Semantyka języków programowania

MSUI, II rok

Ewa Madalińska

http://www.mimuw.edu.pl/ ewama/zajecia.htm

mail: ewama@mimuw.edu.pl

konsultacje pok. 4400

Literatura:

1. K. Apt, E. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 1991.

2. P. Dembiński, J. Małuszyński. *Matematyczne metody definiowania języków programowania*. WNT, 1981.

3. E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

4. M. J. C. Gordon. *The Denotational Description of Programming Languages*, Springer-Verlag, Berlin 1979.

5. H. Riis Nielson, F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1999.

6. G.D. Plotkin. *A structural approach to operational semantics*. Technical Report DAIMI-FN 19, Department of Computer Science, Arrhus University, 1981.

7. G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*, The MIT Press, Cambridge, Massachusetts, London, 1993.

Wstęp

1. Potrzeba formalizacji języka programowania

   - Rozwój języków programowania wysokiego poziomu.

     – Języki wysokiego poziomu – języki niezależne od konkretnych maszyn liczących.

     – poprawność syntaktyczna – **składnia;**

     – treści wyrażane przez instrukcje – **semantyka.**

   Dla języków wewnętrznych semantyka określona jest przez konstrukcję maszyn.

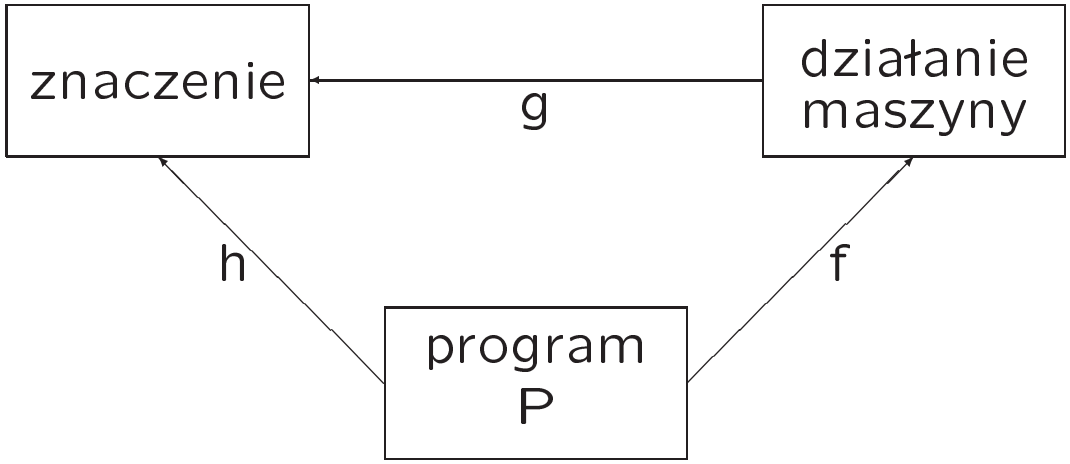
- Różnorodność konstrukcji maszyn prowadzi do oddzielenia opisu składni języka od jego maszynowych realizacji.

- Konieczność weryfikacji programu – czy rozwiązuje postawione zadanie.

- Zapewnienie, aby wykonanie programu na różnych maszynach dawało ten sam wynik.

Wynika stąd konieczność podania **abstrakcyjnej**, nadrzędnej definicji semantyki, która mogłaby służyć jako wzorzec dla każdej realizacji.

Schemat opisu semantycznego



g – określenie efektu działania maszyny w terminach postawionego zadania;

f – realizacja maszynowa programu;

h – określenie znaczenia programu w terminach postawionego zadania.

zgodność realizacji ze wzorcem

$$g(f(P)) = h(P)$$

Kontrprzykłady:

Algol-60, Pascal

2. Różne podejścia do problemu definiowania semantyki

(a) Metoda *operacyjna* – polegająca na wskazaniu kolejnych czynności. Znaczenie zdefiniowane w ten sposób nie musi być jednak ciągiem czynności, może to być wynikowa zmiana stanu.

(b) Metoda *funkcyjna* lub *denotacyjna* – polegająca na określeniu znaczenia poszczególnych elementów składowych konstrukcji języka i zdefiniowania znaczenia całej konstrukcji jako funkcji znaczeń jej składowych.

(c) Metody *logiczne* – polegające na określeniu zbioru zdań prawdziwych definiujących znaczenie konstrukcji językowych.

Opis składni

- Gramatyki bezkontekstowe.

- *Drzewo wywodu* – określa strukturę każdego słowa z języka generowanego przez tę gramatykę.

- Zastosowanie gramatyk bezkontekstowych do opisu języków programowania – BNF

Zalety:

1. Rozstrzygalność języka definiowanego przez gramatykę.

2. Możliwość opisu struktury słów języka za pomocą drzew wywodu.

3. Możliwość modularnego definiowania języka.

4. Możliwość stosowania mnemotechnicznych oznaczeń dla symboli nieterminalnych, co ułatwia naukę języka.

Wady:

1. Niewyrażalność ograniczeń kontekstowych języka:

   (a) każda zmienna w programie musi być zadeklarowana;

   (b) nie mogą występować różne deklaracje tej samej zmiennej w jednym bloku programu.

Podsumowanie

Cele opisu semantycznego:

1. opis jako podstawa implementacji;
2. opis jako podstawa do rozumienia zapisywanego przez programistę algorytmu, jego dokumentacji i weryfikacji oraz komunikowania rozwiązań między ludźmi.

Przez *semantykę* języka programowania będziemy rozumieć przyporządkowanie, które z każdym wyrażeniem (programem) wiąże pewien obiekt – znaczenie ze ściśle określonego zbioru *znaczeń*.

Uwaga 1
Znaczenie będziemy wiązać z drzewem wywodu programu, a nie napisem. ∎

Zmieniając zbiór znaczeń możemy mieć do czynienia z różnymi semantykami tego samego języka.

Z drugiej strony, ten sam obiekt – znaczenie programu – może być wyznaczony na różne sposoby – różne metody definiowania (tej samej) semantyki.

Semantyka operacyjna

Terminologia:

konfiguracja (systemu, maszyny abstrakcyjnej):
mieszanka obiektów syntaktycznych – programów i danych (lokacje lub środowiska).

$$\text{SYSTEM} = \text{PROGRAM} + \text{DANE}$$

system przejść: Zachowanie systemu składa się z elementarnych kroków zwanych **przejściami**.

Przejścia z jednej *konfiguracji systemu* do innej – *relacja binarna* na konfiguracjach.

Definicja 1

Systemem przejść nazywamy strukturę $M = \langle \Gamma, \rightsquigarrow \rangle$, gdzie Γ jest zbiorem konfiguracji, a $\rightsquigarrow \subseteq \Gamma \times \Gamma$ jest relacją binarną zwaną *relacją przejścia*. ∎

Jeśli $\gamma, \gamma' \in \Gamma$, to $\gamma \rightsquigarrow \gamma'$ oznacza, że jest przejście z konfiguracji γ do konfiguracji γ'.

W zależności od definicji konfiguracji relacja przejścia $\rightsquigarrow$ może modelować wykonanie programu na różnych poziomach szczegółowości.

Semantyka zostanie zdefiniowana przy pomocy relacji przejścia $\rightsquigarrow$, dokładniej – przy pomocy $\rightsquigarrow^*$, gdzie $\rightsquigarrow^*$ oznacza domknięcie przechodnie relacji $\rightsquigarrow$.

Definicja 2

1. *Obliczeniem* systemu przejść M nazywamy dowolny, skończony lub nieskończony ciąg konfiguracji $\gamma_0, \ldots, \gamma_i, \ldots$ taki, że dla każdego $(i \geq 0)$ zachodzi $\gamma_i \rightsquigarrow \gamma_{i+1}$.

2. *Konfiguracją końcową* systemu M nazywamy konfigurację γ, dla której nie istnieje konfiguracja γ' taka, że $\gamma \rightsquigarrow \gamma'$.

3. *Obliczeniem właściwym* systemu przejść M nazywamy każde obliczenie nieskończone oraz każde takie obliczenie skończone $\gamma_0, \ldots, \gamma_n$, w którym γ_n jest konfiguracją końcową.

4. Obliczenie nieskończone nazywamy *rozbieżnym*. ∎

Sposób określenia (zdefiniowania) konfiguracji zależy od języka programowania.

Definiowanie semantyki operacyjnej dla konkretnego języka programowania

Sposób postępowania:

1. Definiujemy konfiguracje.

2. Definiujemy relację przejścia za pomocą reguł:

$$\frac{\text{przesłanki}}{\text{wniosek}}$$

3. Definiujemy semantykę za pomocą relacji przejścia.

Prosty język programowania PJP

Składnia

- $e ::= m \mid v \mid (e + e') \mid (e - e') \mid (e * e')$
- $b ::= t \mid b \vee b' \mid e = e' \mid \neg b$
- $c ::= \underline{\text{skip}} \mid v := e \mid c; c' \mid \underline{\text{if}}\ \ b\ \underline{\text{then}}\ \ c\ \underline{\text{else}}\ \ c' \mid \underline{\text{while}}\ \ b\ \underline{\text{do}}\ \ c$

Kategorie syntaktyczne

- $N = \{\ldots, -1, 0, 1, \ldots\}$
- Ide – zbiór identyfikatorów
- $B = \{tt, ff\}$
- $Aexp$ – wyrażenia arytmetyczne
- $Bexp$ – wyrażenia boolowskie
- Com – instrukcje

Wartościowanie zmiennych

- *Zbiór stanów* $\Sigma = [Ide \to N]$

Wyrażenia arytmetyczne

Konfiguracje

$\Gamma = \{\langle e, \sigma\rangle\}$, gdzie $\sigma \in \Sigma$.
Relacja $\langle e, \sigma\rangle \rightsquigarrow \langle e', \sigma\rangle$ oznacza, że jeden krok obliczenia e w stanie σ prowadzi do obliczenia wyrażenia e' w stanie σ (bez zmiany stanu).

Reguły

zmienna

1. $\langle v, \sigma\rangle \rightsquigarrow \langle \sigma(v), \sigma\rangle$

suma

1. $$\frac{\langle e_0, \sigma\rangle \rightsquigarrow \langle e_0', \sigma\rangle}{\langle e_0 + e_1, \sigma\rangle \rightsquigarrow \langle e_0' + e_1, \sigma\rangle}$$

2. $$\frac{\langle e_0, \sigma\rangle \rightsquigarrow \langle e_0', \sigma\rangle}{\langle m + e_0, \sigma\rangle \rightsquigarrow \langle m + e_0', \sigma\rangle}$$

3. $\langle m + m', \sigma\rangle \rightsquigarrow \langle n, \sigma\rangle$ (gdzie $n = m + m'$)

różnica

1. $$\frac{\langle e_0, \sigma\rangle \rightsquigarrow \langle e_0', \sigma\rangle}{\langle e_0 - e_1, \sigma\rangle \rightsquigarrow \langle e_0' - e_1, \sigma\rangle}$$

2. $$\frac{\langle e_0, \sigma\rangle \rightsquigarrow \langle e_0', \sigma\rangle}{\langle m - e_0, \sigma\rangle \rightsquigarrow \langle m - e_0', \sigma\rangle}$$

3. $\langle m - m', \sigma\rangle \rightsquigarrow \langle n, \sigma\rangle$
   (jeśli $n = m - m'$)

iloczyn

1. $$\frac{\langle e_0, \sigma\rangle \rightsquigarrow \langle e_0', \sigma\rangle}{\langle e_0 * e_1, \sigma\rangle \rightsquigarrow \langle e_0' * e_1, \sigma\rangle}$$

2. $$\frac{\langle e_0, \sigma\rangle \rightsquigarrow \langle e_0', \sigma\rangle}{\langle m * e_0, \sigma\rangle \rightsquigarrow \langle m * e_0', \sigma\rangle}$$

3. $\langle m * m', \sigma\rangle \rightsquigarrow \langle n, \sigma\rangle$
   (gdzie $n = m * m'$)

Uwagi

- Symbole +, -, * występują w podwójnym znaczeniu – jako konstruktory syntaktyczne i jako znaki operatorów arytmetycznych.

- Konfiguracje końcowe są to konfiguracje postaci $\langle m, \sigma \rangle$, gdzie m jest wynikiem obliczeń.

Semantyka wyrażeń arytmetycznych jest to wynik ich ewaluacji i jest zdefiniowana poprzez:

$$\mathbf{eval(e}, \sigma) = \mathbf{m} \Leftrightarrow \langle \mathbf{e}, \sigma \rangle \rightsquigarrow^* \langle \mathbf{m}, \sigma \rangle$$

Równoważność wyrażeń arytmetycznych definiujemy: $e_0 \sim e_1$ wtt

$$(\forall m \in N, \forall \sigma \in \Sigma \langle e_0, \sigma \rangle \rightsquigarrow^* \langle m, \sigma \rangle \Leftrightarrow \langle e_1, \sigma \rangle \rightsquigarrow^* \langle m, \sigma \rangle$$

Wyrażenia boolowskie

Konfiguracje

$\Gamma = \{\langle b, \sigma \rangle\}$, gdzie $\sigma \in \Sigma$.

Reguły

alternatywa

Dla alternatywy mamy kilka możliwości nie tylko co do kolejności przejść, ale także co to tego jakie przejścia mają wystąpić.

A. Ewaluacja całkowita

1. $$\frac{\langle b_0, \sigma\rangle \rightsquigarrow \langle b_0', \sigma\rangle}{\langle b_0 \vee b_1, \sigma\rangle \rightsquigarrow \langle b_0' \vee b_1, \sigma\rangle}$$

2. $$\frac{\langle b_0, \sigma\rangle \rightsquigarrow \langle b_0', \sigma\rangle}{\langle t \vee b_0, \sigma\rangle \rightsquigarrow \langle t \vee b_0', \sigma\rangle}$$

3. $\langle t \vee t', \sigma\rangle \rightsquigarrow \langle t'', \sigma\rangle$ (gdzie $t'' = t \vee t'$)

B. Ewaluacja lewostronna

1. $$\frac{\langle b_0, \sigma\rangle \rightsquigarrow \langle b_0', \sigma\rangle}{\langle b_0 \vee b_1, \sigma\rangle \rightsquigarrow \langle b_0' \vee b_1, \sigma\rangle}$$

2. $\langle tt \vee b_0, \sigma\rangle \rightsquigarrow \langle tt, \sigma\rangle$

3. $\langle ff \vee b_0, \sigma\rangle \rightsquigarrow \langle b_0', \sigma\rangle$

C. Ewaluacja prawostronna

analogicznie jak dla strony lewej lecz zastępujemy stronę prawą.

D. Ewaluacja równoległa

reguły dla B + reguły dla C

równość

1. $$\frac{\langle e_0, \sigma\rangle \rightsquigarrow \langle e_0', \sigma\rangle}{\langle e_0 = e_1, \sigma\rangle \rightsquigarrow \langle e_0' = e_1, \sigma\rangle}$$

2. $$\frac{\langle e_0, \sigma\rangle \rightsquigarrow \langle e_0', \sigma\rangle}{\langle m = e_0, \sigma\rangle \rightsquigarrow \langle m = e_0', \sigma\rangle}$$

3. $\langle m = m', \sigma\rangle \rightsquigarrow \langle t, \sigma\rangle$
   (gdzie t jest tt jeśli $m = m'$ a ff w przeciwnym przypadku)

negacja

1. $\dfrac{\langle b, \sigma\rangle \rightsquigarrow \langle b', \sigma\rangle}{\langle \neg b, \sigma\rangle \rightsquigarrow \langle \neg b', \sigma\rangle}$

2. $\langle \neg t, \sigma\rangle \rightsquigarrow \langle t', \sigma\rangle$ (gdzie $t' = \neg t$)

Semantyka wyrażeń boolowskich jest to wynik ich ewaluacji i jest zdefiniowana poprzez:

$$\mathbf{eval(b}, \sigma) = \mathbf{t} \Leftrightarrow \langle \mathbf{b}, \sigma \rangle \rightsquigarrow^* \langle \mathbf{t}, \sigma \rangle$$

Równoważność wyrażeń boolowskich definiujemy: $b_0 \sim b_1$

wtt

$$(\forall t \in B, \forall \sigma \in \Sigma \langle b_0, \sigma \rangle \rightsquigarrow^* \langle t, \sigma \rangle \Leftrightarrow \langle b_1, \sigma \rangle \rightsquigarrow^* \langle t, \sigma \rangle$$

Instrukcje

Konfiguracje

$\Gamma = \{\langle c, \sigma\rangle\} \cup \{\langle \epsilon, \sigma\rangle\}$
gdzie ϵ oznacza pusty program.

Relacja $\langle c, \sigma\rangle \rightsquigarrow \langle c', \sigma'\rangle$ oznacza, że jeden krok wykonania instrukcji c w stanie σ prowadzi do stanu σ' z resztą do wykonania c' (w stanie σ') instrukcji c.
Przyjmiemy następującą notację:
Niech $\sigma \in \Sigma$.

$$\sigma[m/v](x) = \begin{cases} \sigma(x) & \text{jeśli } x \neq v \\ m & \text{jeśli } x = v \end{cases}$$

Reguły

skip

1. $\langle \underline{\text{skip}}, \sigma \rangle \rightsquigarrow \langle \epsilon, \sigma \rangle$

przypisanie (:=)

1. $$\frac{\langle e, \sigma \rangle \rightsquigarrow^* \langle m, \sigma \rangle}{\langle v := e, \sigma \rangle \rightsquigarrow \langle \epsilon, \sigma[m/v] \rangle}$$

złożenie (;)

1. $$\frac{\langle c_0, \sigma\rangle \rightsquigarrow \langle c_0', \sigma'\rangle}{\langle c_0; c_1, \sigma\rangle \rightsquigarrow \langle c_0'; c_1, \sigma'\rangle}$$

2. $$\frac{\langle c_0, \sigma\rangle \rightsquigarrow \langle \epsilon, \sigma'\rangle}{\langle c_0; c_1, \sigma\rangle \rightsquigarrow \langle c_1, \sigma'\rangle}$$

Uwagi

1. W sformułowaniu reguły dla przypisania rozważamy **całą ewaluację** wyrażenia jako **jeden** krok wykonania. Definiując instrukcje zmieniliśmy pogląd co do rozmiaru kroku.

Przykład 1

Rozważmy pierwsze przejście dla wykonania

$$\langle z := x; (x := y; y := z), abc\rangle$$

$$\langle z := x; (x := y; y := z), abc\rangle \overset{\textit{złoż}.1}{\rightsquigarrow} \langle (x := y; y := z), aba\rangle$$

$$\langle z := x, abc\rangle \overset{\textit{przyp}.1}{\rightsquigarrow} \langle \epsilon, aba\rangle$$

$$\langle x, abc\rangle \overset{zm.1}{\rightarrow} \langle a, abc\rangle$$

■

if

Wykonanie if b then c else c' w stanie σ przebiega następująco:

1. Oblicz b w stanie σ.
   (a) Jeśli wynik jest tt wykonaj c w stanie σ.
   (b) Jeśli wynik jest ff wykonaj c' w stanie σ.

Reguły

1. $$\frac{\langle b, \sigma\rangle \rightsquigarrow^* \langle tt, \sigma\rangle}{\langle \text{if } b \text{ then } c \text{ else } c', \sigma\rangle \rightsquigarrow \langle c, \sigma\rangle}$$

2. $$\frac{\langle b, \sigma\rangle \rightsquigarrow^* \langle ff, \sigma\rangle}{\langle \text{if } b \text{ then } c \text{ else } c', \sigma\rangle \rightsquigarrow \langle c', \sigma\rangle}$$

while

Wykonanie while b do c w stanie σ przebiega następująco:

1. Oblicz b w stanie σ.

   (a) Jeśli wynik jest tt wykonaj c w stanie σ. Jeśli c się skończy dając stan σ', wykonaj while b do c w stanie σ'.

   (b) Jeśli wynik jest ff, wykonanie kończy się i stanem wynikowym jest σ.

Reguły

1. $$\frac{\langle b, \sigma\rangle \rightsquigarrow^* \langle tt, \sigma\rangle}{\langle \underline{\text{while}} \quad b \ \underline{\text{do}} \quad c, \sigma\rangle \rightsquigarrow \langle c; \underline{\text{while}} \quad b \ \underline{\text{do}} \quad c, \sigma\rangle}$$

2. $$\frac{\langle b, \sigma\rangle \rightsquigarrow^* \langle ff, \sigma\rangle}{\langle \underline{\text{while}} \quad b \ \underline{\text{do}} \quad c, \sigma\rangle \rightsquigarrow \langle \epsilon, \sigma\rangle}$$

Przykład 2

Rozważmy przykład programu obliczającego *silnia(x)*.

$$y := 1;\ \underbrace{\underline{\text{while}}\ \neg(x = 0)\ \underline{\text{do}}\ \overbrace{(y := y * x; x := x - 1)}^{C}}_{W}$$

oraz jego wykonanie ze stanu $(3, 5)$.

$$
\begin{array}{lll}
\langle y := 1; w, (3,5)\rangle & & \\
\quad \overset{przyp.1}{\rightsquigarrow} & \langle w, (3,1)\rangle & \\
\quad \overset{złoż.2}{\rightsquigarrow} & \langle c; w, (3,1)\rangle & (def.while\ \ 1) \\
\quad \overset{złoż.1}{\rightsquigarrow} & \langle (x := x-1; w), (3,3)\rangle & (złoż.2\ i\ przyp.1) \\
\quad \overset{złoż.2}{\rightsquigarrow} & \langle w, (2,3)\rangle & \\
\quad \overset{złoż.2}{\rightsquigarrow} & \langle c; w, (2,3)\rangle & (def.while\ \ 1) \\
\quad \overset{złoż.1}{\rightsquigarrow} & \langle x := x-1; w, (2,6)\rangle & (złoż.2\ i\ przyp.1) \\
\quad \overset{złoż.1}{\rightsquigarrow} & \langle w, (1,6)\rangle & \\
\quad \overset{złoż.1}{\rightsquigarrow} & \langle c; w, (1,6)\rangle & \\
\quad \overset{złoż.1}{\rightsquigarrow} & \langle x := x-1; w, (1,6)\rangle & \\
\quad \overset{złoż.1}{\rightsquigarrow} & \langle w, (0,6)\rangle & \\
\quad \overset{złoż.2}{\rightsquigarrow} & \langle \epsilon, (0,6)\rangle & (def.while\ \ 2)
\end{array}
$$

■

Semantyka instrukcji jest to stan będący wynikiem jej wykonania i jest zdefiniowana poprzez:

$$\mathbf{exec}(\mathbf{c}, \sigma) = \sigma' \Leftrightarrow \langle \mathbf{c}, \sigma \rangle \rightsquigarrow^* \langle \epsilon, \sigma' \rangle$$

Równoważność instrukcji definiujemy: $c_0 \sim c_1$ wtt

$$\forall \sigma, \sigma' . \langle c_0, \sigma \rangle \rightsquigarrow^* \langle \epsilon, \sigma' \rangle \Leftrightarrow \langle c_1, \sigma \rangle \rightsquigarrow^* \langle \epsilon, \sigma' \rangle$$

Własności

Z podanej semantyki języka PJP można wyprowadzić **formalnie** następujące własności.

Fakt 3
Relacja przejścia dla wyrażeń jest *deterministyczna*. ∎

Fakt 4
Jeśli zmienna v nie występuje po lewej stronie przypisania w c, to wykonanie c nie powoduje zmiany jej wartości. Tzn. jeśli $\langle c, \sigma\rangle \rightsquigarrow^* \langle \epsilon, \sigma'\rangle$ to $\sigma(v) = \sigma'(v)$. ∎

Fakt 5

Dla wszystkich stanów σ, σ'

$$\langle \underline{\text{while}}\ tt\ \underline{\text{do}}\ \underline{\text{skip}}, \sigma \rangle \not\rightarrow^* \langle \epsilon, \sigma' \rangle$$

■

Fakt 6

Niech $w = \underline{\text{while}}\ b\ \underline{\text{do}}\ c.\ \ w \sim \underline{\text{if}}\ b\ \underline{\text{then}}\ c; w\ \underline{\text{else}}\ \underline{\text{skip}}$ ■

Błędy dynamiczne

Załóżmy, że w definicji języka PJP mamy jeszcze operację dzielenia całkowitego (/), tj.:

$e ::= m \mid v \mid (e + e') \mid (e - e') \mid (e * e') \mid (e/e')$

Wymaga to dodatkowych reguł:

dzielenie

1. $$\frac{\langle e_0, \sigma\rangle \rightsquigarrow \langle e_0', \sigma\rangle}{\langle e_0/e_1, \sigma\rangle \rightsquigarrow \langle e_0'/e_1, \sigma\rangle}$$

2. $$\frac{\langle e_0, \sigma\rangle \rightsquigarrow \langle e_0', \sigma\rangle}{\langle m/e_0, \sigma\rangle \rightsquigarrow \langle m/e_0', \sigma\rangle}$$

3. $\langle m/m', \sigma\rangle \rightsquigarrow \langle n, \sigma\rangle$ (jeśli $m' \neq 0$ i $n = m/m'$)

Zauważmy, że mogą wystąpić konfiguracje takie, jak np. $\langle 5+(7/0), \sigma \rangle$, które nie są końcowe, ale nie mają następnika (nazwiemy je *zablokowanymi*). Zapewniliśmy:

$$\gamma \in T \Rightarrow \neg \exists \gamma' . \gamma \rightsquigarrow \gamma'$$

ale nie zapewniliśmy odwrotności. Zwykle zapewnia się to poprzez wykrywanie błędów wykonania i zakończenie obliczeń.

Formalnie: dodajemy do zbioru konfiguracji Γ konfigurację $\langle$błąd$\rangle$. Musimy też dodać stosowne reguły.

Zakładamy, że błędy się propagują.

4. $\langle m/m', \sigma\rangle \rightsquigarrow \langle \text{błąd}\rangle$ (jeśli $m' = 0$)

5. $$\frac{\langle e_0, \sigma\rangle \rightsquigarrow \langle \text{błąd}\rangle}{\langle e_0/e_1, \sigma\rangle \rightsquigarrow \langle \text{błąd}\rangle}$$

6. $$\frac{\langle e_0, \sigma\rangle \rightsquigarrow \langle \text{błąd}\rangle}{\langle m/e_0, \sigma\rangle \rightsquigarrow \langle \text{błąd}\rangle}$$

Analogiczne reguły do 5. i 6. piszemy dla pozostałych konstrukcji.