

NSWI183: SÉMANTIKA PROGRAMŮ

5. ÚPLNÁ SPRÁVNOST

Jan Kofroň



MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

Department of
Distributed and
Dependable
Systems



- Viděli jsme, že částečná správnost je velmi užitečná vlastnost programů, kterou má smysl dokazovat
- Úplná správnost zahrnuje navíc ještě důkaz, že výpočet vždy skončí
 - „**Termination problem**“
 - Pokud je program zároveň částečně správný, vydá vždy správný výsledek, tedy výsledek splňující postcondition

- Program může uvíznout v nekonečném výpočtu buď v důsledku uvíznutí v cyklu (for nebo while), nebo při neomezené rekurzi
- Využijeme princip fundované relace (a indukce)
- Abychom konečnost výpočtu dokázali, musíme najít vhodnou fundovanou relaci
- Poté je třeba najít funkci, která nám mapuje stavy programů na prvky relace
 - Ta je jen technickým usnadněním a způsobem, jakým se konečnost výpočtu dokazuje v PiVC, formálně je pro důkaz nutná jen relace
- Takové funkci říkáme „**ranking function**“
- Funkce odráží existenci fundované relace mezi stavy jednotlivých iterací

- V Pi lze používat n-tice celých čísel, které se lexikograficky porovnávají
- I pokud funkce obsahuje více cyklů, ranking funkce musí být pouze jedna
 - Přes všechny základní cesty v rámci funkce
- Dokázat konečnost je samozřejmě potřeba pro každou funkci (zvláště)
- Verifikují se pak dvě vlastnosti:
 1. Hodnota ranking funkce je na konci základní cesty ostře menší než na jejím začátku
 2. Hodnota ranking funkce je vždy nezáporná

- Stejně jako v případě částečné správnosti je terminace ověřována pomocí verifikačních podmínek vygenerovaných z relevantních základních cest
- Při ověření podmínek se využívá i formulí v anotacích pro částečnou správnost
 - Může být nutné rozšířit anotace pro částečnou správnost, často zejména kvůli důkazu nezápornosti ranking funkce

PŘÍKLAD – BUBBLESORT

```
@pre true
@post sorted(rv, 0, |rv| - 1)
int[] BubbleSort(int[] a_0) {
    int[] a := a_0;
    for @L1: -1 <= i && ...
        # (i + 1, i + 1)
        (int i := |a| - 1; i > 0; i := i - 1) {
            for @L2: 1 <= i && 0 <= j && j <= i && ...
                # (i + 1, i - j)
                (int j := 0; j < i; j := j + 1) {
                    if (a[j] > a[j + 1]) {
                        int t := a[j];
                        a[j] := a[j + 1];
                        a[j + 1] := t;
                    }
                }
            }
        }
    return a;
}
```

PŘÍKLAD – BUBBLESORT

```

@pre true
@post sorted(rv, 0, |rv| - 1)
int[] BubbleSort(int[] a_0) {
  int[] a := a_0;
  for @L1: -1 <= i && ...
    # (i + 1, i + 1)
    (int i := |a| - 1; i > 0; i := i - 1) {
      for @L2: 1 <= i && 0 <= j && j <= i && ...
        # (i + 1, i - j)
        (int j := 0; j < i; j := j + 1) {
          if (a[j] > a[j + 1]) {
            int t := a[j];
            a[j] := a[j + 1];
            a[j + 1] := t;
          } } }
  return a;
}

```

1

$$\begin{aligned}
 &(-1 \leq i) \\
 &\quad \rightarrow \\
 &((i > 0)) \\
 &\quad \rightarrow \\
 &(((i + 1 < i + 1) \vee ((i + 1 = i + 1) \wedge (i - 0 < i + 1))))
 \end{aligned}$$

PŘÍKLAD – BUBBLESORT

```

@pre true
@post sorted(rv, 0, |rv| - 1)
int[] BubbleSort(int[] a_0) {
  int[] a := a_0;
  for @L1: -1 <= i && ...
    # (i + 1, i + 1)
    (int i := |a| - 1; i > 0; i := i - 1) {
  for @L2: 1 <= i && 0 <= j && j <= i && ...
    # (i + 1, i - j)
    (int j := 0; j < i; j := j + 1) {
      if (a[j] > a[j + 1]) {
        int t := a[j];
        a[j] := a[j + 1];
        a[j + 1] := t;
      } } }
  return a;
}

```

2

$$\begin{aligned}
 &(1 \leq i \wedge 0 \leq j \wedge j \leq i) \\
 &\quad \rightarrow \\
 &\quad ((j < i) \\
 &\quad \quad \rightarrow \\
 &\quad \quad ((a[j] > a[j + 1]) \\
 &\quad \quad \quad \rightarrow \\
 &\quad ((i + 1 < i + 1) \vee ((i + 1 = i + 1) \wedge (i - (j + 1) < i - j))))
 \end{aligned}$$

PŘÍKLAD – BUBBLESORT

```

@pre true
@post sorted(rv, 0, |rv| - 1)
int[] BubbleSort(int[] a_0) {
  int[] a := a_0;
  for @L1: -1 <= i && ...
    # (i + 1, i + 1)
    (int i := |a| - 1; i > 0; i := i - 1) {
  for @L2: 1 <= i && 0 <= j && j <= i && ...
    # (i + 1, i - j)
    (int j := 0; j < i; j := j + 1) {
      if (a[j] > a[j + 1]) {
        int t := a[j];
        a[j] := a[j + 1];
        a[j + 1] := t;
      } } }
  return a;
}

```

3

$$\begin{aligned}
 & (1 \leq i \wedge 0 \leq j \wedge j \leq i) \\
 & \quad \rightarrow \\
 & \quad ((j < i) \\
 & \quad \quad \rightarrow \\
 & \quad \quad (\neg(a[j] > a[j + 1]) \\
 & \quad \quad \quad \rightarrow \\
 & \quad \quad ((i + 1 < i + 1) \vee ((i + 1 = i + 1) \wedge (i - (j + 1) < i - j))))))
 \end{aligned}$$

PŘÍKLAD – BUBBLESORT

```

@pre true
@post sorted(rv, 0, |rv| - 1)
int[] BubbleSort(int[] a_0) {
  int[] a := a_0;
  for @L1: -1 <= i && ...
    # (i + 1, i + 1)
    (int i := |a| - 1; i > 0; i := i - 1) {
  for @L2: 1 <= i && 0 <= j && j <= i && ...
    # (i + 1, i - j)
    (int j := 0; j < i; j := j + 1) {
      if (a[j] > a[j + 1]) {
        int t := a[j];
        a[j] := a[j + 1];
        a[j + 1] := t;
      } } }
  return a;
}

```

4

$$(1 \leq i \wedge 0 \leq j \wedge j \leq i)$$

→

$$(\neg(j < i))$$

→

$$((i - 1 + 1 < i + 1) \vee ((i - 1 + 1 = i + 1) \wedge (i - 1 + 1 < i - j))))$$

Uvažme následující úlohu: Máme pytel s **červenými**, **žlutými** a **modrými** žetony. Pokud je v pytli poslední žeton, můžeme ho vytáhnout. V opačném případě vyjmemе dva žetony a postupujeme následujícím způsobem (máme k dispozici libovolně mnoho žetonů):

1. Pokud je jeden ze dvou právě vyjmutých žetonů **červený**, nevložíme zpátky žádný žeton.
2. Pokud jsou oba žetony **žluté**, vložíme do pytle jeden **žlutý** a pět **modrých** žetonů.
3. Pokud je jeden z žetonů **modrý** a druhý **není červený**, vložíme do pytle deset **červených**.

Tento postup specifikuje všechny kombinace pro vytažené žetony. Úloha spočívá v tom rozhodnout, jestli tento proces je vždy konečný, tedy jestli bude pytel bez ohledu na počáteční obsah.

- Řešení spočívá v nalezení ranking funkce, neboli vhodné reprezentace stavů tak, aby její hodnota během výpočtu klesala
- Pro náš případ to může být (**#žlutých**, **#modrých**, **#červených**) ~ (ž, m, č)
 - Ranking funkce, která dokáže terminaci daného programu, obecně nemusí být jen jedna

- Uvažme změnu pravidel u hry s žetony. Zastaví algoritmus, pokud:
 - A. V kroku 1 vrátíme jeden **červený** žeton zpět?
 - B. V kroku 1 přidáme jeden **modrý** žeton?
 - C. V kroku 1 přidáme **modrý** žeton a v kroku 3 vrátíme jeden **modrý** žeton zpět, ale žádné jiné žetony?

- V kroku 1 vrátíme jeden červený žeton zpět
- Důkaz i reprezentace stavu jsou stejné, v prvním kroku jsou možnosti:
 - stav po odebrání je $(\checkmark - 1, m, \checkmark)$, $(\checkmark, m - 1, \checkmark)$ nebo $(\checkmark, m, \checkmark - 1)$. Všechny tyto stavy jsou v relaci $\prec_3$ menší než $(\checkmark, m, \checkmark)$.

- V kroku 1 přidáme **modrý** žeton a v kroku 3 vrátíme jeden **modrý** žeton zpět, ale žádné jiné žetony
- Musíme zvolit jinou reprezentaci stavu: (ž, č, m)
- Pak budou opět fungovat všechny případy a algoritmus tedy zastaví

- V kroku 1. přidáme jeden **modrý** žeton
- Důkaz nebude fungovat, když budou oba žetony červené, protože nový stav bude $(\text{ž}, m+1, \text{č}-2)$
- Jiné pořadí barev bude vždy narušovat nějaký případ (nový stav nebude menší než předchozí)
- Zdá se, že tady to nepůjde (ale dokázat to je jiná věc)
- Obecně se může například stát, že reprezentace bude klesat v každém sudém kroku, a to víc, než bude potenciálně stoupat v lichých krocích
 - Pak může být vhodné ranking funkci zvolit na základě jiných proměnných než obsahu pytle
 - To samozřejmě neplatí pro naši úlohu

Uvažme následující úlohu: Máme pytel s **červenými**, **žlutými** a **modrými** žetony. Pokud je v pytli poslední žeton, můžeme ho vytáhnout. V opačném případě vyjmeme dva žetony a postupujeme následujícím způsobem (máme k dispozici libovolně mnoho žetonů):

1. Pokud je jeden ze dvou právě vyjmutých žetonů **červený**, vložíme do pytle jeden **modrý** žeton.
2. Pokud jsou oba žetony **žluté**, vložíme do pytle jeden **žlutý** a pět **modrých** žetonů.
3. Pokud je jeden z žetonů **modrý** a druhý **není červený**, vložíme do pytle deset **červených**.

Tento postup specifikuje všechny kombinace pro vytažené žetony. Úloha spočívá v tom rozhodnout, jestli tento proces je vždy konečný, tedy jestli bude pytel bez ohledu na počáteční obsah.

Dokažme **neterminaci**. Uvažme stav pytle, kde jsou minimálně čtyři žetony **červené**:

1. Vytáhneme dva **červené** žetony a vložíme zpět jeden **modrý** (nové pravidlo 1).
2. Opět vytáhneme dva **červené** žetony a opět vložíme zpět jeden **modrý**.
3. Vytáhneme dva **modré** a vložíme deset **červených** (pravidlo 3).

Po třetím kroku opět platí, že v pytli jsou nejméně čtyři žetony **červené** (vlastně jich je tam nejméně deset) a můžeme celý postup opakovat do nekonečna, přičemž počet žetonů v pytli bude růst.

PŘÍKLAD S ŽETONY V PI – PŮVODNÍ PRAVIDLA

```
@pre  red >= 0 && blue >= 0 && yellow >= 0
@post true
void compute(int red, int blue, int yellow) {
    int[] state := new int[3];
    state[0] := red;
    state[1] := yellow;
    state[2] := blue;
    while @ ...
        # (state[1], state[2], state[0])
        (state[0] > 0 || state[1] > 0 || state[2] > 0) {
            if (state[0] + state[1] + state[2] = 1) break;
            int first := removeChip(state); state[first] := state[first] - 1;
            int second := removeChip(state); state[second] := state[second] - 1;
            if ((first = 0) || (second = 0)) { }
            else if ((first = 1) && (second = 1)) {
                state[1] := state[1] + 1; state[2] := state[2] + 5;
            }
            else if ((first = 2) || (second = 2)) {
                state[0] := state[0] + 10;
            }
        }
    }
}
```

- Co kdybychom chtěli (potřebovali) mít ranking funkci unární? Bylo by to možné?
 - Potřebujeme realizovat lexikografické uspořádání a porovnání

- Co kdybychom chtěli (potřebovali) mít ranking funkci unární? Bylo by to možné?
 - Potřebujeme realizovat lexikografické uspořádání a porovnání
- **Bylo!** Například: $\# \text{červených} + 11 \cdot \# \text{modrých} + 11 \cdot 6 \cdot \# \text{žlutých}$
 - Konstanty vycházejí z počtů žetonů vrácených žetonů do pytle (10 a 5)

- Žetony
- QuickSort
- MergeSort

- Vedle částečné správnosti je terminace další důležitou vlastností programů
- Dokázat ji bývá jednodušší než dokázat částečnou správnost
 - Spočívá v nalezení vhodné ranking funkce, jejíž hodnota v průběhu výpočtu klesá, ale je stále nezáporná
- Princip dokazování je podobný jako v případě částečné správnosti – program je rozložen na základní cesty, z nichž se následně vygenerují verifikační podmínky

- Doplňte anotace do funkce SelectSort a ověřte **konečnost** výpočtu v PiVC
 - Částečnou správnost ověřovat nemusíte
- Termín pro odevzdání je **10. 11. 2025, 23:59**.
- Zdrojový kód funkcí naleznete na webu předmětu:

`https://d3s.mff.cuni.cz/files/teaching/nswi183/selectsort.pi`