Sémantika programovacích jazyků

Syntaxe a sémantika jednoduchého imperativního jazyka

Pokud chceme formálně dokazovat vlastnosti programů napsaných v nějakém programovacím jazyce, je třeba nejprve popsat jeho:

- **syntaxi** — jak vypadají dobře utvořené programy v tomto jazyce
- **sémantiku** — jak probíhá vykonávaní těchto programů

Tyto pojmy si budeme ilustrovat na příkladu jednoduchého imperativního programovacího jazyka nazvaného **Imp**.

Popíšeme si:

- jak syntaxi a sémantiku definovat formálně matematicky
- jak tyto formální definice reprezentovat v Rocqu

Syntaxe a sémantika jednoduchého imperativního jazyka

Jazyk Imp:

- obsahuje jen základní programové konstrukce (přiřazení, sekvenci, příkaz if a cyklus while)
- celý program je tvořen jedním složeným příkazem (tj. nemá funkce, procedury, ani nic podobného)
- proměnné mohou obsahovat jen jeden typ dat — přirozená čísla

Aritmetické výrazy

Než budeme popisovat syntaxi a sémantiku programů v jazyce Imp, budeme ilustrovat tyto pojmy nejprve na **aritmetických výrazech**:

- nejprve budeme uvažovat pro jednoduchost výrazy, které **neobsahují proměnné**

Abstraktní syntaxi těchto výrazů můžeme popsat následujícím způsobem:

$$\begin{array}{rcl} a & ::= & n \\ & | & a + a \\ & | & a - a \\ & | & a * a \end{array}$$

Poznámka: n zde reprezentuje libovolné přirozené číslo

Abstraktní syntaxe

- Abstraktní syntaxe popisuje, jak vypadají **abstraktní syntaktické stromy** reprezentující daný druh výrazů.
- Jednotlivé položky odpovídají jednotlivým druhům vrcholů abstraktního syntaktického stromu a informacím, které jsou s daným druhem vrcholu spojeny.
- Abstraktní syntaxe neřeší detaily zápisu daných výrazů — věci jako priority operátorů apod.

Abstraktní syntaxe

V Rocqu bude příslušná abstraktní syntaxe reprezentována takto:

```
Inductive aexp : Type :=
  | ANum (n : nat)
  | APlus (a1 a2 : aexp)
  | AMinus (a1 a2 : aexp)
  | AMult (a1 a2 : aexp).
```

Poznámka: Pro usnadnění zápisů takto definovaných výrazů je možné následně definovat příslušné uživatelsky definované notace a koerce.

Implicitní koerce (coercions)

Koerce (**coercions**) jsou uživatelsky definovaná přetypování.

Definice toho, že nějaká dříve definovaná funkce f bude používána jako **implicitní koerce** z typu A na typ B:

Coercion $f : A >\!\!\to B$.

Tento zápis znamená, že:

- Pokud se na místě, kde je očekávána hodnota typu B, objeví term t typu A, bude automaticky implicitně přidáno volání funkce f, tj. na daném místě se bude ve skutečnosti nacházet místo termu t term $(f\ t)$.
- V menu je možné zapnout a vypnout, jestli se mají tato implicitně přidaná volání funkce f zobrazovat nebo ne.
  (*View* / *Display coercions*)

Implicitní koerce (coercions)

Příklad:

```
Coercion ANum : nat >-> aexp.
```

Poznámka: Jako koerce z typu A na typ B může být používán jakýkoli term typu $A \to B$, speciálně tedy i konstruktory induktivně definovaných typů.

Seznam všech aktuálně definovaných koercí je možné vypsat následujícím příkazem:

```
Print Graph.
```

Poznámka: Koerce se spolu mohou skládat.

Aritmetické výrazy — sémantika

Sémantiku výrazů je možné definovat dvěma způsoby:

- jako **funkci** — například typu $\mathrm{aexp} \to \mathrm{nat}$
- jako **relaci** — například typu $\mathrm{aexp} \to \mathrm{nat} \to \mathbf{Prop}$

Výhody definice jako funkce:

- je možné ji přímo volat
- máme přímo dané, že pro daný výraz existuje právě jedna hodnota, na kterou se vyhodnotí

Nevýhody definice jako funkce:

- může být komplikovanější reprezentovat případy, kdy pro některé výrazy není hodnota definována
  (např. výrazy obsahující dělení, kde se vyskytuje dělení nulou)
- není možné zachytit nedeterminismus, kdy se může výraz vyhodnotit na více různých hodnot

Aritmetické výrazy — sémantika

Definice jako relace:

Například induktivně definované relace

$$\mathrm{aeval} : \mathrm{aexp} \to \mathrm{nat} \to \mathbf{Prop}$$

Výraz

$$\mathrm{aeval}\ a\ n$$

označuje, že aritmetický výraz a se může vyhodnotit na hodnotu n.

V matematickém zápise to můžeme reprezentovat například následující notací:

$$a \Rightarrow n$$

Aritmetické výrazy — sémantika

Daná relace je pak reprezentována následující sadou pravidel:

$$\frac{}{n \Rightarrow n}$$

$$\frac{a_1 \Rightarrow n_1 \qquad a_2 \Rightarrow n_2}{a_1 \texttt{ + } a_2 \Rightarrow n} \quad \text{kde } n = n_1 + n_2$$

$$\frac{a_1 \Rightarrow n_1 \qquad a_2 \Rightarrow n_2}{a_1 \texttt{ - } a_2 \Rightarrow n} \quad \text{kde } n = n_1 - n_2$$

$$\frac{a_1 \Rightarrow n_1 \qquad a_2 \Rightarrow n_2}{a_1 \texttt{ * } a_2 \Rightarrow n} \quad \text{kde } n = n_1 * n_2$$

Aritmetické výrazy — sémantika

Zajímavější je případ, kdy výrazy mohou obsahovat **proměnné**.

Abstraktní syntaxe aritmetických výrazů s proměnnými:

$$
\begin{array}{rcll}
a & ::= & n & \\
 & | & x & \longleftarrow \text{ nový případ} \\
 & | & a + a & \\
 & | & a - a & \\
 & | & a * a &
\end{array}
$$

Řekněme, že proměnné budou reprezentovány jako hodnoty nějak definovaného typu var.

Poznámka: Pro implementaci typu var můžeme použít například řetězce nebo přirozená čísla.

Aritmetické výrazy — sémantika

Výraz s proměnnými není možné vyhodnotit sám o sobě.

Výraz s proměnnými se vyhodnocuje vždy vzhledem k nějakému **stavu** σ, který určuje, jaké hodnoty jsou přiřazeny jakým proměnným.

Jednou možností, jak definovat takový stav σ, je definovat ho jako funkci typu

$$\text{var} \longrightarrow \text{nat}$$

- V tomto případě je v daném stavu definována hodnota každé proměnné.
- Alternativně je možné stav definovat jako **parciální** funkci, která určuje hodnoty jen některých proměnných.

Aritmetické výrazy — sémantika

Dvě nejdůležitější operace, které potřebujeme provádět se stavy:

- $\sigma(x)$ — zjištění hodnoty proměnné x ve stavu σ
- $\sigma[x \mapsto n]$ — vytvoří nový stav σ', který se se stavem σ shoduje na všech proměnných kromě x, kde x nabývá v σ' hodnoty n:

$$\sigma'(y) = \begin{cases} n & \text{pokud } y = x \\ \sigma(y) & \text{jinak} \end{cases}$$

Poznámka: Kromě toho musíme mít možnost, jak vytvořit nějaký počáteční stav σ_0.

Aritmetické výrazy — sémantika

Sémantiku pak můžeme definovat jako funkci nebo relaci:

Například induktivně definované relace

$$\text{aeval} : \text{state} \longrightarrow \text{aexp} \longrightarrow \text{nat} \longrightarrow \textbf{Prop}$$

Výraz

$$\text{aeval}\ st\ a\ n$$

označuje, že aritmetický výraz a se může ve stavu st vyhodnotit na hodnotu n.

V matematickém zápise to můžeme reprezentovat například následující notací:

$$\sigma \vdash a \Rightarrow n$$

Aritmetické výrazy — sémantika

$$\frac{}{\sigma \vdash n \Rightarrow n} \qquad \frac{\sigma(x) = n}{\sigma \vdash x \Rightarrow n}$$

$$\frac{\sigma \vdash a_1 \Rightarrow n_1 \qquad \sigma \vdash a_2 \Rightarrow n_2}{\sigma \vdash a_1 \texttt{ + } a_2 \Rightarrow n} \quad \text{kde } n = n_1 + n_2$$

$$\frac{\sigma \vdash a_1 \Rightarrow n_1 \qquad \sigma \vdash a_2 \Rightarrow n_2}{\sigma \vdash a_1 \texttt{ - } a_2 \Rightarrow n} \quad \text{kde } n = n_1 - n_2$$

$$\frac{\sigma \vdash a_1 \Rightarrow n_1 \qquad \sigma \vdash a_2 \Rightarrow n_2}{\sigma \vdash a_1 \texttt{ * } a_2 \Rightarrow n} \quad \text{kde } n = n_1 * n_2$$

Příkazy — syntaxe

Podobně, jako jsme zavedli syntaxi aritmetických výrazů a, můžeme zavést i syntaxi:

- booleovských výrazů b
- příkazů jazyka c

$$
\begin{array}{rcl}
c & ::= & \texttt{skip} \\
& | & x := a \\
& | & c \texttt{ ; } c \\
& | & \texttt{if } b \texttt{ then } c \texttt{ else } c \\
& | & \texttt{while } b \texttt{ do } c
\end{array}
$$

Příkazy — sémantika

Jednou možností, jak definovat sémantiku příkazů, je definovat tzv. **big-step** sémantiku:

- Definovat relaci stanovující, že provedením příkazu c je možné přejít ze stavu σ_1 do stavu σ_2.

$$\sigma_1 \rightrightarrows c \rightrightarrows \sigma_2$$

Příkazy — sémantika (big-step)

$$\sigma \preceq \texttt{skip} \succeq \sigma$$

$$\frac{\sigma \vdash a \Rightarrow n}{\sigma \preceq x := a \succeq \sigma'} \quad \text{kde } \sigma' = \sigma[x \mapsto n]$$

$$\frac{\sigma \preceq c_1 \succeq \sigma' \qquad \sigma' \preceq c_2 \succeq \sigma''}{\sigma \preceq c_1 \,;\, c_2 \succeq \sigma''}$$

$$\frac{\sigma \vdash b \Rightarrow \text{true} \qquad \sigma \preceq c_1 \succeq \sigma'}{\sigma \preceq \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \succeq \sigma'}$$

$$\frac{\sigma \vdash b \Rightarrow \texttt{false} \qquad \sigma \preceq c_2 \succeq \sigma'}{\sigma \preceq \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \succeq \sigma'}$$

$$\frac{\sigma \vdash b \Rightarrow \texttt{false}}{\sigma \preceq \texttt{while } b \texttt{ do } c \succeq \sigma}$$

$$\frac{\sigma \vdash b \Rightarrow \text{true} \qquad \sigma \preceq c \succeq \sigma' \qquad \sigma' \preceq \texttt{while } b \texttt{ do } c \succeq \sigma''}{\sigma \preceq \texttt{while } b \texttt{ do } c \succeq \sigma''}$$

Příkazy — sémantika (small-step)

Alternativně je možné definovat sémantiku příkazů jako tzv. **small-step** sémantiku:

- Celkový stav běžícího programu je dán jako dvojice

$$(\llbracket c \rrbracket, \sigma)$$

kde c je příkaz, který zbývá provést, a σ stanovuje aktuální hodnoty proměnných.

- Definujeme relaci, která stanovuje, kdy může běžící program přejít jedním krokem z jednoho globálního stavu do druhého.

$$(\llbracket c \rrbracket, \sigma) \longmapsto (\llbracket c' \rrbracket, \sigma')$$

Příkazy — sémantika (small-step)

- Výpočet programu je dán jako posloupnost těchto jednotlivých kroků:

$$(\llbracket c_0 \rrbracket, \sigma_0) \longmapsto (\llbracket c_1 \rrbracket, \sigma_1) \longmapsto (\llbracket c_2 \rrbracket, \sigma_2) \longmapsto (\llbracket c_3 \rrbracket, \sigma_3) \longmapsto \dots$$

- Celkově výpočet začne ve stavu

$$(\llbracket c \rrbracket, \sigma_0)$$

kde σ_0 reprezentuje počáteční hodnoty proměnných a c reprezentuje celý program.

- Výpočet programu se zastaví v koncovém stavu tvaru

$$(\llbracket \texttt{skip} \rrbracket, \sigma)$$

Příkazy — sémantika (small-step — ver. 1)

$$\frac{\sigma \vdash a \Rightarrow n}{(\llbracket x := a \rrbracket, \sigma) \longmapsto (\llbracket \texttt{skip} \rrbracket, \sigma')} \quad \text{kde } \sigma' = \sigma[x \mapsto n]$$

$$(\llbracket \texttt{skip} \,;\, c_2 \rrbracket, \sigma) \longmapsto (\llbracket c_2 \rrbracket, \sigma)$$

$$\frac{(\llbracket c_1 \rrbracket, \sigma) \longmapsto (\llbracket c_1' \rrbracket, \sigma')}{(\llbracket c_1 \,;\, c_2 \rrbracket, \sigma) \longmapsto (\llbracket c_1' \,;\, c_2 \rrbracket, \sigma')}$$

$$\frac{\sigma \vdash b \Rightarrow \text{true}}{(\llbracket \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \rrbracket, \sigma) \longmapsto (\llbracket c_1 \rrbracket, \sigma)}$$

$$\frac{\sigma \vdash b \Rightarrow \texttt{false}}{(\llbracket \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \rrbracket, \sigma) \longmapsto (\llbracket c_2 \rrbracket, \sigma)}$$

$$(\llbracket \texttt{while } b \texttt{ do } c_1 \rrbracket, \sigma) \longmapsto$$
$$(\llbracket \texttt{if } b \texttt{ then } (c_1 \,;\, \texttt{while } b \texttt{ do } c_1) \texttt{ else skip} \rrbracket, \sigma)$$

Příkazy — sémantika (small-step — ver. 1)

Alternativně je možné mít místo jednoho pravidla pro `while`:

$$(\llbracket \texttt{while}\ b\ \texttt{do}\ c_1 \rrbracket, \sigma) \longmapsto (\llbracket \texttt{if}\ b\ \texttt{then}\ (c_1\ ;\ \texttt{while}\ b\ \texttt{do}\ c_1)\ \texttt{else}\ \texttt{skip} \rrbracket, \sigma)$$

dvě samostatná pravidla:

$$\frac{\sigma \vdash b \Rightarrow \text{true}}{(\llbracket \texttt{while}\ b\ \texttt{do}\ c_1 \rrbracket, \sigma) \longmapsto (\llbracket c_1\ ;\ \texttt{while}\ b\ \texttt{do}\ c_1 \rrbracket, \sigma)}$$

$$\frac{\sigma \vdash b \Rightarrow \texttt{false}}{(\llbracket \texttt{while}\ b\ \texttt{do}\ c_1 \rrbracket, \sigma) \longmapsto (\llbracket \texttt{skip} \rrbracket, \sigma)}$$

Příkazy — sémantika (small-step — ver. 2)

Uvedená verze small-step sémantiky se při zdůvodňování toho, že je možné provést krok nějakým složeným příkazem tvaru např. $[\![c_1 \; ; \; c_2]\!]$ odvolává na to, jaký krok je možné provést dílčím příkazem $[\![c_1]\!]$.

Existuje alternativní možnost, jak definovat small-step sémantiku takových příkazů bez toho, aby bylo nutné se odvolávat na chování programu na podpříkazech:

- V jednotlivých krocích se transformuje tvar příkazu, který zbývá provést, např. se provádí krok, kdy se změní

  $$[\![(c_1 \; ; \; c_2) \; ; \; c]\!] \qquad \text{na} \qquad [\![c_1 \; ; \; (c_2 \; ; \; c)]\!]$$

- V této variantě jsou typicky příkazy v průběhu výpočtu ve tvaru

  $$[\![c_1 \; ; \; c]\!]$$

  a aktuální činnost v daném kroku závisí na tvaru příkazu c_1.

Příkazy — sémantika (small-step — ver. 2)

- V této variantě small-step sémantiky začne výpočet ve stavu

$$(\llbracket\, c\,;\ \texttt{skip}\,\rrbracket, \sigma_0)$$

kde σ_0 reprezentuje počáteční hodnoty proměnných a c reprezentuje celý původní program.

- Výpočet programu se zastaví v koncovém stavu tvaru

$$(\llbracket\, \texttt{skip}\,\rrbracket, \sigma)$$

Příkazy — sémantika (small-step — ver. 2)

$$\frac{\sigma \vdash a \Rightarrow n}{(\llbracket (x := a)\,;\, c \rrbracket, \sigma) \longmapsto (\llbracket c \rrbracket, \sigma')} \quad \text{kde } \sigma' = \sigma[x \mapsto n]$$

$$(\llbracket \texttt{skip}\,;\, c \rrbracket, \sigma) \longmapsto (\llbracket c \rrbracket, \sigma)$$

$$(\llbracket (c_1\,;\, c_2)\,;\, c \rrbracket, \sigma) \longmapsto (\llbracket c_1\,;\, (c_2\,;\, c) \rrbracket, \sigma)$$

$$\frac{\sigma \vdash b \Rightarrow \text{true}}{(\llbracket (\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2)\,;\, c \rrbracket, \sigma) \longmapsto (\llbracket c_1\,;\, c \rrbracket, \sigma)}$$

$$\frac{\sigma \vdash b \Rightarrow \texttt{false}}{(\llbracket (\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2)\,;\, c \rrbracket, \sigma) \longmapsto (\llbracket c_2\,;\, c \rrbracket, \sigma)}$$

$$(\llbracket (\texttt{while } b \texttt{ do } c_1)\,;\, c \rrbracket, \sigma) \longmapsto$$
$$(\llbracket (\texttt{if } b \texttt{ then } (c_1\,;\, \texttt{while } b \texttt{ do } c_1) \texttt{ else skip})\,;\, c \rrbracket, \sigma)$$

Příkazy — sémantika (small-step — ver. 2)

Alternativně je možné mít místo jednoho pravidla pro `while`:

$$(\llbracket\,(\texttt{while}\ b\ \texttt{do}\ c_1)\ ;\ c\,\rrbracket,\ \sigma)\ \longmapsto$$
$$(\llbracket\,(\texttt{if}\ b\ \texttt{then}\ (c_1\ ;\ \texttt{while}\ b\ \texttt{do}\ c_1)\ \texttt{else}\ \texttt{skip})\ ;\ c\,\rrbracket,\ \sigma)$$

dvě samostatná pravidla:

$$\frac{\sigma \vdash b \Rightarrow \text{true}}{(\llbracket\,(\texttt{while}\ b\ \texttt{do}\ c_1)\ ;\ c\,\rrbracket,\ \sigma)\ \longmapsto\ (\llbracket\,c_1\ ;\ ((\texttt{while}\ b\ \texttt{do}\ c_1)\ ;\ c)\,\rrbracket,\ \sigma)}$$

$$\frac{\sigma \vdash b \Rightarrow \texttt{false}}{(\llbracket\,(\texttt{while}\ b\ \texttt{do}\ c_1)\ ;\ c\,\rrbracket,\ \sigma)\ \longmapsto\ (\llbracket\,c\,\rrbracket,\ \sigma)}$$

Příkazy — sémantika (small-step — ver. 3)

Tuto variantu small-step sémantiky je ještě dále možné upravit následujícím způsobem:

- Místo příkazu $[\![c]\!]$ používat posloupnost příkazů κ.

  Posloupnost κ je buď:

  - prázdná posloupnost $[]$, nebo
  - posloupnost tvaru $[\![c]\!] :: \kappa'$

- Výpočet začíná ve stavu

$$([\![c]\!] :: [], \sigma_0)$$

  kde σ_0 reprezentuje počáteční hodnoty proměnných a c reprezentuje celý původní program.

- Výpočet končí při dosažení prázdné posloupnosti $[]$, tj. ve stavu tvaru

$$([], \sigma)$$

Příkazy — sémantika (small-step — ver. 3)

$$\frac{\sigma \vdash a \Rightarrow n}{(\llbracket x := a \rrbracket :: \kappa,\ \sigma) \longmapsto (\kappa,\ \sigma')} \quad \text{kde } \sigma' = \sigma[x \mapsto n]$$

$$(\llbracket \texttt{skip} \rrbracket :: \kappa,\ \sigma) \longmapsto (\kappa,\ \sigma)$$

$$(\llbracket c_1 \ ;\ c_2 \rrbracket :: \kappa,\ \sigma) \longmapsto (\llbracket c_1 \rrbracket :: \llbracket c_2 \rrbracket :: \kappa,\ \sigma)$$

$$\frac{\sigma \vdash b \Rightarrow \text{true}}{(\llbracket \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \rrbracket :: \kappa,\ \sigma) \longmapsto (\llbracket c_1 \rrbracket :: \kappa,\ \sigma)}$$

$$\frac{\sigma \vdash b \Rightarrow \texttt{false}}{(\llbracket \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \rrbracket :: \kappa,\ \sigma) \longmapsto (\llbracket c_2 \rrbracket :: \kappa,\ \sigma)}$$

$$(\llbracket \texttt{while } b \texttt{ do } c_1 \rrbracket :: \kappa,\ \sigma) \longmapsto$$
$$(\llbracket \texttt{if } b \texttt{ then } (c_1 \ ;\ \texttt{while } b \texttt{ do } c_1) \texttt{ else skip} \rrbracket :: \kappa,\ \sigma)$$

Příkazy — sémantika (small-step — ver. 3)

Alternativně je možné mít místo jednoho pravidla pro `while`:

$$(\llbracket \texttt{while}\ b\ \texttt{do}\ c_1 \rrbracket :: \kappa,\ \sigma) \longmapsto (\llbracket \texttt{if}\ b\ \texttt{then}\ (c_1\ ;\ \texttt{while}\ b\ \texttt{do}\ c_1)\ \texttt{else}\ \texttt{skip} \rrbracket :: \kappa,\ \sigma)$$

dvě samostatná pravidla:

$$\frac{\sigma \vdash b \Rightarrow \text{true}}{(\llbracket \texttt{while}\ b\ \texttt{do}\ c_1 \rrbracket :: \kappa,\ \sigma) \longmapsto (\llbracket c_1 \rrbracket :: \llbracket \texttt{while}\ b\ \texttt{do}\ c_1 \rrbracket :: \kappa,\ \sigma)}$$

$$\frac{\sigma \vdash b \Rightarrow \texttt{false}}{(\llbracket \texttt{while}\ b\ \texttt{do}\ c_1 \rrbracket :: \kappa,\ \sigma) \longmapsto (\kappa,\ \sigma)}$$