Teorija programskih jezikov

Zapiski

Iztok Savnik
Fakulteta za matematiko, naravoslovje in informacijske tehnologije
Univerza na Primorskem, Koper

2. februar 2011

Kazalo

Zapiski so osnova za predavanja pri predmetu *Teorija programskih jezikov* na podiplomskem študiju računalništva in informatike, FAMNIT, Koper. Zapiski so *osnutek* skripte in vsebujejo napake.

Zapiski vsebujejo prevode izbranih poglavij in sekcij iz knjige Benjamina Pierce: *Types and Programming languages* [10] in knjige Roberta Harperja: *Practical Foundations for Programming Languages* [5]. Med pomembnejšimi učbeniki in viri, na katerih temeljijo zapiski je tudi članek Henk Barendregt in Erik Barendsen: *Introduction to Lambda Calculus* [2], učbenik Glyn Winskel: *The Formal Semantics of Programming Languages: An Introduction* [13], knjiga Jean-Yves Girard: *Proofs and Types* [3] in Pfenningovo delo predstavljeno v knjigi: *Computation and Deduction* [9].

Osnovna vodila pri definiciji strukture vsebine zapiskov so naslednja.

1. Podati state-of-the-art *pregled področja* teorije programskih jezikov.
a) Osnovne metode uporabljene za predstavitev semantike programskih jezikov so operacijska semantika, denotacijska semantika in naravna dedukcija.
b) Pregled temelji na *hierarhiji formalnih sistemov* razvitih na osnovi λ-računa, ki predstavljajo osnovne formalne jezike uporabljene za identifikacijo in predstavitev konceptov programskih jezikov.
c) Za vsak nivo hierarhije formalnih sistemov bodo predstavljene: osnovne lastnosti jezika kot so statična semantika in dinamična semantika; pomembnejši rezultati oz. lastnosti jezikov danega nivoja; in množica praktičnih primerov.
d) Podan je pregled konceptov programskih jezikov definiranih v okviru predstavljenih formalnih jezikov.
e) Pregled teorije programskih jezikov temelji na “laboratorijskih” formalnih sistemih, ki so v preteklosti služili za študij formalnih sistemov.

2. Predstaviti uvrstitev teorije programskih jezikov v teoretično računalništvo.
a) Opisati prekrivanje teorije programskih jezikov s telesom znanja o prevajalnikih.
b) Predstaviti povezave in skupne točke med teorijo izračunljivosti in teorijo programskih jezikov.
c) Predstaviti uvrstitev formalnih sistemov teorije programskih jezikov v hierarhijo Chomskega.

Iztok Savnik, 2.12.2010, Koper.

-

Poglavje 1

UVOD

> “Computer Science is no more about computers than astronomy is about telescopes.” - E.W.Dijkstra

Teorija programskih jezikov preučuje prostore, ki jih tvorijo stavki formalnih sistemov.

Opazujemo strukture, ki jih tvorijo stavki, definiramo lahko pomen stavkov in preučujemo pretvorbe stavkov med evaluacijo glede na definiran pomen stavkov.

S formalnimi sistemi predstavimo in preučujemo različne izrazne nivoje modernih programskih jezikov.

Na primer, kot osnovni nivo programskega jezika bi lahko definirali cela števila in operacije za delo s celimi števili. S takšnim jezikom lahko računamo s celimi števili.

Nivoji, ki jih obravnavamo so rekurzivni jeziki, Turingovi jeziki, jeziki s tipi, jeziki z rekurzivnimi tipi, jeziki 2. reda, itd. Za vsak nivo podamo statično in dinamično definicijo ter predstavimo lastnosti struktur, ki jih tvori dani nivo.

Iz drugega vidika, v teoriji programskih jezikov formalno definiramo koncepte programskih jezikov kot so na primer funkcije, tipi, polimorfizem, hierarhija tipov, itd.

Pomen konceptov je v več primerih kompleksen in je prepuščen konkretnim imple-

mentacijam ter prevajalnikom programskih jezikov.

V takšnih primerih nudi teorija programskih jezikov formalna orodja s katerimi lahko natančno preučimo pomen in implementacijo konceptov programskih jezikov.

1.1 Zgodovina

Teorija programskih jezikov temelji na *logiki*, ki jo velikokrat definirajo kot vedo s katero preučujemo izjave ali sodbe. Logiko uporabljamo pri večini intelektualnih aktivnosti zato lahko tudi rečemo, da je logika veda o sklepanju.

Logika je tako rezulat introspekcije: sprašujemo se kaj počnemo pri razmišljanju, na kakšen način dojemamo in obravnavamo koncepte, kako jih povezujemo in sklepamo na posledice iz danih premis?

Aristotelovo delo *Prior Analytics* se šteje kot prva znanstvena razsikava logike. Sicer so logiko preučevali vsi starejši narodi kot npr. Kitajska in Indija.

Izvore formalnih sistemov lahko vidimo v ideji Gottfried Wilhelm Leibniza o univerzalnem matematičnem jeziku–ki ga je imenoval *calculus ratiocinator*–s katerimi se bi dalo opisati in rešiti vse matematične probleme. Calculus ratiocinator si lahko predstavljamo kot formalen sistem ali logiko s katero lahko modeliramo (matematično) razmišljanje[1].

Bertrand Russell je razvil prvo *teorijo tipov* kot odgovor na njegovo odkritje, da je Gottlob Frege-jeva verzija naivne teorije množic vsebovala Russell-ov paradoks. Paradoksu se izogne tako, da najprej definira hierarhijo tipov in priredi vsakemu izrazu tip. Objekti danega tipa so zgrajeni iz objektov predhodnega tipa—tako se izognemo zankam.

Leta 1928 je matematik David Hilbert postavil problem z imenom *Entscheidungspro-*

[1]Obstaja tudi *sintetični pogled*, ki pravi, da je calculus ratiocinator fizični "računalnik", ki omogoča mehansko sklepanje–nekateri trdijo, da je moderen Von Neuman-ov računalnik samo realizacija calculus ratiocinator.

blem. Problem se glasi: ali je mogoče iz opisa formalnega jezika in stavka v tem jeziku napisati algoritem, ki bo odgovoril $true$ v primeru, da je stavek pravilen in $false$ sicer.

Idejo sta uresničila (v istem času, 1932) Turing s strojem, ki zna računati in izražati spekter zahtevnih problemov in Church, ki je definiral jezik, s katerim je mogoče zapisati abstrakcijo matematične funkcije, λ-račun.

Lambda račun je Alonzo Church definiral v okviru raziskave o osnovah matematike. Pokazano je bilo, da orignalen lambda račun vsebuje Kleene-Rosser-jev paradoks. Church je leta 1936 objavil lambda račun brez tipov (untyped lambda calculus) in leta 1940 lambda račun s tipi (typed lambda calculus), ki ne vsebuje več Kleene-Rosser-jevega paradoksa.

Takoj v naslednjih letih po definiciji Turinovega stroja in λ-računa je bilo pokazano, da sta enakovredna po moči in se programe zapisane v enem da prevesti v programe v drugem jeziku.

Lambda račun je v drugi polovici dvajsetega stoletja ...

1.2 Uporaba tipov

Uporaba tipov pripomore k pisanju pravilnih programov. Zaradi definicije tipa vsakega izraza programa se lahko izogne enostavnim kot tudi bolj kompleksnim napakam.

Preverjanje tipov je proces, ki prireja tipe posameznim izrazom programa in primerja izračunane tipe izrazov povsod kjer pravila programskega jezika zahtevajo, da se tipi ujemajo ali so v neki drugi relaciji.

Na primer, tip izraza se mora ujemati s tipom spremenljivke, ki ji je izraz prirejen. Podobno se mora izraz, ki nastopa kot parameter funkcije, ujemati z deklariranim tipo parametra.

Statično preverjanje tipov preveri tipe izrazov v programu v času prevajanja, medtem

ko *dinamično preverjanje tipov* preverja tipe vrednosti in objektov v času izvajanja programa.

Novejši programski jeziki uporabljajo oboje.

1.2.1 Odkrivanje napak

Preverjanje tipov omogoča odkrivanje nekaterih napak v času prevajanja oz. v času izvajanja. V primeru, da se tipi izrazov v programu ne ujemajo nam sistem to sporoči.

Statično preverjanje tipov lahko odkrije presenetljivo veliko napak. Pogosto se zgodi, da programi preprosto “delajo” po tem, ko se odpravijo napake tipov.

Striktno preverjanje tipov torej pogosto uspe odkriti poleg trivialnih napak tudi marsikatero težjo napako. Moč tega učinka je odvisna od izrazne moči sistema za delo s tipi. Čim bolj izrazen je sistem tem več napak uspemo uloviti.

Razlog za ta učinek je predvsem v formi, ki jo daje struktura tipov programom. Programska koda, ki je še posebaj pri objektnih programskih jezikih vezana na tipe oziromo strukture, ki jih tvorijo tipi, je bolj enostavno obvladljiva.

1.2.2 Abstrakcija

Tipi podpirajo proces programiranja tako, da zahtevajo disciplinirano programiranje.

Pri snovanju in implementaciji programskega sistema služijo tipi kot osnovno *okostje sistema* okoli katerega je implementirana koda.

Sistem tipov tvori osnovo za module–tip modula dejansko ustreza vmesniku modula. Strukturiranje večjih sistemov v module omogoča bolj abstraktno zasnovo programskih

sistemov.

1.2.3 Dokumentacija

Strukture tipov, ki definirajo programski sistem služijo kot osnova za dokumentacijo sistema.

Večino objektnih programskih sistemov nudi orodje, ki omogoča enostano generiranje dokumentacije iz parcialnih opisov, ki so vezani na tipe oz. hierarhijo tipov.

Dokumentacija je del samega sistema in jo je tako veliko lažje vzdrževati.

Komentarje običajno lahko enostavno dodajamo na same sintaktične konstrukte programskega jezika kot so na primer razredi, komponente razredov, podatkovne strukture in moduli.

1.2.4 Varnost jezika

Pierce neformalno definira varen jezik kot jezik, ki ti onemgoči, da se ustreliš v lastno nogo medtem, ko programiraš.

Povedano na drug način: jezik je varen, če varuje lastne abstrakcije. Poglejmo si nekaj primerov.

Programski jezik, ki ima definirana polja mora zagotoviti, da programer ne more narediti nič zelo čudnega s polji kot na primer spreminjati vsebino polja tako, da piše preko meja neke druge podtkovne strukture.

Abstrakcija definirana v programskem jeziku kot na primer datoteka mora biti zasnovana in implementirana tako, da programer ne more narediti nič takega, da bi se pro-

gram nedefinirano nehal izvajati.

Spremenljivke, ki so definirane znotraj nekega definicijskega prostora morajo biti dostopne samo znotraj definiranega prostora in ne izven definiranih oblik dostopa.

1.2.5 Učinkovitost

Prvo preverjanje tipov se je začelo pri Fortranu, ko so želeli ločiti med numeričnim računanjem s celimi števili in računanjem z realnimi števili.

Statično preverjanje tipov omogoča bolj učinkovito delo z vrednostmi. Odpade mariskatero preverjanje v času izvajanja.

Marsikatera odločitev pri generiranju kode pride iz tipov izrazov.

Tip kazalcev vpliva na izbiro inštrukcij za delo s kazalci.

Tehnike za izboljšanje algoritmov za čiščenje pomnilniškega prostora.

1.2.6 Načrtovanje jezikov

Programski jeziki morajo biti načrtovani skupaj s sistemom za preverjanje tipov.

Sicer je preverjanje tipov zelo težko.

Dobro načrtovan jezik redko potrebuje podatke tipih izrazov. Večino lahko izpelje sistem sam.

Meta-programski jeziki omogočajo enostavno načrtovanje jezika kot tudi študij lastnosti načrtovanega programskega jezika.

1.3 Praktična uporaba

Področja:

- Meta-programski jeziki
- Načrtovanje in analiza jezikov
- Snovanje novih programskih jezikov

1.3.1 Meta-programski jeziki

V zadnjih desetletjih so se začeli pojavljati meta-programski jeziki, ki na osnovi opisa programskega jezika omogočajo vpogled v sklepanje, kreiranje dokazov lastnosti modeliranega programskega jezika, itd. ...

Novejša programska orodja kot je na primer Twelf nam omogočajo prestavitev formalne semantike jezika v meta-programskem jeziku.

Meta jezik nam omogoča implementacijo analiz programskega jezika kot na primer preverjanje tipov izraza, programsko dokazovanje lastnosti jezikov, interpretacijo izrzov jezika in podobno.

Meta-programski jeziki nudijo okolje za razvoj programskih jezikov. Sintakso in izbrano semantiko načrtovanega programskega jezika lahko formalno definiramo.

Formalna definicija nam lahko služi za iskanje napak v zasnovi programskega jezika. Programsko okolje meta programskega jezika nam omogoča eksperimerntiranje s programskim jezikom.

Relativno enostavno je možno implementirati interpreter jezika, ki nam omogoča vpogled v nekatere lastnosti jezika.

Formalna definicija potemtakem služi tudi kot dokumentacija programskega jezika saj predstavi pomembnejše aspekte načrtovanega programskega jezika.

1.3.2 Načrtovanje in analiza jezikov

Formalna semantika vedno predstavlja *model* računskega pomena programa. Model običajno opiše samo del celotnega pomena programa.

Predstavljen del je definiran zadosti formalno, da lahko analiziramo lastnosti jezika ter opišemo postopke za analizo programa kot so npr. preverjanje tipov, evaluacija, itd.

Rezultat definicije modela nekega jezika z uporabo formalnega jezika je možnost definiranja, implementacije in dokazov lastnosti jezikov.

V zadnjih desetletjih se preko projektov AUTOMATH in Edinburgh Logical Framework, ki sta oba uporabljala inčico λ-računa za predstavitev matematičnih teorij in dokazov izrekov, razvila še množica razširitev lambda računa.

Jezik ima več izraznih nivojev in je definiran na osnovi λ-abstrakcije, notacije, ki omogoča izražanje funkcij, podatkovnih struktur, objektov, ter tudi kompleksnih konceptov programskih jezikov kot so npr. polimorfizem, dedovanje, itd.

Velikokrat bi želeli podrobneje pogledati nekatere lastnosti programov, npr. dokazati, da se program zaključi, preveriti pravilnost in varnost tipov jezika, deterministočnost programa, itd.

Računski pomen je običajno preveč kompleksen zato želimo predstaviti bolj ali manj abstraktni model jezika. Običajno predstavimo samo značilnosti jezika za predstavitev željenega nivoja in aspekta jezika.

Velikokrat zanemarimo značilnosti, ki nas ne zanimajo in ne vplivajo na prerez jezika, ki ga želimo opazovati. Na primer, lahko pozabimo na shranjevanje podatkov, računski čas, itd.

V nadaljevanju uvoda si bomo pogledali bolj natančno nekatere aspekte semantike programskih jezikov in analize, ki jih izvajamo na načrtovanem programskem jeziku.

1.3.3 Snovanje novih programskih jezikov

Zasnova in implementacija enostavnih programskih jezikov je pogosto potrebna v aplikacijah.

Nov jezik tipično uporabljamo pri zasnovi uporabniških vmesnikov, aplikacijskih protokolih v porazdeljenih okoljih, ukaznih jezikih aplikacijskega strežnika ter za definicijo formata za shranjevanje podatkov.

1.4 Zgodovina razvoja programskih jezikov

late 1800s	Origins of formal logic	[Frege]
early 1900s	Formalization of mathematics	[WR25]
30	Untyped lambda-calculus	[Chu41]
40	Simply typed lambda-calculus	[Chu40, CF58]
50	Fortran	[Bac81]
50	Algol	[N+ 63]
60	Automath project	[dB80]
60	Simula	[BDMN79]
70	Martin-Loef type theory	[Mar73, Mar82, SNP90]
60	Curry-Howard isomorphism	[How80]
70	System F, F^{ω}	[Gir72]
70	polymorphic lambda-calculus	[Rey74]
70	CLU	[LAB+ 81]
70	polymorphic type inference	[Mil78, DM82]
70	Logic of Computable Functions	[GMW79]
70	ML	[GMW79]
70	intersection types	[CDC78, CDCS79, Pot80]
80	NuPRL project	[Con86]
80	subtyping	[Rey80, Car84, Mit84a]
80	ADTs as existential types	[MP88]
80	calculus of constructions	[Coq85, CH88]
80	linear logic	[Gir87, GLT89]
80	bounded quantification	[CW85, CG92, CMMS94]
80	Edinburgh Logical Framework	[HHP92]
80	Forsythe	[Rey88]
80	pure type systems	[Bar92]
80	dependent types and modularity	[Mac86]
80	Quest	[Car91]
80	Extended Calculus of Constructions	[Luo90]
80	row variables and extensible records	[Wan87,Rm89 CM91]
90	higher-order subtyping	[Car90, CL91, PT94]
90	typed intermediate languages	[TMC+ 96]
90	Object Calculus	[AC96]
90	translucent types and modarity	[HL94, Ler94]
90	typed assembly language	[MWCG98]

1.5 *Pregled vsebine

Poglavje 2

SKLEPANJE

Teorija je definirana z množico pravil. Pravila lahko definirajo preverjanje tipov stavka danega jezika, definirajo interpretacijo jezika, ...

Sklepanje predstavimo s sekvenco aplikacij pravil na stavku, ki je podan kot vhodni parameter. Sklepanje je torej predstavljeno s sekvencami instanc pravil.

Vse instance pravil predstavljajo interpretacijo jezika, vsa možna obnašanja programov.

Sklepanje je torej tudi interpretacija stavka predstavljena kot sekvenca instanc pravil.

Sekvenca instanc pravil lahko predstavlja: preverjanje strukture stavkov, preverjanje tipov stavkov, interpretacija stavkov, itd.

2.1 Objekti in sodbe

Sodba je *izjava* o *objektih*, ki jih opazujemo. Izraz *sodba* velikokrat zamenjamo z izrazom *izjava*.

Uporabljali bomo najrazličnejše oblike sodb:

n nat	n je naravno število
$n = n_1 + n_2$	n je vsota n_1 in n_2
a ast	a je abstraktno sintaksno drevo
$\mathcal{T}$ type	$\mathcal{T}$ je tip
$e : \mathcal{T}$	izraz e je tipa $\mathcal{T}$
$e \Downarrow v$	izraz e ima vrednost v

(2.1)

Sodba (izjava) pravi, da ima eden ali več objektov neko lastnost ali obstaja razmerje med objekti

Lastnost sodbe ali samo relacijo definirano s sodbo imenujemo *forma sodbe*. Sodba o konkretnem objektu je *instanca* forme sodbe. Formo sodbe imenujemo tudi *predikat* in objekte, ki sodelujejo *subjekti*.

Metaspremenljivko P bomo uporabili za nespecificirano obliko sodbe in meta-spremenljivke $a, b, c, \ldots$ za nespecificirane objekte.

Zdaj lahko napišemo $a\ P$, kar pomeni da ima a lastnost P.

Ko ni pomembna vsebina sodbe uporabimo meta-spremenljivko J, ki določa nespecificirano sodbo.

Namerno bomo predstavili univerzum objektov bolj splošno. Dovolimo katerikoli objekt, ki ga definiramo s končnim postopkom.

Predpostavimo, da unverzum vsebuje vse objekte, ki jih je možno kreirati kot n-terico $(a_1, a_2, \ldots, a_n)$ objektov a_i.

Predpostavimo tudi, da univerzum vsebuje neskončno *simbolov*, ki so zaprti za označevanje.

2.2 Pravila sklepanja

Pravila služijo kot osnoven način za opisovanje pomena programskih jezikov.

V zadnjih desetletjih so se "pravila" zelo različno obravnavala: od enostavnih prehodov med stanji tranzicijskega sistema, pravil operacijske semantike, ki prav tako predstavljajo prehode med različnimi stanji, do pravil LF, ki so zapisane v logiki višjega reda.

Pravila lahko obravnavamo kot *logično implikacijo*. Osnova semantike programskih jezikov je torej spet logika! Izrazna moč pravil je ključnega pomena pri opisovanju semantike jezikov.

Tukaj si bomo ogledali pravila, ki imajo zelo splošno obliko. Pravila obravnavamo kot induktivne definicije forme sodbe.

Definicija forme sodbe J je sestavljena iz množice pravil oblike:

$$\frac{a_1\ J \quad a_2\ J \ldots a_k\ J}{a\ J} \tag{2.2}$$

Izjavo nad črto imenujemo *premisa*. Izjavo pod črto imenujemo *posledica*. Če pravilo nima premis ga imenujemo *aksiom*.

Pravilo beremo kot *logično implikacijo*, kjer so premise zadosten pogoj za sklepanje na posledico pravila. Da bi pokazali $a\ J$ je zadosti pokazati $a_1\ J, \ldots$

Lahko imamo več pravil z isto posledico vsako s svojimi premisami. V primeru, da velja posledica ni nujno, da veljajo premise vseh pravil.

Pravimo, da je sodba J zaprta za dano pravilo, če in samo če izjave $a_1\ J, \ldots, a_k\ J$ implicira $a\ J$. Forma sodbe je definirana z indukcijo nad pravili.

Pravila so potreben kot tudi zadosten pogoj za sklepanje na posledico na osnovi premis.

Primer 2.2.1 *Induktivna definicija sodbe a nat.*

$$\frac{}{\textit{zero nat}} \qquad \frac{a\ \textit{nat}}{\textit{succ}(a)\ \textit{nat}} \tag{2.3}$$

□

Primer 2.2.2 *Iduktivna definicija forme sodbe za drevo.*

$$\frac{}{\textit{empty tree}} \qquad \frac{a\ \textit{tree} \quad b\ \textit{tree}}{\textit{node}(a,b)\ \textit{tree}} \tag{2.4}$$

□

Primer 2.2.3 *Induktivna definicija enakosti naravnih števil.*

$$\frac{}{zero = zero\ \textit{nat}} \qquad \frac{a = b\ \textit{nat}}{\textit{succ}(a) = \textit{succ}(b)\ \textit{nat}} \tag{2.5}$$

□

Primer 2.2.4 *Podobno je definirana induktivno enakost nad drevesi.*

$$\frac{}{\textit{empty} = \textit{empty tree}} \qquad \frac{a_1 = b_1\ \textit{tree} \quad a_2 = b_2\ \textit{tree}}{\textit{node}(a_1,a_2) = \textit{node}(b_1,b_2)\ \textit{tree}} \tag{2.6}$$

□

Spremenljivke a in b imenujemo *metaspremeljivke*. Domena metaspremenljivk so objekti, ki so predmet definicije. Pravila so torej sheme, ki stojijo za neskončno mnogo konkretnih objektov.

2.3 Izpeljave

Pravilnost induktivne definicije sodbe lahko pokažemo tako, da konstruiramo *drevo izpeljave*. Drevo izpeljave dobimo s kompozicijo pravil, ki se začnejo z aksiomi in se zaključijo v sodbi.

Struktura izpeljave je drevo, ki je prikazano kot kopica s pravili naloženimi eno na drugo. Naj bo

$$\frac{a_1\ J \ldots a_k\ J}{a\ J} \tag{2.7}$$

pravilo izpelave in naj bodo $\nabla_1 \ldots \nabla_k$ izpeljave premis, potem je

$$\frac{\nabla_1 \ldots \nabla_k}{a\ J} \tag{2.8}$$

izpeljava posledice pravila $a\,J$. Na primer, izpeljava $\mathrm{succ}(\mathrm{succ}(\mathrm{succ}(\mathrm{zero})))\,\mathrm{nat}$ izgleda takole:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathrm{zero\ nat}}}{\mathrm{succ}(\mathrm{zero})\ \mathrm{nat}}}{\mathrm{succ}(\mathrm{succ}(\mathrm{zero}))\ \mathrm{nat}}}{\mathrm{succ}(\mathrm{succ}(\mathrm{succ}(\mathrm{zero})))\ \mathrm{nat}} \tag{2.9}$$

Podobno predstavlja naslednji izraz izpeljavo $\mathrm{node}(\mathrm{node}(\mathrm{empty}, \mathrm{empty}), \mathrm{empty})\ \mathrm{tree}$:

$$\dfrac{\dfrac{\dfrac{}{\mathrm{empty\ tree}} \quad \dfrac{}{\mathrm{empty\ tree}}}{\mathrm{node}(\mathrm{empty}, \mathrm{empty})\ \mathrm{tree}} \quad \dfrac{}{\mathrm{empty\ tree}}}{\mathrm{node}(\mathrm{node}(\mathrm{empty}, \mathrm{empty}), \mathrm{empty})\ \mathrm{tree}} \tag{2.10}$$

Da bi pokazali, da je neka izjava pravilna moramo poiskati izpeljavo.

Obstajata dve glavni metodi za izpeljave izrazov: *veriženje naprej* ali *konstrukcija od spodaj navzgor* in *veriženje nazaj* oz. *konstrukcija od zgoraj navzdol*.

Veriženje naprej začne z aksiomi in napreduje proti ciljnemu izrazu, medtem ko veriženje nazaj začne z izrazom in napreduje proti aksiomom.

Pri sklepanju naprej začnemo s prazno množico in iterativno širimo množico z uporabo pravil - posledico pravila dodamo množici v primeru, da se premise ujemajo. Proces se konča, ko je željena sodba v množici.

Če uporabljamo vsa pravila pri enem koraku postopka potem pričakujemo, da bomo našli izpeljavo, ne moremo pa zagotovo trditi, da bomo izpeljavo našli. Takšen postopek iskanja ni odločljiv (decidable).

Postopek izpeljave lahko dodaja posledice k množici, brez da bi prišel do izpeljave. Odločljivost posameznih izrazov lahko definiramo samo na osnovi poznavanja strukture jezika in splošnih lastnosti jezika.

Veriženje naprej je neusmerjeno iskanje, ker pri izvajanju koraka postopka ne upoštevamo cilj izpeljave.

Veriženje nazaj usmerjeno iskanje saj algoritem usmerjajo komponente ciljnega izraza.

Na vsaki fazi algoritma imamo seznam ciljev za katere iščemo izpeljave. Začetno vsebuje seznam samo ciljno izjavo. Vsaka faza odstani izjavo iz vrste in jo nadomesti s premisami pravil katerih posledica je dan cilj. Proces se konča, ko je vrsta prazna in smo dosegli vse cilje.

Enako kot pri veriženju naprej se nam lahko zgodi tudi pri veriženju nazaj, da ne uspemo najti izpeljave. V splošnem ne obstaja algoritmična metoda, ki bi nam povedala, da se neka izjava da izpeljati. Seznam ciljev se nam lahko širi ne da bi prišli do točke, kjer lahko rečemo, da je začetni cilj dosežen.

2.4 Indukcija pravil

Induktivno definirana sodba drži samo v primeru, da imamo kakšno izpeljavo s pravili.

Lastnosti sodb lahko dokažemo z uporabo indukacije pravil ali indukcije na izpeljavah.

Napišemo $P(J)$, če hočemo povedati, da lastnost P velja, če je sodba J izpeljiva.

Če hočemo pokazati, P velja za vse izpeljive sodbe J je zadosti, da pokažemo, da je P zaprta za pravila, ki definirajo J.

Za vsako pravilo oblike

$$\frac{J_1 \dots J_k}{J} \tag{2.11}$$

velja

$$\text{if } P(J_1) \dots P(J_k), \text{ then } P(J) \tag{2.12}$$

Konjunkcije lastnosti $P(J_1), \dots, P(J_k)$ imenujemo *induktivne hipoteze*. Dokaz implikacije same imenujemo *induktivni korak.*

Opazka 2.4.1 *Princip indukacije pravil je izraz dejstva, da je sodba J induktivno definirana z množico pravil in je najmočnejša sodba zaprta z danimi pravili.*

J je lahko izpeljiva samo zato, ker obstaja, kjer je J posledica in je vsaka premisa v pravilu tudi izpeljiva.

Po principu indukcije lahko predpostavljamo, da P velja za vse premise in potem pokažemo, da P velja tudi za J.

Če to naredimo za vsako pravilo, potem P mora veljati za vse sodbe J, ki so izpeljive.

V primeru, da pravilo nima premis potem velja P brez pogojev. □

Če je $P(J)$ zaprta za množico pravil, ki definirajo sodbo J, potem to velja tudi za $Q(J) = P(J) \wedge J$. J je avtomatično zaprta za pravila, ki jo definirajo.

To pomeni, da lahko v vsakem indukcijskem koraku predpostavimo, da velja J_i in $P(J_i)$ za vsako od premis pravila za izpeljavo $P(J)$ kot posledico.

Poglejmo si še notacijo. Če ima J obliko C za nek predikat C, včasih napišemo $PC(a)$, ali samo $P(a)$, namesto $P(a\ C)$ pri uporabljanju indukcije v dokazu. Za specifične lastnosti P pogosto uporabljamo ad-hoc notacijo katere pomen je razviden iz konteksta.

Zdaj lahko začnemo z *uporabo indukije* za dokazovanje lastnosti sodb. Najprej si poglejmo primer.

Primer 2.4.1 *Pri specializaciji na pravila 2.3 princip indukcije pravi, da je za veljavnost $P(a$ nat$)$ vedno ko a nat zadosti pokazati:*

1. $P($*zero nat*$)$.
2. $P($*succ*(a) *nat*$)$*, pri predpostavki* $P(a$ *nat*$)$.

To je soroden princip matematični indukciji in poseben primer indukcije pravil.

Primer 2.4.2 *Podobno velja tudi pri indukciji po pravilih 2.4 velja, da če želimo pokazati $P(a$ tree$)$ vedno ko a tree, je zadosti pokazati:*

1. $P($*empty tree*$)$.
2. $P($*node*$(a1; a2)$ *tree*$)$*, če predpostavimo* $P(a_1$ *tree*$)$ *in* $P(a_2$ *tree*$)$.

To imenujemo princip indukcije nad drevesi, ki je tudi primer indukcije pravil.

Primer 2.4.3 *Poglejmo si primer dokaza z indukcijo pravil, ki dokaže da je enakost naravnih števil definiranih s pravili 2.3 refleksivna.*

Lema 2.4.1 *Če a nat, potem $a = a$ nat.*

Dokaz. *Po indukciji na osnovi pravil 2.3:*

Pravilo 2.3a: *Z uporabo prvega pravila dobimo zero $=$ zero nat.*

Pravilo 2.3b: *Predpostavimo $a = a$ nat. Po drugem pravilu velja tudi succ$(a) =$ succ(a) nat.* □

Primer 2.4.4 *Poglejmo si še en primer uporabe indukcije pravil. Pokazali bomo, da je predhodnik naravnega števila tudi naravno število. Dokaz je enostaven, želimo pa pokazati kako je lastnost izpeljana iz začetnih principov.*

Lema 2.4.2 *Če succ(a) nat, potem a nat.*

Dokaz. *Poučno je trditev refolmulirati tako, da bo bolj primerna za induktivni dokaz: če velja b nat in velja da b je succ(a) za nek a, potem a nat. Uporabimo indukcijo po pravilih 2.3:*

Pravilo 2.3a: *Pravilno, ker zero ni oblike succ$(-)$.*

Pravilo 2.3b: *Velja b je succ(b'). Predpostavimo, da trditev velja za b' in velja b' nat. Rezultat sledi direktno. Vidimo, da zato ker velja succ$(b') =$ succ(a) za nek a, potem velja da a je b'.* □

Primer 2.4.5 *Pokažimo zdaj, da je operacija naslednik injektivna.*

Lema 2.4.3 *Če succ$(a_1) =$ succ(a_2) nat, potem $a_1 = a_2$ nat.*

Dokaz. *Spet je bolje prepisati trditev tako, da bomo lažje uporabljali indukcijo pravil.*

Pokazali bomo da če velja $b_1 = b_2\ nat$ *in če velja da* b_1 *je succ*(a_1) *in* b_2 *je succ*(a_2)*, potem velja* $a_1 = a_2$ *nat. Uporabili bomo indukcijo po pravilih 2.5:*

Pravilo 2.5a: *Pravilno, ker zero ni oblike succ*$(-)$.

Pravilo 2.5b: *Če predpostavimo rezultat* $b_1 = b_2$ *nat, in torej premiso* $b_1 = b_2$ *nat, potem pokažemo da: če succ*(b_1') *je succ*(a_1) *in succ*(b_2') *je succ*$(a2)$*, potem predpostavimo, da* $b_1' = b_2'$ *nat in po drugem pravilu 2.5 velja tudi succ*$(b_1') =$ *succ*(b_2') *nat. Torej velja* $b_1' = a_1$ *in* $b_2' = a_2$ *in tudi* $a_1 = a_2$ *nat, ker* $b_1' = b_2'$. □

Oba dokaza se naslanjata na nekatere naravne predpostavke o prostoru objektov.

2.4.1 Iterativne in simultane indukcijske definicije

Induktivne definicije so pogosto podane *iterativno*: ena induktivna definicija je definirana na drugi.

Primer 2.4.6 *Naslednji definiciji opišeta sodbo* a *list, ki pravi da je* a *seznam naravnih števil.*

$$\frac{}{nil\ \textit{list}} \tag{2.13}$$

$$\frac{a\ \textit{nat} \quad b\ \textit{list}}{\textit{cons}(a, b)\ \textit{list}} \tag{2.14}$$

Drugo pravilo vsebuje kot premiso sodbo a *nat, ki je definirana hkrati.* □

Simultana indukcijska definicija množice sodb $J_1, J_2, \ldots, J_n$ je opisana z množico med sabo povezanih pravil. Pravila definirajo vse sodbe hkrati. Vsako pravilo lahko referencira katerokoli sodbo, ki se definira.

Primer 2.4.7 *Poglejmo si naslednja pravila, ki hkrati (simultano) induktivno definirajo sodbe a even, ki pravi, da je a sodo naravno število, in sodba a odd, ki pravi, da je a liho naravno število:*

$$\frac{}{\textit{zero even}} \tag{2.15}$$

$$\frac{a\ \textit{odd}}{\textit{succ}(a)\ \textit{even}} \tag{2.16}$$

$$\frac{a\ \textit{even}}{\textit{succ}(a)\ \textit{odd}} \tag{2.17}$$

Primer 2.4.8 *Poglejmo si še en primer. Princip indukcije za ta pravila pravi, da bi pokazali $P(a\ \textit{even})$ vedno ko velja a even in $P(a\ \textit{odd})$ vedno ko velja a odd, je zadosti, da bi pokazali naslednje:*

1. *P (zero even);*
2. *če $P(a\ \textit{odd})$, potem $P(\textit{succ}(a)\ \textit{even})$;*
3. *če $P(a\ \textit{even})$, potem $P(\textit{succ}(a)\ \textit{odd})$.*

Ko prepišemo izraza z $P_{\textit{even}}(a)$ in $P_{\textit{odd}}(a)$, so pogoji naslednji:

1. *$P_{\textit{even}}(\textit{zero})$;*
2. *če $P_{\textit{odd}}(a)$, potem $P_{\textit{even}}(\textit{succ}(a))$.*
3. *če $P_{\textit{even}}(a)$, potem $P_{\textit{odd}}(\textit{succ}(a))$;*

2.4.2 Definicija funkcij s pravili

Pogosto uporabimo induktivne definicije za opis grafa, ki predstavlja funkcijo. Graf predstavlja razmerje med vhodi in izhodi s pomočjo premis in posledic pravil.

En način za definicijo seštevanja naravnih števil je induktivna definicija sodbe sum(a, b, c), katere pomen je $c = a + b$.

$$\frac{b \text{ nat}}{\text{sum}(zero, b, b)} \tag{2.18}$$

$$\frac{\text{sum}(a, b, c)}{\text{sum}(\text{succ}(a), b, \text{succ}(c))} \tag{2.19}$$

Pokazati želimo, da je c enolično določen z a in b.

Izrek 2.4.1 *Za vsak a nat in b nat obstaja natančno en c nat, kjer velja $c = a + b$.*

Dokaz. Veljati mora, da za vsak a nat in c nat obstaja natančno ena vrednost c nat, tako da sum(a, b, c). Dokaz se sestoji iz dveh delov:

1. (obstoj) Če a nat in b nat potem obstaja c nat tako, da velja sum(a, b, c).
2. (enoličnost) Če a nat in b nat potem obstaja natančno en c nat tako, da velja sum(a, b, c).

Pokažimo najprej obstoj. Naj bo $P(a \text{ nat})$ izjava: če b nat potem obstaja c nat tako, da sum(a, b, c). Dokažemo, da če velja a nat potem velja $P(a \text{ nat})$ z indukcijo po pravilih 2.3. Imamo dva primera.

Pravilo 2.3a Pokazali bomo $P(\text{zero nat})$. Če predpostavimo b nat in če vzamemo da je c enak b potem dobimo sum$(zero, b, c)$ po pravilu 2.18.

Pravilo 2.3b Predpostavimo, da $P(a \text{ nat})$, in pokažimo, da velja $P(\text{succ}(a) \text{ nat})$. Z drugimi besedami, predpostavimo, da če b nat potem obstaja c tako da sum(a, b, c), in pokazali bomo da če b' nat, potem obstaja c' tako, da sum$(\text{succ}(a), b', c')$. Predpostavimo najprej, da velja b' nat. Po indukciji obstaja c tako, da sum(a, b', c). Če vzamemo $c' = \text{succ}(c)$, in apliciramo pravilo 2.18 dobimo sum$(\text{succ}(a), b', c')$.

Pokažimo se unikatnost vrednosti c. Dokažemo, da če velja sum(a, b, c_1) in sum(a, b, c_2), potem velja $c_1 = c_2$ nat po indukciji na osnovi pravil 2.18.

Pravilo 2.18a Velja $a = \text{zero}$ in $c_1 = b$. Po indukciji na istih pravilih lahko pokažemo, da če sum(zero, b, c_2) potem c_2 je b. Po Lemi 2.4.1 dobimo $b = b$ nat.

Pravilo 2.18b Velja $a = succ(a')$ in $c_1 = \text{succ}(c_1')$, kjer sum(a', b, c_1'). Po notranji indukciji po istih pravilih lahko pokažemo, da če velja sum(a, b, c_2), potem velja $c_2 = \text{succ}(c_2')$ nat, kjer je sum(a', b, c_2'). Po zunanji indukcijski hipotezi velja $c_1' = c_2'$ nat in tudi $c_1 = c_2$ nat. □

2.5 Hipotetične sodbe

Kategorična sodba je brezpogojna izjava o nekem objektu univerzuma.

Hipotetična sodba je sestavljena na osnovi ene ali več hipotez ali predpostavk, ki izpeljejo zaključek.

Ogledali si bomo dve vrsti hipotetičnega sklepanja: izpeljevanje in dopustno sklepanje.

2.5.1 Izpeljivost

Ena forma hipotetičnega sklepanja izraža izrazljivost posledic iz danih predpostavk ali hipotez. Takšno sklepanje ima naslednjo obliko.

$$A_1, \ldots, A_n \vdash A. \tag{2.20}$$

Izjave $A_1, \ldots, A_n$ so *hipoteze* in A predstavlja *posledico*.

Pomen zgornje izjave: lahko izpeljemo A s kompozicijo pravil začevši z danimi predpostavkami kot začasnimi aksiomi. Z drugimi besedami, dokaz za pravilnost hipotetične sodbe je izpeljava zaključka s pomočjo instanc pravil iz danih predpostavk.

Primer 2.5.1 *Hipotetična sodba*

$$a\ \mathit{nat} \vdash \mathit{succ}(\mathit{succ}(a))\ \mathit{nat} \tag{2.21}$$

Premiso a nat uporabimo kot aksiom iz katerega izpeljemo posledico succ(succ(a)) nat na osnovi pravil 2.3. Tole je izpeljava posledice.

$$\dfrac{\dfrac{\dfrac{a\ \mathit{nat}}{\mathit{succ}(a)\ \mathit{nat}}}{\mathit{succ}(\mathit{succ}(a))\ \mathit{nat}}}{\mathit{succ}(\mathit{succ}(\mathit{succ}(a)))\ \mathit{nat}} \tag{2.22}$$

Zanimivo je, da je izpeljava popolnoma neodvisna od začetnega objekta a. Lahko bi vzeli namesto a nek poljuben objekt, npr. smet, in izpeljali naslednjo izjavo.

$$smet\ \mathit{nat} \vdash \mathit{succ}(\mathit{succ}(smet))\ \mathit{nat} \tag{2.23}$$

Sodbo $smet$ nat vzamemo kot aksiom iz katerega izpeljemo stavek z uporabo pravil za definicijo naravnih števil. □

Poglejmo si nekaj *strukturnih lastnosti* izpeljav.

Refleksivnost. Vsaka sodba je posledica same sebe: $A \vdash A$. Posledica je upravičena sama s seboj kot aksiom.

Omejevanje. Če $\Gamma \vdash A$ potem velja tudi $\Gamma, A' \vdash A$. Izpeljava A uporablja pravila iz Γ in se ne spremeni ob dodajanju novih sodb.

Tranzitivnost. Če $\Gamma, A' \vdash A$ in $\Gamma' \vdash A'$ potem $\Gamma, \Gamma' \vdash A$. Če dodamo dokaze za neko hipotezo premisi potem ta sodba (hipoteza) ni več potrebna med premisami.

Strukturne lastnosti se dajo izraziti kot pravila izpeljave, ki jih imenujemo *strukturna pravila izpeljave*.

$$\frac{}{\Gamma, A \vdash A} \tag{2.24}$$

$$\frac{\Gamma \vdash A}{\Gamma, A' \vdash A} \tag{2.25}$$

$$\frac{\Gamma \vdash A' \quad \Gamma, A' \vdash A}{\Gamma \vdash A} \tag{2.26}$$

Izpeljivost je relativno močen pogoj, ki je stabilen za razširjanje množice pravil, ki definirajo sodbo. Če je pravilo izpeljivo iz danih pravil potem je izpeljivo tudi iz razširjene množice. Dodatna pravila torej ne spremenijo obstoječih izpeljav.

Te lastnosti nima forma sklepanja, ki jo bomo ogledali v naslednji temi.

Teorija domen predstavlja več o urejenosti in strukturnih lastnostih prostora sodb. Predstavljena bo v enem od naslednjih poglavij.

2.5.2 Dopustno sklepanje

Druga oblika hipotetičnega sklepanja je *dopustno sklepanje*.

$$J \models K \tag{2.27}$$

Sodba K je izpeljiva na osnovi danih pravil vedno, ko je sodba J izpeljiva z isto množice pravil.

Dopustno sklepanje je enostavno pogojna izjava, ki pravi, da če je sodba J izpeljiva s pravili, potem je tudi sodba K.

Kot pri sklepanju z izpeljavo lahko tudi tukaj iteriramo dopustno sodbo tako, da zapišemo $J_1, \ldots, J_n \models J$, kar pomeni, če je J_1 izpeljiva,..., J_n izpeljiva, potem je tudi J izpeljiva.

Pravimo, da je naslednje pravilo dopustno, če in samo če $J_1, \ldots, J_n \models J$.

$$\frac{J_1, \ldots, J_n}{J} \tag{2.28}$$

Primer 2.5.2 *Za poljuben a nat velja naslednja dopustna izpeljava.*

$$\textit{succ}(a)\ \textit{nat} \models a\ \textit{nat} \tag{2.29}$$

Izjava je veljavna glede na pravila (2.3). To lahko pokažemo z indukcijo. Če velja succ(a) nat potem je moralo biti pri izpeljavi succ(a) nat uporabljeno drugo pravilo (2.3b). Toda potem mora veljati tudi a nat, ker je premisa pravila. □

Dopustno sklepanje je striktno šibkejše od izpeljevanja. Če velja $J_1, \ldots, J_n \vdash J$ potem velja tudi $J_1, \ldots, J_n \models J$. Obratno sploh ni nujno, da velja.

Poglejmo dokaz v levo stran. Če predpostavimo $J_1, \ldots, J_n \vdash J$, potem so $J_1, \ldots, J_n$ izpeljivi s pravili zapisano $\vdash J_1, \ldots, \vdash J_n$. Če upoštevamo pravila oslabitve in tranzitivnost, dobimo $\vdash J$, kar pomeni, da je J izpeljiv iz originalnih pravil.

Po drugi strani velja, da succ(a) nat $\nvdash a$ nat: ne obstaja način po katerem bi s pravili izpeljali a nat iz succ(a) nat.

Definicija dopustnega sklepanja pogojuje enake *strukturne lastnosti* kot hipotetično sklepanje. Veljajo naslednje lastnosti: refleksivnost, omejevanje in tranzitivnost.

Za razliko od hipotetičnega sklepanja, dopustno sklepanje je občutljivo na dodajanje pravil. Recimo, da smo razširili pravila (2.3) z naslednjim pravilom:

$$\frac{}{\text{succ}(smet)\ \text{nat}} \tag{2.30}$$

S to razširitvijo pravil ne velja več succ(a) nat $\models a$ nat, ker se novo dodano pravilo ujema z succ(a) nat v posledici, ne obstaja pa premisa, ki katere pravilnost bi dopustila a nat.

2.6 Opombe

Poglavje vsebuje prevode izbranih sekcij iz učbenika *Practical Foundations for Programming Languages* [5] avtorja Roberta Harperja.

Poglavje 3

SINTAKSA

3.1 Osnovni sintaktični objekti

Uporabljali bomo dve vrsti objektov: nize in abstraktno sintaksno drevo.

Nizi so primerna podatkovna struktura za linearno predstavitev jezka, niso pa primerni za analizo.

Abstraktno sintaksno drevo prikažejo hierarhično strukturo sintakse zato je primerno za semantično analizo.

3.1.1 Simboli

Uporabjali bomo množico simbolov, ki bodo služili v različnih vlogah: znaki, spremenljivke, imena polj, ...

Simbolom bomo včasih rekli imena ali atomi ali identifikatorji v odvisnosti od navad v konkretnem delovnem okolju.

O simbolih bomo razmišljali kot o atomih, ki nimajo strukture razen identitete.

Napišemo x sym kar pomeni, da je x simbol in predpostavljamo, da imamo neskončno mnogo simbolov za to identiteto.

Sodba $x \# y$, kjer je x sym in y sym, pove, da sta x in y različna simbola.

Uporabljali bomo različne razrede simbolov. V splošnem bomo predpostavili, da sta katerakoli razreda simbolov med sabo različna, tako da ne bo zmešnjav med njimi.

3.1.2 Nizi nad abecedo

Abeceda je končna množica znakov c_1 sym,...,c_n sym. Naj Σ označuje končno množico takšnih sodb.

Forma sodbe za $\Sigma \vdash s$ str, ki definira nize nad abecedo Σ, je induktivno definirana z naslednjimi pravili.

$$\frac{}{\Sigma \vdash \varepsilon \text{ str}} \tag{3.1}$$

$$\frac{\Sigma \vdash c \text{ sym} \quad \Sigma \vdash s \text{ str}}{\Sigma \vdash c \cdot s \text{ str}} \tag{3.2}$$

Niz je torej sekvenca znakov oz. null v primeru, da ne vsebuje znakov. Pogosto ne omenjamo Σ, če je abeceda jasna iz konteksta.

V primeru nizov uporabljamo indukcijo na sledeč način. Ko hočemo pokazati lastnost P nad nizom s je zadosti, da vidimo sledeče:

- $\epsilon\ P$
- Ob predpostavki $s\ P$ in c sym pokažemo $c \cdot s\ P$

To včasih imenujejmo princip *indukcije nad nizi*. Metoda je ekvivalentna indukciji na dolžino niza le da ni potrebno definirati dolžine niza.

Naslednje pravilo induktivno definira izjavo $s_1\hat{}s_2 = s$ str, ki pravi, da je s konkatenacija s_1 in s_2.

$$\frac{}{\epsilon\hat{}s \text{ str}} \tag{3.3}$$

$$\frac{s_1\hat{}s_2 = s \text{ str}}{(c \cdot s_1)\hat{}s_2 = c \cdot s \text{ str}} \tag{3.4}$$

Enostavno je pokazati z indukcijo, da pravila definirajo totalno funkcijo ($\forall\forall\exists!$) na dveh argumentih.

Nizi so običajno zapisani kot sekvenca simbolov npr. "abcd"je predstavitev niza $a \cdot (b \cdot (c \cdot (d \cdot e)))$.

3.1.3 Abstraktna sintaksna drevesa

Abstraktno sintaksno drevo (okr. ast) je urejeno drevo v katerem so vozlišča označena s simboli, ki jih imenujemo *operatorji*.

Signatura Ω je končna množica sodb oblike $ar(o) = n$, kjer velja o sym in n nat. Te sodbe priredijo številnost n operatorjem o, tako da če $\Omega \vdash ar(o) = n$ in $\Omega \vdash ar(o) = n'$ potem $n = n'$.

Razred abstraktnih sintaksnih dreves nad signaturo Ω je induktivno definiran na sledeč način.

$$\frac{\Omega \vdash ar(o) = n \quad a_1 \text{ ast} \ldots a_n \text{ ast}}{o(a_1, \ldots, a_n) \text{ ast}} \tag{3.5}$$

Osnovni primer induktivne definicije je operator s številnostjo nič. V tem primeru je premisa prazna. Takšna vozlišča drevesa so listi.

Strukturna indukcija

Princip strukturne indukcije je specializacija principa indukcije pravil na pravila, ki definirajo abstraktna sintaksna drevesa nad signaturami.

Če želimo pokazati $P(a \text{ ast})$ je zadosti, da pokažemo da je P zaprta glede na pravilo 3.5. Z drugimi besedami, če $\Omega \vdash ar(o) = n$, potem moramo pokazati, da

$$\text{if } P(a_1 \text{ ast}), \ldots, P(a_n \text{ ast}) \text{ then } P(o(a_1, \ldots, a_n) \text{ ast}). \tag{3.6}$$

V primeru, da je $n = 0$ se dokaz reducira na to, da pokažemo $P(o())$.

Primer 3.1.1 *Poglejmo si sledečo induktivno definicijo višine ast.*

$$\frac{hgt(a_1) = h_1 \ \ldots \ hgt(a_n) = h_n \quad max(h_1, \ldots, h_n) = h}{hgt(o(a_1, \ldots, a_n)) = \mathit{succ}(h)} \tag{3.7}$$

Z uporabo strukturne indukcije lahko pokažemo $(\forall, \exists!)$*, kar pomeni, da ima vsak ast unikatno višino.*

Predpostavimo lahko, da imajo vsa ast višine $1 \leq i \leq n$ *unikatno višino* $hgt(a_i) = h_i$*. Pokažemo lako, da je maksimum vseh unikatno definiran in je torej celotna višina unikatna.* □

Spremenljivke in substitucija

V praksi pogosto potrebujemo abstraktna sintaksna drevesa s spremenljivkami kot vrzeli, ki jih je mogoče napolniti z drugimi drevesi.

Spremenljivke so instancirane z uporabo substitucije spremenljivk ast z dugim drevesom.

Poglejmo si najprej nekaj dogovorov v povezavi z notacijo. Naj bo $\mathcal{X} = x_1$ ast, $\ldots, x_n$ ast množica parametrov in hkrati seznam hipotez $\{x_1, \ldots, x_n\} \mid x_1$ ast, $\ldots, x_n$ ast, kjer velja tudi x_1 sym, $\ldots, x_n$ sym. Naprej, naj izraz $x \# \mathcal{X}$ pomeni $x \notin \{x_1, \ldots, x_n\}$.

Z uporabo te notacije lahko napišemo sodbo $\mathcal{X} \vdash a$ ast, ki je induktivno definirana z naslednjimi pravili:

$$\frac{}{\mathcal{X}, x \text{ ast} \vdash x \text{ ast}} \tag{3.8}$$

$$\frac{\Omega \vdash ar(o) = n \quad \mathcal{X} \vdash a_1 \text{ ast} \ldots \mathcal{X} \vdash a_n \text{ ast}}{\mathcal{X} \vdash o(a_1, \ldots, a_n) \text{ ast}} \tag{3.9}$$

Po principu indukcije pravil je za dokaz $P(\mathcal{X} \vdash a \text{ ast})$ zadosti, da pokažemo:

1. $P(\mathcal{X}, x \text{ ast} \vdash x \text{ ast})$.
2. Če $\Omega \vdash ar(o) = n$ in če $P(\mathcal{X} \vdash a_1 \text{ ast}), \ldots, P(\mathcal{X} \vdash a_n \text{ ast})$, potem $P(\mathcal{X} \vdash o(a_1, \ldots, a_n) \text{ ast})$.

Parametri $\mathcal{X}$ se obravnavajo kot atomični objekti, vendar ima vsak svoje abstraktno sintaksno drevo.

Definiramo sodbo $\mathcal{X} \vdash [a/x]b = c$, kar pomeni da je c rezultat substitucije a za x v b z uporabo naslednjih pravil.

$$\frac{}{\mathcal{X}, x \text{ ast} \vdash [a/x]x = a} \tag{3.10}$$

$$\frac{x \# y}{\mathcal{X}, x \text{ ast}, y \text{ ast} \vdash [a/x]y = y} \tag{3.11}$$

$$\frac{\mathcal{X} \vdash [a/x]b_1 = c_1 \ldots \mathcal{X} \vdash [a/x]b_n = c_n}{\mathcal{X} \vdash [a/x]o(b_1, \ldots, b_n) = o(c_1, \ldots, c_n)} \tag{3.12}$$

Rezultat substitucije je enolično definiran z argumenti. Kot posledico lahko napišemo $[a/x]b$ za unikaten c, tako da $[a/x]b = c$.

Izrek 3.1.1 *Če velja $\mathcal{X} \vdash a$ ast in $\mathcal{X}, x$ ast $\vdash b$ ast, kjer $x \# \mathcal{X}$, potem obstaja enoličen c, tako da $\mathcal{X} \vdash [a/x]b = c$ in $\mathcal{X} \vdash c$ ast.*

Dokaz. Trditev dokažemo s strukturno indukcijo na b relativno na kontekst $\mathcal{X}, x$ ast.

Imamo tri primere za obravnavo:

1. Ker velja $\mathcal{X}, x$ ast $\vdash x$ ast, moramo pokazati da obstaja unikaten c tako da $\mathcal{X} \vdash [a/x]x = c$. Z uporabo pravila 3.10a vidimo da je zamenjava c z a potrebna in zadostna.
2. Če $\mathcal{X}, x$ ast, y ast $\vdash y$ ast za nek $y \# x$, potem po pravilu 3.10b zamenjava c z y je potrebna in zadostna.
3. Končno, če $b = o(b_1, \ldots, b_n)$, potem velja po indukciji, da obstaja enoličen $c_1, \ldots, c_n$ tako da $\mathcal{X} \vdash [a/x]b_1 = c_1, \ldots, \mathcal{X} \vdash [a/x]b_n = c_n$. Po pravilu 3.10c je edina možna izbira za c drevo $o(c_1, \ldots, c_n)$, kar zadostuje. □

3.2 Konkretna sintaksa

Konkretna sintaksa jezika se uporablja za predstavitev gramatike z navadnim tekstom.

Velikokrat uporabimo konktretno sintakso za izboljšanje berljivosti in izločanje dvoumnosti.

Imamo kar nekaj metod za odpravljanje dvoumnosti medtem ko je berljivost velikokrat odvisna od okusa.

V tej sekciji predstavimo glavne metode za specifikacijo konkretne sintakse.

Uporabljali bomo jezik izrazov $\mathcal{L}(\text{nat str})$, ki predstavi elementarno aritmetiko na naravnih številih, enostavne operacije na nizih in enostavno delo s spremenljivkami.

3.2.1 Leksikalna struktura

Prva faza sintaktičnega procesiranja je pretvorba iz predstavitve osnovane na znakih v predstavitev osnovani na simbolih. Ta proces običajno imenujemo *leksikalna analiza*.

Osnovna ideja je združiti znake v simbole, ki so osnova za naslednje faze analize. Na primer, niz znakov 467 se spemeni v celo število 467.

Podobno predstavimo identifikator sestavljen iz niza znakov “temp” v spremenljivko z imenom $temp$.

Prav tako počistimo vse “bele znake” (presledke, tabulatorje, itd.) in komentarje, ki se ne uporabljajo v nadalnji analizi.

Znakovna predstavitev je v največ primerih definirana z regularnimi izrazi. Leksikalna struktura $\mathcal{L}(\text{nat str})$ je določena z naslednjimi pravili.

$$
\begin{array}{llll}
Item & itm & ::= & kwd \mid id \mid num \mid lit \mid spl \\
Keyword & kwd & ::= & l\,.\,e\,.\,t\,.\,\epsilon \mid b\,.\,e\,.\,\epsilon \mid i\,.\,n \\
Identifier & id & ::= & ltr\ (ltr \mid dig)^* \\
Numeral & num & ::= & dig\ dig^* \\
Literal & lit & ::= & qum\ (ltr \mid dig)\ qum \\
Special & spl & ::= & + \mid\ *\ \mid \hat{}\ \mid (\mid) \mid\ \mid \\
Letter & ltr & ::= & a \mid b \mid ... \\
Digit & dig & ::= & 0 \mid 1 \mid ... \\
Quote & qum & ::= & ”
\end{array}
\tag{3.13}
$$

Leksikalni simbol je bodisi ključna beseda, identifikator, število, niz znakov ali posebni simbol. Simboli se med seboj razlikujejo po sestavi znakov, ki jih posamezna vrsta simbola vsebuje.

Nizovni simboli vsebujejo sekvence natisljivih znakov, ki so obdani z narekovaji.

Delo leksikalnega analizatorja je pretvorba nizov znakov v niz simbolov. Zgornja definicija sintakse služi kot vodilo pri pretvorbi.

Vhoden niz znakov se pretvarja v niz simbolov z uporabo naslednjih pravil, ki definirajo simbolni jezik razčlenjevalnika oz. statično semantiko jezika.

$$\frac{s\ str}{\overline{ID[s]\ tok}} \tag{3.14}$$

$$\frac{n\ nat}{\overline{NUM[n]\ tok}} \tag{3.15}$$

$$\frac{s\ str}{\overline{LIT[s]\ tok}} \tag{3.16}$$

$$\overline{\overline{LET\ tok}} \tag{3.17}$$

$$\overline{\overline{BE\ tok}} \tag{3.18}$$

$$\overline{\overline{IN\ tok}} \tag{3.19}$$

$$\overline{\overline{ADD\ tok}} \tag{3.20}$$

$$\overline{\overline{MUL\ tok}} \tag{3.21}$$

$$\overline{\overline{CAT\ tok}} \tag{3.22}$$

$$\overline{\overline{LP\ tok}} \tag{3.23}$$

$$\overline{\overline{RP\ tok}} \tag{3.24}$$

$$\overline{\overline{VB\ tok}} \tag{3.25}$$

Lesikalna analiza je definirana s pretvorbo nizov znakov v nize simbolov. Pretvorba je induktivno definirana s pomočjo naslednjih pravil:

$$\begin{array}{ll} s\ inp \longleftrightarrow t\ tokstr & Scan\ input \\ s\ itm \longleftrightarrow t\ tok & Scan\ an\ item \\ s\ kwd \longleftrightarrow t\ tok & Scan\ a\ keyword \\ s\ id \longleftrightarrow t\ tok & Scan\ an\ identifier \\ s\ num \longleftrightarrow t\ tok & Scan\ a\ number \\ s\ spl \longleftrightarrow t\ tok & Scan\ a\ symbol \\ s\ lit \longleftrightarrow t\ tok & Scan\ a\ string\ literal \\ s\ whs & Skip\ white\ space \end{array} \tag{3.26}$$

Definicija teh form, ki sledijo, uporablja številne pomožne sodbe, ki ustrezajo klasifikaciji znakov in leksikalni strukturi jezika.

Na primer, $s\ whs$ pravi da se niz s sestoji samo iz "belih znakov", izjava $s\ lord$ pravi, da se s sestoji iz alfanumeričnih znakov.

$$\frac{}{\epsilon\ inp \longleftrightarrow \epsilon\ tokstr} \tag{3.27}$$

$$\frac{s = s_1\hat{}s_2\hat{}s_3\ str \quad s_1\ whs \quad s_2\ itm \longleftrightarrow t\ tok \quad s_3\ inp \longleftrightarrow ts\ tokstr}{s\ inp \longleftrightarrow t\ .\ ts\ tokstr} \tag{3.28}$$

$$\frac{s\ kwd \longleftrightarrow t\ tok}{s\ itm \longleftrightarrow t\ tok} \tag{3.29}$$

$$\frac{s\ id \longleftrightarrow t\ tok}{s\ itm \longleftrightarrow t\ tok} \tag{3.30}$$

$$\frac{s\ num \longleftrightarrow t\ tok}{s\ itm \longleftrightarrow t\ tok} \tag{3.31}$$

$$\frac{s\ lit \longleftrightarrow t\ tok}{s\ itm \longleftrightarrow t\ tok} \tag{3.32}$$

$$\frac{s\ spl \longleftrightarrow t\ tok}{s\ itm \longleftrightarrow t\ tok} \tag{3.33}$$

$$\frac{s = l\ .\ e\ .\ t\ .\ \epsilon\ str}{s\ kwd \longleftrightarrow LET\ tok} \tag{3.34}$$

$$\frac{s = b\ .\ e\ .\ \epsilon\ str}{s\ kwd \longleftrightarrow BE\ tok} \tag{3.35}$$

$$\frac{s = i\ .\ n\ .\ \epsilon\ str}{s\ kwd \longleftrightarrow IN\ tok} \tag{3.36}$$

$$\frac{s = s_1 \hat{} s_2\ str \quad s_1\ ltr \quad s_2\ lord}{s\ id \longleftrightarrow ID[s]\ tok} \tag{3.37}$$

$$\frac{s = s_1 \hat{} s_2\ str \quad s_1\ dig \quad s_2\ dgs \quad s\ num \longleftrightarrow n\ nat}{s\ num \longleftrightarrow NUM[n]\ tok} \tag{3.38}$$

$$\frac{s = s_1 \hat{} s_2 \hat{} s_3\ str \quad s_1\ qum \quad s_2\ lord \quad s_3\ qum}{s\ lit \longleftrightarrow LIT[s_2]\ tok} \tag{3.39}$$

$$\frac{s = +\ \centerdot\ \epsilon\ str}{s\ spl \longleftrightarrow ADD\ tok} \tag{3.40}$$

$$\frac{s = *\ \centerdot\ \epsilon\ str}{s\ spl \longleftrightarrow MUL\ tok} \tag{3.41}$$

$$\frac{s = \hat{}\ \centerdot\ \epsilon\ str}{s\ spl \longleftrightarrow CAT\ tok} \tag{3.42}$$

$$\frac{s = (\ \centerdot\ \epsilon\ str}{s\ spl \longleftrightarrow LP\ tok} \tag{3.43}$$

$$\frac{s =)\ \centerdot\ \epsilon\ str}{s\ spl \longleftrightarrow RP\ tok} \tag{3.44}$$

$$\frac{s = |\ \centerdot\ \epsilon\ str}{s\ spl \longleftrightarrow VB\ tok} \tag{3.45}$$

Pravilo 3.37 velja samo, če ne velja nobeno izmed pravil od 3.34 do 3.36. Gledano tehnično ima pravilo 3.37 premise, ki izključijo ključne besede kot možene identifikatorje.

3.2.2 Kontekstno neodvisne gramatike

Standardna metoda za definicijo konkretne sintakse je uporaba kontekstno neodvisne gramatike za jezik. Gramatika se sestoji iz treh komponent:

1. žetonov ali terminalnih simbolov nad katerimi je gramatika definirana,
2. sintaktičnih razredov ali neterminalnih simbolov, ki se razlikujejo od terminalnih simbolov in
3. pravil ali produkcij oblike $A ::= \alpha$, kjer je A neterminalen simbol in je α niz terminalnih in neterminalnih simbolov.

Vsak sintaktični razred je kolekcija nizov žetonov. Pravila povejo kateri niz spada k katerem sintaktičnem razredu.

Pri definiciji gramatike pogosto okrajšamo del produkcij.

$$\begin{aligned} A &::= \alpha_1 \\ &\vdots \\ A &::= \alpha_n \end{aligned} \tag{3.46}$$

Vsaka produkcija ima enak levi del. Če sestavimo zgornje produkcije v eno samo dobimo:

$$A ::= \alpha_1 \mid ... \mid \alpha_n, \tag{3.47}$$

Pravilo definira množico alternativ za razvoj sintaktičnega razreda A.

Kontekstno neodvisna gramatika določa hkratno induktivno definicijo množice sintaktičnih razredov. Vsak neterminalni simbol vidimo kot sodbo $s\ A$ nad nizi terminalnih simbolov.

Neterminalni simboli imajo dvojno vlogo: a) vlogo spremenljivk v produkciji in b) vlogo lastnosti oz. razreda objektov.

Vsaki produkciji naslednje oblike

$$A ::= s_1\ A_1\ s_2\ \ldots\ s_n\ A_n\ s_{n+1} \tag{3.48}$$

priredimo pravilo izpeljave

$$\frac{s'_1\ A_1\ \ldots s'_n\ A_n}{s_1\ s'_1\ s_2\ \ldots\ s_n\ s'_n\ s_{n+1}\ A.} \tag{3.49}$$

Množica takšnih pravil sestavlja induktivno definicijo sintaktičnih razredov gramatike.

Spet se spomnimo, da je postavljanje nizov skupaj krajši zapis za konkatenacijo nizov. Pravilo lahko napišemo takole.

$$\frac{s'_1\ A_1\ \ldots\ s'_n\ A_n \quad s = s_1\hat{}s'_1\hat{}s_2\hat{}\ldots\hat{}s_n\hat{}s'_n\hat{}s_{n+1}}{s\ A} \tag{3.50}$$

Formulacija pove, da A velja če lahko s razdelimo kot je opisano zgoraj: $s'_i\ \mathcal{A}_i$ za $1 \leq i \leq n$. Ker konkatenacija nizov ni enolično inverzibilna, dekompozicija niza ni unikatna in imamo lahko več različnih načinov po katerih ovrednotimo pravilo.

3.2.3 Gramatiča struktura

Konkretno sintakso $\mathcal{L}(\text{nat str})$ lahko definiramo s kontekstno neodvisno gramatiko nad simboli, ki so predstavljeni v 3.13. Gramatika ima samo en sintaktični razred, exp, ki je definiran z naslednjimi pravili:

$$\begin{array}{llll}
Expression & exp & ::= & num \mid lit \mid id \mid LP\ exp\ RP \mid exp\ ADD\ exp \mid \\
 & & & exp\ MUL\ exp \mid exp\ CAT\ exp \mid VB\ exp\ VB \mid \\
 & & & LET\ id\ BE\ exp\ IN\ exp \\
Number & num & ::= & NUM[n] \quad (n\ nat) \\
String & lit & ::= & LIT[s] \quad (s\ str) \\
Identifier & id & ::= & ID[s] \quad (s\ str)
\end{array} \tag{3.51}$$

Ta gramatika uporablja nekatere standardne konvencije za izboljšanje berljivosti: žeton identificiramo s spremenljivko in uporabljamo stik nizov za izražanje konkatenacije.

Če prikažemo interpretacijo gramatike z induktivnimi definicijami, dobimo naslednja pravila.

$$\frac{s\ num}{s\ exp} \tag{3.52}$$

$$\frac{s\ lit}{s\ exp} \tag{3.53}$$

$$\frac{s\ id}{s\ exp} \tag{3.54}$$

$$\frac{s_1\ exp \quad s_2\ exp}{s_1\ ADD\ s_2\ exp} \tag{3.55}$$

$$\frac{s_1\ exp \quad s_2\ exp}{s_1\ MUL\ s_2\ exp} \tag{3.56}$$

$$\frac{s_1\ exp\ s_2\ exp}{s_1\ CAT\ s_2\ exp} \tag{3.57}$$

$$\frac{s\ exp}{VB\ s \quad VB\ exp} \tag{3.58}$$

$$\frac{s\ exp}{LP\ s \quad RP\ exp} \tag{3.59}$$

$$\frac{s1\ id \quad s2\ exp \quad s3\ exp}{LET\ s1\ BE\ s2\ IN\ s3\ exp} \tag{3.60}$$

$$\frac{n\ nat}{NUM[n]\ num} \tag{3.61}$$

$$\frac{s\ str}{LIT[s]\ lit} \tag{3.62}$$

$$\frac{s\ str}{ID[s]\ id} \tag{3.63}$$

Da bi poudarili vlogo konkatenacije nizov bi lahko napisali pravilo 3.56 na naslednji način.

$$\frac{s = s1\ MUL\ s2\ str \quad s1\ exp \quad s2\ exp}{s\ exp} \tag{3.64}$$

Izraz $s\ exp$ je izpeljiv, če je s konkatenacija niza s_1, znaka za množenje in s_2, kjer velja $s_1\ exp$ in $s_2\ exp$.

3.3 Abstraktna sintaksa

Konkretna sintaksa jezika definira linearno predstavitev jezikovnih konstruktov. Program je predstavljen kot niz simbolov.

Osnovni namen konkretne sintakse je izboljšati berljivost jezika kot tudi primernost za obravnavo jezikov s standardnimi orodji za programiranje..

Programski jeziki so tako predmet študije kot tudi instrument za izražanje. Pri matematični obravnavi nas zanima pomen jezika in ne toliko površinske predstavitve.

Abstraktna sintaksa jezika prikaže hierarhično in povezovalno strukturo jezika ter zanemari linearno notacijo konkretne sintakse.

Razčlenjevanje je proces prevajanja iz konkretne sintakse v abstraktno sintakso. Sestoji se iz analize linearne predstavitve jezika preko gramatike jezika ter prevajanja linearne predstavitve v abstraktno sintaksno drevo ali podobno obliko abstraktnega zapisa.

3.3.1 Abstraktna sintaksna drevesa

Abstraktno sintaksno drevo jezika $\mathcal{L}(\text{nat str})$ je definirano z naslednjo signaturo:

$$\begin{array}{l} ar(num[n]) = 0 \quad (n\ nat) \\ ar(str[s]) = 0 \quad (s\ str) \\ ar(id[s]) = 0 \quad (s\ str) \\ ar(plus) = 2 \\ ar(times) = 2 \\ ar(cat) = 2 \\ ar(len) = 1 \\ ar(let[s]) = 2 \end{array} \tag{3.65}$$

Vsak identifikator obravnavamo kot operator s številnostjo 0. Konstruktor 'let' je obrav-

navan kot družina operacij, ki imajo številnost 2 in so indeksirani z identifikatorjem, ki ga povezuje.

Pri specializaciji pravil za abstraktno sintaksna drevesa na predstavljeno signaturo dobimo naslednje induktivne definicije abstraktne sintakse $\mathcal{L}(\text{nat str})$:

$$\frac{n\ nat}{num[n]\ ast} \tag{3.66}$$

$$\frac{s\ str}{str[s]\ ast} \tag{3.67}$$

$$\frac{s\ str}{id[s]\ ast} \tag{3.68}$$

$$\frac{a_1\ ast \quad a_2\ ast}{plus(a_1; a_2)\ ast} \tag{3.69}$$

$$\frac{a_1 ast \quad a_2\ ast}{times(a_1; a_2)\ ast} \tag{3.70}$$

$$\frac{a_1\ ast \quad a_2\ ast}{cat(a_1; a_2)\ ast} \tag{3.71}$$

$$\frac{a\ ast}{len(a)\ ast} \tag{3.72}$$

$$\frac{s\ id \quad a_1\ ast \quad a_2\ ast}{let[s](a_1; a_2)\ ast} \tag{3.73}$$

Zadnje pravilo je specializacija pravila za 'let' stavek. Zahtevamo, da je prvi argument 'let' stavka identifikator.

3.3.2 Razčlenjevanje v ASD

Proces prevajanja iz konkretne sintakse v abstraktno sintakso se imenuje *razčlenjevanje* (angl. parsing).

Razčlenjevanje bomo obravnavali kot sodbo med konkretno in abstraktno sintakso jezika. Za to sodbo bo veljalo ($\forall, \exists!$) za vse nize in ast.

Razčlenjevanje je torej parcialna funkcija med nizi in ast. Nedefinirana je za nize, ki nimajo pravilne gramatične strukture, preostale (pravilno strukturirane) nize pa enolično prevede v ast.

V nadaljevanju si bomo ogledali sodbe za razčlenjevanje jezika $\mathcal{L}(\text{nat str})$.

$$\begin{array}{ll} s\ prg \longleftrightarrow a\ ast & \text{Parse as a program} \\ s\ exp \longleftrightarrow a\ ast & \text{Parse as an expression} \\ s\ trm \longleftrightarrow a\ ast & \text{Parse as a term} \\ s\ fct \longleftrightarrow a\ ast & \text{Parse as a factor} \\ s\ num \longleftrightarrow a\ ast & \text{Parse as a number} \\ s\ lit \longleftrightarrow a\ ast & \text{Parse as a literal} \\ s\ id \longleftrightarrow a\ ast & \text{Parse as an identifier} \end{array} \tag{3.74}$$

Sodbe za razčlenjevanje so definirane induktivno na osnovi sledečih pravil:

$$\frac{n\ nat}{NUM[n]\ num \longleftrightarrow num[n]\ ast} \tag{3.75}$$

$$\frac{s\ str}{LIT[s]\ lit \longleftrightarrow str[s]\ ast} \tag{3.76}$$

$$\frac{s\ str}{ID[s]\ id \longleftrightarrow id[s]\ ast} \tag{3.77}$$

$$\frac{s\ num \longleftrightarrow a\ ast}{s\ fct \longleftrightarrow a\ ast} \tag{3.78}$$

$$\frac{s\ lit \longleftrightarrow a\ ast}{s\ fct \longleftrightarrow a\ ast} \tag{3.79}$$

$$\frac{s\ id \longleftrightarrow a\ ast}{s\ fct \longleftrightarrow a\ ast} \tag{3.80}$$

$$\frac{s\ prg \longleftrightarrow a\ ast}{LP\ s\ RP\ fct \longleftrightarrow a\ ast} \tag{3.81}$$

$$\frac{s\ fct \longleftrightarrow a\ ast}{s\ trm \longleftrightarrow a\ ast} \tag{3.82}$$

$$\frac{s_1\ fct \longleftrightarrow a_1\ ast \quad s_2\ trm \longleftrightarrow a_2\ ast}{s_1\ MUL\ s_2\ trm \longleftrightarrow times(a_1; a_2)\ ast} \tag{3.83}$$

$$\frac{s\ fct \longleftrightarrow a\ ast}{VB\ s\ VB\ trm \longleftrightarrow len(a)\ ast} \tag{3.84}$$

$$\frac{s\ trm \longleftrightarrow a\ ast}{s\ exp \longleftrightarrow a\ ast} \tag{3.85}$$

$$\frac{s_1\ trm \longleftrightarrow a_1\ ast \quad s_2\ exp \longleftrightarrow a_2\ ast}{s_1\ ADD\ s_2\ exp \longleftrightarrow plus(a_1; a_2)\ ast} \tag{3.86}$$

$$\frac{s_1\ trm \longleftrightarrow a_1\ ast \quad s_2\ exp \longleftrightarrow a_2\ ast}{s_1\ CAT\ s_2\ exp \longleftrightarrow cat(a1; a2)ast} \tag{3.87}$$

$$\frac{s\ exp \longleftrightarrow a\ ast}{s\ prg \longleftrightarrow a\ ast} \tag{3.88}$$

$$\frac{s_1\ id \longleftrightarrow id[s]\ ast \quad s_2\ exp \longleftrightarrow a2ast \quad s3\ prg \longleftrightarrow a3ast}{LET\ s_1\ BE\ s_2\ IN\ s_3\ prg \longleftrightarrow let[s](a2; a3)ast} \tag{3.89}$$

Uspešno razčlenjvanje pogojuje, da se niz žetonov razčleni v skladu s pravili gramatike, ki ni dvoumna, in da je rezultat razčlenjevanja dobro-definirano abstraktno sintaktično drevo.

Izrek 3.3.1 *Če $s\ prg \longleftrightarrow a\ ast$, potem $s\ prg$ in $a\ ast$. Enako velja za vsa ostale sodbe.*

Dokaz. Uporaba indukcije pravil po pravilih 3.75-3.89. □

Velja celo več, če je niz generiran po pravilih gramatike, potem ga je mogoče razčleniti v ast.

Izrek 3.3.2 *Če $s\ prg$, potem obstaja enoličen a tako da $s\ prg \longleftrightarrow a\ ast$. Enako velja za ostale sodbe, ki opišejo razčlenjevanje: sodbe imajo lastnost ($\forall\exists!$) za vse dobro-definirane nize in ast.*

Dokaz. Po indukciji pravil določenih z branjem gramatike 3.51 kot induktivne definicije. □

Končno, vsak zapis abstraktne sintakse je mogoče preoblikovati v niz simbolov, ki se razčleni v skladu z danim ast.

Izrek 3.3.3 *Če velja $a\ ast$, potem obstaja (ni nujno enolično) niz s tako, da velja $s\ prg$ in $s\ prg \longleftrightarrow a\ ast$. Z drugimi besedami, sodba za razčlenjevanje ima lastnost ($\exists\forall$).*

Dokaz. Z indukcijo po pravilih gramatike 3.51. □

3.4 Primer enostavnega jezika

V tej sekciji bo predstavljen primer enostavnega jezika $\mathcal{L}(\text{natbool})$, leksikalna struktura $\mathcal{L}(\text{nat bool})$, leksikalna analiza $\mathcal{L}(\text{nat bool})$ ter gramatika $\mathcal{L}(\text{nat bool})$ v obliki pravil.

3.4.1 Jezik $\mathcal{L}(\textbf{nat bool})$

Ogledali si bomo zelo enostaven jezik, ki vsebuje samo nekatere izmed sintaktičnih oblik:

- boolove konstante $true$ in $false$,
- pogojni izraz if $-$ then $-$ else,
- numerično konstanto 0,
- aritmetične operacije $pred$ in $succ$ ter
- pogojni izraz iszero, ki preveri al ima izraz vrednost 0.

Gramatika $\mathcal{L}(\text{nat bool})$ je definirana z naslednjimi izrazi.

$$\begin{array}{lll} \text{Izrazi } t & ::= & \text{true} \\ & & \text{false} \\ & & \text{if } t \text{ then } t \text{ else } t \\ & & 0 \\ & & \text{succ}(t) \\ & & \text{pred}(t) \\ & & \text{iszero}(t) \end{array} \tag{3.90}$$

Konvencija uporabljena pri opisu jezika je podobna BNF obliki. Zacetek opisa t ::= napove definicijo izrazov in definira spremenljivko t, katere zaloga vrednosti so izrazi.

Na desni strani se vrstijo možne oblike izrazov. Spremenljivko t imenujemo *metaspremenljivka*.

Primer 3.4.1 *Poglejmo si nekaj primerov izrazov v predstavljenem jeziku.*

$$\begin{array}{l} \textit{succ}(\textit{succ}(\textit{succ}(0))) \\ \blacktriangleright\ 3 \\ \textit{if } false \textit{ then } 0 \textit{ else } 1 \\ \blacktriangleright\ 1 \\ \textit{iszero}(\textit{pred}(\textit{succ}(0))) \\ \blacktriangleright\ true \end{array} \tag{3.91}$$

- *Najprej, cela števila predstavimo z večkratnim vgnezdenjem operacije succ ter pred.*
- *Drugi primer prikaže uporabo if stavka.*
- *Tretji primer uporabi operacijo iszero nad celim številom.*

□

Zgornja gramatika definira abstraktno sintakso. Z uporabo oklepajev lahko enostavno razrešimo marsikatero dvoumnost v izrazih.

Rezultat evaluacije izrazov so vedno enostavne vrednosti: boolove vrednosti ali cela števila. Kasneje si bomo ogledali pravila za evaluacijo izrazov.

3.4.2 Leksikalna analiza

Leksikalna struktura $\mathcal{L}$(**nat bool**)

Znakovna predstavitev je v največ primerih definirana z regularnimi izrazi. Leksikalna struktura $\mathcal{L}(\text{nat bool})$ je določena z naslednjimi pravili.

$$\begin{array}{llll} Item & itm & ::= & kwd \mid boo \mid zer \mid spl \\ Keyword & kwd & ::= & i\,.\,f\,.\,\epsilon \mid \\ & & & t\,.\,h\,.\,e\,.\,n\,.\,\epsilon \mid \\ & & & s\,.\,u\,.\,c\,.\,c\,.\,\epsilon \mid \\ & & & p\,.\,r\,.\,e\,.\,d\,.\,\epsilon \mid \\ & & & i\,.\,s\,.\,z\,.\,e\,.\,r\,.\,o\,.\,\epsilon \mid \\ Bool & boo & ::= & true \mid false \\ Special & spl & ::= & (\mid) \\ Zero & zer & ::= & 0 \end{array} \tag{3.92}$$

Jezik žetonov $\mathcal{L}$(**nat bool**)

Vhoden niz znakov se pretvarja v niz simbolov (žetonov) z uporabo naslednjih pravil, ki definirajo simbolni jezik razčlenjevalnika oz. statično semantiko jezika.

$$\frac{}{ZERO\ tok} \tag{3.93}$$

$$\frac{l \text{ bool}}{BOOL[l]\ tok} \tag{3.94}$$

$$\frac{}{IF\ tok} \tag{3.95}$$

$$\frac{}{THEN\ tok} \tag{3.96}$$

$$\frac{}{ELSE\ tok} \tag{3.97}$$

$$\frac{}{SUCC\ tok} \tag{3.98}$$

$$\frac{}{PRED\ tok} \tag{3.99}$$

$$\frac{}{ISZERO\ tok} \tag{3.100}$$

$$\frac{}{LP\ tok} \tag{3.101}$$

$$\frac{}{RP\ tok} \tag{3.102}$$

Pretvorba $\mathcal{L}$(nat bool) v nize žetonov

Naslednja pravila predstavlja leksikalno razčlenjevanje stavkov jezika $\mathcal{L}$(nat bool) v niz žetonov.

Pravila na začetku so definirana za usmerjanje pretvorbe izrazov $\mathcal{L}$(nat bool) v žetone.

$$\frac{}{\epsilon\ inp \longleftrightarrow \epsilon\ tokstr} \tag{3.103}$$

$$\frac{s = s_1\hat{}s_2\hat{}s_3\ str \quad s_1\ whs \quad s_2\ itm \longleftrightarrow t\ tok \quad s_3\ inp \longleftrightarrow ts\ tokstr}{s\ inp \longleftrightarrow t\ .\ ts\ tokstr} \tag{3.104}$$

$$\frac{s\ kwd \longleftrightarrow t\ tok}{s\ itm \longleftrightarrow t\ tok} \tag{3.105}$$

$$\frac{s\ boo \longleftrightarrow t\ tok}{s\ itm \longleftrightarrow t\ tok} \tag{3.106}$$

$$\frac{s\ spl \longleftrightarrow t\ tok}{s\ itm \longleftrightarrow t\ tok} \tag{3.107}$$

$$\frac{s = i\ .\ f\ .\ \epsilon\ str}{s\ kwd \longleftrightarrow IF\ tok} \tag{3.108}$$

$$\frac{s = t\ .\ h\ .\ e\ .\ n\ .\ \epsilon\ str}{s\ kwd \longleftrightarrow THEN\ tok} \tag{3.109}$$

$$\frac{s = e \,.\, l \,.\, s \,.\, e\epsilon \; str}{s \; kwd \longleftrightarrow ELSE \; tok} \tag{3.110}$$

$$\frac{s = \; t \,.\, r \,.\, u \,.\, e \,.\, \epsilon \; str}{s \; boo \longleftrightarrow BOOL[t] \; tok} \tag{3.111}$$

$$\frac{s = \; f \,.\, a \,.\, l \,.\, s \,.\, e \,.\, \epsilon \; str}{s \; boo \longleftrightarrow BOOL[f] \; tok} \tag{3.112}$$

$$\frac{s = \; 0 \,.\, \epsilon \; str}{s \; zer \longleftrightarrow ZERO \; tok} \tag{3.113}$$

$$\frac{s = (\,.\epsilon \; str}{s \; spl \longleftrightarrow LP \; tok} \tag{3.114}$$

$$\frac{s =) \,.\, \epsilon \; str}{s \; spl \longleftrightarrow RP \; tok} \tag{3.115}$$

3.4.3 Razčlenjevanje $\mathcal{L}(\textbf{nat bool})$

V prejšnjem razdelku smo definirali kako se nizi znakov pretvarjajo v nize simbolov. Nize simbolov je naprej potrebno razčleniti v skladu z gramatično strukturo jezika $\mathcal{L}(\text{nat bool})$.

Najprej bomo definirali gramatično strukturo jezika v BNF obliki. Zapis bomo pretvorili v pravila s katerim je definirana statična struktura jezika $\mathcal{L}(\text{nat bool})$.

Gramatična struktura $\mathcal{L}(\textbf{nat bool})$

Konkretno sintakso $\mathcal{L}(\text{nat bool})$ lahko definiramo s kontekstno neodvisno gramatiko nad simboli, ki so predstavljeni v 3.92. Spet ima gramatika ima en sam sintaktični razred, exp, ki je definiran z naslednjimi pravili:

$$\begin{array}{llll} Expression & exp & ::= & zer \mid boo \mid \\ & & & \text{if } exp \text{ then } exp \text{ else } exp \mid \\ & & & \text{succ } exp \mid \text{pred } exp \mid \\ & & & \text{iszero } exp \mid LP\ exp\ RP \\ Boolean & boo & ::= & BOOL[l] \qquad (l \text{ bool}) \\ Zero & zer & ::= & ZERO \end{array} \tag{3.116}$$

Če prikažemo interpretacijo gramatike z induktivnimi definicijami, dobimo naslednja pravila.

$$\frac{s\ zer}{s\ exp} \tag{3.117}$$

$$\frac{s\ boo}{s\ exp} \tag{3.118}$$

$$\frac{s_1\ exp \quad s_2\ exp \quad s_3\ exp}{\text{if } s_1 \text{ then } s_2 \text{ else } s_3} \tag{3.119}$$

$$\frac{s\ exp}{\text{succ } s\ exp} \tag{3.120}$$

$$\frac{s\ exp}{\text{pred } s\ exp} \tag{3.121}$$

$$\frac{s\ exp}{\text{pred } s\ exp} \tag{3.122}$$

$$\frac{s\ exp}{LP\ s\ RP\ exp} \tag{3.123}$$

$$\frac{l \text{ bool}}{BOOL[l]\ boo} \tag{3.124}$$

$$\frac{}{ZERO\ zer} \tag{3.125}$$

3.5 Opombe

Poglavje vsebuje prevode izbranih sekcij iz učbenika *Practical Foundations for Programming Languages* [5] avtorja Roberta Harperja.

Poglavje 4

OPERACIJSKA SEMANTIKA

Operacijska semantika definira obnašanje programskega jezika z definicijo enostavnega *abstraktnega stroja*.

Stroj je abstrakten zato, ker uporabljamo izraze programskega jezika kot strojni jezik. Jezik ne prevajamo v bolj konkreten jezik kot to delajo prevajalniki.

V primeru enostavnih jezikov pomeni stanje stroja preprosto stavek. Prehodi med stanji stroja so prehodi iz enega izraza v poenostavitev izraza ali v primeru, da se evaluacija zaključi, prehod v normalni izraz kjer se abstraktni stroj ustavi.

Računalniške jezike bomo predstavljali iz večih izbranih vidikov. Najpomembnejša aspekta jezikov sta statična struktura in evaluacija, ki ju predstavimo s *statično* in *dinamično* semantiko.

Katerikoli aspek programskega jezika bomo predstavili s pomočjo *pravil*, ki ustrezajo logičnim sodbam ali izjavam. Množica pravil predstavlja *teorijo* s katero opišemo željen aspekt jezika.

Lastnosti jezika lahko zdaj opazujemo preko lastnosti teorije. Pomen izrazov jezika izrazimo s pomočjo izbrane teorije. Interpretacija izraza je sekvenca aplikacij pravil teorije na danem izrazu.

Semantiko jezika torej opišemo z apliciranjem pravil teorije na izrazih jezika. Konkreten pomen izraza je *veriga aplikacij pravil* s katero opišemo bodisi izpeljavo statične strukture, izpeljavo ovrednotenja izrazov, preverjanje tipov izrazov, itd.

Izpeljava je torej osnovni način za podrobnejši opis pomena izraza–iz izpeljave lahko vidimo podrobnosti pri strukturi, interpretaciji, evaluaciji, itd.

4.1 Konceptualna zgradba

Pri predstavitvi semantike bomo uporabili primer enostavnega jezika $\mathcal{L}(\text{nat bool})$ s katerim lahko opišemo boolove in numerične vrednosti ter pogojne izraze za boolove in numerične vrednosti.

Najprej bomo predstavili analizo statične strukture jezika z uporabo *statične semantike*. Sintakso jezika bomo predstavili s pravili, ki definirajo zgradbo jezika. Ogledali si bomo tudi alternativne načine opisov pravilnih stavkov danega jezika.

Nadaljevali bomo z *dinamično semantiko*: opisom mehanizmov za predstavitev evaluacije izrazov danega jezika. Evaluacijo bomo predstavili z množico pravil, katerih izvajanje opisuje ovrednotenje izrazov.

Po tem si bomo ogledali kako lahko predstavimo sklepanje z izpeljevanjem pravil. Predstavili bomo osnovne mehanizme uporabljene pri izpeljevanju pravil.

Sledila bo predstavitev dveh osnovnih lastnosti jezikov ter dokazovanje le-teh z indukcijo. Predstavljene lastnosti jezikov bodo:

- *enoličnost* (determinističnost, v teoriji jezikov) izpeljave izraza in
- *zaključitev* izpeljave pravil nad danim izrazom oz. zaključitev izvajanja pravil v vrednosti (t.j. normalni obliki),

Kasneje bomo v Poglavju **??** predstavili še *varne* teorije. Pod tem izrazom razumemo teorije, katerih pravila

1. ne povzročijo mrtvega stanja, kar imenujemo tudi *napredek* in
2. ohranjajo lastnosti jezika oz. tipe izrazov pri izpeljavi, kar imenujemo tudi *ohranitev*.

4.2 Statična semantika

Zaenkrat nas zanimajo samo izrazi, ki so sestavljeni pravilno.

Z uporabo zgornje gramatike lahko sestavimo tudi izraze, ki niso pravilni kot npr. succ$(true)$.

Konstrukcijo takšnih izrazov bomo preprečili z uporabo *statične semantike* jezika.

Statično semantiko lahko predstavimo na več različnih načinov.

Gramatika predstavljena v prejšnjem razdelku definira abstraktno sintakso, ki je vsebovana v naslednjih definicijah.

Definicija 4.2.1 (Induktivno) *Množica izrazov je najmanjša množica $\mathcal{T}$, ki zadošča naslednjim pogojem:*

1. $\{true, false, 0\} \subseteq \mathcal{T}$;
2. *če* $t_1 \in \mathcal{T}$ *potem* $\{$*succ* t_1, *pred* t_1, *iszero* $t_1\} \subseteq \mathcal{T}$
3. *če* $t_1 \in \mathcal{T}, t_2 \in \mathcal{T}$ *in* $t_3 \in \mathcal{T}$ *potem je tudi if* t_1 *then* t_2 *else* $t_3 \in \mathcal{T}$

Poglejmo si definicijo malce bolj podrobno. Prvi stavek definira osnovne tri besede jezika.

Druga in tretja alineja definira strukturo sestavljenih stavkov jezika.

"Najmanjša" množica pomeni, da $\mathcal{T}$ nima drugih elementov kot naštetih.

Naslednja definicija uporablja *pravila izpeljave*, ki se pogosto uporabljajo za *naravno dedukcijo*.

Definicija 4.2.2 (pravila sklepanja) *Množica izrazov jezika $\mathcal{L}(\mathit{nat}\,\mathit{bool})$ je definirana z naslednjimi pravili:*

$$true \in \mathcal{T} \tag{4.1}$$

$$false \in \mathcal{T} \tag{4.2}$$

$$0 \in \mathcal{T} \tag{4.3}$$

$$\frac{t_1 \in \mathcal{T}}{\text{succ } t_1 \in \mathcal{T}} \tag{4.4}$$

$$\frac{t_1 \in \mathcal{T}}{\text{pred } t_1 \in \mathcal{T}} \tag{4.5}$$

$$\frac{t_1 \in \mathcal{T}}{\text{iszero } t_1 \in \mathcal{T}} \tag{4.6}$$

$$\frac{t_1 \in \mathcal{T} \quad t_2 \in \mathcal{T} \quad t_3 \in \mathcal{T}}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3} \tag{4.7}$$

□

Vsako pravilo beremo: "če veljajo pogoji, ki so definirani v premisi, potem lahko izpeljemo posledice pod črto".

Pravila, ki nimajo premise imenujemo tudi *aksiomi*. Te običajno pišemo brez črte.

Pravila lahko vsebujejo metaspremenljivke. Formalno vsako pravilo predstavlja neskončno množico konkretnih pravil, ki jih dobimo tako, da instanciramo metaspremenljivke.

Definicija 4.2.3 (konkretno) *Za vsako naravno število i definiramo množico S_i na naslednji način:*

$$\begin{array}{lcl} S_0 & = & \emptyset \\ S_{i+1} & = & \{true, false, 0\} \\ & \cup & \{\text{succ } t_1, \text{pred } t_1, \text{iszero } t_1 | t_i \in S_i\} \\ & \cup & \{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 | t_1, t_2, t_3 \in S_i\} \end{array}$$

Jezik sestavljajo vsi stavki:

$$S = \bigcup_i S_i \quad \square$$

4.3 Dinamična semantika

Evaluacijo izrazov pogosto imenujemo tudi *dinamična semantika* jezika.

Pri opisu evaluacije uporabimo operacijsko semantiko evaluacije [Plotkin,SOS].

Evaluacija sama predstavlja dinamičen aspekt danega jezika za razliko od predstavitve sestave jezika, ki predstavlja *statično semantiko* jezika.

Pravila za ovrednotenje izrazov definirajo evaluacijsko relacijo med izrazi – predstavljajo en korak pri evaluaciji izrazov.

Če si predstavljamo evaluacijo z abstraktnim strojem, potem en korak evaluacije izraza t pomeni prehod iz enega v drugo stanje stroja: $t \rightarrow t'$.

Vsak korak je definiran z enim pravilom izpeljave.

4.3.1 Evaluacija boolovih izrazov

Poglejmo si zdaj bolj natančno evaluacijo boolovih izrazov. Izraze, ki vsebujejo cela števila si bomo ogledali kasneje.

Če na hitro ponovimo, boolovi izrazi lahko vsebujejo konstanti $true$ in $false$ ter if stavek.

Evaluacija boolovih izrazov je definirana z naslednjimi pravili.

$$\frac{}{\text{if } true \text{ then } t_2 \text{ else } t_3 \rightarrow t_2} \tag{4.8}$$

$$\frac{}{\text{if } false \text{ then } t_2 \text{ else } t_3 \rightarrow t_3} \tag{4.9}$$

$$\frac{t_1 \rightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3} \tag{4.10}$$

Prvo pravilo predstavi ovrednotenje if stavka v primeru, da je logični izraz v pogoju enak $true$. Stroj lahko odvrže vse ostale dele if stavka razen t_2, ki predstavlja novo stanje abstraktnega stroja.

Podobno predstavi drugo pravilo ovrednotenje if stavka, katerega pogoj ima vrednost $false$.

Tretje pravilo pravi, da v primeru, ko pogoj if stavka preide $t_1 \rightarrow t_1'$ potem preide tudi celoten if stavek s pogojem t_1 v stavek s pogojem t_1'.

Primer 4.3.1 *Poglejmo si zdaj interakcijo med pravili. Kot primer bomo uporabili sledeč stavek sestavljen iz dveh if stavkov.*

if $true$ *then* (*if* $false$ *then* $false$ *else* $false$) *else* $true$

Oblika pravil določa vrstni red ovrednotenja stavka.

Najprej se vedno ovrednoti pogoj if stavka in šele nato se lahko ovrednotijo posledice.

V zgornjem primeru se tako stavek ne ovrednotiti v if $true$ *then* $false$ *else* $true$ *ampak se mora ovrednotiti najprej zunanji if stavek in šele nato notranji.* □

Prva dva pravila, ki jih včasih imenujemo tudi *konkretna pravila*, naredita večino dela.

Tretje pravilo pravzaprav samo pomaga usmerjati evaluacijo. Takšna pravila imenujemo tudi *usmerjevalna pravila*.

Zasnova pravil kot tudi vrstni red uporabe pravil določa *evaluacijsko strategijo* abstraktnega stroja.

V naslednjih poglavjih si bomo še večkrat bolj natančno ogledali evaluacijske strategije jezikov.

4.3.2 Evaluacija aritmetičnih izrazov

Poglejmo si zdaj še evaluacijo aritmetičnih izrazov, ki vsebujejo tudi delo s celimi števili.

Najprej obnovimo sintakso dela aritmetičnih izrazov, ki vsebuje cela števila.

$$
\begin{array}{llll}
\text{Izrazi} & t & ::= & \ldots \\
 & & & 0 \\
 & & & \text{succ } t \\
 & & & \text{pred } t \\
 & & & \text{iszero } t \\
\text{Vrednosti} & v & ::= & \ldots \\
 & & & nv \\
\text{Num.vrednosti} & nv & ::= & \ldots \\
 & & & 0 \\
 & & & \text{succ } nv \\
 & & & \text{pred } nv
\end{array}
\tag{4.11}
$$

Evaluacijska pravila za numerične izraze sledijo modelu, ki je uporabljen pri evaluaciji boolovih izrazov.

Definirana so štiri pravila, ki opišejo kako se ovrednotijo operacije pred in iszero.

$$\frac{t_1 \rightarrow t_1'}{\text{succ } t_1 \rightarrow \text{succ } t_1'} \tag{4.12}$$

$$\frac{}{\text{pred } 0 \rightarrow 0} \tag{4.13}$$

$$\frac{}{\text{pred}(\text{succ } nv_1) \rightarrow nv_1} \tag{4.14}$$

$$\frac{t_1 \rightarrow t_1'}{\text{pred } t_1 \rightarrow \text{pred } t_1'} \tag{4.15}$$

$$\frac{}{\text{iszero } 0 \rightarrow true} \tag{4.16}$$

$$\frac{}{\text{iszero}(\text{succ } nv_1) \rightarrow false} \tag{4.17}$$

$$\frac{t_1 \rightarrow t_1'}{\text{iszero } t_1 \rightarrow \text{iszero } t_1'} \tag{4.18}$$

Če bi bili popolnoma natančni bi morali zapisati še definicijo “relacija en-korak-evaluacije je najmanjša relacija, ki zadošča zgornjim pogojem” kot v primeru boolovih izrazov.

Sintaktična kategorija celih števil igra pomembno vlogo pri opisu semantike.

Uporaba metaspremenljivke nv_1 zahteva celo število pri parametrih operacij succ, pred in iszero.

Dinamična semantika majhnih korakov in velikih korakov ...

4.4 Izpeljevanje in sklepanje

Poglejmo si zdaj zveze med stavki in pravili malce bolj formalno.

Definirali bomo naslednje pojme, ki opisujejo evaluacijsko relacijo med stavki: *instanca pravila*, *ujemanje pravil*, *evaluacijo v enem koraku* ter *izpeljivost stavkov*.

Kasneje si bomo ogledali še determinističnost izeljave ter normalne oblike stavkov.

Naš univerzum v katerem bomo preučevali pravila in izpeljave je jezik $\mathcal{L}(\text{nat bool})$, majhen primer jezika, ki vsebuje zadosti strukture, da lahko pokažemo nekatere lastnosti jezikov na slošno.

4.4.1 Pravila in izpeljave

Definicija 4.4.1 Instanco *(primerek) pravila dobimo z zamenjavo metaspremenljivk z istimi izrazi v premisi in posledici pravila.*

Primer 4.4.1 *Naslednji stavek*

$$\textit{if}\ true\ \textit{then}\ true\ \textit{else}\ (\textit{if}\ false\ \textit{then}\ false\ \textit{else}\ false) \rightarrow true \tag{4.19}$$

je instanca pravila 4.8, kjer sta obe instanci t_2 zamenjani s $true$ in t_3 je bil zamenjan z if $false$ then $false$ else $false$. □

Definicija 4.4.2 Evaluacijska relacija v enem koraku *je najmanjša binarna relacija, ki zadošča pravilom evaluacije. Če je par (t, t') v evaluacijski relaciji potem je evaluacijski stavek $t \to t'$* izpeljiv.

Izraz "najmanjša binarna relacija" pravi, da je $t \to t'$ izpeljiv, če imamo bodisi instancno aksioma ali instanco posledice pravila katerega premise so izpeljive, ki se unificira z $t \to t'$.

Izpeljivost izraza lahko pokažemo z drevesom izpeljave, ki je bilo predstavljeno v poglavju o sklepanju. Listi drevesa so premise konkretnih pravil medtem ko so notranja vozlišča premise usmerjevalnih vozlišč.

V našem primeru imamo samo tri pravila med katerimi je samo eno usmerjevalno pravilo, ki ima samo eno premiso.

Edino možno drevo izpeljave v primeru boolovih izrazov je torej ena sama linearna veja. Poglejmo si primer.

Primer 4.4.2 *Najprej bomo definirali nekaj okrajšav za izraze.*

$$
\begin{array}{l}
s = \textit{if}\ true\ \textit{then}\ false\ \textit{else}\ false \\
t = \textit{if}\ s\ \textit{then}\ true\ \textit{else}\ true \\
u = \textit{if}\ false\ \textit{then}\ true\ \textit{else}\ true
\end{array}
$$

Poglejmo si zdaj izpeljavo sledečega izraza.

$$\textit{if}\ t\ \textit{then}\ false\ \textit{else}\ false \to \textit{if}\ u\ \textit{then}\ false\ \textit{else}\ false$$

$$\dfrac{\dfrac{\overline{s \to false}}{t \to u}}{\textit{if}\ t\ \textit{then}\ false\ \textit{else}\ false \to \textit{if}\ u\ \textit{then}\ false\ \textit{else}\ false}$$

□

4.4.2 Normalne oblike in enoličnost izpeljav

Drevo izpeljave lahko torej uporabimo za dokazovanje izpeljivosti izraza.

Ker ima drevo strukturo definirano s posamezhnimi pravili, ki so bila uporabljena pri izpeljavi, ga lahko uporabljamo pri tudi za dokazovanje lastnosti izpeljave.

Zaradi strukture izpeljav lahko uporabljamo strukturno indukcijo kot tehniko za dokazovanje.

Stavek $t \rightarrow t'$ je izpeljiv, če in samo če obstaja izpeljava, ki ima $t \rightarrow t'$ za koren.

Poglejmo si zdaj eno izmed lastnosti, ki velja za boolove izraze izpeljane s podanimi pravili.

Izrek 4.4.1 (determinističnost izpeljave v enem koraku) *Če velja $t \rightarrow t'$ in $t \rightarrow t''$, potem $t' = t''$.* □

Dokaz. Indukcija na strukturo izpeljave izraza. Na vsakem koraku predpostavimo, da željeni rezultat velja za manjše izraze oz. premise korenskega pravila. Potem moramo pokazati, da rezultat velja tudi za koren izpeljave.

V primeru, da je bilo zadnje uporabljeno pravilo 4.8 potem vemo, da ima t obliko if t_1 then t_2 else t_3, kjer velja da je $t_1 = true$. V tem primeru je očitno, da zadnje uporabljeno pravilo izpeljevanja ne more biti 4.9, ker t_1 ne more imeti hkrati vrednosti $t_1 = true$ in $t_1 = false$.

Tudi zadnje pravilo ni zadoščeno, ker zahteva $t \rightarrow t'$, imamo pa vrednost $true$, ki se ne more naprej ovrednotiti. Zadnje pravilo pri evaluacije je lahko torej le 4.8.

Podobno lahko sklepamo tudi v primeru, da velja pravilo 4.9. Ne obstaja tudi drugo pravilo, ki bi se ujelo z $t_1 = false$, torej imamo tudi enolični rezultat.

Končno, preostane nam še zadnje pravilo 4.10. Oblika pravila je if t_1 then t_2 else t_3, kjer velja $t_1 \rightarrow t_1'$ za nek t_1'. Z istim sklepanjem kot že prej lahko vidimo, da je 4.10 edino možno pravilo uporabljeno za v korenu izpeljave t. Izpeljava je spet enolična. □

Naša relacija *izpeljave v enem koraku* pokaže kako se abstrakten stroj premakne iz enega stanja v drugo stanje pri ovrednotenju izraza.

Kot programerje nas zanima končni rezultat evaluacije. Lahko pride do stanja, kjer stroj ne more več narediti naslednjega koraka?

Definicija 4.4.3 *Izraz t je v normalni obliki, če na izrazu ne moremo aplicirati evaluacijskega pravila $t \rightarrow t'$.*

Videli smo že, da sta vrednosti $true$ in $false$ v normalni obliki v definiranem sistemu. Ne obstaja takšno pravilo, ki bi se ujemalo z izrazi $true$ in $false$. To izjavo lahko reformuliramo v naslednjem izreku.

Izrek 4.4.2 *Vsaka vrednost je v normalni obliki.*

To lastnost bomo zagotovili pri vseh jezikih, ki jih bomo definirali. Normalna oblika izraza bo pomenila, da je izraz evaluiran do konca in je vrednost.

Jezik za katerega ne velja izrek 4.4.2 ni dobro definiran.

V obstoječem sistemu boolovih izrazov velja tudi obratno od izreka 4.4.2: vsaka normalna oblika je vrednost.

To lastnost ne bodo imeli vsi jeziki. Normalne oblike, ki niso vrednosti igrajo kritično vlogo pri analizi napak v času izvajanja.

Izrek 4.4.3 *Če je t v normalni obliki, potem je t vrednost.*

Dokaz. Predpostavimo, da t ni vrednost. Enostavno je pokazati z uporabo strukturne indukcije na t, da ni v normalni obliki.

Ker t ni vrednost mora biti oblike if t_1 then t_2 else t_3 za neke t_1, t_2 in t_3. Poglejmo si možne oblike t_1.

Če je $t_1 = true$ potem t sigurno ni v normalni obliki saj bi lahko uporabili pravilo 4.8. Podobno je za primer $t_1 = false$.

V primeru, da t_1 ni niti $true$ niti $false$ potem ni vrednost. Po induktivni hipotezi potem obstaja t_1' tako, da $t_1 \rightarrow t_1'$. Zdaj bi lahko uporabili 4.10 za $t \rightarrow$ if t_1' then t_2 else t_3, torej t ni v normalni obliki. □

Včasih je uporabno videti evaluacijo v večih korakih kot en sam velik korak. To naredimo z definicijo *več-koračne evaluacije*.

Relacija več-koračne evaluacije povezuje izraz t z vsemi izrazi t', ki jih lahko izpeljemo iz t z eno ali več aplikacij pravil.

Definicija 4.4.4 *Več korakov evaluacijske relacije označimo z razmerjem $\rightarrow^*$, refleksivnim in tranzitivnem zaprtju $\rightarrow$. To je najmanjša relacija, za katero velja:*

1. *če velja $t \rightarrow t'$ potem velja tudi $t \rightarrow^* t'$,*
2. *$t \rightarrow^* t$ za vse t ter*
3. *iz $t \rightarrow^* t'$ in $t' \rightarrow^* t''$ lahko sklepamo na $t \rightarrow^* t''$.*

Zdaj, ko imamo eksplicitno definirano relacijo več-koračne evaluacije je lahko definirati dejstva kot je naslednje.

Izrek 4.4.4 (Enoličnost normalne oblike) *Če velja $t \rightarrow^* u$ in $t \rightarrow^* u'$ ter sta u in u' v normalni obliki, potem velja $u = u'$.*

Dokaz. Posledica enoličnosti enega koraka evaluacije. □

Zadnja lastnost, ki jo bomo opazovali je zaključitev evaluacije. Za uporabljen jezik boolovih izrazov velja, da se vsak izraz t enolično ovrednoti v vrednost oz. v normalno obliko.

To je še ena lastnost, ki ni nujno da velja v bolj bogatem jeziku. Dokaz je običajno veliko bolj zapleten kot je dokaz v primeru našega jezika.

Sistem tipov lahko služi kot okostje za dokaz zaključitve programa (Pierce, [10]).

Večina dokazov za zaključitev programov ima neko osnovno obliko. Najprej izberemo dobro-definirano množico S in definiramo funkcijo f, ki preslikuje stanja stroja v vrednosti. Vedno, ko stroj lahko naredi prehod iz t v t' mora veljati $f(t) < f(t')$.

Neskončno verigo evaluacijskih korakov, ki se začnejo s t, lahko preslikamo na verigo elementov S. Ker je S dobro definirana ne moremo imeti neskončnih verig zatorej nimamo neskončne sekvence. Funkcijo f imenujemo mera zaključitve.

Izrek 4.4.5 (Zaključitev evaluacije) *Za vsak izraz t obstaja izraz t', ki je v normalni obliki in velja $t \rightarrow^* t'$.*

Dokaz. Vsak korak evaluacije zmanjša velikost izraza. Velikost izraza je mera zaključitve, ker je običajna urejenost naravnih števil dobro definirana. □

4.5 Opombe

Predstavljen material o operacijski semantiki podaja pregled rezultatov uporabe operacijske semantike pri opisovanju formalnih sistemov. Poglavje vsebuje prevode delov poglavij učbenika B.Pierca [10] in učbenika R.Harperja [5].

Osnovni namen poglavja je podati natančen opis operacijske semantike, primere uporabe operacijske semantike in temeljne lastnosti jezikov, ki so bile odkrite v okviru kompleksnejših jezikov oz. lastnosti, ki so v splošnem zanimive za širšo množico jezikov in jih bomo obdelali bolj podrobno v okviru bolj kompleksnih jezikov.

Poglavje 5

DENOTACIJSKA SEMANTIKA

Denotacijska semantika uporablja bolj abstraktno definicijo pomena od operacijske semantike: namesto sekvence stanj stroja je pomen izrazov definiran z matematičnimi objekti kot so na primer števila, funkcije, itd.

Definicija denotacijske semantike za nek konkreten jezik zahteva najprej definicijo *domen* in nato definicijo *interpretacijskih funkcij*, ki preslikujejo izraze programskega jezika v elemente domen.

Iskanje primernih domen, matematičnih struktur, ki se jih uporablja za definicijo denotacijske semantike se je razvilo v področje, ki ga imenujemo *teorija domen*. Kratek uvod v teorijo domen bo predstavljen v naslednji sekciji.

5.1 Domene

V tej sekciji bo predstavljen uvod v matematično teorijo, *teorijo domen*, ki med drugim nudi formalno ogrodje za konstrukcijo najmanjših fiksnih točk. Konstrukcije bomo potrebovali pri definiciji denotacijske semantike in pri študiju različnih gradnikov (konceptov) programskih jezikov.

Teorija domen preučuje prostore, ki jih sestavljajo izrazi nekega jezika. V naslednjih poglavjih si bomo pogledali statično strukturo in evaluacijo jezikov oz. izrazov neke teorije. Kot osnovno orodje za opisovanje lastnosti jezikov bomo uporabljali *pravila* opisana v predhodnem poglavju.

Teorija domen opisuje stavke jezika bolj iz matematičnega vidika: preučuje urejenost izrazov z uporabo klasičnih matematičnih orodij kot so npr. relacije, množice in funkcije. Objekti, ki jih preučuje predstavljajo abstrakcije stavkov programskega jezika– objekti teorije domen so bolj splošni izrazi, ki jih lahko apliciramo na matematičnih objektih kot tudi na sorodnih jezikih.

5.1.1 Delne urejenosti

Teorija domen uporablja *delno urejene množice*, ki zadoščajo izbranim lastnostim. Delno urejene množice so podane z naslednjo definicijo. Množico D imenujemo *osnovno množico* delne urejenosti $(D, \sqsubseteq)$. Večinoma bomo imenovali osnovno množico D kar delna urejenost; relacijo $\sqsubseteq$ bomo uporabljali za različne delno urejene množice.

Definicija 5.1.1 *Binarna relacija $\sqsubseteq$ nad množico D je delna urejenost, čče je:*

1. *refleksivna:* $\forall d \in D.d \sqsubseteq d$
2. *tranzitivna:* $\forall d, d', d'' \in D.d \sqsubseteq d' \wedge d' \sqsubseteq d'' \Rightarrow d \sqsubseteq d''$
3. *antisimetrična:* $\forall d, d' \in D.d \sqsubseteq d' \wedge d' \sqsubseteq d \Rightarrow d = d'$ □

Par $(D, \sqsubseteq)$ imenujemo delno urejena množica.

Definicija 5.1.2 (Najmanjši element) *Naj bo D delno urejena množica in $S \subseteq D$. Element $d \in S$ je* najmanjši *element, če zadošča naslednjem pogoju.*

$$\forall x \in S.d \sqsubseteq x.$$

Minimalen element bomo imenovali dno *in označili z $\bot_D$ ali samo $\bot$, če je D jasna iz konteksta.* □

Ker je $\sqsubseteq$ anti-simetrična, ima S največ en najmanjši element. Seveda najmanši element množice ni potrebno, da obstaja. Na primer, $\mathbb{Z}$ nima najmanjšega elementa.

Definicija 5.1.3 (Veriga) *Končna naraščajoča veriga D je sekvenvca elementov D, ki zadoščajo naslednjem pogoju.*

$$d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \dots$$

Definicija 5.1.4 (Najmanjša zgornja meja) *Zgornja meja verige $d_0 \sqsubseteq d_1 \sqsubseteq \dots$ je $d \in D$ za katerega velja $\forall n \in \mathbb{N}.d_n \sqsubseteq d$. Če obstaja najmanjša zgornja meja (okr. lub), potem je določena z:*

$$\bigsqcup_{n\geq 0} d_n$$

Poglejmo zakaj je najmanjša zgornja meja dobro definirana. Po definiciji velja $\forall m \in \mathbb{N}.d_m \sqsubseteq \bigsqcup_{n\geq 0} d_n$. Prav tako velja $\forall d \in D$, če $\forall m \in \mathbb{N}.d_m \sqsubseteq d$, potem $\bigsqcup_{n\geq 0} d_n \sqsubseteq d$.

Opazka 5.1.1 *Sledi predstavitev nekaterih lastnosti prej definiranih verig in najmanše zgornje meje.*

(i) Ne bomo se ukvarjali z neskončnimi in padajočimi verigami: "veriga" bo vedno pomenila naraščajoče in končno verigo.

(ii) Elementi verige ni nujno, da so različni. Pravimo, da je veriga $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \dots$ preide v konstanto*, če za nek $N \in \mathbb{N}$ velja $\forall n \geq N.d_n = d_N$. V tem primeru velja $\bigsqcup_{n\geq 0} d_n = d_N$.*

(iii) Kot v primeru najmanjšega elementa poljubne podmnožice delno urejene množice, je najmanjša zgornja meja verige unikatna, če obstaja. Ni pa nujno, da obstaja, npr. veriga $0 \leq 1 \leq 2 \leq 3 \leq \dots$ nima zgornje meje.

(iv) Če izpustimo končno mnogo elementov iz začetka verige se zgornje meja ohranjajo in s tem tudi lub.

$$\bigsqcup_{n\geq 0} d_n = \bigsqcup_{n\geq 0} d_{N+n}, \text{ za vse } N \in \mathbb{N}$$

□

Ukvarjali se bomo z delno urejenimi množicami, ki zadoščajo nekarim lastnostim polnosti: vsaka veriga bo imela najmanjšo zgornjo mejo (lub1+lub2).

Definicija 5.1.5 (ω-polna delno urejena množica; angl. "cpo" t.j. "chain complete poset") *ω-polna delno urejena množica $(D, \sqsubseteq)$ vsebuje samo naraščajoče verige $d_0 \sqsubseteq d_1 \sqsubseteq d_2, ...,$ ki imajo najmanjšo zgornjo mejo $\bigsqcup_{n\geq 0} d_n$:*

(lub1) $\forall m \in \mathbb{N}. d_m \sqsubseteq \bigsqcup_{n\geq 0} d_n$

(lub2) $\forall d \in D. (\forall m \geq 0. d_m \sqsubseteq d) \Rightarrow \bigsqcup_{n\geq 0} d_n \sqsubseteq d$

V literaturi je več različnih definicij *domen*, ki zadoščajo različnim lastnostim. Tu bomo uporabili "minimalno" definicijo: zahtevali bomo, da cpo vsebuje dno.

Definicija 5.1.6 (Domena) *Domena je cpo, ki vsebuje dno $\bot$.*

$$\forall d \in D. \bot \sqsubseteq d$$

Primer 5.1.1 *V tem primeru si bomo ogledali parcialne funkcije $X \rightharpoonup Y$. Domena funkcije f je $dom(f) \subseteq X$ in zaloga vrednosti je Y.*

Delno urejenost definiramo na osnovi domene in zaloge vrednosti f: $f \sqsubseteq g$, čče $dom(f) \subseteq dom(g)$ in $\forall x \in dom(f). f(x) = g(x)$.

Najmanjša zgornja meja verige $f_o \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq ...$ je parcialna funkcija f z domeno $dom(f) = \bigcup_{n\geq 0} dom(f_n)$ in zalogo vrednosti:

$$f(x) = \begin{cases} f_n(x) & x \in dom(f_n) \text{ za nek } n \\ \bot_Y & sicer \end{cases}$$

Definicija funkcije f se ujema z izračunom lub verige $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \ldots : f_{lub} = \bigsqcup_{n\geq 0} f_n$. Domena funkcije f_{lub} je $\bigcup_{n\geq 0} dom(f_n)$ in zaloga vrednosti je definirana $\forall x \in dom(f_i). f_i(x) = f_j(x)$ za vse $i \leq j \in \mathbb{N}$. Velja torej $f_{lub} = f$.

Na koncu definiramo še $\bot_f$, ki predstavlja nedefinirano parcialno funkcijo. □

Primer 5.1.2 *Vsaka delna urejenost $(D, \sqsubseteq)$ katere množica D je končna je ω-polna delna urejenost. V taki množici je vsaka veriga končna. Ni pa potrebno, da ima takšna množica najmanjši element.* □

Primer 5.1.3 *V tem primeru bomo definirali dve domene naravnih števil $\mathbb{N}$. Prvo imenujemo* ravninska *naravna števila, $\mathbb{N}_\perp$:*

$$\begin{array}{ccccccccc} 1 & 2 & 3 & \ldots & n & n+1 & \ldots \\ & & & \perp & & & \end{array}$$

Druga domena predstavlja navpična *naravna števila, Ω:*

$$\begin{array}{c} \omega \\ n+1 \\ n \\ \vdots \\ 2 \\ 1 \\ 0 \end{array}$$

□

Poglejmo si zdaj nekaj primerov delno urejenih množic, ki so oz. niso cpo.

Primer 5.1.4 *Množica naravnih števil $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ opremljena z relacijo $\leq$ ni ω-polna delna urejenost. Naraščajoča veriga $1 \leq 2 \leq 3 \leq 4 \leq \ldots$ nima zgornje meje.* □

Primer 5.1.5 *Poglejmo si zdaj primer delne urejenosti konstruirane iz $\mathbb{N} \cup \{\perp\}$. Za vsako naravno število velja $\forall n \in \mathbb{N}. \perp \sqsubseteq n$. Naravna števila med sabo niso v relaciji $\sqsubseteq$. Tako domeno imenujemo* ravninska naravna števila. □

Primer 5.1.6 *Množico naravnih števil $\mathbb{N}$ lahko uredimo tudi drugače. Množico $\mathbb{N}$ razširimo z elementom ω, ki predstavlja zgornjo mejo $\mathbb{N}$. Relacijo $\sqsubseteq$ lahko zdaj definiramo na sledeč način:*

$$d \sqsubseteq d' \iff \begin{cases} & d, d' \in \mathbb{N} \wedge d \leq d' \\ ali & d \in \mathbb{N} \wedge d' = \omega \\ ali & d = d' = \omega \end{cases}$$

Naraščajoča veriga $1 \leq 2 \leq 3 \leq ...$ iz D ima tako vedno zgornjo mejo ω in D je ω-polna delna urejenost. □

Primer 5.1.7 *Poglejmo si še eno možno definicijo delne urejenosti na osnovi naravnih števil. Tokrat razširimo $\mathbb{N}$ z dvema zgornjima mejama $\{\omega_1, \omega_2\}$. Relacijo $\sqsubseteq$ lahko zdaj definiramo na sledeč način:*

$$d \sqsubseteq d' \iff \begin{cases} & d, d' \in \mathbb{N} \wedge d \leq d' \\ ali & d \in \mathbb{N} \wedge d' \in \{\omega_1, \omega_2\} \\ ali & d = d' = \omega_1 \\ ali & d = d' = \omega_2 \end{cases}$$

Naraščajoča veriga $1 \leq 2 \leq 3 \leq ...$ iz D ima tako dve zgornji meji ω_1 in ω_2. Nima pa najmanjše zgornje meje, ker $\omega_1 \not\sqsubseteq \omega_2$ in $\omega_2 \not\sqsubseteq \omega_1$. D torej ni ω-polna delna urejenost. □

Definicija 5.1.7 (Monotona funkcija) *Funkcija $f : D \to E$ med ω-polnimi delnimi urejenostmi je* monotona, *čče*

$$\forall d, d' \in D. d \sqsubseteq d' \Rightarrow f(d) \sqsubseteq f(d')$$

.

Definicija 5.1.8 (Zvezna funkcija) *Funkcija $f : D \to E$ med ω-polnimi delnimi urejenostmi je* zvezna, *čče je monotona in ohranja najmanjše zgornje meje verig: za vse verige $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \ldots$ iz D velja*

$$f(\bigsqcup_{n \geq 0} d_n) = \bigsqcup_{n \geq 0} f(d_n) \text{ v } E.$$

Definicija 5.1.9 (Striktna funkcija) *Funkcija $f : D \to E$ med ω-polnimi delnimi urejenostmi je* striktna, *čče $f(\bot) = \bot$.*

Opazka 5.1.2 *V primeru, da je $f : D \to E$ monotona in imamo verigo $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \ldots$ v D, potem z aplikacijo f dobimo verigo $f(d_0) \sqsubseteq f(d_1) \sqsubseteq f(d_2) \sqsubseteq \ldots$ v E.*

Še več, v primeru, da je d zgornja meja prve verige, je $f(d)$ zgornja meja druge verige in hkrati tudi večja od najmanjše zgornje meje te verige. Če zapišemo to v matematični notaciji dobimo:

$$\bigsqcup_{n\geq 0} f(d_n) \sqsubseteq f(\bigsqcup_{n\geq 0} d_n)$$

Če zdaj uporabimo asimetričnost $\sqsubseteq$ lahko pokažemo, da je funkcija f med ω-polnimi delnimi urejenostmi zvezna, *tako da za vsako verigo $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \ldots$ iz E preverimo ali velja:*

$$f(\bigsqcup_{n\geq 0} d_n) \sqsubseteq \bigsqcup_{n\geq 0} f(d_n).$$

□

Primer 5.1.8 *Dana sta cpo D in E, za vsak $e \in E$ je enostavno pokazati, da je funkcija $D \to E$, ki ima konstantno vrednost $\lambda d \in D.\ e$, je* zvezna.

Primer 5.1.9 *Naj bo Ω domena vertikalnih naravnih števil, kot je prikazano na Sliki 8.1.1. Funkcija $f : \Omega \to \Omega$ je definirana takole:*

$$f(n) = \begin{cases} n \in \mathbb{N} & f(n) = 0 \\ n = \omega & f(n) = \omega \end{cases}$$

Funkcija je monotona in striktna, ni pa zvezna ker velja:

$$f(\bigsqcup_{n\geq 0} d_n) = f(\omega) = \omega \neq 0 = \bigsqcup_{n\geq 0} 0 = \bigsqcup_{n\geq 0} f(n)$$

□

5.1.2 Tarskijev izrek o fiksni točki

Fiksna točka funkcije $f : D \to D$ je po definiciji $d \in D$, ki zadošča $f(d) = d$. Poglejmo si najprej šibkejši primer *pre-fiksne* točke D je delno urejena množica,

Definicija 5.1.10 (Pre-fiksna točka) *Naj bo D delno urejena množica in naj bo $f : D \to D$ funkcija. Element $d \in D$ je pre-fiksna točka f, če zadošča $f(d) \sqsubseteq d$.* □

Najmanjšo pre-fiksno točko funkcije f bomo označili fix(f).

Definicija 5.1.11 (Najmanjša fiksna točka) *Naj bo D delno urejena množica in naj bo $f : D \to D$ funkcija. Najmanjša fiksna točka funkcije f, fix(f), mora zadoščati naslednjim pogojem:*

lfp1 $f(\mathit{fix}(f)) \sqsubseteq \mathit{fix}(f)$

lfp2 $\forall d \in D. f(d) \sqsubseteq d \Rightarrow \mathit{fix}(f) \sqsubseteq d$. □

Opazka 5.1.3 *(lfp1) fix(f) je tudi pre-fiksna točka, kar pomeni, da se evaluacija f ustavi: $f(\mathit{fix}(f)) \sqsubseteq \mathit{fix}(f)$.*

(lfp2) Vsaka pre-fiksna točka je večja od fiksne točke fix(f): fiksna točka fix(f) je najmanjša pre-fiksna točka. □

Tarskijev izrek o fiksni točki je ključni rezultat, ki omogoča uporabo denotacijske semantike za opis pomena rekurzivnih struktur.

Izrek 5.1.1 (Fiksna točka, Tarski) *Naj bo $f : D \to D$ zvezna funkcija nad domeno D.*

- *f ima* najmanjšo pre-fiksno točko*:*

$$\mathit{fix}(f) = \bigsqcup_{n \geq 0} f^n(\bot)$$

- *$\mathit{fix}(f)$ je fiksna točka f ker zadošča $f(\mathit{fix}(f)) = \mathit{fix}(f)$ in je zato tudi* najmanjša fiksna točka *funkcije f.* □

Opazka 5.1.4 *Notacija $f^n(\bot)$ uporabljena v definiciji:*

$$\begin{cases} f^0(\bot) & = & \bot \\ f^{n+1}(\bot) & = & f(f^n(\bot)) \end{cases} \tag{5.1}$$

Ker velja $\forall d \in D. \bot \sqsubseteq d$, potem velja tudi $f^0(\bot) = \bot \sqsubseteq f^1(\bot)$. Zaradi monotonosti

$$f^n(\bot) \sqsubseteq f^{n+1}(\bot) \Rightarrow f^{n+1}(\bot) = f(f^n(\bot)) \sqsubseteq f(f^{n+1}(\bot)) = f^{n+2}(\bot).$$

Z uporabo indukcije po $n \in \mathbb{N}$ potem velja $\forall n \in \mathbb{N}. f^n(\bot) \sqsubseteq f^{n+1}(\bot)$. Z drugimi besedami elementi $f^n(\bot)$ tvorijo verigo v D. Ker je D cpo je smiselno definirati $\mathit{fix}(f)$. □

Dokaz. [Izrek o fiksni točki]

$$\begin{aligned} f(\mathrm{fix}(f)) &= f(\bigsqcup_{n \geq 0} f^n(\bot)) \\ &= \bigsqcup_{n \geq 0} f(f^n(\bot)) && \text{(zaradi zveznosti)} \\ &= \bigsqcup_{n \geq 0} f^{n+1}(\bot) && () \\ &= \bigsqcup_{n \geq 0} f^n(\bot)) && \text{(zaradi Opazke 5.1.1)} \\ &= \mathrm{fix}(f) \end{aligned}$$

fix(f) je torej fiksna točka za f in zadovolji prvi pogoj (lfp1) v Definiciji 5.1.11. Da bi preverili drugi pogoj $(lfp2)$ definicije najmanjše pre-fiksne točke, predpostavimo da za $d \in D$ velja $f(d) \sqsubseteq d$. Ker je $\bot$ najmanjša vrednost v D velja:

$$f^0(\bot) = \bot \sqsubseteq d$$

in

$$\begin{array}{lll} f^n(\bot) \sqsubseteq d \Rightarrow f^{n+1}(\bot) = f(f^n(\bot)) & \sqsubseteq f(d) & \text{zaradi monotonosti } f \\ & \sqsubseteq d & \text{zaradi predpostavke o } d \end{array}$$

Po indukciji na $n \in \mathbb{N}$ dobimo $\forall n \in \mathbb{N}. f^n(\bot) \sqsubseteq d$. Element d je torej zgornja meja verige in leži nad najmanšo zgornjo mejo:

$$\text{fix}(f) = \bigsqcup_{n \geq 0} f^n(\bot) \sqsubseteq d$$

kot je zahtevano z (lfp2). □

5.1.3 Konstrukcije na domenah

V tej sekciji bodo predstavljeni razni načini za izgradnjo domen in zveznih funkcij. Za konstrukcijo cpo potrebujemo množico, ki je opremljena z binarno relacijo. Potem moramo dokazati, da veljajo naslednji pogoji.

1. Binarna relacija definira delno urejenost.
2. Za vse verige v delni urejenosti obstaja lub.
3. Za domeno je potrebno preveriti še ali ima dno.

V nadaljevanju bodo predstavljene različne metode za konstrukcijo cpo in domen.

Produkti domen

Primer 5.1.10 (Produkt dveh cpo) *Produkt dveh cpo* $(D_1, \sqsubseteq_1)$ *in* $(D_2, \sqsubseteq)$ *ima osnovno množico:*

$$D_1 \times D_2 = \{(d_1, d_2) | d_1 \in D_1 \wedge d_2 \in \mathcal{D}_2\}.$$

Delna urejenost $\sqsubseteq$ *je definirana na sledeč način:*

$$(d_1, d_2) \sqsubseteq (d_1', d_2') \Longleftrightarrow d_1 \sqsubseteq d_1' \wedge d_2 \sqsubseteq d_2'.$$

Najmanjše zgornje meje se izračunajo po komponentah.

$$\bigsqcup_{n \geq 0} (d_{1,n}, d_{2,n}) = (\bigsqcup_{n \geq 0} d_{1,n}, \bigsqcup_{n \geq 0} d_{2,n}) \tag{5.2}$$

Če sta $(D_1, \sqsubseteq_1)$ *in* $(D_2, \sqsubseteq_2)$ *domeni potem je domena tudi* $(D_1 \times D_2, \sqsubseteq)$ *in dno* $\bot_{D_1 \times D_2} = (\bot_{D_1}, \bot_{D_2})$. □

Opazka 5.1.5 *Domena* $(D_1 \times D_2, \sqsubseteq)$ *zahteva uporabo funkcij s katerimi izluščimo eno izmed komponent objekta* (d_1, d_2).

$$\begin{array}{ll} \pi_1 : D_1 \times D_2 \to D_1 & \pi_1(d_1, d_2) =^{def} d_1 \\ \pi_2 : D_1 \times D_2 \to D_2 & \pi_2(d_1, d_2) =^{def} d_2 \end{array}$$

Funkciji π_1 *in* π_2 *sta zvezni, ker sta* D_1 *in* D_2 *domeni.* □

Opazka 5.1.6 *Naj bosta $f_1 : D \to D_1$ in $f_2 : D \to D_2$ zvezni funkciji. Potem je zvezna funkcija tudi:*

$$\langle f_1, f_2\rangle : D \to D_1 \times D_2$$
$$\langle f_1, f_2\rangle(d) =^{def} (f_1(d), f_2(d))$$

Dokaz. Zveznost $\langle f_1, f_2\rangle$ sledi direktno iz definicije najmanjše zgornje meje (lub) verig parov z enačbo 5.2.

$$\begin{aligned}\langle f_1, f_2\rangle(\bigsqcup_{n\geq 0} d_n) &= (f_1(\bigsqcup_{n\geq 0} d_n), f_2(\bigsqcup_{n\geq 0} d_n)) \\ &= (\bigsqcup_{n\geq 0} f_1(d_n), \bigsqcup_{n\geq 0} f_2(d_n)) \\ &= \bigsqcup_{n\geq 0} \langle f_1, f_2\rangle(d_n)\end{aligned}$$

□

Primer 5.1.11 (Odvisni produkti) *Naj bo I množica in $\forall i \in I$ imamo cpo $(D_i, \sqsubseteq)$.* Produkt *cele družine cpo-jev je definiran na sledeč način.*

Osnovna množica I-kratnega kartezijskega produkta je $\Pi_{i\in I} D_i$: sestavljena je iz funkcij p definiranih nad I tako, da je vrednost za vsak $i \in I$ element $p(i) \in D_i$.

Relacija delne urejenosti $\sqsubseteq$ je definirana takole:

$$p \sqsubseteq p' \iff \forall i \in I. p(i) \sqsubseteq_i p'(i)$$

Najmanjše zgornje meje $(\Pi_{i\in I} D_i, \sqsubseteq)$ izračunamo po komponentah. Lub verige $p_0 \sqsubseteq p_1 \sqsubseteq p_2 \sqsubseteq \ldots$ iz produkta cpo-jev je funkcija, ki preslika vsak $i \in I$ v lub verige $p_0(i) \sqsubseteq p_1(i) \sqsubseteq p_2(i) \sqsubseteq \ldots$ iz D_i. Za verige v D_i pa velja:

$$(\bigsqcup_{n\geq 0} p_n)(i) = \bigsqcup_{n\geq 0} p_n(i) \qquad (i \in I)$$

Zdaj lahko definiramo za vsak $i \in I$ i-to projekcijsko funkcijo π_i.

$$\pi_i : \Pi_{i'\in I} D_{i'} \to D_i$$
$$\pi_i(p) =^{def} p(i)$$

Funkcije π_i so zvezne. V primeru, da so vsi D_i domene, potem je tudi $\Pi_{i\in I} D_i$ domena. Lub je funkcija, ki preslika $i \in I$ v najmanjšo zgornjo mejo D_i. □

Funkcijske domene

Iz množice zveznih funkcij med dvema cpo ali domenami lahko naredimo cpo oz. domeno.

Primer 5.1.12 *Naj bosta $(D, \sqsubseteq_D)$ in $(E, \sqsubseteq_E)$ cpo-ja. Funkcijski cpo $(D \to E, \sqsubseteq)$ ima osnovno množico:*

$$D \to E =^{def} \{f | f : D \to E \text{ je zvezna}\}.$$

Delna urejenost je definirana:

$$f \sqsubseteq f' \Leftrightarrow^{def} \forall d \in D. f(d) \sqsubseteq f'(d)$$

Najmanjše zgornje meje se izračunajo z uporabo najmanjših zgornjih mej iz E.

$$(\bigsqcup_{n\geq 0} f_n)(d) = \bigsqcup_{n\geq 0} f_n(d)$$

Če je E domena potem je domena tudi $D \to E$. Dno domene $D \to E$ je $\bot_{D\to E} = \bot_E$. □

Dokaz. Pokazali bi radi, da je lub verige funkcij $\bigsqcup_{n\geq 0} f_n$ zvezna funkcija. Uporabimo zakon zamenjave zveznih funkcij.

$$
\begin{array}{lcll}
(\bigsqcup_{n\geq 0} f_n)(\bigsqcup_{m\geq 0} d_m) & = & \bigsqcup_{n\geq 0}(f_n(\bigsqcup_{m\geq 0} d_m)) & \text{definicija } (\bigsqcup_{n\geq 0} f_n) \\
& = & \bigsqcup_{n\geq 0}(\bigsqcup_{m\geq 0} f_n(d_m)) & \text{zveznost } f_n \\
& = & \bigsqcup_{m\geq 0}(\bigsqcup_{n\geq 0} f_n(d_m)) & \text{zakon zamenjave} \\
& = & \bigsqcup_{m\geq 0}((\bigsqcup_{n\geq 0} f_n)(d_m)) & \text{definicija } (\bigsqcup_{n\geq 0} f_n)
\end{array}
$$

□

Opazka 5.1.7 (Evaluacija in Curry operacija) *Dana sta cpo-ja D in E in zvezna funkcija ev:*

$$
\begin{array}{l}
ev : (D \to E) \times D \to E \\
ev(f, d) =^{def} f(d)
\end{array}
$$

Naj bo $f : D' \times D \to E$ zvezna funkcija, kjer je D' cpo. Za vsak $d' \in D'$ zvezna funkcija $d \in D \to f(d', d)$ določa element funkcijskega cpo $D \to E$, ki ga bomo zapisali cur$(f)(d)$. Funkcija cur(f) je zvezna.

$$
\begin{array}{l}
\mathit{cur} : D' \to (D \to E) \\
\mathit{cur}(f)(d') =^{def} \lambda d \in D.f(d', d)
\end{array}
$$

□

Dokaz. Da bi pokazali zveznost ev spet uporabimo zakon zamenjave zveznih funkcij.

$$
\begin{array}{lcll}
ev(\bigsqcup_{n\geq 0}(f_,d_n)) & = & ev(\bigsqcup_{i\geq 0} f_i, \bigsqcup_{j\geq 0} d_j) & \text{lub para po komponentah} \\
& = & (\bigsqcup_{i\geq 0} f_i)(\bigsqcup_{j\geq 0} d_j) & \text{po definiciji } ev \\
& = & \bigsqcup_{i\geq 0} f_i(\bigsqcup_{j\geq 0} d_j) & \text{lub-i v fun. cpo-jih po argumentih} \\
& = & \bigsqcup_{i\geq 0}\bigsqcup_{j\geq 0} f_i(d_i) & \text{zaradi zveznosti } f_i \\
& = & \bigsqcup_{n\geq 0} f_n(d_n) & \text{zaradi Opazke 8.1.1} \\
& = & \bigsqcup_{n\geq 0} ev(f_n, d_n) & \text{po definiciji } ev
\end{array}
$$

Zveznost vsake funkcije cur$(f)(d')$ in cur(f) sledi iz dejstva, da se lub-i verig iz $D_1 \times D_2$ računajo po komponentah. □

5.2 Programski jezik IMP

IMP je enostaven imperativen jezik z while zanko za katerega bomo definirali denotacijsko semantiko.

Domene:

naravna števila	$\mathbb{N}$
boolove vrednosti	$t = \{true, false\}$
lokacije	Loc
aritmetični izrazi	$AExp$
boolovi izrazi	$BExp$
ukazi	Com

Meta spremenljivke:

$n, m \in N$
$X, Y \in Loc$
$a \in AExp$
$b \in BExp$
$c \in Com$

Imenu je implicitno pripisan tip.

Izrazi IMP:

$$\begin{array}{lll} AExp\, a & ::= & n \mid X \mid \\ & & a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1 \\ BExp\, a & ::= & \text{true} \mid \text{false} \mid \\ & & a_0 = a_1 \mid a_0 \leq a_1 \mid \\ & & \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1 \\ Com\, c & ::= & \text{skip} \mid X := a \mid c_0; c_1 \mid \\ & & \text{if } b \text{ then } c_0 \text{ else } c_1 \mid \\ & & \text{while } b \text{ do } c \end{array} \tag{5.3}$$

IMP je vsebovan v vseh programskih jezikih.

5.3 Denotacijska semantika IMP

Vsakemu izrazu t jezika priredimo *denotacijo* $[\![t]\!]$, matematični objekt, ki predstavlja pomen izraza t.

Denotacija stavka je določena z denotacijami izrazov, ki sestavljajo stavek.

Semantične funkcije $\mathcal{A}$, $\mathcal{B}$ in $\mathcal{C}$ so definirane na osnovi strukturne indukcije.

$$\begin{aligned}\mathcal{A} &: AExp \to (\Sigma \to \mathbf{N})\\ \mathcal{B} &: BExp \to (\Sigma \to \mathbf{T})\\ \mathcal{C} &: Com \to (\Sigma \to \Sigma)\end{aligned}$$

Za vsak tip stavka definiramo parcialno funkcijo $\mathcal{A}, \mathcal{B}, \ldots$:

- predpostavljamo, da je denotacija komponent izrazov že definirana in
- pomen stavka s predstavlja $\mathcal{S}[\![s]\!]$, kjer je $\mathcal{S}$ denotacijska funkcija stavkov tipa s.

5.3.1 Aritmetični izrazi

Aritmetični izrazi $AExp$.

$$\begin{aligned}&\mathcal{A}[\![n]\!] = \{(\sigma, n) \mid \sigma \in \Sigma\}\\ &\mathcal{A}[\![X]\!] = \{(\sigma, \sigma(X)) \mid \sigma \in \Sigma\}\\ &\mathcal{A}[\![a_0 + a_1]\!] = \{(\sigma, n_0 + n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\}\\ &\mathcal{A}[\![a_0 - a_1]\!] = \{(\sigma, n_0 - n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\}\\ &\mathcal{A}[\![a_0 * a_1]\!] = \{(\sigma, n_0 * n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\}\end{aligned}$$

Primer 5.3.1 $\mathcal{A}[\![3 * 5]\!]\sigma = \mathcal{A}[\![3]\!]\sigma + \mathcal{A}[\![5]\!]\sigma = 3 + 5 = 8$ □

Denotacija $\mathcal{A}[\![n]\!]$ je *funkcija*.

Definicija denotacijskih pravil aritmetičnih izrazov v λ-notaciji.

$$\begin{array}{l}
\mathcal{A}[\![n]\!] = \lambda\sigma \in \Sigma.n \\
\mathcal{A}[\![X]\!] = \lambda\sigma \in \Sigma.\sigma(X) \\
\mathcal{A}[\![a_0 + a_1]\!] = \lambda\sigma \in \Sigma.(A[\![a_0]\!]\sigma + \mathcal{A}[\![a_1]\!]\sigma) \\
\mathcal{A}[\![a_0 - a_1]\!] = \lambda\sigma \in \Sigma.(A[\![a_0]\!]\sigma - \mathcal{A}[\![a_1]\!]\sigma) \\
\mathcal{A}[\![a_0 * a_1]\!] = \lambda\sigma \in \Sigma.(A[\![a_0]\!]\sigma * \mathcal{A}[\![a_1]\!]\sigma)
\end{array}$$

5.3.2 Logični izrazi

Uporabimo logične operacije $\wedge_T$, $\vee_T$ in $\neg_T$ nad vrednostmi iz domene $\mathbf{T}$.

$$\begin{array}{lcl}
\mathcal{B}[\![true]\!] & = & \{(\sigma, true) \mid \sigma \in \Sigma\} \\
\mathcal{B}[\![false]\!] & = & \{(\sigma, false) \mid \sigma \in \Sigma\} \\
\mathcal{B}[\![a_0 = a_1]\!] & = & \{(\sigma, true) \mid \sigma \in \Sigma \wedge \mathcal{A}[\![a_0]\!]\sigma = \mathcal{A}[\![a_1]\!]\sigma\} \cup \\
 & & \{(\sigma, false) \mid \sigma \in \Sigma \wedge \mathcal{A}[\![a_0]\!]\sigma \neq \mathcal{A}[\![a_1]\!]\sigma\} \\
\mathcal{B}[\![a_0 \leq a_1]\!] & = & \{(\sigma, true) \mid \sigma \in \Sigma \wedge \mathcal{A}[\![a_0]\!]\sigma \leq \mathcal{A}[\![a_1]\!]\sigma\} \cup \\
 & & \{(\sigma, false) \mid \sigma \in \Sigma \wedge \mathcal{A}[\![a_0]\!]\sigma \not\leq \mathcal{A}[\![a_1]\!]\sigma\} \\
\mathcal{B}[\![\neg b]\!] & = & \{(\sigma, \neg_T t) \mid \sigma \in \Sigma \wedge (\sigma, t) \in \mathcal{B}[\![b]\!]\} \\
\mathcal{B}[\![b_0 \wedge b_1]\!] & = & \{(\sigma, t_0 \wedge_T t_1) \mid \sigma \in \Sigma \wedge (\sigma, t_0) \in \mathcal{B}[\![b_0]\!] \wedge (\sigma, t_1) \in \mathcal{B}[\![b_1]\!]\} \\
\mathcal{B}[\![b_0 \vee b_1]\!] & = & \{(\sigma, t_0 \vee_T t_1) \mid \sigma \in \Sigma \wedge (\sigma, t_0) \in \mathcal{B}[\![b_0]\!] \wedge (\sigma, t_1) \in \mathcal{B}[\![b_1]\!]\}
\end{array}$$

Lahko pokažemo z uporabo strukturne indukcije, da je denotacija $\mathcal{B}[\![b]\!]$ funkcija. Na primer:

$$\mathcal{B}[\![a_0 \leq a_1]\!] = \left\{ \begin{array}{ll} true & \mathcal{A}[\![a_0]\!]\sigma \leq \mathcal{A}[\![a_1]\!]\sigma \\ false & \mathcal{A}[\![a_0]\!]\sigma \not\leq \mathcal{A}[\![a_1]\!]\sigma \end{array} \right.$$

za vsako stanje $\sigma \in \Sigma$.

5.3.3 Stavki IMP

$$\begin{array}{lcl}
\mathcal{C}[\![\text{skip}]\!] & = & \{(\sigma,\sigma) \mid \sigma \in \Sigma\} \\
\mathcal{C}[\![X := a]\!] & = & \{(\sigma, \sigma(n/X)) \mid \sigma \in \Sigma \wedge n = \mathcal{A}[\![a]\!]\sigma\} \\
\mathcal{C}[\![c_0; c_1]\!] & = & \mathcal{C}[\![c_1]\!] \circ \mathcal{C}[\![c_0]\!] \\
\mathcal{C}[\![\text{if } b \text{ then } c_0 \text{ else } c_1]\!] & = & \{(\sigma,\sigma') \mid \mathcal{B}[\![b]\!]\sigma = true \wedge (\sigma,\sigma') \in \mathcal{C}[\![c_0]\!]\} \cup \\
 & & \{(\sigma,\sigma') \mid \mathcal{B}[\![b]\!]\sigma = false \wedge (\sigma,\sigma') \in \mathcal{C}[\![c_1]\!]\}
\end{array}$$

Denotacija stavka while je bolj kompleksna od $AExp$ in $BExp$.

Uporabimo invarianto:

$$w \equiv \text{while } b \text{ do } c$$

$$w \ \sim \ \text{if } b \text{ then } c; w \text{ else skip}$$

Denotacija stavka while:

$$\begin{array}{rcl}
\mathcal{C}[\![w]\!] & = & \{(\sigma,\sigma') \mid \mathcal{B}[\![b]\!]\sigma = true \wedge (\sigma,\sigma') \in \mathcal{C}[\![c; w]\!]\} \cup \\
 & & \{(\sigma,\sigma) \mid \mathcal{B}[\![b]\!]\sigma = false\} \\
 & = & \{(\sigma,\sigma') \mid \mathcal{B}[\![b]\!]\sigma = true \wedge (\sigma,\sigma') \in \mathcal{C}[\![w]\!] \circ \mathcal{C}[\![c]\!]\} \cup \\
 & & \{(\sigma,\sigma) \mid \mathcal{B}[\![b]\!]\sigma = false\}
\end{array}$$

Zapišimo zdaj φ namesto $\mathcal{C}[\![w]\!]$, β namesto $\mathcal{B}[\![b]\!]$ in γ namesto $\mathcal{C}[\![c]\!]$.

$$\begin{array}{rcl}
\varphi = & \{(\sigma,\sigma') \mid \beta(\sigma) = true \wedge (\sigma,\sigma') \in \varphi \circ \gamma\} \cup \\
 & \{(\sigma,\sigma) \mid \beta(\sigma) = false\}
\end{array}$$

Imamo rekurzivno enačbo.

$$\begin{array}{rcl}
\Gamma(\varphi) = & \{(\sigma,\sigma') \mid \exists\sigma''.\beta(\sigma) = true \wedge (\sigma,\sigma'') \in \gamma \wedge (\sigma'',\sigma') \in \varphi\} \cup \\
 & \{(\sigma,\sigma) \mid \beta(\sigma) = false\}
\end{array}$$

- (σ,σ'') - evaluacija c
- (σ'',σ') - evaluacija w

Želimo izračunati fiksno točko φ v izrazu Γ.

$$\varphi = \Gamma(\varphi)$$

Funkcija Γ je enaka $\widehat{R}$, kjer je R operator na množici določen z naslednjim pravilom.

$$\begin{array}{rcl} R & = & \{(\{(\sigma'', \sigma')\}/(\sigma, \sigma') \mid \beta(\sigma) = true \wedge (\sigma, \sigma'') \in \gamma\} \cup \\ & & \{(\emptyset/(\sigma, \sigma)) \mid \beta(\sigma) = false\} \end{array}$$

V naslednji sekciji bomo pokazali, da ima $\widehat{R}$ fiksno točko.

$$\varphi = \text{fix}(\widehat{R})$$

Izračun fiksne točke $\widehat{R}$

Instance pravil imajo obliko $(\emptyset/x)$ ali $(\{x_1, \ldots, x_n\}/x)$

Definicija 5.3.1 *Dana je množica pravil R. I_R je množica izrazov za katere obstaja izpeljava z uporabo pravil v R:*

$$I_R = \{x \mid \Vdash_R x\}$$

Definicija 5.3.2 *Množica Q je zaprta glede na množico pravil R ali preprosto R-zaprta, če in samo če velja:*

$$\forall (X/y) \in R. X \subseteq Q \Rightarrow y \in Q$$

Z drugimi besedami je množica zaprta glede na dana pravila, če velja, da je vedno ko so v množici premise pravila, tam tudi posledica.

I_R je *najmanjša* R-zaprta množica:

Izrek 5.3.1 *Naj bo dana množica pravil R*
(i) I_R je R-zaprta, in
(ii) če je Q R-zaprta množica, potem $I_R \subseteq Q$.

Dokaz. (i) Enostavno je videti, da je I_R zaprt glede na R. Po definiciji I_R, le-ta vsebuje vse izraze, ki se dajo izpeljati iz pravil R.

(ii) Recimo, da je množica Q R-zaprta. Pokazali bi radi $I_R \subseteq Q$. Vsak element $y \in I_R$ ima izpeljavo d, ki jo sestavljajo pravila z zaključkom (X/y) pa vse do osnovnih aksiomov. Ker so vsi elementi I_R izpeljani iz aksiomov, lahko isto naredimo tudi v zaprtju množice B.

Dokaz lahko predstavimo bolj formalno z indukcijo glede na relacijo izpeljave $\prec$. Dokazati je potrebno $\forall y \in I_R.\ d \Vdash_R y \Rightarrow y \in Q$ za vse R-izpeljave d. Iz tega sledi $I_R \subseteq Q$. □

Naj bo R množica primerkov pravil. Operator $\widehat{R}$ je definiran z naslednjim zapisom.

$$\widehat{R}(B) = \{y \mid \exists X \subseteq B.(X/y) \in R\}$$

Operator $\widehat{R}$ omogoča še en način za opis zaprtih množic.

Množica B je zaprta za operator $\widehat{R}$ natako takrat, ko $\widehat{R}(B) \subseteq B$.

Operator $\widehat{R}$ je monoton:

$$A \subseteq B \Rightarrow \widehat{R}(A) \subseteq \widehat{R}(B)$$

Če zaporedoma uporabimo operator $\widehat{R}$ na prazni množici, dobimo sekvenco množic.

$$\begin{aligned}
A_0 &= \widehat{R}^0(\emptyset) = \emptyset \\
A_1 &= \widehat{R}^1(\emptyset) = \widehat{R}(\emptyset) \\
A_2 &= \widehat{R}^2(\emptyset) = \widehat{R}(\widehat{R}(\emptyset)) \\
&\vdots \\
A_n &= \widehat{R}^n(\emptyset)
\end{aligned}$$

Množice so zaporedoma vsebovane druga v drugi.

$$A_0 \subseteq A_1 \subseteq A_2 \subseteq \ldots \subseteq A_n \subseteq \ldots$$

Izrek 5.3.2 *Naj bo* $A = \bigcup_{n \in w} A_n$.
(i) A *je* R*-zaprta.*
(ii) $\widehat{R}(A) = A$.
(iii) A *je najmanjša* R*-zaprta množica.*

Dokaz. (i) Naj $(X/y) \in R$ in $X \subseteq A$, kjer je $A = \bigcup_n A_n$. Ker je X končna množica mora obstajati n tako, da $X \subseteq A_n$. Ker $X \subseteq A$ potem $y \in \widehat{R}(A_n) = A_{n+1}$. A je torej zaprta za R.

(ii) A je R-zaprta, torej $\widehat{R}(A) \subseteq A$. Pokazati je potrebno še obratno. Naj bo $y \in A$, potem $y \in A_n$ za nek n in zaradi tega tudi $y \in \widehat{R}(A_{n-1})$. Obstaja torej pravilo $(X/y) \in R$ in $X \subseteq A_{n-1}$. Ker je $A_{n-1} \subseteq A$ in $X \subseteq A$, potem $y \in \widehat{R}(A)$. Velja torej $\widehat{R}(A) = A$.

(iii) Pokazati moramo, da če je B R-zaprta množica potem velja $A \subseteq B$. Ker je B R-zaprta potem velja $\widehat{R}(B) \subseteq B$. Z indukcijo pokažemo, da so vsi $A_n \subseteq B$.

Osnova indukcije $A_0 \subseteq B$ je očitno res, ker $A_0 = \emptyset$. Indukcijski korak mora pokazati, da $A_{n+1} = \widehat{R}(A_n) \subseteq \widehat{R}(B) \subseteq B$. Ker smo predpostavili, da $A_n \subseteq B$ in zaradi monotonosti $\widehat{R}$ velja potem $A \subseteq B$. □

Končnost števila pravil oblike (X/y) je igralo ključno vlogo pri dokazu (i).

Iz (i) in (iii) vidimo, da je $A = I_R$, množica elementov za katere obstaja R-izpeljava.

Točka (ii) pove, da je I_R fiksna točka $\widehat{R}$. Še več, (iii) pravi, da je I_R *najmanjša fiksna točka* $\widehat{R}$:

$$\widehat{R}(B) = B \Rightarrow I_R \subseteq B.$$

Če bi bila katerakoli druga B fiksna točka R, potem ker je B zaprta za R velja $I_R \subseteq B$.

Množica I_R definirana na osnovi pravil R je *najmanjša fiksna točka* R, fix$(\widehat{R})$:

$$\text{fix}(\widehat{R}) =_{def} \bigcup_{n\in\omega} \widehat{R}^n(\emptyset)$$

Denotacija while

Denotacija stavka w je torej določena s fiksno točko φ.

$$\mathcal{C}[\![\text{ while } b \text{ do } c \,]\!] = fix(\Gamma)$$
$$\Gamma(\varphi) = \{(\sigma,\sigma') \mid \beta(\sigma) = true \wedge (\sigma,\sigma') \in \varphi \circ \mathcal{C}[\![c]\!]\} \cup \{(\sigma,\sigma) \mid \beta(\sigma) = false\}$$

Izrek 5.3.3 *Na bo $w \equiv$ while b do c, potem velja:*

$$\mathcal{C}[\![w]\!] = \mathcal{C}[\![\textit{ if } b \textit{ then } c; w \textit{ else skip }]\!]$$

Dokaz. Denotacija w je fiksna točka funkcije Γ.

$$\begin{aligned}
\mathcal{C}[\![w]\!] = &\ \Gamma(\mathcal{C}[\![w]\!]) \\
= &\ \{(\sigma,\sigma') \mid \mathcal{B}[\![b]\!]\sigma = true \wedge (\sigma,\sigma') \in \mathcal{C}[\![w]\!] \circ \mathcal{C}[\![c]\!]\} \cup \\
&\ \{(\sigma,\sigma) \mid \mathcal{B}[\![b]\!]\sigma = false\} \\
= &\ \{(\sigma,\sigma') \mid \mathcal{B}[\![b]\!]\sigma = true \wedge (\sigma,\sigma') \in \mathcal{C}[\![c; w]\!]\} \cup \\
&\ \{(\sigma,\sigma) \mid \mathcal{B}[\![b]\!]\sigma = false\} \\
= &\ \mathcal{C}[\![\text{if } b \text{ then } c; w \text{ else skip}]\!]
\end{aligned}$$

□

5.3.4 Zanka while kot fiksna točka

Poglejmo si pomen v konkretnem primeru while stavka:

$$\text{while } X > 0 \text{ do } (Y := Y * X;\ X := X - 1)$$

Spremenljivki X in Y predstavljata lokaciji v spominu.

Stanje abstraktne predstavitve stavka while predstavimo s parom celih števil (x, y), ki predstavljata vrednost spremenljivk X in Y.

$$State =_{def} \mathbb{Z} \times \mathbb{Z}$$
$$D =_{def} State \rightharpoonup State$$

Denotacijo stavka while $X > 0$ do $(Y := Y * X;\ X := X - 1)$ definiramo kot funkcijo $w : (\mathbb{Z} \times \mathbb{Z}) \to (\mathbb{Z} \times \mathbb{Z})$.

Denotacija stavka $[\![\text{while } X > 0 \text{ do } (Y := Y * X;\ X := X - 1)]\!] \in D$ je rešitev enačbe fiksne točke $w = f(w)$, kjer $f : D \to D$ označuje eno iteracijo stavka while.

$$f(w)(x, y) = \begin{cases} (x, y) & \text{če } x \leq 0 \\ w(x - 1, x * y) & \text{če } x > 0 \end{cases}$$

Poglejmo si zdaj delno urejenost $\sqsubseteq$ nad D:

$w \sqsubseteq w'$ čče velja $\forall (x, y) \in State:$ če je w definiran za (x, y), potem je definiran tudi w' in velja tudi $w(x, y) = w'(x, y)$

Imamo tudi najmanjši element $\perp \in D$:

$\perp =_{def}$ totalno nedefinirana parcialna funkcija
$\forall w \in D : \perp \sqsubseteq w$

Delna urejenost $\sqsubseteq$ definira neke vrste 'informacijsko urejenost': $w \sqsubseteq w'$ če se w' ujema z w vedno, ko je w definirana, vendar je w' lahko definirana širše.

Sekvenco $w_0, w_1, w_2, \ldots$ dobimo tako, da začnemo z $\perp$ in apliciramo znova in znova funkcijo f nad prejšnjo vrednostjo:

$$\begin{cases} w_0 =_{def} \perp \\ w_{n+1} =_{def} f(w_n) \end{cases}$$

Če zdaj uporabimo definicijo f dobimo sledečo sekvenco funkcij:

$$w_1(x,y) = f(\bot)(x,y) = \begin{cases} (x,y) & \text{če } x \leq 0 \\ \text{nedefinirano} & \text{če } x \geq 1 \end{cases}$$

$$w_2(x,y) = f(w_1)(x,y) = \begin{cases} (x,y) & \text{če } x \leq 0 \\ (0,y) & \text{če } x = 1 \\ \text{nedefinirano} & \text{če } x \geq 2 \end{cases}$$

$$w_3(x,y) = f(w_2)(x,y) = \begin{cases} (x,y) & \text{če } x \leq 0 \\ (0,y) & \text{če } x = 1 \\ (0,2*y) & \text{če } x = 2 \\ \text{nedefinirano} & \text{če } x \geq 3 \end{cases}$$

$$w_4(x,y) = f(w_3)(x,y) = \begin{cases} (x,y) & \text{če } x \leq 0 \\ (0,y) & \text{če } x = 1 \\ (0,2*y) & \text{če } x = 2 \\ (0,6*y) & \text{če } x = 3 \\ \text{nedefinirano} & \text{če } x \geq 4 \end{cases}$$

V splošnem dobimo sledečo funkcijo:

$$w_n(x,y) = f(w_{n-1})(x,y) = \begin{cases} (x,y) & \text{če } x \leq 0 \\ (0,(!x)*y) & \text{če } 0 < x < n \\ \text{nedefinirano} & \text{če } x \geq n \end{cases}$$

Tako smo dobili naraščajočo sekvenco parcialnih funkcij:

$$w_0 \sqsubseteq w_1 \sqsubseteq w_2 \sqsubseteq \ldots \sqsubseteq w_n \sqsubseteq \ldots$$

Unija parcialnih funkcij v sekvenci je element w_∞

$$w_\infty(x,y) = \begin{cases} (x,y) & \text{če } x \leq 0 \\ (0,(!x)*y) & \text{če } x > 0 \end{cases}$$

Funkcija w_∞ je fiksna točka funkcije f zato ker velja:

$$
\begin{aligned}
f(w_\infty)(x,y) &= \begin{cases} (x,y) & \text{če } x \leq 0 \\ w_\infty(x-1, x*y) & \text{če } x > 0 \end{cases} && \text{(po definiciji } f\text{)} \\
&= \begin{cases} (x,y) & \text{če } x \leq 0 \\ (0, 1*y) & \text{če } x = 1 \\ (0, !(x-1)*x*y) & \text{če } x > 1 \end{cases} && \text{(po definiciji } w_\infty\text{)} \\
&= w_\infty(x,y).
\end{aligned}
$$

Pokažemo lahko tudi, da je w_∞ najmanjša fiksna točka f:

$$\forall w \in D : w = f(w) \Rightarrow w_\infty \sqsubseteq w.$$

Konstrukcija denotacija stavka while $X > 0$ do $(Y := Y * X;\ X := X - 1)$ je primerek uporabe Tarskijevega izreka.

5.4 Opombe

Poglavje temelji na prosojnicah Glyn Winskel z naslovom *Denotational semantics* [15] ter na knjigi Glyn Winskel za naslovom *The Formal Semantics of Programming Languages* [13].

Poglavje 6

Λ-RAČUN

6.1 Uvod

Lambda račun je razvil leta 1930 Alonzo Church

λ-račun je bil uporabljen v slavnem članku l.1936 o obstoju neodločljivega problema (Entscheidungsproblem).

Turing je istega leta napisal članek, kjer predstavi svoj stroj.

Izviri λ-računa so v delu Leibniza v 17. stoletju (Calculus Ratiocinator).

Formalna paradigma programskih jezikov:

- enostaven,
- izrazen (iz strani logike in računsko),
- proučevalo ga je veliko ljudi in
- enostavno razširljiv s potebnimi novimi gradniki.

“Whatever the next 700 languages turn out to be, they will surely be variants of lambda calculus.”, Landin 1966 :)

6.2 Sintaksa

Lambda račun ima tri vrste izrazov

$$e ::= x \mid \lambda x.e \mid e_1 e_2$$

x - spremenljivka

$\lambda x.e$ - funkcija z enim argumentom in telesom e

$e_1\ e_2$ - je aplikacija funkcije

Aplikacija funkcije je levo asociativna

$$x\ y\ z \quad \text{pomeni} \quad (x\ y)z$$

Lambda abstrakcija se povezuje na desno

$$\lambda x.x\lambda y.x\ y\ z \quad \text{pomeni} \quad \lambda x.(x\ \lambda y.((x\ y)\ z))$$

6.2.1 Področja definicije spremenljivk

V vseh jezikih je pomembno področje definicije spremenljivk: del programa kjer je definirana neka spremenljivka.

Abstrakcija $\lambda x.E$ povezuje spremenljivko x v E

- x je novo definirana spremenljivka
- E je področje definicije x
- pravimo, da je x *vezana v* E
- podobno kot so argumenti funkcije vezani na telo funkcije

6.2.2 Proste in vezane spremenljivke

Spremenljivka je prosta v izrazu E, če ima pojavitve, ki niso vezane.

Rekurzivna definicija:

$FV(x) = x$
$FV(E_1 E_2) = FV(E_1) \cup FV(E_2)$
$FV(\lambda x.E) = FV(E) - \{x\}$

Primer:
$FV(\lambda x.x(\lambda y.x\ y\ z)) = \{z\}$

Proste spremenljivke so implicitno ali eksplicitno definirane izven izraza

6.2.3 Primeri λ izrazov

Lambda izraz z dvema lambda abstrakcijama in eno prosto spremenljivko.

$$\begin{array}{ll} & \lambda x.\lambda y.x\ z\ y \\ = & \lambda x.(\lambda y.((x\ z)\ y)) \end{array}$$

Še en primer povezovanja - porazdelitev z po dveh parametrih

$$\begin{array}{ll} & \lambda x.\lambda y.\lambda z.(x\ z)\ (y\ z) \\ = & \lambda x.(\lambda y.(\lambda z.((x\ z)\ (y\ z)))) \end{array}$$

In še en primer ...

$$\begin{array}{ll} & \lambda m.\lambda n.\lambda z.\lambda s.m\ (n\ z\ s)\ s \\ = & \lambda m.(\lambda n.(\lambda z.(\lambda s.((m\ ((n\ z)\ s))\ s)))) \end{array}$$

6.3 Evaluacija

6.3.1 Substitucija

Substitucija M za x v N - $[N/x]M$:

1. preimenuje vezane spremenljivke v M in N tako, da so unikatne,
2. izvedi tekstovno zamenjavo M za x v N

Pravila substitucije:

$[N/x]x = N$
$[N/x]z = z$, če $z \neq x$
$[N/x](L\ M) = ([N/x]L)([N/x]M)$
$[N/x](\lambda z.M) = \lambda z.([N/x]M)$, če $z \neq x \wedge z \notin FV(N)$

Primer:

$[y(\lambda x.x)/x]\lambda y.(\lambda x.x)y\ x$
$[y(\lambda v.v)/x]\lambda z.(\lambda u.u)z\ x$
$\lambda z.(\lambda u.u)z\ (y(\lambda v.v))$

6.3.2 Alfa konverzija

λ-izrazi, ki jih lahko pretvorimo med sabo s primenovanjem vezanih spremenljivk so identični

Primer: $\lambda x.x$ je identičen $\lambda y.y$

Konverzija:

$$\lambda x.M = \lambda y.([y/x]M) \quad \text{if } y \notin FV(M)$$

6.3.3 Beta redukcija

V osnovnem lambda računu je edini način za ovrednotenje izrazov aplikacija funkcij na argumentih

Beta redukcija:

$$(\lambda x.M)N \to [N/x]M$$

- $(\lambda x.M)$ imenujemo *redeks* (iz angl. reducable expression)

Redukcija λ-izraza:

- β-redukcija M v N: $M \to_\beta N$ ali $M \to N$
- β-izpeljava M v N: $M \to_\beta^* N$

Primeri:
$(\lambda x.x\ y)(u\ v) \to_\beta u\ v\ y$
$(\lambda x.\lambda y.x)z\ w \to_\beta (\lambda y.z)w \to z$
$(\lambda x.\lambda y.x)z\ w \to_\beta^* z$

M je *beta konvertibilen* z N ali $M =_\beta N$, če

- $M = N$, ali $\exists B : M \to^* B \wedge N \to^* B$.

Primer:

$(\lambda x.x)z =_\beta (\lambda x.\lambda y.x)z\ z$, zato ker
$(\lambda x.\lambda y.x)z\ w \to^* z$ in $(\lambda x.x)z \to^* z$.

6.3.4 Primeri evaluacije

Funkcija identitete:
$(\lambda x.x)E \rightarrow [E/x]x = E$

Še en primer z identiteto:
$(\lambda f.f(\lambda x.x))(\lambda x.x) \rightarrow$
$[(\lambda x.x)/f]f(\lambda x.x) = [(\lambda x.x)/f]f(\lambda y.y) \rightarrow$
$(\lambda x.x)(\lambda y.y) \rightarrow$
$[(\lambda y.y)/x]x = \lambda y.y$

Izpeljava, ki se ne zaključi:
$(\lambda x.xx)(\lambda y.yy) \rightarrow$
$[(\lambda y.yy)/x]xx = (\lambda x.xx)(\lambda y.yy) \rightarrow ...$

6.4 Programiranje v λ-računu

6.4.1 Curry

Lambda račun nima funkcij z večimi argumenti vendar je enostavno doseči isti učinek s *funkcijami višjega reda*.

Naj bo M izraz s prostima spremenljivkama x in y. Želimo napisati funkcijo F, ki za vsak par (N, L) zamenja x z N in y z L v izrazu M. V osnovnem lambda računu ne moremo napisati $F = \lambda(x, y).M$.

$F = \lambda x.\lambda y.M$

- F je funkcija, ki ob danem argumentu N vrne funkcijo, ki ob dani vrednosti za y vrne željeni rezultat.
- $FNL \rightarrow (\lambda y.[N/x]M)L \rightarrow [L/y][N/x]M$

Transformacijo, ki funkcijo z večimi argumenti prevede v funkcijo višjega reda imenujemo *Curry*

6.4.2 Kombinatorji

Znane funkcije, ki se pogosto uporabljajo.

Funkcija identitete

$$I = \lambda x.x$$

Funkcija, ki zavrže dan argument y in izračuna funkcijo identitete

$$K = \lambda y.(\lambda x.x)$$

Porazdelitev zadnjega parametera na prva dva

$$S = \lambda x.\lambda y.\lambda z.(x\ z)(y\ z)$$

Funkcija, ki se ne konča

$$\Omega = (\lambda x.x\ x)(\lambda x.x\ x)$$

Definicija rekurzivnih funkcij

$$Y = \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$$

Pomembna lastnost Y je $Y\ F =_\beta F\ (Y\ F)$.

$$\begin{array}{lcl} Y\ F & = & \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))\ F \\ & \rightarrow & (\lambda x.F(x\ x))(\lambda x.F(x\ x)) \\ & \rightarrow & F((\lambda x.F(x\ x))(\lambda x.F(x\ x))) \\ & \leftarrow & F((\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x)))\ F) \\ & = & F\ (Y\ F) \end{array} \tag{6.1}$$

6.4.3 Logične vrednosti

Kako uvedemo logične vrednosti $\{True, False\}$ v Lambda račun.

Logične vrednosti so funkcije $True,\ False$

$$True = \lambda t.\lambda f.t$$
$$False = \lambda t.\lambda f.f$$

if stavek: if $= \lambda l.\lambda m.\lambda n.\ l\ m\ n$

Primer izpeljave (redukcije) if stavka:

if $True\ M\ N$	$=$	$(\lambda l.\lambda m.\lambda n.\ l\ m\ n)\ True\ M\ N$	po definiciji
	$\rightarrow$	$(\lambda m.\lambda n.\ True\ m\ n)\ M\ N$	β-redukcija
	$\rightarrow$	$(\lambda n.\ True\ M\ n)\ M$	β-redukcija
	$\rightarrow$	$True\ M\ N$	β-redukcija
	$=$	$(\lambda t.\lambda f.t)\ M\ N$	po definiciji
	$\rightarrow$	$(\lambda f.M)\ N$	β-redukcija
	$\rightarrow$	M	β-redukcija

Logični IN

Logični IN lahko zapišemo v naslednji obliki:

$$And = \lambda b.\lambda c.\ b\ c\ False$$

V primeru, da je b resnično potem vrne c sicer $False$.
Izraz c vrne $True$ samo, če je izraz resničen...

Primer aplikacije funkcije And: $And\ True\ False$

6.4.4 Churchova števila

$$
\begin{aligned}
&C_0 = \lambda z.\lambda s.z \\
&C_1 = \lambda z.\lambda s.s\ z \\
&C_2 = \lambda z.\lambda s.s(s\ z) \\
&\vdots \\
&C_n = \lambda z.\lambda s.s(s(\ldots(s\ z)\ldots)
\end{aligned}
$$

Število n je predstavljeno s funkcijo C_n, ki ima dva argumenta z (zero) in s (successor) ter aplicira n kopij funkcije s na z.

Število n je predstavljeno s funkcijo, ki nekaj naredi n-krat.

Običajno aritmetične operacije na Churchovih številih so:

$Plus = \lambda m.\lambda n.\lambda z.\lambda s.m(n\ z\ s)s$
$Times = \lambda m.\lambda n.m\ C_0(Plus\ n)$

$(Plus\ 1\ 2) \rightarrow^* 3$
$Plus\ (\lambda z.\lambda s.s\ z)\ (\lambda z.\lambda s.s(s\ z)) \rightarrow$
$(\lambda m.\lambda n.\lambda z.\lambda s.m(n\ z\ s)s)\ (\lambda z.\lambda s.s\ z)\ (\lambda z.\lambda s.s(s\ z)) \rightarrow$
$(\lambda n.\lambda z.\lambda s.(\lambda z.\lambda s.s\ z)(n\ z\ s)s)\ (\lambda z.\lambda s.s(s\ z)) \rightarrow$
$\lambda z.\lambda s.(\lambda z.\lambda s.s\ z)((\lambda z.\lambda s.s(s\ z))\ z\ s)s \rightarrow$
$\lambda z.\lambda s.(\lambda z.\lambda s.s\ z)((\lambda s.s(s\ z))\ s)s \rightarrow$
$\lambda z.\lambda s.(\lambda z.\lambda s.s\ z)(s(s\ z))s =$
$\lambda z.\lambda s.(((\lambda z.\lambda s.s\ z)\ (s(s\ z)))s) \rightarrow$
$\lambda z.\lambda s.((\lambda s.s(s\ (s\ z)))s) \rightarrow$
$\lambda z.\lambda s.s(s\ (s\ z))$

Seštevanje ni preveč zabavno :(

6.4.5 Fakulteta

Intuitivna definicija funkcije, ki izračuna fakulteto danega števila.

if $n = 0$ then 1
else $n *$ (if $n - 1 = 0$ then 1
else $(n - 1) *$ (if $n - 2 = 0$ then1
else $(n - 2) * \ldots$

Rekurzijo lahko izrazimo s funkcijo $G = \lambda f.\langle telo\ f \rangle$ in $F = Y\ G$

$$\begin{array}{lll} F & = & Y\ G \\ & =_\beta & G\ (Y\ G) \\ & =_\beta & \langle telo\ (Y\ G) \rangle \\ & =_\beta & \langle telo\ \langle telo\ (Y\ G) \rangle \rangle \\ & & \vdots \end{array}$$

Funkcijo $Factorial$ lahko zdaj definiramo na sledeč način.

$$\begin{array}{lll} Fact & = & \lambda fact.\lambda n.\text{if}\ (IsZero\ n)\ C_1\ (Times\ n\ (fact(Pred\ n))) \\ Factorial & = & Y\ Fact \end{array}$$

Poglejmo si izpeljavo fakultete za C_2.

$Factorial\ C_2 = Y\ Fact\ C_2$
$=_\beta\ Fact\ (Y\ Fact)\ C_2$
$=_\beta\ (\lambda fact.\lambda n.\text{if}(IsZero\ n)\ C_1\ (Times\ n(fact(Pred\ n))))\ (Y\ Fact)\ C_2$
$=_\beta\ (\lambda n.\text{if}(IsZero\ n)\ C_1\ (Times\ n(Y\ Fact\ (Pred\ n))))\ C_2$
$=_\beta\ \text{if}(IsZero\ C_2)\ C_1\ (Times\ C_2(Y\ Fact\ (Pred\ C_2)))$
$=_\beta\ \text{if}\ False\ C_1\ (Times\ C_2(Y\ Fact\ C_1)))$
$=_\beta Times\ C_2(Y\ Fact\ C_1)$
$= Times\ C_2(Factorial\ C_1)$

6.4.6 Uporaba λ računa

Preverjanje tipov, lambda račun s tipi

Formalen jezik za opis pomena stavkov v operacijski in denotacijski semantiki

Osnova za interpreter programskega jezika.

Podroben študij gradnikov jezika.

Obstaja veliko razširitv lambda računa.

6.5 Lastnosti λ-računa

6.5.1 *Rekurziven jezik

Uvrstitev v hierarhijo Chomskega.

6.5.2 Church-Rosserjeva lastnost

Church-Rosserjeva (okr. CR) lastnost jezika zagotavlja unikatnost normalne oblike negelde na obstoj normalne oblike.

Church-Rosserjeva lastnost ima pomen za λ-račun brez tipov, kjer normalizacijski izrek ne velja.

Dokaz CR bomo podali malce kasneje. V splošnem velja, da CR ni enostavno dokazati, če pristopimo na običajen način. Poglejmo si najprej zapis izreka CR in nekatere posledice CR. Potem bomo podali še dokaz.

Izrek 6.5.1 (Church-Rosser) *Če* $t \rightarrow_\beta^* u$ *in* $t \rightarrow_\beta^* u'$ *potem obstaja* v *tako da* $u \rightarrow_\beta^* v$ *in* $u' \rightarrow_\beta^* v$.

Takojšnja posledica CR izreka je unikatnost normalne oblike za jezike, kjer velja CR.

Posledica 6.5.1 *Izraz t ima največ eno normalno obliko.*

Dokaz. Če velja $t \rightsquigarrow u, v$, kjer sta u in v v normalni obliki potem velja $u, v \rightsquigarrow w$ za nek w. Toda ker sta u in v v normalni obliki potem jih ni mogoče več naprej reducirati, velja mora torej $u = w = v$. □

Druga posledica CR je *konsistentnost* λ-računa: $\Lambda \not\vdash true =_\beta false$ ali, v splošnem, $\Lambda \not\vdash u =_\beta v$.

Konsistentnost jezika izhaja iz *logičnih* jezikov, kjer konsistentnost pomeni, da ni mogoče dokazati izjave in tudi negacije izjave.

Posledica 6.5.2 *λ-račun je konsistenten: ni res, da bi lahko dokazali enačbo $u =_\beta v$ za poljubna u in v.*

Dokaz.

- Če velja $u \rightsquigarrow v$ potem je enakost $u =_\beta v$ izpeljiva iz pravil za evaluacijo (redukcijo) in aksiomov enakosti.

- Obratno, če iz pravil za redukcijo in aksiomov enakosti sledi $u =_\beta v$ potem je enostavno videti, da obstajajo izrazi $U = t_0, t_1, \ldots, t_{2n-1}, t_{2n}$, tako da za $i \in [0..n-1]$ velja $t_{2i}, t_{2i+2} \rightsquigarrow t_{2i+1}$. S ponavljanjem aplikacije CR izreka končno dobimo $u, v \rightsquigarrow w$.

Če sta u in v dve različni normalni obliki istega tipa, potem ne obstaja takšen w, da bi lahko dokazali $u =_\beta v$. CR izrek torej pokaže denotacijsko konsistenco sistema. □

6.6 Opombe

Klasična predstavitev λ-računa je podana v članku Barendregt in Barendsen, Introduction to Lambda Calculus [2]. Obsežnejša predstavitev, ki vključuje dokaze pomembnejših izrekov, se nahaja v knjigi Barendregt z naslovom The Lambda Calculus: Its Syntax and Semantics [1].

Dokaza Church-Rosserjevega izreka in normalizacijskega izreka za λ-račun je predstavljen v knjigi Barendregt [1].

Poglavje 7

TIPI

V prejšnjih dveh poglavjih smo si ogledali dva jezika: aritmetične izraze in lambda račun.

Večkrat smo omenili, da nismo uporabili natančnega preverjanja zgradbe izrazov. Zaradi tega so bili lahko izrazi, ki so sicer sintaktično pravilni, pomensko napačni.

Tipi so osnovni mehanizem programskih jezikov, ki omogočajo bolj natančen opis izrazov.

V tem poglavju si bomo ogledali prirejanje tipov izrazom z uporabo statične in dinamične semantike ter nekatere lastnosti, ki jih imajo tipi izrazov programskih jezikov.

Delali bomo na primerih jezikov, ki smo jih že preučili: aritmetični izrazi in lambda račun.

Na koncu si bomo ogledeli ujemanje med tipi in klasično logiko, kar običajno imenujemo *Curry-Howardovo ujemanje* po ...

7.1 Aritmetični izrazi s tipi

Poglejmo si še enkrat sintakso aritmetičnih izrazov.

$$\begin{array}{rcl} \text{Izrazi } t & ::= & \text{true} \\ & & \text{false} \\ & & \text{if } t \text{ then } t \text{ else } t \\ & & 0 \\ & & \text{succ } t \\ & & \text{pred } t \\ & & \text{iszero } t \end{array} \tag{7.1}$$

V prejšnjih poglavjih smo videli, da se evaluacija izraza lahko zaključi v vrednosti, lahko pa pride do napake, ker pridemo do izraza, ki ni pomensko pravilen npr. pred $true$.

Označevanje izrazov s tipi ter sklepanje o tipih izrazov nam lahko pomaga pri zagotavljanju pravilnosti izrazov in programov.

Hoteli bi izvedeti—brez evaluacije izrazov—da se evaluacija izraza ne bo ustavila zaradi napake v tipih.

Da bi to lahko ugotovili moramo znati razlikovati med izrazi, katerih evaluacija se zaključi v celem številu in izrazom, katerega vrednost je boolova vrednost.

Za klasifikacijo izrazov bomo uporabili dva tipa: Nat in Bool.

Govorili bomo, da je "izraz t tipa T" ali "t pripada T" ali "t je element T" ..., kar pomeni, da se izraz t ovrednoti v tip T.

Lahko torej vidimo v času preverjanja (prevajanja) programa, da se program ovrednoti v vrednost pravilnega tipa.

Primer 7.1.1 *Izraz if* $true$ *then* $false$ *else* $false$ *je tipa Bool.*
Izraz succ(*pred*(*succ* 0)) *je tipa Nat.*

Naša analiza jezika bo *konzervativna* in bo uporabljala samo statične podatke. Na primer, izraz if $true$ then 0 else $false$ ne bo imel tipa.

7.1.1 Prirejanje tipov

Prirejanje tipov izrazom zapišemo $t : T$. Definirano je z pravili, ki povejo kako se tipi sestavljenih izrazov izračunajo iz tipov podizrazov.

V nadaljevanju bodo predstavljena pravila za prirejanje tipov aritmetičnih izrazov. Poglejmo si najprej boolove izraze t.j. izraze, ki vsebujejo boolove spremenljivke, konstante in operacije.

$$\frac{}{true : \text{Bool}} \tag{7.2}$$

$$\frac{}{false : \text{Bool}} \tag{7.3}$$

$$\frac{t_1 : \text{Bool} \qquad t_2 : T \qquad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \tag{7.4}$$

Naslednja pravila predstavljajo prirejanje tipov aritmetičnim izrazom.

$$\frac{}{0 : \text{Nat}} \tag{7.5}$$

$$\frac{t_1 : \text{Nat}}{\text{succ } t_1 : \text{Nat}} \tag{7.6}$$

$$\frac{t_1 : \text{Nat}}{\text{pred } t_1 : \text{Nat}} \tag{7.7}$$

$$\frac{t_1 : \text{Nat}}{\text{iszero } t_1 : \text{Bool}} \tag{7.8}$$

Definicija 7.1.1 *Prirejanje tipov je najmanjša binarna relacija med izrazi in tipi, ki zadošča vsem instancam pravil statične semantike jezika aritmetičnih izrazov 7.2-7.8.*

Pri sklepanju s tipi bomo večkrat uporabili izjavo: “če ima izraz določeno obliko potem mora imeti določen tip”.

Naslednji izrek podaja vse možne forme aritmetičnih izrazov in tipe podizrazov, ki sledijo iz dane strukture.

Izrek 7.1.1 (Inverzna relacija prirejanju tipov) *:*

1. *Če* $true : R$*, potem* $R =$ *Bool.*
2. *Če* $false : R$*, potem* $R =$ *Bool.*
3. *Če if* t_1 *then* t_2 *else* $t_3 : R$*, potem* t_1 *: Bool,* $t_2 : R$ *in* $t_3 : R$*.*
4. *Če* $0 : R$*, potem* $R =$ *Nat.*
5. *Če succ* t_1 *: Nat, potem* t_1 *: Nat.*
6. *Če pred* t_1 *: Nat, potem* t_1 *: Nat.*
7. *Če iszero* $t_1 : R$*, potem* $R =$ *Bool in* t_1 *: Bool.*

Dokaz. Sledi direktno iz pravil 7.2-7.8. □

Inverzni izrek imenujemo včasih *tvorni izrek* za tipe ker za pravilne tipe pokaže kako bi stavek lahko bil tvorjen.

Inverzni izrek vodi direktno k rekurzivnem algoritmu za računanje tipov izrazov. Izrek pove kako izračunati tipe iz tipov podizrazov za vsako sintaktično formo.

Vaja 7.1.1 *Dokaži, da je vsak pod-izraz dobro-definirnega izraza tudi dobro definiran.*

V prejšnjih poglavjih smo si ogledali izpeljave evaluacije izrazov. Podobno lahko naredimo tudi izpeljave tipov z drevesom, ki vsebuje instance pravil.

Primer 7.1.2 *V naslednjem primeru si bomo ogledali izpeljavo stavka* *if iszero* 0 *then* 0 *else pred* $0 : Nat$.

$$\frac{\dfrac{\dfrac{}{0 : Nat}}{iszero\ 0 : Bool} \qquad \dfrac{}{0 : Nat} \qquad \dfrac{\dfrac{}{0 : Nat}}{pred\ 0 : Nat}}{if\ iszero\ 0\ then\ 0\ else\ pred\ 0 : Nat} \tag{7.9}$$

□

Stavki so torej formalne izjave o tipih programov. *Pravila tipov* so implikacije med stavki in *izpeljave* so deduktivne osnovane na pravilih tipov

Izrek 7.1.2 (Enoličnost tipov) *Vsak izraz t ima natančno en tip, ki ima natančno eno izpeljavo na osnovi pravil 7.2-7.8.*

Dokaz. Strukturna indukcija na t z uporabo primerka inverznih pravil vsak primer strukture t posebaj. □

Aritmetični izrazi so tako enostavni, da ima vsak izraz natančno eno izpeljavo. Videli bomo, da izrazi v bolj kompleksnih jezikih imajo več kot eno izpeljavo.

7.1.2 Varnost = napredek + ohranitev

Osnovna lastnost sistema tipov je *varnost tipov*. Intuitivno pomeni varnost tipov, da pri programih s pravilno označenimi tipi ne pride do napak zaradi tipov.

V prejšnji sekciji smo že povedali, da do takšnih napak lahko pride, ko nimamo več pravil za ovrednotenje vrednosti in se evaluacija ustavi.

Pokazali bi radi, da programi, ki imajo dobro-definirane tipe so varni oz. da pri evaluaciji ne pridemo do mrtvega stanja.

To pakažemo v dveh korakih, ki sta poznana kot izreka o *napredku* in *ohranitvi*.

Napredek = Evaluacija izrazov z dobro-definiranimi tipi ne pride do mrtvega stanja.

Ohranitev = Evaluacija izrazov z dobro-definiranimi tipi v vsakem koraku vodi spet do izrazov z dobro-definiranimi tipi .

Te dve lastnosti skupaj pravita, da dobro definiran izraz med evaluacijo ne more priti do mrtvega stanja.

Za dokaz napredka bomo uporabili nekaj kanoničnih oblik izrazov.

Lema 7.1.1 (Kanonične oblike) *1. Vrednost tipa Bool je bodisi $true$ ali $false$.*

2. Vrednost tipa Nat je numerična vrednost definirana po pravilih 4.12-4.18.

Dokaz. Očitno. □

Izrek 7.1.3 (Napredek) *Predpostavimo, da je t dobro definiran izraz t.j. $t : T$ za nek T. Potem je t bodisi vrednost ali pa obstaja t' tako da $t \to t'$.*

Dokaz. Uporabili bomo indukcijo na izpeljavo $t : T$. Primeri 7.5, 7.2 in 7.3 so očitni, ker je t v vseh primerih vrednost. Za ostale primere si bomo ogledali dokaz.

Primer 7.4:

$$t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$$
$$t_1 : \text{Bool} \qquad t_2 : T \qquad t_3 : T$$

Po indukcijski hipotezi je t_1 bodisi vrednost bodisi obstaja t'_1, tako da $t_1 \to t'_1$. Če je t_1 vrednost potem nam lema o kanonični formi zagotavlja, da je vrednost bodisi $true$

ali $false$. V tem primeru t_1 ustreza pravilu 4.8 ali 4.9. Po drugi strani, če $t_1 \rightarrow t_1'$ potem $t \rightarrow$ if t_1' then t_2 else t_3 po pravilu 4.10.

Primer 7.6:

$$t = \text{succ } t_1 \qquad t_1 : \text{Nat}$$

Po indukcijski hipotezi je t_1 bodisi vrednost ali obstaja $t_1 \rightarrow t_1'$. V primeru vrednosti lema o kanonični obliki zagotavlja, da je numerična vrednost. Sicer velja $t_1 \rightarrow t_1'$ in $t \rightarrow$ succ t_1' po 4.12.

Ostale primere pokažemo na podoben način. □

Izrek 7.1.4 (Ohranitev) *Če $t : T$ in $t \rightarrow t'$ potem $t' : T$.*

Dokaz. Uporabili bomo indukcijo na izpeljavo $t : T$. Na vsakem koraku indukcije predpostavimo, da željena lastnost velja za vse podizraze: če $s \rightarrow s'$ potem s':S, kjer je s podizraz izraza t.

Dokaz bomo razdelili glede na možne oblike t. Izdelali ga bomo samo za nekatere izmed možnih primerov, medtem ko so ostali primeri podobni.

Primer 7.2: $t = true$
Če je zadnje uporabljeno pravilo 7.2 potem potem vemo, da $t = true$ in $B =$ Bool. Torej je t vrednost in ne more obstajati transformacija $t \rightarrow t'$.

Primer 7.4: $t =$ if t_1 then t_2 else t_3 $\quad t_1 :$ Bool $\quad t_2 : T \quad t_3 : T$
Če je zadnje uporabljeno pravilo v izpeljavi 7.4, potem vemo iz oblike pravila, da imamo podizpeljave za $t_1 :$ Bool, $t_2 : T$ in $t_3 : T$. Imamo tri pravila, ki se lahko uporabijo za transformacijo $t \rightarrow t'$: 4.8, 4.9 in 4.10.

Pod-primer 4.8: $t_1 = true \quad t' = t_2$
V primeru, da uporabimo pravilo 4.8 potem vemo, da je $t_1 = true$ in t' mora biti t_2, ki je tipa T! Podobno velja za primer uporabe pravila 4.9.

Pod-primer 4.10: $t_1 \rightarrow t_1' \quad t' =$ if t_1' then t_2 else t_3

Če je bilo uporabljeno pravilo 4.10, potem ob uporabi indukcijskih hipotez lahko sklepamo, da je tip t' enak kot tip t t.j. Bool.

Primer 7.5: $t = 0$ $T = \text{Nat}$
Primer je podoben primeru $t = true$. t je vrednost in se ne more naprej evaluirati.

Preostale primere obravnavamo na podoben način. □

Vaja 7.1.2 ... □

7.2 Jezik $\lambda_{\rightarrow}$

V tej sekaciji bo predstavljen lambda račun s tipi, jezik, ki ga pogosto imenujemo $\lambda_{\rightarrow}$. V skladu z našo terminologijo predstavljeno na začetku bomo ta jezik imenovali $\lambda_{\rightarrow}$.

Najprej si bomo ogledali nov tip $\rightarrow$. Lambda računu bomo dodali samo tip Bool, predvsem zaradi enostavnosti tako dobljenega jezika.

Pogledali si bomo pravila, ki definirajo prirejanje tipov v dinamični semantiki.

V drugem delu sekcije bomo definirali nekaj lastnosti, ki so bile predstavljene že v okviru aritmetičnih izrazov s tipi.

7.2.1 Funkcijski tip

Konstruirali bomo podoben sistem tipov kot v prejšnjem poglavju.

Sistem tipov bo vseboval tip Bool, ki ga bomo uporabili skupaj s čistim lambda lačunom. Zaradi enostavnosti jezika bomo izpustili cela števila.

Jezik, ki ga bomo konstrurali bo

- imel *varne tipe* in
- ne bo preveč konzervativen.

Ker je lambda račun enakovreden Turingovem stroju ne moremo pričakovati, da bomo lahko vedno priredili tip izrazu. Poglejmo si sledeč primer.

$$\text{if } \langle \text{obsežno računanje} \rangle \text{ then } true \text{ else } (\lambda x.x)$$

Stavek vrne bodisi $true$ ali funkcijo! Obsežno računanje se lahko ne ustavi, kar pomeni, da se tudi preverjanje tipov lahko ne ustavi.

Za razširitev boolovih izrazov s funkcijami uvedemo nov tip $\rightarrow$.

Če bi dodali novo pravilo za tipe

$$\lambda x.t :\rightarrow$$

bi lahko klasificirali enostavne izraze kot $\lambda x.x)$ kot tudi sestavljene izraze
if $true$ then $(\lambda x.true)$ else $(\lambda x.\lambda y.y)$ s tipom $\rightarrow$.

Pravilo očitno ni preveč natančno saj prva funkcija slika v Bool, izraz $(\lambda x.\lambda y.y)$ pa je drugačna funkcija.

Da bi lahko opisali koristne tipe potrebujemo več podatkov—moramo vedeti kakšen je tip argumentov in tip rezultata funkcij.

Enostavno karakterizacijo funkcije s tipom $\rightarrow$ moramo torej zamenjati s tipi oblike $T_1 \rightarrow T_2$, kjer je T_1 tip argumenta T_2 tip rezultata funkcije.

Definicija 7.2.1 *Množico enostavnih tipov nad tipom bool generiramo z naslednjo gramatiko.*

$$\begin{array}{lll} T & ::= & \textit{tip} \\ & \textit{Bool} & \textit{tip Bool} \\ & T \to T & \textit{tip funkcija} \end{array}$$

Konstruktor tipa $\to$ *je desno asociativen:* $T_1 \to T_2 \to T_3 \equiv T_1 \to (T_2 \to T_3)$. □

Primer 7.2.1 *Tip Bool* $\to$ *Bool je tip funkcije, ki slika iz argumenta argumenta tipa Bool v vrednost tipa Bool.*

Tip $(\textit{Bool} \to \textit{Bool}) \to (\textit{Bool} \to \textit{Bool})$ *ali ekvivalentno* $(\textit{Bool} \to \textit{Bool}) \to \textit{Bool} \to \textit{Bool}$ *je tip funkcije, ki vzame funkcijo tipa Bool* $\to$ *Bool in vrne drugo funkcijo tipa Bool* $\to$ *Bool.* □

7.2.2 Prirejanje tipov

Da bi lahko dodelili tip lambda abstrakciji $\lambda x.t$ moramo izračunati kaj se bo zgodilo, ko apliciramo abstrakcijo nad nekim objektom. Kako vemo kakšnega tipa bo argument?

Dva odgovora. Lahko označimo spremenljivko v lambda abstrakciji s pričakovanim tipom. Lahko pa tudi analiziramo telo lambda abstrakcije in izračunamo tip spremenljivke in lambda abstrakcije. Pri analizi tipov uporabljamo dedukcijo.

Zaenkrat predpostavimo, da bomo spremenljivke označevali s tipi. Namesto $\lambda x.t$ napišemo $\lambda x : T_1.t_2$, kjer je T_1 tip spremenljivke x.

Jeziki z eksplicitnimi tipi zahtevajo eksplicitno označevanje tipov, ki služijo postopku za preverjanje tipov. Jezike, ki znajo samo izpeljati tipe izrazov imenujemo *jeziki z implicitnimi tipi*.

Ko enkrat vemo tip argumenta lambda abstrakcije $\lambda x : T_1.t_2$ vidimo, da je tip rezultata funkcije enak tipu izraza t_2, ki ga bomo označili z T_2. Pričakujemo, da bodo vse

pojavitve spremenljivke x v t_2 tipa T_1. Lambda abstrakcijo lahko zdaj zapišemo $\lambda x : T_1.t_2 : T_1 \rightarrow T_2$. Pravilo, ki specificira tip lambda abstrakcije je sledeče.

$$\frac{x : T_1 \vdash t_2 : T_2}{\vdash \lambda x : T_1.t_2 : T_1 \rightarrow T_2} \tag{7.10}$$

Lambda abstrakcije lahko vsebujejo vgnezdene lambda abstrakcije, zato bi včasih potrebovali več različnih predpostavk o prostih spremenljivkah v izrazu $t : T$. To spremeni binarno relacijo $t : T$ v ternarno relacijo $\Gamma \vdash t : T$, kjer je Γ množica predpostavk o tipih prostih spremenljivk v t.

Pri sklepanju s tipi bomo uporabljali *kontekst tipov* Γ ali *okolje tipov* Γ. Kontekst tipov Γ vsebuje množico spremenljivk in njihovih tipov. Z operacijo "vejica" lahko razširimo kontekst tipov Γ z novo spremenljivko.

Prazen kontekst zapišemo $\emptyset$, ki ga velikokrat kar izpustimo. Izjava $\vdash t : T$ pomeni: zaprt izraz t ima tip T pri prazni množici predpostavk.

Pravilo, ki opisuje splošno obliko lambda abstrakcije je sledeče.

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.t_2 : T_1 \rightarrow T_2} \tag{7.11}$$

Pravilo, ki definira zvezo med Γ in spremenljivkami je sledeče.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \tag{7.12}$$

Premiso $x : T \in \Gamma$ beremo: "tip, ki ga predpostavljamo za x je T".

Poglejmo si na koncu še pravilo za aplikacijo.

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2} \tag{7.13}$$

Izraz t_1 je funkcija, ki slika argumente tipa T_1 v rezultat tipa T_2. Izraz t_2 se evaluira v vrednost tipa T_1. Rezultat apliciranja t_1 na t_2 je tipa T_2.

Primer 7.2.2 *Pravilo za določitev tipov boolovih izrazov 7.4 lahko zdaj instanciramo na funkcijski tip. V našem primeru dodelimo funkcijski tip vejam pogojnih stavkov.*

if $true$ *then* $(\lambda x : \mathit{Bool}.\ x)$ *else* $(\lambda x : \mathit{Bool}.\ not\ x)$
► $(\lambda x : \mathit{Bool}.\ x) : \mathit{Bool} \to \mathit{Bool}$

Primer 7.2.3 *Poglejmo si še en primer drevesa izpeljave, kot smo naredili za aritmetične izraze. Naredili bomo izpeljavo, ki pokaže da je izraz* $(\lambda x : \mathit{Bool}.x)\ true$ *ima tip Bool v praznem kontekstu.*

$$\dfrac{\dfrac{\dfrac{x : \mathit{Bool} \in x : \mathit{Bool}}{x : \mathit{Bool} \vdash x : \mathit{Bool}}7.12}{\vdash \lambda x : \mathit{Bool}.x : \mathit{Bool} \to \mathit{Bool}}7.11 \qquad \dfrac{}{\vdash true : \mathit{Bool}}7.2}{\vdash (\lambda x : \mathit{Bool}.x)\ true : \mathit{Bool}}7.13 \tag{7.14}$$

□

7.2.3 Statična semantika $\lambda_{\to}$

Poglejmo si zdaj v celoti jezik $\lambda_{\to}$—jezik, ki vsebuje samo funkcije.

Abstraktna sintaksa jezika $\lambda_{\to}$ je predstavljena s sledečo slovnico.

types τ	::=	bool	tip bool
		arr(τ_1, τ_2)	tip $\tau_1 \rightarrow \tau_2$
exprs e	::=	$x \mid$	spremenljivka
		lam$[\tau](x.e) \mid$	abstrakcija
		app(e_1, e_2)	aplikacija
		true	konstanta true
		false	konstanta false
		if(e_1, e_2, e_3)	if stavek

(7.15)

Konkretna sintaksa je predstavljena s sledečo tabelo.

Abstraktna sintaksa	Konkretna sintaksa
true	true
false	false
lam$[\tau](x.e)$	$\lambda x : \tau.e$
app(e_1, e_2)	$e_1\ e_2$
if(e_1, e_2, e_3)	if e_1 then e_2 else e_3

(7.16)

Spremenljivka x je *parameter* abstrakcije in e imenujemo *telo* lambda abstrakcije.

Izraz $e_1\ e_2$ je *aplikacija* funkcije e_1 na *argumentu* e_2. V primeru, da je e_1 λ abstrakcija $\lambda x : \tau.e$, potem se aplikacija $e_1\ e_2$ ovrednoti z $[e_1/x]e_2$ - zamenjava x z e_1 v e_2.

Statično semantiko $\lambda_{\rightarrow}$ definiramo z induktivnimi definicijami sodb (pravil) oblike $\Gamma \vdash x : \tau$, kjer je Γ končna množica hipotez $x : \tau$ in je x *spremenljivka*.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{7.17}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{lam}[\tau_1](x.e) : \text{arr}(\tau_1, \tau_2)} \tag{7.18}$$

$$\frac{\Gamma \vdash : e_1 : \text{arr}(\tau_2, \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \text{app}(e_1, e_2) : \tau} \tag{7.19}$$

Prirejanje tipov boolovim izrazom je predstavljeno s pravili 7.2-7.4.

Definirajmo spet inverzna pravila za določanje tipov izrazov glede na strukturo izraza.

Lema 7.2.1 (Inverzija) *Predpostavimo, da* $\Gamma \vdash e : \tau$.

1. *Če* $e = x$, *potem* $\Gamma = \Gamma', x : \tau$.
2. *Če* $e = lam[\tau_1](x.e)$, *potem* $\tau = arr(\tau_1, \tau_2)$ *in* $\Gamma, x : \tau_1 \vdash e : \tau_2$.
3. *Če* $e = app(e_1, e_2)$, *potem obstaja* τ_2 *tako, da* $\Gamma \vdash e_1 : arr(\tau_2, \tau)$ *in* $\Gamma \vdash e_2 : \tau_2$.

Dokaz. Dokaz sloni na indukciji po pravilih statične semantike. Za vsako pravilo je natančno en primer. Premise posameznih primerov zagotavljajo željen razultat. □

7.2.4 Dinamična semantika $\lambda_{\rightarrow}$

Dinamično semantiko $\lambda_{\rightarrow}$-računa definiramo z operacijsko semantiko na zaprtih izrazih.

Sodba e val, kjer je e zaprt izraz, je definirana induktivno.

$$\frac{}{\text{lam}[\tau](x.e)\ \text{val}} \tag{7.20}$$

Telo lambda abstrakcije e nima definiranih nobenih omejitev.

Imamo dve obliki semantike za funkcije: *klic-po-vrednosti* in *klic-po-imenu*. Pri metodi klic-po-vrednosti najprej ovrednotimo parametre lambda abstrakije in šele potem samo telo. Pri klicu-po-imenu se argument najprej prenese v izraz in se evaluira skupaj z izrazom.

Metoda klic-po-imenu prihrani evaluacijo argumentov v primeru, da sploh niso potrebni. Po drugi strani se lahko parametri ovrednotijo prav zaradi istega vzroka večkrat.

Metoda, klic-po-vrednosti je največkrat privzeta metoda. Semantika je definirana z naslednjimi pravili.

$$\frac{e_1 \mapsto e_1'}{\text{app}(e_1, e_2) \mapsto \text{app}(e_1', e_2)} \tag{7.21}$$

$$\frac{e_1 \text{ val} \quad e_2 \mapsto e_2'}{\text{app}(e_1, e_2) \mapsto \text{app}(e_1, e_2')} \tag{7.22}$$

$$\frac{e_2 \text{ val}}{\text{app}(\text{lam}[\tau_2](x.e_1), e_2) \mapsto [e_2/x]e_1} \tag{7.23}$$

Pravilo 7.21, ki je na prvem mestu, zahteva, da ovrednotimo najprej levo stran aplikacije, katere vrednost je lahko karkoli oz. ni potrebno, da je vrednost.

V čistem lambda računu so lambda abstrakcije edina oblika vrednosti. V primeru, da se e_1 reducira v vrednost, potem to mora biti lambda abstrakcija.

Pravilo 7.22 poskrbi, da se po ovrednotenju leve strani ovrednoti še desna stran aplikacije.

Šele ko tudi desna stran postane vrednost lahko apliciramo pravilo 7.23.

Ker e_2 v pravilu 7.23 mora biti vrednost se lahko pravilo ujame samo v primeru, da je desna stran izraza vrednost.

Vrstni red proženja pravil popolnoma določa vrstni red ovrednotenja aplikacije $e_1\ e_2$:

1. e_1 reduciramo v vrednost,
2. e_2 reduciramo v vrednost in
3. dejansko izvedemo aplikacijo.

Semantika klic-po-imenu je definirana s sledečimi pravili.

$$\frac{e_1 \mapsto e_1'}{\text{app}(e_1, e_2) \mapsto \text{app}(e_1', e_2)} \tag{7.24}$$

$$\frac{}{\text{app}(\text{lam}[\tau_2](x.e_1), e_2) \mapsto [e_2/x]e_1)} \tag{7.25}$$

Aplikacija je zdaj predstavljena z enim samim pravilom 7.24.

Pravilo ovrednoti levo stran aplikacije in šele nato začnemo z evaluacijo lambda abstrakcij.

Za razliko od pravila 7.23 prejšnje pravilo 7.25 nima pogoj, da je e_2 vrednost.

Kot vidimo iz primera pravil za evaluacijo $\lambda_\rightarrow$ je zelo pomemben vrstni red pravil, ki opišejo evaluacijo.

Vaja 7.2.1 *Spreminjaj vrstni red pravil in opazuj kako se spreminja evaluacijska strategija pravil.* □

7.2.5 Lastnosti tipov $\lambda_\rightarrow$

Najprej bomo razvili nekaj osnovnih lem in izrekov, ki nam bodo omogočile dokazati varnost tipov v $\lambda_\rightarrow$. Večina jih je podobnih tistim, ki smo jih prej razvili za $\mathcal{L}(\text{natbool})$.

Večja razlika je v *lemi o substituciji*, ki jo bomo pokazali za tipe $\lambda_\rightarrow$.

Poglejmo si najprej *inverzno lemo*, ki poda množico zaključkov o tem kako so izpeljave zgrajene.

Sklepali bomo nekako takole: “če ima izraz dobro definirane tipe, potem morajo imeti podizrazi naslednjo obliko...”.

Lema 7.2.2 (Inverzija relacije prirejanja tipov) *:*

1. *Če* $\Gamma \vdash x : T$ *then* $x : T \in \Gamma$.
2. *Če* $\Gamma \vdash \lambda x : T_1.t_2 : R$*, then* $R = T_1 \rightarrow R_2$ *za nek* R_2 *tako da* $\Gamma, x : T_1 \vdash t_2 : T_2$
3. *Če* $\Gamma \vdash t_1\ t_2 : R$*, potem obstaja nek* T_1 *tako da* $\Gamma \vdash t_1 : T_1 \rightarrow R$ *in* $\Gamma \vdash t_2 : T_1$
4. *Če* $\Gamma \vdash true : R$*, potem* $R =$ *Bool.*
5. *Če* $\Gamma \vdash false : R$*, potem* $R =$ *Bool.*
6. *Če* $\Gamma \vdash$ *if* t_1 *then* t_2 *else* $t_3 : R$*, potem* $\Gamma \vdash t_1 :$ *Bool,* $\Gamma \vdash t_2 : R$ *in* $\Gamma \vdash t_3 : R$.

□

Dokaz. Direktno po definiciji relacije tipov. □

Vaja 7.2.2 ...

Izrek 7.2.1 (Unikatnost tipov) *V danem kontekstu* Γ *ima izraz* t *enoličen tip. Še več, obstaja ena samo izpeljava tega tipa.* □

Dokaz. Za vajo. □

Za precej sistemov tipov, ki jih bomo videli v nadaljevanju predstavljena zveza med izrazi in izpeljavami ne velja. En izraz bo lahko imel več tipov in vsak bo imel lahko več izpeljav tipov.

Pri takšnih jezikih bomo precej delali na tem, da pokažemo, da lahko izpeljave tipov rekonstruiramo iz izrazov.

Lema 7.2.3 (Kanonične oblike) 1. *Če je* v *vrednost tipa Bool, potem je bodisi* $true$ *ali* $false$.

2. *Če* v *vrednost tipa* $T_1 \rightarrow T_2$*, potem* $v = \lambda x : T_1.t_2$. □

Dokaz. Direktno iz predstavljenih pravil. □

V nadaljevanju bomo dokazali izrek o napredku na podoben način, kot je bil dokazan izrek o napredku aritmetičnih izrazov.

Izjavo v izreku bomo morali prilagoditi tako, da bomo govorili o *zaprtih izrazih*, ki nimajo prostih spremenljivk.

Za odprte izraze izrek dejansko ne velja: izraz $f\ true$ na primer je v normalni obliki ni pa vrednost. Podobni primeri niso napaka v jeziku—korektni programi so vsi zaprti izrazi.

Izrek 7.2.2 (Napredek) *Predpostavimo, da je t zaprt, dobro-definiran izraz $\vdash t : T$. Potem je t bodisi vrednost ali pa obstaja nek t' tako da $t \to t'$.* □

Dokaz. Indukcija na izpeljavi tipa. Primeri za boolove konstante in pogojni izraz so enaki kot pri dokazu za napredek pri aritmetičnih izrazih. Primer spremenljivke se ne more zgoditi, ker je t zaprt. Primer abstrakcije je direkten, ker so abstrakcije vrednosti.

Edini zanimiv primer je aplikacija, kjer je $t = t_1\ t_2$ ter $\vdash t_1 : T_1 \to T_2$ in $\vdash t_2 : T_1$.

Po indukcijski hipotezi je bodisi t_1 vrednost ali lahko naredimo korak evaluacije; enako velja za t_2. Če lahko t_1 razvijemo za en korak, potem lahko uporabimo pravilo za aplikacijo 7.21 nad t. Če je t_1 vrednost in t_2 ni potem lahko uporabimo pravilo 7.22. Končno, če sta t_1 in t_2 vrednosti potem je t_1 oblike $\lambda x : T_3.t_3$ in lahko uporabimo pravilo 7.23. □

Lema 7.2.4 (Permutacije) *Če $\Gamma \vdash t : T$ in je Δ permutacija Γ, potem $\Delta \vdash t : T$. Globina obeh izpeljav je enaka.* □

Dokaz. Direktna indukcija na pravila za izpeljave tipov. □

Lema 7.2.5 (Oslabitev) *Če* $\Gamma \vdash t : T$ *in* $x \notin dom(\Gamma)$*, potem* $\Gamma, x : S \vdash t : T$*. Obe izpeljavi imata isto globino.* □

Dokaz. Indukcija na pravila za izpeljave tipov. □

Z uporabo prej predstavljenih tehničnih lem lahko zdaj dokažemo ključno lastnost relacije tip: dobro-definirani tipi se ohranjajo, če spremnljivko zamenjamo z izrazi primernega tipa.

Podobne leme igrajo pomembno vlogo pri dokazovanju varnosti programskih jezikov—pogosto se jih imenuje “substitucijska lema”.

Lema 7.2.6 (Ohranitev tipov ob substituciji) *Če* $\Gamma, x : S \vdash t : T$ *in* $\Gamma \vdash s : S$*, potem* $\Gamma \vdash [x/s]t : T$*.* □

Dokaz. Po indukciji na izpeljavo stavka $\Gamma, x : S \vdash t : T$. Za dano izpeljavo obdelamo primere na končnem pravilu uporabljenem v dokazu. Najbolj zanimivi primeri so tisti s spremenljivkami in abstrakcijami.

Pravilo 7.17:

$$t = z, \text{ kjer je } z : T \in (\Gamma, x : S)$$

Imamo dva primera, ki jih je potrebno obdelati: $z = x$ in z je neka druga spremenljivka. V primeru $z = x$ velja $[x/s]z = s$. Zahtevan rezultat $\Gamma \vdash s : S$ je med predpostavkami! V drugem primeru pa velja $[x/s]z = z$, kjer je rezultat direkten.

Pravilo 7.18:

$$\begin{array}{l} t = \lambda y : T_2.t_1 \\ T = T_2 \rightarrow T_1 \\ \Gamma, x : S, y : T_2 \vdash t_1 : T_1 \end{array}$$

Lahko predpostavimo, da velja $x \neq y$ in $y \notin FV(s)$. Z uporabo permutacije na podizpeljavi dobimo $\Gamma, y : T_2, x : S \vdash t_1 : T_1$. Z uporabo oslabitve na dani izpeljavi $\Gamma \vdash s : S$ dobimo $\Gamma, y : T_2 \vdash s : S$. Po indukcijski hipotezi velja $\Gamma, y : T_2 \vdash [x/s]t_1 : T_1$. Po pravilu 7.23 velja $\Gamma \vdash \lambda y : T_2.[x/s]t_1 : T_2 \rightarrow T_1$. To pa je natančno to kar potrebujemo za dokaz, ker velja $[x/s]t = \lambda y : T_2.[x/s]t_1$.

Pravilo 7.19:

$$t = t_1\ t_2$$
$$\Gamma, x : S \vdash t_1 : T_2 \to T_1$$
$$\Gamma, x : S \vdash t_2 : T_2$$
$$T = T_1$$

Po indukcijski hipotezi velja $\Gamma \vdash [x/s]t_1 : T_2 \to T_1$ in $\Gamma \vdash [x/s]t_2 : T_2$. Z uporabo pravila 7.13 $\Gamma \vdash [x/s]t_1\ [x/s]t_2$ ali $[x/s](t_1\ t_2) : T$.

Pravilo 7.2:

$$t = true$$
$$T = \text{Bool}$$

Potem velja $[x/s]t = true$ in željen rezultat sledi $\Gamma \vdash [x/s]t : T$.

Pravilo 7.3:

$$t = false$$
$$T = \text{Bool}$$

Podobno.

Pravilo 7.4:

$$t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$$
$$\Gamma, x : S \vdash t_1 : \text{Bool}$$
$$\Gamma, x : S \vdash t_2 : T$$
$$\Gamma, x : S \vdash t_3 : T$$

Tri uporabe induktivnih hipotez vodijo do

$$t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$$
$$\Gamma, x : S \vdash [x/s]t_1 : \text{Bool}$$
$$\Gamma, x : S \vdash [x/s]t_2 : T$$
$$\Gamma, x : S \vdash [x/s]t_3 : T$$

Z uporabo pravila 7.4 dobimo željen rezultat. □

Izrek 7.2.3 (Ohranitev) *Če* $\Gamma \vdash t : T$ *in* $t \to t'$, *potem* $\Gamma \vdash t' : T$. □

Dokaz. Struktura dokaza je podobna kot pri dokazu za ohranitev aritmetičnih izrazov, razen da lahko uporabimo lemo o ohranitvi tipov po substituciji.

... □

7.3 Normalizacija

Normalizacijski izrek zagotavlja eksistenco normalne oblike, medtem ko Church-Rosserjeva lastnost zagotavlja unikatnost normalne oblike.

Normalizacijski izrek ima dve obliki:

- *šibek normalizacijski izrek* pravi, da obstaja neka strategija za normalizacijo izraza, ki se zaključi.
- *Močen normalizacijski izrek* pravi, da se vse možne strategije normalizacije zaključijo.

7.3.1 Definicija λ-računa

Poglejmo si še enkrat varianto lambda računa $\lambda_{\rightarrow}$, ki bo uporabljena za prikaz izomorfizma.

$$
\begin{array}{llcll}
Tip & \tau & ::= & T_1, \ldots, T_n & osnovni\,tipi \\
 & & | & \tau_1 \times \tau_2 & produkt \\
 & & | & \tau_1 \rightarrow \tau_2 & funkcije \\
Izraz & e & ::= & x_0^T, x_1^T, \ldots, x_n^T & spremenljivke \\
 & & | & \langle e_1, e_2 \rangle & pari \\
 & & | & \Pi^1 e & projekcija \\
 & & | & \Pi^2 e & projekcija \\
 & & | & \lambda x.e & abstrakcija \\
 & & | & e\ e & aplikacija
\end{array}
\tag{7.26}
$$

Predpostavljamo, da imamo na voljo atomične tipe $T_1, \ldots, T_n$.

Uporaba tipov $\times, \rightarrow$ bo ustrezala logičnim operacijam $\wedge, \Rightarrow$.

Dokaze v izjavnem računu bomo videli kot izraze $\lambda_{\rightarrow}$ računa. Bolj natančno, dokaz izjave A postane $\lambda_{\rightarrow}$ izraz tipa A.

7.3.2 Šibki normalizacijski izrek

Pri izračunu normalne oblike izraza t imamo lahko več možnosti pri izbiri pravila, ki ga apliciramo za redukcijo izraza t. Na vsakem koraku imamo več možnih redukcij, vendar samo končno mnogo.

Šibki normalizacijski izrek pravi, da lahko najdemo pravilno redukcijo, ki vodi do normalne oblike. Ne izključuje pa možnosti slabih redukcij, ki ne vodijo v normalno obliko. Zato govorimo o šibki normalizaciji.

Izrek 7.3.1 (Šibka normalizacija) *Naj bo t izraz lambda računa brez tipov. Obstaja metoda za evaluacijo t, ki se zaključi.*

Dokaz bo sestavljen iz večih delov. Najprej bomo definirali stopnjo izraza s katero bomo lahko spremljali velikost izraza med redukcijo.

Stopnja $\rho(T)$ tipa T je definirana kot:

- $\rho(T_i)$ = 1 če je T_i atomičen.
- $\rho(U \times V) = \rho(U \to V) = max(\rho(U), \rho(V)) + 1$.

Stopnja $\rho(r)$ redeksa r je definirana z:

- $\rho(\Pi^1\langle u, v\rangle) = \rho(\Pi^2\langle u, v\rangle) = \rho(U \times V)$, kjer je $U \times V$ tip u, v.
- $\rho((\lambda x.v)\ u) = \rho(U \to V)$, kjer je $U \to V$ tip $(\lambda x.v)$.

Stopnja $d(t)$ izraza t je maksimum stopenj redeksov, ki jih vsebuje. Izraz t v normalni obliki ima stopnjo $d(t)$ enako 0.

Redeks r ima dve stopnje: kot redeks in kot izraz. Druga stopnja je večja ali enaka prvi $d(t) \geq \rho(t)$.

Ob uporabi substitucije se zmanjša stopnja izraza. Velikost izraza, ki je rezultat substitucije lahko ocenimo na naslednji način.

Lema 7.3.1 *Če je x tipa U potem* $d([u/x]t) \leq max(d(t), d(u), \rho(U))$.

Dokaz. V izrazu $[u/x]t$ imamo lahko:

- redekse, ki sestavljajo t in kjer je x postal u,
- redeksi u, ki zamenjajo vse pojavitve x,
- nove redekse v primeru, da se x pojavlja v kontekstu $\Pi^1 x$ (ali $\Pi^2 x$ ali $x\ v$) in je $u = \langle u', u'' \rangle$ (ali $\langle u', u'' \rangle$ ali $\lambda y.u'$). Te novi redeksi imajo stopnjo U. □

Poglejmo zdaj še razmerje med stopnjo izraza in stopnjo reduciranega izraza. Najprej vidimo, da v primeru, da imamo redeks r tipa T, potem velja $\rho(r) > \rho(T)$.

Lema 7.3.2 *Če $t \rightsquigarrow u$ potem* $d(u) \leq d(t)$.

Dokaz. Obravnavati moramo samo primer, ko imamo eno samo redukcijo: u dobimo iz t, če zamenjamo r s c. Stanje je zelo podobno tistemu iz prejšnje Leme t.j. u vsebuje naslednje komponente.

- Redekse, ki so bili v t in ne tudi v r spremenjeni z zamenjavo r s c (kar ne vpliva na stopnjo).
- Redekse iz c, ki jih dobimo s poenostavitvijo r ali z interno substitucijo v r: $(\lambda x.s)\ s'$ postane $[x/s']s$, Lema 7.3.1 pa pravi, da $d([x/u]t) \leq max(d(t), d(u), \rho(T))$, kjer je T tip od x. Po drugi strani je $\rho(T) < d(r)$, torej $d(c) \leq d(r)$.
- Redekse, ki pridejo od zamenjave r z c. Situacija je spet podobna tisti pri Lemi 7.3.1: ti redeksi imajo stopnjo enako $\rho(T)$, kjer je T tip r in velja $\rho(T) < \rho(r)$. □

Naslednji korak dokaza šibkega noramlizacijskega izreka je definicija redukcije maksimalne stopnje, ki bo omogočala, da bo stopnja pri vsaki redukciji manjša od stopnje izraza pred redukcijo.

Lema 7.3.3 *Naj bo r redeks maksimalne stopnje n iz t. Predpostavimo, da imajo vsi redeksi vsebovani v r stopnjo manj kot n. Če dobimo u iz t z zamenjavo r s c, potem ima u striktno manjšo stopnjo od n.*

Dokaz. Ob zamenjavi se zgodi sledeče:

- Redeksi izven r ostanejo takšni kot so.
- Redeksi, ki so znotraj r so v splošnem ohranjeni, vendar lahko tudi pomnoženi: na primer če zamenjamo $(\lambda x.x, x)\ s$ z $\langle s, s\rangle$ se redeksi s pomnožijo. Tudi v tem primeru ostane stopnja manj od n.
- Redeks r je zamenjan z redeksi manjše stopnje. □

Dokaz. [Šibki normalizacijski izrek] Naj bo t izraz in funkcija $\mu(t) = (n, m)$, kjer je $n = d(t)$ in $m =$ število redeksov stopnje n.

Lema 7.3.3 pravi, da lahko iz t izberemo redeks r tako, da po konverziji r v c velja $\mu(t') \leq \mu(t)$: če $\mu(n', m')$ potem $n' \leq n$ in $(n' = n \wedge m' \leq m$. Rezultat dokažemo z dvojno indukcijo. □

7.3.3 Močen normalizacijski izrek

7.4 Curry-Howardov izomorfizem

Curry-Howardov izomorfizem prikaže sintaktično in pomensko zvezo med logiko in teorijo tipov. V tej sekciji si bomo ogledali najbolj osnovno zvezo, ki definirana med *izjavnim računom* in *λ-računom s tipi* ($\lambda_{\rightarrow}$). Kasneje bodo omenjene še druge zveze, ki so definirane nad jeziki z večjo izrazno močjo.

Definirali bomo preslikavo med izrazi logike in izrazi $\lambda_{\rightarrow}$ tako, da bo deduktivno sklepanje na strani logike preslikano v normalizacijo izrazov λ-računa oz. bomo hkrati

definirali tudi preslikavo v nasprotno smer. Dokazi torej ustrezajo ovrednotenju izrazov v λ-računu. Preslikava torej ni samo bijektivna ampak tudi ohranja preslikave pri deduktivnem sklepanju na eni strani oz. pri normalizaciji izrazov na drugi, analogni strani.

7.4.1 Definicija izomorfizma

Na eni strani imamo dokaze izjav napisanih v izjavnem računu. Za dokazovanje uporabljamo naravno dedukcijo, ki je bila predstavljena v Poglavju 2.

Na drugi strani imamo izraze $\lambda_{\rightarrow}$, ki imajo tipe. Evaluacija izraza bo ustrezala sklepanju v izjavnem računu.

Poglejmo si zdaj pravila, ki definirajo bijekcijo med izrazi $\lambda_{\rightarrow}$ in izpeljavami dokazov izjavne logike.

1. Dedukcija A ustreza spremenljivki x_i^A.
2. Dedukcija $\dfrac{\begin{array}{cc}\vdots & \vdots \\ A & B\end{array}}{A \wedge B}\wedge\mathcal{I}$ ustreza izrazu $\langle u, v\rangle$, kjer u ustreza izpeljavi A in v izpeljavi B.
3. Dedukciji $\dfrac{\begin{array}{c}\vdots \\ A \wedge B\end{array}}{A}\wedge 1\mathcal{E}$ in $\dfrac{\begin{array}{c}\vdots \\ A \wedge B\end{array}}{B}\wedge 2\mathcal{E}$ ustrezata izrazoma $\Pi^1 t$ in $\Pi^2 t$, kjer t ustreza dedukciji $A \wedge B$.
4. Dedukcija $\dfrac{\begin{array}{c}[A] \\ \vdots \\ B\end{array}}{A \Rightarrow B}\Rightarrow\mathcal{I}$ ustreza izrazu $\lambda x_i^A.v$, kjer izpeljava A ustreza predpostavkam za izpeljavo x_i^A ter izpeljava B iz A ustreza ovrednotenju v.
5. Dedukcija $\dfrac{\begin{array}{cc}\vdots & \vdots \\ A & A \Rightarrow B\end{array}}{B}\Rightarrow\mathcal{E}$ ustreza izrazu $t\ u$, kjer t in u ustrezata izpeljavama $A \Rightarrow B$ in B.

7.4.2 Pomen izomorfizma

7.5 Opombe

...

Noramalizacija izrazov lambda računa s tipi je podana v knjigi Barendregt [1]. Povzetek dokazov šibkega in močnega normalizacijskega izreka za λ-račun s tipi je predstavljen v knjigi Girarda [3].

Različne verzije dokaza normalizacijskega izreka za lambda račun s tipi se nahajajo v člankih avtorjev Hindley [], Berger [], Filinski [], ...

Poglavje 8

REKURZIJA

8.1 $Gödelov$ T

Jezik $\mathcal{L}\{\text{nat} \rightarrow\}$ je bolj poznan kot $Gödelov$ T. Jezik je razširitev jezika $\lambda_{\rightarrow}$ za delo naravnimi števili.

Namesto množice operacij nad celimi števili je uporabljena *primitivna rekurzija* s katero lahko izrazimo vse osnovne operacije nad naravnimi števili.

Primitivna rekurzija je primerna za induktivno definicijo operacij nad celimi števili.

Pomembna lastnost $\mathcal{L}\{\text{nat} \rightarrow\}$ je izražanje samo *totalnih* funkcij. To pomeni, da je jezik $\mathcal{L}\{\text{nat} \rightarrow\}$ zasnovan tako, da se izvajanje izrazov nujno zaključi.

Posledica stroge zahteve glede forme jezika onemogoča izražanje nekaterih rekurzivnih funkcij v $\mathcal{L}\{\text{nat} \rightarrow\}$.

8.1.1 Statična semantika $\mathcal{L}\{\textbf{nat} \rightarrow\}$

Jezik $\mathcal{L}\{\text{nat} \rightarrow\}$ je kombinacija jezikov $\mathcal{L}\{\text{nat}\}$, ki ga bomo definirali zdaj in $\lambda_{\rightarrow}$, ki je bil predstavljen prej.

Sintaksa $\mathcal{L}\{\text{nat}\}$ je definirana z naslednjo gramatiko.

$$\begin{array}{lll} Tipi\ \ \tau & ::= & \text{nat} \\ Izrazi\ e & ::= & z \mid s(e) \mid \text{rec}[\tau](e, e_0, x.y.e_1) \end{array}$$

Naslednja tabela prikazuje povezave s konkretno sintakso.

$Abstraktno$	$Konkretno$
z	z
$s(e)$	$s(e)$
$\text{rec}[\tau](e, e_0, x.y.e_1)$	rec e $\{z \Rightarrow e_0 \mid s(x)$with $y \Rightarrow e_1\}$

Števila so predstavljena kot *Churcheva števila*

Izraz $\text{rec}[\tau](e, e_0, x.y.e_1)$ imenujemo *primitivna rekurzija*.

e-kratna iteracija s transformacijo $x.y.e_1$ in začetno vrednostjo e_0.

Statična semantika $\mathcal{L}\{\text{nat}\}$ je definirana z naslednjimi pravili.

$$\frac{}{\Gamma, x : \text{nat} \vdash x : \text{nat}} \tag{8.1}$$

$$\frac{}{\Gamma \vdash z : \text{nat}} \tag{8.2}$$

$$\frac{\Gamma \vdash e : \text{nat}}{\Gamma \vdash s(e) : \text{nat}} \tag{8.3}$$

$$\frac{\Gamma \vdash e : \text{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \text{nat}, y : \tau \vdash e_1 : \text{nat}}{\Gamma \vdash \text{rec}[\tau](e, e_0, x.y.e_1) : \tau} \tag{8.4}$$

Substitucijska lema pravi, da se tipi v e' ohranijo ob substiciji spremenljivke x z izrazom e.

Lema 8.1.1 (Substitucija) *Če velja* $\Gamma \vdash e : \tau$ *in* $\Gamma, x : \tau \vdash e' : \tau'$*, potem* $\Gamma \vdash [e/x]e' : \tau'$.

8.1.2 Dinamična semantika $\mathcal{L}\{\textbf{nat}\}$

Podobno kot smo razlikovali med klicem-po-vrednosti in klicem-po-referenci, razlikujemo med *takojšnjo* in *leno* interpretacijo operacije naslednika $s()$.

Takojšnja interpretacija najprej ovrednoti parameter in šele nato ovrednoti naslednika. To zagotavlja, da je vsaka zaprta vrednost tipa nat kompozicija naslednikov, ki se začne z nič z.

Lena interpretacija obravnava $s(e)$ kot vrednost, kjer e ni nujno, da je vrednost. Tako se izognemo izračunu predhodnika v primeru, da to ni potrebno. Obe interpretaciji si bomo še ogledali v naslednjih poglavjih.

Takojšnja dinamična semantika je definirana z naslednjimi pravili.

$$\frac{}{z \text{ val}} \tag{8.5}$$

$$\frac{e \text{ val}}{s(e) \text{ val}} \tag{8.6}$$

$$\frac{e \mapsto e'}{s(e) \mapsto s(e')} \tag{8.7}$$

$$\frac{e \mapsto e'}{\text{rec}[\tau](e, e_0, x.y.e_1) \mapsto \text{rec}[\tau](e', e_0, x.y.e_1)} \tag{8.8}$$

$$\frac{}{\text{rec}[\tau](z, e_0, x.y.e_1) \mapsto e_0} \tag{8.9}$$

$$\frac{}{\begin{array}{c}\text{rec}[\tau](s(e), e_0, x.y.e_1) \\ \mapsto \\ [e, \text{rec}[\tau](e, e_0, x.y.e_1)/x, y]e_1\end{array}} \quad (8.10)$$

Pravila 8.6 in 8.7 definirata *takojšnjo* semantiko za operacijo naslednik.

Pravila 8.9 in 8.10 definirata obnašanje rekurzivnega operatorja na z in $s(e)$.

V prvem primeru dobimo rezultat e_0.

V drugem primeru se x poveže z e in y z rekurzivnim klicem istega operatorja. Parameter e zmanjšamo za eno. V primeru, da e_1 ne vsebuje y se rekurzivni klic ne izvede.

Usklajenost med statično in dinamično semantiko ter tipi izrazov lahko preverimo preko varnosti izrazov: za izraze mora veljati *napredek* in *ohranitev*.

Izrek 8.1.1 (Varnost) *:*

1. *Če* $e : \tau$ *in* $e \mapsto e'$ *potem* $e' : \tau$.
2. *Če* $e : \tau$ *potem je lahko kvečjemu* e *val ali* $e \mapsto e'$ *za nek* e'.

8.1.3 Izrazna moč T

T lahko izrazi *primitivne rekurzivne funkcije*.

Primitivna rekurzija je postopek, ki definira vrednost funkcije pri argumentu n z uporabo vrednosti, ki je rezultat vrednosti funkcije z argumentom $n - 1$.

Jezik primitivnih rekurzivnih funkcij je *rekurzivni jezik* v hierarhiji teorije izračunljivosti, jezik za katerega obstaja Turingov stroj, ki se vedno ustavi.

Rekurziven jezik imenujemo tudi izračunlijiv ali rešljiv.

Podobno kot v $\lambda_{\rightarrow}$ imamo v Goedlovem T naslednje kanonične oblike vrednosti.

Lema 8.1.2 (Kanonične oblike) *Če je $e : \tau$ in e val potem velja naslednje.*

1. *Če je $\tau =$ nat potem $e = s(s(\ldots z))$ kjer velja $n \geq 0$.*
2. *Če je $\tau = \tau_1 \rightarrow \tau_2$ potem $e = \lambda x : \tau_1.e_2$ za nek e_2.*

Goedel je T uporabljal za študij izrazne moči formalizmov za izražanje matematičnih funkcij nad celimi števili. Poglejmo si par primerov izrazov s katerimi realiziramo aritmetične operacije..

Primer 8.1.1 *V naslednjem primeru uporabe funkcije rec bomo definirali funkcijo dub, ki podvoji vrednost parametra.*

$$\begin{array}{l} \mathit{dub} = \lambda x : nat.rec\ x\ \{z \rightarrow z \mid s(u)\ with\ v \rightarrow s(s(v))\} \\ \mathit{dub} = \lambda x : nat.rec[\mathsf{nat} \rightarrow \mathsf{nat}](x, z, u.v.s(s(v))) \end{array}$$

□

Z indukcijo lahko pokažemo, da funkcija dub v zgornjem primeru res izračuna funkcijo $\text{dub}(x) = 2 * x$. Podobno lahko dokažemo pravilnost definicije naslednjih primerov aritemetičnih funkcij.

Primer 8.1.2 *Naslednji primer prikaže rekurzivno definicijo funkcije pred izraženo s primitivno rekurzijo*[1].

$$\begin{array}{l} \mathit{pred} = \lambda x : nat.rec\ x\ \{z \rightarrow z \mid s(u)\ with\ v \rightarrow \mathsf{if z}(u, z, s(v))\} \\ \mathit{pred} = \lambda x : nat.rec[\mathsf{nat} \rightarrow \mathsf{nat}](x, z, u.v.\mathsf{if z}(u, z, s(v))) \end{array}$$

□

[1] Funkcijo pred je s primitivno rekurzijo predstavil Kleene.

Primer 8.1.3 *Poglejmo si še seštevanje. Funkcija ima dva parametra: x in y. S primitivno rekurzijo odvijamo (odštevamo 1) vrednost x. Ko pridemo do ničle vrne funkcija rec y. Ob vračanju rekurzije nad y konstruiramo x s funkcijo $s()$. Rezultat je vsota $x + y$.*

$$add = \lambda x : nat.\lambda y : nat.rec\, x\, \{z \rightarrow y \mid s(u)\ with\ v \rightarrow s(v)\}$$

□

Primer 8.1.4 *In še množenje. Funkcija ima dva parametra: x in y. S primitivno rekurzijo odvijamo (odštevamo 1) vrednost x. Ko pridemo do ničle vrne funkcija rec z. Ob vračanju rekurzije prištejemo y k produktu x-krat. Rezultat je produkt $x * y$.*

$$mul = \lambda x : nat.\lambda y : nat.rec\, x\, \{z \rightarrow z \mid s(u)\ with\ v \rightarrow \mathsf{add}(y, v)\}$$

□

Ackermann-ova funkcija je ena od prvih odkritih funkcij, ki je *totalna rekurzivna funkcija* in ni primitivna rekurzivna funkcija.

Vse primitivne rekurzivne funkcije so totalne (definirane za celotno domeno) in izračunljive oz. rekurzivne po hierarhiji teorije jezikov in izračunljivosti.

Primer 8.1.5 *Naslednji primer prikaže definicijo Ackermann-ove funkcije, ki je definirana z naslednjimi rekurzivnimi enačbami.*

$$\begin{aligned} A(0, n) &= n + 1 \\ A(m + 1, 0) &= A(m, 1) \\ A(m + 1, n + 1) &= A(m, A(m + 1, n)) \end{aligned}$$

Pri vsakem rekurzivnem klicu bodisi m pada bodisi ostane enak in se n zmanjšuje. Iz tega stališča so rekurzivni klici dobro definirani.

... □

8.2 Plotkinov PCF

Drug pristop k razširitvi $\lambda_{\rightarrow}$ do uporabnega programskega jezika je opustitev garancije za ustavitev fukcij zato, da bi dobili večjo izraznost jezika.

Dovoljeni so izrazi, ki ni nujno, da se ustavijo. Dokaz ustavitve je v zavesti programerja, ki napiše funkcijo.

Dobimo širši jezik, ki je enakovreden Turingovem stroju.

Jezik $\mathcal{L}(\text{nat} \rightharpoonup)$ kombinira $\mathcal{L}\{\text{nat}\}$ in $\lambda_{\rightarrow}$ s splošno rekurzijo. Dobimo splošno orodje za izražanje samo-referenčnih izrazov.

PCF funkcije so lahko *nedefinirane* za nekatere vrednosti parametorov funkcije, kar pomeni, da so funkcije *parcialne*. Iz tega razloga uporabimo notacijo "zlomljeno puščico": $\rightharpoonup$.

Ključni element $\mathcal{L}(\text{nat} \rightharpoonup)$ je uporaba operatorja fiksne točke za izražanje rekurzivnih definicij.

Spomnimo se kombinatorja Y, ki se uporablja za implementacijo rekurzije.

$$Y = \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$$

Pomembna lastnost Y je $Y\ F =_{\beta} F\ (Y\ F)$.

V matematiki lahko definiramo funkcije z rekurzivnimi enačbami. Na primer:

$$\begin{aligned} f(0) &= 1 \\ f(n+1) &= (n+1) * f(n) \end{aligned}$$

Funkcija, ki jo iščemo je rešitev rekurzivne enačbe.

Videli bomo, da se rekurzivne funkcije lahko izrazijo z operatorejem fix.

8.2.1 Vrednosti in izračuni

Izrazi $\mathcal{L}(\text{nat} \rightharpoonup)$ ni nujno, da se evaluirajo v vrednosti. Za tiste izraze, katerih evaluacija se ne ustavi, pravimo, da *divergirajo*.

Koristno je razlikovati med izrazi, ki vedno divergirajo in izrazi, ki ne divergirajo. Kateri izrazi divergirajo? Kateri izrazi se ovrednotijo v vrednost? Kaj če so nekatere spremenljivke proste?

...

Za enostavnejše razlikovanje med spremenljivkami bomo uporabljali nekaj dogovorov glede notacije.

Imena sestavljena iz majhnih črk iz konca abecede x, y, z in izpeljanke bomo obravnavali kot spremenljivke z vrednostjo tip nat kot tudi λ abstrakcije.

Ista imena sestavljena iz velikih črk X, Y, Z bodo označevala spremenljivke katerih vrednost je izraz, ki predstavlja kodo v rekurzivnih funkcijah.

Predpostavljamo torej x val, medtem ko ni mogoče izpeljati X val, ker lahko izvajanje divergira.

8.2.2 Splošna rekurzija

Splošna rekurzija ima naslednjo obliko.

$$\text{Izrazi } e ::= \text{fix}[\tau](X.e) \tag{8.11}$$

Konkretna sintaksa $\text{fix}[\tau](X.e)$ je fix $X : \tau$ is e. Izraz fix $X : \tau$ is$e$ je *samo-referenčen*, ker X predstavlja sam izraz.

Statična semantika splošne rekurzije je definirana z naslednjim pravilom.

$$\frac{\Gamma, X : \tau \vdash e : \tau}{\Gamma \vdash \text{fix}[\tau](X.e) : \tau} \tag{8.12}$$

To pravilo predstavi samo-referenčno naravo rekurzije. Ker je spremenjlivka X referenca na sam rekurzivni izraz, predpostavimo, da $X : \tau$ in preverimo, če potem $e\ \tau$.

Dinamična semantika splošne rekurzije je podana z naslednjim pravilom.

$$\frac{}{\text{fix}[\tau](X.e) \mapsto [\text{fix}[\tau](X.e)/X]e} \tag{8.13}$$

Pravilo implementira samo-referenco z zamenjavo rekurzivnega izraza samega za spremenljivko X v telesu izraza. To imenujemo tudi *odvijanje* rekurzije.

Poglejmo si zdaj nekaj primerov uporabe operacije fix.

Primer 8.2.1 *Prvi primer prikaže uporabo operacije fix za definicija fakultete.*

$$\textit{fix } f : \textit{nat} \to \textit{nat } is\ \lambda(x : \textit{nat}.\textit{if iszero } x \textit{ then } s(x) \textit{ else } x * f(\textit{pred } x))$$

□

Primer 8.2.2 *Prvi primer uporabi operacijo fix za definicijo funkcije $iseven$, ki vrne $true$ v primeru, da je parameter funkcije sodo število.*

$$
\begin{array}{l}
fix\ f : nat \to \textit{Bool}\ is \\
\quad \lambda(x : nat. \\
\qquad \textit{if iszero}\ x\ \textit{then}\ true \\
\qquad \textit{else if iszero}\ (\textit{pred}\ x)\ \textit{then}\ false \\
\qquad \textit{else}\ f(\textit{pred}\ (\textit{pred}\ x))
\end{array}
$$

□

Spremenljivka X v fix$[\tau](X.e)$ je splošna spremenljivka, ki lahko tudi ne doseže vrednosti na tleh.

Primer izraza, ki divergira je fix$[\tau](X.X)$, kjer se e vedno zamenja sama s seboj. Obstajajo druge in bolj kompleksne funkcije, ki divergirajo.

Še par enostavnih primerov. Izpeljava izraza $x + x$ konvergira proti vrednosti, medtem ko izpeljava $X + X$ ne konvergira.

Razliko med izrazi, ki se ovrednotijo in splošnimi izračuni lahko primerjamo z razliko med parcialnimi in totalnimi funkcijami.

8.2.3 Statična semantika PCF

Sintaksa kompletnega jezika PCF je definirana z naslednjo gramatiko.

$$
\begin{array}{rcll}
 & & Abstraktno & Konkretno \\
Tip\ T & ::= & nat & nat \\
 & | & \text{par}(T_1, T_2) & \tau_1 \rightharpoonup \tau_2 \\
Izraz\ e & ::= & x & x \\
 & | & z & z \\
 & | & s(e) & s(e) \\
 & | & \text{ifz}(e, e_0, X.e_1) & \text{ifz } e\ \{z \to\ e_0\ \mid\ s(x) \to\ e_1\} \\
 & | & \text{lam}[\tau](x.e) & \lambda(x : \tau.e) \\
 & | & \text{app}(e_1, e_2) & e_1\ (e_2) \\
 & | & \text{fix}[\tau](X.e) & \text{fix } X : \tau\ is\ e
\end{array}
\tag{8.14}
$$

Izraz fix$[\tau](X.e)$ imenujemo splošna rekurzija. Operacija je podrobneje predstavljena v prejšnji sekciji.

Izraz ifz$(e, e_0, x.e_1)$ razveji glede na vrednost e. Če ima e vrednost z potem vrne e_0, sicer pa e_1. V zadnjem primeru se rekurzivni klic poveže z X.

Statična semantika $\mathcal{L}(\text{nat} \rightharpoonup)$ je induktivno definirana z naslednjimi pravili.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \tag{8.15}$$

$$\frac{}{z : nat} \tag{8.16}$$

$$\frac{\Gamma \vdash e : nat}{\Gamma \vdash s(e) : nat} \tag{8.17}$$

Premisa v oklepaju definira obnašanje v primeru *takojšnje* semantike evaluacije.

$$\frac{\Gamma \vdash e : nat \qquad \Gamma \vdash e_0 : \tau \qquad \Gamma, x : nat \vdash e_1 : \tau}{\Gamma \vdash \text{ifz}(e, e_0, x.e_1) : \tau} \tag{8.18}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{lam}[\tau_1](x.e) : \text{par}(\tau_1, \tau_2)} \tag{8.19}$$

$$\frac{\Gamma \vdash e_1 : \text{par}(\tau_2, \tau) \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \text{app}(e_1, e_2) : \tau} \tag{8.20}$$

$$\frac{\Gamma, \text{fix}[\tau](X.e) : \tau \vdash [\text{fix}[\tau](X.e)/X]e : \tau}{\Gamma \vdash \text{fix}[\tau](X.e) : \tau} \tag{8.21}$$

Pravilo 8.21 ujame bistvo rekurzivne samo-reference z zamenjavo primerka X z rekurzivnim izrazom med preverjanjem tipov.

Sklepanje je "krožno" v tem, da preveri fix$[\tau](X.e) : \tau$, predpostavimo, da je tako in izpeljemo, da $[\text{fix}[\tau](X.e)/X]e : \tau$.

Alternativno pravilo 8.21 obravnava rekurzivno samo-referenco kot spremenljivko.

$$\frac{\Gamma, X : \tau \vdash e : \tau}{\Gamma \vdash fix[\tau](X.e) : \tau} \tag{8.22}$$

Pri preverjanju rekurzivnega izraza predpostavljamo, da ima spremenljivka X tip τ, medtem ko preverjamo, da ima telo tip τ.

Prednost pravila 8.21 je, da se izognemo obravnavanju X kot spremenljivko. Ker se pomen X ne spreminja, predstavlja sam rekurzivni izraz.

Pravilo 8.22 lahko izpeljemo iz pravila 8.21 kot tudi obratno.

8.2.4 Dinamična semantika PCF

Izjava e val pove, da je izraz (zaprta) vrednost.

Definicija te sodbe se spremenja v odvisnosti od tega ali uporabljamo takojšnjo ali leno semantiko $\mathcal{L}(\text{nat} \rightharpoonup)$.

Izjava val je definirana z naslednjimi pravili:

$$\frac{}{z \text{ val}} \tag{8.23}$$

$$\frac{\{e \text{ val}\}}{s(e) \text{ val}} \tag{8.24}$$

$$\frac{}{\text{lam}[\tau](x.e) \text{ val}} \tag{8.25}$$

Premisa v oklepajih je izpuščena v primeru lene semantike in je vključena v primeru takojšnje semantike.

Dinamična semantika $\mathcal{L}(\text{nat} \rightharpoonup)$ je definirana z naslednjimi pravili:

$$\left\{\frac{e \to e'}{s(e) \to s(e')}\right\} \tag{8.26}$$

$$\frac{e \to e}{\text{ifz}(e, e_0, x.e_1) \to \text{ifz}(e, e_0, x.e_1)} \tag{8.27}$$

$$\frac{}{\text{ifz}(z, e_0, x.e_1) \to e_0} \tag{8.28}$$

$$\frac{}{\text{ifz}(s(e), e_0, x.e_1) \to [e/x]e_1} \tag{8.29}$$

$$\frac{e_1 \to e_1'}{\text{app}(e_1, e_2) \to \text{app}(e_1', e_2)} \tag{8.30}$$

$$\left\{\frac{e_1 \text{ val} \qquad e_2 \to e_2'}{\text{app}(e_1, e_2) \to \text{app}(e_1, e_2')}\right\} \tag{8.31}$$

$$\frac{\{e_2 \text{ val}\}}{\text{app}(\text{lam}[\tau](x.e), e_2) \to [e_2/x]e} \tag{8.32}$$

$$\frac{}{\text{fix}[\tau](X.e) \to [\text{fix}[\tau](X.e)/X]e} \tag{8.33}$$

Enako kot v primeru definicije val se pravila in premise v oklepajih izpustijo v primeru lene semantike in se vključijo v primeru takojšnje semantike $\mathcal{L}(\text{nat} \rightharpoonup)$.

Pravilo 8.33 implementira samo-referenco z zamenjavo samega rekurzivnega izraza za spremenljivko X v telesu. (odvijanje rekurzije).

Izrazi PCF so varni: evaluacija se ne zaključi v izrazu, ki ni vrednost in tipi izrazov se ohranijo pri transformaciji.

Izrek 8.2.1 (Varnost) *1. Če $e : \tau$ in $e \to e'$, potem $e : \tau$. 2. Če $e : \tau$, potem bodisi velja $e\ val$ ali obstaja e, tako da $e \to e'$.*

Dokaz. Dokaz ohranitve je z indukcijo na izpeljavo sodbe tranzicije. Poglejmo si pravilo 8.33.

Predpostavimo fix$[\tau](x.e) : \tau$. Po inverznem izreku velja fix$[\tau](x.e) : \tau \vdash [\text{fix}[\tau](x.e)/x]e : \tau$, iz česar sledi rezultat direktno zaradi tranzitivnosti izjave.

Dokaz napredka sledi po indukciji na izpeljavi tipov. Na primer, za pravilo 8.32 rezultat sledi direktno, ker lahko naredimo napredek z odvijanjem rekurzije. □

8.2.5 Izrazna moč PCF

Splošna rekurzija je zelo fleksibilna tehnika, ki dopušča definicijo širokega nabora funkcij v okviru $\mathcal{L}(\text{nat} \rightharpoonup)$.

Slaba lastnost v primerjavi z *primitivno rekurzivnimi* funkcijami je možnost divergence pri evaluaciji izrazov.

Ni mogoče, tako kot v T, pokazati, da sintaksa in semantika jezika zagotavljata zaključitev izračuna izrazov. Dokaz o zaključitvi evaluacije izraza mora izdelati programer sam.

PCF je po izrazni moči enak *Turingovem jeziku*, jeziku v katerem lahko zapišemo katerikoli izrazljiv program, ki ni nujno, da se zaključi.

Izrazna moč PCF je torej enaka vsem splošnim programskim jezikom.

Poglejmo si kot primer kako lahko primitivno rekurzijo izrazimo v PCF.

Primer 8.2.3 *Stavek rec iz T, ki omogoča izražanje primitivne rekurzije lahko v PCF izrazimo s funkcijo fix.*

$$\textit{rec}\ e\ \{z \to e_0 \mid\ s(x)\ \textit{with}\ y \to e_1]\}$$

Funkcija, ki sprejme izraz e kot parameter in realizira rec, je naslednja.

$$\mathit{fix}\ f : \tau\ is\ \lambda u.\mathit{ifz}\ u\ \{z \rightarrow e_0 \mid\ s(x) \rightarrow [f(x)/y]e_1$$

□

8.2.6 *Denotacijska semantika PCF

8.3 Opombe

Poglavje povzema material o Goedlovem T in Plotkinovem PCF predstavljenem v Harperjem učbeniku z naslovom *Foundations for Programming Languages* [5].

Jezik T je predstavljen v knjigi Jean-Yves Girard z naslovom *Proofs and Types* [3], kjer je podan normalizacijski izrek za T. Primitivne rekurzivne funkcije so predstavljene v člankih Nordstörm [], ...

Obširna predstavitev PCF, ki vključuje dokaze za zaključitev evaluacije, povezavo med denotacijsko in operacijsko semantiko se nahaja v Plotkinovem članku *LCF Considered as A Programming Language* [11].

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 $(\lambda x : \text{Unit}.t_2)\ t_1$.

Izrek 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 $(\lambda x : \text{Unit}.t_2)\ t_1$, kjer je x enolična znotraj t_2.*

- $t \to_E t'$ *čče* $e(t) \to_I e(t')$
- $\Gamma \vdash^E t : T$ *čče* $\Gamma \vdash^I e(t') : T$

□

Dokaz. Po indukciji na konstrukcijo izrazov. □

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 ML, na primer, napišemo let $x = t_1$ in t_2, kar pomeni: naj bo vrednost x enaka t_1 v izrazu t_2.

Sintaksa let stavka je sledeča.

$$\begin{array}{rcl} Tip\, t & ::= & \ldots \\ & & \text{let } x = t \text{ in } t \end{array} \tag{9.1}$$

Statična semantika stavka je definirana z naslednjim pravilom.

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

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

$$\frac{}{\text{let } x = v_1 \text{ in } t_2 \rightarrow [x/v_1]t_2} \tag{E-LETV}$$

$$\frac{t_1 \rightarrow t_1'}{\text{let } x = t_1 \text{ in } t_2 \rightarrow \text{let } x = t_1' \text{ in } t_2} \tag{E-LETV}$$

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 Ničnи in binarni produkt

Abstraktna sintaksa binarnega produkta je definirana s sledečo gramatiko.

$$
\begin{array}{llll}
\mathit{Kategorija} & \mathit{Ime} & \mathit{Abstraktno} & \mathit{Konkretno} \\
\mathit{Tip} & \tau & ::= \text{Unit} & \text{Unit} \\
 & & \mid \text{prod}(\tau_1, \tau_2) & \tau_1 \times \tau_2 \\
\mathit{Izraz} & e & ::= \mathit{triv} & \{\} \\
 & & \mid \text{pair}(e_1, e_2) & \{e_1, e_2\} \\
 & & \mid \text{fst}(e) & e.1 \\
 & & \mid \text{snd}(e) & e.2
\end{array}
\tag{9.2}
$$

Tip $\text{prod}(\tau_1, \tau_2)$ imenujemo tudi binarni produkt tipov τ_1 in τ_2. Tip Unit imenujemo tudi ničnи produkt. Tipi, ki jih dobimo s produktom vključujejo tudi ničnи in binarni produkt.

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

$$\frac{}{\Gamma \vdash \mathit{triv} : \text{Unit}} \tag{S-Triv}$$

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

$$\frac{\Gamma \vdash e : \text{prod}(\tau_1, \tau_2)}{\Gamma \vdash \text{fst}(e) : \tau_1} \tag{S-Proj1}$$

$$\frac{\Gamma \vdash e : \text{prod}(\tau_1, \tau_2)}{\Gamma \vdash \text{snd}(e) : \tau_2} \tag{S-Proj2}$$

Dinamična semantika parov je definirana na sledeč način.

$$\frac{}{\mathit{triv}\ \text{val}} \tag{D-Triv}$$

$$\frac{\{e_1 \text{ val} \quad e_2 \text{ val}\}}{\text{fst}(\text{pair}(e_1, e_2)) \mapsto e_1} \qquad \text{(D-ParBeta1)}$$

$$\frac{\{e_1 \text{ val} \quad e_2 \text{ val}\}}{\text{snd}(\text{pair}(e_1, e_2)) \mapsto e_2} \qquad \text{(D-ParBeta2)}$$

$$\frac{e \mapsto e'}{\text{fst}(e) \mapsto \text{fst}(e')} \qquad \text{(D-Proj1)}$$

$$\frac{e \mapsto e'}{\text{snd}(e) \mapsto \text{snd}(e')} \qquad \text{(D-Proj2)}$$

$$\left\{ \frac{e_1 \mapsto e_1'}{\text{pair}(e_1, e_2) \mapsto \text{pair}(e_1', e_2)} \right\} \qquad \text{(D-Par1)}$$

$$\left\{ \frac{e_1 \text{ val} \quad e_2 \mapsto e_2'}{\text{pair}(e_1, e_2) \mapsto \text{pair}(e_1, e_2')} \right\} \qquad \text{(D-Par2)}$$

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

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.*

$$\begin{array}{ll} & \{\textit{pred}\ 4, \textit{if}\ true\ \textit{then}\ false\ \textit{else}\ false\}.1 \\ \mapsto & \{3, \textit{if}\ true\ \textit{then}\ false\ \textit{else}\ false\}.1 \\ \mapsto & \{3, false\}.1 \\ \mapsto & 3 \end{array} \qquad (9.3)$$

□

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.*

$$
\begin{array}{ll}
 & (\lambda x : \mathit{nat} \times \mathit{nat}.\ x.2)\{\mathit{pred}\ 4, \mathit{pred}\ 5\} \\
\mapsto & (\lambda x : \mathit{nat} \times \mathit{nat}.\ x.2)\{3, \mathit{pred}\ 5\} \\
\mapsto & (\lambda x : \mathit{nat} \times \mathit{nat}.\ x.2)\{3, 4\} \\
\mapsto & \{3, 4\}.2 \\
\mapsto & 4
\end{array}
\tag{9.4}
$$

□

Koherenco med statično in dinamično semantiko pokažemo na običajen način.

Izrek 9.2.1 (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).

Dokaz. ... □

9.2.2 N-kratni produkti

Enostavno je posplošiti binarne produkte na n-kratne produkte. Na primer, $\{1, 2, true\}$ vsebuje števila in boolovo vrednost. Sintaksa n-teric in operacij nad n-tericami je definirana s sledečimi pravili.

$$
\begin{array}{lllll}
izrazi: & t & ::= & \ldots & \\
n-terica & & ::= & \{{t_i}^{i=1..n}\} & \\
projekcija & & & t.i & \\
vrednosti: & v & ::= & \ldots & \text{(Sintaksa)} \\
v.n-terice & & & \{{v_i}^{i=1..n}\} & \\
tipi: & T & ::= & \ldots & \\
t.n-terice & & & \{{T_i}^{i=1..n}\} &
\end{array}
$$

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 $\{T_i^{i \in 1..n}\}$.

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

$$\frac{\forall i \in [1..n] : \Gamma \vdash t_i : T_i}{\Gamma \vdash \{{t_i}^{i\in 1..n}\} : \{{T_i}^{i\in 1..n}\}} \quad \text{(S-Nter)}$$

$$\frac{\Gamma \vdash t_1 : \{{T_i}^{i\in 1..n}\}}{\Gamma \vdash t_1.j : T_j} \quad \text{(S-Proj)}$$

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

$$\frac{}{\{v_i^{i\in 1..n}\}.j \to v_j} \quad \text{(D-Triv)}$$

$$\frac{t_1 \to t_1'}{t_i.i \to t_i'.i} \quad \text{(D-Proj)}$$

$$\frac{t_j \to t_j'}{\{{v_i}^{j\in 1..j-1}, t_j, {t_k}^{k\in j+1..n}\} \to \{{v_i}^{j\in 1..j-1}, t_j', {t_k}^{k\in j+1..n}\}} \quad \text{(D-Nter)}$$

9.2.3 Zapisi

Posplošitev n-teric na označene zapise je precej direktna. Preprosto označimo vsako komponento t_i z oznako $l_i \in \mathcal{L}$.

Sintaksa zapisov je definirana na sledeč način.

$$\begin{array}{lcll} & & & \text{izrazi:} \\ t & ::= & \{l_i = {t_i}^{i=1..n}\} & \text{zapis} \\ & & t.l & \text{projekcija} \\ & & & \text{vrednosti:} \\ v & ::= & \{l_i = {v_i}^{i=1..n}\} & \text{vrednost zapisa} \\ & & & \text{tipi:} \\ T & ::= & \{l_i : {T_i}^{i=1..n}\} & \text{tip zapisa} \end{array} \quad (9.5)$$

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.

$$\frac{\forall i \in [1..n] : \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = {t_i}^{i \in 1..n}\} : \{l_i : {T_i}^{i \in 1..n}\}} \tag{9.6}$$

$$\frac{\Gamma \vdash t_1 : \{{T_i}^{i \in 1..n}\}}{\Gamma \vdash t_1.l_j : T_j} \tag{9.7}$$

Evaluacijska pravila, ki definirajo dinamično semantiko zapisov so naslednja.

$$\overline{\{l_i = {v_i}^{i \in 1..n}\}.j \to v_j} \tag{9.8}$$

$$\frac{t_1 \to t_1'}{t_i.l \to t_i'.l} \tag{9.9}$$

$$\frac{t_j \to t_j'}{\{l_i = {v_i}^{j \in 1..j-1}, l_j = t_j, l_k = t_k^{k \in j+1..n}\} \to \{l_i = {v_i}^{j \in 1..j-1}, l_j = t_j', l_k = t_k^{k \in j+1..n}\}} \tag{9.10}$$

Vaja 9.2.1 *Primer semantike ujemanja vzorcev na zapisih...*

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.

$$\begin{array}{lcl} PhysicalAddress & = & \{firstlast : \text{String}, addr : \text{String}\} \\ VirtualAddress & = & \{name : \text{String}, email : \text{String}\} \end{array} \tag{9.11}$$

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.

$$Addr \quad = \quad PhysicalAddress + VirtualAddress; \tag{9.12}$$

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.

$$\begin{array}{l} \text{inl} : PhysicalAddress \rightarrow Addr \\ \text{inr} : VirtualAddress \rightarrow Addr \end{array} \tag{9.13}$$

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 $Addr$.

$$getName = \lambda a : Addr.\text{case } a \text{ of inl } x \Rightarrow x.firstlast \mid \text{inr } y \Rightarrow y.name; \quad (9.14)$$

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

$$\begin{array}{llll} Tipi & \tau & ::= & \text{void} \mid \text{sum}(\tau_1, \tau_2) \\ Izrazi & e & ::= & \text{inl}[\tau](e) \mid \text{inr}[\tau](e) \mid \\ & & & \text{case}[\tau_1, \tau_2](e, x_1.e_1, x_2.e_2) \end{array} \quad (9.15)$$

Konkretna sintaksa je predstavljena s sledečo tabelo.

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

(9.16)

Tip void 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.

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

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

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

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.

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

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

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

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

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

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

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

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.*

$$\begin{array}{lcl} \textit{Tipi } \tau & ::= & \textit{bool} \\ \textit{Izrazi } e & ::= & \textit{tt} \mid \textit{ff} \mid \textit{if}(e, e_1, e_2) \end{array} \quad (9.17)$$

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

$$\begin{array}{rcl} \mathit{bool} & = & \mathit{sum}(\mathit{Unit}, \mathit{Unit}) \\ \mathit{tt} & = & \mathit{inl}[\mathit{bool}](triv) \\ \mathit{ff} & = & \mathit{inr}[\mathit{bool}](triv) \\ \mathit{if}(e, e_1, e_2) & = & case[\mathit{Unit}, \mathit{Unit}](e, x1.e1, x2.e2) \end{array} \tag{9.18}$$

Spremenljivki x_1 in x_2 sta odveč, ker njihov tip $unit$ določi njihovo vrednost $triv$, 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 $\lambda_{\rightarrow}$ 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. 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 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. Ta možnost je lahko implementirana 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.

$$OptionalNat = \langle none : \text{unit}, some : \text{nat} \rangle; \tag{9.19}$$

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

$$Table = \textsf{nat} \to OptionalNat; \tag{9.20}$$

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

$$\begin{array}{ll} & emptyTable = \lambda n : \textsf{nat}.\langle none : \textsf{unit} \rangle \; as \; OptionalNat; \\ \blacktriangleright & emptyTable : Table \end{array} \tag{9.21}$$

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

$$\begin{array}{lll} & extendTable = & \lambda t : Table.\lambda m : \textsf{nat}.\lambda v : \textsf{nat}.\lambda n : \textsf{nat}. \\ & & \textsf{if} \; equal \; n \; m \; \textsf{then} \langle some = v \rangle \; as \; OptionalNat \\ & & \textsf{else} \; t \; n \\ \blacktriangleright & extendTable : & Table \to \textsf{nat} \to \textsf{nat} \to Table \end{array} \tag{9.22}$$

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

$$\begin{array}{ll} x = & \textit{case}\, t(5)\, \textit{of} \\ & \langle none = u \rangle \rightarrow 0 \mid \\ & \langle some = v \rangle \rightarrow v; \end{array} \tag{9.23}$$

□

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 in spremenljivkami zato

preverjanje tipov zavrne marsikateri program, ki se zdi sprejemljiv. Poglejmo na primer pravilo za aplikacijo funkcije.

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2} \tag{9.24}$$

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.

$$S <: S \qquad \text{(T-REFL)}$$

Drugič, relacija podtip mora biti tranzitivna.

$$\frac{S <: U \qquad U <: T}{S <: T} \qquad \text{(S-TRANS)}$$

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.

$$\frac{}{\{l_i : T_i{}^{i\in 1..n+k}\} <: \{l_i : T_i{}^{i\in 1..n}\}} \quad \text{(S-RCDWIDTH)}$$

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.

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

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 $\{x : \{a : Nat\}, y : \{\}\}$.*

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

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

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

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.

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

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

□

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

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

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 zaloga 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.

$$
\begin{array}{lll}
t & ::= & \text{izrazi} \\
 & x & \text{spremenljivka} \\
 & \lambda x : T.t & \text{abstrakcija} \\
 & t\ t & \text{aplikacija} \\
v & ::= & \text{vrednosti} \\
 & \lambda x : T.t & \text{vrednost abstrakcije} \\
T & ::= & \text{tipi} \\
 & \text{Top} & \text{maksimalen tip} \\
 & T \rightarrow T & \text{funkcijski tip} \\
\Gamma & ::= & \text{konteksti} \\
 & \emptyset & \text{prazen kontekst} \\
 & \Gamma, x : T & \text{vezana spremenljivka}
\end{array}
\tag{9.25}
$$

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

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \tag{9.26}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.t_2 : T_1 \rightarrow T_2} \tag{9.27}$$

$$\frac{\Gamma \vdash x : T_{11} \rightarrow T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \tag{9.28}$$

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T} \tag{9.29}$$

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.

$$\frac{}{S <: S} \tag{9.30}$$

$$\frac{S <: U \qquad U <: T}{S <: T} \tag{9.31}$$

$$\frac{}{S <: \text{Top}} \tag{9.32}$$

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \tag{9.33}$$

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

$$\begin{array}{lcll} t & ::= & \ldots & \text{izrazi} \\ & & \{l_i = t_i{}^{i \in 1..n}\} & \text{zapis} \\ v & ::= & \ldots & \text{vrednosti} \\ & & \{l_i = v_i{}^{i \in 1..n}\} & \text{vrednost zapisa} \\ T & ::= & \ldots & \text{tipi} \\ & & \{l_i = T_i{}^{i \in 1..n}\} & \text{tip zapisa} \end{array} \tag{9.34}$$

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

$$\frac{\forall i \in [1..n] : \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i{}^{i \in 1..n}\} : \{l_i = T_i{}^{i \in 1..n}\}} \tag{9.35}$$

$$\frac{\Gamma \vdash t_1 : \{l_i = T_i{}^{i \in 1..n}\}}{\Gamma \vdash t_1.l_j : T_j} \tag{9.36}$$

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

$$\frac{}{\{l_i = T_i{}^{i \in 1..n+k}\} <: \{l_i = T_i{}^{i \in 1..n}\}} \tag{9.37}$$

$$\frac{\forall i \in [1..n] : S_i <: T_i}{\{l_i = S_i{}^{i \in 1..n}\} <: \{l_i = T_i{}^{i \in 1..n}\}} \tag{9.38}$$

$$\frac{\{l_i = S_i{}^{i\in 1..n}\} \text{ je permutacija } \{l_i = T_i{}^{i\in 1..n}\}}{\{l_i = S_i{}^{i\in 1..n}\} <: \{l_i = T_i{}^{i\in 1..n}\}} \tag{9.39}$$

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

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

$$\frac{t_1 \rightarrow t_1'}{t_1\ t_2 \rightarrow t_1'\ t_2} \tag{9.40}$$

$$\frac{t_2 \rightarrow t_2'}{v_1\ t_2 \rightarrow v_1\ t_2'} \tag{9.41}$$

$$\frac{}{(\lambda x : T_{11}.t_{12})\ v_2 \rightarrow [x/v_2]t_{12}} \tag{9.42}$$

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.

$$\frac{}{\{l_i = S_i{}^{i\in 1..n}\}.t_j \rightarrow v_j} \tag{9.43}$$

$$\frac{t_1 \rightarrow t_1'}{t_1.l \rightarrow t_1'.l} \tag{9.44}$$

$$\frac{t_j \rightarrow t_j'}{\{l_i = v_i{}^{i\in 1..j-1}, l_j = t_j, l_i = v_i{}^{i\in j+1..n}\} \rightarrow \{l_i = v_i{}^{i\in 1..j-1}, l_j = t_j', l_i = v_i{}^{i\in j+1..n}\}} \tag{9.45}$$

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).

Poglavje 10

REKURZIVNI TIPI

Pogledali smo si že kako definirati sezname z gradnikom List(T). Seznami so samo en primer večjega razreda *rekurzivnih tipov*, ki si jih bomo ogledali v tem poglavju.

Poglejmo si zdaj spet seznam celih števil. Izraz, ki pove, da je seznam bodisi nil ali nov seznam je sledeč.

$$\text{NatList} = \langle nil : \text{unit}, cons : \{...\}\rangle; \tag{10.1}$$

Prva komponenta primera določenega z oznako $cons$ je naravno število tipa nat in druga komponenta je spet seznam celih števil NatList, ki ga definiramo.

$$\text{NatList} = \langle nil : \text{unit}, cons : \{\text{nat}, \text{NatList}\}\rangle; \tag{10.2}$$

Ta enačba ni samo preprosta definicija—ne definiramo novega imena na osnovi poznanih. Na desni strani definicije se namreč pojavlja ime, ki ga definiramo. Poglejmo si zapis v obliki neskončnega drevesa.

$$\begin{array}{l}
\text{NatList} = \langle nil : \text{unit}, cons : \text{NatListRec}\rangle; \\
\text{NatListRec} = \{\text{nat}, \text{NatList}_1\}; \\
\text{NatList}_1 = \langle nil : \text{unit}, cons : \text{NatListRec}_1\rangle; \\
\text{NatListRec}_1 = \{\text{nat}, \text{NatList}_2\}; \\
\text{NatList}_2 = \langle nil : \text{unit}, cons : \text{NatListRec}_2\rangle; \\
\text{NatListRec}_2 = \{\text{nat}, \text{NatList}_3\}; \\
\ldots
\end{array} \tag{10.3}$$

Rekurzivna enačba je podobna kot pri definiciji $factorial$. Kot v primeru $factorial$ bomo tudi tukaj izpostavili iteracijo nad celotenim izrazom. Uporabili bomo operator μ nad tipi.

$$\text{NatList} = \mu X.\langle nil : \text{unit}, cons : \{\text{nat}, X\}\rangle; \tag{10.4}$$

Definicijo lahko preberemo: “Naj bo NatList rekurziven tip, ki zadošča enačbi $\mu X.\langle nil : \text{unit}, cons : \{\text{nat}, X\}\rangle;$”.

Poznamo dva načina za definicijo rekurzivnih tipov: equi-rekurzivno in iso-rekurzivno. Načina se razlikujeta po količini informacij o tipih izrazov, ki jih je potrebno specificirati sistemu za delo s tipi.

10.1 Primeri

Končajmo najprej primer, ki smo ga začeli s seznamom celih števil. Za programiranje s celimi števili rabimo konstanto nil in konstruktor cons za dodajanje elementov v glavo seznama, operacijo isnil in funkciji tl in hd, ki za dan seznam vrneta rep ali glavo seznama.

Definicija nil sledi direktno iz definicije NatList.

$$\begin{array}{ll}
 & \text{nil} = \langle \text{nil} = \text{unit}\rangle \; as \; \text{NatList} \\
\blacktriangleright & \text{nil} : \text{NatList} \\
 & \text{cons} = \lambda n : \text{nat}.\lambda l : \text{NatList}.\langle \text{cons} = \{n, l\}\rangle \; as \; \text{NatList} \\
\blacktriangleright & \text{cons} : \text{nat} \to \text{NatList} \to \text{NatList}
\end{array} \tag{10.5}$$

Poglejmo zdaj še preostale operacije, ki testirajo strukturo parametra in izberejo željen del seznama.

$$\begin{array}{l} \text{isnil} = \lambda l : \text{NatList.case } l \text{ of } \langle \text{nil} = u \rangle \ \rightarrow \text{true} \mid \langle \text{cons} = p \rangle \ \rightarrow \text{false} \\ \blacktriangleright \ \text{isnil} : \text{NatList} \rightarrow \text{nat} \\ \text{hd} = \lambda l : \text{NatList.case } l \text{ of } \langle \text{nil} = u \rangle \ \rightarrow 0 \mid \langle \text{cons} = p \rangle \ \rightarrow p.1 \\ \blacktriangleright \ \text{hd} : \text{NatList} \rightarrow \text{nat} \\ \text{tl} = \lambda l : \text{NatList.case } l \text{ of } \langle \text{nil} = u \rangle \ \rightarrow l \mid \langle \text{cons} = p \rangle \ \rightarrow p.2 \\ \blacktriangleright \ \text{tl} : \text{NatList} \rightarrow \text{NatList} \end{array} \tag{10.6}$$

Odločili smo se, da bo glava praznega seznama 0 in da bo rep praznega seznama prazen seznam.

Poglejmo si zdaj primer definicije rekurzivne funkcije, ki sešteje elemente seznama.

$$\begin{array}{l} sumlist = \text{fix}(\lambda s : \text{NatList} \rightarrow \text{nat}.\lambda l : \text{NatList}. \\ \qquad\qquad \text{if isnil } l \text{ then } 0 \text{ else plus } (\text{hd } l)\ (s(\text{tl } l))) \\ \blacktriangleright \ sumlist : \text{NatList} \rightarrow \text{nat} \\ mylist = \text{cons } 2\ (\text{cons } 3\ (\text{cons } 5\ \text{nil})); \\ sumlist\ mylist; \\ \blacktriangleright \ 10 : \text{nat} \end{array} \tag{10.7}$$

Zapis $\text{fix}(\lambda f : T.\langle \text{telo} \rangle)$ je ekvivalenten fix $f : T$ is $\langle \text{telo} \rangle$, ki je bi definiran v Poglavju 8.

10.1.1 Lačne funkcije

Drug primer uporabe rekurzivnih funkcij je malce bolj zapleten. Lačne funkcije sprejmejo poljubno število celoštevilskih argumentov in vrnejo novo funkcijo, ki še vedno lačna.

$$Hungry = \mu A.\text{nat} \rightarrow A \tag{10.8}$$

Element tega tipa lahko definiramo z uporabo operatorja fix.

$$\begin{array}{ll} & f = \text{fix}(\lambda f : \text{nat} \to Hungry.\lambda n : \text{nat}.f) \\ \blacktriangleright & f : Hungry \\ & f\ 0\ 1\ 2\ 3\ 4\ 5; \\ \blacktriangleright & \langle fun \rangle : Hungry \end{array} \tag{10.9}$$

10.1.2 Tokovi

Drugi primer uporabe rekurzivnih tipov je tip $Stream$, ki definira niz funkcij. Lete sprejmejo poljubno število vrednosti unit in vedno vrnejo par sestavljen iz celega števila in nov tok.

$$Stream = \mu A.\text{Unit} \to \{\text{nat}, A\} \tag{10.10}$$

Najprej bomo definirali dve osnovni funkciji nad nizi. Najprej hd s vrne prvo število v toku, ko mu podamo vrednost unit. Druga funkcija tl s vrne tok, ki ga dobimo, če podamo unit toku s.

$$\begin{array}{ll} & \text{hd} = \lambda s : Stream.(s\ \text{unit}).1 \\ \blacktriangleright & \text{hd} : Stream \to \text{nat} \\ & \text{tl} = \lambda s : Stream.(s\ \text{unit}).2 \\ \blacktriangleright & \text{tl} : Stream \to Stream \end{array} \tag{10.11}$$

Tok lahko konstruiramo z naslednjo funkcijo.

$$\begin{array}{ll} & upfrom0 = \text{fix}(\lambda f : \text{Unit} \to Stream.\lambda n : \text{nat}.\lambda\_ : \text{Unit}.\{n, f(\text{succ}\ n)\})\ 0 \\ \blacktriangleright & upFrom0 : Stream \\ & \text{hd}\ upFrom0; \\ \blacktriangleright & 0 : \text{nat} \\ & \text{hd}\ (\text{tl}\ (\text{tl}\ upFrom0))); \\ \blacktriangleright & 3 : \text{nat} \end{array} \tag{10.12}$$

Tokove lahko posplošimo v enostavno obliko *procesa*. Definirali ga bomo kot funkcijo, ki sprejme število in vrne število in novi proces.

$$Proces = \mu A.\text{nat} \to \{\text{nat}, A\} \tag{10.13}$$

Poglejmo si primer procesa, ki na vsakem koraku vrne vsoto vseh sprejetih števil.

$$\begin{array}{l} p = \text{fix}(\lambda f : \text{nat} \to Process.\lambda acc : \text{nat}.\lambda n : \text{nat}. \\ \qquad \text{let } newacc = \text{plus } acc\ n \text{ in} \\ \qquad \{newacc, f\ newacc\})\ 0; \\ \blacktriangleright\ p : Process \end{array} \tag{10.14}$$

Definirajmo še nekaj običajnih funkcij za delo s procesi.

$$\begin{array}{l} curr = \lambda s : Process.(s\ 0).1; \\ \blacktriangleright\ curr : Process \to \text{nat} \\ send = \lambda n : \text{nat}.\lambda s : Process.(s\ n).2; \\ \blacktriangleright\ send : \text{nat} \to Process \to Process \end{array} \tag{10.15}$$

Pošljemo procesu p števila 5, 3 in 20 ter dobimo kot rezultat zadnje interakcije s procesom število 28.

$$\begin{array}{l} curr\ (send\ 20\ (send\ 3\ (send\ 5\ p))); \\ \blacktriangleright\ 28 : \text{nat} \end{array} \tag{10.16}$$

10.2 Iso-rekurzivni in equi-rekurzivni tipi

Ločimo med dvemi osnovnimi pristopi k prestavitvi rekurzivnih tipov: equi-rekurzivni in iso-rekurzivni tipi.

Ključna razlika med pristopama se izraža v odgovoru na enostavno vporašanje: kakšno je razmerje med tipom $\mu X.T$ in njegovim razvitjem v enem koraku?

Na primer, kakšno je razmerje med NatList in $\langle \text{nil} : \text{Unit}, \text{cons} : \{\text{nat}, \text{NatList}\}\rangle$?

Equi-rekurziven pristop obravnava dva zapisa rekurzivnega tipa kot enakost po definiciji–tipa, ki sta zamenljiva v vsakem kontekstu.

Razčlenjevalnik zamenja tipe na vseh mestih, kjer se pričakuje razvitje tipa ali njegova zložena verzija. V obeh primerih rekurzivni tip določa pravzaprav neskončno drevo.

Iso-rekurziven pristop obravnava rekurzivni tip in njegovo razvitje kot različna vendar izomorfna tipa.

Definiramo operaciji fold in unfold, ki sta del jezika. Operacija unfold nad rekurzivnim tipom $\mu X.T$ razvije vse pojavitve tipa X v njegovi definiciji z $\mu X.T$ t.j. lastno definicijo $[X/\mu X.T]\ T$.

Primer 10.2.1 *Tip NatList je definiran z*

$$\mu X.\langle \textit{nil} : \textit{Unit}, \textit{cons} : \{\textit{nat}, X\}\rangle.$$

Razvitje tipa NatList z operacijo unfold da tip

$$\langle \textit{nil} : \textit{Unit}, \textit{cons} : \{\textit{nat}, \mu X.\langle \textit{nil} : \textit{Unit}, \textit{cons} : \{\textit{nat}, X\}\rangle\}\rangle.$$

□

Iso-rekurzivni pristop torej zahteva definicijo operacij fold in unfold s katerima preslikujemo rekurzivni tip v njegovo razvitje in iz razvitega rekurzivnega tipa nazaj v "zloženo" verzijo.

$$\begin{array}{l}\text{unfold}[\mu X.T] : \mu X.T \to [X/\mu X.T]\ T \\ \text{fold}[\mu X.T] : [X/\mu X.T]\ T \to \mu X.T\end{array}$$

Operaciji fold in unfold sta definirani znotraj jezika, ki uporablja rekurzivne tipe. Operaciji fold in unfold imata naslednjo sintakso.

$$
\begin{array}{lll}
t & ::= & \text{izrazi} \\
 & \text{fold}[T]\ t & \text{zloži} \\
 & \text{unfold}[T]\ t & \text{razvitje} \\
v & ::= & \text{vrednosti} \\
 & \text{fold}[T]\ v & \text{zloži} \\
T & ::= & \text{tipi} \\
 & X & \text{spremenljivka tipa} \\
 & \mu X.T & \text{rekurzivni tip}
\end{array}
\tag{10.17}
$$

Evaluacija operacij fold in unfold je predstavljena z naslednjimi pravili.

Izomorfizem operacij je razviden iz pravila 10.18, ki izniči sekvenco $\text{unfold}(\text{fold}(\ldots))$. Kot vidimo tipi v tem pravilu niso pomembni, ker bi sicer bilo potrebno preverjati tipa unfold in fold v času izvajanja.

$$
\frac{}{\text{unfold}[S](\text{fold}[T]\ v_1) \to v_1}
\tag{10.18}
$$

$$
\frac{t_1 \to t_1'}{\text{fold}[T]\ t_1 \to \text{fold}[T]\ t_1'}
\tag{10.19}
$$

$$
\frac{t_1 \to t_1'}{\text{unfold}[T]\ t_1 \to \text{unfold}[T]\ t_1'}
\tag{10.20}
$$

Prirejanje tipov izrazom, ki vsebujejo operaciji fold in unfold je definirano z naslednjimi pravili.

$$
\frac{U = \mu X.T_1 \quad \Gamma \vdash t_1 : [X/U]T_1}{\Gamma \vdash \text{fold}[U]t_1 : U}
\tag{10.21}
$$

$$
\frac{U = \mu X.T_1 \quad \Gamma \vdash t_1 : U}{\Gamma \vdash \text{unfold}[U]t_1 : [X/U]T_1}
\tag{10.22}
$$

V praksi se uporabljata oba pristopa. Equi-rekurziven pristop je bolj intuitiven vendar zahteva od razčlenjevalnika, da ugotovi mesta kjer je potrebno uporabiti operaciji fold in unfold.

Povezovanje equi-rekurzivnih tipov z drugimi jezikovnimi gradniki se je izkazalo kot kompleksno.

Notacija za iso-rekuzivne tipe je težja, ker je potrebno uporabljati operaciji fold in unfold v jeziku.

Konkretni programski jeziki velikokrat združijo razvijanje oz. zlaganje tipa med ostalo anotacijo tipov.

Na primer, v ML družini jezikov je običajno, da vsaka definicija datatype implicitno definiran rekurzivni tip. Vsaka uporaba konstruktorjev implicitno uporabi operacijo fold in vsaka uporaba ujemanja prepostavlja uporabo unfold.

Podobno tudi Java pri vsaki definiciji razreda implicitno uporabi rekurzivni tip. Klic metode implicitno povzroči tudi operacijo unfold.

Primer 10.2.2

10.3 Indukcija in ko-indukcija

Naj bo $\mathcal{U}$ univerzalna množica, ki bo uporabljena kot domena. Elemente $\mathcal{U}$ si lahko predstavljamo kot izraze nekega jezika.

Definicija 10.3.1 (Monotona funkcija) *Funkcija $F \in \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$ je monotona, če za vse $X, Y \in \mathcal{P}(\mathcal{U})$ velja $X \subseteq Y \Rightarrow F(X) \subseteq F(Y)$.* □

V nadaljevanju bomo predpostavljali, da je F monotona. Imenovali jo bomo *generativna funcija*.

Funkcijo F si lahko predstavljamo kot izpeljavo stavkov, evaluacijo ali računanje tipov izrazov iz $\mathcal{U}$. Za dano množico stavkov X jezika z uporabo funkcije F poiščemo vse tiste stavke, ki so izpeljani iz premis v X s pravili, ki jih določa F.

Definicija 10.3.2 *Naj bo* $X \subseteq \mathcal{U}$.

1. X *je* F-zaprta *če* $F(X) \subseteq X$
2. X *je* F-konsistenta *če* $X \subseteq F(X)$
3. X *je* fiksna točka F *če* $F(X) = X$ □

F-zaprta množica je tista, ki je ne moremo več razširiti z uporabo F—vsebuje že vse posledice premis iz X glede na F.

F-konsistentna množica je tista, katere stavki so "izraženi sami s sabo"—vse premise X so vsebovane v množici $F(X)$, ki je izpeljana iz X.

In na koncu še poglejmo fiksno točko F, ki je oboje zaprta in konsistentna—vsebuje vse premise, ki jih potrebujejo elementi množice, ter posledice, ki sledijo iz elementov in nič več.

Primer 10.3.1 *Dana je naslednja generativna funkcija* E_1 *nad množico* $\mathcal{U} = \{a, b, c\}$.

$$\begin{array}{rclrcl} E_1(\emptyset) & = & \{c\} & E_1(\{a,b\}) & = & \{c\} \\ E_1(\{a\}) & = & \{c\} & E_1(\{a,c\}) & = & \{b,c\} \\ E_1(\{b\}) & = & \{c\} & E_1(\{b,c\}) & = & \{a,b,c\} \\ E_1(\{c\}) & = & \{b,c\} & E_1(\{a,b,c\}) & = & \{a,b,c\} \end{array}$$

Obstaja samo ena E_1*-zaprta množica* $\{a, b, c\}$ *in štiri* E_1*-konsistentne množice:* $\emptyset, \{c\}, \{b, c\}$ *in* $\{a, b, c\}$.

E_1 *lahko zapišemo v krajši obliki z uporabo naslednjih pravil.*

$$\frac{}{c} \qquad \frac{c}{b} \qquad \frac{b \quad c}{a}$$

□

Naslednji izrek definira najmanjšo in največjo fiksno točko funkcije F z uporabo pravkar definiranih konceptov F-zaprtja in F-konsistentnosti.

Izrek 10.3.1 (Knaster-Tarski, 1955) *:*

1. *Presek vseh F-zaprtih množic je* najmanjša fiksna točka F.
2. *Unija vseh F-konsistentnih množic je* največja fiksna točka F. □

Dokaz. ... □

Definicija 10.3.3 *Najmanjšo fiksno točko F zapišemo μF. Največjo fiksno točko F zapišemo νF.*

Primer 10.3.2 *Funkcija E_1 predstavljena v primeru zgoraj ima $\mu F = \nu F = \{a, b, c\}$.*

Primer 10.3.3 *Naj bo dana generativna funkcija E_2 na univerzumu $\mathcal{U} = \{a, b, c\}$ definirana z naslednjimi pravili.*

$$\frac{}{a} \qquad \frac{c}{b} \qquad \frac{a \quad b}{c}$$

Funkcijo lahko spet zapišemo v daljši obliki tako, da eksplicitno definiramo preslikavo.

$$\begin{array}{rclrcl} E_2(\emptyset) & = & \{a\} & E_2(\{a,b\}) & = & \{a,c\} \\ E_2(\{a\}) & = & \{a\} & E_2(\{a,c\}) & = & \{a,b\} \\ E_2(\{b\}) & = & \{a\} & E_2(\{b,c\}) & = & \{a,b\} \\ E_2(\{c\}) & = & \{a,b\} & E_2(\{a,b,c\}) & = & \{a,b,c\} \end{array}$$

E_2-zaprte množice so $\{a\}$ in $\{a, b, c\}$ ter E_2-konsistentne množice: $\emptyset, \{a\}$ in $\{a, b, c\}$. □

Opazka 10.3.1 *Množica μF je F-zaprta in je torej najmanjša F-zaprta množica.*

Podobno je množica νF F-konsistentna in je torej največja F-konsistentna množica.

Ta dva razmisleka vodita do para konceptualnih orodij za sklepanje. □

Posledica 10.3.1 (Izreka 10.3.1) *:*

1. *Princip indukcije: X je F-zaprta $\longrightarrow \mu F \subseteq X$.*
2. *Princip ko-indukcije: X je F-konsistentna $\longrightarrow X \subseteq \nu F$.*

Poglejmo si osnovne ideje definiranih principov. Množico X si predstavljamo kot lastnost elementov oz. *predikat* definiran nad elementi univerzalne množice $\mathcal{U}$.

X je karakteristična množica predikata predstavljenega z X—podmnožica $\mathcal{U}$ za katero velja predikat X.

Da bi pokazali da velja predikat X za element $x \in \mathcal{U}$ je zadosti, da pokažemo $x \in X$.

Princip indukcije pravi, da vsaka lastnost katere karakteristična množica X je zaprta za relacijo F, velja za vse elemente induktivno definirane množice μF.

Princip ko-indukcije pravi, da lahko ugotovimo ali je element x v induktivno definirani množici νF, tako da poiščemo karakteristično množico X za katero velja $x \in X$, ki je F-konsistentna.

10.4 Končni in neskončni tipi

10.4.1 Podtipi

Definicija 10.4.1 (Končni podtipi) *Končna tipa T in S sta v relaciji* podtip*, če $(S,T) \in \mu S_f$, kjer je monotona funkcija $S_f \in \mathcal{P}(\mathcal{T}_f \times \mathcal{T}_f) \to \mathcal{P}(\mathcal{T}_f \times \mathcal{T}_f)$ definirana kot:*

$$\begin{array}{rcl} S_f(R) & = & \{(T, Top) \mid T \in \mathcal{T}_f\} \\ & \cup & \{(S_1 \times S_2, T_1 \times T_2) \mid (S_1,T_1),(S_2,T_2) \in R\} \\ & \cup & \{(S_1 \to S_2, T_1 \to T_2) \mid (S_1,T_1),(S_2,T_2) \in R\} \end{array}$$

Definicija 10.4.2 (Neskončni podtipi) *Končna ali neskončna drevesna tipa T in S sta v relaciji* podtip*, če $(S,T) \in \nu S_f$, kjer je monotona funkcija $S_i \in \mathcal{P}(\mathcal{T}_i \times \mathcal{T}_i) \to \mathcal{P}(\mathcal{T}_i \times \mathcal{T}_i)$ definirana kot:*

$$\begin{array}{rcl} S_i(R) & = & \{(T, Top) \mid T \in \mathcal{T}_i\} \\ & \cup & \{(S_1 \times S_2, T_1 \times T_2) \mid (S_1,T_1),(S_2,T_2) \in R\} \\ & \cup & \{(S_1 \to S_2, T_1 \to T_2) \mid (S_1,T_1),(S_2,T_2) \in R\} \end{array}$$

Vaja 10.4.1

Definicija 10.4.3 (Tranzitivnost) *Relacija $R \subseteq \mathcal{U} \times \mathcal{U}$ je tranzitivna, če je R zaprta za monotono funkcijo $TR(R) = \{(x,y) \mid \exists z \in .(x,z),(z,y) \in R\}$: $TR(R) \subseteq R$.*

Lema 10.4.1 *Naj bo $F \in \mathcal{P}(\mathcal{U}\times\mathcal{U}) \to \mathcal{P}(\mathcal{U}\times\mathcal{U})$ monotona funkcija. Če je $TR(T(R)) \subseteq F(TR(R))$ za vsako $R \subseteq \mathcal{U} \times \mathcal{U}$, potem je νF tranzitivna.*

Izrek 10.4.1 *νS je tranzitivna.*

10.5 Preverjanje članstva

Naj bo F konstrukcijska funkcija nad univerzumom $\mathcal{U}$ in naj bo $x \in \mathcal{U}$. Ogledali si bomo algoritme s katerimi lahko pogledamo ali velja $x \in \nu F$ oz. $x \in \mu F$.

...

Definicija 10.5.1 *Na bo F obrnljiva konstrukcijska funkcija in naj bo gfp$_F$ boolova funkcija definirana z naslednjim postopkom.*

$$\begin{array}{rcl} \textit{gfp}(X) & = & \textit{if } support(x) \uparrow \textit{ then false} \\ & & \textit{else if } support(X) \subseteq X \textit{ then true} \\ & & \textit{else gfp}(support(X) \cup X). \end{array}$$

...

Izrek 10.5.1

- *Če gfp$_F(X) = true$, potem $X \subseteq \nu F$.*
- *Če gfp$_F(X) = false$, potem $X \not\subseteq \nu F$.*

...

10.6 μ-Tipi

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 jezikih 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 $\rightarrow$ Nat in X z Nat dobimo izraz

$$\lambda f : \mathit{Nat} \rightarrow \mathit{Nat}.\lambda a : \mathit{Nat}.f(f\ a)$$

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

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

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.

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

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 pari:*

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

□

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.

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

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

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

Primer 11.3.1 *Poglejmo si primer polimorfične funkcije idetitete.*

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

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 za klasifikacijo polimorfičnih funkcij kot so prej predstavljena funkcija id.

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

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

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.

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

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

Kompletna sintaksa Sistema F je definirana z naslednjo slovnico.

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

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

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

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

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

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

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

Naslednja pravila definira prirejanje tipov izrazov Sistema F.

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

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

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

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

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

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*.*

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

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

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

□

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

$$\begin{array}{l} \quad double = \lambda X.\lambda f\colon X \to X.\lambda a\colon X.f(fa) \\ \blacktriangleright \quad double\colon \forall X.(X \to X) \to X \to X \end{array} \tag{11.17}$$

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

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

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

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

□

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.*

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

□

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.*

$$
\begin{array}{l}
\blacktriangleright\ \mathit{nil} : \forall X.\mathit{List}\ X \\
\quad \mathit{cons} : \forall X.X \to \mathit{List}\ X \to \mathit{List}\ X \\
\quad \mathit{isnil} : \forall X.\mathit{List}\ X \to \mathit{Bool} \\
\quad \mathit{head} : \forall X.\mathit{List}\ X \to X \\
\quad \mathit{tail} : \forall X.\mathit{List}\ X \to \mathit{List}\ X
\end{array}
\tag{11.21}
$$

$$
\begin{array}{l}
map = \lambda X.\lambda Y. \\
\qquad \lambda f : X \to Y. \\
\qquad (\mathit{fix}\ (\lambda m : (\mathit{List}\ X) \to (\mathit{List}\ Y). \\
\qquad\quad \lambda l : \mathit{List}\ X. \\
\qquad\quad \mathit{if}\ \mathit{isnil}\ [X]\ l \\
\qquad\quad \mathit{then}\ \mathit{nil}\ [Y] \\
\qquad\quad \mathit{else}\ \mathit{cons}\ [Y]\ (f\ \mathit{head}\ [X]\ l)) \\
\qquad\qquad\qquad\qquad (m\ \mathit{tail}\ [X]\ l)))); \\
\blacktriangleright\ map : \forall X.\forall Y.(X \to Y) \to \mathit{List}\ X \to \mathit{List}\ Y \\
l = \mathit{cons}\ [\mathit{Nat}]\ 4\ (\mathit{cons}\ [\mathit{Nat}]\ 3\ (\mathit{cons}\ [\mathit{Nat}]\ 2\ (\mathit{nil}\ [\mathit{Nat}]))); \\
\blacktriangleright\ l : \mathit{List}\ \mathit{Nat} \\
\mathit{head}\ [\mathit{Nat}]\ (map\ [\mathit{Nat}]\ [\mathit{Nat}]\ (\lambda x : \mathit{Nat}, \mathit{succ}\ x)\ l); \\
\blacktriangleright\ 5 : \mathit{Nat}
\end{array}
\tag{11.22}
$$

□

11.3.3 Osnovne lastnosti jezika F

11.4 Eksistenčni tipi

11.4.1 Osnovne definicije

11.4.2 Podatkovne abstrakcije z eksistenčnimi tipi

Poglavje 12

*JEZIKI VIŠJEGA REDA

12.0.3 Operacije tipov in vrste

12.0.4 Sistem F_ω

Literatura

[1] Barendregt, H. P. The Lambda Calculus: Its Syntax and Semantics. North Holland, Amsterdam, 1984.

[2] Henk Barendregt, Erik Barendsen, Introduction to Lambda Calculus, March 2000.

[3] Jean-Yves Girard, Proofs and Types, Paul Taylor, Yves Lafont (translators), Cambidge University Press, 1990.

[4] Robert Harper, Programming in Standard ML, Draft, 2005.

[5] Robert Harper, Practical Foundations for Programming Languages, Draft, 2010.

[6] R. Harper, F. Honsell, and G. Plotkin, A framework for defining logics, Journal of the ACM, 40(1):143-184, 1993.

[7] Per Martin Löf, On the Meanings of the Logical Constants and the Justifications of the Logical Laws, 1995.

[8] J.C. Mitchell: Concepts in programming languages, Cambridge University Press, 2003.

[9] Frank Pfenning, Computation and Deduction,Carnegie Mellon University, 2001 (1-4).

[10] Benjamin Pierce, Types and Programming languages, MIT Press, 2002.

[11] Gordon Plotkin, LCF Considered as A Programming Language, North-Holland Publishing, TCS 5 (1977), pp.223-255.

[12] Robert D. Tennet, The Denotational Semantics of Programming Languages, CACM, vol. 19, no. 8, 1976.

[13] Glynn Winskel, The Formal Semantics of Programming Languages: An Introduction, MIT Press, 1993 (1-8).

[14] Glynn Winskel, Set Theory for Computer Science, CS Lecture Notes, Cambridge University, 2009.

[15] Glynn Winskel, Denotational Semantics, CS Lecture Notes, Cambridge University, 2005.