Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Florian Haftmann

SS 2009
25. 06. 2009

# Perlen der Informatik 2

**7. Übung**

## 1 Unbounded natural numbers

Hardware platforms have a limit on the largest number they can represent. This is normally fixed by the bit lengths of registers and ALUs used. In order to be able to perform calculations that require arbitrarily large numbers, the provided arithmetic operations need to be extended in order for them to work on an abstract data type representing numbers of arbitrary size.

In this exercise we will build and verify an implementation for such *big_nat*s, a type representing natural numbers of arbitrary size.

### 1.1 Representation

A *big_nat* is represented as a list of natural numbers (cells) in a range supported by the target machine, in ascending significance. In our case, this will be all natural numbers in the range *[0, base - 1]*.

Note that *nat*s in Isabelle have arbitrary size.

First, we give such *big_nat*s a semantic by means of a predicate *valid* :: *nat* ⇒ *nat list* ⇒ *bool* which takes a *base* and checks if the given *big_nat* is valid.

The next step is to specify an evaluation function *val* :: *nat* ⇒ *nat list* ⇒ *nat* that, for a given *base*, computes the actual natural number represented by a specific *big_nat*.

Computations will usually involve a *carry* from a cell to a cell with higher significance. These can easily be expressed using operations *div d* and *mod d*, where *d* again is the *base*. For this purpose we will need some auxiliary lemmas which are, alas, tricky to prove. You need not type these yourself, they are available from the website of this course:

```
lemma plus_div_less_self:
  fixes a b c :: nat
  assumes "a < c" and "b < c"
  shows "(a + b) div c < c"
proof (cases "2 ≤ c")
  case True
  from assms have "a + b ≤ c + (c - 1)" by simp
  then have "(a + b) div c ≤ (c + (c - 1)) div c" by (rule div_le_mono)
  moreover have "(c + (c - 1)) div c < 2" using ‘2 ≤ c‘
    by (simp only: div_add_self1) simp
  ultimately have "(a + b) div c < 2" by simp
  with ‘2 ≤ c‘ show "(a + b) div c < c" by auto
next
  case False then show ?thesis using assms by simp
```

```
qed

lemma times_div_less_self:
  fixes a b c :: nat
  assumes "a < c" and "b < c"
  shows "(a * b) div c < c"
proof (cases "2 ≤ c")
  case True
  then obtain d where "c = Suc d" by (cases c) simp_all
  then have "c * c - 1 = (c - 1) + (c - 1) * c" by simp
  then have "(c * c - 1) div c = ((c - 1) + (c - 1) * c) div c" by simp
  also have "... = c - 1 + (c - 1) div c" using ‘c = Suc d‘
    by (simp only: div_mult_self1)
  also have "... = c - 1" using ‘c = Suc d‘ by simp
  finally have dec_c: "(c * c - 1) div c = c - 1" .
  from assms have "a * b < c * c" by (simp add: mult_strict_mono)
  then have "a * b ≤ c * c - 1" by simp
  then have "(a * b) div c ≤ (c * c - 1) div c" by (rule div_le_mono)
  with dec_c have "(a * b) div c ≤ c - 1" by simp
  then show "(a * b) div c < c" using ‘c = Suc d‘ by simp
next
  case False then have "c = 0 ∨ c = 1" by arith
  then show ?thesis using assms by simp
qed
```

Further algebraic and arithmetic lemmas helpful during proofs can be obtained using the `find_theorems` command. Note that in most situations you should restrict your search to lemmas on natural numbers (or more general) by providing suitable explicit type annotations in the search patterns.

## 1.2 Addition

Define a function `add :: nat ⇒ nat list ⇒ nat list ⇒ nat list` that adds two `big_nat`s with the same `base`. Make sure that your algorithm preserves the validity of the `big_nat` representation.

Using `val`, verify formally that your function `add` computes the sum of two `big_nat`s correctly.

Using `valid`, verify formally that your function `add` preserves the validity of the `big_nat` representation.

*Hints*:

- Use auxiliary functions if necessary.
- Perform induction with a specific induction rule and generalization.

## 1.3 Multiplication

Define a function `mult :: nat ⇒ nat list ⇒ nat list ⇒ nat list` that multiplies two `big_nat`s with the same `base`. You may use already existing operations. Make sure that your algorithm preserves the validity of the `big_nat` representation.

Using `val`, verify formally that your function `mult` computes the product of two `big_nat`s correctly.

Using `valid`, verify formally that your function `mult` preserves the validity of the `big_nat` representation.

*Hints*:

- See above.
- Don't be irritated if things turn out to be considerably easier when expecting the opposite.