Poglavje 9

STRUKTURE

Do zdaj smo si pogledali osnovne formalizme, ki so uporabljeni za predstavitev programskih jezikov. Definirali smo osnovno infrastrukturo za opisovanje in sklepanje z izrazi oz. programi.

Predstavljen je bil osnovni formalni jezik λ-račun, ki je enakovreden Turingovem stroju. Tipi so uporabljeni za klasifikacijo izrazov in vrednosti.

V tem poglavju si bomo pogledali strukture, ki jih uporabljamo za opisovanje strukturiranih gradnikov programskih jezikov ter njihovo formalno obravnavo.

Najprej bomo obravnavali osnovne strukture, ki služijo za definicijo osnovnih tipov in konstruktov programskih jezikov. Med te sodijo tip Unit, sekvence, jokerji, označevanje s tipi in stavek let.

Podrobno si bomo ogledali formalno obravnavo osnovnih struktur v kontekstu že spoznanih teoretičnih osnov.

Sledi predstavitev osnovnih struktur za definicijo sestavljenih objektov programskih jezikov: pari, n-terice, zapisi, unije in nekatere variacije naštetih.

Za vsakega od predstavljenih gradnikov bomo definirali pravila, ki predstavljajo statično strukturo s tipi ter dinamično obnašanje struktur preko evaluacije izrazov.

9.1 Osnovne strukture

9.1.1 Osnovni tipi

Vsak jezik ima množico osnovnih tipov - množico enostavnih vrednosti: cela števila, realna števila, nizi, itd. Vsakemu osnovenemu tipu pripada množica operacij s katerimi lahko delamo z vrednostmi osnovnega tipa.

Včasih ne želimo videti podrobnosti o enostavnih tipih in operacijah nad njimi. V teh primerih bomo uporabljali velike črke $A, B, C,$ itd. kot imena za osnovne tipe.

Poglejmo si nekaj primerov.

Primer 9.1.1. *Izraz $\lambda x : A.x$ definira funkcijo identitete $\langle fun \rangle : A \to A$. Podobno je tudi $\lambda x : B.x$ identiteto $\langle fun \rangle : B \to B$, medtem ko je $\lambda f : A \to A.\lambda x : A.f(f(x))$ funkcija $\langle fun \rangle : (A \to A) \to A \to A$, ki dvakrat ponovi aplikacijo funkcije f na x.*

9.1.2 Tip Unit

Tip, ki je pogosto uporabljen v funkcijskih jezikih kot je npr. ML, je enostaven tip Unit. Tip Unit predstavlja eno samo konstanto unit. Z drugimi besedami interpretacija tipa Unit je konstanta unit.

Pogosto se uporablja tudi pri formalni analizi programskih jezikov s stranskimi učinki. V tem primeru predstavlja velikokrat stranski učinek in ne rezultat izraza, ki ga preučujemo.

Tudi v čistih funkcijskih jezikih je tip Unit lahko koristen. Kasneje si bomo ogledal par primerov.

Tip Unit se uporablja na podoben način kot tip $void$ v programskem jeziku C.

9.1.3 Sekvence

V programskih jezikih s stranskimi učinki zelo pogosto ovrednotimo več stavkov v sekvenci. Notacija za označevanje sekvenc je $t_1; t_2$ pomeni: ovrednoti stavek t_1, odvrzi trivialni rezultat in potem ovrednoti še t_2.

Imamo dva načina formalizacije sekvenc. Prvi način je dodajanje nove sintaktične oblike in novih pravil za definicijo gradnikov. Pravili za evaluacijo sekvence sta naslednji.

$$\frac{t_1 \to t_1'}{t_1; t_2 \to t_1'; t_2} \qquad \text{(E-Seq)}$$

$$\frac{}{unit; t_2 \to t_2} \qquad \text{(E-SeqNext)}$$

Pravilo za definicijo tipov ali statične semantike sekvence je definirano takole.

$$\frac{\Gamma \vdash t_1 : \text{Unit} \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1; t_2 : T_2} \qquad \text{(T-Seq)}$$

Alternativni način formalizacije sekvence je preprosto gledanje na $t_1; t_2$ kot na okrajšavo $(\lambda x : \text{Unit}.t_2)\ t_1$, kjer je x različna od vse ostalih spremenljivk v t_2.

Kar se tiče programerja sta obe definiciji ekvivalentni; pravila za tipe in evaluacijo lahko izpeljemo iz okrajšave $t_1; t_2$ z $(\lambda x : \text{Unit}.t_2)\ t_1$.

Lema 9.1.1. *Jezik, ki ga dobimo iz λ računa, tipa Unit in pravil za delo s sekvencami imenujmo λ^E. Osnovni λ račun s tipom Unit pa bomo imenovali λ^I. Naj bo $e \in \lambda^E \to \lambda^I$ funkcija, ki prevaja izraze iz λ^E v λ^I tako, da zamenjuje vse pojavitve $t_1; t_2$ z $(\lambda x : Unit.t_2)\ t_1$, kjer je x enolična znotraj t_2.*

1. $t \to_E t' \Leftrightarrow e(t) \to_I e(t')$
2. $\Gamma \vdash^E t : T \Leftrightarrow \Gamma \vdash^I e(t') : T$

Dokaz. Po indukciji na konstrukcijo izrazov. □

Lema 9.1.1 opravičuje imenovanje sekvenc stavkov *izpeljana forma*, ker pokaže, da lahko obravnavamo sekvence stavkov kot λ-abstrakcijo in aplikacijo.

Prednost uvajanja novih gradnikov (form) je v razširjanju *površinske* sintakse programskega jezika, medtem ko ostaja interni jezik nespremenjen.

Ta način dodajanja gradnikov programskih jezikov se je uporabljal že pri Algolu-60 in se uporablja tudi dandanes na primer pri definiciji Standard ML.

9.1.4 Stavek let

Ko opisujemo kompleksen izraz je pogosto koristno imenovati podizraze zaradi izboljšanja berljivosti kot tudi zaradi ponavljanja izrazov. Večina jezikov omogoča to na en ali drug način.

V imperativnih programskih jezikih lahko definiramo lokalne spremenljivke, ki hranijo podizraze in vrednosti. V funkcijskih jezikih imamo običajno na razpolago stavek let s katerim lahko izraz razdelimo na več imenovanih pod-izrazov. Tukaj bomo predstavili let stavek kot je definiran v ML.

Sintaksa let stavka je definirana na sledeč način.

$$\text{Izraz } t \ ::= \ \text{let } x = t_1 \text{ in } t_2$$

Stavek let beremo: naj bo vrednost x enaka t_1 v izrazu t_2.

Statična semantika stavka je definirana z naslednjim pravilom.

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2}$$

Tip let stavka izračunamo tako, da izpeljemo tipa pod-izrazov t_1 in t_2. Pri izpeljavi tipa t_2 upoštevamo, da vsebuje spremenljivko, ki je tipa T_1. Le-ta je enak tipu t_1.

Evaluacija stavka let je definirana z naslednjimi pravili, ki definirajo semantiko "klic-po-vrednosti". Izraz t_1 mora biti ovrednoten najprej, šele nato se lahko prenese znotraj t_2.

$$\frac{}{\text{let } x = v_1 \text{ in } t_2 \to [x/v_1]t_2}$$

$$\frac{t_1 \to t_1'}{\text{let } x = t_1 \text{ in } t_2 \to \text{let } x = t_1' \text{ in } t_2}$$

Prvo pravilo je zelo podobno evaluaciji λ-redeksa. Spremenljivki x priredimo vrednost v_1—drugo pravilo omogoča ovrednotenje t_1 do vrednosti.

Landin je pokazal, da se let stavek da izraziti z λ-računom. Enostnavna rešitev je z uporabo abstrakcije in aplikacije.

$$\text{let } x = t_1 \text{ in } t_2 \equiv_{def} (\lambda x : T_1.t_2)\ t_1$$

Če primerjamo izraza na levi in desni vidimo, da imamo na desni strani tip T_1 medtem, ko ga na levi stani ni. Moramo torej imeti tip T_1, ki ga bodisi napiše programer ali pa ga izpelje program za preverjanje tipov.

Podrobnosti glede zamenjave let stavka z λ računom je več; tukaj jih bomo izpustili (glej Piercev učbenik).

9.2 Produkti

Večina programskih jezikov vsebuje več gradnikov za konstrukcijo sestavljenih tipov. Najenostavnejši med njimi so *pari*, bolj kompleksni so *produkti* in *zapisi*.

Binarni produkt je sestavljen iz urejenih parov vrednosti, kjer ima vsaka vrednost tip glede na vrstni red komponent para. Vrednosti lahko izluščimo z uporabo projekcije, ki izbere eno izmed komponent para.

Ničen produkt ali enota je sestavljen iz unikatnega para ”null par”, ki nima komponent.

Bolj splošno pogledano lahko definiramo n-arni produkt tipov, ki ga sestavljajo urejene n-terice vrednosti. Operacije projekcije uporabljamo za izluščenje i-te komponente iz n-terice.

Označeni produkt ali zapis tipov sestavlja označene n-terice, kjer imajo komponente oznake. Projekcija komponent je definirana na osnovi oznak.

Objektni tip je zapis samo-referenčnih funkcij, ki jih imenujemo metode. Samo-referenca je implementirana z izbiro metode, ki priključi metodi objekt preden da kontrolo telesu metode.

9.2.1 Nični in binarni produkt

Abstraktna sintaksa binarnega produkta je definirana s sledečo gramatiko.

$Kategorija$	Ime		$Abstraktno$	$Konkretno$
Tip	τ	::=	Unit	Unit
		|	prod(τ_1, τ_2)	$\tau_1 \times \tau_2$
$Izraz$	e	::=	$unit$	$\{\}$
		|	pair(e_1, e_2)	$\{e_1, e_2\}$
		|	fst(e)	$e.1$
		|	snd(e)	$e.2$

(9.1)

Tip prod(τ_1, τ_2) imenujemo tudi binarni produkt tipov τ_1 in τ_2. Tip Unit imenujemo tudi nični produkt. Tipi, ki jih dobimo s produktom vključujejo tudi nični in binarni produkt.

Statična semantika prudukta je definirana s sledečimi pravili. Uporabljali bomo abstraktno sintakso.

$$\frac{}{\Gamma \vdash unit : \text{Unit}} \tag{9.2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \text{pair}(e_1, e_2) : \text{prod}(\tau_1, \tau_2)} \tag{9.3}$$

$$\frac{\Gamma \vdash e : \text{prod}(\tau_1, \tau_2)}{\Gamma \vdash \text{fst}(e) : \tau_1} \tag{9.4}$$

$$\frac{\Gamma \vdash e : \text{prod}(\tau_1, \tau_2)}{\Gamma \vdash \text{snd}(e) : \tau_2} \tag{9.5}$$

Pravilo 9.2 definira prazen par $unit$, ki je tipa Unit. Pravilo 9.3 predstavi konstrukcijo para iz komponent in tipov komponent. Pravili 9.4 in 9.5 delujeta v obratno smer. Iz para projecirata komponento in tip.

Dinamična semantika parov je definirana na sledeč način.

$$\frac{}{unit \text{ val}} \tag{9.6}$$

$$\frac{\{e_1 \text{ val} \quad e_2 \text{ val}\}}{\text{fst}(\text{pair}(e_1, e_2)) \mapsto e_1} \tag{9.7}$$

$$\frac{\{e_1 \text{ val} \quad e_2 \text{ val}\}}{\text{snd}(\text{pair}(e_1, e_2)) \mapsto e_2} \tag{9.8}$$

$$\frac{e \mapsto e'}{\text{fst}(e) \mapsto \text{fst}(e')} \tag{9.9}$$

$$\frac{e \mapsto e'}{\text{snd}(e) \mapsto \text{snd}(e')} \tag{9.10}$$

$$\left\{ \frac{e_1 \mapsto e_1'}{\text{pair}(e_1, e_2) \mapsto \text{pair}(e_1', e_2)} \right\} \tag{9.11}$$

$$\left\{ \frac{e_1 \text{ val} \quad e_2 \mapsto e_2'}{\text{pair}(e_1, e_2) \mapsto \text{pair}(e_1, e_2')} \right\} \tag{9.12}$$

Pravila v oklepajih se uporabijo samo za takojšnjo semantiko medtem ko niso potrebna za leno semantiko.

Pravilo 9.6 pravi, da je $unit$ tipa Unit. Pravila 9.7 in 9.8 predstavita projekcijo prve in druge komponente para ter pripadajoča tipa. Pravila 9.9-9.12 so usmerjevalna pravila, ki usmerjajo evaluacijo projekcij in evaluacijo parov.

Vrstni red pravil in meta-spremenljivk v pravilih povzroči evaluacijo od leve proti desni.

Primer 9.2.1. *Poglejmo si primer. Uporabili bomo takojšnjo semantiko–ovrednotili bomo obe komponenti para in šele nato izvedli projekcijo.*

$$\begin{array}{ll} & \{\textit{pred}\ 4, \textit{if}\ true\ \textit{then}\ false\ \textit{else}\ false\}.1 \\ \mapsto & \{3, \textit{if}\ true\ \textit{then}\ false\ \textit{else}\ false\}.1 \\ \mapsto & \{3, false\}.1 \\ \mapsto & 3 \end{array} \tag{9.13}$$

□

Primer 9.2.2. *Poglejmo si še evaluacijo funkcije nad parom. Tudi v tem pimeru uporabljamo takojšnjo semantiko saj se najprej ovrednotijo obe komponenti para in šele nato se ovrednoti funkcija.*

$$\begin{array}{ll} & (\lambda x : \textit{nat} \times \textit{nat}.\ x.2)\{\textit{pred}\ 4, \textit{pred}\ 5\} \\ \mapsto & (\lambda x : \textit{nat} \times \textit{nat}.\ x.2)\{3, \textit{pred}\ 5\} \\ \mapsto & (\lambda x : \textit{nat} \times \textit{nat}.\ x.2)\{3, 4\} \\ \mapsto & \{3, 4\}.2 \\ \mapsto & 4 \end{array} \tag{9.14}$$

□

Koherenco med statično in dinamično semantiko pokažemo na običajen način.

Izrek 9.2.1 (Varnost)**.** *1. Če $e : \tau$ in $e \to e'$ za nek $e' : \tau$ (ohranitev).*

2. Če $e : \tau$ potem velja bodisi e val ali $e \to e'$ za nek e' (napredek).

Dokaz. Oboje lahko dokažemo z indukcijo po izpeljavi e. □

9.2.2 N-kratni produkti

Enostavno je posplošiti binarne produkte na n-kratne produkte. Na primer, n-terica $\{1, 2, true\}$ vsebuje števila in boolovo vrednost. Tip te n-terice zapišemo $\{\text{nat}, \text{nat}, \text{bool}\}$.

Cena posplošitve binarnih produktov na n-terice je definicija nove notacije, ki uniformno opiše poljuben produkt n elementov. N-terice z n komponentami opišemo z zapisom $\{t_i^{i\in 1..n}\}$. Tip z n komponentami predstavimo z $\{T_i^{i\in 1..n}\}$.

Sintaksa n-teric in operacij nad n-tericami je definirana s sledečimi pravili.

Tipi T	::=	$\{T_i^{i\in 1..n}\}$	n-terice tipov
Izrazi t	::=	$\{t_i^{i\in 1..n}\}$	n-terice izrazov
		$t.i$	projekcija
Vrednosti v	::=	$\{v_i^{i\in 1..n}\}$	n-terice kot vrednost

Poglejmo si spet najprej statično semantiko, ki definira strukturo n-teric s pravili za prirejanje tipov.

$$\frac{\forall i \in [1..n] : \Gamma \vdash t_i : T_i}{\Gamma \vdash \{t_i{}^{i\in 1..n}\} : \{T_i{}^{i\in 1..n}\}} \tag{9.15}$$

$$\frac{\Gamma \vdash t_1 : \{T_i{}^{i\in 1..n}\}}{\Gamma \vdash t_1.j : T_j} \tag{9.16}$$

Pravilo 9.15 konstruira tip n-terice iz tipov komponent. Drugo pravilo deluje obratno: iz n-terice in pripadajočega tipa projecira j-to komponento in njen tip.

Dinamična semantika n-teric je definirana z naslednjimi evaluacijskimi pravili.

$$\frac{}{\{v_i^{i\in 1..n}\}.j \to v_j} \tag{9.17}$$

$$\frac{t_1 \to t_1'}{t_i.i \to t_i'.i} \tag{9.18}$$

$$\frac{t_j \to t_j'}{\{v_i{}^{j\in 1..j-1}, t_j, t_k{}^{k\in j+1..n}\} \to \{v_i{}^{j\in 1..j-1}, t_j', t_k{}^{k\in j+1..n}\}} \tag{9.19}$$

Pravilo 9.17 definira projekcijo komponente v_j iz n-terice. Pravilo 9.18 omogoča evaluacijo komponent n-terice. Tretje pravilo 9.19 definira strategijo evaluacije n-teric—komponente se ovrednotijo od leve proti desni.

9.2.3 Zapisi

Posplošitev n-teric na označene zapise je precej direktna. Preprosto označimo vsako komponento n-terice t z oznako $l_i \in \mathcal{L}$. Na primer, zapisa $\{x = 7\}$ in $\{id = 234, cena = 110.99\}$ imata tip $\{x : \text{nat}\}$ in $\{id : \text{nat}, cena : \text{float}\}$.

Sintaksa zapisov je definirana na sledeč način.

$$\begin{array}{lcll} \text{Tipi } T & ::= & \{l_i : T_i^{i\in 1..n}\} & \text{tip zapisa} \\ \text{Izrazi } t & ::= & \{l_i = t_i^{i\in 1..n}\} & \text{zapis} \\ & & t.l & \text{projekcija} \\ \text{Vrednosti } v & ::= & \{l_i = v_i^{i\in 1..n}\} & \text{vrednosti} \end{array} \tag{9.20}$$

Programski jeziki se razlikujejo v obravnavanju komponent zapisov. Vrstni red komponent običajno ne vpliva na pomen zapisa. Na primer, zapisa $\{a = 1, b = 2\}$ in $\{b = 2, a = 1\}$ sta pomensko enaka.

Poglejmo si najprej spet statično semantiko. Nova pravila za prirejanje tipov.

$$\frac{\forall i \in [1..n] : \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i^{i\in 1..n}\} : \{l_i : T_i^{i\in 1..n}\}} \tag{9.21}$$

$$\frac{\Gamma \vdash t : \{T_i^{i\in 1..n}\}}{\Gamma \vdash t.l_j : T_j} \tag{9.22}$$

Pravilo 9.21 konstruira zapis iz komponent $t_i : T_i$. Pravilo 9.22 razstavi zapis $t : \{T_i^{i\in 1..n}\}$ s projekcijo, ki je definirana na osnovi imena komponente l_j.

Evaluacijska pravila, ki definirajo dinamično semantiko zapisov so naslednja.

$$\frac{}{\{l_i = v_i^{i\in 1..n}\}.j \to v_j} \tag{9.23}$$

$$\frac{t_1 \to t_1'}{t_i.l \to t_i'.l} \tag{9.24}$$

$$\frac{t_j \to t_j'}{\begin{array}{c}\{l_i = v_i^{j\in 1..j-1}, l_j = t_j, l_k = t_k^{k\in j+1..n}\} \\ \to \\ \{l_i = v_i^{j\in 1..j-1}, l_j = t_j', l_k = t_k^{k\in j+1..n}\}\end{array}} \tag{9.25}$$

Očitno predstavljajo zapisi posplošitev produktov. Če bi dovolili, da so imena komponent cela števila bi lahko zapisali tip $\{\text{bool}, \text{nat}, \text{bool}\}$ kot okrajšavo $\{1 : \text{bool}, 2 : \text{nat}, 3 : \text{bool}\}$. Razlikovanje med zapisi in produkti je običajno posledica različnih implementacij tipov.

Programski jeziki se razlikujejo v obravnavanju vrstnega reda komponent zapisov. Veliko programskih jezikov ne upošteva vrstnega reda komponent. Zapisa $\{id = 234, cena = 110.99\}$ in $\{cena = 110.99, id = 234\}$ sta ekvivalentna v takšnih jezikih. V drugih programskih jezikih je vrstni red komponent pomemben. Na vrstnem redu velikokrat sloni tudi fizična predstavitev zapisa.

Naša predstavitev sloni na drugi možnosti. Zapisa $\{id = 234, cena = 110.99\}$ in $\{cena = 110.99, id = 234\}$ sta obravnavana kot različna. S tem pristopom lahko študiramo lastnosti obeh variant.

9.3 Vsote

Veliko programov potrebuje gradnike za delo s heterogenimi kolekcijami objektov.

V podatkovnih strukturah uporabljamo *alternative* za izražanje razlik med deli podatkovnih struktur kot npr. notranja vozlišča in listi v drevesih ali glava in rep seznama. Izbira alternative definira strukturo vrednosti.

Za opisovanje izbire imamo definirane tipe zgrajene na osnovi *vsote*. Takšni tipi so binarne vsote, nične vsote in n-arne vsote, variante in opcije.

Poglejmo si najprej najenostavnejšo obliko vsot: binarne vsote. V nadaljevanju si bomo ogledali še opcije.

9.3.1 Binarne in večkratne vsote

Vsota dveh tipov opisuje množico vrednosti, ki imajo enega izmed naštetih tipov. Oglejmo si primer.

$$\begin{array}{lcl} PhysicalAddress & = & \{firstlast : \text{String}, addr : \text{String}\} \\ VirtualAddress & = & \{name : \text{String}, email : \text{String}\} \end{array} \tag{9.26}$$

Fizični in virtualni naslov predstavljata dva alternativna načina za opis naslovov. Če želimo delati s kolekcijami obeh naslovov moramo definirati vsoto tipov. Elementi tega tipa so bodisi enega ali drugega tipa.

$$Addr \quad = \quad PhysicalAddress + VirtualAddress; \tag{9.27}$$

Elemente tega tipa kreiramo tako, da *označimo* elemente komponentnih tipov PhysicalAddress in Virtual address. Na primer, če je pa tipa $PhysicalAddress$ potem je inl pa tipa $Addr$. Oznake inl in inr si lahko predstavljamo kot funkcije.

$$\begin{array}{l} \text{inl} : PhysicalAddress \rightarrow Addr \\ \text{inr} : VirtualAddress \rightarrow Addr \end{array} \tag{9.28}$$

Funkciji preslikata elemente $PhysicalAddress$ in $VirtualAddress$ v levo ali desno komponento tipa $Addr$.

Elementi tipa $T_1 + T_2$ vsebujejo elemente tipa T_1 označene z inl in elemente tipa T_2 označene z inr.

Elemente tipa $T_1 + T_2$ lahko uporabljamo s pomočjo case stavka, ki omogoča razlikovanje med elementi na levi strani in tiste na desni.

Naslednja funkcija izlušči ime iz elementa tipa $Addr$.

$$getName = \lambda a : Addr.\text{case } a \text{ of inl } x \Rightarrow x.firstlast \mid \text{inr } y \Rightarrow y.name; \tag{9.29}$$

Oglejmo si zdaj kompletno sintakso gradnikov, ki realizirajo binarne vsote.

$$\begin{array}{llll} Tipi & \tau & ::= & \text{Unit} \mid \text{sum}(\tau_1, \tau_2) \\ Izrazi & e & ::= & \text{inl}[\tau](e) \mid \text{inr}[\tau](e) \mid \\ & & & \text{case}[\tau_1, \tau_2](e, x_1.e_1, x_2.e_2) \end{array} \tag{9.30}$$

Konkretna sintaksa je predstavljena s sledečo tabelo.

Abstraktno	Konkretno
Unit	Unit
$\text{sum}(\tau_1, \tau_2)$	$\tau_1 + \tau_2$
$\text{inl}[\tau](e)$	$\text{inl}_\tau(e)$
$\text{inr}[\tau](e)$	$\text{inr}_\tau(e)$
$\text{case}[\tau_1, \tau_2](e, x1.e1, x2.e2)$	$case\ e\ \{\text{inl}(x1) \Rightarrow e1 \mid \text{inr}(x2) \Rightarrow e2\}$

(9.31)

Tip Unit je nična vsota katere vrednosti so izbrane iz nič možnosti. Tip $\text{sum}(\tau_1, \tau_2)$ je binarna vsota, kjer ima lahko vrednost dve možni oznaki $\text{inl}[\tau](e)$ in $\text{inr}[\tau](e)$.

Statična semantika vsote je definirana s sledečimi pravili.

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau = \text{sum}(\tau_1, \tau_2)}{\Gamma \vdash \text{inl}[\tau](e) : \tau} \tag{S-Inl}$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau = \text{sum}(\tau_1, \tau_2)}{\Gamma \vdash \text{inr}[\tau](e) : \tau} \tag{S-Inr}$$

$$\frac{\Gamma \vdash e : \text{sum}(\tau_1, \tau_2) \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \text{case}[\tau_1, \tau_2](e, x1.e1, x2.e2) : \tau} \tag{S-Case}$$

Kot v primeru pogojnega izraza morata obe veji $case$ imeti isti tip. Izraz predstavlja statično oceno oblike $case$ izraza, ki se ovrednoti v času izvajanja.

Dinamična semantika vsote je definirana na sledeč način.

$$\frac{e \text{ val}}{\text{inl}[\tau](e) \text{ val}} \tag{D-InlVal}$$

$$\frac{e \text{ val}}{\text{inr}[\tau](e) \text{ val}} \tag{D-InrVal}$$

$$\left\{ \frac{e \mapsto e'}{\text{inl}[\tau](e) \mapsto \text{inl}[\tau](e')} \right\} \tag{D-Inl}$$

$$\left\{ \frac{e \mapsto e'}{\text{inr}[\tau](e) \mapsto \text{inr}[\tau](e')} \right\} \tag{D-Inr}$$

$$\frac{e \mapsto e'}{\text{case}[\tau_1, \tau_2](e, x1.e1, x2.e2) \mapsto \text{case}[\tau_1, \tau_2](e', x1.e1, x2.e2)} \tag{D-Case}$$

$$\frac{\{e \text{ val}\}}{\text{case}[\tau_1,\tau_2](\text{inl}[\tau](e), x1.e1, x2.e2) \mapsto [e/x_1]e_1} \quad \text{(D-CaseInl)}$$

$$\frac{\{e \text{ val}\}}{\text{case}[\tau_1,\tau_2](\text{inr}[\tau](e), x1.e1, x2.e2) \mapsto [e/x_2]e_2} \quad \text{(D-CaseInr)}$$

Koherenco med statično in dinamično semantiko pokažemo na običajen način.

Izrek 9.3.1 (Varnost). *1. Če $e : \tau$ in $e \mapsto e'$ za nek $e' : \tau$ (ohranitev).*

2. Če $e : \tau$ potem velja bodisi e val ali $e \mapsto e'$ za nek e' (napredek).

Primer 9.3.1. *Ena možna uporaba vsot je definicija tipa $bool$, ki ima sledečo sintakso.*

$$\begin{array}{lcl} \textit{Tipi } \tau & ::= & \textit{bool} \\ \textit{Izrazi } e & ::= & \textit{tt} \mid \textit{ff} \mid \textit{if}(e, e_1, e_2) \end{array} \quad (9.32)$$

Tip bool lahko definiramo tudi z uporabo vsot in in ničnih produktov na sledeč način.

$$\begin{array}{rcl} \textit{bool} & = & \textit{sum}(\textit{Unit}, \textit{Unit}) \\ \textit{tt} & = & \textit{inl}[\textit{bool}](unit) \\ \textit{ff} & = & \textit{inr}[\textit{bool}](unit) \\ \textit{if}(e, e_1, e_2) & = & case[\textit{Unit}, \textit{Unit}](e, x1.e1, x2.e2) \end{array} \quad (9.33)$$

Spremenljivki x_1 in x_2 sta odveč, ker njihov tip Unit določi njihovo vrednost $unit$, še več, ne pojavijo se proste. □

Poleg binarnih vsot imamo tudi večkratne vsote $\sum_{i \in I} \tau_i$ s katerimi predstavljamo množice objektov tipov τ_i. Semantiko večkratnih vsot dobimo z enostavno raširitvijo binarnih vsot na več tipov.

V programskem jeziku ML z vsotami predstavimo tako podatkovne strukture kot tudi programe. Alternative pri izvajanju, na primer, so definirane z vsotami.

Večino lastnosti relacije tip, ki veljajo za čisti λ-račun s tipi $\lambda_{\rightarrow}$ se ohranijo, če dodamo vsote. Ena izmed ključnih lastnosti pa se ne ohrani: *enoličnost tipov*.

Izrek o enoličnosti tipov pravi, da ima vsak izraz natančno en tip. Enoličnost tipov smo do sedaj implicitno predpostavljali.

V primeru uporabe vsot ima vsak objekt $t_1 : T_1$ vsote vsaj dva tipa: originalnega T_1, tistega definiranega z vsoto inl $t_1 : T_1 + T_2$ kot tudi katerikoli drugo vsoto $T_1 + T$ za poljubni T.

Primer 9.3.2. *Na primer, izpeljemo lahko 5 : Nat, inl 5 : Nat + Nat kot tudi inl 5 : Nat + Bool.*

Brez enoličnosti tipov ne moremo zasnovati algoritma za preverjanje tipov preprosto tako, da beremo pravila od zgoraj navzdol kot smo predpostavljali do sedaj. Imamo več možnosti.

Prva možnost je, da program za preveranje tipa nekako "ugane" tip T_2. Začasno lahko pustimo nedefiniran T_2 in ga določimo kasneje, ko je to jasno iz drugega konteksta.

Druga možnost je, da popravimo jezik tipov tako, da dovoljuje vse možne vrednosti T_2. To možnost lahko implementiramo skupaj s podtipi.

Tretja možnost je, da zahtevamo od programerja, da eksplicitno označi tip T_2. Ta alternativa je najbolj enostavna in tudi ni tako nepraktična kot se najprej lahko zdi.

9.3.2 Opcije

Zelo uporaben gradnik, ki je definiran nad alternativnimi tipi iz vsote je *opcija*. Poglejmo si primer.

$$OptionalNat = \langle none : \text{Unit}, some : \text{nat} \rangle; \tag{9.34}$$

Tip predstavlja bodisi vrednost unit z oznako $none$ ali celoštevilsko vrednost tipa nat, ki ima oznako $some$. Poglejmo si primer uporabe opcijskega tipa $OptionalNat$.

Primer 9.3.3.

$$Table = \textsf{nat} \to OptionalNat; \tag{9.35}$$

Funkcija predstavlja preslikavo med celimi števili in celimi števili z vrednostjo unit. Prazno tabelo lahko definiramo z naslednjim izrazom.

$$\begin{array}{ll} & emptyTable = \lambda n : \textsf{nat}.\langle none : \textsf{unit} \rangle \; as \; OptionalNat; \\ \blacktriangleright & emptyTable : Table \end{array} \tag{9.36}$$

Konstruktor definiran s sledečim izrazom vzame tabelo in ji doda en zapis s ključem m in vrednostjo $\langle some : v \rangle$.

$$\begin{array}{lll} & extendTable = & \lambda t : Table.\lambda m : \textsf{nat}.\lambda v : \textsf{nat}.\lambda n : \textsf{nat}. \\ & & \textsf{if} \; equal \; n \; m \; \textsf{then} \langle some = v \rangle \; as \; OptionalNat \\ & & \textsf{else} \; t \; n \\ \blacktriangleright & extendTable : & Table \to \textsf{nat} \to \textsf{nat} \to Table \end{array} \tag{9.37}$$

Za implementacijo dostopa do tabele preko ključa potrebujemo case stavek. Naj bo t tabela. Želimo dostopati do zapisa, ki ima ključ 5.

$$\begin{array}{lll} x = & \textsf{case} \; t(5) \; \textsf{of} \\ & \langle none = u \rangle \to 0 \mid \\ & \langle some = v \rangle \to v; \end{array} \tag{9.38}$$

□

Nekateri jeziki omogočajo vgrajeno podporo za opcije. Programski jezik OCaml vsebuje predefiniran tip $option$. Funkcije v OCaml velikokrat vrnejo opcije. Veliko jezikov vsebuje $null$ kazalec, ki implicitno definira opcijo.

9.4 Podtipi

V prejšnjih poglavjih smo študirali raznovrstne gradnike programskih jezikov v okviru lambda računa, ki je služil kot osnovni formalizem.

Podtipi so razširitev lambda računa v smeri strukture. Predstavili bomo sintaktične in semantične relacije, ki definirajo strukturo prostora izrazov. Izraze bomo urejali po specifičnosti – nekateri izrazi so posebni primeri drugih izrazov ali z drugimi besedami nekateri tipi so podtipi drugih tipov.

Podtipe uporabljamo pri predstavitvi objektno usmerjenih jezikov. Običajno jih obravnavajo kot potrebno značilnost objektnega jezika.

9.4.1 Vsebovanost

Brez podtipov so lahko enostavna pravila lambda računa zelo rigidna.

Sistem tipov zahteva eksaktno ujemanje med argumenti in spremenljivkami zato preverjanje tipov zavrne marsikateri program, ki se zdi sprejemljiv. Poglejmo na primer pravilo za aplikacijo funkcije.

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2} \tag{9.39}$$

Primer 9.4.1. *Po tem pravilu dobro definiran izraz*

$(\lambda r : \{x : \mathit{Nat}\}.\ r.x)\ \{x = 0, y = 1\}$

nima pravega tipa, ker je tip argumenta $\{x = \mathit{Nat}, y = \mathit{Nat}\}$*, funkcija pa sprejema tip* $\{x : \mathit{Nat}\}$*. Funkcija zahteva argument* x *in dodatne argumente ignorira. Še več, ni potrebno pogledati v telo funkcije, da bi videli, da se dodatni argumenti ne potrebujejo. Vedno je varno podajati argumente z večimi komponentami.* □

Cilj uporabe podtipov je omogočiti izraze kot je zgornji. To dosežemo tako, da formaliziramo idejo, ki pravi, da so nekateri tipi specializirane verzije drugih tipov.

Pravimo, da je S podtip T, napisano $S <: T$, kar pomeni, da se tip T lahko varno uporablja namesto tipa S. Ta pogled na podtipe se pogosto imenuje *princip (varne) substitucije*.

Izraz $S <: T$ intuitivno razumemo: vsaka vrednost opisana z S je opisana tudi z T. Na drugi način povedano: elementi tipa S so pomnožica elementov iz T.

Osnovni princip za intuitivno interpretacijo relacije podtipov je *vsebovanje*. Poglejmo si pravilo vsebovanja.

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T} \tag{T-SUB}$$

Pravilo nam pove, da če velja $S <: T$ potem je vsak element t tipa S tudi element tipa T.

Primer 9.4.2. *Recimo, da velja* $\{x : Nat, y : Nat\} <: \{x : Nat\}$*, potem lahko uporabimo pravilo T-SUB za izpeljavo* $\vdash \{x : 0, y : 1\} : \{x : Nat\}$. □

9.4.2 Relacija podtip

Relacija podtip je definirana z množico pravil, ki opisujejo stavke oblike $S <: T$. Pravimo, da je „S podtip T.“

Relacijo podtip bomo definirali odvisno od strukture tipa: funkcija, zapis, itd. Za vsako vrsto tipa bomo definirali pravila, ki bodo povedala kdaj lahko nek tip zamenjamo z drugim.

Preden gremo k pravilom za posamezno vrsto tipov si oglejmo dva splošna primera.

Prvič, relacija podtip mora biti refleksivna.

$$S <: S \tag{T-REFL}$$

Drugič, relacija podtip mora biti tranzitivna.

$$\frac{S <: U \qquad U <: T}{S <: T} \tag{S-TRANS}$$

Ta pravila sledijo direktno iz zahtev po varni substituciji.

Zapisi

Za zapise smo že videli, da bi želeli definirati, da je $S = \{l_1 : T_1, \ldots, l_m : T_m\}$ podtip $T = \{l_1 : T_1, \ldots, l_n : T_n\}$, če ima T manj komponent kot S. Z drugimi besedami, varno je pozabiti nekatere komponente na koncu zapisa. Pravilo podtipov na osnovi širine se glasi takole.

$$\overline{\{l_i : T_i{}^{i \in 1..n+k}\} <: \{l_i : T_i{}^{i \in 1..n}\}} \tag{S-RCDWIDTH}$$

Mogoče se zdi presenetljivo, da je “manjši tip” tisti, ki ima več komponent. Relacija $<:$ ima dejansko pomen “bolj specifičen”: izraz $S <: T$ pravi, da je S bolj natančno definiran, medtem ko je T bolj splošen tip.

Primer 9.4.3. *Tip $\{x : Nat\}$ ima samo komponento x. Vrednost $\{x = 3\}$ je element tega tipa, kot tudi $\{x = 3, y = 100\}$ ter $\{x = 3, y = 15, z = 100\}$ so elementi tega tipa. Zadnji dve vrednosti sta tudi elementa tipa $\{x : Nat, y : Nat\}$. V splošnem je množica elementiv tipa $\{x : Nat, y : Nat\}$ pomnožica množice elementov tipa $\{x : Nat\}$.* □

Bolj specifičen tip je torej bolj natančno definiran in bolj informativen zato predstavlja manjšo množico vrednosti kot bolj splošen tip.

Pravilo podtipov, ki temelji na širini zapisov, deluje samo na zapisih, ki imajo identične komponente. Varno je dovoliti razlikovanje v posameznih tipih z istimi oznakami, če sta tipa v relaciji podtip. Pravilo zasnovano na tem principu imenujemo relacija podtipov na osnovi globine.

$$\frac{\forall i \in [1..n] : \ S_i <: T_i}{\{l_i : S_i{}^{i\in 1..n}\} <: \{l_i : T_i{}^{i\in 1..n}\}} \qquad \text{(S-RCDDEPTH)}$$

Primer 9.4.4. *Naslednja izpeljava uporablja pravili S-RECWIDTH in S-RECDEPTH za dokaz, da je zapis $\{x : \{a : Nat, b : Nat\}, y : \{m : Nat\}\}$ podtip od $\{x : \{a : Nat\}, y : \{\}\}$.*

$$\frac{\dfrac{}{\{a : Nat, b : Nat\} <: \{a\ Nat\}}\textit{S-RCDWIDTH} \qquad \dfrac{}{\{m : Nat\} <: \{\}}\textit{S-RCDWIDTH}}{\{x : \{a : Nat, b : Nat\}, y : \{m : Nat\}\} <: \{x : \{a : Nat\}, y : \{\}\}}\textit{S-RCDDEPTH}$$

Če se komponenti zapisa, ki ga primerjamo ujemata potem lahko uporabimo pravilo S-REFL, da bi pokazali izpeljavo podtipov.

$$\frac{\dfrac{}{\{a : Nat, b : Nat\} <: \{a\ Nat\}}\textit{S-RCDWIDTH} \qquad \dfrac{}{\{m : Nat\} <: \{m : Nat\}}\textit{S-REFL}}{\{x : \{a : Nat, b : Nat\}, y : \{m : Nat\}\} <: \{x : \{a : Nat\}, y : \{m : Nat\}\}}\textit{S-RCDDEPTH}$$

Uporabimo lahko tudi pravilo tranzitivnosti S-TRANS za kombiniranje pravil na osnovi širine in globine. Na primer, lahko dobimo podtipe, kjer uporabimo pravilo na osnovi širine na eni komponenti in pravilo na osnovi globine na drugi komponenti. Poglejmo si primer, kjer bomo razbili izpeljavo na tri dele.

$$\frac{}{\{x : \{a : Nat, b : Nat\}, y : \{m : Nat\}\} <: \{x : \{a : Nat, b : Nat\}\}}\textit{S-RCDWIDTH}$$

$$\frac{\dfrac{}{\{a : Nat, b : Nat\} <: \{a : Nat\}}\textit{S-RCDWIDTH}}{\{x : \{a : Nat, b : Nat\}\} <: \{x : \{a : Nat\}\}}\textit{S-RCDDEPTH}$$

$$\frac{\dots\textit{S-RCDWIDTH} \qquad \dots\textit{S-RCDDEPTH}}{\{x : \{a : Nat, b : Nat\}, y : \{m : Nat\}\} <: \{x : \{a : Nat\}\}}\textit{S-TRANS}$$

□

Zadnje pravilo za podtipe zapisov pravi, da vrstni red komponent v zapisu ne igra vloge pri relaciji podtip.

$$\frac{\{k_j : S_j{}^{j\in 1..n}\} \text{ je permutacija } \{l_i : T_i{}^{i\in 1..n}\}}{\{k_j : S_j{}^{j\in 1..n}\} <: \{l_i : T_i{}^{i\in 1..n}\}} \qquad \text{(S-RCDPERM)}$$

Na primer, pravilo S-RCDPERM pravi, da $\{c : \text{Nat}, b : \text{Bool}, a : \text{Nat}\}$ je podtip $\{a : \text{Nat}, b : \text{Bool}, c : \text{Nat}\}$ in obratno.

Pravilo S-RCDPERM skupaj s pravili S-RCDWIDTH in S-TRANS lahko uporabimo za primerjavo zapisov, kjer lahko zdaj izpustimo tudi komponento na sredini zapisa in ne samo na koncu zapisa.

Vaja 9.4.1.

...

Funkcije

Ker delamo v jeziku višjega reda, kjer imamo lahko ne samo enostavne tipe in zapise ampak tudi funkcije, moramo definirati relacijo tipov tudi nad funkcijami.

Povedati moramo v katerih primerih je varno podati funkcijo nekega tipa v kontekst, kjer se pričakuje druga funkcija.

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \qquad \text{(S-ARROW)}$$

Relacija podtip med funkcijami $S_1 \to S_2 <: T_1 \to T_2$ zahteva obrnjeno relacijo med domenami funkcij $T_1 <: S_1$ in relacijo definirano v isto smer za zaloge vrednosti.

Relacijo med domenami imenujemo *kontra-variantno*, medtem ko relacijo med zalogami vrednosti je *kovariantna*.

Razlaga za takšno definicijo je sledeča. Recimo, da imamo funkcijo f tipa $S_1 \to S_2$.

- Funkcija f spremjema elemente tipa S_1, torej bo f lahko sprejela tudi katerikoli element podtipa T_1.
- Funkcija f vrača elemente S_2 torej lahko vidimo te iste elemente kot elemente nadtipa T_2.

Funkcijo f torej lahko vidimo tudi kot, da ima tip $T_1 \to T_2$.

Alternativni pogled je: lahko varno dovolimo uporabo funkcije tipa $S_1 \to S_2$ tudi v kontekstu, kjer se pričakuje drugi tip $T_1 \to T_2$, dokler:

- ne bo argument funkcije presenečenje ($T_1 <: S_1$: pričakujemo T_1 torej je funkcija, ki pričakuje bolj splošen tip S_1 tudi v redu) in

- nobeden rezultat ne bo presenečenje ($S_2 <: T_2$: vrnemo S_2 kar je sigurno tudi T_2).

9.4.3 Statična semantika $\mathcal{L}(\rightarrow<:)$

Poglejmo si zdaj jezik $\lambda_{\rightarrow}$, ki so mu dodani podtipi. Pravila, ki smo jih definirali za posamezne vrste tipov glede na strukturo bodo zbrana skupaj s pravili za osnoven lambda račun.

Najprej si bomo ogledali statično semantiko $\lambda_{\rightarrow}$, ki je razširjen z osnovnimi pravili za delo s podtipi. Obravnavani so tudi podtipi funkcij.

$$
\begin{array}{lll}
t & ::= & \text{izrazi} \\
 & x & \text{spremenljivka} \\
 & \lambda x : T.t & \text{abstrakcija} \\
 & t\ t & \text{aplikacija} \\
v & ::= & \text{vrednosti} \\
 & \lambda x : T.t & \text{vrednost abstrakcije} \\
T & ::= & \text{tipi} \\
 & \text{Top} & \text{maksimalen tip} \\
 & T \rightarrow T & \text{funkcijski tip} \\
\Gamma & ::= & \text{konteksti} \\
 & \emptyset & \text{prazen kontekst} \\
 & \Gamma, x : T & \text{vezana spremenljivka}
\end{array}
\tag{9.40}
$$

Pravila, ki definirajo tipe izrazov danega jezika opišejo, kako se sestavljajo izrazi in kakšnega tipa je rezultat.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \tag{9.41}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.t_2 : T_1 \rightarrow T_2} \tag{9.42}$$

$$\frac{\Gamma \vdash x : T_{11} \rightarrow T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \tag{9.43}$$

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T} \tag{9.44}$$

V statično semantiko jezika spada tudi definicija relacije podtip. Pravila za implementacija relacije podtip razširijo množico veljavnih izrazov na vse tiste, kjer tip zamenjamo z bolj splošnim ali bolj specifičnim tipom.

$$\frac{}{S <: S} \tag{9.45}$$

$$\frac{S <: U \qquad U <: T}{S <: T} \tag{9.46}$$

$$\frac{}{S <: \text{Top}} \tag{9.47}$$

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \tag{9.48}$$

Poglejmo si zdaj še pravila, ki opisujejo statično semantiko za zapise. Najprej nove sintaksne oblike izrazov.

$$\begin{array}{lcll} t & ::= & ... & \text{izrazi} \\ & & \{l_i = {t_i}^{i\in 1..n}\} & \text{zapis} \\ v & ::= & ... & \text{vrednosti} \\ & & \{l_i = {v_i}^{i\in 1..n}\} & \text{vrednost zapisa} \\ T & ::= & ... & \text{tipi} \\ & & \{l_i = {T_i}^{i\in 1..n}\} & \text{tip zapisa} \end{array} \tag{9.49}$$

Pravila za definicijo tipov zapisov so sledeča. Bolj podrobno smo si jih že ogledali v poglavju, ki predstavlja strukture.

$$\frac{\forall i \in [1..n] : \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = {t_i}^{i\in 1..n}\} : \{l_i = {T_i}^{i\in 1..n}\}} \tag{9.50}$$

$$\frac{\Gamma \vdash t_1 : \{l_i = {T_i}^{i\in 1..n}\}}{\Gamma \vdash t_1.l_j : T_j} \tag{9.51}$$

Preostanejo samo še nova pravila za definicijo podtipov med izrazi, ki so strukturirani kot zapisi.

$$\frac{}{\{l_i = {T_i}^{i\in 1..n+k}\} <: \{l_i = {T_i}^{i\in 1..n}\}} \tag{9.52}$$

$$\frac{\forall i \in [1..n] : S_i <: T_i}{\{l_i = {S_i}^{i\in 1..n}\} <: \{l_i = {T_i}^{i\in 1..n}\}} \tag{9.53}$$

$$\frac{\{l_i = {S_i}^{i\in 1..n}\} \text{ je permutacija } \{l_i = {T_i}^{i\in 1..n}\}}{\{l_i = {S_i}^{i\in 1..n}\} <: \{l_i = {T_i}^{i\in 1..n}\}} \tag{9.54}$$

9.4.4 Dinamična semantika $\mathcal{L}(\to <:)$

Pravila, ki definirajo evaluacijo lambda računa razširjenega za delo s podtipi $\mathcal{L}(\to <:)$. Enako kot pri statični semantiki bomo tudi tukaj najprej predstavili evaluacijo lambda računa samega in potem šele evaluacijo zapisov.

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2} \tag{9.55}$$

$$\frac{t_2 \to t_2'}{v_1\ t_2 \to v_1\ t_2'} \tag{9.56}$$

$$\frac{}{(\lambda x : T_{11}.t_{12})\ v_2 \to [v_2/x]t_{12}} \tag{9.57}$$

Zgornja pravila definirajo metodo *klic-po-vrednosti*, ki ovrednoti izraze tako, da parametre lambda abstrakcije prevede čim prej v vrednosti.

Poglejmo si zdaj še evaluacijo izrazov, katerih struktura so zapisi. Pravila definirajo evaluacujo struktur zapisov: selekcijo vrednosti komponente, izpeljavo zapisa s selektirano komponento, in izpeljavo komponente zapisa.

$$\frac{}{\{l_i = S_i{}^{i\in 1..n}\}.t_j \to v_j} \tag{9.58}$$

$$\frac{t_1 \to t_1'}{t_1.l \to t_1'.l} \tag{9.59}$$

$$\frac{t_j \to t_j'}{\begin{array}{l}\{l_i = v_i{}^{i\in 1..j-1}, l_j = t_j, l_i = v_i{}^{i\in j+1..n}\} \\ \to \{l_i = v_i{}^{i\in 1..j-1}, l_j = t_j', l_i = v_i{}^{i\in j+1..n}\}\end{array}} \tag{9.60}$$

Na koncu bomo pogledali še varnost izrazov: velja napredek in ohranitev za izraze.

Izrek 9.4.1 (Varnost). *1. Če $e : \tau$ in $e \mapsto e'$ za nek $e' : \tau$ (ohranitev).*

2. Če $e : \tau$ potem velja bodisi e val ali $e \mapsto e'$ za nek e' (napredek).

9.5 Opombe

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