NSWI183: Sémantika programů

11. Dafny IV – Funkcionální programování: Seznamy

Jan Kofroň

MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

Department of
Distributed and
Dependable
Systems

Funkcionální programování

- Funkcionální programy mají často nejen elegantní implementaci, ale i vlastnosti, které usnadňují verifikaci jejich vlastností
- Navíc se s podporou funkcionálního programování můžeme setkat i v moderních procedurálních jazycích (Java, C#, Python)
- Ukážeme si, jak psát vnitřní a vnější specifikaci, moduly a invarianty datových struktur

Seznamy (Lists)

- Seznamy jsou velmi běžnou datovou strukturou ve funkcionálních jazycích
- Jejich struktura se napříč jazyky příliš neliší: hlava (první prvek) + tělo (ostatní prvky)
- My budeme používat následující rekurzivní definici (T je typový parametr):

```
datatype List<T> = Nil | Cons(head: T, tail: List<T>)
```

- `Nil` reprezentuje prázdný seznam, `Cons(x, xs)` reprezentuje seznam, jehož první prvek je `x` a pak následují prvky `xs`
- Seznam `(2, 5, 3)` typu `List<nat>` pak vytvoříme pomocí:

```
Cons(2, Cons(5, Cons(3, Nil)))
```

Délka seznamu

- Délku seznamu, tedy počet prvků různých od posledního `Nil`, zjistíme následující funkcí:

```
function Length<T>(xs: List<T>): nat {
  match xs
  case Nil => 0
  case Cons(_, tail) => 1 + Length(tail)
}
```

... nebo případně můžeme tělo zapsat takhle:

```
if xs == Nil then 0 else 1 + Length(xs.tail)
```

Přidávání na konec

- Někdy je potřeba přidávat prvky na konec seznamu, ne na jeho začátek:

```
function Snoc<T>(xs: List<T>, y: T): List<T> {
  match xs
  case Nil => Cons(y, Nil)
  case Cons(x, tail) => Cons(x, Snoc(tail, y))
}
```

Přidávání na konec

- Někdy je potřeba přidávat prvky na konec seznamu, ne na jeho začátek:

```
function Snoc<T>(xs: List<T>, y: T): List<T> {
  match xs
  case Nil => Cons(y, Nil)
  case Cons(x, tail) => Cons(x, Snoc(tail, y))
}
```

- Abychom se (částečně) ujistili, že je naše definice správná, zformulujeme (a dokážeme) lemma:

```
lemma LengthSnoc<T>(xs: List<T>, x: T)
  ensures Length(Snoc(xs, x)) == Length(xs) + 1
{
}
```

Spojování seznamů (Append)

- Velmi podobně jako funkce Snoc pak vypadá spojování dvou seznamů:

```
function Append<T>(xs: List<T>, ys: List<T>): List<T> {
  match xs
  case Nil => ys
  case Cons(x, tail) => Cons(x, Append(tail, ys))
}
```

Spojování seznamů (Append)

- Velmi podobně jako funkce Snoc pak vypadá spojování dvou seznamů:

```
function Append<T>(xs: List<T>, ys: List<T>): List<T> {
  match xs
  case Nil => ys
  case Cons(x, tail) => Cons(x, Append(tail, ys))
}
```

- Vztah mezi délkou a spojováním seznamů můžeme opět zachytit lemmatem:

```
lemma LengthAppend(xs: List<T> ys: List<T>)
  ensures Length(Append(xs, ys)) == Length(xs) + Length(ys)
{
}
```

Porovnatelné typy

- Někdy je potřeba typový parametr nějak omezit na typy, jejichž hodnoty lze porovnávat, protože to není obecná vlastnost každého typu (například u uživatelem definované třídy):

```
function Find<T(==)>(xs: List, y: T): nat
  // Length of the longest prefix without y
  ensures Find(xs, y) <= Length(xs)
{
  match xs
  case Nil => 0
  case Cons(x, tail) =>
  if x == y then 0 else 1 + Find(tail, y)
}
```

Vnější a vnitřní specifikace

- Vyjadřování vlastností pomocí lemmat se nazývá *vnější (extrinsic)* – **vně** funkce
- Alternativou je zachytit vlastnosti uvnitř funkcí pomocí pre- a postconditions, pak se nazývá *vnitřní (intrinsic)* – **uvnitř** funkce

VNĚJŠÍ A VNITŘNÍ SPECIFIKACE

- Vyjadřování vlastností pomocí lemmat se nazývá *vnější (extrinsic)* – **vně** funkce
- Alternativou je zachytit vlastnosti uvnitř funkcí pomocí pre- a postconditions, pak se nazývá *vnitřní (intrinsic)* – **uvnitř** funkce

```
function Append<T>(xs: List<T>, ys: List<T>): List<T>
  ensures Length(Append(xs, ys)) == Length(xs) + Length(ys)
{
  match xs
  case Nil => ys
  case Cons(x, tail) => Cons(x, Append(tail, ys))
}
```

Vnější a vnitřní specifikace

- Metody jsou v Dafny reprezentovány jejich signaturou (jsou *neprůhledné* (*opaque*)), proto specifikujeme jejich vlastnosti vnitřně
- Funkce jsou naopak viditelné včetně jejich implementace (jsou *průhledné* (*transparent*)), můžeme tedy použít oba způsoby

Vnější a vnitřní specifikace

- Metody jsou v Dafny reprezentovány jejich signaturou (jsou *neprůhledné* (*opaque*)), proto specifikujeme jejich vlastnosti vnitřně
- Funkce jsou naopak viditelné včetně jejich implementace (jsou *průhledné* (*transparent*)), můžeme tedy použít oba způsoby
- Vnitřní specifikace se ověří při každém volání automaticky, vnější se musí explicitně „zavolat“
- Automatické ověřování však může být i nevýhoda, protože verifikační podmínky mohou velmi nabobtnat a tím znemožnit verifikaci jako takovou (i kvůli vyčerpání paměti nebo času)
- Vlastností funkcí tedy specifikujeme a ověřujeme pomocí lemmat
  - Některé vlastnosti ani vnitřně specifikovat nelze: `Mirror(Mirror(t)) == t`

Vynechávání typových parametrů (elision)

- Aby zápis signatur funkcí byl jednodušší, Dafny podporuje automatické doplňování typových parametrů, a to s použitím typů z dané funkce (jejího názvu):

```
function ReverseAux<T>(xs: List, acc: List): List
function Length<T>(xs: List): nat
```

- Navíc je možné úplně vypouštět typové parametry, pokud nejsou nikde jinde (v těle funkce, lemmatu, metody) použity:

```
function ReverseAux(xs: List, acc: List): List
function Length(xs: List): nat
```

Shrnutí

- Viděli jsme základní datový typ `List` používaný ve funkcionálním programování v různých jazycích
- Dokázali jsme některé jeho vlastnosti a viděli, že je můžeme specifikovat vnitřně – pomocí postcondition – nebo jako vnější specifikaci – pomocí lemmat
- Viděli jsme efektivní funkcionální implementaci na otočení seznamu a dokázali jeho ekvivalenci s naivní implementací