Semantyka *call_by_value*

Antoni Kościelski

1 Definicja semantyki

Semantyka *call_by_value* jest pojęciem z teorii języków programowania, w której oprócz analizy różnych konstrukcji stosowanych w językach programowania wyjaśnia się także, na czym polega wykonanie programu lub nawet tworzy i analizuje aparat do opisu działania programów. W przypadku rachunku lambda odpowiednikiem programu jest term, a wykonanie programu polega na wyliczeniu jego wartości. Wyliczenie wartości aplikacji AB jest interpretowane jako wykonanie funkcji A z parametrem B, który w semantyce *call_by_value* jest jakby wywoływany przez wartość, a więc wymaga wyliczenia przed przystąpieniem do wykonywania funkcji. Samo wykonanie funkcji polega na wykonaniu β-redukcji. Jedną z metod definiowania semantyki jest opisanie jednego kroku wykonania programu (jest to tzw. semantyka małych kroków). Wtedy wykonanie programu polega na iterowaniu wykonania pojedyńczych kroków. W związku z tym opiszemy relacją $A \to B$, którą będziemy interpretować jako stwierdzenie, że term B otrzymujemy z A w wyniku wykonania pojedyńczego przekształcenia.

Przejdźmy do formalnej definicji tej semantyki. Przyjmujemy, że symbole T z indeksami będą oznaczać dowolne termy λ-rachunku, a symbole V – wartości.

Wartościami będziemy nazywać dowolne abstrakcje, czyli termy zaczynające się symbolem λ. Oprócz wartości (potencjalnych) zdefiniujemy też pojęcie wartości termu T. Wartością termu T będzie zawsze jedna z wartości, a więc abstrakcja odpowiednio związana z termem T. Teraz tylko zasygnalizujemy, że wartością dowolnej abstrakcji jest ona sama.

Relację redukcji w jednym kroku w semantyce *call_by_value* definiujemy przez indukcję przyjmując, że spełnia następujące reguły:

$$\frac{T_1 \to T_1'}{T_1T_2 \to T_1'T_2}, \quad \frac{T_2 \to T_2'}{VT_2 \to VT_2'} \quad \text{oraz} \quad \frac{}{(\lambda x\ T)V \to T[x := V]}.$$

Aby zrozumieć tę definicję zauważmy, że możemy przekształcać tylko aplikacje, za to na trzy sposoby (zależnie od postaci T_1T_2, VT_2 lub $(\lambda x\ T)V$). W żaden sposób nie przekształcamy zmiennych, abstracji, ani termów postaci xT ze zmienną x. Mając aplikację możemy przedstawić ją w postaci $T_1T_2 \ldots T_{n-1}T_n = (T_1T_2 \ldots T_{n-1})T_n$, gdzie T_1 nie jest już aplikacją, a więc jest albo zmienną, albo wartością. Dzięki pierwszej regule, przekształcanie takiego termu sprowadza się do przekształcania jego fragmentu T_1T_2. Jeżeli T_1 jest zmienną, to obliczenia kończą się bez wyliczenia wartości. Jeżeli T_1 jest wartością, to stosując drugą regułę próbujemy wyliczyć wartość termu T_2. Jeżeli zakończy się to sukcesem, to trzecia reguła pozwala zastosować β-redukcję.

Jest oczywiste, że redukcja $\to$ jest β-redukcją z dodatkowymi ograniczeniami.

Zauważmy, że mamy

Lemat 1.1 *Dla dowolnego T_1 istnieje najwyżej jeden term T_2 taki, że $T_1 \to T_2$.* □

Wartością termu T nazywamy wartość V taką, że $T \to^* V$. Oczywiście, jeżeli wartość termu istnieje, to jest jednoznacznie wyznaczona.

Widać również, że redukowanie termu T może zakończyć się na trzy sposoby: albo utworzeniem nieskończonej redukcji $T = T_0 \to T_1 \to \ldots \to T_n \to \ldots$, albo utworzeniem skończonej redukcji zakończonej termem, który jednak nie jest wartością, lub też znalezieniem wartości termu T.

2 Wartości boolowskie i testowanie

Przypomnijmy, że w rachunku lambda wartości boolowskie reprezentujemy zwykle jako termy $K = \lambda xy.\ x$, odpowiadający prawdzie, oraz $K^* = \lambda xy.y$, oznaczający fałsz.

W dalszych rachunkach symbolem $\overline{A}$ będziemy oznaczać wartość termu A.

Przy takiej definicji wartości logicznych term *test*, czyli if_then_else definiuje się zwykle jako $\lambda xyz.\ xyz$. Prześledźmy, jak w semantyce *call_by_value* redukuje się wyrażenie postaci

$$test\ A\ B\ C \equiv if\ A\ then\ B\ else\ C \equiv (\lambda xyz.\ xyz)ABC \to^* (\lambda xyz.\ xyz)\overline{A}BC \to$$

$$\to (\lambda yz.\ \overline{A}yz)BC \to^* (\lambda yz.\ \overline{A}yz)\overline{B}C \to^* \overline{A}\ \overline{B}\ \overline{C}.$$

Wobec tego

$$test\ K\ B\ C \equiv if\ K\ then\ B\ else\ C \to^* K\overline{B}\overline{C} \to (\lambda y.\ \overline{B})\overline{C} \to \overline{B}$$

oraz

$$test\ K^*\ B\ C \equiv if\ K^*\ then\ B\ else\ C \to^* K^*\overline{B}\overline{C} \to (\lambda y.y)\ \overline{C} \to \overline{C}.$$

Przeprowadzone rachunki wymagają jednak drobnego założenia o istnieniu wartości termów A, B i C. Możemy też posłużyć się bardziej skomplikowanym wyrażeniem

$$test\ K\ (\lambda t.\ B)\ (\lambda t.\ C)\ I,$$

które redukuje się w następujący sposób:

$$test\ K\ (\lambda t.\ B)\ (\lambda t.\ C)\ I \to^* K\ (\lambda t.\ B)\ (\lambda t.\ C)\ I \equiv (\lambda xy.\ x)\ (\lambda t.\ B)\ (\lambda t.\ C)\ I \to$$

$$\to (\lambda y.\ (\lambda t.\ B))\ (\lambda t.\ C)\ I \to (\lambda t.\ B)\ I \to B.$$

Podobnie,

$$test\ K^*\ (\lambda t.\ B)\ (\lambda t.\ C)\ I \to^* C.$$

W dalszym ciągu tego tekstu liczby naturalne będą reprezentowane poprzez numerały Churcha, a najważniejsze będzie testowanie, czy dany numerał reprezentuje 0. Term wykonujący to testowanie będziemy oznaczać symbolem Z. Możemy przyjąć, że

$$Z \equiv \lambda x.\ x\ (\lambda y.\ K^*)\ K.$$

Wtedy

$$Z\ c_0 \equiv Z\ (\lambda fx.\ x) \to (\lambda fx.\ x)\ (\lambda y.\ K^*)\ K \to (\lambda x.x)\ K \to K,$$

a także

$$Z\ c_{n+1} \equiv Z\ (\lambda fx.\ f^{n+1}(x)) \to (\lambda fx.\ f^{n+1}(x))\ (\lambda y.\ K^*)\ K \to^*$$

$$\to^* (\lambda y.\ K^*)^{n+1}(K) \to (\lambda y.\ K^*)^n(K^*) \to (\lambda y.\ K^*)^{n-1}(K^*) \to^* K^*.$$

3 Punkty stałe i rekursja

Spróbujemy teraz pokazać na, mam nadzieję, dostatecznie ogólnym przykładzie, że klasa funkcji obliczalnych zgodnie z semantyką *call_by_value* jest zamknięta ze względu na rekursję prostą. Zrobimy to posługując się odpowiednim operatorem punktu stałego. Jak zwykle, będzie nam potrzebny term reprezentujący operację poprzednika. Na razie będziemy posługiwać się takim termem wierząc, że uda się go zdefiniować, i będziemy go oznaczać symbolem P.

Przypuśćmy, że $f : N \to N$ jest funkcją taką, że

$$f(0) = m \;\text{ oraz }\; f(n+1) = h(f(n), n)$$

dla wszystkich liczb naturalnych n. Przyjmijmy też, że H jest termem definiującym funkcję h. Tak więc term definiujący funkcję f powinien spełniać równość

$$F\ x = if\ (Z\ x)\ then\ c_m\ else\ (H\ (F\ (P\ x))\ (P\ x)).$$

Równości tej nadamy nieco inną postać przy okazji wykorzystując znany już term *test*:

$$F\ x = test\ (Z\ x)\ (\lambda\ t.c_m)\ ((\lambda yt.\ H\ (F\ y)\ y)\ (P\ x))\ I.$$

Niech teraz

$$G = \lambda fx.test\ (Z\ x)\ (\lambda\ t.c_m)\ ((\lambda yt.\ H\ (f\ y)\ y)\ (P\ x))\ I.$$

Term F możemy teraz zdefiniować jako $fix\ G$, czyli wynik zaplikowania operatora punktu stałego fix do termu G. Powinniśmy jednak użyć specjalnego operatora fix.

Zadanie 3.1 Prześledź, co się stanie, jeżeli w roli operatora punkty stałego użyjemy operatora Y lub operatora Turinga Θ.

Niech

$$fix \equiv \lambda f.\ A_f\ A_f, \;\text{ gdzie }\; A_f \equiv \lambda x.\ f(\lambda z.\ xxz).$$

Zauważmy, że

$$fix\ G \equiv (\lambda f.\ A_f\ A_f)G \to A_G\ A_G \to G\ (\lambda z.\ A_G\ A_G\ z).$$

Stąd (przekształcane redeksy zostały podkreślone)

$$\begin{aligned}
&\underline{fix\ G}\ c_{n+1} \to \\
&\to\ \underline{A_G\ A_G}\ c_{n+1} \\
&\to\ \underline{G\ (\lambda x.\ A_G\ A_G\ x)\ c_{n+1}} \\
&\to^*\ test\ (\underline{Z\ c_{n+1}})\ (\lambda\ t.\ c_m)\ ((\lambda yt.\ H\ ((\lambda z.\ A_G\ A_G\ z)\ y)\ y)\ (P\ c_{n+1}))\ I \\
&\to^*\ \underline{test\ K^*\ (\lambda\ t.\ c_m)}\ ((\lambda yt.\ H\ ((\lambda z.\ A_G\ A_G\ z)\ y)\ y)\ (P\ c_{n+1}))\ I \\
&\to^*\ (\lambda z.\ K^*\ (\lambda\ t.\ c_m)\ z)\ ((\lambda yt.\ H\ ((\lambda z.\ A_G\ A_G\ z)\ y)\ y)\ (\underline{P\ c_{n+1}}))\ I \\
&\to^*\ (\lambda z.\ K^*\ (\lambda\ t.\ c_m)\ z)\ (\underline{(\lambda yt.\ H\ ((\lambda z.\ A_G\ A_G\ z)\ y)\ y)\ c_n})\ I \\
&\to\ \underline{(\lambda z.\ K^*\ (\lambda\ t.\ c_m)\ z)\ (\lambda t.\ H\ ((\lambda z.\ A_G\ A_G\ z)\ c_n)\ c_n)}\ I \\
&\to\ \underline{K^*\ (\lambda\ t\ .c_m)\ (\lambda t.\ H\ ((\lambda z.\ A_G\ A_G\ z)\ c_n)\ c_n)}\ I \\
&\to^*\ \underline{(\lambda t.\ H\ ((\lambda z.\ A_G\ A_G\ z)\ c_n)\ c_n)\ I} \\
&\to\ H\ (\underline{(\lambda z.\ A_G\ A_G\ z)\ c_n})\ c_n \\
&\to^*\ H\ (A_G\ A_G\ c_n)\ c_n.
\end{aligned}$$

Jeżeli teraz założymy, że wartością termu $A_G\ A_G\ c_n$ jest $c_{f(n)}$, a wartością $H\ c_{f(n)}\ c_n$ jest $c_{h(f(n),n)} \equiv c_{f(n+1)}$, to otrzymamy, że

$$fix\ G\ c_{n+1} \to A_G\ A_G\ c_{n+1} \to^* H\ (A_G\ A_G\ c_n)\ c_n \to^* H\ c_{f(n)}\ c_n \to^* c_{h(f(n),n)} \equiv c_{f(n+1)}.$$

Łatwo przekonać się powtarzając przedstawione rachunki, że

$$fix\ G\ c_0 \to A_G\ A_G\ c_0 \to^* (\lambda t.\ c_m)\ I \to c_m.$$

Z zasady indukcji matematycznej otrzymujemy więc, że w semantyce *call_by_value* termy $fix\ G$, $A_G\ A_G$, a także będące wartościami termy $\lambda x.\ fix\ G\ x$ oraz $\lambda x.\ A_G\ A_G\ x$ definiują funkcję f, o ile funkcja h jest odpowiednio zdefiniowana.

4 Pary i operacja poprzednika

Funkcje pary $Pair \equiv [\cdot,\cdot]$ i odczytujące współrzędne pary fst i snd definiujemy jak zwykle przyjmując

$$Pair \equiv [\cdot,\cdot] \equiv \lambda xyb.\ bxy, \quad fst \equiv \lambda p.\ pK \quad \text{oraz} \quad snd \equiv \lambda p.\ pK^*.$$

Nietrudno sprawdzić, że

$$Pair\ X\ Y \equiv [\cdot,\cdot]\ X\ Y \to^* \lambda b.\ \overline{X}\ \overline{Y} \equiv [\overline{X},\overline{Y}]$$

oraz

$$fst\ (Pair\ X\ Y) \to^* \overline{X},$$
$$snd\ (Pair\ X\ Y) \to^* \overline{Y}$$

przynajmniej dla termów X i Y mających wartość ($\overline{X}$ oznacza wartość termu X).

Mając operację pary i posługując się numerałami Churcha w standardowy sposób definiujemy poprzednik P. Weźmy

$$ss \equiv \lambda p.\ (\lambda x.\ Pair\ x\ (S\ x))(snd\ p)$$

oraz

$$P \equiv \lambda m.\ fst\ (m\ ss\ [c_0,c_0]).$$

Wtedy, jeżeli term S oznacza operację następnika, to

$$\begin{aligned}
&ss\ [c_m,c_n] \equiv \\
&\equiv\ (\lambda p.\ (\lambda x.\ Pair\ x\ (S\ x))(snd\ p))\ [c_m,c_n] \to (\lambda x.\ Pair\ x\ (S\ x))(snd\ [c_m,c_n]) \to \\
&\to\ (\lambda x.\ Pair\ x\ (S\ x))\ c_n \to Pair\ c_n\ (S\ c_n)) \to (\lambda yb.\ b\ c_n\ y)(S\ c_n) \to^* \\
&\to^*\ (\lambda yb.\ b\ c_n\ y)\ c_{n+1} \to \lambda b.\ b\ c_n\ c_{n+1} \equiv [c_n,c_{n+1}].
\end{aligned}$$

Mamy także

$$\begin{aligned}
&P\ c_0 \equiv \\
&\equiv\ (\lambda m.\ fst\ (m\ ss\ [c_0,c_0]))\ c_0 \to fst\ (c_0\ ss\ [c_0,c_0]) \equiv fst\ ((\lambda fx.\ x)\ ss\ [c_0,c_0]) \to \\
&\to\ fst\ ((\lambda x.\ x)\ [c_0,c_0]) \to fst\ [c_0,c_0] \to^* c_0
\end{aligned}$$

oraz

$$\begin{aligned}
&P\ c_{n+1} \equiv \\
&\equiv\ (\lambda m.\ fst\ (m\ ss\ [c_0,c_0]))\ c_{n+1} \to fst\ (c_{n+1}\ ss\ [c_0,c_0]) \equiv \\
&\equiv\ fst\ ((\lambda fx.\ f^{n+1}(x))\ ss\ [c_0,c_0]) \to^* fst\ ((ss)^{n+1}([c_0,c_0])) \to^* \\
&\to^*\ fst\ ((ss)^n(ss\ [c_0,c_0])) \to^* fst\ ((ss)^n([c_0,c_1])) \to^* fst\ ((ss)^{n-1}([c_1,c_2])) \to^* \\
&\to^*\ fst\ ((ss)^{n-k}([c_k,c_{k+1}])) \to^* fst\ [c_n,c_{n+1}] \to^* c_n
\end{aligned}$$