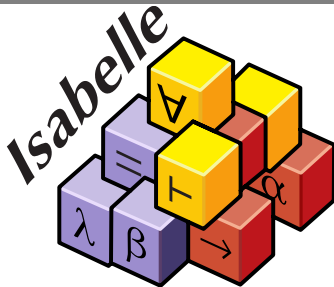


Theorembeweiserpraktikum

Anwendungen in der Sprachtechnologie

LEHRSTUHL PROGRAMMIERPARADIGMEN



Teil XV

Theoreme finden

Befehle, um Theoreme zu finden:

- **find_theorems**
zeigt alle Theoreme an (wenig hilfreich).

Befehle, um Theoreme zu finden:

- **find_theorems**
zeigt alle Theoreme an (wenig hilfreich).
- **find_theorems** *name:classic*
findet alle Theoreme mit *classic* im Namen.

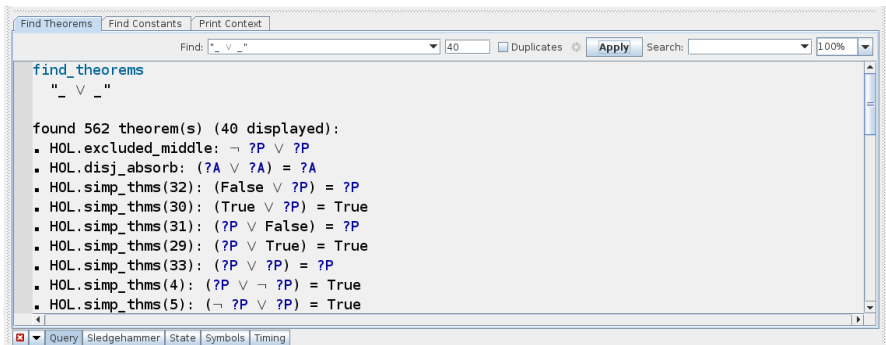
Befehle, um Theoreme zu finden:

- **find_theorems**
zeigt alle Theoreme an (wenig hilfreich).
- **find_theorems** *name:classic*
findet alle Theoreme mit *classic* im Namen.
- **find_theorems** *length*
find_theorems `"?a ∨ ?a"`
find_theorems `"_ ∨ (_ ∨ _)"`
findet alle Theoreme, die den entsprechend Term enthalten.

Befehle, um Theoreme zu finden:

- **find_theorems**
zeigt alle Theoreme an (wenig hilfreich).
- **find_theorems** *name:classic*
findet alle Theoreme mit *classic* im Namen.
- **find_theorems** *length*
find_theorems *"?a ∨ ?a"*
find_theorems *"_ ∨ (_ ∨ _)"*
findet alle Theoreme, die den entsprechend Term enthalten.
- **find_theorems** *length name:induct*
kombiniert die Suchkriterien.
- **find_theorems** *(100) name:induct*
zeigt bis zu 100 Theoreme an.
- **print_theorems**
zeigt alle durch den vorherigen Befehl (z.B. **fun**) erzeugten Theoreme.

Oder:



The screenshot shows a software interface with a search bar containing the query `"_ ∨ _"` and a result count of 40. Below the search bar, the text `find_theorems` is displayed in blue. The main area shows the results of the search, starting with `found 562 theorem(s) (40 displayed):` followed by a list of theorems:

- `HOL.excluded_middle: $\neg ?P \vee ?P$`
- `HOL.disj_absorb: $(?A \vee ?A) = ?A$`
- `HOL.simp_thms(32): $(\text{False} \vee ?P) = ?P$`
- `HOL.simp_thms(30): $(\text{True} \vee ?P) = \text{True}$`
- `HOL.simp_thms(31): $(?P \vee \text{False}) = ?P$`
- `HOL.simp_thms(29): $(?P \vee \text{True}) = \text{True}$`
- `HOL.simp_thms(33): $(?P \vee ?P) = ?P$`
- `HOL.simp_thms(4): $(?P \vee \neg ?P) = \text{True}$`
- `HOL.simp_thms(5): $(\neg ?P \vee ?P) = \text{True}$`

At the bottom of the interface, there are tabs for `Query`, `Sledgehammer`, `State`, `Symbols`, and `Timing`.

Teil XVI

Induktive Prädikate und Mengen

Schlüsselwort: **inductive**, Syntax wie **fun**

Beispiel 1: Die geraden Zahlen als induktives Prädikat

```
inductive even :: "nat  $\Rightarrow$  bool"  
  where "even 0"  
  | "even n  $\implies$  even (n + 2)"
```

Schlüsselwort: **inductive**, Syntax wie **fun**

Beispiel 1: Die geraden Zahlen als induktives Prädikat

```
inductive even :: "nat  $\Rightarrow$  bool"  
  where "even 0"  
  | "even n  $\implies$  even (n + 2)"
```

Beispiel 2:

Welche Eigenschaft über Strings beschreibt folgendes Prädikat?

```
inductive foo :: "string  $\Rightarrow$  bool"  
  where "foo [c]"  
  | "foo [c,c]"  
  | "foo s  $\implies$  foo (c#s@[c])"
```

Schlüsselwort: **inductive**, Syntax wie **fun**

Beispiel 1: Die geraden Zahlen als induktives Prädikat

```
inductive even :: "nat  $\Rightarrow$  bool"  
  where "even 0"  
  | "even n  $\implies$  even (n + 2)"
```

Beispiel 2:

Welche Eigenschaft über Strings beschreibt folgendes Prädikat?

```
inductive foo :: "string  $\Rightarrow$  bool"  
  where "foo [c]"  
  | "foo [c,c]"  
  | "foo s  $\implies$  foo (c#s@[c])"
```

Der Parameterstring ist ein Palindrom!

Induktive Prädikate werden nicht rekursiv über Datentypen definiert, sondern über ein **Regelwerk**, normalerweise bestehend aus

- einer oder mehreren Basisregeln und
- einer oder mehreren induktiven Regeln, wobei das Prädikat in den Prämissen mit “kleineren” Parametern vorkommt (evtl. auch mehrfach).

Das Prädikat gilt für bestimmte Parameter, wenn es durch (endliche) Anwendung der Basis- und induktiven Regeln konstruiert werden kann

Jeder Regel kann einzeln ein Name gegeben werden:

```
inductive palin :: "string  $\Rightarrow$  bool"  
  where OneElem: "palin [c]"  
        | TwoElem: "palin [c,c]"  
        | HdLastRec: "palin s  $\implies$  palin (c#s@[c])"
```

Jeder Regel kann einzeln ein Name gegeben werden:

```
inductive palin :: "string  $\Rightarrow$  bool"  
  where OneElem: "palin [c]"  
        | TwoElem: "palin [c,c]"  
        | HdLastRec: "palin s  $\Longrightarrow$  palin (c#s@[c])"
```

Diese Regeln zusammengefasst als *palin.intros*
(allgemein *Prädikatname.intros*)
sieht wie folgt aus:

```
palin [?c]  
palin [?c, ?c]  
palin ?s  $\Longrightarrow$  palin (?c # ?s @ [?c])
```

Meist verwendet man jedoch die einzelnen Regelnamen.

Da das Prädikat aus Regeln aufgebaut wird ist eine “Fallunterscheidung” möglich, mit welcher Regel das Prädikat erzeugt wurde.

Diese Argumentation über den Regelaufbau heißt *Regelinversion*.

Die entsprechende Regel heißt *Prädikatname.cases* und wird mit der Taktik *cases* oder als Eliminationsregel (in automatischen Taktiken) verwendet:

Da das Prädikat aus Regeln aufgebaut wird ist eine “Fallunterscheidung” möglich, mit welcher Regel das Prädikat erzeugt wurde.

Diese Argumentation über den Regelaufbau heißt *Regelinversion*.

Die entsprechende Regel heißt *Prädikatname.cases* und wird mit der Taktik *cases* oder als Eliminationsregel (in automatischen Taktiken) verwendet:

Beispiel *palin.cases*:

$$\begin{aligned} & \llbracket \text{palin } ?a; \bigwedge c. ?a = [c] \implies ?P; \bigwedge c. ?a = [c, c] \implies ?P; \\ & \bigwedge s c. \llbracket ?a = c \# s @ [c]; \text{palin } s \rrbracket \implies ?P \rrbracket \implies ?P \end{aligned}$$

Da das Prädikat aus Regeln aufgebaut wird ist eine “Fallunterscheidung” möglich, mit welcher Regel das Prädikat erzeugt wurde.

Diese Argumentation über den Regelaufbau heißt *Regelinversion*.

Die entsprechende Regel heißt *Prädikatname.cases* und wird mit der Taktik *cases* oder als Eliminationsregel (in automatischen Taktiken) verwendet:

Beispiel *palin.cases*:

$$\llbracket \text{palin } ?a; \bigwedge c. ?a = [c] \implies ?P; \bigwedge c. ?a = [c, c] \implies ?P; \\ \bigwedge s c. \llbracket ?a = c \# s @ [c]; \text{palin } s \rrbracket \implies ?P \rrbracket \implies ?P$$

from $\langle \text{palin } s \rangle$ **have** "hd s = last s"

proof (*cases rule: palin.cases*)

liefert 3 Teilziele:

1. $\bigwedge c. s = [c] \implies \text{hd } s = \text{last } s$
2. $\bigwedge c. s = [c, c] \implies \text{hd } s = \text{last } s$
3. $\bigwedge sa c. \llbracket s = c \# sa @ [c]; \text{palin } sa \rrbracket \implies \text{hd } s = \text{last } s$

Oftmals ist Fallunterscheidung nicht genug und wir brauchen eine Induktionshypothese für Prädikate in der Prämisse einer Regel.

Dafür gibt es die Induktionsregel *Prädikatname.induct*.

Beispiel *palin.induct*:

$$\begin{aligned} & \llbracket \text{palin } ?x; \wedge c. ?P [c]; \wedge c. ?P [c, c]; \\ & \wedge s c. \llbracket \text{palin } s; ?P s \rrbracket \implies ?P (c \# s @ [c]) \rrbracket \implies ?P ?x \end{aligned}$$

Oftmals ist Fallunterscheidung nicht genug und wir brauchen eine Induktionshypothese für Prädikate in der Prämisse einer Regel.

Dafür gibt es die Induktionsregel *Prädikatname.induct*.

Beispiel *palin.induct*:

$$\llbracket \text{palin } ?x; \bigwedge c. ?P [c]; \bigwedge c. ?P [c, c]; \\ \bigwedge s c. \llbracket \text{palin } s; ?P s \rrbracket \implies ?P (c \# s @ [c]) \rrbracket \implies ?P ?x$$

from $\langle \text{palin } s \rangle$ **have** "hd s = last s "

proof (induction rule: *palin.induct*)

liefert Teilziele

1. $\bigwedge c. \text{hd } [c] = \text{last } [c]$
2. $\bigwedge c. \text{hd } [c, c] = \text{last } [c, c]$
3. $\bigwedge s c. \llbracket \text{palin } s; \text{hd } s = \text{last } s \rrbracket \\ \implies \text{hd } (c \# s @ [c]) = \text{last } (c \# s @ [c])$

Wechselseitigkeit ist auch bei induktiven Definitionen möglich und funktioniert analog zu wechselseitiger Rekursion.

Beispiel:

```
inductive even :: "nat  $\Rightarrow$  bool"  
  and odd :: "nat  $\Rightarrow$  bool"  
  where "even 0"  
    | "odd n  $\implies$  even (Suc n)"  
    | "even n  $\implies$  odd (Suc n)"
```

generiert Regeln *eval.cases*, *odd.cases*, *even_odd.induct* and *even_odd.inducts*

Wie bei **fun**: Erstere Induktionsregel benötigt \wedge -Verknüpfung von *even* und *odd* in Konklusion, zweite liefert *zwei* Regeln für zwei **and**-verbundene Lemmas.

Statt induktiver Präkate sind auch **induktive Mengen** möglich.
Das Schlüsselwort ist **inductive_set** und die Signatur verwendet
entsprechend `'a set` statt `'a \Rightarrow bool`.

Statt induktiver Präkate sind auch **induktive Mengen** möglich.
Das Schlüsselwort ist **inductive_set** und die Signatur verwendet
entsprechend `'a set` statt `'a \Rightarrow bool`.

Manchmal braucht man fixe Parameter, die beim Induktionsschritt der
Induktionsregel konstant bleiben.
Diese müssen nach **for** mit Namen und Signatur angegeben werden.

Beispiel: Reflexive, transitive Hülle

```
inductive rtc :: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a  $\Rightarrow$  'a  $\Rightarrow$  bool
for r :: "'a  $\Rightarrow$  'a  $\Rightarrow$  bool"
where refl: "rtc r a a"
      | trans: "[[ rtc r a b; r b c ] ]  $\implies$  rtc r a c"

rtc.induct: [[rtc ?r ?x ?y;  $\bigwedge$ a. ?P a a;
              $\bigwedge$ a b c. [[rtc ?r a b; ?P a b; ?r b c] ]  $\implies$  ?P a c]
             $\implies$ ?P ?x ?y
```

Statt induktiver Präkate sind auch **induktive Mengen** möglich.
Das Schlüsselwort ist **inductive_set** und die Signatur verwendet
entsprechend `'a set` statt `'a \Rightarrow bool`.

Manchmal braucht man fixe Parameter, die beim Induktionsschritt der
Induktionsregel konstant bleiben.
Diese müssen nach **for** mit Namen und Signatur angegeben werden.

Beispiel: Reflexive, transitive Hülle

```
inductive rtc :: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a  $\Rightarrow$  'a  $\Rightarrow$  bool
```

```
where refl: "rtc r a a"
```

```
| trans: "[[ rtc r a b; r b c ]  $\implies$  rtc r a c"
```

```
rtc.induct: [[rtc ?r ?x ?y;  $\bigwedge$ r a. ?P r a a;  
   $\bigwedge$ r a b c. [[rtc r a b; ?P r a b; r b c]  $\implies$  ?P r a c]  
   $\implies$ ?P ?r ?x ?y
```