The Development of Interactive Theorem Proving

Deduktionstreffen 2025

Christoph Kreitz

Professor (em) für theoretische Informatik

Institut für Informatik und Computational Science

Universität Potsdam

`kreitz@uni-potsdam.de`

What is ITP all about?

- **The semi-official definition: Proof Assistants**
  - Software that interacts with the user to construct formal proofs
- **... but that isn't the complete story**

  ITP is about (formally) proving real theorems
  - Theorems that are out of reach for fully automated provers
  - Proofs with complex structures and elaborate arguments
- **Proof assistants provide a "safe" reasoning environment**
  - Inference engines that guarantee correctness of proofs
  - An infrastructure for developing formal theories
  - Assistance for finding proofs and often much more

For users it's quite different from ATP

- **ITP is no pushbutton technology**
  - Users must provide guidance, know the rules, and have a proof idea
    The system just executes what it's being told
    But it can be pretty smart about that
- … **but one can accomplish much more with interaction**
  - Pushbutton technology is stuck when it doesn't succeed
  - Incomplete ITP proofs permit users to decide locally how to proceed
  - ATP logics like FOL are very small
    They cannot express numbers, induction, or data structures
  - (Most) ITP logics are very rich and cover all of mathematics

For users it's quite different from ATP

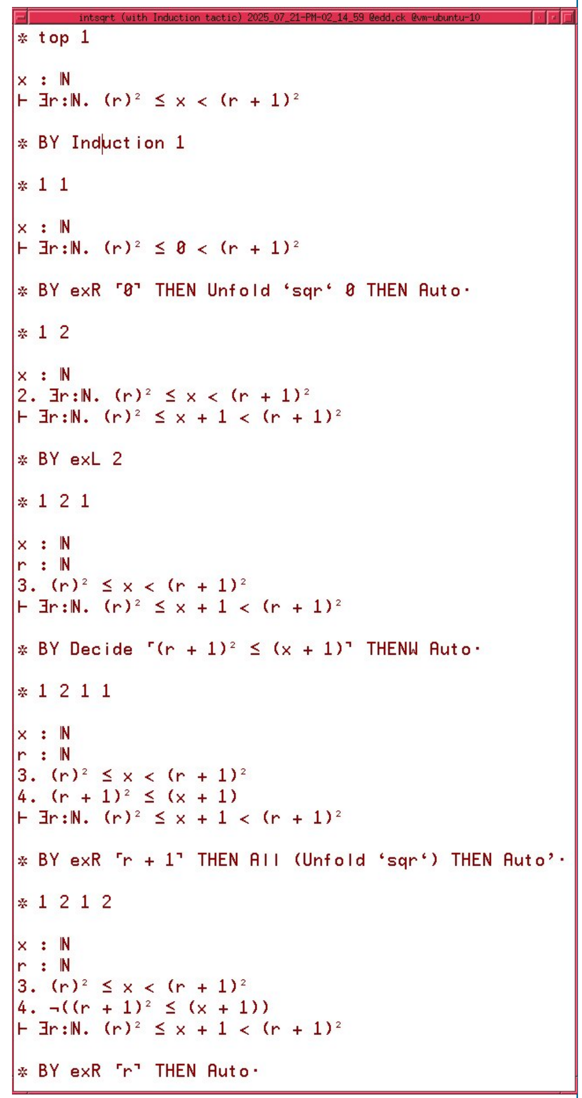
- **ITP is no pushbutton technology**
  - Users must provide guidance, know the rules, and have a proof idea
    The system just executes what it's being told
    But it can be pretty smart about that
- . . . **but one can accomplish much more with interaction**
  - Pushbutton technology is stuck when it doesn't succeed
  - Incomplete ITP proofs permit users to decide locally how to proceed
  - ATP logics like FOL are very small
    They cannot express numbers, induction, or data structures
  - (Most) ITP logics are very rich and cover all of mathematics

Interactive theorem proving is more fun

A SIMPLE EXAMPLE

Prove existence of integer square roots

- **Formalization of "integer square root"**
  - For $x \in \mathbb{N}$ it's the largest number r such that $r^2 \leq x$
  - Better for proofs: a number r with $r^2 \leq x < (r+1)^2$
- **Straightforward proof by induction**
  - For $n=0$ choose $r=0$
  - For $n>0$ assume the existence of a root r' for $n-1$
  - If $(r'+1)^2 \leq x$ choose $r=r'+1$, otherwise $r=r'$
- **Proof is beyond the capabilities of ATP**
  - It needs arithmetic and some user guidance
  - With a proof assistant it's a simple exercise



The early roots

- **Automath (1968): formalizing and checking proofs**
  - Formal proof language based on Curry Howard isomorphism
  - Proof checker based on type checker for λ-Terms
  - Influenced development of type theories (Martin-Löf TT, Nuprl TT, CoC,..)
- **LCF (1972): interactive proof development**
  - Proof rules as metalevel programs that transform sequents
  - Meta-programs (tactics) control application of rules
  - Influenced many ITP systems (Nuprl, Coq, HOL / Isabelle, Lean, ...)
- **NQTHM (1971): proof automation**
  - LISP based quantifier-free computational logic with induction
  - Rewriting based automated proofs about computation
  - Later systems have decision procedures and simplifier (ACL2, PVS,..)
- **MIZAR (1973): formal mathematical libraries**
  - Formal language and checker for real mathematical papers
  - Journal Formalized Mathematics has checked scientific articles

Many different proof assistants are in use

Mizar (1973), Nuprl (1984), Coq (1988, now called Rocq),
PVS (1992), HOL4 (1994), Agda (1999), Twelf (1999),
ACL2 (2000), Isabelle/HOL (1990/2002), Lean (2013), . . .

https://en.wikipedia.org/wiki/Proof_assistant

- **There is no such thing as the best proof assistant**
  - Different systems have different strengths
  - Designs vary significantly and are sometimes incompatible
  - Every design decision has its pros and cons
- **Success depends strongly on the users of a system**
  - It requires a successful cooperation between the two
  - Different users master different systems better than others
  - It's often a matter of personal preference
- . . . **but proof assistants have many aspects in common**

Proof assistants have accomplished a lot

- **Formal proofs of famous mathematical problems**
  - Four color theorem (Coq, 2005)
  - Kepler conjecture (HOL light/Isabelle, 2015)
  - Feit-Thompson theorem (Coq, 2013)
  - Erdös-Graham problem (Lean, 2022)
- **Complex mathematical theories**
  - Cubical Type Theory (NuPRL, 2020)
- **Improving quality of Software Systems** (Nuprl, 1998–2002)
  - Ensemble Group communication system (NYSE)
    Verified optimization improves performance by a factor of 3–10
    Verification of communication protocols detects subtle errors

ITP IS MORE THAN JUST PROOF SEARCH

- **Real proofs are never without context**
  - Theorems are about mathematical theories, programming, security, ...
  - Context determines which concepts, insights, or methods may be used
  - Proofs depend on existing knowledge
- **Interaction requires a user interface**
  - Users have to edit theorems, proofs, definitions, formal theories, etc.
  - Formal constructs should be presented in familiar notation
- **Large proofs need structure and automation**
  - Proofs should not be expressed in terms of primitive inferences
    - Inferences should be grouped into large steps
    - Trivial proof parts should be completely automated

Proof assistants have to offer appropriate support

Aspects of Interactive Theorem Provers

- **Theoretical Foundation**
  – Syntax, semantics, and proof calculus of an expressive theory
- **Knowledge Management**
  – Library of theorems, definitions, specific proof methods, and more
- **User Interface**
  – Visual support for communication with library, inference engine, and other system components
- **Inference Engine**
  – Mechanism that executes proof rules and supports automation
- **Additional components**
  – Code generation, execution, links to external systems, ...

There are different ways to realize these

Theoretical Foundations of Proof Assistants

- **Most ITPs are based on higher order theories**

  HOL, Martin-Löf Type Theory, Nuprl Type Theory, Calculus of Constructions, …
  - Higher-order logics represent concepts by logical properties
  - Type Theories express terms and structures like $\rightarrow$, $\times$, $+$, $\mathbb{N}$ directly
- **Theories can be classical or intuitionistic (constructive)**
  - Classical reasoning permits the law of excluded middle (simpler proofs)
  - Constructive theories support reasoning about programs (more accurate?)
- **Two fundamentally different ways to treat equality**
  - Intensional: only identical terms are equal (not very practical?)
  - Extensional: terms are equal if they have the same value (undecidable)
- **Many systems use a sequent-style proof calculus**
  - Reasoning is synthetic (bottom-up), analytic (top-down), or mixed
- **Some theories include a formal meta-theory**
  - This guarantees correctness of the (implemented) proof calculus

MANAGING FORMAL KNOWLEDGE

Requirements on a proof assistant's library

- **Library should support knowledge management**
  - Constructing definitions, theorems, proofs, methods, documents
  - (Re-)using formal knowledge in proofs, methods, documents
  - Browsing and searching for "relevant" knowledge
  - Grouping knowledge into theories and sub-theories
  - Linking, moving, renaming, removing formal knowledge
- **More than just collecting data**
  - Knowledge changes: insights are gaines or turn out to be irrelevant
    proofs and proof methods may change as well
  - Consistency must be guaranteed (version and dependency control)
  - Knowledge should be certified (justification for storing it)
- **Library should support decentralized development**
  - Export, import, merging, and checking theories
  - Read and write access control

Many questions remain

Library: text-oriented design

- **Easiest and most common approach**

  (Isabelle, Coq, MetaPRL, Agda, ACL2, Lean,...)
  - Objects are stored in a text file
  - Keywords (theory, definition, theorem, proof,...) provide structure
  - Data are read, compiled, and "executed" sequentially
  - Search and other library mechanisms operate on runtime data
- **Pros**
  - \+ Standard editors may be used, search via `grep` or `emacs`
  - \+ Easy exchange of data, small storage space
- **Cons**
  - – Consistency requires strict linear processing
  - – Single user access, objects can be accessed only one at a time
  - – No access control possible
  - – Merging user theories is difficult
  - – System updates may invalidate user libraries

Library design in Isabelle and Coq

Library is a textfile presented in an IDE or emacs mode

```
theory Num
imports Datatype BNF_LFP
begin

subsection {* The @{text num} type *}

datatype num = One | Bit0 num | Bit1 num

text {* Increment function for type @{typ num} *}

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

text {* Converting between type @{typ num} and type @{typ nat} *}

primrec 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)"

primrec 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)"

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

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

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

lemma 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.
```

Library is a simple textfile, no special presentation mode

```
(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))))
```

Library as abstract database

- **Global view on formal knowledge** (Nuprl)
  - "*Mathematical knowledge is universal and not user-dependent*"
  - Knowledge is maintained globally, not on a user's computer
  - Access through a database management system (c.f. online booking)
  - DBMS maintains names, structure, and access rights
- **Cons**
  - Complex design, objects cannot be edited like text
  - Synchronization, theory import/export only through the DBMS
- **Pros**
  - \+ Multi-user cooperation possible, many objects visible simultaneously
  - \+ Selective views and combinations of theories possible
  - \+ Access control and transaction concept with multiple undo/redo
    consistency is guaranteed, version control possible
    increased security against user errors or system crashes

Library design in Nuprl (shown via a tty-like interface)

```
- 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
 ----------------------------------------------
```

– Visual navigation through knowledge base (mouse/arrows)

– Opening objects starts object-specific editors

– Buttons for library commands

DESIGNING

THE

USER INTERFACE

Requirements on a proof assistant's user interface

Visual support for managing knowledge

- **Users have to develop theories interactively**
  – Editing theories, definitions, theorems, proofs, documentation, ...
  – System has to present (intermediate) results of user activities
  – System should support revisiting previous steps,
  simultaneous access to several objects, alternative proof attempts, ...
- **Layout is important**
  – Comprehensibility of formal text is a matter of notation
  Machine-level formalization makes formal theories almost unreadable
  – Interface should support conventional mathematical notation(s)

User Interface: script-oriented design

- **Simple extension of a command line prover**

  (Isabelle, Coq, MetaPRL, ACL2, SpecWare)

  – Definitions, theorems, proof scripts, etc. result from entering keywords, formulas, and commands into a text file.

  – Interface (e.g. ProofGeneral, CoqIDE, jEdit) between text file and system supports serial, sometimes parallel processing of theories and scripts

  – System ouput is shown in a separate window

- **Pros**

  + Easy to learn for beginners, familiar editors may be used

  + Easy to implement

- **Cons**

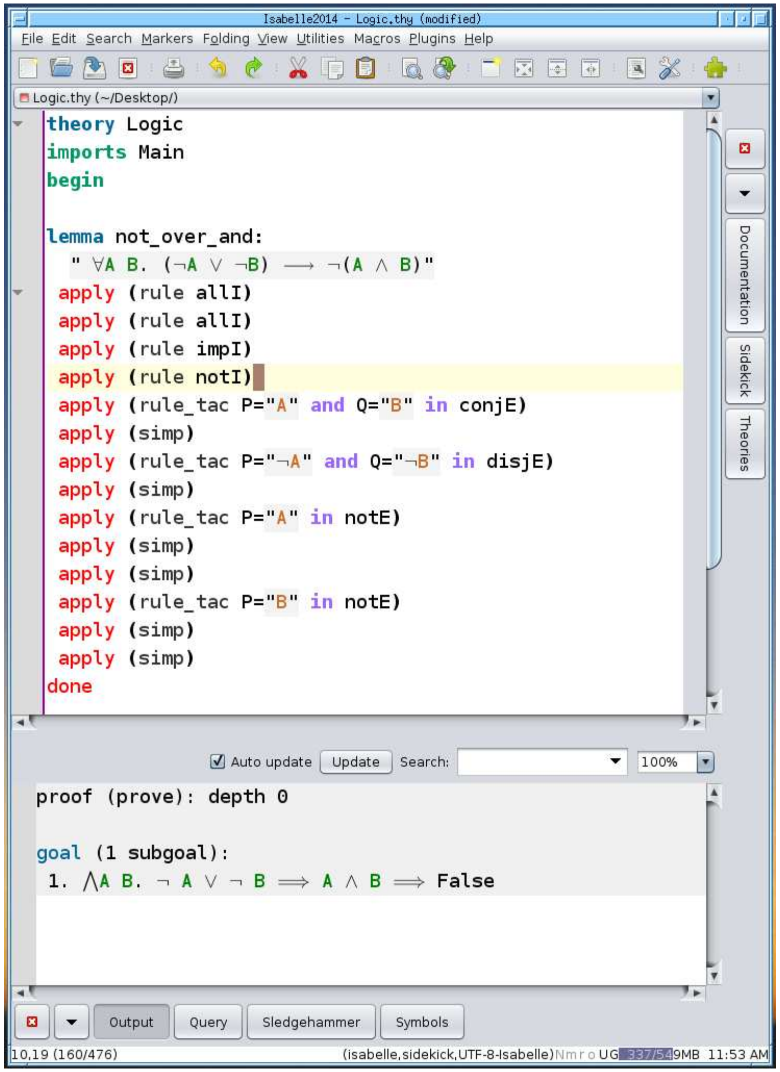
  – Text-oriented approach,

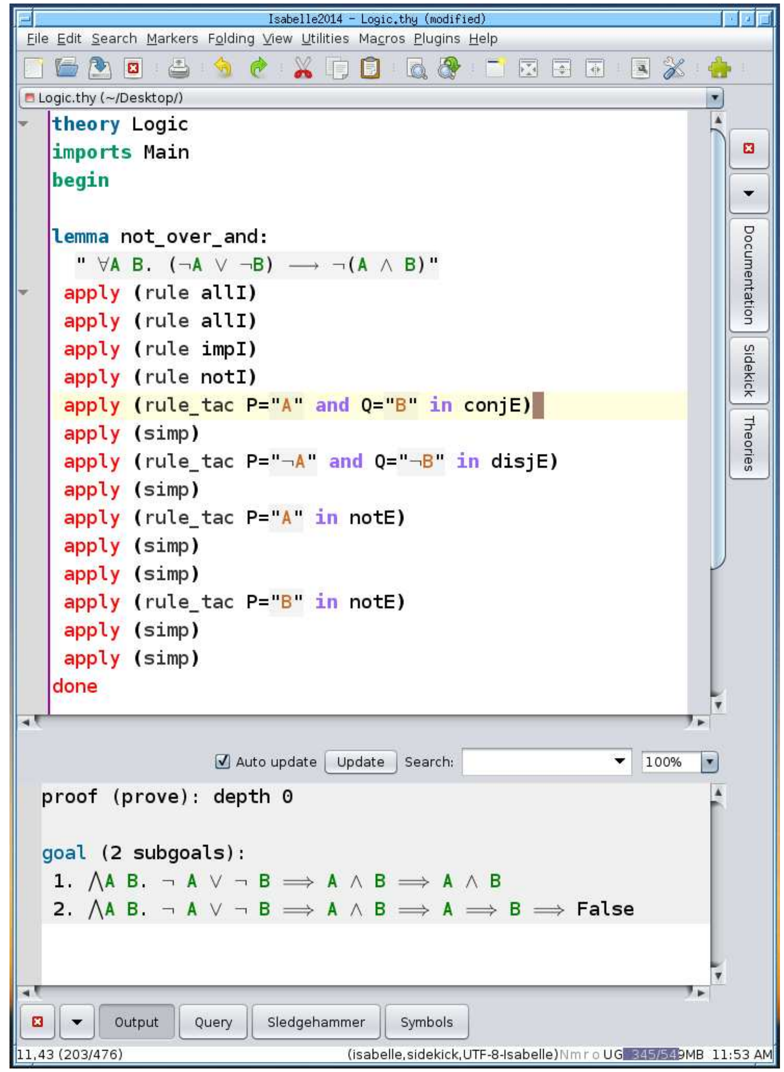
  – At any given time only one proof goal is visible

  – Flexibility of formal notation limited by capabilities of the parser

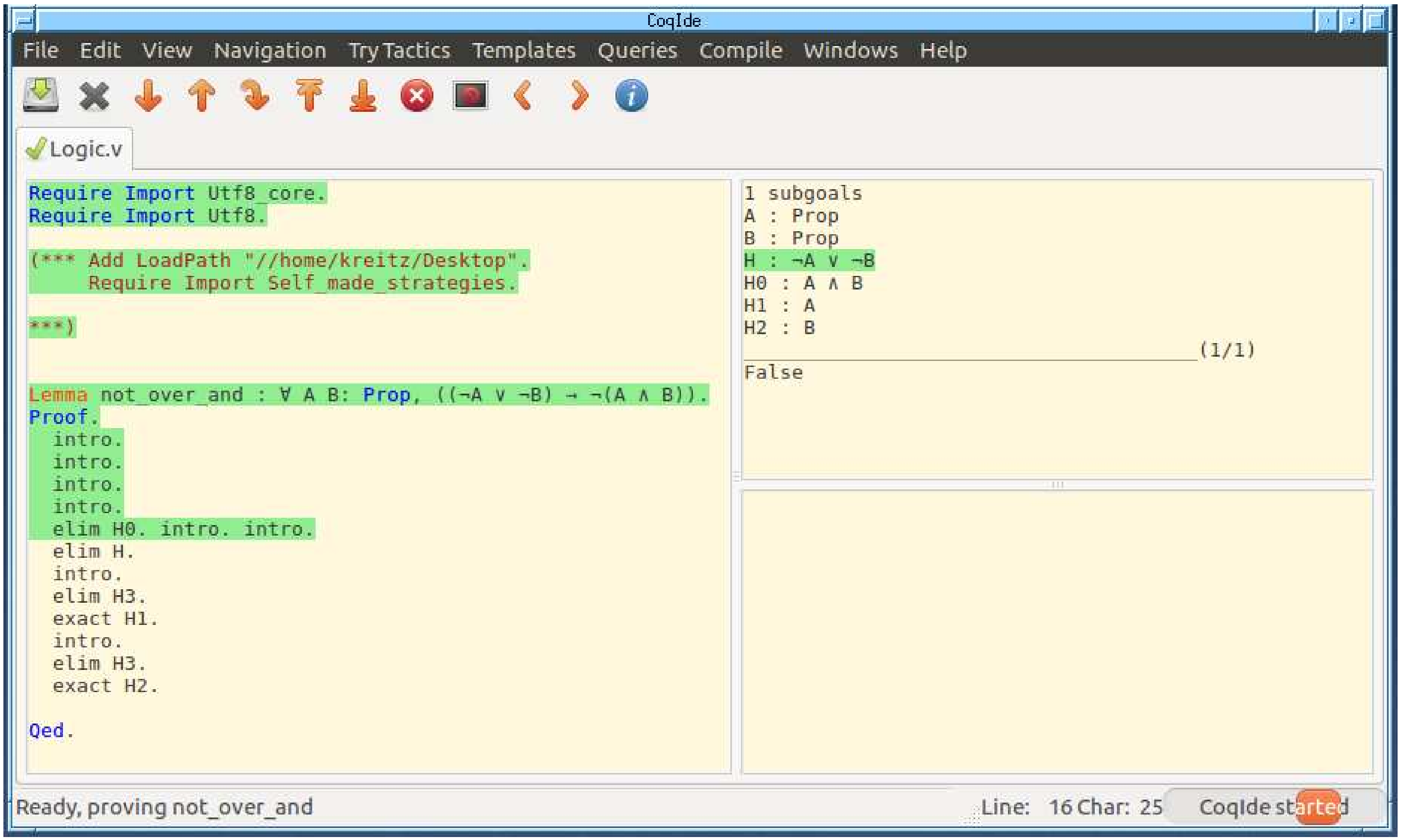
Interface design in Isabelle

Interface shows proof node corresponding to cursor position



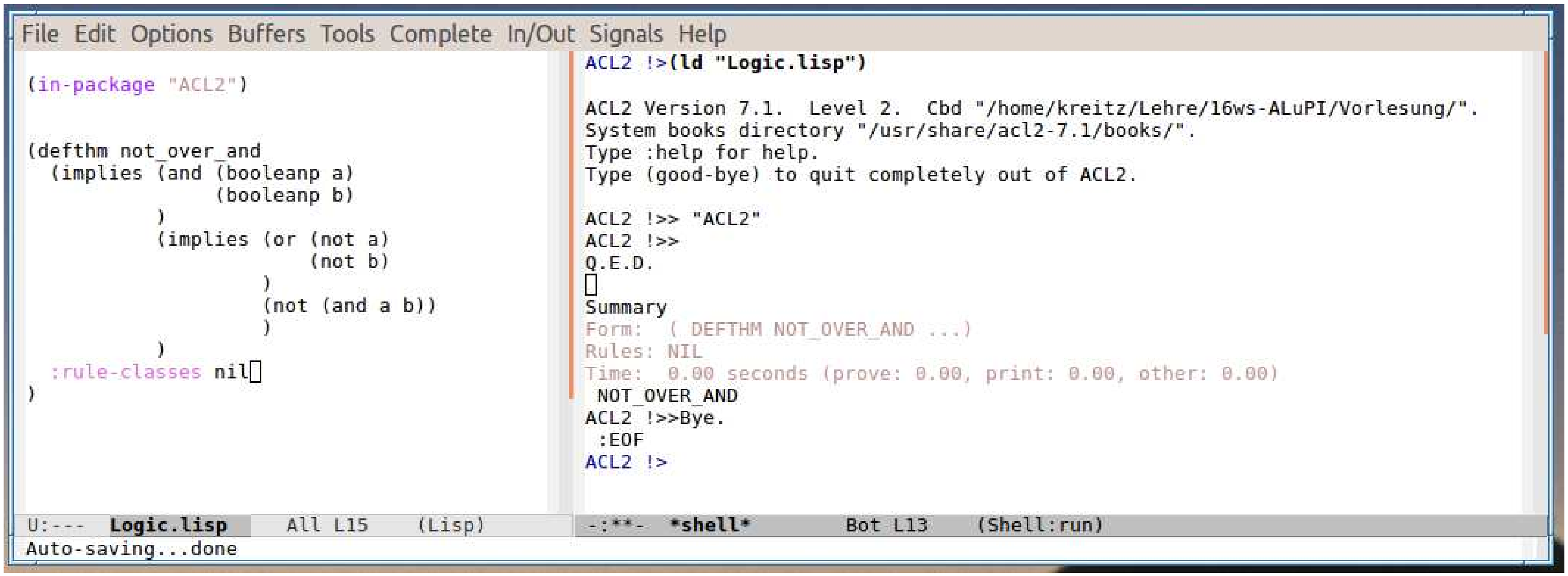


Interface sends script commands to Coq interpreter and shows proof node corresponding to cursor position



Interface design in ACL2

Shell command loads theory file into the system
ALC2 attempts automated proof and shows result
or a detailed error message



User Interface: visual interaction

- **Specialized editors for library objects** (Nuprl)
  - Users navigate through library, proof tree, term tree, etc.
  - Notation for objects is independent from internal representation
  - Structure editors support entering and manipulating objects
- **Cons**
  - Steep learning curve for beginners (much to be learned)
  - Implementation requires significant effort
- **Pros**
  - \+ Simultaneous access to a large variety of information
  - \+ Several proof goals may be processed in parallel
  - \+ Several proof attempts for the same goal may be processed in parallel
  - \+ Flexible syntax without a need for complex parsers
  - \+ Separation between internal representation and external notation permits dapting the vocabulary of formal documents

Interface design in Nuprl

- TERM: Navigator

```
....

MkTHM*  MkML*  AddDef*  AddRecDef* ...

↑↑↑↑  ↑↑↑ ↓↓↓↓  ↓↓↓  <>   ><

Navigator: [kreitz; user; theories]

List Scroll : Total 1,  Point 0,  Visible : 1
 -----------------------------------------------
 -> STM   FFF  not_over_and
 -----------------------------------------------
```

- PRF: not_over_and

```
# top
∀A,B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))
BY allI

# 1
1. A:ℙ
⊢ ∀B:ℙ. (((¬A) ∨ (¬B)) ⇒ (¬(A ∧ B)))
BY |
```

- **Creating theorems**
  - User generates named library object for theorems (button)
  - User opens editor for object (mouse click) and enters proof goal
  - Editor saves proof goal as library object
- **Creating proofs**
  - User enters name of rule or tactic into rule slot
  - Inference engine may be run synchronously or asynchronously
  - Subgoals are stored in the library and shown in proof editor
  - Visible part of proof tree depends only on window size

Components of Nuprl's user interface

- **Navigator**
  – Visual navigation through library and execution of library commands
- **Term editor**
  – Structur editing of terms within the presentation form
- **Proof editor**
  – Proof construction/modification and navigation through proof tree
- **Object specific editors**
  – Editing meta-programs (tactics), presentation forms, comments, ...
- **Command interface**
  – Interpretation of meta-programs and commands
- **Designed as independent process**
  – Several interfaces may access the same library object simultaneously

Graphical interface is purposefully very simple

– Standard text terminal version permits low bandwidth remote access

– Implementing GUIs based on current standards is not a trivial task

FEATURES OF INFERENCE ENGINES

Requirements on an ITP's inference engine

- **Processing inference rules**
  - Proof Checking: testing correctness of (complete) formal proofs
  - Proof Editing: supporting the development of a formal proof
    apply rules to proof goal, and generate/show subgoals
  - Only difference is the form of interaction with users
  - Easy to implement: encode meta-level concepts (proof, rule,...)
    of the theoretical foundation as meta-programs

- **Supporting partial automation of the proof process**
  - Interal extensions to the proof calculus through meta-programs (safe)
  - Built-in proof procedures for specific tasks (verification required)
  - Interaction with external proof systems (trusted or with validation)

Automating the proof process

- **Derived inference rules** (Nuprl, Coq, Lean, Isabelle)
  – Turn theorems of the form $\forall x{:}T.A[x] \Rightarrow B[x]$ into formal rules
  – Implementation via simple pattern matching and instantiation
  – Safe (conservative) extension of the theoretical foundation
- **Tactics** (Nuprl, Coq, Lean)
  – Meta-programs control the application of primitive inference rules
  from combining rules via combinators (e.g. t_1 `THEN` t_2, `Repeat` t, ...)
  to elaborate programs that analyze proof goals and plan proofs
  – Easy if ITP provides a meta-programming language
  – Safe user-definable extension of the proof calculus
  – Proof assistants usually provide many predefined tactics
- **Reflection** (Coq, Lean, Isabelle, Nuprl)
  – Proof procedures bypass primitive inferences
  but have been verified within the proof assistant

Automating the proof process II

- **Built-in proof procedures** (Nuprl, Coq, Lean, Isabelle, ACL2, Agda,...)
  - Procedures that are difficult or too slow to implement as tactics and too hard to be formally verified by reflection
  - Decision procedures for small sub-theories
  - Simplifier, automatic (brute-force) proof search,
- **User-defined extensions of system procedures** (Nuprl, Coq, Lean, Isabelle)
  - Adding theorems and tactics via hooks (e.g. equalities for simplifiers)
  - Risky: may cause the system procedure to loop
- **Control parameters / Hints** (Coq, Isabelle, ACL2)
  - User may change depth and order of proof search
  - Requires understanding the implemented system procedure
- **Calls to external proof systems** (Nuprl, Isabelle, Coq)
  - Logic interface as bridge for syntactical and semantical differences
  - External prover may be run in trusted (unchecked) mode or as proof planner for a tactic that validates the proof

Additional components and features

- **Code generation from constructive proofs**
  (Nuprl, Isabelle, Coq, Lean)
  – Generated code is correct by construction
  – Very difficult for classical theories
- **Code evaluation** (Nuprl, Coq, Lean)
  – Helpful for runtime analysis and verified code optimization
- **Multiple inference engines and interfaces** (Nuprl)
  – Support for distributed and cooperative proof construction
- **Formal document creation** (Nuprl, Coq, Lean)
  – Text documents with integrated library objects and proofs
  – Document changes as objects or proofs are modified

⋮

Conclusion

- **Much has been accomplished**
  ... but so far accomplishing significant results is still tedious
- **Where should we go from here?**
  – More generic ATP and higher speed doesn't really help in practice
  – Focus should be on intelligent automation to make work easier
- **Self-improving proof database** (Nuprl 6†)
  – Successful inferences generated during a proof are stored permanently
  Proof fragments may be improved (e.g. remove unneeded assumptions)
  – Fragment database and proof caching support proof reuse
- **The future may lie in learning proofs**
  – In 7/25 Harmonic AI Inc raised $100M to learn from proofs (Lean)
  – This would require ten thousands of inferences as training data
  Without a proof database these must be generated and stored separately

There is potential for significant improvements