Front-end Technologies for Formal-Methods Tools

Makarius Wenzel
Univ. Paris-Sud, Laboratoire LRI

November 2013

Abstract

Looking at the past decades of interactive (and automated) theorem proving, and tools that integrate both for program verification, we see a considerable technological gap. On the one hand there are sophisticated IDEs for mainstream languages (notably on the Java platform). On the other hand there are deep logical tools implemented in higher-order languages, but with very poor user-interfaces.

The PIDE (Prover IDE) approach combines both the JVM world and the ML world to support sophisticated document-oriented interaction, with semantic information provided by existing logical tools. The architecture is inherently bilingual: Scala is used to bridge the conceptual gap from ML-like languages (SML, OCaml, Haskell) to the JVM, where powerful editors or IDE frameworks already exist. Thus we can extend our tools to a wider world, without giving up good manners of higher-order strongly-typed programming.

Isabelle/jEdit is presently the main example of such a Prover IDE, see also `http://isabelle.in.tum.de` for the current release Isabelle2013-1 (November 2013). The general principles to enhance such formerly command-line tools to work with full-scale IDEs are explained by more basic examples: CoqIDE and Why3.

This demonstrates that classic logic-based tools can be reformed and we can hope to address more users eventually.

Introduction

Motivation

General aims:

- renovate and reform interactive (and automated) theorem proving for new generations of users
- catch up with technological changes: multicore hardware and non-sequentialism
- document-oriented user interaction
- mixed-platform tool integration

Side-conditions:

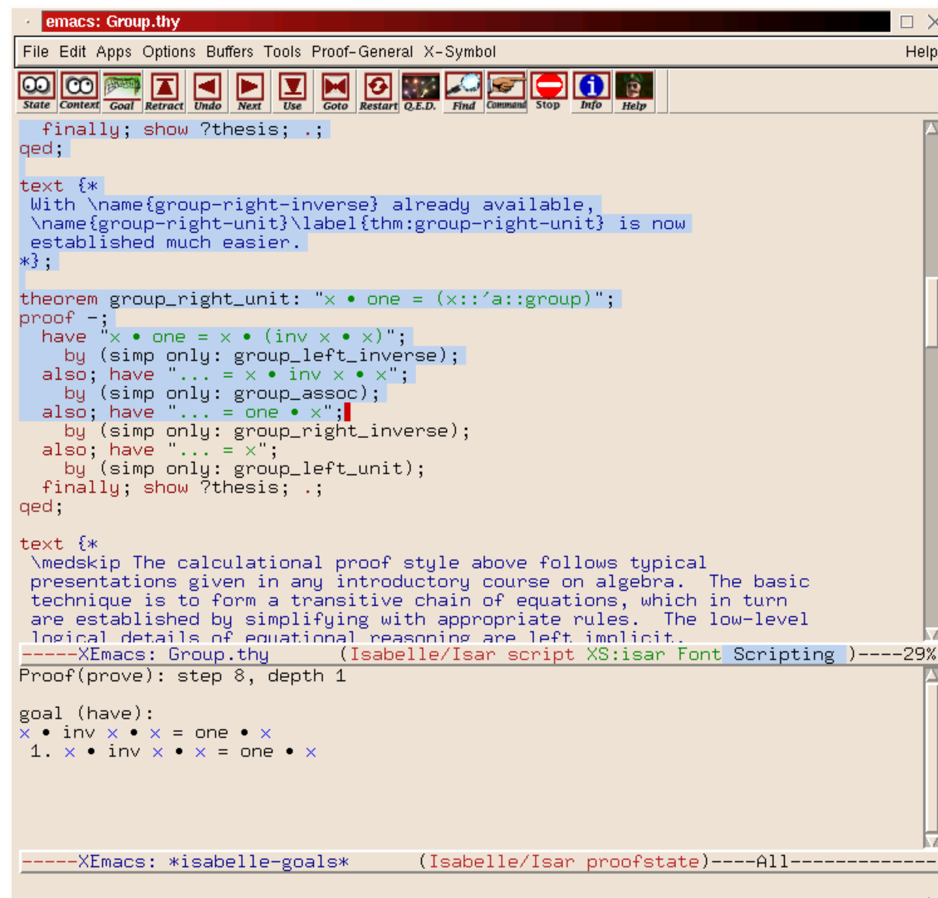
- routine support for Linux, Windows, Mac OS X
- integrated application: download and run
- no "installation"
- no "packaging"
- no "`./configure; make; make install`"

Example: Isabelle/jEdit Prover IDE

```
jEdit - Seq.thy
File Edit Search Markers Folding View Utilities Macros Plugins Help
Seq.thy ($ISABELLE_HOME/src/HOL/ex/)

header {* Finite sequences *}

theory Seq
imports Main
begin

datatype 'a seq = Empty | Seq 'a "'a seq"

fun conc :: "'a seq ⇒ 'a seq ⇒ 'a seq"
where
  "conc Empty ys = ys"
| "conc (Seq x xs) ys = Seq x (conc xs ys)"

fun reverse
where
  "reverse Empty = Empty"
| "reverse (Seq x xs) = conc (reverse xs) (Seq x Empty)"

    constant "Seq.seq.Seq"
    :: 'a ⇒ 'a seq ⇒ 'a seq

Auto update  Update  Detach  100%

Found termination order: "(λp. size (fst p)) <*mlex*> {}"

Console  Find  Output  Sledgehammer

isabelle
Filter:
Seq.thy
  Seq
    header {* Finite sequences *}
      theory Seq
      datatype 'a seq = Empty | S
      fun conc :: "'a seq ⇒ 'a seq
      fun reverse :: "'a seq ⇒ 'a s
      lemma conc_empty: "conc
      lemma conc_assoc: "conc (
      lemma reverse_conc: "reve
      lemma reverse_reverse: "re
      end
Documentation  Sidekick  Theories

14,6 (209/791)    (isabelle,sidekick,UTF-8-Isabelle) N m r o U G 750/1174MB 4:16 PM
```

Antiquated “IDEs”

Emacs Proof General



Characteristics:

- front-end for TTY loop
- sequential *proof scripting*
- one frontier between checked/unchecked text
- one proof state
- one response
- synchronous

CoqIDE

File Edit Navigation Try Tactics Templates Queries Display Compile Windows Help

Intro.v Examples.v

```
rewrite IHn.
reflexivity.
Qed.

Lemma nat_eq_dec : forall (n m : nat), {n = m} +
induction n.
 destruct m as [|m].
  left.
  reflexivity.
 right.
 discriminate.
destruct m as [|m].
 right; discriminate.
destruct (IHn m) as [Hm|Hm].
 left.
 rewrite Hm.
 reflexivity.
right.
intros Hnm.
apply Hm.
injection Hnm.
tauto.
Defined.

Eval compute in (nat_eq_dec 2 2).
Eval compute in (nat_eq_dec 2 1).

Definition pred (n:nat) : option nat :=
match n with
| 0 => None
```

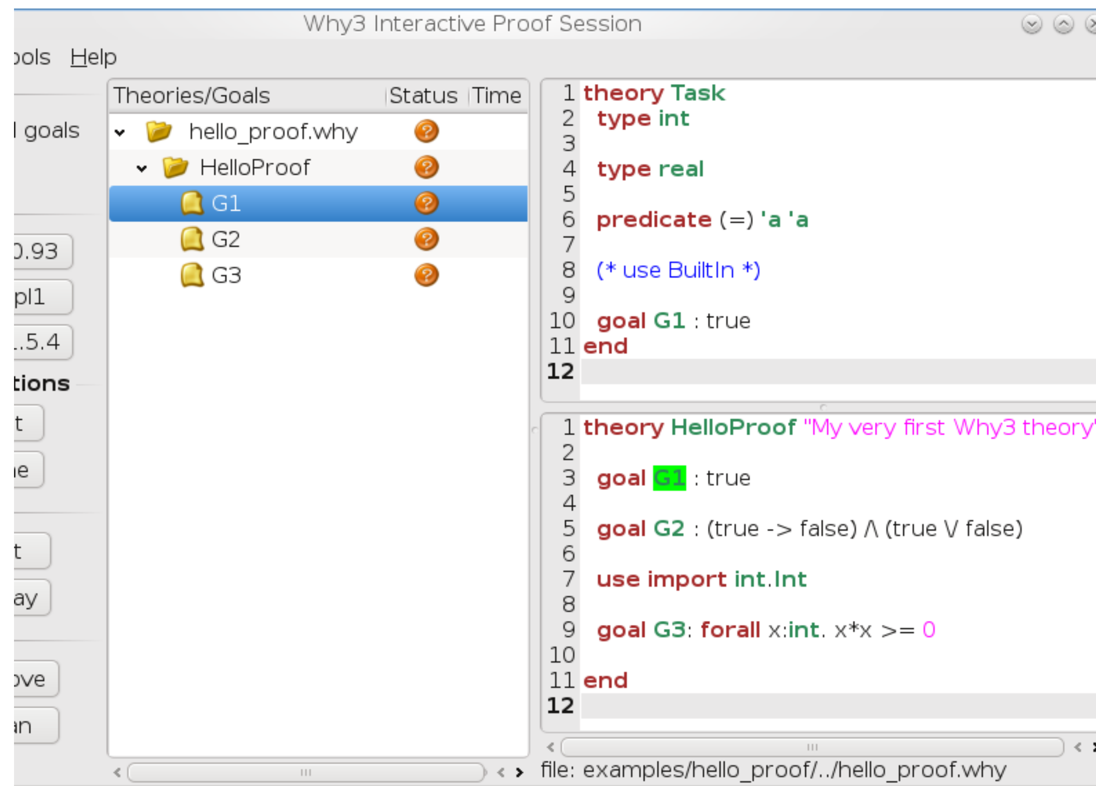
```
2 subgoals
n : nat
IHn : forall m : nat, {n = m} + {n <> m}
m : nat
Hm : n = m
______________________________________(1/2)
S m = S m
______________________________________(2/2)
{S n = S m} + {S n <> S m}
```

Ready in Predicate_Logic, proving nat_eq_dec   Line: 159 Char: 13   CoqIde started

Characteristics:

- clone of Proof General, without Emacs
- OCaml + old GTK
- lacks proper editor

Why3 IDE

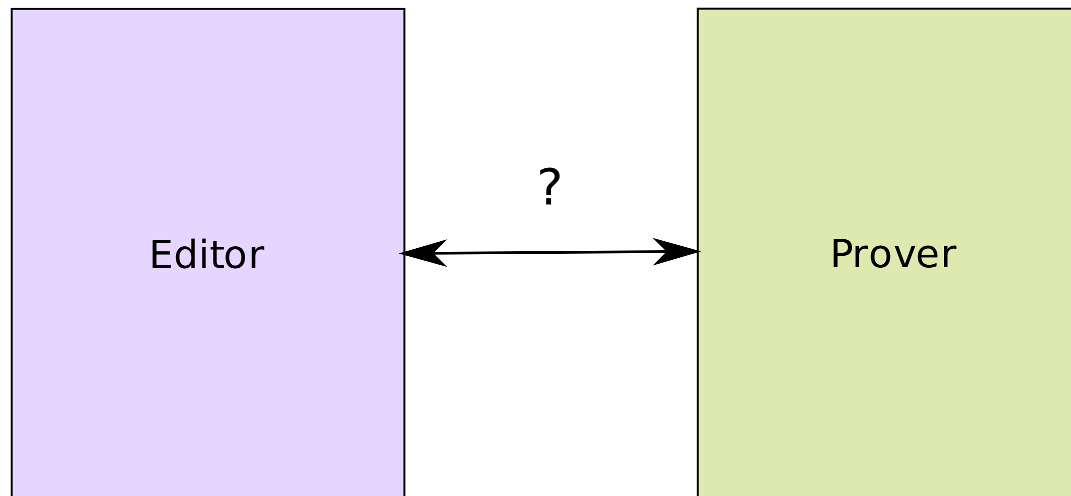


Characteristics:

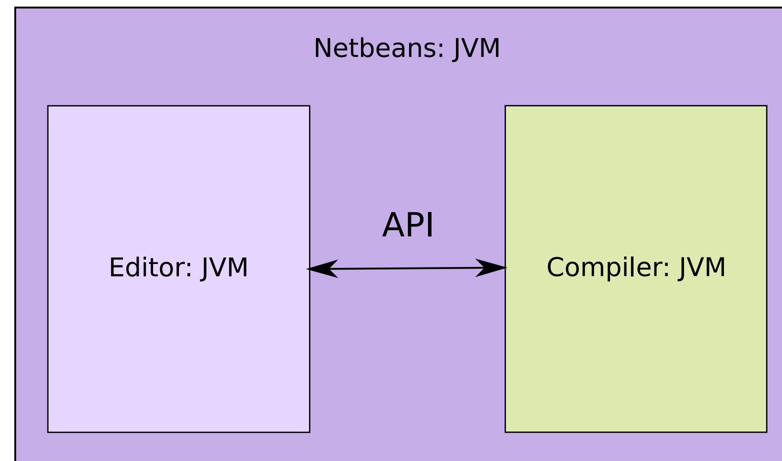
- small add-on for Why3
- minimal integration with CoqIDE
- lacks editor

PIDE architecture

The connectivity problem



Example: Java IDE



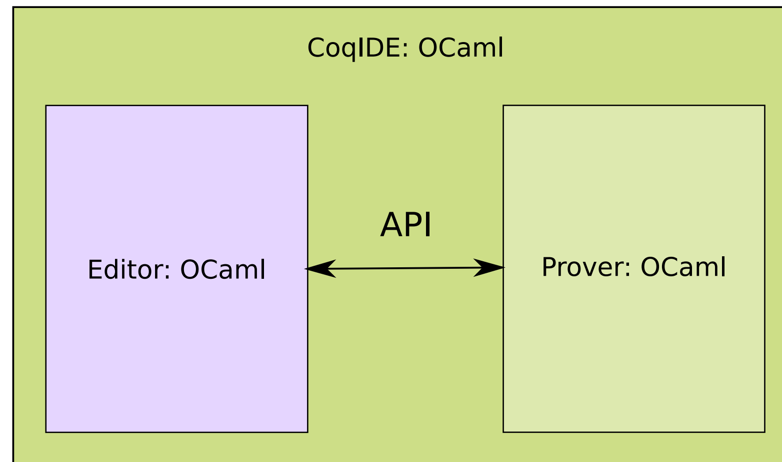
Characteristics:

+ Conceptually simple — no rocket science.
+ It works well — mainstream technology.

–– Provers are not implemented in Java!

– Even with Scala, the JVM is not ideal for hardcore FM.

Example: CoqIDE



Characteristics:

+ Conceptually simple — no rocket science.

+– It works . . . mostly.

– Many Coq power-users ignore it.

– GTK/OCaml is outdated; GTK/SML is unavailable.

– – – Need to duplicate editor implementation efforts.

Bilingual approach

Realistic assumption:

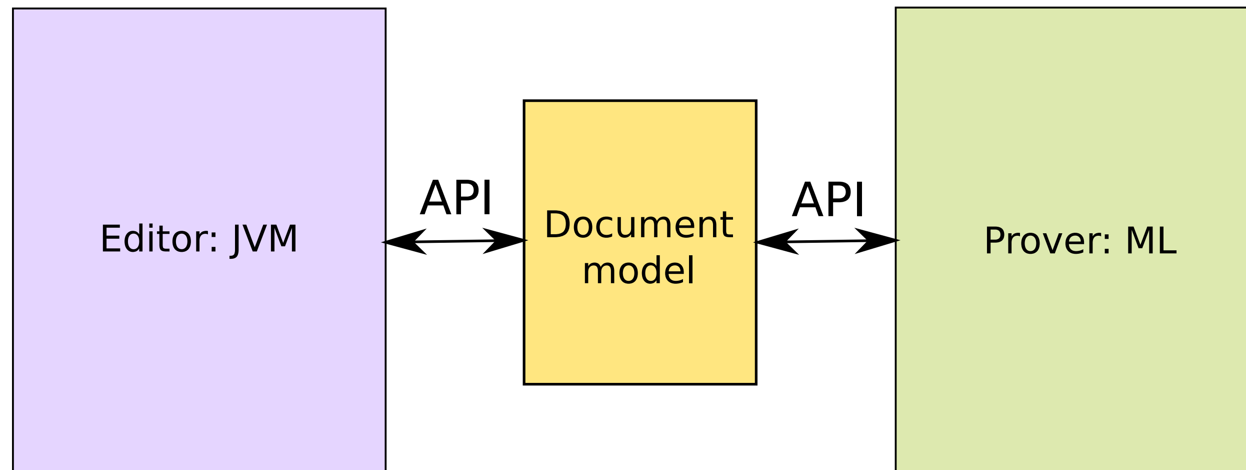
- Prover: ML (SML, OCaml, Haskell)
- Editor: Java

Big problem: How to integrate the two worlds?

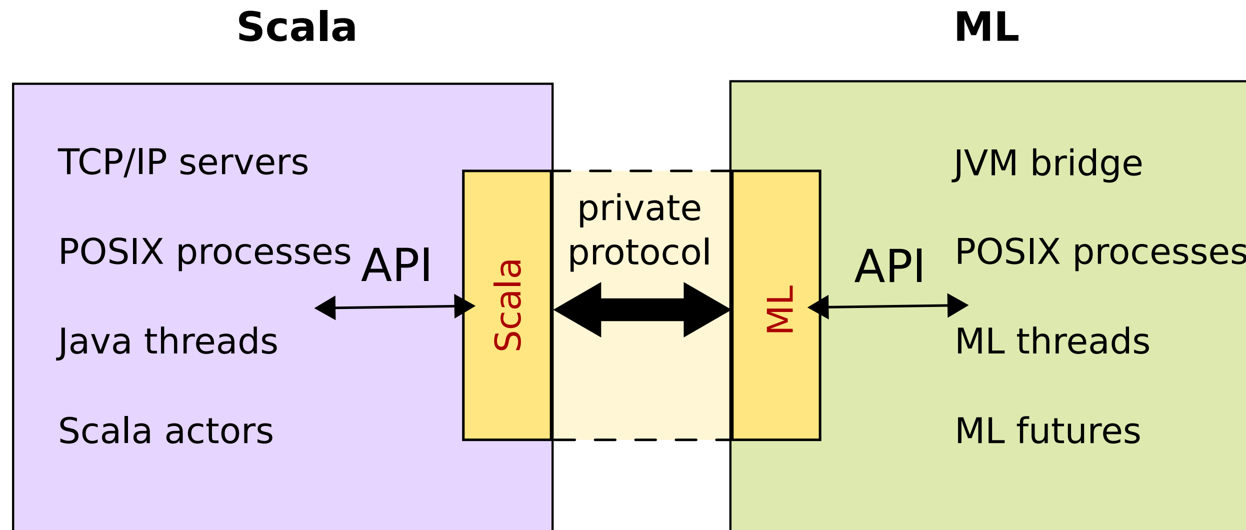
- Separate processes: requires marshalling, serialization, protocols.
- Different implementation languages and programming paradigms.
- Different cultural backgrounds!

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

PIDE architecture: conceptual view



PIDE architecture: implementation view



Design principles:

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

Scala

JVM platform problems

- 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 GUI support is subject to subtle issues ("write once, debug everywhere")
- null (cf. Tony Hoare: *Historically Bad Ideas: "Null References: The Billion Dollar Mistake"*)

Java language problems

– 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, Windows, Mac OS X)
+ despite a lot of junk, some good frameworks are available (e.g. *jEdit* editor)
+ Scala can use existing JVM libraries (with minimal exposure to Java legacy)

Scala language concepts (Martin Odersky et al)

- full compatibility with existing Java/JVM libraries — *asymmetric* upgrade path
- about as 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, scripting languages, "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-kinded parameters
  - type-inference
- incremental compiler ("toplevel loop")
- mainstream IDE support (IntelliJ IDEA, Eclipse, Netbeans)

Isabelle/ML versus Scala

Isabelle/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.)

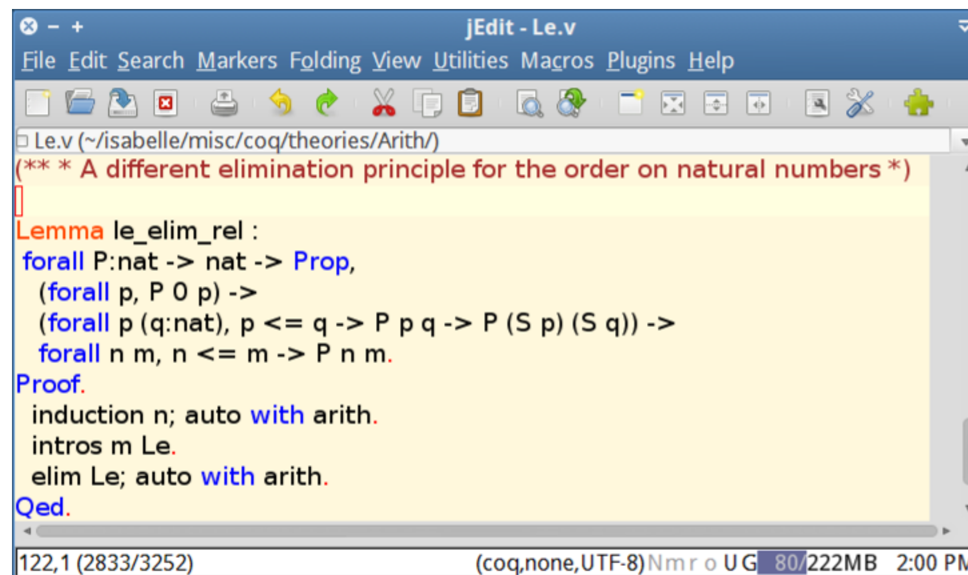
OCaml versus Scala

Left as an exercise for OCaml experts!

PIDE backend implementation

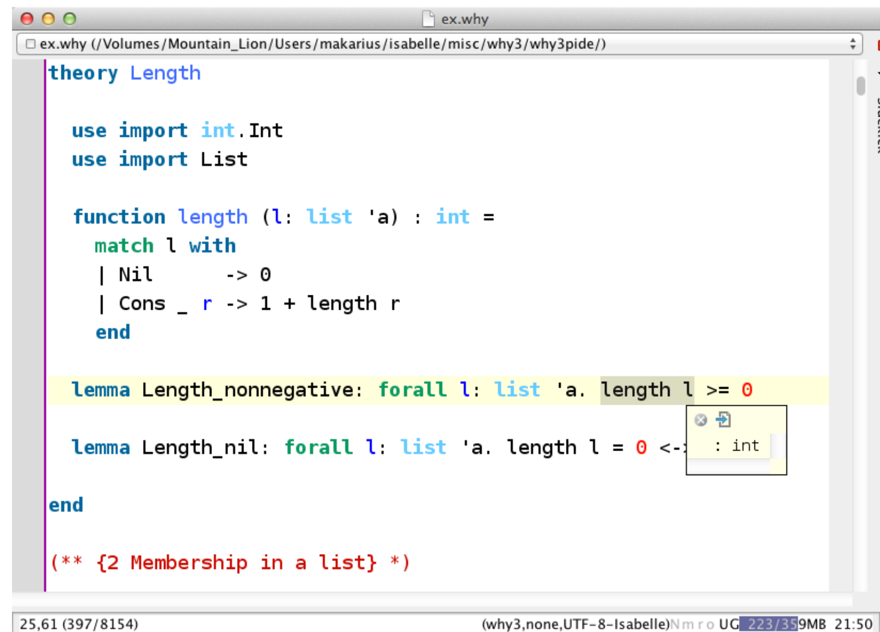
Example: CoqPIDE

- https://bitbucket.org/makarius/coq-pide/src/443d088a72e6/README.PIDE?at=v8.4
- coq-pide/ide/pide.ml (25 kB total; 2 kB payload for Coq)
- formal checking limited to lexical analysis (CoqIDE tokenizer)



Example: Why3PIDE

- `https://bitbucket.org/makarius/why3pide`
- why3pide/why3pide.ml (32 kB total; 8 kB payload for Why3)
- formal checking via reports about theory and term structure
- static syntax tables in jEdit (derived from `share/lang/why.lang`)



PIDE protocol layers (1)

Bidirectional byte-channel:

- pure byte streams
- block-buffering
- high throughput
- Unix: named pipes; Windows: TCP socket; not stdin/stdout

Message chunks:

- explicit length indication
- block-oriented I/O

Text encoding and character positions:

- reconcile ASCII, ISO-Latin-1, UTF-8, UTF-16
- unify Unix / Windows line-endings
- occasional readjustment of positions

PIDE protocol layers (2)

YXML transfer syntax:

- markup trees over plain text
- simple and robust transfer syntax
- easy upgrade of text-based application

XML/ML data representation

- canonical encoding / decoding of ML-like datatypes
- combinator library for each participating language, e.g. OCaml:

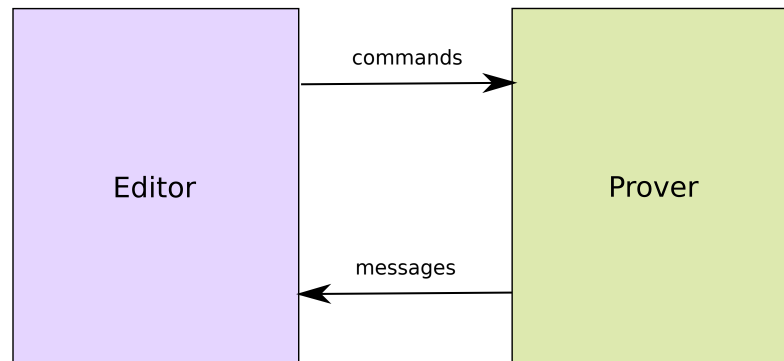
```
type 'a Encode.t = 'a -> XML.tree list
Encode.string: string Encode.t
Encode.pair: 'a Encode.t -> 'b Encode.t -> ('a * 'b) Encode.t
Encode.list: 'a Encode.t -> 'a list Encode.t
```

- untyped data representation of typed data
- typed conversion functions

Protocol functions

- `type protocol_command = name -> input -> unit`
- `type protocol_message = name -> output -> unit`
- outermost state of protocol handlers on each side (pure values)
- asynchronous streaming in each direction

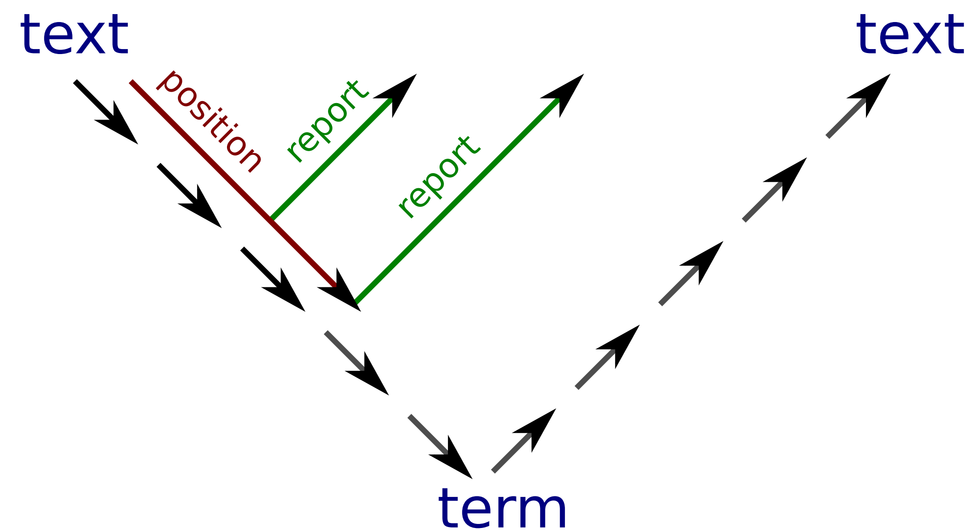
⟶ editor and prover as stream-procession functions



Markup reports

Problem: round-trip through several sophisticated syntax layers

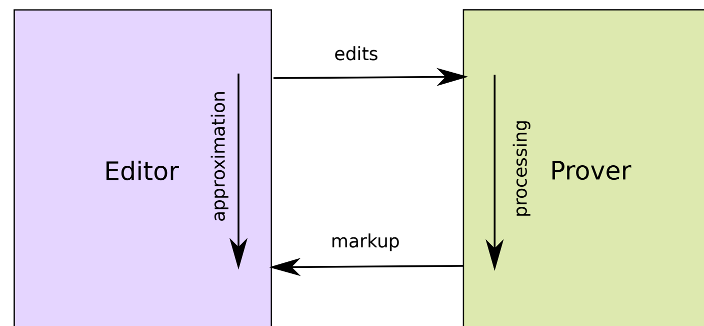
Solution: execution trace with markup reports



Document snapshots

Approximation and convergence:

1. text T, markup M, edits ΔT
2. apply edits: $T' = T + \Delta T$ (immediately in editor)
3. formal processing of T': ΔM after time Δt (eventually in prover)
4. temporary approximation (immediately in editor):
   $\tilde{M} = revert\ \Delta T; retrieve\ M; convert\ \Delta T$
5. convergence after time Δt (eventually in editor):
   $M' = M + \Delta M$



Conclusions

Conclusions

- sophisticated IDEs for sophisticated formal tools are possible, and actually easy with existing PIDE infrastructure
- need to think beyond ML (OCaml)
- need to avoid traps of "software heavy-industry" on JVM platform
- no need to give up good manners: strongly-typed higher-order functional programming with pure values
- feasibility and scalability proven by Isabelle/jEdit
  `http://isabelle.in.tum.de/`