Semantyka Dużych Kroków

MIMUW 2018/19

Michał Szafraniuk

2 lutego 2019

1 Wstęp

Świat napisów: kategorie składniowe

Działamy na pięciu bazowych kategoriach składniowych:

(1) stałe liczbowe

$$n \in \mathit{Num}$$

ze składnią

$$n ::= 0 \mid 1 \mid 2 \mid ...$$

(2) zmienne

$$x \in \mathit{Var}$$

(3) wyrażenia arytmetyczne:

$$e \in \mathit{Expr}$$

ze składnią

$$e ::= n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2$$

(4) wyrażenia logiczne

$$b \in \mathit{BExpr}$$

ze składnią

$$b ::= \texttt{true} \mid \texttt{false} \mid e_1 \leqslant e_2 \mid \neg b' \mid b_1 \wedge b_2$$

(5) instrukcje

$$I \in \mathit{Instr}\ (S \in \mathit{Stmt})$$

ze składnią

$$I ::= x := e \mid \texttt{skip} \mid I_1; I_2 \mid \texttt{if}\ b\ \texttt{then}\ I_1\ \texttt{else}\ I_2 \mid \texttt{while}\ b\ \texttt{do}\ I$$

Ten repertuar będziemy poszerzać lub zawężać w zależności od siły wyrazu jezyków, które będziemy definiować. Kategoria instrukcji jest „główną” kategorią syntaktyczną i tą kategorię zazwyczaj będziemy zajmować się najwięcej. Wszystkie obiekty powyżej to napisy - np. symbol $\leqslant$ powyżej jest tutaj napisem (właściwszym byłby pewnie `<=`).

Świat znaczeń: kategorie semantyczne

(1) zbiór wartości wyrażeń arytmetycznych:

$$\mathsf{Val} = \mathbb{Z}_\perp = \mathbb{Z} \cup \{\perp\}$$

czyli zbiór liczb całkowitych rozszerzony o pewien wygodny obiekt służący do różnych rzeczy wedle potrzeb.

(2) zbiór wartości wyrażeń logicznych

$$\mathsf{Bool} = \{\mathbf{tt}, \mathbf{ff}\}$$

(3) *stany* mapujące zmienne na wartości:

$$s \in \mathsf{State} = Var \to \mathsf{Val}$$

Idea

W semantyce dużych kroków (*naturalnej*) - podobnie jak w semantyce małych kroków - do definiowania *znaczeń kategorii syntaktycznych* operujemy na zbiorach *konfiguracji*, wśród których wyróżniamy *konfiguracje końcowe* i sens składni danego języka (a przynajmniej tych bardziej istotnych kategorii składniowych) opisujemy zazwyczaj przy pomocy *systemu tranzycji.*

W semantyce naturalnej, zamiast pojedynczych obliczeń, które były rozważane w semantyce małych kroków, interesuje nas bezpośrednia relacja między początkową a końcową konfiguracją wykonania programu.

Niech Γ oznacza zbiór konfiguracji a $T \subset \Gamma$ zbiór konfiguracji końcowych. Wówczas interesuje nas *relacja przejścia*

$$\rightsquigarrow \subseteq \Gamma \times T$$

Najczęściej:

$$\Gamma = (Instr \times \mathsf{State}) \cup \underbrace{\mathsf{State}}_{T}$$

Przy pomocy tej relacji możemy opisać system tranzycji dla instrukcji z dwoma typami konfiguracji:

- $\langle I, s\rangle$: konfiguracja reprezentująca instrukcję I do wykonania w stanie s
- s: konfiguracja reprezentująca stan finalny s

Tranzycje opisujemy

$$\langle I, s\rangle \rightsquigarrow s'$$

Intuicyjnie:

- tworzymy zestaw reguł a relację $\rightsquigarrow$ traktujemy jako najmniejszą relację spełniającą te reguły (plus aksjomaty)
- jeśli dane obliczenie startujące w danej konfiguracji początkowej nie kończy się, to tego obliczenia wraz z tą kofniguracją nie powinno być w dziedzienie relacji $\rightsquigarrow$

2 Język TINY

2.1 Duże kroki dla wyrażeń

Chcemy zdefiniować znaczenie wyrażeń arytmetycznych oraz boolowskich w stylu dużych kroków. Wyrażenia rozszerzamy o dzielenie:

$$e ::= \mathtt{q} \mid \mathtt{x} \mid e_1 + e_2 \mid e_1 e_2 \mid e_1 * e_2 \mid e_1 / e_2$$

Używamy metazmiennej `q` zamiast `n` dla podkreślenia, że zbiorem wartości wyrażeń arytmetycznych jest $\mathbb{Q}$ a nie $\mathbb{Z}$.

Pierwszą rzeczą dp zrobienia jest zdefiniowanie konfiguracji i schematu tranzycji. „Zbiorem wartości” wyrażeń logicznych jest $\mathsf{Bool} = \{\mathbf{tt}, \mathbf{ff}\}$ a zbiorem wartości wyrażeń arytmetycznych jest $\mathsf{Val} = \mathbb{Q}$ (potrzebujemy liczb wymiernych bo dodaliśmy dzielenie). A zatem kofniguracje muszą wyglądać następująco:

- konfiguracje robocze/początkowe:

$$\Gamma_I = (\mathit{Expr} \cup \mathit{BExpr} \cup \mathit{Instr}) \times \mathsf{State}$$

- konfiguracje końcowe:

$$\Gamma_T = \mathsf{Val} \cup \mathsf{Bool} \cup \mathsf{State}$$

oraz

$$\mathsf{State} = \mathit{Var} \to \mathsf{Val}$$

Wyrażenia arytmetyczne

Reguły:

- dla stałych numerycznych

$$\frac{}{\langle \mathsf{q}, s\rangle \to q}$$

- dla zmiennych

$$\frac{}{\langle \mathtt{x}, s\rangle \to q} \quad \text{where} \quad q = s(\mathtt{x})$$

lub po prostu

$$\frac{}{\langle \mathtt{x}, s\rangle \to s(x)}$$

- dla sumy

$$\frac{\langle \mathsf{e}_1, s\rangle \to q_1 \qquad \langle \mathsf{e}_2, s\rangle \to q_2}{\langle \mathsf{e}_1 + \mathsf{e}_2, s\rangle \to q_1 + q_2}$$

- dla dzielenia

$$\frac{\langle \mathsf{e}_1, s\rangle \to q_1 \qquad \langle \mathsf{e}_2, s\rangle \to q_2}{\langle \mathsf{e}_1 / \mathsf{e}_2, s\rangle \to q_1 / q_2} \quad \text{if} \quad q_2 \neq 0$$

Zauważmy, że jeżeli $\langle e_2, s\rangle \to 0$ to program się „automatycznie” blokuje, gdyż takiej reguły nie ma a zatem nie ma odpowiadającego mu przejścia z konfiguracji początkowej do końcowej.

Wyrażenia boolowskie

Reguły:

- dla stałych boolowskich

$$\frac{}{\langle \texttt{true}, s\rangle \to \mathbf{tt}}$$

$$\frac{}{\langle \texttt{false}, s\rangle \to \mathbf{ff}}$$

- dla negacji

$$\frac{\langle \texttt{b}, s\rangle \to \mathbf{tt}}{\langle \neg\texttt{b}, s\rangle \to \mathbf{ff}}$$

$$\frac{\langle \texttt{b}, s\rangle \to \mathbf{ff}}{\langle \neg\texttt{b}, s\rangle \to \mathbf{tt}}$$

- dla nierówności

$$\frac{\langle \texttt{e}_1, s\rangle \to q_1 \qquad \langle \texttt{e}_2, s\rangle \to q_2}{\langle \texttt{e}_1 <= \texttt{e}_2, s\rangle \to \mathbf{tt}} \quad \text{if} \quad q_1 \leqslant q_2$$

$$\frac{\langle \texttt{e}_1, s\rangle \to q_1 \qquad \langle \texttt{e}_2, s\rangle \to q_2}{\langle \texttt{e}_1 <= \texttt{e}_2, s\rangle \to \mathbf{ff}} \quad \text{if} \quad q_1 > q_2$$

- dla koniunkcji
  - strategia gorliwa

$$\frac{\langle \texttt{b}_1, s\rangle \to b_1 \qquad \langle \texttt{b}_2, s\rangle \to b_2}{\langle \texttt{b}_1 \wedge \texttt{b}_2, s\rangle \to b} \quad \text{where} \quad b = b_1 \wedge b_2$$

  - strategia lewostronna

$$\frac{\langle \texttt{b}_1, s\rangle \to \mathbf{ff}}{\langle \texttt{b}_1 \wedge \texttt{b}_2, s\rangle \to \mathbf{ff}}$$

$$\frac{\langle \texttt{b}_1, s\rangle \to \mathbf{tt} \qquad \langle \texttt{b}_2, s\rangle \to b_2}{\langle \texttt{b}_1 \wedge \texttt{b}_2, s\rangle \to b_2}$$

  - strategia prawostronna

$$\frac{\langle \texttt{b}_2, s\rangle \to \mathbf{ff}}{\langle \texttt{b}_1 \wedge \texttt{b}_2, s\rangle \to \mathbf{ff}}$$

$$\frac{\langle \texttt{b}_1, s\rangle \to b_1 \qquad \langle \texttt{b}_2, s\rangle \to \mathbf{tt}}{\langle \texttt{b}_1 \wedge \texttt{b}_2, s\rangle \to b_1}$$

– strategia leniwa/równoległa

$$\frac{\langle \mathtt{b}_2, s\rangle \to \mathbf{ff}}{\langle \mathtt{b}_1 \wedge \mathtt{b}_2, s\rangle \to \mathbf{ff}}$$

$$\frac{\langle \mathtt{b}_1, s\rangle \to \mathbf{ff}}{\langle \mathtt{b}_1 \wedge \mathtt{b}_2, s\rangle \to \mathbf{ff}}$$

$$\frac{\langle \mathtt{b}_1, s\rangle \to \mathbf{tt} \qquad \langle \mathtt{b}_2, s\rangle \to \mathbf{tt}}{\langle \mathtt{b}_1 \wedge \mathtt{b}_2, s\rangle \to \mathbf{tt}}$$

Leniwe mnożenie

Rozszerzamy wyrażenia arytmetyczne o leniwe mnożenie

$$\mathtt{e} ::= ... \mid \mathtt{e}_1 \;\mathtt{lmul}\; \mathtt{e}_2$$

o znaczeniu takim, że napis `0 lmul e` ma się wyliczać do zera nawet wówczas, gdy `e` jest nieokreślone, czyli np `0 lmul (1/0)` ma się wyliczać do zera.

Reguły:

$$\frac{\langle \mathtt{e}_1, s\rangle \to 0}{\langle \mathtt{e}_1 \;\mathtt{lmul}\; \mathtt{e}_2, s\rangle \to 0}$$

$$\frac{\langle \mathtt{e}_1, s\rangle \to q_1 \qquad \langle \mathtt{e}_2, s\rangle \to q_2}{\langle \mathtt{e}_1 \;\mathtt{lmul}\; \mathtt{e}_2, s\rangle \to q_1 \cdot q_2} \quad \text{if} \quad q_1 \neq 0$$

Zauważmy, że

- jeśli w regule pierwszej dodalibyśmy

$$\frac{\langle \mathtt{e}_1, s\rangle \to 0 \qquad \langle \mathtt{e}_2, s\rangle \to q_2}{\langle \mathtt{e}_1 \;\mathtt{lmul}\; \mathtt{e}_2, s\rangle \to 0}$$

  to zepsulibyśmy docelowe znaczenie, gdyż program próbowałby wyliczać $\mathtt{e}_2$
- jeśli w regule drugiej usunęlibyśmy warunek $q_1 \neq 0$ to pojawiłyby się redundancje w dowodach ale wymagana semantyka by się nie zepsuła ani też nie pojawiłby się niedeterminizm.

2.2 Duże kroki dla instrukcji (czysty TINY)

Instrukcja `skip`

$$\frac{}{\langle \mathtt{skip}, s\rangle \to s}$$

Instrukcja przypisania

- w wersji pośredniej

$$\frac{}{\langle x := e, s\rangle \to s[x \mapsto \mathcal{E}[\![e]\!]s]}$$

- w wersji pełnych dużych kroków

$$\frac{\langle e, s\rangle \to q}{\langle x := e, s\rangle \to s[x \mapsto q]}$$

Instrukcja złożenia

$$\frac{\langle I_1, s\rangle \to s' \qquad \langle I_2, s'\rangle \to s''}{\langle I_1\,; I_2, s\rangle \to s''}$$

Instrukcja `if`

$$\frac{\langle I_1, s\rangle \to s'}{\langle \texttt{if}\ b\ \texttt{then}\ I_1\ \texttt{else}\ I_2, s\rangle \to s'} \quad \text{if} \quad \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

$$\frac{\langle I_2, s\rangle \to s'}{\langle \texttt{if}\ b\ \texttt{then}\ I_1\ \texttt{else}\ I_2, s\rangle \to s'} \quad \text{if} \quad \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

Instrukcja `while`

$$\frac{\langle I, s\rangle \to s' \qquad \langle \texttt{while}\ b\ \texttt{do}\ I, s'\rangle \to s''}{\langle \texttt{while}\ b\ \texttt{do}\ I, s\rangle \to s''} \quad \text{if} \quad \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

$$\langle \texttt{while}\ b\ \texttt{do}\ I, s\rangle \to s \quad \text{if} \quad \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

2.3 TINY + repeat, for

Instrukcja `repeat`

$$\frac{\langle I, s\rangle \to s'}{\langle \texttt{repeat}\ I\ \texttt{until}\ b, s\rangle \to s'} \quad \text{if} \quad \mathcal{B}[\![b]\!]s' = \mathbf{tt}$$

$$\frac{\langle I, s\rangle \to s' \qquad \langle \texttt{repeat}\ I\ \texttt{until}\ b, s'\rangle \to s''}{\langle \texttt{repeat}\ I\ \texttt{until}\ b, s\rangle \to s''} \quad \text{if} \quad \mathcal{B}[\![b]\!]s' = \mathbf{ff}$$

Instrukcja `for`

$$\frac{\langle \texttt{x} := e_1, s\rangle \to s'}{\langle \texttt{for}\ \texttt{x} := e_1\ \texttt{to}\ e_2\ \texttt{do}\ I, s\rangle \to s} \quad \text{if} \quad \mathcal{B}[\![\texttt{x} \leqslant e_2]\!]s' = \mathbf{ff}$$

$$\frac{\langle \texttt{x} := e_1, s\rangle \to s' \qquad \langle I; \texttt{for}\ \texttt{x} := \texttt{x} + \texttt{1}\ \texttt{to}\ e_2\ \texttt{do}\ I, s'\rangle \to s''}{\langle \texttt{for}\ \texttt{x} := e_1\ \texttt{to}\ e_2\ \texttt{do}\ I, s\rangle \to s''} \quad \text{if} \quad \mathcal{B}[\![\texttt{x} \leqslant e_2]\!]s' = \mathbf{tt}$$

2.4 Duże kroki dla wyrażeń z propagacją błędu

W poprzednim paragrafie dzielenie przez zero po prostu „zawieszało" program, ponieważ nie zdefiniowaliśmy reguł dla błędu dzielenia przez zero. Teraz chcemy aby w przypadku wystąpienia takiego błędu program w kontrolowany sposób zakończył się (natychmiast po wystąpieniu błędu) zwracając swój stan oraz informację o błędzie.

Konfiguracje :

- konfiguracje początkowe bez zmian:

$$\Gamma_I = (\mathit{Expr} \cup \mathit{BExpr} \cup \mathit{Instr}) \times \mathsf{State}$$

- konfiguracje końcowe: Musimy je rozszerzyć o sytuację wystąpienia błędu - w takiej sytuacji chcemy zwrócić flagę błędu $\perp$ oraz stan. A zatem rozszerzamy poprzenie konfiguracje końcowe o zbiór $\{\perp\} \times \mathsf{State}$

$$\Gamma_T = \mathsf{Val} \cup \mathsf{Bool} \cup \mathsf{State} \cup (\{\perp\} \times \mathsf{State})$$

Reguły:

- powstanie błędu:

$$\frac{\langle \mathtt{e}_2, s\rangle \to 0}{\langle \mathtt{e}_1/\mathtt{e}_2, s\rangle \to \langle \perp, s\rangle}$$

- propagacja błędu:
  - nierówność

$$\frac{\langle \mathtt{e}_1, s\rangle \to \langle \perp, s\rangle}{\langle \mathtt{e}_1 <= \mathtt{e}_2, s\rangle \to \langle \perp, s\rangle}$$

$$\frac{\langle \mathtt{e}_2, s\rangle \to \langle \perp, s\rangle}{\langle \mathtt{e}_1 <= \mathtt{e}_2, s\rangle \to \langle \perp, s\rangle}$$

  - `if`
    - wstrzymianie wykonania:

$$\frac{\langle \mathtt{b}, s\rangle \to \langle \perp, s\rangle}{\langle \mathtt{if}\ b\ \mathtt{then}\ I_1\ \mathtt{else}\ I_2, s\rangle \to \langle \perp, s\rangle}$$

    - propagacja:

$$\frac{\langle \mathtt{b}, s\rangle \to \mathbf{tt} \qquad \langle I_1, s\rangle \to \langle \perp, s\rangle}{\langle \mathtt{if}\ b\ \mathtt{then}\ I_1\ \mathtt{else}\ I_2, s\rangle \to \langle \perp, s\rangle}$$

$$\frac{\langle \mathtt{b}, s\rangle \to \mathbf{ff} \qquad \langle I_2, s\rangle \to \langle \perp, s\rangle}{\langle \mathtt{if}\ b\ \mathtt{then}\ I_1\ \mathtt{else}\ I_2, s\rangle \to \langle \perp, s\rangle}$$

  - przypisanie

$$\frac{\langle \mathtt{e}, s\rangle \to \langle \perp, s\rangle}{\langle \mathtt{x} := \mathtt{e}, s\rangle \to \langle \perp, s\rangle}$$

  - `while`
    - wstrzymanie

$$\frac{\langle \mathtt{b}, s\rangle \to \langle \perp, s\rangle}{\langle \mathtt{while\ b\ do\ I}, s\rangle \to \langle \perp, s\rangle}$$

    - propagacja

$$\frac{\langle \mathtt{b}, s\rangle \to \mathbf{tt} \qquad \langle I, s\rangle \to \langle \perp, s\rangle}{\langle \mathtt{while\ b\ do\ I}, s\rangle \to \langle \perp, s\rangle}$$

$$\frac{\langle \mathtt{b}, s\rangle \to \mathbf{tt} \qquad \langle I, s\rangle \to s' \qquad \langle \mathtt{while\ b\ do\ I}, s'\rangle \to \langle \perp, s\rangle}{\langle \mathtt{while\ b\ do\ I}, s\rangle \to \langle \perp, s\rangle}$$

      Ostatnia reguła jest potrzebna, bo obsługuje przypadek, gdy błąd generuje się w co najmniej drugiej iteracji pętli - pierwsza reguła jest przypadkiem bazowym a druga krokiem indukcyjnym. Nie musimy dodawać reguł dla sytuacji gdy dozór wylicza się do fałszu bo taka pętla nigdy się nie odpali a więc nie ma możliwości wystąpienia w niej błędu.

2.5 TINY + loop

Rozszerzamy kategorię instrukcji o

$$I ::= \texttt{loop}\ I \mid \texttt{exit} \mid \texttt{continue}$$

Ponieważ `exit` oraz `continue` skaczą odpowiednio do pierwszej instrukcji poza pętlą oraz do pierwszej instrukcji wewnątrz pętli to aby zrealizować te skoki potrzebujemy rozszerzyć zbiór konfiguracji o znaczniki, które będą „zostawiać ślad” o tym czy napotkana została któraś z powyższych instrukcyj generująca „skok”.

Konfiguracje :

- konfiguracje początkowe bez zmian:

$$\Gamma_I = \mathit{Instr} \times \mathsf{State}$$

- konfiguracje końcowe:

$$\Gamma_T = \mathsf{State} \cup \mathsf{State} \times \{\bot, \top\}$$

  z interpretacją: $\bot$ - ślad po `exit`, $\top$ - ślad po `continue`.

Definiujemy reguły

- reguły zostawiające ślad:

$$\langle \texttt{exit}, s\rangle \to \langle s, \bot\rangle$$

$$\langle \texttt{continue}, s\rangle \to \langle s, \top\rangle$$

- reguły dla `loop`:
  - „czysty” `loop` pętli się normalnie po ostatniej wewnętrzenej instrukcji pętli:

$$\frac{\langle I, s\rangle \to s' \qquad \langle \texttt{loop}\ I, s'\rangle \to s''}{\langle \texttt{loop}\ I\ , s\rangle \to s''}$$

  - po napotkaniu `exit` wychodzimy z pętli w stanie, który osiągnęliśmy do tego napotkania i kasujemy znacznik:

$$\frac{\langle I, s\rangle \to \langle s', \bot\rangle}{\langle \texttt{loop}\ I\ , s\rangle \to s'}$$

  - po napotkaniu `continue` zawijamy się na początek pętli ze stanem, który osiągneliśmy do tego napotkania i kasujemy znacznik:

$$\frac{\langle I, s\rangle \to \langle s', \top\rangle \qquad \langle \texttt{loop}\ I, s'\rangle \to s''}{\langle \texttt{loop}\ I\ , s\rangle \to s''}$$

- pozostałe reguły w sytuacjach, gdy w konfiguracjach wyskakuje znacznik, muszą zagwarantować ignorowanie wykonania kolejnych instrukcji:

– złożenie:

$$\frac{\langle I_1, s\rangle \to \langle s', \bot\rangle}{\langle I_1; I_2, s\rangle \to \langle s', \bot\rangle}$$

$$\frac{\langle I_1, s\rangle \to s' \qquad \langle I_2, s'\rangle \to \langle s'', \bot\rangle}{\langle I_1; I_2, s\rangle \to \langle s'', \bot\rangle}$$

– `if`:

$$\frac{\langle I_1, s\rangle \to \langle s', \bot\rangle}{\langle \texttt{if}\, b\, \texttt{then}\, I_1\, \texttt{else}\, I_2, s\rangle \to \langle s', \bot\rangle} \quad \text{if} \quad \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

$$\frac{\langle I_2, s\rangle \to \langle s', \bot\rangle}{\langle \texttt{if}\, b\, \texttt{then}\, I_1\, \texttt{else}\, I_2, s\rangle \to \langle s', \bot\rangle} \quad \text{if} \quad \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

– `while`:

$$\frac{\langle I, s\rangle \to \langle s', \bot\rangle}{\langle \texttt{while}\, b\, \texttt{do}\, I, s\rangle \to \langle s', \bot\rangle} \quad \text{if} \quad \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

$$\frac{\langle I, s\rangle \to s' \qquad \langle \texttt{while}\, b\, \texttt{do}\, I, s'\rangle \to \langle s'', \bot\rangle}{\langle \texttt{while}\, b\, \texttt{do}\, I, s\rangle \to \langle s'', \bot\rangle} \quad \text{if} \quad \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

- oraz analogiczne reguły dla $\top$

2.6 TINY + loop z etykietami

Rozszerzamy kategorię instrukcji o *etykietowaną* pętlę `loop`:

$$I ::= \texttt{x} : \ \texttt{loop}\, I \mid \texttt{exit x} \mid \texttt{continue x}$$

z intuicyjnym znaczeniem: `exit x / continue x` kończy/wznawia najbliższą otaczającą pętlę `loop` o etykiecie `x`.

W porównaniu z poprzednim zadaniem tym razem musimy nie tylko rozpoznawać sytuacją wystąpienia `exit/continue` ale także etykietę.

To prowadzi do następującego sposobu zdefiniowania konfiguracji :

- konfiguracje początkowe bez zmian:

$$\Gamma_I = \mathit{Instr} \times \mathsf{State}$$

- konfiguracje końcowe:

$$\Gamma_T = \mathsf{State} \cup \mathsf{State} \times \{\bot, \top\} \times \mathit{Var}$$

z interpretacją: $\bot$ - ślad po `exit`, $\top$ - ślad po `continue`.

Definiujemy reguły

- reguły zostawiające ślad:

$$\langle \texttt{exit x}, s\rangle \to \langle s, \bot, \texttt{x}\rangle$$

$$\langle \texttt{continue x}, s\rangle \to \langle s, \top, \texttt{x}\rangle$$

- reguły dla `loop`: Tym razem zamiast trzech przypadków mamy pięć:

  – „czysty” `loop`: standardowo

$$\frac{\langle I, s\rangle \to s' \qquad \langle \mathtt{x:\ loop\,I}, s'\rangle \to s''}{\langle \mathtt{x:\ loop\,I}, s\rangle \to s''}$$

  – „swój” `exit` (tzn. o tej samej etykiecie co `loop`): standardowo wychodzimy z pętli w stanie, który osiągnęliśmy do tego napotkania i kasujemy znacznik:

$$\frac{\langle I, s\rangle \to \langle s', \bot, \mathtt{x}\rangle}{\langle \mathtt{x:\ loop\,I}, s\rangle \to s'}$$

  – „swój” `continue`: zawijamy się na początek pętli ze stanem, który osiągneliśmy do tego napotkania i kasujemy znacznik:

$$\frac{\langle \mathtt{I}, s\rangle \to \langle s', \top, \mathtt{x}\rangle \qquad \langle \mathtt{x:\ loop\,I}, s'\rangle \to s''}{\langle \mathtt{x:\ loop\,I}, s\rangle \to s''}$$

  – „obcy” `exit`: propagujemy dalej

$$\frac{\langle I, s\rangle \to \langle s', \bot, \mathtt{y}\rangle}{\langle \mathtt{x:\ loop\,I}, s\rangle \to \langle s', \bot, \mathtt{y}\rangle} \quad \text{if} \quad \mathtt{x} \neq \mathtt{y}$$

  – „obcy” `continue`: propagujemy dalej

$$\frac{\langle I, s\rangle \to \langle s', \top, \mathtt{y}\rangle}{\langle \mathtt{x:\ loop\,I}, s\rangle \to \langle s', \top, \mathtt{y}\rangle} \quad \text{if} \quad \mathtt{x} \neq \mathtt{y}$$

  Założyliśmy tu *implicite*, że dysponujemy relacją równości/nierówności na *Var*.

- pododbnie jak poprzednio pozostałe reguły w sytuacjach, gdy w konfiguracjach wyskakuje znacznik, muszą zagwarantować ignorowanie wykonania kolejnych instrukcji:

  – złożenie:

$$\frac{\langle I_1, s\rangle \to \langle s', \bot, \mathtt{x}\rangle}{\langle I_1; I_2, s\rangle \to \langle s', \bot, \mathtt{x}\rangle}$$

$$\frac{\langle I_1, s\rangle \to s' \qquad \langle I_2, s'\rangle \to \langle s'', \bot, \mathtt{x}\rangle}{\langle I_1; I_2, s\rangle \to \langle s'', \bot, \mathtt{x}\rangle}$$

  – `if`:

$$\frac{\langle I_1, s\rangle \to \langle s', \bot, \mathtt{x}\rangle}{\langle \mathtt{if}\ b\ \mathtt{then}\ I_1\ \mathtt{else}\ I_2, s\rangle \to \langle s', \bot, \mathtt{x}\rangle} \quad \text{if} \quad \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

$$\frac{\langle I_2, s\rangle \to \langle s', \bot, \mathtt{x}\rangle}{\langle \mathtt{if}\ b\ \mathtt{then}\ I_1\ \mathtt{else}\ I_2, s\rangle \to \langle s', \bot, \mathtt{x}\rangle} \quad \text{if} \quad \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

– `while`:

$$\frac{\langle I, s\rangle \to \langle s', \bot, \mathtt{x}\rangle}{\langle \mathtt{while}\, b\, \mathtt{do}\, I, s\rangle \to \langle s', \bot, \mathtt{x}\rangle} \quad \text{if} \quad \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

$$\frac{\langle I, s\rangle \to s' \qquad \langle \mathtt{while}\, b\, \mathtt{do}\, I, s'\rangle \to \langle s'', \bot, \mathtt{x}\rangle}{\langle \mathtt{while}\, b\, \mathtt{do}\, I, s\rangle \to \langle s'', \bot, \mathtt{x}\rangle} \quad \text{if} \quad \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

- oraz analogiczne reguły dla $\top$

2.7 TINY + efekty uboczne wyrażeń

Rozszerzamy kategorię wyrażeń o *efekty uboczne*:

$$e ::= \ldots \mid \mathtt{do\ I\ then\ e} \mid \mathtt{x} ::= \mathtt{e} \mid \mathtt{x}++$$

o następującym znaczeniu

- `do I then e` najpierw wykonuje `I` jako efekt uboczny a potem oblicza `e`
- `x` ::= `e` oblicza się do `e` a efektem ubocznym jest podstawienie
- `x` + + oblicza się do `x` a efektem ubocznym jest zwiększenie `x` o jeden

Efekty uboczne wewnątrz wyrażeń powodują *zmianę stanu*: dotychczas wyrażenia takiej mocy nie posiadały - stan w którym wyliczało się obliczenie pozostawał bez zmian przez cały proces wyliczania się wyrażenia. A zatem w konfiguracjach końcowych nie wystarczy już samo Val - potrzebujemy dodać zbiór $\mathsf{State} \times \mathsf{Val}$. Ale to nie wszystko: wyrażenia boolowskie takie $\mathtt{e}_1 <= \mathtt{e}_2$ także - „dziedzicząc” po wyrażeniach arytmetycznych - muszą obsługiwać zmienę stanu. Więc moglibyśmy położyć tak:

$$\Gamma_T = \mathsf{State} \cup \mathsf{Bool} \cup \mathsf{Val} \cup \mathsf{State} \times \mathsf{Bool} \cup \mathsf{State} \times \mathsf{Bool}$$

i napisać reguły dla tranzycji zwracających niepary lub pary.

Ale łatwiej tak:

- konfiguracje początkowe bez zmian:

$$\Gamma_I = \mathit{Instr} \times \mathsf{State}$$

- konfiguracje końcowe:

$$\Gamma_T = \mathsf{State} \times (\mathsf{Val} \cup \mathsf{Bool} \cup \{\bot\})$$

gdzie $\bot$ symbolizuje konfigurację, która nie niesie ze sobą wartości obliczenia.

Reguły:

- **dla wyrażeń arytmetycznych**
  – dla stałych numerycznych

$$\frac{}{\langle \mathsf{q}, s\rangle \to \langle s, q\rangle}$$

– dla zmiennych

$$\frac{}{\langle \mathtt{x}, s\rangle \to \langle s, s(x)\rangle}$$

– dla sumy

$$\frac{\langle \mathtt{e}_1, s\rangle \to \langle s', n_1\rangle \qquad \langle \mathtt{e}_2, s'\rangle \to \langle s'', q_2\rangle}{\langle \mathtt{e}_1 + \mathtt{e}_2, s\rangle \to \langle s'', q_1 + q_2\rangle}$$

– dla `do`

$$\frac{\langle \mathtt{I}, s\rangle \to \langle s', \bot\rangle \qquad \langle \mathtt{e}, s'\rangle \to \langle s'', n\rangle}{\langle \mathtt{do\ I\ then\ e}, s\rangle \to \langle s'', n\rangle}$$

Najpierw wykonujemy `I` jako efekt uboczny i przechodzimy do nowego stanu. W tym nowym stanie wykonujemy właściwe obliczenie, być może również z efektami ubocznymi i przechodzimy do jeszcze innego stanu.

– dla ubocznego przypisania

$$\frac{\langle \mathtt{e}, s\rangle \to \langle s', n\rangle}{\langle \mathtt{x} ::= \mathtt{e}, s\rangle \to \langle s'[x \mapsto n], n\rangle}$$

– dla plusplusa:

$$\frac{}{\langle \mathtt{x}++, s\rangle \to \langle s[x \mapsto s(x)+1], s(x)\rangle}$$

- **dla wyrażeń boolowskich**

  – dla stałych boolowskich, negacji i koniunkcji: nie występują efekty uboczne więc jedyne co się zmienia to, że w konfiguracjach końcowych pojawia się stan

  – dla nierówności, np (różne możliwości):

$$\frac{\langle \mathtt{e}_1, s\rangle \to \langle s', n_1\rangle \qquad \langle \mathtt{e}_2, s'\rangle \to \langle s'', n_2\rangle}{\langle \mathtt{e}_1 <= \mathtt{e}_2, s\rangle \to \langle s'', \mathbf{tt}\rangle} \quad \text{if} \quad q_1 \leqslant q_2$$

$$\frac{\langle \mathtt{e}_1, s\rangle \to \langle s', n_1\rangle \qquad \langle \mathtt{e}_2, s'\rangle \to \langle s'', n_2\rangle}{\langle \mathtt{e}_1 <= \mathtt{e}_2, s\rangle \to \langle s'', \mathbf{ff}\rangle} \quad \text{if} \quad q_1 > q_2$$

- **dla instrukcji**

  Konfiguracje finalne dla instrukcji nie niosą ze sobą żadnego obliczenia.

  – zwykłe przypisanie

$$\frac{\langle e, s\rangle \to \langle s', n\rangle}{\langle x := e, s\rangle \to \langle s'[x \mapsto n], \bot\rangle}$$

  – złożenie

$$\frac{\langle I_1, s\rangle \to \langle s', \bot\rangle \qquad \langle I_2, s'\rangle \to \langle s'', \bot\rangle}{\langle I_1 ; I_2, s\rangle \to \langle s'', \bot\rangle}$$

– `if`

$$\frac{\langle b, s\rangle \to \langle s', \mathbf{tt}\rangle \qquad \langle I_1, s'\rangle \to \langle s'', \bot\rangle}{\langle \texttt{if}\ b\ \texttt{then}\ I_1\ \texttt{else}\ I_2, s\rangle \to \langle s', \bot\rangle}$$

$$\frac{\langle b, s\rangle \to \langle s', \mathbf{ff}\rangle \qquad \langle I_2, s'\rangle \to \langle s'', \bot\rangle}{\langle \texttt{if}\ b\ \texttt{then}\ I_1\ \texttt{else}\ I_2, s\rangle \to \langle s', \bot\rangle}$$

– `while`

$$\frac{\langle b, s\rangle \to \langle s', \mathbf{tt}\rangle \qquad \langle I, s'\rangle \to \langle s'', \bot\rangle \qquad \langle \texttt{while}\ b\ \texttt{do}\ I, s''\rangle \to s'''}{\langle \texttt{while}\ b\ \texttt{do}\ I, s\rangle \to s'''}$$

$$\frac{\langle b, s\rangle \to \langle s', \mathbf{ff}\rangle}{\langle \texttt{while}\ b\ \texttt{do}\ I, s\rangle \to s'}$$

2.8 TINY + dziwny for

Rozszerzamy kategorię intrukcji następująco:

$$I ::= ... \mid \texttt{for}\ x = e_1\ \texttt{to}\ e_2\ \texttt{try}\ I_1 \texttt{else}\ I_2 \mid \texttt{fail}$$

Pętla `for` ma następujące znaczenie:

- najpierw oblicz e_1 oraz e_2
- jeśli $e_1 > e_2$ to przywróć wszystkie wartości sprzed rozpoczęcia `for`'a i wykonaj I_2
- jeśli $e_1 \leqslant e_2$ to przypisz e_1 na x i wykonaj I_1. Jeśli wewnątrz I_1 napotkany został `fail` to zwiększ x o jeden, przywróć wszystkie pozostałe zmienne oraz ponownie wykonaj I_1. Jeśli nie napotkano `fail` to zakończ `for`'a przywracając wartość x sprzed (ale reszty zmiennych nie przywracając)

Oczywiście potrzebujemy flagi do oznaczania sytuacji, w której wystąpił `fail`. Ale takie zdarzenie powoduje przywrócenie wartości zmiennych sprzed `for`'a oraz zwiększenie x o jeden - to bardzo upraszcza nam konfiguracje końcowe:

- konfiguracje początkowe:

$$\Gamma_I = (\mathit{Instr} \cup \mathit{Expr}) \times \mathsf{State}$$

- konfiguracje końcowe:

$$\Gamma_T = \mathsf{State} \cup \mathsf{Val} \cup \mathsf{Bool} \cup \{\bot\}$$

Reguły:

- dozór w pętli fałszywy

  Przywracamy wszystko, wykonujemy I_2:

$$\frac{\langle e_1, s\rangle \to n_1 \qquad \langle e_2, s\rangle \to n_2 \qquad \langle I_2, s\rangle \to s'}{\langle \texttt{for}\ x = e_1\ \texttt{to}\ e_2\ \texttt{try}\ I_1 \texttt{else}\ I_2, s\rangle \to s'} \quad \text{if} \quad n_1 > n_2$$

- dozór prawdziwy, wykonujemy I_1, bez `fail` w I_1

  Wychodzimy z pętli przywracając x sprzed pętli:

  $$\frac{\langle e_1, s\rangle \to n_1 \qquad \langle e_2, s\rangle \to n_2 \qquad \langle I_1, s[x \mapsto n_1]\rangle \to s'}{\langle \texttt{for}\; x = e_1 \;\texttt{to}\; e_2 \;\texttt{try}\; I_1 \texttt{else}\; I_2, s\rangle \to s'[x \mapsto s(x)]} \quad \text{if} \quad n_1 \leqslant n_2$$

- dozór prawdziwy, wykonujemy I_1, pojawia się `fail` w I_1

  Generowanie flagi:

  $$\frac{}{\langle \texttt{fail}, s\rangle \to \bot}$$

  Jak widać, konfiugarcja końcowa nie ma stanu - pozwala nam na to specyficzne znaczenie rozpatrywanego `for`'a gdyż:

  $$\frac{\langle e_1, s\rangle \to n_1 \qquad \langle e_2, s\rangle \to n_2 \qquad \langle I_1, s[x \mapsto n_1]\rangle \to \bot \qquad \langle \texttt{for}\; x = n_1 + 1 \;\texttt{to}\; n_2 \;\texttt{try}\; I_1 \texttt{else}\; I_2, s\rangle \to s'}{\langle \texttt{for}\; x = e_1 \;\texttt{to}\; e_2 \;\texttt{try}\; I_1 \texttt{else}\; I_2, s\rangle \to s'}$$

  $$\text{if} \quad n_1 \leqslant n_2$$

- pozostałe reguły powinny zapewnić odpowiednie przerywanie działania analogicznie jak w przykładach wyżej, przykładowa reguła dla `while`'a:

  $$\frac{\langle \texttt{b}, s\rangle \to \texttt{tt} \qquad \langle I, s\rangle \to \bot}{\langle \texttt{while b do I}, s\rangle \to \bot}$$

  $$\frac{\langle \texttt{b}, s\rangle \to \texttt{tt} \qquad \langle I, s\rangle \to s' \qquad \langle \texttt{while b do I}, s'\rangle \to \bot}{\langle \texttt{while b do I}, s\rangle \to \bot}$$

3 Kalkulatory

W tej sekcji rozważamy proste „języki wyrażeń": tj. języki ze stałymi, zmiennymi oraz wyrażeniami arytmetycznymi ale bez instrukcji (i bez wyrażeń boolowskich).

3.1 Kalkulator gorliwy

Rozważmy następujący, prosty „język wyrażeń" (bez instrukcji):

$$e ::= n \mid x \mid e_1 + e_2 \mid \texttt{if}\; e_1 \;\texttt{then}\; e_2 \;\texttt{else}\; e_3 \mid \texttt{let x} = e_1 \;\texttt{in}\; e_2$$

Semantyka `if`'a ma działać następująco: jeśli e_1 wylicza się do nie-zera to wartością wyrażenia jest wartość e_2, wpp wartością jest wartość e_3.

Semantyka `let`'a ma działać następująco:

1. gorliwie (od razu) wylicz e_1
2. podstaw wyliczoną wartość pod `x`
3. wylicz e_2

Odwołania do zmiennej w e_2 dotyczą najbardziej zagnieżdżonej inicjalizacji tej zmiennej, np. przy obliczaniu e_3 w wyrażeniu

$$\texttt{let x} = e_1 \texttt{ in let x} = e_2 \texttt{ in } e_3$$

deklaracja $\texttt{x} = e_2$ przesłania $\texttt{x} = e_1$.

Konfiguracje w tym przypadku mają postać:

$$\Gamma = \mathit{Expr} \times \mathsf{State} \cup \underbrace{\mathsf{Val}}_{T}$$

a stany to

$$\mathsf{State} = \mathit{Var} \to \mathsf{Val}$$

Tranzycje są postaci:

$$\langle e, s \rangle \to n$$

Reguły:

- dla numerałów:

$$\langle n, s \rangle \to n$$

- dla zmiennych:

$$\langle x, s \rangle \to n \quad \text{where} \quad n = s(x)$$

- dla sumy wyrażeń:

$$\frac{\langle e_1, s \rangle \to n_1 \qquad \langle e_2, s \rangle \to n_2}{\langle e_1 + e_2, s \rangle \to n} \quad \text{where} \quad n = n_1 + n_2$$

- dla `if`'a:

$$\frac{\langle e_1, s \rangle \to n_1 \qquad n_1 \neq 0 \qquad \langle e_2, s \rangle \to n_2}{\langle \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, s \rangle \to n_2}$$

$$\frac{\langle e_1, s \rangle \to n_1 \qquad n_1 = 0 \qquad \langle e_3, s \rangle \to n_3}{\langle \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, s \rangle \to n_3}$$

- dla `let`'a:

$$\frac{\langle e_1, s \rangle \to n_1 \qquad \langle e_2, s[x \mapsto n_1] \rangle \to n_2}{\langle \texttt{let x} = e_1 \texttt{ in } e_2, s \rangle \to n_2}$$

3.2 Kalkulator leniwy

Rozważamy język jak poprzednio, z tą różnicą, iż chcemy aby wyliczanie wyrażeń odbywało się *leniwie*: wyrażenie e_1 w $\texttt{let x} = e_1 \texttt{in} e_2$ ma się wyliczać dopiero wówczas, gdy będzie naprawdę potrzebne a nie natychmiast. Na przykład przy obliczaniu e w wyrażeniu

$$\texttt{let x} = 7 \texttt{ in let y} = \underbrace{2 + y}_{e} \texttt{ in } \underbrace{x + x}_{\mathsf{e'}}$$

zmienna y jest niezainicjowana co może prowadzić do pewnych komplikacji (np. w pustym stanie początkowym) ale ponieważ `y` nie jest wykorzystana przy obliczeniu `e'` to przy leniwym podejściu nie powinno to rodzić kłopotów.

Jeżeli chcemy opóźnić wyliczanie wyrażeń to musimy zmodyfikować definicję stanów, gdyż musimy „zapamiętać" dwie rzeczy: nieobliczone wyrażenie oraz stan w którym to wyrażenie pierwotnie miało się wyliczyć. Chcemy więc mieć coś w rodzaju:

$$\mathsf{State} = \mathit{Var} \to \mathit{Expr} \times \mathsf{State}$$

Aby uniknąć definicji *idem per idem* możemy wprowadzić:

$$\mathsf{State}_0 = \{\varnothing\}$$

$$\mathsf{State}_{n+1} = \mathit{Var} \to \mathit{Expr} \times \mathsf{State}_n$$

$$\mathsf{State} = \bigcup_{n \in \mathbb{N}} \mathsf{State}_n$$

Konfiguracje pozostają bez zmian:

$$\Gamma = \mathit{Expr} \times \mathsf{State} \cup \underbrace{\mathsf{Val}}_{T}$$

czyli tranzycje również:

$$\langle e, s \rangle \to n$$

Reguły:

- dla zmiennych:
$$\frac{\langle e, s' \rangle \to n}{\langle x, s \rangle \to n} \quad \text{if} \quad s(x) = \langle e, s' \rangle$$

- dla `let`'a:
$$\frac{\langle e_2, s[x \mapsto \langle e_1, s \rangle] \rangle \to n}{\langle \mathtt{let}\,\mathtt{x} = e_1\,\mathtt{in}\,e_2, s \rangle \to n}$$

Pozostałe reguły pozostają bez zmian.
Przykład:

$$\cfrac{\cfrac{\cfrac{\langle x, \hat{s} \rangle \to 7 \qquad \langle x, \hat{s} \rangle \to 7}{\langle x + x, \underbrace{s[y \mapsto \langle y + y, s[x \mapsto \langle 7, s \rangle] \rangle]}_{\hat{s}} \rangle \to 14}}{\langle \mathtt{let}\,\mathtt{y} = 2 + y\,\mathtt{in}\,\mathtt{x} + \mathtt{x}, s[x \mapsto \langle 7, s \rangle] \rangle \to 14}}{\langle \mathtt{let}\,\mathtt{x} = 7\,\mathtt{in}\,\mathtt{let}\,\mathtt{y} = 2 + y\,\mathtt{in}\,\mathtt{x} + \mathtt{x} \rangle \to 14}$$

gdyż

$$\hat{s}(x) = s[y \mapsto \langle y + y, s[x \mapsto \langle 7, s \rangle] \rangle](x) = \langle 7, s \rangle$$

3.3 Kalkulator z dynamicznym wiązaniem zmiennych

Modyfikujemy poprzeni język wprowadzając *dynamiczne wiązanie zmiennych*: odwołanie do zmiennej odnosimy nie do stanu *w momencie deklaracji* tej zmiennej ale do stanu *w momencie odwołania* do tej zmiennej.

Przykładowo, przy wiązaniu dynamicznym wyrażenie

$$\mathtt{let}\,\mathtt{y} = \mathtt{x} + 1\,\mathtt{in}\,\mathtt{let}\,\mathtt{x} = 10\,\mathtt{in}\,\mathtt{y}$$

powinno wyliczyć się do 11, gdyż w momencie odwołania do `y` mamy $s(x) = 10$. Przy wiązaniu statycznym i pustym stanie początkowym to wyrażenie nie zakończy się poprawnie bo będziemy odwoływać się do niezadeklarowanej zmiennej `x`.

Tym razem nie musimy już pamiętać stanu z momentu deklaracji zmiennej ponieważ przy wiązaniu dynamicznym odwołujemy się do bieżącego stanu. Jedyne co musimy pamiętać to składnię wyrażenia definiującą wartość zmiennej. A zatem:

$$\mathsf{State} = \mathit{Var} \to \mathit{Expr}$$

Konfiguracje i tranzycje pozostają bez zmian:

$$\Gamma = \mathit{Expr} \times \mathsf{State} \cup \underbrace{\mathsf{Val}}_{T}$$

$$\langle e, s \rangle \to n$$

Reguły:

- dla zmiennych:

$$\frac{\langle e, s \rangle \to n}{\langle x, s \rangle \to n} \quad \text{where} \quad s(x) = e$$

- dla `let`'a:

$$\frac{\langle e_2, s[x \mapsto e_1] \rangle \to n}{\langle \mathtt{let}\ \mathtt{x} = e_1\ \mathtt{in}\ e_2, s \rangle \to n}$$

3.4 Kalkulator funkcyjny

Rozszerzamy język wyrażeń kalkulatora do prostego języka funkcyjnego w następujący sposób:

$$e ::= ... \mid \lambda x.e \mid e_1(e_2)$$

czyli wprowadzając funkcje anonimowe oraz mechanizm aplikacji. Np:

$$(\lambda x.x + 10)(2) \to 12$$

W wyrażeniu $e_1(e_2)$ wyrażenie e_1 musi wyliczyć się do funkcji (anonimowej) bo w przeciwnym przypadku obliczenie nie będzie miało sensu (np. `1(2)` nie ma sensu). Przyjmujemy *statyczną widoczność identyfikatorów*. Czyli wyrażenie

$$\mathtt{let\ x} = \mathtt{7\ in\ let\ f} = \lambda\mathtt{z.x} + \mathtt{z\ in\ let\ x} = \mathtt{f(1)\ in\ f(10)}$$

powinno wyliczyć się do 17 ponieważ `x` w ciele funkcji `f` wiąże statycznie czyli zawsze odnosi się do wartości `x` z momentu deklaracji funkcji a nie z momentu jej wywołania.

Rozważamy dwa warianty:

Przekazywanie parametrów *przez wartość*

Pierwszą rzeczą do zrobienia jest określenie zbioru konfiguracji, w tym konfiguracji końcowych. Do tej pory „kalkulatory" obliczały się po prostu do liczb (całkowitych). Tym razem to nie wystarcza bo np. wyrażenie $\lambda \texttt{x}.\texttt{x} + \texttt{10}$ jest poprawnym wyrażeniem i programem. Musimy zatem rozszerzyć dotychczasowy zbiór wartości wyrażeń o obiekty, które będą w stanie trzymać/reprezentować funkcje anonimowe. Do reprezentacji funkcji $\lambda \texttt{x}.\texttt{e}$ niezbędne będą:

- nazwa parametru formalnego $\texttt{x}$
- ciało funkcji $\texttt{e}$
- stan z chwili deklaracji funkcji (ze względu na statyczne wiązanie identyfikatorów)

A zatem do reprezentowania funkcji $\lambda \texttt{x}.\texttt{e}$ potrzeba i wystarcza trójka $\langle \texttt{x}, \texttt{e}, s\rangle$

Innymi słowy konfiguracje będą wyglądały następująco:

$$\Gamma = \mathit{Expr} \times \mathsf{State} \cup \underbrace{\mathsf{Val}}_{T}$$

$$\mathsf{Val} = \mathsf{Val_E} \cup \mathsf{Val_F}$$

$$\mathsf{Val_E} = \mathbb{Z}$$

$$\mathsf{Val_F} = \mathit{Var} \times \mathit{Expr} \times \mathsf{State}$$

czyli ogólny zbiór wartości wyrażeń w tym języku składa się z wartości liczbowych oraz wartości „funkcyjnych". Stany oczywiście muszą wyglądać:

$$\mathsf{State} = \mathit{Var} \to \mathsf{Val}$$

Ponownie wygląda to na definicję *idem per idem* ale radzimy sobie z tym jak poprzednio.

Reguły:

- dla numerałów:
$$\langle n, s\rangle \to n$$
- dla zmiennych:
$$\langle x, s\rangle \to v \quad \text{where} \quad s(x) = v$$
- dla λ-abstrakcji:
$$\langle \lambda \texttt{x}.\texttt{e}, s\rangle \to \langle \texttt{x}, \texttt{e}, s\rangle$$
- dla sumy wyrażeń:
$$\frac{\langle e_1, s\rangle \to n_1 \qquad \langle e_2, s\rangle \to n_2}{\langle e_1 + e_2, s\rangle \to n} \quad \text{where} \quad n = n_1 + n_2$$

  W szczególności widać, że w tym (najprostszym) rozwiązaniu sumować możemy tylko wyrażenia, które obliczają się do liczb (metazmienne $\texttt{n}$ niejawnie to zapewniają) - tzn. nie możemy sumować dwóch funkcji lub funkcji i liczby.

- dla `let`'a:
$$\frac{\langle e_1, s\rangle \to v_1 \qquad \langle e_2, s[x \mapsto v_1]\rangle \to v_2}{\langle \mathtt{let}\, \mathtt{x} = e_1\, \mathtt{in}\, e_2, s\rangle \to v_2}$$

- dla aplikacji:

  Schemat działania:

  1. odtwórz funkcję (parametr formalny, ciało oraz stan w którym ją deklarowaliśmy) z aktualnego stanu
  2. oblicz wartość parametru aktualnego (w obecnym stanie)
  3. przekaż wartość obliczonego parametru aktualnego do odtworzonego ciała funkcji (czyli podstaw tą wartość pod parametr formalny)

  $$\frac{\langle e_1, s\rangle \to \langle x, e, s'\rangle \qquad \langle e_2, s\rangle \to v' \qquad \langle e, s'[x \mapsto v']\rangle \to v}{\langle e_1(e_2), s\rangle \to v}$$

  W szczególności widać, że zapewniliśmy sobie wyliczanie się $\mathtt{e_1}$ do wartości „funkcyjnej"

Przekazywanie parametrów *przez nazwę*

Przy przekazywaniu parametru (w wyrażeniu aplikacji) nie obliczamy wyrażenia, które reprezentuje parametr aktualny ale przekazujemy do funkcji to wyrażenie wraz ze stanem z miejsca wywołania funkcji (*leniwy* odpowiednik przekazywania przez wartość). Ten przekazany stan będzie brany pod uwagę w momencie obliczania wartości parametru (tzn. w momenecie odwołania w ciele funkcji do parametru formalnego).

Dla programu

$$\mathtt{let}\ \mathtt{f} = \lambda\mathtt{x}.7\ \mathtt{in}\ \mathtt{f(y))}$$

wykonywanego w stanie pustym z mechanizmem przekazywania parametru przez wartość obliczenie jest niepoprawne bo odwołanie do zmiennej `y` jest niepoprawne (bo nie jest określona w stanie pustym). Natomiast gdy parametr przekazywany będzie przez zmienną to wartość wyliczy się do 7.

Konfiguracje są identyczne jak poprzenio:

$$\Gamma = \mathit{Expr} \times \mathsf{State} \cup \underbrace{\mathsf{Val}}_{T}$$

$$\mathsf{Val} = \mathsf{Val_E} \cup \mathsf{Val_F}$$

$$\mathsf{Val_E} = \mathbb{Z}$$

$$\mathsf{Val_F} = \mathit{Var} \times \mathit{Expr} \times \mathsf{State}$$

gdyż wciąż jak poprzednio obliczenia mogą wyliczać się do liczb lub do funkcji ale tym razem zbiór stanów jest nieco „prostszy" i wygląda jak zbiór stanów dla zwykłego kalkulatora leniwego:

$$\mathsf{State} = \mathit{Var} \to \mathit{Expr} \times \mathsf{State}$$

Reguły:

- dla zmiennych:

$$\frac{\langle e, s'\rangle \to v}{\langle x, s\rangle \to v} \quad \text{if} \quad s(x) = \langle e, s'\rangle$$

- dla `let`'a:

$$\frac{\langle e_2, s[x \mapsto \langle e_1, s\rangle]\rangle \to v}{\langle \mathtt{let}\, \mathtt{x} = e_1\, \mathtt{in}\, e_2, s\rangle \to v}$$

- dla aplikacji

$$\frac{\langle e_1, s\rangle \to \langle x, e, s'\rangle \qquad \langle e, s'[x \mapsto \langle e_2, s\rangle]\rangle \to v}{\langle e_1(e_2), s\rangle \to v}$$

Przykład:

$$\cfrac{\cfrac{\langle \mathtt{f}, s[\mathtt{f} \mapsto \langle \lambda\mathtt{x}.7, s\rangle]\rangle \to \langle x, e, s'\rangle \qquad \langle x, s'[x \mapsto \langle y, s[\mathtt{f} \mapsto \langle \lambda\mathtt{x}.7, s\rangle]\rangle]\rangle \to v}{\langle \mathtt{f}(\mathtt{y}), s[\mathtt{f} \mapsto \langle \lambda\mathtt{x}.7, s\rangle]\rangle \to v}}{\langle \mathtt{let}\, \mathtt{f} = \lambda\mathtt{x}.7\, \mathtt{in}\, \mathtt{f}(\mathtt{y}), s\rangle \to v}$$

Z reguły dla zmiennych, z faktu, iż $s[\mathtt{f} \mapsto \langle \lambda\mathtt{x}.7, s\rangle](f) = \langle \lambda x.7, s\rangle$ oraz z aksjomatu λ-abstrakcji $\langle \lambda x.7, s\rangle \to \langle x, 7, s\rangle$ wynika, że $v = 7$.

4 Bloki z deklaracjami zmiennych

Rozszerzamy język TINY o *bloki z deklaracjami zmiennych*. Główna idea jest taka, że zmienne zadeklarowane wewnątrz bloków są *lokalne* dla tych bloków i wewnątrz tych bloków przesłaniają zmienne globalne o tych samych identyfikatorach.

Bloki są instrukcjami więc rozszerzamy instrukcje następująco:

$$\mathit{Instr} \ni I ::= x := e \,|\, \mathtt{skip} \,|\, I_1; I_2 \,|\, \mathtt{if}\, b\, \mathtt{then}\, I_1\, \mathtt{else}\, I_2 \,|\, \mathtt{while}\, b\, \mathtt{do}\, I \,|\, \mathtt{begin}\, d; I\, \mathtt{end}$$

Wariant bez środowiska

Następnie wprowadzamy oddzielną kategorię syntaktyczną w postaci deklaracji:

$$\mathit{Decl} \ni d ::= \mathtt{var}\ x := e \mid d_1;d_2$$

Definiujemy:

- konfiguracje początkowe:

$$\Gamma_I = (\mathit{Instr} \cup \mathit{Expr} \cup \mathit{BExpr} \cup \mathit{Decl}) \times \mathsf{State}$$

- konfiguracje końcowe:

$$\Gamma_T = \mathsf{State}$$

- stany:

$$\mathsf{State} = \mathit{Var} \to \mathsf{Val}$$

- zbiór zmiennych deklarowanych:

$$DV(\texttt{varx} := e) = \{x\}$$

$$DV(d_1; d_2) = DV(d_1) \cup DV(d_2)$$

Reguły:

- dla deklaracji zmiennych

$$\frac{\langle e, s\rangle \to n}{\langle \texttt{var x} := e, s\rangle \to s[x \mapsto n]}$$

- dla złożenia deklaracji:

$$\frac{\langle d_1, s\rangle \to s' \qquad \langle d_2, s'\rangle \to s''}{\langle d_1; d_2, s\rangle \to s''}$$

- dla bloku:

$$\frac{\langle d, s\rangle \to s' \qquad \langle I, s'\rangle \to s''}{\langle \texttt{begin} d; I \texttt{end}, s\rangle \to s''[DV(d) \mapsto s]}$$

Ale rodzi to pewne nieścisłości gdy s jest nieokreślone.

Wariant ze środowiskiem

Rozbijamy dotychczasowe stany na dwa odwzorowania.

Pierwsze odwzorowanie - nazwiemy je *środowiskiem* - mapuje identyfikatory na ich *lokacje* (coś w rodzaju *komórek pamięci*) a drugie - *składy* - mapuje lokacji na wartości.

Definiujemy:

- abstrakcyjny, nieskończony zbiór lokacji i zdroworozsądkowo zakładamy, że w każdej chwili wykonania programu tylko skończona ich liczba jest zaalokowana

$$l \ni \mathsf{Loc} = \{l_0, l_1, ...\}$$

- środowisko jako funkcję częściową ze zbioru identyfikatorów w lokacje

$$\rho \ni \mathsf{VEnv} = \mathit{Var} \rightharpoonup \mathsf{Loc}$$

- składy jako funkcję częściową z lokacji w wartości

$$s \ni \mathsf{Store} = \mathsf{Loc} \rightharpoonup \mathsf{Val}$$

- konfiguracje początkowe:

$$\Gamma_I = (\mathit{Instr} \cup \mathit{Expr} \cup \mathit{BExpr} \cup \mathit{Decl}) \times \mathsf{VEnv} \times \mathsf{Store}$$

- konfiguracje końcowe:
$$\Gamma_T = \mathsf{Store}$$

Czyli tranzycje dla np. instrukcji wyglądają tak

$$\langle I, \rho, s \rangle \to s'$$

co czasem zapisuje się tak

$$\rho \vdash \langle I, s \rangle \to s'$$

W momencie deklaracji nowej zmiennej z dodajemy do środowiska ρ parę $\langle z, l \rangle$, gdzie l jest nową, nieużywaną lokacją. Dla wygody można przyjąć, iż mamy do dyspozycji funkcję, która zwraca zaalokowaną `implicite` nową komórkę pamięci:

$$newloc : \mathsf{Store} to \mathsf{Loc}$$

przy czym $newloc(s) \notin dom(s)$

4.1 Bloki, deklaracje, procedury bez parametrów

4.1.1 Statyczne wiązanie identyfikatorów zmiennych i procedur

Rozszerzamy TINY w następujący sposób:

- w instrukcjach mamy dodatkowo bloki
- bloki składają się deklaracji a następnie instrukcji
- deklaracja składają się z deklaracji zmiennych i/lub deklaracji procedur

czyli

$$Instr \ni I ::= x := e | \texttt{skip} | I_1; I_2 | \texttt{if} b \texttt{then} I_1 \texttt{else} I_2 | \texttt{while} b \texttt{do} I | \texttt{begin} d; I \texttt{end} | \texttt{call} p$$

$$Decl \ni d ::= \texttt{var}\, x := e \mid \texttt{proc}\, p\, \texttt{is}\, I \mid d_1;d_2$$

$$PName \ni ::= \texttt{p} \mid \texttt{q} \mid ...$$

Definiujemy:

- składy
$$s \in \mathsf{Store} = \mathsf{Loc} \rightharpoonup \mathsf{Val}$$
- środowisko zmiennych
$$\rho_V \ni \mathsf{VEnv} = Var \rightharpoonup \mathsf{Loc}$$

- środowisko procedur

  Środowisko procedur mapuje identyfikator procedury na krotkę reprezentującą instrukcję danej procedury oraz „zagnieżdżone" obydwa środowiska: procedur i zmiennych. Zagnieżdżenie środowiska procedur jest niezbędne do tego aby wewnątrz deklarowanej procedury statycznie wołać inne procedury - wiązanie identyfikatorów procedur występujących w ciele procedury zostaje zamrożone w chwili deklaracji tej procedury. Podobnie zagnieżdżonego środowiska zmiennych potrzebujemy, żeby związać identyfikatory zmiennych w chwili deklaracji procedury.

$$\rho_P \ni \mathsf{PEnv} = \mathit{PName} \rightharpoonup \mathsf{Proc}$$
$$\mathsf{Proc} = \mathit{Instr} \times \mathsf{VEnv} \times \mathsf{PEnv}$$

Postać tranzycji:

- dla wyrażeń arytmetycznych

  W wyrażeniach arytmetycznych występują zmienne a zatem musimy mieć dostęp do środowiska zmiennych oraz składów, żeby wyłuskiwać wartości tychże zmiennych:
  $$\langle e, \rho_V, s\rangle \to \underline{n}$$

- dla wyrażeń logicznych

  Analogicznie:
  $$\langle b, \rho_V, s\rangle \to \underline{b}$$

- dla instrukcji

  Niewątpliwie instrukcje muszą mieć dostęp do środowisk i składów ale zasadniczym pytaniem jest to czy instrukcje zmieniają środowiska - odpowiedzią jest nie, instrukcje nie zmieniają środowisk.
  $$\langle I, \rho_V, \rho_P, s\rangle \to s'$$

- dla deklaracji

  Deklaracje zmieniają środowiska oraz składy (gdy deklarowane zmienne inicjujemy wartościami wyrażeń) więc:
  $$\langle d, \rho_V, \rho_P, s\rangle \to \langle \rho'_V, \rho'_P, s'\rangle$$

Reguły:

- **dla wyrażeń arytmetycznych**

  Przykładowo:
  $$\frac{}{\langle x, \rho_V, s\rangle \to s(\rho_V(x))}$$

- **dla deklaracji**

  - deklaracja zmiennych

  $$\frac{\langle e, \rho_V, s\rangle \to \underline{n}}{\langle \mathtt{var}\ x := e, \rho_V, \rho_P, s\rangle \to \langle \rho_V[x \mapsto l], \rho_P, s[l \mapsto \underline{n}]\rangle} \quad \text{where} \quad l = \mathit{newloc}(s)$$

– deklaracja procedur

$$\frac{}{\langle \texttt{proc}\, p\, \texttt{is}\, I, \rho_V, \rho_P, s\rangle \to \langle \rho_V, \rho_P[p \mapsto \langle I, \rho_V, \rho_P\rangle], s\rangle}$$

– złożenie procedur

$$\frac{\langle d_1, \rho_V, \rho_P, s\rangle \to \langle \rho'_V, \rho'_P, s'\rangle \qquad \langle d_2, \rho'_V, \rho'_P, s'\rangle \to \langle \rho''_V, \rho''_P, s''\rangle}{\langle d_1; d_2, \rho_V, \rho_P, s\rangle \to \langle \rho''_V, \rho''_P, s''\rangle}$$

Kolejność deklaracji ma oczywiście znaczenie dla semantyki złożenia.

Oczywiście deklaracja zmiennych zmienia skład a deklaracja procedur nie zmienia składu ale przy złożeniu nie wiemy, w którym przypadku jesteśmy.

- **dla instrukcji**

  – instrukcja przypisania

$$\frac{\langle e, \rho_V, s\rangle \to \underline{n}}{\langle x := e, \rho_V, \rho_P, s\rangle \to s[l \mapsto \underline{n}]} \quad \text{where} \quad l = \rho_V(x)$$

  – instrukcja bloku

  Instrukcja wewnętrzna bloku jest wykonywana w środowiskach i składzie zmodyfikowanych przez deklarację:

$$\frac{\langle d, \rho_V, \rho_P, s\rangle \to \langle \rho'_V, \rho'_P, s'\rangle \qquad \langle I \rho'_V, \rho'_P, s'\rangle \to s''}{\langle \texttt{begin}\, d; I\, \texttt{end}, \rho_V, \rho_P, s\rangle \to s''}$$

  Po wyjściu z bloku znajdujemy się w stanie s''. Deklaracja d co prawda mogła zmodyfikować środowiska, w szczególności alokując nowe lokacje, ale te lokacje stają się niewidoczne po wyjściu z bloku.

  – złożenie instrukcji

$$\frac{\langle I_1, \rho_V, \rho_P, s\rangle \to s' \qquad \langle I_2, \rho_V, \rho_P, s'\rangle \to s''}{\langle I_1; I_2, \rho_V, \rho_P, s\rangle \to s''}$$

  Widać wyraźnie, że środowiska nie zmieniają się między instrukcjami wykonywanymi na tym samym poziomie.

  – wywołanie procedury

    * procedury nierekurencyjne

$$\frac{\langle I, \rho'_V, \rho'_P, s\rangle \to s'}{\langle \texttt{call}\, p, \rho_V, \rho_P, s\rangle \to s'} \quad \text{where} \quad \rho_P(p) = \langle I, \rho'_V, \rho'_P\rangle$$

    Wywołując procedurę najpierw ze środowiska procedur „odtwarzamy” jej krotkę, która powstała w chwili jej deklaracji a następnie wywołujemy instrukcję z tej krotki w środkowiskach tej krotki i w składzie z jakim wywołujemy procedurę.

W momencie wywołania procedury p jej instrukcja I operuje w środowisku procedur ρ'_P, które było środowiskiem tuż sprzed deklaracji p. A zatem to środowisko „nie widzi" w sobie deklaracji procedury p (ew. pod p może znajdować się inna procedura, jeszcze wcześniej zadeklarowana pod tym samym identyfikatorem) i w związku z tym wewnątrz p nie jest możliwe rekurencyjne wywołanie p.

* procedury rekurencyjne
  Aby umożliwić wywołania rekurencyjne potrzeba i wystarcza wywołać I w środowisku ρ'_P z dodatkową informacją o tym, jak działa p:

$$\frac{\langle I, \rho'_V, \rho'_P[p \mapsto \langle I, \rho'_V, \rho'_P\rangle], s\rangle \to s'}{\langle \texttt{call}\, p, \rho_V, \rho_P, s\rangle \to s'} \quad \text{where} \quad \rho_P(p) = \langle I, \rho'_V, \rho'_P\rangle$$

4.1.2 Dynamiczne wiązanie identyfikatorów zmiennych i procedur

Jedyne miejsce, w którym mechanizm wiązania identyfikatorów ma wpływ na semantykę programu to wywołanie procedury. Przy wiązaniu statycznym zmiennych i procedur odtwarzaliśmy zagnieżdżony stan środowisk aby w tych odtworzonych środowiskach wywołać procedurę. Teraz jedynie wystarczy odtworzyć ciało procedury z aktualnego środowiska.

Definiujemy:

- składy bez zmian

$$s \ni \mathsf{Store} = \mathsf{Loc} \rightharpoonup \mathsf{Val}$$

- środowisko zmiennych

  De facto, nie potrzebujemy środowiska zmiennych - podobnie jak w przypadku języka TINY z z samymi deklaracjami zmiennych. Ale to podejście wygląda bardziej elegencko:

$$\rho_V \ni \mathsf{VEnv} = \mathit{Var} \rightharpoonup \mathsf{Loc}$$

- środowisko procedur

  Tym razem jest prościej bo jedyne co trzeba zrobić, to dla identyfikatora deklarowanej procedury zapamiętać jej instrukcję:

$$\rho_P \ni \mathsf{PEnv} = \mathit{PName} \rightharpoonup \mathsf{Proc}$$
$$\mathsf{Proc} = \mathit{Instr}$$

Reguły:

- wywołanie procedury

$$\frac{\langle I, \rho_V, \rho_P, s\rangle \to s'}{\langle \texttt{call}\, p, \rho_V, \rho_P, s\rangle \to s'} \quad \text{where} \quad \rho_P(p) = I$$

  czyli wyciągamy ciało procedury ze środowiska procedur i wykonujemy je w aktualnych środowiskach. Oczywiście w ten sposób zapewniamy sobie też możliwość rekurencyjnego wywoływania procedur.

- pozostałe: w miarę oczywiste

4.2 Procedury z parametrami

Dodajemy do schematu przekazywanie parametrów do procedur:

$$\mathit{Instr} \ni i ::= x := e | \texttt{skip} | i_1 ; i_2 | \texttt{if} b \texttt{then} i_1 \texttt{else} i_2 | \texttt{while} b \texttt{do} I | \texttt{begin} d ; I \texttt{end} | \texttt{call} p(x)$$

$$\mathit{Decl} \ni d ::= \texttt{var}\, x := e \mid \texttt{proc}\, p(x)\ \texttt{is}\ I \mid d_1 ; d_2$$

$$\mathit{PName} \ni p ::= \texttt{p} \mid \texttt{q} \mid \ldots$$

Wiązanie identyfikatorów zarówno procedur jak i zmiennych jest statycznei i rozpatrujemy trzy mechanizmy przekazywania parametrów.

4.2.1 Przekazywanie parametrów przez zmienną

Musimy rozszerzyć dotychczasowe środowisko procedur o informację dotyczącą nazwy parametru formalnego.

Definiujemy:

- składy bez zmian
$$s \ni \mathsf{Store} = \mathsf{Loc} \rightharpoonup \mathsf{Val}$$
- środowisko zmiennych bez zmian
$$\rho_V \ni \mathsf{VEnv} = \mathit{Var} \rightharpoonup \mathsf{Loc}$$
- środowisko procedur
$$\rho_P \ni \mathsf{PEnv} = \mathit{PName} \rightharpoonup \mathsf{Proc}$$
$$\mathsf{Proc} = \mathit{Var} \times \mathit{Instr} \times \mathsf{VEnv} \times \mathsf{PEnv}$$

Istotnie zmieniające się reguły:

- deklaracja procedur
$$\frac{}{\langle \texttt{proc}\, p(x)\ \texttt{is}\ I, \rho_V, \rho_P, s\rangle \rightarrow \langle \rho_V, \rho_P[p \mapsto \langle x, I, \rho_V, \rho_P\rangle], s\rangle}$$
- wywołanie procedur (z rekurencją)
$$\frac{\langle I, \rho'_V[y \mapsto l], \rho'_P[p \mapsto \langle y, I, \rho'_V, \rho'_P\rangle], s\rangle \rightarrow s'}{\langle \texttt{call}\, p(x), \rho_V, \rho_P, s\rangle \rightarrow s'} \quad \text{if} \quad \rho_P(p) = \langle y, I, \rho'_V, \rho'_P\rangle, \rho_V(x) = l$$

  Z aktualnego środowiska procedur wyciągamy sygnaturę wołanej procedury, w której nowym elementem jest identyfikator parametru formalnego tej procedury a z aktualnego środowiska zmiennych wyciągamy lokację parametru aktualnego, z którym wołamy procedurę. Następnie wykonujemy ciało procedury I w odczytanych środowiskach ρ'_V, ρ'_P (wiązania statyczne) zmodyfikowanych o

– ρ'_V modyfikujemy tak, aby parametr formalny y odwoływał się do wartości parametru aktualnego x

– ρ'_V modyfikujemy tak jak wyżej, żeby zapewnić rekurencje

4.2.2 Przekazywanie parametrów przez wartość

$$Instr \ni i ::= ... \mid \texttt{call}\, p(e)$$

Tak jak poprzednio odczytujemy ze środowiska procedur krotkę charakteryzującą wołaną procedurę. Trick z przekazywaniem przez wartość polega na tym, aby zaalokować nową komórkę pamięci, przypisać ją do identyfikatora parametru formalnego i nadać jej odpowiednią wartość (oczywiście może się zdarzyć, że w ten sposób przesłonimy już zainicjowaną zmienną pod tym samym identyfikatorem ale nic z tym nie zrobimy bo to kwestia użycia odpowiedniego identyfikatora dla parametru formalnego przy deklaracji procedury):

$$\frac{\langle e, \rho_V, s\rangle \to \underline{n} \qquad \langle i, \rho'_V[x \mapsto l], \rho'_P[p \mapsto \rho_P(p)], s[l \mapsto \underline{n}]\rangle \to s'}{\langle \texttt{call}\, p(e), \rho_V, \rho_P, s\rangle \to s'}$$
$$\text{if} \quad \rho_P(p) = \langle x, i, \rho'_V, \rho'_P\rangle, \quad l = newloc(s)$$

4.2.3 Przekazywanie parametrów in-out

Działania:

- odczytujemy ze środowiska procedur krotkę charakteryzującą wołaną procedurę
- alokujemy nową komórkę pamięci, wiążemy ją zindentyfikatorem parametru formalnego wołanej procedury i nadajemy jej wartość parametru aktualnego
- wewnątrz procedury działamy tak jakby parametr był przekazany przez wartość
- po zakończeniu procedury parametrowi aktualnemu, z którym wywołaliśmy procedurę, nadajemy wartość którą uzyskał parametr formalny w wyniku działania procedury

$$\frac{\langle i, \rho'_V[y \mapsto l], \rho'_P, s[l \mapsto s(\rho_V(x))]\rangle \to s'}{\langle \texttt{call}\, p(x), \rho_V, \rho_P, s\rangle \to s'[\rho_V(x) \mapsto s'(l)]}$$
$$\text{if} \quad \rho_P(p) = \langle y, i, \rho'_V, \rho'_P\rangle, \quad l = newloc(s)$$

5 Zadanka

5.1 Funkcje zwracające wartość

5.1.1 Język

$$
\begin{array}{rcl}
\mathit{Var} \ni x & ::= & x_1 \mid x_2 \mid ... \\
\mathit{FId} \ni f & ::= & f_1 \mid f_2 \mid ... \\
\mathbb{Z} \ni n & ::= & ... \mid -1 \mid 0 \mid 1 \mid ... \\
\mathit{Expr} \ni e & ::= & n \mid x \mid e + e \mid f(e) \\
\mathit{Decl} \ni d & ::= & \texttt{var}\, x = 0 \mid d; d \mid \texttt{fun}\, f(x)\ \{\ i\ \} \\
\mathit{Instr} \ni i & ::= & x := e \mid i; i \mid \texttt{skip} \mid \texttt{begin}\, d\, \texttt{in}\, i\, \texttt{end} \mid \texttt{if}\, e = 0\, \texttt{then}\, i\, \texttt{fi} \mid \\
& & \texttt{while}\, e \neq 0\, \texttt{do}\, i\, \texttt{done} \mid \texttt{return}\, e
\end{array}
$$

5.1.2 Opis

Wyrażeni $f(e)$ oblicza przekazywany przez wartość parametr e a następnie wywołuje funkcję f. Wyrażenie to otrzymuje wartość obliczoną przez pierszwą wykonaną przez f instrukcję `return` e. Jeśli f kończy się bez `return` to wyrażenie zwraca 0. Wykonanie `return` poza ciałem funkcji ma dowolne skutki natomiast wewnątrz ciała kończy natychmiast wykonanie funkcji i przekazuje wynik. Wiązanie identyfikatorów zarówno zmiennych jak i funkcji jest statyczne, funkcje mogą być rekurencyjne.

5.1.3 Rozwiązanie

Zaczynamy od zdefiniowania konfiguracji / przejść. Zasadniczym pytaniem jest to jak powinny wyglądać przejścia dla wyrażeń. Wyrażenia niewątpliwie wciąż zwracają wartości ale wyrażenie $f(e)$ odpala wykonanie funkcji f, która odpala wykonanie instrukcji. A zatem w tym języku wyrażenia mogą zmieniać stany (składy). Aby kończyć natychmiast po `return` potrzebujemy flag do rozpoznawania czy `return` nastąpił czy nie. Potrzebujemy też przenosić wartość zwracaną przez `return` - to oznacza, że będziemy potrzebować dużo flag. Ale spróbujmy inaczej: wyróżnijmy element $\tau \in \mathsf{Loc}$ jako zarezerwowaną komóreczkę na flagi. Loc jest abstrakcyjny więc mało co nas tu ogranicza.

Drugim pytaniem jest to, gdzie modyfikujemy flagi - z pewnością robią to instrukcje. Na pewno nie robią tego deklaracje. A czy wyrażenia mogą modyfikować flagi? Wyrażenie $f(e)$ odpala co prawda jakaś instrukcję, która być może odpala `return`'a, który modyfikuje flagę ale z punktu widzenia wyrażenia $f(e)$ flaga nie ma znaczenia - to wyrażenie po prostu zwraca jakąś wartość.

Trzecim problem jest to jak radzić sobie z `return`'em, który wystąpi poza pętlą. Jeśli taki return ustawi swoją flagę to de facto zakończy program. Wedle polecenia takie rozwiązanie jest akceptowalne więc tak to zostawiamy - `return` zawsze ustawia flagę, `return` poza pętlą kończy program.

Zdefiniujemy:

- wartości

  $$\mathsf{Val} = \mathbb{Z} \cup \{\bot\}$$

  gdzie $\bot$ będzie quasiflagą dla sytuacji braku `returna`'a.

- składy

$$\mathsf{Store} = \mathsf{Loc} \to \mathsf{Val}$$

- środowisko zmiennych

$$\mathsf{VEnv} = \mathit{Var} \rightharpoonup \mathsf{Loc}$$

- środowisko funkcji

$$\mathsf{FEnv} = \mathit{FId} \rightharpoonup \mathsf{Func}$$

$$\mathsf{Func} = \mathit{Var} \times \mathit{Instr} \times \mathsf{VEnv} \times \mathsf{FEnv}$$

Przejścia są następujące:

- dla wyrażeń

  Wyrażenia nie tylko zwracają wartość ale też zmieniają składy (ale nie flagi):

$$\langle e, \rho_V, \rho_F, s\rangle \to \langle \underline{n}, s'\rangle$$

- dla deklaracji

  Deklaracje zmieniają środowiska i składy (ale nie flagi):

$$\langle d, \rho_V, \rho_F, s\rangle \to \langle \rho'_V, \rho'_F, s'\rangle$$

- dla instrukcji

  Instrukcje nie zmieniają środowisk, jedynie składy (w tym flagi):

$$\langle i, \rho_V, \rho_F, s\rangle \to s'$$

Reguły:

- dla wyrażeń
  - stała

$$\frac{}{\langle n, \rho_V, \rho_F, s\rangle \to \langle \underline{n}, s\rangle} \quad \text{if} \quad \underline{n} = \_(n)$$

  - zmienna

$$\frac{}{\langle x, \rho_V, \rho_F, s\rangle \to \langle s(\rho_V(x)), s\rangle} \quad \text{if} \quad \rho_V(x) \in \mathsf{Loc}$$

  - suma

$$\frac{\langle e_1, \rho_V, \rho_F, s\rangle \to \langle \underline{n}_1, s_1\rangle \qquad \langle e_2, \rho_V, \rho_F, s\rangle \to \langle \underline{n}_2, s_2\rangle}{\langle e_1 + e_2, \rho_V, \rho_F, s\rangle \to \langle \underline{n}_1 + \underline{n}_2, s_2\rangle}$$

    Tutaj widać jak przydaje się to, że wyrażenia nie modyfikują flag. Wewnątrz np. e_1 dziać się mogą różne rzeczy z flagami ale po wyjściu z e_1 te działania są dla nas przeźroczyste.

– funkcja

Sytuacja, gdy wewnątrz f pojawił się (niezagnieżdżony w inną funkcję) `return`:

$$\frac{\langle e, \rho_V, \rho_F, s\rangle \to \langle \underline{n}, s'\rangle \qquad \langle i, \rho'_V[x \mapsto l], \rho'_F[f \mapsto \langle x, i, \rho'_V, \rho'_F\rangle], s'[l \mapsto \underline{n}]\rangle \to s''}{\langle f(e), \rho_V, \rho_F, s\rangle \to \langle \underline{m}, s''[\tau \mapsto \bot]\rangle}$$

$$\text{if} \quad \rho_F(f) = \langle x, i, \rho'_V, \rho'_F\rangle, \quad \underline{m} = s''[\tau] \in \mathbb{Z}, \quad l = \mathit{newloc}(s')$$

Gdy nie pojawił się `return`:

$$\frac{\langle e, \rho_V, \rho_F, s\rangle \to \langle \underline{n}, s'\rangle \qquad \langle i, \rho'_V[x \mapsto l], \rho'_F[f \mapsto \langle x, i, \rho'_V, \rho'_F\rangle], s'[l \mapsto \underline{n}]\rangle \to s''}{\langle f(e), \rho_V, \rho_F, s\rangle \to \langle 0, s''\rangle}$$

$$\text{if} \quad \rho_F(f) = \langle x, i, \rho'_V, \rho'_F\rangle, \quad s''[\tau] = \bot, \quad l = \mathit{newloc}(s')$$

Najpierw ewaluujemy e w aktualnych środowiskach i składzie. Następnie standardowo dla statycznego wiązania, rekurencji i przekazywania parametru przez wartość. Jeśli `return` ustawił flagę to musimy tą flagę wymazać.

- dla deklaracji

  – deklaracja zmiennej

$$\frac{}{\langle \mathtt{var}\ x := 0, \rho_V, \rho_F, s\rangle \to \langle \rho_V[x \mapsto l], \rho_F, s[l \mapsto 0]\rangle} \qquad \text{where} \quad l = \mathit{newloc}(s)$$

  – deklaracja funckji

$$\frac{}{\langle \mathtt{fun}\ f(x)\ \{\ i\ \}, \rho_V, \rho_F, s\rangle \to \langle \rho_V, \rho_F[f \mapsto \langle x, i, \rho_V, \rho_F\rangle], s\rangle}$$

  – złożenie deklaracji

$$\frac{\langle d_1, \rho_V, \rho_F, s\rangle \to \langle \rho'_V, \rho'_F, s'\rangle \qquad \langle d_2, \rho'_V, \rho'_F, s'\rangle \to \langle \rho''_V, \rho''_F, s''\rangle}{\langle d_1; d_2, \rho_V, \rho_F, s\rangle \to \langle \rho''_V, \rho''_F, s''\rangle}$$

- dla instrukcji

  – przypisanie

  Tu bez kontrowersji bo na tym poziomie nie musimy dbać o ew. kończenie:

$$\frac{\langle e, \rho_V, \rho_F, s\rangle \to \langle \underline{n}, s'\rangle}{\langle x := e, \rho_V, \rho_F, s\rangle \to s'[\rho_V(x) \mapsto \underline{n}]}$$

  – złożenie instrukcji

  Tutaj już musimy zadbać o szybkie kończenie:

$$\frac{\langle i_1, \rho_V, \rho_F, s\rangle \to s'}{\langle i_1; i_2, \rho_V, \rho_F, s\rangle \to s'} \qquad \text{if} \quad s'(\tau) = \underline{n} \in \mathbb{Z}$$

$$\frac{\langle i_1, \rho_V, \rho_F, s\rangle \to s' \qquad \langle i_2, \rho_V, \rho_F, s\rangle \to s''}{\langle i_1; i_2, \rho_V, \rho_F, s\rangle \to s''} \qquad \text{if} \quad s'(\tau) = \bot$$

– skip

$$\frac{}{\langle \texttt{skip}, \rho_V, \rho_F, s\rangle \to s}$$

– instrukcja bloku

Standardowo - instrukcja i być może ustawi flagę wyjścia ale spropaguje się ona naturalnie:

$$\frac{\langle d, \rho_V, \rho_F, s\rangle \to \langle \rho'_V, \rho'_F, s'\rangle \qquad \langle i, \rho'_V, \rho'_F, s'\rangle \to s''}{\langle \texttt{begin}\, d\, \texttt{in}\, i\, \texttt{end}, \rho_V, \rho_F, s\rangle \to s''}$$

– instrukcja `if`

$$\frac{\langle e, \rho_V, \rho_F, s\rangle \to \langle \underline{n}, s'\rangle}{\langle \texttt{if}\, e = 0\, \texttt{then}\, i\, \texttt{fi}, \rho_V, \rho_F, s\rangle \to s'} \quad \text{if} \quad \underline{n} \neq 0$$

$$\frac{\langle e, \rho_V, \rho_F, s\rangle \to \langle 0, s'\rangle \qquad \langle i, \rho_V, \rho_F, s'\rangle \to s''}{\langle \texttt{if}\, e = 0\, \texttt{then}\, i\, \texttt{fi}, \rho_V, \rho_F, s\rangle \to s''}$$

– instrukcja `while`

Jeśli dozór wylicza się do zera to przechodzimy do następnej instrukcji:

$$\frac{\langle e, \rho_V, \rho_F, s\rangle \to \langle 0, s'\rangle}{\langle \texttt{while}\, e \neq 0\, \texttt{di}\, i\, \texttt{done}, \rho_V, \rho_F, s\rangle \to s'}$$

Jeśli dozór wylicza się do niezera i pierwsze wykonanie instrukcji `while`'a generuje flagę wyjścia to wychodzimy z `while`'a:

$$\frac{\langle e, \rho_V, \rho_F, s\rangle \to \langle \underline{n}, s'\rangle \qquad \langle i, \rho_V, \rho_F, s'\rangle \to s''}{\langle \texttt{while}\, e \neq 0\, \texttt{di}\, i\, \texttt{done}, \rho_V, \rho_F, s\rangle \to s''} \quad \text{if} \quad \underline{n} \neq 0, \quad s''(\tau) = \underline{m} \in \mathbb{Z}$$

Jeśli dozór wylicza się do niezera i pierwsze wykonanie instrukcji `while`'a nie generuje flagi wyjścia to robimy krok indukcyjny propagując ew. flagę:

$$\frac{\langle e, \rho_V, \rho_F, s\rangle \to \langle \underline{n}, s'\rangle \qquad \langle i, \rho_V, \rho_F, s'\rangle \to s'' \qquad \langle \texttt{while}\, e \neq 0\, \texttt{di}\, i\, \texttt{done}, \rho_V, \rho_F, s''\rangle \to s'''}{\langle \texttt{while}\, e \neq 0\, \texttt{di}\, i\, \texttt{done}, \rho_V, \rho_F, s\rangle \to s'''}$$

$$\text{if} \quad \underline{n} \neq 0, \quad s''(\tau) = \bot$$

– instrukcja `return`

Instrukcja `return` po prostu ustawia flagę:

$$\frac{\langle e, \rho_V, \rho_F, s\rangle \to \langle \underline{n}, s'\rangle}{\langle \texttt{return}\, e, \rho_V, \rho_F, s\rangle \to s'[\tau \mapsto \underline{n}]}$$

5.1.4 Rozwiązanie z ignorowanym `return` poza funkcją

Tym razem podczas wystąpienia `return` nieokalanego funkcją chcemy go po prostu ignorować. Musimy dodać jedną flagę do rozpoznawania sytuacji czy jesteśmy wewnątrz jakiejś funkcji czy nie:

$$\mathsf{Val} = \mathbb{Z} \cup \{\bot, \top\}$$

gdzie $\top$ oznacza, że nie jesteśmy zagnieżdżenie w żadną funkcję, $\bot$ gdy jesteśmy zagnieżdżeni ale dotychczas bez `return`a'a.

Ignorująca wersja instrukcji `return`:

$$\frac{\langle e, \rho_V, \rho_F, s\rangle \to \langle \underline{n}, s'\rangle}{\langle \texttt{return}\, e, \rho_V, \rho_F, s\rangle \to s'} \quad \text{if} \quad s(\tau) = \top$$

$$\frac{\langle e, \rho_V, \rho_F, s\rangle \to \langle \underline{n}, s'\rangle}{\langle \texttt{return}\, e, \rho_V, \rho_F, s\rangle \to s'[\tau \mapsto \underline{n}]} \quad \text{if} \quad s(\tau) = \bot$$

Reguły dla funkcji muszą teraz rozpoznawać sytuację, w której dana funkcja jest funkcją niezagnieżdżoną, gdyż taka najbardziej zewnętrzna funkcja musi przywrócić flagę niezagnieżdżenia.

Funkcja niezagnieżdżona z powrotem:

$$\frac{\langle e, \rho_V, \rho_F, s\rangle \to \langle \underline{n}, s'\rangle \qquad \langle i, \rho_V'[x \mapsto l], \rho_F'[f \mapsto \langle x, i, \rho_V', \rho_F'\rangle], s'[l \mapsto \underline{n}][\tau \mapsto \bot]\rangle \to s''}{\langle f(e), \rho_V, \rho_F, s\rangle \to \langle \underline{m}, s''[\tau \mapsto \top]\rangle}$$

$$\text{if} \quad s(\tau) = \top,\ \rho_F(f) = \langle x, i, \rho_V', \rho_F'\rangle, \quad \underline{m} = s''[\tau] \in \mathbb{Z}, \quad l = \mathit{newloc}(s')$$

Funkcja niezagnieżdżona bez powrotu:

$$\frac{\langle e, \rho_V, \rho_F, s\rangle \to \langle \underline{n}, s'\rangle \qquad \langle i, \rho_V'[x \mapsto l], \rho_F'[f \mapsto \langle x, i, \rho_V', \rho_F'\rangle], s'[l \mapsto \underline{n}][\tau \mapsto \bot]\rangle \to s''}{\langle f(e), \rho_V, \rho_F, s\rangle \to \langle 0, s''[\tau \mapsto \top]\rangle}$$

$$\text{if} \quad s(\tau) = \top,\ \rho_F(f) = \langle x, i, \rho_V', \rho_F'\rangle, \quad s''[\tau] = \bot, \quad l = \mathit{newloc}(s')$$

Funkcja zagnieżdżona z powrotem:

$$\frac{\langle e, \rho_V, \rho_F, s\rangle \to \langle \underline{n}, s'\rangle \qquad \langle i, \rho_V'[x \mapsto l], \rho_F'[f \mapsto \langle x, i, \rho_V', \rho_F'\rangle], s'[l \mapsto \underline{n}]\rangle \to s''}{\langle f(e), \rho_V, \rho_F, s\rangle \to \langle \underline{m}, s''[\tau \mapsto \bot]\rangle}$$

$$\text{if} \quad s(\tau) = \bot,\ \rho_F(f) = \langle x, i, \rho_V', \rho_F'\rangle, \quad \underline{m} = s''[\tau] \in \mathbb{Z}, \quad l = \mathit{newloc}(s')$$

Funkcja zagnieżdżona bez powrotu:

$$\frac{\langle e, \rho_V, \rho_F, s\rangle \to \langle \underline{n}, s'\rangle \qquad \langle i, \rho_V'[x \mapsto l], \rho_F'[f \mapsto \langle x, i, \rho_V', \rho_F'\rangle], s'[l \mapsto \underline{n}]\rangle \to s''}{\langle f(e), \rho_V, \rho_F, s\rangle \to \langle 0, s''\rangle}$$

$$\text{if} \quad s(\tau) = \bot,\ \rho_F(f) = \langle x, i, \rho_V', \rho_F'\rangle, \quad s''[\tau] = \bot, \quad l = \mathit{newloc}(s')$$