Semantyka języków programowania

MSUI, II rok

Ewa Madalińska

http://www.mimuw.edu.pl/ ewama/zajecia.htm
mail: ewama@mimuw.edu.pl
konsultacje pok. 4400

Bibliografia

[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.

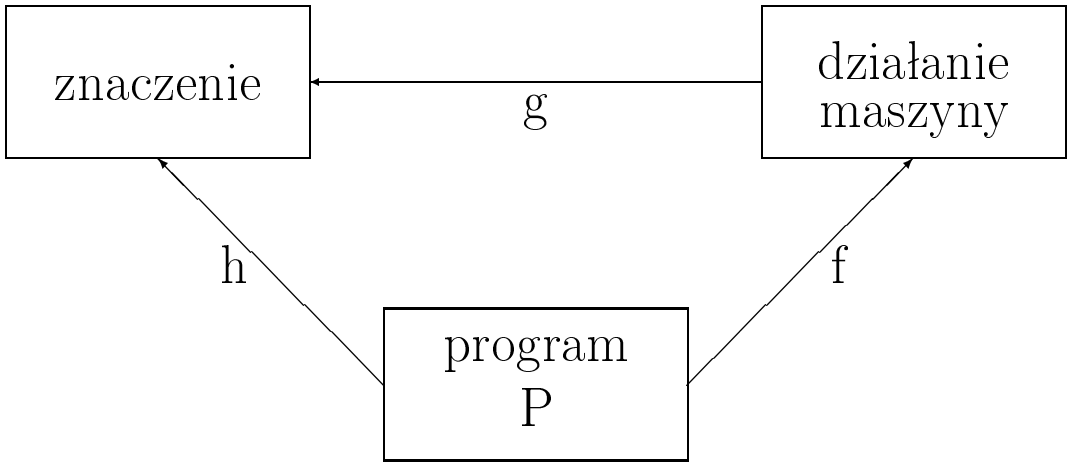
- Różnorodność konstrukcji maszyn prowadzi do oddzielenia opisu składni języka od jego maszynowych realizacji.

- 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).

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

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:

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

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

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

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

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:

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

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

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

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

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

$$\langle x, abc\rangle \overset{zm_v1}{\rightarrow} \langle a, abc\rangle$$

■

if

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

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

while

Wykonanie $\underline{\text{while}}\ \ b\ \underline{\text{do}}\ \ c$ 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 $\underline{\text{while}}\ \ b\ \underline{\text{do}}\ \ c$ w stanie σ'.
   (b) Jeśli wynik jest ff, wykonanie kończy się i stanem wynikowym jest σ.

Reguły

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

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

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 $(3, 5)$.

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

■

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'

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

■

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

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

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 $\langle \mathbf{błąd} \rangle$. Musimy też dodać stosowne reguły. Zakładamy, że błędy się propagują.

dzielenie

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

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

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

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

Semantyka denotacyjna PJP

Motywacje

- **semantyka operacyjna** bliska implementacji języka
- przejścia ściśle oparte na składni
- trudno porównać 2 programy napisane w różnych językach programowania
- sztywne definicje równoważności pomiędzy wyrażeniami arytmetycznymi, boolowskimi czy instrukcjami. Np. $c_0 \sim c_1$ wtt

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

- **semantyka denotacyjna** – jedynie równoważność instrukcji, bardziej bezpośredni sposób uchwycenia semantyki.

$c_0 \sim c_1$ wtt

$$\{(\sigma, \sigma') \mid \langle c_0, \sigma \rangle \rightsquigarrow^* \langle \epsilon, \sigma' \rangle\} = \{(\sigma, \sigma') \mid \langle c_1, \sigma \rangle \rightsquigarrow^* \langle \epsilon, \sigma' \rangle\}$$

Czyli $c_0 \sim c_1$ wtt c_0 i c_1 wyznaczają tę samą funkcję częściową na stanach.

Tak więc dla PJP mamy:

- wyrażenie arytmetyczne $e \in Aexp$ oznacza funkcję $\mathcal{A}[\![e]\!] : \Sigma \to N$
- wyrażenie boolowskie $b \in Bexp$ oznacza funkcję $\mathcal{B}[\![b]\!] : \Sigma \to B$
- instrukcja $c \in Com$ oznacza funkcję $\mathcal{C}[\![c]\!] : \Sigma \to \Sigma$

Definiujemy *funkcje semantyczne*

$$\mathcal{A} : Aexp \to [\Sigma \to N]$$

$$\mathcal{B} : Bexp \to [\Sigma \to B]$$

$$\mathcal{C} : Com \to [\Sigma \to \Sigma]$$

za pomocą indukcji strukturalnej. Będziemy używać λ-notacji.

Aexp

$$\mathcal{A}[\![n]\!] = \lambda\sigma \in \Sigma.n$$

$$\mathcal{A}[\![v]\!] = \lambda\sigma \in \Sigma.\sigma(v)$$

$$\mathcal{A}[\![e_0 + e_1]\!] = \lambda\sigma \in \Sigma.(\mathcal{A}[\![e_0]\!]\sigma + \mathcal{A}[\![e_1]\!]\sigma)$$

$$\mathcal{A}[\![e_0 - e_1]\!] = \lambda\sigma \in \Sigma.(\mathcal{A}[\![e_0]\!]\sigma - \mathcal{A}[\![e_1]\!]\sigma)$$

$$\mathcal{A}[\![e_0 * e_1]\!] = \lambda\sigma \in \Sigma.(\mathcal{A}[\![e_0]\!]\sigma * \mathcal{A}[\![e_1]\!]\sigma)$$

Bexp

$$\mathcal{B}[\![tt]\!] = \lambda\sigma \in \Sigma.tt$$

$$\mathcal{B}[\![f\,f]\!] = \lambda\sigma \in \Sigma.f\,f$$

$$\mathcal{B}[\![b_0 \vee b_1]\!] = \lambda\sigma \in \Sigma.\textbf{if } \mathcal{B}[\![b_0]\!]\sigma = tt \textbf{ then } tt \textbf{ else } \textbf{if } \mathcal{B}[\![b_1]\!]\sigma = tt \textbf{ then } tt \textbf{ else } f\,f$$

$$\mathcal{B}[\![e_0 = e_1]\!] = \lambda\sigma \in \Sigma.\textbf{if } \mathcal{A}[\![e_0]\!]\sigma = \mathcal{A}[\![e_1]\!]\sigma \textbf{ then } tt \textbf{ else } f\,f$$

$$\mathcal{B}[\![\neg b]\!] = \lambda\sigma \in \Sigma.\textbf{if } \mathcal{B}[\![b]\!]\sigma = tt \textbf{ then } f\,f \textbf{ else } tt.$$

Com

$$\mathcal{C}[\![\underline{\text{skip}}]\!] = \lambda\sigma \in \Sigma.\sigma$$

$$\mathcal{C}[\![v := e]\!] = \lambda\sigma \in \Sigma.\sigma[\mathcal{A}[\![e]\!]\sigma/v]$$

$$\mathcal{C}[\![c_1; c_2]\!] = \lambda\sigma \in \Sigma.\mathcal{C}[\![c_2]\!](\mathcal{C}[\![c_1]\!]\sigma)$$

$$\mathcal{C}[\![\underline{\text{if}}\ b\ \underline{\text{then}}\ c_1\ \underline{\text{else}}\ c_2]\!] = \lambda\sigma \in \Sigma.$$
$$\textbf{if}\ \mathcal{B}[\![b]\!]\sigma = tt\ \textbf{then}\ \mathcal{C}[\![c_1]\!]\sigma\ \textbf{else}\ \mathcal{C}[\![c_2]\!]\sigma$$

$$\mathcal{C}[\![\underline{\text{while}}\ b\ \underline{\text{do}}\ c]\!] =$$

Korzystamy z równoważności instrukcji (Fakt 6):

$$\underbrace{\underline{\text{while}}\ b\ \underline{\text{do}}\ c}_{w} = \underline{\text{if}}\ b\ \underline{\text{then}}\ c; \underbrace{\underline{\text{while}}\ b\ \underline{\text{do}}\ c}_{w}\ \underline{\text{else}}\ \underline{\text{skip}}$$

Równanie to ma postać:

$$w = \lambda\sigma \in \Sigma.\mathbf{if}\ \mathcal{B}[\![b]\!]\sigma = tt\ \mathbf{then}\ w(\mathcal{C}[\![c]\!]\sigma)\ \mathbf{else}\ \sigma \qquad (1)$$

$\mathcal{C}[\![\underline{\text{while}}\ b\ \underline{\text{do}}\ c]\!]$ = najmniejsze rozwiązanie w równania (1).
Prawa strona równania (1), dla ustalonych $\mathcal{B}[\![b]\!], \mathcal{C}[\![c]\!]$, – funkcja: $[\Sigma \to \Sigma] \to [\Sigma \to \Sigma]$:

$\mathcal{F}(\mathcal{B}[\![b]\!], \mathcal{C}[\![c]\!]) = \lambda w \in [\Sigma \to \Sigma].\lambda\sigma \in \Sigma.$
$\mathbf{if}\ \mathcal{B}[\![b]\!]\sigma = tt\ \mathbf{then}\ w(\mathcal{C}[\![c]\!]\sigma)\ \mathbf{else}\ \sigma$

$w = \mathcal{F}(\mathcal{B}[\![b]\!], \mathcal{C}[\![c]\!])(w)$

$\mathcal{C}[\![\underline{\text{while}}\ b\ \underline{\text{do}}\ c]\!] = fix(\mathcal{F}(\mathcal{B}[\![b]\!], \mathcal{C}[\![c]\!]))$

Podstawowe pojęcia matematyczne

Porządki zupełne i funkcje ciągłe

Definicja 7
Relację dwuargumentową $R \subseteq X^2$ nazywamy *porządkiem* w zbiorze X, jeżeli jest

1. zwrotna: $\forall x \in X.xRx$
2. antysymetryczna: $\forall x, y \in X.xRy \wedge yRx \Rightarrow x = y$
3. przechodnia: $\forall x, y, z \in X.xRy \wedge yRz \Rightarrow xRz$

Parę $(X, \sqsubseteq)$, gdzie $\sqsubseteq$ jest porządkiem w X nazywamy zbiorem uporządkowanym. ∎

Definicja 8

1. Element $x \in X$ nazywamy *minimalnym (maksymalnym)*, jeżeli nie istnieje $y \in X$ różny od x taki, że $y \sqsubseteq x (x \sqsubseteq y)$.
2. $x \in X$ jest elementem *najmniejszym (największym)*, jeżeli dla każdego $y \in X$ $x \sqsubseteq y$ $(y \sqsubseteq x)$. Element najmniejszy będziemy oznaczać $\bot$.
3. Niech $Y \subseteq X$ i $(X, \sqsubseteq)$. $x \in X$ jest *ograniczeniem górnym (dolnym)* zbioru Y, jeżeli dla każdego $y \in Y$ zachodzi $y \sqsubseteq x (x \sqsubseteq y)$.
4. Jeżeli w zbiorze wszystkich ograniczeń górnych (dolnych) zbioru Y istnieje element najmniejszy (największy), to nazywamy go *kresem górnym Y (kresem dolnym Y)*. Oznaczamy go $\sqcup Y (\sqcap Y)$. ∎

Definicja 9

Niech $(X, \sqsubseteq)$ będzie zbiorem uporządkowanym. Porządek $(X, \sqsubseteq)$ nazywamy *zupełnym* (w skrócie **cpo**), jeżeli każdy ciąg wstępujący $x_1 \sqsubseteq_x x_2 \sqsubseteq_x \ldots$ elementów X posiada kres górny w X. Piszemy wtedy $\sqcup_{i=1}^{\infty} x_i$. Mówimy, że $(X, \sqsubseteq)$ jest porządkiem zupełnym z "pinezką" ($\perp$), jeśli cpo ma element najmniejszy $\perp_x$). ∎

Definicja 10

1. Funkcję $f : X \to Y$ nazywamy monotoniczną, jeżeli dla każdej pary $x_1, x_2 \in X$ warunek $x_1 \sqsubseteq_X x_2$ pociąga za sobą $f(x_1) \sqsubseteq_Y f(x_2)$.
2. Jeżeli $(X, \sqsubseteq_X)$ i $(Y, \sqsubseteq_Y)$ są porządkami zupełnymi, to funkcję f nazywamy ciągłą, jeżeli dla każdego ciągu wstępującego $x_1 \sqsubseteq_X x_2 \sqsubseteq_X \ldots$ elementów zbioru X

$$f(\bigsqcup_{i=1}^{\infty} x_i) = \bigsqcup_{i=1}^{\infty} f(x_i)$$

∎

Definicja 11
Niech $H : D \to D$. $a \in D$ jest najmniejszym rozwiązaniem równania $x = H(x)$ (lub *najmniejszym punktem stałym* funkcji H), jeżeli $a = H(a)$ oraz dla każdego rozwiązania b, $a \sqsubseteq b$. Rozwiązanie to, jeżeli istnieje, będziemy oznaczać $fix(H)$. ∎

Twierdzenie 12
Jeżeli $(D, \sqsubseteq)$ jest porządkiem zupełnym z "pinezką", a H jest funkcją ciągłą, to $fix(H)$ istnieje oraz

$$fix(H) = \bigsqcup_{i=1}^{\infty} H^i(\bot)$$

∎

Dokonujemy następujących modyfikacji:

1. Funkcje częściowe na stanach zastępujemy całkowitymi funkcjami ciągłymi rozszerzając Σ o nowy element $\perp$. Otrzymujemy $\Sigma_\perp$ uporządkowany przez relację:

$$\forall \sigma . \perp \sqsubseteq \sigma,$$

gdzie $\perp$ – stan niezdefiniowany, tzn. wszystkie identyfikatory mają nieokreślone wartości. Zbiór $[\Sigma \to \Sigma]$ funkcji częściowych jest izomorficzny z $[\Sigma \to \Sigma_\perp]$ całkowitych funkcji ciągłych.

2. W konsekwencji rozszerzamy: N do $N_\perp$ oraz B do $B_\perp$, gdzie $\perp$ oznacza "brak wartości" .

3. Zmodyfikowane dziedziny semantyczne:

$$\mathcal{D}_{Aexp} = [\Sigma_\perp \to N_\perp]$$
$$\mathcal{D}_{Bexp} = [\Sigma_\perp \to B_\perp]$$
$$\mathcal{D}_{Com} = [\Sigma_\perp \to \Sigma_\perp]$$

Dla większej jasności symbol $\perp$ będziemy indeksować symbolem dziedziny, do której się odnosi: $\perp_N$, $\perp_B$, $\perp_\Sigma$.
$\perp_\Sigma = \lambda x \in Ide . \perp_N$.

Zastąpienie funkcji częściowych całkowitymi powoduje drobną modyfikację definicji.

Aexp

$$\mathcal{A}[\![n]\!] = \lambda\sigma \in \Sigma.n$$

$$\mathcal{A}[\![v]\!] = \lambda\sigma \in \Sigma.\sigma(v)$$

$$\mathcal{A}[\![e_0 + e_1]\!] = \lambda\sigma \in \Sigma.\textbf{if } \mathcal{A}[\![e_0]\!]\sigma = \perp_N \textbf{ then } \perp_N \\ \textbf{ else if } \mathcal{A}[\![e_1]\!]\sigma = \perp_N \textbf{ then } \perp_N \\ \textbf{ else } (\mathcal{A}[\![e_0]\!]\sigma + \mathcal{A}[\![e_1]\!]\sigma)$$

$$\mathcal{A}[\![e_0 - e_1]\!] = \lambda\sigma \in \Sigma.\textbf{if } \mathcal{A}[\![e_0]\!]\sigma = \perp_N \textbf{ then } \perp_N \\ \textbf{ else if } \mathcal{A}[\![e_1]\!]\sigma = \perp_N \textbf{ then } \perp_N \\ \textbf{ else } (\mathcal{A}[\![e_0]\!]\sigma - \mathcal{A}[\![e_1]\!]\sigma)$$

$$\mathcal{A}[\![e_0 * e_1]\!] = \lambda\sigma \in \Sigma.\textbf{if } \mathcal{A}[\![e_0]\!]\sigma = \perp_N \textbf{ then } \perp_N \\ \textbf{ else if } \mathcal{A}[\![e_1]\!]\sigma = \perp_N \textbf{ then } \perp_N \\ \textbf{ else } (\mathcal{A}[\![e_0]\!]\sigma * \mathcal{A}[\![e_1]\!]\sigma)$$

Bexp

$$\mathcal{B}[\![tt]\!] = \lambda\sigma \in \Sigma.tt$$

$$\mathcal{B}[\![ff]\!] = \lambda\sigma \in \Sigma.ff$$

$$\begin{aligned}\mathcal{B}[\![b_0 \vee b_1]\!] = \lambda\sigma \in \Sigma.&\textbf{if } \mathcal{B}[\![b_0]\!]\sigma = \perp_B \textbf{ then } \perp_B\\ &\textbf{else if } \mathcal{B}[\![b_1]\!]\sigma = \perp_B \textbf{ then } \perp_B\\ &\textbf{else if } \mathcal{B}[\![b_0]\!]\sigma = tt \textbf{ then } tt\\ &\textbf{else if } \mathcal{B}[\![b_1]\!]\sigma = tt \textbf{ then } tt \textbf{ else } ff\end{aligned}$$

$$\begin{aligned}\mathcal{B}[\![e_0 = e_1]\!] = \lambda\sigma \in \Sigma.&\textbf{if } \mathcal{A}[\![e_0]\!]\sigma = \perp_N \textbf{ then } \perp_B\\ &\textbf{else if } \mathcal{A}[\![e_1]\!]\sigma = \perp_N \textbf{ then } \perp_B\\ &\textbf{else if } \mathcal{A}[\![e_0]\!]\sigma = \mathcal{A}[\![e_1]\!]\sigma \textbf{ then } tt\\ &\textbf{else } ff\end{aligned}$$

$$\begin{aligned}\mathcal{B}[\![\neg b]\!] = \lambda\sigma \in \Sigma.&\textbf{if } \mathcal{B}[\![b_0]\!]\sigma = \perp_B \textbf{ then } \perp_B\\ &\textbf{else if } \mathcal{B}[\![b]\!]\sigma = tt \textbf{ then } ff \textbf{ else } tt.\end{aligned}$$

Com

$$\mathcal{C}[\![\underline{\text{skip}}]\!] = \lambda\sigma \in \Sigma.\sigma$$

$$\mathcal{C}[\![v := e]\!] = \lambda\sigma \in \Sigma.\textbf{if } \mathcal{A}[\![e]\!]\sigma = \perp_N \textbf{ then } \perp_\Sigma \textbf{ else } \sigma[\mathcal{A}[\![e]\!]\sigma/v]$$

$$\mathcal{C}[\![c_1; c_2]\!] = \lambda\sigma \in \Sigma.\mathcal{C}[\![c_2]\!](\mathcal{C}[\![c_1]\!]\sigma)$$

$$\mathcal{C}[\![\underline{\text{if}}\ b\ \underline{\text{then}}\ c_1\ \underline{\text{else}}\ c_2]\!] = \lambda\sigma \in \Sigma.$$
$$\textbf{if } \mathcal{B}[\![b]\!]\sigma = \perp_B \textbf{ then } \perp_\Sigma \textbf{ else}$$
$$\textbf{if } \mathcal{B}[\![b]\!]\sigma = tt \textbf{ then } \mathcal{C}[\![c_1]\!]\sigma \textbf{ else } \mathcal{C}[\![c_2]\!]\sigma$$

Wróćmy do instrukcji $\underline{\text{while}}\ b\ \underline{\text{do}}\ c$.

Zgodnie z twierdzeniem 12 mamy:

$$\mathcal{C}[\![\underline{\text{while}}\ b\ \underline{\text{do}}\ c]\!] = \bigsqcup_{i=1}^{\infty} f^i(\bot)$$

,
gdzie $f = \mathcal{F}(\mathcal{B}[\![b]\!], \mathcal{C}[\![c]\!]) : \mathcal{D}_{Com} \to \mathcal{D}_{Com}$,
gdzie $\mathcal{D}_{Com} = [\Sigma_\bot \to \Sigma_\bot]$

$\bot \in \mathcal{D}_{Com}$ jest zdefiniowana $\lambda\sigma \in \Sigma.\bot_\Sigma$.

Obliczymy 2 kolejne aproksymacje $\mathcal{C}[\![\underline{\text{while}}\ b\ \underline{\text{do}}\ c]\!]$.

$$\begin{aligned} f(\bot) &= \lambda\sigma \in \Sigma.\textbf{if } \mathcal{B}[\![b]\!]\sigma = \bot_B \textbf{ then } \bot_\Sigma \textbf{ else} \\ &\quad \textbf{if } \mathcal{B}[\![b]\!]\sigma = ff \textbf{ then } \sigma \textbf{ else } \bot(\mathcal{C}[\![c]\!]\sigma) \\ &= \lambda\sigma \in \Sigma.\textbf{if } \mathcal{B}[\![b]\!]\sigma = \bot_B \textbf{ then } \bot_\Sigma \textbf{ else} \\ &\quad \textbf{if } \mathcal{B}[\![b]\!]\sigma = ff \textbf{ then } \sigma \textbf{ else } \bot_\Sigma. \end{aligned}$$

$$\begin{aligned} f^2(\bot) &= \lambda\sigma \in \Sigma.\textbf{if } \mathcal{B}[\![b]\!]\sigma = \bot_B \textbf{ then } \bot_\Sigma \textbf{ else} \\ &\quad \textbf{if } \mathcal{B}[\![b]\!]\sigma = ff \textbf{ then } \sigma \textbf{ else } f(\bot)(\mathcal{C}[\![c]\!]\sigma) \\ &= \lambda\sigma \in \Sigma.\textbf{if } \mathcal{B}[\![b]\!]\sigma = \bot_B \textbf{ then } \bot_\Sigma \textbf{ else} \\ &\quad \textbf{if } \mathcal{B}[\![b]\!]\sigma = ff \textbf{ then } \sigma \textbf{ else} \\ &\quad \textbf{if } \mathcal{B}[\![b]\!](\mathcal{C}[\![c]\!]\sigma) = ff \textbf{ then } \mathcal{C}[\![c]\!]\sigma \textbf{ else } \bot_\Sigma. \end{aligned}$$

Równoważność semantyki operacyjnej i denotacyjnej dla PJP

Twierdzenie 13

1. Dla dowolnego wyrażenia $e \in Aexp$

$$\mathcal{A}[\![e]\!] = \{(\sigma, n) \mid \langle e, \sigma \rangle \rightsquigarrow^* \langle n, \sigma \rangle\}$$

2. Dla dowolnego $b \in Bexp$

$$\mathcal{B}[\![b]\!] = \{(\sigma, t) \mid \langle b, \sigma \rangle \rightsquigarrow^* \langle t, \sigma \rangle\}$$

3. Dla każdej instrukcji $c \in Com$

$$\mathcal{C}[\![c]\!] = \{(\sigma, \sigma') \mid \langle c, \sigma \rangle \rightsquigarrow^* \langle \epsilon, \sigma' \rangle\}$$

■

Przykład 3

Zdefiniować prosty język funkcyjny (bez rekursji).

Składnia

- $n ::=$ stała
- $v ::=$ identyfikator zmiennych
- $f ::=$ identyfikator funkcji
- $e ::= n \mid v \mid e_1 + e_2 \mid fe \mid let\ f = \lambda v.e\ in\ e_1$

Dziedziny syntaktyczne

- $Ide = VIde + FIde$
- $Aexp$

Dziedziny semantyczne

- $N = \{\ldots, -1, 0, 1, 2, \ldots\}$
- $\Sigma = [VIde \to N]$
- $F\Sigma = [FIde \to [N \to N]]$

Funkcja semantyczna

- $\mathcal{A} : Aexp \to [F\Sigma \to [\Sigma \to N]]$

Równania semantyczne

- $\mathcal{A}[\![n]\!] = \lambda\tau \in F\Sigma.\lambda\sigma \in \Sigma.n$

- $\mathcal{A}[\![v]\!] = \lambda\tau \in F\Sigma.\lambda\sigma \in \Sigma.\sigma(v)$

- $\mathcal{A}[\![e_1 + e_2]\!] = \lambda\tau \in F\Sigma.\lambda\sigma \in \Sigma.\mathcal{A}[\![e_1]\!]\tau\sigma + \mathcal{A}[\![e_2]\!]\tau\sigma$

- $\mathcal{A}[\![fe]\!] = \lambda\tau \in F\Sigma.\lambda\sigma \in \Sigma.\tau(f)(\mathcal{A}[\![e]\!]\tau\sigma)$

- $\mathcal{A}[\![let\ f = \lambda v.e\ in\ e_1]\!] = \lambda\tau \in F\Sigma.\lambda\sigma \in \Sigma.$ $\mathcal{A}[\![e_1]\!](\tau[\lambda x \in N.\mathcal{A}[\![e]\!]\tau(\sigma[x/v])/f])\sigma$

■

Semantyka aksjomatyczna

Motywacje

1. Punkt widzenia użytkownika języka programowania na problem znaczenia przypisywanego programom zwykle jest różny od punktu widzenia projektanta języka czy osoby piszącej kompilator.
2. Potrzeba badania własności programu i jego poprawności:
   (a) jeżeli dane wejściowe programu p spełniają własność α i program poprawnie zakończy obliczenia, to dane wyjściowe spełniają własność β – *częściowa poprawność* p względem α i β.
   (b) jeżeli dane wejściowe programu p spełniają własność α, to program prawidłowo zakończy obliczenia i dane wyjściowe spełniają własność β – *całkowita poprawność* p względem α i β.
3. Potrzeba skonstruowania języka do wyrażania takich stwierdzeń oraz dostarczenia metod ich weryfikacji bez odwoływania się do specyfiki urządzenia wykonującego obliczenia – metody oparte na logice.

Logika Hoare'a

- **Język**
  formuły

$$\{\alpha\}\ c\ \{\beta\},$$

  gdzie α, β – formuły logiki I rzędu,
  α – nazywamy *warunkiem wstępnym*,
  β – *warunkiem końcowym*,
  c – instrukcja (program).

  Wyrażenie takie nazywamy *specyfikacją* lub *trójką Hoare'a*.

  W zależności od przyjętych definicji, formułę taką będziemy rozumieć w sensie 2a lub 2b.

Przykład 4
Niech

$$c = \underline{\text{while}}\ tt\ \underline{\text{do}}\ \underline{\text{skip}}$$

Wykonanie c nie kończy się nigdy. Zgodnie z interpretacją *częściowej poprawności* (2a) następująca specyfikacja jest prawdziwa:

$$\{tt\}\ c\ \{ff\} \tag{2}$$

Co więcej, ponieważ c pętli się, specyfikacja

$$\{\alpha\}\ c\ \{\beta\} \tag{3}$$

jest prawdziwa w sensie 2a dla dowolnych α i β.

Oczywiście nie zachodzi to dla (3) w sensie 2b.
■

- **Język logiki I rzędu**

1. V – zbiór zmiennych indywiduowych;

2. F_m – zbiór m-argumentowych symboli funkcyjnych, $m = 0, 1, \ldots$;
   F_0 – oznacza stałe indywiduowe;

3. tt, ff – stałe logiczne;

4. P_m – zbiór m-argumentowych symboli predykatywnych, $m = 1, \ldots$;

5. zbiór spójników logicznych i kwantyfikatorów:
   $\wedge, \vee, \Rightarrow, \neg, \exists, \forall$;

6. W standardowy sposób tworzymy zbiory:
   termów (TR) i
   formuł (FR).

- **Interpretacja**

1. D – dziedzina interpretacji.

2. $TF = \{tt, ff\}$ – zbiór wartości logicznych.

3. *Wartościowanie* zmiennych – funkcja
   $val : V \to D$.
   $Val = [V \to D]$

4. Interpretacja I – funkcja, która przyporządkowuje:

   - $a \in F_0 \Rightarrow a_I \in D$

   - $x \in V \Rightarrow val(x) \in D$

   - $f \in F_m \Rightarrow f_I : D^m \to D, m = 1, 2, \ldots$

   - $q \in P_m \Rightarrow q_I : D^m \to \{tt, ff\}, m = 1, 2, \ldots$ (relacja na D^m)

• Programy

SP – zbiór specyfikacji.

Specyfikacja – napis formułujący pewne stwierdzenie o programie. Aby związać języki programowania z językiem logiki I rzędu przyjmujemy:

1. $V = Ide$.
2. w konsekwencji przyjmujemy $Val = \Sigma$.
   Czyli $\Sigma = [Ide \to D]$.
   A więc $I(x)(\sigma) = \sigma(x)$ dla $x \in V = Ide$.
3. Rozszerzamy interpretację I na termy, formuły, programy i specyfikacje względem dowolnego stanu $\sigma \in \Sigma$.

- $\alpha \in FR \cup SP$ jest prawdziwa w interpretacji I i w stanie σ, jeżeli $\alpha_I(\sigma) = tt$.

- I jest modelem $\alpha \in FR \cup SP$, jeżeli α jest prawdziwa dla każdego $\sigma \in \Sigma$.

- I jest modelem dla zbioru formuł lub specyfikacji, jeżeli jest modelem dla każdej formuły lub specyfikacji tego zbioru.

- Rozważamy logikę Hoare'a nad zadanym modelem $\mathcal{M} = \{\alpha \in FR : \models_I \alpha\}$.
- System wnioskowania o programach

$$H = (A_{SP}, R_{SP})$$

Wymagamy, aby

1. Własności, które możemy udowodnić w systemie są prawdziwe w rzeczywistości.
2. Fakty rzeczywiście prawdziwe o programach (opisywane przez specyfikacje) dają się w tym systemie udowodnić.

- Związek między systemem wnioskowania o programach dla języka programowania a jego semantyką – semantyka jest modelem dla tego systemu.

- Język programów iteracyjnych

$$c \ ::= \ \underline{\text{skip}} \mid v := t \mid c; c' \mid \underline{\text{if}} \ \ b \ \underline{\text{then}} \ \ c \ \underline{\text{else}} \ \ c' \mid \underline{\text{while}} \ \ b \ \underline{\text{do}} \ \ c$$

  gdzie $t \in TR$ i b jest formułą otwartą.

- Rozszerzenie dowolnej interpretcji I na instrukcje języka programowania definiujemy:
  Dla każdej instrukcji $c \in Com$ i stanu $\sigma \in \Sigma$

$$I(c)(\sigma) = \mathcal{C}[\![c]\!]\sigma$$

  lub równoważnie

$$I(c)(\sigma) = \mathbf{exec}(\mathbf{c}, \sigma)$$

Częściowa poprawność programów

System Hoare'a dla programów iteracyjnych

1. $\underline{\text{skip}}$

   $\{\alpha\}\underline{\text{skip}}\{\alpha\}$

2. $:=$

   $\{\alpha(t/v)\}v := t\{\alpha\}$

3. $;$

   $$\frac{\{\alpha\}c_1\{\beta\},\ \{\beta\}c_2\{\gamma\}}{\{\alpha\}c_1; c_2\{\gamma\}}$$

4. if

$$\frac{\{\alpha \wedge b\}c_1\{\beta\},\ \{\alpha \wedge \neg b\}c_2\{\beta\}}{\{\alpha\}\underline{\text{if}}\ b\ \underline{\text{then}}\ c_1\ \underline{\text{else}}\ c_2\{\beta\}}$$

5. while

$$\frac{\{\alpha \wedge b\}c\{\alpha\}}{\{\alpha\}\underline{\text{while}}\ b\ \underline{\text{do}}\ c\{\alpha \wedge \neg b\}}$$

6. **Reguła konsekwencji**

$$\frac{\alpha \Rightarrow \alpha_1, \{\alpha_1\}c\{\beta_1\}, \beta_1 \Rightarrow \beta}{\{\alpha\}c\{\beta\}}$$

Dla języka PJP mamy:

1. Aexp=TR
2. Bexp= zbiór formuł otwartych
3. Interpretacja – arytmetyka liczb całkowitych.

Przykład 5
Niech $c \equiv x := x + 1;\ y := y + 1$

Chcemy udowodnić $\{x = y\}\ c\ \{x = y\}$.

Dowód:
z **(:=)** mamy

$\{(x = y)[y + 1/y]\}\ y := y + 1\ \{x = y\}$
$\{(x = y + 1\}\ y := y + 1\ \{x = y\}$

z **(:=)** mamy

$\{(x = y + 1)[x + 1/x]\}\ x := x + 1\ \{x = y + 1\}$
$\{x + 1 = y + 1\}\ x := x + 1\ \{x = y + 1\}$

z **(;)** mamy

$\{x + 1 = y + 1\}\ x := x + 1;\ y := y + 1\ \{x = y\}$

z **reguły konsekwencji** i faktu

$(x = y) \Rightarrow (x + 1 = y + 1)$ mamy

$\{x = y\}\ x := x + 1;\ y := y + 1\ \{x = y\}$ ∎

Przykład 6
$c \equiv \underline{\text{if}}\ x \geq y\ \underline{\text{then}}\ m := x\ \underline{\text{else}}\ m := y$

Pokazać, że $\{tt\}\ c\ \{m = max(x, y)\}$.

Dowód:
z **(:=)**

$\{y = max(x, y)\}\ m := y\ \{m = max(x, y)\}$

z **(:=)**

$\{x = max(x, y)\}\ m := x\ \{m = max(x, y)\}$

z **reguły konsekwencji** i faktu

$(x \geq y) \Rightarrow (x = max(x, y))$ mamy

$(x \geq y)\ m := x\ \{m = max(x, y)\}$

z **reguły konsekwencji** i faktu

$(\neg(x \geq y)) \Rightarrow (y = max(x, y))$ mamy

$(x < y)\ m := y\ \{m = max(x, y)\}$

z **if** dostajemy

$\{tt\}\ c\ \{m = max(x, y)\}$. ∎

Przykład 7
$w \equiv \underline{\text{while}}\ x > 0\ \underline{\text{do}}\ y := x * y;\ x := x - 1$

Pokazać, że $\{x = n \wedge n \geq 0 \wedge y = 1\}\ w\ \{y = n!\}$

Weźmy niezmiennik

$$\mathcal{I} \equiv (y * x! = n! \wedge x \geq 0)$$

Pokażemy, że $\mathcal{I}$ jest rzeczywiście niezmiennikiem, tzn.

$\{\mathcal{I} \wedge b\}\ c\ \{\mathcal{I}\}$

z **(:=)**

$\{y * (x-1)! = n! \wedge x - 1 \geq 0\}\ x := x - 1\ \{y * x! = n! \wedge x \geq 0\}$

z **(:=)**

$\{x * y * (x-1)! = n! \wedge x - 1 \geq 0\}\ y := x * y\ \{y * (x-1)! = n! \wedge x - 1 \geq 0\}$

z **reguły konsekwencji** i faktu

$(y * x! = n! \wedge x \geq 0 \wedge x > 0) \Rightarrow (x * y * (x-1)! =$

$n! \wedge x - 1 \geq 0)$

oraz **(;)** mamy

$\{\mathcal{I} \wedge b\}\ c\ \{\mathcal{I}\}$

z **while** dostajemy

$\{y * x! = n! \wedge x \geq 0\}\ w\ \{y * x! = n! \wedge x \geq 0 \wedge x \leq 0\}$

z **reguły konsekwencji** i faktu

$(x = n \wedge n \geq 0 \wedge y = 1) \Rightarrow (y * x! = n! \wedge x \geq 0)$

oraz faktu

$(y * x! = n! \wedge x \geq 0 \wedge x \leq 0) \Rightarrow (y * x! = n! \wedge x = 0) \Rightarrow (y = n!)$

dostajemy

$\{x = n \wedge n \geq 0 \wedge y = 1\}\ w\ \{y = n!\}$ ∎

Własności systemu H

Poprawność

Formalnie:

Definicja 14
System T wnioskowania o programach nazywamy *niesprzecznym* (lub *poprawnym*) względem klasy interpretacji INT, jeżeli dla każdej interpretacji $I \in INT$ i każdej specyfikacji Φ

$$\vdash_{TI} \Phi \Rightarrow \models_I \Phi.$$

■

Przy przyjętych założeniach mamy

Twierdzenie 15
System wnioskowania H jest poprawny. ■

Zupełność

Formalnie:

Definicja 16

System T wnioskowania o programach nazywamy *zupełnym* względem klasy interpretacji INT, jeżeli dla każdej interpretacji $I \in INT$ i każdej specyfikacji Φ

$$\models_I \Phi \Rightarrow \vdash_{TI} \Phi$$

■

Wyrażalność FR względem PR

najsłabszy warunek wstępny

$$wp_I(c,\beta) = \{\sigma \in \Sigma : \text{jeśli } I(c)(\sigma) \text{ jest zdefiniowane, to } \models_I \beta(I(c)(\sigma))\}.$$

Interpretacja I jest *wyrażalna w FR względem PR*, jeżeli dla każdego $c \in PR$ istnieje $\alpha \in FR$ taka, że $\{\sigma : \models_I \alpha(\sigma)\} = wp_I(c,\beta)$.

Twierdzenie 17
System wnioskowania H jest relatywnie zupełny w klasie interpretacji wyrażalnych dla (FR, PR). ∎

Niech I będzie standardową interpretacją dla PJP, a FR językiem formuł generowanym przez wyrażenia języka PJP. Wtedy

Twierdzenie 18
Interpretacja I jest wyrażalna w FR względem PJP.∎

Całkowita poprawność programów

Formalnie:
Dla interpretacji I i dowolnego stanu $\sigma \in \Sigma$

1. *Poprawność cześciowa*
   $I(\{\alpha\}\ c\ \{\beta\})(\sigma) = tt \stackrel{df}{\equiv}$ jeżeli $\models_I \alpha(\sigma)$ i $I(c)(\sigma)$ jest zdefiniowane, to $\models_I \beta(I(c)(\sigma))$
2. *Poprawność całkowita*
   $I(\{\alpha\}\ c\ \{\beta\})(\sigma) = tt \stackrel{df}{\equiv}$ jeżeli $\models_I \alpha(\sigma)$, to $I(c)(\sigma)$ jest zdefiniowane i $\models_I \beta(I(c)(\sigma))$

Poprawność całkowita

=

Poprawność częściowa

+

Własność stopu

Całkowita poprawność programów

System Hoare'a dla programów iteracyjnych

Aksjomaty i reguły 1.–4.,6. oraz

5'. while

$$\frac{\begin{array}{c}\{\alpha \wedge b\}\ c\ \{\alpha\} \\ \{\alpha \wedge b \wedge t = z\}\ c\ \{t < z\} \\ \alpha \Rightarrow t \geq 0\end{array}}{\{\alpha\}\underline{\text{while}}\ b\ \underline{\text{do}}\ c\{\alpha \wedge \neg b\}}$$

gdzie t jest wyrażeniem całkowitoliczbowym, a z zmienną nie występującą w α, b, t ani c. Wyrażenie t nazywamy *funkcją ograniczającą*.

Uwaga 2

Chociaż aksjomaty i reguły 1.-4. i 6. mają tę samą postać, co w systemie poprawności częściowej to występujące w nich specyfikacje $\{\alpha\}\ c\ \{\beta\}$ mają inne znaczenie, mianowicie są interpretowane w sensie poprawności całkowitej.

Metody wyznaczania niezmiennika

"Teoria balona" Griesa.

$$\{\alpha\}\underline{\text{while}}\ b\ \underline{\text{do}}\ c\{\beta\}$$

Niezmiennik $\mathcal{I}$ jest formułą, która jest prawdziwa przed i po każdej iteracji.

Tak więc, zbiór stanów reprezentowany przez $\mathcal{I}$ musi zawierać zarówno stany początkowe jak i stany końcowe.

Należy więc tak osłabić warunek końcowy β, aby objął on także stany początkowe ("nadmuchanie balona").

Każda iteracja powinna "wypuszczać trochę powietrza z balona" tak, aby po ostatniej zredukował się z powrotem do β.

Osłabianie formuły β.

1. **Usunięcie koniunktu**.
   Np. $A \wedge B \wedge C$ może być osłabiony do $A \wedge B$. Wtedy $\neg C$ przyjmujemy za warunek pętli.

2. **Zastąpienie stałej przez zmienną.**
   Np.x=min b[1..10] może być osłabione do x=min b[1..i]$\wedge 1 \leq i \leq 10$, gdzie i jest nową zmienną.

3. **Rozszerzenie zakresu zmiennej.**
   Np. $5 \leq i \leq 10$ może być osłabiony do $0 \leq i \leq 10$.

4. **Dodanie dysjunktu**.
   Np. A może byc osłabiony do $A \vee B$.

Przykład 8
Rozważmy program DIV obliczający iloraz całkowity i resztę z dzielenia liczb całkowitych nieujemnych x i y.

$q := 0;\ r := x;$
$\underline{\text{while}}\ r \geq y\ \underline{\text{do}}\ (r := r - y;\ q := q + 1)$

Rozważmy formuły:

$$\{x \geq 0 \wedge y \geq 0\} DIV \{q * y + r = x \wedge 0 \leq r < y\} \quad (4)$$

$$\{x \geq 0 \wedge y > 0\} DIV \{q * y + r = x \wedge 0 \leq r < y\} \quad (5)$$

Zauważmy, że jeśli udowodnimy prawdziwość specyfikacji (4), to ponieważ

$$(x \geq 0 \wedge y > 0) \Rightarrow (x \geq 0 \wedge y \geq 0),$$

więc z **reguły konsekwencji** otrzymamy prawdziwość (5).

Dowód dla poprawności częściowej

Weźmy jako niezmiennik pętli

$$\mathcal{I} \equiv q * y + r = x \wedge r \geq 0$$

Udowodnimy następujące trzy fakty:

$$\{x \geq 0 \wedge y \geq 0\}\ q := 0;\ r := x;\ \{\mathcal{I}\} \tag{6}$$

$$\{\mathcal{I} \wedge r \geq y\}\ r := r - y;\ q := q + 1\ \{\mathcal{I}\} \tag{7}$$

$$\mathcal{I} \wedge \neg(r \geq y) \Rightarrow (q * y + r = x \wedge 0 \leq r < y) \tag{8}$$

Ad.(6) Stosując dwukrotnie aksjomat dla **(:=)** mamy

$$\{q * y + x = x \wedge x \geq 0\}\ r := x\ \{\mathcal{I}\}$$

oraz

$\{0{*}y{+}x = x{\wedge}x \geq 0\}\ q := 0\ \{q{*}y{+}x = x{\wedge}x \geq 0\}$

więc z **(;)** otrzymujemy

$$\{0 * y + x = x \wedge x \geq 0\}\ q := 0;\ r := x\ \{\mathcal{I}\}$$

co wraz z faktem

$$(x \geq 0 \wedge y \geq 0) \Rightarrow (0 * y + x = x \wedge x \geq 0)$$

daje (6).

Ad.(7) Stosując dwukrotnie aksjomat dla **(:=)** mamy

$$\{(q+1)*y+r=x \wedge r \geq 0\}\ q := q+1\ \{\mathcal{I}\}$$

oraz

$$\begin{array}{l}\{(q+1)*y+(r-y)=x \wedge (r-y) \geq 0\}\\ r := r-y\\ \{(q+1)*y+r=x \wedge r \geq 0\}\end{array}$$

więc z **(;)** otrzymujemy

$$\begin{array}{l}\{(q+1)*y+(r-y)=x \wedge (r-y) \geq 0\}\\ r := r-y;\ q := q+1\\ \{\mathcal{I}\}\end{array}$$

co wraz z faktem

$$\begin{array}{l}(\mathcal{I} \wedge r \geq y) \Rightarrow\\ ((q+1)*y+(r-y)=x \wedge (r-y) \geq 0)\end{array}$$

daje (7).

Ad.(8) Oczywiste.

Ad.(4) (4) otrzymujemy z (6), (7) i (8) stosując regułę (while) oraz (;).

Dowód dla poprawności całkowitej

Udowodnimy specyfikację (5)[1]. Weźmy jako niezmiennik pętli

$$\mathcal{I}' \equiv \mathcal{I} \wedge y > 0$$

i niech funkcją ograniczającą będzie

$$t \equiv r$$

Mamy pokazać, że

$$\{x \geq 0 \wedge y > 0\}\ q := 0;\ r := x;\ \{\mathcal{I}'\} \tag{9}$$

$$\{\mathcal{I}' \wedge r \geq y\}\ r := r - y;\ q := q + 1\ \{\mathcal{I}'\} \tag{10}$$

$$\{\mathcal{I}' \wedge r \geq y \wedge r = z\} \tag{11}$$
$$r := r - y;\ q := q + 1\ \{r < z\}$$

$$\mathcal{I}' \Rightarrow r \geq 0 \tag{12}$$

$$\mathcal{I}' \wedge \neg(r \geq y) \Rightarrow (q * y + r = x \wedge 0 \leq r < y) \tag{13}$$

Dowody (9), (10) i (13) są analogiczne do (6), (7) i (8).

[1]Specyfikacja (4) w tym przypadku nie zachodzi,zauważmy, że program się pętli dla $y = 0$.

Ad.(11) z **(:=)** i **(;)** mamy

$$\{r < z\}\ q := q + 1\ \{r < z\}$$

i

$$\{(r - y) < z\}\ r := r - y\ \{r < z\}.$$

Ale

$$(\mathcal{I} \wedge y > 0 \wedge r \geq y \wedge r = z) \Rightarrow ((r - y) < z)$$

więc (11) zachodzi z reguły konsekwencji.

Ad.(12) Oczywiste.

Ad.(5) (5) otrzymujemy z (9)– (13) stosując regułę (while) oraz (;). ∎

Twierdzenie 19
Problem stopu jest problemem nierozstrzygalnym. ∎

Przykład programu dla którego nie znaleziono dotąd dowodu, że się kończy ani kontrprzykładu:

```
while x ≠ 1   do
if xparzyste  then x := x/2
              else x := 3 * x + 1
```

Deklaracje lokalne – bloki

- Chcemy wydzielić deklaracje jako osobną kategorię syntaktyczną.
- Chcemy aby deklaracja miała znaczenie sama w sobie, a nie tylko w powiązaniu z instrukcją.
- Rozszerzamy składnię języka o deklaracje.

Składnia

- $e ::= m \mid v \mid (e + e') \mid (e - e') \mid (e * e')$
- $b ::= t \mid b \vee b' \mid e = e' \mid \neg b$
- $d ::= \ \underline{\text{var}}\ v \mid d_1; d_2$
- $c ::= \underline{\text{skip}} \mid v := e \mid c; c' \mid \underline{\text{if}}\ \ b\ \underline{\text{then}}\ \ c\ \underline{\text{else}}\ \ c' \mid \underline{\text{while}}\ \ b\ \underline{\text{do}}\ \ c \mid \ \underline{\text{begin}}\ d; c\ \underline{\text{end}}$

Przykład 9

```
begin
  var x; var y;
  x:=1;
  y:=2;
  begin
    var x; var z;
    x:=4;
    z:=x+y;
    y:=y+1;
    print(z)
  end;
  x:=x+y;
  print(x)
end
```

Co się wydrukuje?

6 4

■

Semantyka denotacyjna

Kategorie syntaktyczne

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

Dziedziny semantyczne

- $Loc = \{l_0, l_1, \ldots\}$ – zbiór lokacji (nieskończony).
- $Sv = N$ – zależna od języka dziedzina wartości, które mogą być pamiętane w lokacjach.
- $S = [Loc \to Sv + \{\ \underline{\text{wolna}}\ \}]$ – stany (pamięć)
- $Dv = Loc + \{\ \underline{\text{puste}}\ \}$ – zależna od języka dziedzina wartości, które mogą być związane z (oznaczane) identyfikatorem.

- $Env = [Ide \to Dv]$ – środowiska.

- Będziemy korzystać z funkcji

$$new : S \to Loc \times S$$

dla $s \in S$

$$new(s) = \begin{cases} (l_i, s[0/l_i]) & \exists j.s(l_j) = \underline{\text{wolna}} \wedge \\ & i = min\{j \mid s(l_j) = \underline{\text{wolna}}\} \\ (l_0, s[0/l_0]) & \text{w p.p.} \end{cases}$$

Uwaga 3
Być może zainicjalizowanie zmiennej przez *new* jako $\perp_N$ byłoby bardziej intuicyjne. Sposób inicjalizacji nie odgrywa tu istotnej roli. ∎

Funkcje semantyczne

$\mathcal{A} : Aexp \to [Env \to [S \to N_\perp]]$

$\mathcal{B} : Bexp \to [Env \to [S \to B_\perp]]$

$\mathcal{D} : Decl \to [Env \to [S \to Env \times S]]$

$\mathcal{C} : Com \to [Env \to [S \to S_\perp]]$

Równania semantyczne

Aexp

$$\mathcal{A}[\![n]\!] = \lambda\rho \in Env.\lambda s \in S.n$$

$$\mathcal{A}[\![v]\!] = \lambda\rho \in Env.\lambda s \in S.\textbf{if } \rho v \notin Loc \textbf{ then } \perp_N \textbf{ else if } s(\rho v) = \perp_N \textbf{ then } \perp_N \textbf{ else } s(\rho v)$$

$$\mathcal{A}[\![e_0 + e_1]\!] = \lambda\rho \in Env.\lambda s \in S.\textbf{if } \mathcal{A}[\![e_0]\!]\rho s = \perp_N \textbf{ then } \perp_N \textbf{ else if } \mathcal{A}[\![e_1]\!]\rho s = \perp_N \textbf{ then } \perp_N \textbf{ else } (\mathcal{A}[\![e_0]\!]\rho s + \mathcal{A}[\![e_1]\!]\rho s)$$

$$\mathcal{A}[\![e_0 - e_1]\!] = \lambda\rho \in Env.\lambda s \in S.\textbf{if } \mathcal{A}[\![e_0]\!]\rho s = \perp_N \textbf{ then } \perp_N \textbf{ else if } \mathcal{A}[\![e_1]\!]\rho s = \perp_N \textbf{ then } \perp_N \textbf{ else } (\mathcal{A}[\![e_0]\!]\rho s - \mathcal{A}[\![e_1]\!]\rho s)$$

$$\mathcal{A}[\![e_0 * e_1]\!] = \lambda\rho \in Env.\lambda s \in S.\textbf{if } \mathcal{A}[\![e_0]\!]\rho s = \perp_N \textbf{ then } \perp_N \textbf{ else if } \mathcal{A}[\![e_1]\!]\rho s = \perp_N \textbf{ then } \perp_N \textbf{ else } (\mathcal{A}[\![e_0]\!]\rho s * \mathcal{A}[\![e_1]\!]\rho s)$$

Bexp

$$\mathcal{B}[\![tt]\!] = \lambda\rho \in Env.\lambda s \in S.tt$$

$$\mathcal{B}[\![ff]\!] = \lambda\rho \in Env.\lambda s \in S.ff$$

$$\mathcal{B}[\![b_0 \vee b_1]\!] = \lambda\rho \in Env.\lambda s \in S.\textbf{if } \mathcal{B}[\![b_0]\!]\rho s = \perp_B \textbf{ then } \perp_B \textbf{ else if } \mathcal{B}[\![b_1]\!]\rho s = \perp_B \textbf{ then } \perp_B \textbf{ else if } \mathcal{B}[\![b_0]\!]\rho s = tt \textbf{ then } tt \textbf{ else if } \mathcal{B}[\![b_1]\!]\rho s = tt \textbf{ then } tt \textbf{ else } ff$$

$$\mathcal{B}[\![e_0 = e_1]\!] = \lambda\rho \in Env.\lambda s \in S.\textbf{if } \mathcal{A}[\![e_0]\!]\rho s = \perp_N \textbf{ then } \perp_B \textbf{ else if } \mathcal{A}[\![e_1]\!]\rho s = \perp_N \textbf{ then } \perp_B \textbf{ else if } \mathcal{A}[\![e_0]\!]\rho s = \mathcal{A}[\![e_1]\!]\rho s \textbf{ then } tt \textbf{ else } ff$$

$$\mathcal{B}[\![\neg b]\!] = \lambda\rho \in Env.\lambda s \in S.\textbf{if } \mathcal{B}[\![b_0]\!]\rho s = \perp_B \textbf{ then } \perp_B \textbf{ else if } \mathcal{B}[\![b]\!]\rho s = tt \textbf{ then } ff \textbf{ else } tt.$$

Decl

$$\mathcal{D}[\![\ \underline{\text{var}}\ v]\!] = \lambda\rho \in Env.\lambda s \in S.\\ (\rho[(new(s))_1/v], (new(s))_2)$$

$$\mathcal{D}[\![d_1; d_2]\!] = \lambda\rho \in Env.\lambda s \in S.\\ \mathcal{D}[\![d_2]\!](\mathcal{D}[\![d_1]\!]\rho s)_1(\mathcal{D}[\![d_1]\!]\rho s)_2$$

Com

$$\mathcal{C}[\![\underline{\text{skip}}]\!] = \lambda\rho \in Env.\lambda s \in S.s$$

$$\mathcal{C}[\![v := e]\!] = \lambda\rho \in Env.\lambda s \in S.\textbf{if } \rho v \notin Loc\\ \textbf{ then } \perp_S \textbf{ else if } \mathcal{A}[\![e]\!]\rho s = \perp_N \textbf{ then } \perp_S\\ \textbf{ else } s[\mathcal{A}[\![e]\!]\rho s/\rho v]$$

$$\mathcal{C}[\![c_1; c_2]\!] = \lambda\rho \in Env.\lambda s \in S.\textbf{if } \mathcal{C}[\![c_1]\!]\rho s = \perp_S\\ \textbf{ then } \perp_S \textbf{ else } \mathcal{C}[\![c_2]\!]\rho(\mathcal{C}[\![c_1]\!]\rho s)$$

$$\mathcal{C}[\![\underline{\text{if}}\ b\ \underline{\text{then}}\ c_1\ \underline{\text{else}}\ c_2]\!] = \lambda\rho \in Env.\lambda s \in S.\\ \textbf{if } \mathcal{B}[\![b]\!]\rho s = \perp_B \textbf{ then } \perp_S \textbf{ else}\\ \textbf{if } \mathcal{B}[\![b]\!]\rho s = tt \textbf{ then } \mathcal{C}[\![c_1]\!]\rho s \textbf{ else } \mathcal{C}[\![c_2]\!]\rho s$$

$$\mathcal{C}[\![\underline{\text{while}}\ b\ \underline{\text{do}}\ c]\!] = \lambda\rho \in Env.fix(f)$$

gdzie: $f = \mathcal{F}(\mathcal{B}[\![b]\!], \mathcal{C}[\![c]\!]) : \mathcal{D}'_{Com} \to \mathcal{D}'_{Com}$,

$$\mathcal{D}'_{Com} = [S \to S_\perp]$$

i $f = \lambda w \in \mathcal{D}'_{Com}.\lambda s \in S.$
if $\mathcal{B}[\![b]\!]\rho s = \perp_B$ **then** $\perp_S$ **else**
if $\mathcal{B}[\![b]\!]\rho s = tt$ **then** $w\rho(\mathcal{C}[\![c]\!]\rho s)$ **else** s

$\mathcal{C}[\![\ \underline{\text{begin}}\ d; c\ \underline{\text{end}}\ ]\!] = \lambda\rho \in Env.\lambda s \in S.$
$\mathcal{C}[\![c]\!](\mathcal{D}[\![d]\!]\rho s)_1(\mathcal{D}[\![d]\!]\rho s)_2$

Semantyka aksjomatyczna

Cel:
Skonstruować regułę, która pozwoli wyprowadzić specyfikację

$$\{\alpha\}\ \underline{\text{begin}}\ d; c\ \underline{\text{end}}\ \{\beta\}$$

w zależności od własności samej instrukcji c.

Przykład 10

$$\{(x = -1) \wedge (z < 0)\}\ \underline{\text{begin}}\ \ \underline{\text{var}}\ y; y := 5\ \underline{\text{end}}\ \{x < 0\}$$

Prawdziwość tej specyfikacji wynika z prawdziwości specyfikacji

$$\{(x = -1) \wedge (z < 0)\} y := 5 \{x < 0\}$$

Ale wnioskowania tego nie możemy przenieść na analogiczną specyfikację

$$\{(x = -1) \wedge (z < 0)\}\ \underline{\text{begin}}\ \ \underline{\text{var}}\ x; x := 5\ \underline{\text{end}}\ \{x < 0\}$$

bo specyfikacja

$$\{(x = -1) \wedge (z < 0)\} x := 5 \{x < 0\}$$

jest zawsze fałszywa. ∎

Gdzie leży istota problemu?

Odp: w kolizji nazw zmiennych.

Jak temu zaradzić?

Odp: przemianować zmienne w bloku wewnętrznym.

Wprowadzimy następujące oznaczenia:
Niech $\overline{x} = (x_1, \ldots, x_n), \overline{y} = (y_1, \ldots, y_n)$ oznaczają wektory zmiennych. Przez $c[\overline{y}/\overline{x}]$ będziemy oznaczać instrukcję c, w której wszystkie **wolne** w c wystąpienia x_i zastąpiono jednocześnie przez y_i, dla $i = 1, \ldots, n$.

7. **Reguła instrukcji bloku**

$$\frac{\{\alpha\}c[\overline{y}/\overline{x}]\{\beta\}}{\{\alpha\}\ \underline{\text{begin}}\ d; c\ \underline{\text{end}}\ \{\beta\}}$$

gdzie $\overline{x} = (x_1, \ldots, x_n)$ oznacza wektor zmiennych zadeklarowanych w d, a $\overline{y} = (y_1, \ldots, y_n)$ wektor dowolnych zmiennych takich, że y_i nie występuje w sposób wolny w α, β i c.

Przykład 11
Rozważmy następujące instrukcje:

$$c_1 \equiv \underline{\text{begin}}\ \ \underline{\text{var}}\ x; y := x\ \underline{\text{end}}$$

$$c_2 \equiv \underline{\text{begin}}\ \ \underline{\text{var}}\ u; z := u\ \underline{\text{end}}$$

W interpretacji, którą zdefiniowaliśmy za pomocą semantyki denotacyjnej, specyfikacja (rozumiana w sensie poprawności częciowej)

$$\{tt\}c_1; c_2\{y = z\}$$

jest oczywiście prawdziwa. Z drugiej strony nie da się jej udowodnić w systemie H z regułą (7). Jest więc on niezupełny względem każdej klasy interpretacji zawierającej powyższą interpretację. Nie chcemy jednak "wzmacniać" systemu, ponieważ dowodzenie takich specyfikacji byloby nieracjonalne. Niezupełność nie wynika bowiem z jego "słabości" lecz ze zbyt dokładnego określenia semantyki. ∎

Procedury

Rozszerzamy składnię języka o procedury. Musimy rozszerzyć definicję deklaracji i instrukcji.

- $d ::= \underline{\text{var}}\ v \mid d_1; d_2 \mid \underline{\text{proc}}\ P(v); c$

- $c ::= \underline{\text{skip}} \mid v := e \mid c; c' \mid \underline{\text{if}}\ \ b\ \underline{\text{then}}\ \ c\ \underline{\text{else}}\ \ c' \mid \underline{\text{while}}\ \ b\ \underline{\text{do}}\ \ c \mid \underline{\text{begin}}\ d; c\ \underline{\text{end}} \mid \underline{\text{call}}\ P(v)$

Sposoby przekazywania parametrów:

1. przez zmienną
2. przez wartość
3. przez nazwę

Na początek przyjmujemy następujące założenia:

1. brak rekursji
2. przekazywanie parametrów przez zmienną

Przykład 12

```
 program proc;
   var x:integer;
   procedure druk;
      begin
      writeln(x);
      end;
   procedure stat;
      var x:integer;
      begin
        x:=2;
        druk
      end;
begin
  x:=1;
  stat
end.
```

Pytanie: co wydrukuje procedura **druk** ?
Możliwości są dwie:

1. Pascal, Algol-60 – 1 (*semantyka statyczna*) – zmienne globalne pochodzą ze środowiska deklaracji.
2. Lisp, POP-2 – 2 (*semantyka dynamiczna*) – zmienne globalne pochodzą ze środowiska wywołania.

Semantyka statyczna

Kategorie syntaktyczne

jak poprzednio.

Dziedziny semantyczne

jak poprzednio oraz

- $Proc = [Loc \to [S \to S_\perp]]$
- $Dv = Loc + Proc + \{ \underline{\text{puste}} \}$
- $Env = [Ide \to Dv]$

Równania semantyczne

Aexp i **Bexp** jak poprzednio.

Decl

$$\mathcal{D}[\![\underline{\text{var}}\ v]\!] = \lambda\rho \in Env.\lambda s \in S.\\ (\rho[(new(s))_1/v], (new(s))_2)$$

$$\mathcal{D}[\![d_1; d_2]\!] = \lambda\rho \in Env.\lambda s \in S.\\ \mathcal{D}[\![d_2]\!](\mathcal{D}[\![d_1]\!]\rho s)_1(\mathcal{D}[\![d_1]\!]\rho s)_2$$

$$\mathcal{D}[\![\underline{\text{proc}}\ P(v); c]\!] = \lambda\rho \in Env.\lambda s \in S.(\rho[f/P], s)$$

gdzie $f \in Proc$ jest zdefiniowana następująco:

$$f = \lambda l \in Loc.\lambda s' \in S.\mathcal{C}[\![c]\!](\rho[l/v])s'$$

Com jak poprzednio oraz

$$\mathcal{C}[\![\underline{\text{call}}\ P(v)]\!] = \lambda\rho \in Env.\lambda s \in S.\textbf{if } \rho P \notin Proc$$
$$\textbf{then } \perp_S \textbf{ else if } \rho v \notin Loc \textbf{ then } \perp_S$$
$$\textbf{else } (\rho P)(\rho v)s.$$

Semantyka dynamiczna

Dziedziny semantyczne

jak poprzednio z wyjątkiem dziedziny Proc. Nowa definicja

- $Proc = [Env \to [Loc \to [S \to S_\perp]]]$

Decl

$$\mathcal{D}[\![\underline{\text{proc}}\ P(v); c]\!] = \lambda\rho \in Env.\lambda s \in S.(\rho[f/P], s)$$

gdzie $f \in Proc$ jest zdefiniowana następująco:

$$f = \lambda\rho' \in Env.\lambda l \in Loc.\lambda s' \in S.\mathcal{C}[\![c]\!](\rho'[l/v])s'$$

Com

$$\mathcal{C}[\![\underline{\text{call}}\ P(v)]\!] = \lambda\rho \in Env.\lambda s \in S.\textbf{if } \rho P \notin Proc$$
$$\textbf{then } \perp_S \textbf{ else if } \rho v \notin Loc \textbf{ then } \perp_S$$
$$\textbf{else } (\rho P)\rho(\rho v)s.$$

Semantyka statyczna – rekurencja

Dziedziny semantyczne

wracamy do poprzedniej definicji dziedziny Proc.

- $Proc = [Loc \to [S \to S_\perp]]$

Decl

$$\mathcal{D}[\![\underline{\text{proc}}\ P(v); c]\!] = \lambda\rho \in Env.\lambda s \in S.(\rho[f/P], s)$$

operator $F : Proc \to Proc$
jest zdefiniowany następująco:

$$F = \lambda g \in Proc.\lambda l \in Loc.\lambda s' \in S.\mathcal{C}[\![c]\!](\rho[g/P][l/v])s'$$

$$f = fix(F)$$

Com

wywołanie procedury jak w przypadku semantyki statycznej bez rekurencji.

Przekazywanie parametrów przez wartość

Proces przekazywania parametru przez wartość składa się z dwóch etapów:

1. Obliczenie wartości parametru aktualnego.
2. Przydział lokacji do zapamiętania tej wartości.

Proces wykonania procedury składa się także z dwóch etapów:

1. Wykonanie instrukcji wywołania procedury.
2. Wykonanie treści procedury.

W związku z tym przekazywanie parametru przez wartość możemy zorganizować na kilka różnych sposobów.

1. **Obliczenie** wartości parametru aktualnego i **przydział lokacji** do zapamiętania tej wartości wykonuje **instrukcja wywołania** procedury. Funkcja reprezentująca procedurę "pobiera" wartość parametru i lokację.

   Nowa definicja $Proc$:

   - $Proc = [Sv \to [Loc \to [S \to S_\perp]]]$

Decl

$$\mathcal{D}[\![\underline{\text{proc}}\ P(v); c]\!] = \lambda\rho \in Env.\lambda s \in S.(\rho[f/P], s)$$

gdzie $f \in Proc$ jest zdefiniowana następująco:

$f = \lambda w \in Sv.\lambda l \in Loc.\lambda s' \in S.$
$\mathcal{C}[\![c]\!](\rho[l/v])(s'[w/l])$

Com

$$\mathcal{C}[\![\underline{\text{call}}\ P(e)]\!] = \lambda\rho \in Env.\lambda s \in S.\ \textbf{if}\ \rho P \notin Proc\ \textbf{then}\ \perp_S\ \textbf{else if}\ \mathcal{A}[\![e]\!]\rho s \notin Sv\ \textbf{then}\ \perp_S\ \textbf{else}\ (\rho P)(\mathcal{A}[\![e]\!]\rho s)(new(s))_1(new(s))_2.$$

2. **Obliczenie** wartości parametru aktualnego wykonuje **instrukcja wywołania** procedury, a **przydział lokacji** do jej zapamiętania wykonuje **funkcja reprezentująca procedurę**.
   - $Proc = [Sv \to [S \to S_\perp]]$

Decl

$$\mathcal{D}[\![\underline{\text{proc}}\ P(v); c]\!] = \lambda\rho \in Env.\lambda s \in S.(\rho[f/P], s)$$

gdzie $f \in Proc$ jest zdefiniowana następująco:

$f = \lambda w \in Sv.\lambda s' \in S.$ **let** $(l'', s'') = new(s')$ **in** $\mathcal{C}[\![c]\!](\rho[l''/v])(s''[w/l''])$

Com

$\mathcal{C}[\![\underline{\text{call}}\ P(e)]\!] = \lambda\rho \in Env.\lambda s \in S.$**if** $\rho P \notin Proc$ **then** $\perp_S$ **else if** $\mathcal{A}[\![e]\!]\rho s \notin Sv$ **then** $\perp_S$ **else** $(\rho P)(\mathcal{A}[\![e]\!]\rho s)s.$

3. **Instrukcja wywołania** procedury przekazuje do procedury **nieobliczony** parametr aktualny. **Funkcja reprezentująca procedurę oblicza** wartość parametru i **przydziela lokację**.

   Aby sformalizować pojęcie nieobliczonego parametru aktualnego wprowadzamy pojęcie domknięcia wyrażeń:

   - $Cl = [S \to Sv]$
   - $Proc = [Cl \to [S \to S_\perp]]$

Decl

$$\mathcal{D}[\![\underline{\text{proc}}\ P(v); c]\!] = \lambda\rho \in Env.\lambda s \in S.(\rho[f/P], s)$$

gdzie $f \in Proc$ jest zdefiniowana następująco:

$f = \lambda\xi \in Cl.\lambda s' \in S.$**if** $\xi s' \notin Sv$ **then** $\perp_S$ **else** **let** $(l'', s'') = new(s')$ **in** $\mathcal{C}[\![c]\!](\rho[l''/v])(s''[\xi s'/l''])$

Com

$$\mathcal{C}[\![\ \underline{\text{call}}\ P(e)]\!] = \lambda\rho \in Env.\lambda s \in S.\textbf{if}\ \rho P \notin Proc\ \textbf{then}\ \perp_S\ \textbf{else}\ (\rho P)(\mathcal{A}[\![e]\!]\rho)s.$$

Przekazywanie parametrów przez nazwę

Przykład 13

W poniższym programie parametr przekazywany jest przez **nazwę**.

```
begin
   var x; var z;
   proc p(y); z:=x+y; x:=y+1
   x:=0; z:=0;
   begin
      var x;                    z:=x+z+x;
      x:=1;                     {z=1}
      call p(z+x);              x:=z+x+1
   end;                         {x=3}
   z:=z+x
end
{z=4}
```

Gdyby parametr był przekazywany przez zmienną, wywołanie **call p(z+x)** byłoby błędne.

Gdyby parametr był przekazywany przez wartość, wywołanie **call p(z+x)** przebiegałoby następująco:

```
begin
   var x; var z;
   proc p(y); z:=x+y; x:=y+1
   x:=0; z:=0;
   begin
      var x;
      x:=1;                      y:= z+x; {y=1}
      call p(z+x);               z:= x+y; {z=1}
   end;                          x:= y+1; {x=2}
   z:=z+x
end
{z=3}
```

Formalizacja

Wprowadzamy lewe i prawe wartości wyrażeń:

- $Lv = N + B + Loc$ – lewe wartości
- $Rv = N + B$ – prawe wartości

oraz domknięcia:

- $Cl = [S \to Lv + \{\bot_E\}]$

Definiujemy pozostałe dziedziny semantyczne:

- $Sv = N$
- $S = [Loc \to Sv + \{\ \underline{\text{wolna}}\ \}]$
- $Proc = [Cl \to [S \to S_\bot]]$
- $Dv = Loc + Proc + Cl + \{\ \underline{\text{puste}}\ \}$
- $Env = [Ide \to Dv]$

Dla skrócenia zapisów przyjmiemy

- $Exp = Aexp + Bexp$

Funkcje semantyczne

$$\mathcal{L} : Exp \to [Env \to [S \to Lv + \{\bot_E\}]]$$
$$\mathcal{R} : Exp \to [Env \to [S \to Rv + \{\bot_E\}]]$$
$$\mathcal{D} : Decl \to [Env \to [S \to Env \times S]]$$
$$\mathcal{C} : Com \to [Env \to [S \to S_\bot]]$$

Równania semantyczne

Exp

$$\mathcal{L}[\![n]\!] = \lambda\rho \in Env.\lambda s \in S.n$$

$$\mathcal{R}[\![n]\!] = \lambda\rho \in Env.\lambda s \in S.n$$

$$\mathcal{L}[\![v]\!] = \lambda\rho \in Env.\lambda s \in S.\textbf{if } \rho v \in Loc \textbf{ then } \rho v \textbf{ else if } \rho v \in Cl \textbf{ then } (\rho v)s \textbf{ else } \bot_E$$

$$\mathcal{R}[\![v]\!] = \lambda\rho \in Env.\lambda s \in S.\ \textbf{if } \rho v \in Loc \textbf{ then if } s(\rho v) \notin Rv \textbf{ then } \bot_E \textbf{ else } s(\rho v) \textbf{ else if } \rho v \in Cl \textbf{ then } \textbf{if } (\rho v)s \in Rv \textbf{ then } (\rho v)s \textbf{ else } \textbf{if } (\rho v)s \in Loc \textbf{ then}$$

$$\textbf{if } s((\rho v)s) \in Rv \textbf{ then } s((\rho v)s) \textbf{ else } \perp_E$$
$$\textbf{else } \perp_E \textbf{ else } \perp_E$$

$$\mathcal{L}[\![a_1 + a_2]\!] = \lambda\rho \in Env.\lambda s \in S.$$
$$\textbf{if } \mathcal{R}[\![a_1]\!]\rho s \notin N \textbf{ then } \perp_E \textbf{ else}$$
$$\textbf{if } \mathcal{R}[\![a_2]\!]\rho s \notin N \textbf{ then } \perp_E \textbf{ else}$$
$$\mathcal{R}[\![a_1]\!]\rho s + \mathcal{R}[\![a_2]\!]\rho s$$

Pozostałe wyrażenia definiujemy analogicznie.

Decl

$\mathcal{D}[\![\underline{\text{proc}}\ P(v); c]\!] = \lambda\rho \in Env.\lambda s \in S.((\rho[f/P], s)$
gdzie $f \in Proc$ jest zdefiniowana następująco:

$$f = \lambda\xi \in Cl.\lambda s' \in S.\mathcal{C}[\![c]\!](\rho[\xi/v])s'$$

Com

$$\mathcal{C}[\![v := e]\!] = \lambda\rho \in Env.\lambda s \in S.\textbf{if } \mathcal{L}[\![v]\!]\rho s \notin Loc$$
$$\textbf{then } \perp_S \textbf{ else if } \mathcal{R}[\![e]\!]\rho s \notin Sv \textbf{ then } \perp_S$$
$$\textbf{else } s[\mathcal{R}[\![e]\!]\rho s/\mathcal{L}[\![v]\!]\rho s]$$

$$\mathcal{C}[\![\underline{\text{call}}\ P(e)]\!] = \lambda\rho \in Env.\lambda s \in S.\textbf{if } \rho P \notin Proc$$
$$\textbf{then } \perp_S \textbf{ else } (\rho P)(\mathcal{L}[\![e]\!]\rho)s.$$

Niech $\underline{\text{proc}}\ P(\overline{v}); c$ będzie deklaracją procedury, gdzie $\overline{v}$ jest listą parametrów formalnych.

8. Reguła instrukcji wywołania procedury

$$\frac{\{\alpha\}\ \ c\ \ \{\beta\}}{\forall \overline{v}(roz(\overline{v}) \Rightarrow \{\alpha\}P(\overline{v})\{\beta\})}$$

gdzie β nie może zawierać wolnego wystąpienia żadnego z parametrów z $\overline{v}$ przekazywanych przez wartość, a $roz(\overline{v})$ oznacza, że spełnione są następujące warunki:

(a) Wszystkie parametry aktualne z $\overline{v}$ przekazywane przez zmienną są różne i żaden z nich nie występuje wśród parametrów aktualnych przekazywanych przez wartość.

(b) Żadna z tych zmiennych nielokalnych w c, którym tam przypisuje się wartość, nie występuje w parametrach aktualnych z $\overline{v}$ przekazywanych

przez wartość.

Dla konkretnego wywołania $P(\overline{a})$, w którym lista $\overline{a}$ spełnia powyższe warunki, regułę (8) stosujemy wraz z regułą

9.

$$\frac{roz(\overline{a}), \forall \overline{v}(roz(\overline{v}) \Rightarrow \{\alpha\} P(\overline{v}) \{\beta\})}{\{\alpha[\overline{a}/\overline{v}]\}\ P(\overline{a})\ \{\beta[\overline{a}/\overline{v}]\}}$$

Przykład 14
Rozważmy procedurę:

$\underline{\text{proc}}$ $\quad swap(\ \underline{\text{var}}\ x, y);$
$\underline{\text{begin}}$ $\quad \underline{\text{var}}\ z;$
$z := x;\ \ x := y;\ \ y := z$
$\underline{\text{end}}$

Ciało procedury c spełnia specyfikację

$$\{(x = X) \wedge (y = Y)\}\ c\ \{(x = Y) \wedge (y = X)\} \quad (14)$$

X i Y są tu tzw. zmiennymi specyfikacyjnymi, oznaczającymi wartości początkowe zmiennych x i y. Muszą to być nowe identyfikatory, które nie występują w programie.

Rozważmy wywołanie $swap(a, b)$. Stosując reguły (8) i (9) otrzymujemy

$$\begin{array}{l}\{((x = X) \wedge (y = Y))[a/x, b/y]\} \\ \ swap(a, b) \\ \{((x = Y) \wedge (y = X))[a/x, b/y]\}\end{array}$$

co daje

$$\{(a = X) \wedge (b = Y)\}\ swap(a, b)\ \{(a = Y) \wedge (b = X)\}$$

Ale jeśli wywołanie $swap(a, b)$ wystąpi w sytuacji, gdy $a = 1$, $b = 2$, to chcielibyśmy móc wyprowadzić

$$\{(a = 1) \wedge (b = 2)\}\ swap(a, b)\ \{(a = 2) \wedge (b = 1)\}$$

Ponieważ X i Y reprezentują dowolne wartości początkowe, specyfikacja (14) jest równoważna

$$\forall X, Y.\{(x = X) \wedge (y = Y)\}\ c\ \{(x = Y) \wedge (y = X)\} \quad (15)$$

Możemy teraz wziąć instancję (15) otrzymaną przez zastąpienie X przez 1 i Y przez 2.

Przykład 15
Pokażemy teraz, że warunek rozłączności parametrów aktualnych jest istotny. Rozważmy procedurę:

$$\underline{\text{proc}} \quad dwa(\ \underline{\text{var}}\ x, y);$$
$$x := 1; \quad y := 2$$

i jej wywołanie $dwa(a, a)$. Instrukcja c spełnia specyfikację

$$\{tt\}\ \ c\ \ \{(x = 1) \wedge (y = 2)\}$$

ale nie zachodzi

$$\{tt\}\ \ dwa(a, a)\ \ \{(a = 1) \wedge (a = 2)\}$$

co jest równoważne

$$\{tt\}\ \ dwa(a, a)\ \ \{ff\}$$

Wejście - wyjście: semantyka denotacyjna

Rozszerzamy składnię języka o wejście-wyjście:

Składnia

- $e ::= m \mid v \mid (e + e') \mid (e - e') \mid (e * e') \mid \underline{\text{read}}$
- $b ::= t \mid b \vee b' \mid e = e' \mid \neg b$
- $d ::= \underline{\text{var}}\ v \mid d_1; d_2$
- $c ::= \underline{\text{skip}} \mid v := e \mid c; c' \mid \underline{\text{if}}\ \ b\ \underline{\text{then}}\ \ c\ \underline{\text{else}}\ \ c' \mid \underline{\text{while}}\ \ b\ \underline{\text{do}}\ \ c \mid \underline{\text{begin}}\ d; c\ \underline{\text{end}} \mid \underline{\text{write}}(e)$

Musimy zmodyfikować dziedziny semantyczne. Dotychczas mielismy
$S = [Loc \to Sv + \{\ \underline{\text{wolna}}\ \}]$ – stany (=pamięć)
Teraz rozszerzamy stany o wejście i wyjście. Mamy:

- $Val = \{N + B + \{\perp\}\}$
- $In = Val^*$
- $Out = Val^*$

- pusty ciąg wartości będziemy oznaczać ϵ
- $S = M \times In \times Out$
- $M = [Loc \to Sv + \{ \underline{\text{wolna}} \}]$ – stany (pamięć)

Będziemy używać pomocniczych funkcji:

- $hd : Val^* \to Val$
- $tl : Val^* \to Val*$

Funkcje semantyczne

$\mathcal{A} : Aexp \to [Env \to [S \to N_\perp \times In]]$

$\mathcal{B} : Bexp \to [Env \to [S \to B_\perp \times In]]$

$\mathcal{D} : Decl \to [Env \to [M \to Env \times M]]$

$\mathcal{C} : Com \to [Env \to [S \to S_\perp]]$

Równania semantyczne

Aexp

$$\mathcal{A}[\![n]\!] = \lambda\rho \in Env.\lambda s \in S.(n, (s)_2)$$

$$\mathcal{A}[\![v]\!] = \lambda\rho \in Env.\lambda(m,i,o) \in S.\textbf{if } \rho v \notin Loc \textbf{ then } \bot \textbf{ else if } m(\rho v) = \bot_N \textbf{ then } \bot_N \textbf{ else } (m(\rho v), i)$$

$$\mathcal{A}[\![\underline{\text{read}}]\!] = \lambda\rho \in Env.\lambda s \in S.\textbf{if } i = \epsilon \textbf{ then } \bot \textbf{ else } (hd(i), tl(i))$$

$$\mathcal{A}[\![e_0 + e_1]\!] = \lambda\rho \in Env.\lambda(m,i,o) \in S.$$
$$\textbf{let } (w', i') = \mathcal{A}[\![e_0]\!]\rho(m,i,o) \textbf{ in}$$
$$\textbf{if } w' = \bot_N \textbf{ then } \bot \textbf{ else}$$
$$\textbf{let } (w'', i'') = \mathcal{A}[\![e_1]\!]\rho(m,i',o) \textbf{ in}$$
$$\textbf{if } w'' = \bot_N \textbf{ then } \bot \textbf{ else } (w' + w'', i')$$

Pozostałe wyrażenia analogicznie.

Decl

$$\mathcal{D}[\![\underline{\text{var}}\ v]\!] = \lambda\rho \in Env.\lambda s \in S.$$
$$(\rho[(new(s))_1/v], (new(s))_2)$$

$$\mathcal{D}[\![d_1; d_2]\!] = \lambda\rho \in Env.\lambda s \in S.$$
$$\mathcal{D}[\![d_2]\!](\mathcal{D}[\![d_1]\!]\rho s)_1(\mathcal{D}[\![d_1]\!]\rho s)_2$$

Com

$$\mathcal{C}[\![\underline{\text{skip}}]\!] = \lambda\rho \in Env.\lambda s \in S.s$$

$$\mathcal{C}[\![v := e]\!] = \lambda\rho \in Env.\lambda(m, i, o) \in S$$
$$.\textbf{if } \rho v \notin Loc \textbf{ then } \perp_S \textbf{ else}$$
$$\textbf{if } (\mathcal{A}[\![e]\!]\rho(m, i, o))_1 = \perp_N \textbf{ then } \perp_S \textbf{ else}$$
$$(m[(\mathcal{A}[\![e]\!]\rho(m, i, o))_1/\rho v], (\mathcal{A}[\![e]\!]\rho(m, i, o))_2, o)$$

$$\mathcal{C}[\![\underline{\text{write}}(e)\ ]\!] = \lambda\rho \in Env.\lambda(m, i, o) \in S.$$
$$\textbf{let } (w, i') = \mathcal{A}[\![e]\!]\rho(m, i, o) \textbf{ in}$$
$$\textbf{if } w = \perp_N \textbf{ then } \perp_S \textbf{ else } (m, i', w.o)$$

Pozostałe instrukcje analogicznie.

Niedeterminizm

Rozszerzymy język programowania o następujące instrukcje:

$$c' ::= c \mid \underline{\text{abort}} \mid \underline{\text{if}}\ b_1 \to c_1 [\!] \ldots [\!] b_n \to c_n\ \underline{\text{fi}} \mid \underline{\text{do}}\ b_1 \to c_1 [\!] \ldots [\!] b_n \to c_n\ \underline{\text{od}}$$

Semantyka operacyjna nowych instrukcji

Przyjmiemy następujące oznaczenia:

$$IF \equiv \underline{\text{if}}\ b_1 \to c_1 [\!] \ldots [\!] b_n \to c_n\ \underline{\text{fi}}$$

$$DO \equiv \underline{\text{do}}\ b_1 \to c_1 [\!] \ldots [\!] b_n \to c_n\ \underline{\text{od}}$$

<u>abort</u>

1. $\langle \underline{\text{abort}}, \sigma \rangle \rightsquigarrow \langle \epsilon, \textbf{fail} \rangle$

IF

1. $\dfrac{\langle b_i, \sigma\rangle \rightsquigarrow \langle tt, \sigma\rangle}{\langle IF, \sigma\rangle \rightsquigarrow \langle c_i, \sigma\rangle}$, gdzie $i \in \{1, ..., n\}$.

2. $\dfrac{\langle \wedge_{i=1}^{n} \neg b_i, \sigma\rangle \rightsquigarrow \langle tt, \sigma\rangle}{\langle IF, \sigma\rangle \rightsquigarrow \langle \epsilon, \textbf{fail} \ \rangle}$

DO

1. $\dfrac{\langle b_i, \sigma\rangle \rightsquigarrow \langle tt, \sigma\rangle}{\langle DO, \sigma\rangle \rightsquigarrow \langle c_i; DO, \sigma\rangle}$,

   gdzie $i \in \{1, ..., n\}$.

2. $\dfrac{\langle \wedge_{i=1}^{n} \neg b_i, \sigma\rangle \rightsquigarrow \langle tt, \sigma\rangle}{\langle DO, \sigma\rangle \rightsquigarrow \langle \epsilon, \sigma\rangle}$

Uwaga 4
Zauważmy, że w przypadku, gdy więcej niż jedno wyrażenie b_i (zwane *dozorem*) jest prawdziwe w instrukcji IF lub DO, wybór instrukcji c_i jest niedeterministyczny.

■

Uwaga 5
Zauważmy, że instrukcje IF i DO są uogólnieniem instrukcji $\underline{\text{if}}$ i $\underline{\text{while}}$.

$$\underline{\text{if}}\ b\ \underline{\text{then}}\ c_1\ \underline{\text{else}}\ c_2 \equiv \underline{\text{if}}\ b \to c_1 [\!] \neg b \to c_2\ \underline{\text{fi}}$$

$$\underline{\text{while}}\ b\ \underline{\text{do}}\ c \equiv\ \underline{\text{do}}\ b \to c\ \underline{\text{od}}$$

■

Semantyka Dijkstry

Transformatory formuł:

Dla dowolnej instrukcji c i formuły α:

- $wp(c, \alpha)$ jest formułą, której modelami są dokładnie wszystkie stany takie, że wykonanie c rozpoczęte w którymkolwiek z nich zakończy się w stanie spełniającym warunek α.
- $wlp(c, \alpha)$ jest formułą, której modelami są dokładnie wszystkie stany takie, że jeśli wykonanie c rozpocznie się w którymkolwiek z nich i zakończy, to stan końcowy spełnia warunek α.
- $sp(c, \alpha)$ jest formułą, której modelami są dokładnie te stany, które są osiągalne w wyniku wykonania c, jeśli c się skończy, gdy c rozpocznie się w stanie spełniającym warunek α.

Formalnie:

Zachowujemy dotychczasowe założenia co do języka logiki I rzędu.

Zdefiniujemy pomocnicze funkcje semantyczne:

$$\mathcal{E}[\![c]\!](\sigma) = \{\tau \mid \mathbf{exec}(\mathbf{c}, \sigma) = \tau\}.$$
$$\mathcal{E}_{tot}[\![c]\!](\sigma) = \mathcal{E}[\![c]\!](\sigma) \cup \{\bot \mid \ c \text{ pętli się w } \sigma\}$$

Zauważmy, że jeśli c pętli się w stanie σ, to

$$\mathcal{E}[\![c]\!](\sigma) = \emptyset$$

a

$$\mathcal{E}_{tot}[\![c]\!](\sigma) = \{\bot\}$$

Rozszerzymy powyższe definicje na $\Sigma_\bot$ następująco:

$$\mathcal{E}[\![c]\!](\bot) = \emptyset$$

$$\mathcal{E}_{tot}[\![c]\!](\bot) = \{\bot\}$$

Niech $X \subseteq \Sigma$ będzie dowolnym zbiorem stanów. Wtedy

$$\mathcal{EX}[\![c]\!](X) = \bigcup_{\sigma \in X} \mathcal{E}[\![c]\!](\sigma).$$

Za pomocą tak zdefiniowanej semantyki wyrazimy transformatory formuł.

Niech $\Phi \subseteq \Sigma$. Zdefiniujmy operatory wlp^*, wp^* i sp^* dla stanów:

- $wlp^*(c, \Phi) = \{\sigma \in \Sigma \mid \mathcal{E}[\![c]\!](\sigma) \subseteq \Phi\}$;
- $wp^*(c, \Phi) = \{\sigma \in \Sigma \mid \mathcal{E}_{tot}[\![c]\!](\sigma) \subseteq \Phi\}$;
- $sp^*(c, \Phi) = \cup_{\sigma \in \Phi} \mathcal{E}[\![c]\!](\sigma)$.

Na podstawie powyższej definicji zależność między operatorami wlp^* i sp^* wyraża się wzorami

$$\Phi \subseteq wlp^*(c, sp^*(c, \Phi)) \tag{16}$$

$$sp^*(c, wlp^*(c, \Phi)) \subseteq \Phi \tag{17}$$

Wprowadzimy następujące oznaczenie.
Niech α będzie formułą języka logiki I rzędu $\mathcal{L}(V)$, a $\sigma \in \Sigma$ stanem nad zbiorem zmiennych V.

$$\mid \alpha \mid = \{\sigma \mid \sigma \models \alpha\} \tag{18}$$

Na podstawie twierdzenia o wyrażalności mamy

Twierdzenie 20
Niech c będzie programem, a β formułą.

1. Istnieje formuła α, taka że $\mid \alpha \mid = wlp^*(c, \mid \beta \mid)$.
2. Istnieje formuła α, taka że $\mid \alpha \mid = wp^*(c, \mid \beta \mid)$.
3. Istnieje formuła α, taka że $\mid \alpha \mid = sp^*(c, \mid \beta \mid)$.

■

Teraz możemy już zdefiniować operatory wlp, wp i sp dla formuł.

Definicja 21
Niech α, β będą formułami logiki I rzędu, a c programem.

1. $wlp(c, \alpha) = \beta$ wtt $wlp^*(c, \mid \alpha \mid) = \mid \beta \mid$.
2. $wp(c, \alpha) = \beta$ wtt $wp^*(c, \mid \alpha \mid) = \mid \beta \mid$.
3. $sp(c, \alpha) = \beta$ wtt $sp^*(c, \mid \alpha \mid) = \mid \beta \mid$.

■

Semantyka instrukcji

1. **Instrukcja <u>abort</u>.**

   $wlp(\underline{\text{abort}}, \alpha) \equiv tt$
   $wp(\underline{\text{abort}}, \alpha) \equiv ff$
   $sp(\underline{\text{abort}}, \alpha) \equiv ff$

2. **Instrukcja <u>skip</u>.**

   $wlp(\underline{\text{skip}}, \alpha) \equiv wp(\underline{\text{skip}}, \alpha) \equiv sp(\underline{\text{skip}}, \alpha) \equiv \alpha.$

3. **Instrukcja (:=).**

   $wlp(x := t, \alpha) \equiv wp(x := t, \alpha) \equiv \alpha[t/x].$

   $sp(x := t, \alpha) \equiv$
   $\exists y.((x = t[y/x]) \wedge \alpha[y/x]),$ (19)
   gdzie y jest nową zmienną.

   Jeśli zmienna x nie występuje w wyrażeniu t, równanie (19) można uprościć do postaci

   $sp(x := t, \alpha) \equiv (x = t) \wedge \exists x.\alpha$

4. **Instrukcja (;).**

$$wlp(c_1; c_2, \alpha) \equiv wlp(c_1, wlp(c_2, \alpha));$$

$$wp(c_1; c_2, \alpha) \equiv wp(c_1, wp(c_2, \alpha));$$

$$sp(c_1; c_2, \alpha) \equiv sp(c_2, sp(c_1, \alpha)).$$

5. **Instrukcja IF**

$$wlp(\mathrm{IF}, \alpha) \;=\; \wedge_{i=1}^{n}(b_i \Rightarrow wlp(c_i, \alpha))$$

$$wp(\mathrm{IF}, \alpha) \;=\; \vee_{i=1}^{n} b_i \wedge \wedge_{i=1}^{n}(b_i \Rightarrow wp(c_i, \alpha))$$

$$sp(\mathrm{IF}, \alpha) \;=\; \vee_{i=1}^{n}(sp(c_i, b_i \wedge \alpha)).$$

6. **Instrukcja DO**

$$wlp(DO, \alpha) \wedge b_i \Rightarrow wlp(c_i, wlp(DO, \alpha))$$

gdzie $i \in \{1, ..., n\}$.

$$wlp(DO, \alpha) \wedge \wedge_{i=1}^{n} \neg b_i \Rightarrow \alpha$$

$$wp(DO, \alpha) \wedge b_i \Rightarrow wp(c_i, wlp(DO, \alpha))$$

gdzie $i \in \{1, ..., n\}$.

$$wp(DO, \alpha) \wedge \wedge_{i=1}^{n} \neg b_i \Rightarrow \alpha$$

$$sp(DO, \alpha) \equiv (\wedge_{i=1}^{n} \neg b_i) \wedge (\vee_{i=0}^{\infty} sp(IF^i, \alpha)$$

gdzie $IF \equiv \underline{\text{if}}\ b_1 \rightarrow c_1 [\!] \ldots [\!] b_n \rightarrow c_n\ \underline{\text{fi}}$

Dla $\underline{\text{if}}\ b\ \underline{\text{then}}\ c_1\ \underline{\text{else}}\ c_2$ i $\underline{\text{while}}\ b\ \underline{\text{do}}\ c$ otrzymujemy:

7. **Instrukcja $\underline{\textbf{if}}\ b\ \underline{\textbf{then}}\ c_1\ \underline{\textbf{else}}\ c_2$.**

$$\begin{aligned} wlp(\underline{\text{if}}\ ,\alpha) &\equiv (b \Rightarrow wlp(c_1,\alpha)) \wedge (\neg b \Rightarrow wlp(c_2,\alpha)) \\ &\equiv b \wedge wlp(c_1,\alpha) \vee \neg b \wedge wlp(c_2,\alpha) \end{aligned}$$

$$\begin{aligned} wp(\underline{\text{if}}\ ,\alpha) &\equiv (b \Rightarrow wp(c_1,\alpha)) \wedge (\neg b \Rightarrow wp(c_2,\alpha)) \\ &\equiv b \wedge wp(c_1,\alpha) \vee \neg b \wedge wp(c_2,\alpha) \end{aligned}$$

$$sp(\underline{\text{if}}\ ,\alpha) \equiv sp(c_1, b \wedge \alpha) \vee sp(c_2, \neg b \wedge \alpha).$$

8. **Instrukcja $\underline{\textbf{while}}$** .

Niech $w \equiv \underline{\text{while}}\ b\ \underline{\text{do}}\ c$.

$$wlp(w,\alpha) \wedge b \Rightarrow wlp(c, wlp(w,\alpha))$$
$$wlp(w,\alpha) \wedge \neg b \Rightarrow \alpha$$

$$wp(w,\alpha) \wedge b \Rightarrow wp(c, wlp(w,\alpha))$$
$$wp(w,\alpha) \wedge \neg b \Rightarrow \alpha$$

$$sp(w,\alpha) \equiv \neg b \wedge (\vee_{i=0}^{\infty} sp(IF^i,\alpha)$$

gdzie $IF \equiv \underline{\text{if}}\ b\ \underline{\text{then}}\ c\ \underline{\text{else}}\ \underline{\text{skip}}$

Reguły Hoare'a dla poprawności częściowej

10. Reguła dla IF

$$\frac{\{\alpha \wedge b_i\}\ c_i\ \{\beta\}, i \in \{1, \ldots, n\}}{\{\alpha\}\underline{\text{if}}\ b_1 \to c_1 [\!] \ldots [\!] b_n \to c_n\ \underline{\text{fi}}\ \{\beta\}}$$

11. Reguła dla DO

$$\frac{\{\alpha \wedge b_i\}\ c_i\ \{\alpha\}, i \in \{1, \ldots, n\}}{\{\alpha\}\ \underline{\text{do}}\ b_1 \to c_1 [\!] \ldots [\!] b_n \to c_n\ \underline{\text{od}}\ \{\alpha \wedge \wedge_{i=1}^{n} \neg b\}}$$

Reguły Hoare'a dla poprawności całkowitej

12. Reguła dla IF

$$\frac{p \Rightarrow \vee_{i=1}^{n} b_i \quad \{\alpha \wedge b_i\}\ c_i\ \{\beta\}, i \in \{1, \ldots, n\}}{\{\alpha\} \underline{\text{if}}\ b_1 \rightarrow c_1 [\!] \ldots [\!] b_n \rightarrow c_n\ \underline{\text{fi}}\ \{\beta\}}$$

13. Reguła dla DO

$$\frac{\begin{array}{c}\{\alpha \wedge b_i\}\ c_i\ \{\alpha\}, i \in \{1, \ldots, n\} \\ \{\alpha \wedge b_i \wedge t = z\}\ c_i\ \{t < z\}, i \in \{1, \ldots, n\} \\ \alpha \Rightarrow t \geq 0\end{array}}{\{\alpha\}\ \underline{\text{do}}\ b_1 \rightarrow c_1 [\!] \ldots [\!] b_n \rightarrow c_n\ \underline{\text{od}}\ \{\alpha \wedge \wedge_{i=1}^{n} \neg b\}}$$

gdzie t jest wyrażeniem całkowitoliczbowym, a z zmienną nie występującą w α, b_i, t ani c_i, dla $i \in \{1, \ldots, n\}$.