SEMANTYKA I WERYFIKACJA - ĆW. 4

1. **Zadanie (tylko w drugiej grupie):** Zrobić semantykę liczb binarnych z dodawaniem:

$$e ::= \$ \mid e0 \mid e1 \mid e_1 + e_2$$

(ten dolar to znak początku liczby, tzn. \$1001 reprezentuje liczbę binarną 1001).

Uwaga: to jest dosyć dziwna składnia, bardziej naturalne wydaje się

$$e ::= n \mid e_1 + e_2$$
$$n ::= \$ \mid e0 \mid e1$$

Rozwiązanie: Konfiguracje to wyrażenia. Dodawanie interpretujemy pisemnie:

$$\frac{}{e_1 0 + e_2 0 \to (e_1 + e_2)0} \qquad \frac{}{e_1 0 + e_2 1 \to (e_1 + e_2)1} \qquad \frac{}{e_1 1 + e_2 0 \to (e_1 + e_2)1}$$

kluczowa reguła dla przeniesienia:

$$\frac{}{e_1 1 + e_2 1 \to ((e_1 + e_2) + \$1)1}$$

kończenie obliczeń:

$$\frac{}{e + \$ = e} \qquad \frac{}{\$ + e = e}$$

(uwaga, nie można skorzystać z przemienności: $e_1 + e_2 \to e_2 + e_1$, bo to wprowadziłoby możliwość pętlenia), dodatkowo buchalteria:

$$\frac{e_1 \to e_1'}{e_1 + e_2 \to e_1' + e_2} \qquad \frac{e_2 \to e_2'}{e_1 + e_2 \to e_1 + e_2'}$$

$$\frac{e \to e'}{e0 \to e'0} \qquad \frac{e \to e'}{e1 \to e'1}$$

Pytanie: czy ta kluczowa reguła dla przeniesienia nie sprawia, że obliczenie może się nie zakończyć? Zauważmy, że wyrażenie się skomplikowało! *Odpowiedź:* Nie, dowód terminacji może iść przez indukcję po sumie dodawanych liczb.

Pytanie: A jak byśmy sobie poradzili w tej uproszczonej, naturalnej składni? *Odpowiedź:* Nijak. Konfiguracje muszą być zapisane w bogatszej składni. To częste zjawisko!

2. Semantyka naturalna, czyli semantyka operacyjna dużych kroków:

   - konfiguracje jak poprzednio, w tym konfiguracje końcowe
   - relacja przejścia jest wyłącznie w konfiguracja końcowe.

Jedna z reguł języka TINY:

$$\frac{[\![b]\!]s = tt \qquad S, s \to s' \qquad \texttt{while}\ b\ \texttt{do}\ S,\ s' \to s''}{\texttt{while}\ b\ \texttt{do}\ S,\ s \to s''}$$

3. **Zadanie:** Porównać strukturę obliczeń w semantyce operacyjnej i naturalnej (ciąg drzew vs. jedno drzewo) na przykładzie konfiguracji:

   $$\texttt{while}\ x \leq 2\ \texttt{do}\ x := x + 1,\ [x \mapsto 0]$$

   Zalety semantyki naturalnej:

   - bardziej abstrakcyjna niż semantyka małych kroków,
   - naturalne pojęcie równoważności,
   - proste dowody równoważności przez indukcję po drzewach wyprowadzeń.

   Wady:

   - bardziej abstrakcyjna niż semantyka małych kroków, :-)
   - mniejsza kompozycjonalność (np. `while`), czyli mniej dowodów przez indukcję po strukturze programów,
   - ciężko napisać semantykę języków współbieżnych.

4. **Zadanie:** Napisać semantykę naturalną wyrażeń arytmetycznych i logicznych (w strategii gorliwej albo lewostronnej leniwej) z języka TINY. A jak by wyglądała semantyka wyrażeń `let`?

5. **Zadanie:** Dodajmy do języka TINY dzielenie przez zero. Niech próba dzielenia przez zero skutkuje przerwaniem programu i zwróceniem błędu. Napisać semantykę naturalną.

   Rozwiązanie: Potrzebna jest dodatkowa konfiguracja końcowa *err*. Jest problem decyzyjny: czy zwrócenie błędu zachowuje stan? Jeśli tak, to do konfiguracji dodajemy $\{err\} \times State$, jeśli nie, to tylko $\{err\}$.

   Uwaga: W zasadzie nie trzeba zmieniać istniejących wcześniej reguł! Ale dla jasności lepiej to zrobić, dodając reguły propagacji błędów.

6. Dodajmy do języka TINY pętle z wyskakiwaniem:

   $$S ::= \cdots \mid \texttt{loop}\ S \mid \texttt{break} \mid \texttt{continue}$$

   `loop` to instrukcja pętli. Instrukcja `break` wychodzi z nabliższej otaczającej pętli i kontynuuje wykonanie programu od pierwszej instrukcji za tą pętlą. Instrukcja `continue` powraca na początek instrukcji wewnętrznej najbliższej otaczającej pętli.

   Napisać semantykę operacyjną i naturalną.

Rozwiązanie, semantyka naturalna: Do konfiguracji końcowych dodajemy specjalne wartości: `bylo-exit` i `bylo-continue`. Formalnie, wrzucamy do konfiguracji dodatkowo (niekońcowe)

$$State \times \{\texttt{bylo-exit},\ \texttt{bylo-continue}\}$$

Reguły tworzenia takich konfiguracji:

$$\frac{}{\texttt{exit},\ s \to s,\ \texttt{bylo-exit}} \qquad \frac{}{\texttt{continue},\ s \to s,\ \texttt{bylo-continue}}$$

Reguły korzystania z nich przez pętle:

$$\frac{S, s \to s',\ \texttt{bylo-exit}}{\texttt{loop}\ S,\ s \to s'} \qquad \frac{S, s \to s',\ \texttt{bylo-continue} \qquad \texttt{loop}\ S, s' \to s''}{\texttt{loop}\ S,\ s \to s''}$$

Do tego normalna reguła dla `loop`. Uwaga: pętle nigdy nie zakończą się w konfiguracji `bylo-cos`, więc nie trzeba mnożyć reguł powyżej.

Do tego jeszcze reguły propagowania konfiguracji `bylo-cos`, na przyklad:

$$\frac{S_1, s \to s', \texttt{bylo-exit}}{S_1; S_2, s \to s',\ \texttt{bylo-exit}} \qquad \frac{S_1, s \to s' \qquad S_2, s' \to s'', \texttt{bylo-exit}}{S_1; S_2, s \to s'',\ \texttt{bylo-exit}}$$

(dla `bylo-continue` reguły są takie same).

Rozwiązanie, semantyka operacyjna: Tutaj nie musimy rozbudowywać konfiguracji, ale wygodnie jest rozszerzyć składnię o konstrukcję binarną `then`, służącą do rozwijania pętli, z regułą:

$$\frac{}{\texttt{loop}\ S, s \to S\ \texttt{then loop}\ S, s}$$

Propagowanie wykonania w `then`:

$$\frac{S_1, s \to S_1', s'}{S_1\ \texttt{then}\ S_2, s \to S_1'\ \texttt{then}\ S_2, s'} \qquad \frac{}{\texttt{skip then S}, \texttt{s} \to \texttt{S}, \texttt{s}}$$

Teraz `exit` i `continue` możemy propagować:

$$\frac{}{\texttt{exit}; S, s \to \texttt{exit}, s} \qquad \frac{}{\texttt{continue}; S, s \to \texttt{continue}, s}$$

I wreszcie interpretować:

$$\frac{}{\texttt{exit then}\ S, s \to \texttt{skip}, s} \qquad \frac{}{\texttt{continue then}\ S, s \to S, s}$$