Strukturování důkazů

Strukturování důkazů

Při interaktivním vytváření důkazů máme obecně vždy několik podcílů, které zbývá vyřešit:

- Pokud není uvedeno jinak, je daná taktika použita vždy na **první** z těchto podcílů.
- Je možné specifikovat, že taktika se má použít na nějaký konkrétní podcíl.

  Například, pokud chceme aplikovat taktiku *tac* na třetí podcíl, je možné napsat:

$3 : \mathit{tac}$

Strukturování důkazů

- Je možné aplikovat taktiku na více cílů najednou.

  Aplikuje taktiku *tac* na třetí, čtvrtý, pátý, osmý a jedenáctý podcíl:

3–5, 8, 11 : *tac*

- Taktiku je také možné aplikovat na všechny cíle najednou:

all : *tac*

Strukturování důkazů — odrážky

Důkaz je možné strukturovat pomocí **odrážek** (**bullets**):

- Odrážka je tvořena jedním nebo více z následujících symbolů:

```
-    +    *
```

- Pokud je použito více symbolů, musí se v rámci jedné odrážky opakovat stále stejný symbol:

```
--    +++++    ***
```

- odrážka skryje všechny ostatní podcíle
- na konci sekvence taktik, který začala danou odrážkou, se kontroluje, zda příslušný podcíl byl celý vyřešen

Strukturování důkazů

Další možností strukturování důkazů pomocí **složených závorek**:

$$\{ \quad \ldots \quad \}$$

- skryje všechny ostatní podcíle
- v rámci složených závorek je možné znovu použít stejné odrážky, jako byly použity na vyšší úrovni

Složené závorky je možné použít na jeden konkrétní podcíl
(ale ne na více podcílů najednou):

$$3 : \{ \quad \ldots \quad \}$$

Strukturování důkazů

Nevyřešený podcíl je možné dočasně „vyřešit" pomocí speciální taktiky admit:

```
admit.
```

- Tato taktika označí daný podcíl jako nevyřešený, ale umožní s ním pracovat, jako by už byl vyřešen.
- Pokud zbývá nějaký takový nevyřešený podcíl, označený pomocí admit, není možné důkaz ukončit pomocí **Qed**.

  Důkaz musí být v takovém případě ukončen pomocí **Admitted**.
- Tato taktika by se neměla vyskytovat v hotových důkazech.

  Slouží jako dočasná pomůcka při postupném vytváření důkazů.

Programování taktik

Ltac

Taktiky používané pro interaktivní vytváření důkazů je možno skládat dohromat a vytvářet pomocí nich složitější taktiky.

Jazyk pro vytváření nových složitějších taktik skládaním z již existujících taktik se nazývá **Ltac**.

Konstrukce jazyka Ltac sloužící k tomuto skládání taktik se označují jako **tacticals**.

Obecně je možné vytvářet libovolně složité **výrazy reprezentující taktiky** (**tactic expressions**):

- Při interaktivním provádějí taktik se vždy provede celý výraz najednou.
- Běžné taktiky (intro, apply, ...) jsou nejjednoduším případem tactic expressions.

Ltac

Provádění libovolného Ltac výrazu (a speciálně provedení každé jednotlivé taktiky) může skončit jedním ze dvou způsobů:

- úspěšné provedení:
  - z aktuálního cíle vznikne nějaký libovolný počet nových cílů (může být i nulový)
  - je zároveň vybudována příslušná část důkazu
- neúspěšné provedení — nastane chyba

Ltac

Při provádění taktik:

- Provedení celého výrazu musí skončit úspěšně, jinak se jedná o chybu.
- Provádění některých jeho podvýrazů může skončit chybou.
- To, jesli taková neúspěšná provedení podvýrazů vedou k celkové chybě, záleží na použití konkrétních tacticals, pomocí kterých byly tyto taktiky zkombinovány s dalšími taktikami.
- Některé tacticals umožňují provádění **backtrackingu** — při neúspěchu některé taktiky se vrátí zpět a zkusí jinou možnost.
- Některé výrazy mohou mít více možností úspěchu, které se zkouší při backtrackingu postupně.

  (Toto se většinou netýká základních jednoduchých taktik, ale spíše složitějších taktik vytvořených pomocí tacticals.)

Tacticals

Konkrétní příklady tacticals:

- $tac_0;\ tac_1$
  - provede taktiku tac_0 a na každý vygenerovaný podcíl aplikuje taktiku tac_1
  - jestliže selže taktika tac_0 nebo taktika tac_1 selže na některém z podcílů, celkově výraz selže
  - funguje i v případě, kdy tac_0 daný cíl kompletně vyřeší (tj. když nevzniknou žádné podcíle)
    tac_1 se v takovém případě vůbec nebude volat

- $tac_0;\ [\ tac_1 \mid tac_2 \mid \cdots \mid tac_n\ ]$
  - provede taktiku tac_0, ta musí vygenerovat přesně n podcílů
    na těchto n podcílů jsou aplikovány taktiky $tac_1, tac_2, \ldots, tac_n$
  - libovolné z taktik $tac_1, tac_2, \ldots, tac_n$ je možno vynechat
    — příslušný podcíl bude ponechán beze změny

Tacticals

- $tac_0 + tac_1$
  - nejprve provede taktiku tac_0
  - pokud bylo provedení taktiky tac_0 úspěšné, pracuje se dál s výslednými podcíly této taktiky
  - pokud bylo provedení taktiky tac_0 neúspěšné, zkusí se provést taktika tac_1
  - pokud i taktika tac_1 selže, celkově výraz selže
  - provádí backtracking — jestliže je taktika tac_0 úspěšná, ale selže řešení jí vygenerovaných podcílů, zkusí se provést taktika tac_1

Tacticals

Příklad: $(tac_0 + tac_1);\ tac_2$

- Řekněme, že tac_0 je úspěšná a vygeneruje nějaké podcíle, ale tac_2 na některém z těchto podcílů selže.
- Dojde k backtrackingu:
  - důkaz se vrátí do situace před provedením tac_0
  - použije se taktika tac_1
  - pokud je úspěšná, na vygenerované podcíle se aplikuje taktika tac_2

Tacticals

- $tac_0 \mid\mid tac_1$
  - nejprve provede taktiku tac_0
  - pokud bylo provedení taktiky tac_0 úspěšné, pracuje se dál s výslednými podcíly této taktiky
  - pokud provedení taktiky tac_0 selže, provede se taktika tac_1
  - na rozdíl od $tac_0 + tac_1$ neprovádí backtracking

  Příklad: $(tac_0 \mid\mid tac_1);\ tac_2$

  Pokud bude provedení tac_0 úspěšné, ale selže provedení tac_2 na některém z vygenerovaných podcílů, taktika tac_1 už se zkoušet nebude a celý výraz skončí neúspěchem

Tacticals

- try tac_0
  - zkusí provést taktiku tac_0
  - pokud je úspěšná, výsledek provedení této taktiky je i celkovým výsledkem
  - pokud je neúspěšná, stav důkazu se vrátí do situace před jejím provedením (tj. celkově se neprovede nic)

    a tento výsledek je považován za úspěch
    — tj. taktika try tac_0 nikdy neselže

- tryif tac_0 then tac_1 else tac_2
  - provede taktiku tac_0
  - pokud je úspěšná, provede na její výsledek taktiku tac_1
  - pokud je neúspěšná, provede taktiku tac_2

Tactics

- do n tac_0

  provede n krát taktiku tac_0

- repeat tac_0

  - opakovaně provádí stále dokola taktiku tac_0
  - skončí v okamžiku, kdy buď tac_0 skončí neúspěchem nebo když je sice tac_0 úspěšná, ale nijak nezmění cíl
    (tj. když ve vytváření důkazu nedojde k žádnému pokroku)
  - nikdy není neúspěšná — v nejhorším případě provede taktiku tac_0 nula krát

- progress tac_0

  provede taktiku tac_0 a skončí neúspěchem, pokud tac_0 byla sice úspěšná, ale nijak nezměnila cíl

Tacticals

Následující dvě taktiky nedělají nic, tj. nijak nemění cíl.

Nedává příliš smysl je provádět samostatně, ale v rámci větších výrazů mohou být někdy užitečné.

- idtac

  neudělá nic, ale vždy skončí úspěchem

- fail

  neudělá nic, ale vždy skončí neúspěchem

Obě tyto taktiky mohou být volány s libovolným počtem argumentů — např. s řetězci, čísly nebo hodnotami proměnných.

Hodnoty těchto argumentů jsou vypsány na výstup — např. do okna s debugovacími výpisy či informacemi o chybách.

Tacticals

- time tac_0

  provede taktiku tac_0 a zároveň při tom změří, jak dlouho trvalo provedení této taktiky

- timeout n tac_0

  začne provádět taktiku tac_0

  Pokud provádění této taktiky neskončí do n sekund, je toto provádění násilně ukončeno a taktika skončí chybou timeout.

  Poznámka: Tuto taktiku není vhodné používat v hotových důkazech, ale někdy se může hodit při ladění (například pro automatické odchycení případů, kdy se provádění nějaké taktiky zacyklí).

Programování vlastních taktik

Vlastní taktiky je možné definovat pomocí příkazu **Ltac**:

> **Ltac** my_tactic := tac_0.

Taktika může mít parametry, které je možné používat v rámci výrazu tac_0, který tvoří tělo této taktiky:

> **Ltac** my_tactic x y := tac_0.

Proměnné používané v rámci jazyka taktik:

- jsou **dynamicky typované** — tj. nemají pevně určený typ, je do nich možné přiřadit cokoliv (např. libovolný term, libovolný výraz reprezentující taktiku, apod.)

Programování vlastních taktik

V rámci těla taktiky je možné definovat **lokální proměnné** pomocí konstrukce **let . . . in**:

$$\textbf{let } x := tac_0 \textbf{ in } tac_1$$

Do proměnné je možné uložit jako hodnotu libovolný term t:

$$\textbf{let } x := \text{constr: } t \textbf{ in } tac_1$$

Také je možné do ní uložit výsledek vyhodnocí daného termu t:

$$\textbf{let } x := \text{eval simpl in } t \textbf{ in } tac_1$$

Programování vlastních taktik

V rámci taktik je také možné provádět **pattern matching**:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{match}\ x\ \textbf{with}} \\
| & \ldots & \Rightarrow & tac_1 \\
| & \ldots & \Rightarrow & tac_2 \\
 & \vdots & & \\
\multicolumn{4}{l}{\textbf{end}}
\end{array}
$$

- příkaz **match** provádí backtracking, tj. postupně zkouší všechny větve, které odpovídají dané hodnotě
- místo **match** je možné použít příkaz **lazymatch**
  — neprovádí backtracking, vybere jen první větev, která vyhovuje, ostatní možnosti nezkouší

Programování vlastních taktik

Speciálním případem je použití příkazu **match** vůči aktuálnímu cíli:

$$
\begin{array}{l}
\textbf{match}\ \text{goal}\ \textbf{with} \\
\mid\ [\ \_ : \ ?x = ?a,\ \_ : ?x > 0\ \ \texttt{|-}\ \text{my\_func}\ ?a = \_\ ] \Rightarrow\ \ tac_1 \\
\qquad \vdots \\
\textbf{end}
\end{array}
$$

- V patternu není třeba uvádět všechny předpoklady daného cíle: daný cíl bude odpovídat danému vzoru, pokud se najdou mezi předpoklady daného cíle předpoklady, které odpovídají uvedeným vzorům.
- Do proměnných začínajících ve vzoru otazníkem se přiřadí příslušný podterm ze skutečného cíle — tyto proměnné je pak možné používat ve výraze pro taktiku pro příslušenou větev.

Induktivně definované relace

Induktivně definované relace

Příkaz **Inductive** je možno použít i pro definici **induktivně definovaných relací**:

$$
\begin{aligned}
&\textbf{Inductive}\ \mathsf{even} : \mathsf{nat} \longrightarrow \textbf{Prop} \ := \\
&\qquad |\ \ \mathsf{ev\_0} : \mathsf{even}\ 0 \\
&\qquad |\ \ \mathsf{ev\_succ\_succ} : \forall n : \mathsf{nat},\ \mathsf{even}\ n \longrightarrow \mathsf{even}\ (\mathsf{S}\ (\mathsf{S}\ n)).
\end{aligned}
$$

- Konstruktory představují možnosti, jak je možné zdůvodnit, že daný prvek (nebo prvky) je v dané relaci.
- V dané relaci jsou právě ty prvky, pro které je možné pomocí uvedených konstruktorů zdůvodnit, že v dané relaci jsou.

Induktivně definované relace

Příklad: Relace le (tj. relace "$\leq$" — menší nebo rovno) na typu nat může být definována následujícím způsobem:

$$
\begin{array}{l}
\textbf{Inductive}\ \mathsf{le} : \mathsf{nat} \rightarrow \mathsf{nat} \rightarrow \textbf{Prop} \ := \\
\quad \mid\ \mathsf{le\_n} : \forall n : \mathsf{nat}, \mathsf{le}\ n\ n \\
\quad \mid\ \mathsf{le\_S} : \forall n\ m : \mathsf{nat}, \mathsf{le}\ n\ m \rightarrow \mathsf{le}\ n\ (\mathsf{S}\ m).
\end{array}
$$

Poznámka: Ve skutečnosti je relace le definována ve standardní knihovně takto:

$$
\begin{array}{l}
\textbf{Inductive}\ \mathsf{le}\ (n : \mathsf{nat}) : \mathsf{nat} \rightarrow \textbf{Prop} \ := \\
\quad \mid\ \mathsf{le\_n} : \mathsf{le}\ n\ n \\
\quad \mid\ \mathsf{le\_S} : \forall m : \mathsf{nat}, \mathsf{le}\ n\ m \rightarrow \mathsf{le}\ n\ (\mathsf{S}\ m).
\end{array}
$$

Induktivně definované relace

Řekněme, že máme definováno následující:

- Typ graph — typ reprezentující orientované grafy
- Typ node — typ vrcholů, které se mohou vyskytovat v grafech typu graph
- Relaci node_in_graph : graph $\rightarrow$ node $\rightarrow$ **Prop**

  — kde node_in_graph G v znamená, že vrchol v patří do množiny vrcholů grafu G
- Relaci edge_in_graph : graph $\rightarrow$ node $\rightarrow$ node $\rightarrow$ **Prop**

  — kde edge_in_graph G u v znamená, že v grafu G vede hrana z vrcholu u do vrcholu v
  (a oba tyto vrcholy patří do množiny vrcholů grafu G)

Induktivně definované relace

Jednou z možností, jak definovat, co to znamená, že v grafu G existuje **cesta** z vrcholu u do vrcholu v, je následující:

$$
\begin{array}{l}
\textbf{Inductive}\ \mathsf{path}\ (G : \mathsf{graph}) : \mathsf{node} \longrightarrow \mathsf{node} \longrightarrow \textbf{Prop}\ \ := \\
\quad\quad \mid\ \mathsf{path\_nil} : \forall v : \mathsf{node},\ \ \mathsf{node\_in\_graph}\ \ G\ \ v \longrightarrow \mathsf{path}\ \ G\ \ v\ \ v \\
\quad\quad \mid\ \mathsf{path\_cons} : \forall u\ \ v\ \ w : \mathsf{node}, \\
\quad\quad\quad\quad \mathsf{edge\_in\_graph}\ \ G\ \ u\ \ v \longrightarrow \mathsf{path}\ \ G\ \ v\ \ w \longrightarrow \mathsf{path}\ \ G\ \ u\ \ w.
\end{array}
$$

Tvrzení, že v grafu G existuje cesta z vrcholu u do vrcholu v, je pak možné reprezentovat následujícím zápisem:

$$\mathsf{path}\ \ G\ \ u\ \ v$$

Induktivně definované relace

Příklad: Induktivní definice toho, že jeden seznam nad prvky typu A je **permutací** prvků jiného seznamu nad prvky typu A:

$$
\begin{array}{l}
\textbf{Inductive}\ \mathrm{permut} : \mathrm{list}\ A \rightarrow \mathrm{list}\ A \rightarrow \textbf{Prop}\ := \\
\quad \mid\ \mathrm{permut\_refl} : \forall \ell : \mathrm{list}\ A,\ \mathrm{permut}\ \ell\ \ell \\
\quad \mid\ \mathrm{permut\_cons} : \forall\ (a : A)\ (\ell_0\ \ell_1 : \mathrm{list}\ A), \\
\qquad\qquad \mathrm{permut}\ \ell_0\ \ell_1 \rightarrow \mathrm{permut}\ (\mathrm{cons}\ a\ \ell_0)\ (\mathrm{cons}\ a\ \ell_1) \\
\quad \mid\ \mathrm{permut\_append} : \forall\ (a : A)\ (\ell : \mathrm{list}\ A), \\
\qquad\qquad \mathrm{permut}\ (\mathrm{cons}\ a\ \ell)\ (\ell \mathbin{++} (\mathrm{cons}\ a\ \mathrm{nil})) \\
\quad \mid\ \mathrm{permut\_trans} : \forall\ \ell_0\ \ell_1\ \ell_2 : \mathrm{list}\ A, \\
\qquad\qquad \mathrm{permut}\ \ell_0\ \ell_1 \rightarrow \mathrm{permut}\ \ell_1\ \ell_2 \rightarrow \mathrm{permut}\ \ell_0\ \ell_2.
\end{array}
$$

Sémantika programovacího jazyka

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 Coqu

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:

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

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 Coqu 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 >-> 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 $A \rightarrow B$, 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 aexp $\rightarrow$ nat
- jako **relaci** — například typu aexp $\rightarrow$ nat $\rightarrow$ **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

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

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:

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

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

$$\mathsf{var} \longrightarrow \mathsf{nat}$$

- 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

$$\mathrm{aeval} : \mathrm{state} \longrightarrow \mathrm{aexp} \longrightarrow \mathrm{nat} \longrightarrow \mathbf{Prop}$$

Výraz

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

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

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

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.

$$\sigma_1 \mathrel{=\!\!\prec} c \mathrel{\succ\!\!\Rightarrow} \sigma_2$$

Příkazy — sémantika (big-step)

$$\sigma \models\!\langle\, \texttt{skip} \,\rangle\!\!\Rightarrow \sigma \qquad\qquad \frac{\sigma \vdash a \Rightarrow n}{\sigma \models\!\langle\, x := a \,\rangle\!\!\Rightarrow \sigma'} \quad \text{kde } \sigma' = \sigma[x \mapsto n]$$

$$\frac{\sigma \models\!\langle\, c_1 \,\rangle\!\!\Rightarrow \sigma' \qquad \sigma' \models\!\langle\, c_2 \,\rangle\!\!\Rightarrow \sigma''}{\sigma \models\!\langle\, c_1 \,;\, c_2 \,\rangle\!\!\Rightarrow \sigma''}$$

$$\frac{\sigma \vdash b \Rightarrow \mathsf{true} \qquad \sigma \models\!\langle\, c_1 \,\rangle\!\!\Rightarrow \sigma'}{\sigma \models\!\langle\, \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \,\rangle\!\!\Rightarrow \sigma'}$$

$$\frac{\sigma \vdash b \Rightarrow \texttt{false} \qquad \sigma \models\!\langle\, c_2 \,\rangle\!\!\Rightarrow \sigma'}{\sigma \models\!\langle\, \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \,\rangle\!\!\Rightarrow \sigma'}$$

$$\frac{\sigma \vdash b \Rightarrow \texttt{false}}{\sigma \models\!\langle\, \texttt{while } b \texttt{ do } c \,\rangle\!\!\Rightarrow \sigma}$$

$$\frac{\sigma \vdash b \Rightarrow \mathsf{true} \qquad \sigma \models\!\langle\, c \,\rangle\!\!\Rightarrow \sigma' \qquad \sigma' \models\!\langle\, \texttt{while } b \texttt{ do } c \,\rangle\!\!\Rightarrow \sigma''}{\sigma \models\!\langle\, \texttt{while } b \texttt{ do } c \,\rangle\!\!\Rightarrow \sigma''}$$

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:

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

- 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)

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

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

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

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

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

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

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:

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

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

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

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

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

- Výpočet programu se zastaví v koncovém stavu tvaru

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

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

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

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

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

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

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

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

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:

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

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

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)

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

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

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

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

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

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

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

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

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

dvě samostatná pravidla:

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

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