# Unification Utilities for Isabelle/ML

Kevin Kappelmann

October 4, 2023

**Abstract**

This article provides various unification utilities for Isabelle/ML, most prominently:

1. First-order and higher-order pattern E-unification and E-matching. While unifiers in Isabelle/ML only consider the $\alpha\beta\eta$-equational theory of the $\lambda$-calculus, unifiers in this article may take an extra background theory, in the form of an equational prover, into account. For example, the unification problem $n + 1 \equiv ?m + Suc\ 0$ may be solved by providing a prover for the background theory $\forall n.\ n + 1 \equiv n + Suc\ 0$.
2. Tactics, methods, and attributes with adjustable unifiers (e.g. resolution, fact, assumption, OF).
3. A generalisation of unification hints [1]. Unification hints are a flexible extension for unifiers. Among other things, they can be used for reflective tactics, to provide canonical unification instances, or to simply strengthen the background theory of a unifier in a controlled manner.
4. Simplifier integration for e-unifiers.
5. Practical combinations of unification algorithms, e.g. a combination of first-order and higher-order pattern unification.
6. A hierarchical logger for Isabelle/ML, including per logger configurations with log levels, output channels, message filters.

While this entry works with every object logic, some extra setup for Isabelle/HOL and application examples are provided. All unifiers are tested with SpecCheck [2].

## Contents

# 1 ML Code Utils

**theory** *ML-Code-Utils*
**imports** *Pure*
**begin**

**Summary** Utilities to generate and manipulate (parsed) ML code.

**ML-file**‹*ml-code-util.ML*›
**ML-file**‹*ml-syntax-util.ML*›

**end**

# 2 ML Attributes

**theory** *ML-Attributes*
**imports** *ML-Code-Utils*
**begin**

**Summary** ML code as attributes.

**ML-file**‹*ml-attribute.ML*›

**end**

# 3 ML Logger

**theory** *ML-Logger*
**imports**
*ML-Attributes*
**begin**

**Summary** Generic logging, at some places inspired by Apache's Log4J 2 https://logging.apache.org/log4j/2.x/manual/customloglevels.html.

**ML-file**‹*Data-Structures/map.ML*›
**ML-file**‹*Data-Structures/hoption-tree.ML*›
**ML-file**‹*Data-Structures/binding-tree.ML*›

**ML-file**‹*logger.ML*›
**ML-file**‹*logging-antiquotation.ML*›

**end**

## 3.1 Setup Result Commands

**theory** *Setup-Result-Commands*
  **imports** *Pure*
  **keywords** *setup-result* :: *thy-decl*
  **and** *local-setup-result* :: *thy-decl*
**begin**

**Summary** Setup and local setup with result commands

**ML**‹
```
  let
    fun setup-result finish (name, (source, pos)) =
      ML-Context.expression pos
        (ML-Lex.read val @ name @ ML-Lex.read = Context.>>> ( @ source @
ML-Lex.read ))
      |> finish
    val parse = Parse.embedded-ml
      -- ((keyword ‹=› || keyword ‹≡›)
      |-- Parse.position Parse.embedded-ml)
  in
    Outer-Syntax.command command-keyword ‹setup-result›
      ML setup with result for global theory
      (parse >> (Toplevel.theory o setup-result Context.theory-map));
    Outer-Syntax.local-theory command-keyword ‹local-setup-result›
      ML setup with result for local theory
      (parse >> (setup-result
        (Local-Theory.declaration {pos = here, syntax = false, pervasive = false}
o K)))
  end
```
›

**end**

## 3.2 Examples

**theory** *ML-Logger-Examples*
  **imports**

*ML-Logger*
*Setup-Result-Commands*
**begin**

First some simple, barebone logging: print some information.

**ML-command**‹
— the following two are equivalent
*val - = Logger.log Logger.root-logger Logger.INFO* @{*context*} (*K hello root logger*)
*val - =* @{*log Logger.INFO Logger.root-logger*} @{*context*} (*K hello root logger*)
›

**ML-command**‹
*val logger = Logger.root-logger*
*val - =* @{*log*} @{*context*} (*K hello root logger*)
— @{*log*} is equivalent to *Logger.log logger Logger.INFO*
›

To guarantee the existence of a "logger" in an ML structure, one should use the *HAS-LOGGER* signature.

**ML**‹
*structure My-Struct : sig*
*include HAS-LOGGER*
*val get-n : Proof.context -> int*
*end = struct*
*val logger = Logger.setup-new-logger Logger.root-logger My-Struct*
*fun get-n ctxt =* (@{*log*} *ctxt* (*K retrieving n...*); *42*)
*end*
›

**ML-command**‹*val n = My-Struct.get-n* @{*context*}›

We can set up a hierarchy of loggers

**ML**‹
*val logger = Logger.root-logger*
*val parent1 = Logger.setup-new-logger Logger.root-logger Parent1*
*val child1 = Logger.setup-new-logger parent1 Child1*
*val child2 = Logger.setup-new-logger parent1 Child2*

*val parent2 = Logger.setup-new-logger Logger.root-logger Parent2*
›

**ML-command**‹
(@{*log Logger.INFO Logger.root-logger*} @{*context*} (*K Hello root logger*);
@{*log Logger.INFO parent1*} @{*context*} (*K Hello parent1*);
@{*log Logger.INFO child1*} @{*context*} (*K Hello child1*);
@{*log Logger.INFO child2*} @{*context*} (*K Hello child2*);
@{*log Logger.INFO parent2*} @{*context*} (*K Hello parent2*))
›

We can use different log levels to show/surpress messages. The log levels are based on Apache's Log4J 2 https://logging.apache.org/log4j/2.x/manual/customloglevels.html.

**ML-command**‹@{*log Logger.DEBUG parent1*} @{*context*} (*K Hello parent1*)› — prints nothings
**declare** [[*ML-map-context* ‹*Logger.set-log-level parent1 Logger.DEBUG*›]]
**ML-command**‹@{*log Logger.DEBUG parent1*} @{*context*} (*K Hello parent1*)› — prints message
**ML-command**‹*Logger.ALL*› — ctrl+click on the value to see all log levels

We can set options for all loggers below a given logger. Below, we set the log level for all loggers below (and including) `parent1` to error, thus disabling warning messages.

**ML-command**‹
(@{*log Logger.WARN parent1*} @{*context*} (*K Warning from parent1*);
@{*log Logger.WARN child1*} @{*context*} (*K Warning from child1*))
›
**declare** [[*ML-map-context* ‹*Logger.set-log-levels parent1 Logger.ERR*›]]
**ML-command**‹
(@{*log Logger.WARN parent1*} @{*context*} (*K Warning from parent1*);
@{*log Logger.WARN child1*} @{*context*} (*K Warning from child1*))
›
**declare** [[*ML-map-context* ‹*Logger.set-log-levels parent1 Logger.INFO*›]]

We can set message filters.

**declare** [[*ML-map-context* ‹*Logger.set-msg-filters Logger.root-logger* (*match-string Third*)›]]
**ML-command**‹
(@{*log Logger.INFO parent1*} @{*context*} (*K First message*);
@{*log Logger.INFO child1*} @{*context*} (*K Second message*);
@{*log Logger.INFO child2*} @{*context*} (*K Third message*);
@{*log Logger.INFO parent2*} @{*context*} (*K Fourth message*))
›
**declare** [[*ML-map-context* ‹*Logger.set-msg-filters Logger.root-logger* (*K true*)›]]

One can also use different output channels (e.g. files) and hide/show some additional logging information. Ctrl+click on below values and explore.

**ML-command**‹*Logger.set-output*; *Logger.set-show-logger*; *Logging-Antiquotation.show-log-pos*›

To set up (local) loggers outside ML environments, *ML-Unification.Setup-Result-Commands* contains two commands, **setup-result** and **local-setup-result**.

**experiment**
**begin**
**local-setup-result** *local-logger* = ‹*Logger.new-logger Logger.root-logger Local*›

**ML-command**‹@{*log Logger.INFO local-logger*} @{*context*} (*K Hello local world*)›
**end**

*local-logger* is no longer available. The follow thus does not work:

Let us create another logger in the global context.

**setup-result** *some-logger* = ‹*Logger.new-logger Logger.root-logger Some-Logger*›
**ML-command**‹@{*log Logger.INFO some-logger*} @{*context*} (*K Hello world*)›

Let us delete it again.

**declare** [[*ML-map-context* ‹*Logger.delete-logger some-logger*›]]

The logger can no longer be found in the logger hierarchy

**ML-command**‹@{*log Logger.INFO some-logger*} @{*context*} (*K Hello world*)›

**end**

# 4 ML Attribute Utils

**theory** *ML-Attribute-Utils*
**imports**
*Pure*
**begin**

**Summary** Utilities for attributes.

**ML-file**‹*attribute-util.ML*›

**end**

# 5 ML Conversion Utils

**theory** *ML-Conversion-Utils*
**imports**
*Pure*
**begin**

**Summary** Utilities for conversions.

**lemma** *meta-eq-symmetric*: $(A \equiv B) \equiv (B \equiv A)$
**by** (*rule equal-intr-rule*) *simp-all*
**ML-file**‹*conversion-util.ML*›

**end**

# 6 ML Parsing Utils

**theory** *ML-Parsing-Utils*
**imports**
*ML-Attributes*
*ML-Attribute-Utils*
**begin**

**Summary** Parsing utilities for ML. We provide an antiquotation that takes a list of keys and creates a corresponding record with getters and mappers and a parser for corresponding key-value pairs.

**ML-file**‹*parse-util.ML*›

**ML-file**‹*parse-key-value.ML*›
**ML-file**‹*parse-key-value-antiquot.ML*›

**Example** **ML-command**‹
```
  — Create record type and utility functions.
  @{parse-entries (struct) Test [ABC, DEFG]}

  val parser =
    let
      — Create the key-value parser.
      val parse-entry = Parse-Key-Value.parse-entry
        Test.parse-key — parser for keys
        (Scan.succeed []) — delimiter parser
        (Test.parse-entry — value parser
          Parse.string — parser for ABC
          Parse.int) — parser for DEFG
      val required-keys = [Test.key Test.ABC] — required keys
      val default-entries = Test.empty-entries () — default values for entries
   in Test.parse-entries-required Parse.and-list1 required-keys parse-entry default-entries
end
    — This parses, for example, ABC = hello and DEFG = 3 or DEFG = 3 and
ABC = hello, but not DEFG = 3 since the key "ABC" is missing.
```
›

**end**

# 7 ML Functor Instances

**theory** *ML-Functor-Instances*
**imports**
*ML-Parsing-Utils*
**begin**

**Summary** Utilities for ML functors that create context data.

**ML-file**‹*functor-instance.ML*›
**ML-file**‹*functor-instance-antiquot.ML*›

**Example** **ML-command**‹
```
  — some arbitrary functor
  functor My-Functor(A : sig
    structure FIA : FUNCTOR-INSTANCE-ARGS
    val n : int
```

```
  end) =
  struct
    fun get-n () = (Pretty.writeln (Pretty.block
        [Pretty.str retrieving n from , Pretty.str A.FIA.full-name]);
      A.n)
  end

  — create an instance (structure) called Test-Functor-Instance
  @{functor-instance struct-name = Test-Functor-Instance
    and functor-name = My-Functor
    and id = ‹test›
    and more-args = ‹val n = 42›}

  val - = Test-Functor-Instance.get-n ()
›

end
```

# 8 General ML Utils

**theory** *ML-General-Utils*
  **imports** *Pure*
**begin**

**Summary** General ML utilities.

**ML-file**‹*general-util.ML*›

**end**

# 9 ML Generic Data Utils

**theory** *ML-Generic-Data-Utils*
  **imports** *Pure*
**begin**

**Summary** Utilities for `Generic_Data`.

**ML-file**‹*pair-generic-data-args.ML*›

**end**

# 10 ML Method Utils

**theory** *ML-Method-Utils*
  **imports** *Pure*
**begin**

**Summary** Utilities for methods.

**ML-file**‹*method-util.ML*›

**end**

**theory** *ML-Priorities*
**imports** *ML-Parsing-Utils*
**begin**

**Summary** Priorities for ML tactics.

**ML-file**‹*priority.ML*›

**end**

# 11 ML-Normalisations

**theory** *ML-Normalisations*
**imports**
*Pure*
**begin**

**Summary** Normalisation functions for terms, types, and theorems.

**ML-file**‹*term-normalisation.ML*›
**ML-file**‹*envir-normalisation.ML*›

**end**

# 12 ML-Binders

**theory** *ML-Binders*
**imports**
*ML-General-Utils*
*ML-Normalisations*
**begin**

**Summary** Binders for ML.

**ML-file**‹*binders.ML*›

**end**

# 13 ML Term Utils

**theory** *ML-Term-Utils*
**imports** *ML-Binders*
**begin**

**Summary** Utilities for terms.

**ML-file**‹*term-util.ML*›

**end**

## 14 ML Tactic Utils

**theory** *ML-Tactic-Utils*
**imports**
*ML-Logger*
*ML-Term-Utils*
*ML-Conversion-Utils*
**begin**

**Summary** Utilities for tactics.

**ML-file**‹*tactic-util.ML*›

**end**

## 15 ML Theorem Utils

**theory** *ML-Theorem-Utils*
**imports** *Pure*
**begin**

**Summary** Utilities for theorems.

**ML-file**‹*thm-util.ML*›

**end**

## 16 ML Utils

**theory** *ML-Utils*
**imports**
*ML-Attribute-Utils*
*ML-Conversion-Utils*
*ML-Functor-Instances*
*ML-General-Utils*
*ML-Generic-Data-Utils*
*ML-Method-Utils*
*ML-Attributes*
*ML-Code-Utils*
*ML-Parsing-Utils*
*ML-Priorities*
*ML-Tactic-Utils*
*ML-Term-Utils*

*ML-Theorem-Utils*
**begin**

**end**

# 17 ML Unification Basics

**theory** *ML-Unification-Base*
**imports**
*ML-Logger*
*ML-Binders*
*ML-Normalisations*
**begin**

**Summary** Basic definitions and utilities for unification algorithms.

**ML-file**‹*unification-base.ML*›
**ML-file**‹*unification-util.ML*›

**end**

# 18 Simps To

**theory** *Simps-To*
**imports**
*ML-Tactic-Utils*
*ML-Theorem-Utils*
*ML-Unification-Base*
*Setup-Result-Commands*
**begin**

**Summary** Simple frameworks to ask for the simp-normal form of a term on the user-level.

**setup-result** *simps-to-base-logger* = ‹*Logger.new-logger Logger.root-logger Simps-To-Base*›

**Using Simplification On Left Term** **definition** *SIMPS-TO s t* ≡ (*s* ≡ *t*)

**lemma** *SIMPS-TO-eq*: *SIMPS-TO s t* ≡ (*s* ≡ *t*)
**unfolding** *SIMPS-TO-def* **by** *simp*

Prevent simplification of second/right argument

**lemma** *SIMPS-TO-cong* [*cong*]: *s* ≡ *s'* ⟹ *SIMPS-TO s t* ≡ *SIMPS-TO s' t* **by** *simp*

**lemma** *SIMPS-TOI*: *PROP SIMPS-TO s s* **unfolding** *SIMPS-TO-eq* **by** *simp*
**lemma** *SIMPS-TOD*: *PROP SIMPS-TO s t* ⟹ *s* ≡ *t* **unfolding** *SIMPS-TO-eq* **by** *simp*

**ML-file**‹*simps-to.ML*›

**Using Simplification On Left Term Followed By Unification** **definition** *SIMPS-TO-UNIF s t* ≡ (*s* ≡ *t*)

Prevent simplification

**lemma** *SIMPS-TO-UNIF-cong* [*cong*]: *SIMPS-TO-UNIF s t* ≡ *SIMPS-TO-UNIF s t* **by** *simp*

**lemma** *SIMPS-TO-UNIF-eq*: *SIMPS-TO-UNIF s t* ≡ (*s* ≡ *t*) **unfolding** *SIMPS-TO-UNIF-def* **by** *simp*

**lemma** *SIMPS-TO-UNIFI*: *PROP SIMPS-TO s s'* ⟹ *s'* ≡ *t* ⟹ *PROP SIMPS-TO-UNIF s t*
**unfolding** *SIMPS-TO-UNIF-eq SIMPS-TO-eq* **by** *simp*
**lemma** *SIMPS-TO-UNIFD*: *PROP SIMPS-TO-UNIF s t* ⟹ *s* ≡ *t*
**unfolding** *SIMPS-TO-UNIF-eq* **by** *simp*

**ML-file**‹*simps-to-unif.ML*›

**Examples** **experiment**
**begin**
**lemma**
**assumes** [*simp*]: *P* ≡ *Q*
**and** [*simp*]: *Q* ≡ *R*
**shows** *PROP SIMPS-TO P Q*
**apply** *simp* — Note: only the left-hand side is simplified.
**ML-command**‹ — obtaining the normal form theorem for a term in ML
*Simps-To.SIMPS-TO-thm-resultsq* (*simp-tac* @{*context*}) @{*context*} @{*cterm P*}
|> *Seq.list-of* |> *map* @{*print*}
›
**oops**

**schematic-goal**
**assumes** [*simp*]: *P* ≡ *Q*
**and** [*simp*]: *Q* ≡ *R*
**shows** *PROP SIMPS-TO P ?Q*
**apply** *simp*
**by** (*rule SIMPS-TOI*)
**end**

**end**

# 19 ML Unifiers

**theory** *ML-Unifiers*

**imports**
*ML-Unification-Base*
*ML-Functor-Instances*
*ML-Priorities*
*Simps-To*
**begin**

**Summary** Unification modulo equations and combinators for unifiers.

### Combinators **ML-file**‹*unification-combinator.ML*›

**ML-file**‹*unification-combine.ML*›
**ML**‹
@{*functor-instance struct-name = Standard-Unification-Combine*
*and functor-name = Unification-Combine*
*and id =* ‹›}
›
**local-setup** ‹*Standard-Unification-Combine.setup-attribute NONE*›

### Standard Unifiers **ML-file**‹*unify-copy.ML*›

**ML-file**‹*higher-order-unification.ML*›
**ML-file**‹*higher-order-pattern-unification.ML*›
**ML-file**‹*first-order-unification.ML*›

### Unification via Tactics **ML-file**‹*tactic-unification.ML*›

### Unification via Simplification **ML-file**‹*simplifier-unification.ML*›

### Mixture of Unifiers **ML-file**‹*higher-ordern-pattern-first-decomp-unification.ML*›

**ML-file**‹*mixed-unification.ML*›
**ML**‹
@{*functor-instance struct-name = Standard-Mixed-Unification*
*and functor-name = Mixed-Unification*
*and id =* ‹›
*and more-args =* ‹*structure UC = Standard-Unification-Combine*›}
›

**declare** [[*ucombine add =* ‹*Standard-Unification-Combine.eunif-data*
(*Simplifier-Unification.simp-unify*
|> *Unification-Combinator.norm-closed-unifier*
(#*norm-term Standard-Mixed-Unification.norms-first-higherp-first-comb-higher-unify*)
|> *Unification-Combinator.unifier-from-closed-unifier*
|> *K*)
(*Standard-Unification-Combine.default-metadata* ***binding*** ‹*simp-unif*›)›]]

**end**

# 20 Unification Parsers

**theory** *ML-Unification-Parsers*
**imports**
*ML-Parsing-Utils*
**begin**

**Summary** Common parsers needed for unification attributes, tactics, methods.

**ML-file**‹*unification-parser.ML*›

**end**

## 20.1 Assumption Tactic

**theory** *Unify-Assumption-Tactic*
**imports**
*ML-Functor-Instances*
*ML-Unifiers*
*ML-Unification-Parsers*
**begin**

**Summary** Assumption tactic and method with adjustable unifier.

**ML-file**‹*unify-assumption-base.ML*›
**ML-file**‹*unify-assumption.ML*›

**ML**‹
@{*functor-instance struct-name* = *Standard-Unify-Assumption*
*and functor-name* = *Unify-Assumption*
*and id* = ‹›
*and more-args* = ‹*val init-args* = {
*normalisers* = *SOME Standard-Mixed-Unification.norms-first-higherp-first-comb-higher-unify*,
*unifier* = *SOME Standard-Mixed-Unification.first-higherp-first-comb-higher-unify*
}›}
›
**local-setup** ‹*Standard-Unify-Assumption.setup-attribute NONE*›
**local-setup** ‹*Standard-Unify-Assumption.setup-method NONE*›

**Examples** **experiment**
**begin**

**lemma** *PROP P* $\Longrightarrow$ *PROP P*
**by** *uassm*

**lemma**
**assumes** *h*: $\bigwedge$*P*. *PROP P*
**shows** *PROP P x*
**using** *h* **by** *uassm*

**schematic-goal** $\bigwedge x.\ PROP\ P\ (c :: 'a) \Longrightarrow PROP\ ?Y\ (x :: 'a)$
**by** *uassm*

**schematic-goal** *a*: $PROP\ ?P\ (y :: 'a) \Longrightarrow PROP\ ?P\ (?x :: 'a)$
**by** *uassm* — compare the result with following call

**schematic-goal**
$PROP\ ?P\ (x :: 'a) \Longrightarrow PROP\ P\ (?x :: 'a)$
**by** *uassm* — compare the result with following call

**schematic-goal**
$\bigwedge x.\ PROP\ D \Longrightarrow (\bigwedge P\ y.\ PROP\ P\ y\ x) \Longrightarrow PROP\ C \Longrightarrow PROP\ P\ x$
**by** (*uassm unifier* = *Higher-Order-Unification.unify*) — the line below is equivalent

Unlike *assumption*, *uassm* will not close the goal if the order of premises of the assumption and the goal are different. Compare the following two examples:

**lemma** $\bigwedge x.\ PROP\ D \Longrightarrow (\bigwedge y.\ PROP\ A\ y \Longrightarrow PROP\ B\ x) \Longrightarrow PROP\ C \Longrightarrow PROP\ A\ x \Longrightarrow PROP\ B\ x$
**by** *uassm*

**lemma** $\bigwedge x.\ PROP\ D \Longrightarrow (\bigwedge y.\ PROP\ A\ y \Longrightarrow PROP\ B\ x) \Longrightarrow PROP\ A\ x \Longrightarrow PROP\ C \Longrightarrow PROP\ B\ x$
**by** *assumption*

**end**

**end**

## 20.2 Resolution Tactics

**theory** *Unify-Resolve-Tactics*
**imports**
*Unify-Assumption-Tactic*
*ML-Method-Utils*
**begin**

**Summary** Resolution tactics and methods with adjustable unifier.

**ML-file**‹*unify-resolve-base.ML*›
**ML-file**‹*unify-resolve.ML*›
**ML**‹
@{*functor-instance struct-name* = *Standard-Unify-Resolve*
*and functor-name* = *Unify-Resolve*
*and id* = ‹›

*and more-args* = ‹*val init-args* = {
*normalisers* = *SOME Standard-Mixed-Unification.norms-first-higherp-first-comb-higher-unify*,
*unifier* = *SOME Standard-Mixed-Unification.first-higherp-first-comb-higher-unify*,
*mode* = *SOME* (*Unify-Resolve-Args.PM.key Unify-Resolve-Args.PM.any*),
*chained* = *SOME* (*Unify-Resolve-Args.PCM.key Unify-Resolve-Args.PCM.resolve*)
}›}
›
**local-setup** ‹*Standard-Unify-Resolve.setup-attribute NONE*›
**local-setup** ‹*Standard-Unify-Resolve.setup-method NONE*›

**Examples** **experiment**
**begin**

**lemma**
**assumes** *h*: $\bigwedge x.$ *PROP D x* $\Longrightarrow$ *PROP C x*
**shows** $\bigwedge x.$ *PROP A x* $\Longrightarrow$ *PROP B x* $\Longrightarrow$ *PROP C x*
**apply** (*urule h*) — the line below is equivalent

**oops**

**lemma**
**assumes** *h*: *PROP C x*
**shows** *PROP C x*
**by** (*urule h* **where** *unifier* = *First-Order-Unification.unify*) — the line below is equivalent

**lemma**
**assumes** *h*: $\bigwedge x.$ *PROP A x* $\Longrightarrow$ *PROP D x*
**shows** $\bigwedge x.$ *PROP A x* $\Longrightarrow$ *PROP B x* $\Longrightarrow$ *PROP C x*
— use (r,e,d,f) to specify the resolution mode (resolution, elim, dest, forward)
**apply** (*urule* (*d*) *h*) — the line below is equivalent

**oops**

You can specify how chained facts should be used. By default, *urule* works like *rule*: it uses chained facts to resolve against the premises of the passed rules.

**lemma**
**assumes** *h1*: $\bigwedge x.$ (*PROP F x* $\Longrightarrow$ *PROP E x*) $\Longrightarrow$ *PROP C x*
**and** *h2*: $\bigwedge x.$ *PROP F x* $\Longrightarrow$ *PROP E x*
**shows** $\bigwedge x.$ *PROP A x* $\Longrightarrow$ *PROP B x* $\Longrightarrow$ *PROP C x*
— Compare all of the following calls:

**using** *h2* **apply** (*urule h1* **where** *chained* = *fact*)

**done**

You can specify whether any or every rule must resolve against the goal:

**lemma**
**assumes** *h1*: $\bigwedge x\ y.\ PROP\ C\ y \Longrightarrow PROP\ D\ x \Longrightarrow PROP\ C\ x$
**and** *h2*: $\bigwedge x\ y.\ PROP\ C\ x \Longrightarrow PROP\ D\ x$
**and** *h3*: $\bigwedge x\ y.\ PROP\ C\ x$
**shows** $\bigwedge x.\ PROP\ A\ x \Longrightarrow PROP\ B\ x \Longrightarrow PROP\ C\ x$
**using** *h3* **apply** (*urule h1 h2* **where** *mode* = *every*)

**done**

**lemma**
**assumes** *h1*: $\bigwedge x\ y.\ PROP\ C\ y \Longrightarrow PROP\ A\ x \Longrightarrow PROP\ C\ x$
**and** *h2*: $\bigwedge x\ y.\ PROP\ C\ x \Longrightarrow PROP\ B\ x \Longrightarrow PROP\ D\ x$
**and** *h3*: $\bigwedge x\ y.\ PROP\ C\ x$
**shows** $\bigwedge x.\ PROP\ A\ x \Longrightarrow PROP\ B\ x \Longrightarrow PROP\ C\ x$
**using** *h3* **apply** (*urule* (*d*) *h1 h2* **where** *mode* = *every*)
**oops**

**end**

**end**

## 20.3 Fact Tactic

**theory** *Unify-Fact-Tactic*
**imports**
*Unify-Resolve-Tactics*
**begin**

**Summary** Fact tactic with adjustable unifier.

**ML-file**‹*unify-fact-base.ML*›
**ML-file**‹*unify-fact.ML*›

**ML**‹
@{*functor-instance struct-name* = *Standard-Unify-Fact*
*and functor-name* = *Unify-Fact*
*and id* = ‹›
*and more-args* = ‹*val init-args* = {
*normalisers* = *SOME Standard-Mixed-Unification.norms-first-higherp-first-comb-higher-unify*,
*unifier* = *SOME Standard-Mixed-Unification.first-higherp-first-comb-higher-unify*
}›}
›
**local-setup** ‹*Standard-Unify-Fact.setup-attribute NONE*›
**local-setup** ‹*Standard-Unify-Fact.setup-method NONE*›

**Examples** **experiment**
**begin**

**lemma**
**assumes** *h*: $\bigwedge x\ y.\ PROP\ P\ x\ y$
**shows** $PROP\ P\ x\ y$
**by** (*ufact h*)

**lemma**
**assumes** $\bigwedge P\ y.\ PROP\ P\ y\ x$
**shows** $PROP\ P\ x$
**by** (*ufact assms* **where** *unifier* = *Higher-Order-Unification.unify*) — the line below is equivalent

**lemma**
**assumes** $\bigwedge x\ y.\ PROP\ A\ x \Longrightarrow PROP\ B\ x \Longrightarrow PROP\ P\ x$
**shows** $\bigwedge x\ y.\ PROP\ A\ x \Longrightarrow PROP\ B\ x \Longrightarrow PROP\ P\ x$
**using** *assms* **by** *ufact*
**end**

**end**

# 21 Unification Tactics

**theory** *Unification-Tactics*
**imports**
*Unify-Assumption-Tactic*
*Unify-Resolve-Tactics*
*Unify-Fact-Tactic*
**begin**

**Summary** Tactics with adjustable unifiers.

**end**

# 22 Unification Attributes

**theory** *Unification-Attributes*
**imports** *Unify-Resolve-Tactics*
**begin**

**Summary** OF attribute with adjustable unifier.

**ML-file**‹*unify-of-base.ML*›
**ML-file**‹*unify-of.ML*›

**ML**‹
@{*functor-instance struct-name* = *Standard-Unify-OF*
*and functor-name* = *Unify-OF*
*and id* = ‹›
*and more-args* = ‹*val init-args* = {

*normalisers = SOME Standard-Mixed-Unification.norms-first-higherp-first-comb-higher-unify*,
*unifier = SOME Standard-Mixed-Unification.first-higherp-first-comb-higher-unify*,
*mode = SOME (Unify-OF-Args.PM.key Unify-OF-Args.PM.fact)*
}›}
›
**local-setup** ‹*Standard-Unify-OF.setup-attribute NONE*›

**Examples** **experiment**
**begin**
**lemma**
**assumes** *h1*: (*PROP A* ⟹ *PROP D*) ⟹ *PROP E* ⟹ *PROP C*
**assumes** *h2*: *PROP B* ⟹ *PROP D*
**and** *h3*: *PROP F* ⟹ *PROP E*
**shows** (*PROP A* ⟹ *PROP B*) ⟹ *PROP F* ⟹ *PROP C*
**by** (*fact h1*[*uOF h2 h3* **where** *mode* = *resolve*]) — the line below is equivalent

**lemma**
**assumes** *h1*: (*PROP A* ⟹ *PROP A*)
**assumes** *h2*: (*PROP A* ⟹ *PROP A*) ⟹ *PROP B*
**shows** *PROP B*
**by** (*fact h2*[*uOF h1*]) — the line below is equivalent

— Note: *OF* would not work in this case:

**lemma**
**assumes** *h1*: ⋀*x y z*. *PROP P x y* ⟹ *PROP P y y* ⟹ (*PROP A* ⟹ *PROP A*) ⟹
(*PROP A* ⟹ *PROP B*) ⟹ *PROP C*
**and** *h2*: ⋀*x y*. *PROP P x y*
**and** *h3* : *PROP A* ⟹ *PROP A*
**and** *h4* : *PROP D* ⟹ *PROP B*
**shows** (*PROP A* ⟹ *PROP D*) ⟹ *PROP C*
**by** (*fact h1*[*uOF h2 h2 h3*, *uOF h4* **where** *mode* = *resolve*])

**lemma**
**assumes** *h1*: ⋀*P x*. *PROP P x* ⟹ *PROP E P x*
**and** *h2*: *PROP P x*
**shows** *PROP E P x*
**by** (*fact h1*[*uOF h2*]) — the following line does not work (multiple unifiers error)

We can also specify the unifier to be used:

**lemma**
**assumes** *h1*: ⋀*P*. *PROP P* ⟹ *PROP E*
**and** *h2*: ⋀*P*. *PROP P*
**shows** *PROP E*
**by** (*fact h1*[*uOF h2* **where** *unifier* = *First-Order-Unification.unify*]) — the line below is equivalent

**end**

**end**

# 23 Term Indexing

**theory** *ML-Term-Index*
**imports**
*ML-Normalisations*
**begin**

**Summary** Termin indexes signatures and implementations.

**ML-file**‹*term-index.ML*›
**ML-file**‹*discrimination-tree.ML*›

**ML-file**‹*term-index-data.ML*›

**end**

# 24 Unification Hints

**theory** *ML-Unification-Hints*
**imports**
*ML-Generic-Data-Utils*
*ML-Term-Index*
*ML-Unifiers*
*ML-Unification-Parsers*
**begin**

**Summary** A generalisation of unification hints, originally introduced in [1]. We support a generalisation that

1. allows additional universal variables in premises
2. allows non-atomic left-hand sides for premises
3. allows arbitrary functions to perform the matching/unification of a hint with a disagreement pair.

   General shape of a hint: $\bigwedge y1 ... yn.\ (\bigwedge x1 ... xn1.\ lhs1 \equiv rhs1) \Longrightarrow ... \Longrightarrow (\bigwedge x1 ... xnk.\ lhsk \equiv rhsk) \Longrightarrow lhs \equiv rhs$

**ML-file**‹*unification-hints-base.ML*›
**ML-file**‹*unification-hints.ML*›
**ML-file**‹*term-index-unification-hints.ML*›

**ML**‹
@{*functor-instance struct-name = Standard-Unification-Hints*
*and functor-name = Term-Index-Unification-Hints*
*and id = ‹›*
*and more-args = ‹*
*structure TI = Discrimination-Tree*
*val init-args = {*
*concl-unifier = SOME Higher-Ordern-Pattern-First-Decomp-Unification.unify*,
*normalisers = SOME Standard-Mixed-Unification.norms-first-higherp-first-comb-higher-unify*,
*prems-unifier = SOME (Standard-Mixed-Unification.first-higherp-first-comb-higher-unify*
*|> Unification-Combinator.norm-unifier Envir-Normalisation.beta-norm-term-unif)*,
*retrieval = SOME (Term-Index-Unification-Hints-Args.mk-sym-retrieval*
*TI.norm-term TI.unifiables)*,
*hint-preprocessor = SOME (K I)*
*}›}*
›
**local-setup** ‹*Standard-Unification-Hints.setup-attribute NONE*›

Standard unification hints are accessible via *uhint*.

**declare** [[*ucombine add = ‹Standard-Unification-Combine.eunif-data*
*(Standard-Unification-Hints.try-hints*
*|> Unification-Combinator.norm-unifier*
*(#norm-term Standard-Mixed-Unification.norms-first-higherp-first-comb-higher-unify)*
*|> K)*
*(Standard-Unification-Combine.default-metadata Standard-Unification-Hints.binding)›*]]

Examples see `../Examples`.

**end**

# 25 Setup for HOL

**theory** *ML-Unification-HOL-Setup*
**imports**
*HOL.HOL*
*ML-Unification-Hints*
**begin**

**lemma** *eq-eq-True*: *P ≡ (P ≡ Trueprop True)* **by** *standard+ simp-all*
**declare** [[*uhint* **where** *hint-preprocessor = ‹Unification-Hints-Base.obj-logic-hint-preprocessor*
@{*thm atomize-eq*[*symmetric*]} (*Conv.rewr-conv* @{*thm eq-eq-True*})›]]

**lemma** *eq-TrueI*: *PROP P ⟹ PROP P ≡ Trueprop True* **by** (*standard*) *simp*
**declare** [[*ucombine add = ‹Standard-Unification-Combine.eunif-data*
*(Simplifier-Unification.SIMPS-TO-unify* @{*thm eq-TrueI*}
*|> Unification-Combinator.norm-closed-unifier*
*(#norm-term Standard-Mixed-Unification.norms-first-higherp-first-comb-higher-unify)*
*|> Unification-Combinator.unifier-from-closed-unifier*
*|> K)*
*(Standard-Unification-Combine.default-metadata* ***binding*** ‹*SIMPS-TO-unif*›)›]]

**declare** [[*ucombine add* = ‹*Standard-Unification-Combine.eunif-data*
(*Simplifier-Unification.SIMPS-TO-UNIF-unify* @{*thm eq-TrueI*}
*Standard-Mixed-Unification.norms-first-higherp-first-comb-higher-unify*
(*Standard-Mixed-Unification.first-higherp-first-comb-higher-unify*
|> *Unification-Combinator.norm-unifier Envir-Normalisation.beta-norm-term-unif*)
|> *Unification-Combinator.norm-unifier*
(#*norm-term Standard-Mixed-Unification.norms-first-higherp-first-comb-higher-unify*)
|> *K*)
(*Standard-Unification-Combine.default-metadata* ***binding*** ‹*SIMPS-TO-UNIF-unif*›)›]]

**end**

# 26 E-Unification Examples

**theory** *E-Unification-Examples*
**imports**
*Main*
*ML-Unification-HOL-Setup*
*Unify-Fact-Tactic*
**begin**

**Summary** Sample applications of e-unifiers, methods, etc. introduced in this session.

**experiment**
**begin**

## 26.1 Using The Simplifier For Unification.

**inductive-set** *even* :: *nat set* **where**
*zero*: *0* ∈ *even* |
*step*: *n* ∈ *even* ⟹ *Suc* (*Suc n*) ∈ *even*

Premises of the form *SIMPS-TO-UNIF lhs rhs* are solved by `Simplifier_Unification`. It first normalises *lhs* and then unifies the normalisation with *rhs*. See also *ML-Unification.ML-Unification-HOL-Setup*.

**lemma** [*uhint* **where** *prio* = *Prio.LOW*]: *n* ≠ *0* ⟹ *PROP SIMPS-TO-UNIF* (*n* − *1*) *m* ⟹ *n* ≡ *Suc m*
**unfolding** *SIMPS-TO-UNIF-eq* **by** *linarith*

By default, below unification methods use `Standard_Mixed_Unification.first_higherp_first` which is a combination of various practical unification algorithms.

**schematic-goal** (⋀*x*. *x* + *4* = *n*) ⟹ *Suc ?x* = *n*
**by** *uassm*

**lemma** *6* ∈ *even*
**apply** (*urule step*)
**apply** (*urule step*)

**apply** (*urule step*)
**apply** (*urule zero*)
**done**

**lemma** (*220* + (*80* − *2* ∗ *2*)) ∈ *even*
**apply** (*urule step*)
**apply** (*urule step*)+
**apply** (*urule zero*)
**done**

**lemma**
**assumes** [*a*,*b*,*c*] = [*c*,*b*,*a*]
**shows** [*a*] @ [*b*,*c*] = [*c*,*b*,*a*]
**using** *assms* **by** *uassm*

**lemma** *x* ∈ ({*z*, *y*, *x*} ∪ *S*) ∩ {*x*}
**by** (*ufact TrueI*)

**lemma** (*x* + (*y* :: *nat*))^2 ≤ *x*^2 + *2*∗*x*∗*y* + *y*^2 + *4* ∗ *y* + *x* − *y*
**supply** *power2-sum*[*simp*]
**by** (*ufact TrueI*)

**lemma**
**assumes** ⋀*s*. *P* (*Suc* (*Suc 0*)) (*s*(*x* := (*1* :: *nat*), *x* := *1* + *1* ∗ *4* − *3*))
**shows** *P 2* (*s*(*x* := *2*))
**by** (*ufact assms*[*of s*])

## 26.2 Providing Canonical Solutions With Unification Hints

**lemma** [*uhint*]: *xs* ≡ [] ⟹ *length xs* ≡ *0* **by** *simp*

**schematic-goal** *length ?xs* = *0*
**by** (*ufact refl*)

**lemma** [*uhint*]: (*n* :: *nat*) ≡ *m* ⟹ *n* − *m* ≡ *0* **by** *simp*

**schematic-goal** *n* − *?m* = (*0* :: *nat*)
**by** (*ufact refl*)

The following fails because, by default, `Standard_Unification_Hints.try_hints` uses the higher-order pattern unifier to unify hints against a given disagreement pair, and *0*::′*a* cannot be higher-order pattern unified with *length* []. The unification of the hint requires the use of yet another hint, namely *length xs* = *0* (cf. above).

**schematic-goal** *n* − *?m* = *length* []
— by (ufact refl)
**oops**

There are two ways to fix this:

1. We allow the recursive uses of unification hints when searching for suitable unification hints.
2. We use a different unification hint that the recursive use of hints explicit.

Solution 1: recursive usages of hints. Warning: such recursive applications easily loop.

**schematic-goal** $n - ?m = length\ []$
**using** [[*uhint* **where** *concl-unifier* = *Standard-Mixed-Unification.first-higherp-first-comb-higher-unify*]]
**by** (*ufact refl*)

Solution 2: make the recursion explicit in the hint.

**lemma** [*uhint*]: $k \equiv 0 \Longrightarrow (n :: nat) \equiv m \Longrightarrow n - m \equiv k$ **by** *simp*

**schematic-goal** $n - ?m = length\ []$
**by** (*ufact refl*)

## 26.3 Strenghten Unification With Unification Hints

**lemma**
**assumes** [*uhint*]: $n = m$
**shows** $n - m = (0 :: nat)$
**by** (*ufact refl*)

**lemma**
**assumes** $x = y$
**shows** $y = x$
**supply** *eq-commute*[*uhint*]
**by** (*ufact assms*)

**Unfolding definitions.** **definition** $mysuc\ n = Suc\ n$

**lemma**
**assumes** $\bigwedge m.\ Suc\ n > mysuc\ m$
**shows** $mysuc\ n > Suc\ 3$
**supply** *mysuc-def*[*uhint*]
**by** (*ufact assms*)

**Discharging meta impliciations with object-level implications** **lemma** [*uhint*]:
$Trueprop\ A \equiv A' \Longrightarrow Trueprop\ B \equiv B' \Longrightarrow Trueprop\ (A \longrightarrow B) \equiv (PROP\ A' \Longrightarrow PROP\ B')$
**using** *atomize-imp*[*symmetric*] **by** *simp*

**lemma**
**assumes** $A \longrightarrow (B \longrightarrow C) \longrightarrow D$
**shows** $A \Longrightarrow (B \Longrightarrow C) \Longrightarrow D$
**using** *assms* **by** *ufact*

### 26.4 Better Control Over Meta Variable Instantiations

Consider the following type-inference problem.

**schematic-goal**
**assumes** *app-typeI*: $\bigwedge f\ x.\ (\bigwedge x.\ ArgT\ x \Longrightarrow DomT\ x\ (f\ x)) \Longrightarrow ArgT\ x \Longrightarrow DomT\ x\ (f\ x)$
**and** *f-type*: $\bigwedge x.\ ArgT\ x \Longrightarrow DomT\ x\ (f\ x)$
**and** *x-type*: $ArgT\ x$
**shows** $?T\ (f\ x)$
**apply** (*urule app-typeI*) — compare with the following application, creating an (unintuitive) higher-order instantiation

**oops**

**end**

**end**

## 27 Examples: Reification Via Unification Hints

**theory** *Unification-Hints-Reification-Examples*
**imports**
*HOL.Rat*
*ML-Unification-HOL-Setup*
*Unify-Fact-Tactic*
**begin**

**Summary** Reification via unification hints. For an introduction to unification hints refer to [1]. We support a generalisation of unification hints as described in *ML-Unification.ML-Unification-Hints*.

### 27.1 Setup

One-time setup to obtain a unifier with unification hints for the purpose of reification. We could also simply use the standard unification hints *uhint*, but having separate instances is a cleaner approach.

**ML**‹
@{*functor-instance struct-name = Reification-Unification-Hints*
*and functor-name = Term-Index-Unification-Hints*
*and id =* ‹*reify*›
*and more-args =* ‹
*structure TI = Discrimination-Tree*
*val init-args =* {
*normalisers = SOME Higher-Order-Pattern-Unification.norms-unify*,
*retrieval = SOME* (*Term-Index-Unification-Hints-Args.mk-sym-retrieval*

```
      TI.norm-term TI.unifiables),
     prems-unifier = NONE,
     concl-unifier = NONE,
    hint-preprocessor = SOME (Standard-Unification-Hints.get-hint-preprocessor
      (Context.the-generic-context ()))
   }›}
 val reify-unify = Unification-Combinator.add-fallback-unifier
  (fn unif-theory =>
  Higher-Order-Pattern-Unification.e-unify Unification-Util.unify-types unif-theory
unif-theory)
  (Reification-Unification-Hints.try-hints
  |> Unification-Combinator.norm-unifier (#norm-term Higher-Order-Pattern-Unification.norms-unify))
›
local-setup ‹Reification-Unification-Hints.setup-attribute NONE›
```

Premises of hints should again be unified by the reification unifier.

```
declare [[reify-uhint where prems-unifier = reify-unify]]
```

## 27.2 Formulas with Quantifiers and Environment

The following example is taken from HOL-Library.Reflection_Examples. It is recommended to compare the approach presented here with the reflection tactic presented in said theory.

```
datatype form =
  TrueF
| FalseF
| Less nat nat
| And form form
| Or form form
| Neg form
| ExQ form

primrec interp :: form ⇒ ('a::ord) list ⇒ bool
where
  interp TrueF vs ⟷ True
| interp FalseF vs ⟷ False
| interp (Less i j) vs ⟷ vs ! i < vs ! j
| interp (And f1 f2) vs ⟷ interp f1 vs ∧ interp f2 vs
| interp (Or f1 f2) vs ⟷ interp f1 vs ∨ interp f2 vs
| interp (Neg f) vs ⟷ ¬ interp f vs
| interp (ExQ f) vs ⟷ (∃ v. interp f (v # vs))
```

**Reification with unification and recursive hint unification for conclusion** The following illustrates how to use the equations *interp TrueF ?vs = True*

```
    interp FalseF ?vs = False
    interp (Less ?i ?j) ?vs = (?vs ! ?i < ?vs ! ?j)
    interp (And ?f1.0 ?f2.0) ?vs = (interp ?f1.0 ?vs ∧ interp ?f2.0 ?vs)
```

*interp* (*Or ?f1.0 ?f2.0*) *?vs* = (*interp ?f1.0 ?vs* ∨ *interp ?f2.0 ?vs*)
*interp* (*Neg ?f*) *?vs* = (¬ *interp ?f ?vs*)
*interp* (*ExQ ?f*) *?vs* = (∃ *v*. *interp ?f* (*v* # *?vs*)) directly as unification hints for reification.

**experiment**
**begin**

Hints for list lookup.

**declare** *List.nth-Cons-Suc*[*reify-uhint* **where** *prio* = *Prio.LOW*]
**and** *List.nth-Cons-0*[*reify-uhint*]

Hints to reify formulas of type *bool* into formulas of type *form*.

**declare** *interp.simps*[*reify-uhint*]

We have to allow the hint unifier to recursively look for hints during unification of the hint's conclusion.

**declare** [[*reify-uhint* **where** *concl-unifier* = *reify-unify*]]

**schematic-goal**
*interp ?f* (*?vs* :: (*'a* :: *ord*) *list*) = (∃(*x* :: *'a*). *x* < *y* ∧ ¬(∃(*z* :: *'a*). *v* < *z* ∨ ¬*False*))
**by** (*ufact refl* **where** *unifier* = *reify-unify*)

While this all works nicely if set up correctly, it can be rather difficult to understand and debug the recursive unification process for a hint's conclusion. In the next paragraph, we present an alternative that is closer to the examples presented in the original unification hints paper [1].

**end**

**Reification with matching without recursion for conclusion** We disallow the hint unifier to recursively look for hints while unifying the conclusion; instead, we only allow the hint unifier to match the hint's conclusion against the disagreement terms.

**declare** [[*reify-uhint* **where** *concl-unifier* = *Higher-Order-Pattern-Unification.match*]]

However, this also means that we now have to write our hints such that the hint's conclusion can successfully be matched against the disagreement terms. In particular, the disagreement terms may still contain meta variables that we want to instantiate with the help of the unification hints. Essentially, a hint then describes a canonical instantiation for these meta variables.

**experiment**
**begin**

**lemma** [*reify-uhint* **where** *prio* = *Prio.LOW*]:
*n* ≡ *Suc n′* ⟹ *vs* ≡ *v* # *vs′* ⟹ *vs′* ! *n′* ≡ *x* ⟹ *vs* ! *n* ≡ *x*
**by** *simp*

**lemma** [*reify-uhint*]: *n* ≡ *0* ⟹ *vs* ≡ *x* # *vs′* ⟹ *vs* ! *n* ≡ *x*
**by** *simp*

**lemma** [*reify-uhint*]:
⟦*e* ≡ *ExQ f*; ⋀*v*. *interp f* (*v* # *vs*) ≡ *P v*⟧ ⟹ *interp e vs* ≡ ∃ *v*. *P v*
⟦*e* ≡ *Less i j*; *x* ≡ *vs* ! *i*; *y* ≡ *vs* ! *j*⟧ ⟹ *interp e vs* ≡ *x* < *y*
⟦*e* ≡ *And f1 f2*; *interp f1 vs* ≡ *r1*; *interp f2 vs* ≡ *r2*⟧ ⟹ *interp e vs* ≡ *r1* ∧ *r2*
⟦*e* ≡ *Or f1 f2*; *interp f1 vs* ≡ *r1*; *interp f2 vs* ≡ *r2*⟧ ⟹ *interp e vs* ≡ *r1* ∨ *r2*
*e* ≡ *Neg f* ⟹ *interp f vs* ≡ *r* ⟹ *interp e vs* ≡ ¬*r*
*e* ≡ *TrueF* ⟹ *interp e vs* ≡ *True*
*e* ≡ *FalseF* ⟹ *interp e vs* ≡ *False*
**by** *simp-all*

**schematic-goal**
*interp ?f* (*?vs* :: (*′a* :: *ord*) *list*) = (∃(*x* :: *′a*). *x* < *y* ∧ ¬(∃(*z* :: *′a*). *v* < *z* ∨ ¬*False*))
**by** (*urule refl* **where** *unifier* = *reify-unify*)
**end**

The next examples are modification from [1].

## 27.3 Simple Arithmetic

**datatype** *add-expr* = *Var int* | *Add add-expr add-expr*

**fun** *eval-add-expr* :: *add-expr* ⇒ *int* **where**
*eval-add-expr* (*Var i*) = *i*
| *eval-add-expr* (*Add ex1 ex2*) = *eval-add-expr ex1* + *eval-add-expr ex2*

**lemma** *eval-add-expr-Var* [*reify-uhint* **where** *prio* = *Prio.LOW*]:
*e* ≡ *Var i* ⟹ *eval-add-expr e* ≡ *i* **by** *simp*

**lemma** *eval-add-expr-add* [*reify-uhint*]:
*e* ≡ *Add e1 e2* ⟹ *eval-add-expr e1* ≡ *m* ⟹ *eval-add-expr e2* ≡ *n* ⟹ *eval-add-expr e* ≡ *m* + *n*
**by** *simp*

**ML-command**‹
*val t1* = *Proof-Context.read-term-pattern* @{*context*} *eval-add-expr ?e*
*val t2* = *Proof-Context.read-term-pattern* @{*context*} *1* + (*2* + *7*) :: *int*
*val -* = *Unification-Util.log-unif-results* @{*context*} (*t1*, *t2*) (*reify-unify* [])
›

**schematic-goal** *eval-add-expr ?e* = (*1* + (*2* + *7*) :: *int*)
**by** (*urule refl* **where** *unifier* = *reify-unify*)

## 27.4 Arithmetic with Environment

**datatype** *mul-expr* =
  *Unit*
| *Var nat*
| *Mul mul-expr mul-expr*
| *Inv mul-expr*

**fun** *eval-mul-expr* :: *mul-expr* × *rat list* ⇒ *rat* **where**
  *eval-mul-expr* (*Unit*, Γ) = *1*
| *eval-mul-expr* (*Var i*, Γ) = Γ ! *i*
| *eval-mul-expr* (*Mul e1 e2*, Γ) = *eval-mul-expr* (*e1*, Γ) ∗ *eval-mul-expr* (*e2*, Γ)
| *eval-mul-expr* (*Inv e*, Γ) = *inverse* (*eval-mul-expr* (*e*, Γ))

Split *e* into an expression and an environment.

**lemma** [*reify-uhint* **where** *prio* = *Prio.VERY-LOW*]:
  *e* ≡ (*e1*, Γ) ⟹ *eval-mul-expr* (*e1*, Γ) ≡ *n* ⟹ *eval-mul-expr e* ≡ *n*
  **by** *simp*

Hints for environment lookup.

**lemma** [*reify-uhint* **where** *prio* = *Prio.LOW*]:
  *e* ≡ *Var* (*Suc p*) ⟹ Γ ≡ *s* # Δ ⟹ *n* ≡ *eval-mul-expr* (*Var p*, Δ) ⟹ *eval-mul-expr* (*e*, Γ) ≡ *n*
  **by** *simp*

**lemma** [*reify-uhint*]: *e* ≡ *Var 0* ⟹ Γ ≡ *n* # Θ ⟹ *eval-mul-expr* (*e*, Γ) ≡ *n*
  **by** *simp*

**lemma** [*reify-uhint*]:
  *e1* ≡ *Inv e2* ⟹ *n* ≡ *eval-mul-expr* (*e2*, Γ) ⟹ *eval-mul-expr* (*e1*, Γ) ≡ *inverse n*
  *e* ≡ *Mul e1 e2* ⟹ *m* ≡ *eval-mul-expr* (*e1*, Γ) ⟹
  *n* ≡ *eval-mul-expr* (*e2*, Γ) ⟹ *eval-mul-expr* (*e*, Γ) ≡ *m* ∗ *n*
  *e* ≡ *Unit* ⟹ *eval-mul-expr* (*e*, Γ) ≡ *1*
  **by** *simp-all*

**ML-command**‹
  *val t1* = *Proof-Context.read-term-pattern* @{*context*} *eval-mul-expr ?e*
  *val t2* = *Proof-Context.read-term-pattern* @{*context*} *1* ∗ *inverse 3* ∗ *5* :: *rat*
  *val -* = *Unification-Util.log-unif-results' 1* @{*context*} (*t2*, *t1*) (*reify-unify* [])
›

**schematic-goal** *eval-mul-expr ?e* = (*1* ∗ *inverse 3* ∗ *5* :: *rat*)
  **by** (*ufact refl* **where** *unifier* = *reify-unify*)

**end**

## References

[1] A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi. Hints in unification. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 84–98, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[2] K. Kappelmann, L. Bulwahn, and S. Willenbrink. Speccheck - specification-based testing for isabelle/ml. *Archive of Formal Proofs*, July 2021. https://isa-afp.org/entries/SpecCheck.html, Formal proof development.