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

SS 2009
14. 05. 2009

Perlen der Informatik 2
3. Übung

1 Die ersten Schritte mit Isabelle

1.1 Eine Frage der richtigen Einstellung

Wie jedes nichttriviale System hat auch Isabelle/ProofGeneral eine recht unübersichtliche Vielzahl an Konfigurations- und Einstellungsoptionen.

Die grundlegendste beginnt bei der Installation: halten Sie sich bei der Auswahl der Emacs-Version an die auf der Isabelle-Website `isabelle.in.tum.de` angegebenen Ratschläge. Andere Emacse spielen in der Regel nicht gut mit ProofGeneral zusammen.

Auch beim Start von Isabelle/ProofGeneral hat man die Qual der Wahl:

```
isabelle emacs -m iff -m no_brackets
```

Über die Schalter `-m` werden sogenannte Printmodi angegeben. `iff` druckt dabei die logische Äquivalenz als *"P ⟷ Q"*; ansonsten würde sie als Gleichheit auf *bool* dargestellt (*"P = Q"*), was i.A. eher verwirrend ist. Mit `no_brackets` werden Propositionen mit mehreren Annahmen *gecurried* gedruckt als *"P ⟹ Q ⟹ R"*, ansonsten wird eine geklammerte Darstellung *"⟦P; Q⟧ ⟹ R"* verwendet; die Auswahl ist Geschmackssache.

Printmodi lassen sich auch via Shell-Variable festlegen:

```
export PROOFGENERAL_OPTIONS=-m iff -m no_brackets
```

Nach dem ersten Start von ProofGeneral sollte als erstes XSymbol aktiviert und dies persistent gespeichert werden:

ProofGeneral → *Options* → *XSymbol*
ProofGeneral → *Options* → *Save options*

Einstellungen für Isabelle finden sich im Menü *Isabelle* → *Settings* und lassen sich dort mit *Isabelle* → *Settings* → *Save settings* persistent speichern. Das Menü wird erst *nach* dem Start des Isabelle-Prozesses angezeigt: *Isabelle* → *Start Isabelle*.

Show Types – Typen mit anzeigen; i.A. sind Propositionen *ohne* Typen lesbarer; falls sich Terme nicht wie erwartet verhalten, kann es sinnvoll sein, sich die Typen temporär anzeigen zu lassen.
Eta Contract – implizite Eta-Kontraktion beim Anzeigen von Termen; am Anfang eher verwirrend als hilfreich.
Show Consts – Konstanten mit Typen im Goal gesondert aufführen; kann bei Unklarheiten mit überladenen Operationen (z.B. *op +*, *"0"*) hilfreich sein.
Show Sorts – Typvariablen im Goal gesondert aufführen; hilft ebenfalls bei überladenen Operationen.
Auto Quickcheck – Wendet *Quickcheck* (s.u.) automatisch nach dem Notieren eines **lemma** an.

1.2 Evaluation von Termen und Gegenbeispielsuche für Propositionen

Beim ersten Formulieren der Proposition eines Theorems sind oft noch Fehler enthalten; um nicht unnötig Zeit auf Beweise falscher Theoreme zu verwenden, enthält Isabelle eine automatische Gegenbeispiel-Suche, die mit dem Isar-Kommando **quickcheck**, alternativ *Isabelle* → *Commands* → *Quickcheck* aufgerufen wird. Dies erfolgt bei entsprechender Einstellung (s.o.) auch automatisch.

Unabhängig davon kann es sinnvoll sein, die Plausibilität von Definitionen etc. durch Auswerten von Beispielen zu überprüfen. Verwenden Sie dazu

value term

2 Zum Warmwerden: Listen und natürliche Zahlen

▷ Definieren sie die Fakultätsfunktion `fac :: nat ⇒ nat` auf natürlichen Zahlen primitiv rekursiv. Sie können dazu die Multiplikation `m * n` auf natürlichen Zahlen verwenden.

▷ Definieren sie die Folge der Fibonacci-Zahlen f_n als Funktion `fib :: nat ⇒ nat`. Die Fibonacci-Zahlen gehorchen folgender Rekursionsvorschrift:

f_0 = 0
f_1 = 1
f_n = f_{n-1} + f_{n-2}

▷ Definieren sie eine Funktion `sum :: nat list ⇒ nat`, die eine Liste natürlicher Zahlen aufsummiert. Die Addition `m + n` auf natürlichen Zahlen ist bereits definiert.

▷ Zeigen Sie die folgenden Aussagen:

```
sum_append: sum (xs @ ys) = sum xs + sum ys
sum_rev: sum (rev xs) = sum xs
```

Eine Liste von Listen lässt sich durch Aneinanderhängen der einzelnen Listen "flachklopfen". Dazu gibt es bereits eine vordefinierte Funktion in der Listentheorie *List*. Weiterhin liefert die Funktion `length :: 'a list ⇒ nat` die Länge einer Liste.

▷ Formulieren und beweisen Sie: die Länge einer flachgeklopften Liste von Listen ist die Summe der Längen der einzelnen Listen. Zum Formulieren ist die Funktion *map* hilfreich (oder definieren sie eine geeignete Hilfsfunktion).

▷ Zeigen Sie eine ähnliche Eigenschaft über flachgeklopfte Listen und *sum*.

▷ Definieren Sie eine Funktion `countdown :: nat ⇒ nat list`, so dass

```
countdown n = [n, n-1, n-2, ..., 1]
```

▷ Zeigen Sie:

```
length (countdown n) = n
```

▷ Formulieren Sie einen geschlossenen arithmetischen Ausdruck für `sum (countdown n)`. Denken Sie dabei an Carl Friedrich Gauss. Beweisen Sie die Richtigkeit Ihrer Überlegung mit einem Gleichheitstheorem

```
sum (countdown n) = ....
```

Dabei können Sie zum Halbieren gerader natürlicher Zahlen die Ganzzahl-Division `... div 2` verwenden.

3 Binäre Suchbäume

Natürliche Zahlen und Listen sind beide induktive Datentypen:

```
datatype nat = 0 | Suc nat
datatype 'a list = [] | op # 'a ('a list)
```

Ein weiterer Standardtyp aus dieser Klasse ist:

```
datatype 'a option = None | Some 'a
```

Er drückt das mögliche Nicht-Vorhandensein eines Wertes aus. Weiterhin können damit partielle Funktionen in die totale Logik HOL eingebettet werden: eine partielle Funktion von `'a` nach `'b` wird dargestellt als `"'a ⇒ 'b option"`; da dies häufig verwendet wird, gibt es dafür sogar eine eigene Syntax: `"'a ⇀ 'b option"`.

In einer Anwendung hiervon modellieren wir binäre Suchbäume mit `nat` als Schlüssel und beliebigen Werten als Datentyp:

```
datatype 'a tree = Empty
  | Branch 'a nat "'a tree" "'a tree"
```

Hierbei ist `Empty` der leere Baum und `Branch v k l r` ein Knoten mit Schlüssel `k`, Wert `v`, linkem Teilbaum `l` und rechtem Teilbaum `k`.

Wir wollen Bäume betrachten, die die Suchbaumeigenschaft erfüllen:

> An einem Knoten `Branch v k l r` sind alle Schlüssel in `l` kleiner als `k` und alle Schlüssel in `r` größer als `k`.

Jeder Baum, der dieser Eigenschaft genügt, beschreibt eine partielle Abbildung von Schlüsseln auf Werte, die durch folgende Funktion beschrieben wird:

```
primrec lookup :: "'a tree ⇒ nat ⇀ 'a" where
  "lookup Empty = (λ_. None)"
  | "lookup (Branch v k l r) = (λk'. if k' = k then Some v
       else if k' < k then lookup l k' else lookup r k')"
```

▷ Definieren Sie eine dazu passende Funktion `update :: nat ⇒ 'a ⇒ 'a tree ⇒ 'a tree`, die ein Schlüssel-Wert-Paar in einen bestehenden Baum einfügt bzw. überschreibt, falls der Schlüssel schon vorhanden. Für den ursprünglichen Baum soll dabei die Suchbaumeigenschaft vorausgesetzt werden, für den resultierenden Baum soll sie erhalten werden.

▷ Zeigen Sie damit die folgenden Eigenschaften:

```
lookup (update k v t) k = Some v
```

```
k ≠ k' ⟹ lookup (update k v t) k' = lookup t k'
```

▷ Zeigen Sie ähnliche Eigenschaft(en) für `lookup (update k' v' (update k v t))`.

Innerhalb einer Logik wie HOL ist es selbstverständlich, über Gleichheit auf Funktionstypen zu reden; diese ist definiert über Extensionalität:

```
(⋀x. f x = g x) ⟹ f = g
```

In Beweisen mit *simp* und vergleichbaren Methoden ist es dabei oft angemessen, die verwandte Gleichung

```
f = g ⟷ (∀x. f x = g x)
```

in die Simplifikationsregeln mitaufzunehmen.

Bemerkenswert ist, dass alle hier erbrachten Beweise ohne eine explizite Formulierung der Suchbaumeigenschaft auskommen.