Paral-ITP Front-End Technologies

and Isabelle Prover Architecture

Makarius Wenzel
Univ. Paris-Sud, LRI

October 2013

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

Overview

Papers

1. M. Wenzel: READ-EVAL-PRINT in Parallel and Asynchronous Proof-checking. User Interfaces for Theorem Provers (UITP 2012), postproceedings. EPTCS 118, 2013.
2. B. Barras, H. Herbelin, L. Gonzalez Huesca, Y. Regis-Gianas, E. Tassi, M. Wenzel, B. Wolff: Pervasive Parallelism in Highly-Trustable Interactive Theorem Proving Systems. Conference on Intelligent Computer Mathematics (CICM 2013), 2013. Springer LNCS 7961.
3. C. Lange, M. Caminati, M. Kerber, T. Mossakowski, C. Rowat, M. Wenzel, W. Windsteiger: A Qualitative Comparison of the Suitability of Four Theorem Provers for Basic Auction Theory. Conference on Intelligent Computer Mathematics (CICM 2013), 2013. Springer LNCS 7961.
4. M. Wenzel: Shared-Memory Multiprocessing for Interactive Theorem Proving. Interactive Theorem Proving (ITP 2013), 2013. Springer LNCS 7998.

Releases

Isabelle2013 (February 2013)

- (cf. meeting March 2013)

Isabelle2013-1 (November 2013)

- immutable kernel certificates
- monotonic document updates
- improved magamenent of executions (goal forks, prints)
- asynchronous print functions (e.g. auto sledgehammer)
- query operations (e.g. sledgehammer, find theorems)
- improved multiplatform support (Linux, Windows, Mac OS X)

Parallel Prover Architecture

Summary

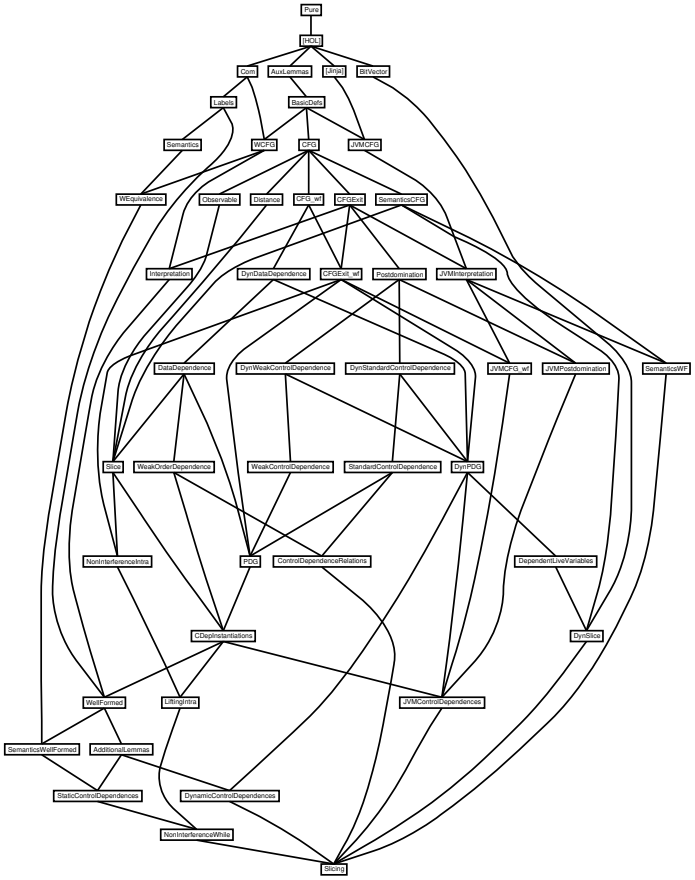
Change of Moore's Law: (since 2005)

- 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:

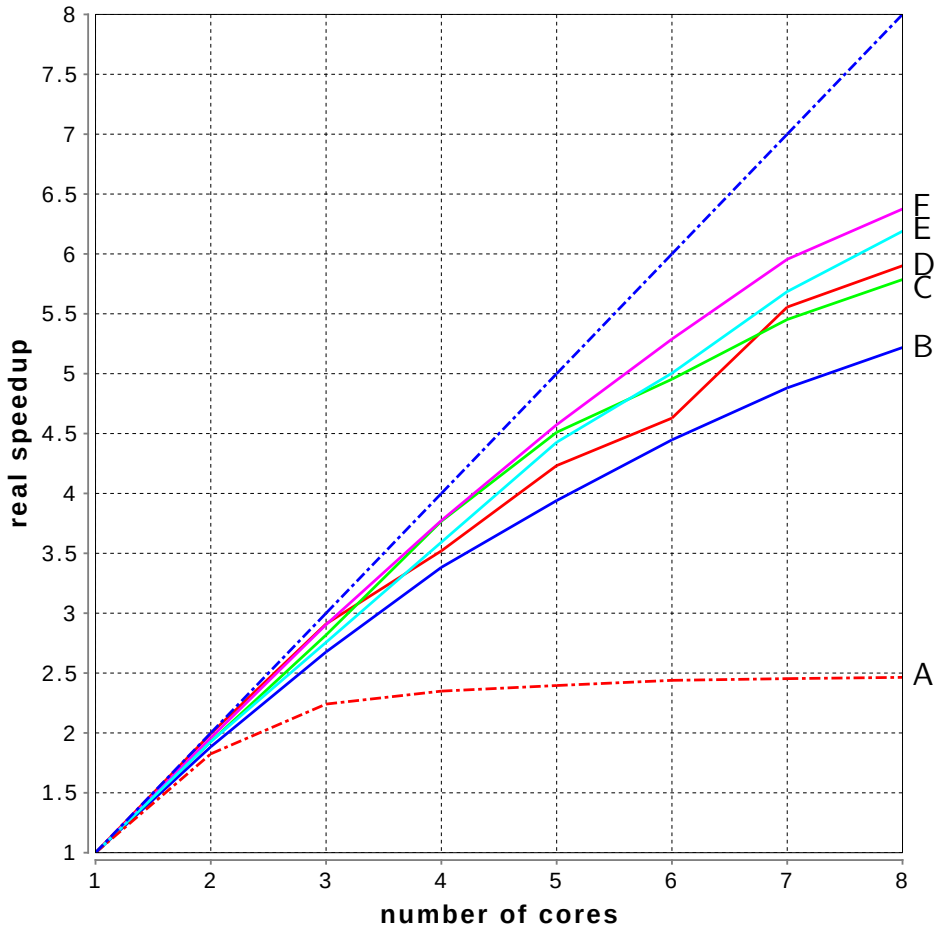
very naively $speedup = m$
before you make an implementation

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

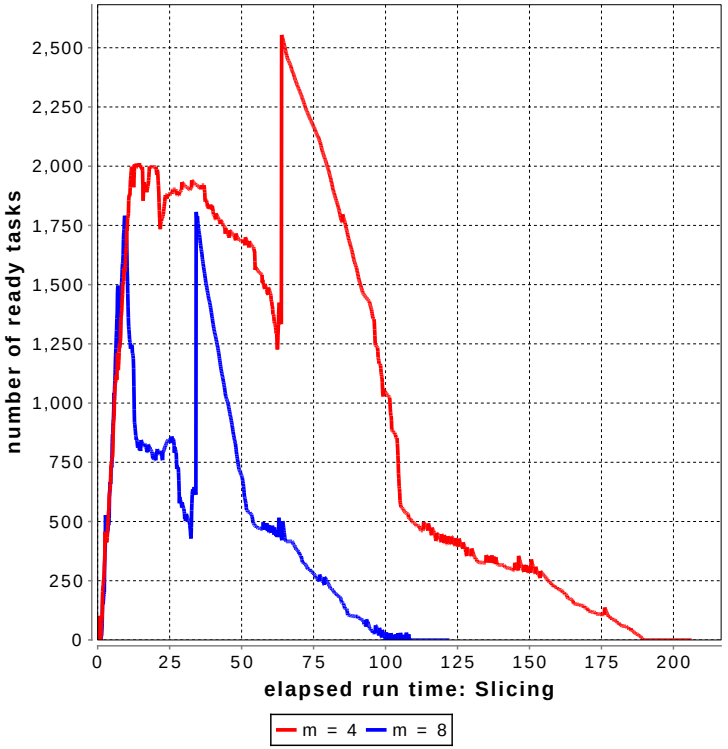
realistically $speedup \approx 0.66 \times m$, for reasonable $m = 4, 8, 16, \ldots$

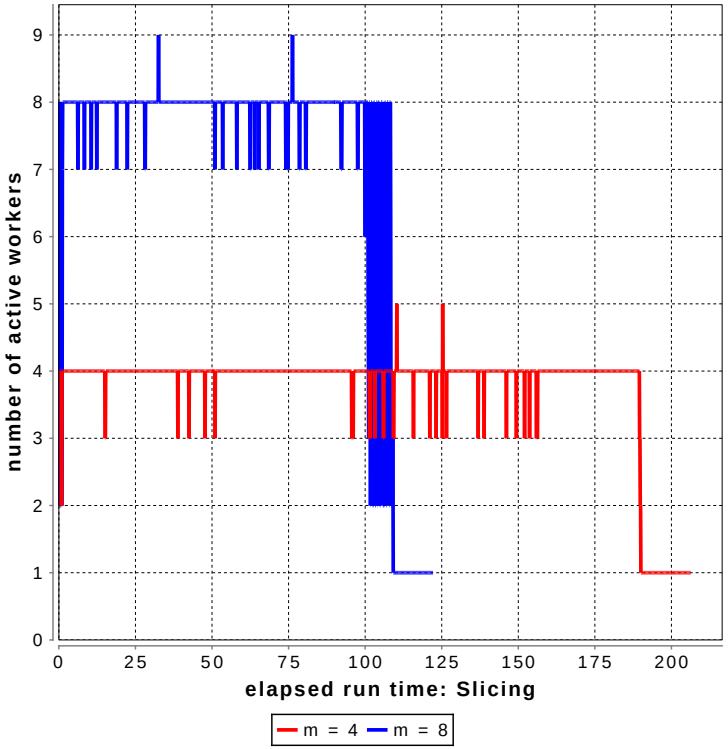
Isabelle2013 (February 2013) on 8 cores



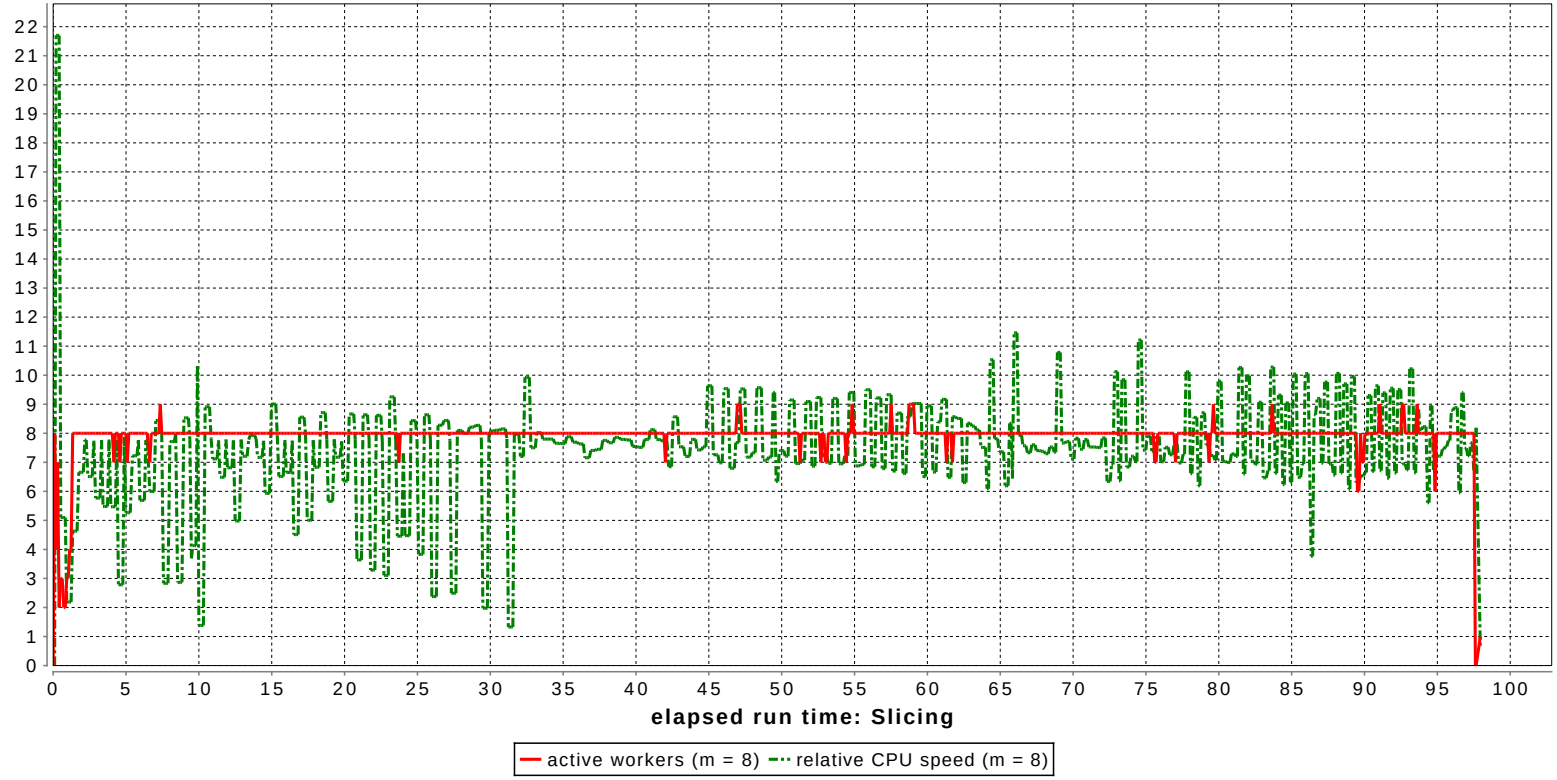
		$\varepsilon(1)$	$\varepsilon(8)$
A	*HOL*	210 s	85 s
B	*HOL-UNITY*	69 s	13 s
C	*HOL-Nominal_Examples*	526 s	91 s
D	*Slicing*	720 s	122 s
E	*HOL-Decision_Procs*	415 s	67 s
F	*HOL-Hoare_Parallel*	218 s	34 s

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

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:

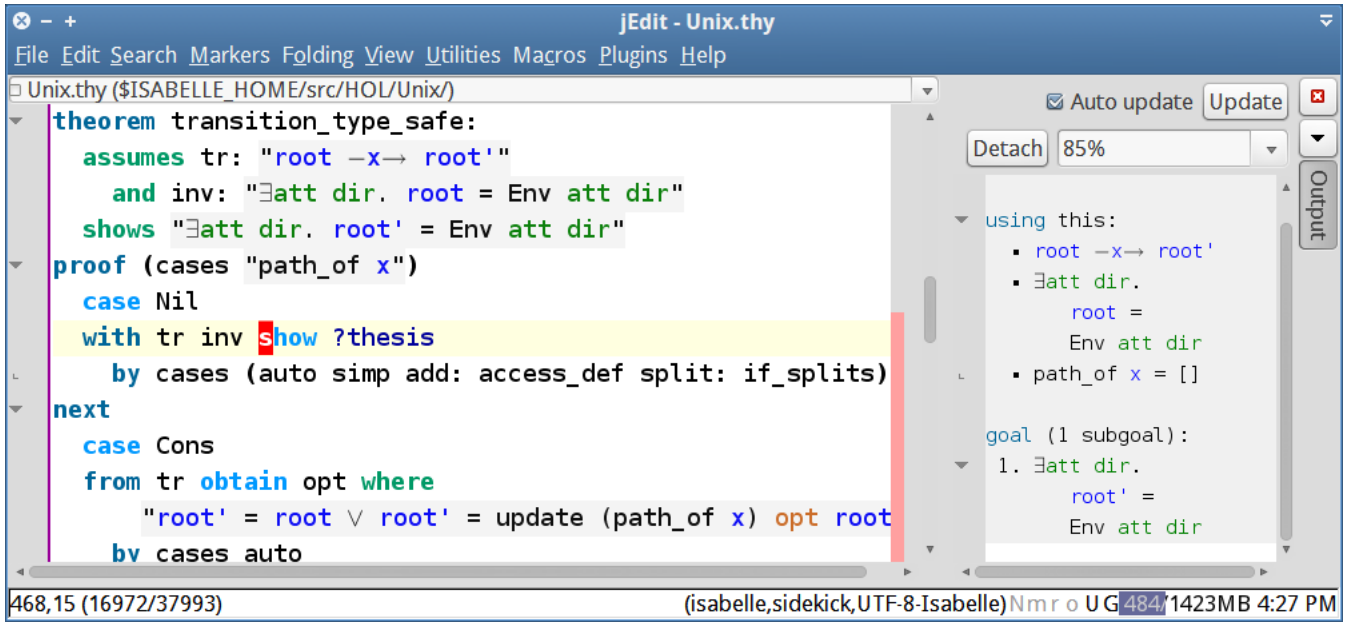
- cumulative PRINT operations consume more time than EVAL (output of goals is slower than most proof steps)
- PRINT depends on user perspective
- PRINT may diverge or fail
- PRINT augments results without changing proof state
- many different PRINTs may be run independently

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

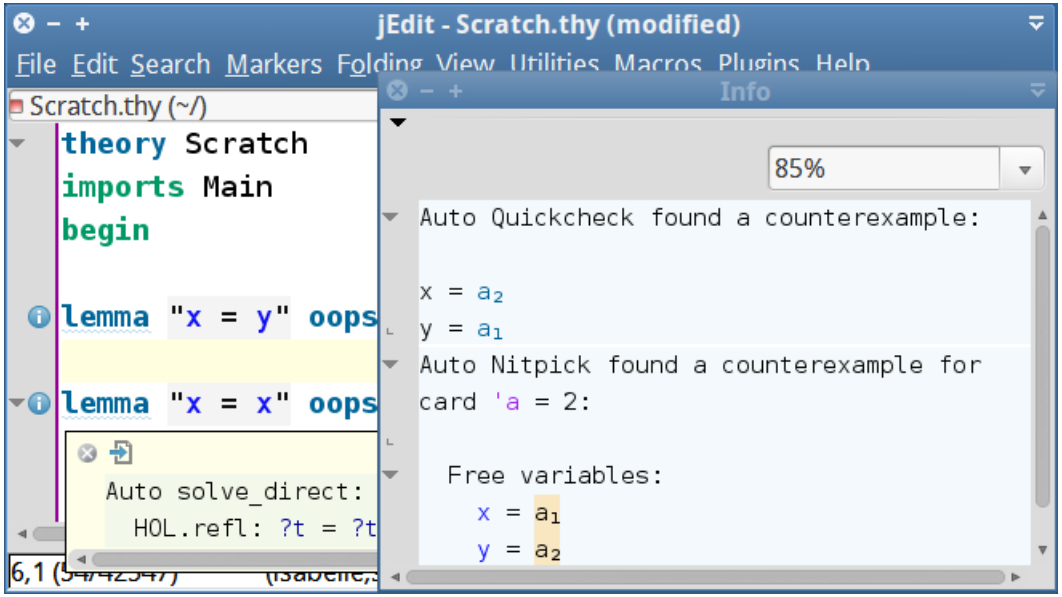
Application: print proof state

- 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



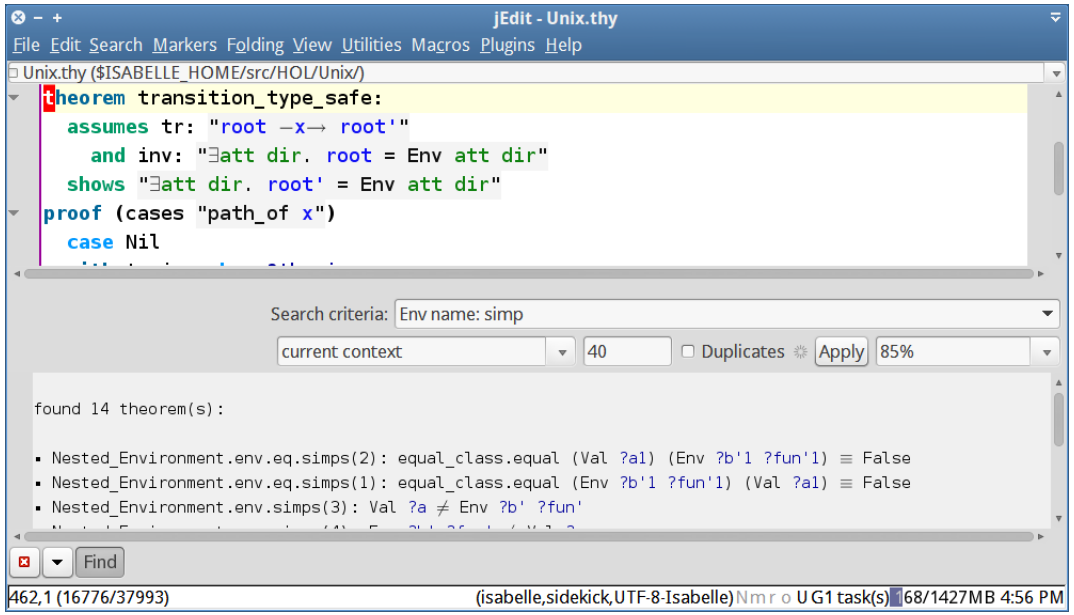
Application: automatically tried tools

- 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