SEMANTYKA I WERYFIKACJA - ĆW. 5

Dalszy ciąg semantyki naturalnej.

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

   Rozwiązanie: Bardzo łatwo, reguła dla `let`:

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

   Zauważmy, jak łatwo przyszło dobrze zdefiniować zmiany i widoczność wartościowań w porównaniu z semantyką małych kroków.

2. **Zadanie:** Zrobić to samo dla strategii leniwej: obliczamy wartość zmiennej `let` tylko wtedy, kiedy jest to konieczne, tzn. np.

   $$\texttt{let } y = x + x \texttt{ in } 7$$

   powinno się obliczyć do 7, a przy naszej dotychczasowej semantyce, w środowisku gdzie wartość x nie jest określona, nie obliczy się w ogóle. (W semantyce, gdzie wartościowania są funkcjami częściowymi.)

   Rozwiązanie: Pierwszy pomysł: wartościowanie powinno przypisywać zmiennej nie liczbę, a wyrażenie. Czyli

   $$\mathbf{State} = \mathbf{Var} \to \mathbf{Exp}$$

   (chodzi oczywiście o funkcje częściowe). Reguły dla zmiennych i `let`:

   $$\frac{s(x), s \to n}{x, s \to n} \qquad \frac{e_2, s[x \mapsto e_1] \to n}{\texttt{let } x = e_1 \texttt{ in } e_2,\ s \to n}$$

   Ćwiczenie: zobaczyć jak to działa dla programów:

   - `let` $x = 2$ `in let` $x = 3$ `in` x
   - `let` $x = x + 1$ `in` x (tu nie da się wyprowadzić żadnej wartości)
   - `let` $y = x + 1$ `in let` $x = 10$ `in` y (to się ewaluuje do 11).

   Coś tu nie gra! Ostatnie wyrażenie nie powinno mieć wartości. Zrobiliśmy tutaj semantykę z *wiązaniem dynamicznym*: zmienna jest interpretowana w takim środowisku, w jakim jest użyta.

   Tymczasem w większości języków (oprócz języków makr) chodzi nam o *wiązanie statyczne*: zmienna jest interpretowana w takim środowisku, w którym była zadeklarowana/związana.

   Drugi pomysł: dla każdej zmiennej pamiętamy nie tylko wyrażenie, ale także wartościowanie z miejsca deklaracji:

   $$\mathbf{State} = \mathbf{Var} \to \mathbf{Exp} \times \mathbf{State}$$

Uwaga: to jest równanie rekurencyjne, które nie ma rozwiązania. Wspomnieć o *teorii dziedzin*. Tutaj jest prostsze rozwiązanie:

$$\begin{aligned}
\mathbf{State}_1 &= \mathbf{Var} \to \mathbf{Exp} \\
\mathbf{State}_2 &= \mathbf{Var} \to \mathbf{Exp} \times \mathbf{State}_1 \\
\mathbf{State}_3 &= \mathbf{Var} \to \mathbf{Exp} \times \mathbf{State}_2 \\
&\dots \\
\mathbf{State} &= \bigcup_{n \in \mathbb{N}} \mathbf{State}_n
\end{aligned}$$

To działa tylko dlatego, że w naszych wyrażeniach nie ma rekursji.

Reguły dla zmiennych `let`:

$$\frac{s(x) = (e, s') \qquad e, s' \to n}{x, s \to n} \qquad\qquad \frac{e_2, s[x \mapsto (e_1, s)] \to n}{\texttt{let } x = e_1 \texttt{ in } e_2,\ s \to n}$$

3. Wróćmy do semantyki gorliwej z zadania 1 i rozszerzmy język o lambda-abstrakcje, otrzymując prosty język funkcyjny (jeszcze bez rekursji):

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

Napisać semantykę naturalną. Ma wyjść wiązanie statyczne, tj. wyrażenie

$$\texttt{let } x = 7 \texttt{ in let } f = \lambda z.z + x \texttt{ in let } x = 3 \texttt{ in } f(10)$$

ma się wyewaluować do 17, a nie do 13. Zróbmy przekazywanie parametrów przez wartość. Ciała funkcji nie obliczamy, dopóki nie przekażemy jej (obliczonego) argumentu.

Przykład: wyrażenie

$$\texttt{let } g = \lambda f.\lambda x.f(f(x)) \texttt{ in } g(\lambda x.x + 1)7$$

powinno się ewaluować do 9.

Rozwiązanie: Wartościami wyrażeń są teraz nie tylko liczby, ale także funkcje. Funkcję $\lambda x.e$ możemy reprezentować jako trójkę: $\langle x, e, s \rangle$, gdzie s to wartościowanie z miejsca deklaracji (żeby uzyskać wiązanie statyczne). Definiujemy więc zbiory:

$$\begin{aligned}
\mathbf{Value} &= \mathbf{Num} \cup (\mathbf{Var} \times \mathbf{Exp} \times \mathbf{State}) \\
\mathbf{State} &= \mathbf{Var} \to \mathbf{Value}
\end{aligned}$$

(to znowu jest równanie rekurencyjne i trzeba tu zastosować ten sam trik co poprzednio).

Reguły dla wartości:

$$\frac{}{n, s \to n} \qquad \frac{}{x, s \to s(x)} \qquad \frac{}{\lambda x.e,\ s \to \langle x, e, s \rangle}$$

Reguły dla dodawania są łatwe. Reguła dla `let` jest taka jak zwykle:

$$\frac{e_1, s \to v_1 \qquad e_2, s[x \mapsto v_1] \to v_2}{\texttt{let } x = e_1 \texttt{ in } e_2,\ s \to v_2}$$

I reguła dla aplikacji:

$$\frac{e_1, s \to \langle x, e_3, s' \rangle \qquad e_2, s \to v \qquad e_3, s'[x \mapsto v] \to v'}{e_1(e_2), s \to v'}$$

4. Teraz przekazywanie leniwe (przez nazwę), ale z zachowaniem statycznej widoczności. Definicje dziedzin:

$$\mathbf{Value} = \mathbf{Num} \cup (\mathbf{Var} \times \mathbf{Exp} \times \mathbf{State})$$
$$\mathbf{State} = \mathbf{Var} \to \mathbf{Exp} \times \mathbf{State}$$

(**Value** jak wyżej, **State** trochę uproszczone). Reguły dla zmiennych i `let` pozostają jak w zadaniu 2:

$$\frac{s(x) = (e, s') \qquad e, s' \to v}{x, s \to v} \qquad\qquad \frac{e_2, s[x \mapsto (e_1, s)] \to v}{\texttt{let } x = e_1 \texttt{ in } e_2,\ s \to v}$$

a dla aplikacji reguła jest nowa:

$$\frac{e_1, s \to \langle x, e_3, s' \rangle \qquad e_3, s'[x \mapsto (e_2, s)] \to v}{e_1(e_2),\ s \to v}$$

5. Wreszcie przekazywanie przez wartość z dynamiczną widocznością:

$$\mathbf{Value} = \mathbf{Num} \cup (\mathbf{Var} \times \mathbf{Exp})$$
$$\mathbf{State} = \mathbf{Var} \to \mathbf{Value}$$

$$\frac{}{n, s \to n} \qquad \frac{}{x, s \to s(x)} \qquad \frac{}{\lambda x.e,\ s \to \langle x, e \rangle}$$

$$\frac{e_1, s \to v_1 \qquad e_2, s[x \mapsto v_1] \to v_2}{\texttt{let } x = e_1 \texttt{ in } e_2,\ s \to v_2}$$

$$\frac{e_1, s \to \langle x, e_3 \rangle \qquad e_2, s \to v \qquad e_3, s[x \mapsto v] \to v'}{e_1(e_2), s \to v'}$$

6. Zrobić przekazywanie przez nazwę z dynamiczną widocznością (czyli po prostu makra).

7. Napisać programy, które rozróżniają te cztery mechanizmy.

   - statyczna widoczność przez wartość vs. statyczna widoczność przez nazwę:

     $$(\lambda x.7)y$$

   - statyczna widoczność przez wartość vs. dynamiczna widoczność przez wartość:

     $$\texttt{let } x = 7 \texttt{ in let } f = \lambda z.z + x \texttt{ in let } x = 3 \texttt{ in } f(10)$$