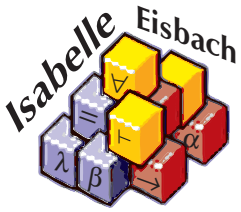


The Eisbach User Manual

Daniel Matichuk
Makarius Wenzel
Toby Murray

18 January 2026

Preface

Eisbach is a collection of tools which form the basis for defining new proof methods in Isabelle/Isar [2]. It can be thought of as a "proof method language", but is more precisely an infrastructure for defining new proof methods out of existing ones.

The core functionality of Eisbach is provided by the Isar **method** command. Here users may define new methods by combining existing ones with the usual Isar syntax. These methods can be abstracted over terms, facts and other methods, as one might expect in any higher-order functional language.

Additional functionality is provided by extending the space of methods and attributes. The new *match* method allows for explicit control-flow, by taking a match target and a list of pattern-method pairs. By using the functionality provided by Eisbach, additional support methods can be easily written. For example, the *catch* method, which provides basic try-catch functionality, only requires a few lines of ML.

Eisbach enables users to implement automated proof tools conveniently via Isabelle/Isar syntax. This is in contrast to the traditional approach to use Isabelle/ML (via **method_setup**), which poses a higher barrier-to-entry for casual users.

This manual is written for readers familiar with Isabelle/Isar, but not necessarily Isabelle/ML. It covers the usage of the **method** as well as the *match* method, as well as discussing their integration with existing Isar concepts such as **named_theorems**.

These commands are provided by theory *HOL−Eisbach.Eisbach*: it needs to be imported by all Eisbach applications. Theory *HOL−Eisbach.Eisbach_Tools* provides additional proof methods and attributes that are occasionally useful.

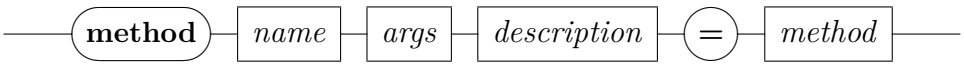
Contents

Chapter 1

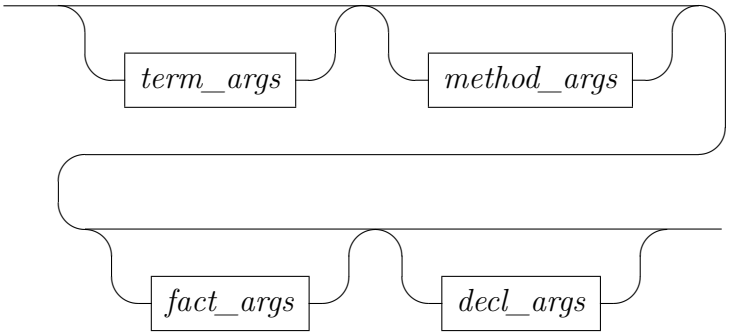
The method command

The **method** command provides the ability to write proof methods by combining existing ones with their usual syntax. Specifically it 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. They may also provide an optional text description for display in **print_methods**.

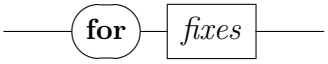
The syntax diagram below refers to some syntactic categories that are further defined in [1].



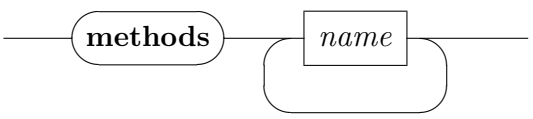
args



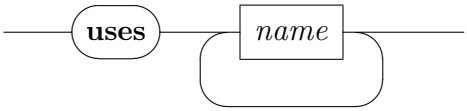
term_args



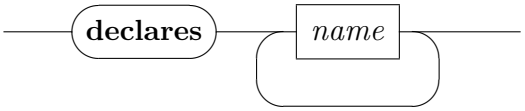
method_args



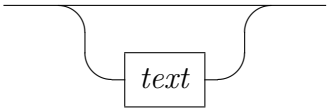
fact_args



decl_args



description



1.1 Basic method definitions

Consider the following proof that makes use of usual Isar method combinators.

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

It is clear that this compound method will be applicable in more cases than this proof alone. 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 $prop\_solver_1$

In this 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 $prop\_solver_1$.

The following example defines the same method and gives it a description for the **print_methods** command.

method *prop_solver*$_2$ ‹*solver for propositional formulae*› =
((*rule impI*, (*erule conjE*)*?*) | *assumption*)+

1.2 Term abstraction

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)

1.3 Fact abstraction

1.3.1 Named theorems

A *named theorem* is a fact whose contents are produced dynamically within the current proof context. The Isar command **named_theorems** declares a dynamic fact with a corresponding *attribute* of the same name. This allows to maintain a collection of facts in the context as follows:

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

having facts hard-coded into the method, as in $prop\_solver_1$, 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*)

1.3.2 Simple fact abstraction

The **declares** keyword requires that a corresponding dynamic fact has been declared with **named_theorems**. This is useful for managing collections of facts which are to be augmented with declarations, but is overkill if we simply want to pass a fact to a method.

We may use the **uses** keyword in the method header to provide a simple fact parameter. In contrast to **declares**, these facts are always implicitly empty unless augmented when the method is invoked.

method *rule_twice* **uses** *my_rule* =
(*rule my_rule*, *rule my_rule*)

lemma $P \Longrightarrow Q \Longrightarrow (P \wedge Q) \wedge Q$
by (*rule_twice my_rule*: *conjI*)

1.4 Higher-order methods

The *structured concatenation* combinator "$method_1$; $method_2$" was introduced in Isabelle2015, motivated by the development of Eisbach. It is similar to "$method_1$, $method_2$", but $method_2$ is invoked 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 $P \wedge P \wedge P$
by (*conj_with rule*: *A*)

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, 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 *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.

1.5 Example

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 $\neg\ P$ 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 — $P \Longrightarrow Q \Longrightarrow P \wedge Q$
  impI — $(P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q$
  disjCI — $(\neg\ Q \Longrightarrow P) \Longrightarrow P \vee Q$
  iffI — $(P \Longrightarrow Q) \Longrightarrow (Q \Longrightarrow P) \Longrightarrow P \longleftrightarrow Q$
  notI — $(P \Longrightarrow \mathit{False}) \Longrightarrow \neg\ P$

lemmas [*elims*] =
  impCE — $P \longrightarrow Q \Longrightarrow (\neg\ P \Longrightarrow R) \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R$
  conjE — $P \wedge Q \Longrightarrow (P \Longrightarrow Q \Longrightarrow R) \Longrightarrow R$
  disjE — $P \vee Q \Longrightarrow (P \Longrightarrow R) \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R$

lemma $(A \vee B) \wedge (A \longrightarrow C) \wedge (B \longrightarrow C) \longrightarrow C$
  by *prop_solver*

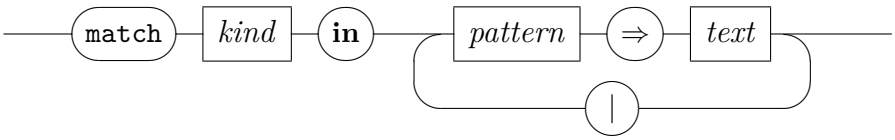
Chapter 2

The match method

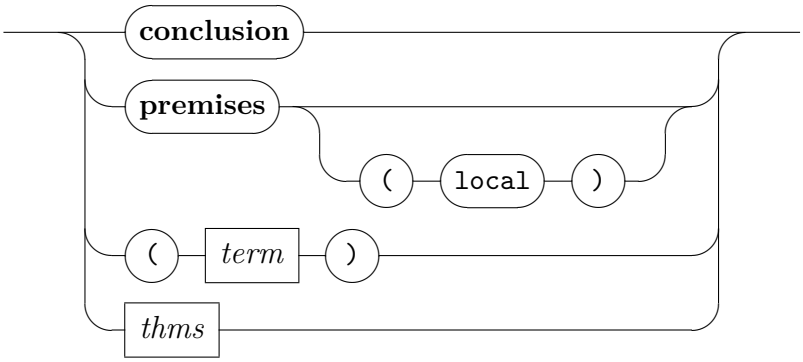
So far we have seen methods defined as simple combinations of other methods. Some familiar programming language concepts have been introduced (i.e. abstraction and recursion). The only control flow has been implicitly the result of backtracking. When designing more sophisticated proof methods this proves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

To address this, we 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.

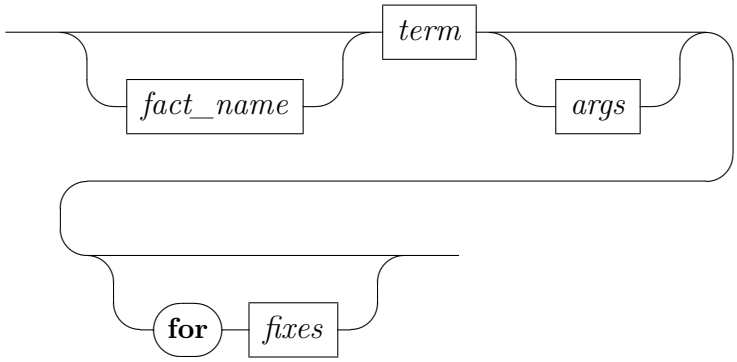
The syntax diagram below refers to some syntactic categories that are further defined in [1].



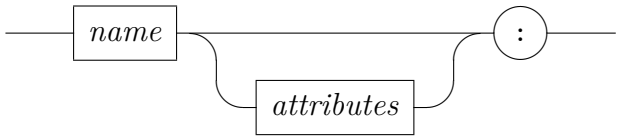
kind



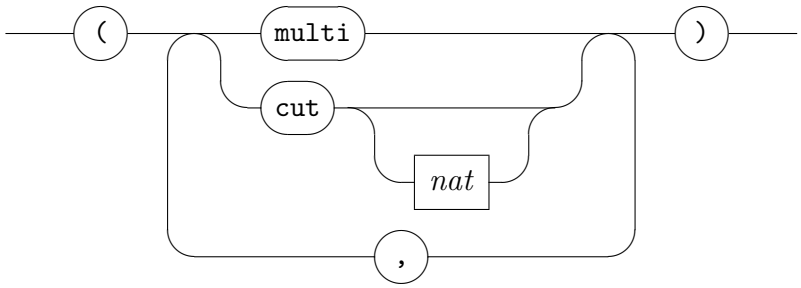
pattern



fact_name



args



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.

lemma
  assumes *X*:
    $Q \longrightarrow P$
    Q
  shows P
    by (*match X* **in** *I*: $Q \longrightarrow P$ **and** I': $Q \Rightarrow$ ‹*insert mp* [*OF I I'*]›)

In this example 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 to 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.

2.1 Subgoal focus

In the previous example 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. To address this, *match* uses *subgoal focusing* 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 $Q \longrightarrow P \Longrightarrow Q \Longrightarrow P$
by (*match* **premises in**
I: $Q \longrightarrow P$ **and** I': $Q \Rightarrow$ ‹*insert mp* [*OF I I'*]›)

Match variables may be specified by giving a list of **for**-fixes after the pattern description. This marks those terms as bound variables, which may be used in the method body.

lemma $Q \longrightarrow P \Longrightarrow Q \Longrightarrow P$
by (*match* **premises in** I: $Q \longrightarrow A$ **and** I': Q **for** $A \Rightarrow$
‹*match conclusion in* $A \Rightarrow$ ‹*insert mp* [*OF I I'*]››)

In this example A is a match variable which is bound to 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.

Schematic terms like *?P* may also be used to specify match variables, but the result of the match is not bound, and thus cannot be used in the inner method body.

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** $Q \Rightarrow$
‹*match premises in U*: $Q\ y$ *for* $y \Rightarrow$
‹*rule exI* [*where* $P = Q$ *and* $x = y$, *OF U*]››)

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*: $P\ x \Longrightarrow \exists\, x.\ P\ x$ 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*

2.1.1 Operating within a 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* or *erule*. 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* $x \Rightarrow$
‹*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. Since **premises** causes a focus, however, there are no subgoal premises to be found and thus *my_allE_bad* will always fail. If focusing instead left the premises in place, using methods like *erule* would lead to unintended behaviour, specifically during backtracking. In our example, *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 directly.

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

lemma ∀ x. P x ⟹ P y
  by (my_allE_almost y)
```

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. Repeated application of this method will 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 when the inner method has finished and the subgoal is unfocused. It can be considered analogous to the existing *thin_tac*.

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)
```

Inner focusing

Premises are *accumulated* for the purposes of subgoal focusing. In contrast to using standard methods like *frule* within focused match, another *match* will have access to all the premises of the outer focus.

```
lemma A ⟹ B ⟹ A ∧ B
  by (match premises in H: A ⇒ ‹intro conjI, rule H,
      match premises in H': B ⇒ ‹rule H'››)
```

In this example, the inner *match* can find the focused premise B. In contrast, the *assumption* method would fail here due to B not being logically accessible.

lemma $A \Longrightarrow A \wedge (B \longrightarrow B)$
by (*match* **premises in** H: $A \Rightarrow$ ‹*intro conjI*, *rule H*, *rule impI*,
match premises (*local*) *in* $A \Rightarrow$ ‹*fail*›
| H': $B \Rightarrow$ ‹*rule* H'››)

In this example, the only premise that exists in the first focus is A. Prior to the inner match, the rule *impI* changes the goal $B \longrightarrow B$ into $B \Longrightarrow B$. A standard premise match would also include A as an original premise of the outer match. The *local* argument limits the match to newly focused premises.

2.2 Attributes

Attributes may throw errors when applied to a given fact. For example, rule instantiation will fail if there is a type mismatch or if a given variable doesn't exist. Within a match or a method definition, it isn't generally possible to guarantee that applied attributes won't fail. For example, in the following method there is no guarantee that the two provided facts will necessarily compose.

method *my_compose* **uses** *rule*1 *rule*2 =
(*rule rule*1 [*OF rule*2])

Some attributes (like *OF*) have been made partially Eisbach-aware. This means that they are able to form a closure despite not necessarily always being applicable. In the case of *OF*, it is up to the proof author to guard attribute application with an appropriate *match*, but there are still no static guarantees.

In contrast to *OF*, the *where* and *of* attributes attempt to provide static guarantees that they will apply whenever possible.

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: $Q \Longrightarrow False$
shows $\neg\ Q$
by (*match intros* **in** X: $\bigwedge P.\ (P \Longrightarrow False) \Longrightarrow \neg\ P \Rightarrow$
‹*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 $(\bigwedge x :: 'a.\ A\ x \Longrightarrow B\ x) \Longrightarrow A\ y \Longrightarrow B\ y$
by (*match* **premises in** *I*: $\bigwedge x :: 'a.\ ?P\ x \Longrightarrow ?Q\ x \Rightarrow$
‹*rule I* [*where* $x = y$]›)

The *of* attribute behaves similarly. It is worth noting, however, that the positional instantiation of *of* occurs against the position of the variables as they are declared *in the match pattern*.

lemma
fixes $A\ B$ **and** $x :: 'a$ **and** $y :: 'b$
assumes *asm*: $(\bigwedge x\ y.\ A\ y\ x \Longrightarrow B\ x\ y\ )$
shows $A\ y\ x \Longrightarrow B\ x\ y$
by (*match asm* **in** *I*: $\bigwedge(x :: 'a)\ (y :: 'b).\ ?P\ x\ y \Longrightarrow ?Q\ x\ y \Rightarrow$
‹*rule I* [*of x y*]›)

In this example, the order of schematics in *asm* is actually *?y ?x*, but we instantiate our matched rule in the opposite order. This is because the effective rule *I* was bound from the match, which declared the $'a$ slot first and the $'b$ slot second.

To get the dynamic behaviour of *of* we can choose to invoke it *unchecked*. This avoids trying to do any type inference for the provided parameters, instead storing them as their most general type and doing type matching at run-time. This, like *OF*, will throw errors if the expected slots don't exist or there is a type mismatch.

lemma
fixes $A\ B$ **and** $x :: 'a$ **and** $y :: 'b$
assumes *asm*: $\bigwedge x\ y.\ A\ y\ x \Longrightarrow B\ x\ y$
shows $A\ y\ x \Longrightarrow B\ x\ y$
by (*match asm* **in** *I*: *PROP ?P* $\Rightarrow$ ‹*rule I* [*of* (*unchecked*) *y x*]›)

Attributes may be applied to matched facts directly as they are matched. Any declarations will therefore be applied in the context of the inner method, as well as any transformations to the rule.

lemma $(\bigwedge x :: 'a.\ A\ x \Longrightarrow B\ x) \Longrightarrow A\ y \longrightarrow B\ y$
by (*match* **premises in** *I* [*of y*, *intros*]: $\bigwedge x :: 'a.\ ?P\ x \Longrightarrow ?Q\ x \Rightarrow$
‹*prop_solver*›)

In this example, the pattern $\bigwedge x :: 'a.\ ?P\ x \Longrightarrow ?Q\ x$ matches against the only premise, giving an appropriately typed slot for *y*. After the match, the

resulting rule is instantiated to y and then declared as an *intros* rule. This is then picked up by *prop_solver* to solve the goal.

2.3 Multi-match

In all previous examples, *match* was only ever searching for a single rule or premise. Each local fact would therefore always have a length of exactly one. We may, however, wish to find *all* matching results. To achieve this, we can simply mark a given pattern with the (*multi*) argument.

```
lemma
  assumes asms: A ⟹ B  A ⟹ D
  shows (A ⟶ B) ∧ (A ⟶ D)
  apply (match asms in I [intros]: ?P ⟹ ?Q  ⇒ ‹solves ‹prop_solver››)?
  apply (match asms in I [intros]: ?P ⟹ ?Q (multi) ⇒ ‹prop_solver›)
  done
```

In the first *match*, without the (*multi*) argument, I is only ever be bound to one of the members of *asms*. This backtracks over both possibilities (see next section), however neither assumption in isolation is sufficient to solve to goal. The use of the *solves* combinator ensures that *prop_solver* has no effect on the goal when it doesn't solve it, and so the first match leaves the goal unchanged. In the second *match*, I is bound to all of *asms*, declaring both results as *intros*. With these rules *prop_solver* is capable of solving the goal.

Using for-fixed variables in patterns imposes additional constraints on the results. In all previous examples, the choice of using *?P* or a for-fixed P only depended on whether or not P was mentioned in another pattern or the inner method. When using a multi-match, however, all for-fixed terms must agree in the results.

```
lemma
  assumes asms: A ⟹ B  A ⟹ D  D ⟹ B
  shows (A ⟶ B) ∧ (A ⟶ D)
  apply (match asms in I [intros]: ?P ⟹ Q (multi) for Q ⇒
        ‹solves ‹prop_solver››)?
  apply (match asms in I [intros]: P ⟹ ?Q (multi) for P ⇒
        ‹prop_solver›)
  done
```

Here we have two seemingly-equivalent applications of *match*, however only the second one is capable of solving the goal. The first *match* selects the first and third members of *asms* (those that agree on their conclusion), which

is not sufficient. The second *match* selects the first and second members of *asms* (those that agree on their assumption), which is enough for *prop_solver* to solve the goal.

2.4 Dummy patterns

Dummy patterns may be given as placeholders for unique schematics in patterns. They implicitly receive all currently bound variables as arguments, and are coerced into the *prop* type whenever possible. For example, the trivial dummy pattern _ will match any proposition. In contrast, by default the pattern *?P* is considered to have type *bool*. It will not bind anything with meta-logical connectives (e.g. _ $\Longrightarrow$ _ or _ &&& _).

lemma
assumes *asms*: A &&& $B \Longrightarrow D$
shows $(A \wedge B \longrightarrow D)$
by (*match asms* **in** *I*: _ $\Rightarrow$ ‹*prop_solver intros*: *I conjunctionI*›)

2.5 Backtracking

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.

method *foo* =
(*match* **conclusion in** *?P* $\wedge$ *?Q* $\Rightarrow$ ‹*fail*› | *?R* $\Rightarrow$ ‹*prop_solver*›)

The failure of an inner method that is executed after a successful match will cause the entire match to fail. This 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. One can always rewrite a match using the combinators "*?*" and "|" to 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 foo_1 =
(*match* **conclusion in** *?P* $\wedge$ *?Q* $\Rightarrow$ ‹*fail*›) |
(*match* **conclusion in** *?R* $\Rightarrow$ ‹*prop_solver*›)

2.5.1 Cut

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 $\Rightarrow$
‹*rule mp* [*OF I' I* [*THEN conjunct*1]]›)

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*.

More precise control is also possible by giving a positive number n as an argument to *cut*. This will limit the number of backtracking results of that match to be at most n. The match argument (*cut* 1) is the same as simply (*cut*).

2.5.2 Multi-match revisited

A multi-match will produce a sequence of potential bindings for for-fixed variables, where each binding environment is the result of matching against at least one element from the match target. For each environment, the match result will be all elements of the match target which agree with the pattern under that environment. This can result in unexpected behaviour when giving very general patterns.

```
lemma
  assumes asms: ⋀x. A x ∧ B x  ⋀y. A y ∧ C y  ⋀z. B z ∧ C z
  shows A x ∧ C x
  by (match asms in I: ⋀x. P x ∧ ?Q x (multi) for P ⇒
    ‹match (P) in A ⇒ ‹fail›
                | _ ⇒ ‹match I in ⋀x. A x ∧ B x ⇒ ‹fail›
                                    | _ ⇒ ‹rule I›››)
```

Intuitively it seems like this proof should fail to check. The first match result, which binds I to the first two members of *asms*, fails the second inner match due to binding P to A. Backtracking then attempts to bind I to the third member of *asms*. This passes all inner matches, but fails when *rule* cannot successfully apply this to the current goal. After this, a valid match that is produced by the unifier is one which binds P to simply $\lambda a.\ A\ ?x$. The first inner match succeeds because $\lambda a.\ A\ ?x$ does not match A. The next inner match succeeds because I has only been bound to the first member of *asms*. This is due to *match* considering $\lambda a.\ A\ ?x$ and $\lambda a.\ A\ ?y$ as distinct terms.

The simplest way to address this is to explicitly disallow term bindings which we would consider invalid.

```
method abs_used for P =
  (match (P) in λa. ?P ⇒ ‹fail› | _ ⇒ ‹-›)
```

This method has no effect on the goal state, but instead serves as a filter on the environment produced from match.

2.6 Uncurrying

The *match* method is not aware of the logical content of match targets. Each pattern is simply matched against the shallow structure of a fact or term. Most facts are in *normal form*, which curries premises via meta-implication $\_ \Longrightarrow \_$.

```
lemma
```

assumes *asms*: $D \Longrightarrow B \Longrightarrow C$ $D \Longrightarrow A$
shows $D \Longrightarrow B \Longrightarrow C \wedge A$
by (*match asms* **in** H: $D \Longrightarrow \_$ (*multi*) $\Rightarrow$ ‹*prop_solver elims*: H›)

For the first member of *asms* the dummy pattern successfully matches against $B \Longrightarrow C$ and so the proof is successful.

lemma
assumes *asms*: $A \Longrightarrow B \Longrightarrow C$ $D \Longrightarrow C$
shows $D \vee (A \wedge B) \Longrightarrow C$
apply (*match asms* **in** H: $\_ \Longrightarrow C$ (*multi*) $\Rightarrow$ ‹*prop_solver elims*: H›)

This proof will fail to solve the goal. Our match pattern will only match rules which have a single premise, and conclusion C, so the first member of *asms* is not bound and thus the proof fails. Matching a pattern of the form $P \Longrightarrow Q$ against this fact will bind P to A and Q to $B \Longrightarrow C$. Our pattern, with a concrete C in the conclusion, will fail to match this fact.

To express our desired match, we may *uncurry* our rules before matching against them. This forms a meta-conjunction of all premises in a fact, so that only one implication remains. For example the uncurried version of $A \Longrightarrow B \Longrightarrow C$ is A &&& $B \Longrightarrow C$. This will now match our desired pattern $\_ \Longrightarrow C$, and can be *curried* after the match to put it back into normal form.

lemma
assumes *asms*: $A \Longrightarrow B \Longrightarrow C$ $D \Longrightarrow C$
shows $D \vee (A \wedge B) \Longrightarrow C$
by (*match asms* [*uncurry*] **in** H [*curry*]: $\_ \Longrightarrow C$ (*multi*) $\Rightarrow$
‹*prop_solver elims*: H›)

2.7 Reverse matching

The *match* method only attempts to perform matching of the pattern against the match target. Specifically this means that it will not instantiate schematic terms in the match target.

lemma
assumes *asms*: $\bigwedge x :: 'a.\ A\ x$
shows $A\ y$
apply (*match asms* **in** H: $A\ y \Rightarrow$ ‹*rule* H›)?
apply (*match asms* **in** H: P **for** $P \Rightarrow$
‹*match* $(A\ y)$ *in* $P \Rightarrow$ ‹*rule* H››)
done

In the first *match* we attempt to find a member of *asms* which matches our goal precisely. This fails due to no such member existing. The second match

reverses the role of the fact in the match, by first giving a general pattern P. This bound pattern is then matched against $A\ y$. In this case, P is bound to $A\ ?x$ and so it successfully matches.

2.8 Type matching

The rule instantiation attributes *where* and *of* attempt to guarantee type-correctness wherever possible. This can require additional invocations of *match* in order to statically ensure that instantiation will succeed.

lemma
assumes *asms*: $\bigwedge x :: 'a.\ A\ x$
shows $A\ y$
by (*match asms* **in** H: $\bigwedge z :: 'b.\ P\ z$ **for** $P \Rightarrow$
‹*match* (*y*) *in* $y :: 'b$ *for* $y \Rightarrow$ ‹*rule H* [*where* $z = y$]››)

In this example the type $'b$ is matched to $'a$, however statically they are formally distinct types. The first match binds $'b$ while the inner match serves to coerce y into having the type $'b$. This allows the rule instantiation to successfully apply.

Chapter 3

Method development

3.1 Tracing methods

Method tracing is supported by auxiliary print methods provided by *HOL−Eisbach.Eisbach_Tools*. These include *print_fact*, *print_term* and *print_type*. Whenever a print method is evaluated it leaves the goal unchanged and writes its argument as tracing output.

Print methods can be combined with the *fail* method to investigate the backtracking behaviour of a method.

lemma
assumes *asms*: *A B C D*
shows *D*
apply (*match asms* **in** *H*: _ ⇒ ‹*print_fact H*, *fail*›)

This proof will fail, but the tracing output will show the order that the assumptions are attempted.

3.2 Integrating with Isabelle/ML

Attributes

A custom rule attribute is a simple way to extend the functionality of Eisbach methods. The dummy rule attribute notation ([[_]]) invokes the given attribute against a dummy fact and evaluates to the result of that attribute. When used as a match target, this can serve as an effective auxiliary function.

```
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)))›
```

In this example, the new attribute *get_split_rule* lifts the ML function of the same name into an attribute. When applied to a case distinction over a

datatype, it retrieves its corresponding split rule.

We can then integrate this into a method that applies the split rule, first matching to ensure that fetching the rule was successful.

```
method splits =
  (match conclusion in ?P f for f ⇒
    ‹match [[get_split_rule f]] in U: (_ :: bool) = _ ⇒
      ‹rule U [THEN iffD2]››)

lemma L ≠ [] ⟹ case L of [] ⇒ False | _ ⇒ True
  apply splits
  apply (prop_solver intros: allI)
  done
```

Here the new *splits* method transforms the goal to use only logical connectives: $L = [] \longrightarrow \mathit{False} \wedge (\forall x\ y.\ L = x\ \#\ y \longrightarrow \mathit{True})$. This goal is then in a form solvable by *prop_solver* when given the universal quantifier introduction rule *allI*.

Bibliography

[1] M. Wenzel. *The Isabelle/Isar Reference Manual*. https://isabelle.in.tum.de/doc/isar-ref.pdf.

[2] M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, Technische Universität München, 2002. https://mediatum.ub.tum.de/doc/601724/601724.pdf.

Index