
Proof Planning in Isabelle and Beyond

Lucas Dixon

May 31, 2006



Fun at the Edinburgh MRG...

- The Rippling Technique...
- Proof Planning & Proof Critics...
- Interfaces (Proof General, HiProofs)...
- Deductive Synthesis...
- Theory Formation...
- Theory and Representation Morphisms...

Proof Planning

- Techniques that capture common patterns of reasoning
- A proof plan is a high level descriptions of (part of) a proof (think: Isar proof scripts)
- A proof planning *technique* operates on a proof plans
- Techniques differ from tactics in that they:
 - operate on proof scripts, not collections of subgoals.
 - the language focuses on providing non-logical machinery for automation (such as search, meta-information for proof guidance etc).

Proof Planning Techniques

- Capture common things people do with proof scripts
- Goals for proof automation (exemplified by Rippling)
 - well-behaved (terminate, unbloated subgoals, progress)
 - introspectable (unpackable into proof script)
 - robust over configuration
 - robust over the representation (relatively)
- Proof Planning suggests a way to interact with proof scripts:
by incrementally morphing them.

Proof Planning Example (1)

lemma "even x & even y --> even (x + y)"

[gap induction_and_rippling]

Choices (or): induction on x or y...

Proof Planning Example (2)

```
lemma "even x & even y --> even (x + y)"  
proof (induct x)  
  show "even 0 & even y --> even (0 + y)" [gap simp]  
next  
  fix x  
  assume [skelton]: "even x & even y --> even (x + y)"  
  show "even (Suc x) & even y --> even ((Suc x) + y)"  
    [gap rippling]  
qed
```

Choices (and): simp the base case and ripple the step-case

Proof Planning Example (3)

```
lemma "even x & even y --> even (x + y)"
proof (induct x)
  show "even 0 & even y --> even (0 + y)" by simp
next
  fix x
  assume [skelton]: "even x & even y --> even (x + y)"
  show "even (Suc x) & even y --> even ((Suc x) + y)"
    [gap rippling]
qed
```

Proof Planning Example (4)

```
lemma "even x & even y --> even (x + y)"
proof (induct x)
  show "even 0 & even y --> even (0 + y)" by simp
next
  fix x
  assume [skelton]: "even x & even y --> even (x + y)"
  have "even (Suc x) & even y --> even (Suc (x + y))"
    [gap (critic consider_induct_revision)]
  from this show "even (Suc x) & even y --> even ((Suc x) + y)"
    by (pp rippling)
qed
```

Proof Critics

Families of fixes to failure in proof planning techniques.

- Lemma speculation
- Generalisations
 - of common subterm
 - introduction of accumulators
- Specialisations (extra assumptions)
- Case analysis (from analysis of different branches)
- Induction revision

Induction Revision (1)

```
lemma "even x & even y --> even (x + y)"
proof (induct x)
  show "even 0 & even y --> even (0 + y)" by simp
next
  fix x
  assume [skelton]: "even x & even y --> even (x + y)"
  hence "even (Suc (Suc x)) & even y --> even ((Suc (Suc x)) + y)"
    by (pp rippling)
  show "even (Suc x) & even y --> even ((Suc x) + y)
    [gap (critic induct_revision with rule: even.induct)]
qed
```

Induction Revision (2)

```
lemma "even x & even y --> even (x + y)"
proof (induct x rule: even.induct)
  show "even 0 & even y --> even (0 + y)" by simp
next
  show "even (Suc 0) & even y --> even ((Suc 0) + y)" [gap simp]
next
  fix x
  assume [skeleton]: "even x & even y --> even (x + y)"
  thus "even (Suc (Suc x)) & even y --> even ((Suc (Suc x)) + y)"
    by (pp rippling)
qed
```

Induction Revision (3)

```
lemma "even x & even y --> even (x + y)"
proof (induct x rule: even.induct)
  show "even 0 & even y --> even (0 + y)" by simp
next
  show "even (Suc 0) & even y --> even ((Suc 0) + y)" by simp
next
  fix x
  assume [skeleton]: "even x & even y --> even (x + y)"
  thus "even (Suc (Suc x)) & even y --> even ((Suc (Suc x)) + y)"
    by (pp rippling)
qed
```

Unfolding a Technique

```
:
next
  fix x
  assume IH[skeleton]: "even x & even y --> even (x + y)"
  show "even (Suc (Suc x)) & even y --> even ((Suc (Suc x)) + y)"
  proof - << pp rippling >>
    from IH have "even (x) & even y --> even ((Suc (Suc x)) + y)"
      by (subst even_sucsuc)
    thus "even (Suc (Suc x)) & even y --> even ((Suc (Suc x)) + y)"
      by (subst even_sucsuc)
  qed
qed
```

From Proof Planning to Proof General

- Both need a representation of proof
- Ideally this would be hierarchical (more than just nested blocks!)
- Interaction can happen by morphisms on proof scripts
- Proof by pointing to script elements (needs representation of proof)
- Richer set of refactorings such as reordering and renaming (needs dependency management)
- Presentation of more information, e.g. dependencies, applicable assumptions etc

Benefits?

- Eases the learning curve: should be much easier for people to learn to write scripts (common errors cannot happen)
- Hierarchical view allows introspection to understand technique behaviour
- Speeds the writing of Isar style proofs: less copy and paste
- More automation by proof planning
- Removes the two-part window: the proof state is in the proof script
- Allows new kinds of interaction/automation from proof critics to smart refactoring macros

Challenges?

- automatic generation of proof scripts requires machine choice of names
- a new representation of proofs in order to maintain the relationship between the script and the actual proof. Two approaches:
 - Symmetry of printing and parsing
 - Richer representation



"Whoa! Watch where that thing lands—
we'll probably need it."

Pushing the Boundaries

- Theory formation
- Represent more than just the proof within planning - include definitions
- Theory morphisms?
- Deductive synthesis proofs: meta variables for term structure in the proof.

Conclusions

- approx 15 Isabelle users in Edinburgh mostly interested in pushing the boundaries of how proof is done.
- Proof planning in Isabelle requires a proof representation which ideally coincides with Isar: a possibility for integration?
- Proof planning & Proof General have many common needs and should have a close relationship
- Leads to new ways of writing proofs: semi-automated, proof-centred, proof by pointing.

So what's next?

thanks!