Automatisierte Logik und Programmierung

Einheit 13

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

    Um eine Aussage durch Fallunterscheidung über ein (entscheidbares) Prädikat P zu beweisen, füge die Zwischenbehauptung $P \vee \neg P$ ein, beweise diese separat und spalte anschließend die Disjunktion auf

- **Taktiken formalisieren Beweisstrategien**
  - Formulierung der Meta-Level Methode als Meta-Level Programm

```
let Decide P = Cut (mk_or_term P (mk_not_term P))
               THENL [ProveDecidable; orL (-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 Decide P

# 1
1. P
⊢ C

# 2
1. ¬P
⊢ C
```

THM cases in steps

```
# top
⊢ C

BY Cut 1 P∨¬P

# 1
⊢ P∨¬P

BY ProveDecidable

# 2
1. P∨¬P
⊢ C

BY orL 1

# 2 1
1. P
⊢ C

# 2 2
1. ¬P
⊢ C
```

Elementarer Beweis kann durch Expansion rekonstruiert werden

Taktiken haben vielfältige Einsatzbereiche

- **Strukturierung von Beweisen**

$\vdash\ ((P \vee Q) \wedge ((P \Rightarrow R) \wedge (Q \Rightarrow R))) \Rightarrow R$	BY `impliesR THEN Repeat andL`
1. $z{:}P \vee Q, g{:}P \Rightarrow R, h{:}Q \Rightarrow R \vdash R$	BY `orL`
1.1. $p{:}P, g{:}P \Rightarrow R, h{:}Q \Rightarrow R \vdash R$	BY `impliesL` g `THEN axiom`
1.2. $q{:}Q, g{:}P \Rightarrow R, h{:}Q \Rightarrow R \vdash R$	BY `impliesL` h `THEN axiom`

  – Überflüssige Details werden versteckt

- **Implementierung spezieller Arten logischer Argumente**
  – z.B. Taktik `Decide` für allgemeines logisches Schließen
  – Abgeleitete Inferenzregeln für Anwendungstheorien ($\mathbb{R}$, Graphen,..)

- **Automatisierung der Beweisfindung**
  – Planung von Beweisen und Suche nach Beweisen

- **Experimente mit neuen Beweistechniken**
  – Leichtes Implementieren und Austesten komplexer Beweisverfahren in sicherer Umgebung

Wie sicher ist taktische Beweisführung?

- **Taktiken sind konservative Erweiterung des Regelsystems**
  - Alle Basistaktiken werden mittels `refine` aus Regelschemata erzeugt
    (einzige Möglichkeit, tatsächliche Beweisobjekte zu generieren)
  - Taktiksprache unterstützt Verknüpfung von Basistaktiken
    ML ist direkte Formalisierung der Metasprache der Logik
  - Auch komplexe Taktiken stützen sich im Endeffekt auf Basistaktiken
  - Damit sind Taktiken Abkürzungen für komplexe Regelanwendungen
- **Taktiken sind immer korrekt**
  - *Das Resultat einer Taktik-Anwendung ist immer ein gültiger Beweis des zugrundeliegenden logischen Kalküls*
    Taktikbeweis kann immer zu Elementarbeweis expandiert werden
  - Taktiken können schlimmstenfalls erfolglos sein oder nicht terminieren

Programmierung von Taktiken

Taktiken sind Basismechanismus des Refiners

- **Explizite Umwandlung von Regeln mittels `refine`**
  - Nur für Erzeugung der Basistaktiken des Systems erforderlich
  - Benutzer sollten `refine` nur in Sonderfällen verwenden
    (Für low-level Kontrolle oder neuer Regeln bei nichtkonservativen Erweiterungen)
- **Standardtaktiken für häufige Anwendungen**
  - Zusammenfassung gleichartiger Regeln unter einheitlichem Namen
  - Automatische Bestimmung passender Werte für Regelparameter
  - Erzeugung einer überschaubaren Kollektion von Basistaktiken
- **Komposition existierender Taktiken durch Tacticals**
  - Operationen, die Taktiken zu neuen Taktiken zusammensetzen
  - Erfordert Programmierung eines kleinen Satz von Standardtacticals
- **Metalevel-Steuerung von Taktikanwendungen**
  - ML-Programme analysieren Beweisziel und Kontext und steuern damit die Anwendung existierender Taktiken

Umwandlung von Regeln in Taktiken

- **Direkte Umwandlung der Regel** `hypothesis`
  - Benötigt Umwandlung der Hypothesennummer in Regelargument

```
let Hyp i proof = Refine ‘hypothesis‘ [mk_int_arg i’] proof ;;
```

- **Unterstütze Zählung der Hypothesen vom Ende**
  - Viele Regeln fügen neue Hypothesen am Ende der Liste hinzu
  - Zugriff auf “letzte Hypothese” oft einfacher als absolute Zahl
  - Hilfsfunktion wandelt negative Hypothesen in Rückwärtszählung um

```
let get_pos_hyp_num i proof =
    if i<0 then length (hypotheses proof)+1+i else i ;;
```

  - Hypothese 0 wird üblicherweise mit Konklusion gleichgesetzt

- **Hypothesenzugriff mit negativen Hypothesennummern**

```
let NthHyp i proof =
    if i=0 then failwith ‘NthHyp: cannot use conclusion‘
    else let i’ = get_pos_hyp_num i proof
         in Refine ‘hypothesis‘ [mk_int_arg i’] proof ;;
```

Umwandlung von Regeln in Taktiken II

- **Ausdünnen überflüssiger Hypothesen**
  - Gleiche Methodik wie bei Programmierung von `NthHyp`

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

- **Umwandlung der Formationsregel `lambdaEquality`**
  - `lambdaEquality` benötigt Universenlevel und neue Variablennamen
  - Explizite Angabe macht Anwendung der Taktik unnötig mühsam
  - Implementiere Hilfsoperationen zur automatischen Parameterbestimmung
    - `new_var`: Auswahl (‘guter’) neuer Namen aus Kontext
    - `level_arg`: Bestimmung eines passenden Universenlevel-Arguments

```
let lambdaEq proof =
    let T = conclusion proof
    in let Uj = level_arg T proof
       and x  = new_var T proof
       in Refine `lambdaEquality` [Uj; mk_var_arg x] proof ;;
```

Standardtaktiken vereinfachen das Inferenzsystem

- **Regelsystem der Typentheorie ist sehr umfangreich**
  - Namensvielfalt ist trotz der Systematik sehr groß
  - Benutzer müssten sich mehr als 150 Inferenzregeln merken
- **Viele Regeln sind strukturell sehr ähnlich**
  - Formationssregeln zerlegen Top-Level Terme in Komponenten
    Ziele der Art $H \vdash x : S \rightarrow T$, $H \vdash \forall x{:}T.P(x)$, $H \vdash A \wedge B$, ...
  - Eliminationsregeln haben eine ähnliche Funktion auf Hypothesen
    Ziele der Art $H, \forall x{:}T.P(x), H' \vdash C$, $H, A \wedge B, H' \vdash C$, ...
  - Gleichheitsregeln zerlegen Terme einer Gleichheit in Komponenten
    Ziele der Art $H \vdash \lambda y.s = \lambda z.t \in x{:}S \rightarrow T$, $H \vdash \langle a, b \rangle \in A \wedge B$, ...
  - Berechnungsregeln reduzieren ein Redex in einer Gleichung
    Ziele der Art $H \vdash (\lambda x.t)\, s = t_2 \in T$, ...

Gleichartige Regeln sollten unter einem Namen zusammengefaßt werden

Programmierung von Standardtaktiken

- **Top-Level Dekomposition von Termen** D i
  - $i{=}0$: Dekomposition der Konklusion (Formationsregeln)
  - $i{\neq}0$: Dekomposition der Hypothese i (Eliminationsregeln)

    Bestimmung der tatsächlichen Hypothese mit `get_pos_hyp_num`
  - Zugriff auf opid des entsprechenden Top-Level Terms

    Auswahl der zur Dekompositionsregel gehörigen Basistaktik

    Soweit möglich, heuristische Bestimmung notwendiger Parameter

```
let D i proof =
    if i=0 then
      let T = conclusion proof in
       let oid = opid T in
             if oid = ‘product‘  then pairI   proof
        else if oid = ‘function‘ then lambdaI proof
        else if ...
    else let i’ = get_pos_hyp_num i proof in
      let T = select i’ (hypotheses proof) in
       let oid = opid T in
             if oid = ‘product‘  then productE i’ proof
        else if oid = ‘function‘ then ...
```

Programmierung von Standardtaktiken II

- **Dekomposition einer Gleichheit** `EqD` i
  - $i{=}0$: Anwendung einer Gleichheitsregel für die Konklusion
  - $i{\neq}0$: Dekomposition einer Gleichheit in einer Hypothese mit Hilfe der Substitutionsregel (nur Produkte / Funktionen)
  - Implementierung analog zur Taktik `D` i

    (Wahl der Basistaktik über opid des Typs der Gleichheit, Parameterbestimmung)

```
let EqD i proof =
    if i=0 then
      let (s,t,T) = dest_equality (conclusion proof) in
       let oid = opid T in
             if oid = ‘product‘  then pairEq   proof
          ⋮
```

Dekomposition einer Typzugehörigkeit `MemD` i

– Aufruf von `EqD` i nach Auffalten der Typzugehörigkeitsdefinition

- **Dekomposition des Typs einer Gleichheit** `EqTypeD` i
  - Notwendig für Teilmengen- / Quotiententypen (`MemTypeD` i analog)
- **Evaluation eines Redex** `Reduce` i
  - $i{=}0$: Anwendung einer Reduktionsregel für die Konklusion
  - $i{\neq}0$: Reduktion von Top-Level (!) Redices mit Hilfe von Substitution

Einfügen von Steuerungsparametern

- **Standardtaktiken bestimmen Regelparameter**
  - Einfach für Variablennamen und Universenlevel
  - Heuristik für Bestimmung von Termen und Abhängigkeiten

    Automatische Bestimmung ist nicht immer möglich oder ‘korrekt’
  - Benutzer muß Parameter ergänzen oder überschreiben können
- **Unterstütze explizites Einfügen von Parametern**

  Anwendung einer Modifikationsfunktion auf die Standardtaktik
  - Name x einer neuen Variablen — `New` $[x]$ `(D 0)`
  - Typ T eines Teilterms im Beweisziel — `With` $\boxed{x{:}S{\to}T}$ `(MemD 0)`

    z.B. für $H \vdash f\ t \in T$
  - Term, s, der für eine Variable einzusetzen ist — `With` $\boxed{s}$ `(D 0)`

    z.B. für $H \vdash \exists x{:}T.P(x)$
  - Level j eines Universums — `At` $\boxed{j}$ `(D 0)`
  - Abhängigkeit eines Terms C von Variable x — `Using` $\boxed{[z,C]}$ `(D 0)`

    z.B. für Substitution in $y{=}x, y{=}f(x) \vdash x{=}f(x)$

  Implementierung: Heuristik zur Parameterbestimmung befragt globale Variable, die von Modifikationsfunktion beschrieben wird

Programmierung einfacher Standardschlüsse

- **Regeln der Logik erster Stufe:** `andR`, `orR`, ..., `exL` i
  - Auffalten der Definition des logischen Operators
  - Anwendung der zugehörigen Standard-Dekompositionstaktik
  - Anwendung der Autotaktik auf das Wohlgeformtheitsziel (Folie 14)
  - Wenn erforderlich, Zurückfalten der Definition des logischen Operators
- **Erweiterte Abschlußregeln** `Hypothesis`, `Declaration`
  - Bestimme anwendbare Hypothese durch Suche und Pattern-Matching (Folie 19)
- **Beweis durch Fallanalyse** (vgl. Folie 2)
  - Allgemeine Fallunterscheidung: `Cases` $[t_1; \ldots; t_n]$
    Erzeugt neben den Einzelfällen das Vollständigkeits-Teilziel $t_1 \vee \ldots \vee t_n$
  - Analyse entscheidbarer Aussagen `Decide` P
    Erzeugt die Fälle P und $\neg P$ und beweist $P \vee \neg P$
    Fehler, wenn Beweis nicht erfolgreich ($P \vee \neg P$ gilt nicht immer)
  - Analyse Boolescher Variablen: `BoolCases` i
    Erzeugt die Fälle x und $\neg_b x$, wenn $x{:}\mathbb{B}$ in Hypothese i deklariert

Programmierung komplexerer Basistaktiken

- **Induktionsregeln** `NatInd` i, `IntInd` i, `ListInd` i
  - Erweiterung der Standard-Dekompositionsregel für Zahlen und Listen
    Erhalte Abhängigkeiten, die von der Basisregel ignoriert werden
    Vor Regelanwendung verschiebe relevante Hypothesen in Konklusion: $\vdash H_i \Rightarrow C$
  - Vollständige Induktion auf natürlichen Zahlen: `CompNatInd` i
    Beweise / instantiiere Induktionsschema als Lemma (Pattern Matching)
    $\vdash \forall P{:}\mathbb{N}\to\mathbb{P}.(\forall i{:}\mathbb{N}.(\forall j{:}\mathbb{N}.j<i \Rightarrow P(j)) \Rightarrow P(i)) \Rightarrow \forall i{:}\mathbb{N}.P(i)$
- **Chaining (Verkettung von Argumenten)**
  - Instantiierung quantifizierter Aussagen (Matching): `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`
  - Versuche alle einfachen Beweisketten automatisch zu finden durch Anwendung aller bekannten (terminierenden) Methoden
  - Sehr hilfreich, aber im Mißerfolgsfall etwas undurchsichtig

Komposition existierender Taktiken

- **Tacticals unterstützen 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 (orR2 THEN Auto)`
  - Wiederholte Anwendung von Taktiken `Repeat` t

    z.B. $H \vdash \forall x_1, x_2, ..x_n{:}T.P$ `BY Repeat allR`
- **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

Funktionskomposition mit Analyse der (Zwischen)ergebnisse

```
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
`OnSomeHyp` 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 und -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 p)) 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 der Typentheorie hängt die Gleichwertigkeit vom Typ ab
- **Explizite Substitution**
  - Ersetze Term t_1 durch t_2 in einer 'Klausel' c — `Subst` $t_1 = t_2 \in T$ c
  - Verwende Gleichheit in Klausel c_1 als Substitution — `HypSubst` c_1 c_2

    Oft gekoppelt mit Einfügen eines Gleichheitslemmas
- **Verwaltung benutzerdefinierter Abstraktionen**
  - Auflösen einer Definition — `Unfold` *name* c
  - Zurückfalten der rechten Seite einer Definition — `Fold` *name* c
- **Auswertung von Teiltermen**
  - Finde und reduziere Redizes in einer Klausel c — `Reduce` c

Simplifier und Rewrite-Pakete

- **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')
- **Nuprls Rewrite Paket** Manual §8.9
  - 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