Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Blaž Repas

Preverjanje pravilnosti programov z odvisnimi tipi v programskem jeziku Idris

diplomsko delo

univerzitetni strokovni študijski program prve stopnje računalništvo in informatika

Delo je pripravljeno v skladu s Pravilnikom o podeljevanju Prešernovih nagrad študentom, pod mentorstvom doc. dr. Jurija Miheliča

Mentor: doc. dr. Jurij Mihelič

Ljubljana 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Programski jeziki so eno izmed področij, ki je kljub svoji starosti še izredno živahno. Večina jezikov programerju nudi osnoven nabor tipov, ki ga je možno razširiti s tipi, zgrajenimi iz že obstoječih tipov. V zadnjem času pa se je pojavilo več jezikov, npr. Agda, Idris, Epigram, ki podpirajo tako imenovane odvisne tipe. Ti so lahko odvisni ne le od drugih tipov, temveč tudi od vrednosti. Zaradi potrebe po izračunu vrednosti takšna odvisnost pogosto vodi v neodločljivost preverjanja skladnosti tipov.

V diplomski nalogi se osredotočite na področje odvisnih tipov in na programske jezike, ki takšne tipe podpirajo. Predstavite prednosti in slabosti odvisnih tipov. Izberite in opišite enega izmed programskih jezikov, ki podpira odvisne tipe. V njem sprogramirajte nekaj algoritmov in podatkovnih struktur ter pri tem ustrezno uporabite odvisne tipe. Primerjajte izvedbe z odvisnimi tipi z izvedbami brez njih.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Blaž Repas, z vpisno številko **63110341**, sem avtor diplomskega dela z naslovom:

Preverjanje pravilnosti programov z odvisnimi tipi v programskem jeziku Idris

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Jurija Miheliča,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 1. septembra 2014 Podpis avtorja:

Zahvaljujem se vsem, ki ste pripomogli k nastajanju tega diplomskega dela. Posebej pa se zahvaljujem mentorju doc. dr. Juriju Miheliču za ves vložen trud in skrb, za vse skrbne popravke ter za odlično strokovno in akademsko usmerjanje.

Kazalo

Povzetek

Formalno dokazovanje pravilnosti programov se v ključnih primerih izvaja ročno. Programski jeziki z odvisnimi tipi kot logična ogrodja predstavljajo avtomatizirano alternativo ročnemu preverjanju pravilnosti. V tem diplomskem delu obravnavamo sisteme tipov ter programske jezike z odvisnimi tipi. Predstavimo programski jezik Idris, ki podpira poljubno kompleksne odvisne tipe. Pokažemo, kako izjave pretvoriti v tipe funkcij in jih z implementacijo funkcij dokazati, pravilnost pa avtomatsko preverja prevajalnik. S pomočjo odvisnih tipov podamo avtomatsko preverjene implementacije podatkovnih struktur seznam, sklad in vrsta. Poglobimo se v algoritem urejanja z vstavljanjem. Z dokazom linearne urejenosti naravnih števil dokažemo, da je njegov izhod urejen seznam, z dokazom permutiranosti pa pokažemo, da je izhodni seznam permutacija vhodnega.

Ključne besede: odvisni tipi, pravilnost programov, programski jezik Idris.

Abstract

Formal verification of program correctness in mission critical applications is still done manually. Programming languages with dependent types used as logical frameworks present an alternative to manual verification of correctness. In this thesis we deal with type systems and programming languages with dependent types. We introduce the Idris programming language which supports arbitrarily complex dependent types. We show how to translate propositions into function types and how to prove them by implementing the functions. Correctness is then automatically verified by the compiler. With use of dependent types we provide automatically verified implementations of data structures like list, stack and queue. To demonstrate usefulness of dependent types we provide automatically verified implementation of insertion sort algorithm. We prove sortedness with linear order of natural numbers and also that the output list is a permutation of the input list.

Keywords: dependent types, program correctness, Idris programming language.

Poglavje 1

Uvod

S pojavom prvih računalnikov, za katere je bilo možno pisati programe, so se začeli razvijati tudi programski jeziki. Sprva, sicer veliko bolj prilagojeni stroju, so hitro začeli dosegati tudi višje, abstraktne nivoje, ki so človeku bolj razumljivi. Razvoj abstrakcij je botroval k drugačnemu, bolj matematičnemu pogledu na pomen programov.

Z razvojem programskih jezikov so se začeli razvijati tudi sistemi tipov z bistvenim namenom, da bi pomagali preprečevati nastanek napak v programih. V programskih jezikih najdemo različne sisteme tipov in tudi različne načine preverjanja skladnosti tipov. V tem diplomskem delu se osredotočamo na programske jezike, ki preverjanje skladnosti tipov izvedejo med prevajanjem in ne med izvajanjem programa. Analiza skladnosti tipov ima pri razvoju programov ključno vlogo. Preprečila naj bi nastanek napak. Opozorila naj bi tako na napake, ob katerih se program nekontrolirano ustavi, kot tudi na tiste, zaradi katerih so izračuni napačni. Poznamo številne primere napak v programih, ki so povzročile ogromno škodo ali pa celo ogrozile človeško življenje. Pravilnost programov je neizpodbitna nujnost na marsikaterem področju, na primer v medicini, bančništvu, vesoljskih programih, energetiki, znanosti in prometu. Na področjih, kjer je napačno delovanje programov nedopustno, je pravilnost programov ponavadi dokazana ročno. To pa ni le izjemno drag, ampak tudi zamuden postopek, ki ga je potrebno ponavljati

za vsako spremembo programa.

Alternativa ročnemu dokazovanju so ogrodja za dokazovanje, ki pa v osnovi temeljijo na programskih jezikih z odvisnimi tipi. V tem diplomskem delu je na kratko predstavljena teorija odvisnih tipov ter analogija med programi in dokazi izjav. Osredotočimo se na programski jezik Idris, ki podpira poljubno kompleksne odvisne tipe, hkrati pa ostaja splošno-namenski. Pokažemo, kako enostavne izjave zapišemo kot tipe funkcij in jih z implementacijo funkcij tudi dokažemo, pravilnost dokazov pa je avtomatsko preverjena s prevajalnikom.

V nadaljevanju pokažemo, kako obstoječe algoritme in podatkovne strukture preoblikujemo tako, da lahko njihovo pravilnost avtomatsko preverjamo. S pomočjo odvisnih tipov pokažemo pravilnost delovanja sklada in vrste, nato pa se poglobimo v pravilnost algoritma za urejanje z vstavljanjem. Dokažemo relacijo linearne urejenosti nad naravnimi števili, ki jo uporabimo v implementaciji urejanja z vstavljanjem. Za algoritem urejanja ne pokažemo le to, da spoštuje urejenost, ampak tudi permutiranost, kar pravilnost algoritma zaokroži v celoto, pravilnost algoritma pa je samodejno preverjena.

V tem delu želimo bralcu približati programske jezike z odvisnimi tipi, s tem pa tudi avtomatsko dokazovanje pravilnosti. Nenazadnje je namen tega diplomskega dela tudi pokazati praktično uporabo programskih jezikov z odvisnimi tipi in pisanje programov, ki so pravilni po konstrukciji. Poleg prikaza uporabnosti programiranja z odvisnimi tipi lahko to diplomsko delo služi tudi kot učni pripomoček pri spoznavanju odvisnih tipov.

Poglavje 2

Programski jeziki in tipi

Programski jeziki so eno izmed ključnih področij računalništva. Poleg množične praktične uporabe so zanimivo in popularno področje tudi za akademsko raziskovanje. Kar nekaj jezikov najdemo, ki so plod akademskega dela, na primer Agda [5], Idris [6] ter Haskell [14]. Drugi, verjetno veliko številčnejši, pa so nastali po bolj pragmatični poti. Med bolj popularnimi v zadnjem času [15] najdemo znana imena: Java, C, C++, PHP, Python [1].

Programske jezike je moč razvrščati in obravnavati po več kriterijih. V sklopu tega diplomskega dela se bom omejil zgolj na tipiziranost, sisteme tipov in deloma tudi na paradigme jezikov.

2.1 Tipiziranost jezika

Tipi in *sistemi tipov* so se in se razvijajo iz pomembnega, a v svojem bistvu čisto pragmatičnega razloga. V članku [9] je motivacijo za nastanek sistemov tipov moč najti predvsem v iskanju mehanizma, ki bi preprečil nastanek programov, ki bi med svojim izvajanjem (lahko) naleteli na napake. V angleški literaturi ta mehanizem imenujemo *type checker*, po slovensko pa ga lahko poimenujemo kot *mehanizem za preverjanje skladnosti tipov*.

[1] Našteti programski jeziki niso niti nujno najbolj popularni niti ne razvrščeni po popularnosti.

Lastnost, da so tipi v programu skladni in pravilni (angl. *type soundness*), seveda ni trivialna. Zahteva določeno mero formalizacije, kar botruje k aktivni akademski dejavnosti na področju sistemov tipov. Izkaže se, da je možno preverjati pravilnosti tipov tudi z dokaj omejeno formalizacijo. Dejstvo je, da v večjem ali manjšem obsegu prevajalniki vsebujejo mehanizem za preverjanje te lastnosti. Dobro je, da sta zasnova jezika in tudi sistem tipov osnovana na formalni podlagi. S tem je doseženo, da je pomen in prav tako tipiziranost jezika enolično določena. To omogoča, da lahko različne implementacije prevajalnikov za isti jezik dajejo konsistentne in enake rezultate.

Preden razdelimo jezike v različne skupine glede na tipiziranost, velja nameniti nekaj besed o tem, kaj tipi sploh so.

2.1.1 Tipi

Tipi nosijo informacijo o pomenu in obnašanju programa preko možnih vrednosti, ki jih lahko spremenljivka nekega tipa zavzame. Na tipe in vrednosti določenega tipa, zapisno kot $a : A$ (prebrano kot vrednost a tipa A), lahko poenostavljeno gledamo kot na množice in elemente množic $a \in A$, vendar pa se striktne matematične definicije tej analogiji izogibajo. V knjigi *Homotopy type theory* [22] je tip definiran kot lastnost, ki jo vsaka vrednost ima že po svoji naravi, torej vrednost sama po sebi brez tipa ne more obstajati. Edino smiselno je torej obravnavati vrednost in njen tip skupaj $a : A$.

Tip nosi lastnosti vrednosti, ki nam omogočajo, da lahko preverjamo pomen programa. Na primer, recimo, da je x naravno število[2]. S tem želimo izraziti, da je x tipa $\mathbb{N}$ oziroma $x : \mathbb{N}$. Spremenljivki x torej (v pravilnem) programu ne moremo prirediti vrednosti, ki ni tipa $\mathbb{N}$, recimo niza črk. Niti ji ne moremo prirediti vrednosti -1, saj bi to kršilo pravilnost tipov. Ob striktnem preverjanju pravilnosti tipov lahko za spremenljivko x upoštevamo, da ima določene lastnosti. V tem konkretnem primeru vemo, da je njena vrednost vedno nenegativna. V konkretnih programskih jezikih teh lastnosti velikokrat niso avtomatsko upoštevane niti implicitno prisotne. Obstajajo

[2]Naravna števila vključno z 0.

pa programski jeziki, v katerih takšne lastnosti lahko izrazimo in dokažemo.

2.1.2 Tipi in odpravljanje napak ob izvajanju

Motivacija za razvoj sistemov tipov in uporabo mehanizmov za preverjanje skladnosti tipov je odprava nekaterih napak, ki bi se zgodile ob izvajanju programa. Te napake niso nujno take, da bi se ob njih program (nasilno) ustavil. Naletimo lahko tudi na take, pri katerih se izvajanje programa ne konča, program pa zaradi napake ne daje pravilnih rezultatov. Odkrivanje teh napak je lahko silno težavno, saj se lahko pojavijo bolj ali manj pogosto, predvsem pa si ne želimo, da bi se pojavile med izvajanjem programa, ki je že v produkcijski rabi.

Avtor v knjigi [21] navaja, da preverjanje skladnosti tipov lahko pomaga zaznati širok spekter napak. Napake se pojavijo v obliki *trivialnih*, za katere je razlog moč iskati v človeški pozabljivosti (na primer: programer pozabi pretvoriti niz znakov v številko, preden jo zmnoži z drugo). Velikokrat pa se pojavijo tudi bolj kompleksne – *konceptualne* napake. Primer takšne napake bi lahko našli v programu, ki računa z različnimi denarnimi valutami. Programer lahko računa z vrednostmi dveh različnih valut brez izvedbe pretvorbe ene valute v drugo. Konceptualne napake so mnogokrat tesno povezane s pomenom programa in jih je brez pomoči preverjanja skladnosti tipov težje odkriti. Ob pravilni (in izkušeni) uporabi sistema tipov se veliko (tudi konceptualnih) napak izrazi kot napaka oziroma neskladnost na nivoju tipov.

Mehanizmi za preverjanje skladnosti tipov pa pripomorejo tudi k vzdrževanju in preoblikovanju (angl. refactoring) kode. Na primer sprememba argumentov funkcije privede do neskladnosti na nivoju tipov in prevajalnik nas opozori na nepravilno rabo. S tem laže najdemo pojavitve in vnesemo spremembe.

Na tej točki velja omeniti, kako lahko delimo programske jezike.

2.1.3 Tipizirani in netipizirani jeziki

Programske jezike lahko delimo na *tipizirane* in *netipizirane*. Vendar pa, kot navaja avtor v članku [9], lahko netipizirane jezike obravnavamo kot tipizirane, le da imajo en univerzalen tip, kateremu pripadajo vse vrednosti. V netipiziranih jezikih se lahko zgodi, da operacije sprejmejo nepravilne argumente in rezultat je lahko poljubna vrednost, napaka ali celo nedefinirano obnašanje. To je seveda neželeno delovanje, ki pa se mu želimo izogniti. Primer netipiziranega jezika je recimo zbirnik.

Tipizirani jeziki imajo večje (glede na možne konstrukcije lahko tudi neskončno) število tipov. Pravilnost oziroma skladnost se preverja ob prevajanju ali izvajanju programa. Glede na to ločimo *statično* in *dinamično* tipizirane jezike. Prvi opravijo analizo skladnosti tipov ob prevajanju, slednji pa ob izvajanju.

2.1.4 Dinamično tipizirani jeziki

Dinamično tipizirani jeziki ali kot avtor knjige [21] navaja dinamično preverjeni jeziki preverjajo skladnost in pravilnost tipov pred oziroma med izvajanjem programa. Dinamičen pristop k preverjanju skladnosti tipov omogoča hitro pisanje prototipov. Zagovorniki dinamično tipiziranih jezikov menijo, da ti jeziki omogočajo hitrejši razvoj. Omeniti velja, da preverjanje skladnosti tipov med izvajanjem programa ne more preprečiti določenih napak ali pa sproži napako zaradi neujemanja tipov. Napakam ob izvajanju se želimo izogniti, saj je to motivacija za uporabo sistemov tipov in mehanizmov za preverjanje skladnosti. V tem pogledu delujejo dinamično tipizirani jeziki rahlo paradoksalno. Sodbo o produktivnosti pa bomo preskočili, saj ni del obsega te diplomske naloge, niti ni moč tega (na enostaven in nepristranski način) preverjati.

2.1.5 Statično tipizirani jeziki

Zaradi statične narave ponujajo statično tipizirani jeziki dobro podlago za bolj znanstveni pristop k preverjanju skladnosti tipov. Mehanizem za preverjanje skladnosti tipov je večinoma vgrajen v prevajalnik, v fazo semantične analize. Različni jeziki uporabljajo različne in različno *močne* sisteme tipov. Odločitev o sistemu tipov je skorajda popolnoma povezana z načrtovanjem programskega jezika in z abstrakcijami, ki jih ponuja. Statično tipizirani jeziki imajo večinoma tipe vgrajene v samo sintakso. To je način, kako prevajalniku posredovati informacije o tipih. Lep primer eksplicitno izraženih tipov je v programskem jeziku Java, kar lahko vidimo na sliki 2.1. Funkcija *idDivisible* ima eksplicitno določen vrnitveni tip *boolean*, določena pa sta tudi tipa argumentov *a* in *b*, ki sta tipa *int*.

```
public static boolean isDivisible(int a, int b) {
    boolean divisible = false;
    if (a % b == 0) {
        divisible = true;
    } else {
        divisible = false;
    }
    return divisible;
}
```

Slika 2.1: Eksplicitno podane informacije o tipih

Vendar pa statična tipiziranost od programerja nujno ne zahteva, da mora podati vse informacije o tipih. S pomočjo *inference tipov* je možno v določenih primerih pridobiti informacije o tipih iz vrednosti in konteksta. Primer inference tipov (v programskem jeziku Haskell) lahko vidimo na sliki 2.2. Prevajalnik lahko za oba argumenta in tudi za vrnitveno vrednost funkcije *isDivisible* izračuna, kakšnega tipa so. Če pogledamo, kakšnega tipa je funkcija *isDivisible*, lahko vidimo, da sprejme dva argumenta, ki sta celoštevilskega

tipa[3] in vrne logično vrednost tipa *Bool*.

```
isDivisible a b = (a `mod` b) == 0
-- preverimo tip v ghci
-- Prelude> :t isDivisible
-- isDivisible :: Integral a => a -> a -> Bool
```

Slika 2.2: Implicitno pridobljene informacije o tipih s pomočjo inference

2.1.6 Varnost jezika

Varnost jezika je velikokrat neformalno omenjena tudi kot varnost tipov. Avtor knjige [21] pravi, da je varnost jezika dosegljiva s pomočjo statičnega preverjanja skladnosti tipov. Le-to je zoprno konkretizirati v enoličnem pomenu. Obstaja več možnih pogledov na varnost jezika. Po definiciji iz knjige [21] je varen jezik tak, ki ščiti svoje abstrakcije. To lahko razumemo kot lastnost, da jezik, ki dovoli uporabo abstrakcij, dovoli programerju samo pravilno, varno uporabo teh abstrakcij. Kot primer poglejmo polja oziroma tabele (angl. array). Programer pričakuje, da se bo polje spreminjalo le preko operacij za delo s poljem. V programskem jeziku C pa lahko polja (ali katerokoli drugo podatkovno strukturo) spreminjamo z direktnim dostopom do pomnilnika. Prevajalnik za jezik C torej ne more zagotoviti varne uporabe polj. V programskem jeziku Java pa direktni dostop do pomnilnika ni mogoč in prevajalnik lahko dovoli le varno uporabo polj.

Prevajalniki oziroma mehanizmi za preverjanje skladnosti tipov v varnih jezikih naj bi zavrnili programe, ki vodijo v napačno ali nedefinirano vedenje. Programski jezik C, ki ga sicer uvrščamo med statično tipizirane jezike, je ilustrativen primer jezika, ki ni varen. Na primer, programerju je dostopna možnost uporabe kazalcev v pomnilnik ter njihova manipulacija, kar omogoča da programer spreminja pomnilnik, kjer prebivajo podatki neke druge podatkovne strukture. Ker so taki posegi dovoljeni, mehanizem za

[3]Inferenca tipov v programskem jeziku Haskell poizkuša izračunati najbolj splošen tip, zato tip argumentov ni omejen na konkreten tip, temveč na katerikoli celoštevilski tip.

preverjanje skladnosti tipov ne more zagotoviti predvidljivega delovanja podatkovne strukture, kateri pripada (nepredvidoma) spremenjeni pomnilnik.

Glede na varnost ločimo jezike na dve skupini, na *varne* in *ne-varne*. V tabeli 2.3 lahko vidimo primere programskih jezikov, kategoriziranih glede na tipiziranost in varnost.

	varni	nevarni
statični	Haskell, Java	C, C++
dinamični	Python, Perl	
netipizirani	(netipizirani) lambda račun	zbirnik

Slika 2.3: Programski jeziki glede na tipiziranost in varnost

Tipični predstavniki varnih jezikov programerju ne dopuščajo direktnega dostopa do pomnilnika. To večinoma gre z roko v roki z avtomatskim upravljanjem s pomnilnikom. Med varne prištevamo tudi bolj ali manj vse dinamične jezike, saj je dodatno delo, ki ga je poleg preverjanja skladnosti tipov potrebno narediti, da se zagotovi varnost, razmeroma majhno. Kot primer netipiziranih jezikov sta dodana še lambda račun brez tipov[4] in zbirnik[5], dve skrajnosti glede na stališče abstraktnosti.

2.2 Sistemi tipov

Sistem tipov je formalni sistem, ki definira pravila skladnosti tipov. Pravila določajo, kdaj se dva tipa ujemata oziroma kdaj sta skladna. Primer takšnega pravila je, da sta dva tipa skladna, kadar imata enako ime. Različni sistemi tipov torej definirajo različna pravila skladnosti tipov. Omeniti velja, da sta sistem tipov in algoritem za preverjanje skladnosti tipov ločeni stvari. Za konkreten sistem tipov lahko obstaja več (različnih) algoritmov za prever-

[4]Več o lambda računu lahko bralec izve v knjigi *Types and Programming Languages* [21] ter v učbeniku *An introduction to Lambda Calculi for Computer Scientists* [12].

[5]V splošnem imamo v zbirniku samo operacije nad vrednostmi, ki so v registrih, programerju pa je prepuščena popolna svoboda.

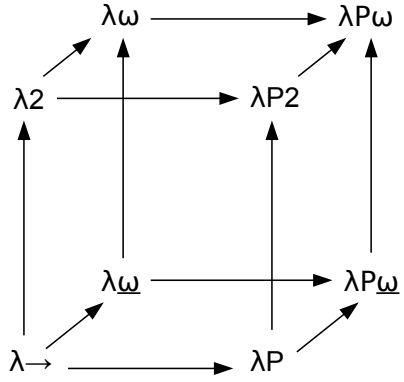
janje skladnosti tipov. Seveda pa mora algoritem za preverjanje upoštevati pravila, ki jih sistem tipov narekuje.

Na sisteme tipov lahko gledamo iz različnih zornih kotov. Lahko bi obravnavali sisteme tipov posameznih programskih jezikov. To bi bil sicer bolj praktičen pogled na sisteme tipov, vendar pa bi jih bilo težko sistematično, formalizirano in kompaktno predstaviti. Namesto konkretnih sistemov tipov si raje oglejmo razdelitev sistemov tipov, ki je opisana v članku [3].

Avtor je možne sisteme tipov razpel na tri osi in jih predstavil v obliki kocke. Kocko je poimenoval po lambda računu: *lambda kocka* (angl. *lambda cube*). Dimenzije pa prihajajo iz različnih družin abstrakcij:

- družina izrazov, ki so indeksirani z izrazi,
- družina izrazov, ki so indeksirani s tipi,
- družina tipov, ki so indeksirani s tipi ter
- družina tipov, ki so indeksirani z izrazi.

Na sliki 2.4 lahko vidimo lambda kocko.



Slika 2.4: Lambda kocka

Kot izhodišče je v spodnjem levem oglišču enostavno tipiziran lambda račun (angl. simply typed lambda calculus), ki je označen z $\lambda\rightarrow$. Enostavno tipiziran lambda račun je pripadnik družine abstrakcij, kjer so izrazi

indeksirani z drugimi izrazi. Zgornja ploskev predstavlja sisteme tipov, kjer so izrazi indeksirani s tipi oziroma sistemi tipov s polimorfizmom. Zadnja ploskev predstavlja dimenzijo, kjer so tipi indeksirani s tipi. Desna ploskev pa predstavlja nam zanimive abstrakcije, kjer so tipi indeksirani z izrazi - sisteme tipov z odvisnimi tipi.

Bolj rigorozna formalizacija sistemov tipov je izven obsega tega diplomskega dela, saj se bomo posvetili predvsem odvisnim tipom. Več o sistemih tipov najdemo v [9, 21].

2.3 Programski jeziki z odvisnimi tipi

V tem razdelku bomo na kratko predstavili programske jezike in paradigme programskih jezikov, ki so pogosto obogatene z odvisnimi tipi. Vrednotenje primernosti paradigme za temelje jezika z odvisnimi tipi je izven obsega tega diplomskega dela, ne glede na to pa je težko podati nepristransko oceno. Predstavili bomo paradigme, ki jih v povezavi z odvisnimi tipi najdemo v literaturi in v praksi.

2.3.1 Imperativni jeziki

Med imperativnimi jeziki najdemo Xanadu [26], ATS [27] in deloma tudi F* [25]. Imperativna paradigma je za raziskovanje odvisnih tipov precej zanimiva, saj je med najbolj pogosto uporabljenimi programskimi jeziki veliko imperativnih. Glavni razlog za uvedbo odvisnih tipov v imperativne jezike je seveda preverjanje pravilnosti z njihovo pomočjo. V imperativnih jezikih so poleg funkcij in podatkovnih struktur z odvisnimi tipi obogateni tudi stavki. Na primer, prirejanje nove vrednosti spremenljivki spremeni predpostavke, ki veljajo za program. S pomočjo avtomatskega preverjanja pravilnosti se v imperativnih jezikih poizkuša izogniti pogostim napakam, kot so na primer preliv izravnalnika (angl. buffer overflow), puščanje pomnilnika (angl. memory leak) ter nepravilni dostopi do pomnilnika. Rezultati na tem področju bi lahko imeli zelo dober vpliv na sorodna področja, pripomogli pa bi tudi k

pravilnosti programov nasploh.

Vendar pa je narava imperativnih jezikov težavna zaradi nekontrolirane uporabe (stranskih) učinkov[6], ki lahko nepredvidljivo vplivajo na pomen programov.

2.3.2 Objektno usmerjeni jeziki

V literaturi [19, 24] najdemo zametke paradigme objektne usmerjenosti v kombinaciji z odvisnimi tipi. Ideja za vpeljavo odvisnih tipov v jezik, ki je objektno usmerjen, je podobna kot pri imperativnih jezikih. Ključnega pomena pri vpeljavi odvisnih tipov v programski jezik skupaj s paradigmo objektne usmerjenosti je formalizacija objektov in razredov v povezavi z odvisnimi tipi. V [24] so objekti obravnavani kot interaktivni programi, metode in vmesniki pa kot interakcije med objekti. V [19] najdemo formalizacijo računa *vObj*, ki služi kot teoretična osnova uporabi odvisnih tipov skupaj z objektno usmerjenostjo. Na žalost pa konkretnega splošno namenskega jezika nismo zasledili.

2.3.3 Funkcijski jeziki

Paradigma funkcijskega programiranja je v svoji čisti različici zelo primerna za dokazovanje. Obravnava učinkov[6] na poseben način zagotavlja, da so ostali deli programa brez njih. Učinki, nedeterminizem in povezava z zunanjim svetom so nepredvidljivi in odvisni od trenutnega stanja. Nepredvidljivost morata prevajalnik in analiza skladnosti tipov seveda upoštevati, kar pa je velikokrat razlog za ločeno ali pa vsaj posebno obravnavo učinkov. To omogoča lažje razumevanje in lažje formalno obravnavanje pomenov programov. Iz tega stališča torej ni presenetljivo, da je veliko programskih jezikov, ki imajo svoje korenine v akademskem svetu, prav funkcijskih. Med njimi

[6] Učinek ali stranski učinek je operacija, ki poleg izračuna vrednosti spreminja stanje programa. Med učinke štejemo tudi funkcije, ki programu omogočajo komunikacijo z zunanjim svetom, na primer pisanje na zaslon. Za funkcije, ki ne izvajajo nobenih učinkov, pravimo, da so čiste.

najdemo programske jezike Agda [5], Epigram [18], Coq [4], DML [28], Cayenne [2] in nenazadnje tudi Idris [6].

Poglavje 3

Odvisni tipi

Kot ostali sistemi tipov, so tudi *sistemi tipov z odvisnimi tipi* odgovor na pereče vprašanje o pravilnosti programov. Idejo za nastanek odvisnih tipov je moč iskati v želji, da bi lahko prevajalniku na nek način podali lastnosti in morebitne zakonitosti, ki jih pričakujemo, da veljajo za naš program, prevajalnik pa bi jih (brez pomoči programerja tj. avtomatsko) preveril. Seveda pa stvar ni tako preprosta, kot morda izgleda na prvi pogled. Da bi se približali ideji o izražanju lastnosti programov, potrebujemo matematično formalizacijo, ki je "razumljiva" tudi prevajalniku. Odvisni tipi spadajo tudi na področje avtomatskega dokazovanja izrekov.

Matematično podlago odvisnim tipom najdemo v intuicionistični (tudi konstruktivistični) logiki in v intuicionistični teoriji tipov po Martin-Löfu [16], ki gradi na izomorfizmu, znanemu tudi kot Curry-Howardova enakovrednost: *tipi* so enakovredni *izjavam*, *programi* pa so enakovredni *dokazom izjav*.

Odvisni tipi so svoje ime dobili po lastnosti, da je lahko tip funkcije odvisen ne le od tipa argumentov, temveč tudi od njihovih vrednosti. Kadar tip vsebuje vrednosti oziroma je od njih odvisen, pravimo, da je *indeksiran* s temi vrednostmi. Odvisnosti izražamo v tipih, kadar pa definiramo nov (podatkovni) tip pa temu pravimo *vrsta* (angl. *kind*) oziroma *vrsta tipa* (angl. *kind of a type*). Vrsti bi lahko rekli tudi tip tipa. Vsak tip ima torej tudi lastnost, ki se ji reče vrsta. Primitivni tipi, kot na primer cela števila, imajo

vrsto enako $*$ ali $Type$, kar označuje, da so ti tipi primitivni (in niso na primer funkcije na nivoju tipov). V programskih jezikih s polimorfizmom, kjer srečamo tudi polimorfične podatkovne tipe, pa obstajajo tudi kompleksnejše vrste.

Kot primer si oglejmo vrsto tipa podatkovne strukture *seznam*, ki je enaka:

$$List : Type \to Type,$$

kar pomeni, da je tip seznama parametriziran z nekim tipom (*parametrični polimorfizem*). Seznamu bi lahko v tip dodali tudi velikost oziroma dolžino seznama. Takrat bi bila vrsta tipa podatkovne strukture seznam enaka

$$List : Nat \to Type \to Type,$$

pri čemer je Nat tip za naravna števila. Sedaj pravimo, da je seznam $List$ indeksiran z vrednostjo tipa Nat oziroma je indeksiran z naravnim številom. Opozoriti velja, da v programskih jezikih z odvisnimi tipi velikokrat ne ločimo med vrsto in med tipom in obema konceptoma pravimo kar tip. Razlog za to je moč iskati tudi v dejstvu, da programski jeziki z odvisnimi tipi pogosto podpirajo splošne funkcije na nivoju tipov, ki prav tako imajo svoj tip. V njihovih tipih bi tudi lahko bila uporabljena kaka druga funkcija na nivoju tipov, ki pa tudi ima svoj tip. Zaradi omenjene splošnosti je torej razlika med vrsto in tipom brez posebnega smisla.

V knjigi [1] najdemo formalizacijo odvisnih tipov. Če nanje pogledamo z malo manj formalnega vidika, lahko vidimo analogijo s predikatnim računom. Univerzalni kvantifikator se izraža kot tip odvisne funkcije, eksistencialni kvantifikator pa kot tip odvisnega para. V nadaljevanju si oglejmo oba.

3.1 Odvisne funkcije

Odvisne funkcije so funkcije, katerih *vrnitveni tip* je odvisen od vrednosti *parametrov* oziroma funkcije, ki imajo za vrnitveni tip odvisen tip.

Tipom odvisne funkcije pravimo tudi *tip odvisnega produkta* (angl. *dependent product type*) ali Π tip. Nanj lahko analogno gledamo kot na univerzalno kvantifikacijo.

Matematično lahko to zapišemo kot:

$$\Pi_{(x:A)} B(x)$$

kar v grobem ustreza univerzalni kvantifikaciji:

$$\forall x : A.\ B(x)$$

Izraziti želimo, da za vsak x tipa A, obstaja tip $B(x)$ oziroma tip B, ki je odvisen od vrednosti x. Vrsta tipa B je torej enaka $A \to Type$. Temu pravimo, da je tip B indeksiran z vrednostjo x.

To bi v programskem jeziku Idris[1] zapisali takole:

```
(x : A) -> B x
```

Primer implementacije odvisne funkcije:

```
replicate : (x : a) -> (n : Nat) -> List n a
replicate x Z = []
replicate x (S k) = x :: replicate x k
```

Funkcija *replicate* prejme argument x poljubnega tipa a in argument n tipa naravno število Nat, vrne pa seznam $List$ indeksiran preko vrednosti n in parametriziran s tipom a (seznam dolžine n, ki vsebuje elemente tipa a).

Če tip odvisne funkcije analogno obravnavamo kot univerzalno kvantifikacijo, lahko pomen tipa interpretiramo na sledeči način. Funkcija *replicate* za vsak x tipa a in vsak n, ki je naravno število, pridela seznam, ki vsebuje n elementov tipa a:

$$\forall x : a\ .\ \forall n : Nat.\ List\ n\ a$$

Njena implementacija je dokaz, da za vsak x tipa a in vsak n, ki je naravno število, obstaja seznam tipa $List\ n\ a$.

[1] Programski jezik Idris bo predstavljen v poglavju 4.

3.2 Odvisni pari

V funkcijskih jezikih poznamo podatkovne tipe parov in n-teric, ki so med drugim uporabni tudi za vračanje več vrednosti iz funkcij. Na področju odvisnih tipov pa poznamo tudi koncept odvisnih parov. Odvisni pari so podobni klasičnim parom, le da je tip drugega elementa para lahko odvisen od vrednosti prvega elementa para. V bistvu so odvisni pari dualen koncept odvisnim funkcijam, tako kot je eksistencialna kvantifikacija dualna univerzalni. Tipom odvisnih parov pravimo zato tudi *tip odvisne vsote* (angl. *dependent sum type*). Včasih pa je tip odvisnega para na kratko označen kar kot Σ tip:

$$\Sigma_{(x:A)} B(x)$$

Podobno kot pri odvisnih funkcijah lahko tudi za odvisne pare v grobem najdemo podobnost s predikatnim računom. Tip odvisnega produkta torej ohlapno ustreza eksistencialni kvantifikaciji:

$$\exists x : A.\ B(x)$$

Splošen tip odvisnega para izraža, da obstaja vrednost x tipa A, tako da velja $B(x)$ oziroma tako, da obstaja vrednost tipa $B(x)$.

V programskem jeziku Idris bi splošen zapis odvisnega para zapisali takole:

```
(x : A ** B x)
```

Primer odvisnega para iz priročnika za programski jezik Idris [7]:

```
vec : (n : Nat ** Vect n Int)
vec = (2 ** [3, 4])
```

Gre za odvisni par, kjer je prvi element n naravno število, drugi odvisni element pa vektor dolžine n. Tip lahko interpretiramo kot izjavo: obstaja tako naravno število n, da obstaja tudi vektor dolžine n.

Odvisni pari so zelo uporabni za dokazovanje lastnosti funkcij. Namesto zgolj ene vrednosti funkcija lahko vrne odvisni par, ki ima kot prvi element

vrednost, kot drugi (odvisni) element pa dokaz o lastnosti funkcije. S tem bi na primer lahko napisali tip funkcije za urejanje seznama, ki bi vračala urejen seznam in dokaz, da je seznam res urejen:

```
sort : (xs : List Nat) -> (ys : List Nat ** IsSorted ys)
```

Tip *IsSorted ys* v tem primeru predstavlja tip dokaza, da je seznam ys urejen. Vrednost tega tipa lahko naredimo le, če je seznam res urejen. Definicijo podobnega tipa bomo spoznali v nadaljevanju. Sedaj želimo le pokazati, da lahko s tipi funkcij izrazimo lastnosti in zahteve, ki jih želimo dokazati. V tem primeru bi lahko tip prevedli v izjavo:

$$\forall xs : List\ Nat.\ \exists ys : List\ Nat.\ IsSorted\ ys$$

Drugače povedano, za vsak seznam naravnih števil xs obstaja tak seznam naravnih števil ys, da velja *IsSorted ys* (da je ys urejen).

3.3 Neodločljivost

Odvisni tipi so izrazno zelo močni in lahko vsebujejo tudi programe za izračun tipov. Preverjanje skladnosti tipov tako lahko (v primeru kompleksnih tipov) pomeni preverjanje enakovrednosti dveh programov, kar pa je v splošnem neodločljiv problem [13]. Težava z neodločljivostjo se lahko omili z uporabo konzervativnih strategij, ki dajo pravilen rezultat o skladnosti tipov ali pa se odločijo negativno. S tem zagotovimo, da preverjanje skladnosti tipov ne sprejme napačnih programov, lahko pa se zgodi, da zavrne nekatere pravilne.

Poglavje 4

Programski jezik Idris

Programski jezik Idris je splošno namenski programski jezik, ki podpira polne odvisne tipe. Njegov sistem tipov je izrazno dovolj močan, da lahko definiramo poljubno kompleksne tipe in dokaze.

Idris je zasnovan na osnovi programskega jezika *Haskell*. V članku [6], kjer je opisana zasnova in implementacija, je Idris predstavljen kot odgovor na vprašanje:

"Kaj če bi Haskell imel polne odvisne tipe?"

Ta zamisel je snovalce Idrisa vodila k programskemu jeziku, ki ima v osnovi dokaj podobne lastnosti. Zaobjema paradigmo funkcijskega programiranja in je, tako kot Haskell, "čist". S tem mislimo, da so kakršni koli učinki nedovoljeni. Funkcije lahko vrednost izračunajo samo preko vhodnih parametrov. S tem pridobimo lepo lastnost – *sklicevalno preglednost* (angl. *referential transparency*), ki je pomembna, da lahko o pomenu programa sklepamo.

Da pa lahko pišemo uporabne programe, mora obstajati način uporabe učinkov. Tudi v tem pogledu Idris gradi na skorajda identičnemu sistemu monadičnih učinkov kot Haskell. Po zasnovi programskega jezika Haskell je narejena tudi podpora *razredom tipov* (angl. *type classes*), prevzete pa je tudi veliko sintakse (na primer *do*-notacija). Vendar pa poleg sistema

tipov najdemo med jezikoma bistveno razliko: Idris sledi načelu *striktnega ovrednotenja* (angl. *strict evaluation*), kar pomeni, da se izrazi poenostavijo – ovrednotijo takoj, ko je to mogoče.

Idris ima sistem tipov, ki podpira polne odvisne tipe. Tipi so torej lahko odvisni od kakršne koli vrednosti. V drugih programskih jezikih, ki le delno podpirajo odvisne tipe, so lahko tipi, na primer odvisni le od naravnih števil in podobno. Ponavadi v programskih jezikih obstaja stroga ločnica med tipi in izrazi [8]. Ti imajo velikokrat ločeno sintakso za opisovanje tipov in izrazov, Idris in tudi drugi programski jeziki s polnimi odvisnimi tipi pa te ločnice nimajo. Ker lahko v tipih nastopajo poljubne vrednosti in izrazi, se s tem prepleta tudi sintaksa.

Programski jezik Idris vsebuje izrazno dovolj močan sistem tipov, da ga lahko uporabljamo kot ogrodje za dokazovanje. S pomočjo interaktivnega načina dokazovanja lahko služi kot *pomočnik za dokazovanje* (angl. *proof assistant*). Vendar pa je bil Idris po besedah avtorjev zasnovan z namenom praktične uporabe, za reševanje realnih problemov. Razlog za izbiro tega programskega jezika za to diplomsko delo je vsekakor funkcijska usmerjenost, podobnost s programskim jezikom Haskell ter bolj praktična usmerjenost. Slednja je pomembna, saj je namen tega diplomskega dela tudi pokazati uporabnost programskih jezikov z odvisnimi tipi.

4.1 Pregled sintakse

4.1.1 Tipi in definicije funkcij

Ker je Idris funkcijski programski jezik, si najprej poglejmo, kako se definirajo funkcije. Definicija funkcije se začne z deklaracijo tipa. Ta vsebuje *ime funkcije*, nato pa dvopičje in tip. Tako kot v programskem jeziku Haskell so tudi tukaj vse funkcije obravnavane tako, da sprejmejo samo en argument, vračajo pa novo funkcijo, ki sprejme naslednji argument. Temu principu v angleški literaturi pravimo *currying*, saj ga je dodelal Haskell Curry. Zapis tipa odraža to lastnost. Posamezni tipi argumentov so ločeni z znakom `->`,

zadnji pa je tip, ki ga funkcija vrača, ko ji podamo vse argumente. Zaradi prej omenjenega principa lahko funkciji podamo manj argumentov in kot rezultat dobimo novo funkcijo, ki ji je potrebno podati še preostanek argumentov.

Primer deklaracije tipa lahko vidimo na primeru funkcije *plus*.

```
plus : Nat -> Nat -> Nat
```

Zaradi Curryevega principa obravnave funkcij je operator za zapis funkcijskih tipov (`->`) desno asociativen. Tip funkcije *plus* bi lahko ekvivalentno zapisali tudi kot tip funkcije, ki sprejme en argument in vrne funkcijo, ki sprejme še naslednjega.

```
plus : Nat -> (Nat -> Nat)
```

Ker so tipi funkcij desno asociativni, ti oklepaji niso potrebni in jih lahko izpustimo.

Kot argumente pa lahko sprejmemo tudi funkcije.

```
apply : (a -> b) -> a -> b
```

V tem primeru pa z oklepaji nakažemo, da kot prvi argument sprejmemo vrednost, ki ima tip `(a -> b)`.

Da definiramo celotno funkcijo, moramo k deklaraciji tipa funkcije dodati še implementacijo oziroma telo funkcije. To storimo tako, da navedemo ime funkcije, imena argumentov, enačaj ter izraz ki predstavlja telo funkcije.

```
apply : (a -> b) -> a -> b
apply f x = f x
```

V definicijah funkcij lahko uporabljamo tudi *prepoznavanje vzorcev* (angl. *pattern matching*). S tem lahko na različne načine obravnavamo vrednosti, ki so bile ustvarjene z različnimi podatkovnimi konstruktorji.

```
plus : Nat -> Nat -> Nat
plus Z m = m
plus (S k) m = S (plus k m)
```

Na primeru funkcije *plus* vidimo seštevanje dveh števil. Prvi argument s prepoznavanjem vzorcev razdelimo na dva dela takrat, ko ima vrednost Z in takrat, ko ima vrednost $(S\ k)$. V vrednost k se nam shrani argument, ki je bil uporabljen skupaj s konstruktorjem S.

Znotraj deklaracije tipa lahko vrednostim argumentov pripišemo ime. To je uporabno, kadar želimo vrednost argumenta uporabiti v nekem odvisnem tipu. Na primer, funkcija *replicate* argument tipa a ponovi n-krat – vrne seznam, ki ima v tipu shranjeno dolžino n. V standardni knjižnici seznamom, ki imajo v tipu shranjeno velikost oziroma dolžino, pravijo vektor.

```
replicate : (n : Nat) -> a -> Vect n a
```

Pri definiciji funkcij lahko uporabljamo tudi omejitve z razredi tipov. Za tip argumenta lahko zahtevamo, da obstaja instanca določenega razreda tipov. S tem v telesu funkcije dovolimo uporabo funkcij, ki so definirane v razredu tipov. Eden izmed razredov tipov je na primer *Ord*, ki definira funkcije za urejenost. Kot primer poglejmo implementacijo funkcije *max*. V tipu zahtevamo, da za tip a obstaja instanca razreda *Ord*. To storimo z operatorjem `=>`, ki ga postavimo pred tip funkcije. S tem v funkciji *max* dovolimo uporabo funkcije `>`.

```
max : Ord a => a -> a -> a
max x y = if x > y
            then x
            else y
```

Funkcija *max* je torej definirana za vsak tip a, za katerega obstaja instanca razreda *Ord*.

4.1.2 Deklaracija podatkovnih tipov

Podatkovne tipe (tudi odvisne) deklariramo s ključno besedo *data*, ki ji sledi *ime podatkovnega tipa*. Nato sledi *dvopičje* in *vrsta tipa*, ki lahko vsebuje tudi tipe vrednosti, od katerih je tip, ki ga deklariramo, odvisen. Nato sledi

ključna beseda *where* in z belimi presledki zamaknjene deklaracije podatkovnih konstruktorjev. Deklaracija konstruktorja sestoji iz imena in tipa, ki ga konstruktor ima.

```
data List : Type -> Type where
  Nil : List a
  (::) : a -> List a -> List a
```

Zgoraj vidimo deklaracijo podatkovnega tipa *List*. Njegova vrsta je enaka

$$Type \to Type,$$

ima pa dva podatkovna konstruktorja: *Nil* in *(::)*.

4.1.3 Sintaksa vgrajenih izrazov

V tem razdelku si bomo ogledali sintakso in primere pogosto uporabljenih izrazov, ki so vgrajeni v programski jezik Idris.

Vezava "let"

Vezava "let" nam omogoča, da vrednost na desni strani enačaja *povežemo* (angl. *bind*) z imenom na levi.

```
let f = plus Z (S Z)
```

Izraz "if"

Skorajda vsak imperativni ali proceduralni programski jezik ima vgrajen stavek "if". V funkcijskih jezikih pa namesto stavka najdemo izraz "if". Ima podobno strukturo, le da se (glede na pogoj) izračuna v vrednost.

```
max : Ord a => a -> a -> a
max x y = if x > y
            then x
            else y
```

Kot primer izraza "if" si poglejmo funkcijo *max*, ki vrne večje od obeh števil.

Izraz "case of"

Idris, Haskell in tudi drugi programski jeziki podpirajo prepoznavanje vzorcev. S pomočjo prepoznavanja vzorcev se da narediti podoben izraz kot "if", le da je ta bolj splošen. V prvi del (med *case* in *of*) vnesemo vrednost (ali izraz), ki se nato izračuna. Nato se izvede primerjava in prepoznavanje vzorcev s posameznimi vrsticami v drugem delu.

Kot primer si poglejmo funkcijo *max*, ki namesto izraza "if" uporablja izraz "case of".

```
max : Ord a => a -> a -> a
max x y = case (x > y) of
            True => x
            False => y
```

Kot smo omenili, lahko z izrazom "case of" prepoznavamo tudi vzorce in izluščimo vrednosti iz podatkovnih konstruktorjev. Primer tega lahko vidimo na funkciji *isZero*, ki vrednost vhodnega argumenta n obravnava z izrazom "case of" in izlušči vrednosti.

```
isZero : Nat -> Bool
isZero n = case n of
             Z => True
             (S k) => False
```

Omeniti velja, da je prepoznavanje vzorcev skorajda identično tistemu, ki smo ga spoznali pri definicijah funkcij.

Pogled "with"

Mnogokrat želimo izvajati prepoznavanje vzorcev nad vmesno vrednostjo, ki jo zgradimo iz vhodnih argumentov. Semantika pogleda "with" je podobna semantiki izraza "case". Vendar pa je pogled "with" drugačen, saj z njim upoštevamo, da lahko prepoznavanje vzorcev na vrednosti z odvisnimi tipi spremeni pomen programa.

```
isLessOrEq : Nat -> Nat -> Bool
isLessOrEq k j with (compare k j)
  | LT = True
  | EQ = True
  | GT = False
```

Zaradi enostavnosti je prikazan primer, ki bi ga lahko implementirali tudi z izrazom "case".

4.1.4 Implicitni parametri in okolje "using"

Programski jezik Idris omogoča, da lahko funkcije sprejmejo tudi *implicitne parametre*. Programerju njihovih vrednosti ni potrebno podajati, saj jih lahko iz konteksta izlušči prevajalnik. Parameter označimo kot implicitnega tako, da ga podamo znotraj zavitih oklepajev. Na primeru vidimo funkcijo *length*, ki kot implicitni parameter sprejme dolžino vektorja in jo vrne kot izhodno vrednost.

```
length : {n : Nat} -> Vect n a -> Nat
length {n=len} vect = len
```

Primer klica funkcije *length* iz interaktivne ukazne vrstice:

```
idris> length (1 :: 2 :: 3 :: Nil)
3 : Nat
```

Nad implicitnimi parametri lahko tako kot nad običajnimi izvajamo prepoznavanje vzorcev. S tem lahko implementiramo funkcijo *isEmpty*, ki vrne resnično logično vrednost, če je vektor prazen.

```
isEmpty : {n : Nat} -> Vect n a -> Bool
isEmpty {n=Z} xs = True
isEmpty {n=(S k)} xs = False
```

Kadar pa v več funkcijah uporabljamo enake implicitne parametre, lahko funkcije prestavimo v okolje "using", ki avtomatsko vstavi implicitne parametre. Kot primer si poglejmo ekvivalentno funkcijo *length* z uporabo okolja "using".

```
using (n : Nat)
  length : Vect n a -> Nat
  length {n=len} vect = len
```

4.1.5 Uporabni podatkovni tipi iz standardne knjižnice

Tip "Maybe" in predstavitev prazne vrednosti

Pogosto imamo pri pisanju programov opravka s *praznimi vrednostmi*. Imperativni in proceduralni jeziki poznajo vrednost *Null*, v funkcijskih jezikih pa so prazne vrednosti zaradi nevarnosti in nedefiniranega pomena prepovedane. Da bi lahko predstavili prazne vrednosti in jih tudi varno obravnavali, je v standardni knjižnici definiran podatkovni tip *Maybe*. Z njim bomo predstavili tip, ki lahko ima vrednost ali pa je prazen.

```
data Maybe: Type -> Type where
  Nothing : Maybe a
  Just : a -> Maybe a
```

Podatkovni tip *Maybe* ima dva podatkovna konstruktorja. Prvi (*Nothing*) predstavlja prazno vrednost, drugi (*Just*) pa zaobjame neko obstoječo vrednost.

S tako predstavitvijo praznih vrednoti lahko napišemo funkcijo, ki vrača glavo seznama tj. prvi element seznama, če jo lahko.

```
head : List a -> Maybe a
head [] = Nothing
head (x :: xs) = Just x
```

Tip "Either"

Tip *Maybe*, ki predstavlja prazne ali neprazne vrednosti, lahko posplošimo na tip, ki predstavlja ali vrednosti tipa a ali vrednosti tipa b, pri čemer sta tipa a in b lahko poljubna.

```
data Either : Type -> Type -> Type where
  Left : (l : a) -> Either a b
  Right : (r : b) -> Either a b
```

V splošnem predstavlja tip *Either* izključujočo izbiro (disjunktno unijo) med dvema tipoma, s pomočjo prepoznavanja vzorcev pa lahko vemo, s katerim izmed konstruktorjev *Left* in *Right* je bila izbira narejena in posledično tudi kateri tip predstavlja. V teoriji tipov pravimo tipu *Either* tudi *tip produkta*.

S takšno definicijo lahko ta tip uporabimo za predstavitev napak. Kot primer si oglejmo funkcijo *head*, ki vrača glavo seznama ali pa vrne sporočilo o napaki.

```
head : List a -> Either String a
head [] = Left "No head of empty list"
head (x :: xs) = Right x
```

Kot smo omenili na začetku, je *Either* posplošitev tipa *Maybe*. Slednjega bi lahko ekvivalentno predstavili kot vrednost, ki ima tip `Either () a` oziroma kot izbiro med tipom () in nekim tipom a. Tip () je tip enote (angl. unit type), ki ima le eno vrednost – (). Interpretiramo ga lahko kot prazno n-terico oziroma 0-terico. V funkcijskih jezikih nadomešča tip *void*, ki ga ponavadi najdemo v imperativnih jezikih.

4.2 Dokazovanje

V tem razdelku si bomo ogledali, kako s pomočjo odvisnih tipov dokazujemo lastnostij.

4.2.1 Dokazovanje s pomočjo odvisnih tipov

S pomočjo izrazne moči, ki jo ima sistem tipov v programskem jeziku Idris, lahko na konstruktivističen način dokazujemo izjave. Konstruktivistični način dokazovanja izjav najdemo v intuicionistični oziroma konstruktivistični logiki. Resničnost izjave lahko dokažemo tako, da napravimo oziroma skonstruiramo dokaz. Opozoriti je potrebno na dejstvo, da neobstoj dokaza o resničnosti ne implicira neresničnosti izjave. Neresničnost izjave je v konstruktivistični logiki potrebno dokazati tako, da skonstruiramo dokaz o protislovju.

Zaradi Curry-Howardove enakovrednosti lahko izjave podamo v obliki tipov funkcij, dokazi pa so implementacije teh funkcij.

Kot primer si oglejmo definicijo enakosti, ki jo najdemo v standardni knjižnici programskega jezika Idris.

```
data (=) : a -> b -> Type where
  refl : x = x -- enakovredno kot refl : (=) x x
```

Enakost lahko predstavimo kot podatkovni tip *(=)*, ki je odvisen od dveh vrednosti tipov a in b. Znak = v tem primeru ni operator ampak ime podatkovnega tipa, programski jezik Idris pa dopušča, da ga lahko uporabljamo tudi infiksno. Dokaz o enakosti dveh vrednosti lahko naredimo le s konstruktorjem *refl*, ki v tipu zahteva enakost vrednosti. Edini način, da naredimo dokaz o enakosti dveh vrednosti, je torej s konstruktorjem *refl*. To pa lahko naredimo le takrat, ko sta vrednosti dejansko enaki.

```
twoEqualsTwo : 2 = 2
twoEqualsTwo = refl
```

Enakost lahko izrazimo tudi med dvema izrazoma, če se izračunata v isto vrednost.

```
onePlusTwo: 1 + 2 = 3
onePlusTwo = refl
```

Pri tem seveda velja omejitev, da algoritem za preverjanje skladnosti tipov vseh izrazov ne zna poenostaviti in je včasih potrebna pomoč programerja.

Pokažimo, kako se gradijo dokazi. Za primer dokazovanja z odvisnimi tipi bomo pokazali, da je enakost *ekvivalenčna relacija*. Pokazali bomo, da za enakost veljajo *refleksivnost*, *simetričnost* in *tranzitivnost*. Po refleksivnosti imenujemo tudi (edini) podatkovni konstruktor za enakost *refl*. Pri vsaki si bomo ogledali matematično izjavo in implementacijo v programskem jeziku Idris. S tem bo na kratko prikazana tudi Curry-Howardova enakovrednost.

Refleksivnost

Izjava:

$$x = x$$

Tip in implementacija funkcije:

```
eqRefl : {x : a} -> x = x
eqRefl = refl
```

Simetričnost

Izjava:

$$x = y \implies y = x$$

Tip in implementacija funkcije:

```
eqSym : {x, y : a} -> x = y -> y = x
eqSym refl = refl
```

Tranzitivnost

Izjava:

$$x = y \land y = z \implies x = z$$

Tip in implementacija funkcije:

```
eqTrans : {x, y, z : a} -> x = y -> y = z -> x = z
eqTrans refl refl = refl
```

Tako kot lahko zapišemo izjave, ki jih ne moremo dokazati, lahko zapišemo tudi tip, za katerega ne znamo narediti dokaza. Napišemo lahko funkcijo s tipom, ki ustreza izjavi $1 = 2$, vendar pa ne moremo narediti dokaza. Če bi poskusili kar s konstruktorjem *refl*, bi dobili napako zaradi neujemanja tipov.

```
wrong : 1 = 2
wrong = refl
```

Ob prevajanju funkcije *wrong* bi prevajalnik sporočil napako, da ne zna poenotiti 2 in 1:

```
Can't unify
        2 = 2
with
        1 = 2
Specifically:
        Can't unify
                2
        with
                1
```

4.2.2 Prepisovanje tipov z dokazi o enakosti

Z dokazi o enakosti lahko dokazujemo tudi druge stvari. Zaradi lastnosti, ki jih ima enakost, lahko z dokazom o enakosti spremenimo tip dokaza, ki ga gradimo.

V naslednjem primeru želimo dokazati, da sta naslednika dveh enakih naravnih števil x in y tudi med seboj enaki naravni števili.

```
succEqual : {x, y : Nat} -> x = y -> S x = S y
succEqual prf = rewrite prf in refl
```

Tip funkcije zahteva, da je izhodni tip enak $S\ x = S\ y$. Imamo pa na voljo dokaz, ki pravi, da sta x in y enaka. S pomočjo vgrajenega mehanizma za prepisovanje tipov lahko z dokazom o enakosti x in y spremenimo tip iz S

$x = S\ y$ na $S\ x = S\ x$. To pa lahko dokažemo kar s konstruktorjem *refl*, saj sta si $S\ x$ med seboj enaka.

Potek prepisa tipa lahko orišemo s pomočjo interaktivnega dokazovalnika, ki je del orodij za programski jezik Idris. Prikazan je ciljni tip dokaza, ki se po uporabi prepisovanja spremeni:

```
----------              Assumptions:              ----------
 x : Nat
 y : Nat
 prf : x = y
----------                 Goal:                  ----------
{hole0} : S x = S y

-Main.succEqual1> rewrite prf

----------              Assumptions:              ----------
 x : Nat
 y : Nat
 prf : x = y
----------                 Goal:                  ----------
{hole1} : S x = S x
```

4.2.3 Preverjanje totalnosti

Ogledali smo si, kako lahko s pomočjo odvisnih tipov izjave izrazimo kot tipe funkcij in jih nato dokažemo z implementacijo teh funkcij. Da pa so ti dokazi resnično veljavni, pa je potrebno zagotoviti, da so te funkcije *totalne*. Za funkcijo pravimo, da je totalna, če zanjo veljata:

- *pokritost* – lastnost, da je funkcija definirana za vse možne vhodne podatke,
- *ustavljivost* – lastnost, da funkcija vrne rezultat za vse možne vhodne podatke oziroma da se izračun funkcije ustavi.

Pokritost lahko prevajalnik preveri s pomočjo zahteve, da je funkcija definirana za vse možne podatkovne konstruktorje vhodnih argumentov. Težji problem je preverjanje ustavljivosti. Na splošno je to neodločljiv problem [13], vendar pa je z uporabo konzervativnih strategij dosežena uporabnost preverjanja ustavljivosti. Omeniti velja, da je predpogoj za totalnost funkcije tudi, da so vse uporabljene pomožne funkcije prav tako totalne.

Pri prevajanju programov s prevajalnikom za programski jezik Idris preverjanje totalnosti ni samodejno vključeno. S pomočjo ključne besede *total* pa nam je omogočeno, da za vsako funkcijo posebej zahtevamo preverjanje totalnosti.

```
total isZero : Nat -> Bool
isZero n = case n of
             Z => True
             (S k) => False
```

Pogosto je nepraktično, da moramo pri vsaki funkciji dopisovati, da želimo zanjo preverjati totalnost. Obstaja tudi način, ki nam omogoča, da za celotno datoteko izvorne kode vklopimo preverjanje totalnosti. To storimo tako, da na vrhu datoteke pripišemo ključni besedi `%default total`. S tem načinom je prevedena in preverjena vsa izvorna koda, ki jo bomo obravnavali v naslednjih poglavjih tega diplomskega dela, ko bomo dokazovali njeno pravilnost.

Poglavje 5

Programi z odvisnimi tipi

Pravilnosti programov lahko opazujemo na več načinov. V tem poglavju so predstavljene pogosto uporabljene podatkovne strukture in algoritmi za njihovo manipulacijo.

Najprej bomo spoznali naravna števila in njihovo podatkovno predstavitev. Nato bomo definirali povezan seznam. Z uporabo odvisnih tipov, naravnih števil in povezanega seznama bomo skonstruirali vektor – povezan seznam z dolžino shranjeno v tipu. Nato bomo pokazali, kako lahko s pomočjo odvisnih tipov izražamo lastnosti programov. Skonstruirali bomo *sklad* in *vrsto*, pokazali implementacije poznanih operacij in nato podali še drugačno, preverjeno implementacijo.

V zadnjem delu tega poglavja bo prikazana konstrukcija urejenega seznama in algoritmov za urejanje. Pokazali bomo, kako lahko skonstruiramo nov podatkovni tip, ki vsebuje potrebne dokaze, in pa kako lahko obstoječim ne-odvisnim tipom *pripnemo* dokaze o neki lastnosti. Na ta način bomo prikazali podatkovni tip *urejen seznam* ter način, kako navadnemu seznamu dodamo odvisni "dokaz" o urejenosti. Podobno kot za urejenost, bomo pokazali tudi dokaz, da je nek seznam *permutacija* nekega drugega seznama. Oba dokaza bosta pokazana na primeru algoritma za urejanje z vstavljanjem (*insertion sort*), pokazali pa bomo tudi, kako oba dokaza združiti. Z obema združenima dokazoma bomo avtomatsko dokazali pravilnost algoritma za

urejanje.

5.1 Naravna števila

Najprej si oglejmo, kako so v progamskem jeziku Idris skonstruirana naravna števila. (Za konstrukcijo naravnih števil ne potrebujemo izrazne moči odvisnih tipov in jih lahko napravimo v večini programskih jezikov).

```
data Nat : Type where
  Z : Nat
  S : Nat -> Nat
```

Naravna števila so predstavljena s pomočjo naravne indukcije in sledijo Peanovim aksiomom.

- Prvo naravno število je 0 in je predstavljeno s podatkovnim konstruktorjem Z.
- Za vsako naravno število n, obstaja naslednik $S(n)$, ki je prav tako naravno število. Naslednik števila n je predstavljen podatkovnim konstruktorjem S n.

Tako je npr. število 3 predstavljeno s S (S (S Z)).

5.2 Povezani seznam

V večini funkcijskih programskih jezikov so *povezani seznami* zgrajeni induktivno:

```
data List : Type -> Type where
  Nil : List a
  (::) : a -> List a -> List a
```

Prvi konstruktor Nil predstavlja prazen seznam. Drugi konstruktor (::) (včasih mu rečemo tudi *cons*) pa sprejme vrednost tipa a in nek seznam

tipa *List a* ter vrne nov seznam tipa *List a*. Semantika konstruktorja (::) je pripenjanje podanega elementa na začetek podanega seznama, pogosto pa se uporablja kot infiksni operator, npr.: $x :: xs$.

Seznam je torej definiran parametrično: za vsak tip a lahko napravimo seznam tipa *List a*. Na to kaže že vrsta tipa *List*, ki je $Type \to Type$, drugače povedano, *List* je *konstruktor tipa*, ki kot parameter sprejme tip in iz njega napravi nov tip.

Nad seznamom navadno želimo definirati funkciji *head* in *tail*, pri čemer prva vrača prvi element v seznamu – glavo, druga pa preostanek seznama – rep.

```
head : List a -> a
head Nil = ???
head (x :: xs) = x

tail : List a -> List a
tail Nil = ???
head (x :: xs) = xs
```

Tukaj pa naletimo na težavo. Glava in rep praznega seznama nista definirana, funkciji sicer imata tip *List a*, kar pomeni, da lahko sprejmeta kakršenkoli seznam, tudi prazen. V programskih jezikih, ki podpirajo izjeme, je standardna praksa, da se ob izračunu glave ali repa praznega seznama sproži zjema.

Programski jezik Idris pa izjem v osnovi ne podpira (izjeme sicer obstajajo kot del razširitve za delo z učinki), torej funkcij *head* in *tail* na praznem seznamu sploh ne moremo definirati. To bi pomenilo, da bi ob izvajanju programa ne bilo druge možnosti, kot da se program ob izračunu glave ali repa praznega seznama nasilno ustavi. To bi sicer lahko omilili z uporabo tipa *Maybe*, tako da vrednosti vračajo vrednost zavito v *Just*, ob praznem seznamu pa vrednost *Nothing*.

Ta rešitev sicer prepreči napako, vendar pa se nekoliko spremeni pomen programa. Programer je s tem prisiljen, da se ukvarja z dekonstrukcijo *Maybe*

```
head : List a -> Maybe a
head Nil = Nothing
head (x :: xs) = Just a
```

in mora vedno preverjati, če je vrednost zavita v *Just*. Rešitev z *Maybe* problema v bistvu ne reši, ampak le prestavi na drugo mesto. Z uporabo odvisnih tipov pa bomo napravili rešitev, ki bo ohranjala pomen, hkrati pa bo preverjanje skladnosti tipov zagotavljajo, da ne moremo napisati programa, ki bi hotel izračunati glavo praznega seznama.

5.3 Povezani seznam z dolžino

Opazili smo, da je tip funkcije *head* : $List\ a \rightarrow a$ presplošen, ker sprejme kakršenkoli seznam. Želeli bi, da bi lahko v tipu povedali, kakšne sezname lahko funkcija *head* sprejme. Bolj natančno, s tipom želimo zagotoviti, da funkcija sprejme samo neprazne sezname. Skonstruirali bomo seznam, ki ne le, da bo parametriziran z nekim tipom, temveč bo indeksiran z naravnimi števili. Novo podatkovno strukturo bomo poimenovali *vektor*, definirana pa je sledeče (povzeto po članku [6]):

```
data Vect : Nat -> Type -> Type where
  Nil : Vect Z a
  (::) : a -> Vect n a -> Vect (S n) a
```

Glavna razlika v primerjavi s seznamom je v vrsti podatkovnega tipa, ki je pri vektorju enak $Nat \rightarrow Type \rightarrow Type$. Torej lahko konkreten tip vektorja ustvarimo le tako, da podamo neko naravno število, ki predstavlja informacijo o dolžini vektorja in nek konkreten tip.

Vektor ima konstruktor *Nil*, ki ustvari prazen vektor, katerega dolžina je seveda enaka Z. Bolj zanimiv je konstruktor ::, ki sprejme neko vrednost tipa a ter vektor dolžine n (tipa $Vect\ n\ a$), vrne pa vektor dolžine $S\ n$, torej za ena večji od podanega vektorja. S tem smo definirali odvisnost med podanimi in

novo nastalim vektorjem ter izrazili, da se dolžina vektorja ob pripenjanju elementa na začetek poveča za ena.

Sedaj, ko imamo dolžino vektorja "shranjeno" v tipu, lahko zapišemo tip funkcije *head* tako, da sprejme le vektorje z dolžino večjo od 0:

```
head : Vect (S n) a -> a
head (x :: xs) = x
```

Edini način, da ustvarimo vrednost vektorja, katerega dolžina je naslednik nekega naravnega števila, je preko konstruktorja :: (Najmanjši naslednik nekega naravnega števila je $S\ Z$). Prav to dejstvo pa nam zagotavlja, da v programu, kjer so tipi skladni, kot argument funkciji *head* ne moremo sprejeti praznega vektorja in lahko vedno varno vrnemo glavo seznama.

Na podoben način implementiramo tudi funkcijo *tail*.

```
tail : Vect (S n) a -> Vect n a
tail (x :: xs) = xs
```

Funkcija tako sprejme le vektorje z dolžino, ki je naslednik nekega naravnega števila n, vrača pa vektorje z dolžino n, torej za ena krajše. V tipu *Vect (S n) a* $\rightarrow$ *Vect n a* se odraža odvisnost izhodnega tipa od dolžine vhodnega vektorja.

Oglejmo si še primer uporabe funkcije *head*:

```
printHead : Vect n String -> IO ()
printHead Nil = putStrLn "Can not print head of an empty vector."
printHead (x :: xs) = putStrLn (head (x :: xs))


main : IO ()
main = do
  vector <- getVectorFromTheRealWorld
  printHead vector
```

V vstopni funkciji *main* s pomočjo funkcije *getVectorFromTheRealWorld* preberemo vektor. Funkcija *getVectorFromTheRealWorld* lahko predstavlja

branje vektorja iz standardnega vhoda ali pa kompleksnejšo zahtevo v podatkovno bazo. Ker to ni relevantno, je implementacija te funkcije izpuščena. Bistvena pa je funkcija *printHead*, ki ob nepraznem vektorju izpiše glavo, ob praznem pa napako. Na prvi pogled ne vidimo bistvene prednosti, vendar pa je potrebno poudariti, da v primeru, ko prejmemo prazen vektor (*printHead Nil = ...*) sploh ni možna uporaba funkcije *head*. To zagotavlja prevajalnik. Če pa na primer razčlenitve vektorja ne bi naredili oziroma ne bi preverili, da je vektor neprazen, nam prevajalnik prav tako ne bi dovolil uporabe funkcije *head*. Ob prevajanju

```
printHead : Vect n String -> IO ()
printHead xs = putStrLn (head xs)
```

bi dobili naslednjo napako

```
When elaborating right hand side of printHead:
When elaborating an application of function head:
        Can't unify
                Vect n String
        with
                Vect (S n) String

        Specifically:
                Can't unify
                        n
                with
                        S n
```

To pomeni, da prevajalnik ni prepričan, da je n naslednik naravnega števila oziroma drugače povedano, ni prepričan, da vektor vsebuje vsaj en element.

5.4 Sklad

Ena izmed bolj uporabljanih podatkovnih struktur je sklad. V nedokazani obliki je enak seznamu. Če pa uporabimo različico, ki ima velikost sklada shranjeno v tipu, pa je na las podoben vektorju. Osnovni in dokazani različici sta povzeti po članku [23].

```
data Stack : Type -> Type where
  StNil : Stack a
  StCons : (x : a) -> Stack a -> Stack a
```

Sklad običajno spremljajo še standardne funkcije za delo z njim. Zaradi enostavnosti bomo pokazali le implementacije na različici sklada, ki je po definiciji tipa enakovredna seznamu. Namenjene so boljšemu razumevanju delovanja seznama, saj ob dodajanju dokazov postane koda manj berljiva.

```
push : (x : a) -> Stack a -> Stack a
push x xs = StCons x xs

pop : Stack a -> (a, Stack a)
pop (StCons x xs) = (x, xs)

peek : Stack a -> a
peek (StCons x xs) = x

drop : Stack a -> Stack a
drop (StCons x xs) = xs
```

Podobno kot pri seznamu, funkcije *pop*, *peek*, *drop* niso definirane za prazen sklad, kar lahko privede do napake. V nadaljevanju bomo zgradili sklad, ki bo z bolj ekspresivnimi tipi zagotavljal, da bodo te funkcije lahko sprejele le sklade, nad katerimi se lahko izvedejo operacije.

Preden implementiramo dokazano različico sklada, velja našteti nekaj lastnosti, ki jih od podatkovne strukture sklad pričakujemo:

- *push* doda element na vrh (začetek) podanega sklada in vrne novi sklad,
- *peek* vrne element, ki je na vrhu nepraznega sklada,
- *drop* zavrže vrh nepraznega sklada in vrne preostanek (sklad, iz katerega je bil podani sklad zgrajen) in
- *pop* vrne par: element na vrhu nepraznega sklada in njegov preostanek.

5.5 Sklad z dokazom

Sedaj pa si poglejmo, kako lastnosti teh operacij lahko avtomatsko dokažemo. Najprej definiramo tip sklada.

```
data Stack : List a -> Type where
  StNil : Stack Nil {a}
  StCons : {xs : List a} -> (x : a) -> Stack xs -> Stack (x :: xs)
```

Sklad definiramo kot podatkovni tip, ki je indeksiran s seznamom. Prazen sklad je indeksiran s praznim seznamom Nil, neprazni sklad pa zgradimo s podatkovnim konstruktorjem $StCons$. Slednji (poleg implicitnega parametra $xs : List\ a$) sprejme element x tipa a in obstoječi sklad tipa $Stack\ xs$ in iz njiju zgradi nov sklad tipa $Stack\ (x :: xs)$. To pomeni, da je indeksiran s seznamom xs, ki mu je spredaj pripet x.

Parameter xs, ki je obdan z zavitimi oklepaji, je t. i. implicitni parameter, katerega vrednosti (v večini primerov) ni potrebno podati, ker jo prevajalnik določi iz konteksta. Vrednost pa potrebujemo, da ji lahko dodelimo ime in tip in s tem omogočimo njeno uporabo v drugih tipih.

Z novo definicijo tipa sklada napišemo funkcije, ki bodo v svojih tipih izražale lastnosti, ki jih od njih pričakujemo. Najprej si oglejmo operacijo *push*.

```
push : {xs : List a} -> (x : a) -> Stack xs -> Stack (x :: xs)
push x xs = StCons x xs
```

Funkcija *push* sprejme x tipa a in sklad tipa *Stack xs* ter vrne nov sklad tipa *Stack (x :: xs)*. Lastnosti funkcije *push* razberemo samo iz tipa. Njena implementacija je s tem skorajda povsem enolično določena, saj drugače sklada s takim tipom sploh ni možno skonstruirati.

```
drop : {x : a} -> {xs : List a} -> Stack (x :: xs) -> Stack xs
drop (StCons y ys) = ys
```

Podobno je tudi s funkcijo *drop*, ki v tipu zahteva, da vrnemo preostanek sklada. Zopet težko najdemo drugačno implementacijo, ki bi ustrezala temu tipu. Ker tip funkcije *drop* zahteva sklad, ki je indeksiran s seznamom $x :: xs$, funkcija ne more sprejeti praznega sklada (ta bi namreč imel tip *Stack Nil*).

Poglejmo si še implementacijo funkcije *peek*.

```
peek : {x : a} -> {xs : List a} -> Stack (x :: xs) -> (y : a ** y = x)
peek (StCons y ys) = (y ** refl)
```

Funkcija *peek* vrača element na vrhu sklada ter dokaz o enakosti tega elementa z dejanskim vrhom sklada. To je zapisano tudi v njenem tipu, ki pomeni, da sprejme sklad tipa *Stack (x :: xs)*, vrne pa odvisni par. Ta vsebuje vrednost y tipa a ter dokaz, da je y enak x. S tem zagotovimo, da ima y lahko le vrednost, ki je enaka vrhu x sklada $x :: xs$.

Implementacija sprejeti sklad najprej razstavi na (edini možni) podatkovni konstruktor *StCons* in tako dobimo vrh sklada y in preostanek ys. Rezultat funkcije je odvisni par, ki ima kot prvi element vrednost y, kot drugi pa dokaz, da je y res enak vrhu sklada.

Kot dokaz uporabimo kar funkcijo *refl* iz standardne knjižnice. Njen tip je $x = x$, kar predstavlja trivialno enakost, tj. enakost, ki jo zna unifikator (ključen del preverjanja skladnosti tipov) brez pomoči dokazati. V našem primeru je to enostavno, saj smo vzeli in vrnili natanko tisto vrednost, ki je na vrhu sklada.

Implementacija funkcije *pop* pa je združena funkcionalnost funkcij *peek* in *drop*.

```
pop : {x : a} -> {xs : List a} -> Stack (x :: xs) ->
        (p : (a, Stack xs) ** (fst p) = x)
pop (StCons y ys) = ((y, ys) ** refl)
```

Funkcija *pop* vrača bolj kompleksen tip. Vrača namreč odvisni par, ki kot prvi – poimenovani element vsebuje par oziroma dvojico p. Dvojica p je sestavljena iz elementa tipa a iz vrha sklada ter iz preostanka sklada tipa *Stack xs*. Kot drugi – odvisni element odvisnega para pa funkcija vrača dokaz, da je element, ki ga vračamo res enak tistem iz vrha sklada. Ker pa je prvi element odvisnega para dvojica p in ne kar element, od katerega je dokaz o enakosti odvisen, je potrebno uporabiti funkcijo fst, ki iz dvojice p izlušči prvi element.

5.6 Vrsta

Poleg sklada je ena izmed najbolj uporabljenih podatkovnih struktur *vrsta*. V knjigi [20] je Chris Okasaki pokazal, da je moč s funkcijskim pristopom zgraditi persistentno [1] podatkovno strukturo, ki ustreza lastnostim vrste z amortizirano časovno zahtevnostjo operacij $O(1)$. V tem razdelku se bomo posvetili manj učinkoviti (naivni) implementaciji, ki pa je zaradi enostavnosti primernejša za namen dokazovanja pravilnosti.

Kot smo pri dokazovanju sklada, bomo tudi tukaj najprej prikazali izvedbo podatkovne strukture brez dokazov. S tem bomo prikazali delovanje in primere, kjer s to različico zaidemo v težave.

Vrsto bomo predstavili skorajda identično seznamu.

```
data Queue : Type -> Type where
  QNil : Queue a
  QCons : a -> Queue a -> Queue a
```

[1] Operacije nad persistentnimi podatkovnimi strukturami ne spreminjajo podatkovne strukture ampak vračajo novo (spremenjeno) različico. Izvorna podatkovna struktura ostane nespremenjena – zaradi tega pravimo, da je persistentna ali ohranjujoča.

Za delo z vrsto bomo definirali še znane funkcije *enqueue*, *dequeue*, *next* in *drop*.

```
enqueue : (x : a) -> Queue a -> Queue a
enqueue x QNil = QCons x QNil
enqueue x (QCons y yq) = QCons y (enqueue x yq)
```

Funkcija *enqueue* sprejme nov element in obstoječo vrsto in vstavi element na konec vrste. Vstavljanje je narejeno rekurzivno in ima časovno zahtevnost $O(n)$.

Skladno s pričakovanji sta implementirani tudi funkciji *next* in *drop*.

```
next : Queue a -> a
next (QCons y yq) = y

drop : Queue a -> Queue a
drop (QCons y yq) = yq
```

Funkcija *next* sprejme vrsto in vrne element na začetku vrste, *drop* pa zavrže element na začetku vrste in vrne preostanek vrste. Njuni funkcionalnosti pa združimo v funkciji *dequeue*, ki vrne element na začetku vrste in preostanek vrste.

```
dequeue : Queue a -> (a, Queue a)
dequeue (QCons y yq) = (y, yq)
```

Na tej točki lahko bralec takoj opazi, da funkcije niso definirane za vse možne vrste. Definirane so le za neprazne vrste, vendar pa tipi funkcij dovoljujejo tako neprazne kot prazne vrste. Pojavlja se znan problem, ki pa ga znamo rešiti z odvisnimi tipi.

5.7 Vrsta z dokazom

Da bi dokazali pravilnost naše implementacije, moramo najprej razmisliti o lastnostih, za katere želimo pokazati, da držijo. Te lastnosti bomo zapisali v tipe funkcij, ki operirajo z vrsto.

Od prej naštetih funkcij zahtevamo, da:

- *enqueue* doda element na konec vrste in vrne novo vrsto,
- *next* vrne element z začetka neprazne vrste,
- *drop* zavrže element z začetka neprazne vrste in vrne preostanek,
- *dequeue* vrne par: element iz začetka neprazne vrste in njen preostanek.

Opazimo lahko, da so opisi teh lastnosti na las podobni opisom delovanja teh funkcij. Razlog za to je, da od teh funkcij pričakujemo točno tako delovanje. Skrbnemu programerju ni težko spisati pravilnih implementacij, katerih opis bo enak zahtevanim lastnostim. Vseeno se lahko zgodi(in velikokrat se), da kljub temu naredi napako. Na primer, Idris bi za pravilno implementacijo (skladno v tipih) razglasil tudi takšno funkcijo *enqueue*:

```
enqueue' : (x : a) -> Queue a -> Queue a
enqueue' x q = QCons x q
```

Seveda takšna implementacija ne bi bila pravilna, saj vrsta ne bi delovala po pričakovanjih programerja. Elementi bi iz vrste prihajali v napačnem vrstnem redu. Pravilnosti ne bomo prepustili naključju in bomo vrsto implementirali z odvisnimi tipi.

```
data Queue : List a -> Type where
  QSome : (xs : List a) -> Queue xs
```

Zgradili smo odvisni tip *Queue*, ki je indeksiran s seznamom. V tem primeru nam zadošča le en konstruktor *QSome*, ki kot parameter sprejme seznam tipa *List a*. To pomeni, da lahko vrsto zgradimo iz kakršnegakoli seznama. Definicija se na prvi pogled zdi ohlapna, vendar za potrebe izražanja lastnosti operacij čisto zadostuje. Lastnosti bomo namreč izrazili s tipi funkcij, ki operirajo nad vrednostmi tipa *Queue*.

Da zagotovimo, da bo dodajanje elementa v vrsto, element vstavilo na njen konec, lahko to sedaj izrazimo s tipom odvisne funkcije.

```
enqueue : {xs : List a} -> (x : a) -> Queue xs -> Queue (xs ++ [x])
enqueue x (QSome xs) = QSome (xs ++ [x])
```

Vstavljanje elementa x na konec vrste smo izrazili kot pripenjanje seznama z enim elementom ([x]) k seznamu, ki definira obstoječo vrsto:

```
(x : a) -> Queue xs ->  Queue (xs ++ [x])
```

Ker smo seznam (in s tem tudi vrsto) definirali parametrično (za katerikoli tip a) in ker smo zahtevali točno tako vrsto, ki ima na koncu pripet element x, je to tudi edina mogoča implementacija funkcije *enqueue*.

Na podoben način zapišemo tudi tip funkcije *drop*.

```
drop : {xs : List a} -> {x : a} -> Queue (x :: xs) -> Queue xs
drop (QSome (x :: xs)) = QSome xs
```

Iz vrste, ki je indeksirana s seznamom $x :: xs$, naredimo novo vrsto, ki je indeksirana s seznamom xs, kar pa natanko ustreza preostanku vrste.

Lastnost funkcije *next*, da vrača točno element na začetku vrste, lahko izraimo z odvinim parom.

```
next : {xs : List a} -> {x : a} -> Queue (x :: xs) -> (y : a ** y = x)
next (QSome (x :: xs)) = (x ** refl)
```

Izhodna vrednost funkcije *next* je odvisni par, ki je sestavljen iz elementa ter dokaza, da je ta element enak tistemu iz začetka vrste. Sestavimo jo kar iz elementa x iz začetka vrste in dokaza za trivialno enakost *refl*, ki pa smo ga srečali že pri dokazu pravilnosti sklada.

Lastnosti funkcij *drop* in *next* podobno kot pri nedokazani različici združimo v funkcijo *dequeue*, ki vrne element iz začetka vrste in novo vrsto s preostankom.

```
dequeue : {x : a} -> {xs : List a} -> Queue (x :: xs) ->
            (p : (a, Queue xs) ** (fst p) = x)
dequeue (QSome (x :: xs)) = ((x, QSome xs) ** refl)
```

Enako kot pri skladu, tudi tukaj tip *Queue (x :: xs)* ustreza samo nepraznim vrstam, tako da smo skupaj z izraženimi lastnostmi pokrili celotno delovanje vrste.

5.8 Urejanje z vstavljanjem

V tem razdelku bo prikazano urejanje seznama. Da se soočimo s problemom, si bomo najprej pogledali nepreverjeno implementacijo urejanja z vstavljanjem (angl. *insertion sort*). Zaradi enostavnosti in preglednosti bomo urejanje prikazali samo na seznamih, ki hranijo naravna števila.

Urejanje z vstavljanjem lahko razdelimo na dva dela:

- vstavljanje enega elementa v že urejen seznam,
- zlaganje elementov seznama v nov, urejen seznam.

5.8.1 Vstavljanje v urejen seznam

Prvi del realiziramo s funkcijo *insert*, drugega pa s funkcijo *insertionSort*.

```
insert : Nat -> List Nat -> List Nat
insert e [] = [e]
insert e (x :: xs) =
  if e <= x
    then e :: x :: xs
    else x :: (insert e xs)
```

Funkcija *insert* sprejme element e in ga vstavi v seznam. V trivialnem primeru, ko je obstoječi seznam prazen, vrne seznam [e] z enim elementom. V induktivnem primeru, ko pa seznam vsebuje vsaj en element x in preostanek xs, pa se izvede primerjava ter vstavljanje. Element e se primerja z glavo x. V primeru, da je e manjši ali enak od x, se ga pripne na začetek obstoječega seznama $x :: xs$. Če pa je e večji, pa je rekurzivno vstavljen v rep xs. Seznamu, ki ga dobimo kot rezultat rekurzivnega klica, pa pripnemo še staro glavo x

5.8.2 Urejanje

Urejen seznam lahko sedaj zgradimo iz elementov neurejenega seznama s pomočjo funkcije *insert*. Tekom postopka se dolžina oziroma velikost urejenega seznama povečuje. Tako je tudi definirana funkcija *insertionSort*:

```
insertionSort : List Nat -> List Nat
insertionSort [] = []
insertionSort (x :: xs) = insert x (insertionSort xs)
```

Ko je vhodni seznam prazen, funkcija vrne prazen seznam, ki je trivialno urejen. Podobno kot pri funkciji *insert* ima tudi funkcija *insertionSort* induktivni primer, ko vhodni seznam ni prazen. Takrat lahko glavo seznama x vstavimo v rekurzivno sortirani rep xs.

Predstavili smo poznano funkcijsko implementacijo urejanja z vstavljanjem. Ker je problem enostaven, ni težko napisati pravilne implementacije. Vendar pa prevajalnik o pravilnosti naše implementacij funkcije *insertionSort* ne more kaj dosti povedati. V tipu zahtevamo, da je vhodni argument seznam naravnih števil, kot izhodno vrednost pa tudi pričakujemo seznam naravnih števil. S stališča skladnosti tipov bi bila povsem pravilna tudi implementacija, ki bi vrnila kar vhodni seznam ali pa celo prazen seznam, na primer:

```
insertionSort' : List Nat -> List Nat
insertionSort' xs = xs

insertionSort'' : List Nat -> List Nat
insertionSort'' xs = []
```

Skratka, pravilna implementacija s stališča skladnosti tipov bi bila kakršnakoli implementacija, ki bi vračala seznam naravnih števil. Ponovno bomo s pomočjo odvisnih tipov poskusili pokazati, kako se takim napakam izogniti.

5.8.3 Pravilnost algoritma

Za začetek orišimo dokaz, da urejanje z vstavljanjem dejansko ureja. V knjigi [10] je opisan dokaz pravilnosti urejanja z vstavljanjem, ki gradi dokaz s pomočjo zančnih invariant, mi pa bomo pokazali pravilnost s pomočjo matematične indukcije.

Najprej pokažimo pravilnost vstavljanja elementa e v urejen seznam xs. Začnimo s trivialnim primerom, z vstavljanjem v prazen seznam. Če v prazen seznam, ki je seveda urejen, vstavimo element, dobimo seznam z enim elementom, ki je prav tako urejen.

$$x, urejen([\,]) \Rightarrow urejen(x :: [\,])$$

To je edina možna stvar, ki jo lahko naredimo. Tudi seznam z enim elementom je trivialno sortiran, torej naš algoritem deluje pravilno na trivialnem primeru.

Če pa je že urejen seznam neprazen $(x :: xs)$, potem ločimo dva različna primera. Če je element e manjši ali enak glavi x urejenega seznama, ga lahko pripnemo na začetek, saj s tem spoštujemo urejenost.

$$urejen(x :: xs) \Leftrightarrow \forall y \in xs.\, x \leq y$$
$$e \leq x \wedge \forall y \in xs.\, x \leq y \Rightarrow \forall y \in xs.\, e \leq y$$
$$e \leq x \wedge \forall y \in xs.\, e \leq y \Rightarrow \forall z \in x :: xs.\, e \leq z$$
$$\forall z \in x :: xs.\, e \leq z \Leftrightarrow urejen(e :: x :: xs)$$

Če pa je element večji od glave urejenega seznama, pa lahko uporabimo induktivno predpostavko: element rekurzivno vstavimo v rep in dobimo urejen seznam. Ker je bila glava začetnega seznama manjša ali enaka od vseh elementov v repu in tudi manjša ali enaka elementu, ki smo ga vstavili v rep, je s tem tudi manjša ali enaka vsem elementom v novem repu in jo lahko pripnemo na začetek. Tako za katerikoli vhodni element in urejen seznam lahko naredimo nov urejen seznam.

$$urejen(x :: xs) \Leftrightarrow \forall y \in xs.\, x \leq y$$

po induktivni predpostavki:

$$urejen(insert(e, xs))$$
$$x \leq e \wedge \forall y \in xs.\, x \leq y \Rightarrow \forall z \in insert(e, xs).\, x \leq z$$
$$\forall z \in insert(e, xs).\, x \leq z \Leftrightarrow urejen(x :: insert(e, xs))$$

S tem je dokaz vstavljanja končan.

Poglejmo si še del, kjer sprejmemo neurejen seznam, uporabimo vstavljanje in vrnemo urejen seznam. Spet bomo uporabili matematično indukcijo. Kot trivialni primer sprejmemo prazen seznam, ki je trivialno urejen in je to tudi naš izhod algoritma. Ob nepraznem vhodnem seznamu pa ga razstavimo na glavo in rep. Uporabimo induktivno predpostavko in z njo rekurzivno uredimo rep. Sedaj uporabimo vstavljanje, ki smo ga dokazali v prejšnjem odstavku, da glavo vstavimo v urejeni rep. Tako smo dokazali, da naš algoritem dejansko ureja sezname.

Da bi dokazali pravilno delovanje urejanja z vstavljanjem, je potrebno dokazati, da algoritem zagotavlja oziroma ohranja ti dve lastnosti:

- **urejenost** – vrstni red elementov v novem seznamu je pravilen in
- **permutiranost** – nov seznam je permutacija starega seznama.

Na tej točki velja omeniti, da v večini literature najdemo le dokaz o pravilnem urejanju urejanja z vstavljanjem, ne pa tudi dokaza o pravilni permutiranosti. V nadaljevanju bomo z odvisnimi tipi izrazili ne le urejenost ampak tudi permutiranost, oboje pa bo avtomatsko preverjal prevajalnik.

5.9 Urejenost naravnih števil

Za zagotavljanje urejenosti najprej potrebujemo definicijo urejenosti, zato si naprej poglejmo, kako je urejenost definirana nad naravnimi števili.

5.9.1 Delna urejenost

V splošnem je urejenost, natančneje *delna urejenost*, relacija $R \subset A \times A$ nad množico A. Delna urejenost je relacija $a \leq b$ med dvema elementoma množice $a, b \in A$, če zanjo veljajo naslednje lastnosti:

- **refleksivnost**: $a \leq a$,
- **tranzitivnost**: če $a \leq b$ in $b \leq c$, potem $a \leq c$,
- **antisimetričnost**: če $a \leq b$ in $b \leq a$, potem $a = b$.

Če za množico lahko najdemo takšno relacijo R, pravimo, da je množica A delno urejena glede na R.

5.9.2 Linearna urejenost

Za urejanje seznamov pa potrebujemo še eno lastnost:

- **stroga sovisnost**: velja $a \leq b$ ali $b \leq a$.

Relacijo, za katero veljajo vse štiri lastnosti, poimenujemo *popolna urejenost* ali *linearna urejenost*.

5.9.3 Tip LTE

V standardni knjižnici programskega jezika Idris najdemo tip *LTE*, ki predstavlja relacijo $\leq$ nad naravnimi števili.

```
||| Proofs that `n` is less than or equal to `m`
||| @ n the smaller number
||| @ m the larger number
data LTE  : (n, m : Nat) -> Type where
  ||| Zero is the smallest Nat
  lteZero : LTE Z    right
  ||| If n <= m, then n + 1 <= m + 1
  lteSucc : LTE left right -> LTE (S left) (S right)
```

Vrednost tipa *LTE n m* predstavlja dokaz, da velja $n \leq m$. Podatkovna konstruktorja *lteZero* in *lteSucc* predstavljata edini način, kako lahko ustvarimo takšen dokaz. Dokaz je zgrajen s pomočjo indukcije. S *lteZero* lahko zgradimo trivialen dokaz, da je vrednost *0* oziroma *Z* manjša ali enaka kateremukoli naravnemu številu. Konstruktor *lteSucc* pa predstavlja induktivni korak v dokazu. Pravi, da če velja $left \leq right$, potem velja to tudi za naslednika teh dveh števil: *(S left)* $\leq$ *(S right)*.

Da je *LTE* res relacija popolne urejenosti, je potrebno zanj dokazati prej omenjene lastnosti. Spet velja omeniti Curry-Howardovo enakovrednost, saj bomo lastnosti zapisali in jih pretvorili v tipe funkcij, ki jih bomo z implementacijo dokazali.

5.9.4 Refleksivnost

Izjava:

$$\forall x \in \mathbb{N}.\, x \leq x$$

Tip in implementacija funkcije:

```
lteRefl : (x : Nat) -> LTE x x
lteRefl Z = lteZero
lteRefl (S k) = lteSucc (lteRefl k)
```

Refleksivnost lahko v trivialnem primeru $Z \leq Z$ dokažemo kar s podatkovnim konstruktorjem *lteZero*.

V primeru, ko pa dokazujemo refleksivnost za naslednika nekega naravnega števila k, pa lahko zopet uporabimo indukcijo. Rekurzivno lahko zgradimo dokaz, da velja $k \leq k$, s pomočjo konstruktorja *lteSucc* pa to prenesemo na njegovega naslednika *S k*.

Primer izvedbe funkcije *lteRefl*:

```
idris> lteRefl (S (S Z))
lteSucc (lteSucc lteZero) : LTE 2 2
```

5.9.5 Tranzitivnost

Izjava:

$$\forall x, y, z \in \mathbb{N}.\, x \leq y \wedge y \leq z \Rightarrow x \leq z$$

Tip in implementacija funkcije:

```
lteTrans : LTE x y -> LTE y z -> LTE x z
lteTrans lteZero ylez = lteZero
lteTrans (lteSucc xprf) (lteSucc yprf) = lteSucc (lteTrans xprf yprf)
```

Pri dokazovanju tranzitivnosti sprejmemo dokaza, da velja $x \leq y$ in da velja $y \leq z$ in iz njiju napravimo nov dokaz $x \leq z$. V prvem primeru, ko je dokaz za $x \leq y$ zgrajen iz *lteZero* (x je torej Z), lahko dokaz $x \leq z$ zgradimo kar s konstruktorjem *lteZero*, saj v vsakem primeru velja $x \leq z$ (ker je x enak Z).

V primeru, ko je x naslednik nekega naravnega števila, je edini način konstruiranja dokaza $x \leq y$ s konstruktorjem *lteSucc*. Kadar velja $x \leq y$ in je x naslednik, potem velja, da je tudi y naslednik nekega naravnega števila. Zato je tudi tokrat dokaz $y \leq z$ lahko zgrajen s konstruktorjem *lteSucc*. Ko razstavimo konstruktor *lteSucc*, dobimo tudi vrednosti dokaza, da je predhodnik x manjši od predhodnika y in da je predhodnik y manjši od predhodnika z. S pomočjo rekurzije (uporabimo induktivno predpostavko, da velja *lteTrans*) lahko pridelamo dokaz, da je predhodnik x manjši od predhodnika z. Nad rekurzivno ustvarjenim dokazom uporabimo še konstruktor *lteSucc*, ki vrne dokaz, ki ga potrebujemo.

5.9.6 Antisimetričnost

Antisimetričnost iz dokazov $x \leq y$ in $y \leq x$ gradi dokaz o enakosti x in y. V prvem primeru, ko je dokaz $x \leq y$ enak *lteZero*, lahko iz tega sklepamo, da je vrednost x enaka Z. Zato je edina možna vrednost dokaza $y \leq x$ tudi *lteZero*, saj je Z lahko manjši ali enak lahko le od Z. Torej je tudi vrednost y enaka Z in med x in y velja trivialna enakost *refl*.

Izjava:

$$\forall x, y \in \mathbb{N}.\, x \leq y \wedge y \leq x \Rightarrow x = y$$

Tip in implementacija funkcije:

```
lteAntiSym : LTE x y -> LTE y x -> x = y
lteAntiSym lteZero lteZero = refl
lteAntiSym (lteSucc xprf) (lteSucc yprf) =
    rewrite (lteAntiSym xprf yprf) in refl
```

V drugem primeru, ko je dokaz o enakosti zgrajen iz *lteSucc*, je x lahko le naslednik nekega naravnega števila i tj. $x = S\ i$. Ker *lteSucc* zgradi dokaz, ki velja za naslednike, je tudi y lahko le naslednik nekega naravnega števila j tj. $y = S\ j$. Iz tega sledi, da je edini način, da je bil zgrajen dokaz $y \leq x$, s konstruktorjem *lteSucc*. Ko razstavimo konstruktorja *lteSucc*, dobimo dokaza *xprf* in *yprf*, ki sta dokaza o $i \leq j$ in $j \leq i$. Iz teh dveh dokazov znamo rekurzivno skonstruirati dokaz, da je i enak j. Ker dokazujemo $S\ i = S\ j$, lahko dokaz o enakosti i in j uporabimo tako, da spremenimo tip dokaza, ki ga konstruiramo iz $S\ i = S\ j$ v $S\ i = S\ i$. To naredimo z vgrajenim orodjem *rewrite*. Dokaz $S\ i = S\ i$ pa lahko zgradimo kar s trivialno enakostjo *refl*. S tem je dokaz antisimetričnosti končan.

5.9.7 Stroga sovisnost

Dokaz stroge sovisnosti zgradimo tako, da za katerikoli dve naravni števili x in y lahko napravimo dokaz o $x \leq y$ ali dokaz o $y \leq x$. Da izrazimo to razmerje med možnima dokazoma, uporabimo tip *Either*, ki smo ga opisali v poglavju 4. Izhodni tip *lteTotal* je torej *Either (LTE x y) (LTE y x)*.

V trivialnih primerih, ko je eno izmed števil (ali obe) enako Z, je dokaz kar *lteZero*. Da zadostimo tipu *Either*, ovijemo ta dokaz še v *Left* ali *Right*, da povemo, za kateri tip smo podali dokaz. V primeru, ko je x enak Z, podamo dokaz za levi tip ($LTE\ x\ y$), v drugem, ko je y enak Z pa dokaz za desni tip ($LTE\ y\ x$). To storimo s konstruktorjema *Left* in *Right*. S tem smo

Izjava:

$$\forall x, y \in \mathbb{N}.\, x \leq y \vee y \leq x$$

Tip in implementacija funkcije:

```
lteTotal : (x, y : Nat) -> Either (LTE x y) (LTE y x)
lteTotal Z y = Left lteZero
lteTotal (S k) Z = Right lteZero
lteTotal (S k) (S j) with (lteTotal k j)
  | Left prf = Left (lteSucc prf)
  | Right prf = Right (lteSucc prf)
```

pokrili primere, ko sta eno ali obe števili enaki Z.

Ostane nam še primer, ko sta x in y naslednika nekih naravnih števil k in j. Ker je tudi *lteTotal* zastavljen induktivno, uporabimo rekurzijo, da pridobimo dokaz za manjši problem, tj. za k in j. Na tej točki je možno s pomočjo *lteSucc* zgraditi dokaz še za x in y. Ker je tip dokaza za k in j enak *Either (LTE k j) (LTE j k)*, ga moramo obravnavati za vsak primer gradnje vrednosti Either posebej.

5.9.8 Maksimum z dokazom

Lastnosti, ki smo jih dokazali, lahko sedaj uporabljamo za primerjanje naravnih števil. Izračunamo lahko npr. maksimum dveh naravnih števil:

```
max : (n: Nat) -> (m : Nat) ->
        (x : Nat ** Either (x = m, LTE n m) (x = n, LTE m n))
max n m with (lteTotal n m)
  | (Left nlem) = (m ** Left (refl, nlem))
  | (Right mlen) = (n ** Right (refl, mlen))
```

Maksimum je definiran kot funkcija, ki sprejme dve naravni števili n in m, ter vrne naravno število x. Za x velja, da je enak m in imamo dokaz, da je n manjši ali enak m, ali pa je x enak n in imamo dokaz, da je m manjši

ali enak n. Funkcija *max* prikazuje uporabo stroge sovisnosti (*lteTotal*), ki jo uporabi, da se odloči, katero izmed števil je večje.

5.10 Urejenost seznama

V prejšnjem razdelku smo si ogledali, kako lahko primerjamo naravna števila s pomočjo lastnosti, ki veljajo za urejenost. V tem razdelku bomo te lastnosti uporabili za konstrukcijo urejenega seznama. Podatkovni tip seznama bomo razširili, da bo vrednost seznama možno skonstruirati le tako, da bo seznam urejen.

5.10.1 Naivni urejeni seznam

Najprej si oglejmo naiven poskus takšne razširitve:

```
mutual
  data SortedList : Type where
    SNil : SortedList
    SCons : (x : Nat) -> (xs : SortedList) ->
      (canPrepend x xs) = True -> SortedList

  canPrepend : (x : Nat) -> (xs : SortedList) -> Bool
  canPrepend x SNil = True
  canPrepend x (SCons y ys prf) = x <= y
```

Ideja za takšno implementacijo je čisto smiselna. Prazen seznam je že urejen. Ko pa dodajamo element v že urejen neprazen seznam, pa potrebujemo dokaz, da ga lahko pripnemo na začetek. Ta dokaz je v obliki funkcije *canPrepend*, ki vrne logično vrednost *True*, kadar to lahko storimo. Na žalost se izkaže, da s takšno definicijo lahko prevajalnik dokaže samo urejenost seznamov, ki imajo vrednosti elementov znanih že ob prevajanju. To so samo vrednosti, ki jih lahko izpeljemo iz konstant, na primer:

```
constantList : SortedList
constantList = SCons 1 (SCons 2 (SCons 3 SNil refl) refl) refl
```

Želimo bolj splošno definicijo seznama, v katero lahko vstavljamo tudi vrednosti, ko jih program pridobi iz zunanjega sveta. Potrebujemo definicijo, s katero bomo lahko obravnavali vse možne primere vrednosti, za vsakega pa tudi dokazali pravilnost delovanja.

5.10.2 Urejeni seznam z mejami

Da bi na malo drugačen način dokazovali vstavljanje, potrebujemo definicijo podatkovnega tipa, ki bo to tudi podpirala. V definicijo urejenega seznama vpeljemo koncept omejenosti. Tipu urejenega seznama dodamo **spodnjo** in **zgornjo** mejo. Za urejeni seznam obstaja spodnja meja l, da velja, da je l manjši ali enak vsem elementom v seznamu. Zaradi urejenosti seznama to velja, če je manjši ali enak prvemu elementu v urejenem seznamu.

Podobno definicijo podamo tudi za zgornjo mejo. Za urejeni seznam obstaja zgornja meja u, da so vsi elementi v urejenem seznamu manjši ali enaki u. Tudi za zgornjo mejo velja, da dovolj, da je od nje manjši ali enak le zadnji element urejenega seznama. Seveda pa mora veljati, da je spodnja meja vedno manjša ali enaka zgornji meji.

V primeru praznega seznama lahko meji izberemo poljubno, le da je spodnja meja manjša ali enaka zgornji. Za podobno definicijo urejenega seznama najdemo implementacijo [17] v programskem jeziku Agda.

Poglejmo si novo definicijo urejenega seznama, ki vsebuje meje.

```
data SortedList : Nat -> Nat -> Type where
  SNil : LTE l u -> SortedList l u
  SCons : (x : Nat) -> (xs : SortedList x u) ->
            LTE l x -> SortedList l u
```

Podatkovni tip *SortedList* je sedaj indeksiran z dvema naravnima številoma l in u, ki predstavljata spodnjo in zgornjo mejo seznama. Prazen seznam, ki

je trivialno urejen, ustvarimo s konstruktorjem *SNil*. Podamo dokaz, da je izbrana spodnja meja l manjša ali enaka izbrani zgornji meji u.

Dodajanje novega elementa x na začetek že urejenega seznama xs pa storimo s konstruktorjem *SCons*. *SCons* v svojem tipu zahteva, da lahko sprejme le tak element x in tak urejen seznam xs, da je xs od spodaj omejen z x. Ta pogoj nam zagotavlja, da ohranjamo urejenost, saj mora biti x kot spodnja meja po definiciji manjši ali enak vsem elementom v xs. Da sestavimo nov urejen seznam potrebujemo še spodnjo mejo, ki mora biti manjša ali enaka x. Temu pa služi dokaz tipa *LTE l x*.

Ta definicija nam zadošča, da lahko napišemo ekvivalent funkciji *insert* (Slika 5.1). Nova implementacija mora poskrbeti, da se ustvarijo dokazi, ki so potrebni, da lahko element vstavimo v seznam.

```
1 insert : {l, u : Nat} -> (x : Nat) -> (xs : SortedList l u) ->
2            (llex : LTE l x) -> (xleu : LTE x u) -> SortedList l u
3 insert x (SNil lleu)       llex xleu = SCons x (SNil xleu) llex
4 insert x (SCons y ys lley) llex xleu with (lteTotal x y)
5   | Left  xley = SCons x (SCons y ys xley)       llex
6   | Right xgty = SCons y (insert x ys xgty xleu) lley
```

Slika 5.1: Funkcija *insert* za definicijo urejenega seznama z mejami.

Funkcija *insert* sprejme element x in obstoječ urejen seznam xs (navzdol omejen z l in navzgor z u) ter dokaza *llex* in *xleu*, da je l manjši ali enak x in da je x manjši ali enak u. Drugače povedano, element x mora biti med spodnjo in zgornjo mejo obstoječega seznama. Iz teh elementov se skonstruira nov urejen seznam, ki je prav tako omejen z l in u. Da dokažemo vstavljanje elementa v seznam, moramo obravnavati več različnih situacij.

Najprej si oglejmo trivialen primer: vstavljanje elementa v prazen seznam (vrstica 3). V tem primeru lahko kar direktno uporabimo konstruktor *SCons*, saj je to edini način, kako iz praznega urejenega seznama naredimo urejen seznam z enim elementom. Na tej točki velja omeniti, da lahko obstoječim urejenim seznamom poljubno spreminjamo meje, dokler lahko dokažemo, da

spoštujejo pogoje, ki jih zahtevamo v definiciji. Da lahko x pritaknemo na začetek praznega seznama, mu moramo spremeniti meje. Element x mora postati nova spodnja meja, zgornjo mejo u pa ohranimo. To lahko storimo z dokazom *xleu*, da velja $x \leq u$, ki ga sprejmemo kot parameter in podamo konstruktorju *SNil*. Nato lahko uporabimo konstruktor *SCons*, ki mu podamo x, prazen seznam z novimi mejami in dokaz *llex*, da je spodnja meja l manjša ali enaka x.

Preostane nam še situacija, kjer bomo izvedli primerjanje in vstavljanje. V vrstici 4 sprejmemo vzorec, da je seznam xs zgrajen s konstruktorjem *SCons*, torej neprazen seznam. V telesu netrivialnega primera moramo izvesti primerjavo med x in prvim elementom seznama xs, torej med x in y. Za izvedbo same primerjave si bomo pomagali z lastnostmi, ki smo jih dokazali za urejenost naravnih števil. Podobno kot smo pri funkciji *max*, bomo uporabili strogo sovisnost, ki je implementirana s funkcijo *lteTotal*. Primerjavo x in y izvedemo z "with" pogledom, kar nam omogoča, da obravnavamo vsakega izmed obeh možnih izidov primerjave in da enostavno izluščimo dokaze, ki jih potrebujemo. V primeru, ko je x manjši ali enak y, dobimo iz primerjave "levo" vrednost (vrstica 5), ki nosi dokaz *xley* tipa *LTE x y*. S tem dokazom lahko kot v primeru praznega seznama spremenimo spodnjo mejo, da omogočimo pripenjanje x na začetek obstoječega seznama xs oziroma *SCons y ys xley*.

Kadar pa je rezultat primerjave "desna" vrednost (vrstica 6), dobimo dokaz *xgty* tipa *LTE y x* oziroma, da je y manjši ali enak x. Tokrat pa x ne moremo direktno pripeti na začetek, lahko pa ga vstavimo v preostanek obstoječega seznama, v ys, kar lahko storimo rekurzivno. Pokličemo funkcijo *insert*, ki ji podamo x in preostanek ys ter dokaza *xgty* in *xleu*. Z *xgty* novemu seznamu nastavimo spodnjo mejo na y, da ga bomo lahko potem pripeli na začetek. Dokaz *xleu* pa potrebujemo, da zadostimo pogoju, da je x manjši ali enak zgornji meji. Rekurzivno zgrajeni seznam ima tip *SortedList y u*, tako da mu lahko na začetek pripnemo y in zgradimo končni urejen seznam. Manjka nam le še dokaz, da je l manjši ali enak y, ki pa ga kot *lley* izluščimo

iz prejšnjega seznama xs.

S tem smo dokazali, da naš algoritem za vstavljanje spoštuje urejenost. Naletimo pa na težavo, ko želimo urediti nek neurejen seznam. Težavo namreč povzroča izbira mej. Za konkreten seznam jih znamo poiskati (minimum in maksimum seznama), v splošnem pa ne. Seveda bi želeli, da naš algoritem za urejanje deluje na seznamih, ki lahko vsebujejo poljubna naravna števila. Za spodnjo mejo bi lahko izbrali Z, zgornje meje pa v splošnem ne moremo določiti, saj lahko seznam vsebuje poljubno naravno število. Za mejo bi morali postaviti največje naravno število, česar zaradi neskončnosti naravnih števil ne moremo.

5.10.3 Urejeni seznam z omejenimi mrežami

Da bi rešili težavo z mejami seznama, bomo potrebovali malo pomoči s strani matematičnih lastnosti naravnih števil. Naslonili se bomo na dejstvo, da je urejenost naravnih števil *mreža* (angl. *lattice*). Formalizacijo in dokaze mrež najdemo v knjigi [11].

Idejo, kako postaviti meje, najdemo pri *omejenih mrežah* (angl. *bounded lattice*). Iz vsake mreže znamo pridelati omejeno mrežo tako, da ji umetno dodamo elementa $\bot$ in $\top$, da velja

$$\forall x \in \mathbb{N} : \bot \leq x \wedge x \leq \top.$$

Z idejo iz omejenih mrež lahko definiramo omejena naravna števila. Definiramo podatkovni tip *BNat*.

```
data BNat: Type where
  Top : BNat
  Bottom : BNat
  BLift : Nat -> BNat
```

Tipu *Nat* tako dodamo dve vrednosti:

- *Bottom*, ki ustreza $\bot$ in

- *Top*, ki ustreza $\top$.

Poleg teh dveh vrednosti imamo še konstruktor *BLift*, s katerim lahko naravno število oziroma vrednosti tipa *Nat dvignemo* v kontekst omejenih naravnih števil *BNat*.

Za tip *BNat* potrebujemo definirati tudi urejenost, ki mora spoštovati prej naštete lastnosti. Definiramo tip dokaza *BLTE* x y, ki predstavlja dokaz, da je x manjši ali enak y, le da sta v tem primeru x in y omejeni naravni števili.

```
using (x, y : Nat)
  using (t : BNat)
    data BLTE: BNat -> BNat -> Type where
      BottomLower : BLTE Bottom t
      TopGreater : BLTE t Top
      BLifted: LTE x y -> BLTE (BLift x) (BLift y)
```

Konstruktorja *BottomLower* in *TopGreater* sta elementarna dokaza, ki nam zagotavljata, da je *Bottom* najmanjši pripadnik, *Top* pa največji pripadnik omejenih naravnih števil. Za potrebe dejanskega urejanja pa imamo še konstruktor *BLifted*, ki podobno kot *BLift* dvigne dokaz o urejenosti dveh naravnih števil v dokaz o urejenosti teh dveh števil, dvignjenih v kontekst omejenih naravnih števil.

Omejena naravna števila nam sedaj omogočajo, da lahko z njimi omejimo urejen seznam.

```
using (l, u : BNat)
  data SortedList : BNat -> BNat -> Type where
    SNil : BLTE l u -> SortedList l u
    SCons : (x : Nat) -> (xs : SortedList (BLift x) u) ->
              BLTE l (BLift x) -> SortedList l u
```

Definicija je skorajda identična prejšnji, le da za meje uporabljamo omejena naravna števila in temu primerno po potrebi dvigamo naravna števila v omejena naravna števila. S to definicijo lahko napišemo tip urejenega

seznama, ki lahko vsebuje poljubne vrednosti naravnih števil: *SortedList Bottom Top*.

Podajmo še popravljeno definicijo funkcije *insert*.

```
insert : {l, u : BNat} -> (x : Nat) -> SortedList l u ->
          BLTE l (BLift x) -> BLTE (BLift x) u -> SortedList l u
insert x (SNil lleu)       llex xleu = SCons x (SNil xleu) llex
insert x (SCons y ys lley) llex xleu with (lteTotal x y)
  | Left  xley = SCons x (SCons y ys (BLifted xley))       llex
  | Right xgty = SCons y (insert x ys (BLifted xgty) xleu) lley
```

Implementacija je na las podobna prejšnji, le da na primernih mestih dvignemo naravna števila v omejena naravna števila, dokaze o urejenosti naravnih števil pa v dokaze o urejenosti omejenih naravnih števil.

Pripravili smo vse potrebno, da lahko spišemo še funkcijo *insertionSort*.

```
insertionSort : List Nat -> SortedList Bottom Top
insertionSort [] = SNil BottomLower
insertionSort (x :: xs) =
   insert x (insertionSort xs) BottomLower TopGreater
```

Popravljena funkcija *insertionSort* sprejme seznam naravnih števil in vrne urejen seznam omejen z *Bottom* in *Top*. V trivialnem primeru, ko je vhodni seznam prazen, vrnemo prazen seznam *SNil* (dokaz *BottomLower* dokazje, da je *Bottom* kot spodnja meja manjši ali enak *Top*). V primeru neprazenega seznama pa *insertionSort* uporabimo rekurzivno na preostanku seznama, v urejen preostanek pa vstavimo glavo x. Dokaza *BottomLower* in *TopGreater* dokazujeta, da je x med spodnjo in zgornjo mejo, kar za delovanje potrebuje funkcija *insert*. S tem je dokaz, da algoritem urejanja z vstavljanjem ohranja urejenost, končan.

5.11 Permutiranost seznamov

V prejšnjem razdelku smo predstavili avtomatski dokaz, da urejanje z vstavljanjem zagotavlja urejenost. Vendar ima naša implementacija (vse podane

implementacije) eno bistveno pomanjkljivost: dokazali smo samo urejenost, ne pa tudi ohranjanja dolžine seznama in ohranjanja vsebovanosti elementov. S stališča skladnosti tipov je naslednja implementacija popolnoma pravilna:

```
insertionSort' : List Nat -> SortedList Bottom Top
insertionSort' xs = SNil BottomLower
```

Ne glede na to, kakšen je vhodni seznam, lahko vedno vrnemo prazen seznam. Ta je namreč trivialno urejen in zadošča vsem pogojem/tipom, ki smo jih zahtevali. Da bi rešili ta problem, moramo na nek način zahtevati tudi, da so v urejenem seznamu vsi elementi, ki so bili v izvornem seznamu. Oglejmo si, kako formalizirati našo zahtevo.

Imejmo seznam xs. Če elemente xs premešamo, dobimo nov seznam ys. Seznam ys vsebuje iste elemente kot xs, le njihov vrstni red je drugačen. Pravimo, da je ys *permutacija* seznama xs.

5.11.1 Tip Perm

Za potrebe dokazovanja bomo definirali tip *Perm xs ys*. Podatkovni tip *Perm* bo indeksiran z dvema seznamoma xs in ys, predstavljal pa bo relacijo med njima. *Perm xs ys* bo torej tip dokaza, da je seznam ys permutacija seznama xs.

```
data Perm : (List a) -> (List a) -> Type where
  PrNil : Perm {a} [] []
  PrPrep : (x : a) -> (xs : List a) -> (ys : List a) ->
             Perm xs ys -> Perm (x :: xs) (x :: ys)
  PrSwap : (x : a) -> (y : a) -> (xs : List a) ->
             Perm (x :: (y :: xs)) (y :: (x :: xs))
  PrTrans : {xs, ys, zs: List a} -> Perm xs ys ->
              Perm ys zs -> Perm xs zs
```

Slika 5.2: Definicija tipa *Perm*

Na sliki 5.2 vidimo, da ima tip *Perm xs ys* štiri konstruktorje. Prvi (*PrNil*) je trivialni konstruktor, ki gradi dokaz, da je prazen seznam permutacija praznega seznama.

Drugi konstruktor (*PrPrep*) pravi, da, če imamo seznama xs in ys in je ys permutacija xs, potem lahko obema seznamoma na začetek pripnemo element x, nov seznam pa bo $x :: ys$ prav tako permutacija seznama $x :: xs$.

Tretji konstruktor (*PrSwap*) pravi, da lahko elementa x in y na začetku seznama $x :: y :: xs$ zamenjamo, nov seznam $y :: x :: xs$ pa je permutacija prejšnjega.

Zadnji konstruktor (*PrTrans*) pa govori o tranzitivnosti permutacij: če je seznam ys permutacija seznama xs in je zs permutacija seznama xs, je tudi zs permutacija seznama xs.

```
PrRefl : (xs : List a) -> Perm xs xs
PrRefl [] = PrNil
PrRefl (x :: xs) = PrPrep x xs xs (PrRefl xs)

PrSame : {xs : List a} -> {ys : List a} -> xs = ys -> Perm xs ys
PrSame {xs=xs} refl = PrRefl xs

PrSym : {xs, ys: List a} -> Perm xs ys -> Perm ys xs
PrSym PrNil = PrNil
PrSym (PrPrep x xs ys perm) = PrPrep x ys xs (PrSym perm)
PrSym (PrSwap x y xs) = PrSwap y x xs
PrSym (PrTrans p1 p2) = PrTrans (PrSym p2) (PrSym p1)
```

Slika 5.3: Lastnosti permutacij

Ta definicija nam zadošča, da izrazimo in dokažemo tudi določene lastnosti permutacij, ki nam bodo koristile pri nadaljnjem dokazovanju. Na sliki 5.3 vidimo definicije in dokaze/implementacije lastnosti: refleksivnosti (*PrRefl*) in simetričnosti (*PrSym*). Podana pa je tudi implementacija pomožne funkcije *PrSame*, s katero izražamo lastnost, da sta dva enaka seznama tudi

permutaciji eden drugega. Tranzitivnost smo izrazili kar kot konstruktor, kar pomeni, da smo to lastnost postulirali oziroma vzeli za resnično.

V naslednjem razdelku bomo permutacije združili z urejanjem.

5.11.2 Urejanje z vstavljanjem in permutiranost

V tem razdelku si bomo ogledali, kako dokazati, da urejanje z vstavljanjem gradi seznam, ki je permutacija neurejene različice.

Da bi izrazili lastnost, da mora urejen seznam biti permutacija neurejenega, bomo popravili tipa funkcij *insert* in *insertionSort*, da bosta namesto samo urejenega seznama v odvisnem paru vračali tudi dokaza o permutiranosti.

Tip funkcije *insert* bomo iz

```
(x : Nat) -> (xs : List Nat) -> List Nat
```

popravili v

```
(x : Nat) -> (xs : List Nat) -> (zs : List Nat ** Perm (x :: xs) zs).
```

Zahtevamo torej, da *insert* vrača seznam zs, ki je permutacija originalnega seznama, kateremu na začetek pripnemo element x. To izraža tudi lastnost, da je x element novega (urejenega) seznama zs.

Na podoben način bomo s tipom funkcije *insertionSort* izrazili našo prvenstveno zahtevo. Tip bomo iz

```
insertionSort : (xs : List Nat) -> List Nat
```

spremenili v

```
insertionSort : (xs : List Nat) -> (zs : List Nat ** Perm xs zs).
```

Ker urejanje z vstavljanjem za delovanje uporablja funkcijo *insert*, bomo najprej pogledali, kako dokažemo njeno delovanje.

Funkcija *insert* s tipom, ki vsebuje permutacije, mora v odvisnem paru vračati zahtevani dokaz. Njeno implementacijo vidimo na sliki 5.4. V primeru, ko element vstavljamo v prazen seznam, je edina vrednost, ki zadošča skladnosti tipov, seznam z enim elementom *[x]* ali *x :: []*. Ker ima seznam enako obliko, kot če bi vhodnemu seznamu na začetek pripeli element x,

```
insert : (x : Nat) -> (xs : List Nat) ->
         (zs : List Nat ** Perm (x :: xs) zs)
insert x [] = ([x] ** PrSame refl)
insert x (y :: ys) =
  if x <= y
     then (x :: y :: ys ** PrSame refl)
     else let (nys ** nperm) = insert x ys in
          let perm1 = PrPrep y (x :: ys) nys nperm in
          let perm2 = PrSwap x y ys in
          let perm3 = PrTrans perm2 perm1 in
            (y :: nys ** perm3)
```

Slika 5.4: Funkcija *insert*, z dokazom o permutaciji urejenega seznama.

lahko dokaz o permutaciji napravimo kar s pomožno funkcijo *PrSame*, ki ji podamo dokaz o trivialni enakosti *refl*. Enak dokaz podamo tudi v primeru, ko element pripenjamo na začetek urejenega seznama.

Malo bolj kompleksen je primer, ko element vstavljamo v preostanek urejenega seznama. Z rekurzivnim vstavljanjem pridobimo novi preostanek *nys*, ki vključuje element x ter dokaz o permutiranosti *nperm*. S pomočjo funkcij za delo s permutacijami lahko ta dokaz predelamo v ustrezen dokaz, ki zadošča tipu funkcije *insert*.

```
insertionSort : (xs : List Nat) -> (zs : List Nat ** Perm xs zs)
insertionSort [] = ([] ** PrNil)
insertionSort (y :: ys) =
  let (nys ** perm) = insertionSort ys in
  let (sorted ** srtperm) = insert y nys in
  let perm1 = PrPrep y ys nys perm in
  let perm2 = PrTrans perm1 srtperm in
      (sorted ** perm2)
```

Slika 5.5: Urejanje z vstavljanjem, ki ohranja permutiranost.

Lastnost, ki jo za urejanje želimo dokazati, pa vidimo v tipu funkcije *insertionSort* (implementacija na sliki 5.5). Ker naša implementacija vstavlja element za elementom in tako gradi urejen seznam, ni težko pokazati, da je urejen seznam, ki ga vrnemo, permutacija vhodnega seznama. Potrebno je le spretno rokovanje z dokazi o permutiranosti, ki jih vrača funkcija *insert*. To pa nam olajšujejo funkcije in lastnosti permutacij, v pravilno implementacijo pa nas vodi prevajalnik.

5.12 Združevanje dokazov

5.12.1 Dokaz o urejenosti navadnega seznama

Pogosto uporabljamo algoritme za dosego nekega bolj kompleksnega cilja. Uporabljamo jih skupaj z drugimi algoritmi in iz njih kujemo nove. Povezovanje različnih algoritmov, ki operirajo nad podatkovnimi strukturami, je včasih lahko težavno, saj moramo rokovati z različnimi podatkovnimi tipi.

Želimo si, da bi algoritmi rokovali tudi z običajnimi podatkovnimi tipi iz standardne knjižnice, hkrati pa bi vseeno lahko dokazali njihovo pravilnost. To želimo tudi pri urejanju. Urejanje z vstavljanjem bi vračalo urejen seznam kot navaden seznam iz standardne knjižnice, zraven pa bi dobili še dokaz, da je seznam sortiran.

Definirajmo podatkovni tip dokaza *Sorted*, ki bo kot odvisne vrednosti vseboval seznam naravnih števil ter zgornjo in spodnjo mejo urejenega seznama. Z drugimi besedami, posebnosti urejenega seznama prestavimo v odvisni podatkovni tip.

```
data Sorted: List Nat -> BNat -> BNat -> Type where
  SrNil : BLTE l u -> Sorted Nil l u
  SrCons : Sorted xs (BLift x) u ->
             BLTE l (BLift x) -> Sorted (x :: xs) l u
```

Konstruktorja *SrNil* in *SrCons* sta zelo podobna konstruktorjema *SNil* in *SCons*. To nikakor ni naključje, saj smo jih izpeljali prav iz njih.

Popravimo lahko tudi definicijo funkcije *insert*, da bo namesto tipa *SortedList* vračala kar navaden seznam v odvisnem paru z dokazom o njegovi urejenosti (*Sorted*). Kodo popravljene funkcije *insert* lahko vidimo na sliki 5.6.

```
insert : {l, u : BNat} -> (x : Nat) -> (xs : List Nat) ->
           Sorted xs l u -> BLTE l (BLift x) -> BLTE (BLift x) u ->
           (ys : List Nat ** Sorted ys l u)
insert x [] (SrNil lleu) llex xleu = ([x] ** SrCons (SrNil xleu) llex)
insert x (y :: ys) (SrCons yssrt lley) llex xleu with (lteTotal x y)
  | Left  xley =
      let nbnd = SrCons (SrCons yssrt (BLifted xley)) llex in
        (x :: y :: ys ** nbnd)
  | Right xgty =
      let (nys ** nbnd) = insert x ys yssrt (BLifted xgty) xleu in
        (y :: nys ** SrCons nbnd lley)
```

Slika 5.6: Funkcija *insert*, ki rokuje z navadnimi seznami.

```
insertionSort : List Nat -> (ys : List Nat ** Sorted ys Bottom Top)
insertionSort [] = ([] ** SrNil BottomLower)
insertionSort (x :: xs) = let (nxs ** nxssrt) = (insertionSort xs) in
                              insert x nxs nxssrt BottomLower TopGreater
```

Slika 5.7: Urejanje z vstavljanjem, ki rokuje nad seznami.

Da zaključimo algoritem za urejanje, ki rokuje z navadnimi seznami, si na sliki 5.7 oglejmo še implementacijo celotnega urejanja, ki vse elemente zaključi v celoto.

5.12.2 Združen dokaz o urejenosti

Do sedaj smo pokazali, kako definirati in dokazati podatkovni tip *SortedList*, ki ohranja urejenost. Nato smo njegove lastnosti izluščili v nov podatkovni

tip *Sorted*, ki služi kot dokaz k vrednostim navadnih seznamov in potrjuje njihovo urejenost. Ugotovili smo, da samo dokaz o pravilnem vrstnem redu elementov oziroma urejenosti ni dovolj, da se izognemo vsem napakam. Da zagotovimo, da v urejen seznam vstavimo vse elemente iz izvornega in nobenega drugega elementa, smo pokazali, kako dokazati relacijo permutranosti med dvema seznamoma. Za naš algoritem smo potrdili, da res vrača permutacije izvornega seznama.

Vendar pa smo vsako od lastnosti dokazali posebej, kar pomeni, da bi se še vedno lahko zmotili. Na primer, implementacija, ki dokazuje permutiranost, bi lahko vračala sezname, ki ne bi bili pravilno urejeni. Prav tako bi lahko implementacija, ki dokazuje urejenost, vračala urejene sezname, ki ne bi bili permutacije izvornih. Temu se lahko izognemo le na en način. Hkrati zahtevamo dokaz o pravilnem vrstnem redu in permutiranosti. Istočasno pa bi želeli ohraniti fleksibilnost navadnih seznamov. Preostane nam tako le znan pristop z odvisnimi pari.

V implementaciji, ki dokazuje urejenost, smo kot odvisni element odvisnega para zahtevali dokaz tipa *Sorted*. Ko smo dokazovali permutiranost, pa je bil v odvisnem paru dokaz tipa *Perm*. Oba dokaza lahko združimo v eni implementaciji, tako da v odvisnem paru zahtevamo par oziroma dvojico, ki vsebuje oba dokaza.

Tip nove funkcije *insert* bo združen iz prejšnjih implementacij. Iz izhodnih tipov

```
(ys : List Nat ** Sorted ys l u)
```

in

```
(ys : List Nat ** Perm (x :: xs) ys)
```

bomo napravili združeno funkcijo *insert*, ki bo imela vrnitveni tip enak

```
(ys : List Nat ** (Sorted ys l u, Perm (x :: xs) ys)).
```

Implementacija je prikazana na sliki 5.8. Vsebuje združeno funkcionalnost prejšnjih implementacij, tako da posebni komentarji niso potrebni. Omeniti velja, da zaradi računanja z dokazi postane izvorna koda veliko manj berljiva, bistvo algoritma pa ni več tako očitno kot v nepreverjeni različici.

```
insert : {l, u : BNat} -> (x : Nat) -> (xs : List Nat) ->
         Sorted xs l u -> BLTE l (BLift x) -> BLTE (BLift x) u ->
         (ys : List Nat ** (Sorted ys l u, Perm (x :: xs) ys))
insert x [] (SrNil lleu) llex xleu =
         ([x] ** (SrCons (SrNil xleu) llex, PrSame refl))
insert x (y :: ys) (SrCons yssrt lley) llex xleu with (lteTotal x y)
   | Left  xley =
       let nbnd = SrCons (SrCons yssrt (BLifted xley)) llex in
         (x :: y :: ys ** (nbnd, PrSame refl))
   | Right xgty =
       let (nys ** (nbnd, nperm)) =
         insert x ys yssrt (BLifted xgty) xleu in
       let perm1 = PrPrep y (x :: ys) nys nperm in
       let perm2 = PrSwap x y ys in
       let perm3 = PrTrans perm2 perm1 in
         (y :: nys ** (SrCons nbnd lley, perm3))
```

Slika 5.8: Funkcija *insert*, ki ohranja urejenost in permutiranost.

Dokaz pravilnosti urejanja z vstavljanjem zaključimo še v celoto s funkcijama *insert* in *insertionSort*. V svojem tipu zahteva, da je seznam zs, ki ga funkcija vrača, urejen (*Sorted zs Bottom Top*) in da je permutacija vhodnega seznama xs (*Perm xs zs*), oboje pa kot pri funkciji *insert* najdemo kot dvojico v odvisnem paru.

Popolnoma dokazana različica vstavljanja z urejanjem je prikazana na sliki 5.9. Služi nam lahko tudi za primer, kako združevati več različnih dokazov v eno implementacijo. Če bi morali dokazati še kakšno lastnost, bi jo prav tako lahko dodali v odvisni par kot dvojico ali trojico, lahko pa bi definirali tudi nov podatkovni tip, ki bi vseboval dokaze več različnih lastnosti.

Za zaključek se spodobi omeniti, da je za dokazovanje pravilnosti podatkovnih struktur in algoritmov potrebno najprej globoko razumevanje problema in njegovih lastnosti. Prvi korak je vsekakor njihovo prepoznavanje,

```
insertionSort : (xs : List Nat) ->
                (zs : List Nat ** (Sorted zs Bottom Top, Perm xs zs))
insertionSort [] = ([] ** (SrNil BottomLower, PrNil))
insertionSort (y :: ys) =
  let (nys ** (nyssrt, perm)) = insertionSort ys in
  let (sorted ** (srtprf, srtperm)) =
    insert y nys nyssrt BottomLower TopGreater in
  let perm1 = PrPrep y ys nys perm in
  let perm2 = PrTrans perm1 srtperm in
      (sorted ** (srtprf, perm2))
```

Slika 5.9: Dokazana različica urejanja z vstavljanjem.

nato pa sledi razmislek, kako jih formalizirati in zapisati v obliki, ki bo "razumljiva" tudi prevajalniku. Ko je oblika lastnosti zastavljena, sledi programiranje in dokazovanje, kjer nam je v veliko pomoč prevajalnik. V večini primerov je iz tipov očitno, kakšen dokaz ali implementacija se od nas zahteva, drugje pa je potrebno malo spretnosti. Je pa pogosto, da do pravilne formulacije potrebujemo več iteracij, čemur pa smo bili priča tudi v tem razdelku o urejenih seznamih in urejanju z vstavljanjem.

Poglavje 6

Sklepne ugotovitve

V tem diplomskem delu smo se osredotočili na tipizirane programske jezike oziroma programske jezike, pri katerih izvajamo analizo skladnosti tipov v času prevajanja programa. Predstavili smo sisteme tipov kot logično ogrodje za preprečevanje napak v programih. Napake v programih se v tipiziranih programskih jezikih lahko izrazijo kot neskladje na nivoju tipov. Osredotočili smo se na sisteme tipov, kjer so tipi lahko odvisni tudi od vrednosti, torej na sisteme tipov z odvisnimi tipi. Orisali smo teoretično podlago odvisnim tipom in predstavili analogijo med izjavami in tipi funkcij ter med dokazi in implementacijami funkcij. Kot primer programskega jezika z odvisnimi tipi smo zaradi uporabnosti in splošno-namenske zasnove izbrali programski jezik Idris. Poglobili smo se v programski jezik Idris in pokazali, kako dokazujemo enostavne izjave.

V jedru diplomskega dela smo se osredotočili na programe z odvisnimi tipi. Pokazali smo kako algoritme in podatkovne strukture implementiramo s pomočjo odvisnih tipov, tako da njihovo pravilnost avtomatsko preverjamo s prevajalnikom. Definirali smo podatkovne strukture, kot so na primer povezani seznam, vrsta in sklad, ter si ogledali pogoste napake pri delu z njimi. Poglobili smo se v lastnosti teh podatkovnih struktur in podali implementacije z odvisnimi tipi, v katerih so lastnosti dokazane, pravilnost pa avtomatsko preverjena. Da bi pokazali uporabnost programskih jezikov z odvisnimi tipi,

smo podali še implementacijo algoritma urejanja seznamov z vstavljanjem. Opazili smo, da je za popolnoma dokazano različico potrebno pokazati, da spoštuje urejenost, prav tako pa tudi permutiranost. V tem diplomskem delu smo pokazali oboje. Za dokaz urejenosti je bilo potrebno pokazati tudi pravilnost relacije urejenosti, ki smo jo integrirali v dokaz pravilnosti urejanja. Na koncu smo dokaza o urejenosti in permutiranosti združili v enotno implementacijo, ki dokazuje pravilnost vstavljanja z urejanjem.

S tem diplomskim delom smo prikazali avtomatsko dokazovanje pravilnosti programov kot alternativo ročnemu dokazovanju pravilnosti. Naše delo služi kot prikaz uporabnosti avtomatskega dokazovanja, ki ga lahko uporabimo na področjih, kjer je pravilnost programov nepogrešljiva, prav tako pa tudi v vsakdanji rabi.

Literatura

[1] David Aspinall in Martin Hoffmann. Advanced topics in types and programming languages, poglavje Dependent types, 2004.

[2] Lennart Augustsson. Cayenne—a language with dependent types. V *Advanced Functional Programming*, strani 240–267. Springer, 1999.

[3] Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[4] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The Coq proof assistant reference manual: Version 6.1. 1997.

[5] Ana Bove, Peter Dybjer, in Ulf Norell. A brief overview of Agda–a functional language with dependent types. V *Theorem Proving in Higher Order Logics*, strani 73–78. Springer, 2009.

[6] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(05):552–593, 2013.

[7] Edwin Brady. Programming in Idris: A Tutorial. Technical report, University of St Andrews, 2013.

[8] Luca Cardelli. Phase distinctions in type theory, 1988.

[9] Luca Cardelli. Type systems. *ACM Computing Surveys*, 28(1):263–264, 1996.

[10] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, in Clifford Stein. *Introduction to algorithms*, druga izdaja. MIT press Cambridge, 2001.

[11] Brian A Davey in Hilary A Priestley. *Introduction to lattices and order*. Cambridge university press, 2002.

[12] Chris Hankin. *An introduction to lambda calculi for computer scientists*. King's College, 2004.

[13] J.E. Hopcroft, R. Motwani, in J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation: Pearson New International Edition*. Always learning. Pearson Education, Limited, 2013.

[14] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, et al. Report on the programming language Haskell: a non-strict, purely functional language version 1.2. *ACM SigPlan notices*, 27(5):1–164, 1992.

[15] Ritchie S King. The top 10 programming languages [the data]. *Spectrum, IEEE*, 48(10):84–84, 2011.

[16] Per Martin-Lof in Giovanni Sambin. *Intuitionistic type theory*, številka 17. Bibliopolis Naples, 1984.

[17] Francesco Mazzoli. Agda by example: Sorting, 2013. [Na spletu; obiskano 10. avgusta 2014; http://mazzo.li/posts/AgdaSort.html].

[18] Conor McBride. Epigram: Practical programming with dependent types. V *Advanced Functional Programming*, strani 130–170. Springer, 2005.

[19] Martin Odersky, Vincent Cremet, Christine Röckl, in Matthias Zenger. A nominal theory of objects with dependent types. V *ECOOP 2003–Object-Oriented Programming*, strani 201–224. Springer, 2003.

[20] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[21] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[22] Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Univalent Foundations, 2013.

[23] Blaž Repas in Jurij Mihelič. Preverjanje pravilnosti podatkovnih struktur z odvisnimi tipi v programskem jeziku Idris. *Zbornik triindvajsete mednarodne Elektrotehniške in računalniške konference ERK 2014*, 2014. (sprejeto).

[24] Anton Setzer. Object-oriented programming in dependent type theory. *Trends in Functional Programming*, 7:91–108, 2006.

[25] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, in Jean Yang. Secure distributed programming with value-dependent types. V *ACM SIGPLAN Notices*, številka 46, strani 266–278. ACM, 2011.

[26] Hongwei Xi. Xanadu: Imperative programming with dependent types, 2001. [Na spletu; obiskano 1. septembra 2014; http://www.cs.bu.edu/ hwxi/Xanadu/Xanadu.html].

[27] Hongwei Xi. Applied type system. V *Types for Proofs and Programs*, strani 394–408. Springer, 2004.

[28] Hongwei Xi. Dependent ML an approach to practical programming with dependent types. *Journal of Functional Programming*, 17(02):215–286, 2007.