Języki Programowania - Projekt

Wojciech Oziębły

wojciechoziebly@gmail.com

1. Składnia

$e := x \mid \underline{n} \mid e + e \mid e * e \mid -e \mid <> \mid < e, e > \mid \text{outl } e \mid \text{outr } e$
$\quad \mid \text{inl}\{\tau + \tau\}\ (e) \mid \text{inr}\{\tau + \tau\}\ (e) \mid$ 'match e with $\mid$ inr $x.e \mid$ inr $x.e$' $\mid \text{abort}\{\tau\}\ (e) \mid *e$
$c := \text{skip} \mid c\,;c \mid \text{if } e \text{ then } c \text{ else } c \mid \text{while } e \text{ do } c \mid \text{var } x := e \text{ in } c \mid \text{var } x := \text{new } e \text{ in } c \mid x := e \mid {}^*x := e$
$\quad \mid x := f(e_1, ..., e_n) \mid \text{switch } e \text{ inl } x\text{: } c \text{ inr } x\text{: } c \mid \text{abort} \mid \text{debug\_print } s$
$\tau := int \mid unit \mid \tau * \tau \mid void \mid \tau + \tau \mid \text{ptr } \tau \mid \text{t} \mid (\tau)$
$d_t := \text{type } t = \tau$
$d_v := x := e \mid x := \text{new } e$
$d_f := f(x_1 : \tau_1, ..., x_n : \tau_n) : \tau_r \text{ in vars } d_v{}^* \text{ in } c \text{ in return } e$
$p := d_t{}^* \text{ in } d_f{}^* \text{ in vars } d_v{}^* \text{ in } c$

Na wyrażenia składają się odczyt zmiennej, operacje na liczbach, parach i sumach rozłącznych. Eliminacja sumy pojawia się w wyrażeniach jako 'match with' i w instrukcjach jako 'switch', podobnie z abortem. Dodatkową instrukcją jest 'debug_print s' która wypisuje aktualne zmienne i ich wartości, stertę oraz identyfikator s. W definicjach funkcji należy podać typy wszystkich argumentów oraz wartości zwracanej, co ułatwia sprawdzenie typów związane z wykorzystaniem funkcji jak i samej definicji funkcji (aby sprawdzić typ funkcji, nie potrzeba informacji z poza jej definicji).

2. Reguły typowania

Mamy osądy typów dla: definicji typów, wyrażeń, instrukcji, definicji funkcji i całych programów. Wykorzystują one następujące konteksty:

Typy użytkownika: $\delta = (t : \tau)^*$
Funkcje: $\Delta = (f : (\tau_1, ..., \tau_2) \to \tau_r)^*$
Zmienne: $\Gamma = (x : \tau)^*$

2.1. Forma osądów

Dla typów użytkownika sprawdzamy czy są dobrze zdefiniowane, a dla jawnych typów (pojawiają się tylko przy definicji funkcji) sprawdzamy czy są dobrze sformułowane:

$$\delta \vdash \tau : \text{wf\_usertype}$$

$$\delta \vdash \tau : \text{type}$$

$$\delta \vdash \text{type } t = \tau : t \to \tau$$

Definicja funkcji podaje wprost jej typ. Jednak należy także potwierdzić, że jest ona dobrze zdefiniowana.

$$\delta; \Delta \vdash f(x_1 : \tau_1, ..., x_n : \tau_n) : \tau_r \text{ in vars } (y := e)^* \text{ in } c \text{ in return } e : \tau_f$$

Wyrażenia posiadają typ:

$$\delta; ; \Gamma \vdash e : \tau$$

Zmienne początkowe mają typ:

$$\delta; ; \Gamma \vdash d_t \to x : \tau$$

Instrukcje są dobrze sformułowane:

$$\delta; \Delta; \Gamma \vdash c \text{ wf}$$

I w końcu sprawdzamy czy cały program jest dobrze zdefiniowany.

$$\vdash d_t* \text{ in } d_f* \text{ in vars } (x := e)* \text{ in } c \text{ wf}$$

2.2. Typy użytkownika

Reguła dla definicji typu użytkownika:

$$\frac{}{\delta, t \to \tau \ \vdash \ \text{type } t = \tau : t \to \tau}$$

$$\frac{\delta \vdash \ \tau \text{ wf\_usertype}}{\delta \ \vdash \ \text{type } t = \tau : t \to \tau}$$

Aby typ został zdefiniowany, musi on spełniać następujące reguły:

1. $$\frac{}{\vdash \ \text{int wf\_usertype}}$$

2. $$\frac{}{\vdash \ \text{unit wf\_usertype}}$$

3. $$\frac{}{\vdash \ \text{void wf\_usertype}}$$

4. $$\frac{\delta \vdash \ \tau_1 \text{ wf\_usertype} \qquad \delta \vdash \ \tau_2 \text{ wf\_usertype}}{\delta \ \vdash \ \tau_1 * \tau_2 \text{ wf\_usertype}}$$

5. $$\frac{\delta \vdash \ \tau_1 \text{ wf\_usertype} \qquad \delta \vdash \ \tau_2 \text{ wf\_usertype}}{\delta \ \vdash \ \tau_1 + \tau_2 \text{ wf\_usertype}}$$

6. $$\frac{\delta \vdash \ \tau \text{ wf\_usertype}}{\delta \ \vdash \ \text{Ptr } \tau \text{ wf\_usertype}}$$

7. $$\frac{}{\delta, t \to \tau \ \vdash \ \text{Ptr } t \text{ wf\_usertype}}$$

Reguła numer 7 mówi o tym że rekurencyjne wystąpienia typów użytkownika mogą pojawiać się tylko pod wskaźnikiem. Relacja type różniłaby się od wf_usertype tylko w siódmej regule:

7. $$\frac{}{\delta, t \to \tau \ \vdash \ t \text{ type}}$$

Typy użytkownika muszą być brane pod uwagę przy porównywaniu typów. Dlatego typy będą porównywane wykorzystując najsilniejszą kongruencję zamkniętą na regułę:

$$\frac{}{\delta, t \to \tau \ \vdash \ t \equiv \tau}$$

Typy użytkownika mogą być rozwijane aby sprawdzić czy dane typy są równe.

2.2.1. Funkcje

Reguła typowania funkcji:

$$\frac{\delta; ; (x_i : \tau_i)_n, (y_i : \tau'_i)_{j-1} \vdash e_j : \tau'_j \qquad \delta; \Delta; (\Gamma = (x_i : \tau_i)_n, (y_i : \tau'_i)_m) \vdash c \text{ wf} \qquad \delta; ; \Gamma \vdash e : \tau_r}{\delta; \Delta \ \vdash \ f(x_1 : \tau_1, ..., x_n : \tau_n) : \tau_r \text{ in vars } (y_i := e_i)_m* \text{ in } c \text{ in return } e : \ (\tau_1, ..., \tau_n) \to \tau_r}$$

Definicje zmiennych mogą korzystać z argumentów i poprzednich zmiennych, instrukcja musi być poprawna i typ zwracanego wyrażenia musi się zgadzać z tym zadeklarowanym.

2.3. Wyrażenia

Reguły typowania wyrażeń:

1. $$\frac{}{\delta;;\Gamma, x:\tau \vdash\ x:\tau}$$

2. $$\frac{}{\vdash\ \underline{n} : \text{int}}$$

3. $$\frac{\delta;;\Gamma \vdash\ e_1 : \text{int} \qquad \delta;;\Gamma \vdash\ e_2 : \text{int}}{\delta;;\Gamma\ \vdash\ e_1 + e_2 : \text{int}}$$

4. $$\frac{\delta;;\Gamma \vdash\ e_1 : \text{int} \qquad \delta;;\Gamma \vdash\ e_2 : \text{int}}{\delta;;\Gamma\ \vdash\ e_1 * e_2 : \text{int}}$$

5. $$\frac{\delta;;\Gamma \vdash\ e : \text{int}}{\delta;;\Gamma\ \vdash\ -e : \text{int}}$$

6. $$\frac{}{\vdash <>\ : \text{unit}}$$

7. $$\frac{\delta;;\Gamma \vdash\ e_1 : \tau_1 \qquad \delta;;\Gamma \vdash\ e_2 : \tau_2}{\delta;;\Gamma\ \vdash\ < e_1, e_2 >: \tau_1\ *\ \tau_2}$$

8. $$\frac{\delta;;\Gamma\ \vdash\ e : \tau_1\ *\ \tau_2}{\delta;;\Gamma\ \vdash\ \text{outl}\ e : \tau_1}$$

9. $$\frac{\delta;;\Gamma\ \vdash\ e : \tau_1\ *\ \tau_2}{\delta;;\Gamma\ \vdash\ \text{outr}\ e : \tau_2}$$

10. $$\frac{\delta;;\Gamma\ \vdash\ e : \tau_1\ *\ \tau_2}{\delta;;\Gamma\ \vdash\ \text{outr}\ e : \tau_2}$$

11. $$\frac{\delta;;\Gamma\ \vdash\ e : \tau_1}{\delta;;\Gamma\ \vdash\ \text{inl}\{\tau_1\ +\ \tau_2\}(e) : \tau_1\ +\ \tau_2}$$

12. $$\frac{\delta;;\Gamma\ \vdash\ e : \tau_2}{\delta;;\Gamma\ \vdash\ \text{inr}\{\tau_1\ +\ \tau_2\}(e) : \tau_1\ +\ \tau_2}$$

13. $$\frac{\delta;;\Gamma \vdash\ e : \tau_1\ +\ \tau_2 \qquad \delta;;\Gamma, x:\tau_1 \vdash\ e_1 : \tau \qquad \delta;;\Gamma, x:\tau_2 \vdash\ e_2 : \tau}{\delta;;\Gamma\ \vdash\ \text{match}\ e\ \text{with}\ |\ \text{inl}\ x.e_1\ |\ \text{inr}\ x.e_2 : \tau}$$

14. $$\frac{\delta;;\Gamma\ \vdash\ e : \text{void}}{\delta;;\Gamma\ \vdash\ \text{abort}\{\tau\}(e) : \tau}$$

15. $$\frac{\delta;;\Gamma\ \vdash\ e : \text{Ptr}\ \tau}{\delta;;\Gamma\ \vdash\ *e : \tau}$$

2.4. Wstępne deklaracje zmiennych

$$\frac{\delta;;\Gamma\ \vdash\ e : \tau}{\delta;;\Gamma\ \vdash\ x := e \rightarrow x : \tau}$$

$$\frac{\delta;;\Gamma\ \vdash\ e : \tau}{\delta;;\Gamma\ \vdash\ x := \text{new}\ e \rightarrow x : \text{Ptr}\ \tau}$$

2.5. Instrukcje

Reguły poprawności instrukcji:

$$1.\quad \frac{}{\delta;\Delta;\Gamma \ \vdash\ \text{skip wf}}$$

$$2.\quad \frac{\delta;\Delta;\Gamma \vdash c_1 \text{ wf} \qquad \delta;\Delta;\Gamma \vdash c_2 \text{ wf}}{\delta;\Delta;\Gamma \ \vdash\ c_1; c_2 \text{ wf}}$$

$$3.\quad \frac{\delta;;\Gamma \vdash e : \text{int} \qquad \delta;\Delta;\Gamma \vdash c_1 \text{ wf} \qquad \delta;\Delta;\Gamma \vdash c_2 \text{ wf}}{\delta;\Delta;\Gamma \ \vdash\ \text{if } e \text{ then } c_1 \text{ else } c_2 \text{ wf}}$$

$$4.\quad \frac{\delta;;\Gamma \vdash e : \text{int} \qquad \delta;\Delta;\Gamma \vdash c \text{ wf}}{\delta;\Delta;\Gamma \ \vdash\ \text{while } e \text{ do } c \text{ wf}}$$

$$5.\quad \frac{\delta;;\Gamma, x : \tau \vdash e : \tau}{\delta;\Delta;\Gamma, x : \tau \ \vdash\ x := e \text{ wf}}$$

$$6.\quad \frac{\delta;;\Gamma, x : \text{Ptr } \tau \vdash e : \tau}{\delta;\Delta;\Gamma, x : \text{Ptr } \tau \ \vdash\ {*x} := e \text{ wf}}$$

$$7.\quad \frac{\delta;;\Gamma \vdash e : \tau \qquad \delta;\Delta;\Gamma, x : \tau \vdash c \text{ wf}}{\delta;\Delta;\Gamma \ \vdash\ \text{var } x := e \text{ in } c \text{ wf}}$$

$$8.\quad \frac{\delta;;\Gamma \vdash e : \tau \qquad \delta;\Delta;\Gamma, x : \text{Ptr } \tau \vdash c \text{ wf}}{\delta;\Delta;\Gamma \ \vdash\ \text{var } x := \text{new } e \text{ in } c \text{ wf}}$$

$$9.\quad \frac{(\delta;;\Gamma, x : \tau \vdash e_i : \tau_i)_n}{\delta;\Delta, f : (\tau_1, ..., \tau_n) \to \tau;\Gamma, x : \tau \ \vdash\ x := f(e_1, ..., e_n) \text{ wf}}$$

$$10.\quad \frac{\delta;;\Gamma \vdash e : \tau_1 + \tau_2 \qquad \delta;\Delta;\Gamma, x : \tau_1 \vdash c_1 \text{ wf} \qquad \delta;\Delta;\Gamma, x : \tau_2 \vdash c_2 \text{ wf}}{\delta;\Delta;\Gamma \ \vdash\ \text{switch } e \text{ inl x: } c_1 \text{ inr x: } c_2 \text{ wf}}$$

$$11.\quad \frac{}{\delta;\Delta;\Gamma \ \vdash\ \text{abort wf}}$$

$$12.\quad \frac{}{\delta;\Delta;\Gamma \ \vdash\ \text{debug\_print } s \text{ wf}}$$

2.6. Program

Reguła poprawności programu:

$$\frac{\begin{array}{ccc}\delta = (t_i \to \tau_{1,i})_k & \Delta = (f_i : (\tau_{2,i,1}, ..., \tau_{2,i,n}) \to \tau_{2,i,r})_m & \Gamma = \delta;;(x_i : \tau_i)_{h-1} \vdash d_{vh} \to x_h : \tau_h \\ \delta \ \vdash\ (d_{t,i} : t_i \to \tau_{1,i})_k & \delta;\Delta \ \vdash\ (d_{f,i} : (\tau_{2,i,1}, ..., \tau_{2,i,n}) \to \tau_{2,i,r})_m & \delta;\Delta;\Gamma \ \vdash\ c \text{ wf}\end{array}}{\vdash\ d_t{*} \text{ in } d_f{*} \text{ in vars } d_v{*} \text{ in } c}$$

3. Wykonanie

Reguły wykonywania instrukcji i wyliczania wyrażeń wymagają trzech kontekstów, wartości funkcji, zmiennych lokalnych oraz sterty. Kontekst funkcji nie będzie się zmieniał podczas wykonywania instrukcji. Kontekst zmiennych lokalnych będzie modyfikowany przez wprowadzanie i przypisanie zmiennych. Sterta jest globalnym kontekstem, nie zależy od tego w jakim bloku jesteśmy.

$$\Delta = (f \to ((x_i)_n, (y_i := e_i)_m, c, e))*$$

$$\Gamma = (x \to v)*$$

$$H = (p \to v)*$$

Wykonanie programu będzie zdefiniowane w semantyce wielkich kroków.

3.1. Forma osądów

Wyliczanie wyrażenia:

$$\Gamma; H \vdash e \Downarrow v$$

Do wyliczenia wyrażenia potrzebujemy wartości aktualnych zmiennych oraz sterty. Operacja 'match with' wymaga lokalnego rozszerzenia kontekstu. Sterta nie zmienia się w trakcie wyliczania wyrażenia.
Deklaracje zmiennych pojawiają się w programie i w deklaracjach funkcji. Na ich podstawie będą tworzone konteksty zmiennych i sterta.

$$\Gamma; H \;||\; (d_t)^* \Downarrow \Gamma; H$$

Wykonanie instrukcji:

$$\Delta \vdash \Gamma; H \;||\; c \Downarrow \Gamma; H$$

Instrukcje mogą dodawać oraz modyfikować zmienne i pola w stercie. Zbiór funkcji natomiast pozostaje niezmieniony.
Wykonanie programu:

$$\vdash p \Downarrow \Gamma; H$$

3.2. Wartość

Wyliczanie wyrażenia kończy się kiedy doliczymy się do wartości. Definicja wartości:

$$\frac{}{\Gamma; H \;\vdash\; \underline{n} \text{ val}}$$

$$\frac{}{\Gamma; H \;\vdash\; <> \text{ val}}$$

$$\frac{}{\Gamma; H, p \Downarrow \_ \;\vdash\; \text{Ptr } p \text{ val}}$$

$$\frac{\Gamma; H \;\vdash\; e_1 \text{ val} \qquad \Gamma; H \;\vdash\; e_2 \text{ val}}{\Gamma; H \;\vdash\; < e_1, e_2 > \text{val}}$$

$$\frac{\Gamma; H, \;\vdash\; e \text{ val}}{\Gamma; H, \;\vdash\; \text{inl } e \text{ val}}$$

$$\frac{\Gamma; H, \;\vdash\; e \text{ val}}{\Gamma; H, \;\vdash\; \text{inr } e \text{ val}}$$

3.3. Wyliczanie wartości wyrażeń

Reguły definiują jak wyrażenia są przeliczane do wartości. Ponieważ 'match with' lokalnie rozszerza kontekst zmiennych, musi on być także modyfikowalny przez obliczenie.

$$\frac{}{\Gamma, x \to v; H \vdash x \;\Downarrow\; v}$$

$$\frac{\Gamma; H \vdash e_1 \;\Downarrow\; v_1 \qquad \Gamma; H \vdash e_2 \;\Downarrow\; v_2}{\Gamma; H \vdash < e_1, e_2 > \Downarrow < v_1, v_2 >}$$

$$\frac{\Gamma; H \vdash e \;\Downarrow < v_1, v_2 >}{\Gamma; H \vdash \text{outl } e \;\Downarrow\; v_1}$$

$$\frac{\Gamma; H \vdash e \;\Downarrow < v_1, v_2 >}{\Gamma; H \vdash \text{outr } e \;\Downarrow\; v_2}$$

$$\frac{\Gamma; H \vdash e \;\Downarrow\; v}{\Gamma; H \vdash \text{inl } (e) \;\Downarrow\; \text{inl } v}$$

$$\frac{\Gamma; H \vdash e \ \Downarrow \ v}{\Gamma; H \vdash \text{inr } (e) \ \Downarrow \ \text{inr } v}$$

$$\frac{\Gamma; H \vdash e \ \Downarrow \ \text{inl } v \qquad \Gamma, x \to v; H \vdash e_1 \ \Downarrow \ v_1}{\Gamma; H \vdash \text{match } e \text{ with } | \text{ inl } x.e_1 \mid \text{inr } x.e_2 \ \Downarrow \ v_1}$$

$$\frac{\Gamma; H \vdash e \ \Downarrow \ \text{inr } v \qquad \Gamma, x \to v; H \vdash e_2 \ \Downarrow \ v_2}{\Gamma; H \vdash \text{match } e \text{ with } | \text{ inl } x.e_1 \mid \text{inr } x.e_2 \ \Downarrow \ v_2}$$

$$\frac{\Gamma; H, p \to v \vdash e \ \Downarrow \ \text{Ptr } p}{\Gamma; H, p \to v \vdash {*e} \ \Downarrow \ v}$$

$$\frac{\Gamma; H \vdash e_1 \ \Downarrow \ \underline{n} \qquad \Gamma; H \vdash e_2 \ \Downarrow \ \underline{m}}{\Gamma; H \vdash e_1 + e_2 \ \Downarrow \ \underline{n+m}}$$

$$\frac{\Gamma; H \vdash e_1 \ \Downarrow \ \underline{n} \qquad \Gamma; H \vdash e_2 \ \Downarrow \ \underline{m}}{\Gamma; H \vdash e_1 * e_2 \ \Downarrow \ \underline{n*m}}$$

$$\frac{\Gamma; H \vdash e \ \Downarrow \ \underline{n}}{\Gamma; H \vdash -e \ \Downarrow \ \underline{-n}}$$

3.4. Wstępne deklaracje zmiennych

$$\frac{}{\Gamma; H \mid\mid \epsilon \ \Downarrow \ \Gamma; H}$$

$$\frac{\Gamma; H \vdash e \ \Downarrow \ v}{\Gamma; H \mid\mid x := e \ \Downarrow \ \Gamma, x \to v; H}$$

$$\frac{\Gamma; H \vdash e \ \Downarrow \ v \qquad p \notin H}{\Gamma; H \mid\mid x := \text{ new } e \ \Downarrow \ \Gamma, x \to \text{Ptr } p; H, p \to v}$$

$$\frac{\Gamma; H \mid\mid d_t \ \Downarrow \ \Gamma'; H' \qquad \Gamma'; H' \mid\mid (d_t)* \ \Downarrow \ \Gamma''; H''}{\Gamma; H \mid\mid d_t \ (d_t)^* \ \Downarrow \ \Gamma''; H''}$$

3.5. Instrukcje

$$\frac{}{\Delta \vdash \Gamma; H \mid\mid \mathit{skip} \ \Downarrow \ \Gamma; H}$$

$$\frac{\Delta \vdash \Gamma; H \mid\mid c_1 \ \Downarrow \ \Gamma_1; H_1 \qquad \Delta \vdash \Gamma_1; H_1 \mid\mid c_2 \ \Downarrow \ \Gamma_2; H_2}{\Delta \vdash \Gamma; H \mid\mid c_1; c_2 \ \Downarrow \ \Gamma_2; H_2}$$

$$\frac{\Gamma; H \vdash e \ \Downarrow \ \underline{n} \neq 0 \qquad \Delta \vdash \Gamma; H \mid\mid c_1 \ \Downarrow \ \Gamma'; H'}{\Delta \vdash \Gamma; H \mid\mid \text{if } e \text{ then } c_1 \text{ else } c_2 \ \Downarrow \ \Gamma'; H'}$$

$$\frac{\Gamma; H \vdash e \ \Downarrow \ \underline{0} \qquad \Delta \vdash \Gamma; H \mid\mid c_2 \ \Downarrow \ \Gamma'; H'}{\Delta \vdash \Gamma; H \mid\mid \text{if } e \text{ then } c_1 \text{ else } c_2 \ \Downarrow \ \Gamma'; H'}$$

$$\frac{\Gamma; H \vdash e \ \Downarrow \ \underline{n} \neq 0 \qquad \Delta \vdash \Gamma; H \mid\mid c_1 \ \Downarrow \ \Gamma'; H'}{\Delta \vdash \Gamma; H \mid\mid \text{while } e \text{ do } c \ \Downarrow \ \Gamma'; H'}$$

$$\frac{\Gamma; H \vdash e \ \Downarrow \ \underline{0}}{\Delta \vdash \Gamma; H \mid\mid \text{while } e \text{ do } c \ \Downarrow \ \Gamma; H}$$

$$\frac{\Gamma; H \vdash e \ \Downarrow \ \underline{n} \neq 0 \qquad \Delta \vdash \Gamma; H \mid\mid c; \text{ while } e \text{ do } c \ \Downarrow \ \Gamma'; H'}{\Delta \vdash \Gamma; H \mid\mid \text{while } e \text{ do } c \ \Downarrow \ \Gamma'; H'}$$

$$\frac{\Gamma, x \to v; H \vdash e \ \Downarrow \ v'}{\Delta \vdash \Gamma, x \to v; H \ || \ x := e \ \Downarrow \ \Gamma, x \to v'; H}$$

$$\frac{\Gamma, x \to \text{Ptr } p; H, p \to v \vdash e \ \Downarrow \ v'}{\Delta \vdash \Gamma, x \to \text{Ptr } p; H, p \to v \ || \ *x := e \ \Downarrow \ \Gamma, x \to \text{Ptr } p; H, p \to v'}$$

$$\frac{\Gamma; H \vdash e \ \Downarrow \ v \qquad \Delta \vdash \Gamma, x \to v; H \ || \ c \ \Downarrow \ \Gamma', x \to \_; H'}{\Delta \vdash \Gamma; H \ || \ \text{var } x := e \text{ in } c \ \Downarrow \ \Gamma'; H'}$$

$$\frac{\Gamma; H \vdash e \ \Downarrow \ v \qquad p \notin H \qquad \Delta \vdash \Gamma, x \to \text{Ptr } p; H, p \to v \ || \ c \ \Downarrow \ \Gamma', x \to \_; H'}{\Delta \vdash \Gamma; H \ || \ \text{var } x := \text{new } e \text{ in } c \ \Downarrow \ \Gamma'; H'}$$

$$\frac{\begin{array}{c}(\Gamma, y \to v; H \vdash e_i \ \Downarrow \ v_i)_n \qquad \Gamma_{args} = (x_i \to v_i)_n \qquad \Gamma_{args}; H \ || \ (d_t)_m \ \Downarrow \ \Gamma_f, H_f \\ \Delta, f \to ((x_i)_n, (d_t)_m, c, e) \vdash \Gamma_f; H_f \ || \ c \ \Downarrow \ \Gamma'_f, ; H'_f \qquad\qquad \Gamma'_f; H'_f \vdash e \ \Downarrow \ v_r\end{array}}{\Delta, f \to ((x_i)_n, (d_t)_m, c, e) \vdash \Gamma, y \to v; H \ || \ y := f(e_1, ..., e_n) \ \Downarrow \ \Gamma, y \to v_r; H'_f}$$

$$\frac{\Gamma; H \vdash e \ \Downarrow \ \text{inl } v \qquad \Delta \vdash \Gamma, x \to v; H \ || \ c_1 \ \Downarrow \ \Gamma', x \to \_; H'}{\Delta \vdash \Gamma; H \ || \ \text{switch } e \text{ inl } x\text{: } c_1 \text{ inr } x\text{: } c_2 \ \Downarrow \ \Gamma'; H'}$$

$$\frac{\Gamma; H \vdash e \ \Downarrow \ \text{inr } v \qquad \Delta \vdash \Gamma, x \to v; H \ || \ c_2 \ \Downarrow \ \Gamma', x \to \_; H'}{\Delta \vdash \Gamma; H \ || \ \text{switch } e \text{ inl } x\text{: } c_1 \text{ inr } x\text{: } c_2 \ \Downarrow \ \Gamma'; H'}$$

W regule dotyczącej wywołania funkcji, najpierw wyliczamy argumenty które tworzą wstępny kontekst (Γ_{args}). Później iterujemy po deklaracjach zmiennych i tworzymy kontekst dla instrukcji funkcji (Γ_f i H_f). W tych deklaracjach mogą się także pojawiać dynamiczne alokacje, stąd zmiana w stercie. Na koniec wyliczamy zmianę kontekstów przez instrukcję (Γ'_f i H'_f) i w nich wyliczamy wartość zwracaną. Sterta nie jest przywracana do stanu z przed wywołania funkcji, natomiast kontekst zmiennych jest jedynie zmodyfikowany of wartość zwracaną.

3.6. Ewaluacja programu

$$\frac{\Delta = (d_f : f \to ((x_i)_n, (y_i := e_i)_m, c, e))^* \qquad || \ (d_t)^* \ \Downarrow \ \Gamma; H \qquad \Delta \vdash \Gamma; H \ || \ c \ \Downarrow \ \Gamma'; H'}{\vdash d_t^* \text{ in } d_f^* \text{ in vars } d_t^* \text{ in } c \ \Downarrow \ \Gamma'; H'}$$

Tworzymy kontekst funkcji i wyznaczamy początkowy kontekst zmiennych i stertę na podstawie deklaracji zmiennych. Z nimi poprostu wykonujemy instrukcję programu.

4. Implementacja

Ewaluator został zaimplementowany w języku OCaml 4.0.7. Do parsowania użyłem biblioteki Menhir. Po parsowaniu kolejnym etapem jest oczywiście sprawdzenie typów i poprawności w programie. Później program zostaje przeliczony żeby otrzymać ostateczne wartości zmiennych i sterty.

4.1. Moduł typechecker

4.1.1. Sprawdzenie typów użytkownika

Najpierw dla każdego zadeklarowanego typu uruchamiamy funkcję *check_user_type*, która sprawdza czy typ należy do relacji wf_usertype. Konstruktor obiektów typu *typ* wymusza należenie do relacji *type*.

```
let check_user_type declared_types t =
        let rec check is_ptr t = match t with
                | TStruct id ->
                    if is_ptr then
                            (if List.mem id declared_types then () else
                                    failwith ("type " ^ id ^ " is never declared"))
                    else failwith ("type " ^ id ^ " is not pointer type")
                | TPtr t -> check true t
                | TSum (t1, t2) -> (check false t1); (check false t2)
                | TProd (t1, t2) -> (check false t1); (check false t2)
                | _ -> ()
        in
        try check false t
        with Failure err ->  failwith ("invalid user type: " ^ (pretty_typ t) ^ " cause: " ^ err)
```

Funkcja ta przyjmuje listę wszystkich zdefiniowanych typów oraz definicję jednego z nich do sprawdzenia. Porównywanie typów jest trochę bardziej skomplikowane niż standardowe strukturalne które byłoby wywołane przez ocamla. Aby porównywać typy, wpierw należy umieć rozwijać typy użytkownika. Tym zajmuje się funkcja *extract_ type*, która rekurencyjnie rozwija zewnętrzny typ użytkownika, aż dojdzie do typu standardowego, np. jeśli b = int + unit, to extract_type b = int + unit. Przydaje się ona także w wyrażeniach "match t with" kiedy chcemy sprawdzić czy typ t ma określoną strukturę, która może być pod typem użytkownika.

```
let rec extract_type type_env t = match t with
        | TStruct id -> extract_type type_env (Env.find id type_env)
        | t -> t
```

Wtedy możemy zdefiniować funkcję do porównywania typów, która potrafi rozwijać napotkane typy użytkownika.

```
let eq_type type_env t1 t2 =
        let extract = extract_type type_env in
        let rec aux a b =
                if a = b then true else
                let b = extract b in
                match extract a with
                | TSum (a1, a2) -> (match b with
                        | TSum (b1, b2) -> (aux a1 b1) && (aux a2 b2)
                        | _ -> false)
                | TProd (a1, a2) -> (match b with
                        | TProd (b1, b2) -> (aux a1 b1) && (aux a2 b2)
                        | _ -> false)
                | TPtr a' -> (match b with
                        | TPtr b' -> aux a' b'
                        | _ -> false)
                | trivial -> b = trivial
        in
        aux t1 t2
```

Mając takie narzędzia, można zdefiniować dedukcję typów wyrażeń i poprawności instrukcji wprost korzystając z reguł przedstawionych wcześniej. Najpierw tworzymy konteksty dla typów i funkcji, nie zmienią się one podczas sprawdzania programu. Wszystkie konteksty są przetrzymywane w strukturze typu Map. Z tymi kontekstami najpierw sprawdzamy poprawność wszystkich zdefiniowanych funkcji.
Aby sprawdzić deklarację funkcji najpierw tworzymy kontekst z typami zmiennych z argumentów i wstępnych deklaracji, następnie sprawdzamy poprawność instrukcji, i na końcu sprawdzamy czy typ wartości zwracanej zgadza się z zadeklarowanym.

```
let check_function_declaration (id, args, r_type, vars, c, e) =
        let arg_types = Env.of_seq (List.to_seq args) in
        let var_env = deduce_vars_types type_env arg_types vars in
        check_command type_env function_env var_env c;
        let actual_r_type = deduce_expr_type type_env var_env e in
        if not (eq_type type_env actual_r_type r_type) then
                failwith ("invalid return type of function " ^ id ^
                                ". Is " ^ (pretty_typ actual_r_type) ^
                                " and should be " ^ (pretty_typ r_type) ^ ".")
```

Możemy także zdefiniować funkcję sprawdzającą poprawność instrukcji, przyjmuje ona dodatkowo środowisko typów zmiennych.

```
let check_command type_env function_env var_env c =
        let rec check var_env c =
        match c with
                | CSkip -> ()
                | CSeq (c1, c2) -> check var_env c1; check var_env c2;
                | CIf (e, c_true, c_false) ->
                        if extract_type type_env (deduce_expr_type type_env var_env e) = TInt then
                                (check var_env c_true; check var_env c_false)
                        else failwith ("if condition" ^ (pretty_expr e) ^ " is not an int.")
                | CWhile (e, c) ->
                        if extract_type type_env (deduce_expr_type type_env var_env e) = TInt
                        then check var_env c
                        else failwith ("while condition" ^ (pretty_expr e) ^ " is not an int.")
        ...
        check var_env c
```

Funkcja ta, kiedy znajdzie błąd, rzuca wyjątek ze źródłem błędu. Porównajmy implementację sprawdzenia wołania funkcji z jej regułą:

$$\frac{(\delta;;\Gamma, x:\tau \vdash e_i : \tau_i)_n}{\delta;\Delta, f:(\tau_1,...,\tau_n)\to\tau;\Gamma, x:\tau \ \vdash \ x := f(e_1,...,e_n) \text{ wf}}$$

```
| CCall (var_id, fun_id, args) ->
 if not (Env.mem var_id var_env) then
         failwith ("Variable " ^ var_id ^ " is not defined at " ^ (pretty_command c))
 else
 if not (Env.mem fun_id function_env) then
         failwith ("Function " ^ fun_id ^ " is not defined at " ^ (pretty_command c))
 else
 let var_type = Env.find var_id var_env in
 let fun_type = Env.find fun_id function_env in
 let return_type = snd fun_type in
 if not (eq_type type_env return_type var_type) then
         failwith ("Function return type and variable type do not match at " ^ (pretty_command c) ^
                 " variable type: " ^ (pretty_typ var_type) ^
                 " function return type: " ^ (pretty_typ return_type))
 else
 let args_expected_types = List.map snd (fst fun_type) in
 let args_actual_types = List.map (deduce_expr_type type_env var_env) args in
 if not (List.for_all2 (eq_type type_env) args_expected_types args_actual_types) then
         failwith ("Invalid argument types at " ^ (pretty_command c) ^
                 "expected: " ^ (String.concat ", " (List.map pretty_typ args_expected_types)) ^
                 " actual: " ^ (String.concat ", " (List.map pretty_typ args_actual_types)))
```

4.2. Ewaluacja

Zauważmy, że środowisko funkcji w trakcie ewaluacji się nie zmienia. Do tego żadne zmiany na stercie nie są cofane, nawet te które powstały podczas wołania funkcji. Z tego powodu, oraz dla uproszczenia kodu, środowisko funkcji i sterta są zmiennymi globalnymi które są czyszczone po ewaluacji każdego programu.

```
let heap = Heap.create 100
let ref_count = Heap.create 100
let fun_env = ref Env.empty
```

Aby znaleźć pola na stercie które można dealokować, zdecydowałem się na zliczanie referencji do nich. Oto funkcje które pomagają w korzystaniu ze sterty oraz zliczaniu referencji:

```
let alloc = (* Alokacja nowej wartości w świeżej komórce na stercie. *)
        let max_key = ref (-1) in
        let aux v = let id = max_key.contents + 1 in
                Heap.add heap id v;
                Heap.add ref_count id 1;
                max_key := id;
                VPtr id
        in
        aux

(* Odczyt pola na stercie podanego przez wstaźnik. Pole musi istnieć. *)
let read_heap_at id = match Heap.find_opt heap id with
        | Some v -> v
        | None -> failwith "Invalid pointer dereference."

(* Zapis pola na stercie podanego przez wstaźnik. Pole musi istnieć. *)
let write_heap_at id v = match Heap.find_opt heap id with
        | Some _ -> Heap.replace heap id v
        | None -> failwith "Invalid pointer dereference."

(* Tworzy listę wszystkich wskaźników które znajdują się w podanej wartości. *)
let rec get_references v = match v with
        | VPtr p -> [p]
        | VInjL v' -> get_references v'
        | VInjR v' -> get_references v'
        | VPar (v1, v2) -> List.rev_append (get_references v1) (get_references v2)
        | _ -> []

(*
Kasuje referencje znajdujące się w podanej wartości. Jeżeli licznik spadnie do zera, to dane
pole jest usuwane, jednak jego referencje też trzeba usunąć. Dzięki temu można np. usunąć listę,
usuwając jej głowę (jeżeli nie ma innych odwołań do niej).
*)
let rec drop_references v =
        let drop_reference p = match Heap.find_opt ref_count p with
                | Some c -> if c = 1 then (
                                        drop_references (Heap.find heap p);
                                        Heap.remove heap p;
                                        Heap.remove ref_count p
                                ) else
                                        Heap.replace ref_count p (c - 1)
                | None -> failwith ("Can't drop reference &" ^ (string_of_int p) ^
                                ". No such key in heap and ref_count.")
        in
        let refs = get_references v in
        List.iter drop_reference refs
```

```
(* Dodaje referencje z podanej wartości. *)
let add_references v =
        let add_reference p = match Heap.find_opt ref_count p with
                | Some c -> Heap.replace ref_count p (c + 1)
                | None -> failwith ("Can't add reference &" ^ (string_of_int p) ^
                                    ". No such key in heap and ref_count.")
        in
        let refs = get_references v in
        List.iter add_reference refs
```

Ewaluacja wyrażeń nie musi się zajmować liczeniem referencji, ponieważ nie dodaje ani usuwa żadnych wartości. Stąd reguły ewaluacji są zaimplementowane wprost. Jedynie sterta nie jest zwracana przez tą funkcję, ponieważ jest ona globalna.

```
let rec eval_expr var_env e =
        let rec eval e = match e with
                | EVar id -> Env.find id var_env
                | EInt n -> VInt n
                | EUnit -> VUnit
                | EPar (e1, e2) -> VPar (eval e1, eval e2)
                | EInjL (t, e) -> VInjL (eval e)
                | EInjR (t, e) -> VInjR (eval e)
                | EMatch (e, (id_left, e_left), (id_right, e_right)) -> (match eval e with
                        | VInjL v -> eval_expr (Env.add id_left v var_env) e_left
                        | VInjR v -> eval_expr (Env.add id_right v var_env) e_right
                        | _ -> failwith ("Stuck at match. " ^ (pretty_expr e) ^
                                         " doesn't evaluate to sum."))
                | EDeref e -> (match eval e with
                        | VPtr p -> read_heap_at p
                        | _ -> failwith ("Stuck at dereference. " ^ (pretty_expr e) ^
                                         " doesn't evaluate to pointer."))
                | ESum (e1, e2) -> (match (eval e1, eval e2) with
                        | (VInt c1, VInt c2) -> VInt (c1 + c2)
                        | _ -> failwith ("Stuck at sum. " ^ (pretty_expr e) ^
                                         " doesn't evaluate to int."))
        ...
```

Bardziej skomplikowana jest ewaluacja funkcji i instrukcji. Funkcja do ewaluowania funkcji przyjmuje wartości argumentów i zwraca tylko wynik zwracany, choć przy okazji może modyfikować stertę.

```
let rec eval_function_call f_id args_v =
        let (args_ids, vars, c, e) = Env.find f_id fun_env.contents in
        (* Stwórz środowisko z argumentów. *)
        let args_env = Env.of_seq (List.to_seq (List.combine args_ids args_v)) in
        Env.iter (fun _ v -> add_references v) args_env;
        (* Dodaj do środowiska wstępne deklaracje zmiennych. *)
        let var_env = eval_vars args_env vars in
        (* Wykonaj instrukcje i wylicz wartość zwracaną. *)
        let var_env' = eval_command var_env c in
        let return_value = eval_expr var_env' e in
        (* Usuń wszystkie referencje utworzone przez wywołanie funkcji, oprócz wartośći zwracanej. *)
        add_references return_value;
        Seq.iter (fun (_, v) -> drop_references v)(Env.to_seq var_env');
        return_value
```

Powyższa funkcja jest zdefiniowana razem z funkcją wykonującą wyrażenia.

```
and eval_command var_env c = match c with
        | CSkip -> var_env
        | CSeq (c1, c2) -> let var_env' = eval_command var_env c1 in
                           eval_command var_env' c2
        | CIf (e, c_true, c_false) -> (match eval_expr var_env e with
               | VInt 0 -> eval_command var_env c_false
               | VInt n -> eval_command var_env c_true
               | _ -> failwith ("If condition" ^ (pretty_expr e) ^ " did not evaluate to int."))
        | CWhile (e, c') -> (match eval_expr var_env e with
               | VInt 0 -> var_env
               | VInt n -> eval_command var_env (CSeq (c', c))
               | _ -> failwith ("While condition" ^ (pretty_expr e) ^ " did not evaluate to int."))
        | CAssign (id, e) ->
               let new_v = eval_expr var_env e in
               let old_v = Env.find id var_env in
               add_references new_v;
               drop_references old_v;
               Env.add id new_v var_env
        ...
        | CPtrVar (id, e, c) ->
               let v = eval_expr var_env e in
               let alloc_ptr = alloc v in
               add_references v;
               let local_var_env = Env.add id alloc_ptr var_env in
               let var_env' = eval_command local_var_env c in
               let v_to_drop = Env.find id var_env' in
               drop_references v_to_drop;
               let fixed_var_env = Env.remove id var_env' in
               fixed_var_env
        | CCall (var_id, f_id, args) -> let args_v = List.map (eval_expr var_env) args in
               let call_result = eval_function_call f_id args_v in
               let old_v = Env.find var_id var_env in
               drop_references old_v;
               Env.add var_id call_result var_env
```

5. Kompilacja, uruchamianie i testowanie

Do zbudowania należy użyć komendy:

```
ocamlbuild -use-menhir -use-ocamlfind -tag thread -pkg core main.native
```

Plik example_list.wpp zawiera przykładową implementację listy oraz mały test dealokacji pamięci. Ewaluator uruchamiamy podając mu nazwę pliku z programem w W++:

```
./main.native example_list.wpp
```

Aby przeprowadzić testy z katalogu tests należy uruchomić skrypt test.py za pomocą pythona 3.

```
python3 test.py main.native tests
```

Są to zarówno testy typecheckera jak i ewaluatora.