An Isabelle Proof Method Language

Daniel Matichuk[1,2], Makarius Wenzel[3], and Toby Murray[1,2]

[1] NICTA, Sydney, Australia*
[2] School of Computer Science and Engineering, UNSW, Sydney, Australia
[3] Univ. Paris-Sud, Laboratoire LRI, UMR8623, Orsay, F-91405, France
CNRS, Orsay, F-91405, France

Abstract. Machine-checked proofs are becoming ever-larger, presenting an increasing maintenance challenge. Isabelle's most popular language interface, Isar, is attractive for new users, and powerful in the hands of experts, but has previously lacked a means to write automated proof procedures. This can lead to more duplication in large proofs than is acceptable. In this paper we present Eisbach, a proof method language for Isabelle, which aims to fill this gap by incorporating Isar language elements, thus making it accessible to existing users. We describe the language and the design principles on which it was developed. We evaluate its effectiveness by implementing some tactics widely-used in the seL4 verification stack, and report on its strengths and limitations.

1 Introduction

Machine-checked proofs, developed using interactive proof assistants, present an increasing maintenance challenge as they become ever larger. For instance, the proofs and specifications that accompany the formally verified seL4 microkernel now comprise 480,000 lines of Isabelle/HOL [9], while Isabelle's Archive of Formal Proofs http://afp.sf.net now comprises over 900,000 lines. Each of these developments is updated to ensure it runs with each new Isabelle release. Large proofs about living software implementations present the additional maintenance challenge of having to be updated as the software to which they apply evolves over time, as is the case with seL4.

The Isabelle proof assistant [15, §6] provides various languages for different purposes, which sometimes overlap and sometimes complement each other. Most commonly used is the Isar language for theory specifications and structured proofs [14]. Isabelle/Isar sits alongside Isabelle/ML, which exposes the full power of system implementation and extension, including the ability to implement new sub-languages of the Isabelle framework. Isar itself is devoid of computation, but it may appeal to arbitrarily complex proof tools from the library: so-called *proof methods*. These are are usually implemented in Isabelle/ML. Isabelle/ML is integrated into the formal context of Isabelle/Isar, and supports referring to logical entities or Isar elements via *antiquotations* [13]. While this makes it reasonably easy to access the full power of ML in proofs, the vast majority of Isabelle theories are written solely in Isar: the AFP comprises just 50 ML files, as compared to 1663 Isar (.thy) files only 6 of which embed ML code.

* NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

The Isar proof language does not support proof procedure definitions directly, but this hasn't prevented large verifications from being completed: the seL4 proofs rely mainly on two custom tactics. This can be partly explained by the power of existing proof tools in Isabelle/HOL. However, it has arguably led to more duplication in these proofs than is acceptable; managing duplication has been a challenge for the seL4 proofs in [1]. This duplication makes proof maintenance difficult, and highlights the barrier to entry when implementing proof tools in Isabelle/ML. If automation can be expressed at a high level, a wider class of users can maintain and extend domain-specific proof procedures, which are often more maintainable than long proof scripts.

In this paper, we present a proof method language for Isabelle, called Eisbach, that allows writing proof procedures by appealing to existing proof tools with their usual syntax. The new Isar command **method-definition** allows proof methods to be combined, named, and abstracted over terms, facts and other methods. Eisbach is inspired by Coq's Ltac [4], and includes similar features such as matching on facts and the current goal. However, Eisbach's matching behaves differently to Ltac's, especially with respect to backtracking (see Section 3.5). Eisbach continues the Isabelle philosophy of exposing carefully designed features to the user while leaving more sophisticated functionality to Isabelle/ML: small snippets of ML may be easily included on demand. Eisbach benefits from general Isabelle concepts, while easing their exposure to users: pervasive backtracking, the structured proof context with named facts, and attributes to declare hints for proof tools.

The following simple example defines a new proof method which identifies a list in the conclusion of the current subgoal and applies the default induction principle to it with the existing method *induct*. All newly emerging subgoals are solved with *fastforce*, with additional simplification rules given as argument.

method-definition *induct-list* **facts** *simp* =
(**match** *?concl* **in** *?P* (*?x* :: *'a list*) ⇒ (*induct ?x* ↦ *fastforce simp*: *simp*))

Now *induct-list* can be called as a proof method to prove simple properties about lists.

lemma *length* (*xs* @ *ys*) = *length xs* + *length ys* **by** *induct-list*

The primary goal of Eisbach is to make writing proofs more productive, to avoid duplication, and thereby lower the costs of proof maintenance. Its design principles are:

- To be easy to use for beginners and experts.
- To expose limited functionality, leaving complex functionality to Isabelle/ML.
- Seamless integration with other Isabelle languages.
- To continue Isar's principle of readable proofs, creating readable proof procedures.

We begin in Section 2 by recalling some concepts of Isabelle and Isar. Section 3 then presents Eisbach, via a tour of its features in a tutorial style, concluding with the development of a solver for first order logic. We describe Eisbach's design and implementation in Section 4, before evaluating it in Section 5 by implementing the two most widely-used proof methods of the seL4 verification stack, and comparing them against their original implementations. Section 6 then surveys related work on proof programming languages, to put Eisbach in proper context. In Section 7 we compare Eisbach to Coq's Ltac and Mtac before considering future work and concluding.

2 Some Isabelle Concepts

Isabelle was originally introduced as yet another *Logical Framework* by Paulson [12], to allow rapid prototyping of implementations of inference systems, especially versions of Martin-Löf type theory. Some key concepts of current Isabelle can be traced back to this heritage, although today most applications are done exclusively in the object-logic Isabelle/HOL, and the general system framework has changed much in 25 years.

Isabelle/Pure is a minimal version of higher-order logic, which serves as general framework for Natural Deduction (with arbitrary nesting of rules). There are Pure connectives for universal parameters $\bigwedge x.\ \square$, premises $A \Longrightarrow \square$, and a notion of schematic variables *?x* (stripped outermost parameters). The Pure connectives outline inference rules declaratively, for example conjunction introduction $A \Longrightarrow B \Longrightarrow A \wedge B$ or well-founded induction $wf\ r \Longrightarrow (\bigwedge x.\ (\bigwedge y.\ (y, x) \in r \Longrightarrow P\ y) \Longrightarrow P\ x) \Longrightarrow P\ a$.

Isabelle/HOL is a rich library of logical theories and tools on top of Isabelle/Pure. It is the main workhorse for big applications, but is subsumed by the general concepts of Isabelle, so w.l.o.g. it is subsequently not explained further.

The logical framework of Isabelle/Pure is augmented by extra-logical infrastructure of Isabelle/Isar, which provides the general setting for structured reasoning. The actual Isar proof language [14] is merely an application of that: it provides particular expressions for human-readable proofs within the generic framework. Some of the underlying concepts of Isabelle architecture are outlined below, as relevant for Eisbach.

Fact. While the inference kernel operates on *thm* entities (as in LCF or HOL), Isabelle users always encounter results as *thm list*, which is called *fact*. This represents the idea of multiple results, without auxiliary conjunctions to encode it within the logic. There is notation to append facts, or to project sub-lists, without any formal reasoning involved.

Goal state. Following [12], the LCF goal state as auxiliary ML data structure is given up, and replaced by a proven theorem that states that the current subgoals imply the main conclusion. Goal refinement means to infer forwards on the negative side of some implication, so it appears like backwards reasoning. The proof starts with the trivial fact $C \Longrightarrow C$ and concludes with zero subgoals $\Longrightarrow C$, i.e. C outright. Administrative goal operations, e.g. shuffling of subgoals or restricted subgoal views, work by elementary inferences involving $\Longrightarrow$ in Isabelle/Pure. While outermost implications represent subgoals, outermost goal parameters correspond to *schematic variables* (or meta-variables), but the latter aspect is subsequently ignored for simplicity.

Tactic. Isabelle tactics due to [12] follow the idea behind LCF tactics, but implement the backwards refinement more directly in the logical framework, without replaying tactic justifications (as still seen in HOL or Coq today). This avoids the brittle concentration of primitive inferences at qed-time. Moreover, backtracking is directly built-in, by producing an unbounded lazy list of results, instead of just zero or one. LCF-style **tacticals** are easily recovered, by composing functions that map a goal state to a sequence of subsequent goal states. Rich varieties of combinators with backtracking are provided, although modern-time proof tools merely use a more focused vocabulary.

Subgoal structure. An intermediate goal state with n open subgoals has the form $H_1 \Longrightarrow \ldots\ H_n \Longrightarrow C$, each with its own substructure $H = (\bigwedge x.\ A\ x \Longrightarrow B\ x)$, for

zero or more *goal parameters* (here x) and *goal premises* (here $A\ x$). Following [12], this local context is implicitly taken into account when natural deduction rules are composed by *lifting*, *higher-order unification*, and *backward chaining*. Isar users encounter this operation frequently in the proof method *rule*, and the rule attributes *OF* or *THEN*.

Other proof tools may prefer direct access to hypothetical terms and premises, when inspecting a subgoal. In Isabelle today the concept of **subgoal focus** achieves that: the proof context is enriched by a fixed term x and assumed fact $A\ x$, and the subgoal restricted to $B\ x$. After refining that, the result is retrofitted into the original situation.

Proof context. Motivated by the Isar proof language [14], the structured proof context provides general administrative structure, to complement primitive *thm* values of the inference kernel. The idea is to provide a first-class representation in ML, of open situations with hypothetical terms (fixed variable x) and assumptions (hypothetical fact A); Hindley-Milner type discipline with schematic polymorphism is covered as well. Proof contexts are not restricted to this logical core, but may contain arbitrary tool-specific **context data**. A typical example is the standard environment of facts (see above), which manages both static and dynamic entries: a statically named fact is interchangeable with its *thm list* as plain value, but a dynamic fact is a function depending on the context.

Attributes. Facts and contexts frequently occur together, and may modify each other by means of attributes (which have their own syntax in Isar). A **rule attribute** modifies a fact depending on the context (e.g. *fact* [*of t*] to instantiate term variables), and a **declaration attribute** modifies the context depending on a fact (e.g. *fact* [*simp*] to add Simplifier rules to the context). Such declarations for automated proof tools also work in hypothetical contexts, with fixed x and assumed $A\ x$. There is standard support to maintain named collections of dynamic facts, with attributes to add or delete list entries.

3 Eisbach

3.1 Isar Proof Methods

Eisbach provides the ability to write automated reasoning procedures to non-expert users of Isabelle, specifically users only familiar with the use of Isabelle/Isar [14].

Isar is a document-oriented proof language, focusing on producing and presenting human-readable formal proofs. Such proofs are a structured argument about why a claim is true, with invocations to proof methods to decompose a claim into multiple goals or to solve outstanding proof goals. For the purposes of this paper, proof method invocations come in two forms: structured and unstructured.

The structured form is "**by** $method_1\ method_2$", where the initial $method_1$ performs the main structural refinement of the goal, and the terminal (optional) $method_2$ may solve emerging subgoals; the proof is always closed by implicit *assumption* steps to finish-off trivial subgoals. For example, "**by** (*induct n*) *simp-all*" splits-up a problem by induction and solves it by simplification, or "**by** (*rule impI*)" applies a single rule and expects the remaining goal state to be trivial up to unification.

The unstructured form is "**apply** *method*", which applies the proof method to the goal without insisting the proof be completed; further **apply** commands may follow

to continue the proof, until it is eventually concluded by the command **done** (without implicit steps for closing). After one or two **apply** steps, the foreseeable structure of the reasoning is usually lost, and the Isar *proof text* degenerates into a *proof script*: understanding it later typically requires stepping through its intermediate goal states.

The *method* expressions above may combine basic proof methods using Isar's method *combinators*. Unlike former tacticals, there is only a minimalistic repertoire for repeated application, alternative choice, and sequential composition (with backtracking). Such methods are used in-place, to address a particular proof problem in a given situation.

Eisbach allows compound proof methods to be named, and extend the name space of basic methods accordingly. Method definitions may abstract over parameters: terms, facts, or other methods. Additionally, Eisbach provides an expressive matching facility that can be used to manage control flow and perform proof goal analysis via unification.

Subsequently, we will follow the development of a small first order logic solver in Eisbach, gradually increasing its scope and demonstrating the main language elements.

3.2 Combinators and Backtracking

There are four combinators in Isar. Firstly, "," is sequential composition of two methods with implicit backtracking: "*meth1*,*meth2*" applies *meth1*, which may produces a set of possible results (new proof goals), before applying *meth2* to all results produced by *meth1*. Effectively this produces all results in which the application of *meth1* followed by *meth2* is successful.

At the end of each **apply** command, the first successful result from all those produced is retained.

The second Isar combinator is "|", alternative composition: "*meth1*|*meth2*" tries *meth1* and falls through to *meth2* when *meth1* fails (yields no results). The third combinator "?" is a unary combinator that suppresses failure: *meth*? returns the original proof state when *meth* fails, rather than failing. Lastly, "+" is a unary combinator for repeated method application: *meth*+ repeatedly applies *meth* until *meth* fails, at which point it yields the proof state obtained before the final failing invocation of *meth*.

A typical method invocation might look as follows:

lemma $P \wedge Q \longrightarrow P$ **by** $((\textit{rule impI}, (\textit{erule conjE})?) \mid \textit{assumption})+$

Which, informally, says: "Apply the implication introduction rule, followed by optionally eliminating any conjunctions in the assumptions. If this fails, solve the goal with an assumption. Repeat this action until it is unsuccessful."

As well as the above lemma, this invocation will prove the correctness a small class of propositional logic tautologies. With the **method-definition** command we can define a proof method that makes the above functionality available generally.

method-definition $\textit{prop-solver}_1 = ((\textit{rule impI}, (\textit{erule conjE})?) \mid \textit{assumption})+$
lemma $P \wedge Q \wedge R \longrightarrow P$ **by** $\textit{prop-solver}_1$

3.3 Abstraction

We can abstract this method over its introduction and elimination rules to make it more generally applicable. The **facts** keyword declares fact parameters for use in the method.

These arguments are provided when the method is invoked, in the form of lists of facts for each, using Isar's standard method-sections syntax. Below we generalise the method above over its *intro* and *elim* rules respectively that it may apply.

method-definition *prop-solver*$_2$ **facts** *intro elim* =
((*rule intro*, (*erule elim*)*?*) | *assumption*)+

lemma $P \wedge Q \longrightarrow P$ **by** (*prop-solver*$_2$ *intro*: *impI elim*: *conjE*)

Above, the introduction and elimination rules need to be provided for each method invocation. Traditionally Isabelle proof methods avoid this by using tool-specific data as part of the proof context, which are managed using *attributes* (see Section 2) to add and remove entries. A method invocation retrieves the facts that it needs to know about whenever it is invoked, using the run-time proof context.

Eisbach supports creating new fact collections in the context using a new Isar command **declare-attributes**. A fact parameter $[p]$ surrounded by square brackets declares that fact to be backed by the fact collection p. It can be augmented further when a method is invoked using the common syntax *meth p*: *facts*, but can also be managed in the proof context with the Isar command **declare**.

declare-attributes *intro elim*

method-definition *prop-solver*$_3$ **facts** [*intro*] [*elim*] =
((*rule intro*, (*erule elim*)*?*) | *assumption*)+

declare *impI* [*intro*] **and** *conjE* [*elim*]

lemma $P \wedge Q \longrightarrow P$ **by** *prop-solver*$_3$

Methods can also abstract over terms using the **for** keyword, optionally providing type constraints. For instance, the following proof method *elim-all* takes a term y of any type, which it uses to instantiate the x-variable of the *allE* (forall elimination) rule before applying that rule as an elimination rule. The instantiation is performed here by Isar's *where* attribute. This has the effect of instantiating a universal quantification $\forall x.\ P\ x$ in one of the current assumptions by replacing it with the term $P\ y$.

method-definition *elim-all* **for** $Q :: 'a \Rightarrow$ *bool* **and** $y :: 'a$ =
(*erule allE* [*where* $P = Q$ **and** $x = y$])

The term parameters y and P can be used arbitrarily inside the method body, as part of attribute applications or arguments to other methods. The expression is type-checked as far as possible when the method is defined, however dynamic type errors can still occur when it is invoked (e.g. when terms are instantiated in a parameterized fact). Actual term arguments are supplied positionally, in the same order as in the method definition.

lemma $\forall x.\ P\ x \Longrightarrow P\ a$ **by** (*elim-all P a*)

3.4 Custom Combinators

The four existing combinators in Isar (mentioned above) quickly prove to be too restrictive when writing tactics in Eisbach. A fifth combinator ("$\mapsto$") was added, which takes two methods and, in contrast to ",", invokes the second method on *all* subgoals

produced by the first. This is necessary to handle cases where the number of subgoals produced by a method cannot be known statically.

lemma *True* ∧ *True* ∧ *True* **by** (*intro conjI* ↦ *rule TrueI*)

To more usefully exploit Isabelle's backtracking, the explicit requirement that a method solve all produced subgoals is frequently useful. This can easily be written as a *higher-order method* using "↦". The **methods** keyword denotes method parameters that are other proof methods to be invoked by the method being defined.

method-definition *solve* **methods** *m* = (*m* ↦ *fail*)

Given some method-argument *m*, *solve m* applies the method *m* and then fails whenever *m* produces any new unsolved subgoals – i.e. when *m* fails to completely discharge the goal it was applied to.

With these simple features we are ready to write our first non-trivial method. Returning to the first order logic example, the following method definition applies various rules with their canonical methods.

method-definition *prop-solver* **facts** [*intro*] [*dest*] [*elim*] [*subst*] =
(*assumption*
| *rule intro* | *drule dest* | *erule elim* | *subst subst* | *subst* (*asm*) *subst* |
(*erule notE* ↦ *solve prop-solver*))+

The only non-trivial part of this method definition is the final alternative (*erule notE* ↦ *solve prop-solver*). Here, in the case that all other alternatives fail, the method takes one of the assumptions ¬ *P* of the current goal and eliminates it with the rule *notE*, causing the goal to be proved to become *P*. The method then recursively invokes itself on the remaining goals. The job of the recursive call is to demonstrate that there is a contradiction in the original assumptions (i.e. that *P* can be derived from them). Note this recursive invocation is applied with the *solve* method to ensure that a contradiction will indeed be shown. In the case where a contradiction cannot be found, backtracking will occur and a different assumption ¬ *Q* will be chosen for elimination.

After declararing some standard rules to the context, e.g. (*P* ⟹ *False*) ⟹ ¬ *P* as [*intro*] and ¬ ¬ *P* ⟹ *P* as [*dest*], the *prop-solver* becomes capable of solving non-trivial propositional tautologies.

lemma (*A* ∨ *B*) ∧ (*A* ⟶ *C*) ∧ (*B* ⟶ *C*) ⟶ *C* **by** *prop-solver*

3.5 Matching

Matching allows the user to introspect the goal state, and to implement more explicit control flow. When performing a match, the user provides a term or fact collection *ts* to match against, along with a collection of pattern-method pairs (*p*, *m*): roughly speaking, when the pattern *p* matches any member of *ts*, the *inner* method *m* will be executed with schematic variables mentioned in *p* appropriately instantiated. In the case of matching against a fact collection, an optional name may be given for each pattern, which will be bound to the fact that was successfully matched out of term ts. The special term *?concl* is always defined to be the conclusion of the first subgoal, and the special fact *prems* is always defined to be the premises of the first subgoal. Using either in the term *t* allows

the user to perform matches against (i.e. to introspect) the current goal-state; doing so causes an implicit *subgoal focus* (see also Section 4) which binds these two names appropriately, creating a local context of local goal parameters (as fixed term variables) and premises (as hypothetical theorems).

In the following example we extract the predicate of an existentially quantified conclusion in the current subgoal and search the current premises for a matching fact. If both matches are successful, we then instantiate the existential introduction rule with both the witness and predicate, solving with the matched premise.

method-definition *solve-ex* =
(**match** *?concl* **in** $\exists\, x.\ ?Q\ x \Rightarrow$
(**match** *prems* **in** U: $Q\ ?y \Rightarrow$ (*rule exI* [*where* $x = y$ **and** $P = Q$, *OF U*]))))

The first match matches the pattern $\exists\, x.\ ?Q\ x$ against the current conclusion, binding the pattern $?Q$ to a particular term Q in the inner match. Next the pattern $Q\ ?y$ is matched against all premises of the current subgoal. Once a match is found, the local fact U is bound to the matching premise and the variable y is bound to the matching witness. The existential introduction rule $P\ x \Longrightarrow \exists\, x.\ P\ x$ is then instantiated with y as the witness and Q as the predicate, with its proof obligation solved by the local fact U (using the Isar attribute *OF*). The following example is a trivial use of this method.

lemma *halts p* $\Longrightarrow \exists\, x.$ *halts x* **by** *solve-ex*

Matching is performed from top to bottom, considering each pattern in turn until a match is found. When attempting to match a pattern, Eisbach tries to match the pattern against all provided terms/facts before moving on to the next pattern. Successful matches serve as cut points for backtracking. Specifically, once a match is made no other patterns will be attempted regardless of the outcome of the inner method m. However, all possible unifiers of that pattern will be explored, re-executing the method m with different variable bindings when backtracking.

The method *foo* below fails for all goals that are conjunctions. Any such goal will match the first pattern, causing the second pattern (that would otherwise match all goals) to never be considered. If multiple unifiers exist for the pattern $?P \wedge ?Q$ against the current goal, then the failing method *fail* will be (uselessly) tried for all of them.

method-definition *foo* =
(**match** *?concl* **in** $?P \wedge ?Q \Rightarrow$ *fail* | $?R \Rightarrow$ *prop-solver*)

This behaviour is in direct contrast to the backtracking done by Coq's Ltac [4], which will attempt all patterns in a match before failing. This means that the failure of an inner method that is executed after a successful match does not, in Ltac, cause the entire match to fail, whereas it does in Eisbach. In Eisbach the distinction is important due to the pervasive use of backtracking. When a method is used in a combinator chain, its failure becomes significant because it signals previously applied methods to move to the next result. Therefore, it is better for match to not mask such failure in Eisbach. One can always rewrite a match in Eisbach using the combinators *?* and | to have it try subsequent patterns in the case of an inner-method failure. The following proof method, for example, always invokes *prop-solver* for all goals because its first alternative either never matches or (if it does match) always fails.

method-definition $foo_1 =$
$((\textbf{match}\ ?concl\ \textbf{in}\ ?P \wedge ?Q \Rightarrow fail) \mid (\textbf{match}\ ?concl\ \textbf{in}\ ?R \Rightarrow prop\text{-}solver))$

Note that matching can be performed against arbitrary terms or facts, with *?concl* and *prems* being special cases. For example, we could match out of a given set of facts to locate rules with matching assumptions and conclusions.

method-definition *match-rules* **facts** *my-facts* =
(**match** *my-facts* **in** U: $?P \Longrightarrow ?Q$ **and** U': $Q \Longrightarrow ?R$
$\Rightarrow$ (*rule* U [*THEN* U']))

This example demonstrates use of the **and** keyword, which chains patterns linearly. First, a fact matching $?P \Longrightarrow ?Q$ is found and named U. Then, having bound Q from the pattern, a fact matching $Q \Longrightarrow ?R$ is matched from *my-facts*. If patterns are matched, then U and U' are bound to local facts and the method body is executed.

lemma
assumes f_1: $A \Longrightarrow B$ **and** f_2: $B \Longrightarrow C$
shows $A \Longrightarrow C$ **by** (*match-rules my-facts*: $f_1\ f_2$)

3.6 Example

We complete our tour of the features of Eisbach by extending the propositional logic solver presented earlier to first-order logic. The following method instantiates universally quantified assumptions by simple guessing, relying on backtracking to find the correct instantiation. Specifically, it instantiates assumptions of the form $\forall x.\ ?P\ x$ by finding some type-correct term y by matching other assumptions against $?H\ ?y$, using type annotations to ensure that the types match correctly. The use of the previously defined *elim-all* method here ensures that the same assumption that was matched is the one that will be eliminated. The same matching is also performed against the conclusion to find possible instantiations there too.

method-definition *guess-all* =
(**match** *prems* **in** U: $\forall x.\ ?P\ (x :: 'a) \Rightarrow$
(**match** *prems* **in** $?H\ (?y :: 'a) \Rightarrow$
(*elim-all P y*)
| **match** *?concl* **in** $?H\ (?y :: 'a) \Rightarrow$
(*elim-all P y*)))

The higher order pattern $?H\ ?y$ is used to find arbitrary subterms y within the premises or conclusion of the current goal. It makes use of Isabelle/Pure's workhorse of higher order unification (although matching involves pattern-matching only). While such a pattern-match need not bind all variables to be valid, to avoid trivial matches, Eisbach considers only those matches that bind all variables mentioned in the pattern.

The inner-match must be duplicated over both the premises and conclusion because of the logical distinction between facts (the premises) and terms (the conclusion). This might look strange to users of Coq's Ltac, where these notions are identified; however, it does not limit the expressivity of Eisbach.

Similar to our previous *solve-ex* method, we introduce a method which attempts to guess at an appropriate witness for an existential proof. In this case, however, the

method simply guesses the witness based on terms found in the current premises, again using higher order matching as in the *guess-all* method above.

```
method-definition guess-ex =
 (match ?concl in
  ∃x. ?P (x :: 'a) ⇒
   (match prems in ?H (?x :: 'a) ⇒
        (rule exI [where x = x and P = P])))
```

These methods can now be combined into a surprisingly powerful first order solver.

```
method-definition fol-solver =
 ((guess-ex | guess-all | prop-solver) ↦ solve fol-solver)
```

The use of *solve* here on the recursive call to the method ensures that the recursive subgoals are solved. Without it, the recursive call could potentially prematurely terminate and leave the goal in an unsolvable state (due to an incorrect guess for a quantifier instantiation).

After declaring some standard rules in the context, this method is capable of solving various standard problems.

```
lemma (∀x. P x) ∧ (∀x. Q x) ⟹ (∀x. P x ∧ Q x)
 and  ∃x. (P x ⟶ (∀x. P x))
 and (∃x. ∀y. R x y) ⟶ (∀y. ∃x. R x y)
 by  fol-solver+
```

4 Design and Implementation

A core design goal of Eisbach is a seamless integration with other Isabelle languages, notably Isar, ML, and object-logics. The primary motivation clearly being to make it accessible to existing Isabelle/Isar users, with a secondary objective of both forward and backward compatibility.

4.1 Static Closure of Concrete Syntax

Isabelle provides a rich selection of powerful proof methods, each with its own parser and invocation style. Additionally, Isabelle's theorem attributes, which perform context and fact transformations, have their own parsers of arbitrary complexity. Rather than re-write these tools to support Eisbach, we exploited an existing feature of the Isabelle parsing framework whereby tokens have values (types, terms and facts) assigned to them implicitly during parsing.

This implicit value assignment mechanism is the main workhorse of Eisbach, allowing it to embed most Isar syntax as uninterpreted token lists. Eisbach then simply serves as an interpretation environment: when a proof method is applied Eisbach instantiates these token values appropriately based on the supplied arguments to the method or results of matching, and then executes the resulting method body.

Although this presents some technical challenges and requires some minor modifications to Isar, this proves to be a very effective solution to performing this kind of

language extension. The necessity of this patching will ideally disappear as the design and implementation principles of Eisbach mature, and thus motivate the incorporation of appropriate concepts into core Isabelle.

4.2 Subgoal Focusing

In Isabelle there is a logical distinction between universally quantified parameters (such as x in $\bigwedge x.\ P\ x$) and arbitrary-but-fixed terms (such as x in $P\ x$). A subgoal in the former form does not allow the x to be explicitly referenced (for example, *my-fact* [*where* $y = x$] does not produce a valid theorem). To deal with this, a set of so-called "improper" methods (like *rule-tac*) have traditionally been used, which are aware of this peculiarity.

It is important to note that premises within a subgoal are not local facts. In a structured Isar proof, assumptions are stated explicitly in the text via **assumes** or **assume** and are accessible to attributes etc. In contrast, the local prefix $\bigwedge x.\ A\ x \Longrightarrow \square$ of a subgoal is not accessible to structured reasoning yet.

To allow the user to write methods that can operate directly on subgoal structure, we decided to expose Isabelle's *subgoal focusing* to Eisbach. Focusing creates a new goal out of a given subgoal, but with its parameters lifted into fixed variables and premises into local assumptions. This allows for uniform treatment of the goal state when matching and parameter passing. In Eisbach, focusing is implicitly triggered whenever the special term *?concl* or special fact *prems* are mentioned. Focusing causes these names to be bound to the conclusion and premises of the current subgoal, respectively.

To support its use in Eisbach, the existing subgoal focusing was enriched to be more generally applicable. Premises, while turned into a local fact, still remain part of the goal. This allows methods like *erule* to still remove premises from the goal.

5 Application and Evaluation

To evaluate Eisbach we re-implemented two existing proof methods: *wp* and *wpc*, which are VCGs currently released as part of the AutoCorres framework [8]. They were used extensively in the full functional correctness proof of seL4 [10] for both invariant and refinement proofs. They were originally designed for performing "weakest-precondition" style reasoning against a shallowly embedded monadic Hoare logic [3]. The intelligence of these methods lies in their large collection of stored facts, and have proven to be more generally useful in other projects [11].

Together these two methods comprise 500 lines of Isabelle/ML, and 60 lines of Isabelle/Isar for setup. However, they may be implemented in Eisbach almost trivially.

The Eisbach implementation of *wp* degenerates into the structured application of some dynamic facts: *wp* supplies facts about monadic functions (e.g. Hoare triples), *wp-comb* contains decomposition rules for postconditions, and *wp-split* splits goals across monadic binds.

method-definition *wp* **facts** [*wp*] [*wp-comb*] [*wp-split*] =
((*rule wp* | (*rule wp-comb*, *rule wp*)) | *rule wp-split*)+

This obscures some details from the original implementation, in particular that the collection of *wp* rules grows quite large and relying exclusively on rule resolution to apply it is costly. This suggests potential improvements to Eisbach, such as allowing facts in the context to be explicitly indexed.

The Eisbach implementation of *wpc* is slightly more involved. It makes use of a simple custom attribute *get-split*, defined in Isabelle/ML, to retrieve the *case-split rule* for a given term; such rules are used to decompose case distinctions on datatypes. The *apply-split* method applies the retrieved case-split rule, specialized to the current goal.

```
method-definition apply-split for f =
  (match [[get-split f]] in U: ?P and TERM ?x ⇒
    (match ?concl in ?R f ⇒
      (rule U [THEN iffD2, of x R])))
```

We defined another higher-order method *repeat-new* to repeatedly apply a provided method *m* to all produced subgoals.

```
method-definition repeat-new methods meth = (meth ↦ (repeat-new meth)?)
```

This method is then used in conjunction with worker lemmas to produce one subgoal for each constructor.

```
method-definition wpc' for f facts [wpc-helper] =
 (apply-split f,
  rule wpc-helperI,
  repeat-new (rule wpc-processors) ↦ (rule wpc-helper))
```

Finally, *wpc* matches the underlying monadic function out of the current Hoare triple subgoal.

```
method-definition wpc =
 (match ?concl in {|?P|} ?f {|?Q|} ⇒ (wpc' f) | {|?P|} ?f {|?Q|},{|?E|} ⇒ (wpc' f))
```

Together, combined with a large body of existing lemmas, these methods calculate weakest-precondition style proof obligations for the monadic Hoare logic of [3]. Additionally, with appropriate lemmas and some additional match conditions for *wpc*, these methods are easily extended to other calculi such as that from [11].

To evaluate the effectiveness of these re-implemented methods, we re-ran the invariant proofs for the seL4 abstract functional specification using them in place of their original implementations. These proofs constitute about 60,000 lines, including whitespace and comments. About 100 lines of Isabelle/ML were required to maintain syntactic compatibility, and an approximately 0.5% change to the proof text itself was required to resolve cases where proofs relied on quirky behaviour of the original methods in very specific situations. The total running time for the proof increased from 8 minutes to 19 minutes (run on an i7 quad-core 2.8Ghz iMac with 8GB of memory), indicating that there is certainly room for optimization, but also that the overhead introduced by Eisbach is not insurmountable.

See https://bitbucket.org/makarius/method_definition/get/6f90e104b1a4.zip for the full sources for these methods, with the implementation of Eisbach, the monadic Hoare logic from AutoCorres and several non-trivial examples.

6 Related Work

The relation of proofs versus programs, proof languages versus programming languages, and ultimately the quest for adequate *proof programming languages* opens a vast space of possibilities that have emerged in the past decades, but the general problem is still not settled satisfactorily. Different interactive provers have their own cultural traditions and approaches, and there is often some confusion about basic notions and terminology. Subsequently we briefly sketch important lines of programmable interactive proof assistants in the LCF tradition, which includes the HOL family, Coq, and Isabelle itself.

The original **LCF** proof assistant [7] has pioneered a notion of *tactics* and *tacticals* (i.e. operators on tactics) that can be still seen in its descendants today. An **LCF tactic** is a proof strategy that reduces a goal state to zero or more subgoals that are sufficient to solve the problem. Tactics work in the opposite direction than inferences of the core logic, which take known facts to derive new ones.

This duality of backward reasoning from goals versus forward reasoning from facts is reconciled by *tactic justifications*: a tactic both performs the goal reduction and records an inference for the inverse direction. At the very end of a tactical proof, all justifications are composed like a proof tree, to produce the final theorem. This could result in a late failure to finish the actual proof, e.g. due to programming errors in the tactic implementation.

ML was invented for LCF as the *Meta Language* to implement tactics and other tools around the core logical engine. Proofs are typically written as ML scripts, but the activity of building up new theory content and associated tactics is often hard to distinguish from mere application of existing tools from some library. The bias towards adhoc proof programming is much stronger than in, for instance, Isabelle theories today.

The **HOL** family [15, §1] continues the LCF tradition with ML as the main integrating platform for all activities of theory and tool development (using Standard ML or OCaml today). Due to the universality of ML, it is of course possible to implement different interface languages on the spot. This has been done as various "Mizar modes" to imitate the mathematical proof language of Mizar [15, §2], or as "SSReflect for HOL Light" that has emerged in the Flyspeck project, inspired by SSReflect for Coq [5].

The HOL family has the advantage that explorations of new possibilities are easy to get started on the bare-bones ML top-level interface. HOL Light is particularly strong in its minimalistic approach. In contrast, Isabelle tools need to take substantial system infrastructure and common conveniences for end-users into account.

Coq [15, §4] started as another branch of the LCF family in 1985, but with quite different answers to old questions of how proofs and programs are related. While the HOL systems have replaced LCF's *Logic of Computable Functions* by simply-typed classical set-theory (retaining the key role of the Meta Language), Coq has internalized computational aspects into its type-theoretic logical environment. Consequently, the OCaml substrate of Coq is mainly seen as the system implementation language, and has become difficult to access for Coq users. Implementing some *Coq plug-in* requires separate compilation of OCaml modules which are then linked with the toplevel application. An alternative is to *drop* into an adhoc OCaml shell interactively, but this only works for the bytecode compiler, not the native compiler (preferred by default).

Since Coq can be understood as a dependently-typed functional programming language in its own right, it is natural to delegate more and more proof tool development into it, to achieve a grand-unified formal system eventually. A well-established approach is to use *computational reflection* in order to turn formally specified and proven proof procedures into inferences that don't leave any trace in the proof object. Recent work on Mtac [16] even incorporates a full tactic programming language into Coq itself.

Ltac is the untyped tactic scripting language for Coq [4], and has been successfully applied in large Coq theory developments [2]. It has familiar functional language elements, such as higher order functions and let-bindings. However, it contains imperative elements as well, namely the implicit passing of the *proof goal* as global state. The main functionality of Ltac is provided by a *match* construct for performing both goal and term analysis. Matching performs *proof search* through implicit backtracking across matches, attempting multiple unifications and falling through to other patterns upon failure. Although syntactically similar to the `match` keyword in the term language of Coq, Ltac tactics have a different formal status than Coq functions. Although this serves to distinguish logical function application from on-line computation, it can result in obscure type errors that happen dynamically at run-time.

Mtac is a recently developed *typed tactic language* for Coq [16]. It follows an approach of dependently-typed functional programming: the behaviour of *Mtactics* may be characterized within the logical language of the prover. Mtac is notable by taking the existing language and type-system of Coq (including type-inference), and merely adds a minimal collection of monadic operations to represent impure aspects of tactical programming as first-class citizens: unbounded search, exceptions, and matching against logical syntax. Thus the formerly separate aspect of tactical programming in Ltac is incorporated into the logical language of Coq, which is made even more expressive to provide a uniform basis for all developments of theories, proofs, and proof tools. Thanks to strong static typing, Mtac avoids the dynamic type errors of Ltac.

This mono-cultural approach is quite elegant for Coq, but it relies on the inherent qualities of the Coq logic and its built-in computational world-view. In contrast, the greater LCF family has always embraced multiple languages that serve different purposes: classic LCF-style systems are more relaxed about separating logical foundations from computation outside of it (potentially with access to external tools and services). Eisbach continues this philosophy. In Isabelle, the art of integrating different languages into one system (not one logic) is particularly emphasized: standard syntactic devices for quotation and anti-quotation support embedded sub-languages.

SSReflect [5] is the common label for various tools and techniques for proof engineering in Coq that have emerged from large verification projects by G. Gonthier. This includes a sophisticated *proof scripting language* that provides fine-grained control over moves within the logical subgoal structure, and nested contexts for single-step equational reasoning. Actual *small-scale reflection* refers to implementation techniques within Coq, for propositional manipulations that could be done in HOL-based systems by more elementary means; the experimental SSReflect for HOL-Light re-uses the proof scripting language and its name, but without doing any reflection.

SSReflect emphasizes concrete proof scripts for particular problems, not general proof automation. Scripts written by an expert of SSReflect can be understood by the

same, without stepping through the sequence of goal states in the proof assistant. General tools may be implemented nonetheless, by going into the Coq logic. The SSReflect toolbox includes specific support for generic theory development based on *canonical structures*. More recent work combines that approach with ideas behind Mtac, to internalize a generic proof programming language into Coq, in analogy to the well-known type-class approach of Haskell, see [6].

7 Conclusion and Future Work

In this paper we have presented Eisbach, a high-level language for writing proof methods in Isabelle/Isar. It supports familiar Isar language elements, such as method combinators and theorem attributes, as well as being compatible with existing Isabelle proof methods. An expressive **match** construct enables the use of higher-order matching against facts and subgoals to provide control flow. We showed that existing methods used in large-scale proofs can be easily implemented in Eisbach. The resulting implementations are far smaller, and easier to understand.

Of the proof programming languages mentioned in Section 6, Eisbach purposefully resembles Coq's Ltac most closely. However, it seamlessly integrates with core Isabelle technologies (fact collections, pervasive backtracking, subgoal focusing) to allow powerful methods to be easily and succinctly written. When building on top of Isabelle/Isar, it made most sense to implement an untyped proof programming language, rather than trying to emulate ideas from languages like Mtac. This is because we wanted Eisbach to be able to invoke existing Isar proof methods, which are untyped. While the absence of typed proof procedures hasn't hindered the development of large-scale proofs, the ability to annotate proof methods with information about how they are expected to transform the proof state is potentially attractive. Although higher order methods can approximate run-time method contracts, we would be free to implement arbitrary contract specification languages because proof methods exist outside the logic of Isabelle/Pure, however this avenue of inquiry remains unexplored.

The evaluation demonstrates that Eisbach can already be effectively used to write real-world proof tools, however it still lacks some important features. Firstly, some debugging features are planned, beyond the current solution of manually printing intermediate goal states. Traces of matches and method applications will be presented, ideally with some level of interaction from the user. Additionally more structured language elements would provide a more natural integration with Isar (e.g. explicit subgoal production and addressing). We would also like Eisbach to support parallel evaluation by default. Method combinators outline a certain structure that should be used as a *parallel skeleton* wherever possible. For example, $\mapsto$ could use a parallel version of the underlying tactical `THEN_ALL_NEW`, analogous to the existing `PARALLEL_GOALS` tactical of Isabelle/ML. Ultimately we plan to include Eisbach in a future Isabelle release, with the aim of it becoming the primary means of writing proof methods.

Acknowledgements

We would like to thank Gerwin Klein, who was involved in the discussions on the design of Eisbach and who provided early feedback on this paper. Thanks also to Peter Gammie, Magnus Myreen, and Thomas Sewell for feedback on drafts of this paper.

References

[1] Bourke, T., Daum, M., Klein, G., Kolanski, R.: Challenges and experiences in managing large-scale proofs. In Wenzel, M., ed.: Conferences on Intelligent Computer Mathematics (CICM) / Mathematical Knowledge Management, Springer (2012)

[2] Chlipala, A.: Mostly-automated verification of low-level programs in computational separation logic. ACM SIGPLAN Notices **46**(6) (Jun 2011) 234

[3] Cock, D., Klein, G., Sewell, T.: Secure microkernels, state monads and scalable refinement. In Mohamed, O.A., Muñoz, C., Tahar, S., eds.: 21st TPHOLs. Volume 5170 of LNCS., Montreal, Canada, Springer (Aug 2008) 167–182

[4] Delahaye, D.: A tactic language for the system Coq. In: Int. Conf. Logic for Progr., Artificial Intelligence & Reasoning. Volume 1955 of LNCS., Springer (Nov 2000)

[5] Gonthier, G., Mahboubi, A.: An introduction to small scale reflection in Coq. J. Formalized Reasoning **3**(2) (2010)

[6] Gonthier, G., Ziliani, B., Nanevski, A., Dreyer, D.: How to make ad hoc proof automation less ad hoc. J. Funct. Program. **23**(4) (2013) 357–401

[7] Gordon, M.J.C., Milner, R., Wadsworth, C.P.: Edinburgh LCF: A Mechanized Logic of Computation. LNCS 78. Springer (1979)

[8] Greenaway, D., Andronick, J., Klein, G.: Bridging the gap: Automatic verified abstraction of C. In Beringer, L., Felty, A., eds.: 3rd ITP. Volume 7406 of LNCS., Princeton, New Jersey, Springer (Aug 2012) 99–115

[9] Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. ACM Transactions on Computer Systems (TOCS) (to appear).

[10] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: SOSP, Big Sky, MT, USA, ACM (Oct 2009) 207–220

[11] Murray, T., Matichuk, D., Brassil, M., Gammie, P., Klein, G.: Noninterference for operating system kernels. In Chris Hawblitzel and Dale Miller, ed.: The Second International Conference on Certified Programs and Proofs, Kyoto, Springer (Dec 2012) 126–142

[12] Paulson, L.C.: Isabelle: the next 700 theorem provers. In Odifreddi, P., ed.: Logic and Computer Science. Academic Press (1990)

[13] Wenzel, M., Chaieb, A.: SML with antiquotations embedded into Isabelle/Isar. In Carette, J., Wiedijk, F., eds.: Workshop on Programming Languages for Mechanized Mathematics (PLMMS 2007). Hagenberg, Austria. (June 2007)

[14] Wenzel, M.: Isabelle/Isar—a versatile environment for human-readable formal proof documents. PhD thesis, Technische Universität München (2002)

[15] Wiedijk, F., ed.: The Seventeen Provers of the World. Volume 3600. (2006)

[16] Ziliani, B., Dreyer, D., Krishnaswami, N.R., Nanevski, A., Vafeiadis, V.: Mtac: a monad for typed tactic programming in Coq. In Morrisett, G., Uustalu, T., eds.: ICFP, ACM (2013)