UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO

Matematika – 1. stopnja

Tine Makovecki

Na videz nemogoči funkcionali

Delo diplomskega seminarja

Mentor: prof. dr. Andrej Bauer

Ljubljana, 2018

Kazalo

Na videz nemogoči funkcionali

Povzetek

V delu obravnavamo računalniško implementacijo funkcionala izbire na Cantorjevi množici. Utemeljimo, da lahko predikate na Cantorjevi množici predstavimo z dvojiškimi drevesi in da nam ta predstavitev pomaga pri implementaciji funkcionala izbire. Dokažemo, da so funkcionali, ki jih obravnavamo, izračunljivi in da so drevesa, ki jih v postopku uporabimo, končna. Funkcional izbire implementiramo v programskem jeziku OCaml na različne načine, ki jih med seboj tudi primerjamo.

Seemingly impossible functionals

Abstract

In this work we discuss the implementation of the selection functional on the Cantor set. We show that continous predicates on the Cantor set can be represented with binary trees and that this representation helps implement the selection functional. It is proven that the functional we discuss is computable and the trees we use are finite. We implement the selection functional in the programming language OCaml in several ways, which we later compare.

Math. Subj. Class. (2010): 68Q05

Ključne besede: funkcional izbire, Cantorjeva množica, funkcijsko programiranje, funkcionali, drevesa, izračunljivost

Keywords: selection functional, Cantor set, functional programming, functionals, trees, computability

1. Uvod

Diplomsko delo obravnava problem računalniške implementacije funkcionala izbire na Cantorjevi množici. Pojem funkcionala izbire izvira iz logike višjega reda in se je pojavil na začetku 20. stoletja, ko so si nekateri matematiki prizadevali formalizirati logični sistem trditev in dokazov v matematiki. Imenujemo ga lahko tudi Hilbertov epsilon po matematiku Davidu Hilbertu, ki ga je vpeljal. Z implementacijo in različnimi možnostmi uporabe funkcionala se je v več člankih ukvarjal Martin Escardo. Članek [3] je služil kot osnova za moje delo, vendar se v tem delu implementacije lotimo na drugačen način. V drugem poglavju razložimo osnovne pojme o funkcionalih in natančno definiramo funkcional, ki ga želimo implementirati. V naslednjem poglavju predikate na Cantorjevi množici predstavimo z dvojiškimi drevesi. Nato zapišemo nekaj lastnosti, ki povezujejo predikate in drevesa, s katerimi jih predstavimo. Četrto poglavje je posvečeno dokazovanju, da so funkcionali, ki nas zanimajo, izračunljivi in jih torej lahko implementiramo. Definiramo pojma izračunljivosti in hereditarne totalnosti, ki ju pri dokazovanju in razlagi uporabimo. Preostanek dela obravnava implementacijo funkcionala izbire, pri kateri si pomagamo z drevesi, ki smo jih prej vpeljali. Najprej premislimo delovanje funkcionala in vpeljemo podatkovne tipe, ki jih bomo uporabili. Nato ustvarimo funkcijo za konstrukcijo dreves in premislimo načine za pregled teh dreves. Na koncu vse skupaj združimo v implementaciji funkcionala izbire. Dosežemo več različnih postopkov, ki jih potem še testiramo in primerjamo.

2. Funkcionali

Prvi razdelek je namenjen uvedbi pojmov, ki jih potrebujemo za opis zastavljenega cilja. Večina izrazov izvira iz logike, uporabljajo pa se tudi na drugih področjih. Najprej uvedimo pojem funkcionala.

Definicija 2.1. Naj bo A množica funkcij. *Funkcional* na množici A je funkcija, katere domena je množica A.

Zanimajo nas posebne vrste funkcionalov, npr. funkcionali, ki imajo za kodomeno končne množice, še posebej pa funkcionali, katerih kodomena je Boolova množica $\{0, 1\}$.

Definicija 2.2. *Predikat* na množici A je funkcija iz A v množico $\{0, 1\}$.

V nadaljevanju bomo v funkcijskih shemah za bolj pregleden zapis Boolovo množico $\{0, 1\}$ označevali z oznako $\mathbf{2}$. Vpeljimo še množico, na kateri bomo običajno preučevali funkcionale.

Definicija 2.3. *Cantorjeva množica* je množica neskončnih zaporedij ničel in enic. Označimo jo z $\mathbf{2}^{\mathbb{N}}$, torej velja $\mathbf{2}^{\mathbb{N}} = \{\alpha \mid \alpha : \mathbb{N} \to \mathbf{2}\}$.

Cantorjevo množico lahko opišemo kot množico predikatov na množici naravnih števil. Večina funkcionalov, ki jih bomo obravnavali, bo definirana na Cantorjevi množici in videli bomo, da imajo nekateri še posebej zanimive lastnosti. Domena funkcionalov na Cantorjevi množici je torej množica neskončnih zaporedij ničel in enic, njihova shemo pa zapišemo kot $\mathbf{2}^{\mathbb{N}} \to A$, kjer je A neka poljubna množica. V primeru predikatov na Cantorjevi množici je kodomena A kar enaka Boolovi množici.

V nadaljevanju elemente Boolove množice obravnavamo tudi kot resničnostne vrednosti, kjer enica označuje *resnico*, ničla pa *neresnico*. V programski kodi se za to

uporabljata oznaki `true` in `false`. Pomensko med pojmi ni prave razlike, uporabljamo tiste, ki v kontekstu omogočijo najbolj razumljiv zapis. Pravimo, da predikat p velja pri argumentu α, oz. je pri njem resničen, če ima izjava $p(\alpha)$ resnično vrednost 1, kar zapišemo kot $p(\alpha) = 1$. V nasproten primeru je predikat p pri tem argumentu neresničen in velja $p(\alpha) = 0$.

Definicija 2.4. Funkcional $\epsilon : (A \to \mathbf{2}) \to A$ je *funkcional izbire* na množici A natanko tedaj, ko velja zveza:

$$p(\epsilon(p)) = 1 \iff \exists\, \alpha \in A.\ p(\alpha) = 1$$

Povedano z besedami, funkcional izbire preslika predikat p v element, pri katerem je predikat p resničen, če tak element obstaja. Funkcional izbire ponavadi označujemo z ϵ iz zgodovinskih razlogov, imenujemo pa ga tudi Hillbertov epsilon. Osredotočimo se na funkcional izbire na Cantorjevi množici, ki ga želimo implementirati, v tem primeru je množica A kar množica $\mathbf{2}^{\mathbb{N}}$.

Opomba 2.5. Definiciji funkcionala izbire v večini primerov ustreza več različnih funkcionalov. Naš cilj je konstruirati in implementirati enega izmed njih, zato govorimo zgolj o enem funkcionalu izbire, kljub temu da jih obstaja več.

Opazimo, da lahko s funkcionalom izbire izrazimo logično kvantifikacijo. Po definiciji velja, da v Cantorjevi množici obstaja element, pri katerem je resničen predikat p, natanko tedaj, ko velja $p(\alpha)$, kjer je $\alpha = \epsilon(p)$ slika funkcionala izbire. S tem dobimo povezavo $\exists\, \beta \in \mathbf{2}^{\mathbb{N}}.\ p(\beta) \Leftrightarrow p(\epsilon(p))$. S pomočjo De Morganovega zakona lahko iz te povezave tudi logični kvantifikator $\forall$ izrazimo s funkcionalom izbire, velja zveza $(\forall \beta \in \mathbf{2}^{\mathbb{N}}.\ p(\beta)) \Leftrightarrow p(\epsilon(\neg p))$.

3. Drevesa

Želimo jasen in pregleden način predstavitve funkcionalov, ki nam bo olajšal računalniško implementacijo. Funkcionale predstavimo kot drevesa, ki jih razumemo v standardnem smislu, privzetem iz teorije grafov. V primeru predikatov na Cantorjevi množici so to dvojiška drevesa z določenim korenom.

V polnem dvojiškem drevesu ima vsako vozlišče, ki ni list, povezavi do dveh poddreves. Te povezavi označimo z 0 in 1. To nam omogoča, da pot od korena drevesa do poljubnega vozlišča v dvojiškem drevesu zapišemo kot zaporedje ničel in enic. Pot do nekega lista v (končnem) drevesu je torej oblike $p_1 \dots p_n$, kjer za vsako naravno število i manjše ali enako n velja $p_i \in \{0, 1\}$. Če je dvojiško drevo neskončno, lahko vsebuje tudi neskončne poti, ki so v našem primeru neskončna zaporedja. Vpeljimo nekaj pojmov, s katerimi lažje izražamo lastnosti takšnih poti.

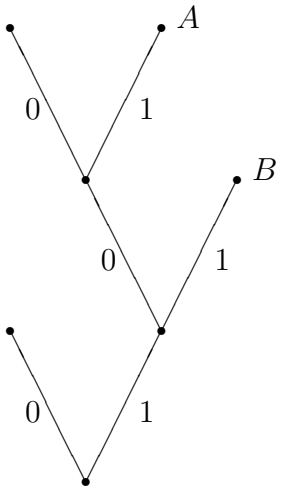
Zgled 3.1. Oglejmo si zgled zapisa poti v drevesu, ki je prikazano na sliki 1.

Pot do lista A opišemo z zaporedjem 101, pot do lista B, pa z zaporedjem 11. $\diamondsuit$

Množico končnih dvojiških zaporedji označimo z $\mathbf{2}^* = \{\alpha_1 \dots \alpha_n | n \in \mathbb{N}_0, \forall i.\ \alpha_i \in \{0, 1\}\}$. Če zaporedje a vsebuje n členov, pravimo, da je dolžine n, oz. zapišemo $|a| = n$.

Definicija 3.2. Zaporedje a je *predpona* zaporedja b, če je dolžina a manjša ali enaka dolžini b in velja $a_k = b_k$ za vsak $k \leq |a|$. To bomo označevali z $a \sqsubseteq b$. Predpono zaporedja b dolžine n označimo z $\overline{b}_n$, oz. $\overline{b}_n = b_1 b_2 \dots b_n$.

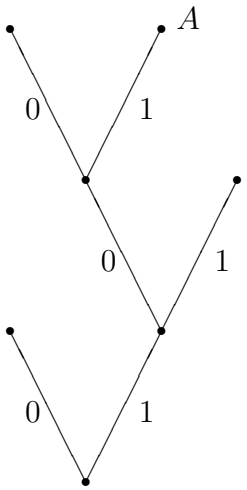
Pojem predpone iz definicije 3.2 lahko uporabljamo tudi za neskončna dvojiška zaporedja.



Slika 1. Primer drevesa za dvojiški zapis poti.

Definicija 3.3. Naj bo T dvojiško drevo in L list v drevesu T. Naj bo p pot od korena drevesa T do lista L. Dvojiško zaporedje α *pripada* listu L, če je zaporedje p predpona zaporedja α.

Zgled 3.4. Za zgled premislimo, katera zaporedja pripadajo listu A iz slike 2.



Slika 2. Primer drevesa za zaporedja, ki pripadajo listu A.

Iz zgleda 3.1 vemo, da je pot do lista A oblike $s = 101$. Pot s je predpona vsakega elementa množice, ki jo iščemo. Vsa zaporedja, ki pripadajo listu A, so torej $\{\alpha \mid \alpha \in \mathbf{2}^{\mathbb{N}} \cup \mathbf{2}^*; 101 \sqsubseteq \alpha\}$. Na nekoliko drugačen način bi lahko množico zapisali tudi kot $\{\alpha \mid \alpha \in \mathbf{2}^{\mathbb{N}} \cup 2^*; 101 = \overline{\alpha}_3\}$. Elementi te množice pa so npr. 1010, 10111, ... ♢

Po definiciji [7, poglavje 10] v drevesu med poljubnima dvema vozliščema obstaja natanko ena pot. Sledi, da v dvojiškem drevesu z določenim korenom tudi od korena do poljubnega lista vodi natanko ena pot. V nadaljevanju vsak list dvojiškega

drevesa označimo z ničlo ali enico. Označimo množico takšnih končnih označenih dvojiških dreves s $\mathbf{T}_2$. Definirajmo povezavo med predikati in drevesi.

Definicija 3.5. Naj bo $T \in \mathbf{T}_2$ dvojiško drevo in naj bo $\Phi : \mathbf{T}_2 \to (\mathbf{2}^{\mathbb{N}} \to \mathbf{2})$ preslikava, ki ustreza predpisu:

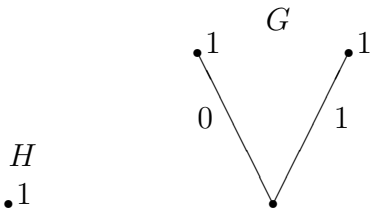
$$\Phi(T)(\alpha) = \begin{cases} 1 & ; \text{ če zaporedje } \alpha \text{ pripada listu z vrednostjo 1} \\ 0 & ; \text{ če zaporedje } \alpha \text{ pripada listu z vrednostjo 0} \end{cases}$$

V nadaljevanju sliko preslikave Φ pri argumentu T označujemo s Φ_T in ne $\Phi(T)$. Pravimo, da drevo T *predstavlja* funkcional p, če velja $\Phi_T = p$.

Pojem predstavitve smo definirali za dvojiška drevesa in predikate na Cantorjevi množici, lahko pa bi ga posplošili še na precej drugih primerov. Če bi množico predikatov na Cantorjevi množici zamenjali s katero drugo množico funkcionalov na Cantorjevi množici, ki imajo za kodomeno neko množico A, bi se spremenile vrednosti v listih dreves. V množici dvojiških dreves $\mathbf{T}_2$ bi bile vrednosti v listih elementi množice A namesto elementov Boolove množice. Če bi množico predikatov na Cantorjevi množici zamenjali z množico predikatov na neki drugi množici zaporedij, bi se spremenile razvejitve drevesa. Recimo, da smo Cantorjevo množico zamenjali z množico $B^{\mathbb{N}}$. Drevesa ne bi bila več dvojiška, temveč bi se razvejila na toliko poddreves, kolikor možnih vrednosti ima vsak člen zaporedja, ki je element množice B. Teh posplošitev predstavitve ne bi bilo težko formalizirati, vendar jih zaenkrat ne bomo potrebovali.

Kadar govorimo o drevesu, kot da bi bilo predikat, to seveda pomeni, da smo ga s preslikavo Φ implicitno pretvorili v predikat. Zanima nas, ali obstaja inverz preslikave Φ. Najprej preverimo, ali je preslikava Φ injektivna, oz. ali lahko različni drevesi predstavljata isti funkcional.

Zgled 3.6. Oglejmo si drevesi na sliki 3.



Slika 3. Drevesi, ki predstavljata isti funkcional.

Kljub temu, da sta drevesi H in G različni, predstavljata isti funkcional. Drevo H predstavlja kostanten funkcional, oz. predikat, ki vse argumente preslika v 1. Funkcional, ki ga predstavlja drevo G, loči zaporedja glede na prvi člen in slika zaporedja, ki se začnejo z 1, v 1 in zaporedja, ki se začnejo z 0, prav tako v 1. Sledi, da je predikat, ki ga predstavlja drevo G, prav tako konstanten in ima sliko 1. $\diamondsuit$

Ker lahko več različnih dreves predstavlja isti predikat, preslikava Φ ni injektivna. Če bi levi inverz preslikave Φ obstajal, bi bila preslikava injektivna. Sledi, da Φ nima levega inverza in posledično tudi njegov inverz ne obstaja. Zanima nas, ali obstaja vsaj desni inverz.

Desni inverz lahko obstaja zgolj za tiste predikate, ki imajo predstavitev z drevesom. Preden se posvetimo obstoju desnega inverza torej želimo ugotoviti, katere

funkcionale lahko predstavimo z drevesi iz $\mathbf{T}_2$. Ugotovimo, da se ne da predstaviti vseh funkcionalov, izkaže pa se, da drevesa iz $\mathbf{T}_2$ predstavljajo natanko zvezne funkcionale. Za dokaz najprej potrebujemo definicijo zveznosti funkcionalov.

Definicija 3.7. Naj bo $d : \mathbf{2}^{\mathbb{N}} \times \mathbf{2}^{\mathbb{N}} \to \mathbb{R}$ preslikava, definirana s predpisom:

$$d(\alpha, \beta) = \begin{cases} 2^{-k} & ; \text{ če je } \overline{\alpha}_{k-1} = \overline{\beta}_{k-1} \text{ in } \alpha_k \neq \beta_k \\ 0 & ; \text{ če je } \alpha = \beta \end{cases}$$

Funkcija d paru zaporedij priredi število glede na to, v koliko mestih se ujemata. Mesto k v definiciji je prvo mesto, pri katerem se zaporedji α in β razlikujeta. Osnovne pojme in definicije povezane z metričnimi prostori je možno najti v knjigi [8].

Trditev 3.8. *Funkcija d je metrika na Cantorjevi množici.*

Dokaz. Po definiciji d velja $d(\alpha, \alpha) = 0$ in za vsako naravno število k velja $2^{-k} > 0$. Sledi, da je $d(\alpha, \beta)$ vedno večje ali enako nič ter da je enako nič natanko tedaj, ko sta zaporedji α in β enaki. Očitno je, da za poljubni zaporedji α in β velja $d(\alpha, \beta) = d(\beta, \alpha)$, torej je funkcija d simetrična.

Pokazati moramo še, da za funkcijo d velja trikotniška neenakost. Želimo, da za poljubno trojico zaporedij α, β in γ velja: $d(\alpha, \gamma) \leq d(\alpha, \beta) + d(\beta, \gamma)$. Denimo, da se α in γ prvič razlikujeta na mestu l, α in β na mestu m ter β in γ na mestu n. Če so vsa tri zaporedja enaka, je vrednost $\min(m, n)$ nedefinirana, vendar v tem primeru trikotniška neenakost očitno velja. Sicer velja, da se α in β ujemata v prvih $\min(n, m) - 1$ mestih, prav tako β in γ, torej se tudi α in γ ujemata v prvih $\min(n, m) - 1$ mestih. Sledi $\min(n, m) \leq l$, torej velja trikotniška neenakost $2^{-l} \leq 2^{-m} + 2^{-n}$. □

S tem, da smo Cantorjevi množici dodali metriko, smo dobili metrični prostor. Cantorjevi množico lahko opazujemo tudi kot topološki prostor, kar nam bo v nadaljevanju koristilo. Osnovni podatki o topologiji so za ponovitev na voljo v knjigi [6]. Vemo, da metrika porodi metrično toplogijo na Cantorjevi množici. To topologijo generiramo z bazo odprtih krogel. Oglejmo si odprte krogle s polmerom $r \in [2^{-m}, 2^{-m-1})$. Vemo, da velja $d(\alpha, \beta) < r$ natanko tedaj, ko imata zaporedji α in β prvih m členov enakih. Torej je krogla s takšnim polmerom odvisna zgolj od prvih m členov središča. Obstaja toliko različnih krogel s polmerom r, kolikor je možnih kombinacij prvih m členov. Z $B_{b_1,\ldots,b_n}$ označujemo kroglo $\{\beta \in \mathbf{2}^{\mathbb{N}} \ ; \ \forall m \leq n \ . \ \beta(m) = b_m\}$. Očitno je, da lahko s tem zapisom predstavimo vse odprte krogle, torej celo bazo metrične topologije.

Spomnimo se, da je produktna topologija na Cantorjevi množici tista topologija, ki jo generira podbaza $\{S_{i,b} \ ; \ i \in \mathbb{N}, b \in \mathbf{2}, S_{i,b} = \{p \in \mathbf{2}^{\mathbb{N}}; p(i) = b\}\}$.

Trditev 3.9. *Metrična topologija in produktna topologija na Cantorjevi množici sta enaki.*

Dokaz. Pokazati moramo, da je množica element metrične topologije natanko tedaj, ko je element produktne topologije. Opazimo, da sta množica $S_{1,b_1} \cap S_{2,b_2}$ in odprta krogla B_{b_1,b_2} enaki. Podobno idejo lahko uporabimo za poljubno odprto kroglo v metriki d; člene, ki so enaki pri vseh elementih odprte krogle, lahko določimo s preseki elementov podbaze. Za odprto kroglo $B_{b_1,b_2,\ldots,b_n}$ velja enakost $B_{b_1,b_2,\ldots,b_n} = S_{1,b_1} \cap S_{2,b_2} \cap \cdots \cap S_{n,b_n}$. Vsak element metrične topologije je unija

odprtih krogel v metriki d in vsaka takšna krogla je enaka končnemu preseku elementov podbaze produktne topologije. Torej je vsak element metrične topologije tudi element produktne topologije.

Naj bo $P = S_{i_1,b_1} \cap S_{i_2,b_2} \cap \cdots \cap S_{i_n,b_n}$ končen presek elementov podbaze produktne topologije. Brez škode za splošnost lahko privzamemo, da za vsak j velja $i_j \leq i_{j+1}$. Trdimo, da je množica P enaka uniji odprtih krogel. Elementi množice P so zaporedja, ki imajo pri indeksu i_j vrednost b_j. Zapisati moramo unijo, kjer bodo vsi elementi pri indeksih i_j imeli ustrezno vrednost, v vseh ostalih členih pa njihova vrednost ne bo fiksirana. To dosežemo tako, da za vsak člen, ki ni določen, v unijo dodamo zaporedja, ki imajo pri tem členu vrednost 0, in tudi zaporedja z vrednostjo 1. Velja enačba $P = \cup\{B_{d_1,d_2,\ldots,d_{i_n}} \ ; \ d_{i_k} = b_k \ \wedge \ (j \leq n \wedge j \notin \{i_1, \ldots, i_n\} \Rightarrow d_j \in \{0,1\})\}$. Torej je vsak element podbaze metrične topologije unija baznih elementov produktne topologije. Sledi, da lahko vsako množico, ki je element metrične topologije, predstavimo s končnimi preseki in unijami baznih množic produktne topologije. Topologiji sta torej enaki. □

Če opremimo še Boolovo množico z diskretno metriko za katero velja $d(0,1) = 1$, lahko opazujemo zveznost predikatov.

Trditev 3.10. *Predikat $p : \mathbf{2}^{\mathbb{N}} \to \mathbf{2}$ je zvezen v $\alpha \in \mathbf{2}^{\mathbb{N}}$ natanko tedaj, ko obstaja tako naravno število $n \in \mathbb{N}$, da za vsak $\beta \in \mathbf{2}^{\mathbb{N}}$ velja:*

$$\overline{\alpha}_n = \overline{\beta}_n \Rightarrow p(\beta) = p(\alpha)$$

Dokaz. Predpostavimo, da je predikat p zvezen v α. Po definiciji zveznosti za vsak pozitiven ϵ obstaja tako pozitivno število δ, da za poljuben β iz Cantorjeve množice iz $d(\alpha,\beta) < \delta$ sledi $d(p(\alpha), p(\beta)) < \epsilon$. Naj bo $\epsilon = 1$. Po definiciji metrike sta vrednosti $p(\alpha)$ in $p(\beta)$ enaki, če je razdalja med njima manjša od 1. Naj bo $\beta \in \mathbf{2}^N$ poljubno zaporedje in m prvo mesto, v katerem se α in β razlikujeta. Zaradi zveznosti predikata p obstaja tak δ, da velja $p(\alpha) = p(\beta)$, če velja $2^{-m} < \delta$. Ta neenakost drži za vse dovolj velike m. Torej obstaja tako naravno število n, da za vsak $m \geq n$ velja $2^{-m} < \delta$. Če velja $\overline{\alpha}_n = \overline{\beta}_n$, velja $d(\alpha,\beta) < 2^{-n} < \delta$ in posledično $p(\alpha) = p(\beta)$. S tem je prvi del ekvivalence dokazan.

Pokažimo še obratno, torej da je predikat p v α zvezen. Po defenciji metrike je za poljubno zaporedje β razdalja med $p(\alpha)$ in $p(\beta)$ manjša ali enaka 1. Za $\epsilon > 1$ pogoj za zveznost očitno velja. Naj bo ϵ manjši ali enak 1. Po predpostavki obstaja tako naravno število n, da iz $\overline{\alpha}_n = \overline{\beta}_n$ sledi $p(\beta) = p(\alpha)$. Če je $d(\alpha,\beta)$ manjše od 2^{-n}, velja $\overline{\alpha}_n = \overline{\beta}_n$. V tem primeru sledi $p(\beta) = p(\alpha)$ in je razdalja med $p(\alpha)$ in $p(\beta)$ manjša od poljubnega pozitivnega ϵ. Za δ, ki izpolni definicijo zveznosti, smo vzeli 2^{-n}. S tem je ekvivalenca dokazana. □

Predikat p je zvezen v točki α, če je njegova slika določena že s predpono $\overline{\alpha}_n$, torej s prvimi n členi zaporedja α. Predikat je zvezen, če je zvezen v vseh elementih Cantorjeve množice. Preden lahko utemeljimo, kateri funkcionali so predstavljeni s končnimi drevesi, se moramo bolj podrobno posvetiti drevesom samim.

Lema 3.11 (Königova lema)**.** *Dvojiško drevo je neskončno natanko tedaj, ko vsebuje neskončno pot.*

Dokaz. Očitno je, da je drevo, ki vsebuje neskončno pot, tudi samo neskončno.

Dokazati moramo, da vsako neskončno dvojiško drevo vsebuje neskončno pot. Naj bo T neskončno dvojiško drevo. Množice poddreves bomo označevali s $T_a = \{b \mid a \sqsubseteq b, b \in T\}$, kjer je a neko končno dvojiško zaporedje, vsebovano v drevesu T.

Velja enakost $T = T_1 \cup T_0$ in množica T je neskončna. Sledi, da je vsaj ena izmed podmnožic T_1 in T_0 neskončna. Prvi člen poti določimo po naslednjem predpisu:

$$\alpha_1 = \begin{cases} 1 & ; \text{ če je množica } T_1 \text{ neskončna} \\ 0 & ; \text{ sicer} \end{cases}$$

Postopek induktivno nadaljujemo. Naj bo konstruiranih prvih $n-1$ členov poti in množica $T_{\alpha_1 \dots \alpha_{n-1}}$ naj bo neskončna. Sledi, da je neskončna vsaj ena izmed podmnožic $T_{\alpha_1 \dots \alpha_{n-1}1}$ in $T_{\alpha_1 \dots \alpha_{n-1}0}$. Naslednji člen poti določimo s predpisom:

$$\alpha_n = \begin{cases} 1 & ; \text{ če je množica } T_{\alpha_1 \dots \alpha_{n-1}1} \text{ neskončna} \\ 0 & ; \text{ sicer} \end{cases}$$

Ker je poddrevo $T_{\alpha_1 \dots \alpha_n}$ neskončno za poljuben n, lahko postopek na vsakem koraku nadaljujemo in tako konstruiramo neskončno zaporedje. Lema je s tem dokazana. □

S tem imamo vsa orodja, ki jih potrebujemo za dokaz naslednjega izreka.

Izrek 3.12. *Predikat p na Cantorjevi množici je zvezen natanko tedaj, ko obstaja tak $T \in \boldsymbol{T}_2$, da velja $p = \Phi_T$.*

Dokaz. Privzemimo, da obstaja tak $T \in \mathbf{T}_2$, da velja $p = \Phi_T$. Dvojiško drevo T je končno, torej po Königovi lemi ne vsebuje neskončne poti. Z L označimo list, ki ima v drevesu T maksimalno globino in pot do njega z $\overline{\alpha}_n = \alpha_1 \alpha_2 \dots \alpha_n$. Vemo, da vsa zaporedja, ki imajo predpono $\overline{\alpha}_n$, pripadajo listu L in jih posledično predikat p slika v vrednost, s katero je označen list L. Predikat p je po trditvi 3.10 zvezen v vseh zaporedjih, ki pripadajo listu L, saj je slika teh zaporedij odvisna zgolj od prvih n členov.

Vemo, da je α_n od vseh poti v drevesu, ki se začnejo v korenu, najdaljša. Vsi listi drevesa so od korena oddaljeni kvečjemu n povezav. Poljubno zaporedje mora pripadati nekemu listu, ki določa njegovo sliko. Sledi, da je slika poljubnega zaporedja določena s prvimi n (ali manj) členi zaporedja. Če se dve zaporedji ujemata v predponi dolžine n, se torej zagotovo preslikata v enako vrednost. Po trditvi 3.10, je predikat p zvezen.

Dokažimo ekvivalenco še v drugo smer. Naj bo predikat p zvezen. Želimo dokazati, da obstaja končno drevo, ki ga predstavlja. Najprej premislimo primer, v katerem je p konstanten predikat. Potem lahko p predstavimo z drevesom, ki vsebuje zgolj koren, ki je označen z vrednostjo, v katero p preslika vse argumente. Takšno drevo je končno.

Naj bo sedaj p predikat, ki ni konstanten. Vemo, da drevo, ki ga predstavlja, ne sme vsebovati neskončne poti. Naj bo α element Cantorjeve množice. To zaporedje mora pripadati nekemu listu drevesa. Da je vozlišče, do katerega vodi pot $\overline{\alpha}_n$, list, mora veljati:

$$\forall \beta \ \in \mathbf{2}^{\mathbb{N}}. \ (\overline{\alpha}_n \sqsubseteq \beta \Rightarrow p(\alpha) = p(\beta))$$
$$\exists \gamma, \delta \in \mathbf{2}^{\mathbb{N}} \ . \ (\overline{\alpha}_{n-1} \sqsubseteq \gamma, \delta \wedge p(\gamma) \neq p(\delta))$$

Utemeljiti želimo, da vsako zaporedje res pripada nekemu takšnemu vozlišču. Vpeljimo funkcijo $F : \mathbf{2}^{\mathbb{N}} \to \mathbb{N}$, ki zaporedje α preslika v najmanjše število n, za katerega velja:

$$\forall \beta \ \in \mathbf{2}^{\mathbb{N}}. \ (\overline{\alpha}_n = \overline{\beta}_n \Rightarrow p(\alpha) = p(\beta))$$

Ker je predikat p zvezen, po trditvi 3.10 tak n obstaja za vsak element Cantorjeve množice. Trdimo, da je funkcija F zvezna glede na diskretno topologijo na $\mathbb{N}$ in metrično topologijo na $\mathbf{2}^{\mathbb{N}}$.

Funkcija F je zvezna, če je za poljubno število $n \in \mathbb{N}$ praslika $F^{-1}(n)$ odprta. Trdimo, da iz $\alpha \in F^{-1}(n)$ sledi, da je vsak $\beta \in \mathbf{2}^{\mathbb{N}}$, ki se z α ujema v predponi dolžine n, prav tako element praslike $F^{-1}(n)$. Denimo, da je $F(\alpha) = n$ in obstaja β, za katerega velja $F(\beta) < n$ in $\overline{\alpha}_n = \overline{\beta}_n$. Iz $\overline{\alpha}_n = \overline{\beta}_n$ sledi $p(\alpha) = p(\beta)$. Ker je po definiciji preslikave F število n najmanjše, za katerega velja ta implikacija, obstaja zaporedje γ, za katerega velja $\overline{\alpha}_{n-1} = \overline{\gamma}_{n-1} = \overline{\beta}_{n-1}$ in $p(\alpha) \neq p(\gamma)$. Ker velja $F(\beta) \leq n-1$, in se γ z β ujema v $n-1$ členih, velja $p(\gamma) = p(\beta) = p(\alpha)$. To je protislovje. Analogno pokažemo, da ne more obstajati β, za katerega velja $F(\beta) > n$ in $\overline{\alpha}_n = \overline{\beta}_n$. Po definiciji funkcije F v tem primeru obstaja zaporedje γ, za katerega velja $\overline{\beta}_n = \overline{\gamma}_n$ in $p(\beta) \neq p(\gamma)$. Posledično velja $\overline{\beta}_n = \overline{\alpha}_n = \overline{\gamma}_n$ in $p(\beta) = p(\alpha) \neq p(\gamma)$, kar je v nasprotju z $F(\alpha) = n$.

Od tod sklepamo, da iz $F(\alpha) = n$ sledi $B_{\alpha(1),\ldots,\alpha(n)} \subseteq F^{-1}(n)$. Praslika je torej enaka uniji odprtih krogel:

$$F^{-1}(n) = \cup\{B_{b_1,\ldots,b_n} \ ; \ \exists \alpha \in \mathbf{2}^{\mathbb{N}} \ . \ (F(\alpha) = n \wedge b_1 \ldots b_n = \overline{\alpha}_n)\}$$

Sledi, da je praslika $F^{-1}(n)$ odprta za poljubno naravno število n.

Trditev 3.9 nam pove, da sta metrična in produktna topologija na $\mathbf{2}^{\mathbb{N}}$ ekvivalentni. Prostor lahko torej gledamo tudi s produktno topologijo. Po izreku Tihonova je $\mathbf{2}^{\mathbb{N}}$ kot produkt kompaktnih prostorov prav tako kompakten prostor. Dokaz izreka Tihonova se nahaja v knjigi [4, str. 11–12]. Ker je F zvezna preslikava, ki slika iz kompaktnega prostora, vemo, da doseže maksimum. Označimo točko, pri kateri je maksimum dosežen, z α.

Naj bo $T \in \mathbf{T}_2$ drevo, ki predstavlja predikat p. Listi drevesa T so vozlišča do katerih vodi takšna pot $s = s_1 s_2 \ldots s_k$, da je predikat p na množici $B_{s_1,\ldots,s_k}$ konstanten in na množici $B_{s_1,\ldots,s_{k-1}}$ ni konstanten. Ostala vozlišča drevesa T pa so tista, do katerih vodi takšna pot $s = s_1 s_2 \ldots s_k$, da predikat na množici $B_{s_1,\ldots,s_k}$ ni konstanten. Trdimo, da je drevo T končno. List, ki mu v drevesu T pripada α, je tisto vozlišče, ki je najbolj oddaljeno od korena. Pot, ki vodi do njega, pa je dolžine $F(\alpha)$. Ker drevo T ne premore neskončne poti, je končno. Sledi, da je predikat p res predstavljen s končnim drevesom iz množice $\mathbf{T}_2$. □

Pokazali smo, da so zvezni predikati predstavljivi z drevesi. Skrčitev preslikave Φ na zvezne predikate je torej surjektivna. V tem primeru lahko definiramo desni inverz.

Trditev 3.13. *Obstaja takšna preslikava $\Psi : \mathcal{C}(\mathbf{2}^{\mathbb{N}}, \mathbf{2}) \to \mathbf{T}_2$, da za vsak zvezen predikat p na Cantorjevi množici velja $\Phi(\Psi(p)) = p$.*

Dokaz. Po izreku 3.12 za vsak zvezen predikat p obstaja drevo $T \in \mathbf{T}_2$, ki ga predstavlja. Naj bo Ψ preslikava, ki zvezen predikat preslika v drevo, ki ga predstavlja. Če je takšnih dreves več, za sliko izberemo enega izmed njih z aksiomom izbire. Po definiciji preslikave Φ in predstavitve velja $\Phi(\Psi(p)) = \Phi_{\Psi(p)} = p$. □

Nismo določili, katero drevo je slika preslikave Ψ, kadar obstaja več možnosti. Izbira se bo določila pri implementaciji Ψ. Preden se posvetimo implementaciji, želimo utemeljiti izračunljivost funkcionalov, ki jih bomo implementirali.

Pokazali bomo, da je funkcional izbire izračunljiv. S tem bomo utemeljili, da je implementacija smiselna. Zanima nas tudi, če obstaja pogoj, ki omejuje, katere funkcionale je vredno implementirati.

4.1. **Osnove izračunljivosti.** Razdelek je namenjen osnovim pojmom iz področja teorije izračunljivosti. Potrebujemo jih, da lahko problem pravilno zastavimo.

Funkcije so preslikave, ki vsakemu elementu domene priredijo sliko v kodomeni. *Delne funkcije* so preslikave, ki niso nujno definirane za vsak element množice, iz katere slikajo. Ta pojem bomo potrebovali pri govoru o izračunljivih funkcijah, saj lahko algoritem naleti na napako v izvajanju in rezultat posledično ni definiran, ali pa pri nekem argumentu postopek preprosto ni smiseln in se algoritem izvaja v neskončnost.

Opomba 4.1. Vsaka povsod definirana funkcija je tudi delna funkcija. Delni funkciji, ki je definirana povsod, pravimo *totalna*.

Definirati moramo, kaj točno pomeni, da je funkcija izračunljiva. Uporabljali bomo standardno definicijo, ki se nanaša na *Turingove stroje*. Podrobno definicijo Turingovih strojev je možno najti v [2, poglavje 1.2.1], v knjigi pa je na voljo tudi bolj podroben uvod v teorijo izračunljivosti.

Definicija 4.2. Delna funkcija f je *Turingovo izračunljiva*, če obstaja takšen Turingov stroj T, da za vsak x velja:

- $T(x)$ račun konča in vrne rezultat natanko tedaj, ko je vrednost $f(x)$ definirana;
- če je $f(x)$ definirana, potem $T(x)$ izračuna vrednost $f(x)$.

Opomba 4.3. Da Turingov stroj vrednost "izračuna", pomeni, da zapiše rezultat in se ustavi.

Če definicijo poenostavimo, lahko rečemo, da je Turingov stroj naprava, ki izpolnjuje končno zaporedje preprostih ukazov. Izračunljivost je lastnost funkcije, da lahko poiščemo rezultat z izvajanjem zaporedja enostavnih ukazov v končnem času. Ukazov je lahko zelo veliko, čas izvajanja pa tako velik, da je v praksi neizvedljiv, vendar se zadovoljimo s tem, da je rezultat možno izračunati. Poudarimo, da lahko dokažemo izračunljivost funkcije, ne da bi poznali dejanski postopek za njen izračun. Izračunljivost nam pove zgolj to, da tak postopek zagotovo obstaja.

Namesto Turingovih strojev lahko obravnavamo programe v nekem splošnem programskem jeziku. Pojem izračunljivosti bomo zato v nadaljevanju uporabljali ne da bi se sklicevali na Turingove stroje. Želimo, da funkcional izbire išče med elementi neskončne množice in vrne rezultat v končnem času. Izračunljivost takšnega funkcionala je treba utemeljiti. Ugotovili smo, da zvezne predikate lahko predstavimo s končnimi drevesi, kar bi bilo pri implementaciji funkcionala zelo koristno. Zanima nas, ali lahko predstavitev z drevesi izkoristimo.

Lema 4.4. *Vsi izračunljivi funkcionali na Cantorjevi množici so zvezni.*

Dokaz leme se nahaja v knjigi [9, str. 30].

4.2. **Hereditarna totalnost.** Delne funkcije in funkcionale želimo uskladiti z matematičnimi funkcijami, da bomo lahko pri implementaciji uprabili sklepe iz poglavja 3. V tem razdelku uvedemo pojem *hereditarne totalnosti*, s katerim si lahko pri

tem pomagamo. Pri programiranju delamo z izračunljivmi delnimi funkcijami. Ko implementiramo funkcional, dobimo delno funkcijo, ki slika delne funkcije v rezultate. Zanimajo nas tisti funkcionali, ki so definirani vsaj na vseh totalnih funkcijah. Pravimo, da so taki funkcionali hereditarno totalni.

Opomba 4.5. Totalen funkcional je funkcional, ki je definiran na vseh delnih funkcijah. Če je tak funkcional izračunljiv, mora prirediti rezultat tudi funkciji, ki je povsod nedefinirana. Sledi, da je rezultat neodvisen od tega, kakšen je argument, torej je izračunljiv totalen funkcional konstanten.

Definicija 4.6. Predikat na Cantorjevi množici je *hereditarno totalen* natanko tedaj, ko je definiran za vsako totalno funkcijo.

Totalna funkcija iz Cantorjeve množice vsakemu naravnemu številu priredi neko Boolovo vrednost. Razumemo jo torej lahko kot matematično zaporedje. Hereditarno totalen predikat na Cantorjevi množici za vsako dvojiško zaporeje izračuna neko Boolovo vrednost. Videli bomo, da je hereditarna totalnost dovolj strog pogoj, da lahko uporabimo lastnosti, ki smo jih izpeljali v poglavju 3.

Hereditarna totalnost ne zagotavlja izračunljivosti. To pokažemo s protiprimerom. Naj bo p predikat na Cantorjevi množici s sledečim predpisom:

$$p(\alpha) = \begin{cases} 1 & ; \text{ če } \forall n \in \mathbb{N}.\ \alpha(n) = 0 \\ 0 & ; \text{ sicer} \end{cases}$$

Predikat p je hereditarno totalen. Da bi se pri nekem argumentu izračunal rezultat predikata, pa bi program moral preveriti neskončno mnogo členov argumenta, česar seveda ne more storiti v končnem času. Torej p ni izračunljiv.

4.3. **Izračunljivost funkcionalov.** Želimo združiti vse prejšne premisleke in utemeljiti, da je funkcional izbire izračunljiv ter je implementacija smiselna. Predstavitev hereditarno totalnih predikatov z drevesi razumemo kot običajno predstavitev predikatov z drevesi. Moramo pa se zavedati, da je ta predstavitev lahko pomanjkljiva, če hereditarno totalnemu predikatu podamo delno funkcijo. V argumentu, kjer ta funkcija ni definirana, ni mogoče preveriti vrednosti. Posledično se lahko zgodi, da ni možno ugotoviti, kateremu listu drevesa bi pripadalo delno zaporedje, in je predikat pri tem argumentu nedefiniran. To bi lahko razumeli kot da so nekateri deli drevesa lahko nedefinirani. V praksi se bomo temu izognili tako, da bomo delali zgolj s totalnimi zaporedji.

Trditev 4.7. *Hereditarno totalni izračunljivi predikati na Cantorjevi množici so predstavljeni s končnimi drevesi.*

Dokaz. Izračunljivi hereditarno totalni predikati na Cantorjevi množici so po lemi 4.4 zvezni. Upoštevamo še izrek 3.12 in vidimo, da lahko vse hereditarno totalne predikate na Cantorjevi množici res predstavimo s končnimi dvojiškimi drevesi. □

Hereditarno totalne izračunljive predikate lahko z implementacijo funkcije Ψ predstavimo s končnim drevesi. Funkcija Ψ slika zvezne predikate, katerih slika je odvisna od končno mnogo členov, v končna drevesa, torej se zdi smiselno, da je izračunljiva. Z implementacijo pa bomo pokazali, da je to res. V končnih drevesih lahko v končnem času poiščemo rezultat funkcionala izbire. Sledi, da je funkcional izbire res izračunljiv, definiran pa je na množici hereditarno totalnih izračunljivih predikatov.

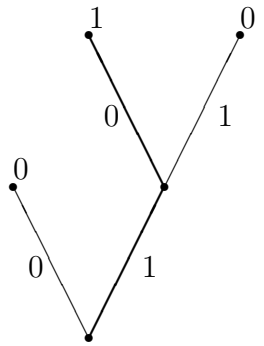
5. Implementacija

V tem poglavju se bomo posvetilil implementaciji. Vemo, da lahko predikate na Cantorjevi množici predstavimo z dvojiškimi drevesi. Videli bomo, da lahko s pomočjo tega poenostavimo postopek delovanja funkcionala izbire. Premislimo, kako bi funkcional izbire deloval, če predikat zamenjamo z drevesom, ki ga predstavlja.

Naj bo p predikat na Cantorjevi množici in T drevo, ki ga predstavlja. Slika funkcionala izbire na Cantorjevi množici je zaporedje α, ki v drevesu T pripada listu z oznako 1, če tak list obstaja. Za določitev $\epsilon(p)$ poiščemo primeren list in izberemo enega izmed zaporedij, ki mu pripadajo. Poljubno lahko izberemo, katero pripadajoče zaporedje določimo kot sliko. Če v drevesu ne obstaja list z vrednostjo 1, je slika poljuben element Cantorjeve množice. Problem izvajanja funkcionala izbire se tako prevede na problem pregledovanja listov drevesa, kar implementacijo močno poenostavi.

Postopek delovanja funkcionala bo takšen, da bo algoritem predikat, ki bo argument, najprej preslikal v drevo. Nato bo pregledal liste dobljenega drevesa. Če bo algoritem našel list z oznako 1, si bo moral zapomniti pot do tega lista. Ta pot bo predpona zaporedja, ki ga bo vrnil algoritem. Preostala mesta zaporedja bodo lahko določena poljubno, npr. tako, da se zapolnijo s samimi enicami. Če algoritem ne bo našel lista z oznako 1, bo za sliko določil poljubno zaporedje.

Primer 5.1. Oglejmo si, kako bi tak funkcional izbire deloval na primeru. Imamo drevo T iz množice $\mathbf{T}_2$ in želimo ugotoviti, kakšno zaporedje bi vrnil funkcional izbire.



Slika 4. Drevo za primer delovanja funkcionala izbire.

V drevesu najdemo le en list z vrednostjo 1. Pot do tega lista je $p = 10$. Vemo, da je odgovor določen s potjo p, preostanek zaporedja pa je lahko poljuben. Določimo torej, da funkcional izbire vrne zaporedje $\alpha = 10111\ldots$ Drugače rečeno, funkcional izbire vrne zaporedje α, za katerega velja:

$$\alpha(n) = \begin{cases} 0 & ;\ n = 2 \\ 1 & ;\ \text{sicer} \end{cases}$$

Vse člene zaporedja, ki jim lahko poljubno izberemo vrednost, smo določili za 1. $\diamondsuit$

Za delovanje funkcionala izbire bomo morali sprogramirati funkcional $\Psi : (\mathbf{2}^{\mathbb{N}} \to \mathbf{2}) \to \mathbf{T}_2$, s katerim bomo lahko poiskali predstavitev predikata z drevesom. Potrebujemo tudi algoritem, ki bo pregledal liste drevesa in iskal list z oznako 1. To bomo

morali oviti v funkcijo, ki bo kot argument prejela predikat na Cantorjevi množici, ga preslikala v drevo, nato pa v drevesu poiskala list in iz podatkov o listu določila ter vrnila primerno zaporedje. Preden lahko algoritme zares zapišemo, moramo določiti, s katerimi podatkovnimi tipi bomo predstavili podatke.

Uporabljali bomo programski jezik *OCaml*. To je funkcijski jezik, ki omogoča tudi imperativno programiranje. Za zapis funkcijskih shem ne bomo več uporabljali matematičnega zapisa, temveč sintakso, prevzeto iz OCamla. Oznake standardnih podatkovnih tipov v OCamlu so dovolj znane, da pomen funkcij ostane jasen. Nekatere podatkovne tipe bomo morali uvesti sami. Podatkovni tip naravnih števil bomo označili z `nat`, kasneje pa bomo ustvarili tudi podatkovni tip dreves. Nekatere funkcije, ki bodo nastopale, ne bodo popolnoma ustrezale standardnemu zapisu algoritmov, in bodo zaradi tega prav tako zapisane v OCaml kodi. Če kakšen del programske kode bralcu ne bo razumljiv, ali pa bo želel dodatno razlago o elementih jezika, so podrobni podatki o sintaksi in gradivo za učenje na voljo v priročniku [5, poglavja 1–8].

Primer 5.2. Podatkovni tip naravnih števil označujemo z `nat`, Boolovo množico pa z `bool`. Če je α element Cantorjeve množice, potem je podatkovnega tipa α : `nat` $\rightarrow$ `bool`. Podatkovni tip predikata `p` na Cantorjevi množici lahko potem zapišemo kot `p: (nat → bool) → bool`. ♢

5.1. **Implementacija dreves.** Implementirati želimo podatkovni tip dvojiškega drevesa, ki bo predstavljal predikate na Cantorjevi množici. Oglejmo si sledečo definicijo podatkovnega tipa.

```
type tree =
  | Answer of bool
  | Question of nat * tree * tree
```

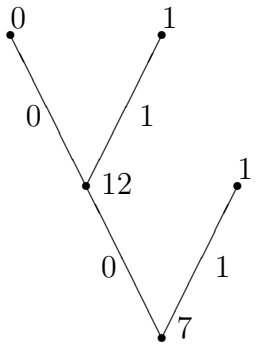
Pove nam, da je drevo bodisi t. i. "odgovor", ki vsebuje ničlo ali enico, bodisi "vprašanje", ki vsebuje produkt naravnega števila in dveh (pod)dreves. Odgovori predstavljajo liste drevesa, vprašanja pa ostala vozlišča. Vprašanja vsebujejo poleg poddreves tudi oznako z naravnim številom. Vozlišča drevesa so označena, da lahko v vsakem vozlišču razberemo, kateri člen zaporedja predstavlja to vozlišče. To je praktično, ker lahko algoritem, kadar ta podatek potrebuje, informacijo prebere iz oznake, namesto da bi moral temu cel čas slediti.

Drevesa so bolj efektivna, če oznake vozlišč niso nujno v vrstnem redu. Prej je koren drevesa določal prvi člen zaporedja, naslednji vozlišči sta določali drugi člen zaporedja, itd. Seveda pa lahko obstaja predikat, za katerega prvi ali drugi člen zaporedja ni pomemben. Zato naj bodo oznake neurejene in vozlišča naj predstavljajo zgolj tiste člene, ki so dejansko pomembni. S tem dosežemo, da so drevesa pogosto lahko precej manjša. Oglejmo si primer nekega predikata, pri katerem je takšna struktura praktična.

Primer 5.3. Naj bo `p` predikat na Cantorjevi množici z naslednjim predpisom.

```
let p a =
  match (a 7) with
    | true -> true
    | false -> a 12
```

Predikat `p` najprej preveri vrednost sedmega člena zaporedja `a`. Če je sedmi člen `true`, kar je za nas enako kot enica, potem je slika predikata enica. V nasprotnem primeru, torej če je sedmi člen zaporedja `false`, pa je slika predikata enaka dvanajstemu členu zaporedja `a`. Lahko ga predstavimo z drevesom na sliki 5.



Slika 5. Drevo z neurejenim vrstnim redom vozlišč.

Opazimo, da je v tem primeru bolj učinkovita predstavitev predikata z drevesom, ki nima urejenih oznak. Če bi drevo imelo urejene oznake, bi moralo biti vsaj globine 12, saj bi šele tam vozlišča predstavljala dvanajsti člen zaporedja. Zgornja predstavitev je mnogo manjša, kar je pomembno, saj velikost dreves z globino hitro narašča in implementacija funkcionala izbire s tem postaja manj učinkovita. Predstavitev z neurejenim vrstnim redom vozlišč nam bo koristila tudi pri vseh podobnih predikatih. ♢

Kljub izboljšavi lahko drevesa hitro postanejo zelo velika. Pomagamo si z lenim izračunavanjem, pri katerem se poddrevo izračuna šele, ko ga algoritem potrebuje. Če tega ne storimo, bi računalnik v spominu vedno hranil celo drevo naenkrat. Oglejmo si sledečo definicijo.

```
type tree =
  | Answer of bool
  | Question of nat * (bool -> tree)
```

Opazimo, da so listi predstavljeni enako kot prej, v zapisu ostalih vozlišč, tj. "vprašanj", pa imamo namesto dveh poddreves funkcijo iz Boolove množice v drevesa. Funkcija preslika enico v eno drevo, ničlo pa v drugo. To sta poddrevesi, ki sta povezani s tem vozliščem. Pri določanju slike predikata pot v danem vozlišču določimo tako, da funkciji podamo člen zaporedja, ki je označen v vozlišču, slika funkcije pa je poddrevo, v katerem nadaljujemo postopek. Ker so poddrevesa zapisana v obliki funkcij, se izračunajo šele, ko je funkcija uporabljena z argumentom. Vsako naslednje vozlišče drevesa se izračuna šele, ko drevo pregledujemo z nekim namenom, pri prvotni definiciji pa bi se celotno drevo izračunalo in shranilo takoj.

5.2. **Pretvorba med predikati in drevesi.** Želimo implementirat preslikavi Φ in Ψ, s katerima bomo lahko preslikali drevesa v predikate in obratno.

Funkcija, ki jo želimo, bo imela shemo Φ: `tree` $\to$ `((nat` $\to$ `bool)` $\to$ `bool)`. Ko funkciji podamo drevo, bo slika predikat s shemo `(nat` $\to$ `bool)` $\to$ `bool`. Za zapis algoritma bomo uporabili nekoliko poenostavljeno sintakso jezika OCaml.

Ukaz `(match t)` pomeni, da je slika odvisna od oblike drevesa `t`. Če je `t` odgovor, oz. list, potem je slika funkcije kar vrednost, s katero je list označen. Če je `t`

Algoritem 1 Algoritem, ki opisuje delovanje predikata Φ_t.

Vhod: dvojiško drevo `t`**, dvojiško zaporedje** `a`
Izhod: slika predikata Φ_t **pri argumentu** `a`

```
let rec iz_drevesa t a =
  match t with
    | Answer b -> b
    | Question (k, vejitev) ->
      (let poddrevo = vejitev (a k)
      in
      iz_drevesa poddrevo a)
```

drevo, ki vsebuje več kot samo list, ga razstavimo na oznako in funkcijo, ki gradi poddrevesa. Funkcijo, s katero pridemo do poddevesa, smo imenovali `vejitev`, koren drevesa pa je bil označen s številom `k`. Funkciji `vejitev` podamo k-ti člen zaporedja, tj. `(a k)` in sliko označimo kot `poddrevo`, saj je to poddrevo, v katerem moramo nadaljevati postopek. Za sliko funkcije rekurzivno pokličemo (isto) funkcijo `iz_drevesa` z argumentoma `poddrevo` in `a`.

Želimo implementirati še preslikavo Ψ, ki slika iz predikatov v drevesa. Glavni izziv implementacije je določanje oznak vozlišč. Koren drevesa želimo označiti s členom zaporedja, ki ga predikat pri izračunu preveri najprej. Vozlišči, ki se nahajata na globini 1, pa želimo označiti s členom, ki ga predikat preveri naslednjega, in tako naprej. Utemeljili smo, da predikat nekaterih členov zaporedja ne preveri in njihove vrednosti niso potrebne za izračun slike predikata. Takšnih členov nočemo predstaviti v drevesu. Zanima nas torej, katere člene zaporedja predikat potrebuje med izračunom rezultata in v kakšnem vrstnem preveri njihove vrednosti. Poznamo tri "orodja", s katerimi bi si lahko pomagali pri implementaciji na tak način:

- izjeme
- reference
- kontinuacije

Osredotočimo se na izjeme. Ko predikat preveri člen zaporedja, ki še ni bil dodan v drevo, lahko to zabeležimo z uporabo izjeme. Ta podatek izjema prenese naprej, da se v drugem delu algoritma drevo ustrezno poveča. Definirajmo izjemo `Neznano`, ki s sabo kot parameter nosi neko naravno število:

```
exception Neznano of nat
```

Oglejmo si predpis algoritma 2 za preslikavo predikatov v drevesa.

Najprej je definirano pomožno zaporedje `a`, ki pri vsakem argumentu, namesto da bi vrnilo vrednost, pokliče izjemo, v kateri pove, kateri člen zaporedja je bil zahtevan. Če to zaporedje podamo kot argument nekemu predikatu, se bo poklicala izjema, katere paremeter bo tisti člen, ki ga je hotel predikat najprej preveriti. To idejo lahko uporabimo tudi naprej pri ugotavljanju, kateri členi zaporedja so pomembni za predikat. Vsakič, ko se pokliče izjema, se zaporedju "doda" člen, ki je izjemo sprožil, torej se bo izjema sprožila šele pri naslednjem zahtevanem členu. Tako je določenih vse več členov zaporedja, dokler jih ni na neki točki dovolj, da lahko določimo sliko predikata.

Definirana je tudi pomožna funkcija `isci`, ki išče nadaljnje dele drevesa, oz. jih konstruira. Funkcija `isci` za argumenta sprejme predikat `f` in zaporedje `b`. Poskusi

Algoritem 2 Algoritem, ki opisuje delovanje preslikave Ψ.

Vhod: predikat `f` na Cantorjevi množici
Izhod: drevo, ki predstavlja predikat `f`

```
let v_drevo f =
  let rec a n =  raise (Neznano n)
  and isci f b =
    try
      Answer (f b)
    with Neznano n ->
      (let vejitev c =
        let novb k = if (k = n) then c else (b k)
        in
        isci f novb
      in
      Question (n, vejitev))
  in
  isci f a
```

vrniti odgovor, tj. list, ki vsebuje sliko predikata `f` pri argumentu `b`. Če se pri izračunu vrednosti `(f b)` pokliče izjema, jo ujame. To pomeni, da vsaj en izmed členov, ki ga potrebuje predikat, še ni določen. Tedaj se drevesu doda nova razvejitev.

Za novo razvejitev se drevesu doda vozlišče, ki ni list, moramo pa mu določiti pripadajočo oznako in funkcijo, ki določa poddrevesi. Če je `n` argument poklicane izjeme, je novo vozlišče označeno z `n`, saj je to člen zaporedja, ki ga je predikat preveril. Definiramo funkcijo `vejitev`, ki je preslikava v poddrevesi tega vozlišča. Poddrevesi bomo prav tako zgradili s funkcijo `isci`, prilagoditi pa ji je treba argumente. Definiramo novo zaporedje, ki je enako prejšnjemu v vseh členih razen `n`-tem. Ta člen zaporedja definiramo tako, da je enak argumentu funkcije `vejitev`. Tako je določenih vse več členov zaporedja in se drevo rekurzivno gradi.

Po pomožnih definicijah pokličemo funkcijo `isci` in ji kot argument podamo zaporedje `a`, ki nima še nobenega določenega člena. S tem klicem se zgradi celotno drevo, ki je slika funkcije `v_drevo` pri argumentu `f`.

5.3. **Pregled dreves.** Za funkcional izbire je treba v drevesni predstavitvi pregledati liste in poiskati takšnega, ki je označen z enico. Potrebujemo torej algoritem za pregled drevesa.

V teoriji grafov je iskanje po grafih pogost problem in obstajajo standardni algoritmi, ki so temu namenjeni. Za pregledovanje grafov sta najbolj znana dva pristopa:

- BFS, tj. iskanje v širino
- DFS, tj. iskanje v globino

Preizkusili bomo oba algoritma, saj nas zanima, v katerem primeru bo funkcional bolj učinkovit. Videli bomo, da oblika drevesa v veliki meri določa, kateri algoritem je bolj učinkovit. Bolj podrobni podatki o teoriji teh dveh algoritmov in njunem ozadju se nahajajo v knjigi [1, poglavji 3–4].

5.3.1. *DFS.* Za implementacijo algoritma DFS (angl. Depth-First Search) bomo potrebovali podatkovno strukturo *sklad*, ki hrani neki tip podatkov tako, da je prvi element tisti, ki smo ga nazadnje dodali. Omogoča nam, da na sklad dodajamo

elemente z operacijo *push*, z operacijo *pop* pa lahko iz sklada dobimo prvi element, ki se iz sklada nato odstrani. Sklad deluje po načelu *LIFO* (angl. Last In First Out), kar pomeni, da operacija pop vedno vrne tisti element, ki je bil zadnji dodan v sklad.

Algoritem 3 Algoritem DFS.

Vhod: graf G, vozlišče v, v katerem algoritem začne
Izhod: Vsa vozlišča grafa G, ki so dosegljiva iz v, so označena kot obiskana

```
1:  function DFS(G, v)
2:      S ← [ ]                                   ▷ ustvarimo sklad S
3:      for all u ∈ V(G) do
4:          obiskan(u) ← false
5:      end for
6:      PUSH(S, v)                                ▷ vozlišče v dodamo na sklad
7:      while S ≠ [ ] do
8:          u ← POP(S)
9:          if not obiskan(u) then
10:             obiskan(u) ← true
11:             for all w; uw ∈ E(G) do           ▷ sosednja vozlišča dodamo na sklad
12:                 PUSH(S,w)
13:             end for
14:         end if
15:     end while
16: end function
```

Algoritem 3 najprej ustvari sklad S in vsa vozlišča grafa označi za neobiskana. Nato se vozlišče v, v katerem se pregledovanje grafa začne, doda na sklad. Iz sklada vzame element, ki ga označi z w, in če je še neobiskan, ga označi za obiskanega. Vozlišča, ki so mu sosedna, pa se dodajo na sklad. Algoritem se konča, ko na skladu ni več elementov.

Postopek obišče vsa vozlišča povezane komponente grafa, kar je dokazano v knjigi [1, poglavje 3.2.1]. V našem primeru so vsi grafi dvojiška drevesa, zato vemo, da lahko algoritem obišče vsa vozlišča. Vozlišče, v katerem se bo postopek začel, bo koren drevesa, saj se tam začne odločitveni proces. Pregled poteka v "globino". Algoritmu dodamo še ukaz, ki preveri vrednost v listih, saj želimo najti list z enico. Za nas bo na koncu pomembno iskanje enice, katera vozlišča so bila obiskana, pa ne zares.

5.3.2. *BFS.* Algoritem BFS (angl. Breadth-First Search) se od DFS razlikuje predvsem v tem, da se namesto sklada uporablja druga podatkovna struktura *vrsta*. Vrsta prav tako omogoča metodi push in pop, vendar operacija pop deluje drugače. Vrsta deluje po načelu *FIFO* (angl. First In First Out), kar pomeni, da se jemljejo elementi, ki so bili v vrsto dodani prej. Element, ki ga vrne metoda pop je tisti, ki je v vrsti najdlje, v nasprotju s skladom, kjer je to element, ki je v skladu najmanj časa.

Pregled poteka v "širino". Najprej so obiskani vsi sosedi prvega vozlišča, šele potem postopek sledi katerikoli izmed poti bolj v globino. Enako velja tudi za naslednja vozlišča. V primeru dreves bodo najprej obiskana vozlišča bližje korena.

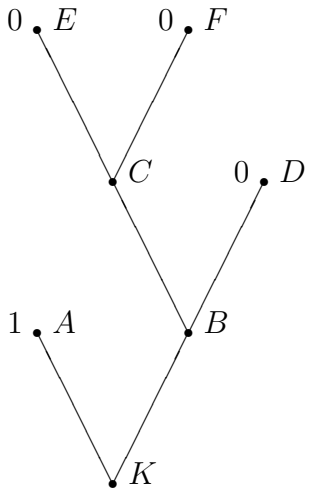
Algoritem 4 Algoritem BFS.

Vhod: graf G, vozlišče v, v katerem algoritem začne
Izhod: Vsa vozlišča grafa G, ki so dosegljiva iz v, so označena kot obiskana

```
1:  function BFS(G, v)
2:      Q ← [ ]                                   ▷ ustvarimo vrsto Q
3:      for all u ∈ V(G) do
4:          obiskan(u) ← false
5:      end for
6:      PUSH(S, v)                                ▷ vozlišče v dodamo v vrsto
7:      while Q ≠ [ ] do
8:          u ← POP(Q)
9:          if not obiskan(u) then
10:             obiskan(u) ← true
11:             for all w; uw ∈ E(G) do           ▷ sosednja vozlišča dodamo v vrsto
12:                 PUSH(S,w)
13:             end for
14:         end if
15:     end while
16: end function
```

5.3.3. *Primerjava.* Oglejmo si primer, kako oba algoritma delujeta na enakem drevesu.

Primer 5.4. Na drevesu iz slike 6 poženemo tako DFS kot BFS algoritme. Zanima nas, v kakšnem vrstnem redu algoritma označita vozlišča drevesa za obiskana.



Slika 6. Drevo za primer delovanja DFS in BFS algoritmov.

Oba algoritma začneta v korenu K. Vrstni red, v katerem DFS obišče vozlišča, je $KBDCFEA$. BFS pa vozlišča pregleda v vrstnem redu $KABCDEF$.

Glede na to, da želimo najti list z enico, kar je v tem primeru vozlišče A, se BFS odreže precej bolje. Opazimo, da je za takšno dobro delovanje BFS-ja ključnega pomena, da se list z enico nahaja blizu korena drevesa. ♢

Čas, ki ga algoritma potrebujeta za iskanje lista z enico, je odvisen od oblike drevesa. Če imamo drevo, kjer se vsi list nahajajo enako daleč od korena, potem je DFS očitno boljša izbira. Algoritem BFS pregleda skoraj polovico vozlišč, predno sploh pride do prvega lista, oz. mu vsi listi ostanejo za na konec, med tem ko lahko DFS do lista pride precej hitreje. V najslabšem primeru DFS potrebuje približno enako količino časa kot BFS, v najboljšem pa je mnogo hitrejši.

V nekaterih drugih primerih deluje BFS praviloma bolje kakor DFS. Tak primer bi bila npr. drevesa, ki imajo nekatere liste precej bližje korenu kot ostale. Če se kakšen list z enico nahaja blizu korena, ga bo BFS zagotovo hitro našel, DFS pa ga lahko velikokrat zgreši in pregleda mnogo večji del drevesa, predno ga najde.

5.4. **Funkcional izbire.** Vse dele algoritma, ki smo jih razložili, moramo še združiti v funkcional izbire. Pri tem moramo algoritma DFS/BFS še prilagoditi, saj želimo v drevesu ne le najti list z enico, temveč dobiti tudi pot do njega. Med izvajanjem algoritma za iskanje po drevesu moramo hraniti podatke o poti.

Določimo podatkovni tip *path*, ki ga bomo uporabljali za opisovanje poti v drevesu. Vemo, da lahko pot od korena do poljubnega vozlišča v dvojiških drevesih množice $\mathbf{T}_2$ opišemo z zaporedjem ničel in enic. Vozlišča drevesa smo označili z naravnimi števili, torej členom poti dodamo naravna števila, ki te oznake predstavljajo. Pot predstavimo kot seznam parov, ki bodo vsebovali ničlo ali enico in pa naravno število. Definicijo zapišemo v sledeči obliki.

```
type path = Steps of (nat * bool) list
```

Želimo, da algoritem za pregled drevesa vrne vozlišče skupaj s potjo do njega. Vozlišča bomo hranili skupaj s potjo do njih. Vrsto/sklad v algoritmu bomo ustrezno priredili, da bo hranil vozlišče in pot, preostanek algoritma pa bo moral, poleg pregleda vozlišča, naprej prenesti podatek o poti. Algoritem 5 je implementirana rekurzivna različica DFS, ki je prirejena našim zahtevam.

Glede na obliko sklada se določi, kakšen bo postopek. Če je sklad prazen, funkcija vrne prazno pot. Sicer je postopek odvisen od prvega elementa sklada. Če je na začetku sklada list, ki vsebuje vrednost `true`, smo našli, kar smo želeli, in vrnemo pot do tega vozlišča. Če je na začetku sklada list z vsebino `false`, vendar je to edino vozlišče v skladu, prav tako vrnemo pot, saj v tem primeru drevo ne vsebuje lista z vsebino `true`, funkcional pa mora vseeno vrniti rezultat. Če je prvi element list z vrednostjo `false`, ki ni edini element sklada, ga odstranimo, saj nas ne zanima, in postopek nadaljujejmo z naslednjim elementom sklada.

Če je prvi element sklada vozlišče, ki ni list, ga razstavimo na vozlišče in pot, shranimo poddrevesi ter ju označimo s `tbranch` in `fbranch`. Sestavimo še poti do vsakega izmed poddreves tako, da poti do trenutnega vozlišča dodamo člen, ki opiše, kam zavije pot pri zadnji razvejitvi. Poddrevesi skupaj s potjo dodamo na sklad in rekurzivno pokličemo funkcijo, s čimer se izvede naslednji korak algoritma DFS.

Opomba 5.5. Če algoritmu podamo list ali prazen sklad, je rezultat prazna pot. Drevo z enim vozliščem ne vsebuje povezave, ki bi lahko tvorila pot, torej je ta rezultat pravilen. Takšna drevesa so predstavitve konstantnih predikatov, ki slikajo vsa zaporedja bodisi v `true` bodisi v `false`. Če gledamo prazen sklad kakor sklad, ki vsebuje le "prazno drevo", je odgovor tudi v tem primeru smiseln.

Želimo dobiti funkcional izbire, ki mu podamo predikat in nam vrne zaporedje iz Cantorjeve množice. Naj ima funkcional sledečo obliko.

Algoritem 5 Implementirani prirejeni algoritem DFS.

Vhod: Sklad `S`, ki vsebuje poddrevo in pot od korena celotnega drevesa do korena poddrevesa
Izhod: Pot do vozlišča, ki vsebuje `true`, oz. prazna pot, če tako vozlišče ne obstaja

```
let rec dfs_path S =
  match S with
    | [] -> Steps []
    | (Answer true, Steps w)::ts -> Steps w
    | (Answer false, Steps w)::[] -> Steps w
    | (Answer false, Steps w)::ts -> dfs_path ts
    | (Question (n, branch), Steps w)::ts ->
      (
      let tbranch = branch true
      and fbranch = branch false
      in
      let new_t1 = (tbranch, Steps ((n, true)::w))
      and new_t2 = (fbranch, Steps ((n, false)::w))
      in
      dfs_path (new_t1 :: new_t2 :: ts)
      )
```

Algoritem 6 Funkcional izbire.

Vhod: Predikat `p`
Izhod: Element Cantorjeve množice, ki ustreza pogojem, ki jih zahteva definicija funkcionala izbire

```
let dfs_epsilon p = dfs_epsilon_tree (to_tree p)
```

V zapisu algoritma 6 je `to_tree` funkcional za pretvorbo predikatov v drevesa, ki smo ga implementirali že prej. Funkcija `dfs_epsilon_tree` mora izvesti prilagojen DFS algoritem na dobljenem drevesu, vendar pa argument še ni v primerni obliki, da bi ga sprejela funkcija `dfs_path`, ki je zapisana v algoritmu 5. Zato jo ovijemo z dodatno funkcijo, ki poskrbi za pretvorbo med podatkovni tipi. Potrebujemo tudi funkcijo `path_seq` (zapisano v algoritmu 7), ki bo poskrbela za pretvorbo med podatkovnima tipoma poti in zaporedij.

```
let dfs_epsilon_tree t =
  let way = dfs_path [(t, Steps [])]
  in
  path_seq way
```

Funkcija združi drevo in prazno pot v par ter ju ovije v seznam, ki bo služil kot sklad. V takšni obliki je to primeren argument za funkcijo `dfs_path`. Rezultat, ki ga dobimo, preoblikujemo v primerno obliko, saj mora funkcional izbire vrniti neskončno zaporedje ničel in enic, prvotni rezultat pa je končna pot.

Ideja je, da začnemo z neskončnim zaporedjem, ki ima vse vrednosti `true`. Potem funkcija prebira člene poti in zaporedje ustrezno prilagaja. Če je v poti člen oblike

`(4, false)`, se četrti člen zaporedja spremeni iz `true` v `false`. Postopek se nadaljuje, dokler ne zmanjka členov poti. Tedaj smo dobili iskano zaporedje. Vsi členi zaporedja, ki niso vključeni v pot, so lahko poljubni, vendar so pri tem načinu vsi takšni členi določeni za `true`. Zaporedje, ki ga bo izmed vseh primernih izbral naš funkcional izbire, je tisto, ki ima najmanj členov `false`. Funkcija je implementirana v algoritmu 7.

Algoritem 7 Funkcija za pretvorbo poti v element Cantorjeve množice.

Vhod: Pot zapisana v podatkovnem tipu `path`
Izhod: Neskončno zaporedje ničel in enic

```
let path_seq path =
  let rec transform p a' =
    match p with
      | Steps [] -> a'
      | Steps ((n, b)::xs) ->
        (
        let b' k = if k = n then b else (a' k)
        and new_p = Steps xs
        in
        transform new_p b'
        )
  and c' n = true
  in
  transform path c'
```

To je samo ena izmed možnih implementacij funkcionala izbire. Namesto algoritma DFS lahko za iskanje uporabimo algoritem BFS in v implementaciji funkcijo `dfs_path` zamenjamo z ekvivalentom `bfs_path`. Tudi za pretvorbo predikatov v drevesa je možnih več različnih pristopov.

Vemo, da lahko s funkcionalom izbire izrazimo in posledično implementiramo tudi logično kvantifikacijo. Sedaj, ko imamo funkcional izbire, lahko npr. eksistenčni kvantifikator zapišemo v obliki:

```
let dfs_exists p = p (dfs_epsilon p)
```

6. Testiranje

V zadnjem poglavju preizkusimo različne implementacije funkcionala izbire. Zanima nas, kako se različni postopki primerjajo med seboj. Za merjenje hitrosti definiramo funkcijo, ki meri čas izvajanja dane funkcije na podanem argumentu. Nato primerjamo, koliko časa traja, da različni postopki rešijo enak primer.

```
let time f x =
  let t1 = Sys.time() in
  let fx = f x in
  let t = (Sys.time() -. t1) in
  Printf.printf "Execution time: %fs\n" t;
  fx
```

Funkciji `time` podamo funkcijo `f` in argument `x`. S klicom funkcije `Sys.time()` se shrani trenutno stanje ure v računalniku, nato se izračuna vrednost funkcije `f` pri

argumentu `x` in po koncu še enkrat zabeleži uro v računalniku. Razlika med obema zabeleženima časoma se izpiše kot čas procesiranja funkcije `f`. Definirajmo nekaj primerov testnih predikatov, na katerih bomo izvajali preizkuse.

```
let h a' =
  match (a' 1) with
    | true -> false
    | false ->
      match (a' 2) with
        | true -> true
        | false -> false

let f1 a' =
  match (a' 1) with
    | _ ->
     match (a' 2) with
      | _ ->
       match (a' 3) with
        | _ ->
         match (a' 4) with
          | _ ->
           match (a' 5) with
            | true -> true
            | false -> false
```

Definiramo še predikata `f2`, `f3`, ki sta oblikovana enako kot `f1`, vendar imata več razvejitev. Drevo, ki predstavlja `f2`, je globine 15, v primeru `f3` pa je globine 20. Definiramo tudi predikat `f4`, ki je prav tako globine 20, ima pa zgolj en list z vrednostjo `true`, ki se nahaja na globini 2, vsi ostali listi pa imajo vrednost `false`. Na teh predikatih preizkusimo, kako hitro se izračunata funkcionala `bfs_epsilon` in `dfs_epsilon`.

Za izračun predikatov `h` in `f1` potrebujeta oba funkcionala zelo malo časa. Izmerjen rezultat pravi, da izračun traja 0 sekund, saj je natančnost uporabljene sistemske ure premajhna. Zanimali nas bodo ostali, bolj kompleksni, primeri.

Meritve za primere `f2`, `f3` in `f4` so zabeležene v tabeli 7.

	`bfs_epsilon`	`dfs_epsilon`
`f2`	0,097	0,00
`f3`	12,445	0,00
`f4`	0,00	5,061

Slika 7. Meritve časa za izračun rezultata funkcionala izbire.

Vidimo, da v primerih `f2` in `f3` algoritem `bfs_epsilon` deluje počasneje kakor `dfs_epsilon`. Primera `f2` in `f3` sta oblike, ki je najslabša za BFS algoritem. Pri primeru `f4` pa je `bfs_epsilon` hitrejši od `dfs_epsilon`. Pri obliki drevesa, kjer se nahaja list z vrednostjo `true` blizu korena drevesa, ostali listi pa so na precej večji globini, se BFS praviloma odreže bolje.

7. Zaključek

V diplomskem delu smo definirali funkcional izbire na Cantorjevi množici. Uvedli smo predstavitev predikatov z dvojiškimi drevesi, s katero smo si pri implementaciji pomagali. Dokazali smo, da je funkcional izračunljiv in da je uporaba dreves smiselna, saj so končna. Nato smo navedli nekaj različnih implementacij in sprogramirali dve. Testiranje pokaže, da je to, kateri postopek deluje bolje, odvisno od primera.

8. Priloga

```
(* ======================================== *)
(* TYPE DEFINITIONS *)
(* ======================================== *)

type nat = int
exception DontKnow of nat
type path = Steps of (nat * bool) list

type tree =
  | Answer of bool
  | Question of nat * (bool -> tree)

(* ======================================== *)
(* AUXILIARY FUNCTIONS *)
(* ======================================== *)
(* Mainly for transformation between types. *)

(* val path_seq : path -> (nat -> bool) *)
let path_seq path =
  (* Transforms a path to a (nat -> bool) sequence.  *)
  let rec transform p a' =
    match p with
      | Steps [] -> a'
      | Steps ((n, b)::xs) ->
        (
        let b' k = if k = n then b else (a' k)
        and new_p = Steps xs
        in
        transform new_p b'
        )
  and c' n = true
  in
  transform path c'

(* val from_tree : tree -> ((nat -> bool) -> bool) *)
let rec from_tree t a' =
  match t with
    | Answer b -> b (* If the tree only contains a leaf
                       we have found the result. *)
```

```
    | Question (k, branch) ->
      (* Otherwise (a' k) decides which branch of the tree we follow.
         val branch : bool -> tree, therefore (branch (a' k))
         is a tree. We continue inspecting the selected subtree. *)
      (
      let subtree =
        branch (a' k)
      in
      from_tree subtree a'
      )


(* ========================================= *)
(* TREE CONSTRUCTION *)
(* ========================================= *)

(* EXCEPTIONS *)

(* val to_tree : ((nat -> bool) -> bool) -> tree *)
let to_tree f =
  (* a' is the starting sequence where the functional
     has not checked any link, we specify more 'known'
     links as more of the sequence is inspected by f *)
  let rec a' n =  raise (DontKnow n)
  (* "search" follows the decision making process of the
     functional and generates the desired tree along the way.
     Vertices "Question (n, branch)" are appended when f
     checks the n-th link of the sequence and the branches
     in a vertex are decided by further actions of f. *)
  and search f b' =
    (* Try to find the result of the functional
       without checking any unknown links of the sequence b'*)
    try
      Answer (f b')
    with DontKnow n ->
      (* If the functional checks an unknown link an exception
         is raised and another "branch" is added to the decision
         making tree. The branches are based on the n-th link
         of the parameter sequence b', "newb'" is a new sequence
         where the n-th link is not unknown anymore. We continue
         figuring out what the result of the functional is now that
         the parameter sequence has one more known link. *)
      (
      let branch b =
        let newb' k =
          if k = n then b else (b' k)
        in
        search f newb'
      in
```

```
      Question (n, branch)
      )
  in
  (* We start a search with sequence a'
    which doesn't have any checked links. *)
  search f a'


(* ======================================== *)
(* TREE SEARCH & SELECTION FUNCTIONALS *)
(* ======================================== *)


(* BFS *)

(* val bfs_path : (tree * path) list -> path *)
let rec bfs_path queue =
  match Queue.pop queue with
  | exception Queue.Empty -> Steps []
  | (Answer true, Steps w) -> Steps w
  | (Answer false, Steps w) -> (
    match Queue.peek queue with
    | exception Queue.Empty -> Steps w
    | _ -> (bfs_path queue))
    | (Question (n, branch), Steps w)-> (
      let tbranch, fbranch = branch true, branch false in
      let new_t1 = (tbranch, Steps ((n, true) :: w)) in
      let new_t2 = (fbranch, Steps ((n, false) :: w)) in
      (Queue.push new_t1 queue;
       Queue.push new_t2 queue;
       bfs_path queue))

(* val bfs_epsilon_tree : tree -> (nat -> bool) *)
(* "bfs_epsilon_tree" uses bfs to construct a sequence for
  which the functional (from_tree t) will evaluate true
  if such a sequence exists. If no such sequence exists
  the function still returns a sequence. *)
let bfs_epsilon_tree t =
  let way = bfs_path [(t, Steps [])]
  in
  path_seq way


(* DFS *)

(* val dfs_path : (tree * path) list -> path *)
let rec dfs_path queue =
(* Performs a dfs search on a tree and returns a path which is
   equivalent to the sequence an epsilon functional would return. *)
```

```
  match queue with (* elements of queue: (tree, Steps w)  *)
    | [] -> Steps []
    | (Answer true, Steps w)::ts -> Steps w
    | (Answer false, Steps w)::[] -> Steps w
    | (Answer false, Steps w)::ts -> dfs_path ts
    | (Question (n, branch), Steps w)::ts ->
      (
      let tbranch = branch true
      and fbranch = branch false
      in
      let new_t1 = (tbranch, Steps ((n, true)::w))
      and new_t2 = (fbranch, Steps ((n, false)::w))
      in
      dfs_path (new_t1 :: new_t2 :: ts)
      )

(* val dfs_epsilon_tree : tree -> (nat -> bool) *)
(* "dfs_epsilon_tree" uses bfs to construct a sequence
   for which the functional (from_tree t) will evaluate true
   if such a sequence exists. If no such sequence exists
   the function still returns a sequence. *)
let dfs_epsilon_tree t =
  let way = dfs_path [(t, Steps [])]
  in
  path_seq way


(* EPSILONS & LOGICS *)

(* val epsilon : ((nat -> bool) -> bool) -> (nat -> bool) *)
(* val exists : ((nat -> bool) -> bool) -> bool *)

let epsilon p = epsilon_tree (to_tree p)
let exists p = p (epsilon p)

let bfs_epsilon p = bfs_epsilon_tree (to_tree p)
let bfs_exists p = p (bfs_epsilon p)

let dfs_epsilon p = dfs_epsilon_tree (to_tree p)
let dfs_exists p = p (dfs_epsilon p)


(* ======================================== *)
(* FUNCTIONS FOR TESTING *)
(* ======================================== *)

let time f x =
  (* Times the execution time of
    a given function and prints it out. *)
```

```
let t1 = Sys.time() in
let fx = f x in
let t = (Sys.time() -. t1) in
Printf.printf "Execution time: %fs\n" t;
fx
```

Slovar strokovnih izrazov

binary tree dvojiško drevo
breadth-first search pregled v širino – algoritem za pregled grafa, ki najprej pregleda bližnja vozlišča
Cantor set Cantorjeva množica
computable izračunljiv – lastnost funkcije, da se jo da računalniško izračunati
depth-first search pregled v globino – algoritem za pregled grafa, ki pregleduje najprej v globino
functional funkcional
selection functional funkcional izbire

Literatura

[1] S. Dasgupta, C. Papadimitriou in U. Vazirani, *Algorithms*, McGraw-Hill, New York, 2008; dostopno tudi na `people.eecs.berkeley.edu/~vazirani/algorithms/chap3.pdf`.

[2] H. Enderton, *Computability theory: An introduction to recursion theory*, Academic Press, Cambridge, Massachusetts, 2010.

[3] M. Escardó in P. Oliva, *What sequential games, the Tychonoff theorem and the double-negation shift have in common*, v: MSFP@ICFP, ACM New York, New York, 2010, str. 21–32.

[4] L. Loomis, *Introduction to abstract harmonic analysis* , Van Nostrand, New York, 1953; dostopno tudi na `archive.org/details/introductiontoab031610mbp`.

[5] Y. Minsky, A. Madhavapeddy in J. Hickey, *Real world OCaml: Functional programming for the masses*, O'Reilly Media, 2013; dostopno tudi na `v1.realworldocaml.org/v1/en/html/index.html`.

[6] P. Pavešić, *Splošna topologija*, Izbrana poglavja iz matematike in računalništva **43**, DMFA – založništvo, Ljubljana, 2008.

[7] P. Potočnik, *Zapiski predavanj iz Diskretne matematike 1*, verzija marec 2011, [ogled 25. 8. 2018], dostopno na `fmf.uni-lj.si/~potocnik/Ucbeniki/DM-Zapiski2010.pdf`.

[8] J. Vrabec, *Metrični prostori*, Matematika – Fizika **31**, DMFA – založništvo, Ljubljana, 1990.

[9] K. Weihrauch, *Computable analysis: an introduction*, Texts in theoretical computer science, an EATCS series, Springer-Verlag, Berlin, 2000.