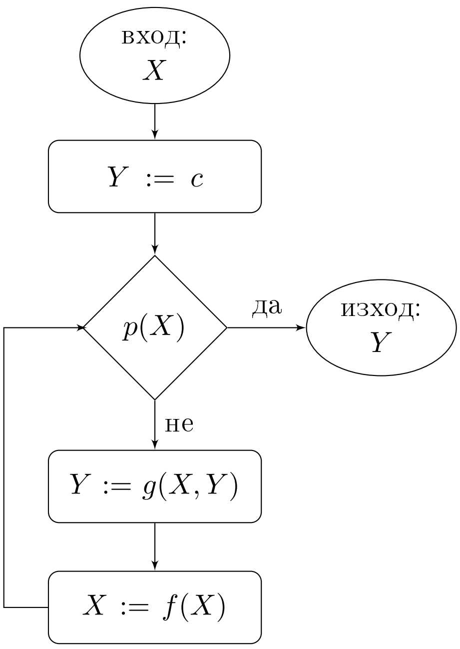
Лекция №12:

Сравнителна схематология

4.1 Схеми на програми

Преди да дадем точните дефиниции, да си изясним разликата между *схема на програма* и програма.
Да разгледаме блок-схемата S от по-долу. Да се запитаме какво прави тази блок-схема; коя функция пресмята тя?



Пример 4.1.

Да, въпросът какво пресмята S е безсмислен, защото тази картинка все още не е програма; тя е само *схема на програма*. Не знаем върху какви обекти работи, не знаем и какъв е смисълът на символите c, f, g и p, участващи в нея.

Една подходяща аналогия в случая е връзката между *формула* от първи ред и нейната *вярност*, за която можем да говорим след като сме придали смисъл на нелогическите символи, участващи във формулата. Например, за формулата φ

$$\forall X \exists Y p(X, Y)$$

е безсмислено да се питаме дали е вярна. За да е смислен въпросът, трябва да знаем къде варират променливите X и Y и какъв е смисълът на буквата p. Това става чрез фиксирането на *структура* в сигнатурата на φ, която в случая се състои само от двуместния предикатен символ p. Ако вземем структурата $\mathcal{N} = (\mathbb{N},\ >)$, в която p се интерпретира като предиката $>$, то очевидно φ не е вярна в $\mathcal{N}$. Ако сменим универсума с множеството $\mathbb{Z}$ на целите числа и разгледаме структурата $\mathcal{Z} = (\mathbb{Z},\ >)$, в нея вече φ ще е вярна.

Да се върнем отново на схемата S от *Пример* 4.1. *Сигнатурата* на схемата е

$$\Sigma \ = \ (c;\ f,\ g;\ p),$$

където c е константен символ, f и g са функционални символи (на 1 и на 2 аргумента, съответно), а p е едноместен предикатен символ. *Структура* $\mathcal{A}$ в тази сигнатура е редица от вида

$$\mathcal{A} \ = \ (D;\ c^{\mathcal{A}};\ f^{\mathcal{A}}, g^{\mathcal{A}};\ p^{\mathcal{A}}),$$

където D е непразно множество — *универсум* (или *носител*) на $\mathcal{A}$, а останалите елементи задават интерпретацията на символите от Σ: $c^{\mathcal{A}} \in D$, $f^{\mathcal{A}}$ и $g^{\mathcal{A}}$ са функции в D (съответно на 1 и 2 аргумента), а $p^{\mathcal{A}}$ е едноместен предикат в D.

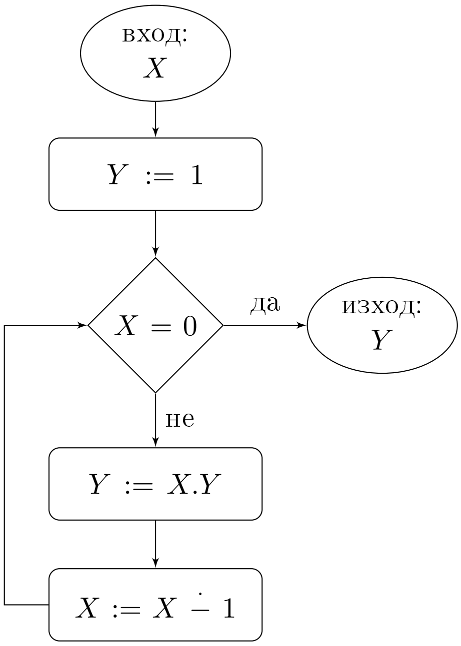
Ето два примера за структури $\mathcal{A}_1$ и $\mathcal{A}_2$ в горната сигнатура $\Sigma \ = \ (c;\ f,\ g;\ p)$. Да дефинираме $\mathcal{A}_1$ по следния начин:

$$\mathcal{A}_1 \ = \ (\mathbb{N};\ 1;\ x \dot{-} 1,\ x.y;\ x = 0?),$$

където функцията $\dot{-}$ (*отсечена разлика*) се определя с равенството:

$$x \dot{-} y = \begin{cases} x - y, & \text{ако} \quad x \geq y \\ 0, & \text{иначе.} \end{cases}$$

В тази структура от схемата S получаваме следната *програма* $(S, \mathcal{A}_1)$:



Пример 4.2.

Лесно се вижда, че програмата $(S, \mathcal{A}_1)$ пресмята функцията $x!$. В този случай ще казваме, че $Sem(S, \mathcal{A}_1)$ е функцията $x!$.

Сега да разгледаме структурата $\mathcal{A}_2$ с носител множеството A^* на всички думи над дадена крайна азбука A:

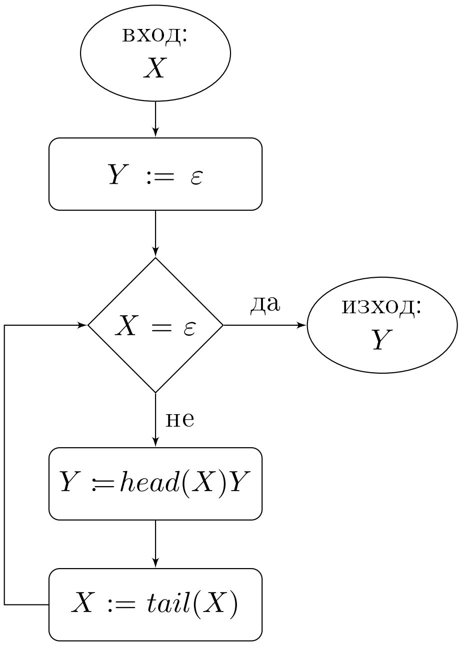
$$\mathcal{A}_2 \ = \ (A^*;\ \varepsilon;\ tail(x),\ head(x)y;\ x = \varepsilon?).$$

Тук ε е празният низ, функциите *head* и *tail* се дефинират по обичайния начин:

$$head(x) \ = \ \begin{cases} a_1, & \text{ако } \ x = a_1 \ldots a_n \text{ и } n > 0 \\ \varepsilon, & \text{ако } \ x = \varepsilon \end{cases} \qquad \text{и}$$

$$tail(x) \ = \ \begin{cases} a_2 \ldots a_n & \text{ако } \ x = a_1 \ldots a_n \text{ и } n > 0 \\ \varepsilon, & \text{ако } \ x = \varepsilon, \end{cases}$$

а $head(x)y$ е *конкатенацията* на $head(x)$ и y. В структурата $\mathcal{A}_2$ от схемата S получаваме следната програма $(S, \mathcal{A}_2)$, която вече работи над низове:



Пример 4.3.

Убедете се, че програмата $(S, \mathcal{A}_2)$ пресмята функцията *reverse*, или формално

$$Sem(S, \mathcal{A}_2) = reverse.$$

Една схема може да прави различни неща, в зависимост от това в каква структура я интерпретираме. Схемата S, която разгледахме, е пример за *стандартна схема*, записана на блок-схемен език. Този тип схеми ще дефинираме формално в следващия раздел.

Под *рекурсивни схеми* ще разбираме синтактични обекти, подобни на рекурсивните програми от езика REC, които разглеждахме досега, като разбира се, те вече няма да са (само) над естествените числа. Точното определение също ще дадем по-нататък; сега ще се ограничим само един пример. Рекурсивната схема R, която следва, е в същата сигнатура $\Sigma = (c;\ f,\ g;\ p)$ като по-горе:

$F(X, c)$ `where`
$F(X, Y) =$ `if` $p(X)$ `then` Y `else` $F(f(X), g(X, Y))$

Рекурсивната програма, която схемата R определя в структурата $\mathcal{A}$, ще отбелязваме с $(R, \mathcal{A})$, а $Sem(R, \mathcal{A})$ ще е нейната семантика, т.е. функцията, която програмата $(R, \mathcal{A})$ пресмята. В тази глава под "семантика" ще разбираме денотационната семантика по стойност на рекурсивната програма $(R, \mathcal{A})$. Защо избираме тази семантика, а не денотационната семантика по име на $(R, \mathcal{A})$ ще стане ясно по-нататък.

4.2 Транслируемост

Определение 4.1. Нека S_1 и S_2 са схеми в една и съща сигнатура Σ. Ще казваме, че те са *еквивалентни* (и ще пишем $S_1 \equiv S_2$), ако за всяка структура $\mathcal{A}$ в сигнатурата Σ е вярно, че

$$Sem(S_1, \mathcal{A}) \;=\; Sem(S_2, \mathcal{A}).$$

С други думи, две схеми са еквивалентни, ако във всяка структура пресмятат една и съща функция.

Да си мислим сега, че имаме два класа от схеми — например стандартните и рекурсивните схеми, които споменахме по-горе. Отново ще предполагаме, че те са в една и съща сигнатура Σ. Можем да ги сравняваме по отношение на тяхната изразителна сила, т.е. по отношение на нещата, които могат да се програмират с техните изразни средства във всяка конкретна структура. Ето по-точните определения:

Определение 4.2. Нека $\mathcal{K}_1$ и $\mathcal{K}_2$ са два класа от схеми в една и съща сигнатура Σ. Ще казваме, че класът $\mathcal{K}_1$ *се транслира* в класа $\mathcal{K}_2$ (и ще пишем $\mathcal{K}_1 \leqslant_t \mathcal{K}_2$), ако

$$\forall S_1 \in \mathcal{K}_1\ \exists S_2 \in \mathcal{K}_2\ \mathcal{S}_1 \equiv S_2.$$

Забележка. Това е т.нар. *силна* (*синтактична* или още *равномерна*) транслируемост, при която на схемно ниво по дадената схема S_1 се конструира еквивалентната на нея схема S_2. Има и по-слабо понятие за неравномерна или *семантична* транслируемост, което се изказва така: $\mathcal{K}_1$ е *слабо транслируем* в $\mathcal{K}_2$, ако

$$\forall S_1 \in \mathcal{K}_1\ \underline{\forall \mathcal{A}\ \exists S_2 \in \mathcal{K}_2}\ Sem(S_1, \mathcal{A}) = Sem(S_2, \mathcal{A}).$$

За да видим по-добре разликата между двете понятия, да разпишем условието определението за транслируемост 4.1:

$$\forall S_1 \in \mathcal{K}_1\ \underline{\exists S_2 \in \mathcal{K}_2\ \forall \mathcal{A}}\ Sem(S_1, \mathcal{A}) = Sem(S_2, \mathcal{A}).$$

Виждаме, че вътрешните два квантора са разменени, което прави второто понятие за транслируемост по-слабо.

Определение 4.3. Ще казваме, че класовете $\mathcal{K}_1$ и $\mathcal{K}_2$ са *еквивалентни* по отношение на транслируемост (и ще пишем $\mathcal{K}_1 \equiv_t \mathcal{K}_2$), ако

$$\mathcal{K}_1 \leqslant_t \mathcal{K}_2 \quad \text{и} \quad \mathcal{K}_2 \leqslant_t \mathcal{K}_1.$$

Записът $\mathcal{K}_1 \nleqslant_t \mathcal{K}_2$ ще означава, че $\mathcal{K}_1$ не се транслира в $\mathcal{K}_2$.

И едно последно понятие за *строга транслируемост* $<_t$, което се получава от нестрогата релация $\leqslant_t$ по обичайния начин:

$$\mathcal{K}_1 <_t \mathcal{K}_2 \quad \overset{\text{деф}}{\iff} \quad \mathcal{K}_1 \leqslant_t \mathcal{K}_2 \text{ и } \mathcal{K}_2 \nleqslant_t \mathcal{K}_1.$$

На картинката по-долу сме използвали следните означения за класове от схеми:

$\mathcal{S}$ — класа на всички стандартни схеми

$\mathcal{R}$ — класа на всички рекурсивни схеми

$\mathcal{S} + nc$ — класа на всички стандартни схеми с n брояча

$\mathcal{S} + ks$ — класа на всички стандартни схеми с k стека

$\mathcal{S} + nc + ks$ — класа на всички стандартни схеми с n брояча и k стека

$\mathcal{LS}$ — класа на всички логически схеми

Ето каква е връзката между тези класове:

$$\begin{array}{ccccccc}
 & & \mathcal{S} & \equiv_t & \mathcal{S}+1c & & \\
 & \wedge_t & & & \wedge_t & & \\
\mathcal{R} & \equiv_t & \mathcal{S}+1s & \substack{\nleqslant_t \\ \ngeqslant_t} & \mathcal{S}+2c & \equiv_t & \mathcal{S}+nc,\ n\geqslant 2 \\
 & & \wedge_t & & \wedge_t & & \\
\mathcal{S}+ns,\ n\geqslant 2 & \equiv_t & \mathcal{S}+2s & \equiv_t & \mathcal{S}+2c+1s & \equiv_t & \mathcal{LS}
\end{array}$$

максимални класове

4.3 Стандартни схеми

4.3.1 Синтаксис

Нашата цел до края на курса ще бъде да докажем неравенството $\mathcal{S} <_t \mathcal{R}$, което може да се изкаже накратко така: *рекурсията е по-мощна от итерацията.* Това включва две неща:

1) $\mathcal{S} \leqslant_t \mathcal{R}$ — *теорема на Маккарти*, която казва, че всяка стандартна схема се транслира в рекурсивна.

2) $\mathcal{R} \not\leqslant_t \mathcal{S}$ — *пример на Патерсън и Хюит*, който показва, че има рекурсивни схеми, които не могат да се преведат в стандартни.

За доказателствата ще ни трябва строга дефиниция на класа на стандартните схеми. Тези схеми могат да се въведат по най-различни (еквивалентни) начини. Тук ще разгледаме една стандартна дефиниция, която е подходяща за нашите цели.

Да фиксираме произволна сигнатура

$$\Sigma = (\mathbf{c}_1, \dots, \mathbf{c}_p;\ \mathbf{f}_1, \dots, \mathbf{f}_s;\ \mathbf{p}_1, \dots,\ \mathbf{p}_t),$$

в която $\mathbf{c}_1, \dots, \mathbf{c}_p$ са константни символи, $\mathbf{f}_1, \dots, \mathbf{f}_s$ — функционални символи, а $\mathbf{p}_1, \dots,\ \mathbf{p}_t$ — предикатни символи. Няма да разглеждаме функция за арност (местност), а ще считаме, че функционалните и предикатните символи вървят със своята местност.

Да фиксираме и една редица от *променливи* (или *регистри*) $X_1, X_2, \dots$.

Следва определението за *оператор* в сигнатурата Σ:

`оператор за присвояване` $::= X_i := X_j \mid X_i := \mathbf{c}_j \mid X_i := \mathbf{f}_j(X_{i_1}, \dots, X_{i_k})$

`оператор за преход` $::=$ `goto` l $\mid$ `if` $\mathbf{p}_j(X_{i_1}, \dots, X_{i_n})$ `then goto` l `else goto` l'

`оператор` $::=$ `оператор за присвояване` $\mid$ `оператор за преход` $\mid$ `stop`

Определение 4.4. *Стандартна схема* в сигнатурата Σ е синтактичен обект от следния вид:

$$
\begin{aligned}
S:\ & \texttt{input}(X_1,\dots,X_n);\ \texttt{output}(X_k)\\
& 1:\ O_1\\
& \vdots\\
& l:\ O_l\\
& \vdots\\
& q:\ O_q
\end{aligned}
$$

където $O_l, 1 \le l \le q$, е оператор в сигнатурата Σ.

Числото l ще наричаме *адрес* на оператора O_l. Ще искаме всички адреси, към които сочат операторите за преход, да са в интервала $[1, q]$, а последният оператор S_q да е `stop`.

Блок-схемата S от началото на тази глава (*Пример* 4.1), записана в този синтаксис, ще изглежда така:

$$
\begin{aligned}
S:\ & \texttt{input}(X);\ \texttt{output}(Y)\\
& 1:\ Y := \mathbf{c}\\
& 2:\ \texttt{if}\ \mathbf{p}(X)\quad \texttt{then goto}\ 6\quad \texttt{else goto}\ 3\\
& 3:\ Y := \mathbf{g}(X, Y)\\
& 4:\ X := \mathbf{f}(X)\\
& 5:\ \texttt{goto}\ 2\\
& 6:\ \texttt{stop}
\end{aligned}
$$

Ще казваме, че променливата X_i *участва* в схемата S, ако тя се среща в някой оператор на S. Нека

$$m \;=\; max\{i \mid X_i \text{ участва в } S\}.$$

$(X_1, \dots, X_m)$ ще наричаме *памет на* S. Ясно е, че броят n на входните променливи е по-малък или равен на m. Разбира се, не е задължително всички X_i, $i \le m$, да се срещат в схемата. Например в нея могат да участват само X_1, X_3 и X_5 (и тогава m ще е 5).

4.3.2 Семантика

Нека S е произволна стандартна схема в сигнатурата

$$\Sigma \;=\; (\mathbf{c}_1, \dots, \mathbf{c}_p;\ \mathbf{f}_1, \dots, \mathbf{f}_s;\ \mathbf{p}_1, \dots,\ \mathbf{p}_t).$$

За да въведем *семантиката* на S, трябва да фиксираме структура в тази сигнатура. Вече разгледахме няколко примера за структури; сега ще дадем официалната дефиниция.

Определение 4.5. *Структура* в сигнатурата Σ е обект от вида

$$\mathcal{A} \;=\; (D;\; c_1,\dots,c_p;\; f_1,\dots,f_s;\; p_1,\dots,p_t),$$

където:

- D е непразно множество — *универсум/носител* на $\mathcal{A}$;
- $c_1 \in D,\dots,c_p \in D$ са интерпретациите на константните символи $\mathbf{c}_1,\dots,\mathbf{c}_p$;
- за всяко $1 \le i \le s$, f_i е частична функция в D, с която се интерпретира функционалният символ $\mathbf{f_i}$ (като местността на f_i се определя от местността на $\mathbf{f_i}$);
- за всяко $1 \le i \le t$, p_i е предикат в D, с който се интерпретира предикатният символ $\mathbf{p_i}$ (като отново местността на p_i се определя от местността на $\mathbf{p_i}$).

Да вземем произволна стандартна схема S:

$S:$ $\mathtt{input}(X_1,\dots,X_n);$ $\mathtt{output}(X_k)$
$1:\; O_1$
$\vdots$
$q:\; O_q$

Схемата S в структурата $\mathcal{A}$ се превръща в *стандартна програма* $(S,\mathcal{A})$, пресмятаща частичната функция

$$\mathit{Sem}(S,\mathcal{A})\colon\; D^n \;\multimap\; D.$$

Да фиксираме и произволна структура $\mathcal{A}$ от сигнатурата на S. Ще дефинираме функцията $\mathit{Sem}(S,\mathcal{A})$ — *семантиката* на S в $\mathcal{A}$ в типичен операционен стил — чрез едностъпково преобразование, което ще итерираме дотогава, докато стигнем до оператора $\mathtt{stop}$ (точно както при машините на Тюринг се итерира δ-функцията на преходите, докато се достигне финално състояние).

За целта дефинираме *конфигурация* (или моментна снимка), която е наредена двойка от вида

$$(l,\;(x_1,\dots,x_m)),$$

където l е адресът на текущия оператор, който се изпълнява в дадения момент, а $(x_1,\dots,x_m) \in D^m$ е *текущото състояние на паметта* в този момент (като x_i е съдържанието на i-тия регистър X_i).

Начална конфигурация за входа $(x_1,\dots,x_n)$ е конфигурацията

$$(1,\;(x_1,\dots,x_n,\underbrace{x_n,\dots,x_n}_{m-n\text{ пъти}})).$$

Да отбележим, че регистрите, които не са входни, считаме, че имат стойност x_n при стартиране на програмата. Със същия успех бихме могли да приемем, че тази стойност е x_1 (обаче няма как да искаме да е 0, защото тя трябва да бъде елемент на D).

Заключителна/финална конфигурация е конфигурация от вида

$$(q,\ (y_1, \dots, y_m)).$$

По-нататък ще пишем $(l, x_1, \dots, x_m)$ вместо $(l,\ (x_1, \dots, x_m))$.

Да означим с $Q = \{1, \dots, q\}$ съвкупността от всички адреси на оператори на S. Ще дефинираме едностъпково преобразование *Step* (или *функция-стъпка*), което ще е изображение от вида:

$$Step : Q \times D^m \ \multimap\!\!\rightarrow\ Q \times D^m$$

Стойността на *Step* върху фиксирана конфигурация $(l, x_1, \dots, x_m)$ дефинираме с разглеждане на различните случаи за вида на оператора O_l както следва:

$$Step(l, x_1, \dots, x_m) \simeq \begin{cases} (l{+}1, x_1, ...x_{i-1}, x_j, x_{i+1}..., x_m), & \text{ако } O_l \text{ е } X_i := X_j \\ (l{+}1, x_1, ...x_{i-1}, c_j, x_{i+1}..., x_m), & \text{ако } O_l \text{ е } X_i := \mathbf{c}_j \\ (l{+}1, x_1, ... \underbrace{f_j(x_{i_1}, .., x_{i_k})}_{(i)} ..., x_m), & \text{ако } O_l \text{ е } X_i := \mathbf{f}_j(X_{i_1}, \dots, X_{i_k}) \\ (l', x_1, \dots, x_m), & \text{ако } O_l \text{ е } \texttt{goto } l' \\ (l', x_1, \dots, x_m), & \text{ако } O_l \text{ е } \texttt{if } \mathbf{p}_j(X_{i_1}, ..., X_{i_n}) \texttt{ then goto } l' \\ & \texttt{else goto } l'' \quad \& \quad p_j(x_{i_1}, ..., x_{i_n}) = \mathbf{t} \\ (l'', x_1, \dots, x_m), & \text{ако } O_l \text{ е } \texttt{if } \mathbf{p}_j(X_{i_1}, ..., X_{i_n}) \texttt{ then goto } l' \\ & \texttt{else goto } l'' \quad \& \quad p_j(x_{i_1}, ..., x_{i_n}) = \mathbf{f} \\ (l, x_1, \dots, x_m), & \text{ако } O_l \text{ е } \texttt{stop}. \end{cases}$$

Да напомним, че ако $f : M \ \multimap\!\!\rightarrow\ M$ е частична функция в M, то

$$f^n \stackrel{\text{деф}}{=} \underbrace{f \circ \cdots \circ f}_{n \text{ пъти}}.$$

Определение 4.6. *Семантиката* на програмата $(S, \mathcal{A})$ е функцията

$$Sem(S, \mathcal{A}) : D^n \ \multimap\!\!\rightarrow\ D,$$

която се дефинира с еквивалентността

$$Sem(S, \mathcal{A})(x_1, \dots, x_n) \simeq y \stackrel{\text{деф}}{\Longleftrightarrow} \exists t \exists y_1 \dots \exists y_m\ Step^t(1, x_1, ..., x_n, \underbrace{x_n, ..., x_n}_{m-n \text{ пъти}})$$
$$\simeq (q, y_1, \dots, y_m)\ \&\ y = y_k. \tag{4.1}$$

Понеже функцията $Sem(S, \mathcal{A})$ е дефинирана с условие, включващо квантори за съществуване, трябва да проверим, че тя е *еднозначна*.

Твърдение 4.1. (Коректност на дефиницията.)

$$Step^{t}(l, \bar{x}) \simeq (q, \bar{y}) \ \&\ Step^{t'}(l, \bar{x}) \simeq (q, \bar{y}') \quad \Longrightarrow \quad \bar{y} = \bar{y}'.$$

Доказателство. Ще използваме, че

$$f^{n+k}(x) \ \stackrel{\text{деф}}{\simeq}\ \underbrace{f \circ \cdots \circ f}_{n+k \text{ пъти}}(x) \ \simeq\ \underbrace{f \circ \cdots \circ f}_{n \text{ пъти}} \circ \underbrace{f \circ \cdots \circ f}_{k \text{ пъти}}(x) \ \simeq\ f^{n}(f^{k}(x)).$$

Без ограничение на общността можем да предполагаме, че $t' \geq t$. Тогава

$$Step^{t'}(l, \bar{x}) \ \simeq\ Step^{t'-t}(Step^{t}(l, \bar{x})) \ \simeq\ Step^{t'-t}(q, \bar{y}) \ \stackrel{\text{деф}}{\simeq}\ (q, \bar{y}),$$

и следователно $(q, \bar{y}) = (q', \bar{y}')$. $\square$