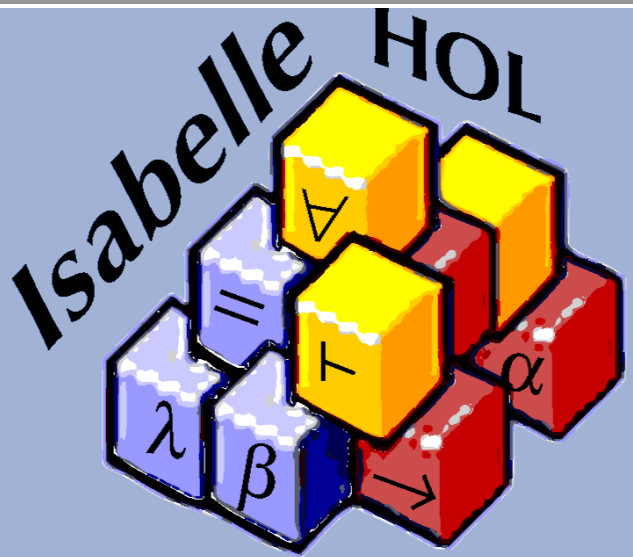
Karlsruher Institut für Technologie

Theorembeweiserpraktikum

Anwendungen in der Sprachtechnologie

LEHRSTUHL PROGRAMMIERPARADIGMEN



Karlsruher Institut für Technologie

Teil VII

Automatische Taktiken

Automatische Taktiken

Bisher kennen wir `simp` als Termersetzungstaktik
Viele weitere automatische Taktiken für den *logical reasoner*
Unterscheiden sich in

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

z.B. entspricht `simp_all` der Taktik `simp`, jedoch werden Termersetzungsregeln auf jedes Zwischenziel angewendet

Im Folgenden ein paar Taktiken vorgestellt

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
von den Entwicklern empfohlene Taktik

`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

`fastsimp`

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

`fastsimp` gilt als langsamste Taktik, von Entwicklern als *legacy* bezeichnet in der Praxis allerdings 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`(fastsimp 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**`(fastsimp dest:foo)`
- alles beliebig kombinierbar, z.B.

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

Karlsruher Institut für Technologie

Teil VIII

Primitive und wechselseitige Rekursion

Deklaration von Funktionen

Festlegen von Namen und Signatur einer Funktion: Deklaration
dazu in Isabelle das Schlüsselwort **consts:**

consts `length :: "'a list ⇒ nat"`

Vorsicht! Gibt der Funktion noch keinerlei Semantik!
Wird in den Übungen verwendet, um Funktionen einzuführen, die sie selbst noch definieren (also mit Semantik versehen) sollen
Selbst bitte vermeiden! Immer gleich Deklaration und Definition

Rekursive Datentypen

Viele Datentypen mit Selbstbezug, z.B.

- natürliche Zahl (ungleich 0) ist Nachfolger einer natürlichen Zahl
- nichtleere Liste ist Liste mit zusätzlichem Kopfelement
- nichtleere Menge ist Menge mit einem zusätzlichen Element

Rekursive Datentypen

Viele Datentypen mit Selbstbezug, z.B.

- natürliche Zahl (ungleich 0) ist Nachfolger einer natürlichen Zahl
- nichtleere Liste ist Liste mit zusätzlichem Kopfelement
- nichtleere Menge ist Menge mit einem zusätzlichen Element

Formalisierung in Isabelle/HOL am Bsp. natürliche Zahlen:

datatype *nat = 0 | Suc nat*

Also Konstruktoren von nat: 0 und Suc (Präfix)

Parametertypen

verschiedene Typen in Containerdatentypen: Parametertyp `'a`
kann bei Verwendung entprechen initialisiert werden (muss aber nicht)

Parametertypen

verschiedene Typen in Containerdatentypen: Parametertyp `'a`
kann bei Verwendung entprechen initialisiert werden (muss aber nicht)

Beispiel: Listen mit Typparameter

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

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

Parametertypen

verschiedene Typen in Containerdatentypen: Parametertyp `'a`
kann bei Verwendung entprechen initialisiert werden (muss aber nicht)

Beispiel: Listen mit Typparameter

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

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

Funktionsdeklaration kann jetzt z.B. so aussehen:

```
consts foo :: "nat list ⇒ bool"
consts bar :: "nat ⇒ bool list ⇒ nat"
consts zip :: "'a list ⇒ 'a"
```

primrec

Definition von Funktionen über rekursive Datentypen: **primrec**
kombiniert Deklaration und Definition (frühere **consts** löschen!)
ein Parameter der Funktion muss in seine Konstruktoren aufgeteilt werden

Beispiel:

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

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

primrec

Es müssen nicht alle Konstruktoren spezifiziert werden:

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

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

Aussagen über nicht-enthaltene Konstruktoren wie z.B. `hd([])` kaum möglich, da nichinterpretierte Funktionssymbole, d.h. nicht vereinfachbar

@ 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 Deklaration/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 Deklaration/Definition?

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

Strukturelle Induktion

Beweise über rekursive Datentypen mittels struktureller Induktion
d.h. Induktion über Konstruktoren des Datentyps

In Isabelle/HOL:

```
 hd_Cons_tl: "xs ≠ [] ⟹ hd xs # tl xs = xs"
apply(induct xs)
apply auto
done
```

wendet strukturelle Induktion mit Datentypkonstruktoren von `xs` an
automatische Taktik beendet Beweis

Strukturelle Induktion

Beweise über rekursive Datentypen mittels struktureller Induktion
d.h. Induktion über Konstruktoren des Datentyps

In Isabelle/HOL:

```
 hd_Cons_tl: "xs ≠ [] ⟹ hd xs # tl xs = xs"
apply(induct xs)
apply auto
done
```

wendet strukturelle Induktion mit Datentypkonstruktoren von `xs` an
automatische Taktik beendet Beweis

Induktionsregel für Listen

List.list.induct:

```
⟦?P []; ⋀a list. ?P list ⟹ ?P (a # list)⟧ ⟹ ?P ?list
```

Probleme mit Induktion

Problem: zu spezielle Induktionshypothesen

lemma `"rev xs = rev ys ⟹ xs = ys"`

- Induktion auf `xs` ermöglicht Lösen des `[]`-Falles
  durch Anwenden der Definition von `rev` (siehe `rev.simps`)

Probleme mit Induktion

Problem: zu spezielle Induktionshypothesen

lemma `"rev xs = rev ys ⟹ xs = ys"`

- Induktion auf `xs` ermöglicht Lösen des `[]`-Falles
  durch Anwenden der Definition von `rev` (siehe `rev.simps`)
- bei Induktionsschritt bleibt:

  ```
  ⋀a xs.
     ⟦rev xs = rev ys ⟹ xs = ys; rev (a # xs) = rev ys ⟧
  ⟹ a # xs = ys
  ```

  `rev ys` kann nicht gleich `rev xs` und `rev (a # xs)` sein!

  ⇒ Induktionshypothese nicht verwendbar

Probleme mit Induktion

Problem: zu spezielle Induktionshypothesen

lemma `"rev xs = rev ys ⟹ xs = ys"`

- Induktion auf `xs` ermöglicht Lösen des `[]`-Falles
  durch Anwenden der Definition von `rev` (siehe `rev.simps`)
- bei Induktionsschritt bleibt:

  ```
  ⋀a xs.
     ⟦rev xs = rev ys ⟹ xs = ys; rev (a # xs) = rev ys ⟧
  ⟹ a # xs = ys
  ```

  `rev ys` kann nicht gleich `rev xs` und `rev (a # xs)` sein!

  ⇒ Induktionshypothese nicht verwendbar

  ⇒ nicht ohne weiteres lösbar!

Probleme mit Induktion

Lösung:

`ys` muss in der Induktionshypothese freie Variable sein!

Probleme mit Induktion

Lösung:

ys muss in der Induktionshypothese freie Variable sein!

Umsetzung:

ys nach *arbitrary* Schlüsselwort in Induktionsanweisung, damit Induktionshypothese für *ys* meta-allquantifiziert:

apply *(induct xs arbitrary: ys)*

Probleme mit Induktion

Lösung:

ys muss in der Induktionshypothese freie Variable sein!

Umsetzung:

ys nach *arbitrary* Schlüsselwort in Induktionsanweisung, damit Induktionshypothese für *ys* meta-allquantifiziert:

apply *(induct xs arbitrary: ys)*

Resultiert in Induktionsschritt:

```
⋀a xs ys.
   ⟦⋀ys. rev xs = rev ys ⟹ xs = ys; rev (a # xs) = rev ys⟧
   ⟹ a # xs = ys
```

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

wechselseitige Rekursion

Probleme bei bisheriger Rekursion: was tun bei

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

bei der Definition verwenden?

wechselseitige Rekursion

Probleme bei bisheriger Rekursion: was tun bei

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

bei der Definition verwenden?

Beispiel:

Bäume mit beliebigem Verzweigungsgrad
jeder Knoten verwaltet eine Liste von Nachfolgerbäumen
Datentyp einfach wie bisher definiert:

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

Datentyp wechselseitig rekursiv definiert für sich und Liste seiner Typen

wechselseitige Rekursion

Karlsruher Institut für Technologie

Definition der Höhenfunktion für solche Bäume

Ansatz:

```
primrec height :: "'a tree ⇒ nat"


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

wechselseitige Rekursion

Karlsruher Institut für Technologie

Definition der Höhenfunktion für solche Bäume

Ansatz:

```
primrec height :: "'a tree ⇒ nat"


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

brauchen Definition der Höhe für Liste von Bäumen!

wechselseitige Rekursion

Karlsruher Institut für Technologie

Definition der Höhenfunktion für solche Bäume

Ansatz:

```
primrec 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)"
```

brauchen Definition der Höhe für Liste von Bäumen!

dazu gleichzeitige Definition der Funktion *height* für *'a tree*
und *heights* für *'a tree list*
sowohl **primrec** als auch **fun** verwendbar

Beweisen mit wechselseitig rekursiv definierten Datentypen

Karlsruher Institut für Technologie

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

```
lemma
        "height t > 0"
 apply(induct 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"
 apply(induct 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"
 apply(induct 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"*
apply(*induct 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 ≥ 0"
 apply(induct 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 ≥ 0"
apply(induct t and ts)
```

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

Beweisen mit wechselseitig rekursiv definierten Datentypen

Karlsruher Institut für Technologie

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 ≥ 0"
 apply(induct t and ts)  apply auto  done
```

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Höhe jedes Baums größer als 1 ist
4. Versuch: Variante mit *fun*

lemma fixes *t::"'a tree"* **and** *ts::"'a tree list"*
shows *"height t > 0"* **and** *"heights ts* $\geq$ *0"*
apply(*induct rule:height_heights.induct*) **apply** *auto* **done**

fun definiert Induktionsregel für wechselseitige Rekursion:
height_heights.induct

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?

Kennen allgemein schon die Lösung: **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?

Kennen allgemein schon die Lösung: **arbitrary**

```
lemma "P t" "Q a t ts"
apply (induct t and ts)
```

`Q` braucht `a` in 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?

Kennen allgemein schon die Lösung: **arbitrary**

```
lemma "P t" "Q a t ts"
apply (induct t and ts arbitrary: and a)
```

auch hinter **arbitrary** die zu quantifizierenden Variablen für jedes Lemma mit **and** trennen