



# Theorembeweiserpraktikum – SS 2018

<http://pp.ipd.kit.edu/lehre/SS2018/tba>

## Blatt 6: Große natürliche Zahlen

Abgabe: 4. Juni 2018, 12:00 Uhr  
Besprechung: 5. Juni 2018

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  $[0, \text{Basis}-1]$  (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 `BigNat`s und seiner Basis die entsprechend dargestellte Zahl zurückgibt.

```
fun valid :: "nat  $\Rightarrow$  bigNat  $\Rightarrow$  bool"  
  where "valid b ns  $\longleftrightarrow$  ...."
```

```
fun val :: "nat  $\Rightarrow$  bigNat  $\Rightarrow$  nat"  
  where "val b ns = ...."
```

## Addition

Definieren Sie jetzt eine Funktion `add`, welche zwei `BigNat`s mit der selben Basis addiert. Stellen Sie sicher, dass ihr Algorithmus 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  $\Rightarrow$  bigNat  $\Rightarrow$  bigNat  $\Rightarrow$  bigNat"  
  where "add b ns ms = ...."
```

```
lemma add_valid: "[[valid d ns; valid d ms]]  $\implies$  valid d (add d ns ms)"  
  <solution>
```

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 ns ms) = val d ns + val d ms"  
<solution>
```

## 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  $\Rightarrow$  bigNat  $\Rightarrow$  bigNat  $\Rightarrow$  bigNat"  
  where "mult b ns ms = ...."
```

```
lemma mult_valid: "[[valid b ns; valid b ms]]  $\implies$  valid b (mult b ns ms)"  
<solution>
```

Auch hier könnte wieder die Forderung nach gültigen Zahlen nötig sein

```
lemma mult_val: "val b (mult b ns ms) = val b ns * val b ms"  
<solution>
```

## 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  $d \leq b \implies b \text{ mod } d + d * (b \text{ div } d) = b$  für natürliche Zahlen durch einfache Anwendung des Simplifiers gelöst. Andere Aussagen, wie z.B.  $m < d * d \implies m \text{ div } d < d$ , 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
```