UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO

Matematika – 1. stopnja

Filip Koprivec

Uporaba tipov za zagotavljanje pravilnosti programov

Delo diplomskega seminarja

Mentor: doc. dr. Matija Pretnar

Ljubljana, 2017

Kazalo

Uporaba tipov za zagotavljanje pravilnosti programov

Povzetek

Eno izmed najpomembnejših orodij v programiranju so tipi, ki nam omogočajo, da že pred zagonom programa zagotovimo, da se v programu ne bo pojavila določena vrsta napak. V diplomskem delu na primeru preprostega jezika najprej predstavimo podajanje programskih jezikov in njihove operacijske semantike ter prikažemo uporabo sistema tipov. S pomočjo sistema tipov dokažemo izrek o varnosti, ki zagotovi, da tekom izvajanja ne prišlo do napake v izvajanju. Jezik s pomočjo funkcij nadgradimo v lambda račun z enostavnimi tipi in tudi zanj pokažemo izrek o varnosti. S pomočjo konstruktorja, ki podaja negibno točko funkcije v lambda račun, dodamo rekurzijo in na primeru pokažemo izvajanje izraza, ki preveri, ali je število sodo.

Using types to provide safety in programming languages

Abstract

One of the most important tools in programming is definitely the type system, which allows us to eliminate a whole class of errors before the actual evaluation of a program. The work presents syntax and evaluation rules while presenting simple programming language and its type system. Safety theorem, which ensures, that well-typed term will not get stuck is presented and proved. A simple programming language is extended into simply typed lambda calculus and corresponding safety theorem is proved. Using fixed point construct, simply typed lambda calculus is extended with recursion. Use of recursion is presented on example function that tests if a non-negative integer is even.

Math. Subj. Class. (2010): 68Q55, 68Q60, 68N18

Ključne besede: Programski jezik, sistem tipov, lambda račun, lambda račun z enostavnimi tipi

Keywords: Programming language, type system, lambda calculus, simply typed lambda calculus

1. Uvod

Najpomembnejša lastnost večine programov je poleg hitrost, majhne porabe pomnilnika, preprostosti za vzdrževanje, zagotovo pravilnost. Da bi zagotovili pravilnost programa, si pomagamo z mnogo različnimi pristopi in orodji (navodila za način pisanja kode, izogibanje slabim delom jezika, formalno dokazovanje pravilnosti), ki imajo različne prednosti in slabosti. Eno izmed preprostejših, a vseeno zelo močnih orodij, so *tipi*. S pomočjo tipov lahko vnaprej preprečimo nekatera stanja, ki bi lahko privedla do napake v izvajanju. Teoretično to utemeljimo z izrekom o varnosti, ki zagotovi, da med izvajanjem programa s tipi ne bo prišlo do napake.

Poleg možnosti za odpravo napak so tipi dobri še na mnogo drugih področjih. Tipi so lahko močno orodje za lažje razumevanje in boljše dokumentiranje kode, sploh ob večji količini kode in delu več različnih ljudi, saj omogočajo preprosto in nevsiljivo vrstično (inline) dokumentacijo in razlago.

Hkrati pa so tipi uspešno orodje za abstrakcijo že najbolj osnovnih pojmov (števila, seznami), mnogi funkcijski jeziki pa s pomočjo algebrajskih tipov uspešno dokazujejo, da lahko tipe uporabimo še za veliko močnejše abstrakcije.

Tudi industrija se v zadnjih časih vse bolj usmerja k strogo tipiziranim jezikom, saj so prednosti statično tipiziranih jezikov prevladale nad slabostmi. Kot primer se lahko navede Googlova odločitev da bo Angular2 uporabljal strogo tipiziran jezik (typescript)[5]. Vse več dinamično tipiziranih programskih jezikov pa omogoča opcijsko označevanje tipov, ki omogoča večino naštetih prednosti tipov, a hkrati ohranja prednosti dinamičnih tipov (mypy projekt za jezik Python, JSDoc Tags za javascript)[4],[6] .

Diplomska naloga je sestavljena iz treh glavnih delov. V poglavju 2 predstavimo način, s katerim podajamo programske jezike in predstavimo operacijsko semantiko preprostega programskega jezika. Ta programski jezik skozi celotno nalogo posodabljamo in dopolnjujemo (najprej s tipi, nato s funkcijami, da dobimo lambda račun, na koncu pa mu dodamo še rekurzijo). V razdelku 2.3 za programski jezik podamo pravila za izpeljavo tipov in s pomočjo tipov v poglavju 3 z izrekom o varnosti formaliziramo varnost v jeziku.

V drugem delu s pomočjo funkcij jezik nadgradimo v lambda račun z enostavnimi tipi, podamo ustrezen izrek o varnosti in ga dokažemo. V poglavju 4 tako predstavimo operacijsko semantiko lambda računa, v poglavju 4.2 lambda računu dodamo pravila za tipe in dobimo jezik, sicer znan kot lambda račun z enostavnimi tipi. V poglavju 5 pa predstavimo in dokažemo izrek o varnosti za lambda račun z enostavnimi tipi.

V tretjem delu lambda računu z enostavnimi tipi dodati fiksno točko. V poglavju 6 v lambda račun dodamo konstrukcijo negibne točke (**fix** kombinator), ki omogoči bolj kompleksne programe. Negibno točko (**fix**) najprej predstavimo in pojasnimo njegovo izpeljavo. S pomočjo negibne točke lahko v tipiziranem lambda računu zapišemo primitivno rekurzijo. V nalogi podamo nekaj primerov uporabe te rekurzije in na koncu na primeru pokažemo zapis funkcije, ki preveri, ali je število sodo v lambda računu z enostavnimi tipi.

2. Operacijska semantika

V tem poglavju predstavimo preprost imperativni programski jezik. Pri tem večinoma sledimo [3], le da namesto naravnih števil uporabljamo cela števila.

Najprej podamo in razložimo sintakso jezika in pokažemo nekaj primerov uporabe. Kasneje jezik dopolnimo s tipi in podamo pravila za izpeljavo le-teh, ter na nekaj primerih pokažemo uporabo in izpeljavo. Jezik, ki ga predstavimo na začetku, je sicer zelo šibak, saj vsebuje le nekaj preprostih konstruktov. Kljub temu na koncu poglavja sestavimo izraz, ki preveri, ali je podan izraz idempotent in zanj podamo izpeljavo tipov.

2.1. **Sintaksa.** Običajno jezike podamo z različico Backus-Naurove oblike (BNF)[7] zapisa. Jezik oziroma množico izrazov, ki jih označujemo s sintaktično spremenljivko t, definiramo tako, da naštejemo vse možne oblike izrazov, ki jih med seboj ločimo z $|$. Če v zapisu nastopa t (ali t_1, t_2, t_3), to pomeni, da namesto t lahko vstavimo poljuben veljaven izraz, kar nam omogoči, da nove izraze dobimo s kombiniranjem že obstoječih. Sintaksa jezika je prikazana v tabeli 1.

$$\begin{aligned}\text{Izraz } t \ ::= \ & \mathbf{true} \mid \mathbf{false} \mid \underline{n} \mid \mathbf{succ}\ t \mid \mathbf{pred}\ t \mid \mathbf{iszero}\ t \\ & \mid \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3\end{aligned}$$

Tabela 1. Osnovna sintaksa

Po pravilih sintakse naš jezik vsebuje: logični konstanti za resnico in neresnico ter konstanto za vsako celo število n. S črto pod številom označimo, da gre za izraz, ki to število predstavlja ($\underline{0}$ je izraz, ki predstavlja število 0). Jezik vsebuje izraza, ki predstavljata naslednika in predhodnika poljubnega drugega izraza t, in izraz, ki predstavlja predikat, ki pove, ali je izraz t enak izrazu, ki predstavlja število 0. Vsebuje pa tudi vejitveni izraz, ki predstavlja pogojni stavek. Ta je sestavljen iz treh delov: pogoja, rezultata, če je pogoj izpolnjen, in rezultata, če pogoj ni izpolnjen.

Opomba 2.1. V formalni definiciji sintakse nismo povedali ničesar o oklepajih, saj smo definirali zgolj abstraktno sintakso. V tem programskem jeziku (in vedno naprej) jih bomo uporabljali zgolj kot pripomoček, da ne pride do nejasnosti v razumevanju izrazov. Nastopali bodo popolnoma enako kot oklepaji v običajni matematiki, kjer natančneje določijo vrstni red računanja izrazov.

Sestavljanje izrazov si najlažje pogledamo na nekaj preprostih primerih. Sintaksa jezika sama po sebi nima še nikakršnega pomena, ta pomen bomo dodali s pravili izvajanja v razdelku 2.2, za pomoč pri predstavitvi jezika pa si mislimo, da se izrazi intuitivno že lahko izvedejo.

Primer 2.2. V predikat **succ** t namesto t vstavimo kar konstanto $\underline{0}$ in dobimo izraz:

$$\mathbf{succ}\ \underline{0}$$

Novi izraz, ki ga dobimo sedaj, uporabimo v predikatu **iszero** t in sestavimo izraz, ki preveri, ali je izraz ki predstavlja naslednika celega števila 0 enak konstanti 0. S pomočjo oklepajev zgolj natančneje določimo vrstni red izvajanja.

$$\mathbf{iszero}\ (\mathbf{succ}\ \underline{0}) \tag{1}$$

♢

Primer 2.3. Čeprav je definiran jezik s stališča teorije programskih jezikov zelo šibak, lahko vseeno sestavimo izraz, ki preveri ali je dan izraz t idempotent (enak 1 ali 0).

(2) $$\textbf{if } (\textbf{iszero } t) \textbf{ then true else } (\textbf{iszero } (\textbf{pred } t))$$

V zgornjem izrazu definiramo preprost pogojni stavek, ki bo v primeru, ko je pogoj izpolnjen (če izraz t predstavlja število 0), vrnil logično konstanto za resnico, če pogoj ne bo izpolnjen, pa vrednost funkcije **iszero** (**pred** t), torej logično konstanto, ki pove, ali predhodnik izraza t predstavlja število 0 ali ne. ♢

Skozi celotno diplomsko delo si bomo prav na primeru izraza (2) ogledali zmožnosti in značilnosti tega programskega jezika ter vseh sledečih izboljšav.

$$\frac{}{\textbf{iszero } \underline{0} \rightsquigarrow \textbf{true}}\ \text{E-IsZeroZero} \qquad \frac{(n \neq 0)}{\textbf{iszero } \underline{n} \rightsquigarrow \textbf{false}}\ \text{E-IsZeroNonZero} \qquad \frac{t \rightsquigarrow t'}{\textbf{iszero } t \rightsquigarrow \textbf{iszero } t'}\ \text{E-IsZero}$$

$$\frac{}{\textbf{pred } \underline{n} \rightsquigarrow \underline{n-1}}\ \text{E-PredNum} \qquad \frac{t \rightsquigarrow t'}{\textbf{pred } t \rightsquigarrow \textbf{pred } t'}\ \text{E-Pred} \qquad \frac{}{\textbf{succ } \underline{n} \rightsquigarrow \underline{n+1}}\ \text{E-SuccNum}$$

$$\frac{t \rightsquigarrow t'}{\textbf{succ } t \rightsquigarrow \textbf{succ } t'}\ \text{E-Succ} \qquad \frac{}{\textbf{if true then } t_2 \textbf{ else } t_3 \rightsquigarrow t_2}\ \text{E-IfTrue}$$

$$\frac{}{\textbf{if false then } t_2 \textbf{ else } t_3 \rightsquigarrow t_3}\ \text{E-IfFalse} \qquad \frac{t_1 \rightsquigarrow t_1'}{\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 \rightsquigarrow \textbf{if } t_1' \textbf{ then } t_2 \textbf{ else } t_3}\ \text{E-If}$$

Tabela 2. Pravila za izvajanje

2.2. **Izvajanje.** Poleg sintakse je potrebno podati tudi pravila za izvajanje jezika. To podamo z relacijo $t \rightsquigarrow t'$, ki pomeni da se izraz t pretvori v t' in je natančneje definirana v 2.5 Ta so podana v tabeli 2. Predstavljena pravila pa potrebujejo krajšo razlago. Najprej si oglejmo prvo, zelo preprosto pravilo:

$$\frac{}{\textbf{iszero } \underline{0} \rightsquigarrow \textbf{true}}\ \text{E-IsZeroZero}$$

Pravila za izvajanje so sestavljena iz dveh delov: iz dela nad vodoravno črto (predpostavk) in dela pod črto (zaključka), ki pove, v kaj se izraz pretvori. V našem primeru imamo preprosto pravilo, ki pravi, da lahko vedno (brez dodatnih predpostavk) zaključimo, da se izraz oblike **iszero** $\underline{0}$ pretvori v **true**. Ime pravila (E-IsZeroZero) služi lažjemu sklicevanju na pravilo.

Sestavljena pravila, kot na primer E-Succ, razumemo tako: »Če se izraz t pretvori v t', potem lahko zaključimo, da se izraz oblike **succ** t pretvori v **succ** t'«. Pravilo E-IsZeroNonZero tako pravi, da lahko za vsak izraz **iszero** $\underline{n}$, kjer je n od 0 različno število zaključimo, da se prvotni izraz pretvori v **false**.

Izvajanje formaliziramo z dvočleno relacijo med izrazi, ki pove, v kaj se pretvori posamezen izraz.

Definicija 2.4. Relacija $\mathcal{R} \subseteq \text{Izraz} \times \text{Izraz}$ je zaprta za pravila (ustreza pravilom) izvajanja, če za vsako obliko pravila za izvajanje velja, da je izpolnjen zaključek pravila, ali pa predpostavke niso. Za naš jezik bolj natančno:

- $\textbf{iszero}\ \underline{0}\ \mathcal{R}\ \textbf{true}$
- $\forall n \in \mathbb{Z} \setminus \{0\}.\ \textbf{iszero}\ \underline{n}\ \mathcal{R}\ \textbf{false}$
- $\vdots$
- $\forall t, t'.\ t\ \mathcal{R}\ t' \implies \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3\ \mathcal{R}\ \textbf{if}\ t_1'\ \textbf{then}\ t_2\ \textbf{else}\ t_3$

Definicija 2.5 (Semantika malih korakov)**.** *Semantika malih korakov* je najmanjša dvomestna relacija med izrazi, ki ustreza pravilom za izvajanje. Pišemo $t \rightsquigarrow t'$ in beremo t se pretvori v t'.

Najprej si na preprostem izrazu **iszero** (**succ** $\underline{0}$) poglejmo, kako deluje izvajanje. Dokaz da za izraz **iszero** $\underline{0}$ velja **iszero** $\underline{0} \rightsquigarrow$ **true** izpeljemo z drevesom izvajanja. Pri tem izraz glede na pravila izvajanja razdelimo na več manjših koščkov, dokler ne pridemo do pravil, katerih predpostavkam je zadoščeno, struktura ime dobi po dejstvu, da je teh predpostavk lahko več. Spodaj vidimo drevo, ki ustreza izvajanju izraza (1) iz primera 2.2.

$$\text{E-IsZero}\ \dfrac{\text{E-SuccNum}\ \dfrac{}{\textbf{succ}\ \underline{0} \rightsquigarrow \underline{1}}}{\textbf{iszero}\ (\textbf{succ}\ \underline{0}) \rightsquigarrow \textbf{iszero}\ \underline{1}}$$

Vidimo, da en korak v izvajanju izraza ni nujno dovolj, da se izraz izvede do konca (od tod ime semantika *malih* korakov). En korak v izvajanju prvotnega izraza tako izraz (1) pretvori v nov izraz

$$\textbf{iszero}\ \underline{1}$$

V izvajanju, ki mu ustreza drevo izvajanja spodaj, se izraz **iszero** $\underline{1}$ z uporabo pravila E-IsZeroNonZero pretovori v konstanto **false**, in konča z izvajanjem.

$$\text{E-IsZeroNonZero}\ \dfrac{}{\textbf{iszero}\ \underline{1} \rightsquigarrow \textbf{false}}$$

Primer 2.6. Poglejmo še izraz za preverjanje idempotentnosti (2). V izraz vstavimo $t = \underline{0}$ in spremljamo potek izvajanja. Za izraz najprej uporabimo pravilo E-If, za katerega moramo izvesti izraz **iszero** $\underline{0}$, zanj pa velja pravilo E-IsZeroZero. Začetni izraz se po pravilu za izvajanje E-If pretvori v

$$\textbf{if true then true else}\ (\textbf{iszero}\ (\textbf{pred}\ \underline{0}))$$

Tej izpeljavi ustreza drevo spodaj.

$$\text{E-If}\ \dfrac{\text{E-IsZeroZero}\ \dfrac{}{\textbf{iszero}\ \underline{0} \rightsquigarrow \textbf{true}}}{\begin{array}{c}\textbf{if}\ (\textbf{iszero}\ \underline{0})\ \textbf{then true else}\ (\textbf{iszero}\ (\textbf{pred}\ \underline{0})) \rightsquigarrow \\ \rightsquigarrow \textbf{if true then true else}\ (\textbf{iszero}\ (\textbf{pred}\ \underline{0}))\end{array}}$$

Če bi zanj izvedli še en korak, bi se po pravilu E-IfTrue izvedel v vrednost **true**.

$$\text{E-If-True}\ \dfrac{}{\textbf{if true then true else}\ (\textbf{iszero}\ (\textbf{pred}\ \underline{0})) \rightsquigarrow \textbf{true}}$$

$\diamondsuit$

Izvajanje obeh izrazov iz primerov smo končali, ko za izvedbo dobljenega izraza ni bilo več ustreznih pravil za izvajanje.

Definicija 2.7 (Normalna oblika)**.** Izrazom, ki nimajo ustreznega pravila za izvajanje, rečemo, da so v *normalni obliki*.

Kot rezultate izvajanja obeh izrazov iz primera smo dobili Booleovi konstanti. Poleg Booleovih konstant so očitno v normalni obliki tudi konstante za cela števila. Izrazi te oblike so zaželjeni rezultati izvajanja.

Definicija 2.8 (Vrednost)**.** Konstantam **true**, **false** in $\underline{n}$ rečemo *vrednosti* in jih označimo s sintaktično spremenljivko v.

$$\text{vrednost } v \ ::= \ \mathbf{true} \ \mid \ \mathbf{false} \ \mid \ \underline{n}$$

Če pa se lotimo izvajanja izraza **if** $\underline{0}$ **then** $\underline{0}$ **else** (**iszero** $\underline{1}$), ki je sintaktično popolnoma pravilen, naletimo na težavo, saj za izvajanje izraza **if** $\underline{0}$ **then** t_1 **else** t_2 nimamo pravila. Za razliko od konstant tak izraz ne predstavlja smiselnega rezultata izvajanja. Izraze, ki se ne morejo več izvesti, a niso vrednosti karakteriziramo kot napako.

Definicija 2.9 (Zataknjen izraz)**.** Za izraz, ki je v normalni obliki, a ni vrednost, pravimo, da je *zataknjen*.

Zataknjeni izrazi tako predstavljajo napako, saj ne predstavljajo rezultata, hkrati pa za njihovo izvajanje nimamo pravil. Poleg zataknjenih izrazov pa nas motijo tudi izrazi, ki sicer niso zataknjeni, a se (lahko) med kasnejšim izvajanjem zataknejo. Preprost primer takega izraza je naslednji izraz

$$\mathbf{iszero}\ (\mathbf{iszero}\ \underline{0}) \tag{3}$$

Sam izraz ni zataknjen, saj se po pravilu E-IsZeroZero pretvori v **iszero true**, ki pa je zataknjen.

2.3. **Tipi.** Naravno se pojavi vprašanje, kako bi lahko izraz, ki se potencialno zatakne, recimo izraz (3), odkrili. Ena izmed možnosti je seveda izvedba celotnega programa. Tak postopek je računsko zelo zahteven, hkrati pa nam ne zagotavlja, da do napake ne bo prišlo. Drugo orodje za odkrivanje potencialno zataknjenih izrazov je sistem tipov.

V tem razdelku predstavimo preprost sistem tipov za programski jezik in na nekaj primerih pokažemo njihovo izpeljavo. S tem položimo temelje za izreke v sledečih poglavjih, kjer s pomočjo tipov zagotovimo, da se ob določenih predpostavkah (in lastnostih programa) v programu ne morejo pojaviti zataknjeni izrazi.

Da v jezik dodamo tipe, najprej v tabeli 3 definiramo sintakso za tipe (ki jih označujemo z veliko črko T) in podamo dva osnovna tipa: tip celih števil (**Int**) in tip Booleovih vrednosti (**Bool**). V tabeli 4 pa so predstavljena pravila za dodeljevanje tipov izrazom.

$$\text{Tip } T \ ::= \ \mathbf{Int} \ \mid \ \mathbf{Bool}$$

Tabela 3. Osnovna tipa **Int** in **Bool**

Dodeljevanje tipov izrazom formaliziramo z binarno relacijo med izrazi in tipi.

$$\text{T-True}\ \frac{}{\mathbf{true} : \mathbf{Bool}} \qquad \text{T-False}\ \frac{}{\mathbf{false} : \mathbf{Bool}} \qquad \text{T-Num}\ \frac{}{\underline{n} : \mathbf{Int}} \qquad \text{T-Succ}\ \frac{t : \mathbf{Int}}{\mathbf{succ}\ t : \mathbf{Int}} \qquad \text{T-Pred}\ \frac{t : \mathbf{Int}}{\mathbf{pred}\ t : \mathbf{Int}}$$

$$\text{T-IsZero}\ \frac{t : \mathbf{Int}}{\mathbf{iszero}\ t : \mathbf{Bool}} \qquad \text{T-If}\ \frac{t_1 : \mathbf{Bool} \qquad t_2 : T \qquad t_3 : T}{\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 : T}$$

Tabela 4. Pravila za tipe

Definicija 2.10. Relacija *imeti tip* je najmanjša dvočlena relacija med izrazi in tipi, ki je zaprta za pravila v tabeli 4. Pišemo $t : T$ in beremo: izraz t ima (je tipa) T.

Pravila za izpeljavo tipov beremo podobno kot pravila za izvajanje. Za primer delovanja si oglejmo pravilo T-IsZero. Če velja, da je t tipa **Int**, potem je **iszero** t tipa **Bool**, ali z oznako: **iszero** t : **Bool**.

Oglejmo si še pogojni stavek. Za izpeljavo tipa pogojnega stavka potrebujemo tri predpostavke: pogoj mora imeti tip **Bool**, obe veji (t_2 in t_3) pa morata imeti isti tip T. Ob teh predpostavkah lahko zaključimo, da ima izraz tip T (enak tipu obeh vej).

Primer 2.11. Na primeru si poglejmo izpeljavo tipa izraza **iszero** (**pred** $\underline{0}$). Če želimo izpeljati tip izraza, uporabimo pravilo T-IsZero. Ob predpostavki, da ima **pred** $\underline{0}$ tip **Int**, iz tega pravila lahko izpeljemo, da ima celoten izraz tip **Bool**. Da pa izpeljemo tip **pred** $\underline{0}$, moramo zadostiti predpostavki pravila T-Pred, torej da ima $\underline{0}$ tip **Int**. Za tip konstante $\underline{0}$ lahko po pravilu T-Num vedno zaključimo da je **Int**. Celotno izpeljavo tipa izraza lahko podobno kot pri izvajanju prikažemo z drevesom spodaj.

$$\text{T-IsZero}\ \frac{\text{T-Pred}\ \dfrac{\text{T-Num}\ \dfrac{}{\underline{0} : \mathbf{Int}}}{\mathbf{pred}\ \underline{0} : \mathbf{Int}}}{\mathbf{iszero}\ (\mathbf{pred}\ \underline{0}) : \mathbf{Bool}}$$

♢

Primer 2.12. Oglejmo si še primer izpeljave tipa izraza (2), če namesto t vstavimo kar $\underline{0}$. Če želimo izpeljati, da ima celoten izraz pod črto tip **Bool**, moramo najprej ugotoviti, kakšne tipe imajo posamezni izrazi v pogojnem stavku. V pogoju imamo izraz **iszero** $\underline{0}$, čigar tip lahko izpeljemo, če vemo, kakšen tip ima izraz, ki predstavlja število 0 ($\underline{0}$). To je konstanta, zato po pravilu T-Num velja $\underline{0}$: **Int**, iz česar po T-IsZero velja **iszero** $\underline{0}$: **Bool**, kar izpolni prvi pogoj za izpeljavo tipa po pravilu T-If. Na podoben način izpeljemo tudi, da velja **true** : **Bool** in **iszero** (**pred** $\underline{0}$) : **Bool**, potem pa uporabimo pravilo T-If in dobimo, da ima celoten izraz tip **Bool**.

$$\dfrac{\dfrac{\dfrac{}{\underline{0} : \mathbf{Int}}\,\text{T-Num}}{\mathbf{iszero}\ \underline{0} : \mathbf{Bool}}\,\text{T-IsZero} \quad \dfrac{}{\mathbf{true} : \mathbf{Bool}}\,\text{T-True} \quad \dfrac{\dfrac{\dfrac{}{\underline{0} : \mathbf{Int}}\,\text{T-Num}}{\mathbf{pred}\ \underline{0} : \mathbf{Int}}\,\text{T-Pred}}{\mathbf{iszero}\ (\mathbf{pred}\ \underline{0}) : \mathbf{Bool}}\,\text{T-IsZero}}{\mathbf{if}\ (\mathbf{iszero}\ \underline{0})\ \mathbf{then}\ (\mathbf{true})\ \mathbf{else}\ (\mathbf{iszero}\ (\mathbf{pred}\ \underline{0})) : \mathbf{Bool}}\,\text{T-If}$$

♢

Opomba 2.13. Zgornja izpeljava tipa izraza bi potekala na povsem enak način, če bi namesto $\underline{0}$ vstavili izraz $\underline{n}$ za poljuben n. Še več, izpeljava tipov bi potekala povsem enako če bi namesto t vstavili poljuben izraz (ne nujno vrednost), za katero bi veljalo $t : \mathbf{Int}$.

Opomba 2.14. Pogosto je dovolj, če zgolj vemo da je izraz tipa T, kjer sama vrednost tipa nima posebnega pomena. Tedaj lahko rečemo tudi zgolj da izraz *ima tip*, in eksplicitno ne navedemo kakšen ta tip je.

Kaj pa se zgodi, če poskušamo izpeljati tip zataknjenega izraza **iszero true**. Tip izraza oblike **iszero** t za nek izraz t lahko izpeljemo edino iz pravila T-IsZero. Pravilo pravi da ob predpostavki, $t : \mathbf{Int}$ lahko zaključimo, da ime celoten izraz tip **Bool**. Da izpeljemo celoten tip moramo torej pokazati, da ima argument (**true**) tip **Int**. Tip izraza **true** pa je po pravilu T-True vedno **Bool**. Predpostavkam ni zadoščeno, torej ne moremo uporabiti pravila T-IsZero. Celoten izraz torej ni v relaciji z nobenim izmed definiranih tipov in torej nima tipa.

3. Izreki o varnosti

V tem poglavju predstavimo in dokažemo izrek o varnosti. V prvih razdelkih najprej predstavimo strukturno indukcijo in razložimo, kako poteka dokazovanje ter dokažemo nekaj pomožnih lem. Na koncu podamo in dokažemo izreka o varnosti in pojasnimo njune posledice.

3.1. Izrek o varnosti.

Ena izmed osnovnih lastnosti programov, ki jo zagotavljajo tipi, je *varnost*. Za izraz, ki ima tip, lahko zagotovimo, da se med izvajanjem ne bo zataknil.

Izrek 3.1 (Izrek o varnosti)**.** *Če ima izraz t tip T ($t : T$), potem izraz t v teku izvajanja ne more obtičati.*

Izrek o varnosti navadno predstavimo (in dokažemo) kot dva podizreka: izrek o napredku 3.5 in izrek o ohranitvi 3.6, ki ju bomo dokazali v naslednjih razdelkih.

Za dokaz obeh izrekov bomo uporabili indukcijo glede na velikost izrazov. Taka indukcija deluje na podoben način kot matematična indukcija, le da v indukcijskem koraku predpostavimo, da izrek velja za vse izraze, ki so manjše velikosti. Velikost izraza definiramo rekurzivno.

Definicija 3.2 (Velikost izraza)**.** Velikost izraza t, ki jo označimo z $|t|$ definiramo za vsako obliko izraza posebej:

- $|\mathbf{true}| := 1$
- $|\mathbf{false}| := 1$
- $|\underline{n}| := 1$
- $|\mathbf{succ}\ t| := 1 + |t|$

- $|\textbf{pred}\ t| := 1 + |t|$
- $|\textbf{iszero}\ t| := 1 + |t|$
- $|\textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3| := 1 + |t_1| + |t_2| + |t_3|$

Izrek 3.3 (Princip indukcije glede na velikost izraza)**.** *Naj bo P poljuben predikat na izrazih. Princip strukturne indukcije glede na velikost izraza se glasi: Če za poljuben izraz s, iz predpostavke, da $P(t)$ velja za vse izraze, za katere velja $|t| < |s|$, lahko pokažemo $P(s)$, potem predikat P velja za poljuben izraz t. Ali matematično*

$$(\forall s.\, (\forall t, |t| < |s|.\ P\,(t)) \implies P\,(s)) \implies \forall t.\ P\,(t) \tag{4}$$

Dokaz. Izrek bom dokazali s pomočjo principa matematične indukcije. Ta pravi, da za poljuben predikat Q na naravnih številih velja

$$(\forall n.\, (\forall m, m < n.\ Q\,(m)) \implies Q\,(n)) \implies \forall n.\ Q\,(n) \tag{5}$$

Za dokaz si definirajmo predikat

$$Q(n) = \forall s, |s| = n.\ P(s)$$

Dokazati moramo, da za poljubno naravno število velja predikat Q, to pa preprosto sledi iz definicije matematične indukcije.

□

Preden začnemo z dokazovanjem izreka o napredku in ohranitvi, si poglejmo še lemo o kanoničnih oblikah, ki nam bo v pomoč pri dokazovanju.

Lema 3.4 (Kanonične oblike)**.**

(1) *Če je v vrednost in velja $v : \textbf{Bool}$, potem je v ali* **true** *ali* **false**.
(2) *Če je v vrednost in velja $v : \textbf{Int}$, potem je v oblike $\underline{n}$ za neko celo število n.*

Dokaz. Dokaz preprosto sledi iz definicije 2.8. Po definiciji je lahko v oblike **true**, **false** ali $\underline{n}$, za nek n. Naravnost iz pravil za tipe sledi $\textbf{true} : \textbf{Bool}$, $\textbf{false} : \textbf{Bool}$ in $\underline{n} : \textbf{Int}$. Od tod pa sledijo zaključki leme. Če za vrednost v velja $v : \textbf{Bool}$, potem je v lahko zgolj **true** ali **false** in če je velja $v : \textbf{Int}$, potem je vrednost v oblike $\underline{n}$ za neko celo število n. □

3.2. **Izrek o napredku.**

Izrek 3.5 (izrek o napredku [3, Izrek 8.3.2])**.** *Izraz t, ki ima tip, ni zataknjen. Torej je t vrednost ali pa obstaja tak izraz t', da velja $t \rightsquigarrow t'$.*

Dokaz. Naj bo t izraz, ki ima tip T ($t : T$). Izrek bomo dokazali z indukcijo glede na velikost izraza. V vsakem koraku po principu strukturne indukcije predpostavimo, da izrek velja za vse izraze, ki so manjše velikosti. Ker iz definicije 3.2 sledi, da je velikost podizrazov vedno manjša od velikosti glavnega izraza, v poljubnem koraku predpostavimo veljavnost izreka za vse podizraze. Nato v vsakem koraku obravnavamo možnosti glede na zadnje uporabljeno pravilo pri izpeljavi tipa.

- *Možnosti* T-True, T-False, T-Num:
  Dokaz lahko dokončamo takoj, saj so izrazi kar vrednosti.
- *Možnost* T-Pred:
  Če je bilo zadnje pravilo uporabljeno za izpeljavo tipa T-Pred, je izraz t oblike $t = \textbf{pred}\ t_1$, za katerega velja $t_1 : \textbf{Int}$. Po indukcijski predpostavki je t_1 ali vrednost ali pa se lahko izvede v t_1'.

Poglejmo najprej primer, ko je t_1 vrednost. Ker velja $t_1 : \mathbf{Int}$, je torej t_1 po lemi 3.4 oblike $\underline{n}$ za neko celo število n. Izraz t v tem primeru lahko opravi korak v izvajanju, saj se po E-PredNum pretvori v $\underline{n-1}$.

Če pa t_1 ni vrednost, po indukcijski predpostavki obstaja tak t_1', da velja $t_1 \rightsquigarrow t_1'$, v tem primeru pa za t velja pravilo E-Pred in dobimo:

$$\mathbf{pred}\ t_1 \rightsquigarrow \mathbf{pred}\ t_1'$$

oziroma

$$t \rightsquigarrow t'$$

kjer je $t' = \mathbf{pred}\ t_1'$

- *Možnosti* T-Succ in T-IsZero:
  Dokaz je zelo podoben dokazu za T-Pred in ga ne bomo ponavljali.
- *Možnost* T-If:
  Če je bilo zadnje pravilo uporabljeno za izpeljavo tipa T-If, imamo izraz $t = \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3$, za katerega velja: $t_1 : \mathbf{Bool}, t_2 : T, t_3 : T$.
  Po indukcijski predpostavki je torej t_1 ali vrednost ali pa se lahko pretvori v t_1'. Če je t_1 vrednost, in ker velja $t_1 : \mathbf{Bool}$, je t_1 lahko zgolj **true** ali **false**. Sledi, da za celoten izraz t obstaja pravilo za izvajanje (v prvem primeru E-IfTrue, v drugem pa E-IfFalse). Če pa t_1 ni vrednost, po indukcijski predpostavki obstaja tak t_1', da velja $t_1 \rightsquigarrow t_1'$, v tem primeru pa se izraz t po pravilu E-If izvede v:

$$\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 \rightsquigarrow \mathbf{if}\ t_1'\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3$$

□

3.3. **Izrek o ohranitvi.**

Izrek 3.6 (izrek o ohranitvi [3, Izrek 8.3.3])**.** *Če izraz tipa T lahko opravi korak v izvajanju, je tudi rezultat izvajanja tipa T. Torej iz $t : T$ in $t \rightsquigarrow t'$ sledi $t' : T$.*

Dokaz. Kot izrek o napredku bomo tudi izrek o ohranitvi dokazali z indukcijo glede na velikost izraza in obravnavali zadnje uporabljeno pravilo pri izpeljavi tipa.

- *Možnost* T-True:
  Če je bilo zadnje pravilo uporabljeno za izpeljavo tipa T-True, vemo, da je velja **true** : **Bool**. Ker se **true** ne more več izvesti, je pogoj izreka izpolnjen na prazno.
- *Možnosti* T-False in T-Num:
  Dokaz je analogen dokazu za T-True.
- *Možnost* T-Pred:
  Če je bilo zadnje pravilo uporabljeno za izpeljavo tipa T-Pred, imamo izraz oblike
  $t = \mathbf{pred}\ t_1$, za katerega velja $t_1 : \mathbf{Int}$.
  Izraz **pred** t_1 se v izvajanju pojavi v natanko dveh primerih: E-PredNum in E-Pred.
  - *Primer* E-PredNum: Velja $t_1 = \underline{n}$ za neko celo število n. Izraz **pred** $\underline{n}$ se po E-PredNum pretvori v vrednost $\underline{n-1}$, ki ima tip **Int**, torej izraz pri izvajanju ohrani tip.

  - *Primer* E-Pred:
    Vemo, da velja $t_1 \rightsquigarrow t_1'$, iz indukcijske predpostavke pa sledi, da velja tudi $t_1' : \mathbf{Int}$. Izraz t se pretvori v $\mathbf{pred}\ t_1'$, za kar pa po pravilu T-Pred velja $\mathbf{pred}\ t_1' : \mathbf{Int}$, torej izraz ohrani tip.
- *Možnosti* T-Succ in T-IsZero:
  Dokaz je zelo podoben dokazu za T-Pred in ga ne bomo ponavljali.
- *Možnost* T-If:
  Če je bilo zadnje pravilo uporabljeno za izpeljavo tipa T-If, imamo izraz oblike
  $\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3$, za katerega velja: $t_1 : \mathbf{Bool}, t_2 : T, t_3 : T$.
  Pogledamo vse možnosti v izvajanju $t \rightsquigarrow t'$, kjer se pojavi pogojni stavek. Temu zadoščajo tri pravila: E-IfTrue, E-IfFalse, E-If, ki jih obravnavamo vsako posebej.
  - *Primer* E-IfTrue:
    Vemo, da velja $t_1 = \mathbf{true}$ in $t' = t_2$.
    Ker je $t' = t_2$ in ker po indukcijski predpostavki velja $t_2 : T$, rezultat izvajanja ohrani tip.
  - *Primer* E-IfFalse:
    Dokažemo podobno kot E-IfTrue.
  - *Primer* E-If:
    Vemo, da velja $t_1 \rightsquigarrow t_1'$ in $t' = \mathbf{if}\ t_1'\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3$.
    Ker v tem primeru vemo, da ima izraz t_1 tip **Bool**, lahko uporabimo indukcijsko predpostavko in iz $t_1 \rightsquigarrow t_1'$ dobimo $t_1' : \mathbf{Bool}$. Skupaj z predpostavkami, da $t_2 : T$ in $t_3 : T$, lahko na izrazu $\mathbf{if}\ t_1'\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3$ uporabimo pravilo za tipe T-If. Sledi, da velja $t' : T$, torej med izvajanjem izraz ohrani tip.

□

Kombinacija izreka o napredku in izreka o ohranitvi nam zagotavlja, da se izraz, ki ima tip, tekom izvajanje ne zataknil. S stališča programskega jezika nam smiselnost tipov tako daje predhodno zagotovilo, da v programu med izvajanjem ne bo prišlo do napake. Seveda pa smiselnost tipov ni pogoj za to, da se izraz ne bo zataknil.

$$\mathbf{iszero}\ (\mathbf{if}\ \mathbf{true}\ \mathbf{then}\ \underline{0}\ \mathbf{else}\ \mathbf{false})$$

Izraz nima tipa, ker imata $\underline{0}$ in **false** različen tip, a se vseeno (po dveh korakih) izvede v vrednost **true**.

Glavna prednost tipov je predvsem to, da lahko tip izraza izpeljemo že pred samim izvajanjem. Še več, v našem jeziku lahko tip izraza izpeljemo celo neodvisno od vhodnih podatkov. Tako lahko v izrazu (2), ki preveri ali je t idempotent zgolj ob predpostavki, kakšen je tip izraza t, neodvisno od vrednosti, določimo tip celotnega izraza.

4. Lambda račun

V tem poglavju prej predstavljenemu programskemu jeziku dodamo lambda funkcije in jezik ustrezno nadgradimo v obliko lambda računa.

4.1. **Operacijska semantika.** V prejšnjih poglavjih smo pokazali kako lahko za preprost jezik s pomočjo tipov zagotovimo varnost. Ker pa je prikazan jezik zelo

šibak, bi ga radi razširili s funkcijami in tako dobili jezik, znan kot lambda račun [3, Poglavje 5].

Za dodajanje funkcij najprej dopolnimo sintakso. Ta poleg do sedaj veljavnih izrazov vsebuje tudi tri nove konstrukte. Prvi je spremenljivka. Z drugim predstavimo funkcijo (abstrakcijo), ki spremenljivki x priredi poljuben veljaven izraz t. Za tako spremenljivko x pravimo da je *vezana* v tej abstrakciji. Tretji pa je izraz, ki predstavlja uporabo)izraza (funkcije predstavljene s) t_1 na argumentu, ki ga predstavlja izraz t_2. Posodobljeno sintakso vsebuje tabela 5.

$$\text{Izraz } t \ ::= \ \cdots \ \Big| \ x \ \Big| \ \lambda x.\ t \ \Big| \ t_1\ t_2$$

Tabela 5. Sintaksa lambda računa

Poglejmo si preprosto funkcijo ki spremenljivki x priredi izraz **iszero** x.

$$\lambda x.\ \textbf{iszero}\ x \tag{6}$$

Ideja funkcije je popolnoma identična matematični funkciji, le da tu funkcije ne poimenujemo in uporabimo namesto $x \mapsto$ **iszero** x nekoliko drugačno obliko zapisa.

Definicija 4.1. Kot je navada v funkcijskih jezikih, so poleg konstant vrednosti tudi abstrakcije. Definicijo vrednosti zato ustrezno posodobimo.

$$\text{vrednost } v \ ::= \ \cdots \ \Big| \ \lambda x.\ t$$

Definicija 4.2 (Vezana spremenljivka)**.** Pojavitev spremenljivke x je *vezana*, če se pojavi v telesu (t) abstrakcije $\lambda x.\ t$. Natančneje, pojavitev x je vezana s to abstrakcijo ($\lambda x.\ t$), pomembno je tudi dejstvo, da lahko spremenljivke vezane v poljubni abstrakciji vedo preimenujemo

Definicija 4.3 (Prosta spremenljivka)**.** Pojavitev spremenljivke x je *prosta* (ni vezana), če se pojavi v izrazu, kjer ni vezana s katero od zunanjih abstrakcij.

Definicija 4.4 (Zaprt izraz)**.** Izrazu, v katerem ni prostih spremenljivk rečemo *zaprt izraz*.

Primer 4.5.

$$\lambda y.\ \textbf{if}\ y\ \textbf{then}\ (\textbf{succ}\ x)\ \textbf{else}\ (\textbf{pred}\ x)$$

V izrazu je y vezana spremenljivka, saj je vezana z $\lambda y.$, medtem, ko je x prosta spremenljivka, saj ni vezana v nobeni abstrakciji. V abstrakciji iz izraza (8) pa sta vezani tako x, kot tudi y, le da je y vezana v bolj notranji abstrakciji.

♢

V tabeli 6 podamo dodatna pravila za izvajanje v lambda računu. Prvo pravilo pove, kako se pretvori prvi izraz (abstrakcija), ki nastopa v uporabi funkcije, drugo, kako se pretvori argument abstrakcije, tretje pa, kako s pomočjo substitucije abstrakcijo uporabimo.

Uporaba abstrakcije na izrazu (in substitucija spremenljivke) poteka na enak način kot v običajnih matematičnih funkcijah.

Primer 4.6.

$$(\lambda x.\ \textbf{iszero}\ x)\ \underline{0} \tag{7}$$

♢

$$\text{E-App1}\ \frac{t_1 \rightsquigarrow t_1'}{t_1\ t_2 \rightsquigarrow t_1'\ t_2} \qquad \text{E-App2}\ \frac{t_2 \rightsquigarrow t_2'}{v\ t_2 \rightsquigarrow v\ t_2'} \qquad \text{E-AppAbs}\ \frac{}{(\lambda x.\ t)(v) \rightsquigarrow [x \mapsto v]t}$$

Tabela 6. Pravila za izvajanje lambda računa

V izrazu (7) abstrakcijo iz izraza (6) uporabimo na izrazu, ki predstavlja število 0. V izvajanju bi, kot v običajni funkciji, vse nastope x v prvotnem izrazu zamenjali z $\underline{0}$ in dobili izraz **iszero** $\underline{0}$. Substitucijo spremenljivke x z izrazom y v izrazu t zapišemo kot: $[x \mapsto y]t$, pri substituciji pa pazimo, da ne zamenjamo nastopov spremenljivke x, ki je vezana v kateri izmed bolj notranjih abstrakcij izraza t. Idejo si najlažje pogledamo na primeru.

Primer 4.7. V izrazu (8) definiramo funkcijo višjega reda. Če to funkcijo uporabimo na nekem številu, dobimo novo funkcijo, ki Booleovi vrednosti priredi naslednik (ali predhodnik) števila, ki smo ga podali prvi funkciji. Vidimo, da smo ime spremenljivke v notranji abstrakciji ($\lambda y.$ **if** y **then** (**succ** x) **else** (**pred** x)) izbrali tako, da ni »prekrila« spremenljivke x iz zunanje.

(8) $$\lambda x.\ (\lambda y.\ \textbf{if}\ y\ \textbf{then}\ (\textbf{succ}\ x)\ \textbf{else}\ (\textbf{pred}\ x))\ \underline{0}$$

Po substituciji se izraz pričakovano pretvori v

$$\lambda y.\ \textbf{if}\ y\ \textbf{then}\ (\textbf{succ}\ \underline{0})\ \textbf{else}\ (\textbf{pred}\ \underline{0})$$

♢

Izvajanje na enak način kot prej formaliziramo z dvočleno relacijo med izrazi.

Definicija 4.8 (Semantika malih korakov v lambda računu)**.** *Semantika malih korakov v lambda računu* je najmanjša dvomestna relacija med izrazi, ki ustreza pravilom podanim v tabeli 2 in tabeli 6. Pišemo $t \rightsquigarrow t'$ in beremo t se pretvori v t'.

Opomba 4.9. Definiciji normalne oblike in zataknjenega izraza ostaneta enaki.

Primer 4.10. Poglejmo si kako lahko s pomočjo izraza (2) v izrazu (9) sestavimo funkcijo, ki preveri ali je njen argument idempotent.

(9) $$\textit{idempotent} := \lambda x.\ \textbf{if}\ (\textbf{iszero}\ x)\ \textbf{then true else}\ (\textbf{iszero}\ (\textbf{pred}\ x))$$

Uporabimo funkcijo *idempotent* na izrazu $\underline{0}$ (seveda bi izvajanje za poljuben izraz $\underline{n}$ potekalo podobno). Izpeljavo izvajanja je prikazana v drevesu spodaj. Edino pravilo, za izvajanje, ki ga lahko uporabimo je E-AppAbs (pravil E-App1 in E-App2 ne moremo uporabiti, saj niti abstrakcija sama niti $\underline{n}$ ne moreta opraviti koraka v izvajanju). Pravilo E-AppAbs pravi, rezultat koraka izvajanja dobimo tako, da vse ponovitve spremenljivke x v izrazu **if** (**iszero** x) **then true else** (**iszero** (**pred** x)) nadomestimo z vrednostjo $\underline{0}$.

$$\text{E-AppAbs}\ \frac{}{\begin{array}{l}(\lambda x.\ \textbf{if}\ (\textbf{iszero}\ x)\ \textbf{then true else}\ (\textbf{iszero}\ (\textbf{pred}\ x)))\ \underline{0} \rightsquigarrow \\ \quad \rightsquigarrow\ \textbf{if}\ (\textbf{iszero}\ \underline{0})\ \textbf{then true else}\ (\textbf{iszero}\ (\textbf{pred}\ \underline{0}))\end{array}}$$

♢

Seveda tudi v lambda računu lahko dobimo zataknjen izraz. Preprost primer lahko vzamemo kar iz prejšnjega jezika: **iszero true**. Drugi tak primer pa je izraz

true false

Sintaktično je izraz popolnoma veljaven (izraz **false** smo uporabili na izrazu **true**), zanj pa ni pravila za izvajanje (da bi lahko uporabili pravilo E-AppAbs bi moral biti izraz **true** abstrakcija) in ker ni vrednost, je zataknjen.

4.2. **Lambda račun z enostavnimi tipi.** Lambda računu dodamo tipe, da bi lahko na enak način kot prej formalno dokazali da med izvajanjem izraza, ki ima tip, ne bo prišlo do napake. Pri tem dobimo jezik, ki je sicer znan kot lambda račun z enostavnimi tipi [3, Tabela 9.1].

Dovolj bo, če povemo kako bo s tipi funkcij, saj pravila za tipe osnovnega jezika ostanejo enaka. Kot v matematiki funkcijo karakteriziramo z domeno (tipom argumenta) in kodomeno (tipom rezultata).

$$\text{Tip } T \ ::= \ \cdots \ \mid \ T_1 \to T_2$$

Ustaljenim tipom dodamo še tip funkcije, ki vsebuje tip argumenta in tip rezultata.

Poglejmo abstrakcijo iz izraza (6): $\lambda x.$ (**iszero** x), ki bi, če bi jo po uporabi na izrazu $\underline{0}$ izvedli do konca, vrnila rezultat **true**. Vidimo, da ima rezultat izvajanja tip **Bool**.

Za splošno funkcijo $\lambda x.\ t$ torej ob predpostavki $x : T_1$ velja $t : T_2$. Za razliko od prej za tip izraza ni več odgovorna samo oblika izraza, ampak tudi tipi spremenljivk (natančneje predpostavke o teh tipih), ki v njem nastopajo. Zato definiramo *kontekst*, ki ima informacije o predpostavkah glede tipov prostih (nevezanih) spremenljivk v izrazu t.

Definicija 4.11 (Kontekst)**.** Kontekst Γ je zaporedje parov različnih spremenljivk in njihovih tipov: (x, T), kjer za spremenljivko x predpostavimo, da ima tip T. Različnost spremenljivk lahko zahtevamo, saj lahko spremenljivke v poljubni abstrakciji preimenujemo.

Definicija 4.12 (Domena konteksta)**.** Domena konteksta $dom(\Gamma)$, je množica vseh spremenljivk, ki se nahajajo v kontekstu. $dom(\Gamma) := \{x \mid \exists T.\ (x, T) \in \Gamma\}$

Formalno relacijo *imeti tip* razširimo na tri elemente: kontekst, izraz in tip: (Γ, t, T) in pišemo $\Gamma \vdash t : T$. Če se spomnimo primera 4.7 je pomembno, da ob definiciji nove abstrakcije uporabimo tako spremenljivko, ki ne »zasenči« kakšne proste spremenljivke. To dejstvo nam zagotovi, da se vsi pari $(x_i : T_i)$ v kontekstu razlikujejo v imenih spremenljivke.

$$\text{T-Var}\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{T-Abs}\ \frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x.\ t_2 : T_1 \to T_2} \qquad \text{T-App}\ \frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}$$

Tabela 7. Pravila za tipe lambda računa z enostavnimi tipi

Definicija 4.13. Relacija *imeti tip* je najmanjša tročlena relacija med konteksti, izrazi in tipi, ki ustreza pravilom v tabeli 4 in tabeli 7. Pišemo $\Gamma \vdash t : T$ in beremo: izraz t ima v kontekstu Γ tip T .

Z dodatnimi pravili za tipe, ki jih vidimo v tabeli 7, lambda račun postane tipiziran, potreben je le kratek komentar. Tip spremenljivke (T-Var) je po novih pravilih kar tip, ki smo ga za to spremenljivko predpostavili v kontekstu Γ. Pri tipu funkcije (T-Abs) razširimo kontekst. Pravilo beremo takole: Če v kontekstu Γ, razširjenem s predpostavko $x : T_1$, velja $t_2 : T_2$, potem lahko zaključimo, da ima funkcija $\lambda x.\ t_2$ v kontekstu Γ tip $T_1 \to T_2$. Pravilo T-App pa nam podaja tip vrednosti uporabe funkcije ob predpostavki, da ima glede na kontekst funkcija t_1 tip $T_1 \to T_2$ in izraz t_2 tip T_1.

Primer 4.14. Primer uporabe pravil se poglejmo na izpeljavi tipa funkcije iz izraza (9), ki preveri, ali je njen argument idempotent.

$$\text{T-Abs}\ \dfrac{\text{T-IsZero}\ \dfrac{\text{T-If}\ \dfrac{\text{T-IsZero}\ \dfrac{\text{T-Var}\ \dfrac{x : \mathbf{Int} \in x : \mathbf{Int}}{x : \mathbf{Int} \vdash x : \mathbf{Int}}}{x : \mathbf{Int} \vdash \mathbf{iszero}\ x : \mathbf{Bool}} \quad \vdots \quad \text{T-IsZero}\ \dfrac{\text{T-Pred}\ \dfrac{\text{T-Var}\ \dfrac{x : \mathbf{Int} \in x : \mathbf{Int}}{x : \mathbf{Int} \vdash x : \mathbf{Int}}}{x : \mathbf{Int} \vdash \mathbf{pred}\ x : \mathbf{Int}}}{x : \mathbf{Int} \vdash \mathbf{iszero}\ (\mathbf{pred}\ x) : \mathbf{Bool}}}{x : \mathbf{Int} \vdash \mathbf{if}\ (\mathbf{iszero}\ x)\ \mathbf{then\ true\ else}\ (\mathbf{iszero}\ (\mathbf{pred}\ x)) : \mathbf{Bool}}}{\emptyset \vdash \lambda x.\ \mathbf{if}\ (\mathbf{iszero}\ x)\ \mathbf{then\ true\ else}\ (\mathbf{iszero}\ (\mathbf{pred}\ x)) : \mathbf{Int} \to \mathbf{Bool}}$$

♢

5. Izrek o varnosti v lambda računu z enostavnimi tipi

Tudi v lambda računu z enostavnimi tipi izrek o varnosti zagotovi, da izraz med izvajanjem ne bo dosegel neveljavnega stanja. Na enak način ga dokažemo v obliki dveh podrizekov. Za pomoč pri dokazovanju izreka o ohranitvi navedemo in dokažemo nekaj pomembnih lem o delovanju kontekstov. Pri tem sledimo [3], Poglavje 9.3.

Izrek 5.1 (izrek o varnosti v lambda računu z enostavnimi tipi)**.** *Če ima zaprt izraz t tip T ($t : T$), potem izraz t v teku izvajanja ne more obtičati.*

Za dokaz bomo enako kot prej uporabili indukcijo na velikost izraza. Pomembna razlika med izreko za lambda račun in izrekom 3.1 je dejstvo, da v izreku za lambda račun zahtevamo, da je izraz zaprt (da ne vsebuje prostih spremenljivk). Ti ni nikakršna pomanjkljivost jezika, ali omejitev izreka, saj nas zanimajo zgolj programi, ki ne vsebujejo prostih spremenljivk. Če v celotnem programu nastopajo proste spremenljivke so le-te nedefinirane in program kot tak je nesmiseln.

Pred izrekom o napredku si poglejmo še lemo o kanoničnih oblikah.

Lema 5.2 (Kanonične oblike)**.**

(1) *Če je v vrednost in velja $v : \mathbf{Bool}$, potem je v ali* **true** *ali* **false**.
(2) *Če je v vrednost in velja $v : \mathbf{Int}$, potem je v oblike $\underline{n}$ za neko celo število n.*
(3) *Če je v vrednost in velja $v : T_1 \to T_2$, potem je v oblike $\lambda x.\ t_2$.*

Dokaz. Dokaz enako kot v lemi 3.4 sledi iz definicije 4.1. □

5.1. Izrek o napredku.

Izrek 5.3 (Izrek o napredku za lambda račun z enostavnimi tipi [3], Izrek 9.3.5)**.** *Zaprt izraz t, ki ima tip, ni zataknjen. Torej če $\emptyset \vdash t : T$, potem je t ali vrednost, ali pa obstaja tak t', da velja $t \rightsquigarrow t'$.*

Dokaz. Naj bo t izraz, ki ima tip T ($t : T$). Izrek bomo dokazali z indukcijo glede na velikost izraza in obravnavali možnosti glede na zadnje uporabljeno pravilo pri izpeljavi tipa. Če je bilo zadnje uporabljeno pravilo pri izpeljavi tipa vzeto iz tabele 4 (torej ni bilo eno izmed dodatnih pravil za tipe v lambda računu), je dokaz povsem enak dokazu izreka o napredku 3.5. Obravnavati moramo torej še tri možnosti iz tabele 7.

- *Možnost* T-Var:
  Izraz t je oblike x, kjer je x spremenljivka. Ker izraz ni zaprt (x je prosta spremenljivka, saj ni vezana v nobeni abstrakciji), je pogoj izreka izpolnjen na prazno.
- *Možnost* T-Abs:
  Če je bilo zadnje pravilo uporabljeno za izpeljavo tipa T-Abs, potem imamo izraz oblike $t = \lambda x.\ t_1$. Ker je abstrakcija vrednost, je izrek dokazan.
- *Možnost* T-App:
  Če je bilo zadnje pravilo uporabljeno za izpeljavo tipa T-App, potem imamo izraz oblike $t = t_1\ t_2$ in $\emptyset \vdash t_1 : T_1 \to T$ in $\emptyset \vdash t_2 : T_1$. Po indukcijski predpostavki, tako za t_1 kot za t_2 velja, da sta ali vrednosti, ali pa lahko opravita še (vsaj en) korak izvajanja. Glede na to, ali sta t_1 in t_2 vrednosti obravnavamo več primerov:
  - t_1 *ni vrednost*:
    Če t_1 ni vrednost, potem po indukcijski predpostavki obstaja tak t_1', da velja $t_1 \rightsquigarrow t_1'$. Neodvisno od t_2, za izraz t velja pravilo za izvajanje E-App1 in izraz $t_1\ t_2$ se pretvori v izraz $t_1'\ t_2$.
  - t_1 *je vrednost,* t_2 *ni vrednost*:
    Če t_2 ni vrednost, potem po indukcijski predpostavki obstaja tak t_2, da velja $t_2 \rightsquigarrow t_2'$. Za izraz t potem velja pravilo E-App2 in izraz $t_1\ t_2$ se pretvori v $t_1\ t_2'$.
  - t_1 *in* t_2 *sta vrednosti*:
    Če je t_1 vrednost velja, da ima tip $T_1 \to T$ in je torej po lemi o kanoničnih oblikah oblike $\lambda x.\ t_1$. Izraz t je torej oblike $(\lambda x.\ t_1)\ t_2$, izraz take oblike pa se po pravilu E-AppAbs pretvori v $[x \mapsto t_2]t_1$.

□

5.2. **Izrek o ohranitvi.** V tem razdelku bomo dokazali izrek o ohranitvi za lambda račun z enostavnimi tipi. Najprej bomo dokazali nekaj dodatnih strukturnih lem, ki jih dokaz potrebuje, na koncu pa dokazali sam izrek. Pri tem bom sledili [3].

Najprej navedemo dve preprosti (in dokaj očitni lemi): prva nam pove, da če ima izraz t tip v nekem kontekstu Γ, potem ima izraz t isti tip v poljubno permutiranem kontekstu. Drugi izrek pa nam pove, da če ima izraz t v kontekstu nek tip, potem ta tip ohrani tudi, če v kontekst dodamo novo spremenljivko.

Lema 5.4 (Permutacijska lema)**.** *Če velja* $\Gamma \vdash t : T$ *in je* Δ *poljubna permutacija konteksta* Γ*,* t *tudi v permutiranem kontekstu ohrani tip:* $\Delta \vdash t : T$*.*

Dokaz. Preprosto z indukcijo glede na velikost izraza. □

Lema 5.5 (Lema o ošibitvi)**.** *Če velja* $\Gamma \vdash t : T$ *in* $x \notin dom(\Gamma)$ *potem* $\Gamma, x : T_1 \vdash t : T$

Dokaz. Preprosto z indukcijo glede na velikost izraza. □

S pomočjo zgornjih lem lahko dokažemo ključno lastnost lambda računa z enostavnimi tipi: tip izraza se ohrani tudi po substituciji spremenljivk z ustreznimi tipi.

Lema 5.6 (Ohranitev tipov med substitucijo)**.** *Če ima izraz t v kontekstu Γ, ki ga razširimo z $x : S$ tip T in ima v kontekstu Γ izraz s tip S, potem ima tudi rezultat substitucije spremenljivke x v izrazu t z izrazom s ($[x \mapsto s]t$) v kontekstu Γ tip T.*

Dokaz. Lemo bomo dokazali z indukcijo glede na velikost izraza, kjer bomo obravnavali možnosti glede na zadnje uporabljeno pravilo pri izpeljavi tipa glede na kontekst ($\Gamma, x : S \vdash t : T$). V vsakem koraku indukcije predpostavimo, da izrek velja za poljuben izraz t in za poljuben kontekst Γ.

- *Možnost* T-True:
  Če je bilo zadnje pravilo uporabljeno za izpeljavo tipa T-True, za trditev izreka velja $t =$ **true** in $T =$ **Bool**. Če v izrazu **true** vse pojavitve spremenljivke x (ki v izrazu sploh ne nastopa) zamenjamo z izrazom s dobimo začetni izraz **true**, za katerega pa po predpostavkah vemo, da velja: $\Gamma \vdash$ **true** : **Bool**
- *Možnosti* T-False, T-Num:
  Dokaz je analogen dokazu za T-True
- *Možnost* T-Pred:
  Če je bilo zadnje pravilo uporabljeno za izpeljavo tipa T-Pred, za trditev izreka velja:
  $t =$ **pred** t_1 in $\Gamma, x : S \vdash t_1 :$ **Int**. Na izrazu t_1 lahko uporabimo indukcijsko predpostavki in dobimo $\Gamma \vdash [x \mapsto s]t_1 :$ **Int**. Na izrazu **pred** $[x \mapsto s]t_1$ uporabimo pravilo T-Pred, po katerem se izraz pretvori v **pred** t_1, ki ima tip **Int**.
- *Možnosti* T-Succ, T-Zero:
  Dokaz je analogen dokazu za T-Pred
- *Možnost* T-If
  Če je bilo zadnje pravilo uporabljeno za izpeljavo tipa T-If, imamo izraz oblike: $t =$ **if** t_1 **then** t_2 **else** t_3 za katerega velja $\Gamma, x : S \vdash t_1 :$ **Bool**, $\Gamma, x : S \vdash t_2 : T$ in $\Gamma, x : S \vdash t_3 : T$. Če za vsakega izmed pravih podizrazov uporabimo indukcijsko predpostavko, lahko po pravilu T-If pa izpeljemo tip celotnega izraza
- *Možnost* T-Var:
  Če je bilo zadnje pravilo uporabljeno za izpeljavo tipa T-Var, za izraz t velja, da je oblike $t = z$ za neko spremenljivko z, velja pa tudi $z : T \in (\Gamma, x : S)$. Obravnavamo dva primera
  - *Primer $z = x$*:
    Če je spremenljivka z enaka x, potem glede na pravilo za substitucijo velja: $[x \mapsto s]z = s$. Zahtevan pogoj izreka $\Gamma \vdash s : S$ je torej izpolnjen, sa je enak drugi predpostavki izreka.
  - *Primer $z \neq x$*:
    Če je z od x različna spremenljivka, glede na pravilo za substitucijo velja: $[x \mapsto s]z = z$, ki pa ima tip T glede na predpostavke izreka.
- *Možnost* T-Abs:
  Če je bilo zadnje pravilo uporabljeno za izpeljavo tipa T-Abs, imamo izraz oblike $t = \lambda y.\ t_1$, $T = T_2 \to T_1$ in $\Gamma, x : S, y : T_2 \vdash t_1 : T_1$.

Z uporabo permutacijske leme velja $\Gamma, y : T_2, x : S \vdash t_1 : T_1$. Z uporabo leme o ošibitvi na drugi predpostavki izreka pa dobimo $\Gamma, y : T_2 \vdash s : S$. Sedaj lahko z uporabo indukcijske predpostavke izpeljemo $\Gamma, y : T_2 \vdash [x \mapsto s]t_1 : T_1$. Ker po definiciji substitucije za izraz t velja: $[x \mapsto s]t = \lambda y.\ [x \mapsto s]t_1$, z uporabo pravila T-Abs dobimo $\Gamma \vdash \lambda y.\ [x \mapsto s]y : T_2 \to T_1$, s čimer je pogoj leme dokazan.

- *Možnost* T-App:
  Če je bilo zadnje pravilo uporabljeno za izpeljavo tipa T-App, imamo izraz oblike $t = t_1\ t_2$, kjer za posamezne podizraze velja: $\Gamma, x : S \vdash t_1 : T_2 \to T_1$, $\Gamma, x : S \vdash t_2 : T_2$ in $T = T_1$.
  Po uporabi indukcijske predpostavke tako na t_1 kot na t_2 dobimo $\Gamma \vdash [x \mapsto s]t_1 : T_2 \to T_1$ in $\Gamma \vdash [x \mapsto s]t_2 : T_2$. Po pravilu T-App velja $\Gamma \vdash [x \mapsto s]t_1\ [x \mapsto s]t_2 : T$, kar pa je po enako $\Gamma \vdash [x \mapsto s](t_1\ t_2) : T$.
  To pa je ravno pogoj leme, ki je s tem dokazan.

□

Izrek 5.7 (Izrek o ohranitvi v lambda računu z enostavnimi tipi [3, Izrek 9.3.4])**.** *Če izraz t, ki ima v danem kontekstu Γ tip T, lahko opravi korak v izvajanju, potem v enakem kontekstu rezultat izvajanja ohrani tip. Torej iz $\Gamma \vdash t : T$ in $t \rightsquigarrow t'$ sledi $\Gamma \vdash t' : T$.*

Dokaz. Izrek bomo dokazali z indukcijo glede na velikost izraza, kjer bomo obravnavali možnosti glede na zadnje uporabljeno pravilo pri izpeljavi tipa. Za vsako možnost bomo preverili, da glede na veljavna pravila za izvajanje izraz t' res ohrani tip. Enako kot pri prejšnjem izreku bomo obravnavali zgolj dodatna pravila za izpeljavo tipov, saj je dokaz ostalih možnosti enak kot v 3.6.

- *Možnost* T-Var:
  Če je bilo zadnje pravilo uporabljeno za izpeljavo tipa T-Var, vemo, da je velja $t = x$ za neko spremenljivko x in $(x, T) \in \Gamma$. Pogoj izreka je na prazno izpolnjen, saj (samo) spremenljivka ne nastopa v nobenem izmed pravil izvajanja.
- *Možnost* T-Abs:
  Če je bilo zadnje pravilo uporabljeno za izpeljavo tipa T-Abs, velja: $t = \lambda x.\ t_1$ in $T = T_1 \to T_2$. Podobno kor pri možnosti T-Var abstrakcija sama ne nastopa v nobenem izmed pravil izvajanja in pogoj izreka je izpolnjen na prazno.
- **Možnost** T-App:
  Če je bilo zadnje pravilo uporabljeno za izpeljavo tipa T-App, velja: $t = t_1\ t_2$, $\Gamma \vdash t_1 : T_1 \to T$, $\Gamma \vdash t_2 : T_1$. Aplikacija izraza se pojavi v treh različnih pravilih za izvajanje, ki jih obravnavamo posebej.
  - *Primer* E-App1:
    V tem primeru velja $t_1 \rightsquigarrow t_1'$ in $t' = t_1'\ t_2$. Iz predpostavk izpeljave tipa T-Abs imamo izpeljavo tipa za t_1: $\Gamma \vdash t_1 : T_1 \to T$ in $\Gamma \vdash t_2 : T_1$. Za prave podizraze lahko uporabimo indukcijsko predpostavko in dobimo $\Gamma \vdash t_1' : T_1 \to T$. Za izraz $t' = t_1'\ t_2$ uporabimo pravilo T-App in dobimo, da za izraz t' velja: $\Gamma \vdash t' : T$.
  - *Primer* E-App2:
    Dokaz poteka analogno dokazu za E-App1, le da indukcijsko predpostavko uporabimo na t_2.

– *Primer* E-AppAbs: V tem primeru velja: $t_1 = \lambda x.\ t_{12}$, $t_2 = v_2$ (da lahko uporabimo to pravilo mora biti t_2 vrednost) in $t' = [x \mapsto v_2]t_{12}$. Edina možnost izpeljave tipa za izraz t_1 nam da $\Gamma, x : T_1 \vdash t_{12} : T$. Substitucija na izrazu t_{12} po lemi 5.6 ohrani tip in dobimo $\Gamma \vdash t' : T$, s čimer je dokaz zaključen.

□

6. Nadgradnja lambda računa z enostavnimi tipi

V tem poglavju si ogledamo, kako lahko s pomočjo negibne točke funkcije v lambda račun z enostavnimi tipi dodamo sposobnost rekurzije.

6.1. **Motivacija in ideja.** Kot smo omenili na začetku je prvi definiran programski jezik dokaj šibak (preverjanje, ali je element enak 0 ali 1, v programiranju ni ravno zavidanja vreden dosežek). Z definicijo funkcij smo dobili močnejši jezik (lambda račun z enostavnimi tipi), ki pa še vedno ni tako močan kot Turingov stroj. Za lambda račun z enostavnimi tipi lahko pokažemo [3, Izrek 12.1.6], da vsak izraz, ki ima tip, konča z izvajanjem (doseže normalno obliko) v končno mnogo korakih. Lambda račun brez tipov je sicer ekvivalenten Turingovemu stroju, vendar pa s tem izgubimo varnost, ki jo zagotavljajo tipi. Pri idejah in motivaciji si pomagamo z [8] in [2].

Primer 6.1. Recimo, da si želimo definirati funkcijo, ki bi za poljubno nenegativno (zaradi preprostosti ne bomo preverjali ali je število negativno ali pozitivno) število preverila, ali je to število sodo ali liho. S pomočjo do sedaj znanega bi lahko napisali izraz viden v enačbi (10).

$$
\begin{aligned}
&\mathit{jeSodo} := \lambda x.\\
&\qquad \textbf{if } (\textbf{iszero } x) \textbf{ then true}\\
&\qquad\quad \textbf{else } (\textbf{if } (\textbf{iszero } (\textbf{pred } x)) \textbf{ then false}\\
&\qquad\qquad \textbf{else } (\mathit{jeSodo}\ (\textbf{pred } (\textbf{pred } x)))\\
&\qquad\quad)
\end{aligned}
\tag{10}
$$

♢

Ključna ideja, ki jo pri tem uporabimo, je, da pri definiciji funkcije uporabimo kar samo sebe. Vendar pa ta način žal ne gre, v zapisu abstrakcije *jeSodo* ne moremo uporabiti kar abstrakcije same. Nam pa ideja da pomemben vpogled v težavo, če nam uspe funkciji na nek način podati samo sebe, smo dosegli cilj.

Primer 6.2.

$$
\begin{aligned}
&\Phi = \lambda f.\\
&\qquad \lambda x.\\
&\qquad\quad \textbf{if } (\textbf{iszero } x) \textbf{ then true}\\
&\qquad\qquad \textbf{else } (\textbf{if } (\textbf{iszero } (\textbf{pred } x)) \textbf{ then false}\\
&\qquad\qquad\quad \textbf{else } (f\ (\textbf{pred } (\textbf{pred } x)))\\
&\qquad\qquad)
\end{aligned}
\tag{11}
$$

♢

V izrazu (11) definiramo funkcijo Φ, ki sprejme (poljubno) funkcijo in jo potem naprej rekurzivno uporabi. Lepo vidimo, da bi za funkcijo, ki preveri, ali je število sodo morala veljati povsem algebraična enačba (12). Iščemo torej negibno točko funkcije Φ.

$$\Phi\ f = f \tag{12}$$

Za uporabo rekurzije bomo v jezik dodali konstrukt **fix**, ki bo poljubni funkciji priredil njeno negibno točko. Očitno more tak konstrukt ustrezati enačbi

$$\mathbf{fix}\ \Phi = \Phi\ (\mathbf{fix}\ \Phi) \tag{13}$$

Če to enakost uporabimo večkrat dobimo:

$$\mathbf{fix}\ \Phi = \Phi\ (\Phi\ (\ldots(\Phi\ (\mathbf{fix}\ \Phi))\ldots)) \tag{14}$$

6.2. **Negibna točka.** V prej definiran jezik tako dodamo nov konstrukt **fix**, ki to posamezni funkciji poda njeno negibno točko. Posodobljena sintaksa jezika je prikazana v tabeli 8.

$$\text{Izraz } t\ ::=\ \cdots\ \mid\ \mathbf{fix}\ t$$

Tabela 8. Sintaksa posodobljenega lambda računa

Pomembnejše od novega sintaktičnega konstrukta pa je pravilo za izvajanje, med pravila za izvajanje dodamo dve pravili za izvajanje. Prvo pove, kako se pretvori izraz podan konstruktu **fix**. Drugo, pomembnejše, pa pove, kako **fix** konstrukt deluje če je uporabljen na abstrakciji. Dodatni pravili za izvajanje sta prikazani v tabeli 9.

$$\text{E-Fix}\ \frac{t \rightsquigarrow t'}{\mathbf{fix}\ t \rightsquigarrow \mathbf{fix}\ t'} \qquad \text{E-FixBeta}\ \frac{}{\mathbf{fix}\ (\lambda x.\ t) \rightsquigarrow [x \mapsto (\mathbf{fix}\ \lambda x.\ t)]t}$$

Tabela 9. Dodatna pravila za izvajanje konstrukta **fix**

Vidimo, da se pravilo E-FixBeta intuitivno obnaša enako kot prvi poskus definicije rekurzivne funkcije v izrazu (10). Negibna točka vsako pojavitev argumenta nadomesti z negibno točko tega argumenta in s tem v jeziku omogoči sklicevanje kot smo ga hoteli.

Podati pa je potrebno tudi pravilo za dodelitev tipa takemu izrazu, to vidimo v tabeli 10.

$$\text{T-Abs}\ \frac{\Gamma \vdash t : T \to T}{\Gamma \vdash \mathbf{fix}\ t : T}$$

Tabela 10. Pravila za tipe konstrukta **fix**

Primer 6.3. S pomočjo negibne točke lahko sedaj definiramo funkcijo, ki preveri ali je število sodo. To bo sedaj negibna točka funkcije f iz izraza (11), ali drugače $jeSodo := \textbf{fix}\ \Phi$.

Poglejmo si, kako deluje rekurzija v izvajanju izraza, ki preveri, ali je število 9 sodo ($sodo\ \underline{9}$). Zapis celotnega izraza vidimo v enačbi(16). Zaradi lažjega branja in izpeljevanja v izrazu uvedemo okrajšavo za notranji del funkcije Φ, ki ga označimo z *sod*. Izraz, ki preveri, ali je število 9 sodo, lahko z okrajšavami prepišemo v izraz (15).

$$jeSodo\ 9 = \textbf{fix}\ \Phi = \textbf{fix}\ \lambda f.\ sod \tag{15}$$

$$\begin{array}{l}(\textbf{fix}\ \lambda f.\quad \lambda x.\\ \qquad\qquad \textbf{if}\ (\textbf{iszero}\ x)\ \textbf{then true}\\ \qquad\qquad\quad \textbf{else}\ (\textbf{if}\ (\textbf{iszero}\ (\textbf{pred}\ x))\ \textbf{then false}\\ \qquad\qquad\qquad \textbf{else}\ (f\ (\textbf{pred}\ (\textbf{pred}\ x)))\\ \qquad\qquad))\ \underline{9}\end{array} \tag{16}$$

Zaradi preglednosti pri izvajanju ne bomo uporabljali drevesa izvajanja. Prvi korak, ki ga izraz pri izvajanju opravi je pretvorba po pravilu E-FixBeta. Po tem pravilu, se vse pojavitve spremenljivke f v telesu funkcije Φ zamenjajo z $\textbf{fix}\ \lambda f.\ sod$. Celoten izraz se tako pretvori v

$$\begin{array}{l}\lambda x.\\ \quad \textbf{if}\ (\textbf{iszero}\ x)\ \textbf{then true}\\ \qquad \textbf{else}\ (\textbf{if}\ (\textbf{iszero}\ (\textbf{pred}\ x))\ \textbf{then false}\\ \qquad\quad \textbf{else}\ ((\textbf{fix}\ \lambda sodo'.\ sod)\ (\textbf{pred}\ (\textbf{pred}\ x)))\\ \qquad)\ \underline{9}\end{array}$$

Za tak, novo pridobljeni izraz, uporabimo pravilo E-AppAbs in vse pojavitve spremenljivke x zamenjamo z izrazom $\underline{9}$, da dobimo izraz

$$\begin{array}{l}\textbf{if}\ (\textbf{iszero}\ \underline{9})\ \textbf{then true}\\ \quad \textbf{else}\ (\textbf{if}\ (\textbf{iszero}\ (\textbf{pred}\ \underline{9}))\ \textbf{then false}\\ \qquad \textbf{else}\ ((\textbf{fix}\ \lambda sodo'.\ sod)\ (\textbf{pred}\ (\textbf{pred}\ \underline{9})))\\ \quad)\end{array}$$

Če sedaj preskočimo nekaj korakov v izvajanju pogojnih stavkov (**iszero** $\underline{9}$ se pretvori v **false**, prav tako se **iszero** (**pred** $\underline{9}$)) pretvori v **false**) se po 4 korakih izraz pretovori v

$$\textbf{if false then false else}\ (jeSodo\ (\textbf{pred}\ (\textbf{pred}\ \underline{9})))$$

Tako pridobljeni izraz se po pravilu E-If pretvori v izraz

$$jeSodo\ (\textbf{pred}\ (\textbf{pred}\ \underline{9}))$$

Vidimo, da je prvi del izraza na las podoben začetnemu izrazu (16), kar je glede na definicijo negibne točke pričakovano.

Celoten izraz se sedaj po pravilu E-FixBeta pretvori v

$\lambda x.$
if (**iszero** x) **then true**
else (**if** (**iszero** (**pred** x)) **then false**
else ($jeSodo$ (**pred** (**pred** x)))
)) (**pred** (**pred** $\underline{9}$))

Ta izraz pa se po dveh korakih in uporabi pravila E-App2 pretvori v

$\lambda x.$
if (**iszero** x) **then true**
else (**if** (**iszero** (**pred** x)) **then false**
else ($jeSodo$ (**pred** (**pred** x)))
)) $\underline{7}$

Če bi z izvajanjem nadaljevali, bi po nekaj korakih prišli do izraza

$\lambda x.$
if (**iszero** x) **then true**
else (**if** (**iszero** (**pred** x)) **then false**
else ($jeSodo$ (**pred** (**pred** x)))
)) $\underline{1}$

Ki bi se pretvoril v

if (**iszero** (**pred** $\underline{1}$)) **then false else** ($jeSodo$ (**pred** (**pred** x)))

Za ta izraz vemo, da bi po treh korakih končal z izvajanjem in se (pričakovano) pretvoril v vrednost **false**. ♢

Če bi v prejšnjem primeru namesto nenegativnega celega števila kot argument uporabili na primer izraz $\underline{-1}$, se izvajanje izraza ne bi končalo, saj niti pogoj **iszero** x, niti pogoj **iszero** (**pred** x) ne bi nikoli veljala.

Definicija 6.4. Izraz, ki nima normalne oblike, je *divergenten*.

Izrek o varnosti lahko brez težav posplošimo na lambda račun z enostavnimi tipi in negibno točko. Dokaza izrekov v večini ostaneta enaka. V koraku indukcije moramo preveriti dodatno možnost za izpeljavo tipa in obravnavati dodatna primera, kjer izraz lahko nastopa v izvajanju. Izrek o varnosti nam še vedno zagotavlja, da se tekom izvajanje izraza, ki ima tip, le-ta ne bo zataknil. Ne zagotovi pa, da bo izraz končal z izvajanjem, saj je to neodločljiv problem. To lahko preprosto vidimo v uporabi izraza $jeSod$ na izrazu $\underline{-1}$. Za izraz $jeSodo$ velja $jeSodo$: **Int** → **Bool**, za aplikacijo tega izraza na poljubnem celem število pa velja, da ima tip **Bool**. Celoten izraz pa divergira.

Slovar strokovnih izrazov

conditional (expression) pogojni stavek
derivation izpeljava
evaluation izvajanje; ∼ **tree** drevo izvajanja
fixed point negibna točka

function funkcija
lambda calculus (LC) lambda račun
normal form normalna oblika
predecessor predhodnik
preservation ohranitev
progress napredek
relation **binary** ∼ dvočlena relacija; **ternary** ∼ tričlena relacija
safety varnost
simply typed lambda calculus (STLC) lambda račun z enostavnimi tipi
small step semantics semantika malih korakov
subterm podizraz
successor naslednik
syntax sintaksa
term izraz; **closed** ∼ zaprt izraz; **stuck** ∼ zataknjen izraz
type tip
type system sistem tipov
value vrednost
variable spremenljivka; **bound** ∼ vezana spremenljivka; **free** ∼ prosta spremenljivka

Literatura

[1] J. E. Hopcroft in J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, 3rd ed, Pearson, 2006

[2] A. Nonaka, *The Y Combinator (no, not that one)*, [ogled ogled 13. 8. 2017], dostopno na `https://medium.com/@ayanonagon/the-y-combinator-no-not-that-one-7268d8d9c46`

[3] B.C. Pierce, *Types and Programming Languages*, The MIT Press, Cambridge Massachusetts, 2002.

[4] G. van Rossum, J. Lehtosalo in Ł. Langa *PEP 484 – Type Hints*, v: Python Enhancement Proposal(PEP), 29. 9. 2014, [ogled 5. 10. 2016], dostopno na `https://www.python.org/dev/peps/pep-0484/`

[5] J. Turner, *Angular 2: Built on TypeScript*, v: Microsoft Developer Tools Blog: Typescript, 5. 3. 2015, [ogled 13. 8. 2017], dostopno na `https://blogs.msdn.microsoft.com/typescript/2015/03/05/angular-2-built-on-typescript/`

[6] *@type*, v: JSDoc Documentation [ogled 13. 8. 2017], dostopno na `http://usejsdoc.org/tags-type.html`

[7] *Backus–Naur form*, v: Wikipedia: The Free Encyclopedia, [ogled 20. 10. 2016], dostopno na `https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form`.

[8] *Haskell/Fix and recursion*, v: Wikibooks: Haskell, [ogled 13. 8. 2017], dostopno na `https://en.wikibooks.org/wiki/Haskell/Fix_and_recursion`.