NSWI183: Sémantika programů

8. Dafny II.

Jan Kofroň

MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

Department of
Distributed and
Dependable
Systems

DAFNY

- Viděli jsme základní koncepty a abstrakce programovacího jazyka DAFNY
- Dnes se seznámíme s dalšími technikami pro specifikaci v tomto prostředí
  - zejména s ohledem na datové struktury

DATOVÉ STRUKTURY

- DAFNY je objektový jazyk, proto můžeme pro definici vlastních datových struktur použít třídy tak, jak jsme zvyklí z běžných programovacích jazyků
- DAFNY má vlastní způsob definice datových typů, které jsou vhodnější pro dokazování jejich vlastností – *datatypes*
- Uvažme binární strom, jehož *listy* jsou buď žluté nebo modré, definice pak bude vypadat takhle:

```
datatype BYTree = BlueLeaf | YellowLeaf | Node(BYTree, BYTree)
```

- Tímto definujeme datový typ **BYTree** se třemi konstruktory instancí
- Strom pak můžeme vytvořit například takhle:

```
var tree := Node(BlueLeaf, Node(YellowLeaf, BlueLeaf))
```

Funkce nad daty

- Pokud chceme definovat nad takovými datovými strukturami nějaké funkce, hodí se nám výraz `match`:

```
function BlueCount(t: BYTree): nat {
  match t
  case BlueLeaf => 1
  case YellowLeaf => 0
  case Node(left, right) => BlueCount(left) + BlueCount(right)
}
```

Funkce nad daty II.

- Pro otestování některých vlastností můžeme použít funkce s návratovým typem `bool`, nebo `predicate`:

```
predicate IsLeaf(t: BYTree) {
  match t
  case BlueLeaf => true
  case YellowLeaf => true
  case Node(left, right) => false
}
```

- Protože testování případů je poměrně častá operace, obsahuje Dafny přímo konstrukt složený z názvu konstruktoru a otazníku pro každý datový typ a každý jeho kontstruktor, např.: **`BlueLeaf?`**, **`YellowLeaf?`** a **`Node?`**

Strukturální indukce

- Definicí induktivních datových typů implicitně definujeme dobře fundovanou relaci $\succ$ dávající do vztahu jeho součásti
- Pro náš případ platí:
  - `Node(left, right)` $\succ$ `left`
  - `Node(left, right)` $\succ$ `right`
- Dafny pak použije tuto relaci ke specifikaci fundované funkce k ověření konečnosti výpočtu
- ... proto u predikátu **`isLeaf`** nepotřebujeme specifikovat fundovanou funkci explicitně

VÝČTY – ENUMERATIONS

- Pokud definujeme datový typ, který má jen bezparametrické konstruktory, říkáme mu **enumeration**:

```
datatype Color = Red | Green | White | Blue | Yellow
```

- Takový typ pak můžeme používat úplně stejně jako standardní datový typ, například:

```
predicate isCzechFlagColor(c: Color) {
  c.Blue? || c.Red? || c.White?
}
```

- ... nebo použít ke zobecnění našeho modrožlutého stromu:

```
datatype ColoredTree = Leaf(Color)
                     | Node(ColoredTree, ColoredTree)
```

Generické typy

- Můžeme ještě dále zobecnit náš binární strom, a to pomocí *typových parametrů*:

```
datatype Tree<T> = Leaf(data: T)
                 | Node(left: Tree<T>, right: Tree<T>)
```

- Takový datový typ se pak nazývá *generický* (*generic type*), což odpovídá tomu, co známe z jiných jazyků
- Funkce pak můžeme vytvářet pro konkrétní typy (např. `Tree<Color>`) nebo obecně pro jakýkoliv strom (`Tree<T>`):

```
function Size<T>(t: Tree<T>): nat {
  match t
  case Leaf(_) => 1
  case Node(left, right) => Size<T>(left) + Size<T>(right)
}
```

PŘÍKLAD – AST

```
datatype Expr = Const(int)
              | Var(string)
              | Node(op: Op, args : List<Expr>)

datatype Op = Add | Mul | Sub | Div

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