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

SS 2009
18. 06. 2009

Perlen der Informatik 2
6. Übung

1 Natürliches Schließen in der Prädikatenlogik

Zusätzlich zu den Regeln der Aussagenlogik kommen jetzt die folgenden Regeln zum Einsatz:

```
exI: P x ⟹ ∃x. P x
exE: ∃x. P x ⟹ (⋀x. P x ⟹ Q) ⟹ Q
allI: (⋀x. P x) ⟹ ∀x. P x
allE: ∀x. P x ⟹ (P x ⟹ R) ⟹ R
```

Die Methoden bleiben weiterhin auf `rule`, `erule` und `assumption` (ggf. auch `drule` und `frule`) beschränkt.

Zeigen Sie die folgenden prädikatenlogischen Aussagen:

```
∀x. (P x ⟶ Q x) ∧ ¬ Q y ⟹ ¬ P y
(∃x. P x) ∧ (∀x. P x ⟶ Q x) ⟹ ∃x. Q x
¬ (∀x. P x) ⟹ ∃x. ¬ P x
```

2 Knobelaufgabe: Rich Grandfather

Ab hier dürfen alle Lemmas verwendet werden; die Methoden bleiben beschränkt auf `rule`, `erule` *und* `assumption` *sowie* `drule` *und* `frule`*; zusätzlich ist Fallunterscheidung auf booleschen Ausdrücken mit* `cases` *gestattet.*

Beweisen Sie die folgende in klassischer Prädikatenlogik gültige Formel zunächst informell mit Papier und Bleistift. Verwenden Sie im Beweis Fallunterscheidung oder Widerspruch.

Wenn jeder Arme einen reichen Vater hat,
dann gibt es einen Reichen mit einem reichen Großvater.

```
(∀x. ¬ rich x ⟶ rich (father x)) ⟶ (∃x. rich x ∧ rich (father (father x)))
```

3 Verfeinerung von Implementierungen: noch einmal Fibonacci

Ab hier und auch in Zukunft dürfen alle Methoden und Lemmas verwendet werden

Gehen Sie nochmal zurück zur Definition der Fibonacci-Zahlen von Blatt 3: `fib :: nat ⇒ nat`.

Wir werden nun Isabelles Code-Generator verwenden, um die 35. Fibonacci-Zahl auszurechnen, und stoppen dabei die Zeit.

```
definition "example _ = fib 35"
```

Das zusätzliche Argument stellt sicher, dass die Zeitmessung erst bei der Ausführung beginnt und nicht bei der Code-Generierung und Compilation.

```
export_code example in SML module_name Fibonacci
ML {* set Toplevel.timing *}
ML {* Fibonacci.example () *}
ML {* reset Toplevel.timing *}
```

Betrachten Sie den Code, der für `fib` generiert wird:

```
export_code fib in SML module_name Fibonacci file -
```

Machen Sie sich durch Inspektion des Codes klar, das die Zahl der Aufrufe von `fib` exponentiell mit der Eingabezahl wächst.

Wir werden nun eine effizientere, äquivalente Implementierung entwickeln. Dazu definieren wir:

```
definition fib_step :: "nat ⇒ nat × nat" where
  fib_step_def: "fib_step n = (fib (Suc n), fib n)"
```

Zeigen Sie die folgenden Rekursions-Gleichungen für `fib_step`:

```
fib_step_Zero: fib_step 0 = (Suc 0, 0)
fib_step_Suc: fib_step (Suc n) = (let (n, m) = fib_step n in (n + m, n))
```

Wir geben diese Gleichungen an den Code-Generator:

```
declare fib_step_def [code del] fib_step_Zero [code] fib_step_Suc [code]
```

Die Implementierung von `fib` soll sich nun auf `fib_step` abstützen und insbesondere *keine* rekursiven Aufrufe für `fib` mehr enthalten. Schauen Sie sich die Funktionsgleichungen von `fib` an, überlegen Sie sich, welche Gleichung(en) ersetzt werden sollen; beweisen Sie für diese geeignete Gegenstücke mit gleicher linker Seite, aber anderer rechter Seite, und geben Sie diese an den Code-Generator. Inspizieren Sie den generierten Code:

```
export_code fib in SML module_name Fibonacci file -
```

Überzeugen Sie sich von der kürzeren Laufzeit:

```
export_code example in SML module_name Fibonacci
ML {* set Toplevel.timing *}
ML {* Fibonacci.example () *}
ML {* reset Toplevel.timing *}
```

Noch einmal zu unserem Vorgehen: wir haben `fib` dadurch effizient(er) implementiert, indem wir eine (zunächst komplizierte) Variante `fib_step` in Abhängigkeit davon definiert haben, und darüber Gleichungen bewiesen haben, die sich als Code-Gleichungen eignen. Der Zusammenhang zwischen `fib` und `fib_step` ergibt sich bei diesem Ansatz aus der Konstruktion und ist trivial zu beweisen.