Poglavje 11

λ-RAČUN 2. REDA

11.1 Rekonstrukcija tipov

Algoritmi za preverjanje tipov, ki smo si jih ogledali do sedaj so odvisni od eksplicitne anotacije tipov v programih. V tej sekciji bo predstavljen algoritem za preverjanje tipov, ki zna izračunati principalen* tip izraza, ki vsebuje samo nekatere anotacije tipov.

V izrazih bomo uporabljali spremenljivke, katerih zaloga vrednosti so tipi. Iz stališča označevanja izrazov jezika s tipi dobimo torej *jezik 2. reda*: izrazi lahko manipulirajo objekte, ki predstavljajo tipe–objekte 2. reda.

Pri preučevanju jezikov 2. reda se bomo lahko spraševali glede lastnosti kot je npr. naslednja: "če instanciramo vrzel X v izrazu t s konkretnim tipom Bool, ali dobimo izraz z enoličnim tipom?"

11.2 Spremenljivke tipov

Pri predstavitvi nekaterih jezikov opisanih v prejšnjih poglavjih smo predpostavili obstoj neskončne množice *neiterpretiranih osnovnih tipov*.

Te tipi nimajo definiranih operacij kot jih imajo npr. konkretni osnovni tipi Bool in Nat. Največkrat služijo samo kot vrzeli, ki jih nadomestijo konkretni tipi, katerih identiteta nas ne zanima.

Neinterpretirane tipe bomo imenoval *spremenljivke tipa*—spremenljivke, ki v izrazih služijo namesto tipov.

Spremenljivke tipa je mogoče *instancirati* tako, da jih zamenjamo s konkretnimi tipi.

Instanciranje bo sestavljeno iz dveh delov: definicije substitucije σ in apliciranje substitucije na izrazu.

Na primer, substitucija je definirana z izrazom $\sigma = [X/\text{Bool}]$ in aplikacija σ na tipu $X \to X$ da izraz $\sigma(X \to X) = \text{Bool} \to \text{Bool}$.

Opazka 11.2.1. *Substitucija tipov je končna preslikava iz množice izrazov tipov v izraze tipov.*

Substitucija spremenljivk tipov, ki se nahajajo v sestavljenih objektih se realizira tako, da se funkcija σ aplicira na vseh komponentah izraza tipa.

$$\sigma(x_1 : T_1, \ldots, x_n : T_n) = (x_1 : \sigma T_1, \ldots, x_n : \sigma T_n)$$

Substitucija lahko tvori kompozicijo dveh substitucij. Recimo, da sta σ in γ substituciji, potem je substitucija tudi $\sigma \circ \gamma$.

$$(\sigma \circ \gamma)S = \sigma(\gamma S)$$

□

Najbolj pomembna lastnost substitucije je ohranjanje dobro definiranih tipov: če ima izraz dobro definirane tipe, potem so dobro definirani tudi po substituciji.

Izrek 11.2.1 (Ohranitev tipov po substituciji)**.** *Naj bo σ substitucijska funkcija in naj $\Gamma \vdash t : T$, potem velja tudi $\Gamma \vdash \sigma t : \sigma T$.*

Dokaz. Indukcija direktno po izpjeljavi tipov oz. po pravilih, ki definirajo tipe izrazov. □

11.2.1 Izpeljava tipov

Naj bo t izraz, ki vsebuje spremenljivke tipa in Γ kontekst, ki tudi vsebuje spremenljivke tipa. Dve vprašanji sta pomembni pri delu z izrazi, ki vsebujejo spremenljivke tipa:

- Imajo vsi možni primerki substitucije dobro definirane tipe? Drugače zapisano: ali velja za vsak σ da $\sigma\Gamma \vdash \sigma t : T$ za nek T?
- Obstaja substitucija tako, da ima t dobro definirane tipe? Drugače zapisano: ali lahko najdemo σ tako da $\sigma\Gamma \vdash \sigma t : T$ za nek T?

V primeru, da imamo takšne izraze, da je vsaka substitucija dobro definirana potem lahko sklepamo s tipi ter obdržimo abstraktno predstavitev tipov med preverjanjem tipov programa.

Primer 11.2.1. *Izraz*

$$\lambda f : X \to X.\lambda a : X.f(f\ a)$$

ima tip $(X \to X) \to X \to X$. *Vedno ko* X *zamenjamo s konkretno instanco dobimo izraz*

$$\lambda f : T \to T.\lambda a : T.f(f\ a)$$

z dobro definiranimi tipi. □

Obravnavanje abstraktnih spremenljivk tipa vodi do *parametričnega polimorfizma*, kjer uporabimo spremenljivke tipa zato, da lahko definiramo parametrične programe uporabne v različnih kontekstih in z različnimi konkretnimi tipi.

Če pogledamo iz vidika drugega vprašanja potem t sploh ni nujno, da ima t dobro definirane tipe. Kar želimo vedeti je ali obstaja takšna substitucija σ, da ima σt dobro definirane tipe.

Primer 11.2.2. *Izraz*

$$\lambda f : Y.\lambda a : X.f(f\ a)$$

nima konsistentnih tipov. Če pa Y *zamenjamo z Nat* $\to$ *Nat in* X *z Nat dobimo izraz*

$$\lambda f : Nat \to Nat.\lambda a : Nat.f(f\ a)$$

z dobro definiranim tipom $(Nat \to Nat) \to Nat \to Nat$. *Druga možnost za zamenjavo tipa* Y *je tip* $X \to X$.

$$\lambda f : X \to X.\lambda a : X.f(f\ a)$$

Dobili smo najbolj splošen izraz z dobro definiranimi tipi za izraz $\lambda f : Y.\lambda a : X.f(f\ a)$ *z omejitvami* X *in* Y. □

Iskanje veljavnih instanc spremenljivk tipov za dan izraz imenujemo *rekonstrukcija tipov* ali tudi *izpeljava tipov*.

Prevajalniki vsebujejo kodo za izpeljavo tipov izrazov, ki jih ni določil programer. V limiti, prevajalnik lahko izpelje tipe vsakega izraza programa in tako razbremeni programerja.

Med razčlenjevanjem programa prevajalnik izračuna najbolj splošen tip za vsak izraz v razčlenjevalnem devesu.

1) Najprej vsem spremenljivkam λ-abstrakcije doda spremenljivko tipa tako, da so unikatne za različne abstrakcije.

2) Potem se izvede rekonstrukcija tipov tako, da poiščemo najbolj splošne tipe spremenljivk.

Za bolj natančno formalizacijo rakonstrukcije tipov bomo potrebovali bolj natančen opis možnih zamenjav spremenljivk tipa s tipi za to, da bi dobili veljaven izraz tipov.

Definicija 11.2.1. *Naj bo* Γ *kontekst in naj bo* t *izraz. Rašitev* (Γ, t) *je par* (σ, T)*, tako da velja* $\sigma\Gamma \vdash \sigma t : T$.

Primer 11.2.3. *Naj bo* $\Gamma = f : X, a : Y$ *in* $t = f\ a$*. Rešitve* (Γ, t) *so:*

$([X/Y \to Nat], Nat)$ $([X/Y \to Z], Z)$
$([X/Y \to Z, Z/Nat], Z)$ $([X/Y \to Nat \to Nat], Nat \to Nat])$
$([X/Nat \to Nat, Y/Nat], Nat)$

□

11.3 Univerzalni tipi

Recimo, da bi želeli napisati funkcijo, ki podvoji klic argumenta funkcije nad danim parametrom, za različne tipe parametra.

$$
\begin{aligned}
doubleNat &= \lambda f : \text{nat} \to \text{nat}.\lambda x : \text{Nat}.f(f(x)) \\
doubleRcd &= \lambda f : \{l : \text{Bool}\} \to \{l : \text{Bool}\}.\lambda x : \{l : \text{Bool}\}.f(f(x)) \\
doubleFun &= \lambda f : (\text{Nat} \to \text{Nat}) \to (\text{Nat} \to \text{Nat}).\lambda x : (\text{Nat} \to \text{Nat}).f(f(x))
\end{aligned}
\tag{11.1}
$$

Vsaka od funkcij je definirana nad različnim parametrom, sicer pa vse izvedejo podobno kodo. Edina razlika med kodo različnih funkcij je v tipu parametrov.

Takšno pisanje programov krši osnovno pravilo programiranja: koda za določeno funkcijo se implementira samo enkrat.

Sprejemljivi del programa v prejšnjem primeru je tip parametrov. To kar potrebujemo je mehanizem za abstrakcijo tipov iz izrazov in nato instanciranje izraza s konkretnimi tipi.

11.3.1 Sistem F

Sistem, ki ga bomo uporabljali v tem poglavju je *sistem F*. Sistem je odkril Jean-Yves Girard leta 1972 v okviru avtomatskega dokazovanja v logiki. Malce kasneje, leta 1974, je John Reynold definiral *polimorfični lambda račun*, ki ima zelo podobno izrazno moč.

Sistem F je bil uporabljan kot raziskovalna platforma za delo na polimorfizmu in za osnovo številnih novih programskih jezikov.

Sistem F velikokrat imenujemo *λ-račun 2. reda*. Eden izmed razlogov za to je Curry-Howardov izomorfizem, ki povezuje Sistem F z logiko 2. reda, ki dovoljuje kvantifikacijo ne samo individualnih spremenljivk ampak tudi spremenljivk tipov.

Sistem F je razširitev jezika $\lambda_{\rightarrow}$, lambda računa s tipi. Ker želimo abstrahirati tipe iz izrazov in jih postaviti kasneje, sta uvedeni nova abstrakcija in nova aplikacija.

Uporabimo *abstrakcijo $\lambda X.t$*, kjer je X spremenljivka tipa in t tip. Definirati moramo tudi novo *aplikacijo $t\ [T]$* s katero apliciramo tip T na izrazu t.

Novo abstrakcijo imenujemo *abstrakcija tipov* in novo aplikacijo imenujemo *aplikacijo tipov* ali *instanciranje*.

Evaluacija aplikacije abstrakcije tipa na konkretnem tipu je definirana z naslednjim pravilom.

$$(\lambda X.t)\ [T] \rightarrow [X/T]t$$

Pravilo je analogno pravilu za lambda abstrakcijo, ki aplicira abstrakcijo na vrednost.

$$(\lambda x.t)\ v \rightarrow [x/v]t$$

Primer 11.3.1. *Poglejmo si primer polimorfične funkcije identitete.*

$$id = \lambda X.\lambda x\colon X.x$$

Če zdaj apliciramo tip Nat na izrazu id, kar zapišemo $id\ [Nat]$, dobimo $[Nat/X]\lambda x\colon X.x$, ki se ovrednoti v $\lambda x\colon Nat.x$, funkcijo identitete nad naravnimi števili. □

Poleg nove abstrakcije in aplikacije potrebujemo zdaj tudi nov tip s katerim bomo lahko klasificirali polimorfične funkcije kot je npr. prej predstavljena funkcija id.

V primeru funkcije id lahko podamo kot parameter tip T in dobimo izraz tipa $T \rightarrow T$. Tako dobljen tip je torej odvisen od tipa T, ki ga podamo kot parameter.

V splošnem lahko v primeru funkcije id uporabimo tip $\forall X.X \rightarrow X$. Kvantifikator $\forall$ zdaj določa tip X: za vsak tip X velja $X \rightarrow X$.

Dobimo torej izraze, kjer je spremenljivka tipa X kvantificirana *univerzalno*: izraz za kvantificirano spremenljivko določa *parametrični tip*, tip katerega vrednost je odvisna od parametra—spremenljivke tipa.

Univerzalni (parametrični) tip bomo v splošnem označevali $\forall X.T$, kjer je T izraz, ki vsebuje spremenljivko tipa X.

Zdaj lahko definiramo tip abstrakcije in aplikacije tipov v obliki pravil, ki so podobna pravilom za abstrakcijo in aplikacijo v običajnem λ-računu.

$$\frac{\Gamma, X \vdash t\colon T}{\Gamma \vdash \lambda X.t\colon \forall X.T} \tag{11.2}$$

$$\frac{\Gamma \vdash t\colon \forall X.T}{\Gamma \vdash t\ [T_1] : [X/T_1]T} \tag{11.3}$$

Sintaksa Sistema F je definirana z naslednjo slovnico.

$$
\begin{array}{lll}
t & ::= & \text{izrazi} \\
 & x & \text{spremenljivka} \\
 & \lambda x\colon T.t & \text{abstrakcija} \\
 & t\ t & \text{aplikacija} \\
 & \lambda X : T.t & \text{abstrakcija tipa} \\
 & t\ [T] & \text{aplikacija tipa} \\
v & ::= & \text{vrednosti} \\
 & \lambda X : T.t & \text{vrednost abstrakcije} \\
 & \lambda X.T & \text{vrednost abstrakcije tipa} \\
T & ::= & \text{tipi} \\
 & X & \text{spremenljivka tipa} \\
 & T \to T & \text{tip funkcije} \\
 & \forall X.T & \text{univerzalni tip} \\
\Gamma & ::= & \text{konteksti} \\
 & \emptyset & \text{prazen kontekst tipa} \\
 & \Gamma, x\colon T & \text{vezava spremenljivke} \\
 & \Gamma, X & \text{vezava spremenljivke tipa}
\end{array}
\tag{11.4}
$$

Pravila s katerimi definiramo evaluacijo izrazov Sistema F so podobna pravilom za evaluacijo jezika $\lambda_{\to}$.

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2} \tag{11.5}$$

$$\frac{t_2 \to t_2'}{v_1\ t_2 \to v_1\ t_2'} \tag{11.6}$$

$$\frac{}{(\lambda x\colon T.t)\ v \to [x/v]t} \tag{11.7}$$

$$\frac{t \to t'}{t\ [T] \to t'\ [T]} \tag{11.8}$$

$$\frac{}{(\lambda X.t)\ [T] \to [X/T]t} \tag{11.9}$$

Naslednja pravila definirajo prirejanje tipov izrazov Sistema F.

$$\frac{x\colon T \in \Gamma}{\Gamma \vdash x\colon T} \tag{11.10}$$

$$\frac{\Gamma, x\colon T_1 \vdash t_2\colon T_2}{\Gamma \vdash \lambda x\colon T_1.t_2 : T_1 \to T_2} \tag{11.11}$$

$$\frac{\Gamma \vdash t_1\colon T_1 \to T_2 \quad \Gamma \vdash t_2\colon T_1}{\Gamma \vdash t_1\ t_2\colon T_2} \tag{11.12}$$

$$\frac{\Gamma, X \vdash t\colon T}{\Gamma \vdash \lambda X.t\colon \forall X.T} \tag{11.13}$$

$$\frac{\Gamma \vdash t_1 : \forall X.T_1}{\Gamma \vdash t_1\ [T_2] : [X/T_2]\ T_1} \tag{11.14}$$

11.3.2 Primeri

V tej sekciji bo predstavljeno nekaj primerov programov zapisanih s Sistemom F.

Primer 11.3.2. *Poglejmo še enkrat definicijo polimorfične verzije funkcije* id*.*

$$\begin{array}{ll} & id = \lambda X.\lambda x : X.x \\ \blacktriangleright & id : \forall X.X \to X \end{array} \tag{11.15}$$

Funkcijo je potrebno instancirati, da bi dobili konkretno funkcijo id*.*

$$\begin{array}{ll} & id\ [Nat]; \\ \blacktriangleright & \langle fun \rangle : Nat \to Nat \end{array} \tag{11.16}$$

□

Primer 11.3.3. *Naslednji primer realizira polimorfično funkcijo, ki podvoji klic parametra.*

$$\begin{array}{ll} & double = \lambda X.\lambda f : X \to X.\lambda a : X.f(fa) \\ \blacktriangleright & double : \forall X.(X \to X) \to X \to X \end{array} \tag{11.17}$$

Zdaj lahko dobimo specifičen tip za različne vrednosti spremeljivke tipa X*.*

$$\begin{array}{ll} & doubleNat = double\ [Nat]; \\ \blacktriangleright & doubleNat : (Nat \to Nat) \to Nat \to Nat \\ & doubleNtoN = double\ [Nat \to Nat]; \\ \blacktriangleright & doubleNtoN : ((Nat \to Nat) \to Nat \to Nat) \to (Nat \to Nat) \to (Nat \to Nat) \end{array} \tag{11.18}$$

Ko je funkcija instancirana s tipom ji lahko podamo še dva parametra: funkcijo, ki jo $double$ *podvoji in začetno vrednost te funkcije.*

$$\begin{array}{ll} & double\ [Nat]\ (\lambda x : Nat.succ(succ(x)))\ 3; \\ \blacktriangleright & 7 : Nat \end{array} \tag{11.19}$$

□

Primer 11.3.4. *Poglejmo zdaj bolj zvit primer polimorafične aplikacije funkcije same nase. Če se spomnimo primera, ki smo ga obdelali pri lambda računu s tipi: ni načina za prireditev tipa izrazu* $\lambda x.x\ x$ *v okviru* $\lambda_{\to}$*. V Sistemu F postane izraz obvladljiv, če spremenljivki* x *priredimo polimorfičen tip.*

$$\begin{array}{ll} & selfApp = \lambda x : \forall X.X \to X.x\ [\forall X.X \to X]\ x; \\ \blacktriangleright & selfApp : \forall X.(X \to X) \to (\forall X.X \to X) \end{array} \tag{11.20}$$

Malce bolj koristen izraz dobimo z aplikacijo polimorfične funkcije $double$ *nad samo sebe s čimer dobimo funkcijo* $quadruple$*.*

$$quadruple = \lambda X.\, double\, [X \to X]\, (double\, [X]) \tag{11.21}$$

□

Večina programov, ki uporablja polimorfizem je veliko bolj običajnih kot prejšnji primeri. Kot primer precej naravne uporabe polimorfizma si bomo ogledali implementacijo seznamov.

Primer 11.3.5. *Recimo, da je naš programski jezik opremljen z običajnim konstruktorjem tipov List in z običajnimi primitivi za delo s seznami.*

$$\begin{array}{l}
\blacktriangleright\ \ \mathit{nil} : \forall X.\mathit{List}\ X \\
\quad\ \mathit{cons} : \forall X.X \to \mathit{List}\ X \to \mathit{List}\ X \\
\quad\ \mathit{isnil} : \forall X.\mathit{List}\ X \to \mathit{Bool} \\
\quad\ \mathit{head} : \forall X.\mathit{List}\ X \to X \\
\quad\ \mathit{tail} : \forall X.\mathit{List}\ X \to \mathit{List}\ X
\end{array} \tag{11.22}$$

$$\begin{array}{l}
map = \lambda X.\lambda Y. \\
\qquad \lambda f : X \to Y. \\
\qquad (\mathit{fix}\ (\lambda m : (\mathit{List}\ X) \to (\mathit{List}\ Y). \\
\qquad\quad \lambda l : \mathit{List}\ X. \\
\qquad\quad \mathit{if\ isnil}\ [X]\ l \\
\qquad\quad \mathit{then\ nil}\ [Y] \\
\qquad\quad \mathit{else\ cons}\ [Y]\ (f\ \mathit{head}\ [X]\ l)) \\
\qquad\qquad\qquad\qquad (m\ \mathit{tail}\ [X]\ l)))); \\
\blacktriangleright\ \ map : \forall X.\forall Y.(X \to Y) \to \mathit{List}\ X \to \mathit{List}\ Y \\
\quad\ l = \mathit{cons}\ [\mathit{Nat}]\ 4\ (\mathit{cons}\ [\mathit{Nat}]\ 3\ (\mathit{cons}\ [\mathit{Nat}]\ 2\ (\mathit{nil}\ [\mathit{Nat}]))); \\
\blacktriangleright\ \ l : \mathit{List\ Nat} \\
\quad\ \mathit{head}\ [\mathit{Nat}]\ (map\ [\mathit{Nat}]\ [\mathit{Nat}]\ (\lambda x : \mathit{Nat}, \mathit{succ}\ x)\ l); \\
\blacktriangleright\ \ 5 : \mathit{Nat}
\end{array} \tag{11.23}$$

□

11.4 Opombe

Poglavje vsebuje prevode izbranih sekcij učbenikov Benjamin Pierce *Types and programming languages* [15] ter Roberta Harperja *Practical foundations for programming languages* [7].