NSWI183: Sémantika programů

3. Pi a PiVC

Jan Kofroň

MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

Department of
Distributed and
Dependable
Systems

Jazyk Pi

- Pi je jednoduchý imperativní programovací jazyk
- Obsahuje podporu anotací
- **Neobsahuje** některé typické konstrukty programovacích jazyků jako jsou reference, ukazatele, globální proměnné
- Podobá se jazyku C s některými omezeními

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

```
@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;
}
```

(Formální) parametry

```
...
LinearSearch(b, 0, |b|-1, v);
...
```

Argumenty

Reference a ukazatele v Pi

- Pi nemá typ ukazatele ani reference, parametry se tedy předávají hodnotou
  - Včetně struktur a polí
- Navíc Pi nedovoluje měnit hodnotu parametrů
  - Musíme tedy hodnotu případně přiřadit do jiné proměnné
  - Tak můžeme vždycky (i v postcondition) odkázat na vstupní hodnoty parametrů
- `return` je jediný způsob, jakým může mít volaná funkce efekt na kontext volajícího
  - … a jak smysluplně napsat postcondition

STRUKTURY

- Kromě polí obsahuje jazyk Pi i podporu pro struktury:

```
struct qs {
    int pivot;
    int[] array;
}
```

- Pomocí tečkové notace se odkazujeme k jednotlivým položkám, např.:

```
qs var;
int p := var.pivot;
var.array[0] := 5;
```

Anotace

- Anotace jsou důležitou vlastností jazyka Pi
- Anotace je formule, jejíž volné proměnné jsou pouze proměnné funkce, ke které se anotace váže
- Anotace F na místě L vyjadřuje, že F je splněná vždycky, když exekuce programu dosáhne L
- Existují tři hlavní typy anotací – **anotace funkcí**, **invarianty cyklů** a **aserce**

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;
}
```

ANOTACE FUNKCÍ

- Jedná se o dvojici anotací – **precondition** a **postcondition**
- **Precondition** je formule, jejíž volné proměnné zahrnují pouze formální parametry dané funkce
  - Vyjadřuje, co by mělo být splněno při vstupu do funkce
  - … nebo jinými slovy, jaké vstupy funkce očekává
- **Postcondition** je formule, jejíž volné proměnné zahrnují formální parametry a speciální proměnnou rv reprezentující návratovou hodnotu funkce
  - Dává do vztahu návratovou hodnotu rv a vstupní parametry

PŘÍKLAD – LINEÁRNÍ VYHLEDÁVÁNÍ

Specifikace funkce pro (lineární) vyhledávání může neformálně znít:

„Funkce LinearSearch vrací $true$ právě tehdy, když pole a obsahuje hodnotu e v rozmezí indexů $[l, u]$. Funkce se chová korektně, pokud $l \geq 0 \wedge u < |a|$.“

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: true
        (int i := l; i <= u; i := i + 1)
    {
        if (a[i] = e)  return true;
    }
    return false;
}
```

Anotace funkcí

- Někdy se může stát, že netriviální (různá od $true$) precondition není přijatelná
  - Např. pokud se jedná o public funkci nějaké knihovny
- Pak je třeba omezení z precondition nějak přenést do postcondition

Příklad – lineární vyhledávání

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

Predikáty v anotacích

- $sorted(a, i, j)$ – platí, pokud pole a je od indexu i do indexu j včetně uspořádané vzestupně:

$$\forall k, l. i \leq k \leq l \leq j \rightarrow a[k] \leq a[l]$$

- $partitioned(a, i, j, k, l)$ – platí, pokud každý prvek v rozsahu $[i, j]$ je menší nebo roven jakémukoliv prvku z rozsahu $[k, l]$ pole a:

$$\forall x, y. i \leq x \leq j \leq k \leq y \leq l \rightarrow a[x] \leq a[y]$$

Invarianty cyklů

- Pokud funkce neobsahuje (ani transitivně) žádné cykly, je počet průchodů a tedy počet formulí, které je třeba ověřit, poměrně malý
- Běžné programy ale cykly obsahují, proto je nutné se s nimi vypořádat
  - Pomocí invariantů cyklů:

while
@ <invariant>
(<podmínka>) {
<tělo cyklu>
}

for
@ <invariant>
(<init>; <podmínka>; <modifikace>) {
<tělo cyklu>
}

Invarianty cyklů

- For cyklus můžeme přepsat jako while cyklus následujícím způsobem:

```
<init>
while
   @ <invariant>
   (<podmínka>) {
   <tělo cyklu>
   <modifikace>
}
```

- Invariant musí platit před každou iterací cyklu a zároveň po poslední iteraci, tedy po skončení cyklu

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;
}
```

Aserce (assertions)

- V jazyku Pi je možné přidat anotace kamkoliv
- Pokud se nejedná o pre- a postcondition ani o invariant cyklu, nazývá se **aserce** (assertion)
- Jedná se vlastně o formální komentář, který vyjadřuje, co má být splněno, například:

```
@ k > 0
i := i + k
```

- Aserce jsou pak ověřeny při překladu nebo při verifikaci
- Speciální typ asercí jsou **runtime aserce**
  - Dělení nulou, modulo nulou, přístup mimo rozsah pole, dereference null, …
  - V Pi se generují runtime aserce pro první tři typy chyb

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;
}
```

Částečná správnost (Partial correctness)

- Pokud máme funkce opatřené anotacemi, můžeme ověřit jejich správnost
- Říkáme, že funkce je **částečně správná** (partially correct), pokud platí, že 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
- 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:
  - Tělo (každé) funkce je rozloženo na konečný počet **základních cest** (basic paths)
  - Pro každou základní cestu se vygeneruje verifikační podmínka

Základní cesty (Basic paths)

- Cesta je posloupnost příkazů
- Anotace (invarianty cyklů, asserce) a větvení programu rozřezávají možné průchody programem na konečný počet konečných základních cest
  - Od precondition k prvnímu řezu (začátek cyklu, přístup do pole, …)
  - Od řezu k následujícímu řezu
  - Od posledního řezu k postcondition
- Volání funkcí, včetně rekurzivních, rozřezává cesty podle pre- a postconditions
- Nakonec máme konečnou množinu základních cest, z každé z nich se vygeneruje verifikační podmínka

Příklad – Lineární vyhledávání – základní cesty

```
@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;
}
```

1

Příklad – Lineární vyhledávání – základní cesty

2

```
@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 cesty

```
@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;
}
```

3

Příklad – Lineární vyhledávání – základní cesty

4

```
@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 cesty

```
@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;
}
```

5

Verifikační podmínka (Verification condition)

- Z každé základní cesty se vygeneruje jedna verifikační podmínka v podobě implikace
- Verifikátor (SMT řešič) se pokusí dokázat **platnost** této formule, tj. že při jakémkoliv ohodnocení volných proměnných bude hodnota formule (verifikační podmínky) $true$
  - Platnost formule se dokazuje pomocí nesplnitelnosti její negace
  - Splnitelné ohodnocení proměnných znegované formule nám pak dává protipříklad

Příklad – Verifikační podmínka

```
@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;
}
```

5

$$
\begin{gathered}
((l \leq i) \wedge (0 \leq l) \wedge (u < |a|) \wedge \\
(\forall j.((l \leq j \wedge j < i \rightarrow a[j] \neq e))) \\
\rightarrow \\
(\neg(i \leq u) \\
\rightarrow \\
(false \leftrightarrow (\exists j.(l \leq j \wedge j \leq u \wedge a[j] = e))))
\end{gathered}
$$

Verifikace

- Verifikace pak spočívá v dokázání platnosti všech verifikačních podmínek
- V případě neplatnosti verifikační podmínky můžeme prozkoumat protipříklad, podle kterého v některých případech můžeme upravit anotace tak, aby byly dostatečně silné, a verifikační podmínka tak mohla být dokázána

PiVC

- Verifikační nástroj pro programy v Pi
- Uživatelské rozhraní v Javě, připojuje se přes TCP k serveru
  - Pro účely tohoto kurzu běží server na adrese lab.d3s.mff.cuni.cz na portu 4242
  - Upravený klient je dostupný na stránkách předmětu

Příklad – 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 – Binární vyhledávání

```
@pre  0 <= l && u < |a| && sorted(a, 0, |a| - 1)
@post rv <-> exists j. (l <= j && j <= u && a[j] = e)
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 – Bubblesort

```
@pre  true
@post sorted(rv, 0, |rv| - 1)
int[] BubbleSort(int[] a_0) {
  int[] a := a_0;
  for @ true
      (int i := |a| - 1; i > 0; i := i - 1) {
      for @ true


          (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 @ sorted(a, i, |a| - 1) && partitioned(a, 0, i, i+1, |a|-1)
      (int i := |a| - 1; i > 0; i := i - 1) {
      for @ partitioned(a, 0, i, i + 1, |a| - 1) && 1 <= i &&
            0 <= j && j <= i && sorted(a, i, |a| - 1) &&
            partitioned(a, 0, j - 1, j, 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;
}
```

Shrnutí

- Pi je jednoduchý imperativní jazyk, jehož hlavní složkou jsou anotace
- Anotace jsou pre- a postconditions funkcí, invarianty cyklů a aserce uvnitř funkcí
- Základní cesty definují verifikační podmínky, jejichž platnost je předmětem verifikace
- Pomocí PiVC lze dokázat platnost anotací funkcí – zejména postconditions – a ověřit tak částečnou správnost programu

První domácí úloha

Doplňte anotace u funkce `containsZero` tak, abyste zachytili její sémantiku, a ověřte postcondition pomocí PiVC. Termín pro odevzdání je **27. 10. 2025, 23:59**.

```
@pre  true
@post true
bool containsZero(int[] a) {
    for @ true
        (int i := 0; i < |a|; i := i + 1) {
            if (a[i] = 0)
                return true;
    }

    return false;
}
```