Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Eisbach

A Proof Method Language for Isabelle

Daniel Matichuk · Toby Murray · Makarius Wenzel

Received : date

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 undesirable duplication in large proofs. 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 end-users. We describe the language and the design principles on which it was developed. We evaluate its effectiveness by implementing the most-widely used proof tools in the seL4 verification stack, and consider 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[1] now comprise 0.5 million lines of Isabelle/HOL [11], while Isabelle's Archive of Formal Proofs[2] now comprises over 1.5 million lines. Each of these developments is updated to ensure it runs with each new Isabelle release. The seL4 proofs are also updated as the seL4 kernel evolves, which is an additional source of proof maintenance common to other verified software projects.

The Isabelle proof assistant [19, §6] provides various languages for different purposes. Most commonly used is the Isar language for theory specifications and

D. Matichuk and T. Murray
NICTA, Sydney, Australia
School of Computer Science and Engineering, UNSW, Sydney, Australia

M. Wenzel
http://sketis.net

[1] https://github.com/seL4/l4v
[2] http://afp.sf.net

structured proofs [17]. Isar itself is devoid of computation, but it incorporates arbitrarily complex proof tools called *proof methods*. Proof methods are traditionally written in Isabelle/ML: Standard ML that is integrated into the formal context of Isabelle/Isar, and supports referring to logical entities or Isar elements via *antiquotations* [18]. 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 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 [3]. 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** allows proof methods to be combined, named, and abstracted over terms, facts and other methods. Eisbach is inspired by Coq's Ltac [6], and includes similar features such as matching on facts and the current goal. However, Eisbach's matching behaves differently from Ltac's, especially with respect to backtracking (see Section 3.6). 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.

As a quick motivating example, consider the following lemma which proves a simple property of lists, by induction on the argument list *xs*, and application of the *auto* proof method with an explicit simplification rule passed as its argument.

```
lemma length (xs @ ys) = length xs + length ys
  by (induct xs ; auto simp: append_Nil)
```

Indeed, as anyone who has worked through the first examples in the Isabelle/HOL tutorial [15] can attest, many simple properties of lists are proved using exactly this same procedure, perhaps varying only on the extra simplification rules to be applied by *auto*.

The following simple usage of Eisbach defines a new proof method which generalises this procedure. The method defined identifies a list in the conclusion of the current subgoal and applies induction to it; all newly emerging subgoals are solved with *auto*, with additional simplification rules given as argument.

```
method induct_list uses simps =
  (match conclusion in ?P (x :: 'a list) for x ⇒
    ⟨induct x ; auto simp: simps⟩)
```

Now *induct_list* can be called as a proof method to prove simple properties about lists such as the one above.

```
lemma length (xs @ ys) = length xs + length ys
```

by (*induct_list simps*: *append_Nil*)

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.

A first version of Eisbach was presented in [13]. The present work is a significant refinement of that, with an implementation that has been reworked from the ground up. This implementation was included in the release of Isabelle (May 2015), and others have already taken advantage of Eisbach's ability to simplify proofs by reducing proof duplication and increasing concision [10]. In this paper we purposefully elide a performance evaluation of the current Eisbach implementation, which represents a compromise between performance and ease of integration with existing Isabelle internals. While already good enough for the vast majority of use cases, we expect that Eisbach's performance will improve yet further with future Isabelle releases.

2 Some Isabelle Concepts

Isabelle was originally introduced as another *Logical Framework* by Paulson [16], 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 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, traditionally $\dfrac{A \quad B}{A \wedge B}$, is: $A \Longrightarrow B \Longrightarrow A \wedge B$
- well-founded induction is: $wf\ r \Longrightarrow (\bigwedge x.\ (\bigwedge y.\ (y,\ x) \in r \Longrightarrow P\ y) \Longrightarrow P\ x) \Longrightarrow P\ a$

To understand the structure of Isabelle/Pure rules, one needs to take the *low* syntactic precedence of $\Longrightarrow/\bigwedge$ into account, compared to the embedded connectives $\forall$, $\exists$, $\wedge$, $\vee$, $\longrightarrow$, $\longleftrightarrow$, $\neg$ etc. from the object-logic. Thus the modus-pones rule looks like this: $P \longrightarrow Q \Longrightarrow P \Longrightarrow Q$, and implication introduction like this: $(P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q$.

Isabelle/HOL is a rich library of 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 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 [17] is merely an application of that: it provides particular expressions for human-readable proofs within the generic framework.

Important concepts of the underlying Isabelle/Isar 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 [16], 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.

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 just x) and *goal premises* (here just $A\ x$). Following [16], 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 rule composition 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 current Isabelle the concept of **subgoal focus** achieves that: the proof context is enriched by the fixed term x and the 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 [17], 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, 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 that produces a *thm list* depending on the context.

Attributes. Facts and contexts frequently occur together, and may modify each other by means of attributes, with concrete 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, notably via the **named_theorems** command.

Tactic. Isabelle tactics due to [16] follow the idea behind LCF tactics, but implement the backwards refinement more directly in the logical framework, without the replay tactic justifications (which is still seen in HOL or Coq today). The approach of Isabelle avoids the brittle concentration of primitive inferences when concluding a proof. 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 present-day proof tools use a more restricted set of combinators.

3 Eisbach

Eisbach provides the ability to write automated reasoning procedures to end-users of Isabelle, specifically users only familiar with the use of Isabelle/Isar [17]. Eisbach allows compound proof methods to be named, and to 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 recount some principles of Isar proof methods and then follow the development of a small first-order logic solver in Eisbach, gradually increasing its scope and demonstrating the main language elements.

3.1 Isar Proof Methods

Isar is a document-oriented proof language, focusing on producing and presenting human-readable formal proofs. Such proofs are an 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. Isar proof method invocations come in two forms: *structured* and *unstructured*.[3]

The structured form is "**by** $method_1$ $method_2$", where the initial $method_1$ performs the main backwards refinement of the goal, and the terminal $method_2$ (which is optional) may solve emerging subgoals; the proof is implicitly closed by zero or more *assumption* steps to finish-off trivial subgoals. For example, "**by** (*induct n*) *simp_all*" splits-up a problem by induction and solves the emerging cases by simplification; or "**by** (*rule impI*)" applies a single rule and expects the remaining goal state to be trivial up to unification. It is also possible to leave arguments to the *rule* method implicit, which means that declarations from the context are used instead: "**by** *rule*" is a structured application of a standard rule, which occurs so frequently that it may be abbreviated as "**..**" (two dots).

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 *assumption* steps). After two or three **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 manual inspection of its intermediate goal states.

Isar *method* expressions may combine basic proof methods using method *combinators*. Unlike former LCF tacticals, there is only a minimal repertoire (with built-in backtracking): sequential composition, alternative choice, repeated application. Such methods are used in-place, to address a particular proof problem in a given situation. At the end of each **apply** command, the first successful result from all those produced is retained. The **back** command invokes explicit backtracking: it shifts the list of results from the previous invocation to access the next possible result. Toplevel backtracking is mainly for experimental exploration, not for production proofs.

3.2 Combinators and Backtracking

The following standard combinators for proof methods are available in Isar:

1. Sequential composition of two methods with implicit backtracking: the expression "$method_1$, $method_2$" applies $method_1$, which may produce a set of possible results (new proof goals), before applying $method_2$ to all possible goal states produced by $method_1$. Effectively this produces all results in which the application of $method_1$ followed by $method_2$ is successful.
2. Alternative composition: "$method_1$ | $method_2$" tries $method_1$ and falls through to $method_2$ when $method_1$ fails (yields no results).
3. Suppression of failure: "*method?*" turns failure of *method* into an identity step on the goal state.
4. Repeated method application: "*method*+" repeatedly applies *method* (at least once) until it fails.

[3] This distinction should not be confused with that between structured and unstructured *proofs*: structured proofs usually contain at most one structured method invocation (the final one); unstructured proofs contain few (if any) unstructured method invocations.

The subsequent example illustrates a proof method expression with combinators:

lemma $P \wedge Q \longrightarrow P$
by ((*rule impI*, (*erule conjE*)*?*) | *assumption*)+

Informally, this 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 fails."

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

method $prop\_solver_1$ =
((*rule impI*, (*erule conjE*)*?*) | *assumption*)+

lemma $P \wedge Q \wedge R \longrightarrow P$
by $prop\_solver_1$

3.3 Fact Abstraction

We can generalize $prop\_solver_1$ by abstracting it over the introduction and elimination rules it currently applies. In the previous example, the facts *impI* and *conjE* are static. They are evaluated once when the method is defined and cannot be changed later. This makes the method stable in the sense of *static scoping*: naming another fact *impI* in a later context won't affect the behaviour of $prop\_solver_1$. To instead pass these facts to the method when it is invoked, we can declare some fact-parameters with the **uses** keyword.

method $prop\_solver_2$ **uses** *intros elims* =
((*rule intros*, (*erule elims*)*?*) | *assumption*)+

lemma $P \wedge Q \wedge R \longrightarrow P \wedge (Q \longrightarrow R)$
by ($prop\_solver_2$ *intros*: *impI conjI elims*: *conjE*)

In this particular example, however, providing these rules on each invocation of $prop\_solver_2$ is cumbersome. In the following section we will see how we can create an Eisbach method that is extensible, but also has a database of fact hints that are implicitly used.

3.3.1 Named Theorems

A *named theorem* is a fact whose contents are produced dynamically within the current proof context. The Isar command **named_theorems** provides simple access to this concept: it declares a dynamic fact with corresponding *attribute* (see Section 2) for managing this particular data slot in the context.

named_theorems *intros*

So far *intros* refers to the empty fact. Using the Isar command **declare** we may apply declaration attributes to the context. Below we declare both *conjI* and *impI* as *intros*, adding them to the named theorem slot.

declare *conjI* [*intros*] **and** *impI* [*intros*]

We can refer to named theorems as dynamic facts within a particular proof context, which are evaluated whenever the method is invoked. Instead of explicitly providing these arguments to $prop\_solver_2$ on each invocation, we can instead refer to these named theorems.

named_theorems *elims*
declare *conjE* [*elims*]

method $prop\_solver_3$ =
  ((*rule intros*, (*erule elims*)*?*) | *assumption*)+

lemma $P \wedge Q \longrightarrow P$
  by $prop\_solver_3$

Often these named theorems need to be augmented on the spot, when a method is invoked. The **declares** keyword in the signature of **method** adds the common method syntax *method decl*: *facts* for each named theorem *decl*.

method $prop\_solver_4$ **declares** *intros elims* =
  ((*rule intros*, (*erule elims*)*?*) | *assumption*)+

lemma $P \wedge (P \longrightarrow Q) \longrightarrow Q \wedge P$
  by ($prop\_solver_4$ *elims*: *impE intros*: *conjI*)

3.4 Term Abstraction

Named theorems, as seen in the previous section, are used to provide large collections of managed facts to methods. The **uses** keyword may alternatively be used to abstract a method over facts which are not named theorems. These fact arguments are not declared in the context and thus must be provided on each invocation of a method, as they are empty by default. This can be used, for example, when a fact is meant to perform a specialized task within a method.

Methods can also abstract over terms using the **for** keyword, optionally providing type constraints. For instance, the following proof method *intro_ex* takes a term y of any type, which it uses to instantiate the x-variable of *exI* (existential introduction) before applying the result as a rule. The instantiation is performed here by Isar's *where* attribute. If the current subgoal is to find a witness for the given predicate Q, then this has the effect of committing to y.

method *intro_ex* **for** $Q :: 'a \Rightarrow bool$ **and** $y :: 'a$ =
  (*rule exI* [*where* $P = Q$ **and** $x = y$])

The term parameters y and Q 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 $P\ a \Longrightarrow \exists x.\ P\ x$
  by (*intro_ex* P a)

3.5 Custom Combinators

The built-in method combinators of Isar (see Section 3.2) were originally meant as an exercise in minimalism, without systematic ways to add new combinators. There were also technical restrictions in the representation of proof states and the intentional lack of sub-goal addressing that made it difficult to provide some popular tactic combinators from the past. This has been refined for the purpose of Eisbach as follows.

First of all, there is a new method combinator for *structured concatenation*: "$method_1$; $method_2$" is similar to "$method_1$, $method_2$", but $method_2$ is invoked on on *all* subgoals that have newly emerged from $method_1$. This is useful to handle cases where the number of subgoals produced by a method is determined dynamically at run-time.

method *conj_with* **uses** *rule* =
(*intro conjI* ; *intro rule*)

lemma
assumes *A*: *P*
shows *P* ∧ *P* ∧ *P*
by (*conj_with rule*: *A*)

Moreover, Eisbach method definitions may take other methods as arguments, and thus implement method combinators with prefix syntax. For example, to more usefully exploit Isabelle's backtracking, it can often be useful to require a method to solve all produced subgoals. 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 *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 proof method. Returning to the first-order logic example, the following method definition applies various rules with their canonical methods.

named_theorems *subst*

method *prop_solver* **declares** *intros elims subst* =
(*assumption* |
rule intros | *erule elims* |
subst subst | *subst* (*asm*) *subst* |
(*erule notE* ; *solve* ‹*prop_solver*›))+

The only non-trivial part above 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 combinator 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 $\neg\ Q$ will be chosen for elimination.

Note that the recursive call to *prop_solver* does not have any parameters passed to it. Recall that fact parameters, e.g. *intros*, *elims*, and *subst*, are managed by declarations in the current proof context. They will therefore be passed to any recursive call to *prop_solver* and, more generally, any invocation of a method which declares these named theorems.

After declaring some standard rules to the context, the *prop_solver* becomes capable of solving non-trivial propositional tautologies.

```
lemmas [intros] =
  conjI — P ⟹ Q ⟹ P ∧ Q
  impI — (P ⟹ Q) ⟹ P ⟶ Q
  disjCI — (¬ Q ⟹ P) ⟹ P ∨ Q
  iffI — (P ⟹ Q) ⟹ (Q ⟹ P) ⟹ P ⟷ Q
  notI — (P ⟹ False) ⟹ ¬ P
lemmas [elims] =
  impCE — P ⟶ Q ⟹ (¬ P ⟹ R) ⟹ (Q ⟹ R) ⟹ R
  conjE — P ∧ Q ⟹ (P ⟹ Q ⟹ R) ⟹ R
  disjE — P ∨ Q ⟹ (P ⟹ R) ⟹ (Q ⟹ R) ⟹ R

lemma (A ∨ B) ∧ (A ⟶ C) ∧ (B ⟶ C) ⟶ C
  by prop_solver
```

3.6 Matching

So far we have seen methods defined as simple combinations of other methods. Some familiar programming language concepts have been introduced (i.e. abstraction and recursion). The only control flow has been implicitly the result of backtracking. When designing more sophisticated proof methods this proves too restrictive and too difficult to manage conceptually.

We therefore introduce the *match* method, which provides more direct access to the higher-order matching facility at the core of Isabelle. It is implemented as a separate proof method (in Isabelle/ML), and thus can be directly applied to proofs. However, it is most useful when applied in the context of writing Eisbach method definitions.

Matching allows methods to introspect the goal state, and to implement more explicit control flow. In the basic case, a term or fact *ts* is given to match against as a *match target*, 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.

Consider the following example:

```
lemma
  fixes P
  assumes X:
    Q ⟶ P
    Q
  shows P
    by (match X in I: Q ⟶ P and I': Q ⇒ ‹rule mp [OF I I']›)
```

Here we have a structured Isar proof, with the named assumption X and a conclusion P. With the match method we can find the local facts $Q \longrightarrow P$ and Q, binding

them separately as I and I'. We then specialize the modus-ponens rule $Q \longrightarrow P \Longrightarrow Q \Longrightarrow P$ to these facts to solve the goal. Here we were able to match against an assumption out of the Isar proof state. In general, however, proof subgoals can be *unstructured*, with goal parameters and premises arising from rule application. For example, an unstructured version of the previous proof state would be the Pure implication $\bigwedge P.\ Q \longrightarrow P \Longrightarrow Q \Longrightarrow P$. Here the premises $Q \longrightarrow P$ and Q are unnamed and P is a quantified variable (or goal parameter) rather than a fixed term.

To handle unstructured subgoals, *match* uses *subgoal focusing* (see also Section 4) to produce structured goals out of unstructured ones. In place of fact or term, we may give the keyword **premises** as the match target. This causes a subgoal focus on the first subgoal, lifting local goal parameters to fixed term variables and premises into hypothetical theorems. The match is performed against these theorems, naming them and binding them as appropriate. Similarly giving the keyword **conclusion** matches against the conclusion of the first subgoal.

An unstructured version of the previous example can then be similarly solved through focusing.

lemma $\bigwedge P.\ Q \longrightarrow P \Longrightarrow Q \Longrightarrow P$
  by (*match* **premises in** I: $Q \longrightarrow ?A$ **and** I': $Q \Rightarrow \langle$*rule mp* [*OF I I'*]$\rangle$)

In this example the goal parameter P is first fixed as an anonymous internal term (e.g. $P\_\_$), and then the premises $Q \longrightarrow P\_\_$ and Q are assumed as hypothetical theorems. In the first pattern, the schematic variable *?A* acts as a wildcard, and thus the pattern $Q \longrightarrow ?A$ matches the first premise by matching *?A* to the newly-fixed $P\_\_$. The second pattern then matches the second premise Q, binding it to I'. Finally the inner method is executed, similar to the previous example, and successfully solves the goal.

Match variables may be specified by giving a list of **for**-fixes after the pattern description. These variables are then considered wildcards, similar to schematics. In contrast to schematic variables, however, **for**-fixed terms are bound to the result of the match, and may be referred to inside of the inner method body. In the previous example we could not give $Q \longrightarrow P\_\_$ as a match pattern, because $P\_\_$ cannot be referred to directly. If we want to refer to $P\_\_$ directly we must first bind it with a **for**-fix in a pattern.

lemma $\bigwedge P.\ Q \longrightarrow P \Longrightarrow Q \Longrightarrow P$
  by (*match* **premises in** I: $Q \longrightarrow A$ **and** I': Q **for** A $\Rightarrow$
    $\langle$*match* **conclusion in** $A \Rightarrow \langle$*rule mp* [*OF I I'*]$\rangle\rangle$)

In this example A is a match variable which is effectively bound to the goal parameter P upon a successful match. The inner *match* then matches the now-bound A (bound to P) against the conclusion (also P), finally applying the specialized rule to solve the goal.

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 *solve_ex* =
  (*match* **conclusion in** $\exists x.\ Q\ x$ **for** Q $\Rightarrow$
    $\langle$*match* **premises in** U: $Q\ y$ **for** y $\Rightarrow$
      $\langle$*rule exI* [*where* $P = Q$ *and* $x = y$, *OF U*]$\rangle\rangle$)

The first *match* matches the pattern $\exists\, x.\ Q\ x$ against the current conclusion, binding the term Q in the inner match. Next the pattern $Q\ y$ is matched against all premises of the current subgoal. In this case Q is fixed and y may be instantiated. 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 $exI\colon P\ x \Longrightarrow \exists\, x.\ P\ x$ 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 ⟹ ∃ x. halts x
  by solve_ex
```

Within a match pattern for a fact, each outermost quantifier specifies the requirement that a matching fact must have a schematic variable at that point. This gives a corresponding name to this "slot" for the purposes of forming a static closure, allowing the *where* attribute to perform an instantiation at run-time.

```
lemma
  assumes A: Q ⟹ False
  shows ¬ Q
  by (match intros in X: ⋀P. (P ⟹ False) ⟹ ¬ P ⇒
      ⟨rule X [where P = Q, OF A]⟩)
```

Subgoal focusing converts the outermost quantifiers of premises into schematics when lifting them to hypothetical facts. This allows us to instantiate them with *where* when using an appropriate match pattern.

```
lemma (⋀x :: 'a. A x ⟹ B x) ⟹ A y ⟹ B y
  by (match premises in I: ⋀x :: 'a. ?P x ⟹ ?Q x ⇒
      ⟨rule I [where x = y]⟩)
```

Multiple pattern-method pairs can be given to *match*, separated by a "|". These patterns are considered top-down, executing the inner method m of the first pattern which is satisfied by the current match target. By default, matching performs extensive backtracking by attempting all valid variable and fact bindings according to the given pattern. In particular, all unifiers for a given pattern will be explored, as well as each matching fact. The inner method m will be re-executed for each different variable/fact binding during backtracking. A successful match is considered a cut-point for backtracking. Specifically, once a match is made no other pattern-method pairs will be considered.

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 foo =
  (match conclusion in ?P ∧ ?Q ⇒ ⟨fail⟩ | ?R ⇒ ⟨prop_solver⟩)
```

This behaviour is in direct contrast to the backtracking done by Coq's Ltac [6], 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 necessary for *match* to not mask such failure. In contrast to supplying multiple pattern-method pairs to a single *match*, we can combine multiple invocations of *match* with the "| combinator. This allows inner methods to instead "fall through" upon 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 foo_1 =
  (*match* **conclusion in** *?P* ∧ *?Q* ⇒ ⟨*fail*⟩)
 | (*match* **conclusion in** *?R* ⇒ ⟨*prop_solver*⟩)

Backtracking may be controlled more precisely by marking individual patterns as *cut*. This causes backtracking to not progress beyond this pattern: once a match is found no others will be considered.

method foo_2 =
  (*match* **premises in** *I*: $P \wedge Q$ (*cut*) **and** *I′*: $P \longrightarrow ?U$ **for** $P\ Q$ ⇒
    ⟨*rule mp* [*OF I′ I* [*THEN conjunct1*]]⟩)

In this example, once a conjunction is found ($P \wedge Q$), all possible implications of P in the premises are considered, evaluating the inner *rule* with each consequent. No other conjunctions will be considered, with method failure occurring once all implications of the form $P \longrightarrow ?U$ have been explored. Here the left-right processing of individual patterns is important, as all patterns after of the cut will maintain their usual backtracking behaviour.

lemma $A \wedge B \Longrightarrow A \longrightarrow D \Longrightarrow A \longrightarrow C \Longrightarrow C$
  by foo_2

lemma $C \wedge D \Longrightarrow A \wedge B \Longrightarrow A \longrightarrow C \Longrightarrow C$
  by (foo_2 | *prop_solver*)

In this example, the first lemma is solved by foo_2, by first picking $A \longrightarrow D$ for I', then backtracking and ultimately succeeding after picking $A \longrightarrow C$. In the second lemma, however, $C \wedge D$ is matched first, the second pattern in the match cannot be found and so the method fails, falling through to *prop_solver*.

3.7 Premises within a Subgoal Focus

Subgoal focusing provides a structured form of a subgoal, allowing for more expressive introspection of the goal state. This requires some consideration in order to be used effectively. When the keyword **premises** is given as the match target, the premises of the subgoal are lifted into hypothetical theorems, which can be found and named via match patterns. Additionally these premises are stripped from the subgoal, leaving only the conclusion. This renders them inaccessible to standard proof methods which operate on the premises, such as *frule* (forward reasoning from premises) or *erule* (eliminating/decomposing premises). Naive usage of these methods within a match will most likely not function as the method author intended.

method *my_allE_bad* **for** $y :: 'a$ =
  (*match* **premises in** *I*: $\forall x :: 'a.\ ?Q\ x$ ⇒
    ⟨*erule allE* [*where* $x = y$]⟩)

Here we take a single parameter y and specialize the universal elimination rule $(\forall x.\ P\ x \Longrightarrow (P\ x \Longrightarrow R) \Longrightarrow R)$ to it, then attempt to apply this specialized rule with *erule*. The method *erule* will attempt to unify with a universal quantifier in the premises that matches the type of y. Suppose we tried to use *my_allE_bad* to prove a trivial lemma the following:

```
lemma ∀x. P x ⟹ P a
  apply (my_allE_bad a)?
  oops
```

When *my_allE_bad* is invoked, since **premises** causes a focus, the premise $\forall x.\ P\ x$ is nowhere to be found, and thus *my_allE_bad* will always fail.[4] If focusing instead left the premises in place, using methods like *erule* would lead to unintended behaviour, specifically during backtracking. In *my_allE_bad*, *erule* could choose an alternate premise while backtracking, while leaving I bound to the original match. In the case of more complex inner methods, where either I or bound terms are used, this would almost certainly not be the intended behaviour.

An alternative implementation would be to specialize the elimination rule to the bound term and apply it with *rule* instead of *erule*.

```
method my_allE_almost for y :: 'a =
  (match premises in I: ∀x :: 'a. ?Q x ⇒
    ⟨rule allE [where x = y, OF I]⟩)
```

This method will insert a specialized duplicate of a universally quantified premise. Although this will successfully apply in the presence of such a premise, it is not likely the intended behaviour. To understand why, consider the following example:

```
lemma ∀x. P x ⟹ P a
 apply (my_allE_almost a)
 apply (my_allE_almost a)
 by assumption
```

Here, after applying *my_allE_almost*, the goal state is: $\forall x.\ P\ x \Longrightarrow P\ a \Longrightarrow P\ a$. Observe that the premise $P\ a$ has been inserted as intended, but that the original premise $\forall x.\ P\ x$ still remains. A second application of *my_allE_almost* therefore succeeds, yielding a goal state of $\forall x.\ P\ x \Longrightarrow P\ a \Longrightarrow P\ a \Longrightarrow P\ a$. Repeated application of *my_allE_almost* would thus produce an infinite stream of duplicate specialized premises, due to the original premise never being removed. To address this, matched premises may be declared with the *thin* attribute. This will hide the premise from subsequent inner matches, and remove it from the list of premises after the inner method has finished. It can be considered analogous to the old-style *thin_tac*, used for removing goal premises that match a given pattern.

To complete our example, the correct implementation of the method will *thin* the premise from the match and then apply it to the specialized elimination rule.

```
method my_allE for y :: 'a =
  (match premises in I [thin]: ∀x :: 'a. ?Q x ⇒
    ⟨rule allE [where x = y, OF I]⟩)

lemma ∀x. P x ⟹ ∀x. Q x ⟹ P y ∧ Q y
  by (my_allE y)+ (rule conjI)
```

[4] This is why we need to use the *?* combinator in this example and the **oops** keyword to terminate an unfinished proof.

Other attributes may also be applied to matched facts. This is most applicable when focusing, in order to inform methods which would otherwise use premises implicitly.

lemma $A \longleftrightarrow B \Longrightarrow (A \longrightarrow B) \wedge (B \longrightarrow A)$
by (*match* **premises in** *I* [*subst*]: $?P \longleftrightarrow ?Q \Rightarrow$ ⟨*prop_solver*⟩)

In this example, the pattern $?P \longleftrightarrow ?Q$ matches against the premise $A \longleftrightarrow B$ and binds it to the local fact *I*. Additionally it declares this fact as a *subst* rule, adding it to the *subst* named theorem for the duration of the match. This is then implicitly used by *prop_solver* to solve the goal.

3.8 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 matched universal quantifier is marked as *thin* to remove it from the premises, while using the universal elimination rule *allE* to specialize U to y. The same matching is also performed against the conclusion to find possible instantiations there as well.

method *guess_all* =
(*match* **premises in** *U* [*thin*]: $\forall x.\ P\ (x :: 'a)$ **for** $P \Rightarrow$
⟨(*match* **premises in** $?H\ (y :: 'a)$ **for** $y \Rightarrow$
⟨*rule allE* [*where* $x = y$, *OF U*]⟩)*?*,
(*match* **conclusion in** $?H\ (y :: 'a)$ **for** $y \Rightarrow$
⟨*rule allE* [*where* $x = y$, *OF U*]⟩)*?*⟩)

The 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, *match* considers only those matches that bind all **for**-fixed variables mentioned in the pattern.

The inner-match must be duplicated over both the premises and conclusion because they are logically different entities: the premises are *facts*, in that they are (assumed) true; the conclusion is not and must be proved, and so is a *term*. This might look strange to users of Coq's Ltac, where these notions are identified; however, it does not limit the expressivity of Eisbach.

The *thin* attribute is necessary here in order to guarantee termination (see Section 3.6). However, since the premise is "consumed", care must be taken to ensure that this does not render the goal unsolvable (i.e. in the case where the premise needs to be specialized multiple times). Here we assume that this is handled by a previous application of *prop_solver*, which decomposes the goal into sufficiently small subgoals such that only a single instantiation is required.

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 guess_ex =
  (match conclusion in
    ∃x. P (x :: 'a) for P ⇒
      ⟨match premises in ?H (x :: 'a) for x ⇒
        ⟨rule exI [where x = x and P = P]⟩⟩)
```

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

```
method fol_solver =
  ((prop_solver | guess_ex | guess_all) ; solve ⟨fol_solver⟩)
```

The use of *solve* above ensures that the recursive subgoals are solved. Without it, the recursive call could terminate prematurely 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 example problems.

```
declare
  allI [intros] — (⋀x. P x) ⟹ ∀x. P x
  exE [elims] — ∃x. P x ⟹ (⋀x. P x ⟹ Q) ⟹ Q
  ex_simps [subst]
  all_simps [subst]

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 Readable Proof Methods

In Isar there is a clear distinction between a structured and an unstructured proof. The former makes use of the rich reasoning framework provided by Isar, while the latter relies more heavily on the implicit behaviour of proof methods. An unstructured proof cannot be understood without checking the proof in Isabelle and inspecting the subgoal state at each stage of the proof. This can create significant issues during proof maintenance phases, where a proof needs to be updated in response to an update to a specification or to Isabelle itself. The original intention of a proof cannot be easily extracted from the unstructured proof script, and a now-failing proof may require significant time and effort to perform the necessary archaeological exploration of its history to recover some insight.

One of the aims of Eisbach is to address this by providing a means to describe reasoning procedures. A proof method designed to solve a particular class of problems serves as a better record of the author's intent than an ad-hoc series of general tools. Arguably this simply shifts the problem of *proof maintenance* to that of *proof method maintenance*, and indeed this is a well-known concern in Ltac today. This indicates the necessity for writing proof methods that are *readable* and thus also maintainable.

In Eisbach, *match* can be considered as a structured language element, and is meant to serve both as implementation and documentation. Many methods shown here could have been implemented without using *match*, but would have been significantly more difficult to understand, and may happen to work in unintended cases. In practice, a match pattern is a much more explicit description of the expected goal state than, for example, the expectation that *erule* successfully finds an appropriate premise for the given rule. As with Isar, Eisbach method authors are free to use as much structure as they consider necessary for their specific application.

4.2 Static Closure of Concrete Syntax

Isabelle provides a rich selection of powerful proof methods, each with its own concrete syntax, which is implemented by parser combinators in ML. Additionally, Isabelle's theorem attributes, which perform context and fact transformations, have their own parsers. Rather than re-write all tools from the libraries to support Eisbach, we build on existing features of the Isabelle parsing framework whereby tokens have values (types, terms, facts etc.) assigned to them implicitly during parsing.

This syntax/value assignment mechanism was originally introduced to support *locale expressions* in the sense of [1]. Thus expressions over facts and attributes became transformable by morphisms, to move them from an abstract locale context to a concrete application context.[5]

The same principle of syntax closure and interpretation is now the main workhorse of Eisbach. After some modifications, it works for method expressions as well, including their embedded facts and attribute expressions. For example, the basic method "(*simp add*: *foo* [*OF bar*])" is wrapped up as static closure, where the embedded fact expression "*foo* [*OF bar*]" is treated like a pre-evaluated constant.

Eisbach then simply serves as an interpretation environment for the carefully prepared method syntax tokens. When a proof method is applied Eisbach instantiates these token values appropriately (via some morphism), 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 required various modifications of the Isar implementation itself, this proves to be a very effective solution to performing this kind of language extension.

[5] See also [2] for a recent exposition of the possibilities of locales and locale interpretations via morphisms in Isabelle.

4.3 Subgoal Focusing

In Isabelle/Pure there is a logical distinction between universally quantified parameters (like x in $\bigwedge x.\ P\ x$) and arbitrary-but-fixed terms (like a in $P\ a$). A subgoal in the former form does not allow the x to be explicitly referenced, because it is hidden within a closed formula; for example, *my_fact* [*where* $y = x$] does not produce a valid theorem. Historically, some special tactics were provided to descend into the sub-goal structure and provide ad-hoc access to its local parameters: these are available in Isar via so-called *improper methods* (like *rule_tac*).

Likewise, premises within a subgoal are not yet 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 $A \Longrightarrow \text{¤}$ of a subgoal is not accessible to structured reasoning.

The Isar proof language accommodates this situation by a canonical nested proof consisting of **fix**/**assume**/**show** of as follows:

have $\bigwedge x.\ A\ x \Longrightarrow B\ x$
proof −
  fix x
  assume $A\ x$
  show $B\ x$ ⟨*proof*⟩
qed

Isabelle/ML already provides systematic support for this *subgoal focusing*. For Eisbach, we incorporated that into the language with some concrete syntax, to allow the user to write methods that can operate within the local subgoal structure as required. Focusing creates a new goal out of a given subgoal, but with its parameters turned into fixed variables (actual terms), and premises into local assumptions (actual facts). This allows for uniform treatment of the goal state when matching and parameter passing. In Eisbach's *match* method, focusing is accessed by specifying the keyword **conclusion** or **premises** as the match target (see Section 3.6).

5 Application and Evaluation

To evaluate Eisbach we re-implemented two existing proof methods: *wp* and *wpc*, which are Verification Condition Generators currently released as part of the seL4 proof. They were used extensively in the full functional correctness proof of seL4 [12] for both invariant and refinement proofs. They were originally designed for performing "weakest-precondition" style reasoning against a shallowly embedded monadic Hoare logic [5]. The intelligence of these methods lies in their large collection of stored facts, and have proven to be more generally useful in other projects [14].

Together these two methods comprise 500 lines of Isabelle/ML, and 60 lines of Isabelle/Isar for setup. However, we implement them (below) in Eisbach almost trivially.

The Eisbach implementation of *wp* is nothing more than 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.

```
named_theorems wp and wp_comb and wp_split

method wp declares 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. Rather than a named theorem or keyword, the given match target is an ML expression. This expression uses a custom attribute *get_split_rule* (defined in ML) that retrieves the *case-split rule* for a given term. Such rules are used to decompose case distinctions on datatypes. The *apply_split* method simply applies the retrieved case-split rule after decomposing it into an implication.

```
attribute_setup get_split_rule =
  ‹Args.term >> (fn t =>
    Thm.rule_attribute (fn context => fn _ =>
      (case get_split_rule (Context.proof_of context) t of
        SOME thm => thm
      | NONE => Drule.dummy_thm)))›

method apply_split for f :: 'a and R :: 'a ⇒ bool=
  (match [[get_split_rule f]] in U: (?x :: bool) = ?y ⇒
    ‹match U[THEN iffD2] in U': ⋀H. ?A ⟹ H (?z :: 'c) ⇒
      ‹match (R) in R' :: 'c ⇒ bool for R' ⇒
        ‹rule U'[where H=R']››››)
```

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

```
method repeat_new methods m = (m ; (repeat_new ‹m›)?)
```

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

```
method wpc' declares wpc_helper =
  (rule wpc_helperI,
    repeat_new ‹rule wpc_processor› ; (rule wpc_helper))
```

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

```
method wpc =
  (match conclusion in
    ⦃P⦄ f ⦃Q⦄ for f P Q ⇒
      ‹apply_split f λf. ⦃P⦄ f ⦃Q⦄›
  | ⦃P⦄ f ⦃Q⦄,⦃E⦄ for f P Q E ⇒
      ‹apply_split f λf. ⦃P⦄ f ⦃Q⦄,⦃E⦄›, wpc')
```

Together, combined with a large body of existing lemmas, these methods calculate weakest-precondition style proof obligations for the monadic Hoare logic of [5]. Additionally, with appropriate lemmas and some additional match conditions for *wpc*, these methods are easily extended to other calculi such as the relational Hoare logic from [14].

To demonstrate the effectiveness of these re-implemented methods, we re-ran the invariant proofs for the seL4 abstract functional specification[6] 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.

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 was never 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 [9] 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 was much stronger in LCF than in Isabelle theories today.

The **HOL** family [19, §1] continues the LCF tradition with ML as the main integrating platform for all activities of theory and proof tool development (using Standard ML or OCaml). 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 [19, §2], or as "SSReflect for HOL Light" that has emerged in the Flyspeck project, inspired by SSReflect for Coq [7].

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 partic-

[6] See https://github.com/NICTA/l4v/tree/master/proof/invariant-abstract

ularly strong in its minimalistic approach. In contrast, Isabelle tools need to take substantial system infrastructure and common conveniences for end-users into account, before anything new can be added.

Coq [19, §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 bootstrap 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 (which is 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 [20] even incorporates a full tactic programming language into Coq itself.

Ltac is the untyped tactic scripting language for Coq [6], and has been successfully applied in large Coq theory developments [4]. 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.

SSReflect [7] is the common label for various tools and techniques for proof engineering in Coq that have emerged from large verification projects by 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 (this is neither possible nor required in HOL).

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*.

Mtac is a recently developed *typed tactic language* for Coq [20]. 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. More recent work combines Mtac with SSReflect [8], to internalize a generic proof programming language into Coq, in analogy to the well-known type-class approach of Haskell.

This uniform proof language approach is quite elegant for Coq, but it relies on the inherent qualities of the Coq logic and its built-in computational approach. 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 programs or network services. Eisbach continues this philosophy of extra-logical additions to an existing system. 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 easily.

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* method 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 resembles Coq's Ltac most closely, which was done on purpose. However, it seamlessly integrates with core Isabelle technologies (fact collections, pervasive backtracking, subgoal focusing) to allow powerful proof 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, like *solve* (see Section 3.5), 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 further research remains unexplored so far.

One of Eisbach's greatest virtues is that it provides a framework for thinking of proof methods like programming language elements. This applies to methods written in Eisbach, like *solve*, but also to those written in Isabelle/ML. A good example of the latter is the *match* method which, while its implementation is entirely independent of the **method** command, became necessary to implement only in the presence of Eisbach. The method language of Isar was already arbitrarily expressive, as methods define their own syntax. Eisbach now opens up this space allowing users to write methods that serve as elements of a high-level *method programming language*, rather than one-off proof tools. Thus methods that may have had little use in proof scripts now become useful, and in the case of *match* incredibly powerful, as elements of Eisbach-defined proof methods.

Eisbach can already be effectively used to write real-world proof tools, although 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, the ";" combinator could use a parallel version of the underlying tactical `THEN_ALL_NEW`, analogous to the existing `PARALLEL_GOALS` tactical of Isabelle/ML.

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.

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

References

1. Ballarin, C.: Locales and locale expressions in Isabelle/Isar. In: S. Berardi, M. Coppo, F. Damiani (eds.) Types for Proofs and Programs (TYPES 2003), *Lecture Notes in Computer Science*, vol. 3085. Springer (2003). DOI 10.1007/978-3-540-24849-1_3
2. Ballarin, C.: Locales: A module system for mathematical theories. Journal of Automated Reasoning **52**(2), 123–153 (2014). DOI 10.1007/s10817-013-9284-7
3. Bourke, T., Daum, M., Klein, G., Kolanski, R.: Challenges and experiences in managing large-scale proofs. In: M. Wenzel (ed.) Conferences on Intelligent Computer Mathematics (CICM) / Mathematical Knowledge Management. Springer (2012). DOI 10.1007/978-3-642-31374-5_3
4. Chlipala, A.: Mostly-automated verification of low-level programs in computational separation logic. ACM SIGPLAN Notices **46**(6), 234 (2011). DOI 10.1145/1993316.1993526

5. Cock, D., Klein, G., Sewell, T.: Secure microkernels, state monads and scalable refinement. In: O.A. Mohamed, C. Muñoz, S. Tahar (eds.) 21st TPHOLs, *LNCS*, vol. 5170, pp. 167–182. Springer, Montreal, Canada (2008). DOI 10.1007/978-3-540-71067-7_16
6. Delahaye, D.: A tactic language for the system Coq. In: Int. Conf. Logic for Progr., Artificial Intelligence & Reasoning, *LNCS*, vol. 1955. Springer (2000). DOI 10.1007/3-540-44404-1_7
7. Gonthier, G., Mahboubi, A.: An introduction to small scale reflection in Coq. J. Formalized Reasoning **3**(2) (2010). DOI 10.6092/issn.1972-5787/1979
8. Gonthier, G., Ziliani, B., Nanevski, A., Dreyer, D.: How to make ad hoc proof automation less ad hoc. J. Funct. Program. **23**(4), 357–401 (2013). DOI 10.1017/S0956796813000051
9. Gordon, M.J.C., Milner, R., Wadsworth, C.P.: Edinburgh LCF: A Mechanized Logic of Computation. LNCS 78. Springer (1979). DOI 10.1007/3-540-09724-4
10. Hölzl, J., Lochbihler, A., Traytel, D.: A formalized hierarchy of probabilistic system types. In: C. Urban, X. Zhang (eds.) Interactive Theorem Proving, *Lecture Notes in Computer Science*, vol. 9236, pp. 203–220. Springer International Publishing (2015). DOI 10.1007/978-3-319-22102-1_13. URL http://dx.doi.org/10.1007/978-3-319-22102-1_13
11. 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) **32**(1), 2 (2014). DOI 10.1145/2560537
12. 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, pp. 207–220. ACM, Big Sky, MT, USA (2009). DOI 10.1145/1629575.1629596
13. Matichuk, D., Wenzel, M., Murray, T.: An Isabelle proof method language. In: G. Klein, R. Gamboa (eds.) Interactive Theorem Proving — 5th International Conference, ITP 2014, Vienna, Austria, *Lecture Notes in Computer Science*, vol. 8558. Springer (2014). DOI 10.1007/978-3-319-08970-6_25
14. 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, pp. 126–142. Springer, Kyoto (2012). DOI 10.1007/978-3-642-35308-6_12
15. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, *Lecture Notes in Computer Science*, vol. 2283. Springer Verlag (2002). DOI 10.1007/3-540-45949-9
16. Paulson, L.C.: Isabelle: the next 700 theorem provers. In: P. Odifreddi (ed.) Logic and Computer Science. Academic Press (1990)
17. Wenzel, M.: Isabelle/Isar — a versatile environment for human-readable formal proof documents. Ph.D. thesis, Technische Universität München (2002)
18. Wenzel, M., Chaieb, A.: SML with antiquotations embedded into Isabelle/Isar. In: J. Carette, F. Wiedijk (eds.) Workshop on Programming Languages for Mechanized Mathematics (PLMMS 2007). Hagenberg, Austria (2007)
19. Wiedijk, F. (ed.): The Seventeen Provers of the World, vol. 3600 (2006). DOI 10.1007/11542384_1
20. Ziliani, B., Dreyer, D., Krishnaswami, N.R., Nanevski, A., Vafeiadis, V.: Mtac: a monad for typed tactic programming in Coq. In: G. Morrisett, T. Uustalu (eds.) ICFP. ACM (2013). DOI 10.1017/S0956796813000051