TIPI V PROGRAMSKIH JEZIKIH IN IZREKI O VARNOSTI PROGRAMOV

FILIP KOPRIVEC

Fakulteta za matematiko in fiziko
Univerza v Ljubljani

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

Eno izmed pomembnejših orodij v programiranju so zagotovo tipi, saj nam omogočajo, da že pred zagonom programa preprečimo določeno vrsto napak. V članku si na primeru preprostega programskega jezika najprej ogledamo pravila sintakse in izvajanja, nato pa zanj dokažemo izrek o varnosti, ki nam s pomočjo tipov zagotavlja, da bo med izvajanjem programa njegovo stanje vedno smiselno. Na kratko predstavimo tudi lambda račun z enostavnimi tipi in njemu pripadajoči izrek o varnosti.

TYPES IN PROGRAMMING LANGUAGES AND CORRESPONDING SAFETY THEOREMS

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 the program even takes place. This article presents a simple programming language which is used to examine syntax and evaluation rules, and proceeds with proving a safety theorem which uses types to ensure that the program state will stay sound during the execution. Simply typed lambda calculus and the corresponding safety theorem are presented at the end.

Uvod

Najpomembnejša lastnost večine programov je zagotovo pravilnost. Da bi zagotovili pravilnost programa, si pomagamo z mnogo različnimi pristopi in orodji, 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 utegnila privesti do napake v izvajanju.

Teoretično osnovo pri delu s tipi nam daje izrek o varnosti, ki zagotovi, da med izvajanjem programa s tipi ne bo prišlo do napake. Izrek o varnosti si bomo ogledali na preprostem programskem jeziku, na koncu pa bomo na kratko omenili, kako lahko to posplošimo tudi na funkcije.

Sintaksa

Običajno jezike podamo z različico Backus-Naurove (BNF) oblike 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 pomeni to število ($\underline{0}$ je izraz, ki pomeni število 0). Obstoječe izraze pa uporabimo pri gradnji novih izrazov. Jezik vsebuje še funkciji, ki vrneta naslednika in predhodnika poljubnega drugega izraza t, in predikat, ki pove, ali je izraz t enak izrazu, ki pomeni število 0. Vsebuje pa tudi pogojni stavek, ki je sestavljen iz treh delov: pogoja, rezultata, če je pogoj izpolnjen, in rezultata, če pogoj ni izpolnjen.

Primer 1. Intuitivno idejo jezika si je najlažje ogledati kar na primeru. Definirani jezik je sicer zelo šibak, a vseeno lahko sestavimo izraz, ki preveri, ali izraz t pomeni število 0 ali 1.

$$\mathbf{if}\ (\mathbf{iszero}\ t)\ \mathbf{then}\ \mathbf{true}\ \mathbf{else}\ (\mathbf{iszero}\ (\mathbf{pred}\ t)) \tag{1}$$

Definiramo preprost pogojni stavek, ki bo tedaj, ko je pogoj izpolnjen (če izraz t pomeni 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 pomeni število 0 ali ne.

Izvajanje

Pravila izvajanja za naš jezik so podana v tabeli 2. Njihova imena se začno s črko E (za angl. »Execution«).

Pravila podamo z dvomestno relacijo med izrazi. Pišemo $t \rightsquigarrow t'$ in beremo: t se pretvori v t'. Rečemo, da je *semantika malih korakov* najmanjša dvomestna relacija na izrazih, ki ustreza pravilom za izvajanje. Relacija zadošča pravilu za izvajanje, če za vsak primer tega pravila velja, da je bodisi izpolnjen zaključek tega pravila bodisi predpostavke niso izpolnjene. 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 služi lažjemu sklicevanju na pravilo. Sestavljena pravila, kot na primer E-Succ, razumemo tako: »Če se izraz t pretvori v t', potem lahko sklepamo, da se izraz oblike **succ** t pretvori v **succ** t'«.

Primer 2. V izraz (1) iz primera 1 vstavimo $t = \underline{0}$ in pogledamo potek izvajanja. Pri izpeljavi izvajanja si pomagamo z drevesom izvajanja, saj izraz razdelimo na manjše koščke, dokler ne pridemo do pravil za izvajanje. Za izraz (1) najprej uporabimo pravilo E-If, za katero moramo izvesti izraz **iszero** $\underline{0}$, zanj pa velja pravilo E-IsZeroZero. Začetni izraz se pretvori v **if true then true else** (**iszero** (**pred** $\underline{0}$)). Če bi zanj izvedli še en korak,

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

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

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

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

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

Tabela 2. Pravila za izvajanje.

bi se po pravilu E-IfTrue izvedel v vrednost **true**. Izpeljavi ustreza drevo spodaj:

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

Če pa se lotimo izvajanja izraza **if** $\underline{0}$ **then** $\underline{0}$ **else** (**iszero** $\underline{0}$), naletimo na težavo, saj za izvajanje izraza **if** $\underline{0}$ **then** t_1 **else** t_2 nimamo pravila.

Definicija 3 (normalna oblika). Izrazom, ki nimajo ustreznega pravila za izvajanje, rečemo, da so v *normalni obliki*.

Očitno so v normalni obliki vse konstante oblike **true**, **false**, $\underline{n}$.

Definicija 4 (vrednost). Konstantam **true**, **false** in $\underline{n}$ rečemo *vrednosti* in jih označimo s sintaktično spremenljivko v:

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

Preostali izrazi v normalni obliki so nezaželeni in jih razglasimo za napačne.

Definicija 5 (zataknjen izraz). Za izraz, ki je v normalni obliki, a ni vrednost, pravimo, da je *zataknjen*.

Primer 6. Zelo preprost zataknjen izraz je tudi **iszero true**.

Tipi

Kot smo videli, lahko zapišemo sintaktično pravilne izraze, ki se zataknejo. Zanima nas, ali lahko že vnaprej ugotovimo, kateri izrazi se med izvajanjem ne morejo zatakniti, pri tem pa si bomo pomagali s tipi.

Najprej v tabeli 3 definiramo dva osnovna tipa, celo število (**Int**) in Booleovo vrednost (**Bool**), nato v tabeli 4 predstavimo pravila za dodeljevanje tipov.

Rečemo, da ima izraz t tip T, če obstaja tak T, da je par (t, T) v relaciji *ima tip*; po pravilih sintakse pišemo $t : T$. Relacijo *imeti tip* formalno predstavimo kot najmanjšo binarno relacijo med izrazi in tipi, ki zadošča vsem pravilom za tipe.

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

$$\text{tip } T \ ::= \ \mathbf{Int} \ \Big| \ \mathbf{Bool}$$

Tabela 3. Osnovni tipi.

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

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

Tabela 4. Pravila za tipe.

Oglejmo si še pogojni stavek. Za izpeljavo tipa pogojnega stavka potrebujemo tri predpostavke: pogoj mora imeti tip **Bool**, rezultata t_2 in t_3 pa morata imeti isti tip T. Iz teh predpostavk lahko sklepamo, da ima izraz tip T (enak tipu obeh rezultatov).

Primer 7. Oglejmo si še primer izpeljave tipa nekoliko večjega izraza. Če želimo izpeljati, da ima celoten izraz pod črto tip **Int**, moramo najprej ugotoviti, kakšne tipe imajo posamezni izrazi v pogojnem stavku. V pogoju imamo izraz **iszero** $\underline{0}$, katerega tip lahko izpeljemo, če vemo, kakšen je tip $\underline{0}$. To je konstanta, po pravilu T-Num velja $\underline{0} : \mathbf{Int}$, iz česar po T-IsZero velja **iszero** $\underline{0} : \mathbf{Bool}$, kar izpolni prvi pogoj za izpeljavo tipa po pravilu T-If. Podobno izpeljemo tudi, da velja **succ** $\underline{0} : \mathbf{Int}$ in $\underline{0} : \mathbf{Int}$, potem pa uporabimo pravilo T-If in dobimo, da ima celoten izraz tip **Int**.

$$\text{T-If}\ \frac{\text{T-IsZero}\ \dfrac{\text{T-Num}\ \dfrac{}{\underline{0} : \mathbf{Int}}}{\mathbf{iszero}\ \underline{0} : \mathbf{Bool}} \qquad \text{T-Succ}\ \dfrac{\text{T-Num}\ \dfrac{}{\underline{0} : \mathbf{Int}}}{\mathbf{succ}\ \underline{0} : \mathbf{Int}} \qquad \dfrac{}{\underline{0} : \mathbf{Int}}\ \text{T-Num}}{\mathbf{if}\ (\mathbf{iszero}\ \underline{0})\ \mathbf{then}\ (\mathbf{succ}\ \underline{0})\ \mathbf{else}\ \underline{0} : \mathbf{Int}}.$$

Izreki 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. Formalno to predstavimo (in dokažemo) z dvema izrekoma: z izrekom o napredku in izrekom o ohranitvi.

Izrek 8 (izrek o napredku). *Izraz t, ki ima tip, ni zataknjen. Torej je t vrednost ali pa obstaja tak izraz t', ki ima tip in zanj velja $t \rightsquigarrow t'$.*

Dokaz. Naj bo t izraz, ki ima tip T ($t : T$). Glede na zadnje uporabljeno pravilo za izpeljavo tipa izraza bomo obravnavali več možnosti. Uporabili bomo strukturno indukcijo, ki deluje podobno kot matematična indukcija, le da na vsakem koraku predpostavimo veljavnost indukcijske predpostavke za vse prave podizraze izraza t. Pogledali bomo zgolj nekaj primerov, saj se preostale dokaže podobno.

- *Možnosti* T-True, T-False, T-Num:

  Dokaz lahko dokončamo takoj, saj so izrazi kar vrednosti.

- *Možnost* T-Pred:

  Po predpostavki izreka imamo izraz 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 $t_1 : \textbf{Int}$, je torej t_1 numerična vrednost in je oblike $\underline{n}$ za neko celo število n (v primeru $t_1 = \textbf{true}$ ali $t_1 = \textbf{false}$ bi bilo to v nasprotju z dejstvom, da velja $t_1 : \textbf{Int}$). 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:

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

  oziroma

  $$t \rightsquigarrow t',$$

  kjer je $t' = \textbf{pred}\ t_1'$.

- *Možnosti* T-Succ in T-IsZero:

  Dokaz je zelo podoben dokazu za T-Pred in ga ne bomo ponavljali.

- *Možnost* T-If:

  Po predpostavki izreka imamo izraz oblike

  $t = \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3$, za katerega velja: $t_1 : \textbf{Bool}, t_2 : T, t_3 : T$.

  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 $t_1 : \textbf{Bool}$, 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:

$$\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 \rightsquigarrow \textbf{if } t_1' \textbf{ then } t_2 \textbf{ else } t_3.$$ ∎

Izrek 9 (izrek o ohranitvi). *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 induktivno glede na izpeljavo tipov z upoštevanjem možnosti, ki nastopajo v pravilih za tipe.

- *Možnost* T-True:

  Po predpostavki izreka imamo izraz oblike **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:

  Po predpostavki izreka imamo izraz oblike $t =$ **pred** t_1, za katerega velja t_1 : **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' : **Int**. Izraz t se pretvori v **pred** t_1', za kar pa po pravilu T-Pred velja **pred** t_1' : **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:

  Po predpostavki izreka imamo izraz oblike **if** t_1 **then** t_2 **else** t_3, za katerega velja: t_1 : **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 tokrat 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 s predpostavkami, da velja $t_2 : T$ 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. ∎

Funkcije

Za preprost programski jezik smo pokazali, kako lahko s tipi zagotovimo varnost. Jezik pa bi radi razširili s funkcijami in si ogledali, kako lahko sestavimo jezik, ki je sicer znan kot *lambda račun z enostavnimi tipi*:

$$\text{izraz } t \ ::= \ \cdots \ \mid \ x \ \mid \ \lambda x.\ t \ \mid \ t_1\ t_2.$$

Sintakso jezika dopolnimo, da omogoča definicijo in uporabo funkcij. Izraz je sedaj lahko poljuben že prej veljaven izraz, lahko pa je tudi spremenljivka. Še pomembnejši pa sta zadnji dve možnosti. S prvo definiramo funkcijo (abstrakcijo), ki spremenljivki x priredi izraz t (tu pravimo, da je x *vezan* v tej abstrakciji), z drugo pa uporabo prvega izraza (funkcije) na drugem izrazu. Vzemimo npr. izraz

$$\lambda x.\ \mathbf{iszero}\ x. \tag{2}$$

V izrazu (2) definiramo preprosto abstrakcijo, ki spremenljivki x priredi izraz $\mathbf{iszero}\ x$. Ideja je popolnoma enaka definiciji funkcije v matematiki, le da uporabimo nekoliko drugačen zapis. Enako je z uporabo funkcije na izrazu.

Primer 10.

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

V izrazu (3) abstrakcijo iz izraza (2) uporabimo na izrazu, ki pomeni število 0. V izvajanju bi, kot pri običajni funkciji, vse nastope x v prvotnem izrazu zamenjali z $\underline{0}$ in dobili izraz $\mathbf{iszero}\ \underline{0}$. Substitucijo spremenjivke 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.

Funkcije pa lahko vračajo tudi nove funkcije. Izraz (4) dani funkciji priredi novo funkcijo, ki prvo funkcijo uporabi dvakrat:

$$\lambda x_1.\ (\lambda x_2.\ x_1\ (x_1\ x_2)). \tag{4}$$

Pravilom za izvajanje zato dodamo še tri pravila: prvo pove, kako se pretvori prvi izraz, ki nastopa v uporabi funkcije; drugo, kako se pretvori argument abstrakcije; tretje pa, kako uporabimo abstrakcijo. Ustrezno posodobimo tudi vrednosti, ki jim dodamo še abstrakcijo:

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

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

Tabela 5. Pravila za izvajanje lambda računa.

Kot pri prejšnjem jeziku bi tudi tu radi s pomočjo tipov prišli do izreka o varnosti. Da to storimo, moramo tipe funkcij najprej definirati. Tako kot v matematiki funkcijo karakteriziramo z domeno (tipom argumenta) in kodomeno (tipom rezultata).

Poglejmo abstrakcijo iz izraza (2): $\lambda x.$ (**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**. Ustaljenim tipom dodamo še tip funkcije, ki vsebuje tip argumenta in tip rezultata:

$$\text{tip } T\ ::=\ \cdots\ \big|\ T_1 \to T_2.$$

Za splošno funkcijo $\lambda x.\ t$ torej ob predpostavki $x : T_1$ velja $t : T_2$. Drugače od prej pa 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. Formalno relacijo *imeti tip* razširimo na tri elemente: kontekst, izraz in tip: (Γ, t, T) in pišemo $\Gamma \vdash t : T$.

Z dodatnimi pravili za tipe, ki jih vidimo v tabeli 6, 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

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

Tabela 6. Pravila za tipe lambda računa z enostavnimi tipi.

takole: Če v kontekstu Γ, razširjenem s predpostavko $x : T_1$, velja $t_2 : T_2$, potem lahko sklenemo, da ima funkcija $\lambda x.\ t_2$ v kontekstu Γ tip $T_1 \to T_2$. Pravilo T-App pa 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 11. S pomočjo izraza iz primera 1 sestavimo funkcijo, ki preveri, ali je argument funkcije enak izrazu za 0 ali 1:

$$\lambda x.\ \textbf{if}\ (\textbf{iszero}\ x)\ \textbf{then true else}\ (\textbf{iszero}\ (\textbf{pred}\ x)).$$

Poglejmo še izpeljavo tipa:

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

Izrek o varnosti lahko posplošimo tudi na lambda račun z enostavnimi tipi. Izrek enako kot prej zagotovi, da se izraz, ki ima tip, med izvajanjem ne bo zataknil. Dokažemo ga s strukturno indukcijo, le da je treba obravnavati več primerov zaradi dodanih pravil za izpeljavo tipov. Tipi nam tudi v lambda računu zagotavljajo varnost. Pomembno dejstvo je, da nam izrek o varnosti jamči zgolj to, da je izraz, ki ima tip, že končal z izvajanjem, ali pa lahko naredi še (vsaj) en korak, nikakor pa nam ne zagotavlja, da se bo izvajanje končalo. V lambda računu z enostavnimi tipi sicer vsi izrazi s tipom končajo z izvajanjem, medtem ko to za druge programske jezike v splošnem ne velja.

LITERATURA

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

[2] P. Wadler, *Propositions as Types*, dostopno na `homepages.inf.ed.ac.uk/wadler/papers/propositions-as-types/propositions-as-types.pdf`, ogled 5. 11. 2016.