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

Makarius Wenzel *
Univ. Paris-Sud, Laboratoire LRI, UMR8623, Orsay, F-91405, France
CNRS, Orsay, F-91405, France

Abstract

The LCF tradition of interactive theorem proving, which was started by Milner in the 1970-ies, appears to be tied to the classic READ-EVAL-PRINT-LOOP of sequential and synchronous evaluation of prover commands. We break up this loop and retrofit the read-eval-print phases into a model of parallel and asynchronous proof processing. Thus we explain some key concepts behind the implementation of the Isabelle/Scala layer for prover interaction and integration, and the Isabelle/jEdit Prover IDE as front-end technology. We hope to open up the scientific discussion about non-trivial interaction models for ITP systems again, and help getting other old-school proof-assistants on a similar track.

1 Introduction

1.1 Motivation

Isabelle [12, §6] is one of the classic members of the LCF prover family, together with Coq [12, §4] and the variety of HOL systems [12, §1]. The survey on Isabelle [11] from 2008 provides some entry points to the diverse tools, packages, and applications of our prover platform. It has started as a Pure logical framework in 1989 and has grown into a general framework for integrating logic-based tools, including automated provers and dis-provers. The Isabelle2008 version also marks the turning point of substantial reforms in the organization of the proof process, such that it works efficiently on multi-core hardware, which is now common-place.

The original work on parallel Poly/ML and Isabelle/ML is reported in [5, 7]. The idea was to provide a parallel LCF-style inference kernel that supports a concept of *proof promises* natively, and to integrate it with the task-parallel library for *future values* in Isabelle/ML. The general principle behind this is *managed evaluation* in ML: some parts of the system take care to organize the execution of user code, similar to an operating system that organizes user processes. Managed evaluation includes external POSIX shell processes run from Isabelle/ML, which is used in Sledgehammer to run external provers implemented in C or to access the TPTP prover farm[1]. Recent extensions also bridge over to evaluations that are initiated in Isabelle/ML, but completed in Isabelle/Scala.

Soon after the initial success of parallel Isabelle it became clear that overly ambitious forking of proofs is in conflict with the received interaction model of the TTY loop, and its canonical front-end Proof General [2]. To illustrate this, we consider the following example:

inductive *path* **for** $R :: 'a \Rightarrow 'a \Rightarrow bool$ — implicit proofs: monotonicity and derived rules
where
base: $path\ R\ x_1\ x_1$
| *step*: $R\ x_1\ x_2 \Longrightarrow path\ R\ x_2\ x_3 \Longrightarrow path\ R\ x_1\ x_3$

theorem *example*: $path\ R\ x_1\ x_3 \Longrightarrow P\ x_1\ x_3$

*Current research supported by Digiteo Foundation and Project Paral-ITP (ANR-11-INSE-001).
[1] `http://www.cs.miami.edu/~tptp`

proof (*induct rule*: *path.induct*) — explicit toplevel proof
case (*base* x_1)
show $P\ x_1\ x_1$ ⟨*proof*⟩ — explicit sub-proof
next
case (*step* $x_1\ x_2\ x_3$)
note ‹$R\ x_1\ x_2$› **and** ‹*path* $R\ x_2\ x_3$›
moreover note ‹$P\ x_2\ x_3$›
ultimately show $P\ x_1\ x_3$ ⟨*proof*⟩ — explicit sub-proof
qed

This formal Isar text follows the basic structure of mathematical documents as a sequence of definition — statement — proof, but the (inductive) definition also involves some implicit proofs internally. Monotonicity of the specification is a prerequisite for further internal derivations of the introduction rules and induction principle (by the Knaster-Tarski fixed-point theorem).

The parallel batch mode of Isabelle (since 2008) forks all these proofs via the future evaluation mechanism, followed by a global join over the whole collection of proofs from all theories that are loaded into the session. This works, because proofs are not relevant for other proofs to proceed: it is sufficient to ensure that ultimately all proofs are finished.

This important principle of *proof irrelevance* holds only in a weaker sense for continuous editing and continuous checking in interactive mode. Some anomalies can occur if implicit proofs are forked blindly, because the TTY loop assumes that commands like **inductive** report synchronously about success or failure, before any other command is started. This limits the scope of parallelism to individual command transactions: all local proofs would have to be joined before committing to work on the next command.

Another bad effect is caused by user interrupts that interfere with parallel evaluation of commands. Implicitly forked proof attempts that are canceled — say by the user updating the source text, and thus causing some local re-evaluation — need to be *restarted*. Otherwise it could happen that the derivation of theorem *example* above might contain memo-ized interrupt exceptions in the justification for the *path.induct* rule.[2]

Apart from these problems of implicit proofs in seemingly atomic commands, parallel processing of explicit proofs given as separate command sequences in the text is even further removed from the received interaction model of step-wise proof scripting. The rich structure of proof texts — with its potential for forking validations of proofs and processing sub-proofs independently — is flattened according to depth-first traversal in the classic scripting model.

These observations should make sufficiently clear that the classic REPL concepts require substantial reforms, to make them fit for the combination of asynchronous interaction with parallel proof processing.

These investigations have already started in summer 2008, but it has required several years to get to reasonably robust implementations in Isabelle/Scala and Isabelle/jEdit. An early version is outlined in [8], the first stable release of Isabelle2011-1 (October 2011) is presented in [10]. In the current release of Isabelle2012 (May 2012) this infrastructure for continuous proof checking and Prover IDE support is consolidated further, but many of the underlying concepts still need to be communicated.

The present paper is a further step to explain the concepts of the Isabelle Prover IDE. This is also the reason why this is a user-interface paper without screenshots!

[2]The deeper problem is the non-monotonic behavior of future cancellation: a parallel evaluation that is not consolidated yet and cancelled cannot be continued afterwards. This does not happen in batch mode, because cancellation means to terminate the whole process (after printing all failures encountered so far.

1.2 Classic REPL Architecture

The classic READ-EVAL-PRINT-LOOP is well-known from long-standing LISP tradition. From there it made its way into many applications of symbolic computation, computer algebra, interactive theorem proving, etc. The basic idea is to process a sequence of *commands* one by one, and report results immediately to the user.

The division into the main phases of the loop can be explained in a first approximation from the perspective of the LISP interpreter, which processes a sequence of LISP expressions as toplevel declarations as follows.

READ: process the syntax of the given expression — *internalize* it into a semantic operation on the program state.

EVAL: evaluate the internalized expression in the current state — *run* it and update the toplevel state accordingly.

PRINT: output the result of the evaluation — *externalize* values, usually in the same notation as the input.

LOOP: continue the above *ad infinitum*, or until the user terminates the command interpreter.

The READ-EVAL-PRINT phases structure various interpreter phases, and the LOOP phase defines the interactive behavior of the system. The latter involves some technical details about organizing interaction that are often taken for granted in the folklore history of these concepts. Subsequently we attempt to recall some of this, and relate them to issues faced by classic REPL front-ends like Proof General [2] and refined versions of its protocols in PGIP [3].

Prompt. The system prints a command prompt and flushes the output channel to ensure the user can see it, and awaits input.[3]

Conceptually, the prompt behavior means full synchronization of the pair of input/output channels. This incurs certain real-time delays, say in local interprocess-communication to flush the buffers of the connecting pipe. For network connections the extra latency of a full round-trip needs to be taken into account. This does not prevent implementation of distributed editors on the World-Wide Web such as Etherpad `http://etherpad.com`, but the throughput of such synchronized interaction is limited by design.

Proof General uses the command prompt as the main protocol marker — the prover is required to decorate its prompt by special control sequences to make it work. This allows to separate command boundaries semantically: all observable output from the evaluation phase between two command prompts is attached to the corresponding command span in the source text. This natural observation of the TTY loop imposes some limitations on command evaluation strategies, though. It is difficult to detach asynchronous commands from the main loop — deferred output can confuse processing of other commands. The user needs to understand the meaning of displaced messages, and occasionally "repair" the protocol by issuing suitable control commands for re-synchronization of the editor with the prover.

Handling of errors. Any of the READ-EVAL-PRINT phases might fail, which results in some error output instead of regular PRINT. The LOOP needs to ensure that command transactions are atomic: the toplevel state is only updated after a successful run; errors should result in a clean *rollback* to the previous state. This means, a failing command transaction essentially

[3] Flushing is sometimes forgotten in implementations and only discovered when the system is run over a pipe for the first time, without the automatic per-line flush of the terminal stream on Unix.

results in an identity function on the state with some extra output, but it depends on details of the prover if it moves one step forward in the command execution, or not. This might affect command history navigation later on, say to *undo* steps.

Classic Proof General and especially PGIP attempt to formalize such notions of "success" or "error" of command transactions, such that both the editor and the prover always agree on it. This still poses problems in boundary cases, with debatable situations of *non-fatal* errors that look like a command failure, but are intended as a strong warning issued by a successful command. It also explains why developers of Isabelle proof tools had to be instructed to emit error messages only if a subsequent failure of the whole command could be guaranteed.

For robustness it is desirable to make the integrity of command transactions independent of accidental prover messages. This opens a spectrum of informative messages, warnings, non-fatal errors, fatal errors etc. without affecting critical aspects of the interaction protocol.

Handling of interrupts. The aim is to allow the user to intercept command execution, say by pressing CTRL-C or pushing some emergency brake button. The standard implementation makes the LOOP itself uninterruptible, but enables interrupts for executing each command (especially in the EVAL phase, which might be non-terminating). This assumes that the runtime environment that executes the command reacts accordingly and aborts the user program.

Even many decades after the introduction of hardware interrupts and process signals (at least on POSIX systems), interruptibility of arbitrary user-code cannot be taken for granted. Servicing of interrupt requests might be too slow (resulting in noticeable delays), or too fast (resulting in inconsistent internal program state). A LISP interpreter might have no problems to poll the interrupt status frequently, but more advanced language platforms need to invest further care to make it work reliably. Poly/ML (which underlies Isabelle/ML) is able to handle interrupts quickly in most practical situations, with well-defined meaning of signals within a multi-threaded process. External signals are dispatched to all threads that are configured to accept them, and internal signals are addressed to selected threads in isolation. The JVM (which underlies Isabelle/Scala) follows a similar model, but is more reluctant to let interrupts interfere with regular user code: `Thread.interrupt` is either serviced implicitly during I/O or needs to be explicitly polled via `Thread.interrupted`.

In any case, external interrupts raise further delicate questions about the integrity of command transactions. It depends on many implementation details if interrupted command transaction are properly rolled-back, or treated as successful without any effect. Adding the aspects of parallel and asynchronous execution makes things even more difficult to handle properly. For example, detached evaluations of older commands might receive a signal from the current command evaluation unintentionally, and thus leave the front-end (and the user) in an unclear situation concerning the state of the prover.

1.3 Command Transactions and Document Structure

Subsequently we introduce a minimal formal model of command transactions and proof document structure, in order to clarify further elaborations of the REP model, and various required extensions for asynchronous interaction and parallel processing. The bigger picture is given by a document-oriented approach to prover interaction. Its content-oriented aspects are explained in [9]. The corresponding interaction model provides first-class notions of document editing with some version management built-in, as sketched below. The idea is to embed "small" toplevel states into "big" document states, and provide some editing operations on that.[4]

[4]Strictly speaking, it is no longer appropriate to use the traditional term "toplevel state" for the many small system configurations that are managed here simultaneously within the big document state.

"Small" toplevel state (isolated commands). The local program configuration that is managed by the toplevel is represented as explicit value *st*. A *command transaction* is essentially a partial function on a toplevel state: we write $st \longrightarrow^{tr} st'$ as relation, or $st' = tr\ st$ as partial function application. The transaction can be internally structured according to the classic READ-EVAL-PRINT phases. As first approximation $tr = read; eval; print$ is merely the sequential composition of certain internal operations.

The original motivation for this sub-structuring was given by the LISP interpreter, with its *intern*−*run*−*extern* phases, but our main purpose is to organize incremental checking of proof documents. So we characterize the three phases by their relation to the toplevel state:

$tr\ st =$
  let $x = read\ src$ **in**
  let $(y,\ st') = eval\ x\ st$ **in**
  let $() = print\ st'\ y$ **in** st'

This means *read* is a prefix of the command transition that does not depend on input state, and *print* a suffix that does not change the output state. Only the core *eval* operation may operate on the semantic state arbitrarily. The *src* input is essentially a parameter of the command transaction, i.e. the concrete command span given in the text.

In reality there might be syntax phases that do require access to the state, but they can be conceptually included in the inner *eval* function.

Document structure. The overall document structure has two main dimensions: *local body* of text as sequence of commands and *global outline* as directed-acyclic graph (DAG). The nodes of this graph may be understood as "modules", which are called "theories" in Isabelle, "vernacular files" in Coq, and "articles" in Mizar.

In some sense this structuring of command transitions is accidental, but motivated by the typical situation in proof assistants: sequences of commands that are evaluated left-to-right and are organized in strictly foundational order of the theory graph. Cyclic module structure is not permitted, in contrast to programming languages like Haskell or Java. Semantically, we can linearize the DAG by producing a canonical walk-through, which means a proof document can be considered (w.l.o.g.) as *locally sequential* as follows:

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

Thus we can ignore the outer DAG structure in many theoretical considerations. Although, the module graph is an important starting point to organize the execution process efficiently. More ambitious re-organization would take the inherent structure of the command sequence into account, as introduced for parallel batch proof-checking in [7, 5].

Our reformed view on READ-EVAL-PRINT shall admit such non-trivial scheduling by the prover in interaction, while retaining a sequential reading of the text and its results that are presented to the user in the editor front-end.

"Big" document state (version history). A single document consists of a certain composition of command transactions as described above. Document *edits* can re-arrange the structure by inserting or removing intervals of command spans. This results in different document *versions* that are related by a certain *history* of edits. Each document version is implicitly associated with an *execution* process that evaluates its content according to the original sequential reading of the text, but implements a certain evaluation strategy on its mathematical meaning, to make good use of the physical resources of the machine.

The global *Document.state* covers all these aspects, by providing a few operations:

Document.init: *Document.state*
Document.update: *version-id* → *version-id* → *edit** → *Document.state* → *Document.state*
Document.remove-versions: *version-id** → *Document.state* → *Document.state*

This means *Document.update* v_1 v_2 *edits* updates the global document state by turning old v_1 into new v_2, applying the given edits on the command structure. The result is registered with the global state. This declarative update on the structure leads to certain modifications of the implicit execution process that is associated with the new version, re-using the partial execution state of the old one. The prover determines the details according to the semantics of the document content; the protocol refrains from speaking about that.

Additional fine-points *Document.update* are determined by the structure of *edit*, which is a concrete datatype with variants to insert or remove command spans from the text, or to indicate node dependencies in the DAG of modules, or to declare the so-called *perspective* of the front-end on the document structure. The latter represents the *visible* parts of the document and thus provides important hints to assign priorities to the incremental evaluation process: compared to a large hidden part of imported theory library and the potentially large unprocessed part of still pending text, the active area seen in the perspective is relatively small. This locality property helps to make document change management reactive and scalable.

The physical text editor is connected to the document model by classic GUI event handlers. Thus various elementary editor options will eventually become a sequence of document edits that are pipe-lined towards the prover: insert or remove text, open or closing windows, scroll within open windows etc. The granularity of document versions is determined implicitly via certain real-time delays (in the range of 50–500 ms), such that edits are grouped and not every single keystroke will be passed through the protocol layer.

Document.remove-versions informs the prover that the editor is no longer interested in certain parts of the history; this amounts to de-allocation of resources in the document model. In practice it is sufficient to keep a short prefix of the editing history alive, one that is sufficient to cover the distance of physical editor buffer from a few document versions that are being processed in the pipeline towards the prover, and the actual execution process that is currently run by the prover. The current implementation prunes the history periodically every 60 s.

2 READ-EVAL-PRINT revisited

2.1 Prover Syntax (READ)

Prover syntax is a surprisingly difficult topic, especially in Isabelle with its many layers, several of them with computationally complete mechanisms to operate on user input: syntax translations, type-reconstruction in multiple stages, etc. A general approach to reform LCF-style provers to reveal some aspects of their internal semantic content is explained in [9].

For the present purpose of prover interaction, it is sufficient to consider the superficial command language, which is called *outer syntax* in Isabelle/Isar, and *vernacular* in Coq. This means we need to cover only the first two layers of Isabelle syntax, and ignore the other 10.

Historically, the Isar language was designed at the same time as early versions of Proof General, which explains some syntactic details of the language that allow a modest Emacs LISP program to discover so-called "command-spans" reliably in the text. Thus users need to write funny quotes around the "inner syntax" of the logical framework, but it leads to simple and robust separation of command boundaries. In contrast, Proof General for Coq involves a few more heuristics and approximations.

Despite such simplifications, the cumulative CPU resources for parsing command spans as the user is editing the text can approach the same order of magnitude as proof checking itself. In typical applications, only few proof commands consume significant evaluation time, but many commands require a certain overhead for the concrete syntax.

In §1.3 we have already isolated the *read* phase of the command transaction as a part that is independent of the semantic state. This means we can reorganize the command application sequence to perform all *read* phases independently, before starting to evaluate the composition:

$$\begin{array}{lllll} \downarrow^{read} & & \downarrow^{read} & & \ldots \\ st & \longrightarrow^{eval} & st' & \longrightarrow^{eval} & st'' \quad \ldots \end{array}$$

The *read* phase is required to be a total operation that terminates quickly. Syntax errors need to be encoded into the result, e.g. by producing error tokens, and postponing actual runtime exceptions to the *eval* phase that runs the internalized command text later on.

Nonetheless, the result of the preliminary *read* phase already contains useful information about the basic structure of the text, such as keywords and quoted text ranges that may be reported back to the front-end to produce some syntax-highlighting, based on authentic information from the prover, not the typical approximations as regular-expressions in the editor.

The diagram above admits at least two further re-organizations to improve performance:

Internalization of results of each *read* of the command source, such that it can be referenced later by some symbolic *id* (notably in operations of the document model). To achieve this we provide an auxiliary operation on the "big" document state:

Document.define-command: *id* → *string* → *string* → *Document.state* → *Document.state*

Document.define-command id name src registers some command *src* text for further use via *id*. The *name* is an aspect of the parsed content that has already been discovered by approximative parsing on the editor side; it helps the prover to organize document processing before commencing the actual *read* phase.

In Isabelle2012 *read* merely means to scan Isabelle "symbols" (ASCII + UTF8 text characters + infinitely many named mathematical symbols like `\<forall>`), and to tokenize according to outer syntax keyword tables and some fixed formats for identifiers and quoted text ranges.

Until Isabelle2011-1, full outer syntax parsing used to be part of the *read* phase, but it is now moved into *eval*. Thus we can support extension of the command language within Isabelle theories smoothly: for the first time of the history of Isabelle, the system does not depend on external keyword tables generated in batch mode, and commands can be used in the same theory body where they were defined. This detail is particularly important for developers of derivative tools in the Isabelle framework, who introduce their own commands in user-space.

Parallelization of the *read* phases, which neither depend on the toplevel state nor on each other. The parsing involved in *Document.define-command* could be forked as future task, and joined only before command evaluation starts.

This simple parallel parsing scheme used to be present in Isabelle2011-1, but was replaced by more modest lazy evaluation in Isabelle2012 in the course of some fine-tuning for 2–4 core machines. The reduced *read* phase no longer justified the (small) overhead for fork/join in the preparatory stage of command transaction. It might become relevant again when the system is optimized for hardware with 8–16 cores, where every tiny potential for parallelism needs to be exploited to make use of the available CPU resources.

2.2 Managed Evaluation (EVAL)

Our standard model for evaluation of user code is that of Standard ML with a few restrictions and extensions. This covers the following in particular:

- strict functional evaluation, without global side-effects; (program state is managed by the value-oriented *context data* concept of the Isabelle framework);
- program exceptions according to Standard ML, to indicate non-local exits from functional programs;
- physical exceptions as intrusion of the environment into the program execution (mapped to the special *Interrupt* exception);
- potential non-terminating, but interruptible execution;
- I/O via official Isabelle/ML channels for *writeln*, *warning*, *tracing* messages etc. or via private temporarily files as input to private external processes (this emulates value-oriented behavior on the file-system).

The Isabelle system infrastructure uses a variety of standard implementation techniques to define an explicit transaction context for user commands. This includes message channels that are explicitly tagged with an execution identifier, to attach output to the proper place, despite physical re-ordering in the parallel execution environment.

Unlike a real operating system that can use hardware mechanisms to enforce integrity of user processes, Isabelle/ML requires user commands to be well-behaved in the above sense. For example, output on raw `TextIO.stdOut` from the Standard ML Basis Library results in a side-effect on that process channel that cannot be retracted by the transaction management of Isabelle (this does not cause any further harm than user confusion about where some raw output is coming or going). In contrast, `writeln` from the Isabelle/ML library attaches a message to the dedicated output stream of the running transaction; it will be located wrt. the original command span in the source text (within a certain document version), and disappear if the transaction is reset or discontinued due to document updates.

Implementation Notes. The Isabelle/ML infrastructure to manage evaluation of user code has emerged over the last five years. Some of the main concepts are as follows.

- *Unevaluated expressions* are represented by existing means of ML, either as unit abstraction `fn () => a` of type `unit -> 'a` or as regular function `fn a => b` of type `'a -> 'b`. There are special combinators (variants of function application) that define a certain "runtime mode" for evaluation. For example, the combinators `uninterruptible` and `interruptible` indicate that an ML expression is run with certain thread attributes.
- *Reified results* as explicit ML datatype that represents the disjoint sum of regular values or exceptional situations:

  ```
  datatype 'a result = Res of 'a | Exn of exn
  val capture: ('a -> 'b) -> 'a -> 'b result
  val release: 'a result -> 'a
  ```

  The corresponding few lines of Isabelle/ML library greatly help to organize evaluation of user code. There are additional means to distinguish regular program exceptions from environmental effects (interrupts).

Reified results are occasionally even communicated explicitly in the interaction protocol. For examples, a malformed Isabelle theory header in the editor buffer is already discovered as part of the organization of files in the front-end; it is passed through the protocol as `Exn` value and produces a runtime error when the corresponding command transaction is run by the prover. Thus we can formally hold up the requirement to make external syntax and protocol operations total, and postpone failures to the runtime environment within the prover.

- Functional wrappers for evaluation strategies, notably the following:
  - Type `'a future` represents value-oriented parallelism, with strict evaluation that is commenced eventually, unless the corresponding future task group is canceled. Regular results and program exceptions are memo-ized; environmental exceptions lead to an explicitly "canceled" state of the future from which it cannot recover. Future *task* identifiers help to organize dependencies within the implicit queue, and hierarchic *group* identifiers allow to define the propagation of exceptions and interrupts between peers and subgroup members.

    This task-parallel concept of Isabelle/ML is used to implement a small library of parallel list operations, with more conventional combinators like `map`, `exists`, etc. as closed expressions with full joining of results.
  - Type `'a promise` is a variant of `'a future` that lacks the built-in policies of parallel evaluation of closed expressions. Instead, there is merely a synchronized single-assignment cell that is associated with a pro-forma future task, so that other future tasks can depend on it. A promise can be fulfilled by external means, and thus cause other future evaluations to be commenced. This admits a form of reactive parallel programming in Isabelle/ML: open promises define the minimal elements of a dependency graph, with outgoing edges of regular futures, and other futures depending on them. After all required promises are fulfilled, the parallel evaluation process starts to run, until completed or cancelled.
  - Type `'a lazy` represents expressions that are fully evaluated at most once, by an explicit force operation. Regular results and program exceptions are memo-ized, but not physical events. Interrupting an attempt to force a lazy value will cause an interrupt of the caller, and keep the lazy value in its unevaluated state.
  - `'a memo` is a synchronized single-assignment cell similar to `'a lazy`, but *with* memoization of interrupts. In user-code, accidental absorption of physical events would lead to anomalies, but here we use it to organize incremental evaluation of different document versions. After cancelling the current attempt to evaluate a document version, the system recovers from the partial result so far, and restarts any command transactions that have produced a result states with persistent interrupts.
- External evaluation via some GNU bash script, to invoke arbitrary POSIX processes from the ML runtime environment, with propagation of interrupts in both directions.
- Remote evaluation via an ML-Scala bridge, to invoke functions of type `String => String` on the JVM. Thus some ML worker thread temporarily transfers its runtime to a Scala counterpart.

Managed evaluation with different strategies is at the core of the Prover IDE concept. It turns out as more important for the user-experience than fancy GUI programming.

2.3 Prover Output (PRINT)

The PRINT phase is somehow dual to READ (§2.1). The original intention of the REPL model is to externalize the result of evaluation in a human-readable form, but this can mean many different things for proof assistants.

Printing may already happen during evaluation, as a trace of the ongoing execution. Conceptually, we decorate all prover messages with the *id* of the running transaction, in order to re-assemble the output stream in the proper order, relatively to the original source of the command span (or local positions within its source, say for warnings and errors that are directly attached to malformed parts).

Traditionally, the main result of an interactive proof step is the subsequent *proof state*, which is printed implicitly for proof commands. Since proof states often consist of large terms that require substantial time for printing (often more than the time for inferencing), it makes sense to organize the *print* phase by continuing our REP diagram in the obvious manner:

$$\begin{array}{ccccccc} \downarrow^{read} & & \downarrow^{read} & & \ldots & & \\ st & \longrightarrow^{eval} & st' & \longrightarrow^{eval} & st'' & \ldots & \\ & & \downarrow^{print} & & \downarrow^{print} & & \ldots \end{array}$$

This means after the last *eval* phase has finished, the system can fork the corresponding *print* and proceed with the next *eval*. So the main evaluation thread will plough through the sequence of commands and fork many parallel *print* jobs. Doing this naively easily saturates the future task queue with relatively insignificant jobs that print proof states of invisible parts of the document, while the user is working elsewhere.

This observation in earlier versions of our document model has motivated the explicit notion of *perspective*. Thus the visible parts of the text are explicitly declared by suitable document edit operations, based on information from the physical editor and its views on the text buffer. The *print* phase is initialized as a lazy expression, which is turned into an active future only if the perspective uncovers it. Afterwards it is guaranteed to finish, without any support to reset or cancel it. This is sufficient under the assumption that printing always terminates in reasonable time.

The above scheme integrates the traditional **pr** command of Isabelle into the document model in a reasonably efficient manner. Considering the potential for long-running or non-terminating *print* tasks not an accident, but a genuine concept to be supported eventually, we could generalize **pr** towards a large class of diagnostic commands over finished command evaluations. This would mean to piggy-back non-trivial analysis tools over prover commands, that analyze the situation and produce additional output for the user. The existing portfolio of Isabelle tools like **nitpick**, **quickcheck**, **sledgehammer** are examples for this.

The implementation in Isabelle2012 still lacks this generalization of the *print* phase towards arbitrary "asynchronous agents" that interact with the document content after evaluation. So far such functionality is simulated by inserting diagnostic commands into the document in the proper place, although it disrupts the evaluation of subsequent commands.

3 Protocol Interpreter

The classic REPL model makes a tight loop around the read-eval-print phases, to synchronize all phases immediately: emit a prompt and flush the output stream to re-synchronize with the input stream, and run the REP phases sequentially on the single main thread of the process.

In contrast, our protocol interpreter that implements the document-oriented model (§1.3) on the prover side works as follows.

- A dedicated *protocol input thread* is connected to a private input channel from where it reads *protocol commands*, and evaluates them immediately. This resembles some rudiments of the former REPL, but we merely do unidirectional stream processing, without re-synchronization by prompting the other side nor printing of results.

- Protocol commands are required to be *total*, i.e. must not raise any ML exceptions. Error conditions need to be internalized into the protocol as separate messages returned to the front-end eventually.[5]

- Protocol commands are required to terminate quickly, to keep the thread *reactive* (say within the range of 10–100 ms). Note that the user-perception on the reactivity of the combination of editor front-end and prover back-end needs to take a full round-trip of certain protocol phases into account that are not explained in the present paper.

- Interrupts are blocked in the protocol thread; all operations on the main document state happen in a runtime context that is protected from physical events. User events stemming from the editing process have already been *internalized* as protocol commands into the stream of edits. This also means that there is no longer any use of POSIX process signals, which are so hard to manage robustly and portably for multi-threaded processes. User code is aborted exclusively via internal signals between ML threads, say as a consequence of cancellation of some future group by the protocol thread.

To make the protocol thread work reliably and efficiently, it is important to understand that protocol commands are not regular user commands. The protocol defines a limited vocabulary of certain editing operations, which need to be applied in-place and reported to the front-end accordingly. Prover commands occur as *data* of such protocol commands, and are dispatched for independent evaluation on a separate thread farm of the future task library in Isabelle/ML.

Implementation Notes. Early versions of the protocol interpreter imitated the classic Isar command loop by using *stdin* and *stdout* with quite concrete syntax for protocol commands and response messages, essentially an extension of the existing prover language with add-on commands like **undo** or **redo** known from TTY mode. This was adequate for prototypes, but had some limitations in robustness and performance.

For example, user-code may interfere with the global *stdin*/*stdout* streams of the process and disrupt the protocol. Classic TTY and Proof General interaction is designed to tolerate this: the user can switch to the raw protocol buffer and recover from the confusion. Such user intervention is no longer feasible in system based on continuous streaming of document edits towards the prover, and results reported back from many transactions run in parallel.

In the current production version the input channel is a private stream that is exclusively available to the protocol thread. On Unix we use named pipes (raw throughput ≈ 500 MB/s) and on Windows the more portable TCP sockets (raw throughput ≈ 100 MB/s).[6] Sockets require some effort to make them work in ML, but the Scala side consists only of a few lines of code. In principle one could also run the protocol on a remote network connection (say via SSH tunneling), but the performance implications have not been explored yet.[7]

[5] Hard crashes of protocol commands are reported to a side-channel that is normally invisible to users.

[6] Interestingly, much of this performance is lost due to the recoding of UTF-8 ML characters versus UTF-16 JVM characters. This explains why Isabelle/Scala sometimes prefers byte vectors that are presented as `CharSequence`, instead of regular `String`.

[7] The protocol uses relatively high band-width, but can afford long latency. Current timeouts for flushing edits are in the comfortable range of 100–500 ms, so one could probably reduce that to take network latency into account. The protocol engine has been tested successfully with 1 ms delays for its local buffers.

Spurious output on raw *stdout*/*stderr* is captured as well, and shown in a special console on demand. Thus we handle tools gracefully that violate the official PRINT conventions (§2.3).

The protocol command syntax has been reduced to the bare minimum to maximize performance and robustness. Errors in the encoding of the protocol would lead to failures that are difficult to repair, so we strive to avoid them by keeping things simple.

Each command consists of a non-empty list of strings: name and arguments. This structure is represented by explicit length indications in the protocol header, so that the protocol interpreter can read precise chunks from the stream without extra parsing. Decoding of arguments is left to the each protocol command implementation.

There have been early attempts (inherited from PGIP) to use standard XML documents to carry protocol data, but it requires awkward maintenance of XML element names and XML attributes to accommodate the quasi-human-readable attitude and other complications of standard XML. Instead, we now use a dedicated library in Isabelle/ML and Isabelle/Scala that performs data encoding of typed ML values over untyped/unnamed XML trees, in the same manner as the ML compiler would do it for untyped bit-strings in memory. These raw XML trees are then transferred via YXML syntax [9, §2.3] in a robust way.

This ML/XML/YXML data exchange is both efficient and easy to use, without demanding extra infrastructure for cross-language meta-programming (ML vs. Scala). Runtime type-safety of such minimalistic marshalling of tuples, lists, algebraic datatypes etc. is ensured by close inspection of a few lines of combinator expressions, both in the ML and the Scala side of the protocol implementation. This works in practice, because these program modules are maintained together in the same code repository. The accidental data formats that are encoded on the byte stream between the Isabelle/ML and Isabelle/Scala process is private to the implementation. The public programming interface is defined by typed functions in ML or Scala, not the protocol messages themselves.

4 Conclusion

The issue of providing sophisticated user-interface support for sophisticated provers has been revisited many times over many years. Early efforts by [4] have eventually found their way into Proof General [2], which is still the de-facto standard. Its approach to wrap up the existing REPL of the prover has been continued by other projects like CoqIde [12, §4] or Matita [1].

The deeper reason for the success of the classic Proof General approach is its conservativity wrt. the prover interaction model. Any prover that provides a reasonable REPL with some formal markup and an **undo** command can participate.

Investigating possibilities beyond Proof General, Aspinall and others have already pointed out the need to reform provers themselves. This eventually lead to the PGIP protocol definition [3] and its proposed front-end PG Eclipse. The idea was to replace Emacs and Emacs LISP by industrial-strength Eclipse and Java. In retrospective, we see a variety of reasons why this approach did not become popular in the proof assistant community: it demands substantial efforts to implement and maintain PGIP in the prover side, and requires higher-order people to engage in profane Java. Moreover, we consider the interaction model of PGIP still too close to classic Proof General, so the returns for the investment to support it were not sufficient.

Our strategy to bridge the cultural gap between ML and the JVM is based on Scala [6]. After non-trivial reforms on the prover side, the current state of concepts and implementation of Isabelle/ML/Scala and Isabelle/jEdit as Prover IDE on top of it should have reached a state where other projects can join the effort, either on the back-end or front-end side. The ultimate

goal is a substantial renovation of the LCF-approach to interactive theorem proving, for coming decades of applications. There are ongoing discussions with some Coq experts to transfer some of the ideas presented here to their world.

Moreover, the Isabelle implementation is expected to improve further in the near future. The main conceptual omission is the management of diagnostic commands over proof documents (cf. the discussion of the PRINT phase). We intend to support a notion of "asynchronous agents" natively, which will allow to attach automated provers and disprovers provided by Sledgehammer and Quickcheck in Isabelle already. Such advanced modes of tool-assisted proof authoring needs to be worked out further and turned into practice.

References

[1] A. Asperti, C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli. User interaction with the Matita proof assistant. *Journal of Automated Reasoning*, 39(2), 2007.

[2] D. Aspinall. Proof General: A generic tool for proof development. In S. Graf and M. Schwartzbach, editors, *European Joint Conferences on Theory and Practice of Software (ETAPS)*, volume 1785 of *LNCS*. Springer, 2000.

[3] D. Aspinall, C. Lüth, and D. Winterstein. A framework for interactive proof. In M. Kauers, M. Kerber, R. Miner, and W. Windsteiger, editors, *Towards Mechanized Mathematical Assistants (CALCULEMUS and MKM 2007)*, volume 4573 of *LNAI*. Springer, 2007.

[4] Y. Bertot and L. Théry. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 25(7), 1998.

[5] D. C. J. Matthews and M. Wenzel. Efficient parallel programming in Poly/ML and Isabelle/ML. In *ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming (DAMP 2010), co-located with POPL*. ACM Press, January 2010.

[6] M. Odersky et al. An overview of the Scala programming language. Technical Report IC/2004/64, EPF Lausanne, 2004.

[7] M. Wenzel. Parallel proof checking in Isabelle/Isar. In G. Dos Reis and L. Théry, editors, *ACM SIGSAM 2009 International Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS)*. ACM Digital library, August 2009.

[8] M. Wenzel. Asynchronous proof processing with Isabelle/Scala and Isabelle/jEdit. In C. Sacerdoti Coen and D. Aspinall, editors, *User Interfaces for Theorem Provers (UITP 2010)*, ENTCS, July 2010. FLOC 2010 Satellite Workshop.

[9] M. Wenzel. Isabelle as document-oriented proof assistant. In J. H. Davenport, W. M. Farmer, F. Rabe, and J. Urban, editors, *Conference on Intelligent Computer Mathematics / Mathematical Knowledge Management (CICM/MKM 2011)*, volume 6824 of *LNAI*. Springer, 2011.

[10] M. Wenzel. Isabelle/jEdit — a Prover IDE within the PIDE framework. In J. Jeuring et al., editors, *Conference on Intelligent Computer Mathematics (CICM 2012)*, volume 7362 of *LNAI*. Springer, 2012.

[11] M. Wenzel, L. Paulson, and T. Nipkow. The Isabelle framework. In *Theorem Proving in Higher Order Logics (TPHOLs 2008)*, LNCS. Springer, 2008.

[12] F. Wiedijk, editor. *The Seventeen Provers of the World*, volume 3600 of *LNAI*. Springer, 2006.