DIAGRAMI INVARIANT ZA TURINGOVE STROJE

JANEZ IGNACIJ JEREB

Fakulteta za matematiko in fiziko
Univerza v Ljubljani

Diagrami invariant so bili uvedeni z namenom lažje zasnove dokazljivo pravilnih programov. V tem članku bodo uporabljeni za zasnovo in dokazovanje pravilnosti Turingovih strojev. Najprej bodo definirani splošni diagrami invariant ter njihova semantika in programska logika. Ti bodo uporabljeni za sestavo diagramov invariant za Turingove stroje. Za lažje dokazovanje bo uvedeno še nekaj notacije in pravil za sklepanje z njimi. Na koncu bodo uporabljeni na primeru.

INVARIANT DIAGRAMS FOR TURING MACHINES

Invariant diagrams were created to facilitate the design of provably correct programs. In this paper, they will be used to design and prove the correctness of Turing machines. Firstly, general invariant diagrams, their semantics, and their program logic will be defined. They will be used to construct invariant diagrams for Turing machines. To facilitate the proofs, some notation and rules will be introduced for reasoning with them. Finally, their application will be demonstrated on an example.

1. Uvod

Dokazovanje pravilnosti programov z diagrami je obstajalo že od vsega začetka računalništva. Alan Turing je že leta 1949 pisal o dokazovanju pravilnosti programov [1], pri čemer si je pomagal z diagrami. Pozneje je bilo, večinoma neodvisno od Turinga, razvitih več metod dokazovanja pravilnosti. Ena od prvih takih metod je bila Floydova metoda [2], ki je bila namenjena dokazovanju pravilnosti diagramov poteka (angl. flowcharts). Ta diagramska metoda je bila uporabljena, ker je bilo tako najlažje prikazati ukaze *GOTO* (skoke). Vendar se je zaradi nasprotovanja GOTOjem [3] uveljavilo strukturirano programiranje, ki je ukaze GOTO nadomestilo z zankami in pogojnimi stavki. Kot glavni orodji za sklepanje o pravilnosti takih programov sta se uveljavili Hoareova logika [4] in Dijkstrovi najšibkejši predpogoji [5]. Ti sistemi sklepanja niso diagramski, ampak simbolični. Pozneje je Reynolds razvil diagramsko metodo [6] za zasnovo programov, pri katerih so bili GOTOji potrebni. Back je te diagrame razširil z gnezdenjem [7], jim dal formalno semantiko [8] in tudi razvil programsko okolje za delo z njimi [9]. Predstavlja jih kot *invariantno usmerjeno programiranje*, kjer programer razmišlja o *situacijah* programa in prehodih med njimi. Tak način razmišljanja omogoča zasnovo dokazano pravilnih programov, prav tako pa je metoda primerna kot učni pripomoček [10].

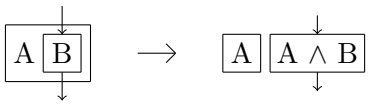
V tem delu se bomo posvetili dokazovanju pravilnosti Turingovih strojev. Ti imajo redko podrobne dokaze pravilnosti, saj nizkonivojska narava Turingovih strojev vodi do velikega števila tehničnih podrobnosti, ki jih je treba obravnavati v dokazu [11]. Diagramov invariant zanje ni težko zasnovati, saj so Turingovi stroji velikokrat predstavljeni z diagrami. Z dodajanjem gnezdenja, invariant in fantomskih spremenljivk dobimo dovolj močno in ne preveč zahtevno orodje za dokazovanje njihove pravilnosti.

2. Diagrami invariant

Diagrami invariant predstavljajo program v obliki usmerjenega grafa, kjer so vozlišča situacije sistema, povezave pa operacije, ki ob določenih pogojih spremenijo stanje spremenljivk in se lahko

premaknejo v novo situacijo. Izvajanje diagrama invariant se začne v začetni situaciji in sledi aktivnim povezavam. Povezave so lahko aktivne ali neaktivne glede na trenutno stanje programa. Če ni aktivnih povezav, se izvajanje ustavi. Tako si lahko predstavljamo izvajanje programa kot sprehod po grafu, kjer nam povezave povedo, kam lahko gremo in kako se spremeni stanje spremenljivk. Vsako situacijo opišemo z *invarianto*, ki je logični izraz, ki pove, kaj velja za stanje spremenljivk v določeni situaciji. Operacije pa napišemo na povezave grafa in so zapisane kot ukazi v nekem imperativnem programskem jeziku. Seveda je treba dokazati, da izvajanje operacij ohranja invariante. Če nam to uspe, potem smo dokazali *delno pravilnost* programa. Če bi dokazali še ustavitev, bi s tem dokazali še *popolno pravilnost* programa. V tem delu bomo dokazovali le delno pravilnost programov.

Za krajši in preglednejši zapis invariant stanj bomo razširili diagrame invariant z gnezdenimi situacijami. Te narišemo tako, da vozlišče označimo z okvirčkom, znotraj katerega lahko narišemo nova vozlišča in jih povezujemo med seboj, z zaobjemajočo situacijo in zunanjimi vozlišči. Lahko si predstavljamo, da okvirji označujejo množico stanj in notranja vozlišča predstavljajo podmnožice stanj. Pri opisovanju stanj nam to prihrani stalno ponavljanje delov opisov. Ne doda pa izrazne moči, saj lahko zunanji diagram jemljemo kot novo vozlišče z označenim opisom, vsem podvozliščem pa pripišemo isti opis kot zunanjemu vozlišču (slika 1).



Slika 1. Pretvorba gnezdenega diagrama v diagram brez gnezdenja.

V nadaljevanju bomo formalizirali diagrame invariant, njihovo semantiko in Hoareovo logiko za dokazovanje pravilnosti. Začnimo z definicijo ukazov, ki jih bomo uporabljali na povezavah diagrama invariant.

Definicija 1 (Ukazi)**.** Gramatika ukazov, kjer je b logični izraz in R relacija med dvema stanjema.

Ukaz	S, T	$::=$	$\{b\}$	trditev
		$\mid$	$S \sqcap T$	demonska izbira
		$\mid$	$[b]$	predpostavka
		$\mid$	$[R]$	demonska posodobitev
		$\mid$	$S; T$	zaporedno izvajanje

Demonska izbira $S \sqcap T$ *demonsko* nedeterministično izvede enega od ukazev S ali T. To pomeni, da se pri dokazovanju predpostavlja, da program izbere „najbolj nezaželeno izbiro“. Na primeru: ne moremo dokazati, da ukaz „zmanjšaj v“ $\sqcap$ „povečaj v“ zmanjša spremenljivko v. To bi lahko dokazali, če bi imeli *angelsko* izbiro. Predpostavka $[b]$ naredi b veljaven na demonsko nedeterminističen način. Demonska posodobitev $[R]$ posodobi stanje z relacijo med stanji R. Če je več možnih stanj, se to izbere demonsko nedeterministično. Zaporedno izvajanje $S; T$ izvede ukaza S in T zaporedno. Pogosto bomo uporabili ukaz $v := e$ za spreminjanje vrednosti spremenljivk. Tega definiramo z $(v{:=}e) := [R']$, kjer R' definiramo z:

$$s \, R' \, s' \;\; :\Leftrightarrow \;\; s' = s[v := e(s)],$$

kjer je

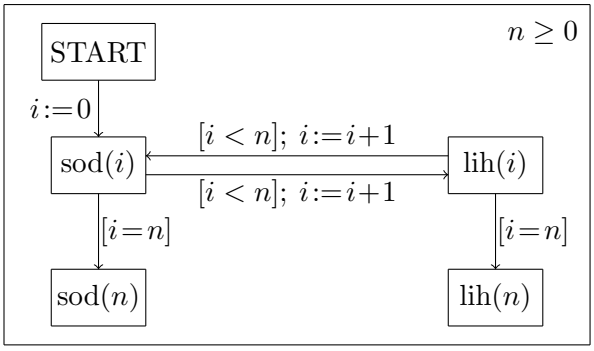
$$f[v := x] = u \mapsto \begin{cases} f(u) & \text{če } u \neq v \\ x & \text{če } u = v \end{cases}.$$

Definiramo tudi substitucijo za izraze z $e[v := x] = s \mapsto e(s[v := x])$. Formalno definiramo diagrame invariant kot usmerjene grafe z oznakami na povezavah. V definiciji ne upoštevamo gnezdenja, saj je samo priročna notacija, ki jo lahko odstranimo (slika 1).

Val je množica vrednosti, kot so števila, znaki ali resničnostne vrednosti, Var pa množica spremenljivk. S Pred bomo označili množico vseh *predikatov*. Ti so logični izrazi, kot $x = 5$, $x > y$ in drugi, in so formalno funkcije tipa $(\text{Var} \to \text{Val}) \to \{\top, \bot\}$, kjer $\top$ pomeni, da je izraz resničen, $\bot$ pa, da ni. Večkrat jih bomo obravnavali kot množice stanj. Takrat vsebovanost interpretiramo z $s \in p :\Leftrightarrow p(s) = \top$.

Definicija 2 (Diagram invariant)**.** *Diagram invariant* je usmerjen graf z oznakami na povezavah (s, V, E, P), kjer je V množica *situacij*, $E : V \times V \to$ Ukaz je funkcija prehodov in $P : V \to$ Pred invariante. S $s \in V$ označimo začetno situacijo. Slednjo v diagramu označimo z oznako `START`.

Primer 3. Primer diagrama invariant (slika 2), ki ugotovi parnost nenegativnega števila n.



Slika 2. Diagram invariant za program, ki ugotovi parnost števila n.

Pri tem diagramu uporabimo gnezdenje le zato, da si prihranimo nekaj pisanja. Nimamo nobenih prehodov na ali iz zaobjemajoče situacije. Predpostavke uporabimo, zato da pogojujemo, kdaj se izvajajo prehodi. Začnemo v situaciji `START`, kjer je $n \geq 0$. Od tam lahko izvedemo prehod $i := 0$, ki nas pripelje v situacijo sod(i). Invarianta sod(i) velja, ker velja sod(0). Potem pa prištevamo i-ju 1 in parnost beležimo s situacijama. Ko dosežemo $i = n$, vemo, da ima n enako parnost kot i, in tako zaključimo.

Sedaj smo spoznali intuitivno izvajanje diagramov invariant. V nadaljevanju bomo to formalizirali z operacijsko semantiko.

2.1 Operacijska semantika diagramov

Formalno operacijsko semantiko diagramov invariant bomo definirali z induktivno definirano relacijo *velikih korakov* za ukaze $(s, S) \downarrow s'$, ki nam pove, kako iz stanja s dobimo stanje s' z izvajanjem ukaza S. Stanja so elementi množice $(\text{Var} \to \text{Val}) \cup \{\texttt{disabled}, \texttt{fail}\}$. Stanji `disabled` in `fail` predstavljata dva dodatna izida. Z `disabled` označimo neaktivnost povezave, oziroma da predpostavka ni izpolnjena ali pa da ni možno izvesti demonske izbire. S `fail` pa označimo, da se je program sesul, oziroma da trditev ni izpolnjena. Ukaze lahko torej izvedemo ali ne, ko pa jih izvedemo, se lahko ti sesujejo. Začnimo s semantiko trditev.

$$\frac{b(s)}{(s, \{b\}) \downarrow s} \qquad \frac{\neg b(s)}{(s, \{b\}) \downarrow \texttt{fail}}$$

Tukaj lahko vidimo, da če je b resničen v stanju s, potem se trditev $\{b\}$ izvede in ohranimo stanje. Sicer pa preidemo v stanje `fail`, ki nam pove, da se program sesuje. Nadaljujmo s semantiko predpostavk.

$$\frac{b(s)}{(s,[b])\downarrow s} \qquad \frac{\neg b(s)}{(s,[b])\downarrow \texttt{disabled}}$$

Predpostavke delujejo podobno kot trditve, le da namesto `fail` dobimo `disabled`. Malo drugačna je semantika demonske posodobitve:

$$\frac{s\ R\ s'}{(s,[R])\downarrow s'} \qquad \frac{\forall s'.\ \neg(s\ R\ s')}{(s,[R])\downarrow \texttt{disabled}}$$

Če lahko izvedemo posodobitev, jo izvedemo, sicer pa preidemo v stanje `disabled`. Tu vidimo še drugačno interpretacijo demonske izbire: ne kot „najslabšo možno izbiro“, ampak kot možnost, da moramo pri dokazovanju upoštevati vse možne izbire (tudi najslabšo). Demonska izbira ima več pravil, vendar so enostavna za razumevanje.

$$\frac{(s,S)\downarrow s'}{(s,S\sqcap T)\downarrow s'} \qquad \frac{(s,T)\downarrow s'}{(s,S\sqcap T)\downarrow s'} \qquad \frac{(s,S)\downarrow \texttt{disabled} \qquad (s,T)\downarrow \texttt{disabled}}{(s,S\sqcap T)\downarrow \texttt{disabled}}$$

Demonska izbira se obnaša podobno kot demonska posodobitev. Izvede se lahko katerakoli izbira. Če se ena izmed njih sesuje, se sesuje tudi celotna izbira. Za neaktivnost pa je potrebno, da sta oba ukaza neaktivna. Nadaljujmo s semantiko zaporednega izvajanja.

$$\frac{(s,S)\downarrow s' \qquad (s',T)\downarrow s''}{(s,S;T)\downarrow s''} \qquad \frac{(s,S)\downarrow \texttt{fail}}{(s,S;T)\downarrow \texttt{fail}}$$

$$\frac{(s,S)\downarrow \texttt{disabled}}{(s,S;T)\downarrow \texttt{disabled}} \qquad \frac{(s,S)\not\downarrow \texttt{fail} \qquad \forall s'.\ ((s,S)\downarrow s' \Rightarrow (s',T)\downarrow \texttt{disabled})}{(s,S;T)\downarrow \texttt{disabled}}$$

Zaporedno izvajanje najprej izvede prvi ukaz in njegov rezultat uporabi za izvedbo drugega. Če se katerikoli izmed ukazov sesuje, se sesuje tudi njuno zaporedno izvajanje. Ukaz pa ni aktiven, če prvi ukaz ni aktiven ali pa je aktiven, vendar pa je drugi ukaz neaktiven za vse možne rezultate prvega ukaza.

Sedaj smo definirali, kako izvesti en prehod v diagramu invariant. Za izvajanje celotnega diagrama pa dodamo novo relacijo velikih korakov $(s,i,E)\Downarrow(s',j)$, ki nam pove, da se iz stanja s v situaciji i z uporabo prehodov diagrama invariant E dobimo stanje s' v situaciji j. Zanjo potrebujemo le tri pravila.

$$\frac{(s,E(i,j))\downarrow s' \qquad s'\neq \texttt{fail} \qquad (s',j,E)\Downarrow(s'',k)}{(s,i,E)\Downarrow(s'',k)}$$

Pri prvem pravilu naredimo en prehod in nato uporabimo novo situacijo in stanje za nadaljnje izvajanje.

$$\frac{(s,E(i,j))\downarrow \texttt{fail}}{(s,i,E)\Downarrow \texttt{fail}} \qquad \frac{\forall j.\ (s,E(i,j))\downarrow \texttt{disabled}}{(s,i,E)\Downarrow(s,i)}$$

Drugo pravilo pravi, da če se en prehod sesuje, se sesuje tudi izvajanje diagrama invariant. Tretje pravilo pa nam pove, da če ni aktivnih prehodov, se izvajanje ustavi in ostanemo v isti situaciji in stanju.

V članku [8] je začetna situacija nedoločena. To pomeni, da se lahko program začne v katerikoli situaciji, ki izpolnjuje začetno invarianto. Ko bomo dokazovali pravilnost Turingovih strojev, tega ne bomo uporabili in bomo označili začetno situacijo z oznako `START`.

2.2 Programska logika za diagrame invariant

V tem poglavju bomo definirali programsko logiko za diagrame invariant, ki je osnovana na Hoareovi logiki [4]. To bomo uporabili za dokazovanje pravilnosti diagramov invariant. Začeli bomo z definicijo Hoareove logike za ukaze. Ta je zelo podobna izvorni Hoareovi logiki, le da je dodan demonski nedeterminizem in predpostavke. Glavni del predstavlja relacija $p \{\!\{S\}\!\} q$, ki nam pove, da če se program začne v stanju, ki izpolnjuje predikat p (predpogoj) in izvede ukaz S, potem bo končno stanje izpolnjevalo predikat q (končni pogoj). Formalno definiramo relacijo $p \{\!\{S\}\!\} q$ z

$$p \{\!\{S\}\!\} q \;\; :\Leftrightarrow \;\; \forall s.\, p(s) \Rightarrow (\forall s'.\, (s, S) \downarrow s' \Rightarrow q(s')).$$

V izvornem članku [8] Hoareova logika vključuje tudi ustavitev. V našem članku smo se odločili, da se bomo osredotočili le na delno pravilnost. Prav tako je bilo v originalnem članku dokazano, da se logika sklada z operacijsko semantiko (skladnost) in tudi da je dovolj močna, da lahko dokažemo vse, kar je mogoče dokazati s semantiko (zadostnost). V nadaljevanju bomo $p \{\!\{S\}\!\} q$ rekli tudi Hoareova trojica ali pa specifikacija.

Začnimo s pravilom za demonsko izbiro.

$$\frac{p \{\!\{S\}\!\} q \qquad p \{\!\{T\}\!\} q}{p \{\!\{S \sqcap T\}\!\} q}$$

Če dokažemo, da obe izbiri izpolnjujeta specifikacijo, potem jo izpolnjuje tudi demonska izbira med njima. Nadaljujmo s pravilom za trditve.

$$\frac{\forall s.\, p(s) \Rightarrow r(s) \wedge q(s)}{p \{\!\{\{r\}\}\!\} q}$$

Ta nam pove, da če predpogoj p implicira trditev r in končni pogoj q, potem velja specifikacija. Predpostavke obravnavamo z naslednjima praviloma

$$\frac{\forall s.\, (p(s) \wedge r(s)) \Rightarrow q(s)}{p \{\!\{[r]\}\!\} q} \qquad \frac{\forall s, s'.\, (p(s) \wedge s\, R\, s') \Rightarrow q(s')}{p \{\!\{[R]\}\!\} q}$$

Pravili nam tukaj jasno pokažeta izvor imena operacije $[\cdot]$. Če trditev q velja ob predpostavki predpogoja p in predpostavki r ali R, potem velja specifikacija. Ker bomo pogosto uporabljali ukaz $v := e$, je dobro, da imamo za to posebej zapisano pravilo.

$$\frac{\forall s.\, p(s) \Rightarrow q(s[v := e])}{p \{\!\{v := e\}\!\} q}$$

To je poseben primer pravila za posodobitev in je podoben standardnemu pravilu za priredbo v Hoareovi logiki. Pravilo za zaporedno izvajanje je enako standardnemu v Hoareovi logiki.

$$\frac{p \{\!\{S\}\!\} r \qquad r \{\!\{T\}\!\} q}{p \{\!\{S; T\}\!\} q}$$

Dodamo še dve strukturni pravili.

$$\frac{\forall s.\, p'(s) \Rightarrow p(s) \qquad p \{\!\{S\}\!\} q \qquad \forall s.\, q(s) \Rightarrow q'(s)}{p' \{\!\{S\}\!\} q'} \qquad \frac{p \{\!\{S\}\!\} q \qquad p' \{\!\{S\}\!\} q'}{p \wedge p' \{\!\{S\}\!\} q \wedge q'}$$

Ti nam omogočata ošibitev predpogojev, okrepitev končnih pogojev in združevanje specifikacij. Dodajmo še dve pravili za praktično dokazovanje.

$$\frac{p \Rightarrow \mathrm{wp}(S, q)}{p \,\{\!\{S\}\!\}\, q} \qquad \frac{\mathrm{sp}(S, p) \Rightarrow q}{p \,\{\!\{S\}\!\}\, q}$$

Pri praktičnem dokazovanju bomo uporabljali še *najšibkejše predpogoje* [5] in *najmočnejše končne pogoje*. Te bomo pozneje definirali z operatorjema wp in sp.

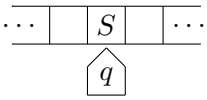
Hoareovo logiko za invariantne diagrame definiramo z naslednjim pravilom

$$\frac{\forall i, j. \vdash P(i) \,\{\!\{E(i,j)\}\!\}\, P(j)}{P \vdash \,\{\!\{E\}\!\}}$$

Tukaj bomo uporabili poenostavljeno različico pravil za invariantne diagrame. V izvornem članku [8] so pravila bistveno bolj zapletena, saj vključujejo še ustavitev.

3. Turingov stroj

V nadaljevanju bomo definirali *Turingove stroje* in diagrame invariant za Turingove stroje. Turingov stroj je matematični model računalnika, ki se uporablja za preučevanje računske zahtevnosti in izračunljivosti. Sestavljen je iz neskončnega traku (slika 3), razdeljenega na celice, in bralno-pisalne glave, ki se lahko premika po traku in spreminja vsebino celic. Glava ima notranje stanje, ki se lahko spreminja med izvajanjem. Teh stanj je končno mnogo. Delovanje glave je opisano z delno definirano tranzicijsko funkcijo, ki za dani vhod in stanje določa novo stanje, nov simbol na traku in smer premika glave. Trak je neskončen v obe smeri in neprazen le na končno mnogih celicah. Prazno celico označimo s znakom $\square$.



Slika 3. Shema Turingovega stroja.

Turingov stroj se začne izvajati na najbolj levi neprazni celici traku, če ta obstaja. V nadaljevanju sledi tranzicijam, dokler so te definirane. Ko ni več definiranih tranzicij, se izvajanje ustavi.

Definicija 4. *Deterministični Turingov stroj* (Γ, Q, δ) je definiran s

- končno množico znakov Γ z $\square \in \Gamma$,
- končno množico stanj Q z $\texttt{START} \in Q$ in
- delno definirano funkcijo $\delta : Q \times \Gamma \rightharpoonup Q \times \Gamma \times \{-1, 0, +1\}$.

Trak definiramo kot funkcijo tipa $\mathbb{Z} \to \Gamma$, ki za vsak celoštevilski indeks vrne znak. Začetno stanje traku je končna beseda, zapisana na zaporednih celicah z indeksi od 0 naprej, medtem ko so vse ostale celice prazne. Turingov stroj lahko na vsakem koraku spremeni le en znak na traku. To pomeni, da na vsakem koraku izvedbe Turingovega stroja obstaja le končno mnogo nepraznih znakov na traku. Začetni trak definiramo z besedo x. Vsaka beseda $x = x_0 x_1 \ldots x_{k-1} \in \Gamma^*$ in $j \in \mathbb{Z}$ definira trak $x@j$.

$$x@j(i) = \begin{cases} x_{i-j} & \text{če } j \le i < j + k \\ \square & \text{sicer} \end{cases}$$

Prav tako vsako tračno stanje in stanje glave opredelimo s *konfiguracijo Turingovega stroja*. Ta nam pove trenutno stanje glave, njen položaj in stanje traku. Formalno je konfiguracija Turingovega stroja trojica (q, t, i), kjer je $q \in Q$ trenutno stanje, t tračno stanje tipa $\mathbb{Z} \to \Gamma$ in $i \in \mathbb{Z}$ trenutna pozicija glave.

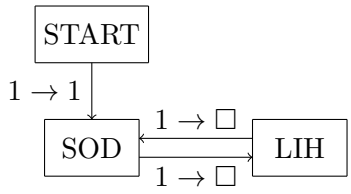
Spremembo traku formalno opišemo s funkcijo step, ki nam pove, kako se konfiguracija spremeni z enim korakom izvedbe Turingovega stroja.

$$\mathrm{step}(q,t,i) = \begin{cases} (q', t[i := a], i+d) & \text{če } \delta(q,t(i)) = (q',a,d) \\ \text{nedefinirano} & \text{če je } \delta(q,t(i)) \text{ nedefiniran} \end{cases}$$

Izvajanje Turingovega stroja je torej definirano s ponavljajočo aplikacijo funkcije step. Poglejmo si primer Turingovega stroja, ki ugotovi parnost dolžine vhodne besede. Tega nam ne sporoči z izhodom, ampak s stanjem, v katerem se ustavi.

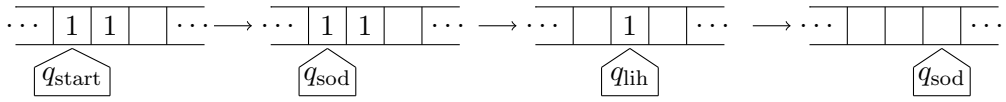
Primer 5. Turingov stroj, ki ugotovi parnost dolžine vhodne besede (slika 4).

$$\begin{aligned} \Gamma &:= \{1, \square\} \\ Q &:= \{\mathrm{start}, q_{\mathrm{sod}}, q_{\mathrm{lih}}\} \\ \delta(\mathrm{start}, 1) &:= (q_{\mathrm{sod}}, 1, 0) \\ \delta(q_{\mathrm{sod}}, 1) &:= (q_{\mathrm{lih}}, \square, +1) \\ \delta(q_{\mathrm{lih}}, 1) &:= (q_{\mathrm{sod}}, \square, +1) \end{aligned}$$



Slika 4. Turingov stroj za ugotavljanje parnosti dolžine vhodne besede.

Skica izvajanja stroja za vhodno besedo 11 je prikazana na sliki 5.



Slika 5. Prikaz izvajanja Turingovega stroja za ugotavljanje parnosti.

Za opisovanje stanj traku bomo uvedli novo notacijo (izvirno za ta članek), ki nam bo omogočala krajši zapis. Prav tako bomo uvedli še pravila sklepanja s to notacijo, v naslednjem poglavju pa tudi transformer najmočnejših končnih pogojev, ki ga bomo uporabljali pri dokazovanju pravilnosti.

Definicija 6. *Opisna tračna notacija* je zapis trditev o traku. Ta je razširjena oblika regularnih izrazov. Glavna razširitev je izražanje možnih položajev bralno-pisalne glave. To se izrazi s podčrtovanjem simbolov. Če je simbol podčrtan, potem je bralno-pisalna glava na tem simbolu. Če ni, potem glava ne sme biti na tem simbolu.

OT notacija	R	$::=$	ε	prazna beseda
		$\mid$	cR	neopazovan simbol c
		$\mid$	c^*R	0 ali več ponovitev neopazovanega simbola c
		$\mid$	$\underline{c}R$	opazovan simbol c
		$\mid$	$\underline{c^*}R$	1 ali več ponovitev simbola c, vsaj eden je opazovan
simbol	c	$::=$	a	znak $a \in \Gamma$
		$\mid$	$\overline{a}$	znak, ki ni $a \in \Gamma$
		$\mid$	$a_1 + \cdots + a_n$	znak, ki je eden izmed $a_1, \ldots, a_n \in \Gamma$

Definiramo še:

$$\gamma := \sum_{c \in \Gamma} c \qquad \text{poljuben znak}$$

$$\tilde{\gamma} := \sum_{c \in \Gamma \setminus \{\square\}} c \qquad \text{poljuben neprazen znak}$$

Primer 7 (Primeri opisne tračne notacije)**.** Za lažje razumevanje opisne tračne notacije si poglejmo nekaj primerov.

- $\underline{\mathbf{1}}\mathbf{11}$ predstavlja stanje, kjer je glava na prvi enici, za njo pa sta še dve enici. Preostali del traku je prazen.
- $\mathbf{0}^*\underline{\mathbf{1}}\mathbf{2}^*$ predstavlja stanje, kjer je glava na enici. Pred njo je 0 ali več ničel, za njo pa 0 ali več dvojic. Preostali del traku je prazen. Nekaj primerov trakov, ki ustrezajo tej označbi: $\square\underline{1}\square$, $\square 00\underline{1}\square$, $\square 0\underline{1}2\square$.
- $\underline{\mathbf{12}}$ ustrezata natanko dve stanji: $\underline{1}2$ in $1\underline{2}$.
- $\underline{\mathbf{1}}^*$ predstavlja stanje, kjer je glava na eni enici, okrog nje pa je poljubno mnogo enic. Primeri: $\underline{1}$, $1\underline{1}$, $\underline{1}11$, $11\underline{1}111$. Označba je enakovredna $1^*\underline{1}1^*$.
- $\underline{\mathbf{0}^*\mathbf{12}^*}$ je bolj kompleksna. Znaki na traku so enaki kot v primeru $0^*\underline{1}2^*$. Sedaj je lahko glava na poljubnem izmed teh znakov. Vsi trakovi, ki ustrezajo označbam $0^*\underline{1}2^*$, $\underline{0}^*12^*$, $0^*1\underline{2}^*$, ustrezajo tudi tej označbi.
- $\underline{\gamma^*}$ ustrezajo vsa stanja trakov.
- $\underline{\tilde{\gamma}^*}$ pa ustreza vsem nepraznim trakom, ki nimajo praznih celic znotraj besede.

Sedaj bomo formalizirali semantiko opisne tračne notacije. Za to bomo uporabili posredno notacijo, katere semantiko lažje definiramo.

Definicija 8 (Posredna tračna notacija)**.** To notacijo bomo interpretirali z bolj preprosto notacijo. Ta notacija ima natanko eno mesto za bralno-pisalno glavo. Prav tako pa je bralno-pisalna glava le na samostojnih simbolih ($\underline{c}$) in na skupinah simbolov ($\underline{c^*}$). Zapis $\underline{c}^*$ ni veljaven.

$$\begin{array}{lrcl}
\text{PT notacija} & T & ::= & cT \\
 & & | & c^*T \\
 & & | & \underline{c}B \\
 & & | & \underline{c^*}B \\
\text{B notacija} & B & ::= & \varepsilon \\
 & & | & cB \\
 & & | & c^*B
\end{array}$$

Semantiko bomo definirali v dveh delih. Najprej definiramo semantiko, ki nam pove možne besede, ki jih določena označba predstavlja. To bomo definirali z množico $W_\Gamma(R)$. Potem bomo definirali semantiko, ki nam pove možne konfiguracije traku z $L_\Gamma(R)$.

$$\begin{aligned}
W_\Gamma(a) &:= \{a\} & W_\Gamma(\varepsilon) &:= \{\varepsilon\} \\
W_\Gamma(\overline{a}) &:= \Gamma \setminus \{a\} & W_\Gamma(cT) &:= \{dt \mid d \in W_\Gamma(c),\ t \in W_\Gamma(T)\} \\
W_\Gamma(a_1 + \cdots + a_n) &:= \{a_1, \ldots, a_n\} & W_\Gamma(c^*T) &:= \{st \mid s \in \bigcup_{n\geq 0} W_\Gamma(c)^n,\ t \in W_\Gamma(T)\}
\end{aligned}$$

Semantika za posredno tračno notacijo je le omejena semantika Kleeneovih regularnih izrazov [12]. Položaj bralno-pisalne glave obravnavamo kot družino znakov ($\underline{c}$), zato je naša semantika definirana tako za PT notacijo kot tudi B notacijo.

Sedaj pa obogatimo besede s položajem na traku. Zaradi definicije traku razlikujemo med besedami na različnih mestih traku, kljub temu, da so obdane s praznimi celicami.

$$L_\Gamma(r'\underline{c'}\,r'') := \{(wcw'@j, j + |w|) \mid j \in \mathbb{Z},\ c \in L_\Gamma(c'),\ w \in L_\Gamma(r'),\ w' \in L_\Gamma(r'')\}$$

Sedaj lahko definiramo semantiko celotne opisne tračne notacije. Za končno semantiko pa potrebujemo še funkcijo rem, ki odstrani podčrtane simbole iz opisne tračne notacije.

$$\begin{aligned}\text{rem} : \text{OT notacija} &\to \text{B notacija}\\ \text{rem}(\varepsilon) &:= \varepsilon\\ \text{rem}(cT) &:= c\,\text{rem}(T)\\ \text{rem}(c^*T) &:= c^*\,\text{rem}(T)\\ \text{rem}(\underline{c}R) &:= c\,\text{rem}(R)\\ \text{rem}(\underline{c^*}R) &:= c^*\,\text{rem}(R)\end{aligned}$$

Sedaj gremo čez vsako podčrtano oznako. Če je podčrtan posamezen simbol, odstranimo ostale podčrtane simbole, sicer pa pretvorimo $\underline{c^*}$ v $c^*\underline{c}c^*$ in odstranimo ostale podčrtane simbole.

$$L_\Gamma(R) := \bigcup_{R=R'\underline{c}R''} L_\Gamma(\text{rem}(R')\,\underline{c}\,\text{rem}(R'')) \;\cup\; \bigcup_{R=R'\underline{c^*}R''} L_\Gamma(\text{rem}(R')c^*\underline{c}c^*\,\text{rem}(R''))$$

Pri dokazovanju pravilnosti bomo potrebovali sklepati o vsebovanosti označb. Za to lahko uporabimo standardna dejstva o regularnih izrazih [12].

$$\begin{aligned}L_\Gamma(\varepsilon) &\subseteq L_\Gamma(c^*)\\ L_\Gamma(c) &\subseteq L_\Gamma(c^*)\\ L_\Gamma(cc^*) &\subseteq L_\Gamma(c^*)\end{aligned}$$

$$\begin{aligned}L_\Gamma(cL) \subseteq L_\Gamma(c'L') &\;\Leftrightarrow\; L_\Gamma(c) \subseteq L_\Gamma(c') \wedge L_\Gamma(L) \subseteq L_\Gamma(L')\\ L_\Gamma(c) \subseteq L_\Gamma(c') &\;\Rightarrow\; L_\Gamma(c^*) \subseteq L_\Gamma(c'^*)\end{aligned}$$

Ker je beseda na traku obdana z neskončno praznimi celicami, veljata tudi naslednji enakosti

$$\begin{aligned}L_\Gamma(L) &= L_\Gamma(L\square)\\ L_\Gamma(L) &= L_\Gamma(\square L)\end{aligned}$$

4. Diagrami invariant za Turingove stroje

Sedaj lahko opišemo diagramski sistem, ki ga bomo uporabili za dokazovanje pravilnosti Turingovih strojev. Ta diagramski sistem bo imel dve semantiki. Ena semantika bo diagram interpretirala kot Turingov stroj. Druga semantika bo interpretirala diagram kot diagram invariant. Obe semantiki predstavljata Turingov stroj, le na drugačen način. Skladnosti semantik ni težko videti, zato v tem delu izpustimo dokaz tega dejstva.

Posebnost diagramov invariant za Turingove stroje je, da ima Turingov stroj zelo omejen nabor ukazov. Ti so oblike $c\,\$\,d$, kjer c predstavlja pogoj izvedbe ukaza (da je glava na simbolu c), $\$ \in \{\leftarrow, \rightarrow, \downarrow\}$ smer premika glave (levo, desno, brez premika) in d simbol, s katerim zamenjamo c. Da te implementiramo v diagramu invariant, bomo uporabili spremenljivke $\mathtt{t}, \mathtt{q}, \mathtt{i}$. V $\mathtt{t}$ bomo shranjevali trenutni trak, v $\mathtt{q}$ trenutno stanje in v $\mathtt{i}$ trenutni položaj glave. Da ohranimo polnost sistema, bomo omogočili še delno uporabo preostalih ukazov. Ti ne bodo smeli spreminjati prej omenjenih spremenljivk, prav tako ne bodo smeli vplivati na aktiviranost prehodov. Takim spremenljivkam pravimo, da so *fantomske*.

Sedaj pa lahko še formaliziramo definicije. Začnimo s formalno definicijo ukazov. Te bomo razširili z vezavo spremenljivk. Vsak ukaz bomo označili z $v : p\ \$\ a$, kjer je $v \in \text{Var}$ spremenljivka, p logična formula, $\$$ smer in $a \in \Gamma$ znak. Pove nam, da če znak na glavi izpolnjuje logični izraz p, potem se znak zamenja z a, zapiše izvorno vrednost v v in se glava premakne v smeri $\$$. Seveda lahko tudi izpustimo vezavo spremenljivk. Namesto logičnega izraza lahko tudi uporabimo opisno tračno notacijo (omejeno na posamezne znake). Imamo lahko torej npr. ukaz $v : \overline{\square} \to 2$. Za vsak

logični izraz ali pa opisno tračno notacijo bomo z $[\![\cdot]\!]$ označili množico znakov, ki jih opisuje. Enako notacijo bomo uporabili za semantiko smeri premika glave in posameznih ukazov.

$$\begin{aligned}[\![\leftarrow]\!] &:= -1\\ [\![\downarrow]\!] &:= 0\\ [\![\rightarrow]\!] &:= 1\end{aligned} \qquad [\![v{:}p \,\$\, a]\!]_{k,\ell} := \begin{pmatrix} [\mathfrak{t}(\mathfrak{i}) \in [\![p]\!]];\\ \mathfrak{q} := \ell;\\ \mathfrak{t} := \mathfrak{t}[\mathfrak{i} := a];\\ \mathfrak{i} := \mathfrak{i} + [\![\$]\!] \end{pmatrix}$$

Spremenljivki k in ℓ predstavljata trenutno situacijo in novo situacijo Turingovega stroja.

Prej omenjene omejitve na ukazih bomo izrazili s pomočjo dveh funkcij W in *vars*. Funkcija W nam pove, katere spremenljivke ukaz (brez predpostavk, trditev in demonske posodobitve) spremeni, funkcija vars pa nam pove, katere spremenljivke so vezane na znake.

$$\begin{aligned}W(v := e) &:= \{v\}\\ W(u; v) &:= W(u) \cup W(v)\\ W(S \sqcap T) &:= W(S) \cup W(T)\end{aligned} \qquad \operatorname{vars}(v{:}p \,\$\, q) := \{v\}$$

Sedaj lahko definiramo diagram invariant za Turingove stroje.

Definicija 9 (Diagram invariant za Turingove stroje)**.** *Diagram invariant za Turingove stroje* je usmerjen graf z oznakami na povezavah (Γ, V, s, E, P), kjer je:

- Γ končna množica znakov,
- V končna množica *situacij*,
- $s \in V$ začetna situacija,
- $E : V \times V \to \{(v{:}p \,\$\, a, u) \mid v \in \mathrm{Var},\ p \text{ je predikat},\ a \in \Gamma,\ u \text{ je ukaz}\}$ funkcija prehodov in
- $P : V \to \mathrm{Pred}$ množica invariant.

Stroj mora za vsak prehod $E(i,j) = (u,\ u')$ dodatno izpolnjevati naslednja pogoja:

- ukaz u' ne sme imeti trditev, predpostavk ali demonske posodobitve,
- množici $\bigcup_{(u,\_)\in\mathrm{cod}(E)} \operatorname{vars}(u)$ in $W(u')$ sta disjunktni,

kjer je $\mathrm{cod}(E)$ zaloga vrednosti funkcije E.

Vsakemu diagramu invariant za Turingove stroje lahko priredimo Turingov stroj. Za diagram invariant za Turingove stroje $G = (\Gamma, V, s, E, P)$ je ustrezni Turingov stroj določen s spodnjo definicijo.

$$Q := V \times \Gamma^{\operatorname{vars}(G)}$$

$$\delta((i,\sigma),a) := \begin{cases}((j, \sigma[v_1 := a]), a', [\![\$]\!]) & \text{če } E(i,j) = (v_1{:}p \,\$\, q, \_) \text{ in } a \in [\![p]\!] \text{ in } a' \in W_\Gamma(q)\\ \text{nedefinirano} & \text{sicer.}\end{cases}$$

Pri tej prevedbi zanemarimo dodatne ukaze in vezave zakodiramo v stanje Turingovega stroja.

Pretvorba v diagram invariant je bila večinoma že prej opisana. Za diagram invariant za Turingove stroje $G = (\Gamma, V, s, E, P)$ je ustrezni diagram invariant (V, s, E', P), kjer velja

$$\begin{aligned}V &:= V\\ s &:= s\\ E'(i,j) &:= [\![E_0(i,j)]\!]; E_1(i,j)\\ P &:= P\end{aligned}$$

diagram invariant. Tu $E_0(i,j)$ predstavlja prvo komponento $E(i,j)$ in $E_1(i,j)$ drugo. $E_0(i,j)$ je napisan pred $E_1(i,j)$, ker lahko ukaz $E_1(i,j)$ potrebuje spremenljivke, ki jih $E_0(i,j)$ veže.

Za lažje pisanje stanja traku bomo dodali še nekaj notacije. Če napišemo samo tračno notacijo R, potem to interpretiramo kot

$$R :\Leftrightarrow (\mathfrak{t}, \mathfrak{i}) \in L_\Gamma(R).$$

Če hočemo dokazati delno pravilnost teh diagramov, moramo podobno kot pri navadnih diagramih pokazati pravilnost prehodov. Ker je pretvorba dovolj enostavna, ne bomo še posebej formalizirali pravila sklepanja. Ker pa imamo definirano začetno situacijo, moramo še dokazati, da velja $\underline{\gamma^*} \Rightarrow P(s)$.

4.1 Praktično dokazovanje pravilnosti diagramov invariant

Pri dokazovanju pravilnosti si bomo pomagali z *najšibkejšimi predpogoji* ukazov. Te označimo s *predikatnim transformerjem* wp, ki nam za $\mathrm{wp}(S,q)$ pove, kaj vsaj mora veljati pred izvajanjem ukaza S, da bo po njegovi izvedbi veljalo q. Če hočemo dokazati pravilnost prehoda iz p na q z wp, potem moramo dokazati, da velja

$$p \implies \mathrm{wp}([\![v]\!], \mathrm{wp}(u, q)).$$

Tu je v ukaz za premik Turingovega stroja, u pa dodatni ukaz za delo s fantomskimi spremenljivkami. V prihodnje bomo prehode zapisovali s $p \xrightarrow[u]{v} q$.

Definicija 10. Definiramo *predikatni transformer* $\mathrm{wp} : \mathrm{Ukaz} \times \mathrm{Pred} \to \mathrm{Pred}$ z induktivno definicijo.

$$\begin{aligned}
\mathrm{wp}(\{p\}, q) &:\Leftrightarrow p \wedge q \\
\mathrm{wp}([p], q) &:\Leftrightarrow (p \Rightarrow q) \\
\mathrm{wp}([R], q)(s) &:\Leftrightarrow (\forall s'.\ s\ R\ s' \Rightarrow s' \in q) \\
\mathrm{wp}(v := e, q) &:\Leftrightarrow q[v := e] \\
\mathrm{wp}(S \sqcap T, q) &:\Leftrightarrow \mathrm{wp}(S, q) \wedge \mathrm{wp}(T, q) \\
\mathrm{wp}(S; T, q) &:\Leftrightarrow \mathrm{wp}(S, \mathrm{wp}(T, q))
\end{aligned}$$

Primer 11. Dokažimo pravilnost prehoda $\mathrm{sod}(i) \xrightarrow{[i\geq 0];\, i:=i+1} \mathrm{lih}(i)$ iz primera 3. Uporabimo podobno pravilo kot zgoraj, le da izpustimo del za upravljanje Turingovega stroja.

$$\begin{array}{lll}
 & \mathrm{sod}(i) \Rightarrow \mathrm{wp}(i{:=}i{+}1, \mathrm{wp}([i \geq 0], \mathrm{lih}(i))) & \\
\iff & \mathrm{sod}(i) \Rightarrow \mathrm{wp}(i{:=}i{+}1, i \geq 0 \Rightarrow \mathrm{lih}(i)) & \text{definicija wp} \\
\iff & \mathrm{sod}(i) \Rightarrow (i{+}1 \geq 0 \Rightarrow \mathrm{lih}(i{+}1)) & \text{definicija wp} \\
\iff & \mathrm{sod}(i) \wedge i{+}1 \geq 0 \Rightarrow \mathrm{lih}(i{+}1) & \text{lastnost implikacije} \\
\iff & \top & \text{lastnost sodosti in lihosti}
\end{array}$$

Prav tako si bomo pomagali z *najmočnejšimi končnimi pogoji* ukazov. Te označimo s *transformerjem končnih pogojev* $\mathrm{sp}(S,q)$, ki nam pove, kaj največ lahko velja po izvedbi ukaza S v stanju, ki izpolnjuje q. Te bomo uporabljali za dokazovanje s tračno notacijo. Če hočemo dokazati pravilnost prehoda $p \xrightarrow[u]{v} q$ z sp, potem moramo dokazati, da velja

$$\mathrm{sp}(u, \mathrm{sp}([\![v]\!], p)) \implies q.$$

Preden definiramo sp, potrebujemo še funkcijo lpos, ki nam za opis brez glave vrne opis z vsemi potencialnimi začetnimi mesti glave. Intuitivno predstavlja premik v nek opis.

$$\begin{aligned}\mathrm{lpos}(cL) &:= \{\underline{c}L\}\\ \mathrm{lpos}(c^*L) &:= \{\underline{c}c^*L\} \cup \mathrm{lpos}(L)\end{aligned}$$

Pri definiciji sp bomo uporabili še funkcijo rev, ki nam obrne opis (torej $\mathrm{rev}(01^*) = 1^*0$). Z N_i' označujemo število znakov i na traku, z N_i pa število znakov i na traku pred izvajanjem programa. Za pogoj p je $\mathbb{1}(p)$ enak 1, če velja p, sicer pa 0. Spremenljivka r je kvantificirana čez OT notacijo, E pa čez enakosti in primerjave (primer: $1 + N_0 < 4$). Oznake $p_1, \ldots, p_n$ pa predstavljajo vse znake, ki jih p opisuje.

Definicija 12. Definiramo *transformer končnih pogojev* $\mathrm{sp} : \mathrm{Ukaz} \times \mathrm{Pred} \to \mathrm{Pred}$ z induktivno definicijo.

$$\begin{aligned}
\mathrm{sp}(v := e, q) &:\Leftrightarrow \exists t.\, v = e[v := t] \wedge q[v := t]\\
\mathrm{sp}(S; T, q) &:\Leftrightarrow \mathrm{sp}(T, \mathrm{sp}(S, q))\\
\mathrm{sp}(v{:}p\,\$\,d, E) &:\Leftrightarrow E[N_d' := N_d' + \mathbb{1}(d \notin p)][N_{p_1}' := N_{p_1}' - \mathbb{1}(p_1 \neq d)] \ldots [N_{p_n}' := N_{p_n}' - \mathbb{1}(p_n \neq d)]\\
\mathrm{sp}(v{:}p\,\$\,d, r) &:\Leftrightarrow \bigvee_{c \in p} \mathrm{sp}(c\,\$\,d, r)\\
\mathrm{sp}(c\downarrow d, r'\underline{c}r'') &:\Leftrightarrow r'\underline{d}r''\\
\mathrm{sp}(c\rightarrow d, r'\underline{c}r'') &:\Leftrightarrow r'd\,\mathrm{lpos}(r'')\\
\mathrm{sp}(c\leftarrow d, r'\underline{c}r'') &:\Leftrightarrow \mathrm{rev}(\mathrm{lpos}(\mathrm{rev}(r')))dr''
\end{aligned}$$

Večina ukazov ima standardno definicijo za sp [13], razen definicij z $v{:}p\,\$\,d$.

Skica dokaza za novi definiciji sp. Treba je pokazati, da sp „zaobjame vse posledice“. Za prve štiri definicije to ni težko pokazati, saj so večinoma standardne. Malo bolj podrobno opišimo preostale definicije. Za negibno glavo ni težko pokazati. Za premik glave v desno pa se lahko zaradi lokalnosti izvajanja Turingovega stroja in oblike PT notacije za $r'\underline{d}c_1^* \ldots c_n^* c' r'$ osredotočimo le na $\underline{d}c_1^* \ldots c_n^* c'$, od tod pa nadaljujemo z indukcijo na n in obravnavo glede na to, ali je c^* prazen ali ne. Podobno naredimo tudi za premik glave v levo. ∎

Za predikatne transformerje velja tudi naslednje.

$$\begin{aligned}
\mathrm{wp}(S, A \wedge B) &\Leftrightarrow \mathrm{wp}(S, A) \wedge \mathrm{wp}(S, B)\\
\mathrm{wp}(S, A \vee B) &\Leftarrow \mathrm{wp}(S, A) \vee \mathrm{wp}(S, B)\\
\mathrm{sp}(S, A \vee B) &\Leftrightarrow \mathrm{sp}(S, A) \vee \mathrm{sp}(S, B)\\
\mathrm{sp}(S, A \wedge B) &\Rightarrow \mathrm{sp}(S, A) \wedge \mathrm{sp}(S, B)
\end{aligned}$$

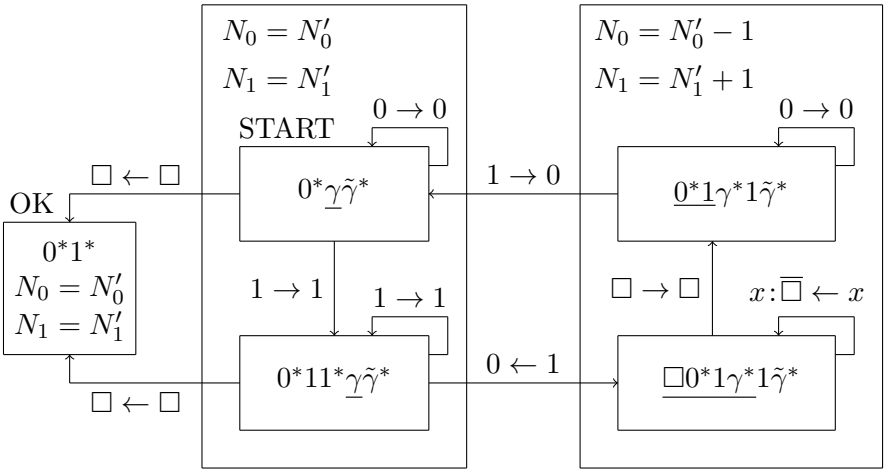
Prav tako lahko vidimo, da velja

$$\mathrm{sp}(c \rightarrow c, E) \Leftrightarrow E. \tag{1}$$

5. Primer uporabe – urejanje niza ničel in enic

Kot večji primer si bomo ogledali diagram invariant (slika 6), ki ureja niz ničel in enic. Program na začetku „predpostavi“, da je niz urejen. Najprej gre čez ničle, nato pa čez enice. Če naleti na ničlo, potem „prestavi“ ničlo na levo.

Za dokaz je potrebno dokazati pravilnost vseh deset prehodov. Za prikaz bomo dokazali pravilnost dveh.



Slika 6. Diagram invariant za Turingov stroj, ki ureja niz ničel in enic.

5.1 Dokaz pravilnosti

Na začetku velja $N_0 = N_0'$, $N_1 = N_1'$ in $\underline{\gamma}\tilde{\gamma}^* \Rightarrow 0^*\underline{\gamma}\tilde{\gamma}^*$. S tem smo dokazali, da je začetek dobro definiran. Sedaj pa moramo dokazati, da se invariante ohranjajo za vsak prehod.

Začnimo z dokazom pravilnosti prehoda

$$\begin{gathered} N_0 = N_0' \wedge N_1 = N_1' \wedge 0^*\underline{\gamma}\tilde{\gamma}^* \\ \xrightarrow{0\to 0} \\ N_0 = N_0' \wedge N_1 = N_1' \wedge 0^*\underline{\gamma}\tilde{\gamma}^*, \end{gathered}$$

ki ga bomo dokazovali s pomočjo sp.

$$\begin{aligned} &\mathrm{sp}(0 \to 0,\ N_0 = N_0' \wedge N_1 = N_1' \wedge 0^*\underline{\gamma}\tilde{\gamma}^*) \\ &\quad \Rightarrow \mathrm{sp}(0 \to 0, N_0 = N_0') \wedge \mathrm{sp}(0 \to 0, N_1 = N_1') \wedge \mathrm{sp}(0 \to 0, 0^*\underline{\gamma}\tilde{\gamma}^*) && \text{distributivnost preko } \wedge \\ &\quad \Leftrightarrow N_0 = N_0' \wedge N_1 = N_1' \wedge (0^*0\underline{\tilde{\gamma}}\tilde{\gamma}^* \vee 0^*0\underline{\square}) && (1) \text{ in definicija sp} \\ &\quad \Rightarrow N_0 = N_0' \wedge N_1 = N_1' \wedge 0^*\underline{\gamma}\tilde{\gamma}^* && \tilde{\gamma} \Rightarrow \gamma,\ \square \Rightarrow \gamma \end{aligned}$$

Sedaj dokažimo pravilnost prehoda

$$\begin{gathered} N_0 = N_0' \wedge N_1 = N_1' \wedge 0^*11^*\underline{\gamma}\tilde{\gamma}^* \\ \xrightarrow{0\leftarrow 1} \\ N_0 = N_0' - 1 \wedge N_1 = N_1' + 1 \wedge \underline{\square 0^*1\gamma^*}1\gamma^*, \end{gathered}$$

$$\begin{aligned} &\mathrm{sp}(0 \leftarrow 1,\ N_0 = N_0' \wedge N_1 = N_1' \wedge 0^*11^*\underline{\gamma}\tilde{\gamma}^*) \\ &\quad \Rightarrow \mathrm{sp}(0 \leftarrow 1, N_0 = N_0') \wedge \mathrm{sp}(0 \leftarrow 1, N_1 = N_1') \wedge \mathrm{sp}(0 \leftarrow 1, 0^*11^*\underline{\gamma}\tilde{\gamma}^*) && \text{distributivnost preko } \wedge \\ &\quad \Leftrightarrow N_0 = N_0' - 1 \wedge N_1 = N_1' + 1 \wedge (0^*\underline{1}1\tilde{\gamma}^* \vee 0^*11^*\underline{1}1\tilde{\gamma}^*) && \text{definicija sp} \\ &\quad \Rightarrow N_0 = N_0' - 1 \wedge N_1 = N_1' + 1 \wedge \underline{\square 0^*1\gamma^*}1\gamma^* && \text{lastnosti tračne notacije} \end{aligned}$$

Podobno lahko dokažemo pravilnost ostalih prehodov in z njimi pravilnost celotnega programa.

6. Zaključek

V tem delu smo si na hitro ogledali diagrame invariant in njihovo prilagoditev za Turingove stroje. Lahko bi jih razširili na nove tipe Turingovih strojev, kot so večtračni, večdimenzijski, nedeterministični in drugi. Za nedeterministične bi bilo potrebno prilagoditi semantiko diagramov invariant, da bi namesto demonskega nedeterminizma uporabljali angelskega. Prav tako bi lahko poenotili način sklepanja, saj trenutno uporabljamo Hoareovo logiko, najšibkejše predpogoje in najmočnejše končne pogoje.

LITERATURA

[1] A. Turing, *Checking a large routine*, v: *The Early British Computer Conferences*, 1989, str. 70–72.

[2] R.W. Floyd, *Assigning meanings to programs*, v: *Program Verification: Fundamental Issues in Computer Science*, Springer, 1993, str. 65–81.

[3] E.W. Dijkstra, *Letters to the editor: go to statement considered harmful*, Communications of the ACM **11** (1968), št. 3, 147–148.

[4] C.A.R. Hoare, *An axiomatic basis for computer programming*, Communications of the ACM **12** (1969), št. 10, 576–580.

[5] E.W. Dijkstra, *Guarded commands, nondeterminacy and formal derivation of programs*, Communications of the ACM **18** (1975), št. 8, 453–457.

[6] J.C. Reynolds, *Programming with transition diagrams*, v: *Programming Methodology: A Collection of Articles by Members of IFIP WG2.3*, Springer, 1978, str. 153–165.

[7] R.-J. Back, *Invariant based programming*, Proceedings of the International Conference on Application and Theory of Petri Nets, Springer, 2006, str. 1–18.

[8] R.-J. Back in V. Preoteasa, *Semantics and proof rules of invariant based programs*, Proceedings of the 2011 ACM Symposium on Applied Computing, 2011, str. 1658–1665.

[9] R.-J. Back, J. Eriksson in M. Myreen, *Testing and verifying invariant based programs in the SOCOS environment*, Proceedings of the International Conference on Tests and Proofs, Springer, 2007, str. 61–78.

[10] R.-J. Back, *Invariant based programming: basic approach and teaching experiences*, Formal Aspects of Computing **21** (2009), št. 3, 227–244.

[11] E.F.A. Lederer, *How to Verify a Turing Machine with Dafny*, arXiv:2601.15230 [cs.LO], 2026.

[12] D. Kozen, *A completeness theorem for Kleene algebras and the algebra of regular events*, Information and Computation **110** (1994), št. 2, 366–390.

[13] M. Gordon in H. Collavizza, *Forward with Hoare*, v: *Reflections on the Work of C.A.R. Hoare*, Springer, 2010, str. 101–121.