SEMANTYKA I WERYFIKACJA - ĆW. 12

Kończymy kontynuacje.

1. Zróbmy pewien odpowiednik konstrukcji `call-cc` w języku imperatywnym: wrzućmy kontynuacje do języka TINY jako obiekty pierwszego rzędu, które mogą być wartościami zmiennych. Intuicja: kontynuacja to pełen stos wywołań i kontrola (tj. miejsce w aktualnej procedurze), ale bez stanu zmiennych. Rozszerzamy język o:

   - zmienne kontynuacyjne L, L' itd. (mogą być tylko globalne, dla uproszczenia)
   - instrukcję `set` L, która przypisuje L aktualną kontynuację,
   - instrukcję `goto` L, która do niej skacze.

   Mamy tu więc `goto` z dynamicznym ustawianiem etykiet i odbudowywaniem stosu przy skokach między procedurami. Uwaga, w odróżnieniu od `longjmp` z języka C, tutaj możemy skakać dowolnie, nie tylko w górę stosu wywołań.

   Rozwiązanie: Mamy dodatkową dziedzinę stanów kontynuacji (nie ma potrzeby rozbijać na stany i środowiska):

   $$\mathbf{Store} = (\mathbf{Loc} \to \mathbf{Num}) \times (\mathbf{CName} \to \mathbf{Cont})$$

   gdzie $\mathbf{Cont} = \mathbf{Store} \to \mathbf{Ans}$, jak zwykle (ale uwaga, definicja zrobiła się rekurencyjna). Typ semantyczny instrukcji jest jak poprzednio. Jeżeli pominiemy aspekty związane z funkcjam/procedurami, to wychodzi

   $$\mathcal{I}[\![]\!] : \mathbf{Instr} \to \mathbf{Env} \to \mathbf{Cont} \to \mathbf{Store} \to \mathbf{Ans}$$

   Równania semantyczne:

   $$\mathcal{I}[\![\texttt{set } G]\!] = \lambda\rho.\lambda\kappa.\lambda s.\kappa(s[G \mapsto \kappa])$$
   $$\mathcal{I}[\![\texttt{goto } G]\!] = \lambda\rho.\lambda\kappa.\lambda s.sGs$$

   Dodanie funkcji wiele nie zmienia, tylko komplikują się dziedziny.

2. Korutyny (coroutines), czyli wątki z przekazywaniem sterowania na ochotnika. Używane np. w programowaniu gier. Współbieżność bez zewnętrznego schedulera.

   $$I ::= \cdots \mid \texttt{fork } I \mid \texttt{yield} \mid \texttt{exit}$$

   Rozwiązanie: W stanie trzymamy kolejkę kontynuacji oczekujących na wykonanie. Zakładamy dziedzinę semantyczną $\mathbf{Queue} = \mathbf{Cont}^*$, z operacjami wstawiania, pobierania i wykrywania pustości.

   $$\mathbf{Cont} = \mathbf{Queue} \to \mathbf{Store} \to \mathbf{Ans}$$

Uwaga, to jest bardzo rekurencyjne równanie. Typ semantyczny instrukcji:

$$\mathcal{I}[\![]\!] : \mathbf{Instr} \to \mathbf{Env} \to \mathbf{Cont} \to \mathbf{Queue} \to \mathbf{Store} \to \mathbf{Ans}$$

Równania semantyczne:

$$\begin{aligned}
\mathcal{I}[\![\texttt{fork } I]\!] &= \lambda\rho.\lambda\kappa.\lambda q.\mathcal{I}[\![I]\!]\rho\kappa(\kappa :: q) \\
\mathcal{I}[\![\texttt{yield}]\!] &= \lambda\rho.\lambda\kappa.\lambda q.\kappa' q' \\
&\qquad \text{gdzie } (q', \kappa') = remove(\kappa :: q) \\
\mathcal{I}[\![\texttt{exit}]\!] &= \lambda\rho.\lambda\kappa.\lambda q.ifte(empty(q), s, \kappa' q') \\
&\qquad \text{gdzie } (q', k') = remove(q)
\end{aligned}$$

3. Śmieszny język programowania:

   $I ::= \texttt{skip} \mid x := e \mid I_1; I_2 \mid \texttt{if } b \texttt{ then } I_1 \texttt{ else } I_2 \mid \texttt{back} \mid \texttt{left } I \mid \texttt{right } I$

   Programy w tym języku mogą wykonywać się w dwie strony: od lewej do prawej oraz od prawej do lewej. Wykonanie rozpoczyna się od pierwszej instrukcji i przebiega od lewej do prawej. Instrukcja `back` powoduje zmianę kierunku wykonania. Instrukcja `left` I powoduje wykonanie I jeśli wykonanie przebiega od prawej do lewej, w przeciwnym razie jest to instrukcja pusta. Symetrycznie działa `right` .

   Rozwiązanie: Nie ma tu potrzeby rozdzielać stanów od środowisk, a więc:

   $$\mathbf{State} = \mathbf{Var} \to \mathbf{Num} \qquad \mathbf{Cont} = \mathbf{State} \to \mathbf{Ans}$$

   Można też zrobić bezpośrednią semantykę wyrażeń.

   Idea: każda instrukcja bierze dwie kontynujace (lewą i prawą) i zwraca dwie kontynuacje (lewą i prawą), czyli ma dwie funkcje semantyczne:

   $$\mathcal{I}_l[\![]\!], \mathcal{I}_r[\![]\!] : \mathbf{Instr} \to \mathbf{Cont} \to \mathbf{Cont}$$

   Większość równań jest bardzo prosta:

   $$\begin{aligned}
   \mathcal{I}_l[\![\texttt{skip}]\!] &= \lambda\kappa_l\lambda\kappa_r.\kappa_l \\
   \mathcal{I}_l[\![x := e]\!] &= \lambda\kappa_l\lambda\kappa_r\lambda s.\kappa_l(s[x \mapsto \mathcal{E}[\![e]\!]s]) \\
   \mathcal{I}_l[\![\texttt{back}]\!] &= \lambda\kappa_l\lambda\kappa_r.\kappa_r \\
   &itd.
   \end{aligned}$$

   Komplikaca pojawia się przy sekwencji (gdzieś musi się pojawić, bo w naszym języku można pisać pętle). Chciałoby się napisać:

   $$\mathcal{I}_l[\![I_1; I_2]\!] = \lambda\kappa_l\lambda\kappa_r.\mathcal{I}_l[\![I_1]\!](\mathcal{I}_r[\![I_2]\!]\kappa_l(\mathcal{I}_r[\![I_1]\!](\cdots)\kappa_r))\kappa_r$$

czyli można napisać tak:

$$\mathcal{I}_l[\![I_1; I_2]\!] = \lambda\kappa_l\lambda\kappa_r.\mathcal{I}_l[\![I_1]\!]F\kappa_r$$

gdzie

$$F = \mathcal{I}_l[\![I_2]\!]\kappa_l(\mathcal{I}_r[\![I_1]\!]F\kappa_r)$$

4. **Uwaga: to jest źle i nie wiem jak poprawić, nie zgadzają się typy w równaniu dla `yield`. Do przemyślenia.**

   Generatory. Weźmy język z funkcjami (dla uproszczenia: bez parametrów), ale zamiast `return` ma instrukcję

   $$I ::= \cdots \mid \texttt{yield}\ x$$

   która działa jak `return` , ale następne wywołanie funkcji skoczy nie do początku funkcji, a do następnej instrukcji po `yield`. (Można by też dopuścić `yield` e, ale dla uproszczenia zwracajmy tylko wartości zmiennych).

   Uwaga: rozważamy język bez rekursji, bo rekursja w połączeniu z `yield` musiałaby działać w jakiś dziwny sposób. Tak naprawdę generatory wyglądają trochę inaczej, tj. są bytami pierwszego rzędu i mogą być wartościami zmiennych.

   Rozwiązanie: Dane o funkcjach powinny teraz być częścią stanu, a nie środowiska. Ale naiwne zrobienie tego wprowadziłoby dynamiczną widoczność nazw funkcji, więc lepiej rozbić dane o funkcjach na część środowiskową i stanową, z odrębnymi "lokacjami na funkcje". Środowisko będzie też pamiętać nazwę aktualnej funkcji, pod odrębnym identyfikatorem *cur*:

   $$\begin{aligned}
   \mathbf{Env} &= (\mathbf{Var} \to \mathbf{Loc}) \times (\mathbf{FName} \to \mathbf{FLoc}) \times (\{cur\} \to \mathbf{FName}) \\
   \mathbf{Store} &= (\mathbf{Loc} \to \mathbf{Num}) \times (\mathbf{FLoc} \to \mathbf{Fun}) \\
   \mathbf{Fun} &= \mathbf{Cont}_E \to \mathbf{Cont} \\
   \mathbf{Cont}_E &= \mathbf{Num} \to \mathbf{Cont} \\
   \mathbf{Cont} &= \mathbf{Store} \to \mathbf{Ans}
   \end{aligned}$$

   Typ **Fun** jest taki jak poprzednio, ale bez parametru. Zauważmy że te równania są teraz bardzo rekurencyjne, bo **Store** jest typem argumentu **Cont** i $\mathbf{Cont}_E$.

   Funkcje semantyczne mają typy takie jak poprzednio:

   $$\begin{aligned}
   \mathcal{I}[\![]\!] &: \mathbf{Exp} \to \mathbf{Env} \to \mathbf{Cont}_E \to \mathbf{Cont} \to \mathbf{Store} \to \mathbf{Ans} \\
   \mathcal{E}[\![]\!] &: \mathbf{Exp} \to \mathbf{Env} \to \mathbf{Cont}_E \to \mathbf{Store} \to \mathbf{Ans} \\
   \mathcal{B}[\![]\!] &: \mathbf{BExp} \to \mathbf{Env} \to \mathbf{Cont}_B \to \mathbf{Store} \to \mathbf{Ans} \\
   \mathcal{D}[\![]\!] &: \mathbf{Decl} \to \mathbf{Env} \to \mathbf{Cont}_D \to \mathbf{Store} \to \mathbf{Ans}
   \end{aligned}$$

   teraz argument $\mathbf{Cont}_E$ w $\mathcal{I}[\![]\!]$ to kontynuacja przy `yield`, a nie przy `return` , którego (dla uproszczenia) nie ma.

Istotne równania semantyczne:

$$\begin{aligned}
\mathcal{I}[\![\texttt{yield}\ x]\!] &= \lambda\rho\lambda\gamma\lambda\kappa\lambda s.\gamma(s(\rho(x)))s[\rho(cur) \mapsto \kappa] \\
\mathcal{I}[\![f()]\!] &= \lambda\rho\lambda\kappa\lambda s.s(\rho(f))\kappa s \\
\mathcal{D}[\![\texttt{fun}\ f\ I]\!] &= \lambda\rho\lambda\kappa\lambda s.\kappa\rho[f \mapsto l]s[l \to F] \\
&\qquad l = \texttt{newfunloc}(s) \\
&\qquad F = \lambda\kappa'.\lambda s'.\mathcal{I}[\![I]\!]\rho[cur \mapsto f]\kappa'(err)s'
\end{aligned}$$

gdzie $err \in \mathbf{Cont}$ to jakaś "błędna" kontynuacja; to odpowiada semantyce, w której generator nigdy nie może się normalnie zakończyć (np. każdy generator jest postaci `while true I`)