NSWI183: Sémantika programů

4. Částečná správnost

Jan Kofroň

MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

Department of Distributed and Dependable Systems
D3S

Částečná správnost (Partial correctness)

- Říkáme, že funkce je částečně správná (**partially correct**), pokud když je splněna precondition funkce při jejím zavolání, její postcondition je splněna při návratu z funkce (pokud nastane)
  - Obecně funkce nemusí skončit – může uvíznout v nekonečném výpočtu („termination problem“)
- Ověření, že funkce skutečně vždy skončí, je jiný a také těžký problém
  - Pro jazyky jako Java, C#, C/C++ se jedná o nerozhodnutelný problém – potřebovali bychom silnější mechanismus než je Turingův stroj, což nemáme
  - I pro programy v Pi musíme verifikátoru pomoci – více na příští přednášce o úplné správnosti
- Obecně rozhodnout o platnosti jiných vlastností než terminace programu je stejně těžké

Připomenutí: Dokazování částečné správnosti

- Principiálně probíhá verifikace v PI tak, že se anotace převedou na konečný počet **verifikačních podmínek**, které se postupně ověří
- Vytvoření verifikačních podmínek probíhá ve dvou krocích:
  1. Tělo (každé) funkce je převedeno na konečný počet **základních cest** (basic paths)
  2. Pro každou základní cestu se vygeneruje verifikační podmínka
- Pokud jsou splněny všechny verifikační podmínky, platí pak i postcondition vždy, když je splněna precondition

PŘÍKLAD – CYKLY – BUBBLESORT

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

Rekurze

- U (rekurzivního) volání se nejprve zkontroluje platnost precondition volané funkce a následně předpokládá platnost postcondition
  - Těmito dvěma anotacemi se nahradí rekurzivní volání
  - Návratová hodnota je definovaná právě postcondition rekurzivně volané funkce
- Tím se tělo funkce opět rozpadne na konečný počet základních cest

Příklad – rekurze – Binární vyhledávání

```
@pre   true
@post  true
bool BinarySearch(int[] a, int l, int u, int e) {
    if (l > u) return false;
    else {
        int m := (l + u) div 2;
        if (a[m] = e) return true;
        else if (a[m] < e) return BinarySearch(a, m + 1, u, e);
        else return BinarySearch(a, l, m - 1, e);
    }
}
```

PŘÍKLAD – REKURZE – BINÁRNÍ VYHLEDÁVÁNÍ

```
else if (a[m] < e) {
  @ 0 <= m + 1 && u < |a| && sorted(a, 0, |a| - 1)
   // _rv = BinarySearch(a, m + 1, u, e);
   @assume _rv <-> exists j. (m + 1 <= j && j <= u && a[j] = e)
   return _rv;
}
else {
  @ 0 <= l && m - 1 < |a| && sorted(a, 0, |a| - 1)
   // _rv = BinarySearch(a, l, m - 1, e);
   @assume _rv <-> exists j. (l <= j && j <= m - 1 && a[j] = e)
   return _rv;
}
```

PŘÍKLAD – REKURZE – BINÁRNÍ VYHLEDÁVÁNÍ

```
else if (a[m] < e) {
  @ 0 <= m + 1 && u < |a| && sorted(a, 0, |a| - 1)
   // _rv = BinarySearch(a, m + 1, u, e);
   @assume _rv <-> exists j. (m + 1 <= j && j <= u && a[j] = e)
   return _rv;
}
else {
  @ 0 <= l && m - 1 < |a| && sorted(a, 0, |a| - 1)
   // _rv = BinarySearch(a, l, m - 1, e);
   @assume _rv <-> exists j. (l <= j && j <= m - 1 && a[j] = e)
   return _rv;
}
```

Stav programu

- **Stavem programu** rozumíme přiřazení hodnot odpovídajících typů ke všem proměnným
  - Proměnná je i „program counter“ (*pc*) obsahující aktuální místo (řádek) vykonávání programu
- Stav programu funkce BubbleSort na místě vykonávání L1 (vnější cyklus) může být:

$$s : \{pc \rightarrow L1, a \rightarrow [2;0;1], i \rightarrow 2, j \rightarrow 0, t \rightarrow 2, rv \rightarrow [\,]\}$$

- Formálně potřebujeme rozšíření teorie o definici typů a interpretaci funkčních a predikátových symbolů, nicméně zde budeme vždy předpokládat běžnou interpretaci a běžné typy hodnot

Verifikační podmínky

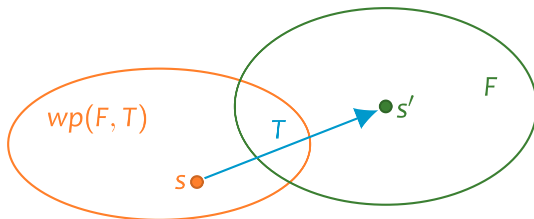
- Kromě anotací obsahuje program i příkazy měnící stav programu
  - Tedy hodnoty proměnných
- Po rozdělení programu na základní cesty je k vytvoření verifikačních podmínek potřeba zahrnout i efekty jednotlivých příkazů
- Princip, který se k tomu používá, se jmenuje „**weakest precondition predicate transformer**“ (predikátový transformátor nejslabšího předpokladu):

$$p : formule \times prikazy \to formule$$

- Ten dvojici (formule, příkaz) přiřazuje novou formuli charakterizující stav po provedení příkazu
  - Zjednodušeně se dá říci, že výsledná formule zachycuje efekt příkazu na původní formuli
  - Pokud je ale například původní formule velmi obecná, může ta samá formule platit i pro stav po provedení příkazu

WEAKEST PRECONDITION

Formule $wp(F, T)$ je **weakest precondition** stavu s, pokud $s \models wp(F, T)$ a po provedení příkazu T, který změní stav programu z s na stav s', platí $s' \models F$



Jak spočítat weakest precondition?

Verifikační podmínky obsahují podformule, které odpovídají:

1. Předpokladům (asumpcím) – podformule tvořící implikace, kromě poslední podformule
   - To jsou typicky části anotací
2. Přiřazení – podformule odrážející efekt přiřazení, tedy změnu hodnoty proměnné

- Pro asumpce platí, že pokud před provedením příkazu platí $c \rightarrow F$, pak po provedení příkazu platí F, tedy $wp(F, assume\ c) \leftrightarrow c \rightarrow F$
  - Například ve verifikační podmínce, kde platí podmínka cyklu
- Pro přiřazení platí, že pokud $F[e]$ platí před provedením příkazu $v := e$, pak po provedení přiřazení platí $F[v]$, tedy $wp(F[v], v := e) \leftrightarrow F[e]$
- Pro posloupnost příkazů $S_1; \ldots; S_n$ definujeme weakest precondition rekurzivně jako $wp(F, S_1, \ldots, S_n) \leftrightarrow wp(wp(F, S_n), S_1; \ldots; S_{n-1})$

Příklad – Weakest precondition

- Pokud máme základní cestu $@F\, S_1; S_2; \ldots S_n; @G$, verifikační podmínka je formule

$$F \rightarrow wp(G, S_1; S_2; \ldots S_n)$$

- Například pro základní cestu $@x \geq 0; x := x + 1; @x \geq 1$ je verifikační podmínka formule

$$x \geq 0 \rightarrow wp(x \geq 1, x = x + 1)$$

- Chceme zjistit, jaká formule odpovídá $wp(x \geq 1, x = x + 1)$, abychom mohli rozhodnout o platnosti verifikační podmínky:

$$\begin{aligned} & wp(x \geq 1, x = x + 1) \\ \leftrightarrow\ & (x \geq 1)\{x \mapsto x + 1\} \\ \leftrightarrow\ & x + 1 \geq 1 \\ \leftrightarrow\ & x \geq 0 \end{aligned}$$

- Po dosazení dostáváme, že $x \geq 0 \rightarrow x \geq 0$, což zřejmě platí

PŘÍKLAD – LINEÁRNÍ VYHLEDÁVÁNÍ

```
@pre   0 <= l && u < |a|
@post rv <-> exists j. (l <= j && j <= u && a[j] = e)
bool LinearSearch(int[] a, int l, int u, int e) {
    for
        @ L: l <= i && 0 <= l && u < |a| &&
             forall j. (l <= j && j < i -> a[j] != e)
        (int i := l; i <= u; i := i + 1)
    {
        @ 0 <= i && i < |a|
        if (a[i] = e)  return true;
    }
    return false;
}
```

Příklad – Lineární vyhledávání

- Základní cesta je v tomto případě posloupnost příkazů:

$$
\begin{aligned}
&@F : l \leq i \wedge \forall j.\, l \leq j < i \rightarrow a[j] \neq e \\
&S_1 : \mathit{assume}\; i \leq u; \\
&S_2 : \mathit{assume}\; a[i] = e; \\
&S_3 : rv := \mathit{true}; \\
&@G : rv \leftrightarrow \exists j.\, l \leq j \leq u \wedge a[j] = e
\end{aligned}
$$

- Verifikační podmínka je pak formule $F \rightarrow wp(G, S_1; S_2; S_3)$
- Spočítejme tedy, čemu odpovídá $wp(G, S_1; S_2; S_3)$

PŘÍKLAD – LINEÁRNÍ VYHLEDÁVÁNÍ

$$
\begin{aligned}
& wp(G, S_1; S_2; S_3) \\
\leftrightarrow\ & wp(wp(rv \leftrightarrow \exists j.\, l \leq j \leq u \wedge a[j] = e, rv := true), S_1; S_2) \\
\leftrightarrow\ & wp(true \leftrightarrow \exists j.\, l \leq j \leq u \wedge a[j] = e, S_1; S_2) \\
\leftrightarrow\ & wp(\exists j.\, l \leq j \leq u \wedge a[j] = e, S_1; S_2) \\
\leftrightarrow\ & wp(wp(\exists j.\, l \leq j \leq u \wedge a[j] = e, assume\, a[i] = e), S_1) \\
\leftrightarrow\ & wp(a[i] = e \rightarrow \exists j.\, l \leq j \leq u \wedge a[j] = e, S_1) \\
\leftrightarrow\ & wp(a[i] = e \rightarrow \exists j.\, l \leq j \leq u \wedge a[j] = e, assume\, i \leq u) \\
\leftrightarrow\ & i \leq u \rightarrow (a[i] = e \rightarrow \exists j.\, l \leq j \leq u \wedge a[j] = e)
\end{aligned}
$$

Příklad – Lineární vyhledávání

- Pokud nyní nahradíme spočítanou formuli ve verifikační podmínce $F \rightarrow wp(\ldots)$, dostáváme:

$$l \leq i \wedge (\forall j.\, l \leq j < i \rightarrow a[j] \neq e) \rightarrow (i \leq u \rightarrow (a[i] = e \rightarrow \exists j.\, l \leq j \leq u \wedge a[j] = e))$$

- což můžeme přepsat jako:

$$l \leq i \wedge (\forall j.\, l \leq j < i \rightarrow a[j] \neq e) \wedge i \leq u \wedge a[i] = e \rightarrow \exists j.\, l \leq j \leq u \wedge a[j] = e$$

- s použitím ekvivalence:

$$(F_1 \wedge F_2 \rightarrow (F_3 \rightarrow (F_4 \rightarrow F_5))) \leftrightarrow ((F_1 \wedge F_2 \wedge F_3 \wedge F_4) \rightarrow F_5)$$

- Výsledná formule je platná (pro případ $j = i$)

Příklad – InsertionSort

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

Příklad – UnsortedUnion

```
int e := ?;
@pre  true
@post true
int[] union(int[] a, int[] b) {
  int[] u := new int[|a| + |b|];
  int j := 0;
  for @ L1: true
      (int i := 0; i < |a|; i := i + 1) {
    u[j] := a[i];
    j := j + 1;
  }
  for @ L2: true
      (int i := 0; i < |b|; i := i + 1) {
    u[j] := b[i];
    j := j + 1;
  }
  return u;
}
```

SHRNUTÍ

- Částečná správnost zachycuje důležitou část korektnosti programu
- Pro její ověření je třeba rozložit program na základní cesty a z každé pak vygenerovat verifikační podmínku
  - Podmínka v podobě formule je pak automaticky ověřena SMT solverem
- Verifikační podmínka využívá principu „weakest precondition“ pro zachycení efektu jednotlivých příkazů