Beyond Script Management

David Aspinall and Christoph Lüth

4th April 2007

Outline

State of the Union

The Proof Engineering Programme

Research

Implementation

Conclusions

Outline

State of the Union

The Proof Engineering Programme

Research

Implementation

Conclusions

Proof General

```
Emacs@Bernays.local
State Context Goal Retract Undo Next Use Goto Q.E.D. Find Command Stop Restart Help

text {* Proper proof text -- \textit{advanced version}. *}

theorem "A & B --> B & A"
proof
  assume "A & B"
  then obtain B and A ..
  then show "B & A" ..
qed


text {* Unstructured proof script. *}

theorem "A & B --> B & A"
  apply (rule impI)
  apply (erule conjE)
  apply (rule conjI)
  apply assumption
  apply assumption
done
--:--  Example.thy     53% L38    CVS:8.3   (Isar script Scripting)-----------

proof (prove): step 3

goal (theorem, 2 subgoals):
 1. [| A; B |] ==> B
 2. [| A; B |] ==> A

-:--  *goals*          All L1      (proofstate)-----------
```

- *Generic* interface:
  - Isabelle,
  - Coq,
  - LEGO,
  - HOL, ACL2,
  - ...
- *Emacs*-based
- Est: 40k lines of code; 4 py effort
- Widely used since 1998
- **EOL** 2008?

Expert Users

- Large proofs developments certainly can be constructed:
  - the IP stack specification
  - type safety and programming logics for Java
  - Gonthier & Werner's proof of the Four Colour Theorem
  - Hales's ongoing project on formalising his proof of the Kepler Conjecture (20 person-years; 1/3 way through)
  - The Mizar Mathematical Library
  - ...

Expert Users

- Large proofs developments certainly can be constructed:
  - the IP stack specification
  - type safety and programming logics for Java
  - Gonthier & Werner's proof of the Four Colour Theorem
  - Hales's ongoing project on formalising his proof of the Kepler Conjecture (20 person-years; 1/3 way through)
  - The Mizar Mathematical Library
  - ...
- But they are very difficult to construct and maintain

Expert Users

- Large proofs developments certainly can be constructed:
  - the IP stack specification
  - type safety and programming logics for Java
  - Gonthier & Werner's proof of the Four Colour Theorem
  - Hales's ongoing project on formalising his proof of the Kepler Conjecture (20 person-years; 1/3 way through)
  - The Mizar Mathematical Library
  - ...
- But they are very difficult to construct and maintain
  — more than they ought to be, that is

New Users

[Interactive] proving is not just a slightly more fussy version of paper proving and neither (Curry-Howard notwithstanding) is it really like programming.
It's a strange new skill, much harder to learn than a new programming language or application.
Coq, or something like it, is the future, if only we could make the initial learning experience a few thousand times less painful.

New Users

> *[Interactive] proving is not just a slightly more fussy version of paper proving and neither (Curry-Howard notwithstanding) is it really like programming.*
> *It's a strange new skill, much harder to learn than a new programming language or application.*
> *Coq, or something like it, is the future, if only we could make the initial learning experience a few thousand times less painful.*

[Nick Benton, First Workshop on Mechanized Metatheory, 2006. Color emphasis mine.]

Outline

State of the Union

The Proof Engineering Programme

Research

Implementation

Conclusions

Proof Engineering

Support the *lifecycle* of large proof developments:

Proof Engineering

Support the *lifecycle* of large proof developments:

- **Construction**: importing, assisting, generating;

Proof Engineering

Support the *lifecycle* of large proof developments:

- **Construction**: importing, assisting, generating;
- **Maintenance**: changing, reusing, exploring choices;

Proof Engineering

Support the *lifecycle* of large proof developments:

- **Construction**: importing, assisting, generating;
- **Maintenance**: changing, reusing, exploring choices;
- **Understanding**: navigating, documenting, exporting.

This is our characterisation of *Proof Engineering* (PE), by (loose) analogy with Software Engineering (SE).

PE != SE

- Writing proofs is often harder than writing programs;

PE != SE

- Writing proofs is often harder than writing programs;
- Formal proofs are complex, dense, and interdependent;

PE != SE

- Writing proofs is often harder than writing programs;
- Formal proofs are complex, dense, and interdependent;
- They are brittle: small changes cause large breakage;

PE != SE

- Writing proofs is often harder than writing programs;
- Formal proofs are complex, dense, and interdependent;
- They are brittle: small changes cause large breakage;
- Phase distinction: checking like compilation but uses computation;

PE != SE

- Writing proofs is often harder than writing programs;
- Formal proofs are complex, dense, and interdependent;
- They are brittle: small changes cause large breakage;
- Phase distinction: checking like compilation but uses computation;
- Proof irrelevance never holds in practice;

PE != SE

- Writing proofs is often harder than writing programs;
- Formal proofs are complex, dense, and interdependent;
- They are brittle: small changes cause large breakage;
- Phase distinction: checking like compilation but uses computation;
- Proof irrelevance never holds in practice;
- Languages and libraries change: bad compatibility;

PE != SE

- Writing proofs is often harder than writing programs;
- Formal proofs are complex, dense, and interdependent;
- They are brittle: small changes cause large breakage;
- Phase distinction: checking like compilation but uses computation;
- Proof irrelevance never holds in practice;
- Languages and libraries change: bad compatibility;
- Effective ways of using, organising libraries lacking.

PE != SE

- Writing proofs is often harder than writing programs;
- Formal proofs are complex, dense, and interdependent;
- They are brittle: small changes cause large breakage;
- Phase distinction: checking like compilation but uses computation;
- Proof irrelevance never holds in practice;
- Languages and libraries change: bad compatibility;
- Effective ways of using, organising libraries lacking.
- Extraordinarily hard to learn and apply...

Our Methodology

- Currently there is a tragic fragmentation across systems:
  - systems hard to relate, compare
  - proofs tricky to port
  - expertise is segregated
  - **tool building effort multiplied**

Our Methodology

- Currently there is a tragic fragmentation across systems:
  - systems hard to relate, compare
  - proofs tricky to port
  - expertise is segregated
  - **tool building effort multiplied**
- We can't afford this, it holds theorem proving back

Our Methodology

- Currently there is a tragic fragmentation across systems:
  - systems hard to relate, compare
  - proofs tricky to port
  - expertise is segregated
  - **tool building effort multiplied**
- We can't afford this, it holds theorem proving back
- ... yet different logics and domain-specifics are expected.

Our Methodology

- Currently there is a tragic fragmentation across systems:
  - systems hard to relate, compare
  - proofs tricky to port
  - expertise is segregated
  - **tool building effort multiplied**
- We can't afford this, it holds theorem proving back
- . . . yet different logics and domain-specifics are expected.
- Some remedies: transfer libraries; compile down; . . .

Our Methodology

- Currently there is a tragic fragmentation across systems:
  - systems hard to relate, compare
  - proofs tricky to port
  - expertise is segregated
  - **tool building effort multiplied**
- We can't afford this, it holds theorem proving back
- . . . yet different logics and domain-specifics are expected.
- Some remedies: transfer libraries; compile down; . . .
- Use *generic* tools with *abstracted* foundations, applied uniformly to different systems.

Our Methodology

- Currently there is a tragic fragmentation across systems:
  - systems hard to relate, compare
  - proofs tricky to port
  - expertise is segregated
  - **tool building effort multiplied**
- We can't afford this, it holds theorem proving back
- . . . yet different logics and domain-specifics are expected.
- Some remedies: transfer libraries; compile down; . . .
- Use *generic* tools with *abstracted* foundations, applied uniformly to different systems.
- Research: **models**, **languages**, **methods**.

Our Methodology

- Currently there is a tragic fragmentation across systems:
  - systems hard to relate, compare
  - proofs tricky to port
  - expertise is segregated
  - **tool building effort multiplied**
- We can't afford this, it holds theorem proving back
- . . . yet different logics and domain-specifics are expected.
- Some remedies: transfer libraries; compile down; . . .
- Use *generic* tools with *abstracted* foundations, applied uniformly to different systems.
- Research: **models**, **languages**, **methods**.
- Implement: **components** in a generic software architecture for interactive proof.

Our Methodology

- Currently there is a tragic fragmentation across systems:
  - systems hard to relate, compare
  - proofs tricky to port
  - expertise is segregated
  - **tool building effort multiplied**
- We can't afford this, it holds theorem proving back
- ...yet different logics and domain-specifics are expected.
- Some remedies: transfer libraries; compile down; ...
- Use *generic* tools with *abstracted* foundations, applied uniformly to different systems.
- Research: **models**, **languages**, **methods**.
- Implement: **components** in a generic software architecture for interactive proof.
- $\Rightarrow$ a natural evolution of Proof General.

Outline

State of the Union

The Proof Engineering Programme

Research

Implementation

Conclusions

PE Foundations: Models

We need models of:

- **Proof at multiple levels**: proof objects, *hi-proofs*, theory extensions, (human-readable) documents, proof sketches, design models, . . .

PE Foundations: Models

We need models of:

- **Proof at multiple levels**: proof objects, *hi-proofs*, theory extensions, (human-readable) documents, proof sketches, design models, . . .
- **Development**: relationship between proofs, explaining development process in moving between versions. Possibility: use *development graphs*.

PE Foundations: Models

We need models of:

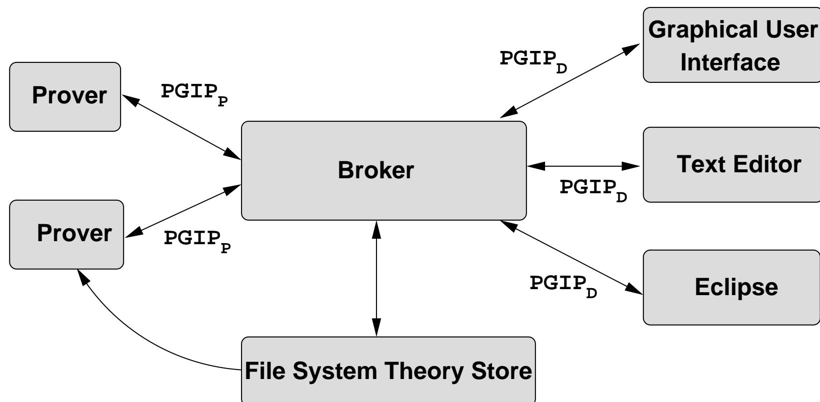
- **Proof at multiple levels**: proof objects, *hi-proofs*, theory extensions, (human-readable) documents, proof sketches, design models, . . .
- **Development**: relationship between proofs, explaining development process in moving between versions. Possibility: use *development graphs*.
- **Dependency**: manifest or implied relationships between parts of a proof.

PE Foundations: Models

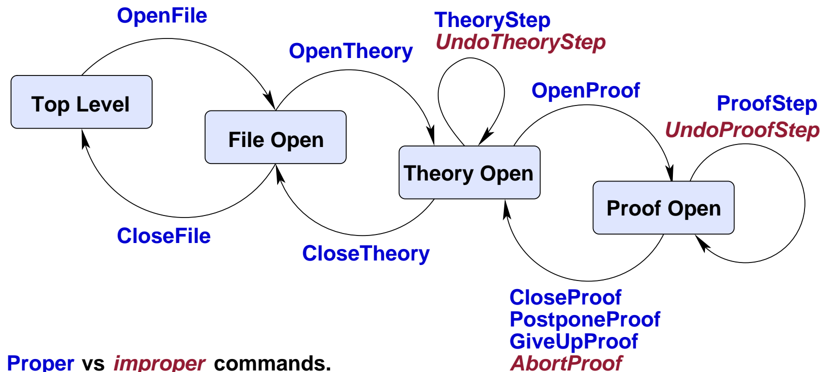
We need models of:

- **Proof at multiple levels**: proof objects, *hi-proofs*, theory extensions, (human-readable) documents, proof sketches, design models, . . .
- **Development**: relationship between proofs, explaining development process in moving between versions. Possibility: use *development graphs*.
- **Dependency**: manifest or implied relationships between parts of a proof.
- **Interaction: how to construct proofs using interactive tools, connecting proof engines to interfaces.**

PG Kit [Bremen, Edinburgh, Munich 2003-]

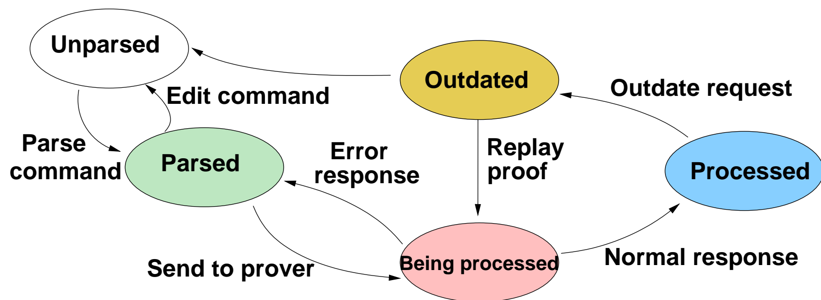


Interaction Model: Prover States in Development

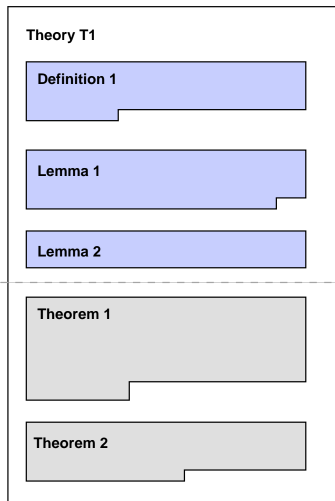


Proper vs ***improper*** **commands.**

Interaction Model: Document States in Development



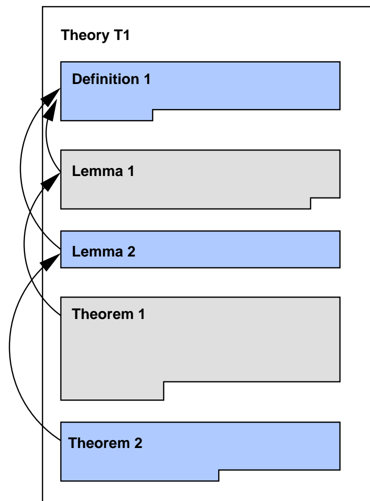
Interaction Model: Script Management



Linear dependency:

- Classical *script management*
- No *prover assistance*
- Potentially *expensive*

Dependencies for Script Management



Explicit dependencies:

- Requires *accurate prover assistance*
- Better *change management*

PE Foundations: Languages

- Generic language: study prover-independent engineering methods, unencumbered by specific features, changing targets.

PE Foundations: Languages

- Generic language: study prover-independent engineering methods, unencumbered by specific features, changing targets.
- Ideally: generic language can be viewed as subset of native language, e.g., by annotations.

PE Foundations: Languages

- Generic language: study prover-independent engineering methods, unencumbered by specific features, changing targets.
- Ideally: generic language can be viewed as subset of native language, e.g., by annotations.
- Do not give full logical meaning to language, just enough structure to do the job (c.f. *Isar* outer syntax without inner syntax).

PE Foundations: Languages

- Generic language: study prover-independent engineering methods, unencumbered by specific features, changing targets.
- Ideally: generic language can be viewed as subset of native language, e.g., by annotations.
- Do not give full logical meaning to language, just enough structure to do the job (c.f. *Isar* outer syntax without inner syntax).
- Semantics in terms of model notions.

Proof Language

Proof script: *native language*

```
lemma fn1: "(∃ x. P (f x)) ⟶ (∃ y. P y)"
proof
  assume "∃ x. P (f x)"
  thus "∃ y. P y"
  proof
    fix a
    assume "P (f a)"
    show ?thesis ..
  qed
qed
```

Proof Language Markup

Proof scripts: *native language* plus *PGIP markup*

```
<opengoal name="fn1">lemma fn1: &quot;(<sym name="exists">EX</sym
<openblock/><proofstep>proof</proofstep>
  <proofstep>assume &quot;<sym name="exists">EX</sym> x. P (f x)&qu
  <proofstep>thus &quot;<sym name="exists">EX</sym> y. P y&quot;</p
  <openblock/><proofstep>proof</proofstep>
    <proofstep>fix a</proofstep>
    <proofstep>assume &quot;P (f a)&quot;</proofstep>
    <proofstep>show ?thesis</proofstep><openblock/>
      <proofstep>..</proofstep><closeblock/>
  <proofstep>qed</proofstep><closeblock/>
<closegoal>qed</closegoal><closeblock/>
```

PE Methods

- *Construction*: content *assistance* (identifiers, templates); generation (tools: planning, resolution, recommendation)

PE Methods

- *Construction*: content *assistance* (identifiers, templates); generation (tools: planning, resolution, recommendation)
- *Views* on models: graphical and textual, supporting navigation, view update. (Hi-proof viewer at Edinburgh).

PE Methods

- *Construction*: content *assistance* (identifiers, templates); generation (tools: planning, resolution, recommendation)
- *Views* on models: graphical and textual, supporting navigation, view update. (Hi-proof viewer at Edinburgh).
- *Refactoring* proofs: simple but with complex pre-conditions (MoveTheorem) or complex operations (SplitProof, ClarifyProof).

PE Methods

- *Construction*: content *assistance* (identifiers, templates); generation (tools: planning, resolution, recommendation)
- *Views* on models: graphical and textual, supporting navigation, view update. (Hi-proof viewer at Edinburgh).
- *Refactoring* proofs: simple but with complex pre-conditions (MoveTheorem) or complex operations (SplitProof, ClarifyProof).
- *Proof Patterns*: recommended strategies for common problems (e.g. PartialFunction, Binder). Needs community effort.

Outline

State of the Union

The Proof Engineering Programme

Research

Implementation

Conclusions

An IDE for Proof based on PGIP

Proof General - HOLex/PER.thy - Proof General Eclipse

```
   x ≅ y ==> f x ≅ g y"
unfolding eqv_fun_def by blast

text {*
  The class of partial equivalence relations is
  closed under function spaces (in \emph{both}
  argument positions).
*}

instance "fun" :: (partial_equiv,partial_equiv)
                 partial_equiv
proof
  fix f g h :: "'a::partial_equiv ⇒
                'b::partial_equiv"
  assume fg: "f ≅ g"
  show "g ≅ f"
  proof
  assume gh: "g ≅ h"
  show "f ≅ h"
  proof
    fix x y :: 'a
    assume x: "x ∈ domain" and y: "y ∈ domain"
       and "x ≅ y"
```

```
proof (state): step 21
goal (1 subgoal):
1. ⋀x y. ⟦x ∈ domain; y ∈ domain; x ≅ y⟧ ⟹ f x ≅ h y
```

- *Many features*
- 45k loc, 2-3py effort

Environments for Formal Proof

- An effective *environment* for theorem proving is much more than a proof engine.
- Rest of environment in total larger than core (UI, file system/database, document processor).
- As software environments gets more complex and yet still diverse, effective genericity like this is catching on: GCC, CIL, VS .NET, Eclipse (even UML).

Things We Need

- To support a modern interface, the underlying proof engine should provide *structured information* and *interaction hints*. . .

Things We Need

- To support a modern interface, the underlying proof engine should provide *structured information* and *interaction hints*...
- rich output display (symbol and structure markup)
- exposing useful information (e.g. abstract syntax of terms and of proof scripts, dependencies in scripts).
- facilitating user interaction (e.g. classifying output, reporting status during long-running operations, querying user, informative error messages)
- providing useful operations (e.g. manipulation and examination of terms, templates for construction)
- facilitating behind-the-scenes programmatic interaction (e.g. parsing, querying, searching), ideally in an auxiliary thread
- ... provided in a stable protocol/API, e.g., PGIP

Outline

State of the Union

The Proof Engineering Programme

Research

Implementation

Conclusions

Conclusions

- Proof Engineering: a research programme for bringing proof development into 21st century.
- It's needed to keep theorem proving viable: (Rushby: TP risks being replaced by "disruptive technology").
- It's needed now: growing number of people want accessible theorem proving technology.
- Proof General Kit with Isabelle: an ideal platform for our research.
- Also, we hope: a compelling Proof Development Environment for users.