SEMWER zadanie 2, am418402

December 30, 2021

Dziedziny semantyczne

Stan

Ponieważ wszystkie zmienne są globalne, oraz traktujemy je jako zadeklarowane i z nadaną wartością, to nie potrzebujemy środowiska zmiennych, a stan ma po prostu postać

$$\mathrm{State} = \mathrm{Var} \longrightarrow \mathrm{Int}$$

i skoro wszystkie zmienne mają wartość, nie jest to funkcja częściowa. Powstaje tutaj pytanie, jaką wartość mają zmienne, którym nie nadano jeszcze w programie żadnej wartości - wszak można się do nich odwołać. To jednak nie zależy od denotacji instrukcji i wyrażeń, więc nie jest częścią zadania. Można to ustalić np. podając stan "początkowy" w denotacji programów.

Typy kontynuacji

Skoro wynikiem działania instrukcji ma być stan końcowy, to kontynuacje mają postać

$$\mathrm{Cont} = \mathrm{State} \rightharpoonup \mathrm{State}$$

Ale dla rozróżnienia wprowadzę typ $\mathrm{Ans} = \mathrm{State}$ i będę pisał

$$\mathrm{Cont} = \mathrm{State} \rightharpoonup \mathrm{Ans}$$

Dalej, dla kontynuacji wyrażeń arytmetycznych:

$$\mathrm{Cont}_E = \mathrm{Int} \longrightarrow \mathrm{State} \rightharpoonup \mathrm{Ans}$$

Tutaj istotne jest, że wyrażenia mogą zmieniać stan, więc nie może być $\mathrm{Cont}_E = \mathrm{Num} \rightharpoonup \mathrm{Ans}$. Dla wyrażeń boolowskich:

$$\mathrm{Cont}_B = \mathrm{Bool} \longrightarrow \mathrm{State} \rightharpoonup \mathrm{Ans}$$

a dla deklaracji:

$$\mathrm{Cont}_D = \mathrm{FEnv} \rightharpoonup \mathrm{Ans}$$

Gdzie FEnv to opisane niżej środowisko funkcji. Tym razem jest istotne, że deklaracje nie zmieniają stanu.

Środowisko funkcji

Środowisko funkcji ma postać

$$\text{FEnv} = \text{FName} \rightharpoonup \text{Fun}$$

Gdzie FName to nazwy funkcji, natomiast funkcje reprezentuje typ Fun:

$$\text{Fun} = (\text{Cont} \longrightarrow \text{Cont}_E \longrightarrow \text{State} \rightharpoonup \text{Ans}) \times \text{EXPR}$$

gdzie

$$\text{EXPR} = \text{FEnv} \longrightarrow \text{Cont}_E \longrightarrow \text{State} \rightharpoonup \text{Ans}$$

Jak warto zauważyć, w pierwszej części produktu podajemy jako argumenty dwie różne kontynuacje. Ma to na celu umożliwienie zarówno wykonania całości ciała funkcji, jak i wyjście z niej wcześniej za pomocą instrukcji `return`. Ta pierwsza część intuicyjnie jest odpowiedzialna za wykonanie ciała funkcji. Natomiast druga część produktu to wyrażenie odpowiedzialne za domyślny wynik działania funkcji. W środowisku należy pamiętać całe to wyrażenie, gdyż będzie ono wyliczane przed każdym wywołaniem funkcji. Typ EXPR jest taki sam jak typ $\mathcal{E}[\![e]\!]$, czyli -co istotne - nie ma jeszcze zaaplikowanego środowiska funkcji. Skutkuje to tym, że przy wyliczaniu wyrażenia przed wywołaniem funkcji obowiązuje każdorazowo środowisko funkcji z momentu wywołania. Jest to świadoma decyzja z mojej strony. W zadaniu nie jest doprecyzowane, czy ma wtedy obowiązywać środowisko z miejsca wywołania, czy deklaracji - mowa tylko o stanie. W mojej opinii takie rozwiązanie jest bardziej intuicyjne.

Typy funkcji semantycznych

Dla instrukcji:

$$\mathcal{I}[\![]\!] : \text{Instr} \longrightarrow \text{FEnv} \longrightarrow \text{Cont} \longrightarrow \text{Cont}_E \longrightarrow \text{State} \rightharpoonup \text{Ans}$$

Ponownie, i z analogicznych powodów, mamy tu dwie różne kontynuacje. Dla wyrażeń arytmetycznych jest:

$$\mathcal{E}[\![]\!] : \text{Expr} \longrightarrow \text{FEnv} \longrightarrow \text{Cont}_E \longrightarrow \text{State} \rightharpoonup \text{Ans}$$

i tym razem oczywiście nie potrzebujemy dwóch różnych kontynuacji. Dla wyrażeń boolowskich analogicznie:

$$\mathcal{B}[\![]\!] : \text{BExpr} \longrightarrow \text{FEnv} \longrightarrow \text{Cont}_B \longrightarrow \text{State} \rightharpoonup \text{Ans}$$

Dla deklaracji:

$$\mathcal{D}[\![]\!] : \text{FDecl} \longrightarrow \text{FEnv} \longrightarrow \text{Cont}_D \rightharpoonup \text{Ans}$$

Denotacje

Denotacje wyrażeń

$$\mathcal{E}[\![n]\!]\rho_F\ \kappa_E\ s = \kappa_E(\mathcal{N}[\![n]\!])\ s$$

$$\mathcal{E}[\![x]\!]\rho_F\ \kappa_E\ s = \kappa_E(s\ x)\ s$$

$$\mathcal{E}[\![e_1 + e_2]\!]\ \rho_F\ \kappa_E\ s = \mathcal{E}[\![e_1]\!]\ \rho_F\ (\lambda n_1.\ \mathcal{E}[\![e_2]\!]\ \rho_F\ (\lambda n_2.\ \kappa_E(n_1 + n_2)))\ s$$

Denotacje dla odejmowania i mnożenia są analogiczne.

$$\mathcal{E}[\![f()]\!]\ \rho_F\ \kappa_E\ s = e_d\ \rho_F\ (\lambda n.\ \beta\ \kappa_E(n)\ \kappa_E)\ s$$

gdzie

$$\rho_F(f) = (\beta, e_d)$$

Innymi słowy, wyciągamy ze środowiska funkcji krotkę oznaczającą ciało funkcji i jej wyrażenie domyślne (tj. określające domyślny wynik), i przekazujemy kontrolę wyrażeniu domyślnemu, mówiąc, że tym, co się stanie z jego wynikiem n, będzie nowo skonstruowana kontynuacja wyrażenia arytmetycznego zależna od n. Ta kontynuacja z kolei jest - nieformalnie mówiąc - wykonaniem ciała funkcji, biorąc jako kontynuację domyślną $\kappa_E(n)$, a jako kontynuację dla instrukcji `return` - κ_E.

$$\mathcal{B}[\![\text{true}]\!]\ \rho_F\ \kappa_B\ s = \kappa_B\ \text{tt}\ s$$

analogicznie dla fałszu.

$$\mathcal{B}[\![\text{not}\ b]\!]\ \rho_F\ \kappa_B\ s = \mathcal{B}[\![b]\!]\ \rho_F\ (\lambda d.\ \kappa_B\ (\neg d))s$$

$$\mathcal{B}[\![b_1 \wedge b_2]\!]\ \rho_F\ \kappa_B\ s = \mathcal{B}[\![b_1]\!]\ \rho_F\ (\lambda d_1.\ \mathcal{B}[\![b_2]\!]\ \rho_F\ (\lambda d_2.\kappa_B(d_1 \wedge d_2)))s$$

$$\mathcal{B}[\![e_1 < e_2]\!]\ \rho_F\ \kappa_B\ s = \mathcal{E}[\![e_1]\!]\ \rho_F\ (\lambda n_1.\ \mathcal{E}[\![e_2]\!]\ \rho_F\ (\lambda n_2.\ \kappa_B(n_1 < n_2)))\ s$$

Dla $e_1 = e_2$ analogicznie. Drobna uwaga: być może użyty tu zapis $n_1 < n_2$ nie jest do końca formalny. Oczywiście chodzi tu o wyrażenie boolowskie które przyjmuje wartość tt, gdy $n_1 < n_2$, a ff w przeciwnym wypadku.

Denotacje instrukcji

$$\mathcal{I}[\![x := e]\!]\ \rho_F\ \kappa\ \kappa_E\ s = \mathcal{E}[\![e]\!]\ \rho_F\ (\lambda n.\lambda s'.\ \kappa\ s'[x \mapsto n])\ \ s$$

$$\mathcal{I}[\![I_1; I_2]\!]\ \rho_F\ \kappa\ \kappa_E\ s = \mathcal{I}[\![I_1]\!]\ \rho_F\ (\mathcal{I}[\![I_2]\!]\ \rho_F\ \kappa\ \kappa_E\ )\ \kappa_E\ s$$

$$\mathcal{I}[\![\text{if}\ b\ \text{then}\ I_1\ \text{else}\ I_2]\!]\ \rho_F\ \kappa\ \kappa_E\ s = \mathcal{B}[\![b]\!]\ \rho_F\ (\lambda d.ifte(d,\ \mathcal{I}[\![I_1]\!]\ \rho_F\ \kappa\ \kappa_E,\ \mathcal{I}[\![I_2]\!]\ \rho_F\ \kappa\ \kappa_E))\ \ s$$

"Robocza", niestałopunktowa denotacja instrukcji `while` wyglądałaby tak:

$$\mathcal{I}[\![\text{while}\ b\ \text{do}\ I]\!]\ \rho_F\ \kappa\ \kappa_E = \mathcal{B}[\![b]\!]\ \rho_F\ (\lambda d.ifte(d,\ \mathcal{I}[\![I]\!]\ \rho_F\ (\mathcal{I}[\![\text{while}\ b\ \text{do}\ I]\!]\ \rho_F\ \kappa\ \kappa_E\ )\ \kappa_E,\ \kappa))$$

Czyli $\mathcal{I}[\![\text{while } b \text{ do } I]\!]\ \rho_F\ \kappa\ \kappa_E\ = fix(\Phi)$ dla

$$\Phi(F) = \mathcal{B}[\![b]\!]\ \rho_F\ (\lambda d. ifte(d,\ \mathcal{I}[\![I]\!]\ \rho_F\ F\ \kappa_E,\ \kappa))$$

$$\mathcal{I}[\![\text{begin } d_F\ I \text{ end}]\!]\ \rho_F\ \kappa\ \kappa_E\ s = \mathcal{D}[\![d_F]\!]\ \rho_F\ (\lambda\rho.\ \mathcal{I}[\![I]\!]\ \rho\ \kappa\ \kappa_E\ s)$$

$$\mathcal{I}[\![\text{return } e]\!]\ \rho_F\ \kappa\ \kappa_E\ s = \mathcal{E}[\![e]\!]\ \rho_F\ (\lambda n.\ \kappa_E(n))\ s$$

Denotacje deklaracji

Tak jak przy instrukcji `while`, możemy roboczo skonstruować ”definicję” deklaracji funkcji (potencjalnie rekurencyjnej), która nie jest kompozycjonalna:

$$\mathcal{D}[\![\text{fun } f \text{ result } e\ do\ (I)]\!]\ \rho_F\ \kappa_D = \kappa_D\ (\rho_F[f \mapsto\ F])$$

gdzie

$$F = \langle \mathcal{I}[\![I]\!]\ \rho_F[f \mapsto F],\ \mathcal{E}[\![e]\!]\rangle$$

I teraz możemy przepisać to na poprawną definicję stałopunktową:

$$\mathcal{D}[\![\text{fun } f \text{ result } e\ do\ (I)]\!]\ \rho_F\ \kappa_D = \kappa_D\ (\rho_F[f \mapsto\ fix(\Phi)])$$

gdzie

$$\Phi(F) = \langle \mathcal{I}[\![I]\!]\ \rho_F[f \mapsto F],\ \mathcal{E}[\![e]\!]\rangle$$

Dalej dla złożenia deklaracji:

$$\mathcal{D}[\![d_{F1}; d_{F2}]\!]\ \rho_F\ \kappa_D = \mathcal{D}[\![d_{F1}]\!]\ \rho_F\ (\lambda\rho.\ \mathcal{D}[\![d_{F2}]\!]\ \rho\ \kappa_D)$$