Parallel and asynchronous interactive theorem proving in Isabelle

from READ-EVAL-PRINT to continuous processing of proof documents

Makarius Wenzel
Univ. Paris-Sud, Laboratoire LRI

September 2013

Project Paral-ITP (ANR-11-INSE-001)
http://paral-itp.lri.fr

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 an inherently synchronous and sequential process, or are these merely accidental limitations of the implementations?

Since 2005 the multicore challenge imposes the burden of explicit parallelism to application developers who still want to participate in Moore's Law for CPU performance. Proof assistants require good performance (and reactivity) when users want to develop large libraries of formal proofs. Isabelle with its underlying Poly/ML platform has managed to support parallel batch mode routinely in 2008, which has impacted the size and complexity of formalizations in the past few years.

This initial success of parallel Isabelle has motivated further research about the combination of pervasive parallelism in the prover back-end with asynchronous

interaction in the front-end. The overall architecture rests on a timeless and stateless document model for formal theories: the editor continuously presents document updates to the prover, and the prover provides continuous feedback via formal markup on the original sources. Document operations apply to explicit immutable versions that are updated monotonically, and document markup is augmented while the prover explores formal content. The prover is free to schedule tasks in parallel, according to the structure of theories and proofs. The editor is free to react on user input and visualize already known document content, according to the real-time demands of the graphical user-interface.

This Prover IDE (PIDE) approach to interactive theorem proving has been under development over several years, and started to become available for production use with Isabelle/jEdit in October 2011. The underlying concepts and implementations have been refined significantly since then. 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 available 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 more advanced applications of our prover technology.

History

The LCF Prover Family

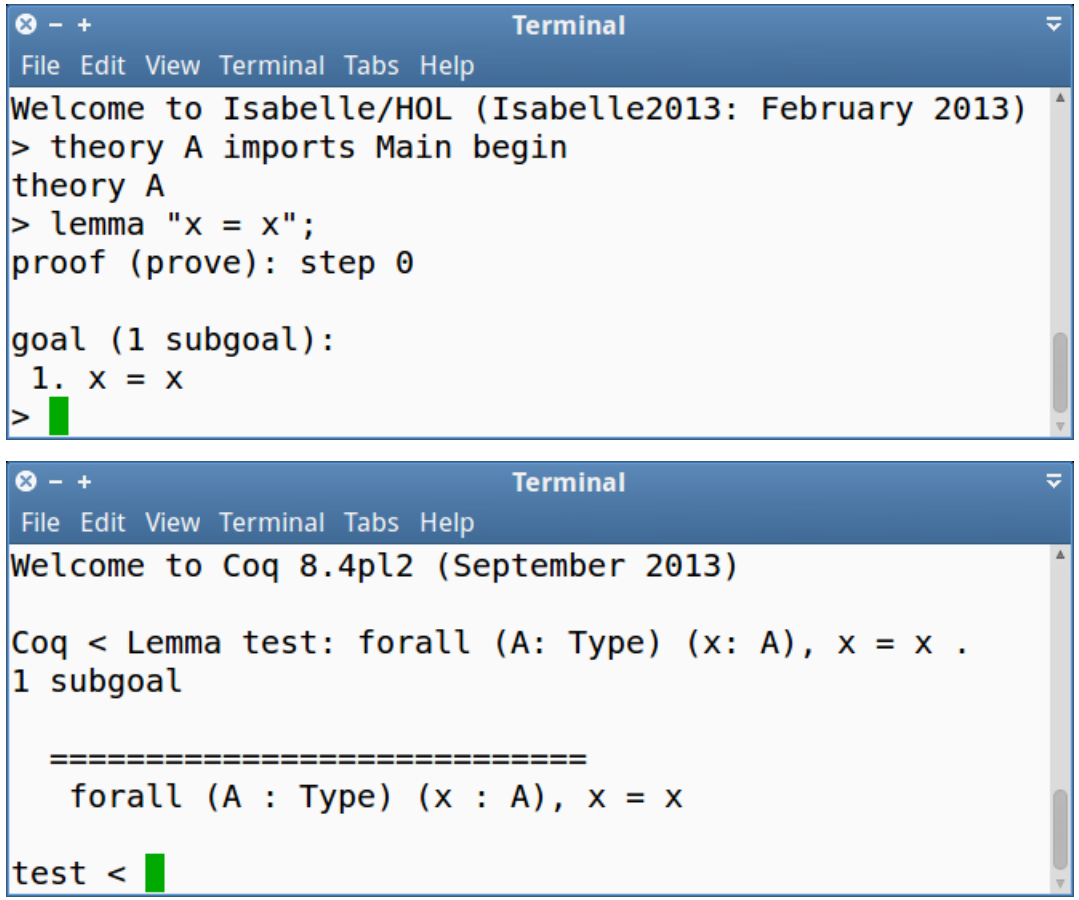
LCF
- Edinburgh LCF (R. Milner & M. Gordon 1979)
- Cambridge LCF (G. Huet & L. Paulson 1985)

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

Coq
- Coc (T. Coquand & G. Huet 1985/1988)
- ⋮
- Coq 8.4 (H. Herbelin 2012/2013, coordinator)

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

TTY interaction ($\approx$ 1979)



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

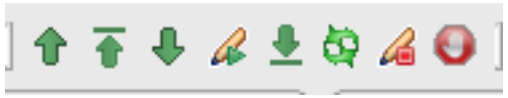
Notes:

- 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)

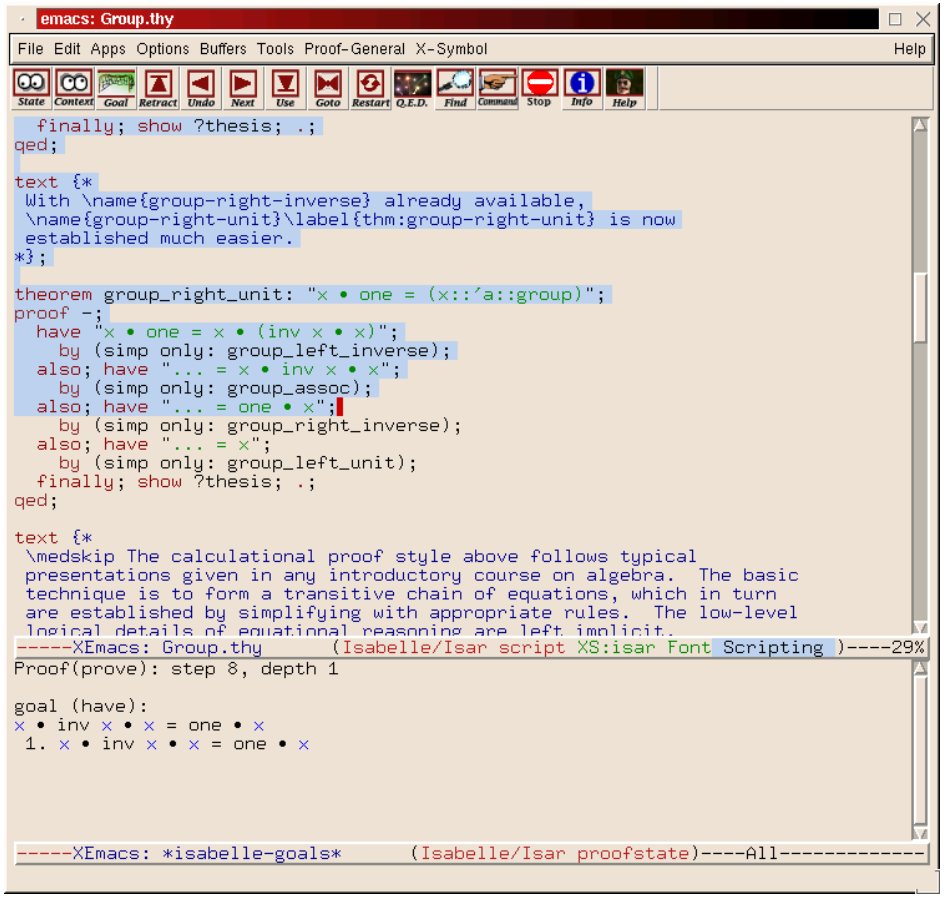
Approach:

- 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

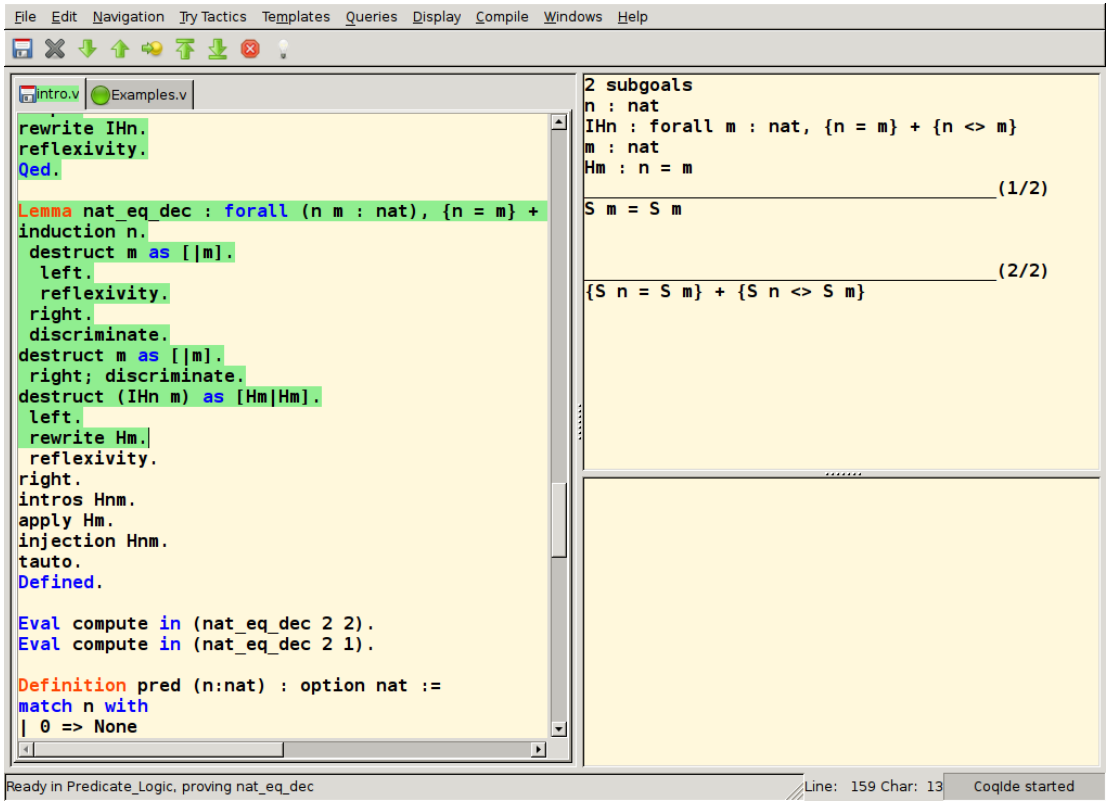


Example: Kopitiam (Eclipse + Coq)

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)

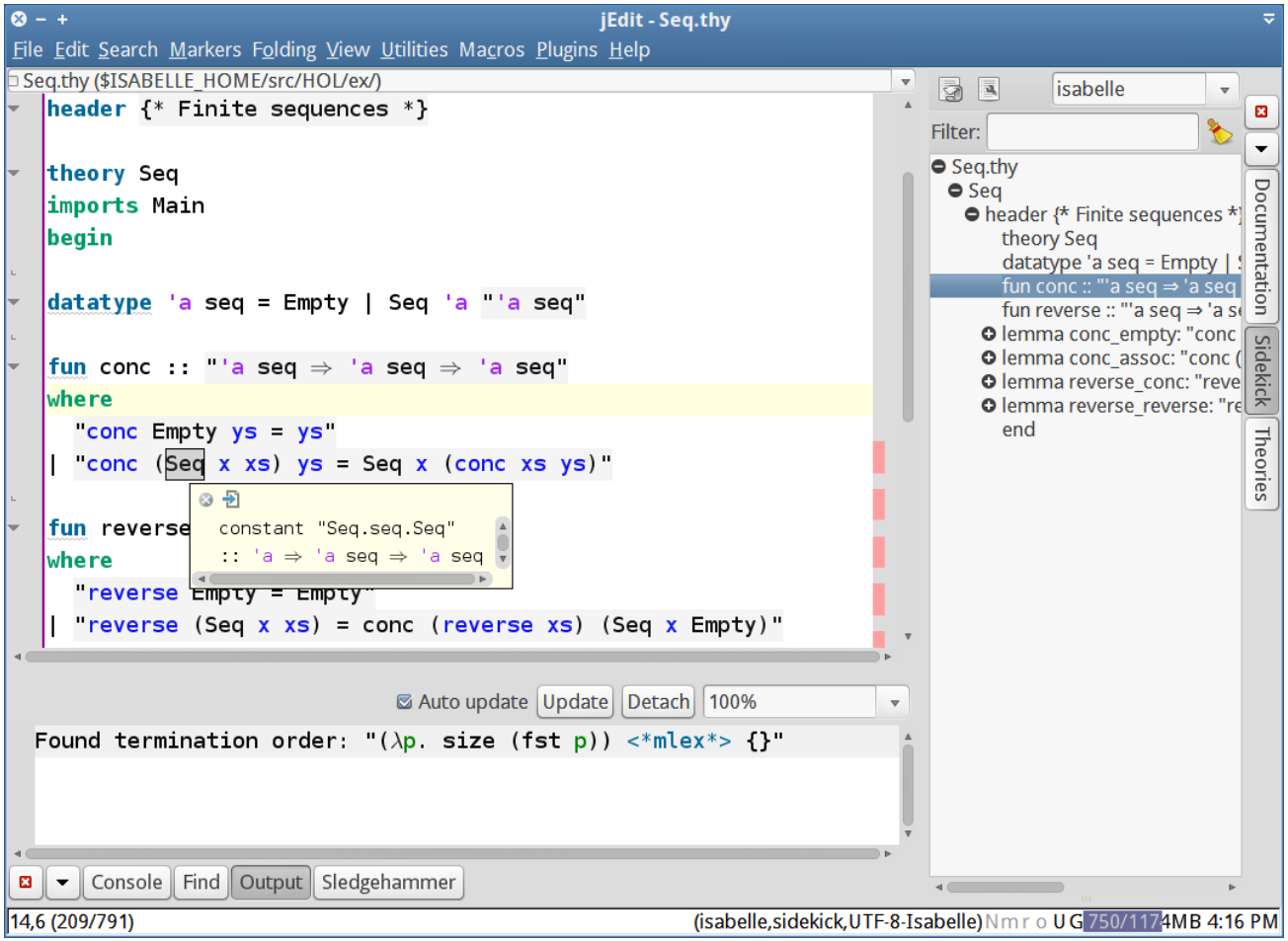
General aims:

- renovate interactive theorem proving for new user generations
- catch up with technological shifts:
  - parallel computing (multicore)
  - asynchronous interaction
  - advanced user-interfaces (IDEs)
- systematic support for user interaction and tool integration

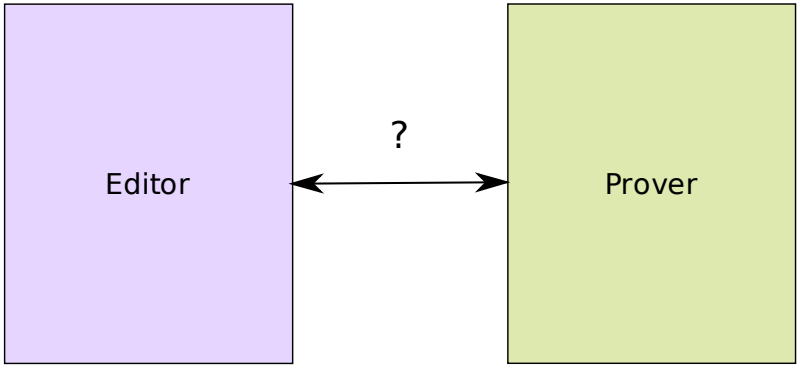
Approach:

- 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

Example: Isabelle/jEdit (September 2013)



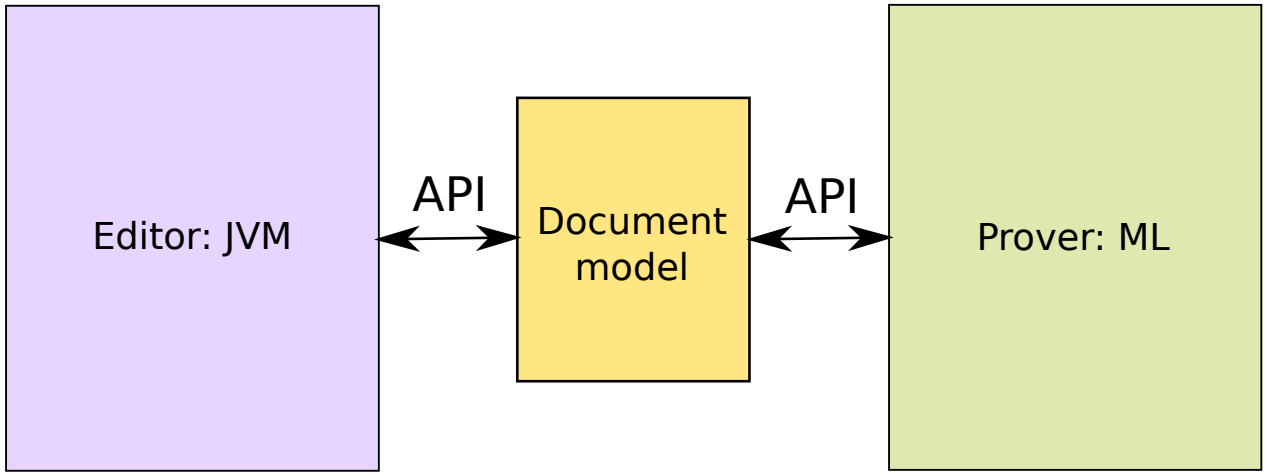
The connectivity problem



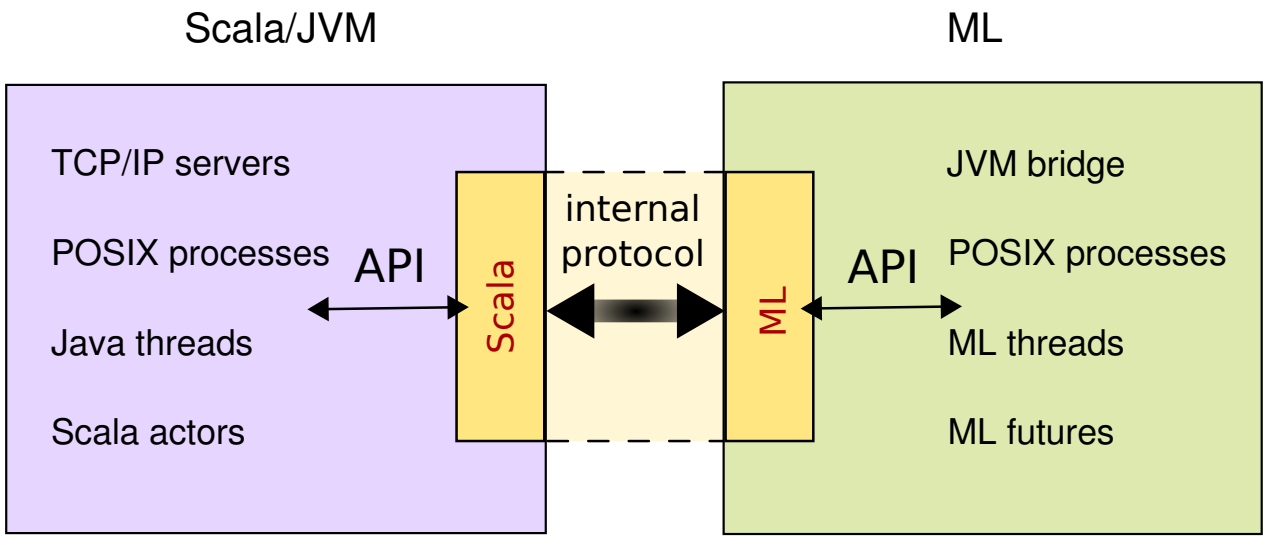
Front-end (editor)	**Back-end** (prover)
"XML"	plain text
weakly structured data	"λ-calculus"
OO programming	higher-order FP
Java	ML

Our answer: bridge gap via Scala/JVM (Martin Odersky, EPFL)

Isabelle/Scala architecture: conceptual view



Isabelle/Scala architecture: implementation view



Design principles:

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

ML versus Scala

ML:

- efficient functional programming with parallel evaluation
- implementation and extension language of logical framework
- ML embedded into the formal context
- leverages decades of research into prover technology

Scala:

- functional object-oriented programming with concurrency
- system programming environment for the prover
- Scala access to formal document content
- leverages JVM frameworks (IDEs, editors, web servers etc.)

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

Parallel Prover Architecture

Summary

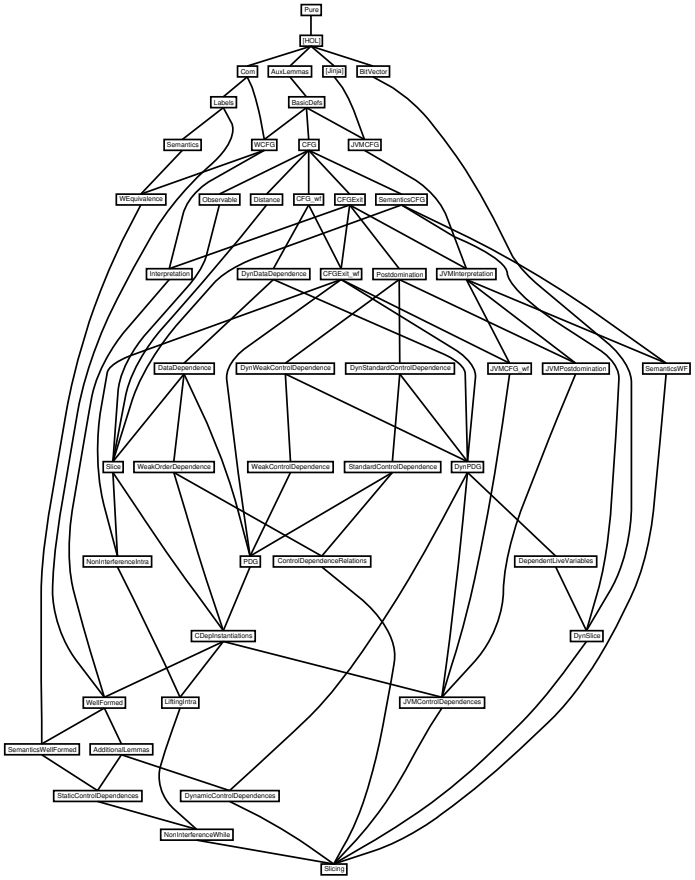
Change of Moore's Law: (since 2005)

- application developers need to care about explicit parallelism
- ignoring the challenge: loose factor 10 on 16 core hardware today
- mastering the challenge: catch up with multiplication of cores (doubling every 18 months)

Parallel LCF proof processing:

- problem structure: practical proof irrelevance
- programming paradigm: purely functional programming in ML (explicit context, immutable data)
- practical problems to achieve actual performance, but feasible

Overall document structure



theory A **imports** $B_1 \ldots B_n$
begin
$\vdots$
inductive $P \ldots$
theorem a: A $\langle proof \rangle$
theorem b: B $\langle proof \rangle$
theorem c: C $\langle proof \rangle$
$\vdots$
  have $A \wedge B$
  proof
    show A **by** $simp$
    show B **by** $blast$
  qed
$\vdots$
end

Stack of parallel system layers

- Multicore Hardware (Intel, AMD)
- Operating System (Linux, Windows, Mac OS X)
- Poly/ML compiler and runtime system (David Matthews)
- Isabelle/ML futures and parallel skeletons: explicit parallelism
- Isabelle theory and proof processing: implicit parallelism
- Isabelle session build management: implicit parellelism, tree of ML processes
- asynchronous and parallel front-end technology (PIDE)

Parallel Poly/ML (David Matthews, 2007, 2012)

Hardware: regular shared-memory multiprocessor (2–32 cores)

Operating system: native POSIX threads (pthreads)

ML multithreading: structures $Thread$, $Mutex$, $ConditionVar$

ML memory management:

- parallel garbage collection (various stages)
- online sharing of immutable values
  (reduced memory bandwidth requirements)

`http://www.polyml.org`

Parallel Isabelle/ML

Future values:

type α *future*
val $Future.fork$: $(unit \to \alpha) \to \alpha$ *future*
val $Future.join$: α *future* $\to \alpha$
val $Future.cancel$: α *future* $\to$ *unit*

strict evaluation: spontaneous execution via thread-pool

synchronous exceptions: propagation within nested task groups

asynchronous interrupts: cancellation and signalling of tasks

nested groups: implicit block structure of parallel program

dependencies: implicit graph of tasks determined statically

Explicit theory context (Paulson 1989)

Main judgment:

$$\boxed{\Theta,\ \Gamma \vdash \varphi}$$

- background theory Θ
  (polymorphic types, constants, axioms; global data)
- proof context Γ (fixed variables, assumptions; local data)

Operations on theories:

- extend and merge: $\Theta_3 = \Theta_1 \cup \Theta_2 \cup \tau \cup c :: \tau \cup c \equiv t$
- symbolic sub-theory check: $\Theta_1 \subseteq \Theta_2$
- transfer of results: $\Theta_1 \subseteq \Theta_2 \Longrightarrow \Theta_1 \vdash \varphi \Longrightarrow \Theta_2 \vdash \varphi$

Key benefit: timeless and stateless prover kernel

Proof promises

Main ideas:

- closed proof constants as place-holders for future proofs
- substitution by finished proofs – from proper theory context!
- special support for schematic polymorphism of proofs

New kernel inferences: wrt. proof promise environment Π

$$\frac{\mathrm{FV}\, A = \emptyset \quad \mathrm{TV}\, A = \{?\overline{\alpha}\}}{\Theta,\, \{a : A\},\, \emptyset \vdash a[?\overline{\alpha}] : A[?\overline{\alpha}]}\ (\mathit{promise})$$

$$\frac{\Theta,\, \Pi,\, \Gamma \vdash p : B \quad \Theta_0,\, \emptyset,\, \emptyset \vdash q : A \quad \Theta_0 \subseteq \Theta}{\Theta,\, \Pi - \{a : A\},\, \Gamma \vdash p[a := q] : B}\ (\mathit{fulfill})$$

Goal forks

Main ideas:

- specific infrastructure for goal-directed proof (via tactics)
- global accounting of forked proofs, avoid flooding by futures
- systematic tracking of errors

ML interfaces:

val $Goal.prove$: $Proof.context \rightarrow term \rightarrow tactic \rightarrow thm$
val $Goal.prove\_future$: $Proof.context \rightarrow term \rightarrow tactic \rightarrow thm$

- same signature
- same semantics for succesful tactic (without side-effects)
- same semantics for failing tactic, where it indicates global breakdown without local error handling

Performance and Scalability

Expected speedup in practice

Real speedup: $\varepsilon(1)$ / $\varepsilon(m)$ for m cores,
relating elapsed run-time of sequential vs. parallel application

Rules of Thumb:

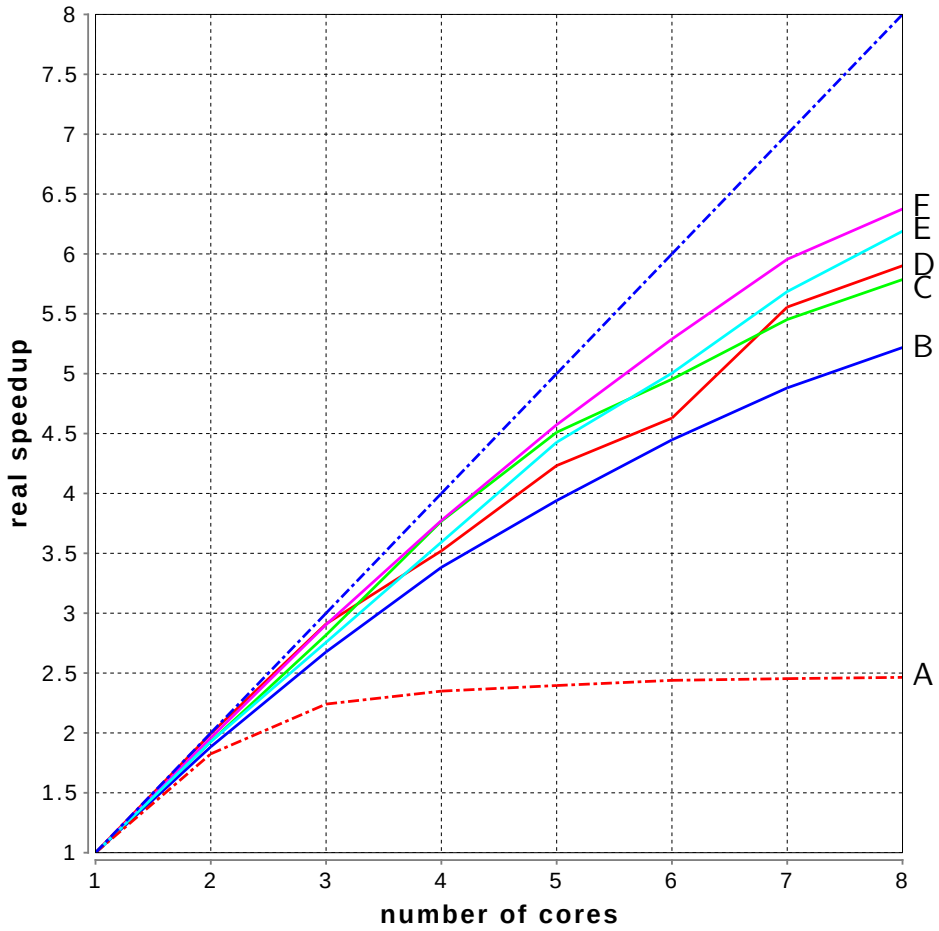
very naively $speedup = m$
before you make an implementation

naively $speedup \approx 1$
first attempt with too little parallelism in the application

asymptotically $speedup \approx 0$
worse than Amdahl's law, excessive overhead for many cores

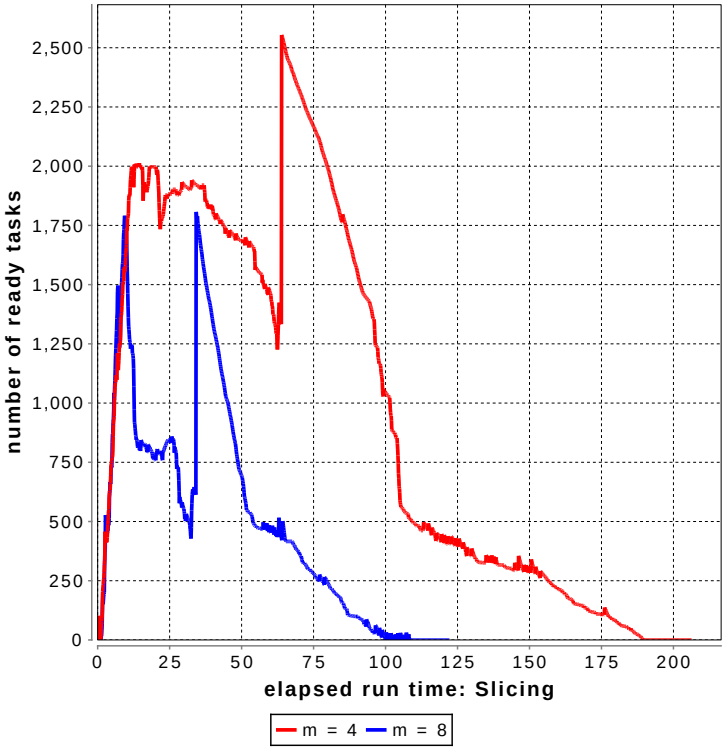
realistically $speedup \approx 0.66 \times m$, for reasonable $m = 4, 8, 16, \ldots$

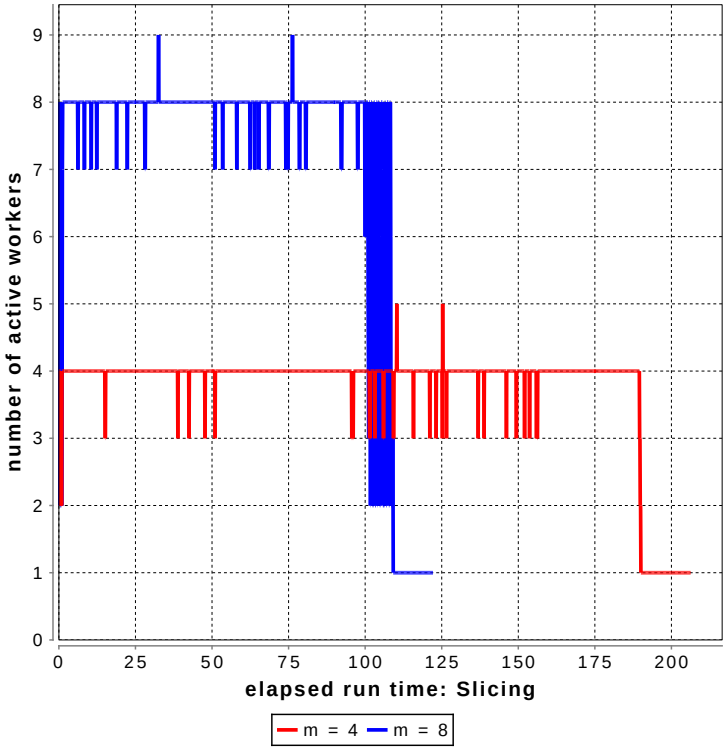
Isabelle2013 (February 2013) on 8 cores



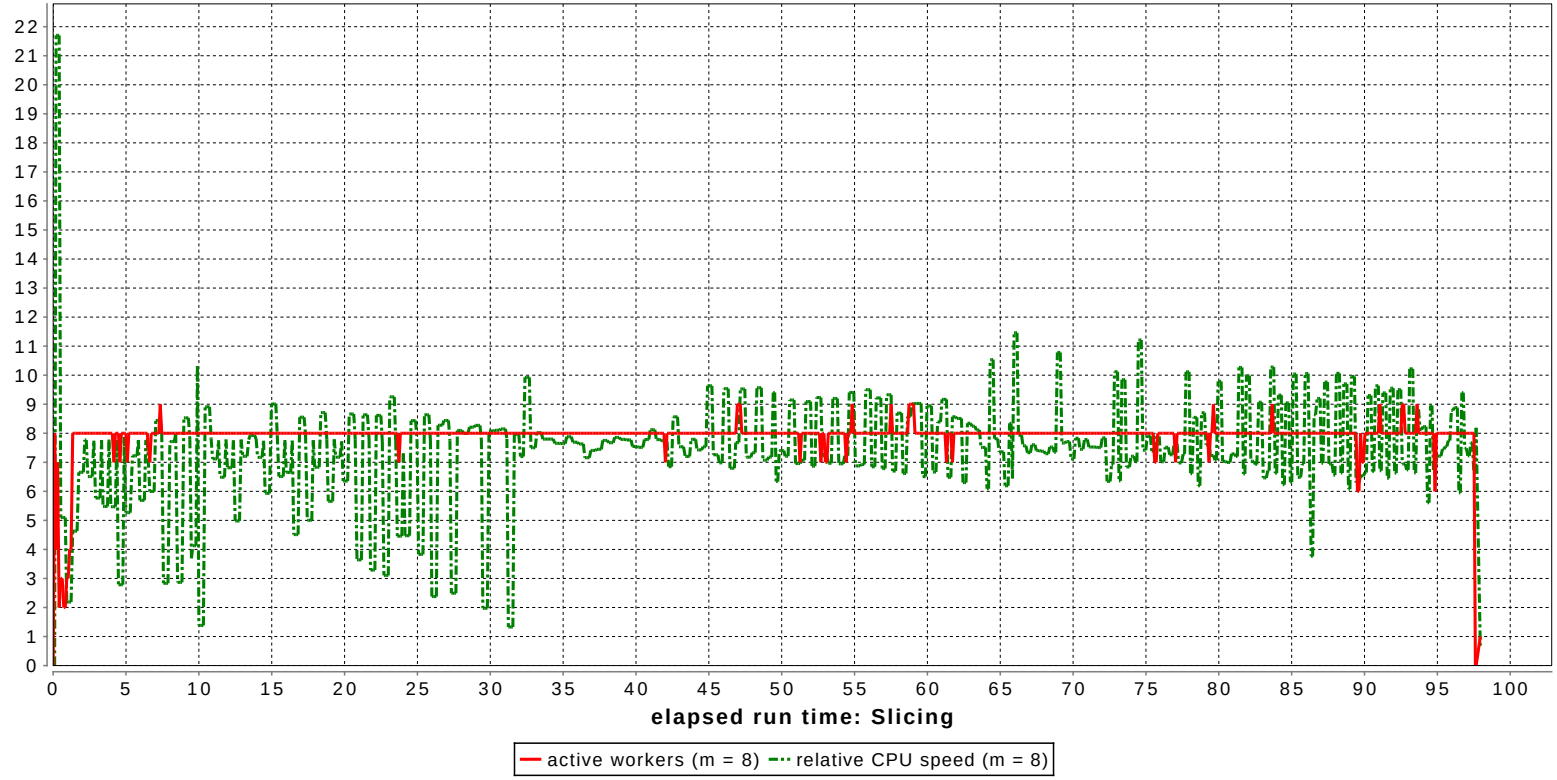
		$\varepsilon(1)$	$\varepsilon(8)$
A	*HOL*	210 s	85 s
B	*HOL-UNITY*	69 s	13 s
C	*HOL-Nominal_Examples*	526 s	91 s
D	*Slicing*	720 s	122 s
E	*HOL-Decision_Procs*	415 s	67 s
F	*HOL-Hoare_Parallel*	218 s	34 s

Task queue population and worker thread utilization





Saturation of threads vs CPUs (March 2013)



Best speedups so far

AFP/Slicing:

- factor 6.5: 8 threads on 8 cores
  February 2013 (official Isabelle2013)
- factor 9.5: 16 threads on 8 cores $\times$ hyperthreading,
  March 2013 with "prescient scheduling" of tasks (from last run)
- factor 12.5: 20 threads on 32 cores,
  April 2013 (measured by David Rager, Univ. of Texas)

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:

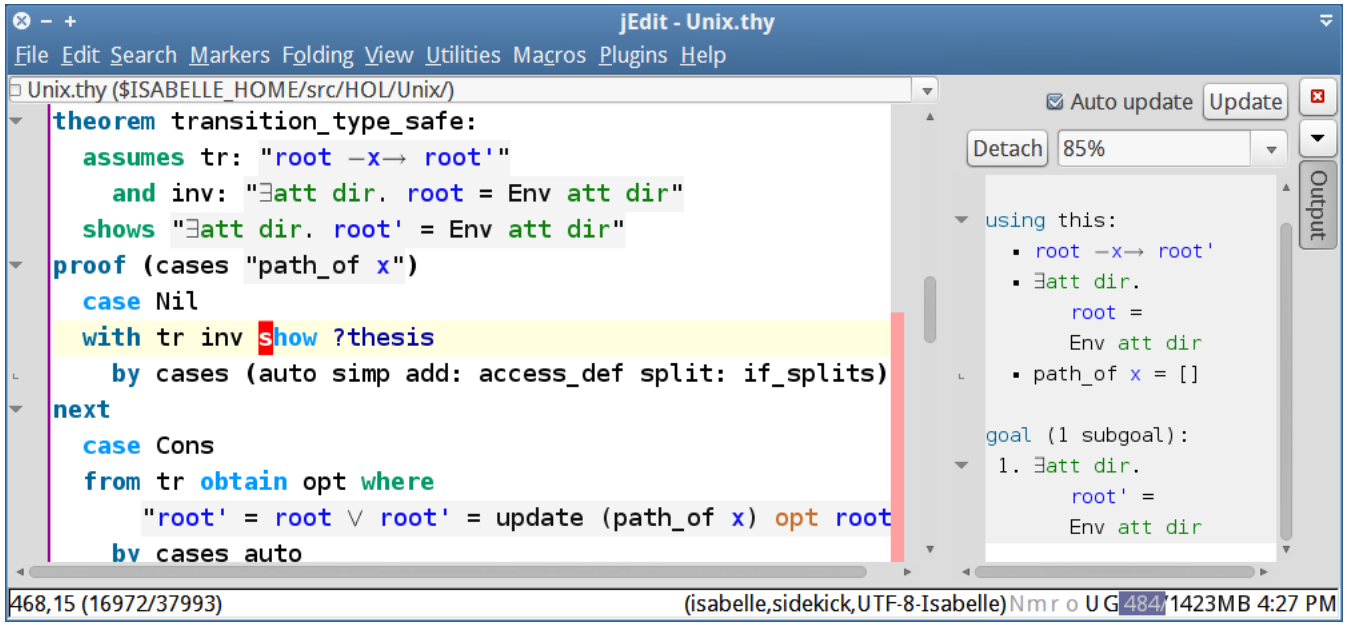
- cumulative PRINT operations consume more time than EVAL (output of goals is slower than most proof steps)
- PRINT depends on user perspective
- PRINT may diverge or fail
- PRINT augments results without changing proof state
- many different PRINTs may be run independently

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

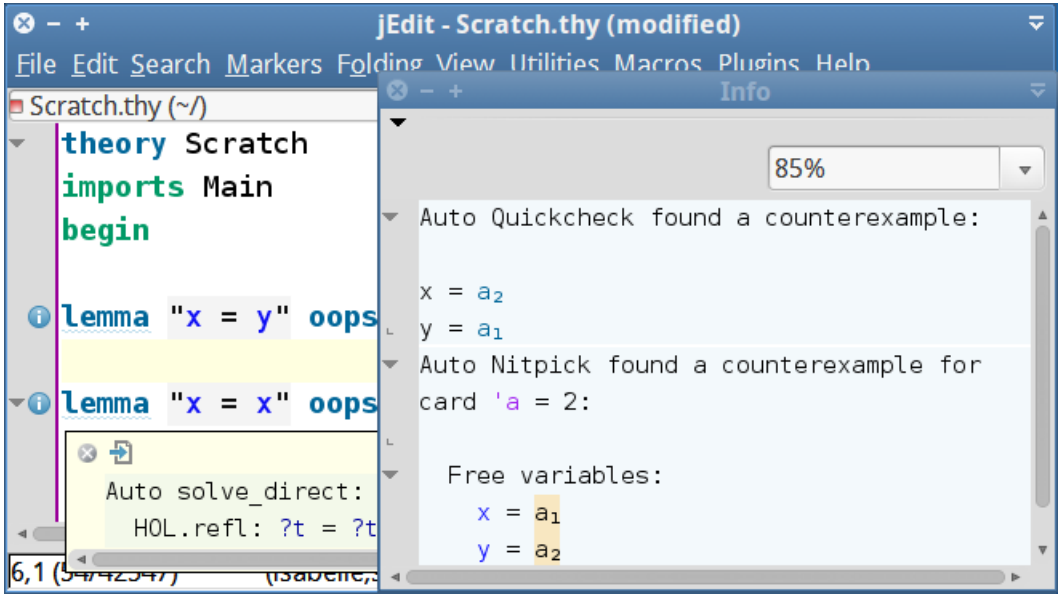
Application: print proof state

- 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



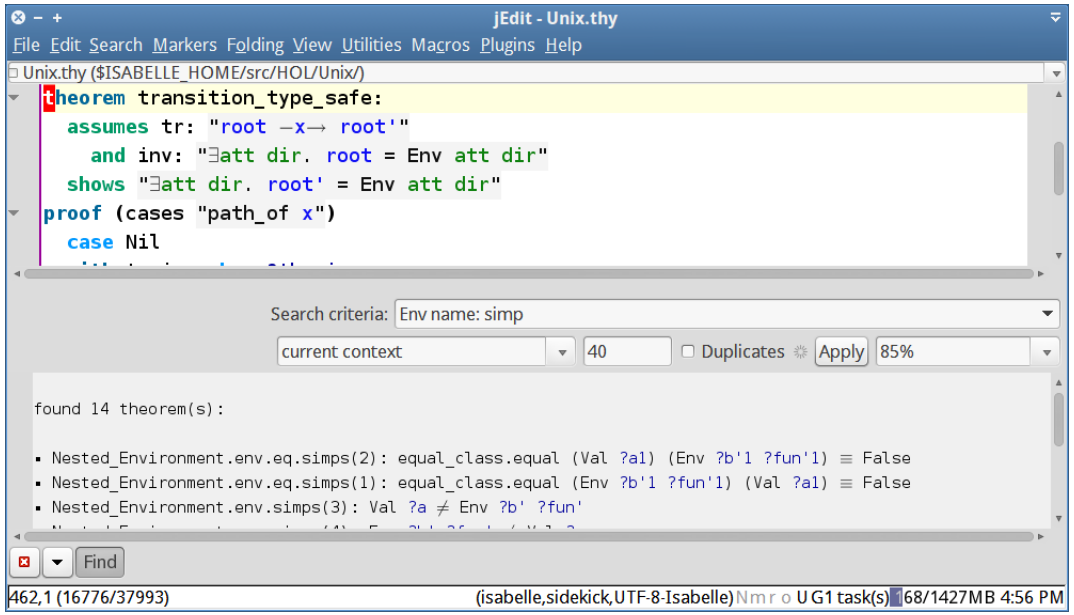
Application: automatically tried tools

- 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



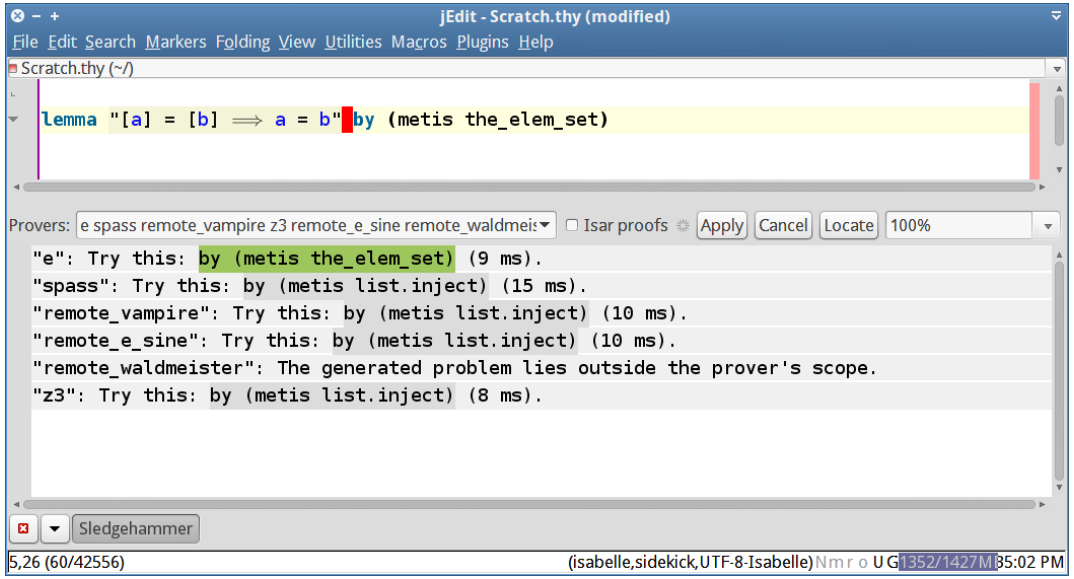
Application: query operations with user input

- 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.
- Try out Isabelle Prover IDE today!
  http://isabelle.in.tum.de
- Encourage your local Coq gurus to move forward!