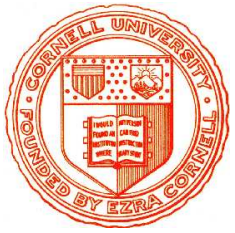
Automatisierte Logik und Programmierung

Einheit 12



Taktiken
Benutzerdefinierbare Beweisstrategien

1. Grundkonzept und Arbeitsweise
2. Programmierung von Taktiken
3. Rewriting als Taktiken
4. Erfahrungen im praktischen Umgang

Benutzerdefinierbare Beweisstrategien

- **Rein interaktive Beweisführung ist mühsam**
  - Fast alle formalen Beweise benötigen sehr viele Beweisschritte
  - Die meisten Beweisschritte sind naheliegend
  - Benutzer verwenden implizite Strategie bei der Beweisführung
  - Ein Computer könnte diese Strategie genauso gut ausführen
- **Vordefinierte Beweisprozeduren helfen nicht immer**
  - Sie können nur allgemeine Situationen abdecken und lösen
    - Die "Standardtricks" von Anwendungstheorien sind nicht erfaßt
  - Sie liefern unnatürliche Argumente in konkreten Anwendungen
    - Bei Schlüssen über $\mathbb{R}$ würden z.B. ständig Cauchy-Folgen sichtbar
- **Unterstütze benutzerdefinierte Beweisstrategien**
  - Benutzer muß System um Strategieprogramme selbst ergänzen dürfen
  - Die Korrekheit derartig erstellter Beweise muß sichergestellt sein

Beweisautomatisierung durch Taktiken

- **Beweisstrategien sind Meta-Level Konzepte**
  - Methode, die beschreibt wann welche Inferenzregel zu verwenden ist
  - Präzisiert Vorgehensweise, die aus Erfahrung und Planung entsteht
  - Meist formuliert als Vorschrift in natürlicher Sprache
    - z.B. *Um eine Fallunterscheidung* $P \vee \neg P$ *einzuführen, verwende zunächst* `cut` *und anschließend* `magic` *und* `orL`
- **Taktiken formalisieren Beweisstrategien**
  - Formulierung der Meta-Level Methode als Meta-Level Programm
    - ```
      let cases P = Cut (mk_or_term P (mk_not_term P))
                    THENL [magic; orE (-1)]
      ```
  - Programmierung der Strategie kombiniert vorhandene Regeln/Taktiken
  - System stellt Mechanismen für einfache Kombinationen bereit
  - Verwendung von ML erlaubt auch komplexere Strategieprogramme
      ```

Taktikanwendung vs. Elementarbeweis

THM cases

```
# top
⊢ C

BY cases A

# 1
1. A
⊢ C

# 2
1. ¬A
⊢ C
```

THM cases in steps

```
# top
⊢ C

BY cut 1 A∨¬A

# 1
⊢ A∨¬A

BY magic

# 2
1. A∨¬A
⊢ C

BY orE 1

# 2 1
1. A
⊢ C

# 2 2
1. ¬A
⊢ C
```

Taktische Beweisführung

- **Refiner verwendet Taktiken als Basismechanismus**
  - Basistaktiken werden mittels `refine` aus Regelschemata erzeugt
    (einzige Möglichkeit, tatsächliche Beweisobjekte zu generieren)
  - Auswahlstrategie analysiert Beweisziel und ruft existierende Taktiken auf
- **Flexibler Mechanismus für vielfältige Einsatzbereiche**
  - Planung von Beweisen und Suche nach Beweisen
  - Strukturierung von Beweisen durch Verstecken überflüssiger Details
  - Abgeleitete Inferenzregeln für Anwendungstheorien (z.B. $\mathbb{R}$, Graphen,..)
  - Austesten komplexer Beweis-/Syntheseverfahren in sicherer Umgebung
- **Sicherer Mechanismus: Taktiken sind immer korrekt**
  - Das Resultat einer Taktik-Anwendung ist immer ein gültiger Beweis des zugrundeliegenden logischen Kalküls
  - Taktiken können schlimmstenfalls erfolglos sein oder nicht terminieren

Programmierung von Taktiken

- **Explizite Umwandlung von Regeln mittels `refine`**
  - Erforderlich für Erzeugung der Basistaktiken im System
  - Benutzer sollten `refine` nur in Sonderfällen verwenden
    - z.B. um neu eingeführte Regeln umzuwandeln
- **Verwendung vordefinierter Standardtaktiken**
  - System sollte kleinen Satz von Basistaktiken bereitstellen
    - Alle Inferenzregeln sind repräsentiert
    - Gleichartige Regeln werden zusammengefaßt
    - Einfachste Regelparameter werden automatisch bestimmt
- **Komposition von Taktiken durch Tacticals**
  - Operationen, die Taktiken zu neuen Taktiken zusammensetzen
  - System sollte kleinen Satz von Standardtacticals bereitstellen
  - Hinreichend für einfache Anwendungen
- **Metalevel-Steuerung von Taktikanwendungen**
  - ML-Programme analysieren Beweisziel und Kontext

Umwandlung von Regeln in Taktiken

- **Umwandlung der Regel `hypothesis`**

```
let get_pos_hyp_num i proof =
  if i<0 then length (hypotheses proof)+1+i else i ;;
let NthHyp i proof =
  let i' = get_pos_hyp_num i proof
  in
     Refine 'hypothesis' [mk_int_arg i'] proof ;;
```

- **Ausdünnen überflüssiger Hypothesen**

```
let Thin i proof =
  let i' = get_pos_hyp_num i p roof
  in
    if i=0 then failwith 'Thin: cannot thin conclusion'
    else Refine 'thin' [mk_int_arg i'] proof ;;
```

- **Wichtige Hilfsoperationen**
  - `new_var`: automatische Auswahl ('guter') neuer Namen
  - `level_arg`: automatische Bestimmung des Universums

  ⋮

Basistaktiken in Nuprl: Dekomposition

Manual §8

- **D i : Top-Level Dekomposition**

  i=0: Dekomposition der Konklusion (Formationsregeln)

  · Ziele $H \vdash x : S \to T$, $H \vdash \forall x{:}T.P(x)$, $H \vdash A \wedge B$, ...

  i>0: Dekomposition der Hypothese i (Eliminationsregeln)

  i<0: Dekomposition der Hypothese n-i

  · Ziele $H, \forall x{:}T.P(x), H' \vdash C$, $H, A \wedge B, H' \vdash C$, ...

- **EqD i / MemD i : Dekomposition einer Gleichheit**

  – Gleichheits bzw. Typzugehörigkeitsregeln

  · Ziele $H \vdash \lambda y.s = \lambda z.t \in x : S \to T$, $H \vdash \langle a, b \rangle \in A \wedge B$, ...

  – In Hypothesen mit Substitution realisiert (nur Produkte / Funktionen)

- **EqTypeD i / MemTypeD i :**

  – Dekomposition des Typs einer Gleichheit (für Teilmengen/Quotienten)

- **Einfach zu programmieren**

  – Bestimme opid der Hypothese, wähle passende Regel und Parameter

Einfügen von Steuerungsparametern

- **Standardtaktiken bestimmen Regelparameter**
  - Einfach für Variablennamen und Universenlevel
  - Heuristik für Bestimmung von Abhängigkeiten
  - Automatische Bestimmung ist nicht immer möglich oder 'korrekt'
  - Benutzer muß Parameter ergänzen oder überschreiben können
- **Funktionen modifizieren Standardtaktiken**
  - Name x einer neuen Variablen — `New [x] (D 0)`
  - Typ T eines Teilterms im Beweisziel — `With` $x{:}S{\to}T$ `(MemD 0)`
    - z.B. für $H \vdash f\ t \in T$
  - Term, s, der für eine Variable einzusetzen ist — `With` s `(D 0)`
    - z.B. für $H \vdash \exists x{:}T.P(x)$
  - Level j eines Universums — `At` j `(D 0)`
  - Abhängigkeit eines Terms C von Variable x — `Using` $[z,C]$ `(D 0)`
    - z.B. für Substitution in $y{=}x, y{=}f(x) \vdash x{=}f(x)$

Basistaktiken in Nuprl (2)

- **Berechnungsregeln**
  - `Reduce` i: Top-Level Reduktion von Hypothese oder Konklusion
    · Ziele $H \vdash (\lambda y.s)t{=}t' \in T$, $H \vdash \mathsf{let}\ \langle x, y\rangle = \langle s, t\rangle\ \mathsf{in}\ u = t' \in T$,...
- **Strukturelle Regeln**
  - `Hypothesis`, `Declaration`: Regel formuliert als Taktik
  - `Assert` t : Einführen von Zwischenbehauptungen (`cut`)
- **Regeln der Logik erster Stufe:** `andR`, `orR`, ..., `exL` i
  - Umformulierung der Standard-Dekompositionsregeln
  - Versuchen, Wohlgeformtheitsziele automatisch zu beweisen
- **Induktionsregeln** `NatInd` i, `IntInd` i, `ListInd` i
  - Erweiterung der Standard-Dekompositionsregel für Zahlen und Listen
    Erhält Abhängigkeiten, die von der Basisregel ignoriert werden
  - Vollständige Induktion auf natürlichen Zahlen: `CompNatInd` i
    Lemma $\vdash$ `∀P:ℕ→ℙ.(∀i:ℕ.(∀j:ℕ.j<i⇒P(j))⇒P(i))⇒∀i:ℕ.P(i)`

Unterstützung für komplexere Beweise

Standardschlußmethoden in interaktiven Beweisen

- **Fallanalysen**
  - Analyse Boolescher Variablen: `BoolCases` *i*
  - Allgemeine Fallunterscheidung: `Cases [`t_1`;..;`t_n`]`
  - Analyse entscheidbarer Aussagen (Fälle P und $\neg P$) `Decide` P
- **Chaining (Verkettung von Argumenten)**
  - Instantiierung quantifizierter Aussagen: `InstHyp [`t_1`;..;`t_n`]` *i*
  - Vorwärtsverkettung von Hypothesen: `FHyp` *i* `[`h_1`;..;`h_n`]`,
  - Rückwärtsverkettung im Beweisknoten: `BHyp` *i*,
    `" "` durch Hypothesen & Lemmata: `Backchain` *bc_names*
- **Autotaktik (triviale Schlüsse)** `Auto`
  - Versucht alle einfachen Beweisketten automatisch zu finden
  - Sehr hilfreich aber üblicherweise etwas undurchsichtig

Tacticals

- **Unterstützung für einfache Komposition von Taktiken**
  - Hintereinanderausführung von Taktiken t_1 `THEN` t_2
    - z.B. $A \wedge (B \wedge C) \vdash G$ `BY andL (-1) THEN andL (-1)`
  - Unterschiedliche Behandlung der Unterziele t `THENL` $[t_1; ..; t_n]$
    - z.B. $H \vdash \forall x{:}\mathbb{N}.P(x) \wedge Q(x)$ `BY D 0 THENL [andR; Auto]`
  - Wahl alternativer Taktiken t_1 `ORELSE` t_2
    - z.B. $H \vdash A \vee B$ `BY (orR1 THEN Auto) ORELSE (orR1 THEN Auto)`
  - Wiederholte Anwendung von Taktiken `Repeat` t
    - z.B. $H \vdash \forall x_1, x_2, ..x_n{:}T.P$ `BY Repeat allI`
- **Formulierung als kommandoartige Taktiksprache**
  - Adäquater für Beschreibung der Vorgehensweise
- **Implementierung als Funktionen höherer Ordnung**
  - Tactical setzt Dekompositions- und Validierungsfunktionen zusammen
  - Infixschreibweise versteckt funktionale Sichtweise der Komposition

Implementierung von Tacticals

```
ml_curried_infix ‘THENL‘  ;;
ml_curried_infix ‘ORELSE‘ ;;

let $THENL (tac: tactic) (tac_list : tactic list) (pf:proof) =
  let subgoals, val = tac pf
  in
    if not length tac_list = length subgoals then fail
    else let subL, valL = map_apply tac_list subgoals
         in
            (flatten subL),
            \pfs.val((mapshape(map length subL) valL) pfs) ;;

let $ORELSE (t1:tactic) (t2:tactic) pf =  t1 pf ? t2 pf   ;;

% Complete t: Apply t only if it completes the proof %
let Complete (tac:tactic) (pf:proof) =
  let subgoals, val = tac pf
  in
     if null subgoals then subgoals, val else fail;;
```

Die wichtigsten vordefinierten Tacticals

t_1 `THEN` t_2:	Wende t_2 auf alle von t_1 erzeugten Teilziele an
t `THENL` `[`$t_1; ..; t_n$`]`:	Wende t_i auf das i-te von t erzeugte Teilziel an
t_1 `ORELSE` t_2:	Wende t_1 an. Falls dies fehlschlägt, wende t_2 an
`Repeat` t:	Wiederhole Taktik t bis sie fehlschlägt
t_1 `THENA` t_2:	Wende t_2 auf alle von t_1 erzeugten Hilfsziele an
t_1 `THENW` t_2:	Wende t_2 auf alle von t_1 erzeugten wf-ziele an
`Try` t:	Wende t an, Bei Fehlschlag lasse den Beweis unverändert
`Complete` t:	Wende t nur an, wenn der Beweis vollständig wird
`Progress` t:	Wende t nur an, wenn ein "Fortschritt" erzielt wird
`RepeatFor` i t:	Wiederhole t genau i mal
`AllHyps` t:	Wende t auf alle möglichen Hypothesen an
`OnSomHyp` t:	Wende t auf die erstmögliche Hypothese an

Entwurf einer komplexeren Taktik mit Tacticals

```
let Equality = Refine ‘equality‘ [] ;;
and Arith    = Refine ‘arith‘
                      [mk_term_arg
                        (mk_universe_term
                        (mk_level_exp [‘i‘,0]))];;

% Immediate: Perform trivial steps automatically %
let Immediate =
              Declaration
      ORELSE Hypothesis
      ORELSE OnSomeHyp falseL
      ORELSE Contradiction
      ORELSE Equality
      ORELSE Arith
      ORELSE D 0   ORELSE OnSomeHyp D
      ORELSE EqD 0 ORELSE OnSomeHyp EqD ;;
```

Verwendet ausschließlich Standardtaktiken/-tacticals

Programmierung der Taktik Hypothesis

Anwendung von NthHyp auf alle Hypothesen

```
let OnHyp i (T: int->tactic) p =
  T (get_pos_hyp_num i p) p
;;
let OnSomeHyp T p =
  letrec Aux i p' =
    if i = 0 then failwith 'OnSomeHyp'
    else (OnHyp i T ORELSE Aux (i-1)) p'
  in
    Aux (length (hypotheses proof)) p
;;
let Hypothesis = OnSomeHyp NthHyp ;;
```

Programmierung der Taktik `Contradiction`: Metalevel-Steuerung von Taktikanwendungen

Finde zwei einander widersprechende Hypothesen

```
let Contradiction proof =
  let facts    = map (fst o snd o dest_declaration)(hypotheses proof) in
  let facts_and_nums = (map2 o pair) facts (upto 1 (length facts))    in
  let negative_hyps  = filter (\t,i. is_not_term t) facts_and_nums     in
  let positive_hyp, negative_hyp =
     (first_value
        (\t,i. let negated_term = dest_not t in
               let pos_hyp_num  =
                 first_value (\t,i. if t = negated_term then i else fail)
                             facts_and_nums
               in
                  pos_hyp_num,i
         )
        negative_hyps
      ?
      failwith 'Contradiction'
     )
  in   if negative_hyp > positive_hyp
          then (notL negative_hyp THEN NthHyp  positive_hyp   ) proof
          else (notL negative_hyp THEN NthHyp (positive_hyp-1)) proof
;;
```

Metalevel Analyse: Instantiierung von Quantoren

Bestimme Werte für Variablen durch Matching

```
let match_subEx quantified_term assumption  =
   letrec matchAux vars exprop =
     map (\var.assoc var (match vars exprop assumption)) (rev vars)
          % Terms must fit quantifier order%
     ?
     let var,type,prop = dest_exists exprop in matchAux (var.vars) prop
   in
      matchAux [] quantified_term
;;
letrec exIon terms pf =
    let t.rest = terms in (exI t THEN exIon rest) pf
    ?
    Id pf
;;
 let InstantiateEx =
   let InstEx_aux pos pf =
      let sigma = match_subEx (conclusion pf) (type_of_hyp pos pf)   in
         (exIon (map snd sigma) THEN (NthHyp pos)) pf
   in
      OnSomeHyp InstEx_aux
;;
```

Rewriting (Termersetzung)

- **Eine der häufigsten Argumentationsformen**
  - **Ersetze Terme in Beweisen durch gleichwertige Terme**
  - In CTT hängt die Gleichwertigkeit vom Typ ab
- **Auswertung von Teiltermen**
  - Finde und reduziere Redizes alle in einer 'Klausel' c `Reduce` c
- **Verwaltung benutzerdefinierter Abstraktionen**
  - Auflösen einer Definition `Unfold` *name* c
  - Zurückfalten der rechten Seite einer Definition `Fold` *name* c
- **Explizite Substitution**
  - Ersetze Term t_1 durch t_2 in Klausel c `Subst` $t_1 = t_2 \in T$ c
  - Verwende Gleichheit in Klausel c_1 als Substitution `HypSubst` c_1 c_2

Simplifier und Rewrite-Pakete

Manual §8.9

- **Termersetzung lebt von Automatisierung**
  - Beweise basieren oft auf einer Serie von Ersetzungsschritten
  - Ersetzungen stützen sich auf große Menge möglicher Gleichheiten
  - Strategie sollte Gleichungen automatisch identifizieren und anwenden
- **Konflikt zwischen Effizienz und Korrektheit**
  - Ersetzungen sollten schnell gefunden/ausgeführt werden (untypisiert)
  - Abstützung auf Basiskalkül benötigt Serie von Lemmata zur Rechtfertigung der durchgeführten Ersetzungen
  - Trenne Termersetzung vom Beweiskalkül ('reiche Beweise nach')
- **Nuprl's Rewrite Paket**
  - Funktionen zur Ersetzung von Termen in Ausdrücken (conversions)
  - Programmierbar durch vordefinierte conversions und conversionals
  - Verwendet Rewrite Lemmas zur Rechtfertigung der Ersetzungen
  - Unterstützt Vielfalt von Äquivalenzrelationen (nicht nur Gleichheit)
  - Taktiken zur Anwendung von conversions auf Klauseln im Beweis

Isabelle Simplifier arbeitet ähnlich (aber verzahnte Operation, Caching, ...)

Erfahrungen im praktischen Umgang

- **Sehr hilfreich für Anwendungen**
  - Keine Veränderung des eigentlichen Inferenzsystems erforderlich
  - Benutzer können Inferenzsystem an eigene Bedürfnisse anpassen
  - Benutzerdefinierte Strategien produzieren keine falschen Ergebnisse
- **Gut für Experimente**
  - Ideen können unmittelbar ausprobiert werden
- **Sinnvoll für "kontrollierte" Inferenzen**
  - Repräsentation von Schlüssen in speziellen Anwendungsbereichen
  - Macro-Inferenzen: Schließen auf höherem Niveau
  - Begrenzte Suche nach Beweisen
- **Nicht sinnvoll für "universelle" Beweissuche**
  - Zu langsam: jeder Taktikschritt modifiziert den Beweisbaum
  - Zu unkontrolliert: Taktik kann unverständliche Teilziele zurückgeben