Preverjanje pravilnosti podatkovnih struktur z odvisnimi tipi v programskem jeziku Idris

Blaž Repas, Jurij Mihelič

Fakulteta za računalništvo in informatiko, Univerza v Ljubljani
E-pošta: blaz@bss.si

Checking correctness of data structures with dependent types in the Idris programming lanugage

Programming languages and software engineering represent one of the key areas of computing and also have a great influence on other fields. Languages that use sophisticated type systems and type checking are usually considered more safe in the sense that their programs are less prone to runtime errors. Classical type systems offer types that may only depend on other types whereas their extension, called dependent types, may also depend on values. Such types enable programmers to take the correctness of the program to a higher level. In the paper we focus on the programming language Idris which supports full dependent types. First we describe standard data types such as natural numbers, a linked list, and a stack. Then we give implementations of several dependent data structures such as vector, i.e., a list of specific length, and a stack. We focus on stack and provide proven implementation of its operations. Both of our implementations include proofs of correctness within their described meaning.

1 Uvod

Programiranje in programski jeziki so eno izmed temeljnih področij računalništva. Pomembnost področja izhaja predvsem iz tega, da s programi v različnih programskih jezikih lahko opisujemo postopke, algoritme, podatkovne strukture, uporabniške vmesnike, besedilne in večpredstavitvene dokumente itd. Takšni opisi pa so uporabni ne samo v računalništvu, temveč tudi v drugih vedah.

Poznamo ogromno različnih vrst programskih jezikov, v tem članku pa se osredotočamo predvsem na *tipizirane* jezike, tj. jezike, v katerih ima vsaka vrednost natančno določen *tip*. Tip nosi informacijo o pomenu in obnašanju programa preko možnih vrednosti, ki jih lahko spremenljivka nekega tipa zavzame. Netipizirane jezike pravzaprav lahko obravnavamo tudi kot tipizirane [1], le da podpirajo samo en *univerzalen tip*, kateremu pripadajo vse vrednosti. Primer takšnega jezika je zbirnik. V praksi pa navadno rečemo, da je jezik tipiziran, v kolikor podpira večje število tipov (lahko tudi neskončno) oz. sistem tipov. Primera takšnih jezikov sta npr. Java in Haskell.

Tipi in sistemi tipov se razvijajo iz pomembnega, čisto pragmatičnega razloga: gre za iskanje mehanizma, ki bi programerju kar se da onemogočil implementacijo programov, ki bi med svojim izvajanjem lahko naleteli na napake. Takšen mehanizem torej zmanjšuje število programskih hroščev, posledično pa so programi bolj varni. Varnost programa je seveda izredno zaželena lastnost, še posebej pa je ključna v kritičnih aplikacijah, katerih nepravilno delovanje lahko povzroči materialno ali finančno škodo ali celo ogrozi človekovo življenje. Področja, ki zahtevajo takšne aplikacije, so številna, npr. bančništvo, nadzor elektrarn, komunikacije, vesoljski programi in podobno.

Preverjanje skladnosti tipov v programu lahko poteka tekom izvajanja programa, čemur pravimo *dinamično preverjanje*, ali pa tekom prevajanja programa, čemur pravimo *statično preverjanje*. Dinamično preverjanje marsikatero napako ulovi šele tekom izvajanja programa ali pa sproži napako zaradi neujemanja tipov. Takšnim napakam se želimo v življenjski dobi programa izogniti čimprej, zato je v nadaljevanju članka za nas zanimivo predvsem statično preverjanje tipov. Seveda pa to, da je nek jezik statičen, še ne pomeni, da je varen. Na primer, statičen programski jezik C omogoča uporabo in manipulacijo kazalcev, s čimer lahko programer neposredno spreminja pomnilnik, ki ga zaseda neka druga podatkovna struktura. Posledično tudi skladnost tipov ne more zagotoviti predvidljivega delovanja takšne podatkovne strukture.

V naslednjem razdelku na kratko predstavimo področje odvisnih tipov in z njimi povezanih problemov in rešitev, pri čemer se osredotočimo na programski jezik Idris, ki takšne tipe podpira. V tretjem razdelku opišemo osnovne principe delovanja odvisnih tipov. Najprej prikažemo izvedbo „neodvisnih" naravnih števil in povezanega seznama, nato pa še dveh odvisnih tipov vektorja in sklada. V zadnjem razdelku zaključimo in podamo možnosti za nadaljnje delo.

2 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) 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 [3], 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 (oz. je odvisen od) vrednosti, pravimo, da je *indeksiran* s to vrednostjo.

Kot primer si oglejmo vrsto (angl. *kind*) tipa *seznam* (implementiran in razložen v nadaljevanju), ki je

$$List : Type \to Type,$$

kar pomeni, da je tip seznama parametriziran z nekim tipom (*parametrični polimorfizem*). Seznamu bi lahko v tip dodali tudi dolžino. Takrat bi bila vrsta tipa 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* oz. je indeksiran z naravnim številom.

2.1 Odvisne funkcije

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

Primer implementacije odvisne funkcije:

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

Funkcija $replicate$, torej 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).

2.2 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. Primer odvisnega tipa iz priročnika za programski jezik Idris [4]:

```
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.

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)
```

2.3 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 [5]. 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 pravilne.

3 Programi z odvisnimi tipi

3.1 Naravna števila

Najprej si oglejmo, kako so v programskem 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 s podatkovnim konstruktorjem $S\ n$.

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

3.2 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 $Type \rightarrow Type$, drugače povedano, $List$ je konstruktor tipa, ki kot parameter sprejme 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 izjema.

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. Natančno takšne napake pa želimo z uporabo odvisnih tipov preprečiti.

3.3 Vektor - seznam z dolžino, shranjeno v tipu

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 nek konkreten tip in neko naravno število, ki predstavlja informacijo o dolžini vektorja.

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 podanim 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 lahko implementiramo tudi funkcijo *tail*.

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

Funkcija tako lahko 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.

3.4 Sklad

Ena izmed bolj uporabljanih podatkovnih struktur je sklad. V nedokazani obliki je identičen seznamu. Če pa uporabimo različico, ki ima velikost sklada shranjeno v tipu, pa je na las podoben vektorju.

```
data SimpleStack : Type -> Type where
  SiStNil : SimpleStack a
  SiStCons : (x : a) -> SimpleStack a ->
               SimpleStack 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 ekvivalentna seznamu. Namenjene so boljšemu razumevanju delovanja seznama, saj ob dodajanju dokazov postane koda manj berljiva.

```
push : (x : a) -> SimpleStack a ->
         SimpleStack a
push x xs = SiStCons x xs

pop : SimpleStack a -> (a, SimpleStack a)
pop (SiStCons x xs) = (x, xs)

peek : SimpleStack a -> a
peek (SiStCons x xs) = x

drop : SimpleStack a -> SimpleStack a
drop (SiStCons x xs) = xs
```

Podobno kot pri seznamu, funkcije *pop*, *peek*, *drop* niso definirane za vse možne sklade, 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 lahko izvajajo 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 sklada,
- *drop* zavrže vrh sklada in vrne preostanek sklada (sklad, iz katerega je bil podani sklad zgrajen) in
- *pop* vrne par: element na vrhu sklada in preostanek sklada.

3.5 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 sprejme element *x* tipa *a* in obstoječi sklad tipa *Stack xs* in iz njiju zgradi nov sklad tipa *Stack (x :: xs)*, kar 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 vrednost (v večini primerov) ni potrebno podati, ker jo prevajalnik določi iz konteksta, potrebujemo pa jo, da ji lahko dodelimo ime in tip in s tem omogočimo njihovo uporabo v drugih tipih.

Z novo definicijo tipa sklada napišemo funkcije, ki bodo v svojih tipih izražale lastnosti, ki jih zanje pričakujemo.

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

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

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. 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 funkcij *peek* in *pop*.

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

Funkcija *peek* vrača element iz vrha sklada, kar 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, t. j. enakost, ki jo zna unifikator (ključen del preverjanja skladnosti tipov) brez pomoči dokazati. V našem primeru je to enostavno, saj smo vzeli 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)
```

4 Zaključek

V članku smo predstavili področje odvisnih tipov. Dva klasična primera sta odvisne funkcije in odvisni pari. Osredotočili smo se na programski jezik Idris, ki podpira polne odvisne tipe in v njem implementirali nekaj osnovnih podatkovnih struktur. Prikazali smo (nedokazano) izvedbo naravnih števil, povezanega seznama in sklada, ki ne podpirajo odvisnosti. Nato smo predstavili odvisni tip vektor, ki ima dolžino shranjeno v tipu, pa tudi avtomatski dokaz pravilnosti operacij $head$ in $tail$, poleg tega pa še odvisni tip sklad z avtomatskimi dokazi pravilnosti standardnih operacij nad njim. Naše nadaljnje delo se osredotoča na izvedbo tipa *urejen seznam* in nekaterih algoritmov za urejanje, pri čemer bi že sama izvedba algoritma avtomatsko dokazovala njegovo pravilnost.

Literatura

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

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

[3] Per Martin-Löf. Intuitionistic type theory. Ed. Giovanni Sambin. Vol. 17. Naples: Bibliopolis, 1984.

[4] Edwin Brady. Programming in Idris: a tutorial. Technical report, University of St Andrews, 2013.

[5] John E. Hopcroft and Rajeev Motwani and Jeffrey D. Ullman. Introduction to Automata Theory, Languages, and Computation, 2. izdaja. Addison-Wesley, 2001.

[6] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of Functional Programming 23(5): 552-593, 2013.