READ-EVAL-PRINT in Parallel and Asynchronous Proof-checking

Makarius Wenzel
Univ. Paris-Sud, LRI

July 2012

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

Introduction

Motivation

Renovation of LCF-style theorem proving:

- overcome sequential-synchronous TTY mode
- overcome Emacs and Emacs LISP
- extend parallel Isabelle/ML into the Scala/JVM world
- facilitate integration of ITP systems into plain editors
  $\longrightarrow$ Isabelle/jEdit as Prover IDE

Transfer of concepts and implementation:

- help other provers to join the movement
- encourage front-end applications within Prover IDE

Example

`$ISABELLE_HOME/src/HOL/Unix/Unix.thy`

Observations:

- continuous editing can have non-local effects on prover commands
- special care is required for parallel evaluation, with local forks and global joins
  (not shown here; disabled in current production version Isabelle2012)

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

$\rightarrow$ Proof General sometimes needs manual "repair" of protocol

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 $(print,\ st') = eval\ st$ **in**	— main transaction
let $() = print\ ()$ **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

Version history:

- "big" document state $state$: $Document.state$
- each version contains full document structure + evaluation state
- document operations update versions in purely functional manner

$Document.init$: $Document.state$
$Document.update$: $version\text{-}id \to version\text{-}id \to edit^* \to Document.state \to Document.state$
$Document.remove\text{-}versions$: $version\text{-}id^* \to Document.state \to Document.state$

$\to$ main protocol operations on document-oriented proof processing

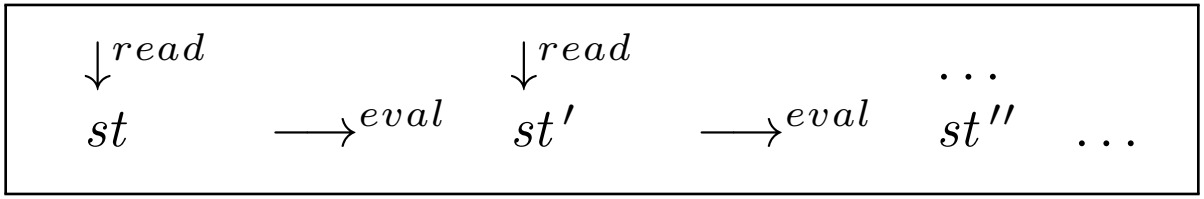
READ-EVAL-PRINT revisited

Prover Syntax: READ

Characteristics:

- many syntax layers ($\approx$ 12 in Isabelle)
- some syntax layers computationally complete (not really "syntax")
- solution for IDE document content: see [Wenzel, CICM 2012]
- solution for interaction model: restrict to outer syntax (command spans corresponding to transactions boundaries)

Scheduling: READ vs. remaining EVAL



- potentially parallel $read$ (independent of st)
- total $read$ (syntax errors postponed to $eval$)
- further tuning via command internalization before applying edits

Managed Evaluation: EVAL

Standard model of $SML + x - y$:

- strict functional evaluation, without global side-effects
- program exceptions
- physical exceptions (interrupts)
- potentially non-terminating, but interruptible

Portfolio of parallel Isabelle/ML:

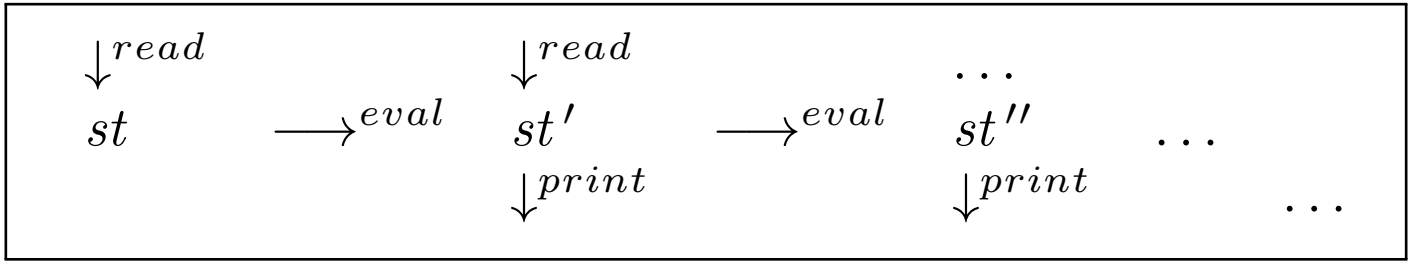
- type `'a future` — value-oriented parallelism; strict evaluation with exceptions and cancellation; groups and dependencies
- type `'a promise` — externally fulfilled futures
- type `'a lazy` — non-strict lazy evaluation conforming to parallelism and interrupts (non-persistent)
- type `'a memo` — low-level memo-cells for raw execution *with* persistent interrupts

Prover Output: PRINT

Characteristics:

- PRINT basically dual to READ
- PRINT often more time-consuming than EVAL
- assume total terminating PRINT, errors internalized into output
- all output decorated by command transaction id

Scheduling: READ vs. EVAL vs. PRINT



- potentially parallel $print$, e.g. lazy $\rightsquigarrow$ future via perspective
- not yet: potentially diverging and interruptible $print$ operations

$\rightarrow$ Asynchronous Agents Framework for Sledgehammer, Nitpick, etc.

Protocol Interpreter: breaking up LOOP

Main reforms:

- dedicated protocol thread (uninterruptible)
- private input and output stream, *not* stdin/stdout/stderr
- undirectional stream processing without synchronization (delayed flushing of output)
- protocol commands total, terminating, quick (max. 10–100 ms)
- representation of protocol data via YXML/XML/ML encoding (simple, robust, efficient)

Notes:

- prover commands are data of the protocol
- errors and interrupts are internalized — no POSIX signals between parallel processes

Conclusion

Lessons Learned

- READ and PRINT phases consume considerable runtime in practice and require extra organization
- EVAL opens a broad range of concepts for managed evaluation, independently of user-interfaces (initial motivation for the project)
- LOOP is a genuine artifact of TTY interaction, replaced by protocol interpreter threads in ML and Scala

- parallel evaluation + asynchronous interaction is very delicate (5 years until quite stable Isabelle2012, but still not finished)
- making proof-of-concept implementation for Coq is very easy (5 days for some experiment in June 2012, including full protocol stack, excluding READ-EVAL-PRINT management)

Further work

- generalize PRINT to Asynchronous Agents Framework
  (e.g. Sledgehammer, Nitpick, Quickcheck in Isabelle)
- full parallelism with local forks and global join as in Isabelle batch mode, including forked sub-proofs
- encourage other prover people to join the movement away from TTY loop