NSWI183: Sémantika programů

12. Dafny V – Abstrakce, Třídění

Jan Kofroň

MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

Department of
Distributed and
Dependable
Systems

Abstrakce

Abstrakce – Moduly

- Abstrakce je klíčová pro dobrý návrh programů
- V praxi to znamená vytváření objektů na vyšší úrovni abstrakce: funkce, metody, třídy, moduly
- V Dafny existuje na nejvyšší úrovni koncept **modulů**, které lze importovat mezi zdrojovými soubory:

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

  function Append(xs: List, ys: List): List
  // ....

  lemma ....
}
```

IMPORT

- Pokud chceme použít kód z jiného modulu – a to i v případě, že se jedná o stejný soubor –, musíme modul explicitně importovat:

```
module Sorting {
  import ListLibrary

  function Sort<T>(xs: ListLibrary.List): ListLibrary.List {
    // using lists here as e.g., ListLibrary.Nil
  }
}
```

Import

- Název modulu může být neprakticky dlouhý, můžeme ho importovat pod vlastním jménem:

```
module Sorting {
  import LL = ListLibrary
  function Sort<T>(xs: LL.List): LL.List {
    // using lists here as e.g., LL.Nil
  }
}
```

- Nebo můžeme použít klíčové slovo `opened` a pak používat identifikátory bez prefixu:

```
module Sorting {
  import opened ListLibrary
  var list := Nil
}
```

Import–Export

- Importy nesmějí být cyklické, tvoří tedy hierarchii modulů
- Moduly nejen že rozdělují jmenný prostor programu, ale umožňují i skrývání informací (o implementaci), vytvářejíce tak rozhraní (interface)
- To samozřejmě dává volnost implementaci, která se může v čase měnit bez porušení kontraktů, které má modul se svými klienty (moduly, které tento modul používají)
- V Dafny se to, co modul „vystavuje“ navenek, specifikuje pomocí deklarace `export`
- Exportovat typ (datový typ, funkce, lemma, metoda) pak může modul pomocí `provides` (pouze signatura), nebo `reveals` (signatura i tělo)

Import–Export

- Předpokládejme následující specifikaci modulu:

```
module ModuleC {
  export
    provides Color
    reveals Double

  datatype Color = Blue | Yellow | Green | Red
  function Double(x: int): nat
    requires 0 <= x
    ensures Double(x) % 2 == 0
  {
    if x == 0 then 0 else 2 + Double(x - 1)
  }
}
```

Import–Export

- Předpokládejme následující specifikaci modulu:

```
module ModuleC {
  export
    provides Color
    reveals Double

  datatype Color = Blue | Yellow | Green | Red
  function Double(x: int): nat
    requires 0 <= x
    ensures Double(x) % 2 == 0
  {
    if x == 0 then 0 else 2 + Double(x - 1)
  }
}
```

IMPORT–EXPORT

- Modul pak importujeme následovně:

```
module ModuleD {
  import MC = ModuleC

  method Test() {
    var c: MC.Color;
    c := MC.Yellow;
    assert MC.Double(3) == 6;
  }
}
```

- Pokud se modul nachází v jiném souboru, musíme tento soubor „inkludovat“ pomocí:

```
include "filename.dfy"
```

IMPORT–EXPORT

- Modul pak importujeme následovně:

```
module ModuleD {
  import MC = ModuleC

  method Test() {
    var c: MC.Color; // OK protože Color je provided
    c := MC.Yellow;
    assert MC.Double(3) == 6;
  }
}
```

- Pokud se modul nachází v jiném souboru, musíme tento soubor „inkludovat“ pomocí:

```
include "filename.dfy"
```

Import–Export

- Modul pak importujeme následovně:

```
module ModuleD {
  import MC = ModuleC

  method Test() {
    var c: MC.Color; // OK protože Color je provided
    c := MC.Yellow; // chyba: unresolved identifier 'Yellow'
    assert MC.Double(3) == 6;
  }
}
```

- Pokud se modul nachází v jiném souboru, musíme tento soubor „inkludovat“ pomocí:

```
include "filename.dfy"
```

IMPORT–EXPORT

- Modul pak importujeme následovně:

```
module ModuleD {
  import MC = ModuleC

  method Test() {
    var c: MC.Color; // OK protože Color je provided
    c := MC.Yellow; // chyba: unresolved identifier 'Yellow'
    assert MC.Double(3) == 6; // verifikuje se, Double je revealed
  }
}
```

- Pokud se modul nachází v jiném souboru, musíme tento soubor „inkludovat“ pomocí:

```
include "filename.dfy"
```

Konzistence exportů

- Exporty musejí být konzistentní – když smažeme části, které nejsou exportované, program musí být stále typově správný
- Předpokládejme následující definici modulu:

```
module ModuleC {
  datatype Parity = Even | Odd
  function F(x: int): Parity
  {
    if x % 2 == 0 then Even else Odd
  }
}
```

KONZISTENCE EXPORTŮ

```
module ModuleC {
   datatype Parity = Even | Odd
   function F(x: int): Parity
   {
     if x % 2 == 0 then Even else Odd
   }
}
```

Nekonzistentní exporty

Konzistentní exporty

KONZISTENCE EXPORTŮ

```
module ModuleC {
   datatype Parity = Even | Odd
   function F(x: int): Parity
   {
     if x % 2 == 0 then Even else Odd
   }
}
```

Nekonzistentní exporty

```
export provides F
```

Konzistentní exporty

KONZISTENCE EXPORTŮ

```
module ModuleC {
    datatype Parity = Even | Odd
    function F(x: int): Parity
    {
      if x % 2 == 0 then Even else Odd
    }
}
```

Nekonzistentní exporty

```
export provides F
export provides Parity
       reveals F
```

Konzistentní exporty

KONZISTENCE EXPORTŮ

```
module ModuleC {
   datatype Parity = Even | Odd
   function F(x: int): Parity
   {
     if x % 2 == 0 then Even else Odd
   }
}
```

Nekonzistentní exporty

```
export provides F
export provides Parity
       reveals F
```

Konzistentní exporty

```
export provides Parity, F
```

KONZISTENCE EXPORTŮ

```
module ModuleC {
    datatype Parity = Even | Odd
    function F(x: int): Parity
    {
      if x % 2 == 0 then Even else Odd
    }
}
```

Nekonzistentní exporty

```
export provides F
export provides Parity
       reveals F
```

Konzistentní exporty

```
export provides Parity, F
export reveals Parity, F
export reveals *
```

Třídění

Třídění

- Přirozenou operací nad seznamy je setřídění prvků (u čísel podle velikosti)
- Ukážeme si, jak postupovat při specifikaci a implementaci takové operace
- Specifikace označíme jako `ghost`, aby bylo jasné, že se jedná o nekompilovaný kód použitý jen pro ověření vlastností

- Definice seznamu (pro připomenutí):

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

Predikát Ordered

- K setřídění (a ověření setříděnosti) seznamů potřebujeme tentokrát víc, než jen schopnost porovnat (na rovnost) jednotlivé prvky seznamu – potřebujeme testovat, který je větší
- Proto se omezíme na typ `int`:

```
ghost predicate Ordered(xs: List<int>) {
  match xs
  case Nil => true
  case Cons(x, Nil) => true
  case Cons(x, Cons(y, _)) => x <= y && Ordered(xs.tail)
}
```

Ověření predikátu Ordered

- Pokud máme složitější predikát nebo obecně nějakou definici, je dobré ověřit její vlastnosti, které očekáváme, že má
- V případě predikátu `Ordered` to může být například následující lemma:

```
lemma AllOrdered(xs: List<int>, i: nat, j: nat)
  requires Ordered(xs) && i <= j < Length(xs)
  ensures At(xs, i) <= At(xs, j)
{
  ...
}
```

PERMUTACE

- Ne každá funkce, která vrací setříděný seznam, je třídící funkce
  - například funkce vždy vracející prázdný nebo jednoprvkový seznam
- Další důležitou vlastností je, že návratová hodnota je permutací vstupu
  - přímé vyjádření a dokázání permutace jsou ale složité – to jsme už viděli v PiVC
  - využijeme princip projekce

Permutace

```
ghost function Count(xs: List<int>, p: int): nat {
  match xs
  case Nil => 0
  case Cons(x, tail) =>
    if x == p then (1 + Count(tail, p)) else Count(tail, p)
}

ghost function Project(xs: List<int>, p: int): List<int> {
  match xs
  case Nil => Nil
  case Cons(x, tail) => if x == p then Cons(x, Project(tail, p))
                                  else Project(tail, p)
}
```

Permutace

```
ghost function Count(xs: List<int>, p: int): nat {
  match xs
  case Nil => 0
  case Cons(x, tail) =>
    if x == p then (1 + Count(tail, p)) else Count(tail, p)
}

ghost function Project(xs: List<int>, p: int): List<int> {
  match xs
  case Nil => Nil
  case Cons(x, tail) => if x == p then Cons(x, Project(tail, p))
                                  else Project(tail, p)
}

lemma SortingCorrectness(xs: List<int>, p: int)
  ensures Project(xs, p) == Project(WhateverSort(xs), p)
```

Insertion Sort

- Třídění vkládáním (insertion sort) je založeno na postupném zatřiďování jednotlivých prvků do už setříděného seznamu, jeho složitost je $\mathcal{O}(n^2)$
- Funkcionální implementace je přímočará:

Insertion Sort

- Třídění vkládáním (insertion sort) je založeno na postupném zatřiďování jednotlivých prvků do už setříděného seznamu, jeho složitost je $\mathcal{O}(n^2)$
- Funkcionální implementace je přímočará:

```
function InsertionSort(xs: List<int>): List<int> {
  match xs
  case Nil => Nil
  case Cons(x, tail) => Insert(x, InsertionSort(tail))
}

function Insert(y: int, xs: List<int>): List<int> {
  match xs
  case Nil => Cons(y, Nil)
  case Cons(x, tail) => if y < x then Cons(y, xs)
                                 else Cons(x, Insert(y, tail))
}
```

Ověření Insertion Sort

- Nyní využijeme dříve definovaných predikátů a funkcí k ověření správnosti insertion sortu:

```
lemma InsertionSortOrdered(xs: List<int>)
  ensures Ordered(InsertionSort(xs))

lemma InsertOrdered(y: int, xs: List<int>)
  requires Ordered(xs)
  ensures Ordered(Insert(y, xs))

lemma InsertionSortSameElements(xs: List<int>, p: int)
  ensures Project(xs, p) == Project(InsertionSort(xs), p)

lemma InsertSameElements(y: int, xs: List<int>, p: int)
  ensures Project(Cons(y, xs), p) == Project(Insert(y, xs), p)
```

SHRNUTÍ

- Viděli jsme, jak je abstrakce pro členění a zapouzdření kódu realizovaná v DAFNY pomocí modulů
- Specifikovali, implementovali a ověřili jsme správnost třídícího algoritmu InsertionSort ve funkcionálním prostředí