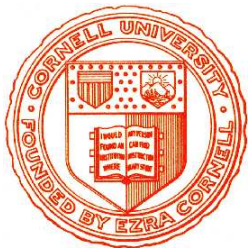
# Automatisierte Logik und Programmierung

## Einheit 4




## Computergestützte Beweisführung


1. Aufgaben interaktiver Beweissysteme
2. Realisierungsvarianten
   - Wissensverwaltung
   - Benutzerinteraktion
   - Beweisautomatisierung
3. Arbeiten mit Nuprl

# Computergestützte Beweiser – wozu?

- **Formale Beweisführung ist sehr schematisch**
  - Vorgabe eines Beweisziels
  - Iteratives Anwenden von Beweisregeln und Erzeugen von Teilzielen
  - Kreativität liegt in Wahl geeigneter Regeln
- **Formale Beweisführung ist nichts für Menschen**
  - Zu viele Details: mühsame Schreibarbeit, fehleranfällig in Ausführung
  - Als Training für gründliche Vorgehensweise geeignet, aber ungeeignet für praktisches Arbeiten mit großen Beweisen
- **Anwendung formaler Regeln ist leicht zu programmieren**
  - Pattern Matching vergleicht Beweisziel mit Hauptziel der Regel
  - Teilziele können durch Instantiieren der Parameter generiert werden
- **Ein erster Schritt zur Beweisautomatisierung**
  - Benutzer konstruieren Beweise interaktiv durch Angabe der Regeln
  - Ermöglicht Experimente mit verschiedenen Beweisansätzen
  - Ermöglicht Programmierung von Beweisstrategien

# Die einfachste Variante - Kommandozeile

- **System verwaltet Beweisbaum nur intern**
  - Benutzer gibt Theorem als initiales Beweisziel
  - Benutzer gibt Namen der Regel und ggf Parameter ein
  - System zeigt berechnete Teilziele an
  - Benutzer muß mit Kommandos durch Beweisbaum navigieren
- **Schnell zu implementieren**
  - Parser für Terme, Formeln, Regelnamen und Navigationskommandos
  - Datenstruktur für Beweisbäume
  - Regeln des Systems als Operationen auf Beweisbäumen
- **Nur für kleine Experimente geeignet**
  - Fokus liegt auf Programm zur Ausführung formaler Inferenzen
  - Term- und Formelsprache oft zu einfach (z.B $\texttt{and}(A, B)$ statt $A \wedge B$)
  - Beweis wird nach Fertigstellung “vergessen”
  - Interaktion mit Benutzer sehr primitiv

# Theorembeweisen ist mehr als nur Inferenzen

## Es geht um Computerassistenz bei der Beweisführung

- **Beweisassistenten müssen Wissen verwalten können**
  - "Echte Beweisführung" ist niemals isoliert, sondern in einem Kontext
    - Mathematik: Algebra, Analysis, Logik, Kategorientheorie, ...
    - Programmierung: Verifikation, Synthese, Optimierung, Security, ...
  - Kontext bestimmt Begriffswelt, Erkenntnisse, Methoden, ...
  - Beweisführung stützt sich massiv auf bereits bekanntes Wissen
- **Beweisassistenten benötigen "visuelles" Benutzerinterface**
  - Unterstützung für interaktive Bearbeitung von Statements/Theoremen, Beweisen, Definitionen, Theorien, informalen Kommentaren, ...
  - Präsentation formaler Konstrukte muß verständlich bleiben
- **Beweisassistenten sollten Automatisierung unterstützen**
  - Elementare Inferenzregeln sind ungeeignet für große Beweise
    Beweise werden unlesbar und sind extrem mühsam zu führen
  - Automatisierung kann komplexe Folgen von Beweischritten in Gruppen zusammenfassen und selbständig nach "trivialen" Beweisen suchen

# Kernbestandteile eines Beweisassistenzsystems

- **Bibliothek**
  - Verwaltung von des formalen Wissens
  - Zugriff, Suche, Modifikation, globale Theorieoperationen
- **Benutzerinterface**
  - Kommunikation mit der Bibliothek
  - Visuelle Bearbeitung von Termen, Beweisen, Definitionen, ...
- **Inferenzmaschine**
  - Anwendung von Inferenzregeln auf Beweisziele
  - Automatisierung der Beweisführung
- **Optionale Komponenten**
  - Extraktion von Programmen aus Beweisen
  - Evaluator: Ausführung von Programmen
  - Konvertierung in andere Formate (PDF, HTML, Fremdbeweiser, ...)

**Es gibt unterschiedliche Arten der Gestaltung**

# Beweisassistenten: Aufgaben der Bibliothek

- **Bibliothek muß Wissensverarbeitung unterstützen**
  - Erzeugung formaler Definitionen, Sätze, Beweise, Methoden, Texte
  - Strukturierung formalen Wissens in Theorien und Sub-Theorien
  - Umbenennen, Verschieben, Verlinken, Entfernen von Wissen
  - Verwendung formalen Wissens in Beweisen, Methoden und Texten
  - Durchsuchen gespeicherten Wissens, Suche nach "relevantem" Wissen
- **Wissensverarbeitung ist mehr als "Sammeln"**
  - Neue Erkenntnisse kommen hinzu, andere werden entfernt
  - Spezifische Beweise und Beweismethoden ändern sich
  - Konsistenz des gespeicherten Wissens muß sichergestellt sein (Versions- / Abhängigkeitskontrolle, Vermeidung zirkulärer Beweise)
  - Vorhandensein gespeicherten Wissens benötigt eine Rechtfertigung z.B. durch Verweise auf Inferenzregeln oder externe "Autoritäten"
- **Unterstützung für dezentrales Arbeiten**
  - Export, Import, Mischen und Prüfung von (Teil-)Theorien
  - Einschränkung von Schreib- und Zugriffsrechten

# Bibliothek: Textorientierte Gestaltung

- **Einfachste und häufigste Form der Wissensverwaltung**
  (Isabelle, Coq, MetaPRL, Agda, ACL2,...)
  - Objekte werden konventionell in Textdatei gelagert
  - Strukturierung durch Schlüsselworte (Definition, Theorem, Proof, ...)
  - Dateien werden wie Programmcode sequentiell gelesen und compiliert
  - Spezielle Suchwerkzeuge operieren auf geladenen internen Daten
- **Vorteile**
  + Konventionell editierbar, vertraute Textsuche mit grep oder Emacs
  + Informationen leicht austauschbar, geringer Platzbedarf
- **Nachteile**
  - Konsistenz nur durch strikt lineare Verarbeitung gesichert
  - Nur ein Benutzer, nur ein aktuell sichtbares Objekt
  - Keine Zugriffskontrolle: jeder kann alles überschreiben / löschen
  - Lokale Bibliotheken verschiedener Nutzer nicht leicht zu integrieren
  - Systemupdates können existierende Nutzerbibliotheken ungültig machen

# Bibliothek: Realisierung in Isabelle und Coq

## Library ist Textdatei, wird angezeigt in passendem Emacs mode

```
eory Num
ports Datatype BNF_LFP
gin

bsection {* The @{text num3} type *3

tatype num = One | Bit0 num | Bit1 num

xt {* Increment function for type @{typ num3} *3

imrec inc :: "num ⇒ num" where
"inc One = Bit0 One" |
"inc (Bit0 x) = Bit1 x" |
"inc (Bit1 x) = Bit0 (inc x)"

xt {* Converting between type @{typ num3} and type @{typ nat3} *3

imrec nat_of_num :: "num ⇒ nat" where
"nat_of_num One = Suc 0" |
"nat_of_num (Bit0 x) = nat_of_num x + nat_of_num x" |
"nat_of_num (Bit1 x) = Suc (nat_of_num x + nat_of_num x)"

imrec num_of_nat :: "nat ⇒ num" where
"num_of_nat 0 = One" |
"num_of_nat (Suc n) = (if 0 < n then inc (num_of_nat n) else One)"

mma nat_of_num_pos: "0 < nat_of_num x"
by (induct x) simp_all

mma nat_of_num_neq_0: " nat_of_num x ≠ 0"
by (induct x) simp_all

mma nat_of_num_inc: "nat_of_num (inc x) = Suc (nat_of_num x)"
by (induct x) simp_all

mma num_of_nat_double:
"0 < n ⟹ num_of_nat (n + n) = Bit0 (num_of_nat n)"
by (induct n) simp_all
```

```
(** Interpretation of booleans as propositions *)

Definition Is_true (b:bool) :=
  match b with
    | true => True
    | false => False
  end.

(*******************)
(** * Decidability *)
(*******************)

Lemma bool_dec : forall b1 b2 : bool, {b1 = b2} + {b1 <> b2}.
Proof.
  decide equality.
Defined.

(*********************)
(** * Discrimination *)
(*********************)

Lemma diff_true_false : true <> false.
Proof.
  discriminate.
Qed.
Hint Resolve diff_true_false : bool v62.

Lemma diff_false_true : false <> true.
Proof.
  discriminate.
Qed.
Hint Resolve diff_false_true : bool v62.
Hint Extern 1 (false <> true) => exact diff_false_true.
```

# Bibliothek: Realisierung in ACL2

## Library ist einfache Textdatei, keine optische Unterstützung

```
(in-package "ACL2")

(include-book "inequalities")

; theorems about natp, posp

(defthm natp-fc-1
  (implies (natp x)
           (<= 0 x))
  :rule-classes :forward-chaining)

(defthm natp-fc-2
  (implies (natp x)
           (integerp x))
  :rule-classes :forward-chaining)

(defthm posp-fc-1
  (implies (posp x)
           (< 0 x))
  :rule-classes :forward-chaining)

(defthm posp-fc-2
  (implies (posp x)
           (integerp x))
  :rule-classes :forward-chaining)
```

```
(defthm natp-rw
  (implies (and (integerp x)
                (<= 0 x))
           (natp x)))

(defthm posp-rw
  (implies (and (integerp x)
                (< 0 x))
           (posp x)))

(defthm |(natp a)  <=>  (posp a+1)|
  (implies (integerp a)
           (equal (posp (+ 1 a))
                  (natp a))))

(encapsulate
 ()
 (local
  (defthm posp-natp-l1
    (implies (posp (+ -1 x))
             (natp (+ -1 (+ -1 x))))))

 (defthm posp-natp
   (implies (posp (+ -1 x))
            (natp (+ -2 x)))
   :hints (("goal" :use posp-natp-l1))))
```

# Bibliothek als Abstrakte Datenbank

- **Globales Verständnis formalen Wissens** (Nuprl)
  - "*Mathematisches Wissen ist universell und nicht für jeden anders*"
  - Wissen wird nicht lokal sondern uniform verwaltet
  - Zugriffe über Datenbankmanagementsystem (vgl. Buchungssysteme)
    Datenbanksprache codiert in Menüs und Buttons
  - DBMS verwaltet Namensgebung, Strukturierung und Zugriffsrechte
- **Nachteile**
  - Komplexeres System, kein einfaches Editieren von Text möglich
  - Synchronisation, Import/Export von Theorien nur über das DBMS
- **Vorteile**
  + Multiuser-Kooperationen möglich, viele Objekte simultan sichtbar
  + Selektive Sichten und Kombinationen von Theorien möglich
  + Zugriffskontrolle und Transaktionskonzept mit vielfachem Undo/Redo
    sichert Konsistenz, ermöglicht Versionskontrolle,
    erhöhte Sicherheit gegenüber Mißbrauch, Irrtümern, Absturz

# Bibliothek: Realisierung in Nuprl

```
- TERM: Navigator

MkTHY*  OpenThy*  CloseThy*  ExportThy*  ChkThy*  ChkAllThys*  ChkOpenThy*
CheckMinTHY*  MinTHY*  EphTHY*  ExTHY*

Mill*  ObidCollector*  NameSearch*  PathStack*  RaiseTopLoops*
PrintObjTerm*  PrintObj*  MkThyDocObj*  ProofHelp*  FixRefEnvs*
CpObj*  reNameObj*  EditProperty*  SaveObj*  RmLink*  MkLink*  RmGroup*

MkTHM*  MkML*  AddDef*  AddRecDef*  AddRecMod*  AddDefDisp*  AbReduce*
Act*  DeAct*  MkThyDir*  RmThyObj*  MvThyObj*

↑↑↑↑  ↑↑↑  ↓↓↓↓  ↓↓↓  <>  ><

Navigator: [num_thy_1; standard; theories]

List Scroll : Total 159,  Point 5,  Visible : 10
 ---------------------------------------------
    CODE  TTF  RE_init_num_thy_1
    COM   TTF  num_thy_1_begin
    COM   TTF  num_thy_1_summary
    COM   TTF  num_thy_1_intro
    DISP  TTF  divides_df
 -> ABS   TTF  divides
    STM   TTF  divides_wf
    STM   TTF  comb_for_divides_wf
    STM   TTF  zero_divs_only_zero
    STM   TTF  one_divs_any
 ---------------------------------------------
```

– Navigieren durch Bibliothek mit Maus und Pfeiltasten

– Öffnen von Objekten mit passendem Editor (Mausclick oder Pfeil)

– Ausführen von Bibliothekskommandos mit Buttons

# Aufgaben des Benutzerinterface

## Visuelle Unterstützung zur Bearbeitung von Wissen

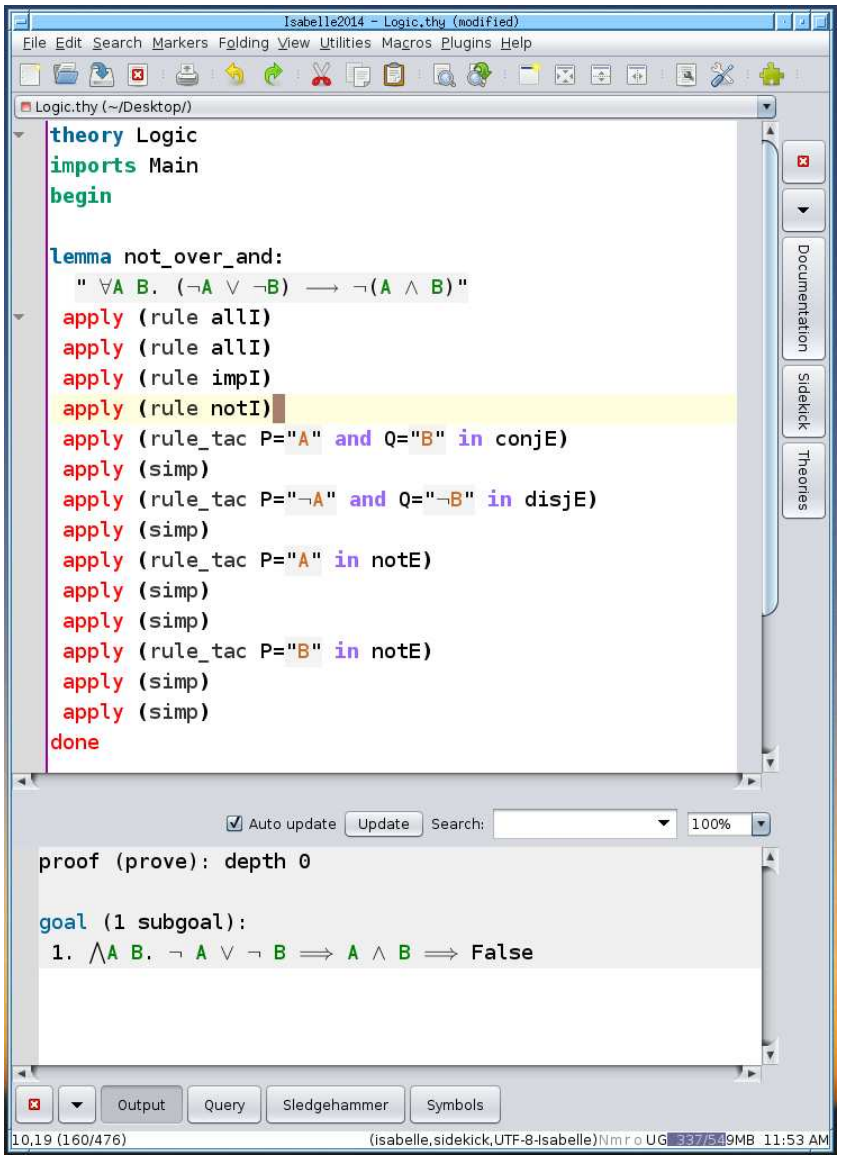
- **Benutzer muß Theorien interaktiv entwickeln können**
  - Erzeugung von Definitionen, Statements, Führen von Beweisen, Strukturierung in Theorien, informale Dokumentation, ...
  - System muß (Zwischen-)Ergebnisse der Bearbeitung anzeigen
  - System sollte Unterstützung für typische Arbeitstechniken anbieten
    z.B. Rückblättern, gleichzeitige Sicht auf viele Daten, alternative Beweisansätze, ...
- **Layoutfragen sind wichtig**
  - Verständnis formaler Texte ist of abhängig von gewählter Notation
    Computernahe Formalisierung erschwert Entwurf "guter" Theorien
  - Formales System sollte gängige mathematische Notation unterstützen

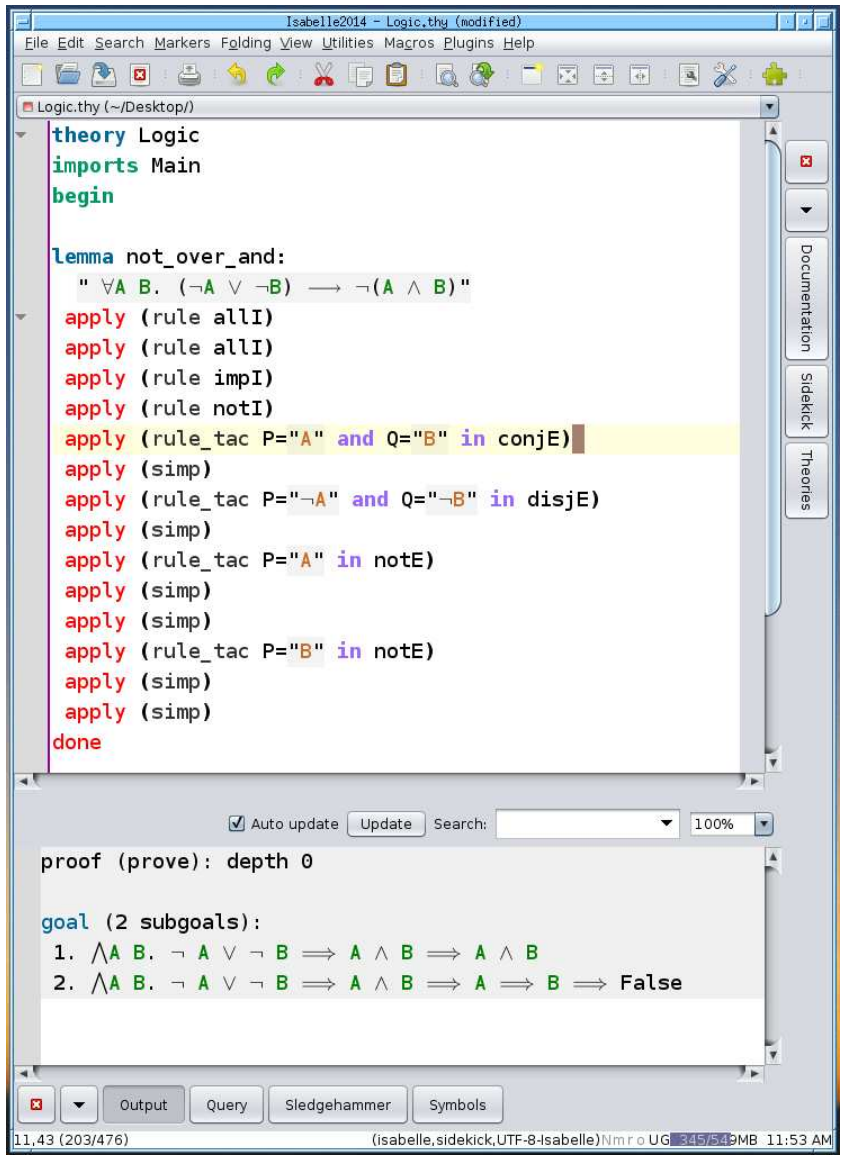
# Benutzerinterface: Skriptorientierte Gestaltung

- **Einfache Erweiterung der Kommandozeile**
  (Isabelle, Coq, MetaPRL, ACL2, SpecWare)
  - Definitionen, Sätze, Beweisskripte, etc. entstehen durch Eingabe von Schlüsselworten, Formeln und Kommandos in Textdatei
  - Interface (e.g. ProofGeneral, CoqIDE, jEdit) zwischen Textdatei und Kommandozeile des "eigentlichen" Systems unterstützt serielle (ggf. auch parallele) Verarbeitung von Theorien und Beweisskripten
  - Ausgaben des Systems werden separaten Fenstern angezeigt
- **Vorteile**
  + Für Anfänger leichter zu erlernen, Einsatz vertrauter Editoren möglich
  + Geringer Implementierungsaufwand
- **Nachteile**
  - Textbasiertes Vorgehen, nur aktuelles Beweisziel sichtbar
  - Formale Notation eingeschränkt durch Fähigkeiten des Parsers

# Benutzerinterface: Realisierung in Isabelle

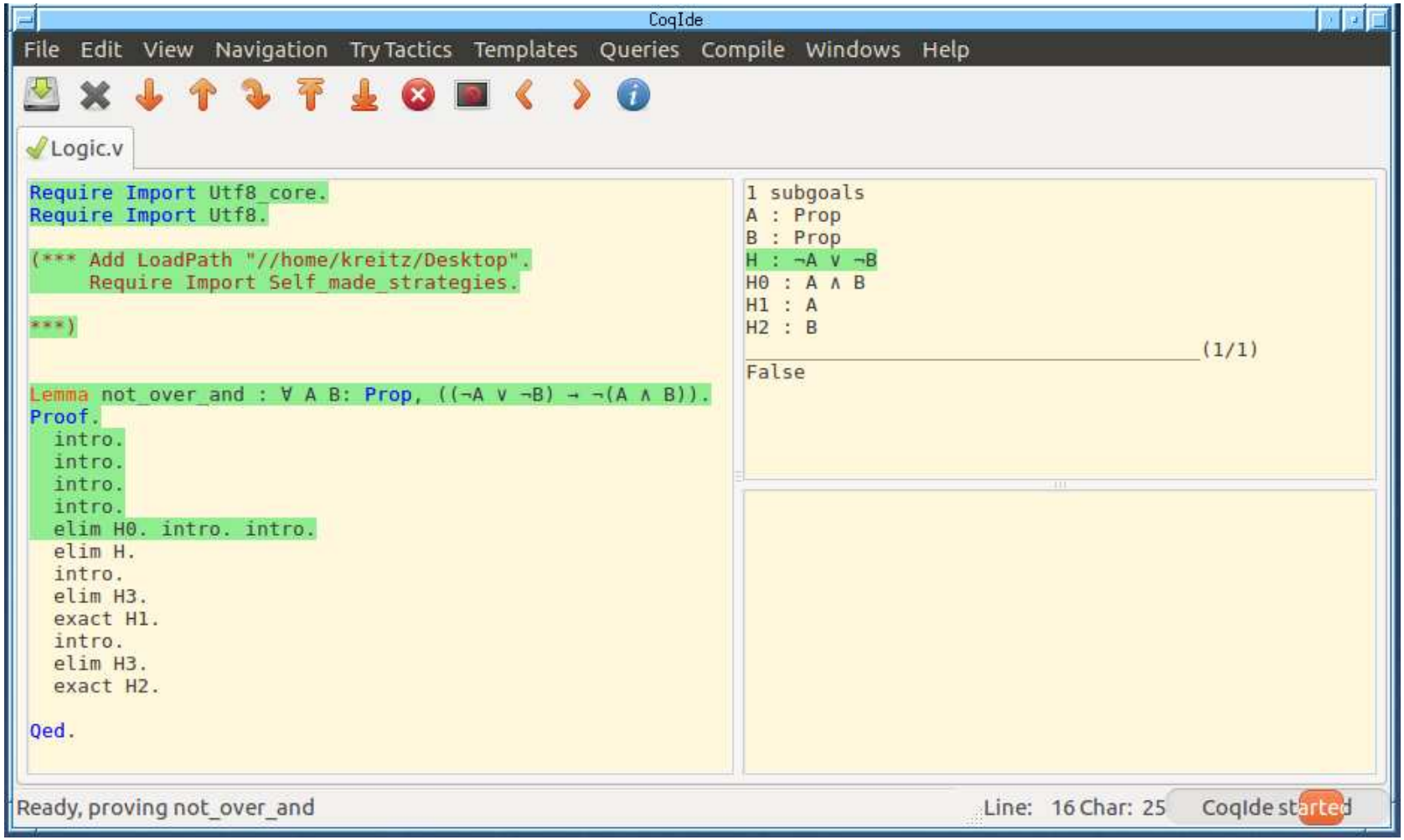
## Interface zeigt Beweisknoten entsprechend der Cursorposition



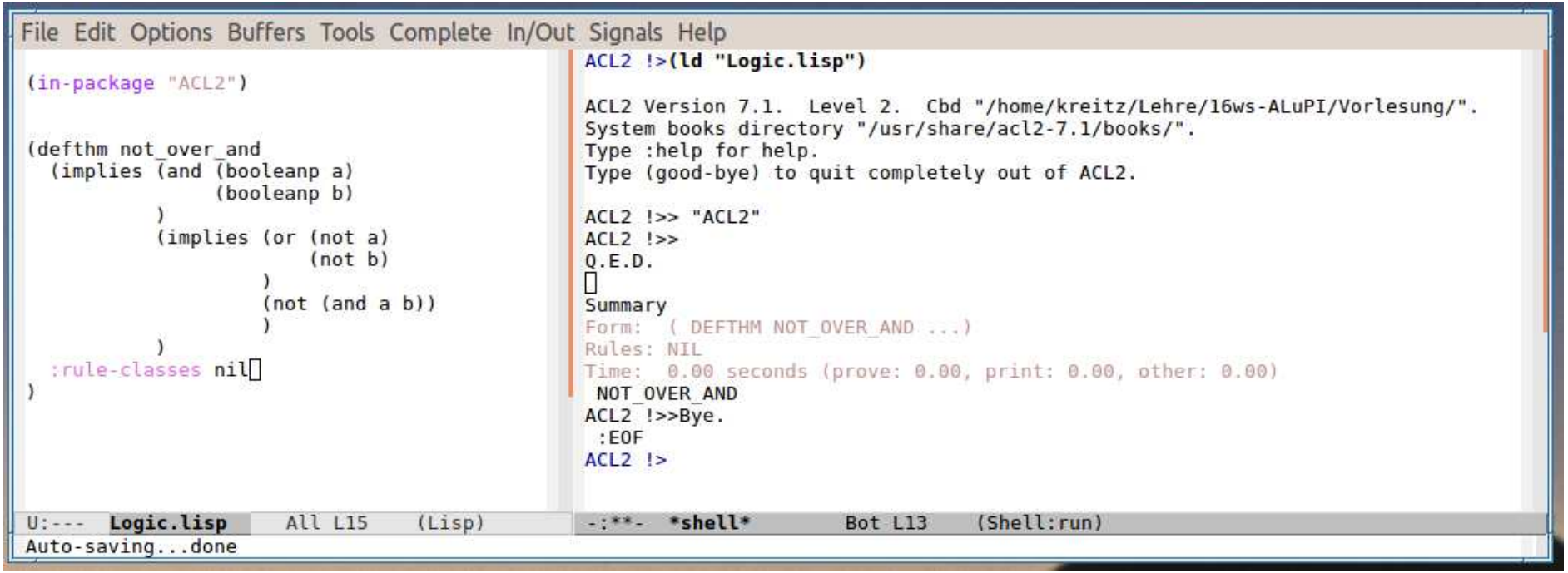
# Benutzerinterface: Realisierung in Coq

**Interface reicht Skript-Kommandos and Coq-Interpreter und zeigt Beweisknoten entsprechend der Cursorposition**



# Benutzerinterface: Realisierung in ACL2

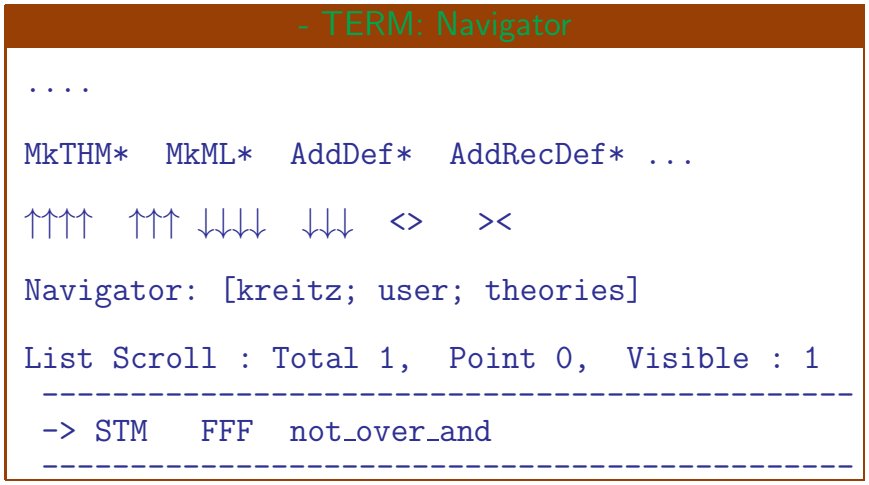
**Shell Kommando lädt Theoriedatei in ACL2-System**
**System versucht automatischen Beweis und zeigt Ergebnis oder ausführliche Fehlermeldung**

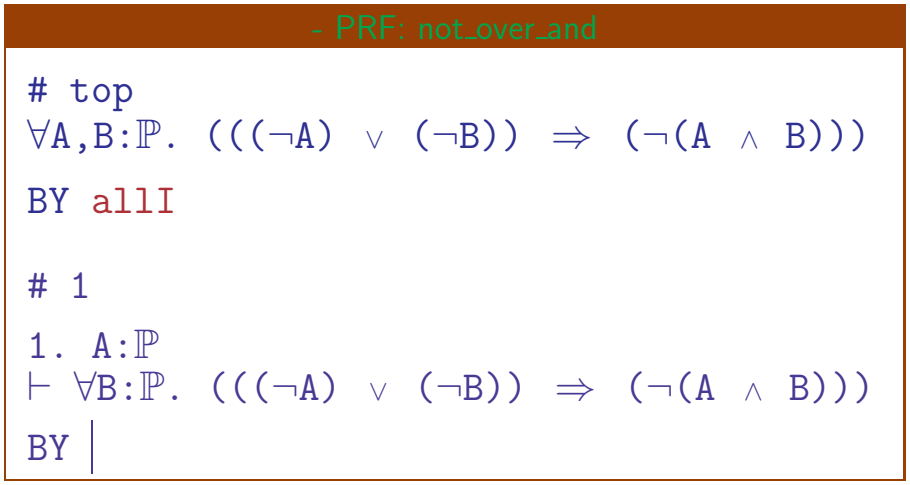


# Benutzerinterface: Visuelle Interaktion

- **Spezialeditoren für Objekte der Wissensbank** (Nuprl)
  - Benutzer navigiert visuell durch Bibliothek, Beweisbaum, ...
  - Notation für formale Objekte unabhängig von interner Darstellung
  - Strukturediteren unterstützen Eingabe und Manipulation verschiedenartiger Objekte
- **Nachteile**
  - − Steilere Lernkurve für Anfänger (es gibt mehr zu erlernen)
  - − Aufwendigere Implementierung
- **Vorteile**
  - + Beliebig paralleler Zugriff erlaubt gleichzeitig sichtbare Information
  - + Parallele Bearbeitung mehrerer Beweisziele
  - + Parallele Bearbeitung mehrerer Beweisversuche für dasselbe Ziel
  - + Flexible Syntax ohne Notwendigkeit für aufwendige Parser
  - + Trennung zwischen Notation und Bedeutung ermöglicht Anpassung der Präsentation an Papierform und nachträgliche(!) Veränderungen

# Benutzerinterface: Realisierung in Nuprl




- **Erzeugung von Theoremen**
  - Benuzter generiert Theorem-Objekt in Wissensbank (MkTHM)
  - Benutzer öffnet Beweiseditor (Maus-Click) und gibt Beweisziel ein
  - Beweisziel wird unmittelbar in Bibliothek gesichert
- **Beweisführung**
  - Benutzer gibt Beweisregel (oder Taktik) im Regelslot an
  - Inferenzmaschine kann synchron oder asynchron aufgerufen werden
  - Teilziele werden unmittelbar in Bibliothek gespeichert und angezeigt
  - Sichtbarer Anteil des Beweisbaums nur abhängig von Fenstergröße Benutzer navigiert ggf. visuell in Unterziele

# Aufgaben der Inferenzmaschine

- **Verarbeitung von Regeln und Beweisstrategien**
  - Prüfen der Anwendbarkeit und Instantiierung von Parametern
  - Berechnen (und Abspeichern) der zu erzeugenden Unterziele
- **Unterstützung von Beweisautomatisierung**
  - Ergänzung des Regelsystems durch Laden neuer Beweisprogramme
    Programmierte Anwendung von Inferenzregeln, Simplifikation, Entscheidungsprozeduren, automatische Beweissuche, ...
    Spezialstrategien für bestimmte Anwendungsbereiche
  - Erweiterung existierender Beweisprogramme
    Integration von Modulen, Erweiterung von Parameterlisten, ...
  - Optimierung komplexer Beweisprogramme
    Compilierung, Caching, ...

# Mögliche Arten von Beweisautomatisierung

- **Abgeleitete Inferenzregeln** (Nuprl, Coq, Isabelle)
  - Anwendung bewiesener Theoreme der Form $(\forall \hat{x}) A[\hat{x}] \Rightarrow B[\hat{x}]$ als Regel Matching von $B[\hat{x}]$ gegen Beweisziel, Instantiiertes $A[\hat{x}]$ wird Unterziel
  - Mechanismus implementierbar als leicht anzuwendende `Lemma`-Regel
  - Ermöglicht massive Erweiterung der Menge "primitiver" Regeln
- **Beweistaktiken** (Nuprl, Coq)
  - Programmierte Steuerung der Anwendung elementarer Inferenzregeln z.B. Zusammensetzung von Regeln durch Verkettung, Iteration, etc.
  - Komplexe Taktiken dürfen für die Analyse auf alle Teile des Systems zugreifen um passende Regeln auszuwählen
  - Benutzerdefinierbare Erweiterung des Inferenzsystems ohne das Risiko inkorrekte Beweise zu generieren
  - Einfach zu implementierender Mechanismus

# Mögliche Arten von Beweisautomatisierung II

- **Komplexe Beweisuchprogramme für spezifische Aufgaben**
  (Nuprl, Coq, Isabelle, ACL2, Agda, ...)
  - Entscheidungsprozeduren, Simplifikatoren, Auto-Prozeduren
  - Oft sehr mächtig und vielseitig verwendbar
  - Implementierung durch Systementwickler als komplexe Taktik oder als (meist unverifizierte) Systemprozedur
- **Benutzerdefinierbare Erweiterung von Systemprozeduren**
  (Nuprl, Coq, Isabelle)
  - Theoreme oder Taktiken können in "hooks" eingefügt werden z.B. neue beweisene Gleichheiten für Simplifikation oder neue Schlußformen für Auto
  - Naiver Einsatz führt schnell zu Endlossschleifen in der Prozedur
- **Steuerungsparameter / Hints** ( Coq, Isabelle, ACL2)
  - Benutzer kann Suchtiefe, Abarbeitungsreihenfolge, etc. beeinflussen
  - Braucht Verständnis der konkreten Arbeitsweise von Simplifikator/Auto

# Das interaktive Beweissystem Nuprl

## "Proof Refinement Language, Version $\nu$"

- **Anfänge liegen mehr als 30 Jahre zurück** (1984)
  - Nuprl 1 (Symbolics): Beweissystem für intuitionistische Typentheorie
  - Nuprl 2: Unix Version
  - Nuprl 3: Strategische Beweisführung (komplexe Taktiken) (1987)
  - Nuprl 4: Logisches Framework (Kalkül in Bibliothek enthalten) (1992)
  - Nuprl 5: Systemkomponenten als kommunizierende Prozesse (1999)
- **Anwendungen in Mathematik und Programmierung**
  - Girard's Paradox, Higman's Lemma (1987)
  - Verifikation & Optimierung von Software
    - Logikschaltungen, SCI cache coherency protocol (1993)
    - Ensemble Kommunikationssystem (100000 Zeilen ML Code) (1998)
    - Paxos und Consensus Protokolle (2015)
  - Cubical Type Theory (2016)

# Arbeiten mit Nuprl

- **Basiskomponenten des Systems sind separate Prozesse**
  - Datenbank: Managementsystem für alle Datenobjekte
  - Bibliothek: Verwaltung des Wissens, Gateway zur Außenwelt
  - Refiner: Ausführung der Inferenzmechanismen (mehrfache Prozesse möglich)
  - Editor: Benutzerinterface (mehrfache Prozesse möglich)
- **Benutzerstart**
  - Empfohlen: Zugriff auf laufende Version über VNC
    - Datenbank, Bibliothek, Refiner und Editoren sind bereits gestartet
    - Benutzer greift auf laufenden Editorprozess zu oder startet ihn neu
    - System zeigt Navigator und Kommandoloop
  - Virtuelle Maschine (Standalone):
    - System mit Extras der Vorlesung ist vorinstalliert und gestartet
  - Eigene Installation (Standalone):
    - Download der Binaries und Bibliothek als komprimiertes Archiv
    - Derzeit nicht zu empfehlen, da nur veraltete Versionen verfügbar

# Arbeiten mit Nuprl II

- **Erzeuge Objekte im Navigator**
  - Theorien zur Strukturierung
  - Definitionen: Abstraktionen und separate Displayform
  - Theoreme (für Beweise),
  - Code Objekte (für Taktiken), ...

```
- TERM: Navigator

CpObj*  reNameObj*  SaveObj*  RmLink*  MkLink*
MkTHM*  MkML*  AddDef*  AddRecDef* AddDefDisp*
Act*  DeAct*  MkTHY* MkThyDir*  RmThyObj* MvThyObj*
↑↑↑↑  ↑↑↑  ↓↓↓↓  ↓↓↓  <>  ><
Navigator: [num_thy_1; standard; theories]
List Scroll : Total 159,  Point 4,  Visible : 5
 ------------------------------------------
    CODE  TTF  RE_init_num_thy_1
    COM   TTF  num_thy_1_begin
    DISP  TTF  divides_df
 -> ABS   TTF  divides
    STM   TTF  divides_wf
 ------------------------------------------
```

- **Öffnen des Objektes startet geeigneten Editor**
  - Termeditor: Strukturelles Editieren von Termen in Präsentationsform
  - Beweiseditor: Beweisführung und Navigation durch Beweisbäume
  - weitere Objekteditoren für spezifische Objekte
- **Graphische Interaktion bewußt einfach gestaltet**
  - Textterminal-Version erlaubt Fernzugriff bei geringer Bandbreite
  - Für Standalone wären moderne GUIs möglich (aufwendig)

# Dienstleistungen des Navigators

- **Visuelle Navigation durch Bibliothek**
  - Keyboard- oder Maus-gesteuertes Durchlaufen
  - Patterngesteuerte Namenssuche
  - Springen zu gespeicherten Positionen
- **Ausführung von Bibliothekskommandos**
  - Vorbereitete "Buttons" für die wichtigsten Operationen
    - Erzeugung von Objekten, Theorien, Definitionen, Modulen
    - Löschen, Kopieren, Verschieben, Umbenennen, Drucken, ...
    - Import, Export, Drucken und Dokumentation von Theorien
  - Aufruf der Operationen öffnet Kommandomenü
- **Undo und Redo für jede Operation**
- **Anpassbar**
  - Buttons und Erscheinungsbild durch Bibliotheksobjekte definiert

# Eigenschaften des Termeditors

- **Mathematische Notation erlaubt keine Parser**
  - Zu reichhaltig (nicht kontextfrei) und nicht einheitlich geregelt
  - Notation ist keine gute Repräsentationform für logische Konzepte
- **Nuprl trennt Notation von Struktur**
  - Logische Struktur leichter zu verarbeiten
  - Separate Darstellungsform sorgt für verständliche Notation
- **Termeditor ist Strukturediteor**
  - Editiere logische Struktur von Termen bei gleichzeitiger Präsentation der Darstellungsform auf dem Bildschirm
  - Erzeugung des Termbaums durch Eintrag in Slots der Darstellungsform
  - Benutzer kann Layout von Termen separat modifizieren
  - Kenntnis der genauen Syntax nicht erforderlich
  - Umdenken erforderlich: keine lineare Eingabe von Text

**Benutzer kann mit verständlicher Notation arbeiten**

# Arbeiten mit Nuprls Termeditor

**Aktiv, wenn Cursor in einem Termslot ist**

- **Eingabe neuer Terme**
  - Eingabe des Termnamens erzeugt
    Template mit Subtermslots
  - z.B. Eingabe von `all` ↵ erzeugt
    $\forall$`[var]:[type]. [prop]`
    Typisierte Prädikatenlogik mit geringfügig anderer Syntax
  - Unbekannte Namen werden als Variablen interpretiert

| | |
|---|---|
| `not` | $\neg A$ |
| `and` | $A \wedge B$ |
| `or` | $A \vee B$ |
| `implies` | $A \Rightarrow B$ |
| `all` | $\forall x{:}T\,.\,A$ |
| `exists` | $\exists x{:}T\,.\,A$ |

- **Modifikation von Termen**
  - Navigation im Syntaxbaum mit Emacs-ähnlichen Kommandos
  - Variablennamen und Zahlen können wie Text verändert werden
  - Teilterme müssen gelöscht und neu erzeugt oder kopiert werden

# Eigenschaften des Beweiseditors

- **Visuelle Entwicklung von Top-Down Beweisen**
  - Strukturierte Bearbeitung von Beweisbäumen
  - Navigation durch Beweisbaum mit Maus und Keyboard
  - Lokales Arbeiten im einzelnen Beweisknoten möglich
  - Kontrolliertes Interface zum Refiner (via Library Gateway)
  - Graphische Interaktion einfach gestaltet
- **Operationen auf Beweisen** (vgl. Folie 10)
  - Erzeugung von Beweiszielen mit Term-Editor
    Beweisziel wird nach Eingabe in Bibliothek gesichert
  - Synchrone oder asynchrone Ausführung von Beweisregeln
    Anzeige der erzeugten Teilziele
  - Komprimierung und Expansion bis zu elementaren Schritten
  - Verarbeitung von Backup-Beweisen und 'Schmierblatt'-Beweisen
  - Erzeugung von Evidenz-Termen

# Typischer Beweisknoten

```
- PRF: intsqrt
① # top 1
②
③ 1. x:ℕ
   ⊢ ∃y:ℕ. y²≤x ∧ x<(y+1)²
④ BY NatInd 1
⑤ # 1 1
   .....basecase.....
   ⊢ ∃y:ℕ. y²≤0 ∧ 0<(y+1)²
⑥ BY exR ⌜0⌝
   There is 1 hidden subgoal
⑤ # 1 2
   .....upcase.....
   1. x:ℤ
   2. 0<x
   3. ∃y:ℕ. y²≤x-1 ∧ x-1<(y+1)²
   ⊢ ∃y:ℕ. y²≤x ∧ x<(y+1)²
⑤ BY
```

```
- PRF: intsqrt
① # top 1 2
② .....upcase.....
③ 1. x:ℤ
   2. 0<x
   3. ∃y:ℕ. y²≤x-1 ∧ x-1<(y+1)²
   ⊢ ∃y:ℕ. y²≤x ∧ x<(y+1)²
④ BY
```

① Status und Adresse im Beweisbaum

② Annotation des Beweisknotens

③ Beweisziel (Sequenz)

④ Angewandte Beweistaktik

⑤ Teilziele mit Status, Adresse, Sequenz (neue Hypothesen)

⑥ Beweise der Teilziele, sofern vorhanden

# Beweistaktiken in Nuprl

- **Benutzerprogrammierbare Beweisstrategien**
  - Planung und Suche von Beweisen
  - Strukturierung von Beweisen (Verstecken überflüssiger Details)
  - Abgeleitete Inferenzregeln für benutzerdefinierte Theorien
  - Austesten komplexer Beweis-/Syntheseverfahren in sicherer Umgebung
- **Formalisiert als Metalevel-Programme in** Classic-ML
  - Formale metasprachliche Ausdrücke steuern Regelanwendungen
  - Einfache Komposition von Regeln und Taktiken durch Tacticals

| | |
|---|---|
| $t_1$ `THEN` $t_2$: | Wende $t_2$ auf alle von $t_1$ erzeugten Teilziele an |
| $t$ `THENL` `[`$t_1; \ldots; t_n$`]`: | 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 |
| `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 |

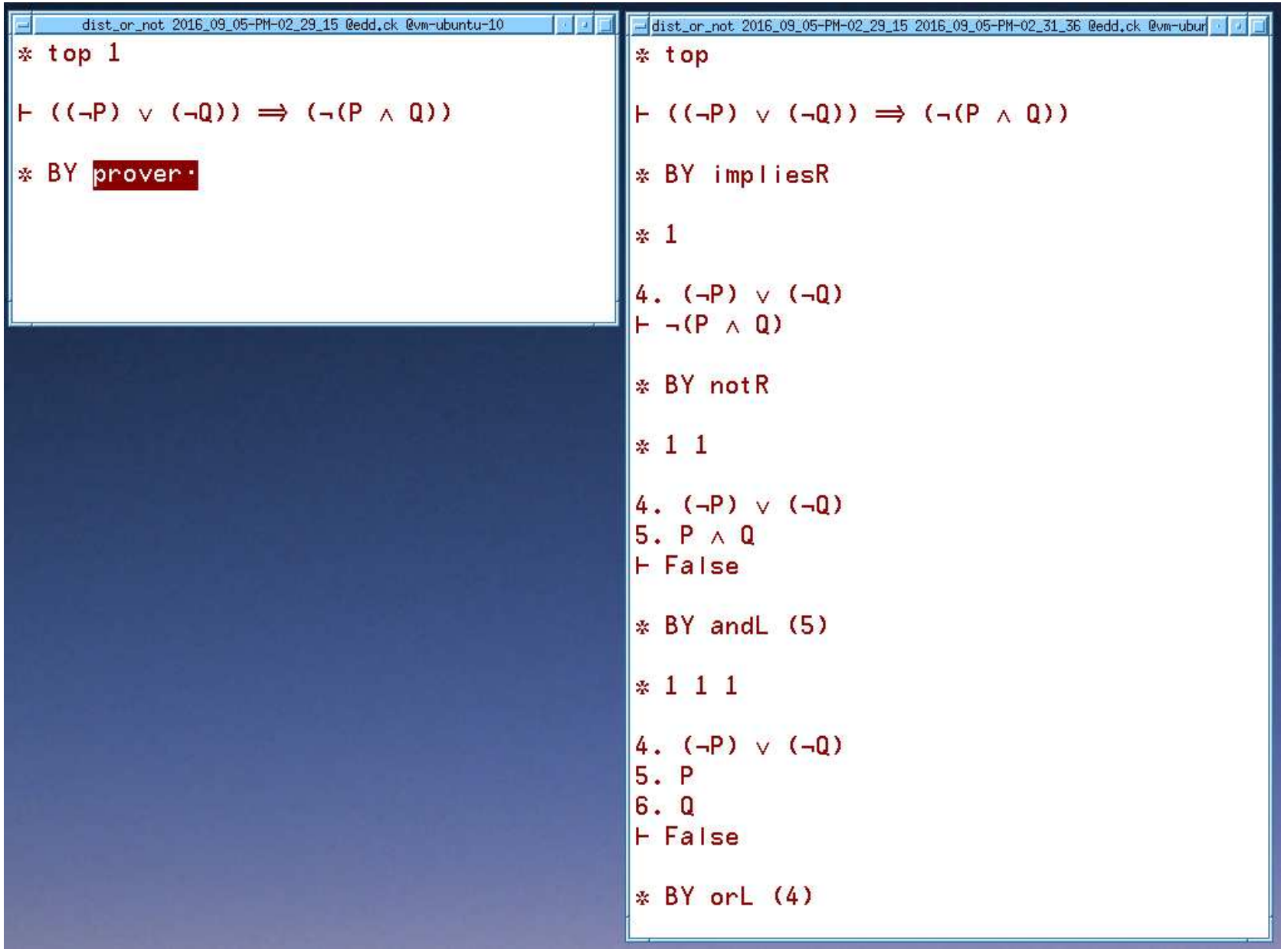
- **Taktiken sind immer korrekt**
  - Taktikbeweis wird expandiert zu Beweis mit elementaren Regeln

# Programmierung eines taktischen Beweisers

## Sortiere Regelanwendungen nach Aufwand für Beweissuche

```
let simple_prover = Repeat
                    (       Hypothesis
                    ORELSE contradiction
                    ORELSE InstantiateAll
                    ORELSE InstantiateEx
                    ORELSE conjunctionE
                    ORELSE existentialE
                    ORELSE nondangerousI
                    ORELSE disjunctionE
                    ORELSE not_chain
                    ORELSE iff_chain
                    ORELSE imp_chain
                    );;
letrec prover = simple_prover
                THEN Try (          Complete (orI1 THEN prover)
                          ORELSE (Complete (orI2 THEN prover)))
                ;;
```

# Anwendung von prover und erzeugter Beweis



# Arbeiten mit Nuprl in Potsdam

- Nuprl 5 **ist immer noch Experimentalsoftware**
  - Hohe Flexibilität führt zu steiler Lernkurve für Interface
  - Manual auf Webseite zu finden (Vorlesungsseite bzw. nuprl.org) müsste aber in einigen wichtigen Punkten überarbeitet werden
- **Installation auf eigenem Rechner zur Zeit schwierig**
  - Download Version ist einige Jahre alt und konzipiert für Ubuntu 10
  - Benötigt CMU Common Lisp oder SBCL für Linux
    · Systemanforderungen: $\geq$2GB RAM, 10GB freier Plattenplatz
- **Verfügbare virtuelle Maschinen**
  - Geschlossene Ubuntu-10 Installation: mit Extras der Vorlesung
    Version für VMPlayer (ca. 8GB) von 2012, historisch gewachsen
  - Cornell Version von 2015: aktuelle Binaries und Bibliothek
    ohne eigenes X-Window System (Zugang über lokales VNC)
    (zur Zeit noch) ohne Extras der Vorlesung
  - VNC Version auf Server der Theoriegruppe
    Cornell Version von 2015, vorgestartet, mit einigen Extras
    Anleitungen zum Aufruf werden demnächst bereitgestellt