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

SS 2009
28. 05. 2009

Perlen der Informatik 2
4. Übung

1 Vedische Arithmetik

Ein Buch über die vedische Mathematik beschreibt drei Methoden, die Berechnung des Quadrats von natürlichen Zahlen zu vereinfachen:

- `MM1`: Zahlen, für die das Quadrat des Vorgängers entweder bekannt oder einfach zu berechnen ist. Zum Beispiel:
  Gesucht: `61 ^ 2`
  Gegeben: `60 ^ 2 = 3600`
  Folglich: `61 ^ 2 = 3600 + 60 + 61 = 3721`
- `MM2`: Zahlen größer, aber nahe `100`. Zum Beispiel:
  Gesucht: `102 ^ 2`
  Sei `h = 102 - 100 = 2`, somit `h ^ 2 = 4`
  Folglich: `102 ^ 2 = 102 + h um 2 Stellen nach links verschoben + h ^ 2 = 10404`
- `MM3`: Zahlen, die mit `5` enden. Zum Beispiel:
  Gesucht: `85 ^ 2`
  Folglich: `85 ^ 2 = 8 * 9 gefolgt von 25 = 7225`
  Gesucht: `995 ^ 2`
  Folglich: `995 ^ 2 = 99 * 100 gefolgt von 25 = 990025`

In dieser Übung werden wir zeigen, dass diese Methoden letztlich gar nicht so magisch sind:

- ▷ Fassen Sie `MM1` in eine primitiv rekursive Funktion `sq`.
- ▷ Zeigen Sie, dass Ihre Definition von `sq` korrekt ist: `sq n = n * n`.
- ▷ Formulieren und beweisen Sie die Korrektheit von `MM2` und `MM3`.

  Hinweise:
  - Stützen Sie ihre Formulierung auf die Funktion `sq` ab.
  - Verallgemeinern Sie `MM2` auf eine beliebige Konstante (statt `100`).
  - Versuchen Sie die Eigenschaft ‘die mit `5` enden’ so zu formulieren, dass es einfach ist, den Rest der Zahl zu erhalten.
  - Die Subtraktion auf natürlichen Zahlen kann beim Beweisen unhandlich sein (warum?). Evtl. ist es günstiger, Subtraktion geschickt in Vorbedingungen einzukodieren.
  - Wenn Sie Induktion machen, verallgemeinern Sie, wenn nötig, alle Variablen außer der Induktionsvariable: `(induct ... arbitrary: ...)`
  - Benutzen Sie `find_theorems` zur Suche nach arithmetischen Regeln, die Sie benötigen. Es kann auch nötig sein, Regeln für `sq` zu beweisen (z.B. binomische Formeln).

2 Das Pascalsche Dreieck

Die ersten 8 Zeilen des *Pascalschen Dreiecks* lauten:

$n=0$							1							
$n=1$						1		1						
$n=2$					1		2		1					
$n=3$				1		3		3		1				
$n=4$			1		4		6		4		1			
$n=5$		1		5		10		10		5		1		
$n=6$	1		6		15		20		15		6		1	
$n=7$	1	7		21		35		35		21		7		1

Die inneren Elemente der $n+1$-ten Zeile im Pascalschen Dreieck werden konstruiert, indem man jeweils die Summe zweier benachbarter Elemente der n-ten Zeile bildet.

▷ Benutzen Sie die Funktion

```
primrec zipadd :: "nat ⇒ nat list ⇒ nat list"  where
    "zipadd n [] = [1]"
  | "zipadd n (m # ms) = (n + m) # zipadd m ms"
```

um das Pascalsche Dreieck in Isabelle zu konstruieren. Die `n`-te Zeile des Dreiecks soll dabei durch eine Funktion `row :: nat ⇒ nat list` berechnet werden:

▷ Beweisen Sie, dass `row n` für `n = 0, 1, 2, 7` genau die Werte in der obigen Abbildung berechnet. *Hinweis:* Numeraldarstellungen natürlicher Zahlen können mit `apply (simp add: nat_number)` in `Suc`-Darstellung aufgefaltet werden.

Im folgenden wird es des öfteren nötig sein, Variablen bei der Induktion zu verallgemeinern.

▷ Beweisen Sie `length (row n) = Suc n`. Beweisen Sie dazu erst `length (zipadd m ms) = Suc (length ms)`.

▷ Erinnern Sie sich an die Funktion `sum :: nat list ⇒ nat` von Blatt 3; zeigen Sie:

```
sum (row n) = 2 ^ n
```

Dabei steht `m ^ n` für Exponentation m^n.

Zum Beweis ist folgendes Lemma hilfreich:

```
sum_zipadd: ms ≠ [] ⟹ sum (zipadd m ms) = 2 * sum ms + m + 1 - last ms
```

Zu dessen Beweis wiederum ein kleiner Griff in die Trickkiste. Wie bereits erwähnt, ist Subtraktion auf `nat` manchmal unhandlich; daher beweisen wir zuerst:

```
sum_zipadd_aux: ms ≠ [] ⟹ sum (zipadd m ms) + last ms = 2 * sum ms + m + 1
```

Dieses setzen wir dann wie folgt ein:

```
lemma sum_zipadd: "ms ≠ [] ⟹ sum (zipadd m ms) = 2 * sum ms + m + 1 - last ms"
  unfolding sum_zipadd_aux [symmetric] by simp
```

Die Annotation `[symmetric]` ist ein *Attribut* und kehrt die Orientierung einer (ggf. mit Seitenbedingungen versehenen) Gleichung um; **unfolding** wendet seine Argumente als Rewrite-Regeln auf den Goal-Kontext an.

Darüber hinaus sind sicherlich noch weitere Lemmata nötig.

3 Darstellung logischer Formeln als Polynome

Gegeben sei der folgende Datentyp für aussagenlogische Formeln:

```
datatype form = T | Var nat | And form form | Xor form form
```

Hierbei steht `T` für die Formel, die immer wahr ist, `Var n` bezeichne eine Aussagenvariable, wobei Variablennamen als natürliche Zahlen dargestellt werden, `And f1 f2` bezeichne die und-Verknüpfung und `Xor f1 f2` die exklusiv-oder-Verknüpfung zweier Formeln. Ein Konstruktor `F` für die immer falsche Formel ist unnötig, da dies durch `Xor T T` ausgedrückt werden könnte.

▷ Definieren Sie eine Funktion `evalf :: (nat ⇒ bool) ⇒ form ⇒ bool`, die eine Formel unter einer gegebenen Variablenbelegung auswertet.

Aussagenlogische Formeln lassen sich als sogenannte *Polynome* darstellen. Unter einem Polynom versteht man eine Liste von Listen von Aussagevariablen, d.h. ein Element vom Typ `nat list list`. Hierbei interpretiert man die inneren Listen (die sogenannten *Monome*) als und-Verknüpfung der Variablen, während die äußere Liste als exklusiv-oder-Verknüpfung der inneren Listen interpretiert wird.

▷ Definieren Sie zwei Funktionen `evalm :: (nat ⇒ bool) ⇒ nat list ⇒ bool` und `evalp :: (nat ⇒ bool) ⇒ nat list list ⇒ bool` zur Auswertung von Monomen und Polynomen unter einer gegebenen Variablenbelegung. Überlegen Sie sich insbesondere, wie leere Listen ausgewertet werden müssen.

▷ Definieren Sie eine Funktion `poly :: form ⇒ nat list list`, die eine Formel in ein Polynom umwandelt. Sie benötigen hierfür eine Hilfsfunktion `mult_pp :: nat list list ⇒ nat list list ⇒ nat list list` zur "Multiplikation" zweier Polynome, d.h. zur Berechnung von

$$((v_1^1 \odot \cdots \odot v_{m_1}^1) \oplus \cdots \oplus (v_1^k \odot \cdots \odot v_{m_k}^k)) \odot ((w_1^1 \odot \cdots \odot w_{n_1}^1) \oplus \cdots \oplus (w_1^l \odot \cdots \odot w_{n_l}^l))$$

wobei $\oplus$ "exklusiv-oder" und $\odot$ "und" bedeutet. Dies geschieht unter Anwendung der üblichen Rechenregeln für Addition und Multiplikation.

▷ Zeigen Sie nun die Korrektheit der Funktion `poly`: `evalf e f ⟷ evalp e (poly f)`.

Es ist nützlich, hierfür zunächst ein analoges Korrektheitstheorem für `mult_pp` zu beweisen.