Teil XIV

Theoreme finden

find_theorems & Co

Befehle, um Theoreme zu finden:

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

find_theorems & Co

Befehle, um Theoreme zu finden:

- **find_theorems**
  zeigt alle Theoreme an (wenig hilfreich).
- **find_theorems** *`length`*
  findet alle Theoreme zur Konstanten *`length`*.

find_theorems & Co

Befehle, um Theoreme zu finden:

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

find_theorems & Co

Befehle, um Theoreme zu finden:

- **find_theorems**
  zeigt alle Theoreme an (wenig hilfreich).
- **find_theorems** *length*
  findet alle Theoreme zur Konstanten *length*.
- **find_theorems** *name:classic*
  findet alle Theoreme mit *classic* im Namen.
- **find_theorems** *"?a ∨ (?b ∨ ?c)"*
  findet alle Theoreme, die den entsprechendend Term enthalten.

find_theorems & Co

Befehle, um Theoreme zu finden:

- **find_theorems**
  zeigt alle Theoreme an (wenig hilfreich).
- **find_theorems** *length*
  findet alle Theoreme zur Konstanten *length*.
- **find_theorems** *name:classic*
  findet alle Theoreme mit *classic* im Namen.
- **find_theorems** *"?a ∨ (?b ∨ ?c)"*
  findet alle Theoreme, die den entsprechendend Term enthalten.
- **find_theorems** *length name:induct*
  kombiniert die Suchkritieren.
- **print_theorems**
  zeigt alle durch den vorherigen Befehl (z.B. **fun**) erzeugten Theoreme.

find_theorems & Co

Befehle, um Theoreme zu finden:

- **find_theorems**
  zeigt alle Theoreme an (wenig hilfreich).
- **find_theorems** *length*
  findet alle Theoreme zur Konstanten *length*.
- **find_theorems** *name:classic*
  findet alle Theoreme mit *classic* im Namen.
- **find_theorems** *"?a ∨ (?b ∨ ?c)"*
  findet alle Theoreme, die den entsprechendend Term enthalten.
- **find_theorems** *length name:induct*
  kombiniert die Suchkritieren.
- **print_theorems**
  zeigt alle durch den vorherigen Befehl (z.B. **fun**) erzeugten Theoreme.
- **thm** *classical*
  zeigt das angegebene Lemma an.

Karlsruher Institut für Technologie

Teil XV

Induktive Prädikate und Mengen

Syntax und Beispiel

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

Beispiel 1: Die geraden Zahlen als induktives Prädikat

```
inductive even :: "nat ⇒ bool"
  where "even 0"
  | "even n ⟹ even (n + 2)"
```

Syntax und Beispiel

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

Beispiel 1: Die geraden Zahlen als induktives Prädikat

```
inductive even :: "nat ⇒ bool"
  where "even 0"
  | "even n ⟹ even (n + 2)"
```

Beispiel 2:

Welche Eigenschaft über Strings beschreibt folgendes Prädikat?

```
inductive foo :: "string ⇒ bool"
  where "foo [c]"
  | "foo [c,c]"
  | "foo s ⟹ foo (c#s@[c])"
```

Syntax und Beispiel

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

Beispiel 1: Die geraden Zahlen als induktives Prädikat

```
inductive even :: "nat ⇒ bool"
  where "even 0"
  | "even n ⟹ even (n + 2)"
```

Beispiel 2:

Welche Eigenschaft über Strings beschreibt folgendes Prädikat?

```
inductive foo :: "string ⇒ bool"
  where "foo [c]"
  | "foo [c,c]"
  | "foo s ⟹ foo (c#s@[c])"
```

Der Parameterstring ist ein Palindrom!

Induktive Prädikate

Induktive Prädikate werden nicht rekursiv über Datentypen definiert, sondern über ein Regelwerk, 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

Introduktionsregeln

Jeder Regel kann einzeln ein Name gegeben werden:

```
inductive palin :: "string ⇒ bool"
  where OneElem: "palin [c]"
  | TwoElem: "palin [c,c]"
  | HdLastRec: "palin s ⟹ palin (c#s@[c])"
```

Introduktionsregeln

Jeder Regel kann einzeln ein Name gegeben werden:

```
inductive palin :: "string ⇒ bool"
  where OneElem: "palin [c]"
  | TwoElem: "palin [c,c]"
  | HdLastRec: "palin s ⟹ palin (c#s@[c])"
```

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

```
palin [?c]
palin [?c, ?c]
palin ?s ⟹ palin (?c # ?s @ [?c])
```

Meist verwendet man jedoch die einzelnen Regelnamen.

Regelinversion

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:

Regelinversion

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`**:**

```
⟦palin ?a; ⋀c. ?a = [c] ⟹ ?P; ⋀c. ?a = [c, c] ⟹ ?P;
⋀s c. ⟦?a = c # s @ [c]; palin s⟧ ⟹ ?P⟧ ⟹ ?P
```

Regelinversion

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`**:**

```
⟦palin ?a; ⋀c. ?a = [c] ⟹ ?P; ⋀c. ?a = [c, c] ⟹ ?P;
⋀s c. ⟦?a = c # s @ [c]; palin s⟧ ⟹ ?P⟧ ⟹ ?P
```

from ‘`palin s`‘ **have** `"hd s = last s"`
proof`(cases rule: palin.cases)`

liefert 3 Teilziele:

```
1. ⋀c. s = [c] ⟹ hd s = last s
2. ⋀c. s = [c, c] ⟹ hd s = last s
3. ⋀sa c. ⟦s = c # sa @ [c]; palin sa⟧ ⟹ hd s = last s
```

Induktionsregel

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*:

```
⟦palin ?x; ⋀c. ?P [c]; ⋀c. ?P [c, c];
⋀s c. ⟦palin s; ?P s⟧ ⟹ ?P (c # s @ [c])⟧ ⟹ ?P ?x
```

Induktionsregel

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*:

```
⟦palin ?x; ⋀c. ?P [c]; ⋀c. ?P [c, c];
⋀s c. ⟦palin s; ?P s⟧ ⟹ ?P (c # s @ [c])⟧ ⟹ ?P ?x
```

from *‘palin s‘* **have** *"hd s = last s"*
proof *(induction rule: palin.induct)*

liefert Teilziele

```
1. ⋀c. hd [c] = last [c]
2. ⋀c. hd [c, c] = last [c, c]
3. ⋀s c. ⟦palin s; hd s = last s⟧
   ⟹ hd (c # s @ [c]) = last (c # s @ [c])
```

Induktiv wechselseitig

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

Beispiel:

```
inductive even :: "nat ⇒ bool"
      and odd :: "nat ⇒ bool"
  where "even 0"
  | "odd n ⟹ even (Suc n)"
  | "even n ⟹ 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, zweitere liefert *zwei* Regeln für zwei **and**-verbundene Lemmas.

Ausblick

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

Ausblick

Statt induktiver Präkate sind auch induktive Mengen möglich.
Das Schlüsselwort ist **inductive_set** und die Signatur verwendet entsprechend *'a set* statt *'a ⇒ 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 ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool
  for r :: "'a ⇒ 'a ⇒ bool"
   where refl: "rtc r a a"
   | trans: "⟦ rtc r a b; r b c ⟧ ⟹ rtc r a c"

rtc.induct: ⟦rtc ?r ?x ?y; ⋀a. ?P a a;
    ⋀a b c. ⟦rtc ?r a b; ?P a b; ?r b c⟧ ⟹ ?P a c⟧
   ⟹?P ?x ?y
```

Ausblick

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

   where refl: "rtc r a a"
   | trans: "⟦ rtc r a b; r b c ⟧ ⟹ rtc r a c"

rtc.induct: ⟦rtc ?r ?x ?y; ⋀r a. ?P r a a;
    ⋀r a b c. ⟦rtc r a b; ?P r a b; r b c⟧ ⟹ ?P r a c⟧
   ⟹?P ?r ?x ?y
```