

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)

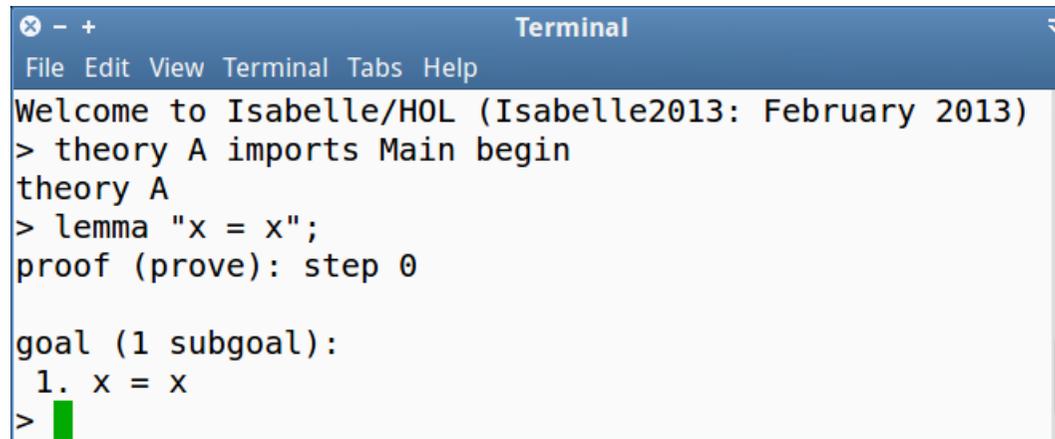
Isabelle/HOL (T. Nipkow 1992)

Isabelle/Isar (M. Wenzel 1999)

⋮

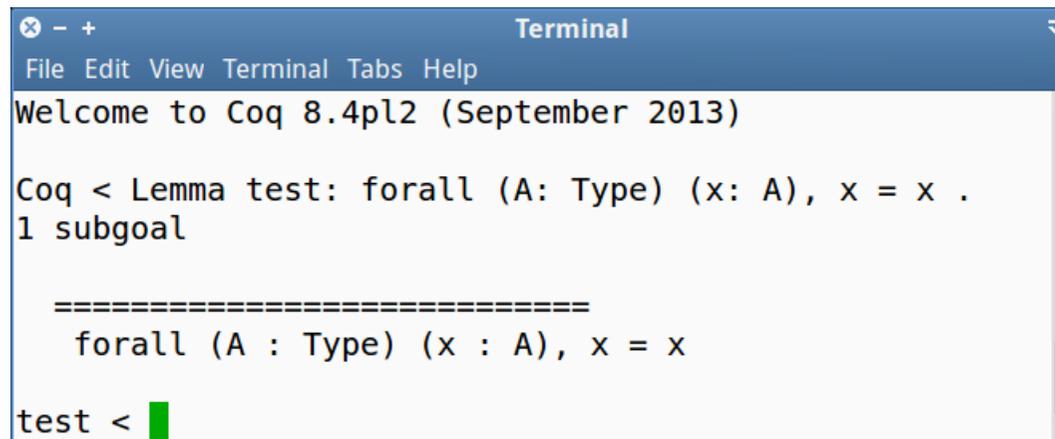
Isabelle2013-2 (M. Wenzel 2013, coordinator)

TTY interaction (\approx 1979)



```
Terminal
File Edit View Terminal Tabs Help
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
>
```



```
Terminal
File Edit View Terminal Tabs Help
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

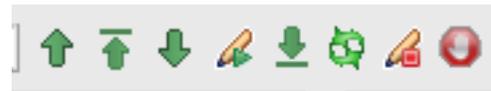
Notes:

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

Proof General (\approx 1999)

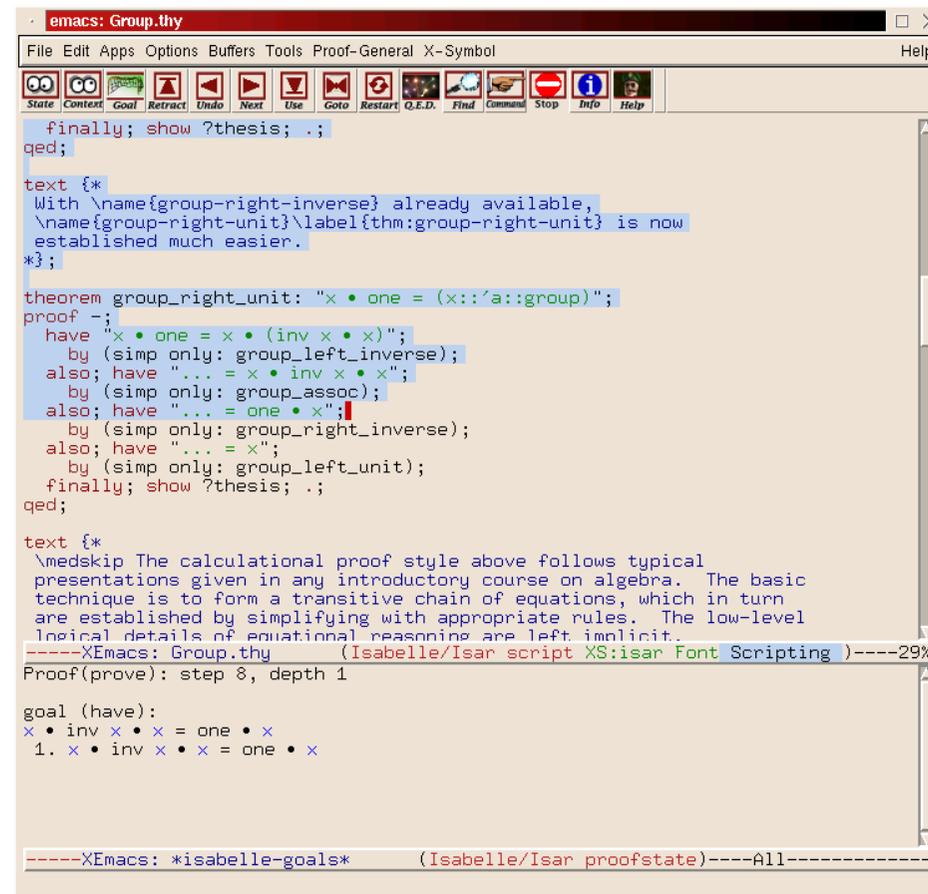
Approach:

- 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



The screenshot shows the Emacs editor interface with a file named 'Group.thy'. The editor contains Isabelle/HOL code for proving a theorem about the right identity in a group. The code includes a theorem statement, a proof using the 'simp' tactic, and a text block explaining the proof style. The status bar at the bottom indicates the current proof state: 'Proof(prove): step 8, depth 1' and 'goal (have): x • inv x • x = one • x'. The goal is shown as a list of equations to be proved.

```
emacs: Group.thy
File Edit Apps Options Buffers Tools Proof-General X-Symbol Help
State Context Goal Retract Undo Next Use Goto Restart Q.E.D. Find Command Stop Info Help

  finally; show ?thesis; .;
qed;

text {*
  With \name{group-right-inverse} already available,
  \name{group-right-unit}\label{thm:group-right-unit} is now
  established much easier.
*};

theorem group_right_unit: "x • one = (x::'a::group)";
proof -;
  have "x • one = x • (inv x • x)";
  by (simp only: group_left_inverse);
  also; have "... = x • inv x • x";
  by (simp only: group_assoc);
  also; have "... = one • x";
  by (simp only: group_right_inverse);
  also; have "... = x";
  by (simp only: group_left_unit);
  finally; show ?thesis; .;
qed;

text {*
  \medskip The calculational proof style above follows typical
  presentations given in any introductory course on algebra. The basic
  technique is to form a transitive chain of equations, which in turn
  are established by simplifying with appropriate rules. The low-level
  logical details of equational reasoning are left implicit.
  -----XEmacs: Group.thy (Isabelle/Isar script XS:isar Font Scripting)-----29%
Proof(prove): step 8, depth 1

goal (have):
x • inv x • x = one • x
1. x • inv x • x = one • x

-----XEmacs: *isabelle-goals* (Isabelle/Isar proofstate)-----All-----
```

Example: CoqIDE

The screenshot shows the CoqIDE interface with a Coq script on the left and a list of subgoals on the right. The script defines a lemma `nat_eq_dec` and a definition `pred`. The subgoals show the current state of the proof, including the induction hypothesis `IHn` and the goal `S m = S m`.

```
File Edit Navigation Try Tactics Templates Queries Display Compile Windows Help
Intro.v Examples.v
rewrite IHn.
reflexivity.
Qed.

Lemma nat_eq_dec : forall (n m : nat), {n = m} +
induction n.
destruct m as [|m].
left.
reflexivity.
right.
discriminate.
destruct m as [|m].
right; discriminate.
destruct (IHn m) as [Hm|Hm].
left.
rewrite Hm.
reflexivity.
right.
intros Hnm.
apply Hm.
injection Hnm.
tauto.
Defined.

Eval compute in (nat_eq_dec 2 2).
Eval compute in (nat_eq_dec 2 1).

Definition pred (n:nat) : option nat :=
match n with
| 0 => None
```

2 subgoals
n : nat
IHn : forall m : nat, {n = m} + {n <> m}
m : nat
Hm : n = m
----- (1/2)
S m = S m
----- (2/2)
{S n = S m} + {S n <> S m}

Ready in Predicate_Logic, proving nat_eq_dec Line: 159 Char: 13 CoqIde started

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 continuously 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

```
header {* Finite sequences *}

theory Seq
imports Main
begin

datatype 'a seq = Empty | Seq 'a "'a seq"

fun conc :: "'a seq => 'a seq => 'a seq"
where
  "conc Empty ys = ys"
| "conc (Seq x xs) ys = Seq x (conc xs ys)"

fun reverse
where
  "reverse Empty = Empty"
| "reverse (Seq x xs) = conc (reverse xs) (Seq x Empty)"

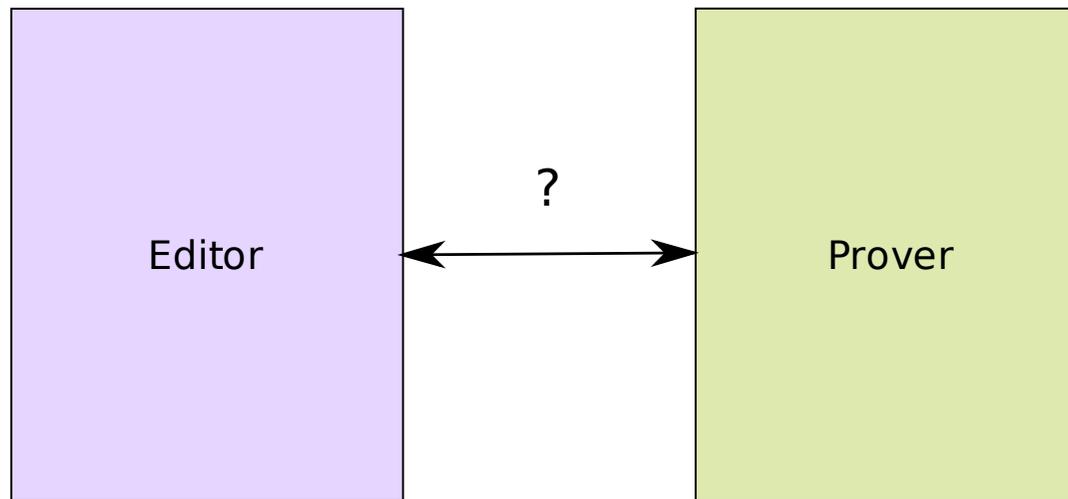
constant "Seq.seq.Seq"
  :: 'a => 'a seq => 'a seq
```

Found termination order: $(\lambda p. \text{size}(\text{fst } p)) \text{ < } *mlex* \text{ > } \{\}$

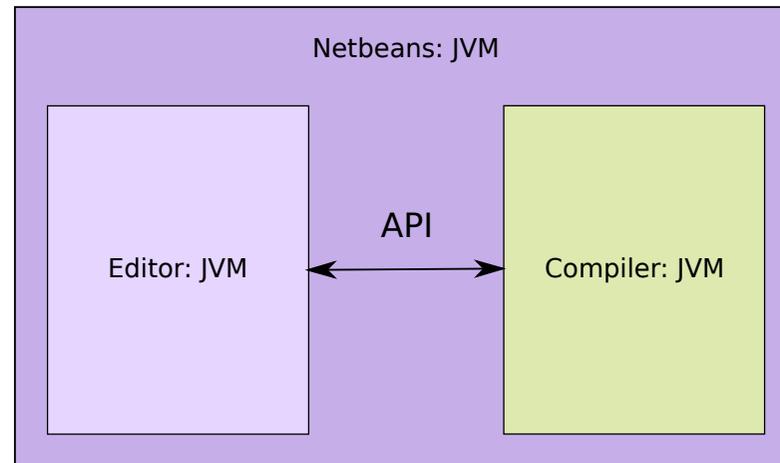
14,6 (209/791) (isabelle,sidekick,UTF-8-Isabelle)Nm r o U G 750/1174MB 4:16 PM

PIDE architecture

The connectivity problem



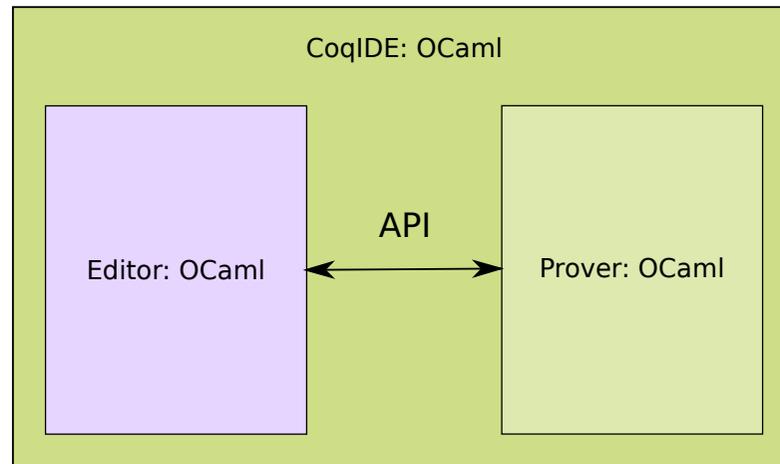
Example: Java IDE



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:

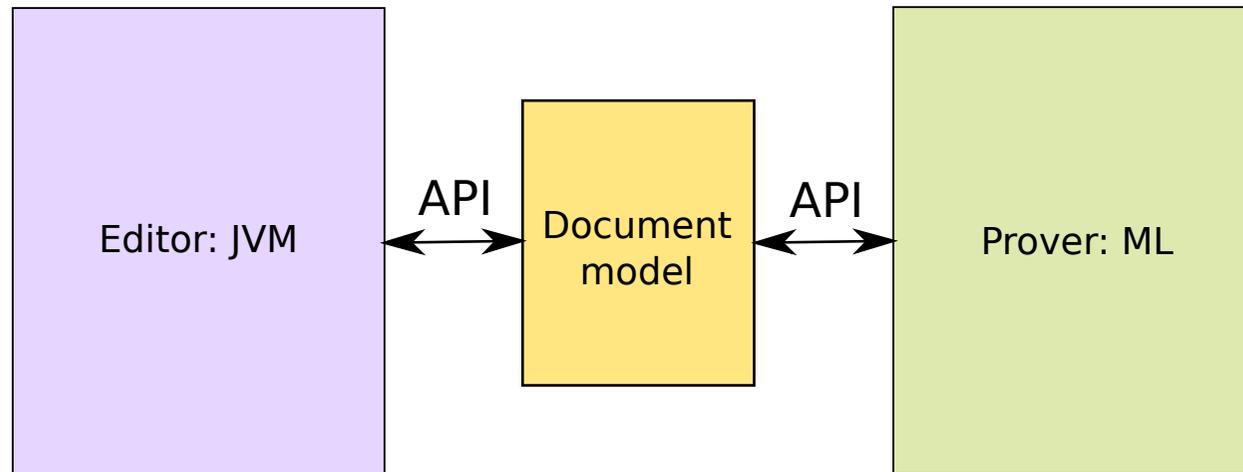
- Prover: ML (SML, OCaml, Haskell)
- Editor: Java

Big problem: How to integrate the two worlds?

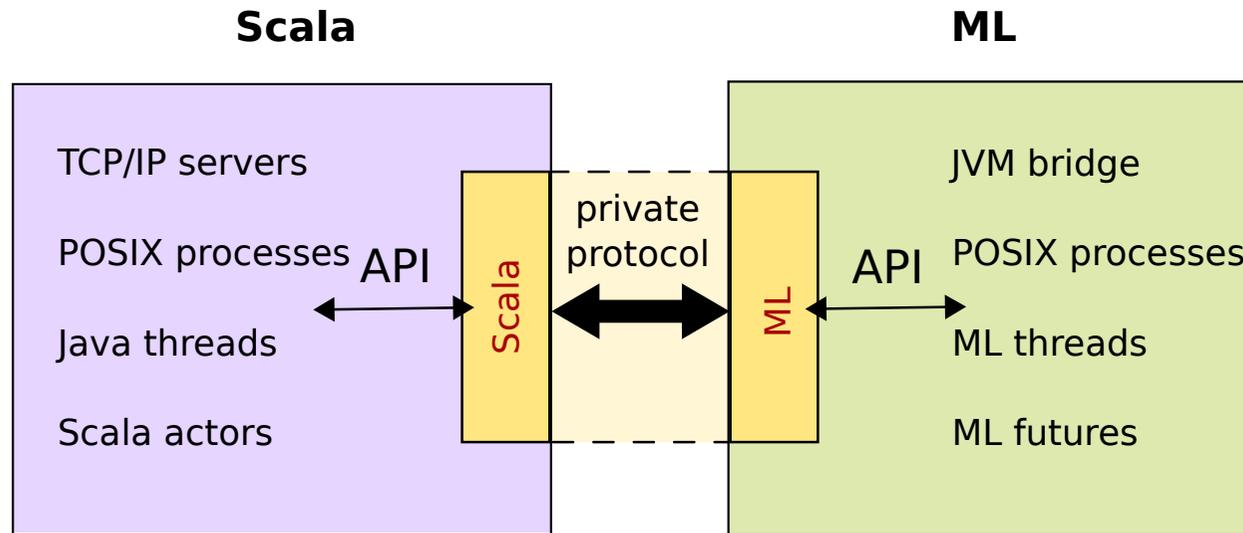
- Separate processes: requires marshalling, serialization, protocols.
- Different implementation languages and programming paradigms.
- Different cultural backgrounds!

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 $st \xrightarrow{tr} st'$ 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

$$st \xrightarrow{tr} st' \xrightarrow{tr'} st'' \dots$$

- 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 → src → state → state

Document.update: version_id → version_id → edit → state → state*

Document.remove_versions: version_id → state → state*

edit ≈ insert | remove | dependencies | perspective

- global execution management

Execution.start: unit → execution_id

Execution.discontinue: unit → unit

Execution.running: execution_id → exec_id → bool

Execution.fork → exec_id → (α → unit) → α future

Execution.cancel: exec_id → unit

Asynchronous print functions

Observations:

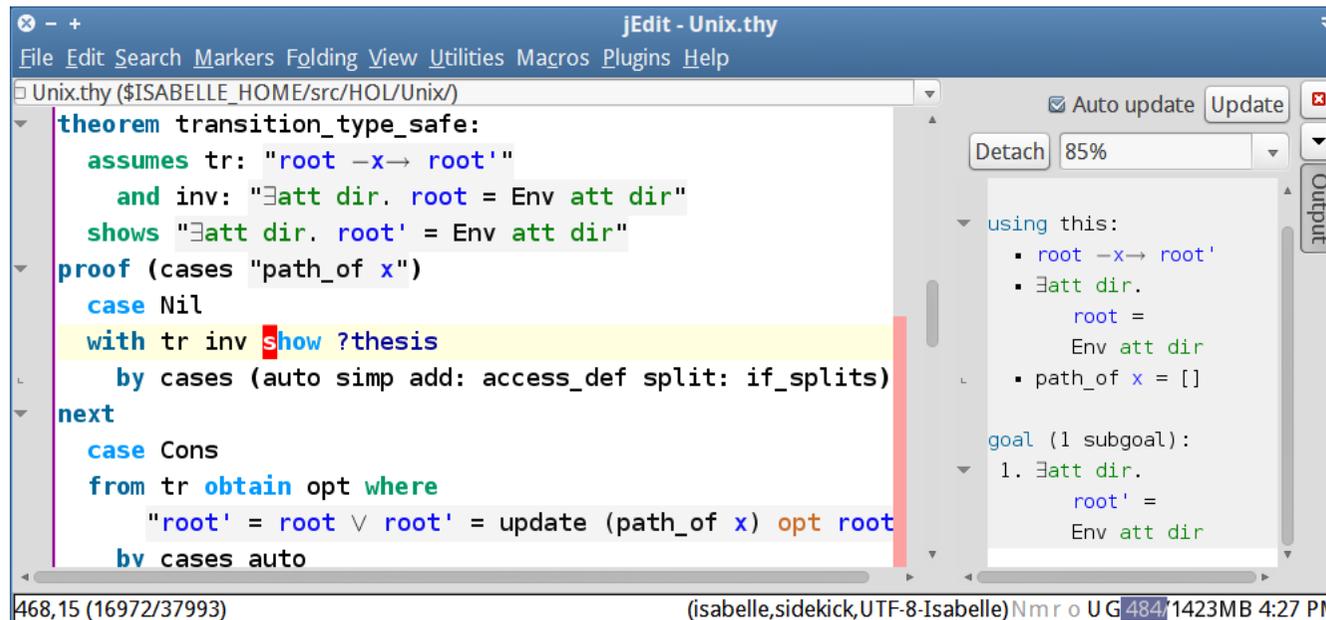
- cumulative PRINT operations consume more time than EVAL (output of goals is slower than most proof steps)
- PRINT depends on user perspective
- PRINT may diverge or fail
- PRINT augments results without changing proof state
- many different PRINTs may be run independently

Approach:

- each command transaction is associated with several *exec_ids*: 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 spontaneously
- GUI panel follows cursor movement to display content



The screenshot shows the jEdit IDE with a file named Unix.thy. The main editor displays a theorem proof:

```
theorem transition_type_safe:
  assumes tr: "root -x→ root'"
  and inv: "∃att dir. root = Env att dir"
  shows "∃att dir. root' = Env att dir"
proof (cases "path_of x")
  case Nil
  with tr inv show ?thesis
  by cases (auto simp add: access_def split: if_splits)
next
  case Cons
  from tr obtain opt where
    "root' = root ∨ root' = update (path_of x) opt root"
  bv cases auto
```

The Output panel on the right shows the current proof state:

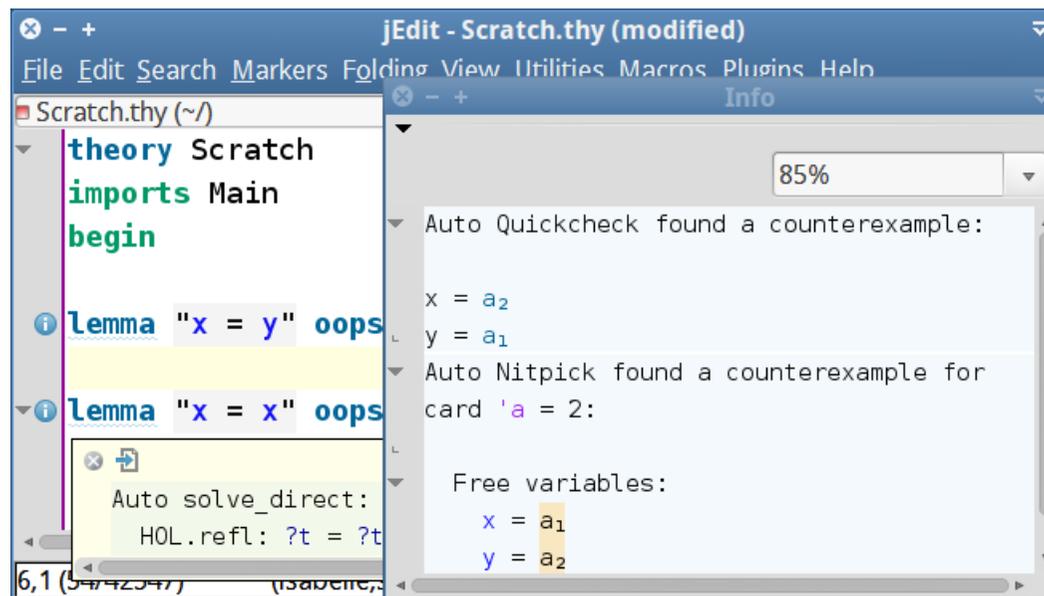
```
using this:
  • root -x→ root'
  • ∃att dir.
    root =
      Env att dir
  • path_of x = []

goal (1 subgoal):
  1. ∃att dir.
    root' =
      Env att dir
```

The status bar at the bottom indicates the cursor is at line 468, column 15 (16972/37993) in the file Unix.thy.

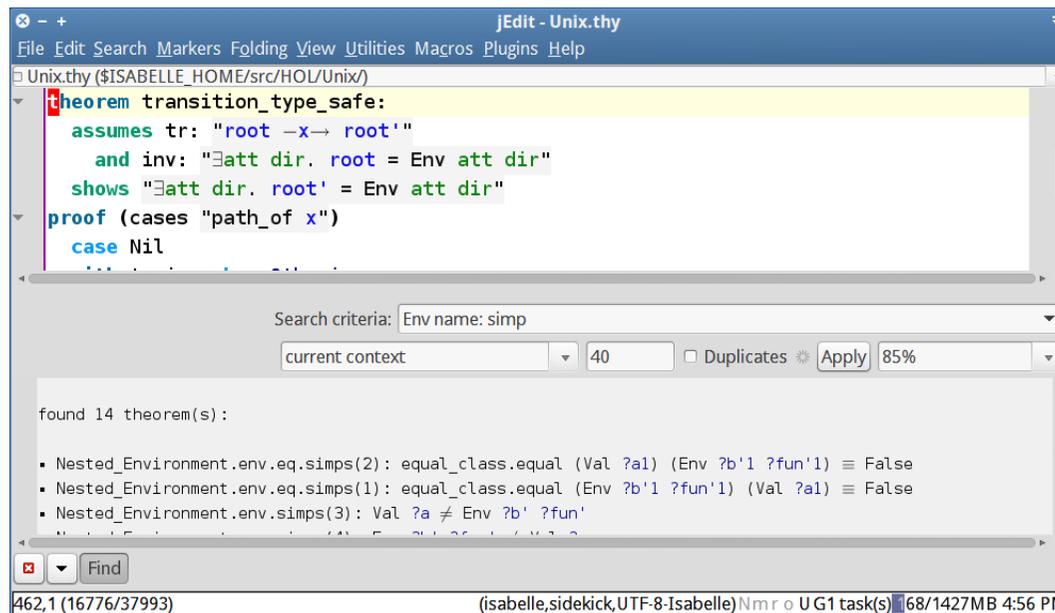
Application: automatically tried tools

- 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)



```
File Edit Search Markers Folding View Utilities Macros Plugins Help
Unix.thy ($ISABELLE_HOME/src/HOL/Unix)
theorem transition_type_safe:
  assumes tr: "root -x→ root'"
  and inv: "∃att dir. root = Env att dir"
  shows "∃att dir. root' = Env att dir"
proof (cases "path_of x")
case Nil
```

Search criteria: Env name: simp
current context 40 Duplicates Apply 85%

found 14 theorem(s):

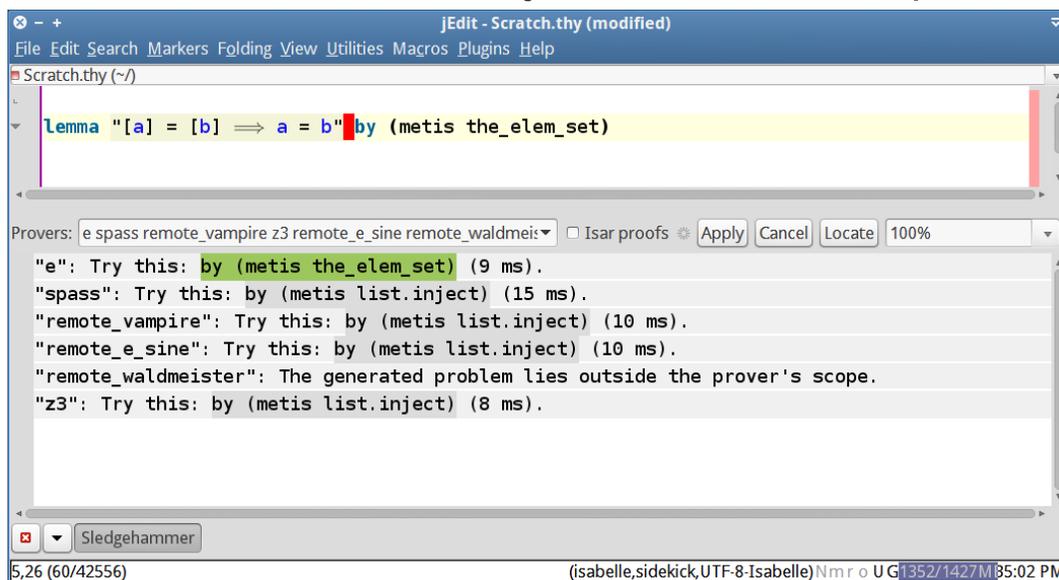
- Nested_Environment.env.eq.simps(2): equal_class.equal (Val ?a1) (Env ?b'1 ?fun'1) ≡ False
- Nested_Environment.env.eq.simps(1): equal_class.equal (Env ?b'1 ?fun'1) (Val ?a1) ≡ False
- Nested_Environment.env.simps(3): Val ?a ≠ Env ?b' ?fun'

Find

462,1 (16776/37993) (isabelle,sidekick,UTF-8-Isabelle) Nmr o U G1 task(s) 68/1427MB 4:56 PM

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



The screenshot shows the Sledgehammer application interface. At the top, there is a menu bar with options: File, Edit, Search, Markers, Folding, View, Utilities, Macros, Plugins, Help. Below the menu bar, the file name "Scratch.thy (~)" is displayed. The main text area contains a lemma: `Lemma "[a] = [b] ==> a = b" by (metis the_elem_set)`. Below the lemma, there is a "Provers" section with a dropdown menu showing "e spass remote_vampire z3 remote_e_sine remote_waldmeister" and a checkbox for "Isar proofs". There are buttons for "Apply", "Cancel", and "Locate", and a progress indicator set to "100%". The output area shows the results for each prover: "e": Try this: by (metis the_elem_set) (9 ms). "spass": Try this: by (metis list.inject) (15 ms). "remote_vampire": Try this: by (metis list.inject) (10 ms). "remote_e_sine": Try this: by (metis list.inject) (10 ms). "remote_waldmeister": The generated problem lies outside the prover's scope. "z3": Try this: by (metis list.inject) (8 ms). At the bottom, there is a status bar with the text "Sledgehammer" and a small icon. The bottom right corner of the window shows the system tray information: "(isabelle,sidekick,UTF-8-Isabelle)Nm r o U G 1352/1427 M 85:02 PM".

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/>