Karlsruher Institut für Technologie

Theorembeweiserpraktikum

Anwendungen in der Sprachtechnologie

LEHRSTUHL PROGRAMMIERPARADIGMEN

Karlsruher Institut für Technologie

Teil I

Einleitung

Ziele des Praktikums

- Kennenlernen der Arbeit mit Theorembeweisern
- Erlernen des Beweisassistenten Isabelle/HOL
- Eigenständige Verifikation eines Projekts aus der Sprachtechnologie

Organisatorisches (I)

- Termin: Di, 14.00 – 15.30, Praktikumspool -143, Geb. 50.34
- Unterlagen: auf der Webseite
  `http://pp.ipd.kit.edu/lehre/SS2016/tba/`

- Master-Studenten:
  - Veranstaltung Teil der Module

[IN4INSPT]	Sprachtechnologien
[IN4INFM]	Formale Methoden
[IN4INPRAK2]	Informatik-Praktikum 2

- Diplom-Studenten:
  - Veranstaltung des Hauptstudium
  - Teil der Vertiefungsfächer

VF 1	Theoretische Grundlagen
VF 6	Softwaretechnik und Übersetzerbau

  - Veranstaltung ist prüfbar

Organisatorisches (II)

Das Praktikum teilt sich in 2 Hälften

- 1. Hälfte à 7 Veranstaltungen:
  - Übungsblätter
  - eigenständige Bearbeitung
  - Abgabe jeweils folgender Montag, 12.00 Uhr
  - Über `https://praktomat.cs.kit.edu/tba_2016_SS/`, SCC-Login
- 2. Hälfte:
  - Bearbeitung eines Projekts
  - in Zweiergruppen
  - Bearbeitungszeitraum: 31.5. – 11.7.2016, 12.00 Uhr
  - wieder über den Praktomaten
  - 12.7.2016: Informationen zur Projektpräsentation
  - letzter Termin 19.7.2016, **16.00 Uhr**, Projektpräsentation im Oberseminar
  - An den Dienstagsterminen Zeit für Fragen, Problembesprechung, etc. (bitte vorher Bescheid geben)

Was wird erwartet?

- Bearbeitung und Abgabe aller Übungsblätter (einzeln)
- Bearbeitung und Abgabe des Projekts als Zweiergruppe
- Anwesenheit an allen Übungsterminen, bei Projektvorstellung und -präsentation
- kurze Abschlusspräsentation im Oberseminar des Lehrstuhls
- keine schriftliche Ausarbeitungen

Teil II

Was ist ein Theorembeweiser?

Was ist ein Theorembeweiser?

Ein Theorembeweiser beweist Aussagen über formale Strukturen durch Anwendung von Regeln.

Was ist ein Theorembeweiser?

Karlsruher Institut für Technologie

Ein Theorembeweiser beweist Aussagen über formale Strukturen durch Anwendung von Regeln.

Was ist ein Theorembeweiser?

Ein Theorembeweiser beweist Aussagen über formale Strukturen durch Anwendung von Regeln.

- Typen und Datentypen (natürliche Zahlen, Listen, Paare, . . .)

Was ist ein Theorembeweiser?

Ein Theorembeweiser beweist Aussagen über formale Strukturen
durch Anwendung von Regeln.

- Typen und Datentypen (natürliche Zahlen, Listen, Paare, ...)
- Mengen, Relationen, Funktionen

Was ist ein Theorembeweiser?

Ein Theorembeweiser beweist Aussagen über formale Strukturen durch Anwendung von Regeln.

- Typen und Datentypen (natürliche Zahlen, Listen, Paare, . . .)
- Mengen, Relationen, Funktionen
- funktionale Programmierung ermöglicht selbstdefinierte Strukturen (durch Rekursion, Fallunterscheidung etc.)

Was ist ein Theorembeweiser?

Ein Theorembeweiser beweist Aussagen über formale Strukturen durch Anwendung von Regeln.

- Typen und Datentypen (natürliche Zahlen, Listen, Paare, . . .)
- Mengen, Relationen, Funktionen
- funktionale Programmierung ermöglicht selbstdefinierte Strukturen (durch Rekursion, Fallunterscheidung etc.)
- definiert im jeweiligen System!

Was ist ein Theorembeweiser?

Ein Theorembeweiser beweist Aussagen über formale Strukturen durch Anwendung von Regeln.

Was ist ein Theorembeweiser?

Ein Theorembeweiser beweist Aussagen über formale Strukturen durch Anwendung von Regeln.

automatisch

prozedural

deklarativ

Was ist ein Theorembeweiser?

Ein Theorembeweiser beweist Aussagen über formale Strukturen durch Anwendung von Regeln.

automatisch

- Theorembeweiser versucht Ziel eigenständig zu lösen
- bei Nichtgelingen Meldung, woran gescheitert und Abbruch
- Hilfslemmas zeigen und zum Beweisprozess hinzufügen
- nochmals versuchen, Ziel zu zeigen

prozedural

deklarativ

Was ist ein Theorembeweiser?

Ein Theorembeweiser beweist Aussagen über formale Strukturen durch Anwendung von Regeln.

automatisch

prozedural

- Taktiken für bestimmte automatisierte Prozesse
- können durch vorher gezeigte Hilfslemmas erweitert werden
- Beweisprozess wird nicht abgebrochen falls erfolglos, sondern an den Benutzer übergeben
- mittels Beweisskripten ‚Dirigieren' der Schlussfolgerung

deklarativ

Was ist ein Theorembeweiser?

Ein Theorembeweiser beweist Aussagen über formale Strukturen durch Anwendung von Regeln.

automatisch

prozedural

deklarativ

- Benutzer schreibt kompletten Beweis
- System prüft den Beweis
- nicht korrekte Schlussfolgerungen werden aufgezeigt

Was ist ein Theorembeweiser?

Ein Theorembeweiser beweist Aussagen über formale Strukturen durch Anwendung von Regeln.

Was ist ein Theorembeweiser?

Ein Theorembeweiser beweist Aussagen über formale Strukturen durch Anwendung von Regeln.

- Unifikation und Substitution

Was ist ein Theorembeweiser?

Ein Theorembeweiser beweist Aussagen über formale Strukturen durch Anwendung von Regeln.

- Unifikation und Substitution
- Simplifikation

Was ist ein Theorembeweiser?

Ein Theorembeweiser beweist Aussagen über formale Strukturen durch Anwendung von Regeln.

- Unifikation und Substitution
- Simplifikation
- (natürliche) Deduktion inkl. Quantoren

Was ist ein Theorembeweiser?

Ein Theorembeweiser beweist Aussagen über formale Strukturen durch Anwendung von Regeln.

- Unifikation und Substitution
- Simplifikation
- (natürliche) Deduktion inkl. Quantoren
- Induktion (natürlich, wohlfundiert, strukturell, Regel-Indunktion, . . .)

Was ist Unifikation?

- Verfahren, um zwei Terme identisch zu machen
- eventuell durch Ersetzen der schematischen Variablen durch Terme
- Anderes Beispiel: Pattern Matching

Beispiel: Unifikation der Terme ($?x, ?y, ?z$ Variablen, a Konstante)

$$f(?x, g(?x, ?x)) \qquad \text{und } f(h(?y), g(?z, h(a)))$$

Was ist Unifikation?

- Verfahren, um zwei Terme identisch zu machen
- eventuell durch Ersetzen der schematischen Variablen durch Terme
- Anderes Beispiel: Pattern Matching

Beispiel: Unifikation der Terme ($?x, ?y, ?z$ Variablen, a Konstante)

$$f(?x, g(?x, ?x)) \qquad \text{und } f(h(?y), g(?z, h(a)))$$

1. Schritt:

Was ist Unifikation?

- Verfahren, um zwei Terme identisch zu machen
- eventuell durch Ersetzen der schematischen Variablen durch Terme
- Anderes Beispiel: Pattern Matching

Beispiel: Unifikation der Terme ($?x, ?y, ?z$ Variablen, a Konstante)

$$f(h(?y), g(h(?y), h(?y))) \text{ und } f(h(?y), g(?z, h(a)))$$

1. Schritt: $?x = h(?y)$

Was ist Unifikation?

- Verfahren, um zwei Terme identisch zu machen
- eventuell durch Ersetzen der schematischen Variablen durch Terme
- Anderes Beispiel: Pattern Matching

Beispiel: Unifikation der Terme ($?x, ?y, ?z$ Variablen, a Konstante)

$$f(h(?y), g(h(?y), h(?y))) \text{ und } f(h(?y), g(?z, h(a)))$$

1. Schritt: $?x = h(?y)$
2. Schritt:

Was ist Unifikation?

- Verfahren, um zwei Terme identisch zu machen
- eventuell durch Ersetzen der schematischen Variablen durch Terme
- Anderes Beispiel: Pattern Matching

Beispiel: Unifikation der Terme ($?x, ?y, ?z$ Variablen, a Konstante)

$$f(h(?y), g(h(?y), h(?y))) \text{ und } f(h(?y), g(h(?y), h(a)))$$

1. Schritt: $?x = h(?y)$
2. Schritt: $?z = h(?y)$

Was ist Unifikation?

- Verfahren, um zwei Terme identisch zu machen
- eventuell durch Ersetzen der schematischen Variablen durch Terme
- Anderes Beispiel: Pattern Matching

Beispiel: Unifikation der Terme ($?x, ?y, ?z$ Variablen, a Konstante)

$$f(h(?y), g(h(?y), h(?y))) \text{ und } f(h(?y), g(h(?y), h(a)))$$

1. Schritt: $?x = h(?y)$
2. Schritt: $?z = h(?y)$
3. Schritt:

Was ist Unifikation?

- Verfahren, um zwei Terme identisch zu machen
- eventuell durch Ersetzen der schematischen Variablen durch Terme
- Anderes Beispiel: Pattern Matching

Beispiel: Unifikation der Terme ($?x, ?y, ?z$ Variablen, a Konstante)

$$f(h(a), g(h(a), h(a))) \qquad \text{und } f(h(a), g(h(a), h(a)))$$

1. Schritt: $?x = h(a)$
2. Schritt: $?z = h(a)$
3. Schritt: $?y = a$

Was ist Substitution?

Ersetzung von (logisch) äquivalenten Termen.

Regel:

$$\frac{s = t \qquad P[s/x]}{P[t/x]}$$

$P[t/x]$: ersetze x in P durch t

Eselsbrücke: Multiplikation mit einem Bruch: $(t/x) \cdot x = t$

Was ist Deduktion?

- meist Inferenzregeln (aus Prämissen folgt Konklusion)
- lassen sich in zwei Klassen aufteilen:

  Introduktion: wie erhalte ich diese Formel?
  Elimination: was kann ich aus dieser Formel folgern?

Was ist Deduktion?

- meist Inferenzregeln (aus Prämissen folgt Konklusion)
- lassen sich in zwei Klassen aufteilen:

Introduktion: wie erhalte ich diese Formel?
Elimination: was kann ich aus dieser Formel folgern?

Beispiel:

Introduktion von Konjunktion: $\dfrac{P \qquad Q}{P \wedge Q}$

Elimination von Konjunktion: $\dfrac{P \wedge Q \qquad \dfrac{P \qquad Q}{R}}{R}$

mehr in den Übungen!

Was ist Induktion?

Was ist Induktion?

natürliche Induktion:

zeige $P(0)$ und $P(n) \longrightarrow P(n+1)$

Was ist Induktion?

natürliche Induktion:

zeige $P(0)$ und $P(n) \longrightarrow P(n+1)$

strukturelle Induktion:

Induktion über rekursive Datentypen

Beispiel: Polymorphe Listen

datatype $'a\ list = [] \mid 'a \# ('a\ list)$

($[] =$ leere Liste, $\# =$ Konkatenation)

Induktion: zeige $P([])$ und $P(xs) \longrightarrow P(x \# xs)$

Was ist Induktion?

natürliche Induktion:

zeige $P(0)$ und $P(n) \longrightarrow P(n+1)$

strukturelle Induktion:

Induktion über rekursive Datentypen

Beispiel: Polymorphe Listen

datatype $'a\ list = [] \mid 'a \# ('a\ list)$

([] = leere Liste, # = Konkatenation)

Induktion: zeige $P([])$ und $P(xs) \longrightarrow P(x \# xs)$

wohlfundierte Induktion:

Induktion über Relationen

Beispiel: $<$ (auch: *starke Induktion*) $(\forall k < n.\ P(k)) \longrightarrow P(n)$

Was ist Induktion?

natürliche Induktion:

zeige $P(0)$ und $P(n) \longrightarrow P(n+1)$

strukturelle Induktion:

Induktion über rekursive Datentypen

Beispiel: Polymorphe Listen

datatype $'a\ list = [] \mid {'a} \# ({'a}\ list)$

([] = leere Liste, # = Konkatenation)

Induktion: zeige $P([])$ und $P(xs) \longrightarrow P(x \# xs)$

wohlfundierte Induktion:

Induktion über Relationen

Beispiel: $<$ (auch: *starke Induktion*) $(\forall k < n.\ P(k)) \longrightarrow P(n)$

mehr in den Übungen!

Fazit

Theorembeweiser sind mächtig, aber kein „goldener Hammer“!

- Kann Sicherheit bzgl. Aussagen beträchtlich erhöhen
- aber ‚schnell mal etwas formalisieren und beweisen‘ unmöglich
- meistens werden Aussagen über Kernprobleme formalisiert und bewiesen

Fragen

Sind „Papier und Bleistift“ Beweise nicht einfacher?

Fragen

Sind „Papier und Bleistift“ Beweise nicht einfacher?

- Formalisierung in Theorembeweiser braucht viel Formalisierungsarbeit, auch für scheinbar ‚triviale‘ Dinge

Fragen

Sind „Papier und Bleistift“ Beweise nicht einfacher?

- Formalisierung in Theorembeweiser braucht viel Formalisierungsarbeit, auch für scheinbar ‚triviale‘ Dinge
- doch Beweise von Hand enthalten oftmals Fehler, vor allem für komplexe Strukturen

Fragen

Sind „Papier und Bleistift“ Beweise nicht einfacher?

- Formalisierung in Theorembeweiser braucht viel Formalisierungsarbeit, auch für scheinbar ‚triviale‘ Dinge
- doch Beweise von Hand enthalten oftmals Fehler, vor allem für komplexe Strukturen
- Viel Aufwand, dafür garantierte Korrektheit!

Fragen

Wie kann ich sicher sein, dass meine Abstraktion das gewählte Problem beschreibt?

Fragen

Wie kann ich sicher sein, dass meine Abstraktion das gewählte Problem beschreibt?

- Im Allgemeinen: gar nicht!

Fragen

Wie kann ich sicher sein, dass meine Abstraktion das gewählte Problem beschreibt?

- Im Allgemeinen: gar nicht!
- Je genauer am konkreten Problem, desto größer die Sicherheit, aber ‚Formalisierungslücke' bleibt

Karlsruher Institut für Technologie

Teil III

Einführung in Isabelle/HOL

Isabelle

Karlsruher Institut für Technologie

Autoren: Larry Paulson, Tobias Nipkow, Markus (Makarius) Wenzel
Land: Großbritannien, Deutschland
Sprache: *SML*
Webseite: `isabelle.in.tum.de`

prozeduraler/deklarativer Beweiser
generisch, d.h. instantiierbar z.B. mit Typ- (HOL)
oder Mengentheorie (Zermelo-Fraenkel) (als *Objektlogiken*)

Beweiserstellung

- **prozedural** mittels unstrukturierten Taktikskripten („*apply-Skripten*“)
- **deklarativ** mittels strukturierten Isar Beweisskripten (nahe an üblicher mathematischer Notation)

Installation

Im Praktikumspool schon vorinstalliert unter
`/opt/Isabelle2016/bin/isabelle`

Für eigene Installation:

- Auf Seite `http://isabelle.in.tum.de/download.html` gehen
- Isabelle2016-Bundle herunterladen und installieren (ist erklärt)

Starten:

- *Isabelle-Pfad*`/bin/isabelle jedit`
- oder *Isabelle-Pfad*`/Isabelle2016`

Dateien in Isabelle

- Isabelle-Dateien haben die Endung `.thy`
- Eine Datei beginnt mit:
  theory *Dateiname* **imports** *Basisdateiname* (Standard: **Main**) **begin**
- Dann folgt die Formalisierung, die automatisch im Hintergrund geprüft wird
- Wichtig sind dabei die Informationen in den Fenstern „State“ und „Output“
- Das Ende der Datei wird mit **end** markiert

Regeln

- allgemeine Form: $P1 \Longrightarrow P2 \Longrightarrow P3 \Longrightarrow Q$
- $P1$, $P2$, $P3$ sind Prämissen der Regel (Annahmen)
- Q die Konklusion (Schlussfolgerung)
- $\Longrightarrow$ heißt “Meta-Implikation” und ist *rechts*-assoziativ.
- Also: “Wenn $P1$, $P2$ und $P3$, dann Q”
- Beispiel Modus Ponens: $P \longrightarrow Q \Longrightarrow P \Longrightarrow Q$

- alternative Form: $[\![P1;\ P2;\ P3]\!] \Longrightarrow Q$

  `Plugins`→`Plugin Options...`→`Isabelle`→`General`→`Print Mode=`*`brackets`*

Operatoren

Es gibt folgende logische Operatoren in Isabelle/HOL:

Name	*Anzeige*	*Sourcecode*	*jEdit-Kürzel*
Negation	$\neg$	\<not>	~
Konjunktion	$\wedge$	\<and>	/\
Disjunktion	$\vee$	\<or>	\/
Implikation	$\longrightarrow$	\<longrightarrow>	-->
Gleichheit	=		
Ungleichheit	$\neq$	\<noteq>	~=

Die jEdit-Kürzel werden evt. mit einem TAB abgeschlossen.

Achtung: $\longrightarrow$ und $\Longrightarrow$ sind verschieden

Introduktion und Elimination

- Jeder Operator besitzt eine Introduktionsregel, wobei der Operator in der Konklusion steht (Standardname $\ldots\mathtt{I}$)
„Was brauche ich, damit die Formel gilt?“
Beispiel: $\mathtt{conjI}$: $P \Longrightarrow Q \Longrightarrow P \wedge Q$

- Jeder Operator besitzt eine Eliminationsregel, wobei der Operator in der ersten Prämisse steht (Standardname $\ldots\mathtt{E}$)
„Was kann ich aus der Formel folgern?“
Beispiel: $\mathtt{conjE}$: $P \wedge Q \Longrightarrow (P \Longrightarrow Q \Longrightarrow R) \Longrightarrow R$

- Regeln kann man mittels **thm** ⟨*lemma-Name*⟩ anzeigen lassen

Aussagen

- In Isabelle werden zu zeigende Aussagen mit dem Schlüsselwort **lemma** eingeleitet (auch möglich: **corollary** und **theorem**)
- danach folgt optional ein Name, beendet durch :
- danach folgt die zu zeigende Aussage in Anführungszeichen

Beispiel:

lemma *imp_uncurry:* "$(P \longrightarrow (Q \longrightarrow R)) \longrightarrow P \wedge Q \longrightarrow R$"

Dem Lemma folgt dann der Beweis. . .

Beweis

- Ein Beweis beginnt mit **proof** `(rule` ⟨*Regel*⟩`)` und endet mit **qed**
- Dazwischen werden Zwischenschritte angegeben, in drei Varianten:
  - **assume** `"`⟨*Aussage*⟩`"` führt eine Aussage ein, die angenommen wird (also nicht bewiesen werden muss).
  - **have** `"`⟨*Aussage*⟩`"` ⟨*Beweis*⟩ führt eine bewiesene Hilfsaussage ein.
  - **show** `"`⟨*Aussage*⟩`"` ⟨*Beweis*⟩ beweist eine Aussage, die einen Fall des Beweises abschließt.
- Mehrere Fälle werden durch **next** getrennt

proof unifiziert die Konklusion der Regel mit der zu zeigenden Aussage. Die Prämissen der Regel sind die zu zeigenden Teilziele.

Zur Erinnerung:

$conjI\colon P \Longrightarrow Q \Longrightarrow P \wedge Q$

Beispiel:

lemma $"foo \wedge bar"$
proof `(rule conjI)`
  show `"foo"` ⟨*proof*⟩
next
  show `"bar"` ⟨*proof*⟩
qed

Annahmen aufsammeln

- Aussagen, die der **proof** von **have** und **show** direkt verwenden soll, werden mit **from** aufgezählt. Sie werden benannt durch
  - Die Aussage in Cartouches ‹*Aussage*› (`\<open>` bzw. `<<`, `\<close>` bzw. `>>`),
  - oder Namen, die der Aussage optional mit Doppelpunkt vorangestellt wurden oder
  - mit *this*, was stets die letzte gemachte Aussage ist.
- Diese werden, in der angegebenen Reihenfolge, mit den (Konklusionen der) Annahmen der Regel unifiziert
- Bei „**proof** -“ wird *keine* Regel angewandt.
- Bei **qed** dürfen nur noch Fälle offen sein, deren Ziele bereits unter den aufgesammelten Annahmen sind.

Beispiel:

from ‹$A \wedge B$›
have "A"
proof (*rule conjE*)
  assume "A"
  from *this*
  show "A".
qed

Zur Erinnerung:

conjE: $P \wedge Q \Longrightarrow (P \Longrightarrow Q \Longrightarrow R) \Longrightarrow R$

Abkürzungen

- **then** $\equiv$ **from** *this*
- **with** *a b* $\equiv$ **from** *a b this* (Reihenfolge beachten!)
- **by** *(rule* $\langle$*Regel*$\rangle$*)* $\equiv$ **proof** *(rule* $\langle$*Regel*$\rangle$*)* **qed**
- **proof** $\equiv$ **proof** *(rule)* (passende Regel wird automatisch gewählt)
- **..** $\equiv$ **by** *(rule)*
- **.** $\equiv$ **by**- $\equiv$ **proof**- **qed**

Abkürzungen

- **then** $\equiv$ **from** *this*
- **with** *a b* $\equiv$ **from** *a b this* (Reihenfolge beachten!)
- **by** *(rule* $\langle$*Regel*$\rangle$*)* $\equiv$ **proof** *(rule* $\langle$*Regel*$\rangle$*)* **qed**
- **proof** $\equiv$ **proof** *(rule)* (passende Regel wird automatisch gewählt)
- **..** $\equiv$ **by** *(rule)*
- **.** $\equiv$ **by**- $\equiv$ **proof**- **qed**

Und Abkürzungen anderer Art sind die Beweise

- **oops**: Bricht den aktuellen Beweis ab.
- **sorry**: Beweist alles (und sollte in fertigen Theorien nicht stehen).

Und jetzt Sie

Viel Spaß beim Ausprobieren!

Teil IV

Quantoren in Isabelle/HOL

Quantoren in Isabelle/HOL

Die üblichen zwei Quantoren der Logik:

Existenzquantor: ∃ (geschrieben `\<exists>`, Kürzel `?`), Syntax: $\exists x.\ P\ x$

Allquantor: ∀ (geschrieben `\<forall>`, Kürzel `!`), Syntax: $\forall x.\ P\ x$

Quantoren in Isabelle/HOL

Die üblichen zwei Quantoren der Logik:

Existenzquantor: $\exists$ (geschrieben `\<exists>`, Kürzel `?`), Syntax: $\exists$ `x. P x`

Allquantor: $\forall$ (geschrieben `\<forall>`, Kürzel `!`), Syntax: $\forall$ `x. P x`

Gültigkeitsbereich der gebundenen Variablen:
bis zum nächsten ; bzw. $\Longrightarrow$

Quantoren in Isabelle/HOL

Die üblichen zwei Quantoren der Logik:

Existenzquantor: $\exists$ (geschrieben `\<exists>`, Kürzel `?`), Syntax: $\exists\, x.\ P\ x$

Allquantor: $\forall$ (geschrieben `\<forall>`, Kürzel `!`), Syntax: $\forall\, x.\ P\ x$

Gültigkeitsbereich der gebundenen Variablen:
bis zum nächsten ; bzw. $\Longrightarrow$

Beispiele

$\forall\, x.\ P\ x \Longrightarrow Q\ x$ $\quad x$ in Konklusion nicht gebunden durch Allquantor

Quantoren in Isabelle/HOL

Die üblichen zwei Quantoren der Logik:

Existenzquantor: $\exists$ (geschrieben `\<exists>`, Kürzel `?`), Syntax: $\exists x.\ P\ x$

Allquantor: $\forall$ (geschrieben `\<forall>`, Kürzel `!`), Syntax: $\forall x.\ P\ x$

Gültigkeitsbereich der gebundenen Variablen:
bis zum nächsten ; bzw. $\Longrightarrow$

Beispiele

$\forall x.\ P\ x \Longrightarrow Q\ x$ x in Konklusion nicht gebunden durch Allquantor

$P\ y \Longrightarrow \exists y.\ P\ y$ y in Prämisse nicht gebunden durch Existenzquantor

Quantoren in Isabelle/HOL

Die üblichen zwei Quantoren der Logik:

Existenzquantor: $\exists$ (geschrieben `\<exists>`, Kürzel `?`), Syntax: $\exists x.\ P\ x$

Allquantor: $\forall$ (geschrieben `\<forall>`, Kürzel `!`), Syntax: $\forall x.\ P\ x$

Gültigkeitsbereich der gebundenen Variablen:
bis zum nächsten ; bzw. $\Longrightarrow$

Beispiele

$\forall x.\ P\ x \Longrightarrow Q\ x$ — x in Konklusion nicht gebunden durch Allquantor

$P\ y \Longrightarrow \exists y.\ P\ y$ — y in Prämisse nicht gebunden durch Existenzquantor

$\forall x.\ P\ x \Longrightarrow \exists x.\ Q\ x \Longrightarrow R$

Zwei verschiedene x in den Annahmen
gleichbedeutend mit $\forall y.\ P\ y \Longrightarrow \exists z.\ Q\ z \Longrightarrow R$
(*gebundene Namen sind Schall und Rauch*)

Quantoren in Isabelle/HOL

Die üblichen zwei Quantoren der Logik:

Existenzquantor: $\exists$ (geschrieben `\<exists>`, Kürzel `?`), Syntax: $\exists x.\ P\ x$

Allquantor: $\forall$ (geschrieben `\<forall>`, Kürzel `!`), Syntax: $\forall x.\ P\ x$

Gültigkeitsbereich der gebundenen Variablen:
bis zum nächsten ; bzw. $\Longrightarrow$

Beispiele

$\forall x.\ P\ x \Longrightarrow Q\ x$	x in Konklusion nicht gebunden durch Allquantor
$P\ y \Longrightarrow \exists y.\ P\ y$	y in Prämisse nicht gebunden durch Existenzquantor
$\forall x.\ P\ x \Longrightarrow \exists x.\ Q\ x \Longrightarrow R$	
	Zwei verschiedene x in den Annahmen gleichbedeutend mit $\forall y.\ P\ y \Longrightarrow \exists z.\ Q\ z \Longrightarrow R$ (*gebundene Namen sind Schall und Rauch*)
$\forall x.\ P\ x \longrightarrow Q\ x$	*gleiches* x für P und Q

Wie sagt man es Isabelle...?

Argumentation mit Quantoren erfordert Aussagen über *beliebige* Werte
Nur: Wie weiß Isabelle, dass ein Wert *beliebig* ist?

Wie sagt man es Isabelle...?

Argumentation mit Quantoren erfordert Aussagen über *beliebige* Werte
Nur: Wie weiß Isabelle, dass ein Wert *beliebig* ist?

Lösung: Meta-Logik

Syntax: $\bigwedge x.\ \ldots \Longrightarrow \ldots$

$\bigwedge$ heisst Meta-Allquantor, Variablen dahinter Parameter

Gültigkeitsbereich der Parameter: ganzes Teilziel

Beispiel: $\bigwedge x\ y.\ \forall y.\ P\ y \longrightarrow Q\ z\ y \Longrightarrow Q\ x\ y \Longrightarrow \exists x.\ Q\ x\ y$

entspricht $\bigwedge x\ y.\ \forall y_1.\ P\ y_1 \longrightarrow Q\ z\ y_1 \Longrightarrow Q\ x\ y \Longrightarrow \exists x_1.\ Q\ x_1\ y$

Wie sagt man es Isabelle...?

Argumentation mit Quantoren erfordert Aussagen über *beliebige* Werte
Nur: Wie weiß Isabelle, dass ein Wert *beliebig* ist?

Lösung: Meta-Logik

Syntax: $\bigwedge x.\ \ldots \Longrightarrow \ldots$

$\bigwedge$ heisst Meta-Allquantor, Variablen dahinter Parameter

Gültigkeitsbereich der Parameter: ganzes Teilziel

Beispiel: $\bigwedge x\ y.\ \forall y.\ P\ y \longrightarrow Q\ z\ y \Longrightarrow Q\ x\ y \Longrightarrow \exists x.\ Q\ x\ y$

entspricht $\bigwedge x\ y.\ \forall y_1.\ P\ y_1 \longrightarrow Q\ z\ y_1 \Longrightarrow Q\ x\ y \Longrightarrow \exists x_1.\ Q\ x_1\ y$

Auch die Meta-Implikation $\Longrightarrow$ ist Teil der Meta-Logik

Wie sagt man es Isabelle...?

Argumentation mit Quantoren erfordert Aussagen über *beliebige* Werte
Nur: Wie weiß Isabelle, dass ein Wert *beliebig* ist?

Lösung: Meta-Logik

Syntax: $\bigwedge x.\ \ldots \Longrightarrow \ldots$

$\bigwedge$ heisst Meta-Allquantor, Variablen dahinter Parameter

Gültigkeitsbereich der Parameter: ganzes Teilziel

Beispiel: $\bigwedge x\ y.\ \forall y.\ P\ y \longrightarrow Q\ z\ y \Longrightarrow Q\ x\ y \Longrightarrow \exists x.\ Q\ x\ y$

entspricht $\bigwedge x\ y.\ \forall y_1.\ P\ y_1 \longrightarrow Q\ z\ y_1 \Longrightarrow Q\ x\ y \Longrightarrow \exists x_1.\ Q\ x_1\ y$

Auch die Meta-Implikation $\Longrightarrow$ ist Teil der Meta-Logik

$\forall$ und $\longrightarrow$ entsprechen nicht $\bigwedge$ und $\Longrightarrow$, die ersten beiden nur in HOL!

Regeln

Karlsruher Institut für Technologie

Jeder Quantor hat Introduktions- und Eliminationsregel:

Regeln

Jeder Quantor hat Introduktions- und Eliminationsregel:

- `allI:` $(\bigwedge x.\ P\ x) \Longrightarrow \forall x.\ P\ x$
  Eine Aussage gilt für beliebige x (Meta-Ebene),
  also gilt sie auch für alle (HOL-Ebene)

Regeln

Jeder Quantor hat Introduktions- und Eliminationsregel:

- *allI:* $(\bigwedge x.\ P\ x) \Longrightarrow \forall x.\ P\ x$
  Eine Aussage gilt für beliebige x (Meta-Ebene),
  also gilt sie auch für alle (HOL-Ebene)
- *allE:* $\forall x.\ P\ x \Longrightarrow (P\ ?x \Longrightarrow R) \Longrightarrow R$
  Eine Aussage gilt für alle x, also folgt die Konklusion, wenn ich sie unter Verwendung der Aussage für einen (selbst wählbaren) Term zeigen kann

Regeln

Jeder Quantor hat Introduktions- und Eliminationsregel:

- *allI:* $(\bigwedge x.\ P\ x) \Longrightarrow \forall x.\ P\ x$
  Eine Aussage gilt für beliebige x (Meta-Ebene),
  also gilt sie auch für alle (HOL-Ebene)
- *allE:* $\forall x.\ P\ x \Longrightarrow (P\ ?x \Longrightarrow R) \Longrightarrow R$
  Eine Aussage gilt für alle x, also folgt die Konklusion, wenn ich sie unter Verwendung der Aussage für einen (selbst wählbaren) Term zeigen kann
- *exI:* $P\ ?x \Longrightarrow \exists x.\ P\ x$
  Eine Aussage gilt für einen Term $?x$, also gibt es ein x, wofür sie gilt

Regeln

Jeder Quantor hat Introduktions- und Eliminationsregel:

- $\mathtt{allI}$: $(\bigwedge x.\ P\ x) \Longrightarrow \forall x.\ P\ x$
  Eine Aussage gilt für beliebige x (Meta-Ebene),
  also gilt sie auch für alle (HOL-Ebene)
- $\mathtt{allE}$: $\forall x.\ P\ x \Longrightarrow (P\ ?x \Longrightarrow R) \Longrightarrow R$
  Eine Aussage gilt für alle x, also folgt die Konklusion, wenn ich sie unter Verwendung der Aussage für einen (selbst wählbaren) Term zeigen kann
- $\mathtt{exI}$: $P\ ?x \Longrightarrow \exists x.\ P\ x$
  Eine Aussage gilt für einen Term $?x$, also gibt es ein x, wofür sie gilt
- $\mathtt{exE}$: $\exists x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q) \Longrightarrow Q$
  Eine Aussage gilt für ein x, also folgt die Konklusion, wenn ich sie unter Verwendung der Aussage für einen beliebigen, nicht weiter bestimmten Term zeigen kann

Quantoren in Isar

Der Befehl **fix** korrespondiert mit $\bigwedge$.

have *"∀x. P x"*
proof *(rule allI)*
  fix *x*
  show *"P x"* ⟨*Beweis*⟩
qed

from ⟨∀x. P x⟩
have *"Q (f x)"*
proof *(rule allE)*
  fix *x*
  assume *"P (f x)"*
  then show *"Q (f x)"* ⟨*Beweis*⟩
qed

have *"∃x. P x"*
proof *(rule exI)*
  show *"P (f 0)"* ⟨*Beweis*⟩
qed

from ⟨∃x. P x⟩
have *"R"*
proof *(rule exE)*
  fix *x*
  assume *"P x"*
  show *"R"* ⟨*Beweis*⟩
qed

Letzteres geht auch ohne neuen Scope:

from ⟨∃x. P x⟩
obtain *x* **where** *"P x"* **by** *(rule exE)*

Teil V

Fallunterscheidung

Fallunterscheidung

In (klassischen) Beweisen Fallunterscheidung wichtiges Hilfsmittel

$$\frac{\dfrac{P}{Q} \qquad \dfrac{\neg P}{Q}}{Q}$$

Fallunterscheidung

In (klassischen) Beweisen Fallunterscheidung wichtiges Hilfsmittel

$$\dfrac{\dfrac{P}{Q} \qquad \dfrac{\neg P}{Q}}{Q}$$

In Isabelle: Mit der Regel *case_split*: $(P \Longrightarrow Q) \Longrightarrow (\neg P \Longrightarrow Q) \Longrightarrow Q$

Beispiel:

have *"BB ∨ ¬ BB"*
proof *(rule case_split)*
  assume *"BB"*
  then show *"BB ∨ ¬BB"*..
next
  assume *"¬ BB"*
  then show *"BB ∨ ¬BB"*..
qed

Statt **proof** *(rule case_split)* geht auch **proof** *(cases "BB")*. Vorteil: Die Annahmen sind gleich die richtigen (sonst erfährt Isabelle erst beim ersten **show** was *P* sein sollte – das klappt ggf. nicht zuverlässig).

Fallunterscheidung: Beispiel

Dieses Lemma wäre ohne Fallunterscheidung so nicht (einfach) lösbar!

```
lemma "B ∧ C ⟶ (A ∧ B) ∨ (¬ A ∧ C)"
proof
  assume "B ∧ C" then have "B"..
  from ‹B ∧ C›    have "C"..

  show "(A ∧ B) ∨ (¬ A ∧ C)"
  proof(cases A)
    assume A
    from ‹A› ‹B› have "A ∧ B"..
    then show ?thesis..
  next
    assume "¬A"
    from ‹¬A› ‹C› have "¬A ∧ C"..
    then show ?thesis..
  qed
qed
```

Teil VI

Definition

definition

Ermöglicht, einen Term zu benennen, so darüber zu abstrahieren und die Abstraktion gezielt zu öffnen

definition

Ermöglicht, einen Term zu benennen, so darüber zu abstrahieren und die Abstraktion gezielt zu öffnen

Beispiel:

definition *solution* :: *"nat"*
 where *"solution = 42"*

definition

Ermöglicht, einen Term zu benennen, so darüber zu abstrahieren und die Abstraktion gezielt zu öffnen

Beispiel:

definition *solution* :: *"nat"*
 where *"solution = 42"*

Erzeugt Regel: *solution_def: solution = 42*

definition

Ermöglicht, einen Term zu benennen, so darüber zu abstrahieren und die Abstraktion gezielt zu öffnen

Beispiel:

definition *solution* :: *"nat"*
where *"solution = 42"*

Erzeugt Regel: *solution_def: solution = 42*

So können auch Funktionen definiert werden:

Beispiel:

definition *nand* :: *"bool* $\Rightarrow$ *bool* $\Rightarrow$ *bool"*
where *"nand A B =* $(\neg\ (A \wedge B))$*"*

definition

Ermöglicht, einen Term zu benennen, so darüber zu abstrahieren und die Abstraktion gezielt zu öffnen

Beispiel:

definition *solution* :: *"nat"*
 where *"solution = 42"*

Erzeugt Regel: *solution_def: solution = 42*

So können auch Funktionen definiert werden:

Beispiel:

definition *nand* :: *"bool $\Rightarrow$ bool $\Rightarrow$ bool"*
 where *"nand A B = ($\neg$ (A $\wedge$ B))"*

Erzeugt Regel: *nand_def: nand A B = ($\neg$ (A $\wedge$ B))*

definition – Syntaxdefinition

`nand` ist binärer Operator

definition – Syntaxdefinition

`nand` ist binärer Operator
$\Longrightarrow$ Infixoperator bietet sich an

definition – Syntaxdefinition

`nand` ist binärer Operator
$\Longrightarrow$ Infixoperator bietet sich an

Syntaxdefinition (Infix-Notation)

Schreibe (**infixl** *"Operatorsymbol" n*) an die Deklarationszeile, wobei

- **infixl** für linksgebundenen Infixoperator steht,
  infixr für rechtsgebundene
- *Operatorsymbol* ein beliebig wählbares Symbol für den Operator ist,
- *n* eine Zahl ist, welche die Präzedenz dieses Operators angibt

definition – Syntaxdefinition

`nand` ist binärer Operator
$\Longrightarrow$ Infixoperator bietet sich an

Syntaxdefinition (Infix-Notation)

Schreibe (**infixl** "*Operatorsymbol*" *n*) an die Deklarationszeile, wobei

- **infixl** für linksgebundenen Infixoperator steht,
  infixr für rechtsgebundene
- *Operatorsymbol* ein beliebig wählbares Symbol für den Operator ist,
- *n* eine Zahl ist, welche die Präzedenz dieses Operators angibt

Beispiel: Operator `nand`

definition *nand* :: "*bool* $\Rightarrow$ *bool* $\Rightarrow$ *bool*" (**infixl** "$\bowtie$" *36*)
where "$A \bowtie B = (\neg\ (A \wedge B))$"

Jetzt: $A \bowtie B \bowtie C$ gleichbedeutend mit *nand (nand A B) C*

Teil VII
Gleichungen

Gleichungen

Das Arbeiten mit Gleichungen ist eine sehr wichtige Beweistechnik. Die Hauptregel dabei ist die Substitution:

$$\frac{s = t \qquad P[s/x]}{P[t/x]}$$

Diese Regel gibt es auch in Isabelle:

subst: $s = t \Longrightarrow P\ s \Longrightarrow P\ t$

Gleichungen

Das Arbeiten mit Gleichungen ist eine sehr wichtige Beweistechnik. Die Hauptregel dabei ist die Substitution:

$$\frac{s = t \qquad P[s/x]}{P[t/x]}$$

Diese Regel gibt es auch in Isabelle:

subst: $s = t \Longrightarrow P\ s \Longrightarrow P\ t$

Beispiel:

assume *"correct solution"*
with *solution_def*
have *"correct 42"* **by** *(rule subst)*

Auch nützlich:

ssubst: $s = t \Longrightarrow P\ t \Longrightarrow P\ s$

arg_cong: $x = y \Longrightarrow f\ x = f\ y$

Transitivität

Gleichheit ist transitiv, auch in Isabelle:

$trans\colon r = s \Longrightarrow s = t \Longrightarrow r = t$

Aber umständlich:

lemma *"foo = qux"*
proof *(rule trans)*
  show *"foo = bar"* ⟨*Beweis 1*⟩
next
  show *"bar = qux"*
  proof *(rule trans)*
    show *"bar = baz"* ⟨*Beweis 2*⟩
  next
    show *"baz = qux"* ⟨*Beweis 3*⟩
  qed
qed

Transitivität

Gleichheit ist transitiv, auch in Isabelle:

trans: $r = s \Longrightarrow s = t \Longrightarrow r = t$

Aber umständlich:

```
lemma "foo = qux"
proof (rule trans)
  show "foo = bar" ⟨Beweis 1⟩
next
  show "bar = qux"
  proof (rule trans)
    show "bar = baz" ⟨Beweis 2⟩
  next
    show "baz = qux" ⟨Beweis 3⟩
  qed
qed
```

Besser mit **also** und **finally**:

```
lemma "foo = qux"
proof -
  have "foo = bar" ⟨Beweis 1⟩
  also
  have "bar = baz" ⟨Beweis 2⟩
  also
  have "baz = qux" ⟨Beweis 3⟩
  finally show ?thesis.
qed
```

Gleichungsketten

Aber oft **proof** *(rule trans)* zu schreiben wäre sehr umständlich.
Statt dessen: Gleichungsketten!

- Die Befehle **also** und **finally** sollten jeweils einer Aussage (**assume** oder **have**) folgen, die eine Gleichung ist.
- Das abschließende **finally** kombiniert die Aussagen per Transitivität und stellt das Ergebnis (wie **from**) bereit.
- In Ausdrücken steht ... für die rechte Seite der letzten Aussage.
- Ist ein Lemma *foo* falsch herum, kann man *foo[symmetric]* verwenden.
- Die Abkürzung *?thesis* steht für die Konklusion des aktuell zu beweisende Lemmas (vor Anwendung von Regeln!).
  Typisches Muster:

```
proof -
  ⟨...⟩
  finally show ?thesis.
qed
```

Gleichungsketten (Beispiel)

```
lemma "(A ∧ (A ∨ B)) = A"
proof-
  from conj_disj_distribL
  have "(A ∧ (A ∨ B)) = ((A ∧ A) ∨ (A ∧ B))".
  also
  from conj_absorb
  have "((A ∧ A) ∨ (A ∧ B)) = (A ∨ (A ∧ B))" by (rule arg_cong)
  also
  have "(A ∨ (A ∧ B)) = A"
  proof
    ⟨a nested proof⟩
  qed
  finally
  show "(A ∧ (A ∨ B)) = A".
qed
```

Gleichungsketten (Beispiel mit Abkürzungen)

```
lemma "(A ∧ (A ∨ B)) = A"
proof-
  from conj_disj_distribL
  have "(A ∧ (A ∨ B)) = ((A ∧ A) ∨ (A ∧ B))".
  also
  from conj_absorb
  have "... = (A ∨ (A ∧ B))" by (rule arg_cong)
  also
  have "... = A"
  proof
    ⟨a nested proof⟩
  qed
  finally
  show ?thesis.
qed
```

Randbemerkung: moreover und ultimately

Die **also**..**finally**-Struktur hat einen kleinen Bruder: **moreover**..**ultimately**. Hier werden die Aussagen nicht per Transitivität verbunden, sondern einfach gesammelt.

Randbemerkung: moreover und ultimately

Die **also**..**finally**-Struktur hat einen kleinen Bruder: **moreover**..**ultimately**. Hier werden die Aussagen nicht per Transitivität verbunden, sondern einfach gesammelt.

Damit können Verschachtelungen vermieden werden, die Beweise natürlicher aufgebaut und Wiederholungen in **from**-Befehlen verringert werden:

have "$A \wedge B$"
proof
  show A
    ⟨*Beweis A*⟩
next
  show B
    ⟨*Beweis B*⟩
qed

have A
  ⟨*Beweis A*⟩
moreover
have B
  ⟨*Beweis B*⟩
ultimately
  have "$A \wedge B$"**..**

Teil VIII

Quick and Dirty: apply-Skripte Oder: Wie es wirklich geht!

Hintergrundwissen: Modi in Isabelle

Isabelle arbeitet im Wesentlichen mit 3 Modi

theory mode: Deklarationen, Definitionen, Lemmas
state mode: Zwischenaussagen, Fixes, etc.
prove mode: Anwendung von Beweistaktiken

Befehle schalten zwischen den Modi hin und her

Beispiele:

definition: *theory mode* → *theory mode*
lemma: *theory mode* → *prove mode*
proof: *prove mode* → *state mode*
assume: *state mode* → *state mode*
show: *state mode* → *prove mode*
by: *prove mode* → *state mode* | *theory mode*
qed: *state mode* → *state mode* | *theory mode*

apply-Skripte

apply: *prove mode* → *prove mode*

verändert das aktuelle Beweisziel (`subgoal`) durch Anwendung der gegebenen *Taktik(en)*

Beispiel:

```
lemma conjCommutes: "A ∧ B ⟹ B ∧ A"
apply (rule conjI)
 apply (erule conjE)
 apply assumption
apply (erule conjE)
apply assumption
done
```

done: *prove mode* → *state mode* | *theory mode*

beendet einen Beweis

Manuelle Taktiken

Beweisziel	ist immer das aktuell zu zeigende Ziel unter `subgoal`.
aktuelle Fakten	sind die gesammelten Fakten unter `this`.
gegebene ...	sind die Parameter der Taktik.

manuelle Taktiken

– (minus):	Fügt die aktuellen Fakten dem Beweisziel hinzu.
fact:	Setzt aus den gegebenen Fakten das Beweisziel zusammen (modulo Unifikation und schematischen Typ- und Termvariablen).
assumption:	Löst das Beweisziel, wenn eine passende Annahme vorhanden ist.
this:	Wendet die aktuellen Fakten der Reihe nach als Regel auf das Beweisziel an.
rule:	Wendet die gegebene(n) Regel(n) auf das Beweisziel an. Die aktuellen Fakten werden verwendet, um die Annahmen der Regel zu instanziieren.

weitere manuelle Taktiken

unfold: Ersetzt die gegebenen Definitionen in allen Beweiszielen.

fold: Kollabiert die gegebenen Definitionen in allen Beweiszielen.

insert: Fügt die gegebenen Fakten in alle Beweisziele ein.

erule: Wendet die gegebene Regel als Eliminationsregel an.

drule: Wendet die gegebene Regel als Destruktionsregel an.

frule: Wie *drule*, aber hält die verwendete Annahme im Beweisziel.

intro: Wendet die gegebenen Regeln **wiederholt** als Introduktionsregel an.

elim: Wendet die gegebenen Regeln **wiederholt** als Eliminationsregel an.

Introduktion und Elimination

Karlsruher Institut für Technologie

Introduktion: Operator steht in der Konklusion (Standardname $\ldots I$)
„Was brauche ich, damit die Formel gilt?“
Beispiel: $conjI$: $[\![P;\ Q]\!] \Longrightarrow P \wedge Q$

Was passiert?
Konklusionen der Regel und des Beweisziels werden unifiziert. Jede Prämisse der Regel wird als neues Beweisziel hinzugefügt.

Elimination: Operator steht in der ersten Prämisse (Standardname $\ldots E$)
„Was kann ich aus der Formel folgern?“
Beispiel: $conjE$: $[\![P \wedge Q;\ [\![P;\ Q]\!] \Longrightarrow R]\!] \Longrightarrow R$

Was passiert?
Konklusionen der Regel und des Beweisziels werden unifiziert. Dann wird die erste Prämisse der Regel mit der ersten passenden Prämisse des Beweisziels unifiziert. Die übrigen Prämissen der Regel werden als neue Beweisziele hinzugefügt.

Destruktion

Destruktion: Operator steht in der ersten Prämisse
„Ich benötige eine schwächere Aussage.“
Beispiel: `conjunct1`: $P \wedge Q \Longrightarrow P$

Was passiert?
Die erste Prämisse der Regel wird mit der ersten passenden Prämisse des Beweisziels unifiziert.
Die übrigen Prämissen der Regel werden als neue Beweisziele hinzugefügt.
Als letztes wird ein neues Beweisziel hinzugefügt, welches die Konklusion der Regel als Prämisse enthält.

Teil IX

Automatische Taktiken

Automatische Taktiken

Viele automatische Taktiken für den *logical reasoner*
Unterscheiden sich in

- der verwendeten Regelmenge,
- ob nur auf ein oder alle Zwischenziele angewendet,
- ob Abbruch bei nichtgelösten Zielen oder Rückgabe an Benutzer,
- ob der Simplifier mitverwendet wird oder nicht,
- ob mehr Zwischenziele erzeugt werden dürfen oder nicht

Im Folgenden werden ein paar Taktiken vorgestellt

Simplifikation

- Simplifikationsregeln: Gleichungen
- entsprechende Taktik: `simp` (nur erstes Ziel) bzw. `simp_all` (alle Ziele)
  - besitzt Pool an Termersetzungsregeln
  - prüft für jede solche Regel, ob Term mit linker Seite einer Gleichung unifizierbar
  - falls ja, ersetzen mit entsprechend unifizierter rechten Seite
- genauer: Termersetzung (weil Ausdruck rechts in der Gleichung nicht notwendigerweise einfacher)

Simplifikation

- Simplifikationsregeln: Gleichungen
- entsprechende Taktik: `simp` (nur erstes Ziel) bzw. `simp_all` (alle Ziele)
  - besitzt Pool an Termersetzungsregeln
  - prüft für jede solche Regel, ob Term mit linker Seite einer Gleichung unifizierbar
  - falls ja, ersetzen mit entsprechend unifizierter rechten Seite
- genauer: Termersetzung (weil Ausdruck rechts in der Gleichung nicht notwendigerweise einfacher)

Beispiel:

aktuelles Subgoal: $C \Longrightarrow P\ (\mathit{if}\ \mathit{False}\ \mathit{then}\ A\ \mathit{else}\ B \longrightarrow D)$

`simp` wendet folgende Termersetzungsregel an:

$\mathit{HOL.if\_False}\colon (\mathit{if}\ \mathit{False}\ \mathit{then}\ ?x\ \mathit{else}\ ?y) = ?y$

Simplifikation

- Simplifikationsregeln: Gleichungen
- entsprechende Taktik: `simp` (nur erstes Ziel) bzw. `simp_all` (alle Ziele)
  - besitzt Pool an Termersetzungsregeln
  - prüft für jede solche Regel, ob Term mit linker Seite einer Gleichung unifizierbar
  - falls ja, ersetzen mit entsprechend unifizierter rechten Seite
- genauer: Termersetzung (weil Ausdruck rechts in der Gleichung nicht notwendigerweise einfacher)

Beispiel:

aktuelles Subgoal: $C \Longrightarrow P\ (\mathit{if\ False\ then\ A\ else\ B} \longrightarrow D)$

`simp` wendet folgende Termersetzungsregel an:

$\mathit{HOL.if\_False}:\ (\mathit{if\ False\ then\ ?x\ else\ ?y}) = \mathit{?y}$

Resultat: $C \Longrightarrow P\ (B \longrightarrow D)$

Simplifikation

Auch bedingte Ersetzungsregeln sind möglich, also in der Form

$[\![\ldots]\!] \Longrightarrow \ldots = \ldots$

Dazu: Prämissen der Regel aus aktuellen Annahmen *via Simplifikation* herleitbar

Simplifikation

Auch bedingte Ersetzungsregeln sind möglich, also in der Form

$[\![\ldots]\!] \Longrightarrow \ \ldots\ = \ldots$

Dazu: Prämissen der Regel aus aktuellen Annahmen *via Simplifikation* herleitbar

Simplifier modifizieren:

- selbstgeschriebene Simplifikationslemmas zu Taktik hinzufügen:
  apply `(simp add:` *Regel*$_1$ *Regel*$_2$ ... `)`
- nur bestimmte Ersetzungsregeln verwenden:
  apply `(simp only:` *Regel*$_1$ *Regel*$_2$ ... `)`
- Ersetzungsregeln aus dem Standardpool von `simp` entfernen:
  apply `(simp del:` *Regel*$_1$ *Regel*$_2$ ... `)`

Simplifikation

Auch möglich: Ersetzungsregeln in den Standardpool von `simp` einfügen
Zwei Varianten:

- Zusatz *[simp]* hinter Lemmanamen
  Beispiel: **lemma** *bla [simp]: "A = True* $\Longrightarrow$ *A* $\wedge$ *B = B"*
- mittels **declare** *... [simp]*
  Beispiel: **declare** *foo [simp] bar [simp]*
  Analog: mittels **declare** *[simp del]* Ersetzungsregeln
  aus Standardpool entfernen
- Analog in einem **proof**-Block: Mit **have** resp. **note**.

Simplifikation

Auch möglich: Ersetzungsregeln in den Standardpool von `simp` einfügen
Zwei Varianten:

- Zusatz *[simp]* hinter Lemmanamen
  Beispiel: **lemma** *bla [simp]: "A = True* $\Longrightarrow$ *A* $\wedge$ *B = B"*
- mittels **declare** *... [simp]*
  Beispiel: **declare** *foo [simp] bar [simp]*
  Analog: mittels **declare** *[simp del]* Ersetzungsregeln
  aus Standardpool entfernen
- Analog in einem **proof**-Block: Mit **have** resp. **note**.

Vorsicht!

- Nur Regeln zu Standardpool hinzufügen, dessen rechte Seite einfacher als linke Seite!
- Sicherstellen, dass `simp` durch neue Regeln nicht in Endlosschleifen hängenbleibt!

`clarify` und `clarsimp`

- nur „offensichtliche“ logische Regeln
- nur auf oberstes Zwischenziel angewendet
- nach Vereinfachung Rückgabe an Benutzer
- `clarify` ohne Simplifier, `clarsimp` = `clarify` + `simp`
- kein Aufteilen in Zwischenziele

Oft verwendet um aktuelles Zwischenziel leichter lesbar zu machen

`blast`

- mächtige Regelmenge
- nur auf oberstes Zwischenziel angewendet
- versucht Ziel komplett zu lösen, ansonsten Abbruch
- kein Simplifier

Schnellster *logical reasoner*, gut auch bei Quantoren

`auto`

- mächtige Regelmenge, da basierend auf `blast`
- auf alle Zwischenziele angewendet
- nach Vereinfachung Rückgabe an Benutzer, wenn nicht gelöst
- mit Simplifier
- Aufteilen in Zwischenziele

oft verwendete „ad-hoc“-Taktik

`force` wie `auto`, nur sehr aggressives Lösen und Aufteilen,
deswegen anfällig für Endlosschleifen und Aufhängen

`fastforce`

- große Regelmenge
- nur auf oberstes Zwischenziel angewendet
- versucht Ziel komplett zu lösen, ansonsten Abbruch
- Simplifier, `fast` nur *logical reasoner*

`fastforce` ist eine in der Praxis oft gut brauchbare Taktik für das Lösen eines Zwischenziels

Regeln zu Taktiken hinzufügen

- Simplifikationsregeln: möglich bei allen Taktiken mit Simplifier
  normalerweise *simp:*Regelname, z.B. **apply** *(auto simp:bla)*
  bei *simp* und *simp_all* stattdessen *add:*, z.B. **apply** *(simp add:bla)*
  für Einschränken der Regelmenge: *simp del:* bzw. *simp only:*
  apply *(fastforce simp del:bla)* bzw. **apply** *(auto simp only:bla)*
- Deduktionsregeln: alle Taktiken mit *logical reasoner*
  Unterscheidung zwischen Introduktions-, Eliminations- und Destruktionsregeln
  - Introduktion: *intro:*, z.B. **apply** *(blast intro:foo)*
  - Elimination: *elim:*, z.B. **apply** *(auto elim:foo)*
  - Destruktion: *dest:*, z.B. **apply** *(fastforce dest:foo)*
- alles beliebig kombinierbar, z.B.

  apply *(auto dest:foo intro:bar simp:zip zap)*

- Resolutionskalkül
- nur auf oberstes Zwischenziel angewendet
- versucht Ziel komplett zu lösen, ansonsten Abbruch
- es müssen alle Regeln angegeben werden, die verwendet werden sollen

`metis` kann sehr gut mit Gleichungen umgehen, wenn diese nicht nur von links nach rechts angewendet werden sollen.

Teil X

Datentypen und primitive Rekursion

Algebraische Datentypen

Algebraische Datentypen werden über eine Menge von *Konstruktoren* beschrieben, die wiederum weitere Typen als *Parameter* haben.
Jeder Wert des Typs ist durch genau einen Konstruktor und den Werten dessen Parameters beschrieben.

Ein Parameter kann auch der definierte Datentyp selbst sein. In dem Fall spricht man von einem rekursiven Datentyp.

Algebraische Datentypen

Algebraische Datentypen werden über eine Menge von *Konstruktoren* beschrieben, die wiederum weitere Typen als *Parameter* haben.
Jeder Wert des Typs ist durch genau einen Konstruktor und den Werten dessen Parameters beschrieben.

Ein Parameter kann auch der definierte Datentyp selbst sein. In dem Fall spricht man von einem rekursiven Datentyp.

Beispiele:

- Eine natürliche Zahl ist entweder Null, oder der Nachfolger einer natürlichen Zahl
- Eine Liste ist entweder leer oder eine weitere List mit zusätzlichem Kopfelement.

Algebraische Datentypen

Algebraische Datentypen werden über eine Menge von *Konstruktoren* beschrieben, die wiederum weitere Typen als *Parameter* haben.
Jeder Wert des Typs ist durch genau einen Konstruktor und den Werten dessen Parameters beschrieben.

Ein Parameter kann auch der definierte Datentyp selbst sein. In dem Fall spricht man von einem rekursiven Datentyp.

Beispiele:

- Eine natürliche Zahl ist entweder Null, oder der Nachfolger einer natürlichen Zahl
- Eine Liste ist entweder leer oder eine weitere List mit zusätzlichem Kopfelement.

Formalisierung in Isabelle/HOL am Bsp. natürliche Zahlen:

datatype *nat = 0 | Suc nat*

Also Konstruktoren von nat: 0 und Suc

Parametertypen

Soll der Typ eines Konstruktorparameters nicht festgelegt sein, verwendet man den Parametertyp `'a`.

Parametertypen

Soll der Typ eines Konstruktorparameters nicht festgelegt sein, verwendet man den Parametertyp `'a`.

Beispiel: Listen mit Typparameter

```
datatype 'a list =
  Nil    ("[]")
| Cons 'a  "'a list"    (infixr "#" 65)
```

Konstruktoren von list: [] und # (Infix) (x#[] = [x])

Parametertypen

Soll der Typ eines Konstruktorparameters nicht festgelegt sein, verwendet man den Parametertyp `'a`.

Beispiel: Listen mit Typparameter

```
datatype 'a list =
  Nil    ("[]")
| Cons 'a  "'a list"    (infixr "#" 65)
```

Konstruktoren von list: [] und # (Infix) (x#[] = [x])

Damit kann man z.B. folgende Typen bilden:

```
foo :: "nat list ⇒ bool"
bar :: "nat ⇒ bool list ⇒ nat"
baz :: "'a list ⇒ 'a"
```

Rekursive Funktionen

Definition von Funktionen über rekursive Datentypen: **fun**
ein Parameter der Funktion kann dabei in seine Konstruktoren aufgeteilt werden.
Auf der rechten Seite der Gleichungen sollte die definierte Funktion höchstens auf die Parameter des Konstruktors angewandt werden.

Beispiel:

```
fun length :: "'a list ⇒ nat"
  where "length [] = 0"
  | "length (x#xs) = Suc (length xs)"

fun tl :: "'a list ⇒ 'a list"
  where "tl [] = []"
  |"tl (x#xs) = xs"
```

Rekursive Funktionen

Es müssen nicht alle Konstruktoren spezifiziert werden:

```
fun hd :: "'a list ⇒ 'a"
  where "hd (x#xs) = x"

fun last :: "'a list ⇒ 'a"
  where "last (x#xs) = (if xs=[] then x else last xs)"
```

Bei nicht enthaltenen Konstruktoren wie z.B. *hd []* nimmt die Funktion den Wert *undefined* an, ein fester Wert in jedem Typ, über den nichts bekannt ist.

@ und rev

weiterer Infixoperator: `@` hängt Listen zusammen
Beispiel: *`[0,4] @ [2] = [0,4,2]`*

@ und rev

weiterer Infixoperator: `@` hängt Listen zusammen
Beispiel: `[0,4] @ [2] = [0,4,2]`

`rev` dreht Listen um, also `rev [0,4,2] = [2,4,0]`

Wie lautet die entsprechende Definition?

@ und rev

weiterer Infixoperator: `@` hängt Listen zusammen
Beispiel: *[0,4] @ [2] = [0,4,2]*

rev dreht Listen um, also *rev [0,4,2] = [2,4,0]*

Wie lautet die entsprechende Definition?

```
fun  rev :: "'a list ⇒ 'a list"
  where "rev [] = []"
  | "rev (x#xs) = rev xs @ [x]"
```

Fallunterscheidung

„Wenn eine Aussage für jeden möglichen Konstruktor gilt, dann auch für alle Werte des Typs.“

Regel für Listen:

list.exhaust: $(xs = [] \Longrightarrow P) \Longrightarrow (\bigwedge a\ as.\ xs = a\ \#\ as \Longrightarrow P) \Longrightarrow P$

Fallunterscheidung

„Wenn eine Aussage für jeden möglichen Konstruktor gilt, dann auch für alle Werte des Typs.“

Regel für Listen:

list.exhaust: $(xs = [] \Longrightarrow P) \Longrightarrow (\bigwedge a\ as.\ xs = a \# as \Longrightarrow P) \Longrightarrow P$

Wird meist mit der Beweistaktik *cases* verwendet:

```
lemma hd_Cons_tl: "xs ≠ [] ⟶ hd xs # tl xs = xs"
proof (rule impI)
  assume "xs ≠ []" show "hd xs # tl xs = xs"
  proof (cases xs)
    assume "xs = []"
    with ‹xs ≠ []› have False..
    then show ?thesis..
  next
    fix y ys
    assume "xs = y # ys"
    then show ?thesis by simp
  qed
qed
```

Fallnamen

Warum **proof** *(cases ..)* und nicht einfach **proof** *(rule List.exhaust)*?
Wegen benamter Fälle!

lemma *hd_Cons_tl: "xs ≠ [] ⟶ hd xs # tl xs = xs"*
proof *(rule impI)*
  assume *"xs ≠ []"*
  show *"hd xs # tl xs = xs"*
  proof *(cases xs)*
    case *Nil*
    with *⟨xs ≠ []⟩* **have** *False*..
    thus *?thesis*..
  next
    case *(Cons y ys)*
    then show *?thesis* **by** *simp*
  qed
qed

(Bei *cases* auf Aussagen heißen die Fälle *True* und *False*.)

Fallnamen

Warum **proof** *(cases ..)* und nicht einfach **proof** *(rule List.exhaust)*?
Wegen benamter Fälle!

lemma *hd_Cons_tl: "xs ≠ [] ⟶ hd xs # tl xs = xs"*
proof *(rule impI)*
  assume *"xs ≠ []"*
  show *"hd xs # tl xs = xs"*
  proof *(cases xs)*
    case *Nil*
    with *⟨xs ≠ []⟩* **have** *False*..
    thus *?thesis*..
  next
    case *(Cons y ys)*
    then show *?thesis* **by** *simp*
  qed
qed

(Bei *cases* auf Aussagen heißen die Fälle *True* und *False*.)
Hier geht natürlich auch einfach

lemma *hd_Cons_tl: "xs ≠ [] ⟹ hd xs # tl xs = xs"*
by *(cases xs)auto*

Strukturelle Induktion

Fallunterscheidung genügt bei rekursiven Datentypen nicht immer, weil man die Aussage schon für die rekursiven Parameter braucht. Dann hilft strukturelle Induktion

Strukturelle Induktion

Fallunterscheidung genügt bei rekursiven Datentypen nicht immer, weil man die Aussage schon für die rekursiven Parameter braucht. Dann hilft strukturelle Induktion

z.B. für Listen sieht die Regel wie folgt aus:

list.induct: $[\![P\ [];\ \bigwedge x\ xs.\ P\ xs \Longrightarrow P\ (x\ \#\ xs)]\!] \Longrightarrow P\ xs$

Strukturelle Induktion

Fallunterscheidung genügt bei rekursiven Datentypen nicht immer, weil man die Aussage schon für die rekursiven Parameter braucht. Dann hilft strukturelle Induktion

z.B. für Listen sieht die Regel wie folgt aus:

list.induct: $\llbracket P\ []; \bigwedge x\ xs.\ P\ xs \Longrightarrow P\ (x\ \#\ xs)\rrbracket \Longrightarrow P\ xs$

Komfortabler als *rule* ist die Methode *induction*:

- Benamte Fälle wie bei *cases*
- *Fall.IH* ist die Induktionshypothese, also die Aussage für die rekursiven Parameter
- ggf. *Fall.prems* sind die Annahmen, wie sie im Lemma stehen.
- ggf. *Fall.hyps* sind zusätzliche Aussagen, die durch die Induktionsregel eingefügt werden.

Beispiel für strukturelle Induktion

```
lemma "length (xs @ ys) = length xs + length ys"
proof(induction xs)
case Nil
  have "length ([] @ ys) = length ys" by simp
  also have "... = 0 + length ys" by simp
  also have "... = length [] + length ys" by simp
  finally show ?case.
next
case (Cons x xs)
  have "length ((x # xs) @ ys) = length (x # (xs @ ys))" by simp
  also have "... = Suc (length (xs @ ys))" by simp
  also from Cons.IH have "... = Suc (length xs + length ys)"..
  also have "... = Suc (length xs) + length ys" by simp
  also have "... = length (x # xs) + length ys" by simp
  finally show ?case.
qed
```

Beispiel für strukturelle Induktion

```
lemma "length (xs @ ys) = length xs + length ys"
proof(induction xs)
case Nil
  have "length ([] @ ys) = length ys" by simp
  also have "... = 0 + length ys" by simp
  also have "... = length [] + length ys" by simp
  finally show ?case.
next
case (Cons x xs)
  have "length ((x # xs) @ ys) = length (x # (xs @ ys))" by simp
  also have "... = Suc (length (xs @ ys))" by simp
  also from Cons.IH have "... = Suc (length xs + length ys)"..
  also have "... = Suc (length xs) + length ys" by simp
  also have "... = length (x # xs) + length ys" by simp
  finally show ?case.
qed
```

oder natürlich einfach

```
by(induction xs) auto
```

Probleme mit Induktion

Problem: zu spezielle Induktionshypothesen

lemma *"rev xs = rev ys* $\Longrightarrow$ *xs = ys"*
proof *(induction xs)*

- Fall *[]* mit *simp* lösbar:
  case *Nil* **then show** *?case* **by** *simp*

Probleme mit Induktion

Problem: zu spezielle Induktionshypothesen

lemma *"rev xs = rev ys $\Longrightarrow$ xs = ys"*
proof *(induction xs)*

- Fall *[]* mit *simp* lösbar:
  case *Nil* **then show** *?case* **by** *simp*
- bei Induktionsschritt bekommen wir:
  Cons.IH: rev xs = rev ys $\Longrightarrow$ xs = ys
  Cons.prems: rev (x # xs) = rev ys
  aber *rev ys* kann nicht gleichzeitig *rev xs* und *rev (a # xs)* sein!

  $\Rightarrow$ Induktionshypothese nicht verwendbar

  $\Rightarrow$ So kommen wir nicht weiter.

Probleme mit Induktion

Lösung:

ys muss in der Induktionshypothese eine freie Variable sein!

Probleme mit Induktion

Lösung:

ys muss in der Induktionshypothese eine freie Variable sein!

Umsetzung:

ys nach Schlüsselwort *arbitrary* in der Induktionsanweisung. Damit wird die Induktionshypothese für *ys* meta-allquantifiziert:

lemma *"rev xs = rev ys ⟹ xs = ys"*
proof *(induction xs arbitrary: ys)*

Probleme mit Induktion

Lösung:
ys muss in der Induktionshypothese eine freie Variable sein!

Umsetzung:
ys nach Schlüsselwort *arbitrary* in der Induktionsanweisung. Damit wird die Induktionshypothese für *ys* meta-allquantifiziert:

lemma *"rev xs = rev ys $\Longrightarrow$ xs = ys"*
proof *(induction xs arbitrary: ys)*

Nun liefert uns
case *(Cons x xs ys)*
diese Aussagen:

Cons.IH: rev xs = rev ?ys $\Longrightarrow$ xs = ?ys

Cons.prems: rev (x # xs) = rev ys

wobei die Variable mit dem Fragezeichen frei ist, also mit jeder beliebigen Liste verwendet werden kann.

Probleme mit Induktion – Lösungsideen

Heuristiken für (bisher scheiternde) Induktionen:

- alle freien Variablen (außer Induktionsvariable) mit *`arbitrary`*
- Induktion immer über das Argument, über das die Funktion rekursiv definiert ist
- Generalisiere zu zeigendes Ziel: Ersetze Konstanten durch Variablen

Teil XI

Eigene Lemmas als Regeln

Lemmas und Regeln

Man kann bewiesene Lemmas als Regeln verwenden!
(Wenig überraschend)

```
lemma mylemma: "even (42::nat)" by simp

lemma "∃x. even (x::nat)"
proof
  show "even (42::nat)" by (rule mylemma)
qed
```

Lemmas und Regeln: Annahmen

Man kann bewiesene Lemmas als Regeln verwenden!
Dabei sollten Annahmen mit der Meta-Implikation angegeben werden.

lemma *mylemma2: "even n ⟹ even (3 * n)"* **by** *simp*

lemma *"even 126"*
proof-
  have *"even 42"* **by** *(rule mylemma)*
  then have *"even (3 * 42)"* **by** *(rule mylemma2)*
  also have *"3 * 42 = (126::nat)"* **by** *simp*
  finally show *?thesis*.
qed

Lemmas und Regeln: Annahmen vs. Isar

Man kann bewiesene Lemmas als Regeln verwenden!
Dabei sollten Annahmen mit der Meta-Implikation angegeben werden.
Das ist in Isar-beweisen nicht so schön (Annahmen müssen zweimal genannt werden):

```
lemma "even n ⟹ even (9 * n)"
proof-
  assume "even n"
  have "even (3 * (3 * n))"
  proof(rule mylemma2)
    from ‹even n›
    show "even (3 * n)" by (rule mylemma2)
  qed
  then show ?thesis by simp
qed
```

Lemmas und Regeln: Strukturierte Lemma

In Isar sind mit den Schlüsselwörter **assumes** und **shows** die Annahmen direkt verfügbar, sie können benannt werden und Attribute wie *[simp]* gesetzt werden.
Außerdem bezeichnet *assms* immer alle Annahmen.

```
lemma times9:
  assumes n_is_even: "even n"
  shows "even (9 * n)"
proof-
  have "even (3 * (3 * n))"
  proof(rule mylemma2)
    from n_is_even — oder from ‹even n› oder from assms
    show "even (3 * n)" by (rule mylemma2)
  qed
  then show ?thesis by simp
qed
```

Will man die Annahme(n) der ersten Beweismethode übergeben, so fügt man vor **proof** noch **using** *assms* ein. Wichtig bei Induktionsbeweisen!

Lemmas und Regeln: Freie Variablen

So wie **assumes** dem $\Longrightarrow$ entspricht, entspricht **fixes** dem $\bigwedge$.
Damit lassen sich die freien Variablen des Lemmas besser betonen und ihr Typ kann festgelegt werden:

```
lemma times9:
  fixes n :: nat
  assumes n_is_even: "even n"
  shows "even (9 * n)"
proof-
  from n_is_even
  have "even (3 * n)"  by (rule mylemma2)
  then have "even (3 * (3 * n))"  by (rule mylemma2)
  also have "3 * (3 * n) = 9 * n" by simp
  finally show ?thesis.
qed
```

Teil XII

Allgemeine Rekursion

Allgemeine Rekursion

Oftmals ist primitive Rekursion mit einer Regel pro Konstruktor zu einschränkend.

Manche rekursive Definitionen haben z.B. zwei Basisfälle oder brauchen Rekursion in mehr als einem Parameter.

Allgemeine Rekursion

Oftmals ist primitive Rekursion mit einer Regel pro Konstruktor zu einschränkend.

Manche rekursive Definitionen haben z.B. zwei Basisfälle oder brauchen Rekursion in mehr als einem Parameter.

Beispiel “mehrere Basisfälle”: Fibonacci-Zahlen

```
fun fib :: "nat ⇒ nat"
  where "fib 0 = 1"
  | "fib (Suc 0) = 1"
  | "fib (Suc (Suc n)) = fib n + fib (Suc n)"
```

Allgemeine Rekursion

Oftmals ist primitive Rekursion mit einer Regel pro Konstruktor zu einschränkend.

Manche rekursive Definitionen haben z.B. zwei Basisfälle oder brauchen Rekursion in mehr als einem Parameter.

Beispiel "mehrere Basisfälle": Fibonacci-Zahlen

```
fun fib :: "nat ⇒ nat"
  where "fib 0 = 1"
  | "fib (Suc 0) = 1"
  | "fib (Suc (Suc n)) = fib n + fib (Suc n)"
```

Beispiel "Rekursion in mehreren Parametern": Zippen von Listen

```
fun zip :: "'a list ⇒ 'b list ⇒ ('a × 'b) list"
  where "zip [] [] = []"
  | "zip (a#as) (b#bs) = (a,b)#zip as bs"
```

fun

fun definiert Funktionen durch *Pattern Matching*.

Dabei werden nur “lineare Patterns” unterstützt: Variablen dürfen auf den linken Seiten jeweils nur höchstens einmal vorkommen.

fun

fun definiert Funktionen durch *Pattern Matching*.

Dabei werden nur “lineare Patterns” unterstützt: Variablen dürfen auf den linken Seiten jeweils nur höchstens einmal vorkommen.

Es ist erlaubt, dass sich Pattern überlappen. Es wird die erste passende Regel angewandt.
Damit sind default-Regeln möglich, die alle restlichen Fälle behandeln.

fun

fun definiert Funktionen durch *Pattern Matching*.

Dabei werden nur "lineare Patterns" unterstützt: Variablen dürfen auf den linken Seiten jeweils nur höchstens einmal vorkommen.

Es ist erlaubt, dass sich Pattern überlappen. Es wird die erste passende Regel angewandt.
Damit sind default-Regeln möglich, die alle restlichen Fälle behandeln.

Beispiel: Separatorzeichen zwischen je zwei Elemente einer Liste

```
fun sep :: "'a ⇒ 'a list ⇒ 'a list"
where "sep a (x#y#zs) = x#a#sep a (y#zs)"
  | "sep a xs         = xs"
```

Simplifikationsregeln

In **fun** definierte Regeln landen im Simplifier, können auch direkt mit *Funktionsname.simps* angesprochen werden.

Simplifikationsregeln

In **fun** definierte Regeln landen im Simplifier, können auch direkt mit *Funktionsname.simps* angesprochen werden.

Beispiel: *fib.simps***:**

```
fib 0 = 1
fib (Suc 0) = 1
fib (Suc (Suc ?n)) = fib ?n + fib (Suc ?n)
```

Simplifikationsregeln

In **fun** definierte Regeln landen im Simplifier, können auch direkt mit *Funktionsname.simps* angesprochen werden.

Beispiel: *fib.simps*:

```
fib 0 = 1
fib (Suc 0) = 1
fib (Suc (Suc ?n)) = fib ?n + fib (Suc ?n)
```

Beispiel: *sep.simps*

```
sep ?a (?x # ?y # ?zs) = ?x # ?a # sep ?a (?y # ?zs)
sep ?a [] = []
sep ?a [?v] = [?v]
```

Beachte: Die Defaultregel (*sep a xs = xs*) generiert **zwei** Regeln, damit das Pattern-Matching vollständig ist.

Induktionsregeln

Analog definiert **fun** auch für jede Funktion eine Induktionsregel *Funktionsname.induct*

Diese kann man im Induktionsbeweis verwenden:

proof *(induction Funktionsparameter rule:Funktionsname.induct)*

Das nennt man *Regelinduktion*.

Induktionsregeln

Analog definiert **fun** auch für jede Funktion eine Induktionsregel
Funktionsname.induct
Diese kann man im Induktionsbeweis verwenden:

proof *(induction Funktionsparameter rule:Funktionsname.induct)*

Das nennt man *Regelinduktion*.

Beispiel: *sep.induct*

$[\![\bigwedge a\ x\ y\ zs.\ ?P\ a\ (y\ \#\ zs) \Longrightarrow ?P\ a\ (x\ \#\ y\ \#\ zs);\ \bigwedge a.\ ?P\ a\ [\,];$
$\bigwedge a\ v.\ ?P\ a\ [v]]\!] \Longrightarrow ?P\ ?a0.0\ ?a1.0$

Induktionsregeln

Analog definiert **fun** auch für jede Funktion eine Induktionsregel
Funktionsname.induct
Diese kann man im Induktionsbeweis verwenden:

proof *(induction Funktionsparameter rule:Funktionsname.induct)*

Das nennt man *Regelinduktion*.

Beispiel: *sep.induct*

$[\![\bigwedge a\ x\ y\ zs.\ ?P\ a\ (y\ \#\ zs) \Longrightarrow\ ?P\ a\ (x\ \#\ y\ \#\ zs);\ \bigwedge a.\ ?P\ a\ [];$
$\bigwedge a\ v.\ ?P\ a\ [v]]\!] \Longrightarrow\ ?P\ ?a0.0\ ?a1.0$

lemma *"map f (sep x ys) = sep (f x) (map f ys)"*
proof *(induction x ys rule:sep.induct)* generiert folgende 3 subgoals:

Induktionsregeln

Analog definiert **fun** auch für jede Funktion eine Induktionsregel
Funktionsname.induct
Diese kann man im Induktionsbeweis verwenden:

proof *(induction Funktionsparameter rule:Funktionsname.induct)*

Das nennt man *Regelinduktion*.

Beispiel: *sep.induct*

$\llbracket \bigwedge a\ x\ y\ zs.\ ?P\ a\ (y\ \#\ zs) \Longrightarrow ?P\ a\ (x\ \#\ y\ \#\ zs);\ \bigwedge a.\ ?P\ a\ [];$
$\bigwedge a\ v.\ ?P\ a\ [v] \rrbracket \Longrightarrow ?P\ ?a0.0\ ?a1.0$

lemma *"map f (sep x ys) = sep (f x) (map f ys)"*
proof *(induction x ys rule:sep.induct)* generiert folgende 3 subgoals:

1. $\bigwedge a\ x\ y\ zs.\ map\ f\ (sep\ a\ (y\ \#\ zs)) = sep\ (f\ a)\ (map\ f\ (y\ \#\ zs)) \Longrightarrow$
   $map\ f\ (sep\ a\ (x\ \#\ y\ \#\ zs)) = sep\ (f\ a)\ (map\ f\ (x\ \#\ y\ \#\ zs))$
2. $\bigwedge a.\ map\ f\ (sep\ a\ []) = sep\ (f\ a)\ (map\ f\ [])$
3. $\bigwedge a\ v.\ map\ f\ (sep\ a\ [v]) = sep\ (f\ a)\ (map\ f\ [v])$

Ausblick

fun ist sehr mächtig und in den meisten Fällen ausreichend, um rekursive Funktionen zu definieren.

Aber: auch mit **fun** kann es Probleme geben, z.B. bei wechselseitiger Rekursion oder falls **fun** die Termination nicht selbst beweisen kann.

Lösung: function
Braucht jedoch selbstgeschriebenen Vollständigkeits- und Terminationsbeweis...

Mehr dazu im **function**-Tutorial unter
`http://isabelle.in.tum.de/dist/Isabelle/doc/functions.pdf`

Teil XIII

Wechselseitige Rekursion

Wechselseitige Rekursion

Ein bisher ungelöstes Problem bei Rekursion: Was tun bei

- mehreren Datentypen, die sich gegenseitig oder
- Datentypen, die eine Liste ihres eigenen Typs

bei der Definition verwenden?

Wechselseitige Rekursion

Ein bisher ungelöstes Problem bei Rekursion: Was tun bei

- mehreren Datentypen, die sich gegenseitig oder
- Datentypen, die eine Liste ihres eigenen Typs

bei der Definition verwenden?

Beispiel:

Bäume mit beliebigem Verzweigungsgrad, d.h. jeder Knoten verwaltet eine Liste von Nachfolgerbäumen. Der Datentyp wird wie bisher definiert:

```
datatype 'a tree = Leaf 'a
  | Node 'a "'a tree list"
```

Damit ist der Datentyp wechselseitig rekursiv definiert für sich und Liste seiner selbst.

Wechselseitige Rekursion

Definition der Höhenfunktion für solche Bäume

Ansatz:

```
fun height :: "'a tree ⇒ nat"


  where "height (Leaf l) = 1"
  | "height (Node n ts) = ?
```

Wechselseitige Rekursion

Definition der Höhenfunktion für solche Bäume

Ansatz:

```
fun height :: "'a tree ⇒ nat"

  where "height (Leaf l) = 1"
  | "height (Node n ts) = heights ts + 1"
```

Wir brauchen eine Definition der Höhe für Liste von Bäumen!

Wechselseitige Rekursion

Definition der Höhenfunktion für solche Bäume

Ansatz:

```
fun height :: "'a tree ⇒ nat"
  and heights :: "'a tree list ⇒ nat"

  where "height (Leaf l) = 1"
  | "height (Node n ts) = heights ts + 1"

  | "heights [] = 0"
  | "heights (t#ts) = max (height t) (heights ts)"
```

Wir brauchen eine Definition der Höhe für Liste von Bäumen!

Dazu gleichzeitige Definition der Funktion *height* für *'a tree* und *heights* für *'a tree list*.

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Höhe jedes Baums größer als 1 ist

1. Versuch:

lemma

"height t > 0"

proof *(induction t)*

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Höhe jedes Baums größer als 1 ist
1. Versuch:

```
lemma
      "height t > 0"
proof (induction t)
```

komisches subgoal: `?P2.0 []`

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Höhe jedes Baums größer als 1 ist
1. Versuch:

lemma
```
      "height t > 0"
```
proof *(induction t)*

komisches subgoal: *?P2.0 []*
wir haben keine Aussage über die Höhe von Baumlisten!

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Höhe jedes Baums größer als 1 ist
2. Versuch:

lemma
    "height t > 0" **and** *"heights ts $\geq$ 0"*
proof *(induction t)*

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Höhe jedes Baums größer als 1 ist
2. Versuch:

lemma
    "height t > 0" **and** *"heights ts $\geq$ 0"*
proof *(induction t* **and** *ts)*

müssen beide Parameter getrennt durch **and** angeben

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Höhe jedes Baums größer als 1 ist
2. Versuch:

lemma
    "height t > 0" **and** *"heights ts* $\geq$ *0"*
proof *(induction t* **and** *ts)*

Problem: wegen generischem Elementtyp *'a* werden *t* und *ts* unterschiedliche Typen zugeordnet! *t::"'a tree", ts::"'b tree list"*

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Höhe jedes Baums größer als 1 ist

3. Versuch:

lemma fixes *t::"'a tree"* **and** *ts::"'a tree list"*
shows *"height t > 0"* **and** *"heights ts $\geq$ 0"*
proof *(induction t* **and** *ts)* **qed** *auto*

Wechselseitige Rekursion und arbitrary

Lemma für wechselseitige Rekursion hat so viele “Teillemmas” wie Datentypen rekursiv definiert.

Was passiert jedoch, wenn ein “Teillemma” nur gezeigt werden kann, wenn in der Induktionshypothese bestimmte Variablen allquantifiziert werden müssen?

Wir kennen die Lösung schon: **arbitrary**

Wechselseitige Rekursion und arbitrary

Lemma für wechselseitige Rekursion hat so viele “Teillemmas” wie Datentypen rekursiv definiert.

Was passiert jedoch, wenn ein “Teillemma” nur gezeigt werden kann, wenn in der Induktionshypothese bestimmte Variablen allquantifiziert werden müssen?

Wir kennen die Lösung schon: **arbitrary**

```
lemma "P t" "Q a t ts"
proof (induction t and ts)
```

`Q` braucht `a` in der Induktionshypothese quantifiziert, also in einem **arbitrary**

Wechselseitige Rekursion und arbitrary

Lemma für wechselseitige Rekursion hat so viele “Teillemmas” wie Datentypen rekursiv definiert.

Was passiert jedoch, wenn ein “Teillemma” nur gezeigt werden kann, wenn in der Induktionshypothese bestimmte Variablen allquantifiziert werden müssen?

Wir kennen die Lösung schon: **arbitrary**

lemma *"P t" "Q a t ts"*
proof *(induction t* **and** *ts* **arbitrary***:* **and** *a)*

`Q` braucht `a` in der Induktionshypothese quantifiziert, also in einem **arbitrary**
Auch hinter **arbitrary** werden die zu quantifizierenden Variablen für jedes Lemma mit **and** getrennt.

Teil XIV

Abkürzungen

let

Innerhalb eines Isar-Beweises können Terme **let**-gebunden werden.

Beispiel „Cantors Theorem“

theorem *"∃S. S ∉ range (f :: 'a ⇒ 'a set)"*
proof
  let *?S = "{x. x ∉ f x}"*
  show *"?S ∉ range f"*
  ...

Auch möglich: **def** führt eine neue (lokale) Konstante ein.

  def *S ≡ "{x. x ∉ f x}"*
Die definierende Gleichung von *S* heißt *S_def*.

type_synonym

Auch Typen kann man neu benamen:

type_synonym *'a env = "'a ⇒ nat"*

Ab jetzt sind *'a ⇒ nat* und *'a env* äquivalent/austauschbar.

Teil XV

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.

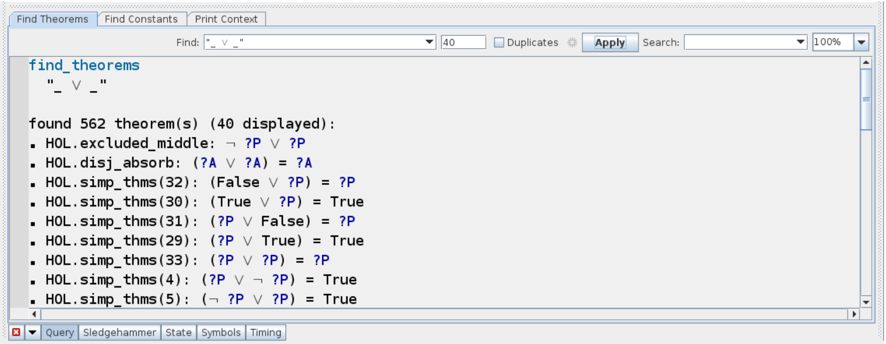
find_theorems & Co

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.
- **find_theorems** *(100) name:induct*
  zeigt bis zu 100 Theoreme an.
- **print_theorems**
  zeigt alle durch den vorherigen Befehl (z.B. **fun**) erzeugten Theoreme.

Query-Panel

Oder:



Karlsruher Institut für Technologie

Teil XVI

Induktive Prädikate und Mengen

Syntax und Beispiel

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

Beispiel 1: Die geraden Zahlen als induktives Prädikat

inductive *even :: "nat* $\Rightarrow$ *bool"*
  where *"even 0"*
  | *"even n* $\Longrightarrow$ *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`:

$\llbracket palin\ ?a;\ \bigwedge c.\ ?a = [c] \Longrightarrow\ ?P;\ \bigwedge c.\ ?a = [c, c] \Longrightarrow\ ?P;$
$\bigwedge s\ c.\ \llbracket ?a = c \# s\ @\ [c];\ palin\ s\rrbracket \Longrightarrow\ ?P\rrbracket \Longrightarrow\ ?P$

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

liefert 3 Teilziele:

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

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*:

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

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*:

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

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

liefert Teilziele

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

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 ∧-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 ⇒ bool*.

Ausblick

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

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 $refl$: "$rtc\ r\ a\ a$"
   | $trans$: "$\llbracket\ rtc\ r\ a\ b;\ r\ b\ c\ \rrbracket \Longrightarrow rtc\ r\ a\ c$"

$rtc.induct$: $\llbracket rtc\ ?r\ ?x\ ?y;\ \bigwedge a.\ ?P\ a\ a;$
  $\bigwedge a\ b\ c.\ \llbracket rtc\ ?r\ a\ b;\ ?P\ a\ b;\ ?r\ b\ c\rrbracket \Longrightarrow\ ?P\ a\ c\rrbracket$
  $\Longrightarrow ?P\ ?x\ ?y$

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
```

Karlsruher Institut für Technologie

Teil XVII

Verknüpfung von Beweismethoden

Hintereinanderausführung mit Backtracking

lemma "$\forall$ *x. P x* $\Longrightarrow$ $\forall$ *x. Q x* $\Longrightarrow$ *Q a*"
apply *(erule allE)*
by *assumption* $\longleftarrow$ geht schief!

stattdessen:
lemma "$\forall$ *x. P x* $\Longrightarrow$ $\forall$ *x. Q x* $\Longrightarrow$ *Q a*"
by *(erule allE, assumption)* $\longleftarrow$ geht gut!

, wendet Beweismethoden hintereinander an und verwendet dabei gemeinsamen Backtracking-Stack.

Weitere

Structural Composition

`apply` $(m_1;m_2)$ wendet m_2 auf alle Teilziele an, die durch die Anwendung von m_1 neu entstehen.

Alternative Choices

`apply` $(m_1|m_2)$ wendet m_1 an. (Nur) falls m_1 fehlschlägt wird m_2 angewendet.

Try

`apply` $m?$ wendet m an, bricht aber nicht ab falls Anwendung fehlschlägt.

Repeat

`apply` $m+$ wendet m mehrfach an (min. 1 mal), bis m fehlschlägt.

Restriction to subgoals

`apply` $m[n]$ wendet m an und schränkt den Beweiszustand dabei auf die ersten n Teilziele ein.

Teil XVIII

Automatische Beweissuche

Sledgehammer

Das Kommando **sledgehammer** führt eine automatische Beweissuche auf dem aktuellen Teilziel durch.

Optionen

timeout Setzt die maximale Suchdauer in Sekunden fest.
Beispiel *[timeout = 360]*

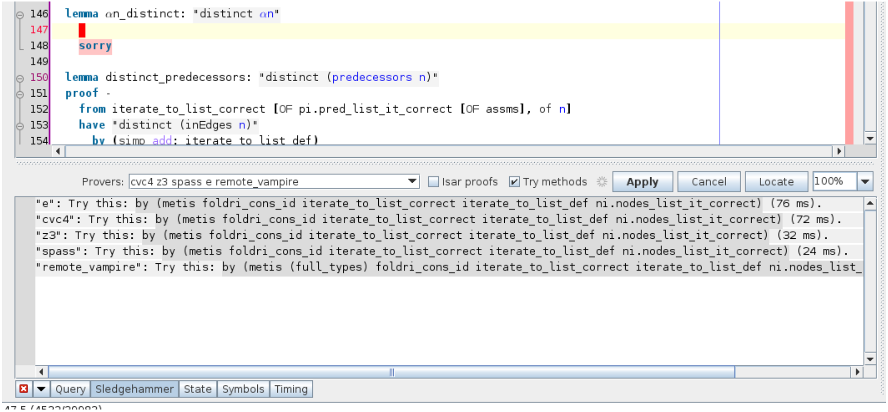
isar_proofs Erzeuge Isar-Beweise anstelle von Ein-Zeilern (verwendet nur ATPs keine SMT-Solver).

provers Verwende nur die angegebenen Beweiser.
Beispiel *[provers = e spass remote_vampire]*

Siehe auch: `isabelle doc sledgehammer`

Sledgehammer Panel

sledgehammer ist in Isabelle/jEdit auch als Panel verfügbar:



Teil XIX

Split-Regeln

`case`-**expressions**

Für jeden mit **datatype** erzeugten Typ T, wird auch eine Konstante $\texttt{case\_}T$ erzeugt, welche für *Pattern Matching* verwendet wird.

Beispiel

datatype $color = Red \mid Black$

datatype $'a\ rbt = Empty \mid Node\ color\ "'a\ rbt"\ 'a\ "'a\ rbt"$

liefert

$case\_color :: "'a \Rightarrow 'a \Rightarrow color \Rightarrow 'a"$

$case\_rbt :: "'a \Rightarrow (color \Rightarrow 'b\ rbt \Rightarrow 'b \Rightarrow 'b\ rbt \Rightarrow 'a) \Rightarrow 'b\ rbt \Rightarrow 'a"$

mit Syntax

$case\_color\ f\ g\ c \equiv$ **case** c **of** $Red \Rightarrow f \mid Black \Rightarrow g$

$case\_rbt\ f\ g\ t \equiv$ **case** t **of** $Empty \Rightarrow f \mid Node\ c\ l\ v\ r \Rightarrow g\ c\ l\ v\ r$

Simplifikation von `case`-expressions

Wie kann der Simplifier mit `case`-expressions umgehen?

Simplifikation von `case`-expressions

Wie kann der Simplifier mit `case`-expressions umgehen?

Lösung: Splitter

- Der *Splitter* ist Teil des Simplifiers.
- Wird aufgerufen, wenn Rewriting abgeschlossen.
- Wendet eine Split-Regel an (falls möglich).
- Startet Simplifikation auf dem/n neue/n Teilziel/en.

Split-Regel für `color`:

$$\mathit{color.split}\colon P\ (\textbf{case}\ \mathit{color}\ \textbf{of}\ \mathit{Red} \Rightarrow \mathit{f1} \mid \mathit{Black} \Rightarrow \mathit{f2})$$
$$\longleftrightarrow (\mathit{color} = \mathit{Red} \longrightarrow P\ \mathit{f1}) \wedge (\mathit{color} = \mathit{Black} \longrightarrow P\ \mathit{f2})$$

Split-Regeln

Split-Regeln gibt es in zwei Varianten:

für Konklusion: `?P (f ?a ?b ?c)` $\longleftrightarrow$ `?Q ?a ?b ?c`

- Splitter unifiziert linke Seite mit Teiltermen der Konklusion aus dem aktuellen Ziel.
- Ersetze unifizierten Teilterm durch rechte Seite.

für Annahmen: `?P (f ?a ?b ?c)` $\longleftrightarrow$ $\neg$ `?Q ?a ?b ?c`

- Splitter unifiziert linke Seite mit Teiltermen der Annahmen aus dem aktuellen Ziel.
- Ersetze unifizierten Teilterm durch rechte Seite
- und löse einige logische Verknüpfungen auf:
  - Top-Level Disjunktionen in `?Q` werden zu Teilzielen
  - Konjunktionen in `?Q` werden zu einzelnen Annahmen (des jeweiligen Teilziels)
  - Negationen von `?P` in `?Q` werden „richtig rumgedreht".

Split-Regeln – Beispiel

Split in Konklusion:

lemma fixes *n* :: *nat*
  shows *"even* (**case** *c* **of** *Red* ⇒ *2*n* | *Black* ⇒ *4*n)"*
by *(simp split: color.split)*

color.split: P (**case** *color* **of** *Red* ⇒ *f1* | *Black* ⇒ *f2)*
  ⟷ *(color = Red* ⟶ *P f1)* ∧ *(color = Black* ⟶ *P f2)*

Split-Regeln – Beispiel

Split in Konklusion:

lemma fixes *n* :: *nat*
  shows *"even* (**case** *c* **of** *Red* ⇒ *2*n* | *Black* ⇒ *4*n)"*
by *(simp split: color.split)*

Split in Annahme:

lemma fixes *n* :: *nat*
  assumes *"even* (**case** *c* **of** *Red* ⇒ *n* | *Black* ⇒ *m)"*
  shows *"even (n*m)"*
by *(simp split: color.split_asm)*

color.split: P (**case** *color* **of** *Red* ⇒ *f1* | *Black* ⇒ *f2*)
  $\longleftrightarrow$ (*color* = *Red* $\longrightarrow$ *P f1*) $\wedge$ (*color* = *Black* $\longrightarrow$ *P f2*)

color.split_asm: P (**case** *color* **of** *Red* ⇒ *f1* | *Black* ⇒ *f2*)
  $\longleftrightarrow$ $\neg$ (*color* = *Red* $\wedge$ $\neg$ *P f1* $\vee$ *color* = *Black* $\wedge$ $\neg$ *P f2*)

Split-Regeln – Beispiel

Split in Konklusion:

lemma fixes *n* :: *nat*
  shows "*even* (**case** *c* **of** *Red* ⇒ 2*n | *Black* ⇒ 4*n)"
by (*simp split: color.split*)

Split in Annahme:

lemma fixes *n* :: *nat*
  assumes "*even* (**case** *c* **of** *Red* ⇒ *n* | *Black* ⇒ *m*)"
  shows "*even* (n*m)"
by (*simp split: color.split_asm*)

color.split: P (**case** *color* **of** *Red* $\Rightarrow$ *f1* | *Black* $\Rightarrow$ *f2*)
$\longleftrightarrow$ (*color* = *Red* $\longrightarrow$ *P f1*) $\wedge$ (*color* = *Black* $\longrightarrow$ *P f2*)

color.split_asm: P (**case** *color* **of** *Red* $\Rightarrow$ *f1* | *Black* $\Rightarrow$ *f2*)
$\longleftrightarrow$ $\neg$ (*color* = *Red* $\wedge$ $\neg$ *P f1* $\vee$ *color* = *Black* $\wedge$ $\neg$ *P f2*)

color.splits = *color.split color.split_asm*

Noch mehr Split-Regeln

Split-Regeln gibt es nicht nur für *case*-expressions eines Datentyps, sondern für beliebige Funktionssymbole.

Beispiel: *if*

split_if:

$P\ (\mathit{if}\ Q\ \mathit{then}\ x\ \mathit{else}\ y) \longleftrightarrow (Q \longrightarrow P\ x) \wedge (\neg\ Q \longrightarrow P\ y)$

split_if_asm:

$P\ (\mathit{if}\ Q\ \mathit{then}\ x\ \mathit{else}\ y) \longleftrightarrow \neg\ (Q \wedge \neg\ P\ x \vee \neg\ Q \wedge \neg\ P\ y)$

Teil XX

Maps

Map

Map: partielle Abbildung, also rechte Seite undefiniert für manche linke Seite

Typen inklusive Undefiniertheit in Isabelle: *'a option*

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

- enthält alle Werte in *'a* mit *Some* vornangesetzt
- spezieller Wert *None* für Undefiniertheit

Beispiel: *bool option* besitzt die Elemente *None*, *Some True* und *Some False*

Map

also Maps in Isabelle von Typ *'a* nach *'b* haben Typ $'a \Rightarrow\ 'b$ *option* oder kurz $'a \rightharpoonup\ 'b$ ($\rightharpoonup$ ist `\<rightharpoonup>`, Kürzel `.>`)

- leere Map (also überall undefiniert): *empty*
- Update der Map M, so dass x auf y zeigt: $M(x \mapsto y)$ ($\mapsto$: $|-\ >$)
- Wert von x in Map M auf undefiniert setzen: $M(x := None)$
  (Hinweis: $M(x \mapsto y)$ entspricht $M(x := Some\ y)$)
- x hat in Map M Wert y, wenn gilt: $M\ x = Some\ y$
- x ist in Map M undefiniert, wenn gilt: $M\ x = None$
- um Map eigenen Typnamen zu geben: **type_synonym**
  Beispiel: **type_synonym** $nenv = nat \rightharpoonup bool$

Falls mehr Infos zu Maps nötig: *Isabelle-Verzeichnis*/src/HOL/Map.thy

Organisatorisches zum Projekt

- Projektbeginn: 31.5.2016 (heute)
- Bearbeitung in Zweierteams
- Isabelle-Rahmen zu den Projekten sind vorgegeben
- Abgabe: 11.7.2016, 12.00 Uhr via Praktomat
- ab jetzt: freiwillige Dienstagstermine mit weiterführendem Material
  Wünsche dazu gerne an uns.
- bei Problemen frühzeitig melden!
- Projektpräsentation: 19.7.2016, 16.00 Uhr (SR 010)
- Infos dazu: 12.7.2016, 14.00 Uhr (Poolraum -143)

Karlsruher Institut für Technologie

Teil XXI

Formale Semantik

Was ist Semantik?

Zwei Konzepte bei Programmiersprachen (analog zu natürlicher Sprache), *Syntax* und *Semantik*

Was ist Semantik?

Zwei Konzepte bei Programmiersprachen (analog zu natürlicher Sprache), *Syntax* und *Semantik*

Syntax:
- Regeln für korrekte Anordnung von Sprachkonstrukten
- In Programmiersprachen meist durch Grammatik, vor allem in BNF (Backus-Naur-Form) gegeben
- Angegeben im Sprachstandard

Was ist Semantik?

Zwei Konzepte bei Programmiersprachen (analog zu natürlicher Sprache), *Syntax* und *Semantik*

Syntax:
- Regeln für korrekte Anordnung von Sprachkonstrukten
- In Programmiersprachen meist durch Grammatik, vor allem in BNF (Backus-Naur-Form) gegeben
- Angegeben im Sprachstandard

Semantik:
- Bedeutung der einzelnen Sprachkonstrukte
- Bei Programmiersprachen verschiedenste Darstellungsweisen:
  - implizit (über eine Implementierung definiert)
  - informal (Beispiele, erläuternder Text etc.)
  - formal (Regelsysteme, Funktionen etc.)
- Angegeben im Sprachstandard (oft sehr vermischt mit Syntax)

Operationale Semantik

- Simuliert Zustandsübergänge auf abstrakter Maschine
- nahe an tatsächlichem Programmverhalten
- *Small-Step-Semantik*:
  Programm (= initiale Anweisung) + Startzustand wertet je einen Schritt zu Folgeprogramm + Folgezustand aus
  Syntax: $\langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle$
  Anweisung c in Zustand σ wertet zu Anweisung c' und Zustand σ' aus
- *Big-Step-Semantik*:
  Programm (= initiale Anweisung) + Startzustand wertet zu Endzustand aus
  Syntax: $\langle c, \sigma \rangle \Rightarrow \sigma'$ Anweisung c in Zustand σ liefert Endzustand σ'

einfache While-Sprache

arithmetische/boole'sche Ausdrücke: Zwei Werte `Intg` und `Bool`

- Konstanten `Val`
- Variablenzugriffe `Var`
- binäre Operatoren «*Eq*», «*And*», «*Less*», «*Add*» und «*Sub*» für ==, &&, $<$, $+$, $-$

einfache While-Sprache

arithmetische/boole'sche Ausdrücke: Zwei Werte $\mathtt{Intg}$ und $\mathtt{Bool}$

- Konstanten $\mathtt{Val}$
- Variablenzugriffe $\mathtt{Var}$
- binäre Operatoren «$\mathtt{Eq}$», «$\mathtt{And}$», «$\mathtt{Less}$», «$\mathtt{Add}$» und «$\mathtt{Sub}$» für ==, &&, $<$, $+$, $-$

Programmanweisungen:

- `Skip`
- Variablenzuweisung $x ::= e$
- sequentielle Komposition (Hintereinanderausführung) $c_1;; c_2$
- if-then-else $\mathtt{IF}\ (b)\ c_1\ \mathtt{ELSE}\ c_2$
- while-Schleifen $\mathtt{WHILE}\ (b)\ c'$

einfache While-Sprache

arithmetische/boole'sche Ausdrücke: Zwei Werte `Intg` und `Bool`

- Konstanten `Val`
- Variablenzugriffe `Var`
- binäre Operatoren «`Eq`», «`And`», «`Less`», «`Add`» und «`Sub`»
  für ==, &&, $<$, $+$, $-$

Programmanweisungen:

- `Skip`
- Variablenzuweisung $x ::= e$
- sequentielle Komposition (Hintereinanderausführung) $c_1 ;; c_2$
- if-then-else `IF` (b) c_1 `ELSE` c_2
- while-Schleifen `WHILE` (b) c'

Zustand:
beschreibt, welche Werte aktuell in den Variablen (Map)

Small-Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck *a* in Zustand σ
Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 `+` `true`)

Small-Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck a in Zustand σ
Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

$$\langle x ::= a, \sigma\rangle \rightarrow \langle \texttt{Skip}, \sigma(x \mapsto [\![a]\!]\sigma)\rangle$$

Small-Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck a in Zustand σ
Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

$$\langle x ::= a, \sigma\rangle \rightarrow \langle \texttt{Skip}, \sigma(x \mapsto [\![a]\!]\sigma)\rangle$$

$$\langle \texttt{Skip};;c, \sigma\rangle \rightarrow \langle c, \sigma\rangle \qquad \frac{\langle c, \sigma\rangle \rightarrow \langle c', \sigma'\rangle}{\langle c;;c'', \sigma\rangle \rightarrow \langle c';;c'', \sigma'\rangle}$$

Small-Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck *a* in Zustand σ

Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

$$\langle x ::= a, \sigma\rangle \to \langle \texttt{Skip}, \sigma(x \mapsto [\![a]\!]\sigma)\rangle$$

$$\langle \texttt{Skip};; c, \sigma\rangle \to \langle c, \sigma\rangle \qquad \frac{\langle c, \sigma\rangle \to \langle c', \sigma'\rangle}{\langle c;; c'', \sigma\rangle \to \langle c';; c'', \sigma'\rangle}$$

$$\frac{[\![b]\!]\sigma = \texttt{Some true}}{\langle \texttt{IF}\ (b)\ c\ \texttt{ELSE}\ c', \sigma\rangle \to \langle c, \sigma'\rangle} \qquad \frac{[\![b]\!]\sigma = \texttt{Some false}}{\langle \texttt{IF}\ (b)\ c\ \texttt{ELSE}\ c', \sigma\rangle \to \langle c', \sigma'\rangle}$$

Small-Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck a in Zustand σ
Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

$$\langle x ::= a, \sigma\rangle \to \langle \texttt{Skip}, \sigma(x \mapsto [\![a]\!]\sigma)\rangle$$

$$\langle \texttt{Skip};; c, \sigma\rangle \to \langle c, \sigma\rangle \qquad \frac{\langle c, \sigma\rangle \to \langle c', \sigma'\rangle}{\langle c;; c'', \sigma\rangle \to \langle c';; c'', \sigma'\rangle}$$

$$\frac{[\![b]\!]\sigma = \texttt{Some true}}{\langle \texttt{IF}\ (b)\ c\ \texttt{ELSE}\ c', \sigma\rangle \to \langle c, \sigma'\rangle} \qquad \frac{[\![b]\!]\sigma = \texttt{Some false}}{\langle \texttt{IF}\ (b)\ c\ \texttt{ELSE}\ c', \sigma\rangle \to \langle c', \sigma'\rangle}$$

$$\langle \texttt{WHILE}\ (b)\ c, \sigma\rangle \to \langle \texttt{IF}\ (b)\ c;; \texttt{WHILE}\ (b)\ c\ \texttt{ELSE}\ \texttt{Skip}, \sigma\rangle$$

Big Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck *a* in Zustand σ

Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

Big Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck *a* in Zustand σ
Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

$$\langle \texttt{Skip}, \sigma \rangle \Rightarrow \sigma$$

Big Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck a in Zustand σ
Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

$$\langle \texttt{Skip}, \sigma\rangle \Rightarrow \sigma \qquad \langle x ::= a, \sigma\rangle \Rightarrow \sigma(x \mapsto [\![a]\!]\sigma)$$

Big Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck a in Zustand σ
Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

$$\langle \texttt{Skip}, \sigma\rangle \Rightarrow \sigma \qquad \langle x ::= a, \sigma\rangle \Rightarrow \sigma(x \mapsto [\![a]\!]\sigma)$$

$$\frac{[\![b]\!]\sigma = \texttt{Some true} \qquad \langle c, \sigma\rangle \Rightarrow \sigma'}{\langle \texttt{IF}\ (b)\ c\ \texttt{ELSE}\ c', \sigma\rangle \Rightarrow \sigma'} \qquad \frac{[\![b]\!]\sigma = \texttt{Some false} \qquad \langle c', \sigma\rangle \Rightarrow \sigma'}{\langle \texttt{IF}\ (b)\ c\ \texttt{ELSE}\ c', \sigma\rangle \Rightarrow \sigma'}$$

Big Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck a in Zustand σ
Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

$$\langle \texttt{Skip}, \sigma\rangle \Rightarrow \sigma \qquad \langle x ::= a, \sigma\rangle \Rightarrow \sigma(x \mapsto [\![a]\!]\sigma)$$

$$\frac{[\![b]\!]\sigma = \texttt{Some true} \qquad \langle c, \sigma\rangle \Rightarrow \sigma'}{\langle \texttt{IF}\ (b)\ c\ \texttt{ELSE}\ c', \sigma\rangle \Rightarrow \sigma'} \qquad \frac{[\![b]\!]\sigma = \texttt{Some false} \qquad \langle c', \sigma\rangle \Rightarrow \sigma'}{\langle \texttt{IF}\ (b)\ c\ \texttt{ELSE}\ c', \sigma\rangle \Rightarrow \sigma'}$$

$$\frac{[\![b]\!]\sigma = \texttt{Some true} \qquad \langle c', \sigma\rangle \Rightarrow \sigma' \qquad \langle \texttt{WHILE}\ (b)\ c', \sigma'\rangle \Rightarrow \sigma''}{\langle \texttt{WHILE}\ (b)\ c', \sigma\rangle \Rightarrow \sigma''}$$

$$\frac{[\![b]\!]\sigma = \texttt{Some false}}{\langle \texttt{WHILE}\ (b)\ c', \sigma\rangle \Rightarrow \sigma}$$

Big Step Regeln

$[\![a]\!]\sigma$ Auswertung von arithm. oder boole'schem Ausdruck a in Zustand σ
Verwende Map, da Resultat undefiniert sein kann (z.B. bei 5 + `true`)

$$\langle \texttt{Skip}, \sigma \rangle \Rightarrow \sigma \qquad \langle x ::= a, \sigma \rangle \Rightarrow \sigma(x \mapsto [\![a]\!]\sigma)$$

$$\frac{[\![b]\!]\sigma = \texttt{Some true} \qquad \langle c, \sigma \rangle \Rightarrow \sigma'}{\langle \texttt{IF}\ (b)\ c\ \texttt{ELSE}\ c', \sigma \rangle \Rightarrow \sigma'} \qquad \frac{[\![b]\!]\sigma = \texttt{Some false} \qquad \langle c', \sigma \rangle \Rightarrow \sigma'}{\langle \texttt{IF}\ (b)\ c\ \texttt{ELSE}\ c', \sigma \rangle \Rightarrow \sigma'}$$

$$\frac{[\![b]\!]\sigma = \texttt{Some true} \qquad \langle c', \sigma \rangle \Rightarrow \sigma' \qquad \langle \texttt{WHILE}\ (b)\ c', \sigma' \rangle \Rightarrow \sigma''}{\langle \texttt{WHILE}\ (b)\ c', \sigma \rangle \Rightarrow \sigma''}$$

$$\frac{[\![b]\!]\sigma = \texttt{Some false}}{\langle \texttt{WHILE}\ (b)\ c', \sigma \rangle \Rightarrow \sigma} \qquad \frac{\langle c, \sigma \rangle \Rightarrow \sigma' \qquad \langle c', \sigma' \rangle \Rightarrow \sigma''}{\langle c;;c', \sigma \rangle \Rightarrow \sigma''}$$

Ausführungsdauer

Erweiterung der Auswertungsfunktionen für Ausdrücke und der Big-Step-Semantik um einen Zeitbegriff:

Konstanten:	1
Variablen:	1
je bin. Operation:	1
Skip:	1
LAss:	1 + Auswertung des arith. Ausdrucks
If:	1 + Auswertung des bool. Ausdrucks + Dauer des gew. Zweigs
While-False:	1 + Auswertung des bool. Ausdrucks
While-True:	1 + Auswertung des bool. Ausdrucks + Dauer für Rumpf + Rest-Dauer

Formalisierung in Isabelle

Siehe Rahmen

Teil XXII

Typsystem

Typsystem

Typsystem ordnet jedem Ausdruck Typ zu

Zwei Typen: `Boolean` und `Integer`

Typumgebung Γ: Map von Variablen nach Typ

Zwei Stufen:

1. jeder Ausdruck e bekommt unter Typumgebung Γ Typ T zugeordnet
   Syntax: $\Gamma \vdash e : T$
2. Anweisung c ist typbar unter Typumgebung Γ
   Syntax: $\Gamma \vdash c$

auch Typsystem definiert als induktive Menge

Regeln

Ausdrücke:

- Konstanten haben Typ des Werts
- Variablen haben den in Typumgebung gespeicherten Typ
- Operatoren haben, wenn Unterausdrücke Typen passend zu Operator, Typ des Resultats
  z.B. bei «*Less*»: Unterausdrücke *Integer*, ganzer Operator *Boolean*

Regeln

Ausdrücke:

- Konstanten haben Typ des Werts
- Variablen haben den in Typumgebung gespeicherten Typ
- Operatoren haben, wenn Unterausdrücke Typen passend zu Operator, Typ des Resultats
  z.B. bei «*Less*»: Unterausdrücke *Integer*, ganzer Operator *Boolean*

Anweisungen:

- `Skip` typt immer
- $x ::= e$ typt, wenn Typ der Variable x in Typumgebung Γ gleich Typ des Ausdruck e
- Sequenz typt, wenn beide Unteranweisungen typen
- `if` und `while` typen, wenn Unteranweisungen typen und Prädikat vom Typ *Boolean*

Formalisierung in Isabelle

Siehe Rahmen

Teil XXIII

Projekt: Konstantenfaltung und -propagation

Motivation

- Konstantenfaltung und -propagation sind wichtige Optimierungen in Compilern
- verringern *Registerdruck* (Anzahl der benötigten Register)
- Korrektheit dieser Optimierungen essentiell
- Korrektheit zu zeigen bzgl. formaler Semantik
- wir beschränken uns auf Konstantenfaltung und -propagation rein auf Semantikebene

Konstantenfaltung

Optimierung für Ausdrücke

- Wenn Berechnungen nur auf Konstanten, Ergebnis einfach einsetzen:
  Val(Intg 5) «Add» Val(Intg 3) wird zu *Val(Intg 8)*
  Val(Intg 4) «Eq» Val(Intg 7) wird zu *Val false*
- Wenn mind. eine Variable, einfach beibehalten:
  Var y «Sub» Val(Intg 3) bleibt *Var y «Sub» Val(Intg 3)*
- nicht sinvolle Ausdrücke auch beibehalten:
  Val(Intg 5) «And» Val true bleibt *Val(Intg 5) «And» Val true*
- Wenn Ausdruck nur Konstante oder Variable, auch beibehalten:
  Val(Intg 5) bleibt *Val(Intg 5)*, *Var y* bleibt *Var y*

Konstantenpropagation

Optimierung für Anweisungen

- Idee: Merken von Variablen, die konstant deklariert sind
- ermöglicht Ersetzen der Variable durch konstanten Wert
- dadurch möglich, `if`- Anweisungen zu vereinfachen
- Benötigt *Map* von Variablen nach Werten
- verwendet auch Konstantenfaltung

Beispiele

$x ::= \texttt{Val(Intg 7)};;$
$y ::= \texttt{Val(Intg 3)};;$
$\texttt{IF}\ (\texttt{Var}\ x\ «Eq»\ \texttt{Var}\ y)$
$\quad (y ::= \texttt{Var}\ x\ «Add»\ \texttt{Val(Intg 2)})$
$\texttt{ELSE}\ (y ::= \texttt{Var}\ x\ «Sub»\ \texttt{Var}\ z);;$
$z ::= \texttt{Var}\ y$

wird zu

$x ::= \texttt{Val(Intg 2)};;$
$y ::= \texttt{Var}\ x;;$
$b ::= \texttt{Var}\ x\ «Eq»\ \texttt{Var}\ y;;$
$\texttt{IF}\ (\texttt{Var}\ b)$
$\quad (z ::= \texttt{Var}\ x\ «Add»\ \texttt{Var}\ y)$
$\texttt{ELSE}\ (z ::= \texttt{Var}\ x)$

wird zu

Beispiele

$x ::= \texttt{Val}(\texttt{Intg}\ 7);;$
$y ::= \texttt{Val}(\texttt{Intg}\ 3);;$
$\texttt{IF}\ (\texttt{Var}\ x\ «Eq»\ \texttt{Var}\ y)$
$\quad (y ::= \texttt{Var}\ x\ «Add»\ \texttt{Val}(\texttt{Intg}\ 2))$
$\texttt{ELSE}\ (y ::= \texttt{Var}\ x\ «Sub»\ \texttt{Var}\ z);;$
$z ::= \texttt{Var}\ y$

wird zu

$x ::= \texttt{Val}(\texttt{Intg}\ 7);;$
$y ::= \texttt{Val}(\texttt{Intg}\ 3);;$
$y ::= \texttt{Val}(\texttt{Intg}\ 7)\ «Sub»\ \texttt{Var}\ z;;$
$z ::= \texttt{Var}\ y$

finale Map: $(x \mapsto \texttt{Val}(\texttt{Intg}\ 7))$

$x ::= \texttt{Val}(\texttt{Intg}\ 2);;$
$y ::= \texttt{Var}\ x;;$
$b ::= \texttt{Var}\ x\ «Eq»\ \texttt{Var}\ y;;$
$\texttt{IF}\ (\texttt{Var}\ b)$
$\quad (z ::= \texttt{Var}\ x\ «Add»\ \texttt{Var}\ y)$
$\texttt{ELSE}\ (z ::= \texttt{Var}\ x)$

wird zu

Beispiele

$x ::= \texttt{Val(Intg 7)};;$
$y ::= \texttt{Val(Intg 3)};;$
$\mathrm{IF}\ (\texttt{Var}\ x \ll Eq\gg \texttt{Var}\ y)$
$\quad (y ::= \texttt{Var}\ x \ll Add\gg \texttt{Val(Intg 2)})$
$\mathrm{ELSE}\ (y ::= \texttt{Var}\ x \ll Sub\gg \texttt{Var}\ z);;$
$z ::= \texttt{Var}\ y$

wird zu

$x ::= \texttt{Val(Intg 7)};;$
$y ::= \texttt{Val(Intg 3)};;$
$y ::= \texttt{Val(Intg 7)} \ll Sub\gg \texttt{Var}\ z;;$
$z ::= \texttt{Var}\ y$

finale Map: $(x \mapsto \texttt{Val(Intg 7)})$

$x ::= \texttt{Val(Intg 2)};;$
$y ::= \texttt{Var}\ x;;$
$b ::= \texttt{Var}\ x \ll Eq\gg \texttt{Var}\ y;;$
$\mathrm{IF}\ (\texttt{Var}\ b)$
$\quad (z ::= \texttt{Var}\ x \ll Add\gg \texttt{Var}\ y)$
$\mathrm{ELSE}\ (z ::= \texttt{Var}\ x)$

wird zu

$x ::= \texttt{Val(Intg 2)};;$
$y ::= \texttt{Val(Intg 2)};;$
$b ::= \texttt{Val true};;$
$z ::= \texttt{Val(Intg 4)}$

finale Map: $(x \mapsto \texttt{Val(Intg 2)}, y \mapsto \texttt{Val(Intg 2)}, b \mapsto \texttt{Val true}, z \mapsto \texttt{Val(Intg 4)})$

`WHILE`

Wie `IF` könnte man auch `WHILE` vereinfachen:

- falls Prädikat konstant *`false`*, komplettes `WHILE` durch `Skip` ersetzen
- falls Prädikat konstant *`true`*, Prädikat umschreiben, ansonsten Schleife beibehalten und in Schleifenkörper weiter Konstanten propagieren

Wie `IF` könnte man auch `WHILE` vereinfachen:

- falls Prädikat konstant *`false`*, komplettes `WHILE` durch `Skip` ersetzen
- falls Prädikat konstant *`true`*, Prädikat umschreiben, ansonsten Schleife beibehalten und in Schleifenkörper weiter Konstanten propagieren

Problem: Konstanten im Schleifenkörper beeinflussen auch Prädikat!
Beispiel:

```
x ::= Val(Intg 5);; y ::= Val(Intg 1);;
WHILE (Var x «Less» Val(Intg 7))
  (IF (Var y «Eq» Val(Intg 4))
     (x ::= Val(Intg 9))
   ELSE Skip;;
   y ::= Var y «Add» Val(Intg 1))
```

Darf das Prädikat von `WHILE` vereinfacht werden?

`WHILE`

- Kompletter Algorithmus bräuchte Fixpunktiteration!
- Zu kompliziert, deshalb Vereinfachung:
  Ist das Prädikat konstant `false` ist alles in Ordnung, ansonsten löschen wir beim `WHILE` die bisher gesammelte Konstanteninformation, verwenden also *`empty`* Map
- Ergebnis ist immer noch korrekt, aber nicht optimal vereinfacht
- Algorithmus so aber viel einfacher zu formalisieren

Projektaufgaben

1. Beweis, dass die vorgeg. Semantik deterministisch ist
(sowohl im Endzustand, als auch im Zeitbegriff)
2. Formalisierung von Konstantenpropagation inklusive -faltung
3. Beweis, dass Konstantenpropagation Semantik erhält
anders gesagt: “Endzustand ist der Gleiche, egal ob man im gleichen Anfangszustand Originalanweisung oder resultierende Anweisung der Konstantenpropagation verwendet”
4. Beweis, dass sich die Ausführungsgeschwindigkeit durch Konstantenpropagation erhöht
5. Beweis, dass zwei-/mehrfache Anwendung der Konstantenpropagation das Programm nicht weiter verändert
6. Beweis, dass Konstantenpropagation Typisierung erhält
anders gesagt: “Wenn Originalanweisung typt, dann auch resultierende Anweisung der Konstantenpropagation”

Beweise sollten verständlich und (komplett) in Isar geschrieben werden

Hinweise

- Isabelle-Dokumentation verwenden! Vor allem die Tutorials zu Isabelle/HOL, Isar und Function Definitions sollten helfen
- erst formalisieren, dann beweisen!
  Beispiele mittels `value` prüfen (z.B. Beispielprogramme in Semantics.thy)
- verwendet `quickcheck`, `nitpick` oder `sledgehammer` um Aussagen *vor* einem Beweis zu prüfen (spart oft unnötige Arbeit)
- falls Funktionsdefinitionen mit `fun` nicht funktionieren:
  - oftmals Probleme mit Termination
  - Fehlermeldung genau ansehen (wo Probleme mit Termination?)
    oft hilft eigene [simp] Regel
  - auch möglich: Zu `function` übergehen und versuchen, Termination explizit zu zeigen (siehe Tutorial zu Function Definitions)
- für die Beweise überlegen: welche Beziehungen müssen zwischen Semantikzustand, Typumgebung und Konstantenmap existieren?

Hinweise II

- *case*-Ausdrücke statt *if-then-else* verwenden wo möglich
  ⇒ Entsprechende *split*-Regeln verwenden
  ⇒ Mehr Automatismus

Beispiel

lemma "$case\ v\ of\ None \Rightarrow f\ 0 \mid Some\ x \Rightarrow f\ x \Longrightarrow \exists n.\ f\ n$"
  by (cases v) auto

Hinweise II

- `case`-Ausdrücke statt `if-then-else` verwenden wo möglich
  ⇒ Entsprechende `split`-Regeln verwenden
  ⇒ Mehr Automatismus

Beispiel

lemma *"case v of None ⇒ f 0 | Some x ⇒ f x ⟹ ∃n. f n"*
  by *(auto split: option.splits)*

Teil XXIV

Projekt: Eulerkreise

Hintergrund

Definition:
Ein Eulerkreis eines Graphen ist ein geschlossener Pfad, der jede Kante eines Graphen genau einmal enthält.

Satz:
Ein nicht-leerer gerichteter Multi-Graph hat genau dann einen Eulerkreis, wenn er stark zusammenhängend ist und für jeden Knoten Eingangsgrad und Ausgangsgrad übereinstimmen.

Hintergrund

Definition:
Ein Eulerkreis eines Graphen ist ein geschlossener Pfad, der jede Kante eines Graphen genau einmal enthält.

Satz:
Ein nicht-leerer gerichteter Multi-Graph hat genau dann einen Eulerkreis, wenn er stark zusammenhängend ist und für jeden Knoten Eingangsgrad und Ausgangsgrad übereinstimmen.

Das wollen wir formalisieren.

Multi-Mengen

Isabelle bietet (in der Theorie `"~~/src/HOL/Library/Multiset"`) einen Datentypen `'a multiset` für Multisets, also für Mengen mit Vielfachheiten. Ein paar Schreibweisen:

`{#}`	für die leere Multi-Menge
`{# x #}`	für Multi-Menge die nur `x` einmal enthält.
`size M`	für die Anzahl der Einträge in `M`.
`set_mset M`	um aus einer Multi-Menge eine Menge zu machen.
`x ∈# M`	wenn `x` mind. einmal in `M` vorkommt.
`M + M'`	für die Vereinigung von zwei Multimengen.
`M ⊆# M'`	wenn eine Menge eine Teil-Multi-Menge einer anderen ist.
`{# x ∈# M . P x #}`	für eine Filter-Operation auf Multi-Mengen.
`mset xs`	um aus Listen Multi-Mengen zu machen.

Unsere Modellierung von Graphen

Eine Kante ist ein geordnetes Paar, und ein Graph ist eine Multi-Menge solcher Paare:

type_synonym *'node graph = "('node × 'node) multiset"*

Unsere Modellierung von Graphen

Eine Kante ist ein geordnetes Paar, und ein Graph ist eine Multi-Menge solcher Paare:

type_synonym *'node graph = "('node × 'node) multiset"*

Ein paar Eigenschaften von Graphen:

definition *inDegree :: "'node graph ⇒ 'node ⇒ nat"*
where *"inDegree G n = size {# e ∈# G . snd e = n #}"*

definition *outDegree :: "'node graph ⇒ 'node ⇒ nat"*
where *"outDegree G n = size {# e ∈# G . fst e = n #}"*

definition *nodes :: "'node graph ⇒ 'node set"*
where *"nodes G = fst ` (set_mset G) ∪ snd ` (set_mset G)"*

Unsere Modellierung von Wegen

Ein Weg ist eine nicht-leere Liste von Kanten (*"('node × 'node) list"*), so dass der Zielknoten einer Kante der Anfangsknoten der nächsten Kante ist:

inductive *path :: "'node graph ⇒ ('node × 'node) list ⇒ bool"*
  where *"..."*

Unsere Modellierung von Wegen

Ein Weg ist eine nicht-leere Liste von Kanten (*"('node × 'node) list"*), so dass der Zielknoten einer Kante der Anfangsknoten der nächsten Kante ist:

inductive *path :: "'node graph ⇒ ('node × 'node) list ⇒ bool"*
  where *"..."*

Ein Graph ist stark zusammenhängend, wenn es zu je zwei Knoten v_1, v_2 im Graphen einen Pfad gibt, dessen erste Kante bei v_1 beginnt und dessen letzte Kante bei v_2 endet.

definition *strongly_connected :: "'node graph ⇒ bool"*
  where *"strongly_connected G = ..."*

Unsere Modellierung von Wegen

Ein Weg ist eine nicht-leere Liste von Kanten (*"('node × 'node) list"*), so dass der Zielknoten einer Kante der Anfangsknoten der nächsten Kante ist:

inductive *path :: "'node graph ⇒ ('node × 'node) list ⇒ bool"*
  where *"..."*

Ein Graph ist stark zusammenhängend, wenn es zu je zwei Knoten v_1, v_2 im Graphen einen Pfad gibt, dessen erste Kante bei v_1 beginnt und dessen letzte Kante bei v_2 endet.

definition *strongly_connected :: "'node graph ⇒ bool"*
  where *"strongly_connected G = ..."*

Ein Eulerkreis eines Graphen ist ein geschlossener Pfad, der jede Kante eines Graphen genau einmal enthält:

definition *eulerian :: "'node graph ⇒ bool"* **where** *"eulerian G =*
 (∃ es. path G es ∧ fst (hd es) = snd (last es) ∧ G = mset es)"

Die Aufgabe

Beweisen Sie:

theorem *euleriant_characterization:*

$$\text{"}G \neq \{\#\} \Longrightarrow \textit{eulerian } G \longleftrightarrow \textit{strongly\_connected } G \wedge (\forall v.\ \textit{inDegree } G\ v = \textit{outDegree } G\ v)\text{"}$$

Die Aufgabe

Beweisen Sie:

theorem *eulerian_characterization:*
  $"G \neq \{\#\} \Longrightarrow \mathit{eulerian}\ G \longleftrightarrow$
    $\mathit{strongly\_connected}\ G \wedge (\forall v.\ \mathit{inDegree}\ G\ v = \mathit{outDegree}\ G\ v)"$

Ein paar Hinweise:

- Die Richtung $\longrightarrow$ ist (scheinbar) leichter.
- Ist *es* ein Pfad, so ist *mset es* ein Graph; so kann man *inDegree* etc. auch damit verwenden.
- Aussagen über Ein- und Ausgangsgrade in Pfaden sind schöner, wenn der Pfad (ggf. künstlich) geschlossen wird!
- Bei komischen Induktionen (z.B. um Schrittweise von *23* auf *42* zu schließen) ist es schön, die Induktionsregel, sofern nicht in der Standardbibliothek vorhanden, als eigenes Lemma zu extrahieren und dann zu benutzen.

Teil XXV

Attribute

Attribute

Allgemein: Attribute verändern Theoreme.

Attribute

Allgemein: Attribute verändern Theoreme.

Syntax:

theoremname[attribut1, attribut2, attribut mit optionen]

Attribute

Allgemein: Attribute verändern Theoreme.

Syntax:

theoremname[attribut1, attribut2, attribut mit optionen]

Kann überall verwendet werden, wo ein Theorem referenziert wird:

. . . **by** *(rule foo[bar])*

from *foo[bar]* **have**. . .

declare *neuer_name = foo[bar]*

note *neuer_name = foo[bar]*

Variablen in Regeln spezifizieren mittels `of`

Manchmal nötig, um Variablen vor Regelanwendung festzulegen
(z.B. wenn Isabelle passende Terme nicht inferieren kann), dann:

- Attribut `of`, danach einer oder mehrere Terme
- müssen natürlich zu Typ der Variable passen
- Reihenfolge wie erstes Auftreten in Regel
- _ für Variablen, die man nicht instantiieren möchte

Variablen in Regeln spezifizieren mittels `of`

Manchmal nötig, um Variablen vor Regelanwendung festzulegen (z.B. wenn Isabelle passende Terme nicht inferieren kann), dann:

- Attribut `of`, danach einer oder mehrere Terme
- müssen natürlich zu Typ der Variable passen
- Reihenfolge wie erstes Auftreten in Regel
- _ für Variablen, die man nicht instantiieren möchte

Beispiel:

`iffE:`	$[\![?P = ?Q;\ [\![?P \longrightarrow ?Q;\ ?Q \longrightarrow ?P]\!] \Longrightarrow ?R]\!] \Longrightarrow ?R$
`iffE[of X]:`	$[\![X = ?Q;\ [\![X \longrightarrow ?Q;\ ?Q \longrightarrow X]\!] \Longrightarrow ?R]\!] \Longrightarrow ?R$
`iffE[of _ Y]:`	$[\![?P = Y;\ [\![?P \longrightarrow Y;\ Y \longrightarrow ?P]\!] \Longrightarrow ?R]\!] \Longrightarrow ?R$
`iffE[of X Y Z]:`	$[\![X = Y;\ [\![X \longrightarrow Y;\ Y \longrightarrow X]\!] \Longrightarrow Z]\!] \Longrightarrow Z$

Variablen in Regeln spezifizieren mittels `where`

Syntax:

`Regel[where v=T]`

Wobei

- v die zu spezifizierende Variable in der Regel `Regel` ist
- T der einzusetzende Term ist

Beispiel:

`iffE:` $\llbracket ?P = ?Q;\ \llbracket ?P \longrightarrow ?Q;\ ?Q \longrightarrow ?P \rrbracket \Longrightarrow ?R \rrbracket \Longrightarrow ?R$

`iffE[where Q="X ∧ Y"]:` $\llbracket ?P = X \wedge Y;\ \llbracket ?P \longrightarrow X \wedge Y;\ X \wedge Y \longrightarrow ?P \rrbracket \Longrightarrow ?R \rrbracket \Longrightarrow ?R$

Prämissen in Regeln spezifizieren mittels `OF`

Analog zu `of`: Ganze Prämissen instantiieren

- Attribut `OF` gefolgt von Regelnamen.
- Konklusion der Regel und entspr. Prämisse müssen unifizieren.
- Entspr. Prämissen werden durch Prämissen der eingefügten Regel ersetzt.
- Mit _ werden Prämissen Überspringen.
- Gut bei Induktionshypothesen in Isar einsetzbar (`Foo.IH[OF bar]`).

Prämissen in Regeln spezifizieren mittels `OF`

Analog zu `of`: Ganze Prämissen instantiieren

- Attribut `OF` gefolgt von Regelnamen.
- Konklusion der Regel und entspr. Prämisse müssen unifizieren.
- Entspr. Prämissen werden durch Prämissen der eingefügten Regel ersetzt.
- Mit _ werden Prämissen Überspringen.
- Gut bei Induktionshypothesen in Isar einsetzbar (`Foo.IH[OF bar]`).

Beispiel:

conjI:	$[\![?P;\ ?Q]\!] \Longrightarrow ?P \wedge ?Q$
ccontr:	$(\neg\ ?P \Longrightarrow False) \Longrightarrow ?P$
conjI[OF ccontr]:	$[\![\neg\ ?P \Longrightarrow False;\ ?Q]\!] \Longrightarrow ?P \wedge ?Q$
conjI[OF ccontr,of X]:	$[\![\neg\ X \Longrightarrow False;\ ?Q]\!] \Longrightarrow X \wedge ?Q$

Konklusion umdrehen mit `symmetric`

Wenn die Konklusion einer Regel eine Gleichheit falsch herum hat, hilft `foo[symmetric]`:

Konklusion umdrehen mit `symmetric`

Wenn die Konklusion einer Regel eine Gleichheit falsch herum hat, hilft *foo[symmetric]*:

Beispiel:

drop_all:	length ?xs $\leq$?n $\Longrightarrow$ drop ?n ?xs = []
drop_all[symmetric]:	length ?xs $\leq$?n $\Longrightarrow$ [] = drop ?n ?xs

Definitionen falten mit `folded` und `unfolded`

Man kann eine Gleichung (meist eine Definition) in einer Regel substituieren, je nach Richtung mit `foo[folded equality]` oder `foo[unfolded equality]`:

Definitionen falten mit *folded* und *unfolded*

Man kann eine Gleichung (meist eine Definition) in einer Regel substituieren, je nach Richtung mit *foo[folded equality]* oder *foo[unfolded equality]*:

Beispiel:

solution_def	*solution = 42*
foo:	*?P solution* $\Longrightarrow$ *?Q 42*
foo[unfolded solution_def]:	*?P 42* $\Longrightarrow$ *?Q 42*
foo[folded solution_def]:	*?P solution* $\Longrightarrow$ *?Q solution*

Regeln vereinfachen mit `simplified`

Das Attribut `[simplified]` lässt den Simplifier eine Regel vereinfachen. Das sollte man bei bewiesenen Lemmas eigentlich nicht brauchen (die kann man direkt „richtig“ formulieren), aber in Kombination mit `OF` oder `of` ist es oft der beste Weg die Regel wieder in eine Form zu kriegen, mit der z.B. `auto intro:` arbeiten kann.

Regeln vereinfachen mit *simplified*

Das Attribut *[simplified]* lässt den Simplifier eine Regel vereinfachen. Das sollte man bei bewiesenen Lemmas eigentlich nicht brauchen (die kann man direkt „richtig“ formulieren), aber in Kombination mit *OF* oder *of* ist es oft der beste Weg die Regel wieder in eine Form zu kriegen, mit der z.B. *auto intro:* arbeiten kann.

(Sehr konstruiertes) Beispiel:

```
take_add:
  take (?i + ?j) ?xs = take ?i ?xs @ take ?j (drop ?i ?xs)
take_add[of 5 10]:
  take (5 + 10) ?xs = take 5 ?xs @ take 10 (drop 5 ?xs)
take_add[of 5 10, simplified]:
  take 15 ?xs = take 5 ?xs @ take 10 (drop 5 ?xs)
```

Regeln vereinfachen mit `simplified`

Das Attribut *[simplified]* lässt den Simplifier eine Regel vereinfachen. Das sollte man bei bewiesenen Lemmas eigentlich nicht brauchen (die kann man direkt „richtig“ formulieren), aber in Kombination mit *OF* oder *of* ist es oft der beste Weg die Regel wieder in eine Form zu kriegen, mit der z.B. *auto intro:* arbeiten kann.

(Sehr konstruiertes) Beispiel:

```
take_add:
  take (?i + ?j) ?xs = take ?i ?xs @ take ?j (drop ?i ?xs)
take_add[of 5 10]:
  take (5 + 10) ?xs = take 5 ?xs @ take 10 (drop 5 ?xs)
take_add[of 5 10, simplified]:
  take 15 ?xs = take 5 ?xs @ take 10 (drop 5 ?xs)
```

Das Attribut kann auch in der Form *[simplified regel1 regel2...]* verwendet werden. Dann verwendet der Simplifier nur die angegebenen Regeln.

Karlsruher Institut für Technologie

Teil XXVI

Universelle Fallunterscheidung

Fallunterscheidungsregeln

Wir kennen bereits Fallunterscheidung

- klassisch (mit `case_split`),
- nach Datentypkonstruktor (Bsp. `list.exhaust`),
- als Regelinversion bei induktiven Prädikaten (Bsp. `palin.cases`),
- nach Pattern-Matching bei **fun**-Definitione (Bsp. `BigNat.add'.cases`).

Alle diese Regeln folgen dem Muster:

$$
\begin{aligned}
&(\mathit{Fall1} \Longrightarrow P) \Longrightarrow \\
&(\mathit{Fall2} \Longrightarrow P) \Longrightarrow \\
&(\mathit{Fall3} \Longrightarrow P) \Longrightarrow \\
&\ldots \Longrightarrow P
\end{aligned}
$$

Im Allgemeinen kann jede Regel dieser Form als Fallunterscheidungsregel verwendet werden. Z.B.:

$$(\mathit{even}\ n \Longrightarrow P) \Longrightarrow (\mathit{odd}\ n \Longrightarrow P) \Longrightarrow P$$

Eigene Fallunterscheidungsregeln anwenden

```
lemma even_odd_cases:
  assumes "even n ⟹ P"
    and "odd n ⟹ P"
  shows "P"
```

Freie Variablen der Regel müssen instanziiert werden:

```
have "P (n::nat)"
proof (cases n rule: even_odd_cases)
  case 1
  ...
qed
```

Eigene Fallunterscheidungsregeln anwenden

```
lemma even_odd_cases [case_names even odd]:
  assumes "even n ⟹ P"
    and "odd n ⟹ P"
  shows "P"
```

Freie Variablen der Regel müssen instanziiert werden:

```
have "P (n::nat)"
proof (cases n rule: even_odd_cases)
  case even
  ...
qed
```

lokale Fallunterscheidung

Fallunterscheidung ist auch lokal in einem Isar-Beweis mit dem Kommando **consider** möglich:

consider *(even) "even n"* | *(odd) "odd n"* **by** *blast*
then show *?thesis*
proof *cases*
  case *even*
  ...
next
  case *odd*
  ...
qed

lokale Fallunterscheidung

Fallunterscheidung ist auch lokal in einem Isar-Beweis mit dem Kommando **consider** möglich:

```
consider (even) "even n" | (odd) "odd n" by blast
then show ?thesis
proof cases
  case even
  ...
next
  case odd
  ...
qed
```

Dabei auch „obtainen“ von Variablen möglich:

```
consider (zero) "n = 0" | (succ) x where "n = Suc x"
then have "even n"
proof cases
  case (succ x) ...
```

Teil XXVII

Strukturierte Zwischenziele

Strukturierte Beweisziele

Ähnlich wie **assumes** und **shows** kann man auch für Zwischenziele (**have**) eines Isar-Beweises die Aussage strukturieren:

have *"P x"*
  if *"Q x"*
  and *"R x"*

- Annahmen nach **if** *nicht* im Beweiszustand.
  Dafür gibt es die Variable *that*, welche alle **if**-Annahmen enthält.

Strukturierte Beweisziele

Ähnlich wie **assumes** und **shows** kann man auch für Zwischenziele (**have**) eines Isar-Beweises die Aussage strukturieren:

have *"P x"*
  if *meine_annahme_1: "Q x"*
  and *meine_annahme_2: "R x"*

- Annahmen nach **if** *nicht* im Beweiszustand.
  Dafür gibt es die Variable *that*, welche alle **if**-Annahmen enthält.
- Benamte Annahmen

Strukturierte Beweisziele

Ähnlich wie **assumes** und **shows** kann man auch für Zwischenziele (**have**) eines Isar-Beweises die Aussage strukturieren:

have *"P x"*
  if *meine_annahme_1: "Q x"*
  and *meine_annahme_2: "R x"*
  for *x :: nat*

- Annahmen nach **if** *nicht* im Beweiszustand.
  Dafür gibt es die Variable *that*, welche alle **if**-Annahmen enthält.
- Benamte Annahmen
- (Meta-)Allquantifizierte Variablen

Strukturierte Beweisziele

Ähnlich wie **assumes** und **shows** kann man auch für Zwischenziele (**have**) eines Isar-Beweises die Aussage strukturieren:

have *"P x"*
  if *meine_annahme_1: "Q x"*
  and *meine_annahme_2: "R x"*
  for *x :: nat*

- Annahmen nach **if** *nicht* im Beweiszustand.
  Dafür gibt es die Variable *that*, welche alle **if**-Annahmen enthält.
- Benamte Annahmen
- (Meta-)Allquantifizierte Variablen

if und **for** auch mit **assume** möglich – nicht aber mit **assumes**. Also nur innerhalb eines Beweises.

Teil XXVIII

Typedef

Eigene Typen in HOL definieren

In HOL, und damit in Isabelle, können eigene Datentypen definiert werden. Dazu benötigt man

- eine Teilmenge eines existierenden Typs sowie
- ein Beweis, dass diese Teilmenge nicht leer ist.

(Leere Typen würden HOL inkonsistent machen, d.h. man könnte *False* beweisen.)

Eigene Typen in Isabelle definieren: typedef

Karlsruher Institut für Technologie

Syntax

typedef *typname* = *"Menge"* **morphisms** *rep_fun abs_fun* **by** *proof*

- *typname* ist der Name des neuen Typs. Hier dürfen auch Typvariablen verwendet werden (*('a, 'b) typname*).
- *Menge* ist ein Ausdruck vom Typ *irgendwas set*.
- Morphismen konvertieren zwischen der Menge und dem neuen Typ: $rep\_fun :: typname \Rightarrow irgendwas$ und $abs\_fun :: irgendwas \Rightarrow typname$
- Default-Morphismennamen: *Rep_typname* und *Abs_typname*.
- Das Beweisziel ist $\exists x.\ x \in Menge$.
- Erzeugt (u. a. und v. a.) diese Lemmas:

rep_fun:	$rep\_fun\ ?x \in Menge$
rep_fun_inverse:	$abs\_fun\ (rep\_fun\ ?x) = ?x$
rep_abs_inverse:	$?y \in Menge \Longrightarrow rep\_fun\ (abs\_fun\ ?y) = ?y$

Beispiel: Nicht-Leere Liste

Wir erstellen einen Typ für nicht-leere Listen und beginnen mit der Typ-Definition:

Beispiel: Nicht-Leere Liste

Wir erstellen einen Typ für nicht-leere Listen und beginnen mit der Typ-Definition:

typedef *'a ne = "{xs :: 'a list . xs ≠ []}"*
  by *(rule exI[***where*** x = "[undefined]"], simp)*

Beispiel: Nicht-Leere Liste

Wir erstellen einen Typ für nicht-leere Listen und beginnen mit der Typ-Definition:

```
typedef 'a ne = "{xs :: 'a list . xs ≠ []}"
  by (rule exI[where x = "[undefined]"], simp)
```

Weiter ein paar Funktionen auf nicht-leeren Listen:

```
definition singleton :: "'a ⇒ 'a ne"
  where "singleton x = Abs_ne [x]"

definition append :: "'a ne ⇒ 'a ne ⇒ 'a ne"
  where "append l1 l2 = Abs_ne (Rep_ne l1 @ Rep_ne l2)"

definition head :: "'a ne ⇒ 'a"
  where "head l = hd (Rep_ne l)"

definition tail :: "'a ne ⇒ 'a ne"
  where "tail l = Abs_ne (tl (Rep_ne l))"
```

Beispiel: Lemmas zu Nicht-Leeren Liste

Bei Append kommt der Head der Liste immer von der linken Liste (für allgemeine Listen nicht wahr!):

```
lemma "head (append l1 l2) = head l1"
  unfolding head_def append_def
  apply (subst Abs_ne_inverse)
  using Rep_ne[of l1] apply simp
  using Rep_ne[of l1] apply simp
  done
```

Beispiel: Mehr Lemmas zu Nicht-Leeren Liste

Head und Tail ergeben wieder die gesamte Liste:

```
lemma "append (singleton (head l)) (tail l) = l"
  unfolding head_def append_def singleton_def tail_def
  apply (subst Abs_ne_inverse)
  apply simp
  apply (subst Abs_ne_inverse)
  defer
  using Rep_ne[of l]
  apply simp
  apply (rule Rep_ne_inverse)
  apply simp
  oops
```

Beispiel: Mehr Lemmas zu Nicht-Leeren Liste

Head und Tail ergeben wieder die gesamte Liste:

```
lemma "append (singleton (head l)) (tail l) = l"
  unfolding head_def append_def singleton_def tail_def
  apply (subst Abs_ne_inverse)
  apply simp
  apply (subst Abs_ne_inverse)
  defer
  using Rep_ne[of l]
  apply simp
  apply (rule Rep_ne_inverse)
  apply simp
  oops
```

Problem: Das Lemma ist „eigentlich“ richtig, aber `tail [a]` ist undefiniert, da keine nicht-leere Liste.

Beispiel: Richtige Lemmas zu Nicht-Leeren Liste

Tail muss eine „normale“ Liste zurückgeben:

```
definition tail' :: "'a ne ⇒ 'a list"
  where "tail' l = tl (Rep_ne l)"

definition append' :: "'a ne ⇒ 'a list ⇒ 'a ne"
  where "append' l1 l2 = Abs_ne (Rep_ne l1 @ l2)"

lemma "append' (singleton (head l)) (tail' l) = l"
  unfolding head_def append'_def singleton_def tail'_def
  apply (subst Abs_ne_inverse, simp)
  using Rep_ne[of l, simplified]
  apply simp
  apply (rule Rep_ne_inverse)
  done
```

Ausblick: lifting und transfer

Karlsruher Institut für Technologie

Das Beweisen mit den Abstraktions- und Representationsfunktionen ist mühsam und unnatürlich: So wird die Erhaltung einer Invariante beim Verwenden der Funktion bewiesen, und nicht beim Definieren (siehe `tail`).

Die Isabelle-Pakete Lifting und Transfer erlauben es, Funktionen einmal bei der Definition als „korrekt“ zu beweisen und Lemmas mit einem Methodenaufruf in die Welt der zugrundeliegenden Repräsentation zu übertragen und dann dort zu beweisen.

Teil XXIX

Locales

Beweise modularisieren

$\Longrightarrow$ Verwende **Locales**

locale: Definiert neuen Beweiskontext

fixes: Legt Funktionssymbol fest (wird zu Parameter der Locale)

assumes: Macht Annahmen über die Locale-Parameter

context ... begin: Öffnet Beweiskontext

end: Schließt Beweiskontext

Beispiel:

```
locale Magma =
  fixes M :: "'a set"
  fixes bop :: "'a ⇒'a ⇒'a"
  assumes closed: "a ∈ M ⟹ b ∈ M ⟹ bop a b ∈ M"

context Magma begin <Definitionen, Beweise, ...> end
```

Locales Erweitern

Locales lassen sich mit “+” erweitern:

Beispiel:

locale *Semigroup = Magma +*
 assumes *assoc: "..."*

Auch “verschmelzen” von Locales möglich:

Beispiel:

locale *Ring = AbelieanGroup "M" "add" "zero" + Magma "M" "mul"*
 for *M :: "'a set"*
 and *add :: "'a ⇒ 'a ⇒ 'a"*
 and *zero :: "'a"*
 and *mul :: "'a ⇒ 'a ⇒ 'a"*
+ **assumes** *assoc: "..."*

Interpretationen

Instanzieerung der Locales mit

interpretation: im Theoriekontext

interpret: in Beweiskontexten

Vorgehen:

- Angabe der konkreten Parameter
- Locale-Definition “auspacken” mit Taktik **unfold_locales**
- Beweis der Locale-Annahmen

Beispiel:

interpretation *Mod3:*
*Ring "{0::nat,1,2}" "λa b. a + b mod 3" "0" "λa b. a * b mod 3"*
by *(unfold_locales) auto*

Evaluation

Teil XXX

Lifting und Transfer

Rückblick: Eigene Typen in HOL definieren

```
typedef 'a ne = "{xs :: 'a list . xs ≠ []}"
  by (rule exI[where x = "[undefined]"], simp)

definition singleton :: "'a ⇒ 'a ne"
  where "singleton x = Abs_ne [x]"
definition append :: "'a ne ⇒ 'a ne ⇒ 'a ne"
  where "append l1 l2 = Abs_ne (Rep_ne l1 @ Rep_ne l2)"
definition head :: "'a ne ⇒ 'a"
  where "head l = hd (Rep_ne l)"

lemma "head (append l1 l2) = head l1"
  unfolding head_def append_def
  apply (subst Abs_ne_inverse)
  using Rep_ne[of l1] apply simp
  using Rep_ne[of l1] apply simp
  done
```

Heute: lifting und transfer

Das Beweisen mit den Abstraktions- und Representationsfunktionen ist mühsam und unnatürlich: So wird die Erhaltung einer Invariante beim Verwenden der Funktion bewiesen, und nicht beim Definieren (siehe *`tail`*).

Die Isabelle-Pakete Lifting und Transfer erlauben es, Funktionen einmal bei der Definition als „korrekt“ zu beweisen und Lemmas mit einem Methodenaufruf in die Welt der zugrundeliegenden Repräsentation zu übertragen und dann dort zu beweisen.

Lifting und Transfer verwenden

1. Typ registrieren:

   setup_lifting *type_definition_typname*

Lifting und Transfer verwenden

1. Typ registrieren:

   setup_lifting *type_definition_typname*

2. Definitionen liften:

   lift_definition *name :: type* **is** *"ausdruck"*

   Beweis

   wobei *ausdruck* die Definition von *name* auf den konkreten Datentyp ist und der *Beweis* beweist dass die Typ-Invarianten respektiert werden.

Lifting und Transfer verwenden

1. Typ registrieren:

   setup_lifting *type_definition_typname*

2. Definitionen liften:

   lift_definition *name :: type* **is** *"ausdruck"*

   Beweis

   wobei *ausdruck* die Definition von *name* auf den konkreten Datentyp ist und der *Beweis* beweist dass die Typ-Invarianten respektiert werden.

3. Aussagen auf die konkreten Typen übertragen: **apply** *transfer*
   Ersetzt das aktuelle Ziel durch ein gleichwertiges auf dem konkreten Datentyp, indem die per **lift_definition** definierten Funktionen durch ihre konkrete Definition ersetzt werden.

Beispiel: Sortierte Listen

Typ registrieren:

```
typedef slist = "{xs. sorted xs}" morphisms list_of as_sorted
  by (rule_tac x="[]" in exI) simp

setup_lifting type_definition_slist
```

Beispiel: Sortierte Listen

Typ registrieren:

typedef *slist = "{xs. sorted xs}"* **morphisms** *list_of as_sorted*
by *(rule_tac x="[]"* **in** *exI) simp*

setup_lifting *type_definition_slist*

Definitionen:

lift_definition *Singleton :: "nat ⇒ slist"* **is** *"λx. [x]"* **by** *simp*
lift_definition *hd :: "slist ⇒ nat"* **is** *"List.hd"* **..**
lift_definition *take :: "nat ⇒ slist ⇒ slist"* **is** *"List.take"* **..**
lift_definition *smerge :: "slist ⇒ slist ⇒ slist"* **is** *"Scratch.merge"* **by** *(rule sorted_merge_sorted)*

Beispiel: Sortierte Listen

Lemmas zu Definitionen auf dem abstrakten Typ:

lemma *set_of_Singleton [simp]: "set_of (Singleton x) = {x}"*

Aktuelles Ziel: *set_of (Singleton x) = {x}*

apply *transfer*

Aktuelles Ziel: $\bigwedge x.$ *set [x] = {x}*

apply *simp*

Aktuelles Ziel: *No subgoals!*

done

oder gleich

by *transfer simp*

Beispiel: Sortierte Listen

Lemmas können Invarianten nutzen:

lemma *"list_of xs = a#b#ys* $\Longrightarrow$ *a* $\leq$ *b"*

Aktuelles Ziel: *list_of xs = a # b # ys* $\Longrightarrow$ *a* $\leq$ *b*

apply *transfer*

Aktuelles Ziel: $\bigwedge$*xs a b ys.* $[\![$*sorted xs; xs = a # b # ys*$]\!]$ $\Longrightarrow$ *a* $\leq$ *b*

apply *simp*

Aktuelles Ziel: *No subgoals!*

done

Beispiel: Sortierte Listen

Lemmas mit rein abstrakten Definitionen:

definition *insert :: "nat ⇒ slist ⇒ slist"*
  where *"insert x xs = smerge xs (Singleton x)"*

lemma *set_of_insert [simp]: "x ∈ set_of (insert x xs)"*

Beispiel: Sortierte Listen

Lemmas mit rein abstrakten Definitionen:

definition *insert :: "nat ⇒ slist ⇒ slist"*
  where *"insert x xs = smerge xs (Singleton x)"*

lemma *set_of_insert [simp]: "x ∈ set_of (insert x xs)"*

Erster Versuch:

  apply *transfer*

Hier bringt *transfer* einen nicht weiter!

Beispiel: Sortierte Listen

Lemmas mit rein abstrakten Definitionen:

definition *insert :: "nat ⇒ slist ⇒ slist"*
  where *"insert x xs = smerge xs (Singleton x)"*

lemma *set_of_insert [simp]: "x ∈ set_of (insert x xs)"*

Erster Versuch:

  apply *transfer*

Hier bringt *transfer* einen nicht weiter!

Zweiter Versuch:

  unfolding *insert_def* **by** *transfer simp*

Beispiel: Sortierte Listen

Lemmas mit rein abstrakten Definitionen:

definition *insert :: "nat ⇒ slist ⇒ slist"*
  where *"insert x xs = smerge xs (Singleton x)"*

lemma *set_of_insert [simp]: "x ∈ set_of (insert x xs)"*

Erster Versuch:

  apply *transfer*

Hier bringt *transfer* einen nicht weiter!

Zweiter Versuch:

  unfolding *insert_def* **by** *transfer simp*

Schöner ist:

lemma *set_of_smerge: "set_of (smerge xs ys) = set_of xs ∪ set_of ys"*
  by *transfer simp*

und dann

  unfolding *insert_def* **by** *(simp add: set_of_smerge)*

Karlsruher Institut für Technologie

Teil XXXI

Erzeugung von ausführbarem Code

Code exportieren

Isabelle kann Formalisierungen nach **SML**, **OCaml**, **Haskell** bzw. **Scala** exportieren.

$\Longrightarrow$ Dadurch sind verifizierte *ausführbare* Programme möglich.

- Jede HOL-Funktion wird in eine entsprechende Funktion der Zielsprache übersetzt.
- Jeder HOL-Typ wird ein entsprechenden Typ der Zielsprache übersetzt.
- **Basis:** Code-Gleichungen

Code-Erzeugung mit Befehl:

export_code *f* **in** *Sprache* **module_name** *Modul* **file** *Datei*

Die definierenden HOL-Gleichungen von *f* werden 1:1 in die Zielsprache übersetzt.

Nicht ausführbare Definitionen

Nicht alle HOL-Funktionen können direkt übersetzt werden.

Beispiel

Wie kann die Funktion

```
doubled xs =
  (if (∃ys. xs = ys @ ys) then Some (THE ys. xs = ys @ ys) else None)
```

übersetzt werden?

Nicht ausführbare Definitionen

Nicht alle HOL-Funktionen können direkt übersetzt werden.

Beispiel

Wie kann die Funktion

```
doubled xs =
  (if (∃ys. xs = ys @ ys) then Some (THE ys. xs = ys @ ys) else None)
```

übersetzt werden?

Problem: Existenzquantor nur für enum-Typen ausführbar.

Nicht ausführbare Definitionen

Nicht alle HOL-Funktionen können direkt übersetzt werden.

Beispiel

Wie kann die Funktion

doubled xs =
  (if (∃ys. xs = ys @ ys) then Some (THE ys. xs = ys @ ys) else None)

übersetzt werden?

Problem: Existenzquantor nur für enum-Typen ausführbar.

Lösung: Beweise alternative Code-Gleichung:

```
lemma doubled_code [code]: "doubled xs =
  (let ys = take (length xs div 2) xs in
    (if (xs = ys @ ys) then Some ys else None)"
```

Code-Gleichungen

Eine Gleichung kann als Code-Gleichung für *f* verwendet werden, wenn

- *f* das oberste (und einzige) Funktionssymbol im linken Term ist,
- Patternmatching auf die Parameter von *f* nur via Datentyp-Konstruktoren erfolgt, und
- für alle Funktionssymbole auf der rechten Seite Code-Gleichungen existieren.

Insbesondere ist es nicht (direkt) möglich „partielle“ Code-Gleichungen anzugeben.

Späteres hinzufügen einer Gleichung als Code-Gleichung mit

declare *lemma [code]*

möglich.

Beispiel

Siehe Formalisierung

Diverses

- Das Kommando

  value *[code]* "*t*"

  übersetzt den Term *t* und wertet ihn aus.

- **eval** ist eine Beweis-Taktik, welche versucht, das aktuelle Ziel durch „ausrechnen“ (Brute-Force) zu zeigen.

- **code_thms** *f* zeigt alle registrierten Code-Gleichungen an, die zur Auswertung von *f* benötigt werden.

- **print_codesetup** zeigt *alle* registrierten Code-Gleichungen an.

Datatype Refinement mit Invarianten

Lifting arbeitet gut mit dem Code-Generator zusammen: Es registriert *as_sorted* als Konstruktor für den Typ *slist* und definierte alle Operationen darauf. Man kann keine Code-Gleichung angeben die mittels *as_sorted x* ein Wert vom Typ *slist* konstruiert, ohne bewiesen zu haben, dass *sorted x* gilt.

```
export_code insert hd take list_of set_of
  in Haskell
  file "-"
```

Manuell: Auch möglich, dann mit **code_datatype**, *[code abstype]* und *[code abstract]* arbeiten.

$\Longrightarrow$ siehe nachher, sowie `isabelle doc codegen`.

Code-Präprozessor

Vor Anwendung der Code-Gleichungen werden diese vom Code-Präprozessor bearbeitet.

- Rewrite-System mit ähnlicher Mächtigkeit wie Simplifier
- Attribut **code_unfold** verwenden, um Gleichungen zu registrieren
- **print_codeproc** zeigt das Präprozessor-Setup an

Teil XXXII

Koinduktion

Was ist Koinduktion?

Duales Prinzip zu Induktion

Induktive Definition:

kleinster Fixpunkt, der die definierende Gleichung erfüllt.

Koinduktive Definition:

größter Fixpunkt, der die definierende Gleichung erfüllt.

Was ist Koinduktion?

Duales Prinzip zu Induktion

Induktive Definition:

> *kleinster* Fixpunkt, der die definierende Gleichung erfüllt.
> Induktionsprinzip: Um eine Eigenschaft für alle Elemente zu zeigen, genügt es sie für eine beliebige Menge zu zeigen, die die definierende Gleichung erfüllt. (Der kleinste Fixpunkt muss darin enthalten sein)

Koinduktive Definition:

> *größter* Fixpunkt, der die definierende Gleichung erfüllt.
> Koinduktionsprinzip: Jede Menge, die die definierende Gleichung erfüllt, ist in der koinduktiven Definition enthalten.

Koinduktive Prädikate in Isabelle

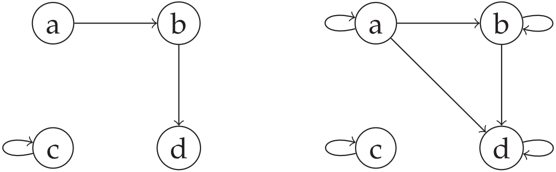
Beispiel: Reflexiv transitive Hülle

inductive *rtc :: "('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool"*
for *r :: "('a ⇒ 'a ⇒ bool)"*
where *refl: "rtc r x x"*
  | *trans: "r x y ⟹ rtc r y z ⟹ rtc r x z"*

Koinduktive Prädikate in Isabelle

Beispiel: Reflexiv transitive Hülle

```
inductive rtc :: "('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool"
for r :: "('a ⇒ 'a ⇒ bool)"
where refl: "rtc r x x"
  | trans: "r x y ⟹ rtc r y z ⟹ rtc r x z"
```



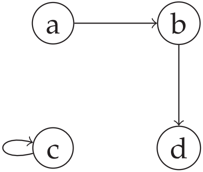
(a) Graph r **(b)** Inductive RTC r

aus © Andreas Lochbihler, DOI 10.5445/KSP/1000028867, KIT Scientific Publishing, Karlsruhe, 2012

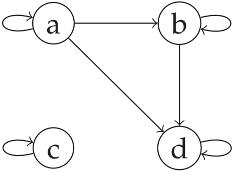
Koinduktive Prädikate in Isabelle

Beispiel: Reflexiv transitive Hülle

coinductive *rtc :: "('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool"*
for *r :: "('a ⇒ 'a ⇒ bool)"*
where *refl: "rtc r x x"*
  | *trans: "r x y ⟹ rtc r y z ⟹ rtc r x z"*



(a) Graph r



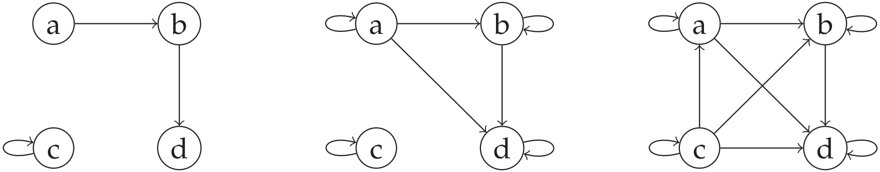
(b) Inductive RTC r

aus © Andreas Lochbihler, DOI 10.5445/KSP/1000028867, KIT Scientific Publishing, Karlsruhe, 2012

Koinduktive Prädikate in Isabelle

Beispiel: Reflexiv transitive Hülle

coinductive *rtc :: "('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool"*
for *r :: "('a ⇒ 'a ⇒ bool)"*
where *refl: "rtc r x x"*
| *trans: "r x y ⟹ rtc r y z ⟹ rtc r x z"*



(a) Graph r **(b)** Inductive RTC r **(c)** Coinductive RTC r

aus © Andreas Lochbihler, DOI 10.5445/KSP/1000028867, KIT Scientific Publishing, Karlsruhe, 2012

Koinduktive Beweise

Induktionsschema für `rtc`: `rtc.induct`

$$\mathit{rtc}\ r\ a\ b \Longrightarrow (\bigwedge x.\ P\ x\ x)$$
$$\Longrightarrow (\bigwedge x\ y\ z.\ r\ x\ y \Longrightarrow \mathit{rtc}\ r\ y\ z \Longrightarrow P\ y\ z \Longrightarrow P\ x\ z)$$
$$\Longrightarrow P\ a\ b$$

Koinduktive Beweise

Induktionsschema für *rtc*: *rtc.induct*

$$\mathit{rtc}\ r\ a\ b \Longrightarrow (\bigwedge\ x.\ P\ x\ x)$$
$$\Longrightarrow (\bigwedge\ x\ y\ z.\ r\ x\ y \Longrightarrow \mathit{rtc}\ r\ y\ z \Longrightarrow P\ y\ z \Longrightarrow P\ x\ z)$$
$$\Longrightarrow P\ a\ b$$

Koinduktionsschema für *rtc*: *rtc.coinduct*

$$X\ a\ b \Longrightarrow$$
$$(\bigwedge a\ b.\ X\ a\ b \Longrightarrow$$
$$(\exists x.\ a = x \wedge b = x) \vee$$
$$(\exists x\ y\ z.\ a = x \wedge b = z \wedge r\ x\ y \wedge (X\ y\ z \vee \mathit{rtc}\ r\ y\ z))) \Longrightarrow$$
$$\mathit{rtc}\ r\ a\ b$$

Beweismethoden: *coinduct*/*coinduction* – analog zu *induct*/*induction*

Koinduktive Datentypen und Funktionen

Beispiel: Lazy Listen

codatatype *'a llist = lnull: LNil | LCons (lhd: 'a) (ltl: "'a llist")*

Auch hier *duale Sichtweise*: Elemente werden erzeugt, statt abgebaut.

Beispiel: Iterate

primcorec *literate :: "('a ⇒ 'a) ⇒ 'a ⇒ 'a llist"*
where *"literate f s = LCons s (literate f (f s))"*

Mehr: siehe Sitzung.

Teil XXXIII

Dokumentenerzeugung

Dokumentenerzeugung

Isabelle kann Theorien mit LaTeX schön setzen.

Dazu muss man eine *Sitzung* definieren. Am einfachsten geht das mit
`isabelle mkroot -d name`.

Die Datei `ROOT` führt alle verwendeten Theorien auf.
Die Datei `document/root.tex` enthält den LaTeX-Rahmen.

Man lässt Isabelle mit

```
isabelle build -D .
```

die Theorien verarbeiten und die PDF-Dateien erzeugen.

LaTeX in Isabelle-Theorien

Normaler Text (einschließlich LaTeX-Makros) kann mittels

text ‹*bla bla* ›

eingefügt werden.

Kommentare (`(* bla bla *)`) erscheinen *nicht* im Dokument!

Strukturierung

Statt LaTeX-Befehle wie `\section`, `\subsection` etc. in **text**-Blöcke einzubauen kann man die entsprechenden Isabelle-Befehle

- **chapter** (bei geeigneter `\documentclass`)
- **section**
- **subsection**
- **subsubsection**

Ähnlich wie in Markdown können Listen auch ohne LaTeX-Befehl angegeben werden. Dazu werden spezielle Bullet-Points verwendet. Diese kann man in jEdit mittles `\item` für ungeordnete und `\enum` für geordnete Listen eingeben.

Anti-Quotations

Man kann Ausdrücke verschiedener Art von Isabelle in das Dokument einfügen lassen:
Nach

```
definition N :: nat where "N = 0"
theorem great_result: "N = N * P" unfolding N_def by simp
```

wird aus

```
text ⟨
  After defining @{thm N_def} we were finally able
  to prove @{thm great_result}.
⟩
```

in der Dokumentausgabe

After defining `N = 0` we were finally able to prove `N = N * ?P`.

Weitere Anti-Quotations

Neben `@{thm ...}` sind noch nützlich:

- `@{theory ...}` verweist auf einen (importierten) Theorie-Namen,
- `@{term ...}` setzt einen Term,
- `@{term_type ...}` ebenso, aber mit Typ,
- `@{typ ...}` setzt einen Typ,
- `@{value ...}` evaluiert einen Term und zeigt das Ergebnis,
- `@{text ...}` setzt beliebigen Text im Isabelle-Stil.

Während `@{thm ...}` garantiert, dass nur bewiesenes gedruckt wird, überprüfen die anderen nur die Typisierung, und mit `@{text ...}` lässt sich alles ausgeben.

Beim Ausgeben von Lemmas ist oft `@{thm great_result[no_vars]}` schöner als `@{thm great_result}`.

Reihenfolge der Dateien

Standardmäßig enthält `document/root.tex` den Befehl `\input{session}` und `session.tex` (von Isabelle erstellt) enthält für jede Theorie *foo* eine Zeile `\input{Example.tex}`.

Man kann natürlich auch die Theorie-Dateien direkt in `document/root.tex` einbinden, etwa um dazwischen noch Text wie Kapitelüberschriften oder Einleitungen zu setzten.

Auch will man vielleicht in der Einleitung schon auf alle Definitionen und Ergebnisse vorgreifen. Dazu erstellt man z.B. eine Theorie *Introduction* und bindet diese in `document/root.tex` am Anfang ein.

Für Theorien, die in `ROOT` mit der Option `document = false` versehen sind, werden nicht in das Dokument aufgenommen (die trotzdem erzeugte `.tex`-Datei ist leer).

Mehr Informationen

zu mehr Anti-Quotations siehe das Isabelle Referenz-Handbuch (`isabelle doc isar-ref`).

Für mehr LaTeX-Spielereien wie z.B. die Ausgabe

$$\frac{P\ 0 \qquad \bigwedge nat.\ \dfrac{P\ nat}{P\ (Suc\ nat)}}{P\ nat}$$

für

```
text ⟨\begin{center}
  @{thm[mode=Rule] nat.induct[no_vars]}
\end{center} ⟩
```

siehe `isabelle doc sugar`.

Karlsruher Institut für Technologie

Teil XXXIV

Projektvorstellungsvorbereitung

Projektvorstellung

- Ergebnisse werden im Oberseminar des Lehrstuhls vorgestellt
- Termin: 19. Juli 2016, **16 Uhr**
- Ort: Seminarraum 010
- 10 Minuten pro Gruppe
- Folien als PDF vorher per Mail an `breitner@kit.edu`
  oder `denis.lohner@kit.edu`
  oder auf USB-Stick mitbringen.

Ablauf (Teil 1)

- 1. Gruppe: Flatt, Kaiser, Stumpf
  - Aufgabe und Beweis vorstellen
  - Erfahrungen, Feedback, Statistiken (lines of code, Anzahl der Lemmas)

Ablauf (Teil 2)

- 2. Gruppe: Priesner, Priesner
  - Syntax und Semantik vorstellen
  - Determinismusbeweis (nur sehr kurz)
  - Definition der Konstantenpropagation und -Faltung
  - Semantikerhaltung und Beschleunigungsaussage
  - Erfahrungen, Feedback, Statistiken (lines of code, Anzahl der Lemmas)
- 3. Gruppe: Böhm, Scholz
  - Definition der Konstantenpropagation und -Faltung
  - Semantikerhaltung und Beschleunigungsaussage
  - Idempotenzbeweis
  - Erfahrungen, Feedback, Statistiken (lines of code, Anzahl der Lemmas)
- 4. Gruppe: Graf, Kiefer
  - Definition der Konstantenpropagation und -Faltung
  - Semantikerhaltung und Beschleunigungsaussage
  - Typisierbarkeitserhaltung
  - Erfahrungen, Feedback, Statistiken (lines of code, Anzahl der Lemmas)

Allgemeine Hinweise

- Beweisideen deutlich machen
- Beweismethode nennen (Induktion etc.)
- Bei Induktion Verallgemeinerung und Invarianten zeigen und begründen
- Nicht jeden Fall eines Beweises einzeln durchnudeln

Siehe auch Termin zur Dokumentenerzeugung! Generierten LaTex-Code aus `output/document/Theorie.tex` kopieren ist ok und üblich.