Universität Karlsruhe (TH)
Lehrstuhl für Programmierparadigmen

Theorembeweiser und ihre Anwendungen SS 2009 http://pp.info.uni-karlsruhe.de/
Übungsleiter: Daniel Wasserrab wasserra@ipd.info.uni-karlsruhe.de

Übungsblatt 12 Besprechung: 14.07.2009

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.

```
types 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.

```
consts
  valid :: "nat ⇒ bigNat ⇒ bool"
  val :: "nat ⇒ bigNat ⇒ nat"
```

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.

```
consts
  add :: "nat ⇒ bigNat ⇒ bigNat ⇒ bigNat"

lemma add_valid: "⟦valid d ns; valid d ms⟧ ⟹ valid d (add d ns ms)"
oops
```

Eventuell brauchen Sie in folgenden Lemma die Voraussetzungen `valid d ns` und `valid d ms`

```
lemma add_val: "val d (add d ns ms) = val d ns + val d ms"
oops
```

Multiplikation

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

```
consts
  mult :: "nat ⇒ bigNat ⇒ bigNat ⇒ bigNat"

lemma mult_valid: "⟦valid d ns; valid d ms⟧ ⟹ valid d (mult d ns ms)"
oops
```

Auch hier könnte wieder die Forderung nach gültigen Zahlen nötig sein

```
lemma mult_val: "val d (mult d ns ms) = val d ns * val d ms"
oops
```

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.

Versuchen Sie ihre Beweise in einem möglichst klaren Isar-Skript darzustellen.

Verwenden Sie für ihre Definitionen die Funktionen `div` und `mod`, 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.

Folgende Aussagen werden in den Beweisen wahrscheinlich zur Lösung benötigt werden, sie sind hier mit einem Beweis angegeben. Sie müssen sie also nur noch an die benötigte Stelle kopieren (oder Sie beweisen sie selbst).

```
lemma less_sqr_div_less: "m < d * d ⟹ m div d < (d::nat)"
lemma less_less_add_div_less:
  "⟦a < d; b < d⟧ ⟹ (a + b) div d < (d::nat)"
lemma less_less_less_add_div_less:
  "⟦a < d; b < d; c < d⟧ ⟹ (a + b + c) div d < (d::nat)"
lemma le_le_le_add_mult_div_Suc_le:
  "⟦a ≤ d; b ≤ d; c ≤ d⟧ ⟹ (a * b + c) div (Suc d) ≤ d"
lemma less_less_less_add_mult_div_less:
  "⟦a < d; b < d; c < d⟧ ⟹ (a * b + c) div d < (d::nat)"
```

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
```