrestuj`, ter bančni račun z eno izmed deterministčnih obrestnih mer, odvisno od pozitivnega oz. negativnega stanja denarja, ki ga ima uporabnik na bančnem računu.

Simulacijo smo želeli napisati čim bolj splošno, saj je to tisti del kode, ki bo ves čas ostajal nespremenjen. Kljub temu pa bomo na njem izvajali različne izračune in realizacije finančnih modelov. V simulaciji imamo tri nedeterministične operacije, za katere bomo v nadaljevanju napisali različne prestreznike.

6.1 Prestrezniki za operaciji `Trguj` in `Shrani_zgodovino`

Operaciji `Trguj` in `Shrani_zgodovino` sta med seboj povezana, saj je trgovalna strategija dostikrat odvisna od tega, kako se je trg razvijal v preteklosti. Prestreznik `shrani_zgodovino` bo torej shranil tiste informacije iz preteklosti, ki bodo potrebne za oblikovanje portfelija.

Prestrezniki za operacijo `Trguj` bodo v vsakem stanju določili, na kakšen način se oblikuje portfelij. V tem razdelku bomo na začetku imeli oblikovan portfelij, v

katerem bodo količine delnic večje od 0. Potem bomo določili trgovalno strategijo, kot čas ustavljanja [3]. To je pravilo, ki v vsakem času pove ali še zadržimo delnico, ki jo imamo ali delnico prodamo.

Naj cena delnice sledi slučajnemu procesu $(S_t)_{t=0}^n$. Spodaj si lahko pogledamo dva primera časov ustavljanja za takšno delnico in trgovalno strategijo, ki jo implicirata.

1. **Čas prvega padca vrednosti** τ^1 je čas ustavljanja in ga lahko definiramo na sledeči način:

$$\tau^1 = \begin{cases} min\{t, S_t < S_{t-1}\} & , če\{t; S_t < S_{t-1}\} \neq \emptyset; \\ T & , sicer. \end{cases}$$

   Delnice bomo v portfelju držali dokler ne pride do prvega padca njihove vrednosti, nato pa jo bomo prodali. Ker imamo v portfelju več različnih delnic takšno pravilo velja za vsako posebej. Da bomo vedeli ali je prišlo do padca vrednosti delnice, moramo v vsakem času poznati tudi vrednost iz prejšnjega stanja, zato definiramo prestreznik za operacijo `Shrani_zgodovino` na sledeči način:

```
let prejsno_stanje = handler
    | effect (Shrani_zgodovino (zgo, s)) k ->
    (continue k (cene_delnic s))
;;
```

   Zgornji prestreznik bo v spremenljivko `zgodovina` shranil podatke o ceni delnic pred evolucijo trga in jih prenesel v naslednje obdobje. Sedaj lahko definiramo tudi prestreznik za trgovalno strategijo:

```
let drzimo_do_prvega_padca_vrednosti = handler
  | effect (Trguj (zgo, premozenje, s)) k ->
  let cena_s = cene_delnic s in
  let rec cas_ustavlanja zgo cena_delnic =
  match zgo, cena_s with
  | [], [] -> []
  | (x::xs), (y::ys) ->
  (if x <= y then 1 else 0) :: (cas_ustavlanja xs ys) in

  (continue k (cas_ustavlanja zgo cena_s))
;;
```

2. **Čas prvega povečanja vrednosti za** ${}^1/_4\tau^2$ je prav tako čas ustavljanja. Definiramo ga kot:

$$\tau^2 = \begin{cases} min\{t, S_t > \frac{5}{4}S_0\} & , če\{t; S_t > \frac{5}{4}S_0\} \neq \emptyset; \\ T & , sicer. \end{cases}$$

V tem primeru je potrebna informacija o cenah delnic v času 0. Oba potrebna prestreznika sta definirana na sledeči način:

```
let zacetno_stanje = handler
    | effect (Shrani_zgodovino (zacetno_st, s)) k ->
```

```
    (continue k zacetno_st)
;;

let drzimo_do_povecanja_za_cetrtino = handler
    | effect (Trguj (zgo, premozenje, s)) k ->
    let cena_s = cene_delnic s in
    let rec cas_ustavlanja zgo cena_s =
      match zgo, cena_s with
      | [], [] -> []
      | (x::xs), (y::ys) ->
      (if (5. /. 4. *. x) >= y then 1
      else 0) :: (cas_ustavlanja xs ys)
      in
      (continue k (cas_ustavlanja zgo cena_s))
;;
```

6.2 Prestrezniki za operacijo `Obrestuj`

Preostane nam še napisati prestreznik za operacijo `Obrestuj`, pri čemer želimo (kot v prejšnjih poglavjih) imeti možnost računanja z več različnimi tipi, npr. zgolj z eno samo realizacijo razvojev trga, s končno porazdelitvijo ali pričakovano vrednostjo. Hkrati pa želimo obrestovanje na trgu predpisati s poljubno končno diskretno porazdelitvijo. V ta namen si najprej poglejmo definicijo algoritma, ki izbere naključno vrednost na podlagi dane končne diskretne porazdelitve.

Algoritem za generiranje naključnih števil na podlagi končne diskretne porazdelitve

Najprej definiramo operacijo `Izberi`, ki predstavlja Bernoullijevo porazdelitev:

```
effect Izberi : float -> bool;;
```

Operacija kot vhodni parameter sprejme realno število (spodaj označeno s `p`), ki predstavlja verjetnost s katero je izbrana logična vrednost `true` [6], torej ima naslednjo porazdelitev:

$$Izberi(p) \sim \begin{pmatrix} true & false \\ p & (1-p) \end{pmatrix}$$

Na podlagi operacije `Izberi` bo sprejeta odločitev, ali iz dane porazdelitve izberemo posamezno vrednost ali ne. Pri tem bo verjetnosti, na podlagi katere bo sprejeta takšna odločitev ustrezala verjetnosti iz dane porazdelitve.

Algoritem bo deloval rekurzivno, robni pogoj pokriva primere v katerih vhodna porazdelitev vsebuje en sam element, tedaj je ta očitno enak izbranemu elementu. V osnovnem koraku rekurzije je z ustrezno verjetnostjo (spodaj `normiran_p`) sprejeta odločitev o izbiri prvega elementa, če je le ta izbran se rekurzija konča, sicer v nadaljevanju izbiramo elemente iz preostanka porazdelitve, torej porazdelitve brez prvega elementa. Opisanemu algoritmu sledi naslednja funkcija:

[6] Funkcije v tem delu so povzete po [8] in nekoliko prirejene za naš primer.

```
let izberi_iz_porazdelitve l =
  let rec izberi_iz_porazdelitve' acc = function
  | [(x, _)] -> x
  | (x, p)::xs ->
    let normiran_p = (p/.(1. -. acc)) in
    if perform (Izberi normiran_p) then x
    else
        izberi_iz_porazdelitve' (acc +. p) xs
  in
  izberi_iz_porazdelitve' 0. l
;;
```

V akumulatorju `acc` je shranjena vsota verjetnosti vseh elementov, ki jih v preteklosti nismo izbrali. V zgornji funkciji lahko opazimo, da je verjetnost dana v vhodni porazdelitvi na vsakem koraku normirana z vrednostjo `1 - acc`. Poglejmo si idejo dokaza, zakaj tako normirane verjetnosti ustrezajo verjetnostim iz dane porazdelitve. Denimo, da slučajno spremenljivko izbiramo na podlagi spodnje porazdelitve, kjer je $n > 2$:

$$\begin{pmatrix} a_1 & a_2 & \dots & a_n \\ p_1 & p_2 & \dots & p_n \end{pmatrix}$$

Ker je vrednost akumulatorja v začetku enaka 0, je `normiran _p` tedaj kar enak p_1 in torej velja $P(a_1) = p_1$. Nadalje si poglejmo izračun verjetnosti s katero sta izbrana elementa a_2 oz. a_3:

$$P(a_2) = (1 - p_1)(\frac{p_2}{1 - p_1}) = p_2$$

pri čemer prvi oklepaj predstavlja verjetnost, da ne izberemo prvi elementa iz porazdelitve, ki ga množimo z verjetnostjo, da je izbran drugi element. Vidimo lahko, da se dobljena verjetnost ujema s tisto iz vhodne porazdelitve. Po enakem razmisleku zapišemo še verjetnost izbire tretji elementa iz porazdelitve:

$$P(a_3) = (1 - p_1)(1 - \frac{p_2}{1 - p_1})(\frac{p_3}{1 - p_1 - p_2}) = p_3$$

Prva dva oklepaja zopet ustrezata verjetnosti, da ne izberemo niti prvega niti drugega elementa v porazdelitvi. V tretjem oklepaju pa se nahaja verjetnost, da je hkrati izbran tretji element.

S pomočjo podobnega razmisleka, bi lahko pokazali, da se verjetnosti ujemajo za poljuben element iz porazdelitve in torej zgornji algoritem ustrezno deluje.

Delovanje operacije `Izberi` predpišemo s prestrezniki, tako da bo ustrezal binomski porazdelitvi, hkrati pa bo eden izmed prestreznikov vseboval predpis za računanje ene realizacije izračuna, drugi za računanje porazdelitve vseh možnih realizacij, ter tretji za pričakovano upanje. Poglejmo si najprej prvi predpis:

```
let realizacija = handler
  | v -> v
  | effect (Izberi p) k ->
  let izbira = perform (Random_float 1.) < p in
  continue k izbira
;;
```

Zgoraj je najprej definiran prestreznik, ki ustreza eni realizaciji izbire, tako da vrne eno izmed logičnih vrednosti `true` oz. `false` z vhodno verjetnostjo p oz. $1 - p$.

Nadalje definiramo prestreznik za računanje s porazdelitvami. Predpis sledi razmisleku s katerim smo se že srečali v zgornjih poglavjih. Računanje nadaljujemo za obe logični vrednosti, ter nato končne izračune najprej pretvorimo v porazdelitve, nato pa te porazdelitve smiselno združimo s funkcijo `combine`, katere predpis smo tudi že zapisali zgoraj. Poglejmo si opisani prestreznik za porazdelitev:

```
let porazdelitev = handler
  | v -> [(v, 1.0)]
  | effect (Izberi p) k ->
     combine p (continue k true) (continue k false)
;;
```

Pri nedeterminističnem programiranju brez prestreznikov (opisanem v 5. poglavju zgoraj), smo morali uporabiti monade, kadar smo želeli, da identična koda vrača izračune različnih tipov. Torej, kadar npr. želimo da so rezultati naših izračunov enkrat zapisani v realnih številih, drugič v seznamih realnih števil, v seznamih parov itd.

Sedaj, ko smo se že seznanili z delovanjem monad, smo lahko pozorni na to, kako „zamenjavo tipa“ naredijo prestrezniki. Prestreznik upošteva, da mora operacija `Izberi` vračati logično vrednost, da se bo v funkciji `izberi_iz_porazdelitve` ujemal z operacijami, ki s temi vrednostmi računajo naprej. Zaradi kontunuacije `k` se računanje nadaljuje, prestrezniki pa nato končnim izračunom spremeni tip v diskretno porazdelitev.

Prestreznik sam opravi vse potrebne transformacije, brez potrebe po dodajanju bolj kompleksnih konstruktov, kot so monade.

Dodajamo še predpis prestreznika za pričakovano vrednost, ki prav tako kot v primeru porazdelitve, računanje nadaljuje za obe logični vrednosti, ko pa pride do končnih izračunov te množi z ustreznimi verjetnostmi ter sešteva med seboj:

```
let rec pomnozi p = function
  | (x, xs) ->
     (p *. x, (map1 (fun x -> (1.0 -. p) *. x) xs))
;;

let rec zdruzi prvi drugi =
  match prvi, drugi with
  | (x, xs), (y, ys) -> (x +. y, (map2 ( +. ) xs ys))
;;

let pricakovana_vrednost = handler
  | v -> v
  | effect (Izberi p) k ->
  (zdruzi (pomnozi p (continue k true))
              (pomnozi (1.0 -. p) (continue k false)))
;;
```

Sedaj lahko zgornje funkcije uporabimo za definicijo prestreznika, ki bo delnice obrestoval z naključnimi obrestnimi merami izbranimi glede na končno diskretno

porazdelitev. Spodaj podajamo primer v katerem imamo trg s tremi delnicami in tremi možnimi stanji ekonomije:

```
let obrestuj_iz_diskretne_por = handler
  | v -> v
  | effect (Obrestuj s) k ->
    let obresti1 =  [1.2; 0.94; 0.99] in
    let obresti2 = [0.89; 1.1; 1.04] in
    let obresti3 = [0.92; 1.0; 0.87] in
  (continue k (izberi_iz_porazdelitve
     [(obrestuj obresti1 s,0.3);(obrestuj obresti2 s,0.4);
     (obrestuj obresti3 s,0.3)]))
;;
```

Imamo torej primer gnezdenih prestreznikov, saj je v predpisu uporabljena funkcija `izberi_iz_porazdelitve` v kateri se nahaja operacija `Izberi`, zato moramo prestreznik `obrestuj_iz_diskretne_por` zajeti še z enim prestreznikom, ki bo določal njeno delovanje. V nasprotnem primeru bi dobili izjemo „uncaught effect Izberi“.

Sedaj lahko na simulaciji trga definirani v začetku šestega poglavja, uporabljamo poljubne kombinacije zgoraj definiranih prestreznikov in na ta način izvajamo različne izračune. Spodaj kod primer navajamo eno izmed kombinacij:

```
let primer n =
  with obrestuj_iz_diskretne_por handle
  with drzimo_do_prvega_padca_vrednosti handle
  with prejsno_stanje handle
  simulacija n 1.001 0.9 200.0
          [(30.0,1); (40.0,1); (50.0,1)] [1; 1; 1]
;;
```

Najprej smo simulacijo trga prestregli s tremi prestrezniki, ki ustrezajo definicijam operacij `Obrestuj`, `Trguj` in `Shrani_zgodovino`. Na ta način imamo v primeru določeno trgovalno strategijo in način obrestovanja s katerim trg prehaja v naslednje obdobje.

Potrebujemo še prestreznik za operacijo `Izberi`, s katerim bomo za zgornji primer določili ali pri obrestovanju želimo računati s porazdelitvijo, eno verjetnostno realizacijo ali s pričakovano vrednostjo. V spodnji kodi zaporedno izvedemo vse tri izračune:

```
with porazdelitev handle
  primer 3
;;

with realizacija handle
  primer 3
;;

with pricakovana_vrednost handle
  primer 3
;;
```

Seveda bi v primeru lahko uporabili drugačno kombinacijo zgoraj definiranih prestreznikov in dobili drugačne izračune. Prav tako bi lahko definirali še nove prestreznike

tako, da bi npr. iskali trgovalno strategijo, ki optimizira pričakovani dobiček, množico več trgovalnih strategij itd. tako, da možnosti za različne izračune, ki jih odpira primer še zdaleč nismo izčrpali. Vidimo pa, kako lahko na preprost in pregleden način s prestrezniki kombiniramo in gnezdimo več različno delujočih operacije.

Na tem mestu zaključujemo s primeri simulacije trga, ki so bili tekom celotne diplomske naloge do sedaj med seboj povezani. Tudi v naslednjem poglavju se bomo srečali s simulacijo trga delnic in sicer na podlagi Brownovega gibanja, vendar primer lahko stoji sam zase in ne potrebuje več nobene od dosedanjih definicij.

7 Prestreznik za spreminjanje stanja in Brownovo gibanje

Ogledali smo si že, kako s prestrezniki lahko spreminjamo in preusmerimo izračun kode, vendar imajo ti lahko tudi drugačne vloge. V tem poglavju si bomo najprej na primeru slučajnega sprehoda pogledali uporabo prestreznika za spreminjanje stanj. Pri tem stanje razumemo kot eno možno konfiguracijo pomnilnika ali kot vse (do sedaj) definirane spremenljivke. Stanje bi lahko zajeli tudi širše, kot stanje celotnega računalnika ali celo računalnika skupaj z njegovim uporabnikom.

Recimo, da imamo stanje, katerega tip označimo z `s` in je lahko predstavljeno z realnim številom, urejenim parom ali urejeno n-terico, odvisno od tega koliko spremenljivk želimo imeti. Poleg stanja imamo še dve operaciji, ki spreminjata to stanje v pomnilniku. Operacijo `Get: unit -> s` ter operacijo `Set: s -> unit` s katerima beremo in nastavljamo stanje.

V ukaznih programskih jezikih (Python, C, C++, Java) programiramo s spremenljivkami, ki hranijo stanje. V teh jezikih pišemo `x := e` namesto operacije `Set` in preprosto `x` namesto `Get x`, vendar je to le poenostavljen način pisanja, v ozadju pa sta ti dve operaciji še vedno prisotni.

Omenjeni operaciji, ki spreminja vrednost spremenljivk v pomnilniku sta učinka. V čistih programskih jezikih (npr. Haskell) so namreč funkcije definirane tako, da ne moremo spreminjati nobenega globalnega stanja ali spremenljivke, lahko le delamo izračune in vračamo njihove rezultate [5]. Zaradi te lastnosti nam ni treba skrbeti, da bo prišlo do napake, saj v vsakem času točno vemo, kakšna je vrednost posamezne spremenljivke. Ker pa imamo določene primere, ki so neločljivo povezani s spreminjanjem stanj, želimo vseeno imeti tudi to možnost, hkrati pa na drugi strani ohranjati čistost programa. V programskem jeziku Haskell to omogočajo posebne monade za stanja.

Na kakšen način delujejo monade za stanje v Haskell-u si lahko preberete v 13. poglavju knjige Learn You a Haskell for Great Good! [5]. Tukaj pa si bomo podrobneje pogledali, koko spreminjanje stanj, torej operaciji `Get` in `Set`, obvladujejo prestrezniki, pri čemer si bomo pomagali s primerom slučajnega sprehoda.

7.1 Prestreznik za stanje in definicija slučajnega sprehoda

Pri slučajnem sprehodu se v začetku nahajamo v neki vrednosti `v_0`, ki bo pri nas predstavljena z realnim številom, nato pa se na vsakem koraku iz te vrednosti naključno premaknemo v neko novo stanje. Novo stanje je torej odvisno od prejšnjega

stanja ter prirastka dodanega temu stanju.

Preden se lotimo definicije slučajnega sprehoda najprej predpišimo dve novi operaciji, s katerima bomo nadzorovali spremembo stanj v slučajnem sprehodu:

```
type s = float

effect Get: unit -> s
effect Set: s -> unit
```

Zaradi boljše preglednosti smo definirali še tip `s`, ki je predstavljen z realnim številom in pomeni stanje oz. vrednost v kateri se slučajni sprehod nahaja na posameznem koraku. Operacija `Set`, katere glavna vloga bo nastavitev novega stanja, nima nobene pomembne vrednosti, ki bi jo morala vrniti, zato vrača samo `()`.

Pri prestrezniku za stanje imamo torej dve operaciji `Get` in`Set` s prestreznikom spodaj povemo, kako delujeta [7] [8]:

```
let stanje v_0 = handler
  | y -> (fun _ -> y)
  | effect (Get ()) k -> (fun s -> (continue k s) s)
  | effect (Set s') k -> (fun _ -> (continue k ()) s')
  | finally g -> g v_0
;;
```

Podani prestreznik je nekoliko bolj zapleten za razumevanje, zato bomo v nadaljevanju za vsakega izmed predpisov v prestrezniku posebej naredili razmislek o tem, kako deluje:

1. `y -> (fun _ -> y)`:
   Če imamo „čisto“ vrednost, jo prestreznik pretvori v funkcijo. Ta funkcija bo sprejela nek vhodni parameter, ga zavrgla in vrnila vrednost, ki jo je zajel prestreznik.[8]

2. `effect (Get ()) k -> (fun s -> (continue k s) s)`:
   Nadalje imamo operacijo `Get`, ki jo prav tako pretvorimo v funkcijo. Predpisana funkcija sprejme trenutno stanje in ga vrne v kontinuacijo `k`. Ker funkcija vrne stanje izpoljuje predpis tipa operacije `Get`, saj mora ta vrniti stanje. V nadaljevanju bo prestreznik tudi kontinuacijo pretvoril v funkcijo, ki bo naprej računala s stanjem `s`. Funkcija pa bo kot parameter potrebovala (še eno) stanje, zato ji ga podamo z dodatnim parametrom `s` (zadnji v predpisu).

3. `effect (Set s') k -> (fun _ -> (continue k ()) s')`:
   Parameter `s'`, ki ga operacija `Set` sprejme, predstavlja novo stanje. `Set` spremenimo v funkcijo, ki bo sprejela trenutno stanje, vendar ga ignoriramo, ker ga ne potrebujemo več in nato rečemo kontinuaciji, da je rezultat `()`, kar ustreza tipu operacije `Set`. V kontinuaciji `k` se nato nadaljuje računanje, ki je prav tako zajeto s prestreznikom, kar pomeni, da bo tip izračuna, ki ga bo vrnila kontinuacija, funkcija. To sledi iz predpisa prestreznika, ki „nečisti“ operaciji

[7] Prestreznik deluje podobno kot monade za stanje v Haskell-u.

[8] Ta predpis je, če vzporedno gledamo monade za stanje v Haskell-u, podoben definiciji funkcije `return`. Omenjena funkcija bi običajno vrednost `v` spremenila v izračun stanja, ki vrne enako vrednost `v` in nespremenjeno stanje.

in tudi same vrednosti spreminja v funkcije. Ta funkcija bo kot parameter potrebovala stanje, zato ji ga podamo z `s'`.

4. `finally g -> g v_0`:
   Prvič se srečamo tudi z zadnjim predpisom v prestrezniku. Videli smo že, da so vsi omenjeni predpisi v prestrezniku vračali funkcije in bodo torej kodo, ki jo bodo zajeli spremenili v funkcijo. Funkcija pa bo vrnila izračun šele, ko jo bomo klicali na nekem parametru. Zadnji predpis torej pove, naj se na koncu dobljena funkcija, v katero se je pretvorila koda zajeta s prestreznikom, izvede tako, da kot parameter sprejme vrednost `v_0`. Iz definicije slučajnega sprehoda v nadaljevanju bo sledilo, da je potreben parameter `v_0`, ki predstavlja vrednost v kateri se slučajni sprehod nahaja na začetku.

Sedaj s pomočjo zgornjega prestreznika definiramo slučajni sprehod, v katerem se nahaja tudi operacija `Prirastek`, ki bo vračala velikost prirastka na posameznem koraku slučajnega sprehoda in še čaka, da ji prestreznik pove, kako naj deluje. Vhodni parametri napisane funkcije so sledeči:

- `v_0` – začetna vrednost slučajnega sprehoda
- `n` – število korakov slučajnega sprehoda

Predpis funkcije za slučajni sprefod skupaj z operacijo `Prirastek` je sledeč:

```
effect Prirastek: unit -> float

  let slucajni_sprehod v_0 n =
  with stanje v_0 handle
    (let rec sprehajanje m =
       let novo_stanje =
         perform (Get ()) +. perform (Prirastek ())
       in
       perform (Set novo_stanje);
       if m < n then sprehajanje (m + 1)
       else perform (Get ())
     in
     sprehajanje 0)
  ;;
```

V zgornjem predpisu slučajnega sprehoda je rekurzivna funkcija `sprehajanje` že ujeta s prestreznikom za stanja. Funkcija deluje tako, da na vsakem koraku rekurzije najprej vzame stanje, mu prišteje prirastek in vrednost shrani v lokalno spremenljivko `novo_stanje`. Nato to stanje sprejme funkcija `Set` in ga prenese v naslednji korak. Prestreznik spremembo stanja izvaja tako, da si operaciji `Get` in `Set` na vsakem koraku (kot parameter, ki ga sprejme funkcija, v katero se operaciji spremenita,) na nek način „podajata“ spreminjajoče se stanje?

Da bomo slučajni sprehod lahko izvedli potrebujemo še prestreznik, ki bo na verjetnosten način predpisal vrednosti prirastkov. Tega bomo v nadaljevanju definirali tako, da bo ustrezal procesu Brownovega gibanja.

7.2 Prirastki za Brownovo gibanje

Ime Brownovo gibanje je bilo najprej uporabljeno za pojav kaotično gibanja cvetnega prahu plavajočega v vodi, ki ga je leta 1828 opazoval botanik Robert Brown [4]. Danes ima več različnih uporabnosti, v fiziki običajno popisuje pojav naključnega gibanja drobnih delcev v tekočini, v matematičnem modeliranju pa z njim generiramo zgornjemu pojavu podobno naključno gibanje. Za nas bo zanimiva predvsem uporaba v ekonomiji, kjer se Brownovo gibanje lahko uporablja pri simulacijah nihanja trga delnic, takšno gibanje vsebuje tudi več stohastičnih modelov za simuliranje različnih ekonomskih indeksov (kot so Geometrijsko in aritmetično Brownovo gibanje, Ornstein-Uhlenbeckov proces itd.).

Brownovo gibanje je najpreprostejši stohastični proces in ga popisuje Wienerjev proces W_t, ki ga karakterizirajo naslednje štiri lastnosti:

- $W_0 = 0$
- W_t je skoraj gotovo zvezen
- W_t ima neodvisne prirastke
- prirastki $(W_t - W_s)$, kjer je $0 \leq s \leq t$, so porazdeljeni normalno s povprečjem 0 in varianco $t - s$.

Za naš primer zadostujejo do sedaj povedane lastnosti Brownovega gibanja, pri čemer bomo potrebovali predvsem neodvisnost prirastkov $(W_t - W_s)$ in njihovo porazdelitev. Bralec si lahko podrobnejšo definicijo prebere v knjigi Brownian Motion and Stochastic Calculus [4].

Na Brownovo gibanje lahko gledamo kot na slučajni sprehod z normalno porazdeljenimi prirastki [9]. Za definicijo prirastkov slučajnega sprehoda na način, ki ustreza Brownovem gibanju moramo napisati še algoritem za vzorčenje slučajnih spremenljivk iz normalne porazdelitve, saj je iz prej opisane teorije razvidno, da so tako porazdeljeni prirastki. Kode z opisom tega algoritma tukaj ne bomo dodajali, priložena pa je kot dodatek v datoteki z imenom „brownovoGibanje.eff“.

Tukaj bomo privzeli, da v Eff-u že imamo funkcijo `gauss e s`, ki vrne eno realizacijo slučajne spremenljivke porezdeljene normalno z pričakovano vrednostjo `e` in standardnim odklonom `s`.

V spodnji kodi bomo uporabili parametre s sledečim pomenom:

- `v_0` – začetno stanje delnice oz. trenutna vrednost delnice
- `s` – letni standardni odklon vrednosti delnice od začetne vrednosti
- `dt` – sprememba časa, kjer je enota leto
- `n` – število korakov slučajnega sprehoda

Najprej definiramo prestreznik za prirastke nato pa z njim prestrežemo funkcijo `slucajni_sprehod`, ter jo na ta način spremenimo v Brownovo gibanje:

[9] Pri tem prirastke ne gledamo v zveznem, temveč v diskretnem času za neko konstantno spremembo časa $dt > 0$

```
let normalni_prirastki s dt = handler
  |effect (Prirastek ()) k ->
     continue k (gauss 0.0 (s *. dt))
;;

let brownianMotion n s dt v_0 =
with normalni_prirastki s dt handle
  slucajni_sprehod v_0 n
  ;;
```

Če imamo sedaj npr. delnico s trenutno vrednostjo 5 in na preteklih podatkih ocenimo pričakovan letni standardni odklon 3, lahko simuliramo gibanje vrednosti delnice v prihodnosti z Brownovim gibanjem. Za spremembo časa vzamemo 1/252, kjer število 252 v imenovalcu predstavlja število delavnih dni v letu. To pomeni, da se bo v simulaciji vrednost delnice spreminjala na vsak delavni dan in bo torej število korakov pomenilo slučajnega sprehoda predstavljalo število delavnih dni.

Spodnja funkcija sprožimo eno realizacijo Brownovega gibanja za navedene parametre in število korakov 25:

```
brownianMotion 25 4.0 (1.0 /. 252.0) 5.0;;
```

Če bi izvedli večje število izračunov z zgornjo funkcijo, bi dobili približen interval na katerem se bo po 25 delovnih dneh domnevno nahaja vrednost delnice. Seveda Gre za napoved vrednosti z Brownovim gibanjem, ki je preprost matematičen model in ima pri napovedovanju svoje omejitve.

8 Zaključek

Strnimo še posebnosti ter prednosti programiranja s prestrezniki in algebrajskimi učinki, ki smo ga predstavljali tekom diplomske naloge.

Pri programiranju se pogosto srečujemo s primeri, v katerih lahko podoben postopek oz. enak algoritem uporabimo za več različnih izračunov, kar pomeni, da se nam naravno ponuja nedeterminističen način programiranja. Prestrezniki za algebrajske učinke, ki smo jih uporabljali v programskem jeziku Eff, nam s pisanjem kode, ki deluje zelo splošno, omogočajo lažje in hitrejše programiranje na nedeterminističen način.

S prestrezniki je spreminjanje delovanja posameznih operacij v kodi enostavno, prav tako je prestreznike za različne računske učinke mogoče pregledno gnezditi in med seboj kombinirati, kar nam omogoča hitro prilagajanje delovanja kode tako, da ta pokriva širok spekter sorodnih primerov.

S prestrezniki lahko pokrijemo večino primerov za katere se uporabljajo monade, te so sicer nekoliko splošnejše od prestreznikov, vendar pa jih je mnogo težje kombinirati med seboj. Prestrezniki nam ne omogočajo le spreminjanja izračunov kode v druge vrednosti istega tipa, temveč lahko z njimi spreminjamo tudi tipe funkcij, v katerih spremenljive operacije nastopajo. To smo pokazali s primerih zgoraj, kjer smo imeli lahko rezultate izračunov v seznamih realnih števil, v njihovih končnih porazdelitvah, ali pa pričakovanih vrednosti.

Prestrezniki in algebrajski učinki so v programiranju mnogo širši koncept, kot je bil predstavljen v tej diplomski nalogi in se ne uporabljajo le za nedeterministično

programiranje ter spreminjanje stanj v transakcijah. Zaradi prednosti, ki jih ponujajo, vzbujajo veliko pozornosti na področju razvoja programskih jezikov. Vedno več je tudi pobud o njihovi implementaciji v že obstoječe programske jezike.

Slovar strokovnih izrazov

algebraic effect algebrajski učinek
pure computation čisti izračun
stateful computation izračun stanja
handler prestreznik

Literatura

[1] Andrej Bauer, What is algebraic about algebraic effects and handlers? *ArXiv e-prints*, July 2018.

[2] Andrej Bauer in Matija Pretnar, Programming with algebraic effects and handlers, *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 2015.

[3] K.J. Hastings, *Introduction to Financial Mathematics*, Advances in Applied Mathematics. CRC Press, 2015.

[4] I. Karatzas, S. Shreve, S.E. Shreve in S.E. Shreve, *Brownian Motion and Stochastic Calculus*. Graduate Texts in Mathematics. Springer New York, 1991.

[5] Miran Lipovaca, *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA, 1st edition, 2011.

[6] Y. Minsky, *Real World Ocaml. Functional Programming for the Masses*. Shroff Publishers & Distr, 2013.

[7] Matija Pretnar in Andrej Bauer, Programski jezik Eff, [ogled 15. 7. 2018], dostopno na `https://www.eff-lang.org`.

[8] Matija Pretnar, Eff: examples. *GitHub repository*, [ogled 15. 7. 2018], dostopno na `https://github.com/matijapretnar/eff/blob/master/examples`.

[9] Matija Pretnar, An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic Notes in Theoretical Computer Science*, 319:19–35, 2015.