Karlsruher Institut für Technologie

Institut für Programmstrukturen und Datenorganisation
Lehrstuhl Programmierparadigmen
Am Fasanengarten 5
76131 Karlsruhe
http://pp.ipd.kit.edu/

Theorembeweiserpraktikum – SS 2016

http://pp.ipd.kit.edu/lehre/SS2016/tba

Lösung 6: Große natürliche Zahlen

Abgabe: 30. Mai 2016, 12:00 Uhr
Besprechung: 31. Mai 2016

Hardware-Plattformen haben ein Limit, welches die größte darstellbare Zahl ist; dies ist normalerweise durch die Bitlänge der verwendeten Register und ALU festgelegt. Um mit beliebig großen Zahlen rechnen zu können, müssen die entsprechenden arithmetischen Operationen erweitert werden, um auf abstrakten Datentypen, welche Zahlen beliebiger Größe repräsentieren, arbeiten zu können.
Wir werden im folgenden eine Implementation für BigNat, einen abstrakten Datentypen zur Darstellung von natürlichen Zahlen beliebiger Größe, erstellen und verifizieren.

Darstellung

Ein BigNat wird als Liste von natürlichen Zahlen innerhalb eines von der Zielmaschine unterstützten Bereichs dargestellt. In unserem Fall sind das alle natürlichen Zahlen im Bereich [0, Basis-1] (den Sonderfall, dass die Basis 1 ist, können wir für diese Aufgabe ignorieren). In Isabelle selbst sind natürliche Zahlen von beliebiger Größe.

```
type_synonym bigNat = "nat list"
```

Definieren Sie jetzt zwei Funktionen: `valid`, welche einen Wert für die Basis nimmt und prüft, ob der gegebene BigNat dafür gültig ist, und `val`, welches mittels eines BigNats und seiner Basis die entsprechend dargestellte Zahl zurückgibt. Beachten Sie: wenn die Basis nicht größer als 1 ist, darf die Zahl auch nicht in dieser Basis gültig sein.

```
fun valid :: "nat ⇒ bigNat ⇒ bool"
where "valid b [] ⟷ b > 1"
    | "valid b (n#ns) ⟷ n < b ∧ valid b ns"

fun val :: "nat ⇒ bigNat ⇒ nat"
  where "val b [] = 0"
      | "val b (n#ns) = n + b * val b ns"
```

Addition

Definieren Sie jetzt eine Funktion `add`, welche zwei BigNats mit der selben Basis addiert. Stellen Sie sicher, dass ihr Algorihmus die Gültigkeit der BigNat-Darstellung beibehält. Beweisen Sie danach mittels `val` und `valid`, dass der Algorithmus das korrekte Resultat berechnet und die Gültigkeit der Darstellung beibehält.

```
definition add :: "nat ⇒ bigNat ⇒ bigNat ⇒ bigNat"
  where "add b ns ms = ...."
```

Wir führen eine Hilfsfunktion mit einem weiteren Parameter für den Übertrag ein.

```
fun add' :: "nat ⇒ nat ⇒ bigNat ⇒ bigNat ⇒ bigNat"
  where "add' b c [] []         = [c]"
      | "add' b c (n#ns) []     = (if n + c ≥ b
                                   then (n + c - b) # add' b 1 ns []
                                   else (n + c) # add' b 0 ns [])"
      | "add' b c []     (m#ms) = (if m + c ≥ b
                                   then (m + c - b) # add' b 1 [] ms
                                   else (m + c) # add' b 0 [] ms)"
      | "add' b c (n#ns) (m#ms) = (if n + m + c ≥ b
                                    then (n + m + c - b) # add' b 1 ns ms
                                    else (n + m + c) # add' b 0 ns ms)"

definition add :: "nat ⇒ bigNat ⇒ bigNat ⇒ bigNat"
  where "add b ns ms = add' b 0 ns ms"

lemma add'_valid: "⟦valid d ns; valid d ms; c ∈ {0,1}⟧ ⟹ valid d (add' d c ns ms)"
by (induction d c ns ms rule:add'.induct) auto

lemma add_valid: "⟦valid d ns; valid d ms⟧ ⟹ valid d (add d ns ms)"
  by (auto intro: add'_valid simp add: add_def)
```

Abhängig von ihren Definitionen brauchen Sie in folgendem Lemma möglicherweise die Voraussetzungen `valid d ns` und `valid d ms`

```
lemma add'_val: "val d (add' d c ns ms) = c + val d ns + val d ms"
by (induction d c ns ms rule:add'.induct)(auto simp add: add_mult_distrib2)

lemma add_val: "val d (add d ns ms) = val d ns + val d ms"
by (simp add: add'_val add: add_def)
```

Multiplikation

Jetzt sollen Sie analog zur Addition auch noch die Multiplikation definieren und die entsprechenden Lemmas für `valid` und `val` zeigen. Sie können sich an der normalen Papiermultiplikation, die Sie in der Schule gelernt haben, orientieren. Vergessen Sie dabei aber das Shiften um eine Stelle nicht!

```
fun mult :: "nat ⇒ bigNat ⇒ bigNat ⇒ bigNat"
  where "mult b ns ms = ...."
fun mult_one :: "nat ⇒ nat ⇒ nat => bigNat ⇒ bigNat"
  where "mult_one b n c [] = [c]"
    | "mult_one b n c (m#ms) = ((n * m + c) mod b) # mult_one b n ((n * m + c) div b) ms"

lemma less_sqr_div_less:
  assumes "m < d * d"
  shows "m div d < (d::nat)"
proof(rule ccontr)
  assume "¬ m div d < d"
  then have "d ≤ m div d" by simp
  then have "d * d ≤ d * (m div d)" by simp
  also have "... ≤ m" by (simp add: mult_div_cancel)
```

```
  also note ‹... < d * d›
  finally
  show False by simp
qed

lemma mult_one_valid: "⟦valid b ns; valid b ms; n < b ; c < b⟧ ⟹ valid b (mult_one b
n c ms)"
proof (induction b n c ms rule:mult_one.induct)
  case 1 then show ?case by simp
next
  case (2 b n c m ms)
    from ‹valid b (m # ms)› have "m < b" by (cases rule: valid.cases, auto)
    then have "n * m + c ≤ n * b + c" by (simp add: add_le_mono mult_le_mono)
    also have "... < n * b + b" using ‹c < b› by simp
    also have "... = (Suc n) * b" by simp
    also have "... ≤ b * b" using ‹n < b› by (simp add: mult_le_mono del: mult_Suc)
    finally
    have "(n * m + c) div b < b" by (rule less_sqr_div_less)
    with 2
    show ?case by (auto intro: add_valid)
qed

fun mult :: "nat ⇒ bigNat ⇒ bigNat ⇒ bigNat"
where "mult b ns []= []"
    | "mult b ns (m#ms) = add b (mult_one b m 0 ns) (0#mult b ns ms)"

lemma mult_valid: "⟦valid b ns; valid b ms⟧ ⟹ valid b (mult b ns ms)"
by (induction b ns ms rule:mult.induct)(auto intro!: add_valid mult_one_valid)
```

Auch hier könnte wieder die Forderung nach gültigen Zahlen nötig sein

```
lemma mult_one_val: "val b (mult_one b m c ns) = m * val b ns + c"
by (induction b m c ns rule:mult_one.induct)(auto simp add: add_val add_mult_distrib2)

lemma mult_val: "val b (mult b ns ms) = val b ns * val b ms"
by (induction b ns ms rule:mult.induct)(auto simp add: add_val mult_one_val add_mult_distrib2)
```

Wenn man sich etwas von der Schulmathematik entfernt kann man auch eine Definition finden, die ohne Hilfsfunktionen auskommt und mit der Isabelle besser zurecht kommt:

```
fun mult' :: "nat ⇒ bigNat ⇒ bigNat ⇒ bigNat"
where "mult' b [] ms = []"
    | "mult' b (0#ns) ms = 0 # (mult' b ns ms)"
    | "mult' b ((Suc n)#ns) ms = add b ms (mult' b (n#ns) ms)"

lemma mult'_valid: "⟦valid b ns; valid b ms⟧ ⟹ valid b (mult' b ns ms)"
by (induction b ns ms rule:mult'.induct)(auto intro: add_valid)

lemma mult'_val: "val b (mult' b ns ms) = val b ns * val b ms"
by (induction b ns ms rule:mult'.induct)(auto simp add: add_val)
```

Hinweise

Machen sie sich Gedanken, ob Sie die Zahldarstellung in „big endian“ oder „little endian“ machen möchten, mit einer der beiden ist deutlich angenehmer zu arbeiten als mit der anderen.
Auch hier gilt: Je eleganter die Definition, desto kürzer der Beweis. Versuchen Sie ihre Beweise, sofern keine Einzeiler, in einem möglichst klaren Isar-Skript darzustellen.
Sie können, falls Sie das benötigen, für ihre Definitionen die Funktionen `div` und `mod` verwenden, die sich in Isabelle wie erwartet verhalten; z.B. wird die Aussage `d ≤ b ⟶ b mod d + d * (b div d) = b` für natürliche Zahlen durch einfache Anwendung des Simplifiers gelöst. Andere Aussagen, wie z.B. `m < d * d ⟹ m div d < d`, werden Sie eventuell als Hilfslemma selbst zeigen müssen.
An einigen Stellen werden Sie dem Simplifier mit einem der folgenden Lemmas auf die Sprünge helfen müssen:

```
add_mult_distrib: (m + n) * k = m * k + n * k
add_mult_distrib2: k * (m + n) = k * m + k * n
```