Verifikáció a Why3 tételbizonyító környezetben

Kabódi László

2015. december 9.

VerifyThis 2012 verseny

- Verseny programok helyességének bizonyítására
- A verseny feladatai
  1. Leghosszabb közös prefix
  2. Prefix összeg
  3. Bináris keresőfában iteratív törlés
- Az első két feladathoz adott Java implementáció
- A harmadikhoz pszeudokód

A Why3 áttekintése

- A Why3 egy elsőrendű logikán alapuló helyességbizonyító rendszer
- Kétféle nyelvet használ
  - Specifikációt leíró nyelv: elsőrendű logika, ML alapokon
  - Programot leíró nyelv: WhyML, ML alapokon
- A helyességet több automatikus bizonyítóval ellenőrzi
- Interaktív bizonyítók is használhatók

Specifikációs nyelv

- Alapvető típusok elérhetőek, például `int`, `real`, `list`, `tree`
- Függvények és predikátumok lehetnek rekurzívok.
  - Automatikus terminálódás ellenőrzés
  - Csak algebrai típusokra
- Induktív predikátumokat kezel
- Termek és formulák: elsőrendű logikát kiegészíti mintafelismeréssel és feltételes kifejezésekkel.
- `Theorems`: tisztán logikai definíciók, axiómák és lemmák gyűjteménye.

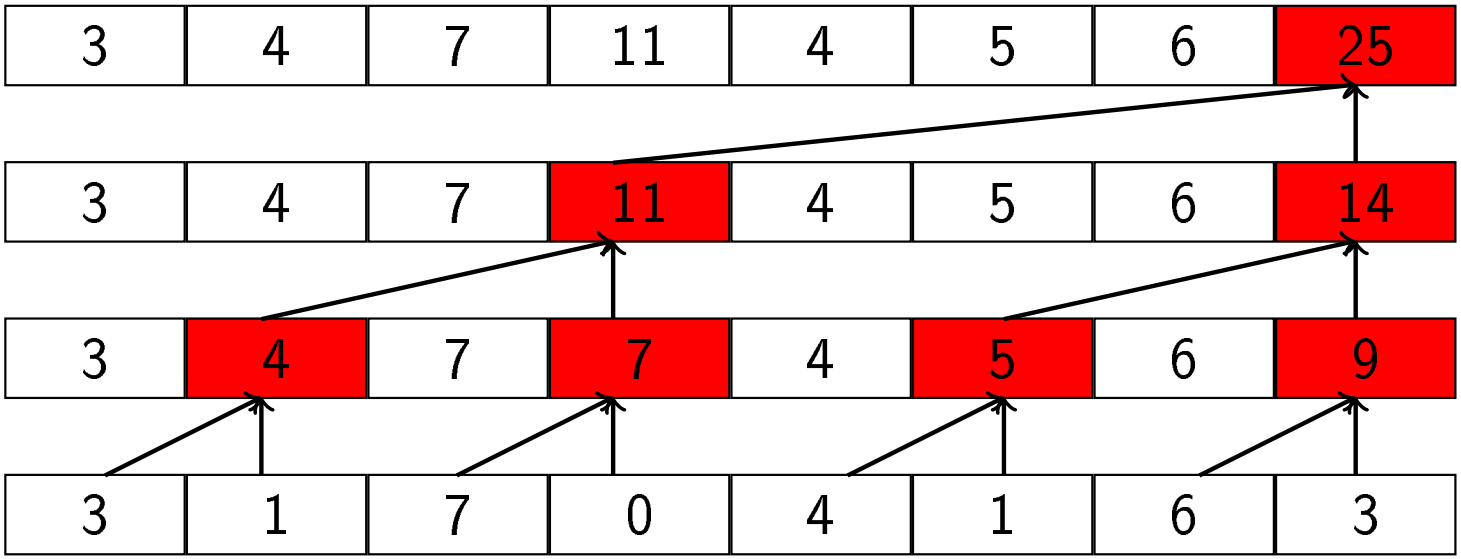
Programozási nyelv

- A WhyML az ML egy verifikációt elősegítő elemekkel kiegészített változata
  - Ciklusokban megadhatók invariánsok
  - Rekurzióban megadhatók olyan változók, amik folyamatosan csökkennek
  - Típusok
    - A változtatható elemek `mutable` jelzéssel
    - Típusinvariáns megadható
    - Úgynevezett `ghost` változók, verifikációs céllal
- Modulokat is kezel

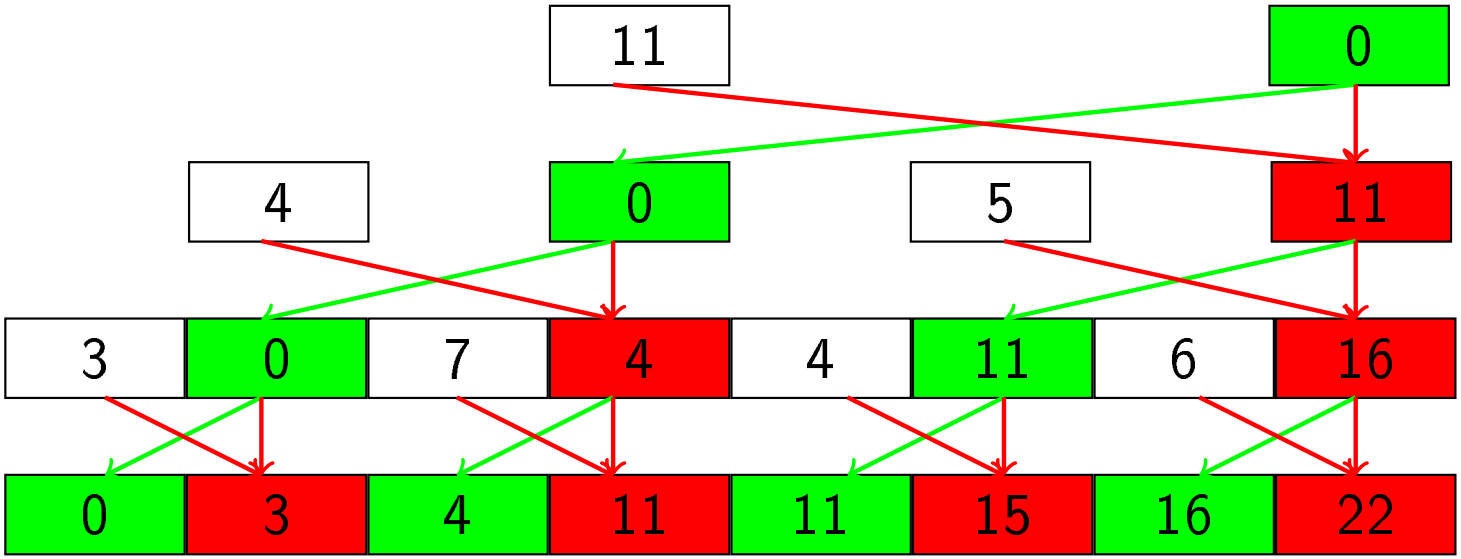
Algoritmus

- Adott egy kettőhatvány méterű tömb. A feladat ebben minden elemre kiszámítani a prefixösszeget: $a[i] = \sum_{j=0}^{i-1} a_j$
- A tömb elemei kezdetben egy teljes bináris fa levelei
- A megadott algoritmus két részből áll
  1. „Upsweep”: minden csomópontba a részfájában tárolt elemek összege kerül. Ezt a tömb megfelelő elemeinek felülírásával tároljuk.
  2. „Downsweep”: Az utolsó elem lesz a fa gyökere, nullára cseréljük. Ez után minden csomópont minden szinten:
     - A bal gyereknek a saját értékét adja
     - A jobb gyereknek az „Upsweep” rész szerinti bal szomszédja és a saját értékének összegét adja

„Upsweep”



„Downsweep”



Specifikáció

- A tömb kettőhatvány méretű

```
predicate is_power_of_2 (x : int) =
  exists k:int. (k ≥ 0 ∧ x = power 2 k)
```

- Az eredmény helyes

```
val compute_sums (a : array int)
  requires { a.length ≥ 2 ∧
    is_power_of_2 a.length }
  writes   { a }
  ensures  { forall i:int. 0 ≤ i < a.length →
              a[i] = sum (old a) 0 i }
```

A bináris fa adatszerkezet - ötlet

- Induktív predikátum, ami jól modellezi a rekurzív algoritmust
- Két index, „`left`” és „`right`”
- Segédváltozó, melynek mérete kettőhatvány (konkrét változóként csak később fordul elő): `space = right - left`
- A tömbben a vizsgált részfa elemei `left - space + 1` és `right` között helyezkednek el.
- A predikátum két tömböt használ, a módosítottat (`a`) és az eredetit (`a0`)
- A predikátum `a[left]` értékeit adja meg. Mivel induktív, `right-1` az utolsó.

A bináris fa adatszerkezet - kód

```
inductive sumtree (left right : int)
                  (a0 a : array int) =
  | Leaf : forall left right : int, a0 a : array int.
      right = left + 1 → a[left] = a0[left] →
      sumtree left right a0 a
  | Node : forall left right : int, a0 a : array int.
      right > left + 1 →
      sumtree (go_left left right) left a0 a →
      sumtree (go_right left right) right a0 a →
      a[left] =
        sum a0 (left-(right-left)+1) (left+1) →
      sumtree left right a0 a
```

A bináris fa adatszerkezet - bejárás

- A bal részfa: `(go_left left right) left`

```
function go_left (left right:int) : int =
  let space = right - left in left - div space 2
```

- A jobb részfa: `(go_right left right) right`

```
function go_right (left right:int) : int =
  let space = right - left in right - div space 2
```

Az „upsweep” fázis - ötlet

- A `(left, right)` részfára meghívva legyen `a[right]` a részfa elemeinek összege.
- A `sumtree` az `a[left]`-et pont erre állította, miután a bal részfa elemein végigment.
- Az `a[right]` elemet be kell állítani.
- A `(left, right)` részfán kívüli elemek ne változzanak.

Az „upsweep” fázis - kód

```
let rec upsweep (left right : int) (a : array int)
  requires { 0 ≤ left < right < a.length ∧
    -1 ≤ left - (right - left) ∧
    is_power_of_2 (right - left) }
  variant { right - left }
  ensures { let space = right - left in
    sumtree left right (old a) a ∧
    a[right] = sum (old a) (left-space+1) (right+1) ∧
    (forall i: int. i ≤ left-space →
                                  a[i] = (old a)[i]) ∧
    (forall i: int. i > right → a[i] = (old a)[i]) }
```

A „downsweep” fázis - ötlet

- Mivel **`old`** a a „downsweep” és „upsweep” közötti állapotot tárolja, kell egy szellem változó, `a0`.
- Az `a[right]` értéke a részfa bal szomszédjainak összege (`a[right] = sum a0 0 (left - space + 1)`, fehér mezőből piros nyíl).
- Végeredményként biztosítania kell a prefix összeg teljesülését az egész `(left right)` részfára, ehhez kell egy segédpredikátum (`partial_sum left right a0 a`).

A „downsweep” fázis - kód I.

```
predicate partial_sum (left right : int)
                      (a0 a : array int) =
  forall i : int. (left-(right-left)) < i ≤ right →
    a[i] = sum a0 0 i
```

A „downsweep” fázis - kód II.

```
let rec downsweep (left right : int)
    (ghost a0 : array int) (a : array int)
  requires { 0 ≤ left < right < a.length ∧
    -1 ≤ left - (right - left) ∧
    is_power_of_2 (right - left) ∧
    a[right] = sum a0 0 (left - (right - left) + 1) ∧
    sumtree left right a0 a }
  variant { right - left }
  ensures { partial_sum left right a0 a ∧
    (forall i: int. i ≤ left - (right - left) →
                                  a[i] = (old a)[i]) ∧
    (forall i: int. i > right → a[i] = (old a)[i]) }
```

Teljes program

```
let compute_sums (a : array int)
  requires { a.length ≥ 2 ∧
    is_power_of_2 a.length }
  ensures { forall i : int. 0 ≤ i < a.length →
                           a[i] = sum (old a) 0 i }
  = let a0 = ghost (copy a) in
  let l = a.length in
  let left = div l 2 - 1 in
  let right = l - 1 in
  upsweep left right a;
  downsweep left right a0 a;
```

Tanulságok

- A bizonyítások nagy része automatikusan sikerült
- De csak `Assert`-ek segítségével
- Két keretlemmához interaktív bizonyító is szükséges volt
- A legnagyobb probléma a specifikáció hibás felírása volt, az indexeket elrontották
- Specifikáció lépésenkénti ellenőrzése hasznos plusz funkció lenne