Asynchronous Proof Processing with Isabelle/Scala and Isabelle/jEdit

Makarius Wenzel
Univ. Paris-Sud, LRI

July 2010
Introduction
Motivation

Aims:

- **U-ITP**: usable interactive theorem proving
  → Make our provers accessible to many more people out there.

- **TP-UI**: theorem provers for user interfaces
  → Make building front-ends for provers really easy.
Motivation

Aims:
• **U-ITP**: usable interactive theorem proving
  → Make our provers accessible to many more people out there.
• **TP-UI**: theorem provers for user interfaces
  → Make building front-ends for provers really easy.

Issues:
• Viability of editor framework? *Emacs*?
• Viability of interaction model? *Read-eval-print loop*?
Beyond Proof General?

Implementations of “Proof General”:
- 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: based on large IDE platform (Netbeans)
Beyond Proof General?

Implementations of “Proof General”:
- 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: based on large IDE platform (Netbeans)

Limitations:
1. editor framework: single-core or even single-threaded (except for JVM-based frameworks)
2. interaction model: synchronous command loop with undo
Parallel proof checking and asynchronous interaction
Isabelle/Isar proof document structure

theory C imports A B begin

inductive path for rel :: α ⇒ α ⇒ bool where
  base: path rel x x
| step: rel x y ⇒ path rel y z ⇒ path rel x z

theorem example: fixes x z :: α assumes path rel x z shows P x z
using assms
proof induct
  fix x show P x x ⟨proof⟩
next
  fix x y z assume rel x y and path rel y z
  moreover assume P y z
  ultimately show P x z ⟨proof⟩
qed

end
theory C imports A B begin

inductive path for rel :: α ⇒ α ⇒ bool where
  base: path rel x x
  | step: rel x y ⇒ path rel y z ⇒ path rel x z ⟨internal proof⟩

theorem example: fixes x z :: α assumes path rel x z shows P x z

  using assms
  proof induct
    fix x show P x x ⟨proof⟩
  next
    fix x y z assume rel x y and path rel y z
    moreover assume P y z
    ultimately show P x z ⟨proof⟩
  qed

end
Potential for parallelism

1. **DAG structure** of theory development graph: cf. GNU `make -j`
2. **toplevel AND/OR structure**: explicit statements, irrelevant proofs
3. **modularity of structured proofs**: practically requires Isar
4. **general parallelism in ML**: practically requires immutable data
Potential for parallelism

1. **DAG structure** of theory development graph: cf. GNU `make -j`
2. **toplevel AND/OR structure**: explicit statements, irrelevant proofs
3. **modularity of structured proofs**: practically requires Isar
4. **general parallelism in ML**: practically requires immutable data

**Notes:**
- all of this available in recent Isabelle2009-2 and Poly/ML 5.3.0
- max. speedup 3.0 for 4 cores, and 5.0 for 8 cores
- technical and conceptual correlation with asynchronous interaction
Asynchronous proof documents

Main ideas:

• replace Isabelle command loop by document model for direct editing

• manage persistent history of versions to decouple editor from prover
Asynchronous proof documents

Main ideas:
• replace Isabelle command loop by document model for direct editing
• manage persistent history of versions to decouple editor from prover

Primitives:
• begin-document and end-document bracketing
• static define-command
• dynamic edit-document (wrt. command spans)
• detailed source addressing for prover input/output
Asynchronous proof documents

Main ideas:
- replace Isabelle command loop by document model for direct editing
- manage persistent history of versions to decouple editor from prover

Primitives:
- begin-document and end-document bracketing
- static define-command
- dynamic edit-document (wrt. command spans)
- detailed source addressing for prover input/output

Note: “command prompt” finally abolished

Parallel proof checking and asynchronous interaction
Scala/JVM
JVM problems (Sun/Apple implementation)

- reasonably fast only after long startup time
- small stack/heap default size, determined at boot time
- no tail recursion for methods
- delicate semantics of object initialization; mutual scopes but sequential (strict) evaluation
- plain values (e.g. int) vs. objects (e.g. Integer) live in separate worlds — cannot have bignums that are unboxed for small values
- multi-platform is subject to subtle issues ("write once, debug everywhere")
- null (cf. Tony Hoare: Historically Bad Ideas: "Null References: The Billion Dollar Mistake")
Java problems (source language)

- very verbose, code inflation factor $\approx 2$–$10$
- outdated language design, inability of further evolution
- huge development tools (software Heavy Industry)
Java problems (source language)

- very verbose, code inflation factor $\approx 2$–$10$
- outdated language design, inability of further evolution
- huge development tools (software Heavy Industry)

**But:**
+ reasonably well-established on a broad range of platforms (Linux, Mac OS, Windows)
+ despite a lot of junk, some good frameworks are available (e.g. *jEdit* editor or *Jetty* web server)
+ Scala/JVM can use existing JVM libraries (without too much exposure to musty Java legacy)
Scala language concepts (Martin Odersky et al)

- **100% compatibility** with existing Java/JVM libraries — *asymmetric* upgrade path
- about as (in)efficient as Java
- fully **object-oriented** (unlike Java)
- **higher-order** functional concepts (like ML/Haskell)
- **algebraic datatypes** ("case classes") with usual constructor terms and **pattern matching** ("extractors")
- good standard **libraries**
  - tuples, lists, options, functions, partial functions
  - iterators and collections
  - actors (concurrency, interaction, parallel computation)
- **flexible syntax**, supporting a broad range of styles (e.g. deflated Java, or ML/Haskell style), or "domain-specific languages"
• very powerful static type-system:
  – parametric polymorphism (similar to ML)
  – subtyping (“OO” typing)
  – coercions (“conversions”, “views”)
  – auto-boxing
  – self types
  – existential types
  – higher-kindled parameters
  – type-inference

• incremental compiler (“toplevel loop”)
Isabelle/Scala for prover interaction
The connectivity problem

Problems:
- JVM: provers are better not implemented in Java
- JVM: even with Scala, the JVM is suboptimal for our purposes
- ML: lack of connectivity to GUI / Web frameworks etc.
- ML: even GTK/OCaml is a niche market
  (no serious editor frameworks; we depend on SML anyway)
Realistic assumption:
- Prover: SML (e.g. Isabelle)
- Editor: Java/JVM (e.g. jEdit)

Question: How to integrate the two worlds?
- separate processes: requires marshalling, serialization, protocols
- different implementation languages and programming paradigms
- different cultural backgrounds
Realistic assumption:
• Prover: SML (e.g. Isabelle)
• Editor: Java/JVM (e.g. jEdit)

Question: How to integrate the two worlds?
• separate processes: requires marshalling, serialization, protocols
• different implementation languages and programming paradigms
• different cultural backgrounds

Our answer: bridge the gap via Scala (by Martin Odersky, EPFL)
Isabelle/Scala architecture

Conceptual view:

Editor: JVM  Document model  Isabelle: SML
Isabelle/Scala architecture

Conceptual view:

Implementation view:

Isabelle/Scala for prover interaction
Isabelle/Scala library

- public API, private internal protocol
- integral part of Isabelle distribution
- imitate Isabelle/ML style
- duplicate few Isabelle/ML modules on the Scala side (e.g. pretty printing, outer syntax lexer and command parsers)
- reduce public standards to required functionality (e.g. YXML encoding for pure XML trees)
Example: markup trees

**Raw trees**: untyped, uninterpreted

```scala
sealed abstract class Tree
case class Elem(name: String, attributes: Attributes, body: List[Tree]) extends Tree
case class Text(content: String) extends Tree
```
Example: markup trees

**Raw trees:** untyped, uninterpreted

```scala
sealed abstract class Tree
case class Elem(name: String, attributes: Attributes, body: List[Tree]) extends Tree
case class Text(content: String) extends Tree
```

**Pretty markup:** typed views on certain tree nodes

- derived objects `Block`, `Break` with `apply` and `unapply` methods
- pattern matching on extractors, e.g. in our `pretty.scala`

```scala
def format(trees: List[XML.Tree], ...): Text =
  trees match {
    case Nil => text
    case Block(indent, body) :: ts => ...; format(ts1, blockin, after, btext)
    case Break(wd) :: ts =>
      if (...) format(ts, blockin, after, text.blanks(wd))
      else format(ts, blockin, after, text.newline.blanks(blockin.toInt))

    case XML.Elem(name, atts, body) :: ts => ...; format(ts1, blockin, after, btext1)
    case XML.Text(s) :: ts => ts => format(...)
  }
```
Basic Isabelle/Scala services

- platform abstractions (Linux, Mac OS, Windows/Cygwin)
- Isabelle symbols vs. Unicode (UTF-8)
- minimal AWT/Swing support, including Isabelle font
- XML trees and YXML encoding (simple and efficient)
- process management (prover, other tools)
- pretty printing and HTML rendering of prover output
- asynchronous document model (long story — to be continued)
Application: Isabelle/jEdit
Main characteristics:

- very powerful editor framework
- well-focused and well-written
- pure Java/Swing application for standard JVM 1.6
- easily extensible via plugins (officially in Java, we use Scala)
- worthy successor to Emacs
- general GUI metaphors similar to full-scale IDEs
Isabelle/jEdit

- “IDE” both for Isar and ML
- discontinues “locked region” of Proof General
- asynchronous proof processing
- sneak preview in Isabelle2009-2: run isabelle jedit
Conclusion
Stocktaking

Achievements:
• bridging the gap between ML and Java/JVM — thanks to Scala
• towards routine use of prover IDE technology
Stocktaking

Achievements:
• bridging the gap between ML and Java/JVM — thanks to Scala
• towards routine use of prover IDE technology

Lessons learned:
• LCF-style provers can be adapted to accommodate interfaces
• many seemingly marginal issues need to be addressed (process, encodings, fonts, rich text rendering)
• actual GUI programming rather marginal
• asynchronous proof processing mostly concerned about persistent history management (cf. Mercurial SCM)
Future work

**Scaling up:**
1. large buffers — now: up to 5–10 pages
2. multiple buffers — without locking
3. connectivity with actual SCM history (Mercurial)
4. distributed multi-author editing (Wiki?)
Future work

**Scaling up:**
1. large buffers — now: up to 5–10 pages
2. multiple buffers — without locking
3. connectivity with actual SCM history (Mercurial)
4. distributed multi-author editing (Wiki?)

**Exploiting semantic content from prover:**
- generic CSS rules for GUI metaphors
- formal cross references everywhere
- highlighting of scopes, renaming of bindings
- templates, proof skeletons etc.