Karlsruher Institut für Technologie

Institut für Programmstrukturen und Datenorganisation
Lehrstuhl Programmierparadigmen
Am Fasanengarten 5
76131 Karlsruhe
http://pp.ipd.kit.edu/

Theorembeweiserpraktikum – SS 2018

http://pp.ipd.kit.edu/lehre/SS2018/tba

Lösung 2: Simplifikation

Abgabe: 7. Mai 2018, 12:00 Uhr
Besprechung: 8. Mai 2018

1 Prädikatenlogik

Es geht wieder um Beweise mit den Regeln des Kalküls des natürlichen Schließens. Zusätzlich zu den Regeln der letzten Übung können Sie nun auch folgende Regeln verwenden:

allI: $(\bigwedge x.\ P\ x) \Longrightarrow \forall x.\ P\ x$ *allE*: $\forall x.\ P\ x \Longrightarrow (P\ x \Longrightarrow R) \Longrightarrow R$

exI: $P\ x \Longrightarrow \exists x.\ P\ x$ *exE*: $\exists x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q) \Longrightarrow Q$

Es dürfen wieder nur die Befehle **proof** (mit (*rule* ...)), **assume**, **have**, **show**, **next**, **qed** und **from** sowie die darauf aufbauenden Abkürzungen verwendet werden. Zusätzlich dürfen die Befehle **fix** und **obtain** verwendet werden.

Beispiel

```
lemma "∀x. P x ⟶ (∃x. P x)"
proof (rule allI)
  fix x
  show "P x ⟶ (∃x. P x)"
  proof
    assume "P x"
    then show "∃x. P x" by (rule exI)
  qed
qed

lemma "(∀x. P x) ⟷ ¬ (∃x. ¬ P x)"
proof
  assume "∀x. P x"
  show "¬ (∃x. ¬ P x)"
  proof
    assume "∃x. ¬ P x"
    then obtain x where "¬ P x" by (rule exE)
    from ‹∀x. P x›
    have "P x" by (rule allE)
    with ‹¬ P x›
    show False..
  qed
next
  assume lhs: "¬ (∃x. ¬ P x)"
  show "∀x. P x"
  proof (rule allI)
```

```
    fix x
    show "P x"
    proof (rule ccontr)
      assume "¬ P x"
      then have "∃x. ¬ P x" by (rule exI)
      with lhs
      show False..
    qed
  qed
qed

lemma "(¬ (∀x. P x)) ⟷ (∃x. ¬ P x)"
proof
  assume "¬ (∀x. P x)"
  show "∃x. ¬ P x"
  proof (rule ccontr)  — Regel contrapos_np: ¬ A ⟹ (¬ B ⟹ A) ⟹ B wäre kürzer
    assume "¬ (∃x. ¬ P x)"
    have "∀ x. P x"
    proof
      fix x
      show "P x"
      proof (rule ccontr)
        assume "¬ P x"
        then have "∃x. ¬ P x"..
        with ‹¬ ?this› — ?this ist die letzte Aussage, also ∃x. ¬ P x
        show False..
      qed
    qed
    with ‹¬ ?this›
    show False..
  qed
next
  assume "∃x. ¬ P x"
  then obtain x where "¬ P x"..
  show "¬ (∀x. P x)"
  proof
    assume "∀x. P x"
    then have "P x"..
    with ‹¬ ?this›
    show False..
  qed
qed

lemma "(∀x. P x ⟶ Q) ⟷ ((∃x. P x) ⟶ Q)"
proof
  assume "∀x. P x ⟶ Q"
  show "(∃x. P x) ⟶ Q"
  proof
    assume "∃x. P x"
    then obtain x where "P x"..
    from ‹∀x. P x ⟶ Q› have "P x ⟶ Q"..
    from this ‹P x›
```

```
    show Q..
  qed
next
  assume "(∃x. P x) ⟶ Q"
  show "∀x. P x ⟶ Q"
  proof
    fix x
    show "P x ⟶ Q"
    proof
      assume "P x"
      then have "∃x. P x"..
      with ‹(∃x. P x) ⟶ Q›
      show Q..
    qed
  qed
qed

lemma "(∃x. ∀y. P x y) ⟶ (∀y. ∃x. P x y)"
proof
  assume "∃x. ∀y. P x y"
  then obtain x where "∀ y. P x y"..
  show "∀y. ∃x. P x y"
  proof
    fix y
    from ‹∀ y. P x y›
    have "P x y"..
    then show "∃x. P x y"..
  qed
qed

lemma "(∀x. P x) ∧ (∀x. Q x) ⟷ (∀x. (P x ∧ Q x))"
proof
  assume "(∀x. P x) ∧ (∀x. Q x)"
  then have "∀ x. P x"..
  from ‹(∀x. P x) ∧ (∀x. Q x)›
  have "∀ x. Q x"..

  show "∀x. (P x ∧ Q x)"
  proof
    fix x
    show "P x ∧ Q x"
    proof
      from ‹∀x. P x› show "P x"..
    next
      from ‹∀x. Q x› show "Q x"..
    qed
  qed
next
  assume "∀x. (P x ∧ Q x)"
  show "(∀x. P x) ∧ (∀x. Q x)"
  proof
    show "∀x. P x"
```

```
    proof
      fix x
      from ‹∀x. (P x ∧ Q x)›
      have "P x ∧ Q x"..
      then show "P x"..
    qed
  next
    show "∀x. Q x"
    proof
      fix x
      from ‹∀x. (P x ∧ Q x)›
      have "P x ∧ Q x"..
      then show "Q x"..
    qed
  qed
qed

lemma "(∃x. P x) ∨ (∃x. Q x) ⟷ (∃x. (P x ∨ Q x))"
proof
  assume "(∃x. P x) ∨ (∃x. Q x)"
  then show "∃x. (P x ∨ Q x)"
  proof
    assume "∃ x. P x"
    then obtain x where "P x"..
    then have "P x ∨ Q x"..
    then show "∃x. (P x ∨ Q x)"..
  next
    assume "∃ x. Q x"
    then obtain x where "Q x"..
    then have "P x ∨ Q x"..
    then show "∃x. (P x ∨ Q x)"..
  qed
next
  assume "∃x. (P x ∨ Q x)"
  then obtain x where "P x ∨ Q x"..
  then show "(∃x. P x) ∨ (∃x. Q x)"
  proof
    assume "P x"
    then have "∃ x. P x"..
    then show "(∃x. P x) ∨ (∃x. Q x)"..
  next
    assume "Q x"
    then have "∃ x. Q x"..
    then show "(∃x. P x) ∨ (∃x. Q x)"..
  qed
qed
```

Bei diesem Lemma darf mit Fallunterscheidung (Methode `cases`) gearbeitet werden. Erinnerung: Eine Variable, über die Sie nichts wissen (brauchen), erhalten Sie mit **fix**.

```
lemma "∃x. P x ⟶ (∀x. P x)"
proof(cases "∃x. ¬ P x")
  assume "∃ x. ¬ (P x)"
```

```
  then obtain x where "¬ P x"..
  have "P x ⟶ (∀x. P x)"
  proof
    assume "P x"
    with ‹¬ P x›
    show "∀x. P x"..
  qed
  then show "∃x. P x ⟶ (∀x. P x)"..
next
  assume "¬ (∃ x. ¬ P x)"
  have "∀ x. P x"
  proof
    fix x
    show "P x"
    proof (rule ccontr)
      assume "¬ P x"
      then have "∃ x. ¬ P x"..
      with ‹¬ ?this›
      show False..
    qed
  qed

  fix x
  from ‹∀x. P x›
  have "P x ⟶ (∀x. P x)"..
  then show "∃x. P x ⟶ (∀x. P x)"..
qed
```

2 Definitionen und Arbeiten mit Gleichheit

In klassischer Aussagenlogik können alle Aussagen allein aus `False` und `nor` gebildet werden. Definieren Sie den `nor`-Operator:

```
definition
  nor :: "bool ⇒ bool ⇒ bool"  (infix "↓" 37)
  where "A ↓ B ⟷ ¬ (A ∨ B)"
```

Nun leiten Sie Lemmas her, die die üblichen boolschen Junktionen nur mit `False` und `op ↓` darstellen. Theoretisch ist `op ↓` alleine schon universell. Der Einfachheit halber dürfen Sie in dieser Aufgabe aber auch noch `False` zusätzlich verwenden. Gehen Sie dabei wie folgt vor:

- Wenden Sie zu Beginn keine Regel an (**proof**-).
- Schreiben Sie erst mit **have** `.... = ....` den Ausdruck in eine Form um, die neben Ausdrücken der Form `¬ (.... ∨ ....)` nur Junktionen verwendet, für die Sie bereits Regeln geschrieben haben.
- Führen Sie diese dann Schrittweise mit **also have** in die gewünschte Form über. Dabei sollten Sie neben `nor_def[symmetric]` nur die davor bewiesenen Regeln `rewrite_foo` in ihren **from**-Befehlen aufführen müssen, und als Beweis dann **.** oder **by** (`rule arg_cong`) verwenden.

- Wenn die Gleichungskette zum Lemma passt, lässt sie sich mit **finally show** *?thesis* **.** abschließen.

Um *True* zu zeigen verwenden Sie die Regel *TrueI: True*.

```
lemma rewrite_not: "¬ A ⟷ False ↓ A"
proof -
  have "¬ A ⟷ ¬ (False ∨ A)"
  proof
    assume "¬ A"
    show "¬ (False ∨ A)"
    proof
      assume "False ∨ A"
      then show False
      proof
        assume A
        with ‹¬A›
        show False..
      qed
    qed
  next
    assume "¬ (False ∨ A)"
    show "¬ A"
    proof
      assume A
      then have "False ∨ A"..
      with ‹¬ ?this›
      show False..
    qed
  qed
  also
  from nor_def[symmetric]
  have "... ⟷ False ↓ A".
  finally
  show ?thesis.
qed

lemma rewrite_or: "A ∨ B ⟷ False ↓ (A ↓ B)"
proof -
  have "A ∨ B ⟷ ¬ (¬ (A ∨ B))"
  proof
    assume "A ∨ B"
    show "¬ (¬ (A ∨ B))"
    proof
      assume "¬ (A ∨ B)"
      from this ‹A ∨ B›
      show False..
    qed
  next
    assume "¬ (¬ (A ∨ B))"
    show "A ∨ B"
    proof (rule ccontr)
```

```
      assume "¬ (A ∨ B)"
      with ‹¬ (¬ (A ∨ B))›
      show False..
    qed
  qed
  also
  from nor_def[symmetric]
  have "... ⟷ ¬ (A ↓ B)" by (rule arg_cong)
  also
  from rewrite_not
  have "... ⟷ False ↓ (A ↓ B)".
  finally
  show "A ∨ B ⟷ False ↓ (A ↓ B)".
qed

lemma rewrite_and: "A ∧ B ⟷ (False ↓ A) ↓ (False ↓ B)"
proof -
  have "A ∧ B ⟷ ¬ (¬ A ∨ ¬ B)"
  proof
    assume "A ∧ B" then have A..
    from ‹A ∧ B› have B..
    show "¬ (¬ A ∨ ¬ B)"
    proof
      assume "¬ A ∨ ¬ B"
      then show False
      proof
        assume "¬ A"
        from this ‹A›
        show False..
      next
        assume "¬ B"
        from this ‹B›
        show False..
      qed
    qed
  next
    assume "¬ (¬ A ∨ ¬ B)"
    show "A ∧ B"
    proof
      show A
      proof (rule ccontr)
        assume "¬ A"
        then have "¬ A ∨ ¬ B"..
        with ‹¬ (¬ A ∨ ¬ B)›
        show False..
      qed
    next
      show B
      proof (rule ccontr)
        assume "¬ B"
        then have "¬ A ∨ ¬ B"..
        with ‹¬ (¬ A ∨ ¬ B)›
```

```
        show False..
      qed
    qed
  qed
  also
  from nor_def[symmetric]
  have "... ⟷ (¬ A) ↓ (¬ B)".
  also
  from rewrite_not
  have "... ⟷ (False ↓ A) ↓ (¬ B)" by (rule arg_cong)
  also
  from rewrite_not
  have "... ⟷ (False ↓ A) ↓ (False ↓ B)" by (rule arg_cong)
  finally
  show "A ∧ B ⟷ (False ↓ A) ↓ (False ↓ B)".
qed

lemma rewrite_imp: "(A ⟶ B) ⟷ False ↓ ((False ↓ A) ↓ B)"
proof -
  have "(A ⟶ B) ⟷ ¬ A ∨ B"
  proof
    assume "A ⟶ B"
    show "¬ A ∨ B"
    proof (cases A)
      assume "¬ A"
      then show "¬ A ∨ B"..
    next
      assume A
      with ‹A ⟶ B›
      have B..
      then show "¬ A ∨ B"..
    qed
  next
    assume "¬ A ∨ B"
    then show "A ⟶ B"
    proof
      assume "¬ A"
      show "A ⟶ B"
      proof
        assume A
        with ‹¬ A›
        show B..
      qed
    next
      assume B
      then show "A ⟶ B"..
    qed
  qed
  also
  from rewrite_or
  have "... ⟷ False ↓ ((¬ A) ↓ B)".
  also
```

```
  from rewrite_not
  have "... ⟷ False ↓ ((False ↓ A) ↓ B)" by (rule arg_cong)
  finally
  show "(A ⟶ B) ⟷ False ↓ ((False ↓ A) ↓ B)".
qed

lemma rewrite_True: "True ⟷ False ↓ False"
proof -
  have "True ⟷ ¬ False"
  proof
    show True by (rule TrueI)
  next
    show "¬ False" by (rule notI)
  qed
  also from rewrite_not have "... ⟷ False ↓ False" .
  finally show ?thesis.
qed
```

Schreiben Sie nun den folgenden Ausdruck schrittweise so um, das er nur mit `op ↓` und `False` gebildet wird. Verwenden Sie dabei das **also** ... **finally** Konstrukt.

```
lemma  "(A ⟶ (A ∨ B)) ∧ ¬ B ⟷
  (False ↓ (False ↓ ((False ↓ A) ↓ (False ↓ (A ↓ B))))) ↓ (False ↓ (False ↓ B))"
proof -
  from rewrite_and
  have "(A ⟶ (A ∨ B)) ∧ ¬ B ⟷ (False ↓ (A ⟶ (A ∨ B))) ↓ (False ↓ (¬ B))".
  also
  from rewrite_not
  have "... ⟷ (False ↓ (A ⟶ (A ∨ B))) ↓ (False ↓ (False ↓ B))"
    by (rule arg_cong)
  also
  from rewrite_imp
  have "... ⟷ (False ↓ (False ↓ ((False ↓ A) ↓ (A ∨ B)))) ↓ (False ↓ (False ↓ B))"
    by (rule arg_cong)
  also
  from rewrite_or
  have "... ⟷ (False ↓ (False ↓ ((False ↓ A) ↓ (False ↓ (A ↓ B))))) ↓ (False ↓ (False
↓ B))"
    by (rule arg_cong)
  finally
  show ?thesis.
qed
```