Document-oriented Prover Interaction with Isabelle/PIDE

Makarius Wenzel
Univ. Paris-Sud, Laboratoire LRI

December 2013

Project **Paral-ITP**
ANR-11-INSE-001

Abstract

LCF-style proof assistants like Coq, HOL, and Isabelle have been traditionally tied to a sequential READ-EVAL-PRINT loop, with linear refinement of proof states via proof scripts. This limits both the user and the system to a single spot of interest. Already 10-15 years ago, prover front-ends like Proof General (with its many clones such as CoqIDE) have perfected this command-line interaction, but left fundamental questions open. Is interactive theorem proving a necessarily synchronous and sequential process? Is step-by-step command-line execution inherent to the approach? Or are these merely accidental limitations of historic implementations?

The PIDE (Prover IDE) approach to interactive theorem proving puts a conceptual Document-Model at the center of any proof development activity. It is the formal text that the user develops with the help of the system (including all background libraries). The editor front-end and prover back-end are smoothly integrated, in order to provide a metaphor of continuous proof checking of the whole formalization project (not just a single file). As the user continues editing text, the system performs formal checking in the background (usually in parallel on multiple cores), and produces output in the form of rich markup over the sources, with hints,

suggestions etc. This may involve arbitrarily complex proof tools, such as ATPs and SMTs via Isabelle/Sledgehammer.

The combination of asynchronous editing by the user and parallel checking by the prover poses some challenges to the overall architecture, with many technical side-conditions. To cope with this, Isabelle/PIDE is implemented as a hybrid of Isabelle/ML and Isabelle/Scala. This enables the pure logical environment to reach out into the JVM world, where many interesting frameworks for text editors, IDEs, web services etc. already exist. Scala allows to continue the manner and style of ML on the JVM, with strongly-typed higher-order functional programming and pure values. This helps to achieve good performance and reliability in a highly concurrent environment.

The main example application of the PIDE framework is Isabelle/jEdit, which has first become available for production use with Isabelle2011-1 (October 2011). The underlying concepts and implementations have been refined significantly in the past 2 years, such that Isabelle/jEdit is now the default user interface of Isabelle2013-2 (December 2013). Recent improvements revisit the old READ-EVAL-PRINT model within the new document-oriented environment, in order to integrate long-running print tasks efficiently. Applications of such document query operations range from traditional proof state output (which may consume substantial time in interactive development) to automated provers and dis-provers that report on existing proof document content (e.g. Sledgehammer, Nitpick, Quickcheck in Isabelle/HOL).

So more and more of the parallel hardware resources are employed to assist the user in developing formal proofs, within a front-end that presents itself like well-known IDEs for programming languages. Thus we hope to address more users and support more advanced applications of our vintage prover technology.

History

The LCF Prover Family

LCF

Edinburgh LCF (R. Milner & M. Gordon 1979)
Cambridge LCF (G. Huet & L. Paulson 1985)

HOL (HOL4, HOL-Light, HOL Zero, ProofPower)

Coq

Coc (T. Coquand & G. Huet 1985/1988)
⋮
Coq 8.4pl2 (H. Herbelin 2013, coordinator)

Isabelle

Isabelle/Pure (L. Paulson 1986/1989)
Isabelle/HOL (T. Nipkow 1992)
Isabelle/Isar (M. Wenzel 1999)
⋮
Isabelle2013-2 (M. Wenzel 2013, coordinator)

TTY interaction (≈ 1979)

```
Welcome to Isabelle/HOL (Isabelle2013: February 2013)
> theory A imports Main begin
theory A
> lemma "x = x";
proof (prove): step 0

goal (1 subgoal):
 1. x = x
>
```

```
Welcome to Coq 8.4pl2 (September 2013)

Coq < Lemma test: forall (A: Type) (x: A), x = x .
1 subgoal

  ============================
   forall (A : Type) (x : A), x = x

test <
```

Classic REPL architecture (from LISP)

READ: internalize input (parsing)
EVAL: run command (toplevel state update + optional messages)
PRINT: externalize output (pretty printing)
LOOP: emit prompt + flush output; continue until terminated

Notes:

- prompt incurs full synchronization between input/output (tight loop with full round-trip: slow)
- errors during READ-EVAL-PRINT may loose synchronization
- interrupts often undefined: might be treated like error or not

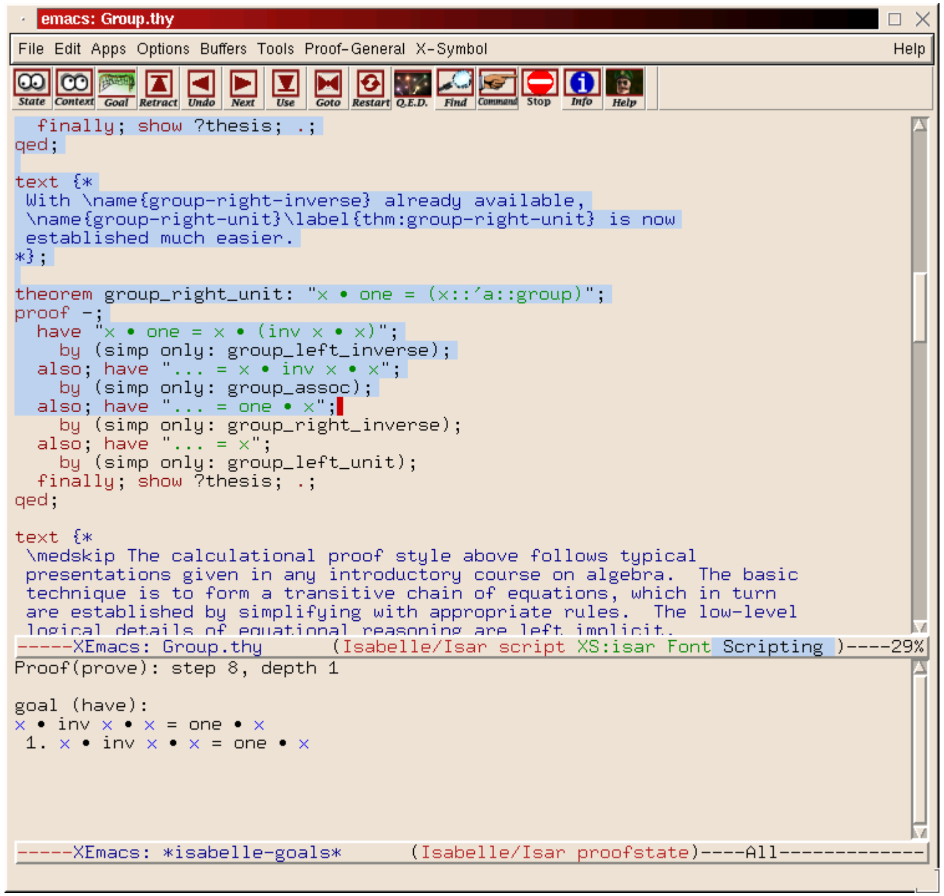
Proof General ($\approx$ 1999)

Approach:

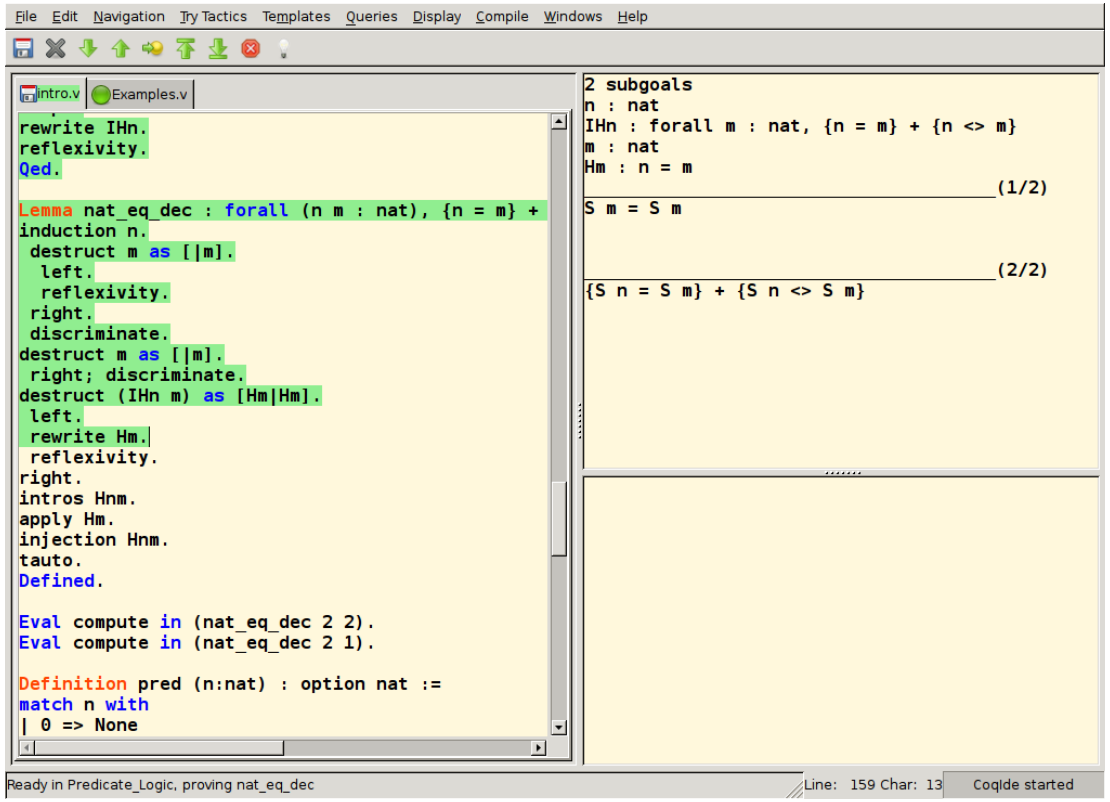
- Prover TTY loop with prompt and undo
- Editor with locked region
- User controls frontier between checked vs. unchecked text
  - move one backwards
  - move all backwards
  - move one forwards
  - move to point
  - move all forwards
  - refresh output
  - restart prover
  - interrupt prover

Example: Kopitiam (Eclipse + Coq)

Example: Isabelle Proof General



Example: CoqIDE



The “Proof General” standard

Implementations:

- Proof General / Emacs
- CoqIDE: based on OCaml/Gtk
- Matita: based on OCaml/Gtk
- ProofWeb: based on HTML text field in Firefox
- PG/Eclipse: based on huge IDE platform
- I3P for Isabelle: based on large IDE platform (Netbeans)
- Kopitiam for Coq: based on huge IDE platform (Eclipse)

Limitations:

- sequential proof scripting
- synchronous interaction
- single focus

Documented-oriented
Prover Interaction

PIDE — Prover IDE ($\approx$ 2009)

General aims:

- renovate and reform interactive theorem proving for new generations of users
- catch up with technological changes: multicore hardware and non-sequentialism
- document-oriented user interaction
- mixed-platform tool integration

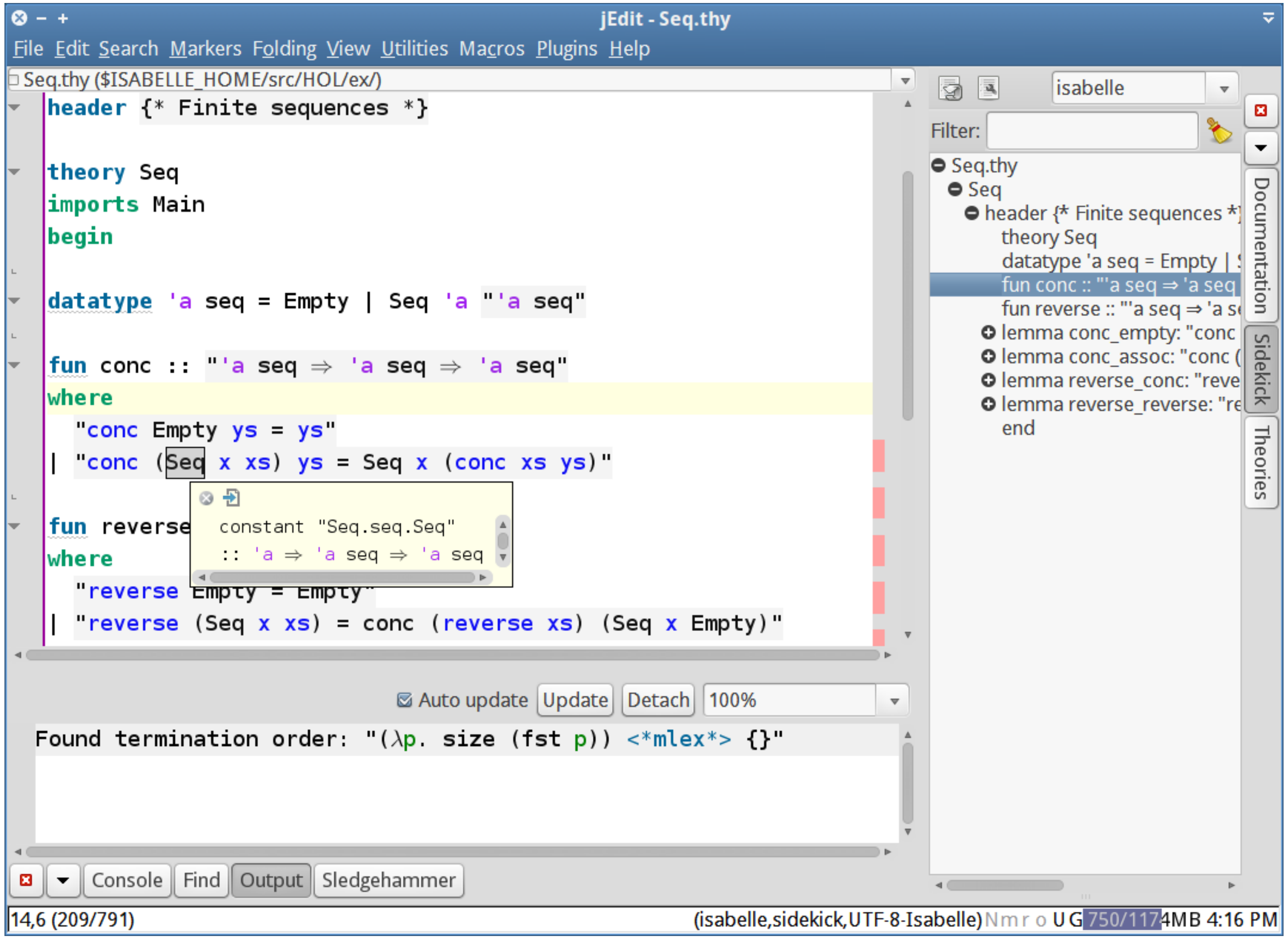
Approach:

- Prover supports document model natively
- Editor continously sends source edits and receives markup reports
- User constructs document content, assisted by GUI rendering of formal markup

Technical side-conditions:

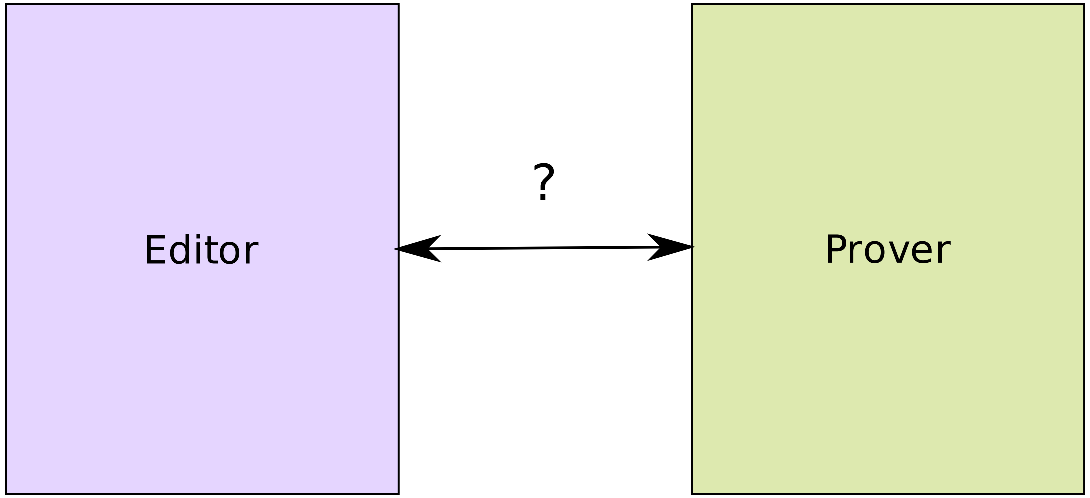
- routine support for Linux, Windows, Mac OS X
- integrated application: download and run
- no "installation"
- no "packaging"
- no "`./configure; make; make install`"

Example: Isabelle/jEdit Prover IDE

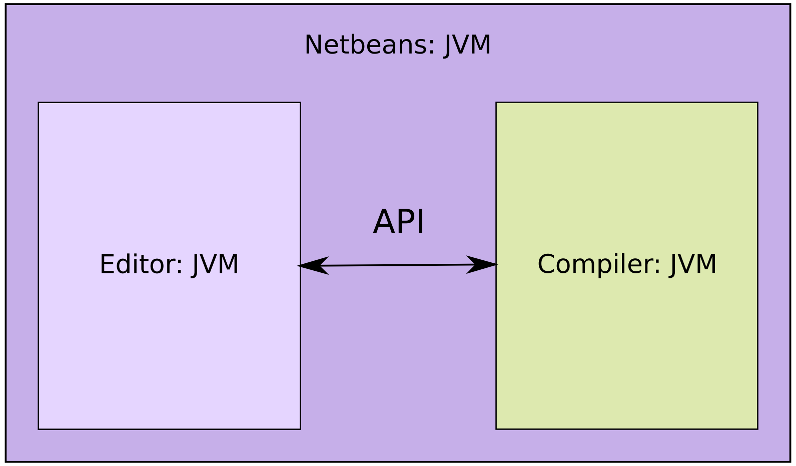


PIDE architecture

The connectivity problem



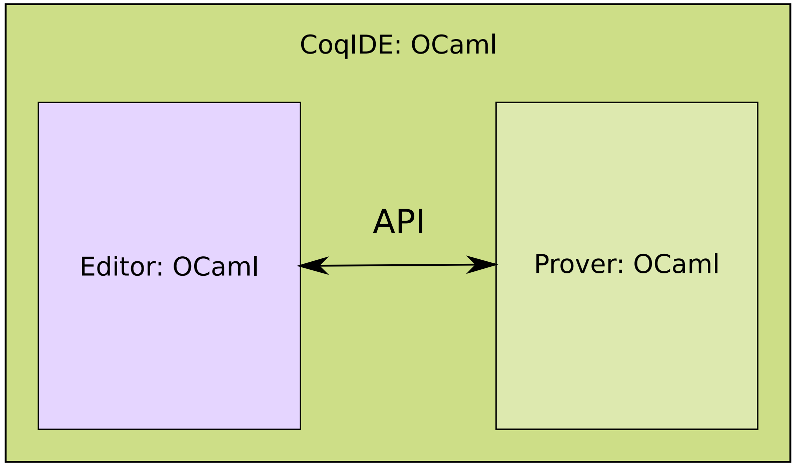
Example: Java IDE



Characteristics:

+ Conceptually simple — no rocket science.
+ It works well — mainstream technology.
−− Provers are not implemented in Java!
− Even with Scala, the JVM is not ideal for hardcore FM.

Example: CoqIDE



Characteristics:

+ Conceptually simple — no rocket science.

+– It works . . . mostly.

– Many Coq power-users ignore it.

– GTK/OCaml is outdated; GTK/SML is unavailable.

– – – Need to duplicate editor implementation efforts.

Bilingual approach

Realistic assumption:

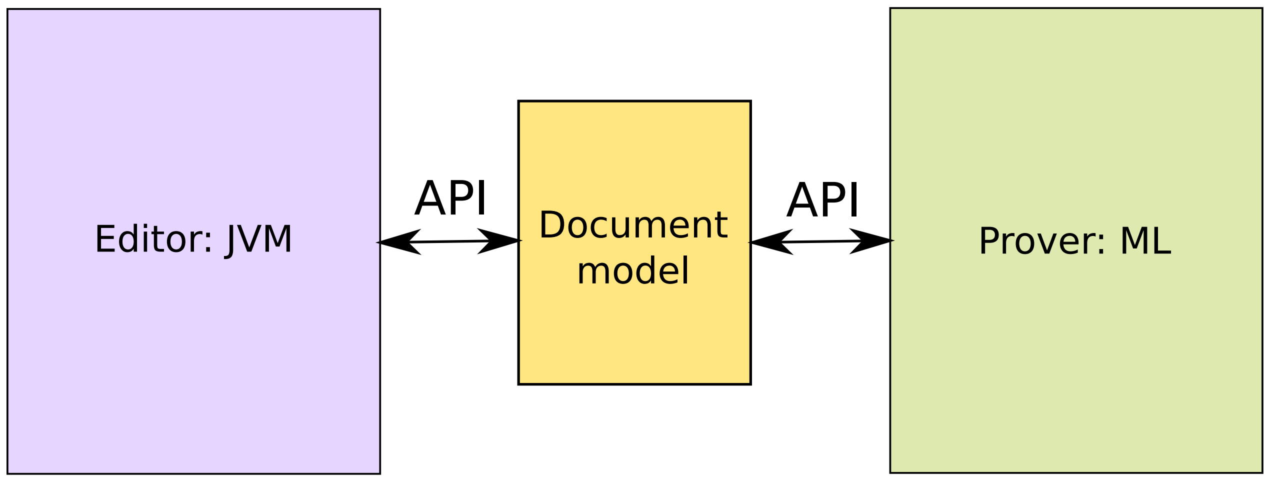
- Prover: ML (SML, OCaml, Haskell)
- Editor: Java

Big problem: How to integrate the two worlds?

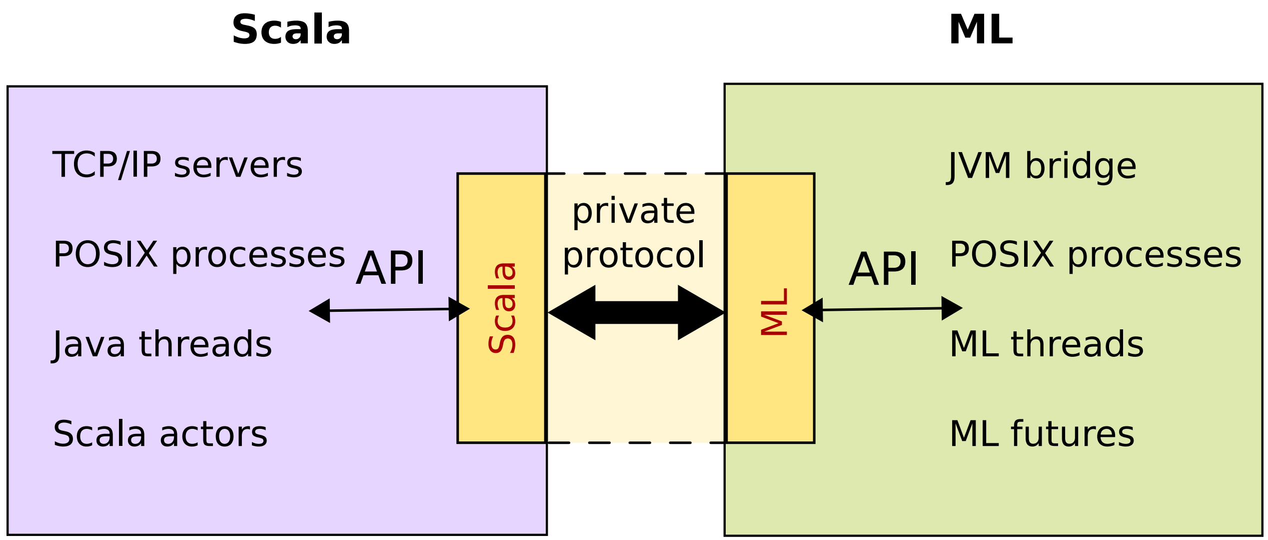
- Separate processes: requires marshalling, serialization, protocols.
- Different implementation languages and programming paradigms.
- Different cultural backgrounds!

Front-end (editor)	**Back-end** (prover)
"XML"	plain text
weakly structured data	"λ-calculus"
OO programming	higher-order FP
Java	ML

PIDE architecture: conceptual view



PIDE architecture: implementation view



Design principles:

- private protocol for prover connectivity
  (asynchronous interaction, parallel evaluation)
- public Scala API
  (timeless, stateless, static typing)

PIDE applications

Isabelle/jEdit:

- included in Isabelle distribution as default prover interface
- main application to demonstrate PIDE concepts in reality
- ready for everyday use since October 2011

Isabelle/Eclipse: (Andrius Velykis)

- `https://github.com/andriusvelykis/isabelle-eclipse`
- port of Isabelle2012 Prover IDE to Eclipse
- demonstrates viability and portability of PIDE concepts

Isabelle/Clide: (Christoph Lüth, Martin Ring)

- `https://github.com/martinring/clide`
- Prover IDE based on Isabelle/Scala and Play web framework
- demonstrates flexibility of PIDE concepts: web service instead of rich-client

Asynchronous READ-EVAL-PRINT
(without LOOP)

Command Transactions

Isolated commands:

- "small" toplevel state st: $Toplevel.state$
- command transaction tr as partial function over st
  we write $st \longrightarrow^{tr} st'$ for $st' = tr\ st$
- general structure: $tr = read;\ eval;\ print$
  (for example $tr = intern;\ run;\ extern$ in LISP)

Interaction view:

$tr\ st =$

let $eval = read\ src$ **in**	— $read$ does not use st
let $(y,\ st') = eval\ st$ **in**	— main transaction
let $() = print\ st'\ x$ **in** st'	— $print$ does not update st'

Note: flexibility in separating $read;\ eval;\ print$

Document Structure

Traditional structure:

- local body: linear sequence of command spans
- global outline: directed acyclic graph (DAG) of theories

Notes:

- in theory: document consists single linear sequence

  $st \longrightarrow^{tr} st' \longrightarrow^{tr'} st'' \ldots$

- in practice: independent paths in graph important for parallelism

Approach:

- incremental editing of command sequences
- parallel scheduling of resulting R-E-P phases
- continuous processing while the user is editing

Document model with immutable versions

- overall $Document.state$ with associated $Execution$
- document version contains command structure and assignment of "exec ids" for command transactions
- implicit sharing between versions (content and running commands)
- functional document update

  $Document.define\_command: command\_id \to src \to state \to state$
  $Document.update: version\_id \to version\_id \to edit^* \to state \to state$
  $Document.remove\_versions: version\_id^* \to state \to state$

  $edit \approx insert \mid remove \mid dependencies \mid perspective$

- global execution management

  $Execution.start: unit \to execution\_id$
  $Execution.discontinue: unit \to unit$
  $Execution.running: execution\_id \to exec\_id \to bool$
  $Execution.fork \to exec\_id \to (\alpha \to unit) \to \alpha\ future$
  $Execution.cancel: exec\_id \to unit$

Asynchronous print functions

Observations:

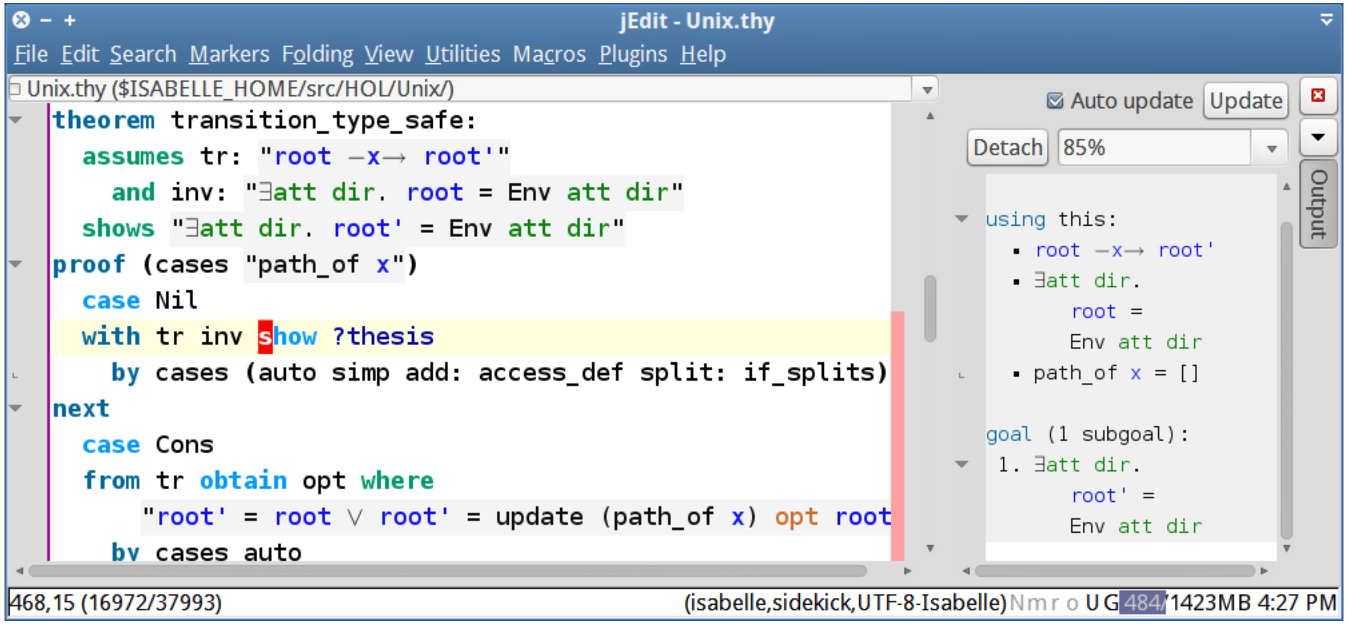
- cumulative PRINT operations consume more time than EVAL (output of goals is slower than most proof steps)
- PRINT depends on user perspective
- PRINT may diverge or fail
- PRINT augments results without changing proof state
- many different PRINTs may be run independently

Approach:

- each command transaction is associated with several $exec_id$s: one $eval$ + many $prints$
- document content forms union of markup
- print management via declarative parameters: startup delay, time-out, task priority, persistence, strictness wrt. eval state

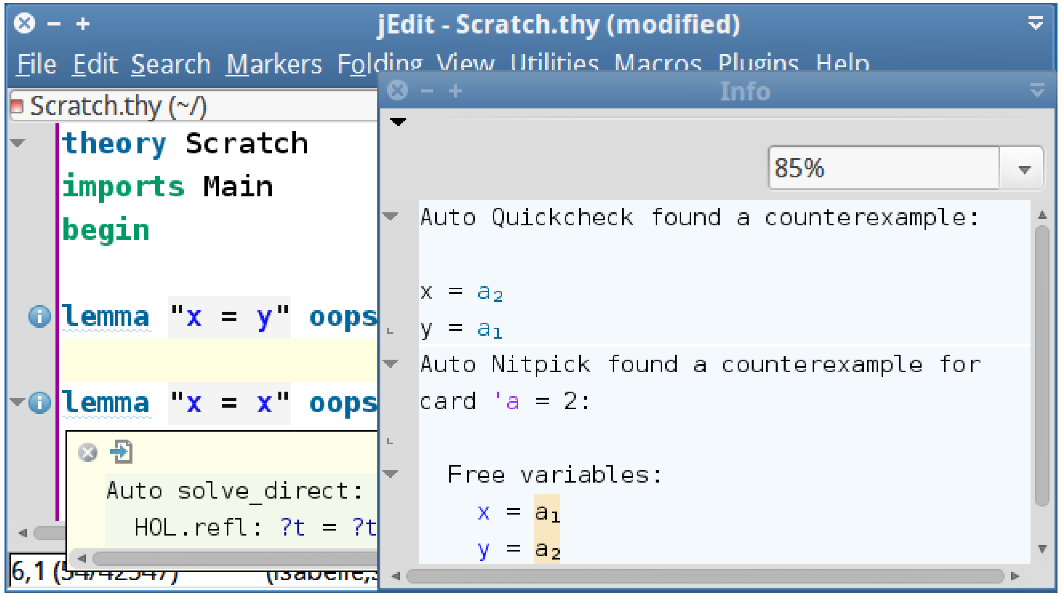
Application: print proof state

- parameters: $\{pri = 1,\ persistent = false,\ strict = true\}$
- change of perspective invokes or revokes asynchronous / parallel prints sponteneously
- GUI panel follows cursor movement to display content



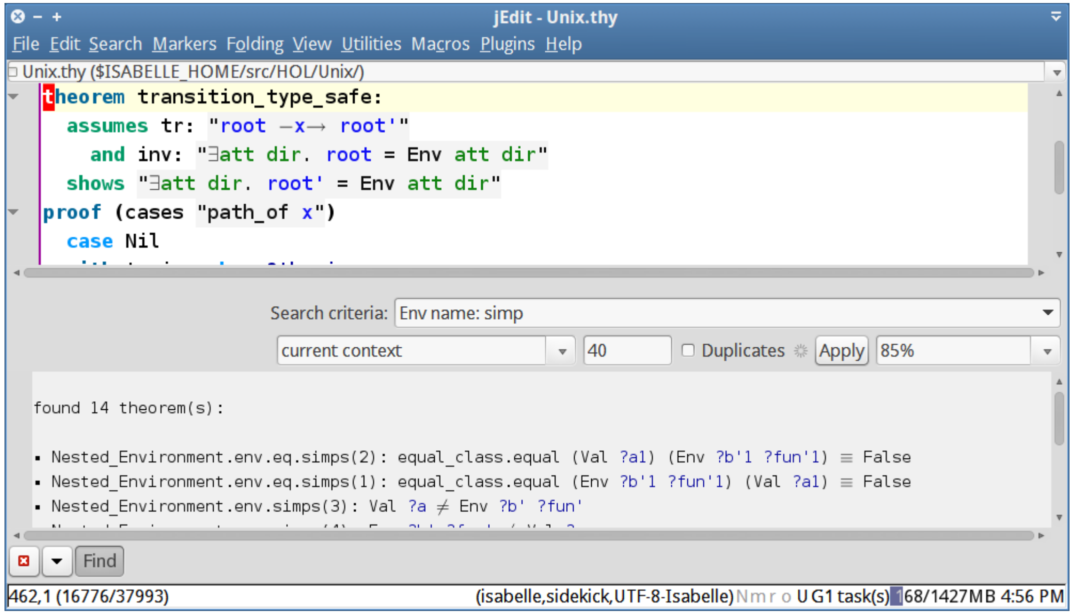
Application: automatically tried tools

- parameters: $\{delay = 1s,\ timeout = 4s,\ pri = -10,\ persistent = true,\ strict = true\}$
- long-running tasks with little output, e.g. automated (dis-)provers
- comment on existing document content via information message



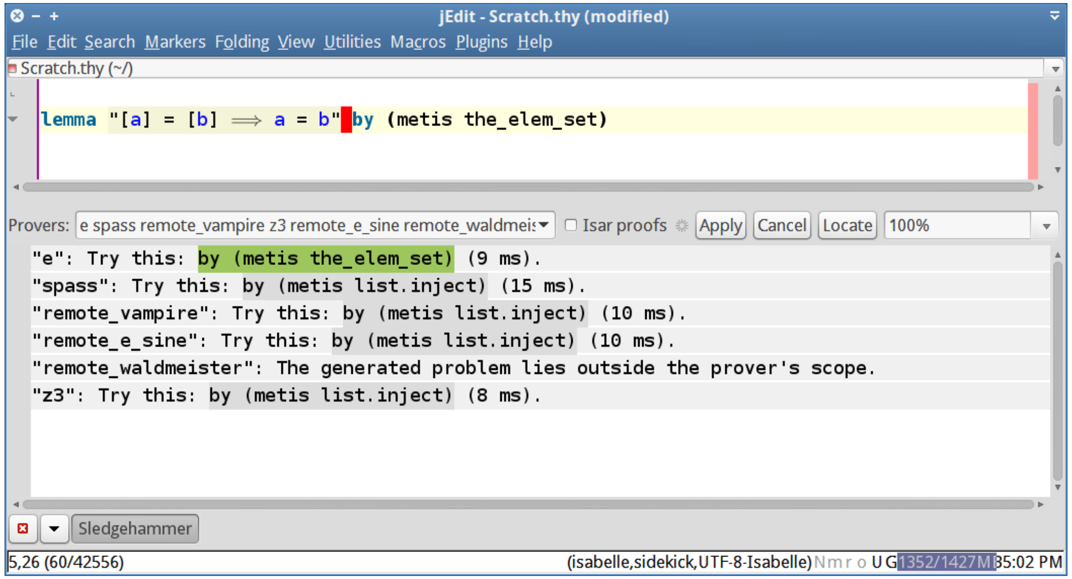
Application: query operations with user input

- parameters: $\{pri = 0,\ persistent = false,\ strict = false\}$
- separate infrastructure to manage temporary document overlays
- stateful GUI panel with user input, system output, and control of corresponding command transaction (status icon, cancel button)



Application: Sledgehammer

- heavy-duty query operation, with long-running ATPs and SMTs in the background (local or remote)
- progress indicator (spinning disk)
- clickable output
- implementation: trivial corollary of above concepts



Conclusions

Conclusions

- Substantial reforms of LCF-style theorem proving is possible.
- Reforms do not break with the history, but learn from it.
- Need to think beyond ML.
- Need to change user habits.
- Feasibility and scalability of PIDE proven by Isabelle/jEdit
  `http://isabelle.in.tum.de/`