UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO
FAKULTETA ZA MATEMATIKO IN FIZIKO

Luka Sabotič

Implementacija rekurzivnih podatkovnih tipov

DIPLOMSKO DELO
UNIVERZITETNI ŠTUDIJSKI PROGRAM PRVE STOPNJE
RAČUNALNIŠTVO IN MATEMATIKA

Mentor: prof. dr. Andrej Bauer

Ljubljana, 2023

Besedilo je oblikovano z urejevalnikom besedil LaTeX.
Slike so izdelane s pomočjo jezika PGF/TikZ.

Namesto te strani **vstavite** original izdane teme diplomskega dela s podpisom mentorja in dekana ter žigom fakultete, ki ga diplomant dvigne v študentskem referatu, preden odda izdelek v vezavo!

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani Luka Sabotič,

z vpisno številko 63200031,

sem avtor diplomskega dela z naslovom:

Implementacija rekurzivnih podatkovnih tipov

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom
  prof. dr. Andrej Bauer
- 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 v zbirki "Dela FRI".

V Ljubljani, dne 05.09.2023 Podpis avtorja:

Zahvala

Zahvaljujem se mentorju prof. dr. Andreju Bauerju za veliko zagretost in izjemno odzivnost pri izdelavi tega diplomskega dela.

Kazalo

Povzetek

Diplomsko delo obravnava rekurzivne podatkovne tipe, tako induktivne, kot tudi koinduktivne. Podaja njihovo definicijo in predstavi nekaj primerov, ter opiše različne pristope k njihovi implementaciji. V nadaljevanju se posveti vsotam tipov in izrazu `case`, ki lahko razčleni vsote na posamezne variante in je potreben za delo z rekurzivnimi tipi. Predstavi korake implementacije teh struktur v programskem jeziku MiniHaskell in pokaže potrebne spremembe v leksičnem analizatorju, razčlenjevalniku, preverjevalniku tipov in tolmaču. Osnovni poudarek je na praktični integraciji teh struktur v progrmaski jezik MiniHaskell, v katerem je prikazanih več primerov uporabe, ki ilustrirajo uporabnost teh struktur in odpirajo možnosti za morebitne nadaljnje razširitve.

Ključne besede:

rekurzivni tip, induktivni tip, koinduktivni tip, funkcijsko programiranje, vsota tipov

Abstract

The thesis addresses recursive data types, both inductive and coinductive. It provides their definition and presents some examples, as well as describing various approaches to their implementation. Subsequently, it focuses on sum types and the expression case, which can decompose sums into individual variants and is essential for working with recursive types. It outlines the steps for implementing these structures in the MiniHaskell programming language and illustrates the necessary changes in the lexer, parser, type checker and interpreter. The primary emphasis lies on the practical integration of these structures into the MiniHaskell programming language, where several usage examples are demonstrated, showcasing the utility of these structures and paving the way for potential further extensions.

Key words:

recursive type, inductive type, coinductive type, functional programming, sum type

Poglavje 1

Rekurzivni tipi

V programiranju se vsakodnevno srečujemo z velikim številom konceptov in idej, ki nam na različne načine omogočajo iskati rešitve. Eden najbolj osnovnih in popularnih je tudi rekurzija. V osnovi rekurzija predstavlja eleganten način reševanja zapletenih problemov z razčlenitvijo na manjše probleme iste vrste, ki se nato lahko razčlenjujejo naprej. Ta sposobnost reševanja zapletenih izzivov postopoma ni le preoblikovala načina, kako se programerji lotevajo kodiranja, temveč je tudi postavila temelje za ustvarjanje razreda podatkovnih struktur, imenovanih rekurzivne podatkovne strukture.

Tako kot rekurzivna funkcija pokliče samo sebe, da reši problem v manjših korakih, rekurzivni tipi opredeljujejo strukture, ki vsebujejo podatke istega tipa in ustvarjajo strukturirane samo-referenčne vzorce. S tem, ko dovoljujejo da so elementi strukture sestavljeni iz primerkov iste strukture, oponašajo način, kako zaznavamo in opisujemo svet okoli nas. Na primer datotečni sistem, kjer lahko mape vsebujejo podmape, ki pa spet vsebujejo več map in datotek. Podobno v družinskem drevesu: posamezniki imajo otroke, ki sčasoma sami postanejo starši. Rekurzivne strukture omogočajo, da te kompleksne odnose opišemo na preprost in intuitiven način. Vsebina tega pogavja je povzeta po viru [2].

V programskih jezikih podatkovne strukture predstavimo s podatkovnimi tipi. Če so strukture rekurzivne, so taki tudi tipi. Rekurzivne tipe lahko ločimo na induktivne in koinduktivne, elementi prvih lahko nosijo le končne podatke, elementi drugih pa so lahko tudi neskončni [3]. Morda najbolj osnoven primer rekurzivnega tipa je seznam. Ta je lahko prazen, ali pa vsebuje urejen par nekega elementa in drugega seznama. Elementu ponavadi rečemo glava, seznamu, ki glavi sledi, pa rep. Tako ima poljubno dolžino, lahko pa je tudi prazen. V prikazovanju primerov bom za prazen seznam uporabljal konstruktor `Empty`, ki ne sprejme nobenega argumenta in predstavlja odsotnost vrednosti, pogosto označena kot `nil`. Oglejmo si seznam celih števil:

```
data Seznam = Empty | Cons Int Seznam
```

Primer seznama celih števil, ki vsebuje elemente 1, 2 in 3:

```
let enaDoTri = Cons 1 (Cons 2 (Cons 3 Empty))
val enaDoTri : Seznam
```

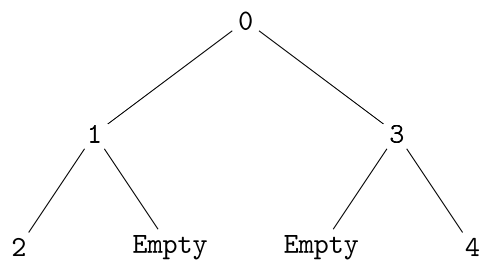
Seznamom podoben primer so drevesa. Ta so sestavljena iz vozlišč, vsako je lahko končno, ali pa ima potomce, ki so spet drevesa.

```
data Drevo = Empty | List Int
                   | Vozlisce Int Drevo Drevo
```

Primer drevesa, ki vsebuje cela števila:

```
let smreka = Vozlisce 0 (Vozlisce 1 (List 2) Empty)
             (Vozlisce 3 Empty (List 4))
val smreka : Drevo
```

Drevesa si pogosto predstavljamo kot grafe. Vrednost spremenljivke `smreka` torej izgleda kot



V svetu programiranja so seznami in drevesa ključne podatkovne strukture, ki jih uporabljamo na različne načine. Razvijajo se inovativne implementacije in algoritmi za delo z njimi in so nujni za razumevanje programiranja. Seveda pa so to hkrati le osnovni primeri rekurzivnih tipov. V nadaljevanju bom predstavil nekaj bolj zanimivih primerov, ki so prav tako zelo uporabni.

1.1 Koinduktivni podatkovni tipi

Seznami in drevesa so induktivni rekurzivni tipi, kar je lahko zelo prikladno, ker jih lahko podamo v algoritme. Lahko pa se zgodi, da porebujemo podatkovni tip, ki nam omogoča dostop do neomejene količine podatkov, za kar potrebujemo koinduktivne tipe. Koinduktivne tipe v splošnem najdemo v procesih, ki so lahko neskončni, najbolj so povezani z uporabo v komunikacijah, kjer ni potrebe, da se kanal kdaj zapre.

Primer koinduktivnih tipov je tip lačnih funkcij. To so funkcije, ki sprejmejo argument in vrnejo novo funkcijo, ki je lačna novega argumenta.

$$\texttt{Lacna} = \texttt{Argument} \to \texttt{Lacna}$$

Na primer, lahko imamo funkcijo, ki sprejme število in vrne funkcijo, ki sprejme novo število:

```
let f = λ n : Int. f
val f : Lacna
```

Ko pokličemo f, bo ta vrnila novo funkcijo, ki bo lačna novega argumenta:

```
let lacna1 = f 1
val lacna1 : Lacna
```

Rezultatu lahko nato podamo nov argument in dobili bomo enak rezultat:

```
let lacna2 = lacna1 2
val lacna2 : Lacna
```

Tako funkcijo lahko gledamo tudi kot funkcijo, ki sprejme neomejeno količino argumentov in bo še vedno lačna novih:

```
let zeloLacna = f 1 2 3 4 5 6 7 8 9
val zeloLacna : Lacna
```

V resnici je `zeloLacna` sestavljena iz več lačnih funkcij, ki se poračunajo sproti z vsakim argumentom. Lahko bi jo zapisali tudi kot:

```
let zeloLacna = ((((((((f 1) 2) 3) 4) 5) 6) 7) 8) 9
val zeloLacna : Lacna
```

Tako se `1` uporabi kot argument na funkciji `f`, `2` na funkciji, ki jo vrne `f`, ko sprejme `1` in tako naprej.

Zgornji primer je zabaven, ampak ni preveč uporaben v praksi. Poglejmo si bolj znane lačne funkcije – tokove. To so funkcije, ki sprejmejo enotske vrednosti (`Unit`) in vrnejo pare elementov in novih tokov:

$$\texttt{Tok} = \texttt{Unit} \to \texttt{Int} \times \texttt{X}$$

Tokove si lahko predstavljamo kot neskončne sezname, sestavljene iz parov njihovih elementov in novih tokov. Na primer, lahko imamo tok naravnih števil:

```
let naravna = λn : Int. (n, naravna (n+1))
naravna : Tok
```

Za delo s tako funkcijo potrebujemo še nekaj pomožnih funkcij. Najprej funkcijo, ki vrne prvi element, ali glavo toka: (imejmo 1 za indeks prvega elementa in 2 za indeks drugega)

```
let glava = λt : Tok. t.1
val glava : Tok → Int
```

in še funkcijo, ki vrne rep toka:

```
let rep = λt:Tok. t.2
val rep : Tok → Tok
```

Tako lahko dostopamo do poljubnega elementa v toku:

```
glava (rep (rep (rep naravna)))
- : 3 : Int
```

Tokovi se uporabljajo, ko pričakujemo neskončno zaporedje podatkov. V lenih programskih jezikih, kot je tudi MiniHaskell, rep neskončne dolžine ne predstavlja problema, saj se vrednosti računajo le po potrebi. Oglejmo si še nadgradnjo tokov, enostavne procese. To so funkcije, ki sprejmejo nek element in vrnejo par elementa in novega procesa:

$$\texttt{Proces} = \texttt{Int} \to \texttt{Int} \times \texttt{Proces}$$

Primer procesa je recimo enocelični pomnilnik, ki shranjuje po eno vrednost in ob prejemu argumenta vrne do sedaj shranjen element in shrani novega:

```
let pomnilnik = λe : Int.
                λn : Int. (e , pomnilnik (n ))
val pomnilnik : Proces
```

Podobno kot pri tokovih, za delo s procesi potrebujemo pomožne funkcije, kot je na primer funkcija, ki vrne vrnjeno vrednost procesa:

```
let vrednost = λp : Proces. p.1
val vrednost : Proces → Int
```

Koinduktivni tipi so torej zelo uporabni, ker nam omogočajo delo z neskončnimi podatki. Vendar je potrebno biti previden, ker lahko hitro pride do neskončnih zank. Lažje jih je obvladovati v lenih programskih jezikih, ker se njihova vsebina nikoli ne izračuna do konca, vedno samo po potrebi.

1.2 Ekvirekurziven ali izorekurziven pristop?

Ko implementiramo rekurzivne tipe, se moramo slej ko prej vprašati, kaj je razlika med tipom in čemer dobimo, ko ta tip enkrat »odvijemo«. Na primer, kaj je razlika med tipom `Seznam` in njegovim enkratnim odvojem `Empty` | `Cons Int Seznam`? V literaturi pojavita dva pristopa k temu vprašanju: ekvirekurziven in izorekurziven.

Definirajmo operator μ, ki računa negibne točke. Za dano funkcijo F, ki slika tipe v tipe, naj bo $\mu(F)$ tip, za katerega velja $\mu(F) = F(\mu(F))$. Običajno namesto $\mu(\lambda X.T)$ pišem $\mu X.T$. Na primer, tip seznamov celih števil lahko zapišemo kot $\mu(\lambda X.\ \texttt{Empty} \mid \texttt{Cons Int}\ X)$ ali raje $\mu X.\texttt{Empty} \mid \texttt{Cons Int}\ X$.

1.2.1 Ekvirekurziven pristop

Ekvirekurziven pristop pravi, da je tip *μX.F(X)* po definiciji enak njegovemu odvoju *F(μX.F(X))*, ker oba predstavljata enaka neskončna drevesa. Ta pristop nato od preverjevalnika tipov zahteva, da upošteva enačbo $\mu X.F(X) = F(\mu X.F(X))$, kakor tudi vse enačbe, ki sledijo iz te, na primer $\mu X.F(X) = F(F(F(\mu X.F(X))))$.

1.2.2 Izorekurziven pristop

Izorekurziven način pa ubere na prvi pogled nekoliko bolj zapleteno pot. Definira dve preslikavi, imenujmo ju `fold` in `unfold`, ki nam omogočata prehode med rekurzivnimi tipi in njihovimi odvoji. `Fold` sprejme element odvitega tipa in ga preslika v element rekurzivnega tipa, `unfold` pa ravno obratno, element rekurzivnega tipa preslika v element odvitega tipa.

$$\begin{aligned} \texttt{fold} &: F\ (\mu F) \mapsto (\mu F) \\ \texttt{unfold} &: (\mu F) \mapsto F\ (\mu F) \end{aligned}$$

Preslikavi sta inverzni, torej sta izomorfizma.

$$\begin{aligned} \texttt{fold}(\texttt{unfold}(\texttt{x})) &= \texttt{x} \\ \texttt{unfold}(\texttt{fold}(\texttt{y})) &= \texttt{y} \end{aligned}$$

Na primeru seznamov:

$$\begin{aligned} \texttt{fold} &: (\texttt{Empty} \mid \texttt{Cons Int Seznam}) \mapsto \texttt{Seznam} \\ \texttt{fold} &: \texttt{Seznam} \mapsto (\texttt{Empty} \mid \texttt{Cons Int Seznam}) \end{aligned}$$

Oba pristopa se uporabljata pri konstrukciji programskih jezikov in teoretičnih besedilih. Ekvirekurziven je bolj intuitiven, vednar zahteva več dela od preverjevalnika tipov in lahko privede do težav pri kombinaciji z drugimi konstrukti, na primer operatorjih na tipih. Medtem izorekurziven pristop zahteva uporabo izomorfizmov in več dela preloži na programerja. V svoji implementaciji sem uporabil slednjega, ker je pogosteje uporabljen, zahteva pa tudi manj dela na prevajalniku. Funkciji `fold` in `unfold` lahko uporabnik definira po potrebi, vendar v veliko primerih to ni potrebno, ker zadoščuje uporaba konstruktorjev in izrazov `case`.

Poglavje 2

Vsote tipov

Programerji se pogosto srečujemo z različnimi strukturami ali spremenljivkami, ki lahko zavzamejo vrednosti iz množice možnosti. Na primer, vozlišče v drevesu je lahko prazno, list, ali notranje vozlišče. Element v povezanem seznamu je lahko trivialna vrednost `Empty`, ali pa vozlišče `Cons` z glavo in repom seznama. Takih primerov je veliko, zato poznamo vsote tipov. Snov tega poglavja se zgleduje po viru [2].

Vsote tipov so podatkovni tipi, ki izvirajo iz množice vrednosti, dobljene iz kombinacije več tipov. Na primer, če imamo tipa

```
data Odrasel = Odr Int Bool Int
data Otrok = Otr Int
```

s katerima povemo koliko je odrasel človek star, če je poročen in koliko otrok ima, ter koliko je star otrok in ju želimo združiti v en tip, da bi lahko na primer naredili seznam, ki vključuje odrasle in otroke, lahko definiramo vsoto `Oseba`.

```
data Oseba = Odr Int Bool Int | Otr Int
```

Vsak element tipa `Oseba` je označen kot varianta `Odr`, ki ustreza tipu `Odrasel`, ali kot varianta `Otr`, ki ustreza tipu `Otrok`.

Da lahko nato delamo z vsotami, moramo imeti funkcije, ki znajo ločevati med njihovimi elementi in jih obravnavati posebej. To naredimo z uporabo izraza `case`. Če nas na primer zanima starost vrednosti tipa `Oseba`, lahko definiramo funkcijo `starost`:

```
let starost = fun o:Oseba ⇒
    case o of
        Odr x y z → x
      | Otr x → x
    end
```

Ko je argument `o` tipa `Odrasel`, se izvede prva veja in vrne izobrazba odraslega, ko pa je argument o tipa `Otrok`, se izvede druga veja. Tako je tip celotne funkcije `Oseba` → `Int`. Seveda zahtevamo, da vse veje vrnejo izraz istega tipa.

Ker v izrazu `case` preverjamo tip argumenta na način, da samo preverimo ali ustreza kakšni izmed variant in se ne oziramo na preostale tipe v tisti vsoti, se lahko zgodi, da je izrazu možno dodeliti več kot en tip. To se zgodi, če ima več vsot enak seštevanec. Za primer vzemimo vsoti

```
data A = Foo Int | Bar Bool
data B = Foo Int | Baz Int
```

Obe imata varianto, označeno s `Foo`, torej kateremu tipu pripada izraz `Foo 42`? V praksi se ta problem rešuje na različne načine, jaz pa sem v svoji implementaciji določil, da se vrednosti dodeli tip, ki ga preverjevalnik tipov najprej najde, torej tistega, ki je bil definiran nazadnje. V tem primeru je to tip `B`.

```
Foo 42
- : B = 42
```

Morda zanimiva skupina vsot so tiste, ki lahko vsebujejo tudi trivialne vrednosti `Unit`, ki je tip z enim elementom:

```
data Unit = T
```

Ker ima tip `Unit` lahko le en element `T`, sta zapisa `Nic Unit` in `Nic` izomorfna. Oba imata le eno možno vrednost. Vsote, ksterih elementi lahko zavzamejo tudi trivialno vrednost, izgledajo kot:

```
data Opcijsko = Nic | Vrednost Tip
```

Ti tipi so izomorfni s tistimi, katerih elementi imajo tip `Tip` in razširjeni z možnostjo trivialne vrednosti. To so na primer tipi, ki jih poznamo iz priljubljenih programskih jezikov in dopuščajo vrednosti kot so `null`, `None` ali `nil`.

Še dve posebni vrsti vsot sta dovolj zanimivi za posebno obravnavo.

2.1 Oštevilčenja

Poleg tipa `Unit`, ki sestoji iz ene konstante, imamo tudi vsote, sestavljene iz več njih. To so vsote, ki vsebujejo zgolj trivialne vrendosti in so namenjene predstavljanju tipov, ki sestojijo iz končno mnogo konstant. Na primer, če želimo predstaviti dele dneva, definiramo vsoto

```
data DelDneva = Jutro | Dopoldan | Popoldan | Vecer
                | Noc
```

Lahko sestavimo tudi funkcije, ki obravnavajo take vrednosti. Na primer

```
let primerenZaZajtrkovanje = fun d:DelDneva ⇒
    case d of
        Jutro → true
      | Dopoldan → true
      | _ → false
    end
```

je funkcija, ki sprejme eno izmed delov dneva in vrne odgovor na vprašanje, če takrat običajno zajtrkujemp. Tip te funkcije je `DelDneva` → `Bool`.

2.2 Vsote z eno varianto

Možno je ustvariti tudi vsote s samo eno varianto, ki vsebuje vrednosti danega tipa `T`:

```
data V = C T
```

To je lahko zelo uporabno, ker tako elementov tipa `T` ni možno zamešati za elemente tipa `V` in posledično na njih ne moremo izvajati operacij, ki jih lahko na tipu `V`. Tako se lahko izognemo nesmiselnim primerom. Recimo, da uporabljamo podatke o valutah. Količino denarja v posamezni valuti lahko predstavimo s tipom `Float`, kot decimalno število. Težava nastane, ko definiramo funkcijo, ki pretvarja med valutami:

```
let dolarjiVEvre = fun d:Float ⇒ x * 0.92
val dolarjiVEvre : Float → Float
```

Če je `d`, ki ga podamo funkciji `dolarjiVEvre` količina denarja v dolarjih, je vse v redu. Vendar pa nam nič ne preprečuje, da bi kot argument v `dolarjiVEvre` podali katerokoli drugo število, na primer količino denarja v frankih, ali še huje, število število komarjev v nekem prostoru. Taka uporaba te funkcije je nesmiselna in se ji želimo izogniti, katr storimo tako, da definiramo vsote z eno varianto:

```
data Evri = Evri Float
data Dolarji = Dolarji Float
```

Pozorni moramo biti na različne pomene pojavitev besed z veliko začetnico. V zgornji definiciji prva pojavitev besede `Evri` predstavlja ime tipa, druga pa konstruktor, ki sprejme en argument in vrne vrednost tipa `Evri`. Funkcijo `dolarjiVEvre` spremenimo tako, da deluje s pravilnimi tipi, torej pretvorbo omogočimo samo iz dolarjev v evre in nič drugega:

```
let dolarjiVEvre = fun d:Dolarji ⇒
    case d of
        Dolarji x → Evri (x * 0.92)
    end
val dolarjiVEvre : Dolarji → Evri
```

Tako se zavarujemo pred napakami.

Poglavje 3

Implementacija

Predstavljene konstrukte sem tudi sam implementiral. To sem storil v programskem jeziku MiniHaskell, ki ga je ustvaril prof. dr. Andrej Bauer in je dostopen v repozitoriju [1].

MiniHaskell je programski jezik, ki je namenjen predstavitvi osnovnih konceptov funkcijskega programiranja. Napisan je v programskem jeziku OCaml in po strukturi in načinu uporabe sledi jeziku Haskell. Omogoča uporabo celih števil, boolovih vrednosti z logičnimi operacijami in primerjavami celih števil, urejenih parov, seznamov, funkcij in rekurzije. Temu sem dodal možnost definiranja novih tipov, rekurzivnih ali ne, in uporabe izrazov `case` za delo z njimi. Programski jezik sestavljajo leksična analiza, razčlenjevalnik, tolmač in preverjevalnik tipov. V nadaljevanju bom predstavil kako delujejo v MiniHaskellu, ter kako sem jih dopolnil.

3.1 Leksična analiza

Prvi korak, ki ga je potrebno narediti, ko dobimo izvorno kodo in jo želimo izvesti, je leksična analiza. To je postopek, kjer vhodni niz znakov očistimo znakov, ki ne nosijo pomembnih informacij, kot na primer presledki, in jih razdelimo na gradnike. To so vse ključne besede, ki jih jezik pozna, števila, znaki in podobno. To stori leksični analizator ali lekser. Ta je v našem primeru napisan v datoteki `lexer.mll`. Je datoteka, kjer so shranjeni regularni izrazi, ki opisujejo gradnike, ki jih lahko uporabljamo v programskem jeziku. Kot je to storjeno v Haskellu, sem določil, da bom imena spremenljvk dovolil le z majhno začetnico, imena konstruktorjev tipov pa le z veliko. Tako sem si olajšal delo v razčlenjevalniku. Gradnik, ki bo predstavljal imena konstruktorjev in imena tipov sem poimenoval `cname`, kot okrajšavo za *capital name* ali ime z veliko začetnico. Poleg načina poimenovanja konstruktov je v lekserju definiranih še veliko drugih gradnikov, kot so olkepaji, znaki za operacije, ključne besede, ukaz `:quit` za izhod iz programa in podobno. Za definicijo podatkovnih tipov sem dodal ključno besedo `data`, z idejo, da bo imela enako vlogo kot tista v Haskellu. Tako imamo vse potrebno za definicijo novih podatkovnih tipov, ker smo definirali besedo `data`, imamo `cname` za definicijo konstruktorjev in že od prej možnost definiranja spremenljivk. Za delo s podatkovnimi tipi, torej izraz `case`, je bilo potrebno dodati še gradnike za ključne besede `case`, `of` in `end`.[1][2] V Haskellu sicer gradnika za `end` ne poznamo, vendar ker za razliko od Haskella MiniHaskell ni občutljiv na zamik vrstic v kodi, in ker nisem želel z implementacijo še tega zaiti s smeri moje diplomske naloge, sem za lažje delo v razčlenjevalnikju dodal tudi ta gradnik. Da bi razumeli zakaj je potreben, si poglejmo naslednji primer: recimo, da imamo izraz `case`, ki v eni od svojih vej vsebuje še en, gnezden izraz `case`:

```
case x of
  a → _
  b → case y of
        c → _
        d → _
  e → _
```

Če bi bil MiniHaskell občutljiv na zamik vrstic, bi lahko zgornji izraz razčlenili enolično. Če pa zamikov ne upoštevamo, ga je možno razčleniti na dva načina:

```
case x of
  a → _
  b → (case y of
         c → _
         d → _)
  e → _
```

ali pa:

```
case x of
  a → _
  b → (case y of
         c → _)
  d → _
```

[1]Gradnik `ALTERNATIVE` za znak | je že bil definiran, ker je MiniHaskell že imel sezname in funkcijo `match` za delo z njimi.

[2]Gradnik *TARROW* za znak →, ki bo v veji *case* izraza ločil med izrazom, ki ga primerjamo in posledico, ki se sproži v primeru ujemanja, je prav tako že bil definiran.

```
  e → _
```

Ker imajo različni programerji različna mnenja o tem, kateri od obeh je bolj smiselen in sem se želel izogniti kakeršnikoli dvoumnosti, sem dodal gradnik `end`, ki nam omogoča, da enostavno določimo, kdaj se izraz `case` konča:

```
case x of
  a → _
  b → case y of
        c → _
        d → _
      end
  e → _
end
```

Prav tako imamo v MiniHaskellu težavo z vpisovanjem izrazov na zgornji način, saj je prilagojen le vpisovanju ukazov v ukazno vsrtico, ki pa sprejema le nize znakov brez znaka za novo vrstico (*newline* ali `\textbackslash n`). Tako nastane težava pri iskanju meje med posameznimi vejami `case`-a. Haskell za to uporablja le znake za novo vrstico, jaz pa sem dodal obvezno uporabo znaka | na začetku vsake veje, ki ni prva. Tako bi zgornji primer izgledal takole:

```
case x of
    a → _
  | b → case y of
            c → _
          | d → _
        end
  | e → _
end
```

3.2 Razčlenjevanje

Razčlenjevalnik ali parser je naslednji korak v procesu prevajanja progamov. Kot je to početo v praksi, tudi v MiniHaskellu ni razčlenjevalnik napisan na roke, temveč je uporabljen generator razčlenjevalnikov. To je program, ki iz podanih slovničnih pravil sestavi kodo za pripadajoči razčlenjevalnik. Ker se tukaj uporabi Menhirjev OCaml parser generator, končnica datoteke ni .ml, ampak .mly. Lekser posreduje tok gradnikov, ki jih je prepoznal, naloga razčlenjevalnikja pa je prepoznati slovnično pravilne stavke in zgraditi abstraktna sintaktična drevesa. To stori s pomočjo slovničnih pravil, ki povejo, v kakšnem zaporedju pričakujemo gradnike in kje se ti smejo pojaviti. V svoji implementaciji sem dodal dve gramatični pravili: eno za definicijo novih tipov in eno za izraz `case`. Pravilo za definicijo novih tipov je sledeče:

```
datadef:
| DATA CNAME EQUAL data_variants,

data_variants:
| data_variant
| data_variant ALTERNATIVE data_variants

data_variant:
| CNAME list(ty)
```

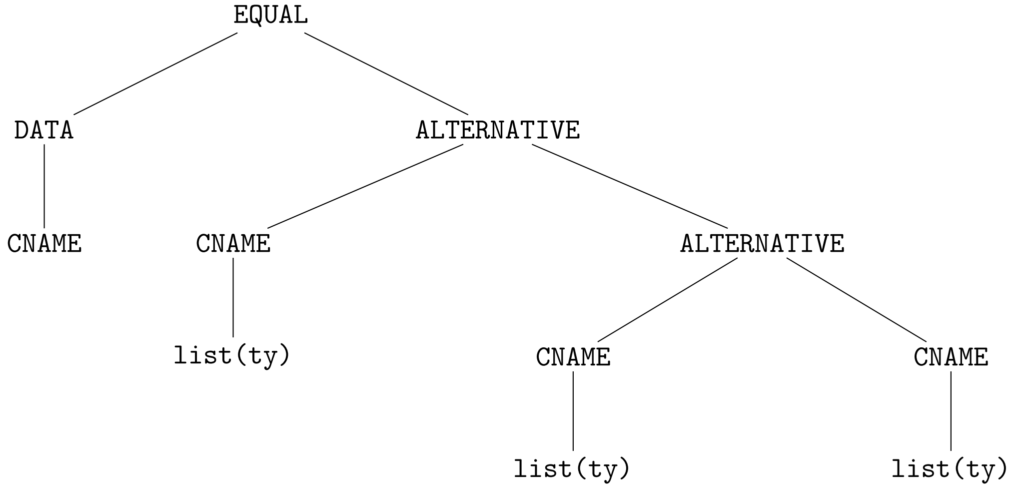
ki omogoča definicijo tipov na način:

```
data Type = Constr_1 args_1 | Constr_2 args_2 |
            ... | Constr_n args_n.
```

Seveda morajo biti TYPE in CONSTR imena z veliko začetnico. Slovnična pravila lahko zapišemo tudi z abstraktnimi sintaktičnimi drevesi. Na primer, definicijo tipa:

```
data Type = Constr_1 args_1 | Constr_2 args_2 |
            Constr_3 args_3,
```

lahko predstavimo z drevesom:



Definicijo tipov sem kot `datadeftop` definiral kot enega izmed štirih osnovnih ukazov. Pred njim so bili že definirani `lettop`, ki omogoča definicijo spremenljivk, `exprtop`, ki omogoča definicijo izrazov in `cmdtop`, ki vsebuje posebne ukaze, kot so na primer `quit`.

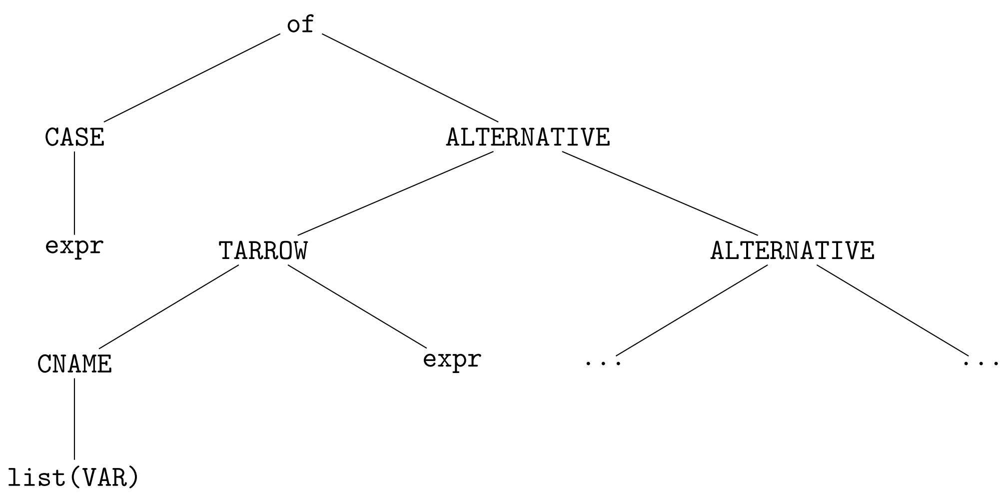
Slovnično pravilo za izraz `case` pa sem dodal samo pod pravilo za `expr` ali izraze, ker se `case` ne uporabljav v definicijo spremenljivk ali definiciji tipov, niti to ni poseben ukaz. Pravilo je:

```
expr:
...
| CASE expr OF cases END

case_variants:
| case_variant
| case_variant ALTERNATIVE case_variants

  case_variant:
| pattern TARROW expr

  pattern:
| CNAME list(VAR)
```

in omogoča uporabo izrazov `case`, kot so:

```
case x of
  Constr 1 → _
  Constr 2 → _
  ...
  Constr n → _
end
```

ki jih seveda lahko prav tako zapišemo kot abstraktno sintaktično drevo:



3.3 Abstraktna sintaksa

Ko imamo zgrajena drevesa iz gradnikov, je potrebno ugotoviti, kaj pomenijo in čemu so namenjeni, preden lahko program začnemo izvajati. Tu pride na vrsto abstraktna sintaksa. V datoteki `syntax.ml` so najprej definirani podatkovni tipi, v katere razčlenjevalnik predela izvorno kodo. Eden izmed njih je na primer `htype`, krajše za *haskell type*, ki hrani vrednosti tipov, ki jih MiniHaskell pozna na začetku, torej `TInt`, `TBool`, `TTimes`, `Tarrow`, `TList` in `TData`. Črke `T` na začetku poimenovanj služijo kot oznaka, da gre za tip. Morda nesamoumeven tip je `TArrow`, ki predstavlja funkcije. Hrani dva tipa, tip argumenta in tip vrnjene vrednosti. Spomnimo se, da v Haskellu ne obstajajo funkcije večih argumentov, temveč funkcije lahko vrnejo tudi nove funkcije, ki sprejmejo nove argumente in tako simulirajo sprejem večih argumetnov. Na primer: funkcija, ki sešteje dve celi števili, tipa `Int` → `Int` → `Int`, je v resnici funkcija, ki sprejme prvi argument tipa `Int` in vrne novo funkcijo, ki sprejme drugi argument in vrne rezultat:

```
sestej :: Int → Int → Int
sestej x y = x + y
```

v resnici izgleda kot:

```
sestej :: Int → (Int → Int)
sestej = \x → \y → x + y
```

Med tipe, ki jih pozna MiniHaskell sem moral dodati tip `TData`, ki je namenjen shranjevanju imen konstruktorjev in imen tipov. Sam ima tip `string`, ampak kot omenjeno, se vanj shranjejo le besede z veliko začetnico.

Morda bolj zanimiv je tip `toplevel_cmd` ali *toplevel command*, ki kot omenjeno zgoraj, lahko zavzame eno izmed štirih vrendnosti: `Expr`, ki shranjuje izraze, `Def`, ki shrnajuje definicije spremenljivk, `DataDef`, ki shranjuje definicije novih tipov in `Quit`, ki označuje ukaz za izhod iz `toplevel`, torej *minihaskell.exe*. Odločil sem se, da bo `DataDef` shranjeval definicije novih tipov kot urejen par imena tipa in seznama konstruktorjev in njihovih argumentov:

```
type toplevel_cmd =
| Expr of expr
| Def of name * expr
| DataDef of cname * datadef
| Quit

type datadef = (cname * htype list) list
```

Kot vidimo, so argumenti konstruktorjev tipa `htype`. To pomeni, da lahko konstruktorjem podamo tudi argumente, ki so novih tipov. To nam omogoča definicijo rekurzivnih tipov:

```
data Seznam = Empty | Cons Int Seznam
```

Izraz `case` ima bolj zapleteno strukturo. Odločil sem se, da ga bom najprej razbil na dva dela: vhodni izraz in seznam vej. V abstraktnem sintaktičnem drevesu prikazanem zgoraj, sta ta dva dela prva potomca korenskega vozlišča. Vsako vejo v seznamu sem nato razbil še naprej, na vzorec, ki ga primerjamo z vhodnim izrazom in posledico, ki se sproži v primeru ujemanja. Vzorec pa je sestavljen iz imena kontruktorja in seznamom spremenljivk, ki predstavljajo njegove morebitne argumente. V kodi ta definicija igleda takole:

```
type expr =
    ...
    | Case of expr * (pattern * expr) list

and pattern = cname * name list
```

kjer `expr` predstavlja izraz, `pattern` vzorec v veji, `cname` ime konstruktorja z veliko začetnico in `name` ime spremenljivke z malo začetnico, ki predstavlja morebitni argument konstruktorja.

Zdaj imamo že dovolj konstrukcij, da lahko definiramo svoj tip in funkcijo z izrazom `case`:

```
data Oseba = Odrasel Int Bool Int | Otrok Int

let starost = fun o : Oseba ⇒
case o of
    Odrasel x z y → x
  | Otrok x → x
end
```

Definiral sem tudi funkcijo za izpis izraza `case`, vendar je zaradi njene preprostosti ne bom ipostavljal. Na voljo je v repozitoriju.

3.4 Pravilnost tipov

Da se program lahko izvede, se morajo vsi tipi spremenljivk, funkcij in izrazov ujemati. Preveriti, da je temu tako, je naloga preverjevalnika tipov. V MiniHaskellu je zapisan v datoteki `type_check.ml`. Peverjevalnik tipov programskega jezika MiniHaskell je zelo preprost. Poleg pomožnih funkcij vsebuje kontekst, kjer hrani tipe spremenljivk, funkcijo `check`, ki preveri, če je tip danega izraza pravilen in funkcijo `type_of`, ki kakšnega tipa je dani izraz.

Da lahko preverjevalnik tipov deluje, mora poznati tipe obstoječih spremenljivk. To hrani kot seznam urejenih parov imen spremenljivk in njihovih tipov. Da pa lahko poznamo tipe spremenljivk, ki so rezultat novih konstruktorjev, moramo poznati tudi njihove tipe.

```
data Oseba = Odrasel Int Bool Int | Otrok Int

let student = Odrasel 20 false 0
val student : Oseba
```

Da lahko razberemo tip spremenljivke `student`, moramo vedeti, da konstruktor `Odrasel` pripada tipu `Oseba` in da sprejme tri argumente tipov `Int`, `Bool` in `Int`. Te informacije shranimo kot seznam urejenih parov imen tipov in njihovih konstruktorjev. Tako dobimo kontekst:

```
type context = {vars: (string * Syntax.htype) list;
     datadefs: (Syntax.cname * Syntax.datadef) list}
```

definiramo tudi začeten, prazen kontekst, ki ga potrebujemo na začetku izvajanja, ko še ne poznamo nobenih spremenljivk in pomožni funkciji za dodajanje novih spremenljivk in tipov:

```
let empty_ctx = {vars = []; datadefs = []}

let extend_var x ty ctx =
    {ctx with vars = (x, ty)::ctx.vars}

let extend_datadef x constrs ctx =
    {ctx with datadefs = (x, constrs)::ctx.datadefs}.
```

Funkcija `check` deluje zelo preprosto, primerja izračunan tip danega izraza s tistim, ki ga pričakuje in sproži napako, če se ne ujemata. Funkcija `type_of` pa s pomočjo informacij v kontekstu preoblikuje podan izraz v izrazno drevo in rekurzivno izračuna tipe podizrazov. Hkrati preverja pravilnost tipov s pomočjo funkcije `check`. Na primer, v operaciji seštevanja dovolimo le dve celi števili, torej izračunamo tipa obeh podizrazov in če nista celi števili, sprožimo napako.

Poglejmo, kako sem dopolnil `type_of`, da pravilno preverja tipe novih konstruktorjev in izrazov `case`:

```
and type_of (ctx:context) = function
  ...
```

```
      | Syntax.Constr c → type_of_constr c ctx.datadefs
      | Syntax.Case (e, cases) →
        let t = type_of ctx e in
        (match t with
          | Syntax.TData u →
          let u_def = find_u u ctx.datadefs in
          let ret_type = cases_type u_def cases ctx in
          ret_type
          | _ → type_error "%s cannot occur in a case
                expression" (Syntax.string_of_type t))
```

Osredotočimo se najprej na definicije novih tipov, torej `Syntax.Constr` vejo. Tu enostavno kličemo funkcijo `type_of_constr`, s konstruktorjem, katerega tip nas zanima in delom konteksta, kjer je ta shranjen. Funkcija `type_of_constr` izgleda takole:

```
let rec type_of_constr c = function
  | [] → type_error "unknown constructor %s" c
  | (type_name, constrs)::datadefs →
    begin
        match List.assoc_opt c constrs with
          | None → type_of_constr c datadefs
          | Some arg_types → List.fold_right
            (fun arg_type t → Syntax.TArrow
                (arg_type, t)) arg_types
                    (Syntax.TData type_name)
    end
```

Funkcija se rekurzivno sprehodi skozi del konteksta, kjer so shranjeni novi tipi in njihovi konstruktorji. Za vsak tip preveri, če se med njegovimi konstruktorji nahaja želeni in če se, izračuna njegov tip.

Malo več dela je bilo s preverjanjem tipa v izrazu `case`. Najprej izračunamo tip izraza, ki ga primerjamo z vejami. Ker je `case` smiselen le za novo definirane tipe, ker ne bi ničesar pridobili, če bi vstavili na primer celo število, v nasprotnem primeru sprožimo napako. Sedaj poznamo vhodni tip. Nato poiščemo konstruktorje tega tipa, da jih bomo lahko primerjali z vzorci v vejah. To storimo rekurzivno, na zelo podoben način, kot smo to storili v funkciji `type_of_constr`. Preostane nam le še izračunati tip, ki ga bomo vrnili. Pri tem nam pomaga pomožna funkcija `cases_type`:

```
and cases_type u_def cases ctx =
match cases with
    | [] → type_error "empty case expression"
```

```
| ((cname, xs), action)::cases' →
let xs_types = find_u cname u_def in
let ctx = extend_ctx xs xs_types ctx in
let t1 = type_of ctx action in
let rec rest_of_cases cases =
  match cases with
  | [] → t1
  | ((cname, xs), action)::cases' →
  let xs_types = find_u cname u_def in
  let ctx' = extend_ctx xs xs_types ctx in
  let t2 = type_of ctx' action in
  if t1 ⟨⟩ t2 then
      type_error "case expressions have
        different types"
  else
      rest_of_cases cases'
in rest_of_cases cases'
```

Seveda ne želimo praznega izraza `case`, zato v takem primeru sprožimo napako. Nato izračunamo tipe spremenljivk, ki predstavljajo argumente za konstruktor v dani veji. To storimo z uporabo funkcije `find_u`, ki poišče konstruktor v seznamu konstruktorjev in vrne tipe njegovih argumentov. Nato to dodamo v kontekst, da jih lahko uporabimo pri izračunu tipa posledice. Opazimo, da, ko dodajamo spremenljivke in njihove tipe v kontekst, posredno preverimo, da je število spremenljivk enako številu potrebnih argumentov, torej pravilno. Ko tega izračunamo in ga poimenujemo `t1`, se lotimo preostalih vej. Te pregledamo rekurzivno, z enakim postopkom kot pri prvi veji, vendar zdaj namesto da shranimo tip posledice, ga primerjamo s `t1` in če se ne ujemata, sprožimo napako. To storimo, ker želimo, da izraz `case` vrne rezultat enakega tipa, neodvisno od izbrane veje.

3.5 Tolmač

Čas je za izračunanje vrednosti izrazov v programu. Tega se lahko brez težav lotimo, ker nam preverjevalnik tipov zagotavlja, da so ti pravilni. V MiniHaskellu izraze računamo znotraj datoteke `interpret.ml`. Ta deluje precej preprosto, ker smo vso pripravo naredili že prej. Najprej definira okolje za spremenljivke, nato kako shranjujemo izračunane vrednosti, funkcijo `interp`, ki računa vrednosti izrazov, in izpiše rezultat.

Okolje ali `environment` je seznam urejenih parov imen spremenljivk in njihovih vrednosti. Uporablja ga funkcija `interp`, ki vrednosti spremenljivk uporablja za izračun vrednosti izrazov. Izračune vrednosti hranimo v enem izmed konstruktov tipa `value`:

```
and value =
| VInt of int
| VBool of bool
| VNil of Syntax.htype
| VClosure of environment * Syntax.expr
| VConstr of Syntax.cname * (environment * Syntax.expr)
    list
```

Potrebujemo način za shranjevanje osnovnih tipov MiniHaskella, torej `Int` in `Bool`, prazen seznam `Nil` in funkcije. Za hranjenje vrednosti novih tipov sem dodal še `VConstr`, ki vsebuje ime konstruktorja in seznam urejenih parov okolja in izraza, ki predstavlja njegove argumente. To nam med drugim tudi omogoča izpisovanje tipov in njihovih konstruktorjev:

```
let rec print_result n v =
  (if n = 0 then
     print_string "..."
   else
     match v with
     ...
     | VConstr (c, args) →
      print_string c;
      if args ⟨⟩ [] then begin
        print_string " (";
        print_args n args;
        print_string ")"
      end
      ...
) ;
```

Za `case` je bilo potrebno dopolniti funkcijo `interp`, da zna poiskati pravo vejo in vrniti njeno posledico. Kot je tudi navada v splošnem, sem omogočil tudi vzorec oblike `_`, ki pomeni 'vsi preostali primeri'. Ker vemo, da preverjevalnik tipov zagotavlja pravilno število spremenljivk za konstruktorji v vejah, lahko le enostavno primerjamo ime podanega konstruktorja z vsemi v izrazu `case`. Če najdemo pravega, dodamo njegove argumente v okolje in izračunamo posledico. V nasprotnem primeru sprožimo napako.

```
let rec extend_env env xs vs =
    match xs, vs with
```

```
...
| Syntax.Case (e, cases) →
  (match interp env e with
  | VConstr (c, args) →
      let rec find_case = function
      | [] → runtime_error
          ("Unmatched constructor " ^ c)
      | ((c', xs), action) :: l →
        if c = c' then
          let env' = extend_env env xs args in
          interp env' action
        else if "_" = c' then
          let env' = extend_env env xs args in
          interp env' action
        else find_case l
      in find_case cases
  | _ → runtime_error "Constructor expected in case"
  )
```

3.6 Glavni program

Preostane le še glavna datoteka, ki poganja programe z uporabo vseh zgoraj opisanih datotek. Sklicuje se na leksični analizator in razčlenjevalnik, ter definira okolje in funkcijo `exec`, ki izvaja ukaze. Okolje je sestavljeno iz konteksta preverjevalnika tipov in okolja tolmača. Tako pozna vrednosti in tipe vseh spremenljivk. Moral sem le dodati, kaj se zgodi, ko uporabnik definira nov tip:

```
let exec (ctx, env) = function
  ...
    | Syntax.DataDef (name, constructors) →
      Zoo.print_info "type %s is defined@." name ;
      (Type_check.extend_datadef name constructors ctx,
        env)
  ...
```

Imamo programski jezik z delujočimi funkcionalnostmi definiranja novih tipov in delanja z njimi. Poglejmo si še, kako ga uporabljamo.

Poglavje 4

Uporaba

Vsak programski jezik je razvit z namenom, da v njem programiramo. Z implementacijo definicije novih podatkovnih tipov MiniHaskell programerju ponuja veliko več možnosti načina pristopa k problemu. Vendar najprej poglejmo, če lahko programski jezik nekoliko poenostavimo, brez da izgubimo funkcionalnost.

Spomnimo se, da ima jezik od prej že osnovne tipe celih števil, booleanov, urejenih parov, seznamov in funkcij. Najbolj očitno je, da lahko z definiranjem nekega novega tipa nadomestimo posebej definiran tip seznama:

```
data Seznam = Empty | Cons Int Seznam
```

Definirali smo seznam kot nov podatkovni tip, ki lahko vsebuje cela števila. V moji implementaciji je možno definirati le monomorfne sezname, torej take, ki vsebujejo le en tip, ki ga določimo v naprej, torej ob definiciji. Pogoste funkcije ze delo s seznami so `head`, ki vrne prvi element seznama, `tail`,ki vrne vse elemente razen prvega, `length` za računanje dolžine seznama, `append`, ki doda element na konec seznama in `map`, ki uporabi funkcijo na vsakem elementu seznama.

```
let head = fun s : Seznam ⇒
  case s of
      Empty → Nil
    | Cons x xs → x
  end

let tail = fun s : Seznam ⇒
  case s of
      Empty → Empty
    | Cons x xs → xs
```

```
  end

let length = rec length : Seznam → Int is
  fun s : Seznam ⇒
    case s of
        Empty → 0
      | Cons x xs → 1 + length xs
    end

let append = rec append : Seznam → Seznam → Seznam is
  fun s1 : Seznam ⇒ fun s2 : Seznam ⇒
    case s1 of
        Empty → s2
      | Cons x xs → Cons x (append xs s2)
    end

let map = rec map : (Int → Int) → Seznam → Seznam
  is
    fun f : (Int → Int) ⇒
      fun s : Seznam ⇒
        case s of
            Empty → Empty
          | Cons x xs → Cons (f x) (map f xs)
        end
```

Imamo funkcije za delo s seznami. Potrebujemo še seznam, na katerem jih bomo uporabili

```
let enke = Cons 1 (Cons 1 (Cons 1 Empty))
```

`Enke` so torej seznam sestavljen iz treh enic. Dodajmo še funkcijo, ki jo bomo kot argument podali funkciji `map`:

```
let plusEna = fun x : Int ⇒ x + 1
```

Sedaj imamo vse potrebno. Poglejmo, kako deluje naš primer.

```
MiniHaskell⟩ head enke
- : Int = 1

MiniHaskell⟩ tail enke
- : Seznam = Cons ((1) (Cons (1) (Empty)))

MiniHaskell⟩ length enke
- : Int = 3
```

```
MiniHaskell⟩ append (Cons 2 Empty) enke
- : Seznam = Cons (2) (Cons (1)
             (Cons (1) (Cons (1) (Empty))))

MiniHaskell⟩ map plus_ena enke
- : Seznam = Cons (2) (Cons (2)
             (Cons (2) (Empty)))
```

Z implementacijo novih tipov smo dobili tudi možnost delanja s seznami. Torej seznamov kot so bili definirani v MiniHaskellu, ne potrebujemo več. Vprašanje je, kaj vse še lahko nadomestimo. Poskusimo še ustvariti nov tip `Boolean`, ki bo zamenjal osnovni tip `Bool` in nekaj pripadajočih funkcij:

```
data Boolean = True | False

let not = fun b : Boolean ⇒
  case b of
      True → False
    | False → True
  end

let xor = fun b1 : Boolean ⇒ fun b2 : Boolean ⇒
  case of b1 of
      True → case b2 of
                   True → False
                      | False → True
                     end
    | False → case b2 of
                               True → True
                          | False → False
                       end
  end
```

Podobno bi lahko definirali tudi funkciji `and` in `or`, vendar tu nista ključnega pomena. Oglejmo si rezultate:

```
MiniHaskell⟩ let a = True
val a : Boolean

MiniHaskell⟩ let b = False
val b : Boolean

MiniHaskell⟩ not a
```

```
- : Boolean = False

MiniHaskell⟩ not b
- : Boolean = True

MiniHaskell⟩ xor a b
- : Boolean = True

MiniHaskell⟩ xor a a
- : Boolean = False
```

Zelo pogosta uporaba boolovih vrednosti je pogojni stavek. Tudi tega lahko zamenjamo z novimi tipi in izrazom `case`.

```
if e1 then r1
else if e2 then r2
else r3
```

Zgornjo kodo lahko zamenjamo z

```
case e1 of
    True → r1
  | False →  case e2 of
                 True → r2
               | False → r3
             end
end
```

Preostanejo nam še cela števila ali `Int`, urejeni pari in funkcije. Slednje se samo s tipi ne dajo reproducirati, lahko pa se lotimo števil. Najbolj enostaven primer in tak, ki se izogne pisanju neštetega števila konstruktorjev, so Peanova naravna števila. Delujejo tako, da definirajo število nič in naslednika števila. Tako lahko zapišemo vsa naravna števila, kot bi to naredili v eniškem sistemu. Z negativnimi števili je več težav, zato se jim ne bomo posebej posvečali. Definiramo tudi funkcijo `plus`, ki sešteje dve Peanovi števili na način, da vemo, da je vsota nič in nekemu številu to drugo število in to uporabimo tako, da rekurzivno zmanjšujemo prvi seštevanec do nič in sproti povečujemo vsoto. Na koncu samo prištejemo drugi seštevanec.

```
data Stevilo = Nic | Naslednik Stevilo

let plus = rec plus : Stevilo → Stevilo → Stevilo is
  fun x : Stevilo ⇒ fun y : Stevilo ⇒
    case x of
```

```
        Nic → y
      | Naslednik a → Naslednik (plus a y)
    end

let ena = Naslednik Nic

let dva = Naslednik (Naslednik Nic)

MiniHaskell⟩ plus ena dva
- : Stevilo = Naslednik (Naslednik (Naslednik (Nic)))
```

Opazimo, da seštevanje deluje pravilno.

Urejeni pari so tudi dokaj preprosta struktura. Potrebujemo tip in funkciji za pridobivanje prvega in drugega elementa.

```
data Par = Par Stevilo Stevilo

let prvi = fun p : Par ⇒
  case p of
    Par x y → x
  end

let drugi = fun p : Par ⇒
  case p of
    Par x y → y
  end

let par = Par (Naslednik (Naslednik Nic))
              (Naslednik Nic)

MiniHaskell⟩ prvi par
- : Stevilo = Naslednik (Naslednik (Nic))
MiniHaskell⟩ drugi par
- : Stevilo = Naslednik (Nic)
```

Tako lahko nadomestimo tudi urejene pare. Funkcij pa ne moremo, ker jih potrebujemo za delo s tipi. Vanje vstavljamo izraze `case`.

Zelo pogosto upoabljen primer rekurzivnih tipov so drevesa. Definirajmo tip `Drevo`, ki bo predstavljalo binarno drevo in funkcijo, ki bo preštela število elementov v drevesu:

```
data Drevo = Empty | List Stevilo |
    Vozlisce Stevilo Drevo Drevo
```

```
let stVozlisc = rec st_vozlisc : Drevo → Stevilo is
  fun d : Drevo ⇒
    case d of
        Empty → Nic
      | List s → Naslednik Nic
      | Vozlisce s l r →
        plus (Naslednik Nic) (plus (stVozlisc l)
             (stVozlisc r))
    end

let lipa = Vozlisce Nic (List (Naslednik Nic))
           (List (Naslednik (Naslednik Nic)))

MiniHaskell⟩ st_vozlisc lipa
- : Stevilo = Naslednik (Naslednik (Naslednik (Nic)))
```

Definirali smo drevo `lipa`, ki ima korensko vozlišče in dva lista. Funkcija `st_vozlisc` vrne število tri.

V prvem poglavju smo spoznali lačne funkcije. Spomnimo se, da so to funkcije, ki sprejmejo argument in vrnejo funkcijo, ki sprejme spet nov argument in vrne funkcijo, ... Definicija takih funkcij je zaradi iso rekurzivnega pristopa bolj zapletena, ker potrebujemo izomorfizem med tipom in njegovim odvojem. V tem primeru bomo potrebovali funkcijo `fold`. Definirajmo tip `Lacna`, ki bo predstavljal lačne funkcije. Potrebujemo konstruktor, poimenujmo ga `Fun`:

```
data Lacna = Fun (Int → Lacna)
```

Definirajmo še izomorfizem `fold`, ki nam bo omogočal, da bomo lahko slikali tipe `Int` → (`Int` → `Lacna`) v `Int` → `Lacna` ker tipa izomorfna, torej bomo lahko izhode lačnih funkcij obravnavali kot lačne funkcije.

```
let fold = fun l : Lacna ⇒
  case l of
    Fun f → f
  end
```

Definirajmo še primer lačne funkcije.

```
let lacna = rec lacna : Int → Lacna is
  (fun x : Int ⇒ Fun lacna)

MiniHaskell⟩ lacna 42
- : Lacna = Fun (⟨fun⟩)
```

```
MiniHaskell⟩ fold (lacna 42)
- : Int → Lacna = ⟨fun⟩
```

Ko lačni funkciji podamo argument in rezultat preslikamo v njegov izomorfizem, dobimo spet lačno funkcijo. Poskusimo ustvariti daljše zaporedje:

```
MiniHaskell⟩ fold (fold (fold (fold ((fold (lacna 1
             )) 2) 3) 4) 5)
- : Int → Lacna = ⟨fun⟩
```

Še malo bolj zapletene lačne funkcije so tokovi, ki za vsak prejeto `Unit` vrednost, vrnejo naslednji element toka. Ker vračajo urejeni par elementa in toka, potrebujemo še dodatni funkciji, ki vračata posamezne elemente para.

```
data Unit = T

data Par = Par Int Tok

data Tok = Fun (Unit → Par)

let fold = fun t : Tok ⇒
  case t of
    Fun f → f
  end

let prvi = fun p : Par ⇒
  case p of
    Par e l → e
  end

let drugi = fun p : Par ⇒
  case p of
    Par e l → l
  end
```

Konstantni tokovi, ki vedno vračajo isto vrednost, niso zanimivi. Poglejmo primer toka naravnih števil:

```
let naslednik = fun n : Int ⇒ n + 1

let naravna = rec naravna : (Int → Int) →
Int → Unit → Par is
  fun f : (Int → Int) ⇒
    fun n : Int ⇒ fun u : Unit ⇒
      Par n (Fun (naravna f (f n)))
```

```
MiniHaskell⟩ prvi ((fold (drugi ((fold (drugi
             ((naravna naslednik 0) T))) T))) T)
- : Int = 2
```

Ker lahko na tokove gledamo tudi kot na neskončne seznаme, jih lahko tudi definiramo podobno kot smo sezname zgoraj. Seveda tudi tukaj potrebujemo funkcijo na začetnem elementu.

```
data Tok = Cons Int Tok

let naslednik = rec funkcija : Tok → Tok is
  fun t : Tok ⇒
    case t of
      Cons e tok → Cons (e+1) (naslednik tok)
    end

MiniHaskell⟩ let naravna = rec naravna : Tok is
             Cons 0 (naslednik naravna)
val naravna : Tok

MiniHaskell⟩ naravna
- : Tok = Cons (0 (Cons (1 (Cons (2 (Cons (3 (Cons (4
(Cons (5 (Cons (6 (Cons (7 (Cons (8 (Cons (9 (Cons (10
(Cons (11 (Cons (12 (Cons (13 (Cons (14 (Cons (15 (Cons
(16 (Cons (17 (Cons (18 (Cons (19 (Cons (20 (Cons (21
(Cons (22 (Cons (23 (Cons (24 (Cons (25 (Cons (26 (Cons
(27 (Cons (28 (Cons (29 (Cons (30 (Cons (31 (Cons (32
(Cons (33 (Cons (34 (Cons (35 (Cons (36 (Cons (37 (Cons
(38 (Cons (39 (Cons (40 (Cons (41 (Cons (42 (Cons (43
(Cons (44 (Cons (45 (Cons (46 (Cons (47 (Cons (48 (Cons
(49 (Cons (50 (Cons (51 (Cons (52 (Cons (53 (Cons (54
(Cons (55 (Cons (56 (Cons (57 (Cons (58 (Cons (59 (Cons
(60 (Cons (61 (Cons (62 (Cons (63 (Cons (64 (Cons (65
(Cons (66 (Cons (67 (Cons (68 (Cons (69 (Cons (70 (Cons
(71 (Cons (72 (Cons (73 (Cons (74 (Cons (75 (Cons (76
(Cons (77 (Cons (78 (Cons (79 (Cons (80 (Cons (81 (Cons
(82 (Cons (83 (Cons (84 (Cons (85 (Cons (86 (Cons (87
(Cons (88 (Cons (89 (Cons (90 (Cons (91 (Cons (92 (Cons
(93 (Cons (94 (Cons (95 (Cons (96 (Cons (97 (Cons (98
(Cons (... (...))))))))))))))))))))))))))))))))))))))))
)))))))))))))))))))))))))))))))))))))))))))))))))))))))
)))))))))))))))))))))))))))))))))))))))))))))))))))))))
```

```
))))))))))))))))))))))))))))))))))))))))))))))))))
```

Dobimo neskončen seznam naravnih števil. Program seveda ne izpiše vseh, da lahko nadaljujemo z vpisovanjem ukazev. Taka definicija nam poda neskončen seznam, medtem ko definicija z uporabo lačne funkcije vsakič vrne le en par. Seveda lahko tudi iz neskončnega seznama izluščimo končno mnogo elementov, če definiramo funkciji `glava`, ki vrne prvi element seznama in `rep`, ki vrne vse neprve elemente.

```
let glava = fun t : Tok ⇒
  case t of
    Cons e tok → e
  end

let rep = fun t : Tok ⇒
  case t of
    Cons e tok → tok
  end
```

V lenih programskih jezikih delujeta obe definiciji, medtem ko bi neučakani jeziki po drugi definiciji hoteli najprej izračunati vse vrednosti do konca, kar seveda ni mogoče.

Če definicijo tokov malenkost spremenimo, lahko definiramo tudi procese. Ti namesto `Unit` vrednosti sprejemejo poljubne elemente, ki vplivajo na izhod funkcije. Ohranimo lahko funkciji `prvi` in `drugi` iz definicije tokov. Poglejmo si primer, ki vrne vsoto vseh dosedanjih vhodov:

```
data Par = Par Int Proces

data Proces = Fun (Int → Par)

let fold = fun t : Proces ⇒
  case t of
    Fun f → f
  end

let sestej = fun e : Int ⇒
  fun n : Int ⇒
    e + n

let proces = rec proces : (Int → Int → Int) → Int
    → Int → Par is
  fun f : (Int → Int → Int) ⇒
```

```
      fun n : Int ⇒
        fun e : Int ⇒
          Par (f e n) (Fun (proces f (f e n)))

MiniHaskell⟩ prvi ((fold (drugi (proces sestej 1 2
    ))) 3)
- : Int = 6
```

Dobili smo šest, kar je vsota enke, dvojice in trojke.

Poglavje 5

Zaključek

Področje rekurzivnih tipov sega še veliko globje, kot prikazujejo zgornji primeri in nam tako daje veliko možnosti za nadgradnjo implementacije. Opazna možnost je implementacija polimorfizma. Tipi so sedaj lahko le monomorfni, torej ne moremo definirati seznama, ki bi lahko dopuščal elemente različnih tipov, vendar je tip seznama potrebno določiti ob definiciji tipa. Polimorfizem na seznamih bi izgledal takole:

```
data Seznam a = Empty | Cons a (Seznam a)
let seznamStevil = Cons 1 (Cons 2 (Cons 3 Empty))
```

spremenljivka `a` je tukaj lahko poljubnega tipa.

Prav tako znotraj izrazov `case` ne moremo uporabljati kompleksnih vzorcev. Recimo, ne moremo definirati izraza

```
case e of
     Empty → ...
   | Cons x l → ...
   | Cons x (Cons y l) → ...
end
```

Sicer bi zgornji izraz lahko zapisali kot gnezdeni `case`

```
case e of
     Empty → ...
   | Cons x l →
      case l of
           Empty → ...
         | Cons y l → ...
     end
end
```

vednar bi bilo zanimivo uporabljati prvi primer, vsekakor pa bi bila koda tako lepša in bolj pregledna.

Še ena razširitev bi bila omogočanje medsebojno rekurzivnih tipov, na primer

```
data A = B
data B = A
```

s katero bi lahko našli veliko zanimivih primerov malo bolj zapletenih rekurzivnih tipov.

Literatura

[1] Andrej Bauer. Programming Languages Zoo. `https://plzoo.andrej.com`.

[2] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts, USA, 2002.

[3] Jure Slak. Induktivni in koinduktivni podatkovni tipi. Diplomsko delo, Univerza v Ljubljani – Fakulteta za matematiko in fiziko, 2015.