The Isabelle/Isar Implementation

Makarius Wenzel

With Contributions by Stefan Berghofer,
Florian Haftmann and Larry Paulson

12 December 2021

Abstract

We describe the key concepts underlying the Isabelle/Isar implementation, including ML references for the most important functions. The aim is to give some insight into the overall system architecture, and provide clues on implementing applications within this framework.

Isabelle was not designed; it evolved. Not everyone likes this idea. Specification experts rightly abhor trial-and-error programming. They suggest that no one should write a program without first writing a complete formal specification. But university departments are not software houses. Programs like Isabelle are not products: when they have served their purpose, they are discarded.

Lawrence C. Paulson, “Isabelle: The Next 700 Theorem Provers”

As I did 20 years ago, I still fervently believe that the only way to make software secure, reliable, and fast is to make it small. Fight features.

Andrew S. Tanenbaum

One thing that UNIX does not need is more features. It is successful in part because it has a small number of good ideas that work well together. Merely adding features does not make it easier for users to do things — it just makes the manual thicker. The right solution in the right place is always more effective than haphazard hacking.

Rob Pike and Brian W. Kernighan

If you look at software today, through the lens of the history of engineering, it’s certainly engineering of a sort–but it’s the kind of engineering that people without the concept of the arch did. Most software today is very much like an Egyptian pyramid with millions of bricks piled on top of each other, with no structural integrity, but just done by brute force and thousands of slaves.

Alan Kay

Contents

List of Figures

Chapter 0

Isabelle/ML

Isabelle/ML is best understood as a certain culture based on Standard ML. Thus it is not a new programming language, but a certain way to use SML at an advanced level within the Isabelle environment. This covers a variety of aspects that are geared towards an efficient and robust platform for applications of formal logic with fully foundational proof construction — according to the well-known *LCF principle*. There is specific infrastructure with library modules to address the needs of this difficult task. For example, the raw parallel programming model of Poly/ML is presented as considerably more abstract concept of *futures*, which is then used to augment the inference kernel, Isar theory and proof interpreter, and PIDE document management.

The main aspects of Isabelle/ML are introduced below. These first-hand explanations should help to understand how proper Isabelle/ML is to be read and written, and to get access to the wealth of experience that is expressed in the source text and its history of changes.[1]

0.1 Style and orthography

The sources of Isabelle/Isar are optimized for *readability* and *maintainability*. The main purpose is to tell an informed reader what is really going on and how things really work. This is a non-trivial aim, but it is supported by a certain style of writing Isabelle/ML that has emerged from long years of system development.[2]

The main principle behind any coding style is *consistency*. For a single author of a small program this merely means "choose your style and stick to it". A complex project like Isabelle, with long years of development and different contributors, requires more standardization. A coding style that

[1] See https://isabelle.in.tum.de/repos/isabelle for the full Mercurial history. There are symbolic tags to refer to official Isabelle releases, as opposed to arbitrary *tip* versions that merely reflect snapshots that are never really up-to-date.

[2] See also the interesting style guide for OCaml https://caml.inria.fr/resources/doc/guides/guidelines.en.html which shares many of our means and ends.

is changed every few years or with every new contributor is no style at all, because consistency is quickly lost. Global consistency is hard to achieve, though. Nonetheless, one should always strive at least for local consistency of modules and sub-systems, without deviating from some general principles how to write Isabelle/ML.

In a sense, good coding style is like an *orthography* for the sources: it helps to read quickly over the text and see through the main points, without getting distracted by accidental presentation of free-style code.

0.1.1 Header and sectioning

Isabelle source files have a certain standardized header format (with precise spacing) that follows ancient traditions reaching back to the earliest versions of the system by Larry Paulson. See `~~/src/Pure/thm.ML`, for example.

The header includes at least `Title` and `Author` entries, followed by a prose description of the purpose of the module. The latter can range from a single line to several paragraphs of explanations.

The rest of the file is divided into chapters, sections, subsections, subsubsections, paragraphs etc. using a simple layout via ML comments as follows.

```
(**** chapter ****)

(*** section ***)

(** subsection **)

(* subsubsection *)

(*short paragraph*)

(*
  long paragraph,
  with more text
*)
```

As in regular typography, there is some extra space *before* section headings that are adjacent to plain text, but not other headings as in the example above.

The precise wording of the prose text given in these headings is chosen carefully to introduce the main theme of the subsequent formal ML text.

0.1.2 Naming conventions

Since ML is the primary medium to express the meaning of the source text, naming of ML entities requires special care.

Notation. A name consists of 1–3 *words* (rarely 4, but not more) that are separated by underscore. There are three variants concerning upper or lower case letters, which are used for certain ML categories as follows:

variant	example	ML categories
lower-case	`foo_bar`	values, types, record fields
capitalized	`Foo_Bar`	datatype constructors, structures, functors
upper-case	`FOO_BAR`	special values, exception constructors, signatures

For historical reasons, many capitalized names omit underscores, e.g. old-style `FooBar` instead of `Foo_Bar`. Genuine mixed-case names are *not* used, because clear division of words is essential for readability.[3]

A single (capital) character does not count as "word" in this respect: some Isabelle/ML names are suffixed by extra markers like this: `foo_barT`.

Name variants are produced by adding 1–3 primes, e.g. `foo'`, `foo''`, or `foo'''`, but not `foo''''` or more. Decimal digits scale better to larger numbers, e.g. `foo0`, `foo1`, `foo42`.

Scopes. Apart from very basic library modules, ML structures are not "opened", but names are referenced with explicit qualification, as in `Syntax.string_of_term` for example. When devising names for structures and their components it is important to aim at eye-catching compositions of both parts, because this is how they are seen in the sources and documentation. For the same reasons, aliases of well-known library functions should be avoided.

Local names of function abstraction or case/let bindings are typically shorter, sometimes using only rudiments of "words", while still avoiding cryptic shorthands. An auxiliary function called `helper`, `aux`, or `f` is considered bad style.

Example:

```
  (* RIGHT *)
```

[3] Camel-case was invented to workaround the lack of underscore in some early non-ASCII character sets. Later it became habitual in some language communities that are now strong in numbers.

```
fun print_foo ctxt foo =
  let
    fun print t = ... Syntax.string_of_term ctxt t ...
  in ... end;


(* RIGHT *)

fun print_foo ctxt foo =
  let
    val string_of_term = Syntax.string_of_term ctxt;
    fun print t = ... string_of_term t ...
  in ... end;


(* WRONG *)

val string_of_term = Syntax.string_of_term;

fun print_foo ctxt foo =
  let
    fun aux t = ... string_of_term ctxt t ...
  in ... end;
```

Specific conventions. Here are some specific name forms that occur frequently in the sources.

- A function that maps `foo` to `bar` is called `foo_to_bar` or `bar_of_foo` (never `foo2bar`, nor `bar_from_foo`, nor `bar_for_foo`, nor `bar4foo`).
- The name component `legacy` means that the operation is about to be discontinued soon.
- The name component `global` means that this works with the background theory instead of the regular local context (§1.1), sometimes for historical reasons, sometimes due a genuine lack of locality of the concept involved, sometimes as a fall-back for the lack of a proper context in the application code. Whenever there is a non-global variant available, the application should be migrated to use it with a proper local context.

- Variables of the main context types of the Isabelle/Isar framework (§1.1 and chapter 8) have firm naming conventions as follows:

  – theories are called `thy`, rarely `theory` (never `thry`)

  – proof contexts are called `ctxt`, rarely `context` (never `ctx`)

  – generic contexts are called `context`

  – local theories are called `lthy`, except for local theories that are treated as proof context (which is a semantic super-type)

  Variations with primed or decimal numbers are always possible, as well as semantic prefixes like `foo_thy` or `bar_ctxt`, but the base conventions above need to be preserved. This allows to emphasize their data flow via plain regular expressions in the text editor.

- The main logical entities (§2) have established naming convention as follows:

  – sorts are called `S`

  – types are called `T`, `U`, or `ty` (never `t`)

  – terms are called `t`, `u`, or `tm` (never `trm`)

  – certified types are called `cT`, rarely `T`, with variants as for types

  – certified terms are called `ct`, rarely `t`, with variants as for terms (never `ctrm`)

  – theorems are called `th`, or `thm`

  Proper semantic names override these conventions completely. For example, the left-hand side of an equation (as a term) can be called `lhs` (not `lhs_tm`). Or a term that is known to be a variable can be called `v` or `x`.

- Tactics (§4.2) are sufficiently important to have specific naming conventions. The name of a basic tactic definition always has a `_tac` suffix, the subgoal index (if applicable) is always called `i`, and the goal state (if made explicit) is usually called `st` instead of the somewhat misleading `thm`. Any other arguments are given before the latter two, and the general context is given first. Example:

  ```
  fun my_tac ctxt arg1 arg2 i st = ...
  ```

Note that the goal state `st` above is rarely made explicit, if tactic combinators (tacticals) are used as usual.

A tactic that requires a proof context needs to make that explicit as seen in the `ctxt` argument above. Do not refer to the background theory of `st` – it is not a proper context, but merely a formal certificate.

0.1.3 General source layout

The general Isabelle/ML source layout imitates regular type-setting conventions, augmented by the requirements for deeply nested expressions that are commonplace in functional programming.

Line length is limited to 80 characters according to ancient standards, but we allow as much as 100 characters (not more).[4] The extra 20 characters acknowledge the space requirements due to qualified library references in Isabelle/ML.

White-space is used to emphasize the structure of expressions, following mostly standard conventions for mathematical typesetting, as can be seen in plain TEX or LATEX. This defines positioning of spaces for parentheses, punctuation, and infixes as illustrated here:

```
val x = y + z * (a + b);
val pair = (a, b);
val record = {foo = 1, bar = 2};
```

Lines are normally broken *after* an infix operator or punctuation character. For example:

```
val x =
  a +
  b +
  c;
```

[4] Readability requires to keep the beginning of a line in view while watching its end. Modern wide-screen displays do not change the way how the human brain works. Sources also need to be printable on plain paper with reasonable font-size.

```
val tuple =
 (a,
  b,
  c);
```

Some special infixes (e.g. `|>`) work better at the start of the line, but punctuation is always at the end.

Function application follows the tradition of λ-calculus, not informal mathematics. For example: `f a b` for a curried function, or `g (a, b)` for a tupled function. Note that the space between `g` and the pair `(a, b)` follows the important principle of *compositionality*: the layout of `g p` does not change when `p` is refined to the concrete pair `(a,`

```
b).
```

Indentation uses plain spaces, never hard tabulators.[5]

Each level of nesting is indented by 2 spaces, sometimes 1, very rarely 4, never 8 or any other odd number.

Indentation follows a simple logical format that only depends on the nesting depth, not the accidental length of the text that initiates a level of nesting. Example:

```
(* RIGHT *)

if b then
  expr1_part1
  expr1_part2
else
  expr2_part1
  expr2_part2


(* WRONG *)

if b then expr1_part1
          expr1_part2
else expr2_part1
     expr2_part2
```

[5] Tabulators were invented to move the carriage of a type-writer to certain predefined positions. In software they could be used as a primitive run-length compression of consecutive spaces, but the precise result would depend on non-standardized text editor configuration.

The second form has many problems: it assumes a fixed-width font when viewing the sources, it uses more space on the line and thus makes it hard to observe its strict length limit (working against *readability*), it requires extra editing to adapt the layout to changes of the initial text (working against *maintainability*) etc.

For similar reasons, any kind of two-dimensional or tabular layouts, ASCII-art with lines or boxes of asterisks etc. should be avoided.

Complex expressions that consist of multi-clausal function definitions, `handle`, `case`, `let` (and combinations) require special attention. The syntax of Standard ML is quite ambitious and admits a lot of variance that can distort the meaning of the text.

Multiple clauses of `fun`, `fn`, `handle`, `case` get extra indentation to indicate the nesting clearly. Example:

```
(* RIGHT *)

fun foo p1 =
      expr1
  | foo p2 =
      expr2


(* WRONG *)

fun foo p1 =
  expr1
  | foo p2 =
  expr2
```

Body expressions consisting of `case` or `let` require care to maintain compositionality, to prevent loss of logical indentation where it is especially important to see the structure of the text. Example:

```
(* RIGHT *)

fun foo p1 =
      (case e of
        q1 => ...
      | q2 => ...)
  | foo p2 =
      let
        ...
      in
        ...
      end
```

```
(* WRONG *)

fun foo p1 = case e of
    q1 => ...
  | q2 => ...
  | foo p2 =
  let
    ...
  in
    ...
  end
```

Extra parentheses around `case` expressions are optional, but help to analyse the nesting based on character matching in the text editor.

There are two main exceptions to the overall principle of compositionality in the layout of complex expressions.

1. `if` expressions are iterated as if ML had multi-branch conditionals, e.g.

   ```
   (* RIGHT *)

   if b1 then e1
   else if b2 then e2
   else e3
   ```

2. `fn` abstractions are often layed-out as if they would lack any structure by themselves. This traditional form is motivated by the possibility to shift function arguments back and forth wrt. additional combinators. Example:

   ```
   (* RIGHT *)

   fun foo x y = fold (fn z =>
     expr)
   ```

Here the visual appearance is that of three arguments `x`, `y`, `z` in a row.

Such weakly structured layout should be use with great care. Here are some counter-examples involving `let` expressions:

```
(* WRONG *)

fun foo x = let
    val y = ...
  in ... end


(* WRONG *)

fun foo x = let
  val y = ...
in ... end


(* WRONG *)

fun foo x =
let
  val y = ...
in ... end


(* WRONG *)

fun foo x =
  let
    val y = ...
  in
    ... end
```

In general the source layout is meant to emphasize the structure of complex language expressions, not to pretend that SML had a completely different syntax (say that of Haskell, Scala, Java).

0.2 ML embedded into Isabelle/Isar

ML and Isar are intertwined via an open-ended bootstrap process that provides more and more programming facilities and logical content in an alternating manner. Bootstrapping starts from the raw environment of existing implementations of Standard ML (mainly Poly/ML).

Isabelle/Pure marks the point where the raw ML toplevel is superseded by Isabelle/ML within the Isar theory and proof language, with a uniform context for arbitrary ML values (see also §1.1). This formal environment holds ML compiler bindings, logical entities, and many other things.

Object-logics like Isabelle/HOL are built within the Isabelle/ML/Isar environment by introducing suitable theories with associated ML modules, either inlined within `.thy` files, or as separate `.ML` files that are loading from some theory. Thus Isabelle/HOL is defined as a regular user-space application within the Isabelle framework. Further add-on tools can be implemented in ML within the Isar context in the same manner: ML is part of the standard repertoire of Isabelle, and there is no distinction between "users" and "developers" in this respect.

0.2.1 Isar ML commands

The primary Isar source language provides facilities to "open a window" to the underlying ML compiler. Especially see the Isar commands **ML_file** and **ML**: both work the same way, but the source text is provided differently, via a file vs. inlined, respectively. Apart from embedding ML into the main theory definition like that, there are many more commands that refer to ML source, such as **setup** or **declaration**. Even more fine-grained embedding of ML into Isar is encountered in the proof method *tactic*, which refines the pending goal state via a given expression of type `tactic`.

ML Examples

The following artificial example demonstrates some ML toplevel declarations within the implicit Isar theory context. This is regular functional programming without referring to logical entities yet.

```
ML ‹
  fun factorial 0 = 1
    | factorial n = n * factorial (n - 1)
›
```

Here the ML environment is already managed by Isabelle, i.e. the `factorial` function is not yet accessible in the preceding paragraph, nor in a different theory that is independent from the current one in the import hierarchy.

Removing the above ML declaration from the source text will remove any trace of this definition, as expected. The Isabelle/ML toplevel environment is managed in a *stateless* way: in contrast to the raw ML toplevel, there are no global side-effects involved here.[6]

The next example shows how to embed ML into Isar proofs, using **ML_prf** instead of **ML**. As illustrated below, the effect on the ML environment is local to the whole proof body, but ignoring the block structure.

```
notepad
begin
 ML_prf ‹val a = 1›
  {
    ML_prf ‹val b = a + 1›
  } — Isar block structure ignored by ML environment
  ML_prf ‹val c = b + 1›
end
```

By side-stepping the normal scoping rules for Isar proof blocks, embedded ML code can refer to the different contexts and manipulate corresponding entities, e.g. export a fact from a block context.

Two further ML commands are useful in certain situations: **ML_val** and **ML_command** are *diagnostic* in the sense that there is no effect on the underlying environment, and can thus be used anywhere. The examples below produce long strings of digits by invoking `factorial`: **ML_val** takes care of printing the ML toplevel result, but **ML_command** is silent so we produce an explicit output message.

```
ML_val ‹factorial 100›
ML_command ‹writeln (string_of_int (factorial 100))›

notepad
begin
 ML_val ‹factorial 100›
  ML_command ‹writeln (string_of_int (factorial 100))›
end
```

0.2.2 Compile-time context

Whenever the ML compiler is invoked within Isabelle/Isar, the formal context is passed as a thread-local reference variable. Thus ML code may access the theory context during compilation, by reading or writing the (local) theory

[6] Such a stateless compilation environment is also a prerequisite for robust parallel compilation within independent nodes of the implicit theory development graph.

under construction. Note that such direct access to the compile-time context is rare. In practice it is typically done via some derived ML functions instead.

ML Reference

```
Context.the_generic_context: unit -> Context.generic
Context.>> : (Context.generic -> Context.generic) -> unit
ML_Thms.bind_thms: string * thm list -> unit
ML_Thms.bind_thm: string * thm -> unit
```

`Context.the_generic_context ()` refers to the theory context of the ML toplevel — at compile time. ML code needs to take care to refer to `Context.the_generic_context ()` correctly. Recall that evaluation of a function body is delayed until actual run-time.

`Context.>>` *f* applies context transformation *f* to the implicit context of the ML toplevel.

`ML_Thms.bind_thms` (*name*, *thms*) stores a list of theorems produced in ML both in the (global) theory context and the ML toplevel, associating it with the provided name.

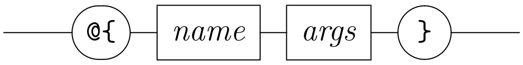
`ML_Thms.bind_thm` is similar to `ML_Thms.bind_thms` but refers to a singleton fact.

It is important to note that the above functions are really restricted to the compile time, even though the ML compiler is invoked at run-time. The majority of ML code either uses static antiquotations (§0.2.3) or refers to the theory or proof context at run-time, by explicit functional abstraction.

0.2.3 Antiquotations

A very important consequence of embedding ML into Isar is the concept of *ML antiquotation*. The standard token language of ML is augmented by special syntactic entities of the following form:

antiquote



Here *name* and *args* are outer syntax categories, as defined in [19].

A regular antiquotation @{*name args*} processes its arguments by the usual means of the Isar source language, and produces corresponding ML source text, either as literal *inline* text (e.g. @{*term t*}) or abstract *value* (e.g. @{*thm th*}). This pre-compilation scheme allows to refer to formal entities in a robust manner, with proper static scoping and with some degree of logical checking of small portions of the code.

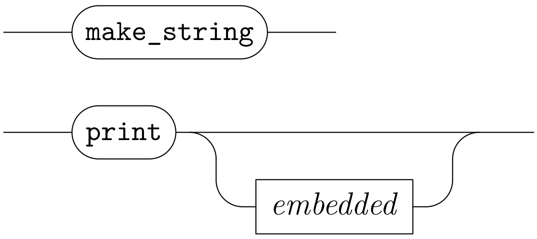
0.2.4 Printing ML values

The ML compiler knows about the structure of values according to their static type, and can print them in the manner of its toplevel, although the details are non-portable. The antiquotations *make_string* and *print* provide a quasi-portable way to refer to this potential capability of the underlying ML system in generic Isabelle/ML sources.

This is occasionally useful for diagnostic or demonstration purposes. Note that production-quality tools require proper user-level error messages, avoiding raw ML values in the output.

ML Antiquotations

make_string	:	*ML_antiquotation*
print	:	*ML_antiquotation*



@{*make_string*} inlines a function to print arbitrary values similar to the ML toplevel. The result is compiler dependent and may fall back on "?" in certain situations. The value of configuration option *ML_print_depth* determines further details of output.

@{*print f*} uses the ML function *f*: *string* −> *unit* to output the result of @{*make_string*} above, together with the source position of the antiquotation. The default output function is `writeln`.

ML

Examples

The following artificial examples show how to produce adhoc output of ML values for debugging purposes.

```
ML_val ‹
  val x = 42;
  val y = true;

  writeln (make_string {x = x, y = y});

  print {x = x, y = y};
  print‹tracing› {x = x, y = y};
›
```

0.3 Canonical argument order

Standard ML is a language in the tradition of λ-calculus and *higher-order functional programming*, similar to OCaml, Haskell, or Isabelle/Pure and HOL as logical languages. Getting acquainted with the native style of representing functions in that setting can save a lot of extra boiler-plate of redundant shuffling of arguments, auxiliary abstractions etc.

Functions are usually *curried*: the idea of turning arguments of type τ_i (for $i \in \{1, \ldots n\}$) into a result of type τ is represented by the iterated function space $\tau_1 \to \ldots \to \tau_n \to \tau$. This is isomorphic to the well-known encoding via tuples $\tau_1 \times \ldots \times \tau_n \to \tau$, but the curried version fits more smoothly into the basic calculus.[7]

Currying gives some flexibility due to *partial application*. A function f: $\tau_1 \to \tau_2 \to \tau$ can be applied to x: τ_1 and the remaining $(f\ x)$: $\tau_2 \to \tau$ passed to another function etc. How well this works in practice depends on the order of arguments. In the worst case, arguments are arranged erratically, and using a function in a certain situation always requires some glue code. Thus we would get exponentially many opportunities to decorate the code with meaningless permutations of arguments.

This can be avoided by *canonical argument order*, which observes certain standard patterns and minimizes adhoc permutations in their application. In Isabelle/ML, large portions of text can be written without auxiliary operations like *swap*: $\alpha \times \beta \to \beta \times \alpha$ or C: $(\alpha \to \beta \to \gamma) \to (\beta \to \alpha \to \gamma)$ (the latter is not present in the Isabelle/ML library).

[7] The difference is even more significant in HOL, because the redundant tuple structure needs to be accommodated extraneous proof steps.

The main idea is that arguments that vary less are moved further to the left than those that vary more. Two particularly important categories of functions are *selectors* and *updates*.

The subsequent scheme is based on a hypothetical set-like container of type β that manages elements of type α. Both the names and types of the associated operations are canonical for Isabelle/ML.

kind	canonical name and type
selector	member: $\beta \to \alpha \to \mathit{bool}$
update	insert: $\alpha \to \beta \to \beta$

Given a container B: β, the partially applied *member B* is a predicate over elements $\alpha \to \mathit{bool}$, and thus represents the intended denotation directly. It is customary to pass the abstract predicate to further operations, not the concrete container. The argument order makes it easy to use other combinators: *forall* (*member B*) *list* will check a list of elements for membership in B etc. Often the explicit *list* is pointless and can be contracted to *forall* (*member B*) to get directly a predicate again.

In contrast, an update operation varies the container, so it moves to the right: *insert a* is a function $\beta \to \beta$ to insert a value a. These can be composed naturally as $\mathit{insert}\ c \circ \mathit{insert}\ b \circ \mathit{insert}\ a$. The slightly awkward inversion of the composition order is due to conventional mathematical notation, which can be easily amended as explained below.

0.3.1 Forward application and composition

Regular function application and infix notation works best for relatively deeply structured expressions, e.g. $h\ (f\ x\ y + g\ z)$. The important special case of *linear transformation* applies a cascade of functions f_n $(\ldots$ $(f_1$ $x))$. This becomes hard to read and maintain if the functions are themselves given as complex expressions. The notation can be significantly improved by introducing *forward* versions of application and composition as follows:

$$\begin{array}{lcl} x \mathbin{|\!>} f & \equiv & f\,x \\ (f \mathbin{\#\!>} g)\ x & \equiv & x \mathbin{|\!>} f \mathbin{|\!>} g \end{array}$$

This enables to write conveniently $x \mathbin{|\!>} f_1 \mathbin{|\!>} \ldots \mathbin{|\!>} f_n$ or $f_1 \mathbin{\#\!>} \ldots \mathbin{\#\!>} f_n$ for its functional abstraction over x.

There is an additional set of combinators to accommodate multiple results (via pairs) that are passed on as multiple arguments (via currying).

$$\begin{array}{lcl} (x,\ y) \mid\!-\!> f & \equiv & f\ x\ y \\ (f\ \#\!-\!>\ g)\ x & \equiv & x \mid\!> f \mid\!-\!> g \end{array}$$

ML Reference

```
infix |> : 'a * ('a -> 'b) -> 'b
infix |-> : ('c * 'a) * ('c -> 'a -> 'b) -> 'b
infix #> : ('a -> 'b) * ('b -> 'c) -> 'a -> 'c
infix #-> : ('a -> 'c * 'b) * ('c -> 'b -> 'd) -> 'a -> 'd
```

0.3.2 Canonical iteration

As explained above, a function f: $\alpha \to \beta \to \beta$ can be understood as update on a configuration of type β, parameterized by an argument of type α. Given a: α the partial application (f a): $\beta \to \beta$ operates homogeneously on β. This can be iterated naturally over a list of parameters $[a_1, \ldots, a_n]$ as f a_1 #> … #> f a_n. The latter expression is again a function $\beta \to \beta$. It can be applied to an initial configuration b: β to start the iteration over the given list of arguments: each a in $a_1, \ldots, a_n$ is applied consecutively by updating a cumulative configuration.

The *fold* combinator in Isabelle/ML lifts a function *f* as above to its iterated version over a list of arguments. Lifting can be repeated, e.g. (*fold* ∘ *fold*) *f* iterates over a list of lists as expected.

The variant *fold_rev* works inside-out over the list of arguments, such that *fold_rev f* ≡ *fold f* ∘ *rev* holds.

The *fold_map* combinator essentially performs *fold* and *map* simultaneously: each application of *f* produces an updated configuration together with a side-result; the iteration collects all such side-results as a separate list.

ML Reference

```
fold: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
fold_rev: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
fold_map: ('a -> 'b -> 'c * 'b) -> 'a list -> 'b -> 'c list * 'b
```

`fold` *f* lifts the parametrized update function *f* to a list of parameters.

`fold_rev` *f* is similar to `fold` *f*, but works inside-out, as if the list would be reversed.

`fold_map` *f* lifts the parametrized update function *f* (with side-result) to a list of parameters and cumulative side-results.

! The literature on functional programming provides a confusing multitude of combinators called *foldl*, *foldr* etc. SML97 provides its own variations as `List.foldl` and `List.foldr`, while the classic Isabelle library also has the historic `Library.foldl` and `Library.foldr`. To avoid unnecessary complication, all these historical versions should be ignored, and the canonical `fold` (or `fold_rev`) used exclusively.

ML

Examples

The following example shows how to fill a text buffer incrementally by adding strings, either individually or from a given list.

```
ML_val ‹
  val s =
    Buffer.empty
    |> Buffer.add "digits: "
    |> fold (Buffer.add o string_of_int) (0 upto 9)
    |> Buffer.content;

  assert (s = "digits: 0123456789");
›
```

Note how `fold (Buffer.add o string_of_int)` above saves an extra `map` over the given list. This kind of peephole optimization reduces both the code size and the tree structures in memory (“deforestation”), but it requires some practice to read and write fluently.

The next example elaborates the idea of canonical iteration, demonstrating fast accumulation of tree content using a text buffer.

```
ML ‹
  datatype tree = Text of string | Elem of string * tree list;

  fun slow_content (Text txt) = txt
    | slow_content (Elem (name, ts)) =
        "<" ^ name ^ ">" ^
        implode (map slow_content ts) ^
        "</" ^ name ^ ">"

  fun add_content (Text txt) = Buffer.add txt
```

```
    | add_content (Elem (name, ts)) =
        Buffer.add ("<" ^ name ^ ">") #>
        fold add_content ts #>
        Buffer.add ("</" ^ name ^ ">");

  fun fast_content tree =
    Buffer.empty |> add_content tree |> Buffer.content;
›
```

The slowness of `slow_content` is due to the `implode` of the recursive results, because it copies previously produced strings again and again.

The incremental `add_content` avoids this by operating on a buffer that is passed through in a linear fashion. Using `#>` and contraction over the actual buffer argument saves some additional boiler-plate. Of course, the two `Buffer.add` invocations with concatenated strings could have been split into smaller parts, but this would have obfuscated the source without making a big difference in performance. Here we have done some peephole-optimization for the sake of readability.

Another benefit of `add_content` is its "open" form as a function on buffers that can be continued in further linear transformations, folding etc. Thus it is more compositional than the naive `slow_content`. As realistic example, compare the old-style `Term.maxidx_of_term: term -> int` with the newer `Term.maxidx_term: term -> int -> int` in Isabelle/Pure.

Note that `fast_content` above is only defined as example. In many practical situations, it is customary to provide the incremental `add_content` only and leave the initialization and termination to the concrete application to the user.

0.4 Message output channels

Isabelle provides output channels for different kinds of messages: regular output, high-volume tracing information, warnings, and errors.

Depending on the user interface involved, these messages may appear in different text styles or colours. The standard output for batch sessions prefixes each line of warnings by `###` and errors by `***`, but leaves anything else unchanged. The message body may contain further markup and formatting, which is routinely used in the Prover IDE [20].

Messages are associated with the transaction context of the running Isar command. This enables the front-end to manage commands and resulting messages together. For example, after deleting a command from a given theory document version, the corresponding message output can be retracted from the display.

ML Reference

```
writeln: string -> unit
tracing: string -> unit
warning: string -> unit
error: string -> 'a
```

`writeln` *text* outputs *text* as regular message. This is the primary message output operation of Isabelle and should be used by default.

`tracing` *text* outputs *text* as special tracing message, indicating potential high-volume output to the front-end (hundreds or thousands of messages issued by a single command). The idea is to allow the user-interface to downgrade the quality of message display to achieve higher throughput.

Note that the user might have to take special actions to see tracing output, e.g. switch to a different output window. So this channel should not be used for regular output.

`warning` *text* outputs *text* as warning, which typically means some extra emphasis on the front-end side (color highlighting, icons, etc.).

`error` *text* raises exception `ERROR` *text* and thus lets the Isar toplevel print *text* on the error channel, which typically means some extra emphasis on the front-end side (color highlighting, icons, etc.).

This assumes that the exception is not handled before the command terminates. Handling exception `ERROR` *text* is a perfectly legal alternative: it means that the error is absorbed without any message output.

! The actual error channel is accessed via `Output.error_message`, but this is normally not used directly in user code.

! Regular Isabelle/ML code should output messages exclusively by the official channels. Using raw I/O on *stdout* or *stderr* instead (e.g. via `TextIO.output`) is apt to cause problems in the presence of parallel and asynchronous processing of Isabelle theories. Such raw output might be displayed by the front-end in some system console log, with a low chance that the user will ever see it. Moreover, as a genuine side-effect on global process channels, there is no proper way to retract output when Isar command transactions are reset by the system.

! The message channels should be used in a message-oriented manner. This means that multi-line output that logically belongs together is issued by a single invocation of `writeln` etc. with the functional concatenation of all message constituents.

ML Examples

The following example demonstrates a multi-line warning. Note that in some situations the user sees only the first line, so the most important point should be made first.

```
ML_command ‹
  warning (cat_lines
   ["Beware the Jabberwock, my son!",
    "The jaws that bite, the claws that catch!",
    "Beware the Jubjub Bird, and shun",
    "The frumious Bandersnatch!"]);
›
```

An alternative is to make a paragraph of freely-floating words as follows.

```
ML_command ‹
  warning (Pretty.string_of (Pretty.para
    "Beware the Jabberwock, my son! \
    \The jaws that bite, the claws that catch! \
    \Beware the Jubjub Bird, and shun \
    \The frumious Bandersnatch!"))
›
```

This has advantages with variable window / popup sizes, but might make it harder to search for message content systematically, e.g. by other tools or by humans expecting the "verse" of a formal message in a fixed layout.

0.5 Exceptions

The Standard ML semantics of strict functional evaluation together with exceptions is rather well defined, but some delicate points need to be observed to avoid that ML programs go wrong despite static type-checking. Exceptions in Isabelle/ML are subsequently categorized as follows.

Regular user errors. These are meant to provide informative feedback about malformed input etc.

The *error* function raises the corresponding `ERROR` exception, with a plain text message as argument. `ERROR` exceptions can be handled internally, in order to be ignored, turned into other exceptions, or cascaded by appending messages. If the corresponding Isabelle/Isar command terminates with an `ERROR` exception state, the system will print the result on the error channel (see §0.4).

It is considered bad style to refer to internal function names or values in ML source notation in user error messages. Do not use @{*make_string*} nor @{*here*}!

Grammatical correctness of error messages can be improved by *omitting* final punctuation: messages are often concatenated or put into a larger context (e.g. augmented with source position). Note that punctuation after formal entities (types, terms, theorems) is particularly prone to user confusion.

Program failures. There is a handful of standard exceptions that indicate general failure situations, or failures of core operations on logical entities (types, terms, theorems, theories, see chapter 2).

These exceptions indicate a genuine breakdown of the program, so the main purpose is to determine quickly what has happened where. Traditionally, the (short) exception message would include the name of an ML function, although this is no longer necessary, because the ML runtime system attaches detailed source position stemming from the corresponding `raise` keyword.

User modules can always introduce their own custom exceptions locally, e.g. to organize internal failures robustly without overlapping with existing exceptions. Exceptions that are exposed in module signatures require extra care, though, and should *not* be introduced by default. Surprise by users of a module can be often minimized by using plain user errors instead.

Interrupts. These indicate arbitrary system events: both the ML runtime system and the Isabelle/ML infrastructure signal various exceptional situations by raising the special `Exn.Interrupt` exception in user code.

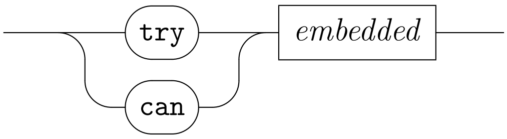
This is the one and only way that physical events can intrude an Isabelle/ML program. Such an interrupt can mean out-of-memory, stack overflow, timeout, internal signaling of threads, or a POSIX process signal. An Isabelle/ML program that intercepts interrupts becomes dependent on physical effects of the environment. Even worse, exception handling patterns that are too general by accident, e.g. by misspelled exception constructors, will cover interrupts unintentionally and thus render the program semantics ill-defined.

Note that the Interrupt exception dates back to the original SML90 language definition. It was excluded from the SML97 version to avoid its malign impact on ML program semantics, but without providing a viable alternative. Isabelle/ML recovers physical interruptibility (which is an indispensable tool to implement managed evaluation of command transactions), but requires user code to be strictly transparent wrt. interrupts.

Isabelle/ML user code needs to terminate promptly on interruption, without guessing at its meaning to the system infrastructure. Temporary handling of interrupts for cleanup of global resources etc. needs to be followed immediately by re-raising of the original exception.

ML Reference

```
try: ('a -> 'b) -> 'a -> 'b option
can: ('a -> 'b) -> 'a -> bool
exception ERROR of string
exception Fail of string
Exn.is_interrupt: exn -> bool
Exn.reraise: exn -> 'a
Runtime.exn_trace: (unit -> 'a) -> 'a
```



`try` *f x* makes the partiality of evaluating *f x* explicit via the option datatype. Interrupts are *not* handled here, i.e. this form serves as safe replacement for the *unsafe* version `(SOME` *f x* `handle _ => NONE)` that is occasionally seen in books about SML97, but not in Isabelle/ML.

`can` is similar to `try` with more abstract result.

`ERROR` *msg* represents user errors; this exception is normally raised indirectly via the `error` function (see §0.4).

`Fail` *msg* represents general program failures.

`Exn.is_interrupt` identifies interrupts robustly, without mentioning concrete exception constructors in user code. Handled interrupts need to be re-raised promptly!

`Exn.reraise` *exn* raises exception *exn* while preserving its implicit position information (if possible, depending on the ML platform).

`Runtime.exn_trace (fn () =>` *e*`)` evaluates expression *e* while printing a full trace of its stack of nested exceptions (if possible, depending on the ML platform).

Inserting `Runtime.exn_trace` into ML code temporarily is useful for debugging, but not suitable for production code.

ML Antiquotations

try	:	*ML_antiquotation*
can	:	*ML_antiquotation*
assert	:	*ML_antiquotation*
undefined	:	*ML_antiquotation*

@{*try*} and {*can*} are similar to the corresponding functions, but the argument is taken directly as ML expression: functional abstraction and application is done implicitly.

@{*assert*} inlines a function `bool -> unit` that raises `Fail` if the argument is `false`. Due to inlining the source position of failed assertions is included in the error output.

@{*undefined*} inlines `raise Match`, i.e. the ML program behaves as in some function application of an undefined case.

ML Examples

We define a total version of division: any failures are swept under the carpet and mapped to a default value. Thus division-by-zero becomes 0, but there could be other exceptions like overflow that produce the same result (for unbounded integers this does not happen).

```
ML ‹
  fun div_total x y = try‹x div y› |> the_default 0;

  assert (div_total 1 0 = 0);
›
```

The ML function `undefined` is defined in `~~/src/Pure/library.ML` as follows:

ML ‹*fun undefined _ = raise Match*›

The following variant uses the antiquotation *undefined* instead:

ML ‹*fun undefined _ =* ***undefined***›

Here is the same, using control-symbol notation for the antiquotation, with special rendering of `\<^undefined>`:

ML ‹*fun undefined _ =* ***undefined***›

Semantically, all forms are equivalent to a function definition without any clauses, but that is syntactically not allowed in ML.

0.6 Strings of symbols

A *symbol* constitutes the smallest textual unit in Isabelle/ML — raw ML characters are normally not encountered at all. Isabelle strings consist of a sequence of symbols, represented as a packed string or an exploded list of strings. Each symbol is in itself a small string, which has either one of the following forms:

1. a single ASCII character "*c*", for example "`a`",
2. a codepoint according to UTF-8 (non-ASCII byte sequence),
3. a regular symbol "`\<ident>`", for example "`\<alpha>`",
4. a control symbol "`\<^ident>`", for example "`\<^bold>`",

The *ident* syntax for symbol names is *letter* (*letter* | *digit*)*, where *letter* = *A..Za..z* and *digit* = 0..9. There are infinitely many regular symbols and control symbols, but a fixed collection of standard symbols is treated specifically. For example, "`\<alpha>`" is classified as a letter, which means it may occur within regular Isabelle identifiers.

The character set underlying Isabelle symbols is 7-bit ASCII, but 8-bit character sequences are passed-through unchanged. Unicode/UCS data in UTF-8 encoding is processed in a non-strict fashion, such that well-formed code sequences are recognized accordingly. Unicode provides its own collection of mathematical symbols, but within the core Isabelle/ML world there is no link to the standard collection of Isabelle regular symbols.

Output of Isabelle symbols depends on the print mode. For example, the standard LaTeX setup of the Isabelle document preparation system would present "`\<alpha>`" as α, and "`\<^bold>\<alpha>`" as $\boldsymbol{\alpha}$. On-screen rendering usually works by mapping a finite subset of Isabelle symbols to suitable Unicode characters.

ML Reference

```
type Symbol.symbol = string
Symbol.explode: string -> Symbol.symbol list
Symbol.is_letter: Symbol.symbol -> bool
Symbol.is_digit: Symbol.symbol -> bool
Symbol.is_quasi: Symbol.symbol -> bool
Symbol.is_blank: Symbol.symbol -> bool
```

```
type Symbol.sym
Symbol.decode: Symbol.symbol -> Symbol.sym
```

Type `Symbol.symbol` represents individual Isabelle symbols.

`Symbol.explode` *str* produces a symbol list from the packed form. This function supersedes `String.explode` for virtually all purposes of manipulating text in Isabelle![8]

`Symbol.is_letter`, `Symbol.is_digit`, `Symbol.is_quasi`, `Symbol.is_blank` classify standard symbols according to fixed syntactic conventions of Isabelle, cf. [19].

Type `Symbol.sym` is a concrete datatype that represents the different kinds of symbols explicitly, with constructors `Symbol.Char`, `Symbol.UTF8`, `Symbol.Sym`, `Symbol.Control`, `Symbol.Malformed`.

`Symbol.decode` converts the string representation of a symbol into the datatype version.

Historical note. In the original SML90 standard the primitive ML type `char` did not exists, and `explode: string -> string list` produced a list of singleton strings like `raw_explode: string -> string list` in

[8] The runtime overhead for exploded strings is mainly that of the list structure: individual symbols that happen to be a singleton string do not require extra memory in Poly/ML.

Isabelle/ML today. When SML97 came out, Isabelle did not adopt its somewhat anachronistic 8-bit or 16-bit characters, but the idea of exploding a string into a list of small strings was extended to "symbols" as explained above. Thus Isabelle sources can refer to an infinite store of user-defined symbols, without having to worry about the multitude of Unicode encodings that have emerged over the years.

0.7 Basic data types

The basis library proposal of SML97 needs to be treated with caution. Many of its operations simply do not fit with important Isabelle/ML conventions (like "canonical argument order", see §0.3), others cause problems with the parallel evaluation model of Isabelle/ML (such as `TextIO.print` or `OS.Process.system`).

Subsequently we give a brief overview of important operations on basic ML data types.

0.7.1 Characters

ML **Reference**

```
type char
```

Type `char` is *not* used. The smallest textual unit in Isabelle is represented as a "symbol" (see §0.6).

0.7.2 Strings

ML **Reference**

```
type string
```

Type `string` represents immutable vectors of 8-bit characters. There are operations in SML to convert back and forth to actual byte vectors, which are seldom used.

This historically important raw text representation is used for Isabelle-specific purposes with the following implicit substructures packed into the string content:

1. sequence of Isabelle symbols (see also §0.6), with `Symbol.explode` as key operation;
2. XML tree structure via YXML (see also [18]), with `YXML.parse_body` as key operation.

Note that Isabelle/ML string literals may refer Isabelle symbols like "`\<alpha>`" natively, *without* escaping the backslash. This is a consequence of Isabelle treating all source text as strings of symbols, instead of raw characters.

ML Examples

The subsequent example illustrates the difference of physical addressing of bytes versus logical addressing of symbols in Isabelle strings.

```
ML_val ‹
  val s = "𝒜";

  assert (length (Symbol.explode s) = 1);
  assert (size s = 4);
›
```

Note that in Unicode renderings of the symbol 𝒜, variations of encodings like UTF-8 or UTF-16 pose delicate questions about the multi-byte representations of its codepoint, which is outside of the 16-bit address space of the original Unicode standard from the 1990-ies. In Isabelle/ML it is just "`\<A>`" literally, using plain ASCII characters beyond any doubts.

0.7.3 Integers

ML Reference

```
type int
```

Type `int` represents regular mathematical integers, which are *unbounded.* Overflow is treated properly, but should never happen in practice.[9]

Structure `IntInf` of SML97 is obsolete and superseded by `Int`. Structure `Integer` in `~~/src/Pure/General/integer.ML` provides some additional operations.

[9] The size limit for integer bit patterns in memory is 64 MB for 32-bit Poly/ML, and much higher for 64-bit systems.

0.7.4 Rational numbers

ML Reference

```
type Rat.rat
```

Type `Rat.rat` represents rational numbers, based on the unbounded integers of Poly/ML.

Literal rationals may be written with special antiquotation syntax `@`*int*`/`*nat* or `@`*int* (without any white space). For example `@~1/4` or `@10`. The ML toplevel pretty printer uses the same format.

Standard operations are provided via ad-hoc overloading of `+`, `-`, `*`, `/`, etc.

0.7.5 Time

ML Reference

```
type Time.time
seconds: real -> Time.time
```

Type `Time.time` represents time abstractly according to the SML97 basis library definition. This is adequate for internal ML operations, but awkward in concrete time specifications.

`seconds` *s* turns the concrete scalar *s* (measured in seconds) into an abstract time value. Floating point numbers are easy to use as configuration options in the context (see §1.1.5) or system options that are maintained externally.

0.7.6 Options

ML Reference

```
Option.map: ('a -> 'b) -> 'a option -> 'b option
is_some: 'a option -> bool
is_none: 'a option -> bool
the: 'a option -> 'a
these: 'a list option -> 'a list
the_list: 'a option -> 'a list
the_default: 'a -> 'a option -> 'a
```

Apart from `Option.map` most other operations defined in structure `Option` are alien to Isabelle/ML and never used. The operations shown above are defined in `~~/src/Pure/General/basics.ML`.

0.7.7 Lists

Lists are ubiquitous in ML as simple and light-weight "collections" for many everyday programming tasks. Isabelle/ML provides important additions and improvements over operations that are predefined in the SML97 library.

ML Reference

```
cons: 'a -> 'a list -> 'a list
member: ('b * 'a -> bool) -> 'a list -> 'b -> bool
insert: ('a * 'a -> bool) -> 'a -> 'a list -> 'a list
remove: ('b * 'a -> bool) -> 'b -> 'a list -> 'a list
update: ('a * 'a -> bool) -> 'a -> 'a list -> 'a list
```

`cons` x xs evaluates to $x :: xs$.

Tupled infix operators are a historical accident in Standard ML. The curried `cons` amends this, but it should be only used when partial application is required.

`member`, `insert`, `remove`, `update` treat lists as a set-like container that maintains the order of elements. See `~~/src/Pure/library.ML` for the full specifications (written in ML). There are some further derived operations like `union` or `inter`.

Note that `insert` is conservative about elements that are already a `member` of the list, while `update` ensures that the latest entry is always put in front. The latter discipline is often more appropriate in declarations of context data (§1.1.4) that are issued by the user in Isar source: later declarations take precedence over earlier ones.

ML Examples

Using canonical `fold` together with `cons` (or similar standard operations) alternates the orientation of data. The is quite natural and should not be altered forcible by inserting extra applications of `rev`. The alternative `fold_rev` can be used in the few situations, where alternation should be prevented.

ML_val ‹

```
  val items = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

  val list1 = fold cons items [];
  assert (list1 = rev items);

  val list2 = fold_rev cons items [];
  assert (list2 = items);
```

›

The subsequent example demonstrates how to *merge* two lists in a natural way.

ML_val ‹

```
  fun merge_lists eq (xs, ys) = fold_rev (insert eq) ys xs;
```

›

Here the first list is treated conservatively: only the new elements from the second list are inserted. The inside-out order of insertion via `fold_rev` attempts to preserve the order of elements in the result.

This way of merging lists is typical for context data (§1.1.4). See also `merge` as defined in `~~/src/Pure/library.ML`.

0.7.8 Association lists

The operations for association lists interpret a concrete list of pairs as a finite function from keys to values. Redundant representations with multiple occurrences of the same key are implicitly normalized: lookup and update only take the first occurrence into account.

```
AList.lookup: ('a * 'b -> bool) -> ('b * 'c) list -> 'a -> 'c option
AList.defined: ('a * 'b -> bool) -> ('b * 'c) list -> 'a -> bool
AList.update: ('a * 'a -> bool) -> 'a * 'b -> ('a * 'b) list -> ('a * 'b) list
```

`AList.lookup`, `AList.defined`, `AList.update` implement the main "framework operations" for mappings in Isabelle/ML, following standard conventions for their names and types.

Note that a function called `lookup` is obliged to express its partiality via an explicit option element. There is no choice to raise an exception, without changing the name to something like *the_element* or *get*.

The *defined* operation is essentially a contraction of `is_some` and `lookup`, but this is sufficiently frequent to justify its independent existence. This also gives the implementation some opportunity for peephole optimization.

Association lists are adequate as simple implementation of finite mappings in many practical situations. A more advanced table structure is defined in `~~/src/Pure/General/table.ML`; that version scales easily to thousands or millions of elements.

0.7.9 Unsynchronized references

ML **Reference**

```
type 'a Unsynchronized.ref
Unsynchronized.ref: 'a -> 'a Unsynchronized.ref
! : 'a Unsynchronized.ref -> 'a
infix := : 'a Unsynchronized.ref * 'a -> unit
```

Due to ubiquitous parallelism in Isabelle/ML (see also §0.8), the mutable reference cells of Standard ML are notorious for causing problems. In a highly parallel system, both correctness *and* performance are easily degraded when using mutable data.

The unwieldy name of `Unsynchronized.ref` for the constructor for references in Isabelle/ML emphasizes the inconveniences caused by mutability. Existing operations `!` and `:=` are unchanged, but should be used with special precautions, say in a strictly local situation that is guaranteed to be restricted to sequential evaluation — now and in the future.

! Never `open Unsynchronized`, not even in a local scope! Pretending that mutable state is no problem is a very bad idea.

0.8 Thread-safe programming

Multi-threaded execution has become an everyday reality in Isabelle since Poly/ML 5.2.1 and Isabelle2008. Isabelle/ML provides implicit and explicit parallelism by default, and there is no way for user-space tools to "opt out". ML programs that are purely functional, output messages only via the official channels (§0.4), and do not intercept interrupts (§0.5) can participate in the multi-threaded environment immediately without further ado.

More ambitious tools with more fine-grained interaction with the environment need to observe the principles explained below.

0.8.1 Multi-threading with shared memory

Multiple threads help to organize advanced operations of the system, such as real-time conditions on command transactions, sub-components with explicit communication, general asynchronous interaction etc. Moreover, parallel evaluation is a prerequisite to make adequate use of the CPU resources that are available on multi-core systems.[10]

Isabelle/Isar exploits the inherent structure of theories and proofs to support *implicit parallelism* to a large extent. LCF-style theorem proving provides almost ideal conditions for that, see also [21]. This means, significant parts of theory and proof checking is parallelized by default. In Isabelle2013, a maximum speedup-factor of 3.5 on 4 cores and 6.5 on 8 cores can be expected [22].

ML threads lack the memory protection of separate processes, and operate concurrently on shared heap memory. This has the advantage that results of independent computations are directly available to other threads: abstract values can be passed without copying or awkward serialization that is typically required for separate processes.

To make shared-memory multi-threading work robustly and efficiently, some programming guidelines need to be observed. While the ML system is responsible to maintain basic integrity of the representation of ML values in memory, the application programmer needs to ensure that multi-threaded execution does not break the intended semantics.

! To participate in implicit parallelism, tools need to be thread-safe. A single ill-behaved tool can affect the stability and performance of the whole system.

Apart from observing the principles of thread-safeness passively, advanced tools may also exploit parallelism actively, e.g. by using library functions for parallel list operations (§0.9.1).

! Parallel computing resources are managed centrally by the Isabelle/ML infrastructure. User programs should not fork their own ML threads to perform heavy computations.

[10] Multi-core computing does not mean that there are "spare cycles" to be wasted. It means that the continued exponential speedup of CPU performance due to "Moore's Law" follows different rules: clock frequency has reached its peak around 2005, and applications need to be parallelized in order to avoid a perceived loss of performance. See also [17].

0.8.2 Critical shared resources

Thread-safeness is mainly concerned about concurrent read/write access to shared resources, which are outside the purely functional world of ML. This covers the following in particular.

- Global references (or arrays), i.e. mutable memory cells that persist over several invocations of associated operations.[11]
- Global state of the running Isabelle/ML process, i.e. raw I/O channels, environment variables, current working directory.
- Writable resources in the file-system that are shared among different threads or external processes.

Isabelle/ML provides various mechanisms to avoid critical shared resources in most situations. As last resort there are some mechanisms for explicit synchronization. The following guidelines help to make Isabelle/ML programs work smoothly in a concurrent environment.

- Avoid global references altogether. Isabelle/Isar maintains a uniform context that incorporates arbitrary data declared by user programs (§1.1.4). This context is passed as plain value and user tools can get/map their own data in a purely functional manner. Configuration options within the context (§1.1.5) provide simple drop-in replacements for historic reference variables.
- Keep components with local state information re-entrant. Instead of poking initial values into (private) global references, a new state record can be created on each invocation, and passed through any auxiliary functions of the component. The state record contain mutable references in special situations, without requiring any synchronization, as long as each invocation gets its own copy and the tool itself is single-threaded.
- Avoid raw output on *stdout* or *stderr*. The Poly/ML library is thread-safe for each individual output operation, but the ordering of parallel invocations is arbitrary. This means raw output will appear on some system console with unpredictable interleaving of atomic chunks.

  Note that this does not affect regular message output channels (§0.4). An official message id is associated with the command transaction from

[11] This is independent of the visibility of such mutable values in the toplevel scope.

where it originates, independently of other transactions. This means each running Isar command has effectively its own set of message channels, and interleaving can only happen when commands use parallelism internally (and only at message boundaries).

- Treat environment variables and the current working directory of the running process as read-only.
- Restrict writing to the file-system to unique temporary files. Isabelle already provides a temporary directory that is unique for the running process, and there is a centralized source of unique serial numbers in Isabelle/ML. Thus temporary files that are passed to to some external process will be always disjoint, and thus thread-safe.

ML Reference

```
File.tmp_path: Path.T -> Path.T
serial_string: unit -> string
```

`File.tmp_path` *path* relocates the base component of *path* into the unique temporary directory of the running Isabelle/ML process.

`serial_string` () creates a new serial number that is unique over the runtime of the Isabelle/ML process.

ML Examples

The following example shows how to create unique temporary file names.

```
ML_val ‹
  val tmp1 = File.tmp_path (Path.basic ("foo" ^ serial_string ()));
  val tmp2 = File.tmp_path (Path.basic ("foo" ^ serial_string ()));
  assert (tmp1 <> tmp2);
›
```

0.8.3 Explicit synchronization

Isabelle/ML provides explicit synchronization for mutable variables over immutable data, which may be updated atomically and exclusively. This addresses the rare situations where mutable shared resources are really required.

Synchronization in Isabelle/ML is based on primitives of Poly/ML, which have been adapted to the specific assumptions of the concurrent Isabelle environment. User code should not break this abstraction, but stay within the confines of concurrent Isabelle/ML.

A *synchronized variable* is an explicit state component associated with mechanisms for locking and signaling. There are operations to await a condition, change the state, and signal the change to all other waiting threads. Synchronized access to the state variable is *not* re-entrant: direct or indirect nesting within the same thread will cause a deadlock!

ML Reference

```
type 'a Synchronized.var
Synchronized.var: string -> 'a -> 'a Synchronized.var
Synchronized.guarded_access: 'a Synchronized.var ->
  ('a -> ('b * 'a) option) -> 'b
```

Type `'a Synchronized.var` represents synchronized variables with state of type `'a`.

`Synchronized.var` *name* x creates a synchronized variable that is initialized with value x. The *name* is used for tracing.

`Synchronized.guarded_access` *var* f lets the function f operate within a critical section on the state x as follows: if f x produces `NONE`, it continues to wait on the internal condition variable, expecting that some other thread will eventually change the content in a suitable manner; if f x produces `SOME` (y, x') it is satisfied and assigns the new state value x', broadcasts a signal to all waiting threads on the associated condition variable, and returns the result y.

There are some further variants of the `Synchronized.guarded_access` combinator, see `~~/src/Pure/Concurrent/synchronized.ML` for details.

ML Examples

The following example implements a counter that produces positive integers that are unique over the runtime of the Isabelle process:

ML_val ‹

```
local
  val counter = Synchronized.var "counter" 0;
in
  fun next () =
    Synchronized.guarded_access counter
      (fn i =>
        let val j = i + 1
        in SOME (j, j) end);
end;

val a = next ();
val b = next ();
assert (a <> b);
>
```

See ~~/src/Pure/Concurrent/mailbox.ML how to implement a mailbox as synchronized variable over a purely functional list.

0.9 Managed evaluation

Execution of Standard ML follows the model of strict functional evaluation with optional exceptions. Evaluation happens whenever some function is applied to (sufficiently many) arguments. The result is either an explicit value or an implicit exception.

Managed evaluation in Isabelle/ML organizes expressions and results to control certain physical side-conditions, to say more specifically when and how evaluation happens. For example, the Isabelle/ML library supports lazy evaluation with memoing, parallel evaluation via futures, asynchronous evaluation via promises, evaluation with time limit etc.

An *unevaluated expression* is represented either as unit abstraction `fn () => a` of type `unit -> 'a` or as regular function `fn a => b` of type `'a -> 'b`. Both forms occur routinely, and special care is required to tell them apart — the static type-system of SML is only of limited help here.

The first form is more intuitive: some combinator `(unit -> 'a) -> 'a` applies the given function to `()` to initiate the postponed evaluation process. The second form is more flexible: some combinator `('a -> 'b) -> 'a -> 'b` acts like a modified form of function application; several such combinators may be cascaded to modify a given function, before it is ultimately applied to some argument.

Reified results make the disjoint sum of regular values versions exceptional situations explicit as ML datatype: *'a result = Res of 'a | Exn of exn*. This is typically used for administrative purposes, to store the overall outcome of an evaluation process.

Parallel exceptions aggregate reified results, such that multiple exceptions are digested as a collection in canonical form that identifies exceptions according to their original occurrence. This is particular important for parallel evaluation via futures §0.9.3, which are organized as acyclic graph of evaluations that depend on other evaluations: exceptions stemming from shared subgraphs are exposed exactly once and in the order of their original occurrence (e.g. when printed at the toplevel). Interrupt counts as neutral element here: it is treated as minimal information about some canceled evaluation process, and is absorbed by the presence of regular program exceptions.

ML Reference

```
type 'a Exn.result
Exn.capture: ('a -> 'b) -> 'a -> 'b Exn.result
Exn.interruptible_capture: ('a -> 'b) -> 'a -> 'b Exn.result
Exn.release: 'a Exn.result -> 'a
Par_Exn.release_all: 'a Exn.result list -> 'a list
Par_Exn.release_first: 'a Exn.result list -> 'a list
```

Type `'a Exn.result` represents the disjoint sum of ML results explicitly, with constructor `Exn.Res` for regular values and `Exn.Exn` for exceptions.

`Exn.capture` *f x* manages the evaluation of *f x* such that exceptions are made explicit as `Exn.Exn`. Note that this includes physical interrupts (see also §0.5), so the same precautions apply to user code: interrupts must not be absorbed accidentally!

`Exn.interruptible_capture` is similar to `Exn.capture`, but interrupts are immediately re-raised as required for user code.

`Exn.release` *result* releases the original runtime result, exposing its regular value or raising the reified exception.

`Par_Exn.release_all` *results* combines results that were produced independently (e.g. by parallel evaluation). If all results are regular values, that list is returned. Otherwise, the collection of all exceptions is raised, wrapped-up as collective parallel exception. Note that the latter prevents access to individual exceptions by conventional `handle` of ML.

`Par_Exn.release_first` is similar to `Par_Exn.release_all`, but only the first (meaningful) exception that has occurred in the original evaluation process is raised again, the others are ignored. That single exception may get handled by conventional means in ML.

0.9.1 Parallel skeletons

Algorithmic skeletons are combinators that operate on lists in parallel, in the manner of well-known *map*, *exists*, *forall* etc. Management of futures (§0.9.3) and their results as reified exceptions is wrapped up into simple programming interfaces that resemble the sequential versions.

What remains is the application-specific problem to present expressions with suitable *granularity*: each list element corresponds to one evaluation task. If the granularity is too coarse, the available CPUs are not saturated. If it is too fine-grained, CPU cycles are wasted due to the overhead of organizing parallel processing. In the worst case, parallel performance will be less than the sequential counterpart!

ML Reference

```
Par_List.map: ('a -> 'b) -> 'a list -> 'b list
Par_List.get_some: ('a -> 'b option) -> 'a list -> 'b option
```

`Par_List.map` f $[x_1, \ldots, x_n]$ is like `map` f $[x_1, \ldots, x_n]$, but the evaluation of $f\ x_i$ for $i = 1, \ldots, n$ is performed in parallel.

An exception in any $f\ x_i$ cancels the overall evaluation process. The final result is produced via `Par_Exn.release_first` as explained above, which means the first program exception that happened to occur in the parallel evaluation is propagated, and all other failures are ignored.

`Par_List.get_some` f $[x_1, \ldots, x_n]$ produces some $f\ x_i$ that is of the form *SOME* y_i, if that exists, otherwise *NONE*. Thus it is similar to `Library.get_first`, but subject to a non-deterministic parallel choice process. The first successful result cancels the overall evaluation process; other exceptions are propagated as for `Par_List.map`.

This generic parallel choice combinator is the basis for derived forms, such as `Par_List.find_some`, `Par_List.exists`, `Par_List.forall`.

ML Examples

Subsequently, the Ackermann function is evaluated in parallel for some ranges of arguments.

ML_val ‹

```
  fun ackermann 0 n = n + 1
    | ackermann m 0 = ackermann (m - 1) 1
    | ackermann m n = ackermann (m - 1) (ackermann m (n - 1));

  Par_List.map (ackermann 2) (500 upto 1000);
  Par_List.map (ackermann 3) (5 upto 10);
```

›

0.9.2 Lazy evaluation

Classic lazy evaluation works via the *lazy* / *force* pair of operations: *lazy* to wrap an unevaluated expression, and *force* to evaluate it once and store its result persistently. Later invocations of *force* retrieve the stored result without another evaluation. Isabelle/ML refines this idea to accommodate the aspects of multi-threading, synchronous program exceptions and asynchronous interrupts.

The first thread that invokes *force* on an unfinished lazy value changes its state into a *promise* of the eventual result and starts evaluating it. Any other threads that *force* the same lazy value in the meantime need to wait for it to finish, by producing a regular result or program exception. If the evaluation attempt is interrupted, this event is propagated to all waiting threads and the lazy value is reset to its original state.

This means a lazy value is completely evaluated at most once, in a thread-safe manner. There might be multiple interrupted evaluation attempts, and multiple receivers of intermediate interrupt events. Interrupts are *not* made persistent: later evaluation attempts start again from the original expression.

ML Reference

```
type 'a lazy
Lazy.lazy: (unit -> 'a) -> 'a lazy
Lazy.value: 'a -> 'a lazy
Lazy.force: 'a lazy -> 'a
```

Type `'a lazy` represents lazy values over type `'a`.

`Lazy.lazy` (*fn* () => *e*) wraps the unevaluated expression *e* as unfinished lazy value.

`Lazy.value` *a* wraps the value *a* as finished lazy value. When forced, it returns *a* without any further evaluation.

There is very low overhead for this proforma wrapping of strict values as lazy values.

`Lazy.force` *x* produces the result of the lazy value in a thread-safe manner as explained above. Thus it may cause the current thread to wait on a pending evaluation attempt by another thread.

0.9.3 Futures

Futures help to organize parallel execution in a value-oriented manner, with *fork* / *join* as the main pair of operations, and some further variants; see also [21, 22]. Unlike lazy values, futures are evaluated strictly and spontaneously on separate worker threads. Futures may be canceled, which leads to interrupts on running evaluation attempts, and forces structurally related futures to fail for all time; already finished futures remain unchanged. Exceptions between related futures are propagated as well, and turned into parallel exceptions (see above).

Technically, a future is a single-assignment variable together with a *task* that serves administrative purposes, notably within the *task queue* where new futures are registered for eventual evaluation and the worker threads retrieve their work.

The pool of worker threads is limited, in correlation with the number of physical cores on the machine. Note that allocation of runtime resources may be distorted either if workers yield CPU time (e.g. via system sleep or wait operations), or if non-worker threads contend for significant runtime resources independently. There is a limited number of replacement worker threads that get activated in certain explicit wait conditions, after a timeout.

Each future task belongs to some *task group*, which represents the hierarchic structure of related tasks, together with the exception status a that point. By default, the task group of a newly created future is a new sub-group of the presently running one, but it is also possible to indicate different group layouts under program control.

Cancellation of futures actually refers to the corresponding task group and all its sub-groups. Thus interrupts are propagated down the group hierarchy.

Regular program exceptions are treated likewise: failure of the evaluation of some future task affects its own group and all sub-groups. Given a particular task group, its *group status* cumulates all relevant exceptions according to its position within the group hierarchy. Interrupted tasks that lack regular result information, will pick up parallel exceptions from the cumulative group status.

A *passive future* or *promise* is a future with slightly different evaluation policies: there is only a single-assignment variable and some expression to evaluate for the *failed* case (e.g. to clean up resources when canceled). A regular result is produced by external means, using a separate *fulfill* operation.

Promises are managed in the same task queue, so regular futures may depend on them. This allows a form of reactive programming, where some promises are used as minimal elements (or guards) within the future dependency graph: when these promises are fulfilled the evaluation of subsequent futures starts spontaneously, according to their own inter-dependencies.

ML Reference

```
type 'a future
Future.fork: (unit -> 'a) -> 'a future
Future.forks: Future.params -> (unit -> 'a) list -> 'a future list
Future.join: 'a future -> 'a
Future.joins: 'a future list -> 'a list
Future.value: 'a -> 'a future
Future.map: ('a -> 'b) -> 'a future -> 'b future
Future.cancel: 'a future -> unit
Future.cancel_group: Future.group -> unit
Future.promise: (unit -> unit) -> 'a future
Future.fulfill: 'a future -> 'a -> unit
```

Type `'a future` represents future values over type `'a`.

`Future.fork` (*fn* () => *e*) registers the unevaluated expression *e* as unfinished future value, to be evaluated eventually on the parallel worker-thread farm. This is a shorthand for `Future.forks` below, with default parameters and a single expression.

`Future.forks` *params exprs* is the general interface to fork several futures simultaneously. The *params* consist of the following fields:

- *name* : *string* (default `""`) specifies a common name for the tasks of the forked futures, which serves diagnostic purposes.

- *group* : *Future.group option* (default `NONE`) specifies an optional task group for the forked futures. `NONE` means that a new subgroup of the current worker-thread task context is created. If this is not a worker thread, the group will be a new root in the group hierarchy.
- *deps* : *Future.task list* (default `[]`) specifies dependencies on other future tasks, i.e. the adjacency relation in the global task queue. Dependencies on already finished tasks are ignored.
- *pri* : *int* (default `0`) specifies a priority within the task queue.

  Typically there is only little deviation from the default priority `0`. As a rule of thumb, `~1` means "low priority" and `1` means "high priority".

  Note that the task priority only affects the position in the queue, not the thread priority. When a worker thread picks up a task for processing, it runs with the normal thread priority to the end (or until canceled). Higher priority tasks that are queued later need to wait until this (or another) worker thread becomes free again.
- *interrupts* : *bool* (default `true`) tells whether the worker thread that processes the corresponding task is initially put into interruptible state. This state may change again while running, by modifying the thread attributes.

  With interrupts disabled, a running future task cannot be canceled. It is the responsibility of the programmer that this special state is retained only briefly.

`Future.join` *x* retrieves the value of an already finished future, which may lead to an exception, according to the result of its previous evaluation.

For an unfinished future there are several cases depending on the role of the current thread and the status of the future. A non-worker thread waits passively until the future is eventually evaluated. A worker thread temporarily changes its task context and takes over the responsibility to evaluate the future expression on the spot. The latter is done in a thread-safe manner: other threads that intend to join the same future need to wait until the ongoing evaluation is finished.

Note that excessive use of dynamic dependencies of futures by adhoc joining may lead to bad utilization of CPU cores, due to threads waiting on other threads to finish required futures. The future task farm has a limited amount of replacement threads that continue working on unrelated tasks after some timeout.

Whenever possible, static dependencies of futures should be specified explicitly when forked (see *deps* above). Thus the evaluation can work from the bottom up, without join conflicts and wait states.

`Future.joins` *xs* joins the given list of futures simultaneously, which is more efficient than `map Future.join` *xs*.

Based on the dependency graph of tasks, the current thread takes over the responsibility to evaluate future expressions that are required for the main result, working from the bottom up. Waiting on future results that are presently evaluated on other threads only happens as last resort, when no other unfinished futures are left over.

`Future.value` *a* wraps the value *a* as finished future value, bypassing the worker-thread farm. When joined, it returns *a* without any further evaluation.

There is very low overhead for this proforma wrapping of strict values as futures.

`Future.map` *f x* is a fast-path implementation of `Future.fork` (*fn* () => *f* (`Future.join` *x*)), which avoids the full overhead of the task queue and worker-thread farm as far as possible. The function *f* is supposed to be some trivial post-processing or projection of the future result.

`Future.cancel` *x* cancels the task group of the given future, using `Future.cancel_group` below.

`Future.cancel_group` *group* cancels all tasks of the given task group for all time. Threads that are presently processing a task of the given group are interrupted: it may take some time until they are actually terminated. Tasks that are queued but not yet processed are dequeued and forced into interrupted state. Since the task group is itself invalidated, any further attempt to fork a future that belongs to it will yield a canceled result as well.

`Future.promise` *abort* registers a passive future with the given *abort* operation: it is invoked when the future task group is canceled.

`Future.fulfill` *x a* finishes the passive future *x* by the given value *a*. If the promise has already been canceled, the attempt to fulfill it causes an exception.

Chapter 1

Preliminaries

1.1 Contexts

A logical context represents the background that is required for formulating statements and composing proofs. It acts as a medium to produce formal content, depending on earlier material (declarations, results etc.).

For example, derivations within the Isabelle/Pure logic can be described as a judgment $\Gamma \vdash_\Theta \varphi$, which means that a proposition φ is derivable from hypotheses Γ within the theory Θ. There are logical reasons for keeping Θ and Γ separate: theories can be liberal about supporting type constructors and schematic polymorphism of constants and axioms, while the inner calculus of $\Gamma \vdash \varphi$ is strictly limited to Simple Type Theory (with fixed type variables in the assumptions).

Contexts and derivations are linked by the following key principles:

- Transfer: monotonicity of derivations admits results to be transferred into a *larger* context, i.e. $\Gamma \vdash_\Theta \varphi$ implies $\Gamma' \vdash_{\Theta'} \varphi$ for contexts $\Theta' \supseteq \Theta$ and $\Gamma' \supseteq \Gamma$.

- Export: discharge of hypotheses admits results to be exported into a *smaller* context, i.e. $\Gamma' \vdash_\Theta \varphi$ implies $\Gamma \vdash_\Theta \Delta \Longrightarrow \varphi$ where $\Gamma' \supseteq \Gamma$ and $\Delta = \Gamma' - \Gamma$. Note that Θ remains unchanged here, only the Γ part is affected.

By modeling the main characteristics of the primitive Θ and Γ above, and abstracting over any particular logical content, we arrive at the fundamental notions of *theory context* and *proof context* in Isabelle/Isar. These implement a certain policy to manage arbitrary *context data*. There is a strongly-typed mechanism to declare new kinds of data at compile time.

The internal bootstrap process of Isabelle/Pure eventually reaches a stage where certain data slots provide the logical content of Θ and Γ sketched above, but this does not stop there! Various additional data slots support all kinds of mechanisms that are not necessarily part of the core logic.

For example, there would be data for canonical introduction and elimination rules for arbitrary operators (depending on the object-logic and application), which enables users to perform standard proof steps implicitly (cf. the *rule* method [19]).

Thus Isabelle/Isar is able to bring forth more and more concepts successively. In particular, an object-logic like Isabelle/HOL continues the Isabelle/Pure setup by adding specific components for automated reasoning (classical reasoner, tableau prover, structured induction etc.) and derived specification mechanisms (inductive predicates, recursive functions etc.). All of this is ultimately based on the generic data management by theory and proof contexts introduced here.

1.1.1 Theory context

A *theory* is a data container with explicit name and unique identifier. Theories are related by a (nominal) sub-theory relation, which corresponds to the dependency graph of the original construction; each theory is derived from a certain sub-graph of ancestor theories. To this end, the system maintains a set of symbolic "identification stamps" within each theory.

The *begin* operation starts a new theory by importing several parent theories (with merged contents) and entering a special mode of nameless incremental updates, until the final *end* operation is performed.

The example in figure 1.1 below shows a theory graph derived from *Pure*, with theory *Length* importing *Nat* and *List*. The body of *Length* consists of a sequence of updates, resulting in locally a linear sub-theory relation for each intermediate step.

Derived formal entities may retain a reference to the background theory in order to indicate the formal context from which they were produced. This provides an immutable certificate of the background theory.

ML Reference

```
type theory
Context.eq_thy: theory * theory -> bool
Context.subthy: theory * theory -> bool
Theory.begin_theory: string * Position.T -> theory list -> theory
Theory.parents_of: theory -> theory list
Theory.ancestors_of: theory -> theory list
```

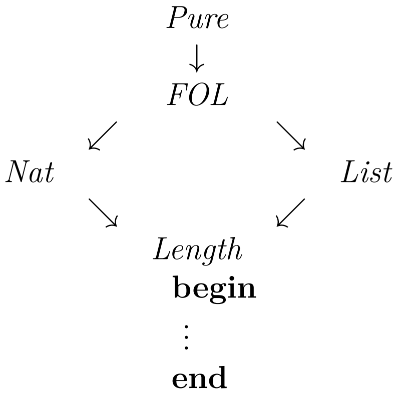


Figure 1.1: A theory definition depending on ancestors

Type `theory` represents theory contexts.

`Context.eq_thy` (*thy*$_1$, *thy*$_2$) check strict identity of two theories.

`Context.subthy` (*thy*$_1$, *thy*$_2$) compares theories according to the intrinsic graph structure of the construction. This sub-theory relation is a nominal approximation of inclusion ($\subseteq$) of the corresponding content (according to the semantics of the ML modules that implement the data).

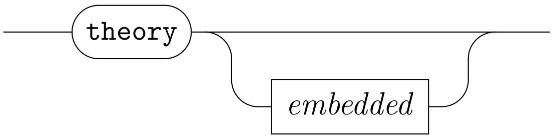
`Theory.begin_theory` *name parents* constructs a new theory based on the given parents. This ML function is normally not invoked directly.

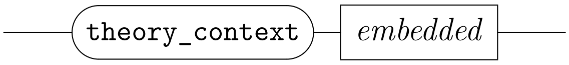
`Theory.parents_of` *thy* returns the direct ancestors of *thy*.

`Theory.ancestors_of` *thy* returns all ancestors of *thy* (not including *thy* itself).

ML Antiquotations

theory : *ML_antiquotation*
theory_context : *ML_antiquotation*





@{*theory*} refers to the background theory of the current context — as abstract value.

@{*theory A*} refers to an explicitly named ancestor theory *A* of the background theory of the current context — as abstract value.

@{*theory_context A*} is similar to @{*theory A*}, but presents the result as initial `Proof.context` (see also `Proof_Context.init_global`).

1.1.2 Proof context

A proof context is a container for pure data that refers to the theory from which it is derived. The *init* operation creates a proof context from a given theory. There is an explicit *transfer* operation to force resynchronization with updates to the background theory – this is rarely required in practice.

Entities derived in a proof context need to record logical requirements explicitly, since there is no separate context identification or symbolic inclusion as for theories. For example, hypotheses used in primitive derivations (cf. §2.3) are recorded separately within the sequent $\Gamma \vdash \varphi$, just to make double sure. Results could still leak into an alien proof context due to programming errors, but Isabelle/Isar includes some extra validity checks in critical positions, notably at the end of a sub-proof.

Proof contexts may be manipulated arbitrarily, although the common discipline is to follow block structure as a mental model: a given context is extended consecutively, and results are exported back into the original context. Note that an Isar proof state models block-structured reasoning explicitly, using a stack of proof contexts internally. For various technical reasons, the background theory of an Isar proof state must not be changed while the proof is still under construction!

ML Reference

```
type Proof.context
Proof_Context.init_global: theory -> Proof.context
Proof_Context.theory_of: Proof.context -> theory
Proof_Context.transfer: theory -> Proof.context -> Proof.context
```

Type `Proof.context` represents proof contexts.

`Proof_Context.init_global` *thy* produces a proof context derived from *thy*, initializing all data.

`Proof_Context.theory_of` *ctxt* selects the background theory from *ctxt*.

`Proof_Context.transfer` *thy ctxt* promotes the background theory of *ctxt* to the super theory *thy*.

ML Antiquotations

context : *ML_antiquotation*

@{*context*} refers to *the* context at compile-time — as abstract value. Independently of (local) theory or proof mode, this always produces a meaningful result.

This is probably the most common antiquotation in interactive experimentation with ML inside Isar.

1.1.3 Generic contexts

A generic context is the disjoint sum of either a theory or proof context. Occasionally, this enables uniform treatment of generic context data, typically extra-logical information. Operations on generic contexts include the usual injections, partial selections, and combinators for lifting operations on either component of the disjoint sum.

Moreover, there are total operations *theory_of* and *proof_of* to convert a generic context into either kind: a theory can always be selected from the sum, while a proof context might have to be constructed by an ad-hoc *init* operation, which incurs a small runtime overhead.

ML Reference

```
type Context.generic
Context.theory_of: Context.generic -> theory
Context.proof_of: Context.generic -> Proof.context
```

Type `Context.generic` is the direct sum of `theory` and `Proof.context`, with the datatype constructors `Context.Theory` and `Context.Proof`.

`Context.theory_of` *context* always produces a theory from the generic *context*, using `Proof_Context.theory_of` as required.

`Context.proof_of` *context* always produces a proof context from the generic *context*, using `Proof_Context.init_global` as required (note that this re-initializes the context data with each invocation).

1.1.4 Context data

The main purpose of theory and proof contexts is to manage arbitrary (pure) data. New data types can be declared incrementally at compile time. There are separate declaration mechanisms for any of the three kinds of contexts: theory, proof, generic.

Theory data declarations need to implement the following ML signature:

type T	representing type
val $empty$: T	empty default value
val $extend$: $T \rightarrow T$	obsolete (identity function)
val $merge$: $T \times T \rightarrow T$	merge data

The *empty* value acts as initial default for *any* theory that does not declare actual data content; *extend* is obsolete: it needs to be the identity function.

The *merge* operation needs to join the data from two theories in a conservative manner. The standard scheme for $merge\ (data_1,\ data_2)$ inserts those parts of $data_2$ into $data_1$ that are not yet present, while keeping the general order of things. The `Library.merge` function on plain lists may serve as canonical template. Particularly note that shared parts of the data must not be duplicated by naive concatenation, or a theory graph that resembles a chain of diamonds would cause an exponential blowup!

Sometimes, the data consists of a single item that cannot be "merged" in a sensible manner. Then the standard scheme degenerates to the projection to $data_1$, ignoring $data_2$ outright.

Proof context data declarations need to implement the following ML signature:

type T	representing type
val $init$: $theory \to T$	produce initial value

The *init* operation is supposed to produce a pure value from the given background theory and should be somehow "immediate". Whenever a proof context is initialized, which happens frequently, the the system invokes the *init* operation of *all* theory data slots ever declared. This also means that one needs to be economic about the total number of proof data declarations in the system, i.e. each ML module should declare at most one, sometimes two data slots for its internal use. Repeated data declarations to simulate a record type should be avoided!

Generic data provides a hybrid interface for both theory and proof data. The *init* operation for proof contexts is predefined to select the current data value from the background theory.

Any of the above data declarations over type T result in an ML structure with the following signature:

get: $context \to T$
put: $T \to context \to context$
map: $(T \to T) \to context \to context$

These other operations provide exclusive access for the particular kind of context (theory, proof, or generic context). This interface observes the ML discipline for types and scopes: there is no other way to access the corresponding data slot of a context. By keeping these operations private, an Isabelle/ML module may maintain abstract values authentically.

ML Reference

```
functor Theory_Data
functor Proof_Data
functor Generic_Data
```

`Theory_Data`(*spec*) declares data for type `theory` according to the specification provided as argument structure. The resulting structure provides data init and access operations as described above.

`Proof_Data`(*spec*) is analogous to `Theory_Data` for type `Proof.context`.

`Generic_Data`(*spec*) is analogous to `Theory_Data` for type `Context.generic`.

ML Examples

The following artificial example demonstrates theory data: we maintain a set of terms that are supposed to be wellformed wrt. the enclosing theory. The public interface is as follows:

ML ‹

```
  signature WELLFORMED_TERMS =
  sig
    val get: theory -> term list
    val add: term -> theory -> theory
  end;
```

›

The implementation uses private theory data internally, and only exposes an operation that involves explicit argument checking wrt. the given theory.

ML ‹

```
  structure Wellformed_Terms: WELLFORMED_TERMS =
  struct

  structure Terms = Theory_Data
  (
    type T = term Ord_List.T;
    val empty = [];
    fun merge (ts1, ts2) =
      Ord_List.union Term_Ord.fast_term_ord ts1 ts2;
  );

  val get = Terms.get;

  fun add raw_t thy =
    let
      val t = Sign.cert_term thy raw_t;
    in
      Terms.map (Ord_List.insert Term_Ord.fast_term_ord t) thy
    end;

  end;
```

›

Type `term Ord_List.T` is used for reasonably efficient representation of a set of terms: all operations are linear in the number of stored elements. Here we assume that users of this module do not care about the declaration order, since that data structure forces its own arrangement of elements.

Observe how the `merge` operation joins the data slots of the two constituents: `Ord_List.union` prevents duplication of common data from different branches, thus avoiding the danger of exponential blowup. Plain list append etc. must never be used for theory data merges!

Our intended invariant is achieved as follows:

1. `Wellformed_Terms.add` only admits terms that have passed the `Sign.cert_term` check of the given theory at that point.
2. Wellformedness in the sense of `Sign.cert_term` is monotonic wrt. the sub-theory relation. So our data can move upwards in the hierarchy (via extension or merges), and maintain wellformedness without further checks.

Note that all basic operations of the inference kernel (which includes `Sign.cert_term`) observe this monotonicity principle, but other user-space tools don't. For example, fully-featured type-inference via `Syntax.check_term` (cf. §3.3) is not necessarily monotonic wrt. the background theory, since constraints of term constants can be modified by later declarations, for example.

In most cases, user-space context data does not have to take such invariants too seriously. The situation is different in the implementation of the inference kernel itself, which uses the very same data mechanisms for types, constants, axioms etc.

1.1.5 Configuration options

A *configuration option* is a named optional value of some basic type (Boolean, integer, string) that is stored in the context. It is a simple application of general context data (§1.1.4) that is sufficiently common to justify customized setup, which includes some concrete declarations for end-users using existing notation for attributes (cf. §7.3).

For example, the predefined configuration option *show_types* controls output of explicit type constraints for variables in printed terms (cf. §3.1). Its value can be modified within Isar text like this:

experiment
begin

declare [[*show_types* = *false*]]
— declaration within (local) theory context

notepad
begin
  note [[*show_types* = *true*]]
    — declaration within proof (forward mode)
  term x

  have $x = x$
    using [[*show_types* = *false*]]
      — declaration within proof (backward mode)
    ..
end

end

Configuration options that are not set explicitly hold a default value that can depend on the application context. This allows to retrieve the value from another slot within the context, or fall back on a global preference mechanism, for example.

The operations to declare configuration options and get/map their values are modeled as direct replacements for historic global references, only that the context is made explicit. This allows easy configuration of tools, without relying on the execution order as required for old-style mutable references.

ML Reference

```
Config.get: Proof.context -> 'a Config.T -> 'a
Config.map: 'a Config.T -> ('a -> 'a) -> Proof.context -> Proof.context
Attrib.setup_config_bool: binding -> (Context.generic -> bool) ->
  bool Config.T
Attrib.setup_config_int: binding -> (Context.generic -> int) ->
  int Config.T
Attrib.setup_config_real: binding -> (Context.generic -> real) ->
  real Config.T
Attrib.setup_config_string: binding -> (Context.generic -> string) ->
  string Config.T
```

`Config.get` *ctxt config* gets the value of *config* in the given context.

`Config.map` *config f ctxt* updates the context by updating the value of *config*.

config = `Attrib.setup_config_bool` *name default* creates a named configuration option of type `bool`, with the given *default* depending on the application context. The resulting *config* can be used to get/map its value in a given context. There is an implicit update of the background theory that registers the option as attribute with some concrete syntax.

`Attrib.config_int`, `Attrib.config_real`, and `Attrib.config_string` work like `Attrib.config_bool`, but for types `int` and `string`, respectively.

ML Examples

The following example shows how to declare and use a Boolean configuration option called *my_flag* with constant default value `false`.

```
ML ‹
  val my_flag =
    Attrib.setup_config_bool binding‹my_flag› (K false)
›
```

Now the user can refer to *my_flag* in declarations, while ML tools can retrieve the current value from the context via `Config.get`.

```
ML_val ‹assert (Config.get context my_flag = false)›

declare [[my_flag = true]]

ML_val ‹assert (Config.get context my_flag = true)›

notepad
begin
  {
    note [[my_flag = false]]
    ML_val ‹assert (Config.get context my_flag = false)›
  }
  ML_val ‹assert (Config.get context my_flag = true)›
end
```

Here is another example involving ML type `real` (floating-point numbers).

```
ML ‹
  val airspeed_velocity =
    Attrib.setup_config_real binding‹airspeed_velocity› (K 0.0)
›

declare [[airspeed_velocity = 10]]
declare [[airspeed_velocity = 9.9]]
```

1.2 Names

In principle, a name is just a string, but there are various conventions for representing additional structure. For example, "*Foo.bar.baz*" is considered as a long name consisting of qualifier *Foo.bar* and base name *baz*. The individual constituents of a name may have further substructure, e.g. the string "`\<alpha>`" encodes as a single symbol (§0.6).

Subsequently, we shall introduce specific categories of names. Roughly speaking these correspond to logical entities as follows:

- Basic names (§1.2.1): free and bound variables.
- Indexed names (§1.2.2): schematic variables.
- Long names (§1.2.3): constants of any kind (type constructors, term constants, other concepts defined in user space). Such entities are typically managed via name spaces (§1.2.4).

1.2.1 Basic names

A *basic name* essentially consists of a single Isabelle identifier. There are conventions to mark separate classes of basic names, by attaching a suffix of underscores: one underscore means *internal name*, two underscores means *Skolem name*, three underscores means *internal Skolem name*.

For example, the basic name *foo* has the internal version *foo*_, with Skolem versions *foo*__ and *foo*___, respectively.

These special versions provide copies of the basic name space, apart from anything that normally appears in the user text. For example, system generated variables in Isar proof contexts are usually marked as internal, which prevents mysterious names like *xaa* to appear in human-readable text.

Manipulating binding scopes often requires on-the-fly renamings. A *name context* contains a collection of already used names. The *declare* operation adds names to the context.

The *invents* operation derives a number of fresh names from a given starting point. For example, the first three names derived from *a* are *a*, *b*, *c*.

The *variants* operation produces fresh names by incrementing tentative names as base-26 numbers (with digits *a..z*) until all clashes are resolved. For example, name *foo* results in variants *fooa*, *foob*, *fooc*, ..., *fooaa*, *fooab* etc.; each renaming step picks the next unused variant from this sequence.

ML Reference

```
Name.internal: string -> string
Name.skolem: string -> string

type Name.context
Name.context: Name.context
Name.declare: string -> Name.context -> Name.context
Name.invent: Name.context -> string -> int -> string list
Name.variant: string -> Name.context -> string * Name.context

Variable.names_of: Proof.context -> Name.context
```

`Name.internal` *name* produces an internal name by adding one underscore.

`Name.skolem` *name* produces a Skolem name by adding two underscores.

Type `Name.context` represents the context of already used names; the initial value is `Name.context`.

`Name.declare` *name* enters a used name into the context.

`Name.invent` *context name n* produces *n* fresh names derived from *name*.

`Name.variant` *name context* produces a fresh variant of *name*; the result is declared to the context.

`Variable.names_of` *ctxt* retrieves the context of declared type and term variable names. Projecting a proof context down to a primitive name context is occasionally useful when invoking lower-level operations. Regular management of "fresh variables" is done by suitable operations of structure `Variable`, which is also able to provide an official status of "locally fixed variable" within the logical environment (cf. §6.1).

ML Examples

The following simple examples demonstrate how to produce fresh names from the initial `Name.context`.

ML_val ‹

```
  val list1 = Name.invent Name.context "a" 5;
  assert (list1 = ["a", "b", "c", "d", "e"]);

  val list2 =
    #1 (fold_map Name.variant ["x", "x", "a", "a", "'a", "'a"]
Name.context);
  assert (list2 = ["x", "xa", "a", "aa", "'a", "'aa"]);
>
```

The same works relatively to the formal context as follows.

experiment fixes *a b c* :: *'a*
begin

```
ML_val <
  val names = Variable.names_of context;

  val list1 = Name.invent names "a" 5;
  assert (list1 = ["d", "e", "f", "g", "h"]);

  val list2 =
    #1 (fold_map Name.variant ["x", "x", "a", "a", "'a", "'a"]
names);
  assert (list2 = ["x", "xa", "aa", "ab", "'aa", "'ab"]);
>
```

end

1.2.2 Indexed names

An *indexed name* (or *indexname*) is a pair of a basic name and a natural number. This representation allows efficient renaming by incrementing the second component only. The canonical way to rename two collections of indexnames apart from each other is this: determine the maximum index *maxidx* of the first collection, then increment all indexes of the second collection by *maxidx* + 1; the maximum index of an empty collection is −1.

Occasionally, basic names are injected into the same pair type of indexed names: then $(x, -1)$ is used to encode the basic name x.

Isabelle syntax observes the following rules for representing an indexname (x, i) as a packed string:

- *?x* if x does not end with a digit and $i = 0$,
- *?xi* if x does not end with a digit,
- *?x.i* otherwise.

Indexnames may acquire large index numbers after several maxidx shifts have been applied. Results are usually normalized towards 0 at certain checkpoints, notably at the end of a proof. This works by producing variants of the corresponding basic name components. For example, the collection $?x1$, $?x7$, $?x42$ becomes $?x$, $?xa$, $?xb$.

ML Reference

```
type indexname = string * int
```

Type `indexname` represents indexed names. This is an abbreviation for `string * int`. The second component is usually non-negative, except for situations where $(x, -1)$ is used to inject basic names into this type. Other negative indexes should not be used.

1.2.3 Long names

A *long name* consists of a sequence of non-empty name components. The packed representation uses a dot as separator, as in "$A.b.c$". The last component is called *base name*, the remaining prefix is called *qualifier* (which may be empty). The qualifier can be understood as the access path to the named entity while passing through some nested block-structure, although our free-form long names do not really enforce any strict discipline.

For example, an item named "$A.b.c$" may be understood as a local entity c, within a local structure b, within a global structure A. In practice, long names usually represent 1–3 levels of qualification. User ML code should not make any assumptions about the particular structure of long names!

The empty name is commonly used as an indication of unnamed entities, or entities that are not entered into the corresponding name space, whenever this makes any sense. The basic operations on long names map empty names again to empty names.

ML Reference

```
Long_Name.base_name: string -> string
Long_Name.qualifier: string -> string
Long_Name.append: string -> string -> string
Long_Name.implode: string list -> string
Long_Name.explode: string -> string list
```

`Long_Name.base_name` *name* returns the base name of a long name.

`Long_Name.qualifier` *name* returns the qualifier of a long name.

`Long_Name.append` $name_1$ $name_2$ appends two long names.

`Long_Name.implode` *names* and `Long_Name.explode` *name* convert between the packed string representation and the explicit list form of long names.

1.2.4 Name spaces

A *name space* manages a collection of long names, together with a mapping between partially qualified external names and fully qualified internal names (in both directions). Note that the corresponding *intern* and *extern* operations are mostly used for parsing and printing only! The *declare* operation augments a name space according to the accesses determined by a given binding, and a naming policy from the context.

A *binding* specifies details about the prospective long name of a newly introduced formal entity. It consists of a base name, prefixes for qualification (separate ones for system infrastructure and user-space mechanisms), a slot for the original source position, and some additional flags.

A *naming* provides some additional details for producing a long name from a binding. Normally, the naming is implicit in the theory or proof context. The *full* operation (and its variants for different context types) produces a fully qualified internal name to be entered into a name space. The main equation of this "chemical reaction" when binding new entities in a context is as follows:

$$\mathit{binding} + \mathit{naming} \longrightarrow \mathit{long\ name} + \mathit{name\ space\ accesses}$$

As a general principle, there is a separate name space for each kind of formal entity, e.g. fact, logical constant, type constructor, type class. It is usually clear from the occurrence in concrete syntax (or from the scope) which kind of entity a name refers to. For example, the very same name *c* may be used uniformly for a constant, type constructor, and type class.

There are common schemes to name derived entities systematically according to the name of the main logical entity involved, e.g. fact *c.intro* for a canonical introduction rule related to constant *c*. This technique of mapping names from one space into another requires some care in order to avoid conflicts.

In particular, theorem names derived from a type constructor or type class should get an additional suffix in addition to the usual qualification. This leads to the following conventions for derived names:

logical entity	fact name
constant *c*	*c.intro*
type *c*	*c_type.intro*
class *c*	*c_class.intro*

ML Reference

```
type binding
Binding.empty: binding
Binding.name: string -> binding
Binding.qualify: bool -> string -> binding -> binding
Binding.prefix: bool -> string -> binding -> binding
Binding.concealed: binding -> binding
Binding.print: binding -> string
```

```
type Name_Space.naming
Name_Space.global_naming: Name_Space.naming
Name_Space.add_path: string -> Name_Space.naming -> Name_Space.naming
Name_Space.full_name: Name_Space.naming -> binding -> string
```

```
type Name_Space.T
Name_Space.empty: string -> Name_Space.T
Name_Space.merge: Name_Space.T * Name_Space.T -> Name_Space.T
Name_Space.declare: Context.generic -> bool ->
  binding -> Name_Space.T -> string * Name_Space.T
Name_Space.intern: Name_Space.T -> string -> string
Name_Space.extern: Proof.context -> Name_Space.T -> string -> string
Name_Space.is_concealed: Name_Space.T -> string -> bool
```

Type `binding` represents the abstract concept of name bindings.

`Binding.empty` is the empty binding.

`Binding.name` *name* produces a binding with base name *name*. Note that this lacks proper source position information; see also the ML antiquotation *binding*.

`Binding.qualify` *mandatory name binding* prefixes qualifier *name* to *binding*. The *mandatory* flag tells if this name component always needs to be given in name space accesses — this is mostly *false* in practice. Note that this part of qualification is typically used in derived specification mechanisms.

`Binding.prefix` is similar to `Binding.qualify`, but affects the system prefix. This part of extra qualification is typically used in the infrastructure for modular specifications, notably "local theory targets" (see also chapter 8).

`Binding.concealed` *binding* indicates that the binding shall refer to an entity that serves foundational purposes only. This flag helps to mark implementation details of specification mechanism etc. Other tools should not depend on the particulars of concealed entities (cf. `Name_Space.is_concealed`).

`Binding.print` *binding* produces a string representation for human-readable output, together with some formal markup that might get used in GUI front-ends, for example.

Type `Name_Space.naming` represents the abstract concept of a naming policy.

`Name_Space.global_naming` is the default naming policy: it is global and lacks any path prefix. In a regular theory context this is augmented by a path prefix consisting of the theory name.

`Name_Space.add_path` *path naming* augments the naming policy by extending its path component.

`Name_Space.full_name` *naming binding* turns a name binding (usually a basic name) into the fully qualified internal name, according to the given naming policy.

Type `Name_Space.T` represents name spaces.

`Name_Space.empty` *kind* and `Name_Space.merge` (*space*$_1$, *space*$_2$) are the canonical operations for maintaining name spaces according to theory data management (§1.1.4); *kind* is a formal comment to characterize the purpose of a name space.

`Name_Space.declare` *context strict binding space* enters a name binding as fully qualified internal name into the name space, using the naming of the context.

`Name_Space.intern` *space name* internalizes a (partially qualified) external name.

This operation is mostly for parsing! Note that fully qualified names stemming from declarations are produced via `Name_Space.full_name` and `Name_Space.declare` (or their derivatives for `theory` and `Proof.context`).

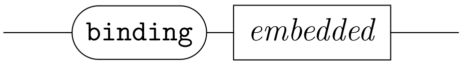
`Name_Space.extern` *ctxt space name* externalizes a (fully qualified) internal name.

This operation is mostly for printing! User code should not rely on the precise result too much.

`Name_Space.is_concealed` *space name* indicates whether *name* refers to a strictly private entity that other tools are supposed to ignore!

ML Antiquotations

binding : *ML_antiquotation*



@{*binding name*} produces a binding with base name *name* and the source position taken from the concrete syntax of this antiquotation. In many situations this is more appropriate than the more basic `Binding.name` function.

ML Examples

The following example yields the source position of some concrete binding inlined into the text:

ML_val ‹*Binding.pos_of* ***binding***‹*here*››

That position can be also printed in a message as follows:

ML_command

```
 ‹writeln
   ("Look here" ^ Position.here (Binding.pos_of binding‹here›))›
```

This illustrates a key virtue of formalized bindings as opposed to raw specifications of base names: the system can use this additional information for feedback given to the user (error messages etc.).

The following example refers to its source position directly, which is occasionally useful for experimentation and diagnostic purposes:

ML_command ‹*warning ("Look here" ^ Position.here* ***here***)›

Chapter 2

Primitive logic

The logical foundations of Isabelle/Isar are that of the Pure logic, which has been introduced as a Natural Deduction framework in [14]. This is essentially the same logic as "λHOL" in the more abstract setting of Pure Type Systems (PTS) [1], although there are some key differences in the specific treatment of simple types in Isabelle/Pure.

Following type-theoretic parlance, the Pure logic consists of three levels of λ-calculus with corresponding arrows, $\Rightarrow$ for syntactic function space (terms depending on terms), $\bigwedge$ for universal quantification (proofs depending on terms), and $\Longrightarrow$ for implication (proofs depending on proofs).

Derivations are relative to a logical theory, which declares type constructors, constants, and axioms. Theory declarations support schematic polymorphism, which is strictly speaking outside the logic.[1]

2.1 Types

The language of types is an uninterpreted order-sorted first-order algebra; types are qualified by ordered type classes.

A *type class* is an abstract syntactic entity declared in the theory context. The *subclass relation* $c_1 \subseteq c_2$ is specified by stating an acyclic generating relation; the transitive closure is maintained internally. The resulting relation is an ordering: reflexive, transitive, and antisymmetric.

A *sort* is a list of type classes written as $s = \{c_1, \ldots, c_m\}$, it represents symbolic intersection. Notationally, the curly braces are omitted for singleton intersections, i.e. any class c may be read as a sort $\{c\}$. The ordering on type classes is extended to sorts according to the meaning of intersections: $\{c_1, \ldots c_m\} \subseteq \{d_1, \ldots, d_n\}$ iff $\forall j.\ \exists i.\ c_i \subseteq d_j$. The empty intersection $\{\}$ refers to

[1] This is the deeper logical reason, why the theory context Θ is separate from the proof context Γ of the core calculus: type constructors, term constants, and facts (proof constants) may involve arbitrary type schemes, but the type of a locally fixed term parameter is also fixed!

the universal sort, which is the largest element wrt. the sort order. Thus $\{\}$ represents the "full sort", not the empty one! The intersection of all (finitely many) classes declared in the current theory is the least element wrt. the sort ordering.

A *fixed type variable* is a pair of a basic name (starting with a $'$ character) and a sort constraint, e.g. $('a, s)$ which is usually printed as α_s. A *schematic type variable* is a pair of an indexname and a sort constraint, e.g. $(('a, 0), s)$ which is usually printed as $?\alpha_s$.

Note that *all* syntactic components contribute to the identity of type variables: basic name, index, and sort constraint. The core logic handles type variables with the same name but different sorts as different, although the type-inference layer (which is outside the core) rejects anything like that.

A *type constructor* κ is a k-ary operator on types declared in the theory. Type constructor application is written postfix as $(\alpha_1, \ldots, \alpha_k)\kappa$. For $k = 0$ the argument tuple is omitted, e.g. $prop$ instead of $()prop$. For $k = 1$ the parentheses are omitted, e.g. α *list* instead of $(\alpha)list$. Further notation is provided for specific constructors, notably the right-associative infix $\alpha \Rightarrow \beta$ instead of $(\alpha, \beta)fun$.

The logical category *type* is defined inductively over type variables and type constructors as follows: $\tau = \alpha_s \mid ?\alpha_s \mid (\tau_1, \ldots, \tau_k)\kappa$.

A *type abbreviation* is a syntactic definition $(\vec{\alpha})\kappa = \tau$ of an arbitrary type expression τ over variables $\vec{\alpha}$. Type abbreviations appear as type constructors in the syntax, but are expanded before entering the logical core.

A *type arity* declares the image behavior of a type constructor wrt. the algebra of sorts: $\kappa :: (s_1, \ldots, s_k)s$ means that $(\tau_1, \ldots, \tau_k)\kappa$ is of sort s if every argument type τ_i is of sort s_i. Arity declarations are implicitly completed, i.e. $\kappa :: (\vec{s})c$ entails $\kappa :: (\vec{s})c'$ for any $c' \supseteq c$.

The sort algebra is always maintained as *coregular*, which means that type arities are consistent with the subclass relation: for any type constructor κ, and classes $c_1 \subseteq c_2$, and arities $\kappa :: (\vec{s}_1)c_1$ and $\kappa :: (\vec{s}_2)c_2$ holds $\vec{s}_1 \subseteq \vec{s}_2$ component-wise.

The key property of a coregular order-sorted algebra is that sort constraints can be solved in a most general fashion: for each type constructor κ and sort s there is a most general vector of argument sorts $(s_1, \ldots, s_k)$ such that a type scheme $(\alpha_{s_1}, \ldots, \alpha_{s_k})\kappa$ is of sort s. Consequently, type unification has most general solutions (modulo equivalence of sorts), so type-inference produces primary types as expected [12].

ML Reference

```
type class = string
type sort = class list
type arity = string * sort list * sort
type typ
Term.map_atyps: (typ -> typ) -> typ -> typ
Term.fold_atyps: (typ -> 'a -> 'a) -> typ -> 'a -> 'a
```

```
Sign.subsort: theory -> sort * sort -> bool
Sign.of_sort: theory -> typ * sort -> bool
Sign.add_type: Proof.context -> binding * int * mixfix -> theory -> theory
Sign.add_type_abbrev: Proof.context ->
  binding * string list * typ -> theory -> theory
Sign.primitive_class: binding * class list -> theory -> theory
Sign.primitive_classrel: class * class -> theory -> theory
Sign.primitive_arity: arity -> theory -> theory
```

Type `class` represents type classes.

Type `sort` represents sorts, i.e. finite intersections of classes. The empty list `[]: sort` refers to the empty class intersection, i.e. the "full sort".

Type `arity` represents type arities. A triple (κ, $\vec{s}$, s) : *arity* represents κ :: $(\vec{s})s$ as described above.

Type `typ` represents types; this is a datatype with constructors `TFree`, `TVar`, `Type`.

`Term.map_atyps` f τ applies the mapping f to all atomic types (`TFree`, `TVar`) occurring in τ.

`Term.fold_atyps` f τ iterates the operation f over all occurrences of atomic types (`TFree`, `TVar`) in τ; the type structure is traversed from left to right.

`Sign.subsort` *thy* (s_1, s_2) tests the subsort relation $s_1 \subseteq s_2$.

`Sign.of_sort` *thy* (τ, s) tests whether type τ is of sort s.

`Sign.add_type` *ctxt* (κ, k, *mx*) declares a new type constructors κ with k arguments and optional mixfix syntax.

`Sign.add_type_abbrev` *ctxt* (κ, $\vec{\alpha}$, τ) defines a new type abbreviation $(\vec{\alpha})\kappa = \tau$.

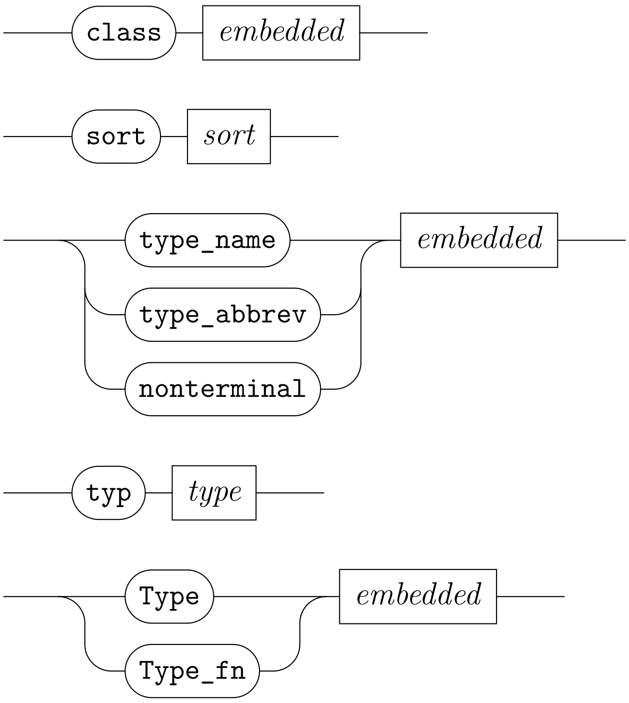
`Sign.primitive_class` (c, [c_1, …, c_n]) declares a new class c, together with class relations $c \subseteq c_i$, for $i = 1, \ldots, n$.

`Sign.primitive_classrel` (c_1, c_2) declares the class relation $c_1 \subseteq c_2$.

`Sign.primitive_arity` $(\kappa, \vec{s}, s)$ declares the arity $\kappa :: (\vec{s})s$.

ML Antiquotations

class	:	*ML_antiquotation*
sort	:	*ML_antiquotation*
type_name	:	*ML_antiquotation*
type_abbrev	:	*ML_antiquotation*
nonterminal	:	*ML_antiquotation*
typ	:	*ML_antiquotation*
Type	:	*ML_antiquotation*
Type_fn	:	*ML_antiquotation*



@{*class c*} inlines the internalized class c — as `string` literal.

@{*sort s*} inlines the internalized sort s — as `string list` literal.

@{*type_name c*} inlines the internalized type constructor *c* — as `string` literal.

@{*type_abbrev c*} inlines the internalized type abbreviation *c* — as `string` literal.

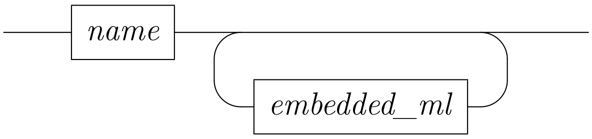
@{*nonterminal c*} inlines the internalized syntactic type / grammar nonterminal *c* — as `string` literal.

@{*typ* τ} inlines the internalized type τ — as constructor term for datatype `typ`.

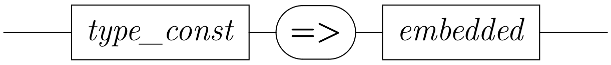
@{*Type source*} refers to embedded source text to produce a type constructor with name (formally checked) and arguments (inlined ML text). The embedded *source* follows the syntax category *type_const* defined below.

@{*Type_fn source*} is similar to @{*Type source*}, but produces a partial ML function that matches against a type constructor pattern, following syntax category *type_const_fn* below.

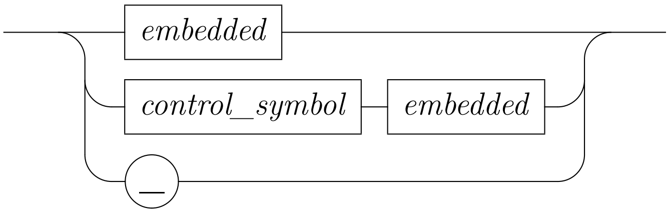
type_const



type_const_fn



embedded_ml



The text provided as *embedded_ml* is treated as regular Isabelle/ML source, possibly with nested antiquotations. ML text that only consists of a single antiquotation in compact control-cartouche notation, can be written without an extra level of nesting embedded text (see the last example below).

ML Examples

Here are some minimal examples for type constructor antiquotations.

```
ML ‹
  — type constructor without arguments:
  val natT = Type‹nat›;

  — type constructor with arguments:
  fun mk_funT (A, B) = Type‹fun A B›;

  — type constructor decomposition as partial function:
  val dest_funT = Type_fn‹fun A B => ‹(A, B)››;

  — nested type constructors:
  fun mk_relT A = Type‹fun A  Type‹fun A  Type‹bool›››;

  assert (mk_relT natT = typ‹nat ⇒ nat ⇒ bool›);
›
```

2.2 Terms

The language of terms is that of simply-typed λ-calculus with de-Bruijn indices for bound variables (cf. [6] or [15]), with the types being determined by the corresponding binders. In contrast, free variables and constants have an explicit name and type in each occurrence.

A *bound variable* is a natural number b, which accounts for the number of intermediate binders between the variable occurrence in the body and its binding position. For example, the de-Bruijn term $\lambda_{bool}.\ \lambda_{bool}.\ 1 \wedge 0$ would correspond to $\lambda x_{bool}.\ \lambda y_{bool}.\ x \wedge y$ in a named representation. Note that a bound variable may be represented by different de-Bruijn indices at different occurrences, depending on the nesting of abstractions.

A *loose variable* is a bound variable that is outside the scope of local binders. The types (and names) for loose variables can be managed as a separate context, that is maintained as a stack of hypothetical binders. The core logic operates on closed terms, without any loose variables.

A *fixed variable* is a pair of a basic name and a type, e.g. (x, τ) which is usually printed x_τ here. A *schematic variable* is a pair of an indexname and a type, e.g. $((x, 0), \tau)$ which is likewise printed as $?x_\tau$.

A *constant* is a pair of a basic name and a type, e.g. (c, τ) which is usually printed as c_τ here. Constants are declared in the context as polymorphic families $c :: \sigma$, meaning that all substitution instances c_τ for $\tau = \sigma\theta$ are valid.

The vector of *type arguments* of constant c_τ wrt. the declaration $c :: \sigma$ is defined as the codomain of the matcher $\theta = \{?\alpha_1 \mapsto \tau_1, \ldots, ?\alpha_n \mapsto \tau_n\}$ presented in canonical order $(\tau_1, \ldots, \tau_n)$, corresponding to the left-to-right occurrences of the α_i in σ. Within a given theory context, there is a one-to-one correspondence between any constant c_τ and the application $c(\tau_1, \ldots, \tau_n)$ of its type arguments. For example, with $plus :: \alpha \Rightarrow \alpha \Rightarrow \alpha$, the instance $plus_{nat \Rightarrow nat \Rightarrow nat}$ corresponds to $plus(nat)$.

Constant declarations $c :: \sigma$ may contain sort constraints for type variables in σ. These are observed by type-inference as expected, but *ignored* by the core logic. This means the primitive logic is able to reason with instances of polymorphic constants that the user-level type-checker would reject due to violation of type class restrictions.

An *atomic term* is either a variable or constant. The logical category *term* is defined inductively over atomic terms, with abstraction and application as follows: $t = b \mid x_\tau \mid ?x_\tau \mid c_\tau \mid \lambda_\tau.\ t \mid t_1\ t_2$. Parsing and printing takes care of converting between an external representation with named bound variables. Subsequently, we shall use the latter notation instead of internal de-Bruijn representation.

The inductive relation $t :: \tau$ assigns a (unique) type to a term according to the structure of atomic terms, abstractions, and applications:

$$\frac{}{a_\tau :: \tau} \qquad \frac{t :: \sigma}{(\lambda x_\tau.\ t) :: \tau \Rightarrow \sigma} \qquad \frac{t :: \tau \Rightarrow \sigma \quad u :: \tau}{t\ u :: \sigma}$$

A *well-typed term* is a term that can be typed according to these rules.

Typing information can be omitted: type-inference is able to reconstruct the most general type of a raw term, while assigning most general types to all of its variables and constants. Type-inference depends on a context of type constraints for fixed variables, and declarations for polymorphic constants.

The identity of atomic terms consists both of the name and the type component. This means that different variables x_{τ_1} and x_{τ_2} may become the same after type instantiation. Type-inference rejects variables of the same name, but different types. In contrast, mixed instances of polymorphic constants occur routinely.

The *hidden polymorphism* of a term $t :: \sigma$ is the set of type variables occurring in t, but not in its type σ. This means that the term implicitly depends on type arguments that are not accounted in the result type, i.e. there are different type instances $t\theta :: \sigma$ and $t\theta' :: \sigma$ with the same type. This slightly pathological situation notoriously demands additional care.

A *term abbreviation* is a syntactic definition $c_\sigma \equiv t$ of a closed term t of type σ, without any hidden polymorphism. A term abbreviation looks like a constant in the syntax, but is expanded before entering the logical core. Abbreviations are usually reverted when printing terms, using $t \rightarrow c_\sigma$ as rules for higher-order rewriting.

Canonical operations on λ-terms include $\alpha\beta\eta$-conversion: α-conversion refers to capture-free renaming of bound variables; β-conversion contracts an abstraction applied to an argument term, substituting the argument in the body: $(\lambda x.\ b)a$ becomes $b[a/x]$; η-conversion contracts vacuous application-abstraction: $\lambda x.\ f\ x$ becomes f, provided that the bound variable does not occur in f.

Terms are normally treated modulo α-conversion, which is implicit in the de-Bruijn representation. Names for bound variables in abstractions are maintained separately as (meaningless) comments, mostly for parsing and printing. Full $\alpha\beta\eta$-conversion is commonplace in various standard operations (§2.4) that are based on higher-order unification and matching.

ML Reference

```
type term
infix aconv: term * term -> bool
Term.map_types: (typ -> typ) -> term -> term
Term.fold_types: (typ -> 'a -> 'a) -> term -> 'a -> 'a
Term.map_aterms: (term -> term) -> term -> term
Term.fold_aterms: (term -> 'a -> 'a) -> term -> 'a -> 'a

fastype_of: term -> typ
lambda: term -> term -> term
betapply: term * term -> term
incr_boundvars: int -> term -> term
Sign.declare_const: Proof.context ->
  (binding * typ) * mixfix -> theory -> term * theory
Sign.add_abbrev: string -> binding * term ->
  theory -> (term * term) * theory
Sign.const_typargs: theory -> string * typ -> typ list
Sign.const_instance: theory -> string * typ list -> typ
```

Type `term` represents de-Bruijn terms, with comments in abstractions, and explicitly named free variables and constants; this is a datatype with constructors `Bound`, `Free`, `Var`, `Const`, `Abs`, `infix $`.

t `aconv` u checks α-equivalence of two terms. This is the basic equality relation on type `term`; raw datatype equality should only be used for operations related to parsing or printing!

`Term.map_types` f t applies the mapping f to all types occurring in t.

`Term.fold_types` f t iterates the operation f over all occurrences of types in t; the term structure is traversed from left to right.

`Term.map_aterms` f t applies the mapping f to all atomic terms (`Bound`, `Free`, `Var`, `Const`) occurring in t.

`Term.fold_aterms` f t iterates the operation f over all occurrences of atomic terms (`Bound`, `Free`, `Var`, `Const`) in t; the term structure is traversed from left to right.

`fastype_of` t determines the type of a well-typed term. This operation is relatively slow, despite the omission of any sanity checks.

`lambda` a b produces an abstraction $\lambda a.\ b$, where occurrences of the atomic term a in the body b are replaced by bound variables.

`betapply` (t, u) produces an application t u, with topmost β-conversion if t is an abstraction.

`incr_boundvars` j increments a term's dangling bound variables by the offset j. This is required when moving a subterm into a context where it is enclosed by a different number of abstractions. Bound variables with a matching abstraction are unaffected.

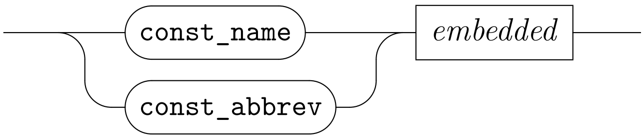
`Sign.declare_const` *ctxt* ((c, σ), *mx*) declares a new constant $c :: \sigma$ with optional mixfix syntax.

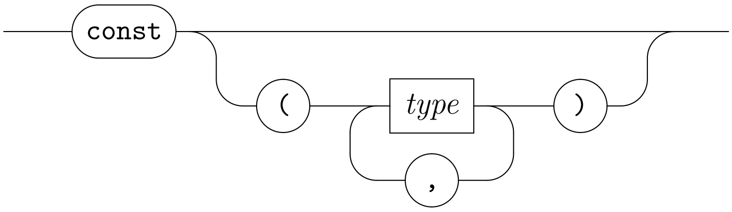
`Sign.add_abbrev` *print_mode* (c, t) introduces a new term abbreviation $c \equiv t$.

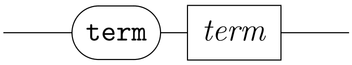
`Sign.const_typargs` *thy* (c, τ) and `Sign.const_instance` *thy* (c, [τ_1, $\ldots$, τ_n]) convert between two representations of polymorphic constants: full type instance vs. compact type arguments form.

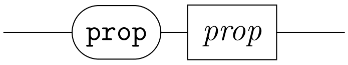
ML Antiquotations

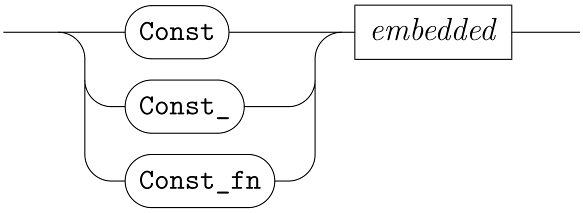
const_name	:	*ML_antiquotation*
const_abbrev	:	*ML_antiquotation*
const	:	*ML_antiquotation*
term	:	*ML_antiquotation*
prop	:	*ML_antiquotation*
Const	:	*ML_antiquotation*
*Const_*	:	*ML_antiquotation*
Const_fn	:	*ML_antiquotation*











@{*const_name c*} inlines the internalized logical constant name *c* — as `string` literal.

@{*const_abbrev c*} inlines the internalized abbreviated constant name *c* — as `string` literal.

@{*const* $c(\vec{\tau})$} inlines the internalized constant c with precise type instantiation in the sense of `Sign.const_instance` — as `Const` constructor term for datatype `term`.

@{*term* t} inlines the internalized term t — as constructor term for datatype `term`.

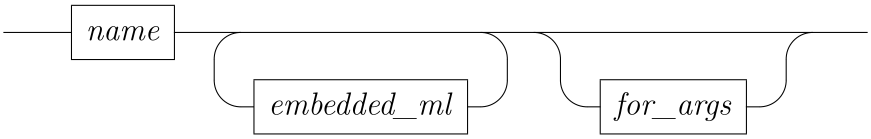
@{*prop* φ} inlines the internalized proposition φ — as constructor term for datatype `term`.

@{*Const source*} refers to embedded source text to produce a term constructor with name (formally checked), type arguments and term arguments (inlined ML text). The embedded *source* follows the syntax category *term_const* defined below, using *embedded_ml* from antiquotation *Type* (§2.1).

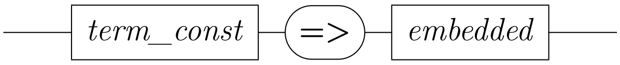
@{*Const_ source*} is similar to @{*Const source*} for patterns instead of closed values. This distinction is required due to redundant type information within term constants: subtrees with duplicate ML pattern variable need to be suppressed (replaced by dummy patterns). The embedded *source* follows the syntax category *term_const* defined below.

@{*Const_fn source*} is similar to @{*Const_ source*}, but produces a proper `fn` expression with function body. The embedded *source* follows the syntax category *term_const_fn* defined below.

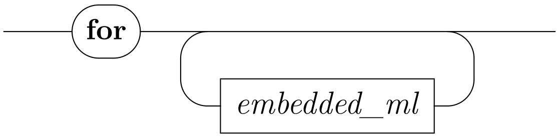
term_const



term_const_fn



for_args



ML

Examples

Here are some minimal examples for term constant antiquotations. Type arguments for constants are analogous to type constructors (§2.1). Term arguments are different: a flexible number of arguments is inserted via curried applications (`op $`).

```
ML ‹
  — constant without type argument:
  fun mk_conj (A, B) = Const‹conj for A B›;
  val dest_conj = Const_fn‹conj for A B => ‹(A, B)››;

  — constant with type argument:
  fun mk_eq T (t, u) = Const‹HOL.eq T for t u›;
  val dest_eq = Const_fn‹HOL.eq T for t u => ‹(T, (t, u))››;

  — constant with variable number of term arguments:
  val c = Const (const_name‹conj›, typ‹bool ⇒ bool ⇒ bool›);
  val P = term‹P::bool›;
  val Q = term‹Q::bool›;
  assert (Const‹conj› = c);
  assert (Const‹conj for P› = c $ P);
  assert (Const‹conj for P Q› = c $ P $ Q);
›
```

2.3 Theorems

A *proposition* is a well-typed term of type *prop*, a *theorem* is a proven proposition (depending on a context of hypotheses and the background theory). Primitive inferences include plain Natural Deduction rules for the primary connectives $\bigwedge$ and $\Longrightarrow$ of the framework. There is also a builtin notion of equality/equivalence $\equiv$.

2.3.1 Primitive connectives and rules

The theory *Pure* contains constant declarations for the primitive connectives $\bigwedge$, $\Longrightarrow$, and $\equiv$ of the logical framework, see figure 2.1. The derivability judgment $A_1, \ldots, A_n \vdash B$ is defined inductively by the primitive inferences given in figure 2.2, with the global restriction that the hypotheses must *not* contain any schematic variables. The builtin equality is conceptually axiomatized as shown in figure 2.3, although the implementation works directly with derived inferences.

$all :: (\alpha \Rightarrow prop) \Rightarrow prop$	universal quantification (binder $\bigwedge$)
$\Longrightarrow :: prop \Rightarrow prop \Rightarrow prop$	implication (right associative infix)
$\equiv :: \alpha \Rightarrow \alpha \Rightarrow prop$	equality relation (infix)

Figure 2.1: Primitive connectives of Pure

$$\frac{A \in \Theta}{\vdash A}\ (axiom) \qquad \frac{}{A \vdash A}\ (assume)$$

$$\frac{\Gamma \vdash B[x] \quad x \notin \Gamma}{\Gamma \vdash \bigwedge x.\ B[x]}\ (\bigwedge\text{-}intro) \qquad \frac{\Gamma \vdash \bigwedge x.\ B[x]}{\Gamma \vdash B[a]}\ (\bigwedge\text{-}elim)$$

$$\frac{\Gamma \vdash B}{\Gamma - A \vdash A \Longrightarrow B}\ (\Longrightarrow\text{-}intro) \qquad \frac{\Gamma_1 \vdash A \Longrightarrow B \quad \Gamma_2 \vdash A}{\Gamma_1 \cup \Gamma_2 \vdash B}\ (\Longrightarrow\text{-}elim)$$

Figure 2.2: Primitive inferences of Pure

The introduction and elimination rules for $\bigwedge$ and $\Longrightarrow$ are analogous to formation of dependently typed λ-terms representing the underlying proof objects. Proof terms are irrelevant in the Pure logic, though; they cannot occur within propositions. The system provides a runtime option to record explicit proof terms for primitive inferences, see also §2.5. Thus all three levels of λ-calculus become explicit: $\Rightarrow$ for terms, and $\bigwedge/\Longrightarrow$ for proofs (cf. [2]).

Observe that locally fixed parameters (as in $\bigwedge$-*intro*) need not be recorded in the hypotheses, because the simple syntactic types of Pure are always inhabitable. "Assumptions" $x :: \tau$ for type-membership are only present as long as some x_τ occurs in the statement body.[2]

[2] This is the key difference to "λHOL" in the PTS framework [1], where hypotheses $x : A$ are treated uniformly for propositions and types.

$\vdash (\lambda x.\ b[x])\ a \equiv b[a]$	β-conversion
$\vdash x \equiv x$	reflexivity
$\vdash x \equiv y \Longrightarrow P\ x \Longrightarrow P\ y$	substitution
$\vdash (\bigwedge x.\ f\, x \equiv g\ x) \Longrightarrow f \equiv g$	extensionality
$\vdash (A \Longrightarrow B) \Longrightarrow (B \Longrightarrow A) \Longrightarrow A \equiv B$	logical equivalence

Figure 2.3: Conceptual axiomatization of Pure equality

The axiomatization of a theory is implicitly closed by forming all instances of type and term variables: $\vdash A\theta$ holds for any substitution instance of an axiom $\vdash A$. By pushing substitutions through derivations inductively, we also get admissible *generalize* and *instantiate* rules as shown in figure 2.4.

$$\frac{\Gamma \vdash B[\alpha] \quad \alpha \notin \Gamma}{\Gamma \vdash B[?\alpha]} \quad \frac{\Gamma \vdash B[x] \quad x \notin \Gamma}{\Gamma \vdash B[?x]} \quad (\textit{generalize})$$

$$\frac{\Gamma \vdash B[?\alpha]}{\Gamma \vdash B[\tau]} \quad \frac{\Gamma \vdash B[?x]}{\Gamma \vdash B[t]} \quad (\textit{instantiate})$$

Figure 2.4: Admissible substitution rules

Note that *instantiate* does not require an explicit side-condition, because Γ may never contain schematic variables.

In principle, variables could be substituted in hypotheses as well, but this would disrupt the monotonicity of reasoning: deriving $\Gamma\theta \vdash B\theta$ from $\Gamma \vdash B$ is correct, but $\Gamma\theta \supseteq \Gamma$ does not necessarily hold: the result belongs to a different proof context.

An *oracle* is a function that produces axioms on the fly. Logically, this is an instance of the *axiom* rule (figure 2.2), but there is an operational difference. The inference kernel records oracle invocations within derivations of theorems by a unique tag. This also includes implicit type-class reasoning via the order-sorted algebra of class relations and type arities (see also **instantiation** and **instance**).

Axiomatizations should be limited to the bare minimum, typically as part of the initial logical basis of an object-logic formalization. Later on, theories are usually developed in a strictly definitional fashion, by stating only certain equalities over new constants.

A *simple definition* consists of a constant declaration $c :: \sigma$ together with an axiom $\vdash c \equiv t$, where $t :: \sigma$ is a closed term without any hidden polymorphism. The RHS may depend on further defined constants, but not c itself. Definitions of functions may be presented as $c\ \vec{x} \equiv t$ instead of the puristic $c \equiv \lambda\vec{x}.\ t$.

An *overloaded definition* consists of a collection of axioms for the same constant, with zero or one equations $c((\vec{\alpha})\kappa) \equiv t$ for each type constructor κ (for distinct variables $\vec{\alpha}$). The RHS may mention previously defined constants as above, or arbitrary constants $d(\alpha_i)$ for some α_i projected from $\vec{\alpha}$. Thus overloaded definitions essentially work by primitive recursion over the syntactic structure of a single type argument. See also [8, §4.3].

ML Reference

```
Logic.all: term -> term -> term
Logic.mk_implies: term * term -> term

type ctyp
type cterm
Thm.ctyp_of: Proof.context -> typ -> ctyp
Thm.cterm_of: Proof.context -> term -> cterm
Thm.apply: cterm -> cterm -> cterm
Thm.lambda: cterm -> cterm -> cterm
Thm.all: Proof.context -> cterm -> cterm -> cterm
Drule.mk_implies: cterm * cterm -> cterm

type thm
Thm.transfer: theory -> thm -> thm
Thm.assume: cterm -> thm
Thm.forall_intr: cterm -> thm -> thm
Thm.forall_elim: cterm -> thm -> thm
Thm.implies_intr: cterm -> thm -> thm
Thm.implies_elim: thm -> thm -> thm
Thm.generalize: Names.set * Names.set -> int -> thm -> thm
Thm.instantiate: ctyp TVars.table * cterm Vars.table -> thm -> thm
Thm.add_axiom: Proof.context ->
  binding * term -> theory -> (string * thm) * theory
Thm.add_oracle: binding * (’a -> cterm) -> theory ->
  (string * (’a -> thm)) * theory
Thm.add_def: Defs.context -> bool -> bool ->
  binding * term -> theory -> (string * thm) * theory

Theory.add_deps: Defs.context -> string ->
  Defs.entry -> Defs.entry list -> theory -> theory
Thm_Deps.all_oracles: thm list -> Proofterm.oracle list
```

`Logic.all` a B produces a Pure quantification $\bigwedge a.\ B$, where occurrences of the atomic term a in the body proposition B are replaced by bound variables. (See also `lambda` on terms.)

`Logic.mk_implies` $(A,\ B)$ produces a Pure implication $A \Longrightarrow B$.

Types `ctyp` and `cterm` represent certified types and terms, respectively. These are abstract datatypes that guarantee that its values have passed the full well-formedness (and well-typedness) checks, relative to the declarations of type constructors, constants etc. in the background theory.

The abstract types `ctyp` and `cterm` are part of the same inference kernel that is mainly responsible for `thm`. Thus syntactic operations on `ctyp` and `cterm` are located in the `Thm` module, even though theorems are not yet involved at that stage.

`Thm.ctyp_of` *ctxt* τ and `Thm.cterm_of` *ctxt* t explicitly check types and terms, respectively. This also involves some basic normalizations, such expansion of type and term abbreviations from the underlying theory context. Full re-certification is relatively slow and should be avoided in tight reasoning loops.

`Thm.apply`, `Thm.lambda`, `Thm.all`, `Drule.mk_implies` etc. compose certified terms (or propositions) incrementally. This is equivalent to `Thm.cterm_of` after unchecked `$`, `lambda`, `Logic.all`, `Logic.mk_implies` etc., but there can be a big difference in performance when large existing entities are composed by a few extra constructions on top. There are separate operations to decompose certified terms and theorems to produce certified terms again.

Type `thm` represents proven propositions. This is an abstract datatype that guarantees that its values have been constructed by basic principles of the `Thm` module. Every `thm` value refers its background theory, cf. §1.1.1.

`Thm.transfer` *thy thm* transfers the given theorem to a *larger* theory, see also §1.1. This formal adjustment of the background context has no logical significance, but is occasionally required for formal reasons, e.g. when theorems that are imported from more basic theories are used in the current situation.

`Thm.assume`, `Thm.forall_intr`, `Thm.forall_elim`, `Thm.implies_intr`, and `Thm.implies_elim` correspond to the primitive inferences of figure 2.2.

`Thm.generalize` $(\vec{\alpha}, \vec{x})$ corresponds to the *generalize* rules of figure 2.4. Here collections of type and term variables are generalized simultaneously, specified by the given sets of basic names.

`Thm.instantiate` $(\vec{\alpha}_s, \vec{x}_\tau)$ corresponds to the *instantiate* rules of figure 2.4. Type variables are substituted before term variables. Note that the types in $\vec{x}_\tau$ refer to the instantiated versions.

`Thm.add_axiom` *ctxt* (*name*, A) declares an arbitrary proposition as axiom, and retrieves it as a theorem from the resulting theory, cf. *axiom* in figure 2.2. Note that the low-level representation in the axiom table may differ slightly from the returned theorem.

`Thm.add_oracle` (*binding*, *oracle*) produces a named oracle rule, essentially generating arbitrary axioms on the fly, cf. *axiom* in figure 2.2.

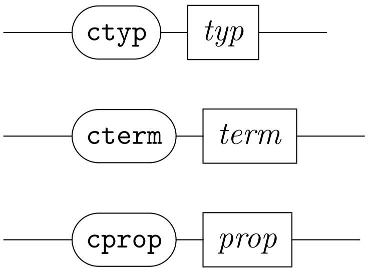
`Thm.add_def` *ctxt unchecked overloaded* (*name*, $c\ \vec{x} \equiv t$) states a definitional axiom for an existing constant c. Dependencies are recorded via `Theory.add_deps`, unless the *unchecked* option is set. Note that the low-level representation in the axiom table may differ slightly from the returned theorem.

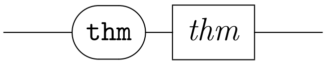
`Theory.add_deps` *ctxt name* c_τ $\vec{d}_\sigma$ declares dependencies of a named specification for constant c_τ, relative to existing specifications for constants $\vec{d}_\sigma$. This also works for type constructors.

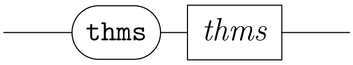
`Thm_Deps.all_oracles` *thms* returns all oracles used in the internal derivation of the given theorems; this covers the full graph of transitive dependencies. The result contains an authentic oracle name; if `Proofterm.proofs` was at least `1` during the oracle inference, it also contains the position of the oracle invocation and its proposition. See also the command **thm_oracles**.

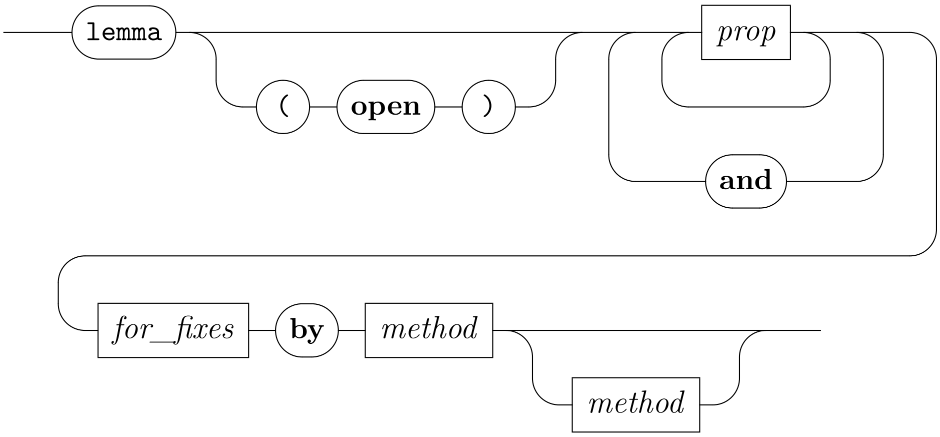
ML Antiquotations

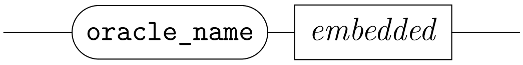
ctyp	:	*ML_antiquotation*
cterm	:	*ML_antiquotation*
cprop	:	*ML_antiquotation*
thm	:	*ML_antiquotation*
thms	:	*ML_antiquotation*
lemma	:	*ML_antiquotation*
oracle_name	:	*ML_antiquotation*











@{*ctyp* τ} produces a certified type wrt. the current background theory — as abstract value of type `ctyp`.

@{*cterm t*} and @{*cprop* φ} produce a certified term wrt. the current background theory — as abstract value of type `cterm`.

@{*thm a*} produces a singleton fact — as abstract value of type `thm`.

@{*thms a*} produces a general fact — as abstract value of type `thm list`.

@{*lemma* φ *by meth*} produces a fact that is proven on the spot according to the minimal proof, which imitates a terminal Isar proof. The result is an abstract value of type `thm` or `thm list`, depending on the number of propositions given here.

The internal derivation object lacks a proper theorem name, but it is formally closed, unless the (*open*) option is specified (this may impact performance of applications with proof terms).

Since ML antiquotations are always evaluated at compile-time, there is no run-time overhead even for non-trivial proofs. Nonetheless, the justification is syntactically limited to a single **by** step. More complex

Isar proofs should be done in regular theory source, before compiling the corresponding ML text that uses the result.

@{*oracle_name a*} inlines the internalized oracle name a — as `string` literal.

2.3.2 Auxiliary connectives

Theory *Pure* provides a few auxiliary connectives that are defined on top of the primitive ones, see figure 2.5. These special constants are useful in certain internal encodings, and are normally not directly exposed to the user.

$\mathit{conjunction} :: \mathit{prop} \Rightarrow \mathit{prop} \Rightarrow \mathit{prop}$ (infix &&&)
$\vdash A \mathbin{\&\&\&} B \equiv (\bigwedge C.\ (A \Longrightarrow B \Longrightarrow C) \Longrightarrow C)$

$\mathit{prop} :: \mathit{prop} \Rightarrow \mathit{prop}$ (prefix #, suppressed)
$\#A \equiv A$

$\mathit{term} :: \alpha \Rightarrow \mathit{prop}$ (prefix *TERM*)
$\mathit{term}\ x \equiv (\bigwedge A.\ A \Longrightarrow A)$

$\mathit{type} :: \alpha\ \mathit{itself}$ (prefix *TYPE*)
(*unspecified*)

Figure 2.5: Definitions of auxiliary connectives

The introduction $A \Longrightarrow B \Longrightarrow A \mathbin{\&\&\&} B$, and eliminations (projections) $A \mathbin{\&\&\&} B \Longrightarrow A$ and $A \mathbin{\&\&\&} B \Longrightarrow B$ are available as derived rules. Conjunction allows to treat simultaneous assumptions and conclusions uniformly, e.g. consider $A \Longrightarrow B \Longrightarrow C \mathbin{\&\&\&} D$. In particular, the goal mechanism represents multiple claims as explicit conjunction internally, but this is refined (via backwards introduction) into separate sub-goals before the user commences the proof; the final result is projected into a list of theorems using eliminations (cf. §4.1).

The *prop* marker (#) makes arbitrarily complex propositions appear as atomic, without changing the meaning: $\Gamma \vdash A$ and $\Gamma \vdash \#A$ are interchangeable. See §4.1 for specific operations.

The *term* marker turns any well-typed term into a derivable proposition: $\vdash \mathit{TERM}\ t$ holds unconditionally. Although this is logically vacuous, it allows to treat terms and proofs uniformly, similar to a type-theoretic framework.

The *TYPE* constructor is the canonical representative of the unspecified type α *itself*; it essentially injects the language of types into that of terms. There is specific notation $TYPE(\tau)$ for $TYPE_{\tau\ itself}$. Although being devoid of any particular meaning, the term $TYPE(\tau)$ accounts for the type τ within the term language. In particular, $TYPE(\alpha)$ may be used as formal argument in primitive definitions, in order to circumvent hidden polymorphism (cf. §2.2). For example, $c\ TYPE(\alpha) \equiv A[\alpha]$ defines $c :: \alpha\ itself \Rightarrow prop$ in terms of a proposition A that depends on an additional type argument, which is essentially a predicate on types.

ML Reference

```
Conjunction.intr: thm -> thm -> thm
Conjunction.elim: thm -> thm * thm
Drule.mk_term: cterm -> thm
Drule.dest_term: thm -> cterm
Logic.mk_type: typ -> term
Logic.dest_type: term -> typ
```

`Conjunction.intr` derives A &&& B from A and B.

`Conjunction.elim` derives A and B from A &&& B.

`Drule.mk_term` derives $TERM\ t$.

`Drule.dest_term` recovers term t from $TERM\ t$.

`Logic.mk_type` τ produces the term $TYPE(\tau)$.

`Logic.dest_type` $TYPE(\tau)$ recovers the type τ.

2.3.3 Sort hypotheses

Type variables are decorated with sorts, as explained in §2.1. This constrains type instantiation to certain ranges of types: variable α_s may only be assigned to types τ that belong to sort s. Within the logic, sort constraints act like implicit preconditions on the result $(\!|\alpha_1 : s_1|\!), \ldots, (\!|\alpha_n : s_n|\!), \Gamma \vdash \varphi$ where the type variables $\alpha_1, \ldots, \alpha_n$ cover the propositions Γ, φ, as well as the proof of $\Gamma \vdash \varphi$.

These *sort hypothesis* of a theorem are passed monotonically through further derivations. They are redundant, as long as the statement of a theorem still contains the type variables that are accounted here. The logical significance of sort hypotheses is limited to the boundary case where type variables disappear from the proposition, e.g. $(\!|\alpha_s : s|\!) \vdash \varphi$. Since such dangling type variables can be renamed arbitrarily without changing the proposition φ, the inference kernel maintains sort hypotheses in anonymous form $s \vdash \varphi$.

In most practical situations, such extra sort hypotheses may be stripped in a final bookkeeping step, e.g. at the end of a proof: they are typically left over from intermediate reasoning with type classes that can be satisfied by some concrete type τ of sort s to replace the hypothetical type variable α_s.

ML Reference

```
Thm.extra_shyps: thm -> sort list
Thm.strip_shyps: thm -> thm
```

`Thm.extra_shyps` *thm* determines the extraneous sort hypotheses of the given theorem, i.e. the sorts that are not present within type variables of the statement.

`Thm.strip_shyps` *thm* removes any extraneous sort hypotheses that can be witnessed from the type signature.

ML Examples

The following artificial example demonstrates the derivation of *False* with a pending sort hypothesis involving a logically empty sort.

class *empty* =
  assumes *bad*: $\bigwedge(x{::}'a)\ y.\ x \neq y$

theorem (**in** *empty*) *false*: *False*
  using *bad* **by** *blast*

ML_val ‹*assert* (Thm.extra_shyps @{thm false} = [*sort*‹empty›])›

Thanks to the inference kernel managing sort hypothesis according to their logical significance, this example is merely an instance of *ex falso quodlibet consequitur* — not a collapse of the logical framework!

2.4 Object-level rules

The primitive inferences covered so far mostly serve foundational purposes. User-level reasoning usually works via object-level rules that are represented as theorems of Pure. Composition of rules involves *backchaining*, *higher-order unification* modulo $\alpha\beta\eta$-conversion of λ-terms, and so-called *lifting* of rules into a context of $\bigwedge$ and $\Longrightarrow$ connectives. Thus the full power of higher-order Natural Deduction in Isabelle/Pure becomes readily available.

2.4.1 Hereditary Harrop Formulae

The idea of object-level rules is to model Natural Deduction inferences in the style of Gentzen [7], but we allow arbitrary nesting similar to [16]. The most basic rule format is that of a *Horn Clause*:

$$\frac{A_1 \quad \ldots \quad A_n}{A}$$

where A, A_1, …, A_n are atomic propositions of the framework, usually of the form *Trueprop B*, where B is a (compound) object-level statement. This object-level inference corresponds to an iterated implication in Pure like this:

$$A_1 \Longrightarrow \ldots A_n \Longrightarrow A$$

As an example consider conjunction introduction: $A \Longrightarrow B \Longrightarrow A \wedge B$. Any parameters occurring in such rule statements are conceptionally treated as arbitrary:

$$\bigwedge x_1 \ldots x_m.\ A_1\ x_1 \ldots x_m \Longrightarrow \ldots A_n\ x_1 \ldots x_m \Longrightarrow A\ x_1 \ldots x_m$$

Nesting of rules means that the positions of A_i may again hold compound rules, not just atomic propositions. Propositions of this format are called *Hereditary Harrop Formulae* in the literature [10]. Here we give an inductive characterization as follows:

$\mathbf{x}$	set of variables
$\mathbf{A}$	set of atomic propositions
$\mathbf{H} = \bigwedge \mathbf{x}^*.\ \mathbf{H}^* \Longrightarrow \mathbf{A}$	set of Hereditary Harrop Formulas

Thus we essentially impose nesting levels on propositions formed from $\bigwedge$ and $\Longrightarrow$. At each level there is a prefix of parameters and compound premises, concluding an atomic proposition. Typical examples are $\longrightarrow$-introduction $(A \Longrightarrow B) \Longrightarrow A \longrightarrow B$ or mathematical induction $P\ 0 \Longrightarrow (\bigwedge n.\ P\ n \Longrightarrow P\ (Suc\ n)) \Longrightarrow P\ n$. Even deeper nesting occurs in well-founded induction $(\bigwedge x.\ (\bigwedge y.\ y \prec x \Longrightarrow P\ y) \Longrightarrow P\ x) \Longrightarrow P\ x$, but this already marks the limit of rule complexity that is usually seen in practice.

Regular user-level inferences in Isabelle/Pure always maintain the following canonical form of results:

- Normalization by $(A \Longrightarrow (\bigwedge x.\ B\ x)) \equiv (\bigwedge x.\ A \Longrightarrow B\ x)$, which is a theorem of Pure, means that quantifiers are pushed in front of implication at each level of nesting. The normal form is a Hereditary Harrop Formula.
- The outermost prefix of parameters is represented via schematic variables: instead of $\bigwedge \vec{x}.\ \vec{H}\ \vec{x} \Longrightarrow A\ \vec{x}$ we have $\vec{H}\ ?\vec{x} \Longrightarrow A\ ?\vec{x}$. Note that this representation looses information about the order of parameters, and vacuous quantifiers vanish automatically.

ML Reference

```
Simplifier.norm_hhf: Proof.context -> thm -> thm
```

`Simplifier.norm_hhf` *ctxt thm* normalizes the given theorem according to the canonical form specified above. This is occasionally helpful to repair some low-level tools that do not handle Hereditary Harrop Formulae properly.

2.4.2 Rule composition

The rule calculus of Isabelle/Pure provides two main inferences: *resolution* (i.e. back-chaining of rules) and *assumption* (i.e. closing a branch), both modulo higher-order unification. There are also combined variants, notably *elim_resolution* and *dest_resolution*.

To understand the all-important *resolution* principle, we first consider raw *composition* (modulo higher-order unification with substitution θ):

$$\frac{\vec{A} \Longrightarrow B \quad B' \Longrightarrow C \quad B\theta = B'\theta}{\vec{A}\theta \Longrightarrow C\theta}\ (composition)$$

Here the conclusion of the first rule is unified with the premise of the second; the resulting rule instance inherits the premises of the first and conclusion of the second. Note that C can again consist of iterated implications. We can also permute the premises of the second rule back-and-forth in order to compose with B' in any position (subsequently we shall always refer to position 1 w.l.o.g.).

In *composition* the internal structure of the common part B and B' is not taken into account. For proper *resolution* we require B to be atomic, and explicitly observe the structure $\bigwedge\vec{x}.\ \vec{H}\ \vec{x} \Longrightarrow B'\ \vec{x}$ of the premise of the second rule. The idea is to adapt the first rule by "lifting" it into this context, by means of iterated application of the following inferences:

$$\frac{\vec{A} \Longrightarrow B}{(\vec{H} \Longrightarrow \vec{A}) \Longrightarrow (\vec{H} \Longrightarrow B)}\ (\mathit{imp\_lift})$$

$$\frac{\vec{A}\ ?\vec{a} \Longrightarrow B\ ?\vec{a}}{(\bigwedge\vec{x}.\ \vec{A}\ (?\vec{a}\ \vec{x})) \Longrightarrow (\bigwedge\vec{x}.\ B\ (?\vec{a}\ \vec{x}))}\ (\mathit{all\_lift})$$

By combining raw composition with lifting, we get full *resolution* as follows:

$$\frac{\begin{array}{l}\vec{A}\ ?\vec{a} \Longrightarrow B\ ?\vec{a} \\ (\bigwedge\vec{x}.\ \vec{H}\ \vec{x} \Longrightarrow B'\ \vec{x}) \Longrightarrow C \\ (\lambda\vec{x}.\ B\ (?\vec{a}\ \vec{x}))\theta = B'\theta\end{array}}{(\bigwedge\vec{x}.\ \vec{H}\ \vec{x} \Longrightarrow \vec{A}\ (?\vec{a}\ \vec{x}))\theta \Longrightarrow C\theta}\ (\mathit{resolution})$$

Continued resolution of rules allows to back-chain a problem towards more and sub-problems. Branches are closed either by resolving with a rule of 0 premises, or by producing a "short-circuit" within a solved situation (again modulo unification):

$$\frac{(\bigwedge\vec{x}.\ \vec{H}\ \vec{x} \Longrightarrow A\ \vec{x}) \Longrightarrow C \quad A\theta = H_i\theta\ \text{ (for some } i\text{)}}{C\theta}\ (\mathit{assumption})$$

ML Reference

```
infix RSN: thm * (int * thm) -> thm
infix RS: thm * thm -> thm
infix RLN: thm list * (int * thm list) -> thm list
infix RL: thm list * thm list -> thm list
infix MRS: thm list * thm -> thm
infix OF: thm * thm list -> thm
```

rule_1 *RSN* $(i,\ \mathit{rule}_2)$ resolves the conclusion of rule_1 with the i-th premise of rule_2, according to the *resolution* principle explained above. Unless there is precisely one resolvent it raises exception `THM`.

This corresponds to the rule attribute *THEN* in Isar source language.

rule_1 *RS* rule_2 abbreviates rule_1 *RSN* $(1,\ \mathit{rule}_2)$.

$rules_1$ RLN (i, *$rules_2$*) joins lists of rules. For every *$rule_1$* in *$rules_1$* and *$rule_2$* in *$rules_2$*, it resolves the conclusion of *$rule_1$* with the i-th premise of *$rule_2$*, accumulating multiple results in one big list. Note that such strict enumerations of higher-order unifications can be inefficient compared to the lazy variant seen in elementary tactics like `resolve_tac`.

$rules_1$ RL $rules_2$ abbreviates *$rules_1$ RLN* (1, *$rules_2$*).

[*$rule_1$*, ..., *$rule_n$*] *MRS rule* resolves *$rule_i$* against premise i of *rule*, for $i = n, \ldots, 1$. By working from right to left, newly emerging premises are concatenated in the result, without interfering.

rule OF rules is an alternative notation for *rules MRS rule*, which makes rule composition look more like function application. Note that the argument *rules* need not be atomic.

This corresponds to the rule attribute *OF* in Isar source language.

2.5 Proof terms

The Isabelle/Pure inference kernel can record the proof of each theorem as a proof term that contains all logical inferences in detail. Rule composition by resolution (§2.4) and type-class reasoning is broken down to primitive rules of the logical framework. The proof term can be inspected by a separate proof-checker, for example.

According to the well-known *Curry-Howard isomorphism*, a proof can be viewed as a λ-term. Following this idea, proofs in Isabelle are internally represented by a datatype similar to the one for terms described in §2.2. On top of these syntactic terms, two more layers of λ-calculus are added, which correspond to $\bigwedge x :: \alpha.\ B\ x$ and $A \Longrightarrow B$ according to the propositions-as-types principle. The resulting 3-level λ-calculus resembles "λHOL" in the more abstract setting of Pure Type Systems (PTS) [1], if some fine points like schematic polymorphism and type classes are ignored.

Proof abstractions of the form $\boldsymbol{\lambda} x :: \alpha.\ prf$ or $\boldsymbol{\lambda} p : A.\ prf$ correspond to introduction of $\bigwedge/\Longrightarrow$, and *proof applications* of the form $p \cdot t$ or $p \bullet q$ correspond to elimination of $\bigwedge/\Longrightarrow$. Actual types α, propositions A, and terms t might be suppressed and reconstructed from the overall proof term.

Various atomic proofs indicate special situations within the proof construction as follows.

A *bound proof variable* is a natural number b that acts as de-Bruijn index for proof term abstractions.

A *minimal proof* "*?*" is a dummy proof term. This indicates some unrecorded part of the proof.

Hyp A refers to some pending hypothesis by giving its proposition. This indicates an open context of implicit hypotheses, similar to loose bound variables or free variables within a term (§2.2).

An *axiom* or *oracle* $a : A[\vec{\tau}]$ refers some postulated *proof constant*, which is subject to schematic polymorphism of theory content, and the particular type instantiation may be given explicitly. The vector of types $\vec{\tau}$ refers to the schematic type variables in the generic proposition A in canonical order.

A *proof promise* $a : A[\vec{\tau}]$ is a placeholder for some proof of polymorphic proposition A, with explicit type instantiation as given by the vector $\vec{\tau}$, as above. Unlike axioms or oracles, proof promises may be *fulfilled* eventually, by substituting a by some particular proof q at the corresponding type instance. This acts like Hindley-Milner *let*-polymorphism: a generic local proof definition may get used at different type instances, and is replaced by the concrete instance eventually.

A *named theorem* wraps up some concrete proof as a closed formal entity, in the manner of constant definitions for proof terms. The *proof body* of such boxed theorems involves some digest about oracles and promises occurring in the original proof. This allows the inference kernel to manage this critical information without the full overhead of explicit proof terms.

2.5.1 Reconstructing and checking proof terms

Fully explicit proof terms can be large, but most of this information is redundant and can be reconstructed from the context. Therefore, the Isabelle/Pure inference kernel records only *implicit* proof terms, by omitting all typing information in terms, all term and type labels of proof abstractions, and some argument terms of applications $p \cdot t$ (if possible).

There are separate operations to reconstruct the full proof term later on, using *higher-order pattern unification* [11, 2].

The *proof checker* expects a fully reconstructed proof term, and can turn it into a theorem by replaying its primitive inferences within the kernel.

2.5.2 Concrete syntax of proof terms

The concrete syntax of proof terms is a slight extension of the regular inner syntax of Isabelle/Pure [19]. Its main syntactic category *proof* is defined as follows:

$$\begin{array}{rcl}
\mathit{proof} & = & \boldsymbol{\lambda}\ \mathit{params}\ \texttt{.}\ \mathit{proof} \\
 & | & \mathit{proof}\ \cdot\ \mathit{any} \\
 & | & \mathit{proof}\ \bullet\ \mathit{proof} \\
 & | & \mathit{id}\ \ |\ \ \mathit{longid} \\
 & & \\
\mathit{param} & = & \mathit{idt} \\
 & | & \mathit{idt}\ \texttt{:}\ \mathit{prop} \\
 & | & \texttt{(}\ \mathit{param}\ \texttt{)} \\
 & & \\
\mathit{params} & = & \mathit{param} \\
 & | & \mathit{param}\ \mathit{params}
\end{array}$$

Implicit term arguments in partial proofs are indicated by “_”. Type arguments for theorems and axioms may be specified using $p \cdot \mathit{TYPE}(\mathit{type})$ (they must appear before any other term argument of a theorem or axiom, but may be omitted altogether).

There are separate read and print operations for proof terms, in order to avoid conflicts with the regular term language.

ML Reference

```
type proof
type proof_body
Proofterm.proofs: int Unsynchronized.ref
Proofterm.reconstruct_proof: theory -> term -> proof -> proof
Proofterm.expand_proof: theory ->
  (Proofterm.thm_header -> string option) -> proof -> proof
Proof_Checker.thm_of_proof: theory -> proof -> thm
Proof_Syntax.read_proof: theory -> bool -> bool -> string -> proof
Proof_Syntax.pretty_proof: Proof.context -> proof -> Pretty.T
```

Type `proof` represents proof terms; this is a datatype with constructors `Abst`, `AbsP`, `infix %`, `infix %%`, `PBound`, `MinProof`, `Hyp`, `PAxm`, `Oracle`, `PThm` as explained above.

Type `proof_body` represents the nested proof information of a named theorem, consisting of a digest of oracles and named theorem over some proof term. The digest only covers the directly visible part of the proof: in order to get the full information, the implicit graph of nested theorems needs to be traversed (e.g. using `Proofterm.fold_body_thms`).

`Thm.proof_of` *thm* and `Thm.proof_body_of` *thm* produce the proof term or proof body (with digest of oracles and theorems) from a given theorem. Note that this involves a full join of internal futures that fulfill pending proof promises, and thus disrupts the natural bottom-up construction of proofs by introducing dynamic ad-hoc dependencies. Parallel performance may suffer by inspecting proof terms at run-time.

`Proofterm.proofs` specifies the detail of proof recording within `thm` values produced by the inference kernel: `0` records only the names of oracles, `1` records oracle names and propositions, `2` additionally records full proof terms. Officially named theorems that contribute to a result are recorded in any case.

`Proofterm.reconstruct_proof` *thy prop prf* turns the implicit proof term *prf* into a full proof of the given proposition.

Reconstruction may fail if *prf* is not a proof of *prop*, or if it does not contain sufficient information for reconstruction. Failure may only happen for proofs that are constructed manually, but not for those produced automatically by the inference kernel.

`Proofterm.expand_proof` *thy expand prf* reconstructs and expands the proofs of nested theorems according to the given *expand* function: a result of `SOME ""` means full expansion, `SOME` *name* means to keep the theorem node but replace its internal name, `NONE` means no change.

`Proof_Checker.thm_of_proof` *thy prf* turns the given (full) proof into a theorem, by replaying it using only primitive rules of the inference kernel.

`Proof_Syntax.read_proof` *thy* b_1 b_2 *s* reads in a proof term. The Boolean flags indicate the use of sort and type information. Usually, typing information is left implicit and is inferred during proof reconstruction.

`Proof_Syntax.pretty_proof` *ctxt prf* pretty-prints the given proof term.

ML Examples

- `~~/src/HOL/Proofs/ex/Proof_Terms.thy` provides basic examples involving proof terms.

- `~~/src/HOL/Proofs/ex/XML_Data.thy` demonstrates export and import of proof terms via XML/ML data representation.

Chapter 3

Concrete syntax and type-checking

Pure λ-calculus as introduced in chapter 2 is an adequate foundation for logical languages — in the tradition of *higher-order abstract syntax* — but end-users require additional means for reading and printing of terms and types. This important add-on outside the logical core is called *inner syntax* in Isabelle jargon, as opposed to the *outer syntax* of the theory and proof language [19].

For example, according to [4] quantifiers are represented as higher-order constants $All :: ('a \Rightarrow bool) \Rightarrow bool$ such that $All\ (\lambda x{::}'a.\ B\ x)$ faithfully represents the idea that is displayed in Isabelle as $\forall\, x{::}'a.\ B\ x$ via **binder** notation. Moreover, type-inference in the style of Hindley-Milner [5] (and extensions) enables users to write $\forall\, x.\ B\ x$ concisely, when the type $'a$ is already clear from the context.[1]

The main inner syntax operations are *read* for parsing together with type-checking, and *pretty* for formatted output. See also §3.1.

Furthermore, the input and output syntax layers are sub-divided into separate phases for *concrete syntax* versus *abstract syntax*, see also §3.2 and §3.3, respectively. This results in the following decomposition of the main operations:

- $read = parse;\ check$
- $pretty = uncheck;\ unparse$

For example, some specification package might thus intercept syntax processing at a well-defined stage after *parse*, to a augment the resulting pre-term before full type-reconstruction is performed by *check*. Note that the formal status of bound variables, versus free variables, versus constants must not be changed between these phases.

[1] Type-inference taken to the extreme can easily confuse users. Beginners often stumble over unexpectedly general types inferred by the system.

In general, *check* and *uncheck* operate simultaneously on a list of terms. This is particular important for type-checking, to reconstruct types for several terms of the same context and scope. In contrast, *parse* and *unparse* operate separately on single terms.

There are analogous operations to read and print types, with the same subdivision into phases.

3.1 Reading and pretty printing

Read and print operations are roughly dual to each other, such that for the user $s' = \mathit{pretty}\ (\mathit{read}\ s)$ looks similar to the original source text s, but the details depend on many side-conditions. There are also explicit options to control the removal of type information in the output. The default configuration routinely looses information, so $t' = \mathit{read}\ (\mathit{pretty}\ t)$ might fail, or produce a differently typed term, or a completely different term in the face of syntactic overloading.

ML Reference

```
Syntax.read_typs: Proof.context -> string list -> typ list
Syntax.read_terms: Proof.context -> string list -> term list
Syntax.read_props: Proof.context -> string list -> term list
Syntax.read_typ: Proof.context -> string -> typ
Syntax.read_term: Proof.context -> string -> term
Syntax.read_prop: Proof.context -> string -> term
Syntax.pretty_typ: Proof.context -> typ -> Pretty.T
Syntax.pretty_term: Proof.context -> term -> Pretty.T
Syntax.string_of_typ: Proof.context -> typ -> string
Syntax.string_of_term: Proof.context -> term -> string
```

`Syntax.read_typs` *ctxt strs* parses and checks a simultaneous list of source strings as types of the logic.

`Syntax.read_terms` *ctxt strs* parses and checks a simultaneous list of source strings as terms of the logic. Type-reconstruction puts all parsed terms into the same scope: types of free variables ultimately need to coincide.

If particular type-constraints are required for some of the arguments, the read operations needs to be split into its parse and check phases. Then it is possible to use `Type.constraint` on the intermediate preterms (§3.3).

`Syntax.read_props` *ctxt strs* parses and checks a simultaneous list of source strings as terms of the logic, with an implicit type-constraint for each argument to enforce type *prop*; this also affects the inner syntax for parsing. The remaining type-reconstruction works as for `Syntax.read_terms`.

`Syntax.read_typ`, `Syntax.read_term`, `Syntax.read_prop` are like the simultaneous versions, but operate on a single argument only. This convenient shorthand is adequate in situations where a single item in its own scope is processed. Do not use `map o Syntax.read_term` where `Syntax.read_terms` is actually intended!

`Syntax.pretty_typ` *ctxt T* and `Syntax.pretty_term` *ctxt t* uncheck and pretty-print the given type or term, respectively. Although the uncheck phase acts on a simultaneous list as well, this is rarely used in practice, so only the singleton case is provided as combined pretty operation. There is no distinction of term vs. proposition.

`Syntax.string_of_typ` and `Syntax.string_of_term` are convenient compositions of `Syntax.pretty_typ` and `Syntax.pretty_term` with `Pretty.string_of` for output. The result may be concatenated with other strings, as long as there is no further formatting and line-breaking involved.

`Syntax.read_term`, `Syntax.read_prop`, and `Syntax.string_of_term` are the most important operations in practice.

Note that the string values that are passed in and out are annotated by the system, to carry further markup that is relevant for the Prover IDE [20]. User code should neither compose its own input strings, nor try to analyze the output strings. Conceptually this is an abstract datatype, encoded as concrete string for historical reasons.

The standard way to provide the required position markup for input works via the outer syntax parser wrapper `Parse.inner_syntax`, which is already part of `Parse.typ`, `Parse.term`, `Parse.prop`. So a string obtained from one of the latter may be directly passed to the corresponding read operation: this yields PIDE markup of the input and precise positions for warning and error messages.

3.2 Parsing and unparsing

Parsing and unparsing converts between actual source text and a certain *pre-term* format, where all bindings and scopes are already resolved faithfully. Thus the names of free variables or constants are determined in the sense of the logical context, but type information might be still missing. Pre-terms support an explicit language of *type constraints* that may be augmented by user code to guide the later *check* phase.

Actual parsing is based on traditional lexical analysis and Earley parsing for arbitrary context-free grammars. The user can specify the grammar declaratively via mixfix annotations. Moreover, there are *syntax translations* that can be augmented by the user, either declaratively via **translations** or programmatically via **parse_translation**, **print_translation** [19]. The final scope-resolution is performed by the system, according to name spaces for types, term variables and constants determined by the context.

ML Reference

```
Syntax.parse_typ: Proof.context -> string -> typ
Syntax.parse_term: Proof.context -> string -> term
Syntax.parse_prop: Proof.context -> string -> term
Syntax.unparse_typ: Proof.context -> typ -> Pretty.T
Syntax.unparse_term: Proof.context -> term -> Pretty.T
```

`Syntax.parse_typ` *ctxt str* parses a source string as pre-type that is ready to be used with subsequent check operations.

`Syntax.parse_term` *ctxt str* parses a source string as pre-term that is ready to be used with subsequent check operations.

`Syntax.parse_prop` *ctxt str* parses a source string as pre-term that is ready to be used with subsequent check operations. The inner syntax category is *prop* and a suitable type-constraint is included to ensure that this information is observed in subsequent type reconstruction.

`Syntax.unparse_typ` *ctxt T* unparses a type after uncheck operations, to turn it into a pretty tree.

`Syntax.unparse_term` *ctxt T* unparses a term after uncheck operations, to turn it into a pretty tree. There is no distinction for propositions here.

These operations always operate on a single item; use the combinator `map` to apply them to a list.

3.3 Checking and unchecking

These operations define the transition from pre-terms and fully-annotated terms in the sense of the logical core (chapter 2).

The *check* phase is meant to subsume a variety of mechanisms in the manner of "type-inference" or "type-reconstruction" or "type-improvement", not just type-checking in the narrow sense. The *uncheck* phase is roughly dual, it prunes type-information before pretty printing.

A typical add-on for the check/uncheck syntax layer is the **abbreviation** mechanism [19]. Here the user specifies syntactic definitions that are managed by the system as polymorphic *let* bindings. These are expanded during the *check* phase, and contracted during the *uncheck* phase, without affecting the type-assignment of the given terms.

The precise meaning of type checking depends on the context — additional check/uncheck modules might be defined in user space.

For example, the **class** command defines a context where *check* treats certain type instances of overloaded constants according to the "dictionary construction" of its logical foundation. This involves "type improvement" (specialization of slightly too general types) and replacement by certain locale parameters. See also [9].

ML Reference

```
Syntax.check_typs: Proof.context -> typ list -> typ list
Syntax.check_terms: Proof.context -> term list -> term list
Syntax.check_props: Proof.context -> term list -> term list
Syntax.uncheck_typs: Proof.context -> typ list -> typ list
Syntax.uncheck_terms: Proof.context -> term list -> term list
```

`Syntax.check_typs` *ctxt Ts* checks a simultaneous list of pre-types as types of the logic. Typically, this involves normalization of type synonyms.

`Syntax.check_terms` *ctxt ts* checks a simultaneous list of pre-terms as terms of the logic. Typically, this involves type-inference and normalization term abbreviations. The types within the given terms are treated in the same way as for `Syntax.check_typs`.

Applications sometimes need to check several types and terms together. The standard approach uses `Logic.mk_type` to embed the language of types into that of terms; all arguments are appended into one list of terms that is checked; afterwards the type arguments are recovered with `Logic.dest_type`.

`Syntax.check_props` *ctxt ts* checks a simultaneous list of pre-terms as terms of the logic, such that all terms are constrained by type *prop*. The remaining check operation works as `Syntax.check_terms` above.

`Syntax.uncheck_typs` *ctxt Ts* unchecks a simultaneous list of types of the logic, in preparation of pretty printing.

`Syntax.uncheck_terms` *ctxt ts* unchecks a simultaneous list of terms of the logic, in preparation of pretty printing. There is no distinction for propositions here.

These operations always operate simultaneously on a list; use the combinator `singleton` to apply them to a single item.

Chapter 4

Tactical reasoning

Tactical reasoning works by refining an initial claim in a backwards fashion, until a solved form is reached. A *goal* consists of several subgoals that need to be solved in order to achieve the main statement; zero subgoals means that the proof may be finished. A *tactic* is a refinement operation that maps a goal to a lazy sequence of potential successors. A *tactical* is a combinator for composing tactics.

4.1 Goals

Isabelle/Pure represents a goal as a theorem stating that the subgoals imply the main goal: $A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow C$. The outermost goal structure is that of a Horn Clause: i.e. an iterated implication without any quantifiers[1]. For $n = 0$ a goal is called "solved".

The structure of each subgoal A_i is that of a general Hereditary Harrop Formula $\bigwedge x_1 \ldots \bigwedge x_k.\ H_1 \Longrightarrow \ldots \Longrightarrow H_m \Longrightarrow B$. Here $x_1, \ldots, x_k$ are goal parameters, i.e. arbitrary-but-fixed entities of certain types, and $H_1, \ldots, H_m$ are goal hypotheses, i.e. facts that may be assumed locally. Together, this forms the goal context of the conclusion B to be established. The goal hypotheses may be again arbitrary Hereditary Harrop Formulas, although the level of nesting rarely exceeds 1–2 in practice.

The main conclusion C is internally marked as a protected proposition, which is represented explicitly by the notation $\#C$ here. This ensures that the decomposition into subgoals and main conclusion is well-defined for arbitrarily structured claims.

Basic goal management is performed via the following Isabelle/Pure rules:

$$\frac{}{C \Longrightarrow \#C}\ (\mathit{init}) \qquad \frac{\#C}{C}\ (\mathit{finish})$$

[1] Recall that outermost $\bigwedge x.\ \varphi[x]$ is always represented via schematic variables in the body: $\varphi[?x]$. These variables may get instantiated during the course of reasoning.

The following low-level variants admit general reasoning with protected propositions:

$$\frac{A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow C}{A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow \#C}\ (\mathit{protect}\ n)$$

$$\frac{A \Longrightarrow \ldots \Longrightarrow \#C}{A \Longrightarrow \ldots \Longrightarrow C}\ (\mathit{conclude})$$

ML Reference

```
Goal.init: cterm -> thm
Goal.finish: Proof.context -> thm -> thm
Goal.protect: int -> thm -> thm
Goal.conclude: thm -> thm
```

`Goal.init` *C* initializes a tactical goal from the well-formed proposition *C*.

`Goal.finish` *ctxt thm* checks whether theorem *thm* is a solved goal (no subgoals), and concludes the result by removing the goal protection. The context is only required for printing error messages.

`Goal.protect` *n thm* protects the statement of theorem *thm*. The parameter *n* indicates the number of premises to be retained.

`Goal.conclude` *thm* removes the goal protection, even if there are pending subgoals.

4.2 Tactics

A *tactic* is a function $goal \rightarrow goal^{**}$ that maps a given goal state (represented as a theorem, cf. §4.1) to a lazy sequence of potential successor states. The underlying sequence implementation is lazy both in head and tail, and is purely functional in *not* supporting memoing.[2]

An *empty result sequence* means that the tactic has failed: in a compound tactic expression other tactics might be tried instead, or the whole refinement

[2] The lack of memoing and the strict nature of ML requires some care when working with low-level sequence operations, to avoid duplicate or premature evaluation of results. It also means that modified runtime behavior, such as timeout, is very hard to achieve for general tactics.

step might fail outright, producing a toplevel error message in the end. When implementing tactics from scratch, one should take care to observe the basic protocol of mapping regular error conditions to an empty result; only serious faults should emerge as exceptions.

By enumerating *multiple results*, a tactic can easily express the potential outcome of an internal search process. There are also combinators for building proof tools that involve search systematically, see also §4.3.

As explained before, a goal state essentially consists of a list of subgoals that imply the main goal (conclusion). Tactics may operate on all subgoals or on a particularly specified subgoal, but must not change the main conclusion (apart from instantiating schematic goal variables).

Tactics with explicit *subgoal addressing* are of the form $int \to tactic$ and may be applied to a particular subgoal (counting from 1). If the subgoal number is out of range, the tactic should fail with an empty result sequence, but must not raise an exception!

Operating on a particular subgoal means to replace it by an interval of zero or more subgoals in the same place; other subgoals must not be affected, apart from instantiating schematic variables ranging over the whole goal state.

A common pattern of composing tactics with subgoal addressing is to try the first one, and then the second one only if the subgoal has not been solved yet. Special care is required here to avoid bumping into unrelated subgoals that happen to come after the original subgoal. Assuming that there is only a single initial subgoal is a very common error when implementing tactics!

Tactics with internal subgoal addressing should expose the subgoal index as *int* argument in full generality; a hardwired subgoal 1 is not acceptable.

The main well-formedness conditions for proper tactics are summarized as follows.

- General tactic failure is indicated by an empty result, only serious faults may produce an exception.
- The main conclusion must not be changed, apart from instantiating schematic variables.
- A tactic operates either uniformly on all subgoals, or specifically on a selected subgoal (without bumping into unrelated subgoals).
- Range errors in subgoal addressing produce an empty result.

Some of these conditions are checked by higher-level goal infrastructure (§6.3); others are not checked explicitly, and violating them merely results in ill-behaved tactics experienced by the user (e.g. tactics that insist in being applicable only to singleton goals, or prevent composition via standard tacticals such as REPEAT).

ML Reference

```
type tactic = thm -> thm Seq.seq
no_tac: tactic
all_tac: tactic
print_tac: Proof.context -> string -> tactic
PRIMITIVE: (thm -> thm) -> tactic
SUBGOAL: (term * int -> tactic) -> int -> tactic
CSUBGOAL: (cterm * int -> tactic) -> int -> tactic
SELECT_GOAL: tactic -> int -> tactic
PREFER_GOAL: tactic -> int -> tactic
```

Type `tactic` represents tactics. The well-formedness conditions described above need to be observed. See also `~~/src/Pure/General/seq.ML` for the underlying implementation of lazy sequences.

Type `int -> tactic` represents tactics with explicit subgoal addressing, with well-formedness conditions as described above.

`no_tac` is a tactic that always fails, returning the empty sequence.

`all_tac` is a tactic that always succeeds, returning a singleton sequence with unchanged goal state.

`print_tac` *ctxt message* is like `all_tac`, but prints a message together with the goal state on the tracing channel.

`PRIMITIVE` *rule* turns a primitive inference rule into a tactic with unique result. Exception `THM` is considered a regular tactic failure and produces an empty result; other exceptions are passed through.

`SUBGOAL` (*fn* (*subgoal*, *i*) => *tactic*) is the most basic form to produce a tactic with subgoal addressing. The given abstraction over the subgoal term and subgoal number allows to peek at the relevant information of the full goal state. The subgoal range is checked as required above.

`CSUBGOAL` is similar to `SUBGOAL`, but passes the subgoal as `cterm` instead of raw `term`. This avoids expensive re-certification in situations where the subgoal is used directly for primitive inferences.

`SELECT_GOAL` *tac i* confines a tactic to the specified subgoal *i*. This rearranges subgoals and the main goal protection (§4.1), while retaining the syntactic context of the overall goal state (concerning schematic variables etc.).

`PREFER_GOAL` *tac i* rearranges subgoals to put *i* in front. This is similar to `SELECT_GOAL`, but without changing the main goal protection.

4.2.1 Resolution and assumption tactics

Resolution is the most basic mechanism for refining a subgoal using a theorem as object-level rule. *Elim-resolution* is particularly suited for elimination rules: it resolves with a rule, proves its first premise by assumption, and finally deletes that assumption from any new subgoals. *Destruct-resolution* is like elim-resolution, but the given destruction rules are first turned into canonical elimination format. *Forward-resolution* is like destruct-resolution, but without deleting the selected assumption. The *r*/*e*/*d*/*f* naming convention is maintained for several different kinds of resolution rules and tactics.

Assumption tactics close a subgoal by unifying some of its premises against its conclusion.

All the tactics in this section operate on a subgoal designated by a positive integer. Other subgoals might be affected indirectly, due to instantiation of schematic variables.

There are various sources of non-determinism, the tactic result sequence enumerates all possibilities of the following choices (if applicable):

1. selecting one of the rules given as argument to the tactic;
2. selecting a subgoal premise to eliminate, unifying it against the first premise of the rule;
3. unifying the conclusion of the subgoal to the conclusion of the rule.

Recall that higher-order unification may produce multiple results that are enumerated here.

ML Reference

```
resolve_tac: Proof.context -> thm list -> int -> tactic
eresolve_tac: Proof.context -> thm list -> int -> tactic
dresolve_tac: Proof.context -> thm list -> int -> tactic
forward_tac: Proof.context -> thm list -> int -> tactic
biresolve_tac: Proof.context -> (bool * thm) list -> int -> tactic
assume_tac: Proof.context -> int -> tactic
eq_assume_tac: int -> tactic
match_tac: Proof.context -> thm list -> int -> tactic
ematch_tac: Proof.context -> thm list -> int -> tactic
dmatch_tac: Proof.context -> thm list -> int -> tactic
bimatch_tac: Proof.context -> (bool * thm) list -> int -> tactic
```

`resolve_tac` *ctxt thms i* refines the goal state using the given theorems, which should normally be introduction rules. The tactic resolves a rule's conclusion with subgoal *i*, replacing it by the corresponding versions of the rule's premises.

`eresolve_tac` *ctxt thms i* performs elim-resolution with the given theorems, which are normally be elimination rules.

Note that `eresolve_tac ctxt [asm_rl]` is equivalent to `assume_tac ctxt`, which facilitates mixing of assumption steps with genuine eliminations.

`dresolve_tac` *ctxt thms i* performs destruct-resolution with the given theorems, which should normally be destruction rules. This replaces an assumption by the result of applying one of the rules.

`forward_tac` is like `dresolve_tac` except that the selected assumption is not deleted. It applies a rule to an assumption, adding the result as a new assumption.

`biresolve_tac` *ctxt brls i* refines the proof state by resolution or elim-resolution on each rule, as indicated by its flag. It affects subgoal *i* of the proof state.

For each pair (*flag*, *rule*), it applies resolution if the flag is *false* and elim-resolution if the flag is *true*. A single tactic call handles a mixture of introduction and elimination rules, which is useful to organize the search process systematically in proof tools.

`assume_tac` *ctxt i* attempts to solve subgoal *i* by assumption (modulo higher-order unification).

`eq_assume_tac` is similar to `assume_tac`, but checks only for immediate α-convertibility instead of using unification. It succeeds (with a unique next state) if one of the assumptions is equal to the subgoal's conclusion. Since it does not instantiate variables, it cannot make other subgoals unprovable.

`match_tac`, `ematch_tac`, `dmatch_tac`, and `bimatch_tac` are similar to `resolve_tac`, `eresolve_tac`, `dresolve_tac`, and `biresolve_tac`, respectively, but do not instantiate schematic variables in the goal state.[3] These tactics were written for a specific application within the classical reasoner.

Flexible subgoals are not updated at will, but are left alone.

4.2.2 Explicit instantiation within a subgoal context

The main resolution tactics (§4.2.1) use higher-order unification, which works well in many practical situations despite its daunting theoretical properties. Nonetheless, there are important problem classes where unguided higher-order unification is not so useful. This typically involves rules like universal elimination, existential introduction, or equational substitution. Here the unification problem involves fully flexible *?P ?x* schemes, which are hard to manage without further hints.

By providing a (small) rigid term for *?x* explicitly, the remaining unification problem is to assign a (large) term to *?P*, according to the shape of the given subgoal. This is sufficiently well-behaved in most practical situations.

Isabelle provides separate versions of the standard $r/e/d/f$ resolution tactics that allow to provide explicit instantiations of unknowns of the given rule, wrt. terms that refer to the implicit context of the selected subgoal.

An instantiation consists of a list of pairs of the form (*?x*, t), where *?x* is a schematic variable occurring in the given rule, and t is a term from the current proof context, augmented by the local goal parameters of the selected subgoal; cf. the *focus* operation described in §6.1.

Entering the syntactic context of a subgoal is a brittle operation, because its exact form is somewhat accidental, and the choice of bound variable names

[3] Strictly speaking, matching means to treat the unknowns in the goal state as constants, but these tactics merely discard unifiers that would update the goal state. In rare situations (where the conclusion and goal state have flexible terms at the same position), the tactic will fail even though an acceptable unifier exists.

depends on the presence of other local and global names. Explicit renaming of subgoal parameters prior to explicit instantiation might help to achieve a bit more robustness.

Type instantiations may be given as well, via pairs like $(?'a, \tau)$. Type instantiations are distinguished from term instantiations by the syntactic form of the schematic variable. Types are instantiated before terms are. Since term instantiation already performs simple type-inference, so explicit type instantiations are seldom necessary.

ML Reference

```
Rule_Insts.res_inst_tac: Proof.context ->
    ((indexname * Position.T) * string) list -> (binding * string option * mixfix) list ->
    thm -> int -> tactic
Rule_Insts.eres_inst_tac: Proof.context ->
    ((indexname * Position.T) * string) list -> (binding * string option * mixfix) list ->
    thm -> int -> tactic
Rule_Insts.dres_inst_tac: Proof.context ->
    ((indexname * Position.T) * string) list -> (binding * string option * mixfix) list ->
    thm -> int -> tactic
Rule_Insts.forw_inst_tac: Proof.context ->
    ((indexname * Position.T) * string) list -> (binding * string option * mixfix) list ->
    thm -> int -> tactic
Rule_Insts.subgoal_tac: Proof.context -> string ->
    (binding * string option * mixfix) list -> int -> tactic
Rule_Insts.thin_tac: Proof.context -> string ->
    (binding * string option * mixfix) list -> int -> tactic
rename_tac: string list -> int -> tactic
```

`Rule_Insts.res_inst_tac` *ctxt insts thm i* instantiates the rule *thm* with the instantiations *insts*, as described above, and then performs resolution on subgoal *i*.

`Rule_Insts.eres_inst_tac` is like `Rule_Insts.res_inst_tac`, but performs elim-resolution.

`Rule_Insts.dres_inst_tac` is like `Rule_Insts.res_inst_tac`, but performs destruct-resolution.

`Rule_Insts.forw_inst_tac` is like `Rule_Insts.dres_inst_tac` except that the selected assumption is not deleted.

`Rule_Insts.subgoal_tac` *ctxt* φ *i* adds the proposition φ as local premise to subgoal *i*, and poses the same as a new subgoal $i + 1$ (in the original context).

`Rule_Insts.thin_tac` *ctxt* φ *i* deletes the specified premise from subgoal *i*. Note that φ may contain schematic variables, to abbreviate the intended proposition; the first matching subgoal premise will be deleted. Removing useless premises from a subgoal increases its readability and can make search tactics run faster.

`rename_tac` *names i* renames the innermost parameters of subgoal *i* according to the provided *names* (which need to be distinct identifiers).

For historical reasons, the above instantiation tactics take unparsed string arguments, which makes them hard to use in general ML code. The slightly more advanced `Subgoal.FOCUS` combinator of §6.3 allows to refer to internal goal structure with explicit context management.

4.2.3 Rearranging goal states

In rare situations there is a need to rearrange goal states: either the overall collection of subgoals, or the local structure of a subgoal. Various administrative tactics allow to operate on the concrete presentation these conceptual sets of formulae.

ML Reference

```
rotate_tac: int -> int -> tactic
distinct_subgoals_tac: tactic
flexflex_tac: Proof.context -> tactic
```

`rotate_tac` *n i* rotates the premises of subgoal *i* by *n* positions: from right to left if *n* is positive, and from left to right if *n* is negative.

`distinct_subgoals_tac` removes duplicate subgoals from a proof state. This is potentially inefficient.

`flexflex_tac` removes all flex-flex pairs from the proof state by applying the trivial unifier. This drastic step loses information. It is already part of the Isar infrastructure for facts resulting from goals, and rarely needs to be invoked manually.

Flex-flex constraints arise from difficult cases of higher-order unification. To prevent this, use `Rule_Insts.res_inst_tac` to instantiate some variables in a rule. Normally flex-flex constraints can be ignored; they often disappear as unknowns get instantiated.

4.2.4 Raw composition: resolution without lifting

Raw composition of two rules means resolving them without prior lifting or renaming of unknowns. This low-level operation, which underlies the resolution tactics, may occasionally be useful for special effects. Schematic variables are not renamed by default, so beware of clashes!

ML **Reference**

```
compose_tac: Proof.context -> (bool * thm * int) -> int -> tactic
Drule.compose: thm * int * thm -> thm
infix COMP: thm * thm -> thm
```

`compose_tac` *ctxt* (*flag*, *rule*, *m*) *i* refines subgoal *i* using *rule*, without lifting. The *rule* is taken to have the form $\psi_1 \Longrightarrow \ldots \psi_m \Longrightarrow \psi$, where ψ need not be atomic; thus *m* determines the number of new subgoals. If *flag* is *true* then it performs elim-resolution — it solves the first premise of *rule* by assumption and deletes that assumption.

`Drule.compose` (thm_1, *i*, thm_2) uses thm_1, regarded as an atomic formula, to solve premise *i* of thm_2. Let thm_1 and thm_2 be ψ and $\varphi_1 \Longrightarrow \ldots \varphi_n \Longrightarrow \varphi$. The unique *s* that unifies ψ and φ_i yields the theorem $(\varphi_1 \Longrightarrow \ldots \varphi_{i-1} \Longrightarrow \varphi_{i+1} \Longrightarrow \ldots \varphi_n \Longrightarrow \varphi)s$. Multiple results are considered as error (exception `THM`).

thm_1 *COMP* thm_2 is the same as *Drule.compose* (thm_1, 1, thm_2).

! These low-level operations are stepping outside the structure imposed by regular rule resolution. Used without understanding of the consequences, they may produce results that cause problems with standard rules and tactics later on.

4.3 Tacticals

A *tactical* is a functional combinator for building up complex tactics from simpler ones. Common tacticals perform sequential composition, disjunctive choice, iteration, or goal addressing. Various search strategies may be expressed via tacticals.

4.3.1 Combining tactics

Sequential composition and alternative choices are the most basic ways to combine tactics, similarly to "`,`" and "`|`" in Isar method notation. This corresponds to `THEN` and `ORELSE` in ML, but there are further possibilities for fine-tuning alternation of tactics such as `APPEND`. Further details become visible in ML due to explicit subgoal addressing.

ML Reference

```
infix THEN: tactic * tactic -> tactic
infix ORELSE: tactic * tactic -> tactic
infix APPEND: tactic * tactic -> tactic
EVERY: tactic list -> tactic
FIRST: tactic list -> tactic
infix THEN': ('a -> tactic) * ('a -> tactic) -> 'a -> tactic
infix ORELSE': ('a -> tactic) * ('a -> tactic) -> 'a -> tactic
infix APPEND': ('a -> tactic) * ('a -> tactic) -> 'a -> tactic
EVERY': ('a -> tactic) list -> 'a -> tactic
FIRST': ('a -> tactic) list -> 'a -> tactic
```

tac_1 `THEN` tac_2 is the sequential composition of tac_1 and tac_2. Applied to a goal state, it returns all states reachable in two steps by applying tac_1 followed by tac_2. First, it applies tac_1 to the goal state, getting a sequence of possible next states; then, it applies tac_2 to each of these and concatenates the results to produce again one flat sequence of states.

tac_1 `ORELSE` tac_2 makes a choice between tac_1 and tac_2. Applied to a state, it tries tac_1 and returns the result if non-empty; if tac_1 fails then it uses tac_2. This is a deterministic choice: if tac_1 succeeds then tac_2 is excluded from the result.

tac_1 `APPEND` tac_2 concatenates the possible results of tac_1 and tac_2. Unlike `ORELSE` there is *no commitment* to either tactic, so `APPEND` helps to avoid incompleteness during search, at the cost of potential inefficiencies.

`EVERY` $[tac_1, \ldots, tac_n]$ abbreviates tac_1 `THEN` ... `THEN` tac_n. Note that `EVERY []` is the same as `all_tac`: it always succeeds.

`FIRST` $[tac_1, \ldots, tac_n]$ abbreviates tac_1 `ORELSE` ... `ORELSE` tac_n. Note that `FIRST []` is the same as `no_tac`: it always fails.

`THEN'` is the lifted version of `THEN`, for tactics with explicit subgoal addressing. So (tac_1 `THEN'` tac_2) i is the same as (tac_1 i `THEN` tac_2 i).

The other primed tacticals work analogously.

4.3.2 Repetition tacticals

These tacticals provide further control over repetition of tactics, beyond the stylized forms of "`?`" and "`+`" in Isar method expressions.

ML Reference

```
TRY: tactic -> tactic
REPEAT: tactic -> tactic
REPEAT1: tactic -> tactic
REPEAT_DETERM: tactic -> tactic
REPEAT_DETERM_N: int -> tactic -> tactic
```

`TRY` *tac* applies *tac* to the goal state and returns the resulting sequence, if non-empty; otherwise it returns the original state. Thus, it applies *tac* at most once.

Note that for tactics with subgoal addressing, the combinator can be applied via functional composition: `TRY o` *tac*. There is no need for `TRY'`.

`REPEAT` *tac* applies *tac* to the goal state and, recursively, to each element of the resulting sequence. The resulting sequence consists of those states that make *tac* fail. Thus, it applies *tac* as many times as possible (including zero times), and allows backtracking over each invocation of *tac*. `REPEAT` is more general than `REPEAT_DETERM`, but requires more space.

`REPEAT1` *tac* is like `REPEAT` *tac* but it always applies *tac* at least once, failing if this is impossible.

`REPEAT_DETERM` *tac* applies *tac* to the goal state and, recursively, to the head of the resulting sequence. It returns the first state to make *tac* fail. It is deterministic, discarding alternative outcomes.

`REPEAT_DETERM_N` n *tac* is like `REPEAT_DETERM` *tac* but the number of repetitions is bound by n (where `~1` means ∞).

ML Examples

The basic tactics and tacticals considered above follow some algebraic laws:

- `all_tac` is the identity element of the tactical `THEN`.
- `no_tac` is the identity element of `ORELSE` and `APPEND`. Also, it is a zero element for `THEN`, which means that *tac* `THEN` `no_tac` is equivalent to `no_tac`.
- `TRY` and `REPEAT` can be expressed as (recursive) functions over more basic combinators (ignoring some internal implementation tricks):

ML ‹

```
  fun TRY tac = tac ORELSE all_tac;
  fun REPEAT tac st = ((tac THEN REPEAT tac) ORELSE all_tac) st;
```

›

If *tac* can return multiple outcomes then so can `REPEAT` *tac*. `REPEAT` uses `ORELSE` and not `APPEND`, it applies *tac* as many times as possible in each outcome.

! Note the explicit abstraction over the goal state in the ML definition of `REPEAT`. Recursive tacticals must be coded in this awkward fashion to avoid infinite recursion of eager functional evaluation in Standard ML. The following attempt would make `REPEAT` *tac* loop:

ML_val ‹

```
  (*BAD -- does not terminate!*)
  fun REPEAT tac = (tac THEN REPEAT tac) ORELSE all_tac;
```

›

4.3.3 Applying tactics to subgoal ranges

Tactics with explicit subgoal addressing `int -> tactic` can be used together with tacticals that act like "subgoal quantifiers": guided by success of the body tactic a certain range of subgoals is covered. Thus the body tactic is applied to *all* subgoals, *some* subgoal etc.

Suppose that the goal state has $n \geq 0$ subgoals. Many of these tacticals address subgoal ranges counting downwards from n towards 1. This has the fortunate effect that newly emerging subgoals are concatenated in the result, without interfering each other. Nonetheless, there might be situations where a different order is desired.

ML Reference

```
ALLGOALS: (int -> tactic) -> tactic
SOMEGOAL: (int -> tactic) -> tactic
FIRSTGOAL: (int -> tactic) -> tactic
HEADGOAL: (int -> tactic) -> tactic
REPEAT_SOME: (int -> tactic) -> tactic
REPEAT_FIRST: (int -> tactic) -> tactic
RANGE: (int -> tactic) list -> int -> tactic
```

`ALLGOALS` *tac* is equivalent to *tac n* `THEN` ... `THEN` *tac* 1. It applies the *tac* to all the subgoals, counting downwards.

`SOMEGOAL` *tac* is equivalent to *tac n* `ORELSE` ... `ORELSE` *tac* 1. It applies *tac* to one subgoal, counting downwards.

`FIRSTGOAL` *tac* is equivalent to *tac* 1 `ORELSE` ... `ORELSE` *tac n*. It applies *tac* to one subgoal, counting upwards.

`HEADGOAL` *tac* is equivalent to *tac* 1. It applies *tac* unconditionally to the first subgoal.

`REPEAT_SOME` *tac* applies *tac* once or more to a subgoal, counting downwards.

`REPEAT_FIRST` *tac* applies *tac* once or more to a subgoal, counting upwards.

`RANGE` [tac_1, ..., tac_k] i is equivalent to tac_k $(i + k - 1)$ `THEN` ... `THEN` tac_1 i. It applies the given list of tactics to the corresponding range of subgoals, counting downwards.

4.3.4 Control and search tacticals

A predicate on theorems `thm -> bool` can test whether a goal state enjoys some desirable property — such as having no subgoals. Tactics that search for satisfactory goal states are easy to express. The main search procedures, depth-first, breadth-first and best-first, are provided as tacticals. They generate the search tree by repeatedly applying a given tactic.

ML Reference

Filtering a tactic's results

```
FILTER: (thm -> bool) -> tactic -> tactic
CHANGED: tactic -> tactic
```

`FILTER` *sat tac* applies *tac* to the goal state and returns a sequence consisting of those result goal states that are satisfactory in the sense of *sat*.

`CHANGED` *tac* applies *tac* to the goal state and returns precisely those states that differ from the original state (according to `Thm.eq_thm`). Thus `CHANGED` *tac* always has some effect on the state.

Depth-first search

```
DEPTH_FIRST: (thm -> bool) -> tactic -> tactic
DEPTH_SOLVE: tactic -> tactic
DEPTH_SOLVE_1: tactic -> tactic
```

`DEPTH_FIRST` *sat tac* returns the goal state if *sat* returns true. Otherwise it applies *tac*, then recursively searches from each element of the resulting sequence. The code uses a stack for efficiency, in effect applying *tac* `THEN` `DEPTH_FIRST` *sat tac* to the state.

`DEPTH_SOLVE`*tac* uses `DEPTH_FIRST` to search for states having no subgoals.

`DEPTH_SOLVE_1` *tac* uses `DEPTH_FIRST` to search for states having fewer subgoals than the given state. Thus, it insists upon solving at least one subgoal.

Other search strategies

```
BREADTH_FIRST: (thm -> bool) -> tactic -> tactic
BEST_FIRST: (thm -> bool) * (thm -> int) -> tactic -> tactic
THEN_BEST_FIRST: tactic -> (thm -> bool) * (thm -> int) -> tactic -> tactic
```

These search strategies will find a solution if one exists. However, they do not enumerate all solutions; they terminate after the first satisfactory result from *tac*.

`BREADTH_FIRST` *sat tac* uses breadth-first search to find states for which *sat* is true. For most applications, it is too slow.

BEST_FIRST (*sat*, *dist*) *tac* does a heuristic search, using *dist* to estimate the distance from a satisfactory state (in the sense of *sat*). It maintains a list of states ordered by distance. It applies *tac* to the head of this list; if the result contains any satisfactory states, then it returns them. Otherwise, BEST_FIRST adds the new states to the list, and continues.

The distance function is typically size_of_thm, which computes the size of the state. The smaller the state, the fewer and simpler subgoals it has.

THEN_BEST_FIRST tac_0 (*sat*, *dist*) *tac* is like BEST_FIRST, except that the priority queue initially contains the result of applying tac_0 to the goal state. This tactical permits separate tactics for starting the search and continuing the search.

Auxiliary tacticals for searching

```
COND: (thm -> bool) -> tactic -> tactic -> tactic
IF_UNSOLVED: tactic -> tactic
SOLVE: tactic -> tactic
DETERM: tactic -> tactic
```

COND *sat* tac_1 tac_2 applies tac_1 to the goal state if it satisfies predicate *sat*, and applies tac_2. It is a conditional tactical in that only one of tac_1 and tac_2 is applied to a goal state. However, both tac_1 and tac_2 are evaluated because ML uses eager evaluation.

IF_UNSOLVED *tac* applies *tac* to the goal state if it has any subgoals, and simply returns the goal state otherwise. Many common tactics, such as resolve_tac, fail if applied to a goal state that has no subgoals.

SOLVE *tac* applies *tac* to the goal state and then fails iff there are subgoals left.

DETERM *tac* applies *tac* to the goal state and returns the head of the resulting sequence. DETERM limits the search space by making its argument deterministic.

Predicates and functions useful for searching

```
has_fewer_prems: int -> thm -> bool
Thm.eq_thm: thm * thm -> bool
Thm.eq_thm_prop: thm * thm -> bool
size_of_thm: thm -> int
```

`has_fewer_prems` *n thm* reports whether *thm* has fewer than *n* premises.

`Thm.eq_thm` (thm_1, thm_2) reports whether thm_1 and thm_2 are equal. Both theorems must have the same conclusions, the same set of hypotheses, and the same set of sort hypotheses. Names of bound variables are ignored as usual.

`Thm.eq_thm_prop` (thm_1, thm_2) reports whether the propositions of thm_1 and thm_2 are equal. Names of bound variables are ignored.

`size_of_thm` *thm* computes the size of *thm*, namely the number of variables, constants and abstractions in its conclusion. It may serve as a distance function for `BEST_FIRST`.

Chapter 5

Equational reasoning

Equality is one of the most fundamental concepts of mathematics. The Isabelle/Pure logic (chapter 2) provides a builtin relation $\equiv :: \alpha \Rightarrow \alpha \Rightarrow prop$ that expresses equality of arbitrary terms (or propositions) at the framework level, as expressed by certain basic inference rules (§5.1).

Equational reasoning means to replace equals by equals, using reflexivity and transitivity to form chains of replacement steps, and congruence rules to access sub-structures. Conversions (§5.2) provide a convenient framework to compose basic equational steps to build specific equational reasoning tools.

Higher-order matching is able to provide suitable instantiations for giving equality rules, which leads to the versatile concept of λ-term rewriting (§5.3). Internally this is based on the general-purpose Simplifier engine of Isabelle, which is more specific and more efficient than plain conversions.

Object-logics usually introduce specific notions of equality or equivalence, and relate it with the Pure equality. This enables to re-use the Pure tools for equational reasoning for particular object-logic connectives as well.

5.1 Basic equality rules

Isabelle/Pure uses $\equiv$ for equality of arbitrary terms, which includes equivalence of propositions of the logical framework. The conceptual axiomatization of the constant $\equiv :: \alpha \Rightarrow \alpha \Rightarrow prop$ is given in figure 2.3. The inference kernel presents slightly different equality rules, which may be understood as derived rules from this minimal axiomatization. The Pure theory also provides some theorems that express the same reasoning schemes as theorems that can be composed like object-level rules as explained in §2.4.

For example, `Thm.symmetric` as Pure inference is an ML function that maps a theorem *th* stating $t \equiv u$ to one stating $u \equiv t$. In contrast, *Pure.symmetric* as Pure theorem expresses the same reasoning in declarative form. If used like *th* [*THEN Pure.symmetric*] in Isar source notation, it achieves a similar effect as the ML inference function, although the rule attribute *THEN* or

ML operator `op RS` involve the full machinery of higher-order unification (modulo $\beta\eta$-conversion) and lifting of $\bigwedge/\Longrightarrow$ contexts.

ML Reference

```
Thm.reflexive: cterm -> thm
Thm.symmetric: thm -> thm
Thm.transitive: thm -> thm -> thm
Thm.abstract_rule: string -> cterm -> thm -> thm
Thm.combination: thm -> thm -> thm
Thm.equal_intr: thm -> thm -> thm
Thm.equal_elim: thm -> thm -> thm
```

See also `~~/src/Pure/thm.ML` for further description of these inference rules, and a few more for primitive β and η conversions. Note that α conversion is implicit due to the representation of terms with de-Bruijn indices (§2.2).

5.2 Conversions

The classic article [13] introduces the concept of conversion for Cambridge LCF. This was historically important to implement all kinds of "simplifiers", but the Isabelle Simplifier is done quite differently, see [19, §9.3].

ML Reference

```
structure Conv
type conv
Simplifier.asm_full_rewrite: Proof.context -> conv
```

`Conv` is a library of combinators to build conversions, over type `conv` (see also `~~/src/Pure/conv.ML`). This is one of the few Isabelle/ML modules that are usually used with `open`: finding examples works by searching for "`open Conv`" instead of "`Conv.`".

`Simplifier.asm_full_rewrite` invokes the Simplifier as a conversion. There are a few related operations, corresponding to the various modes of simplification.

5.3 Rewriting

Rewriting normalizes a given term (theorem or goal) by replacing instances of given equalities $t \equiv u$ in subterms. Rewriting continues until no rewrites are applicable to any subterm. This may be used to unfold simple definitions of the form $f\, x_1\, \ldots\, x_n \equiv u$, but is slightly more general than that.

ML Reference

```
rewrite_rule: Proof.context -> thm list -> thm -> thm
rewrite_goals_rule: Proof.context -> thm list -> thm -> thm
rewrite_goal_tac: Proof.context -> thm list -> int -> tactic
rewrite_goals_tac: Proof.context -> thm list -> tactic
fold_goals_tac: Proof.context -> thm list -> tactic
```

`rewrite_rule` *ctxt rules thm* rewrites the whole theorem by the given rules.

`rewrite_goals_rule` *ctxt rules thm* rewrites the outer premises of the given theorem. Interpreting the same as a goal state (§4.1) it means to rewrite all subgoals (in the same manner as `rewrite_goals_tac`).

`rewrite_goal_tac` *ctxt rules i* rewrites subgoal *i* by the given rewrite rules.

`rewrite_goals_tac` *ctxt rules* rewrites all subgoals by the given rewrite rules.

`fold_goals_tac` *ctxt rules* essentially uses `rewrite_goals_tac` with the symmetric form of each member of *rules*, re-ordered to fold longer expression first. This supports to idea to fold primitive definitions that appear in expended form in the proof state.

Chapter 6

Structured proofs

6.1 Variables

Any variable that is not explicitly bound by λ-abstraction is considered as "free". Logically, free variables act like outermost universal quantification at the sequent level: $A_1(x)$, …, $A_n(x) \vdash B(x)$ means that the result holds *for all* values of x. Free variables for terms (not types) can be fully internalized into the logic: $\vdash B(x)$ and $\vdash \bigwedge x.\ B(x)$ are interchangeable, provided that x does not occur elsewhere in the context. Inspecting $\vdash \bigwedge x.\ B(x)$ more closely, we see that inside the quantifier, x is essentially "arbitrary, but fixed", while from outside it appears as a place-holder for instantiation (thanks to $\bigwedge$ elimination).

The Pure logic represents the idea of variables being either inside or outside the current scope by providing separate syntactic categories for *fixed variables* (e.g. x) vs. *schematic variables* (e.g. $?x$). Incidently, a universal result $\vdash \bigwedge x.\ B(x)$ has the HHF normal form $\vdash B(?x)$, which represents its generality without requiring an explicit quantifier. The same principle works for type variables: $\vdash B(?\alpha)$ represents the idea of "$\vdash \forall\alpha.\ B(\alpha)$" without demanding a truly polymorphic framework.

Additional care is required to treat type variables in a way that facilitates type-inference. In principle, term variables depend on type variables, which means that type variables would have to be declared first. For example, a raw type-theoretic framework would demand the context to be constructed in stages as follows: $\Gamma = \alpha\colon \mathit{type},\ x\colon \alpha,\ a\colon A(x_\alpha)$.

We allow a slightly less formalistic mode of operation: term variables x are fixed without specifying a type yet (essentially *all* potential occurrences of some instance x_τ are fixed); the first occurrence of x within a specific term assigns its most general type, which is then maintained consistently in the context. The above example becomes $\Gamma = x\colon \mathit{term},\ \alpha\colon \mathit{type},\ A(x_\alpha)$, where type α is fixed *after* term x, and the constraint $x :: \alpha$ is an implicit consequence of the occurrence of x_α in the subsequent proposition.

This twist of dependencies is also accommodated by the reverse operation of exporting results from a context: a type variable α is considered fixed as long as it occurs in some fixed term variable of the context. For example, exporting x: *term*, α: *type* $\vdash x_\alpha \equiv x_\alpha$ produces in the first step x: *term* $\vdash x_\alpha \equiv x_\alpha$ for fixed α, and only in the second step $\vdash ?x_{?\alpha} \equiv ?x_{?\alpha}$ for schematic $?x$ and $?\alpha$. The following Isar source text illustrates this scenario.

```
notepad
begin
  {
    fix x  — all potential occurrences of some x::τ are fixed
    {
      have x::'a ≡ x  — implicit type assignment by concrete occurrence
        by (rule reflexive)
    }
    thm this  — result still with fixed type 'a
  }
  thm this  — fully general result for arbitrary ?x::?'a
end
```

The Isabelle/Isar proof context manages the details of term vs. type variables, with high-level principles for moving the frontier between fixed and schematic variables.

The *add_fixes* operation explicitly declares fixed variables; the *declare_term* operation absorbs a term into a context by fixing new type variables and adding syntactic constraints.

The *export* operation is able to perform the main work of generalizing term and type variables as sketched above, assuming that fixing variables and terms have been declared properly.

There *import* operation makes a generalized fact a genuine part of the context, by inventing fixed variables for the schematic ones. The effect can be reversed by using *export* later, potentially with an extended context; the result is equivalent to the original modulo renaming of schematic variables.

The *focus* operation provides a variant of *import* for nested propositions (with explicit quantification): $\bigwedge x_1 \ldots x_n.\ B(x_1, \ldots, x_n)$ is decomposed by inventing fixed variables $x_1, \ldots, x_n$ for the body.

ML Reference

```
Variable.add_fixes:
  string list -> Proof.context -> string list * Proof.context
Variable.variant_fixes:
  string list -> Proof.context -> string list * Proof.context
Variable.declare_term: term -> Proof.context -> Proof.context
Variable.declare_constraints: term -> Proof.context -> Proof.context
Variable.export: Proof.context -> Proof.context -> thm list -> thm list
Variable.polymorphic: Proof.context -> term list -> term list
Variable.import: bool -> thm list -> Proof.context ->
  ((ctyp TVars.table * cterm Vars.table) * thm list)
    * Proof.context
Variable.focus: binding list option -> term -> Proof.context ->
  ((string * (string * typ)) list * term) * Proof.context
```

`Variable.add_fixes` *xs ctxt* fixes term variables *xs*, returning the resulting internal names. By default, the internal representation coincides with the external one, which also means that the given variables must not be fixed already. There is a different policy within a local proof body: the given names are just hints for newly invented Skolem variables.

`Variable.variant_fixes` is similar to `Variable.add_fixes`, but always produces fresh variants of the given names.

`Variable.declare_term` *t ctxt* declares term *t* to belong to the context. This automatically fixes new type variables, but not term variables. Syntactic constraints for type and term variables are declared uniformly, though.

`Variable.declare_constraints` *t ctxt* declares syntactic constraints from term *t*, without making it part of the context yet.

`Variable.export` *inner outer thms* generalizes fixed type and term variables in *thms* according to the difference of the *inner* and *outer* context, following the principles sketched above.

`Variable.polymorphic` *ctxt ts* generalizes type variables in *ts* as far as possible, even those occurring in fixed term variables. The default policy of type-inference is to fix newly introduced type variables, which is essentially reversed with `Variable.polymorphic`: here the given terms are detached from the context as far as possible.

`Variable.import` *open thms ctxt* invents fixed type and term variables for the schematic ones occurring in *thms*. The *open* flag indicates whether the fixed names should be accessible to the user, otherwise newly introduced names are marked as “internal” (§1.2).

`Variable.focus` *bindings B* decomposes the outermost $\bigwedge$ prefix of proposition B, using the given name bindings.

ML Examples

The following example shows how to work with fixed term and type parameters and with type-inference.

```
ML_val ‹
  (*static compile-time context -- for testing only*)
  val ctxt0 = context;

  (*locally fixed parameters -- no type assignment yet*)
  val ([x, y], ctxt1) = ctxt0 |> Variable.add_fixes ["x", "y"];

  (*t1: most general fixed type; t1’: most general arbitrary type*)
  val t1 = Syntax.read_term ctxt1 "x";
  val t1’ = singleton (Variable.polymorphic ctxt1) t1;

  (*term u enforces specific type assignment*)
  val u = Syntax.read_term ctxt1 "(x::nat) ≡ y";

  (*official declaration of u -- propagates constraints etc.*)
  val ctxt2 = ctxt1 |> Variable.declare_term u;
  val t2 = Syntax.read_term ctxt2 "x";  (*x::nat is enforced*)
›
```

In the above example, the starting context is derived from the toplevel theory, which means that fixed variables are internalized literally: x is mapped again to x, and attempting to fix it again in the subsequent context is an error. Alternatively, fixed parameters can be renamed explicitly as follows:

```
ML_val ‹
  val ctxt0 = context;
  val ([x1, x2, x3], ctxt1) =
    ctxt0 |> Variable.variant_fixes ["x", "x", "x"];
›
```

The following ML code can now work with the invented names of $x1$, $x2$, $x3$, without depending on the details on the system policy for introducing these variants. Recall that within a proof body the system always invents fresh “Skolem constants”, e.g. as follows:

```
notepad
begin
 ML_prf
  ‹val ctxt0 = context;

   val ([x1], ctxt1) = ctxt0 |> Variable.add_fixes ["x"];
   val ([x2], ctxt2) = ctxt1 |> Variable.add_fixes ["x"];
   val ([x3], ctxt3) = ctxt2 |> Variable.add_fixes ["x"];

   val ([y1, y2], ctxt4) =
     ctxt3 |> Variable.variant_fixes ["y", "y"];›
end
```

In this situation `Variable.add_fixes` and `Variable.variant_fixes` are very similar, but identical name proposals given in a row are only accepted by the second version.

6.2 Assumptions

An *assumption* is a proposition that it is postulated in the current context. Local conclusions may use assumptions as additional facts, but this imposes implicit hypotheses that weaken the overall statement.

Assumptions are restricted to fixed non-schematic statements, i.e. all generality needs to be expressed by explicit quantifiers. Nevertheless, the result will be in HHF normal form with outermost quantifiers stripped. For example, by assuming $\bigwedge x :: \alpha.\ P\ x$ we get $\bigwedge x :: \alpha.\ P\ x \vdash P\ ?x$ for schematic $?x$ of fixed type α. Local derivations accumulate more and more explicit references to hypotheses: $A_1, \ldots, A_n \vdash B$ where $A_1, \ldots, A_n$ needs to be covered by the assumptions of the current context.

The *add_assms* operation augments the context by local assumptions, which are parameterized by an arbitrary *export* rule (see below).

The *export* operation moves facts from a (larger) inner context into a (smaller) outer context, by discharging the difference of the assumptions as specified by the associated export rules. Note that the discharged portion is determined by the difference of contexts, not the facts being exported! There is a separate flag to indicate a goal context, where the result is meant to refine an enclosing sub-goal of a structured proof state.

The most basic export rule discharges assumptions directly by means of the

$\Longrightarrow$ introduction rule:

$$\frac{\Gamma \vdash B}{\Gamma - A \vdash A \Longrightarrow B}\ (\Longrightarrow\text{-}\mathit{intro})$$

The variant for goal refinements marks the newly introduced premises, which causes the canonical Isar goal refinement scheme to enforce unification with local premises within the goal:

$$\frac{\Gamma \vdash B}{\Gamma - A \vdash \#A \Longrightarrow B}\ (\#\Longrightarrow\text{-}\mathit{intro})$$

Alternative versions of assumptions may perform arbitrary transformations on export, as long as the corresponding portion of hypotheses is removed from the given facts. For example, a local definition works by fixing x and assuming $x \equiv t$, with the following export rule to reverse the effect:

$$\frac{\Gamma \vdash B\ x}{\Gamma - (x \equiv t) \vdash B\ t}\ (\equiv\text{-}\mathit{expand})$$

This works, because the assumption $x \equiv t$ was introduced in a context with x being fresh, so x does not occur in Γ here.

ML Reference

```
type Assumption.export
Assumption.assume: Proof.context -> cterm -> thm
Assumption.add_assms: Assumption.export ->
  cterm list -> Proof.context -> thm list * Proof.context
Assumption.add_assumes:
  cterm list -> Proof.context -> thm list * Proof.context
Assumption.export: bool -> Proof.context -> Proof.context -> thm -> thm
```

Type `Assumption.export` represents arbitrary export rules, which is any function of type `bool -> cterm list -> thm -> thm`, where the `bool` indicates goal mode, and the `cterm list` the collection of assumptions to be discharged simultaneously.

`Assumption.assume` *ctxt* A turns proposition A into a primitive assumption $A \vdash A'$, where the conclusion A' is in HHF normal form.

`Assumption.add_assms` *r As* augments the context by assumptions *As* with export rule *r*. The resulting facts are hypothetical theorems as produced by the raw `Assumption.assume`.

`Assumption.add_assumes` *As* is a special case of `Assumption.add_assms` where the export rule performs $\Longrightarrow$-*intro* or $\#\Longrightarrow$-*intro*, depending on goal mode.

`Assumption.export` *is_goal inner outer thm* exports result *thm* from the the *inner* context back into the *outer* one; *is_goal* = *true* means this is a goal context. The result is in HHF normal form. Note that `Proof_Context.export` combines `Variable.export` and `Assumption.export` in the canonical way.

ML Examples

The following example demonstrates how rules can be derived by building up a context of assumptions first, and exporting some local fact afterwards. We refer to *Pure* equality here for testing purposes.

```
ML_val ‹
  (*static compile-time context -- for testing only*)
  val ctxt0 = context;

  val ([eq], ctxt1) =
    ctxt0 |> Assumption.add_assumes [cprop ‹x ≡ y›];
  val eq' = Thm.symmetric eq;

  (*back to original context -- discharges assumption*)
  val r = Assumption.export false ctxt1 ctxt0 eq';
›
```

Note that the variables of the resulting rule are not generalized. This would have required to fix them properly in the context beforehand, and export wrt. variables afterwards (cf. `Variable.export` or the combined `Proof_Context.export`).

6.3 Structured goals and results

Local results are established by monotonic reasoning from facts within a context. This allows common combinations of theorems, e.g. via $\bigwedge/\Longrightarrow$ elimination, resolution rules, or equational reasoning, see §2.3. Unaccounted context manipulations should be avoided, notably raw $\bigwedge/\Longrightarrow$ introduction or ad-hoc references to free variables or assumptions not present in the proof context.

The *SUBPROOF* combinator allows to structure a tactical proof recursively by decomposing a selected sub-goal: $(\bigwedge x.\ A(x) \Longrightarrow B(x)) \Longrightarrow \ldots$ is turned into $B(x) \Longrightarrow \ldots$ after fixing x and assuming $A(x)$. This means the tactic needs to solve the conclusion, but may use the premise as a local fact, for locally fixed variables.

The family of *FOCUS* combinators is similar to *SUBPROOF*, but allows to retain schematic variables and pending subgoals in the resulting goal state.

The *prove* operation provides an interface for structured backwards reasoning under program control, with some explicit sanity checks of the result. The goal context can be augmented by additional fixed variables (cf. §6.1) and assumptions (cf. §6.2), which will be available as local facts during the proof and discharged into implications in the result. Type and term variables are generalized as usual, according to the context.

The *obtain* operation produces results by eliminating existing facts by means of a given tactic. This acts like a dual conclusion: the proof demonstrates that the context may be augmented by parameters and assumptions, without affecting any conclusions that do not mention these parameters. See also [19] for the corresponding Isar proof command **obtain**. Final results, which may not refer to the parameters in the conclusion, need to exported explicitly into the original context.

ML Reference

```
SUBPROOF: (Subgoal.focus -> tactic) ->
  Proof.context -> int -> tactic
Subgoal.FOCUS: (Subgoal.focus -> tactic) ->
  Proof.context -> int -> tactic
Subgoal.FOCUS_PREMS: (Subgoal.focus -> tactic) ->
  Proof.context -> int -> tactic
Subgoal.FOCUS_PARAMS: (Subgoal.focus -> tactic) ->
  Proof.context -> int -> tactic
Subgoal.focus: Proof.context -> int -> binding list option ->
  thm -> Subgoal.focus * thm
Subgoal.focus_prems: Proof.context -> int -> binding list option ->
  thm -> Subgoal.focus * thm
Subgoal.focus_params: Proof.context -> int -> binding list option ->
  thm -> Subgoal.focus * thm
```

```
Goal.prove: Proof.context -> string list -> term list -> term ->
  ({prems: thm list, context: Proof.context} -> tactic) -> thm
Goal.prove_common: Proof.context -> int option ->
  string list -> term list -> term list ->
  ({prems: thm list, context: Proof.context} -> tactic) -> thm list
```

```
Obtain.result: (Proof.context -> tactic) -> thm list ->
  Proof.context -> ((string * cterm) list * thm list) * Proof.context
```

`SUBPROOF` *tac ctxt i* decomposes the structure of the specified sub-goal, producing an extended context and a reduced goal, which needs to be solved by the given tactic. All schematic parameters of the goal are imported into the context as fixed ones, which may not be instantiated in the sub-proof.

`Subgoal.FOCUS`, `Subgoal.FOCUS_PREMS`, and `Subgoal.FOCUS_PARAMS` are similar to `SUBPROOF`, but are slightly more flexible: only the specified parts of the subgoal are imported into the context, and the body tactic may introduce new subgoals and schematic variables.

`Subgoal.focus`, `Subgoal.focus_prems`, `Subgoal.focus_params` extract the focus information from a goal state in the same way as the corresponding tacticals above. This is occasionally useful to experiment without writing actual tactics yet.

`Goal.prove` *ctxt xs As C tac* states goal *C* in the context augmented by fixed variables *xs* and assumptions *As*, and applies tactic *tac* to solve it. The latter may depend on the local assumptions being presented as facts. The result is in HHF normal form.

`Goal.prove_common` *ctxt fork_pri* is the common form to state and prove a simultaneous goal statement, where `Goal.prove` is a convenient shorthand that is most frequently used in applications.

The given list of simultaneous conclusions is encoded in the goal state by means of Pure conjunction: `Goal.conjunction_tac` will turn this into a collection of individual subgoals, but note that the original multi-goal state is usually required for advanced induction.

It is possible to provide an optional priority for a forked proof, typically `SOME ~1`, while `NONE` means the proof is immediate (sequential) as for `Goal.prove`. Note that a forked proof does not exhibit any failures in the usual way via exceptions in ML, but accumulates error situations under the execution id of the running transaction. Thus the system is able to expose error messages ultimately to the end-user, even though the subsequent ML code misses them.

`Obtain.result` *tac thms ctxt* eliminates the given facts using a tactic, which results in additional fixed variables and assumptions in the context. Final results need to be exported explicitly.

ML Examples

The following minimal example illustrates how to access the focus information of a structured goal state.

```
notepad
begin
 fix A B C :: 'a ⇒ bool

 have ⋀x. A x ⟹ B x ⟹ C x
   ML_val
     ‹val {goal, context = goal_ctxt, ...} = @{Isar.goal};
      val (focus as {params, asms, concl, ...}, goal') =
        Subgoal.focus goal_ctxt 1 (SOME [binding ‹x›]) goal;
      val [A, B] = #prems focus;
      val [(_, x)] = #params focus;›
   sorry
end
```

The next example demonstrates forward-elimination in a local context, using `Obtain.result`.

```
notepad
begin
 assume ex: ∃x. B x

 ML_prf
   ‹val ctxt0 = context;
    val (([(_, x)], [B]), ctxt1) = ctxt0
      |> Obtain.result (fn _ => eresolve_tac ctxt0 @{thms exE} 1)
[@{thm ex}];›
  ML_prf
   ‹singleton (Proof_Context.export ctxt1 ctxt0) @{thm refl};›
  ML_prf
   ‹Proof_Context.export ctxt1 ctxt0 [Thm.reflexive x]
      handle ERROR msg => (warning msg; []);›
end
```

Chapter 7

Isar language elements

The Isar proof language (see also [19, §2]) consists of three main categories of language elements:

1. Proof *commands* define the primary language of transactions of the underlying Isar/VM interpreter. Typical examples are **fix**, **assume**, **show**, **proof**, and **qed**.

   Composing proof commands according to the rules of the Isar/VM leads to expressions of structured proof text, such that both the machine and the human reader can give it a meaning as formal reasoning.

2. Proof *methods* define a secondary language of mixed forward-backward refinement steps involving facts and goals. Typical examples are *rule*, *unfold*, and *simp*.

   Methods can occur in certain well-defined parts of the Isar proof language, say as arguments to **proof**, **qed**, or **by**.

3. *Attributes* define a tertiary language of small annotations to theorems being defined or referenced. Attributes can modify both the context and the theorem.

   Typical examples are *intro* (which affects the context), and *symmetric* (which affects the theorem).

7.1 Proof commands

A *proof command* is state transition of the Isar/VM proof interpreter.

In principle, Isar proof commands could be defined in user-space as well. The system is built like that in the first place: one part of the commands are primitive, the other part is defined as derived elements. Adding to the genuine structured proof language requires profound understanding of the Isar/VM machinery, though, so this is beyond the scope of this manual.

What can be done realistically is to define some diagnostic commands that inspect the general state of the Isar/VM, and report some feedback to the user. Typically this involves checking of the linguistic *mode* of a proof state, or peeking at the pending goals (if available).

Another common application is to define a toplevel command that poses a problem to the user as Isar proof state and processes the final result relatively to the context. Thus a proof can be incorporated into the context of some user-space tool, without modifying the Isar proof language itself.

ML Reference

```
type Proof.state
Proof.assert_forward: Proof.state -> Proof.state
Proof.assert_chain: Proof.state -> Proof.state
Proof.assert_backward: Proof.state -> Proof.state
Proof.simple_goal: Proof.state -> {context: Proof.context, goal: thm}
Proof.goal: Proof.state ->
  {context: Proof.context, facts: thm list, goal: thm}
Proof.raw_goal: Proof.state ->
  {context: Proof.context, facts: thm list, goal: thm}
Proof.theorem: Method.text option ->
  (thm list list -> Proof.context -> Proof.context) ->
  (term * term list) list list -> Proof.context -> Proof.state
```

Type `Proof.state` represents Isar proof states. This is a block-structured configuration with proof context, linguistic mode, and optional goal. The latter consists of goal context, goal facts ("*using*"), and tactical goal state (see §4.1).

The general idea is that the facts shall contribute to the refinement of some parts of the tactical goal — how exactly is defined by the proof method that is applied in that situation.

`Proof.assert_forward`, `Proof.assert_chain`, `Proof.assert_backward` are partial identity functions that fail unless a certain linguistic mode is active, namely "*proof*(*state*)", "*proof*(*chain*)", "*proof*(*prove*)", respectively (using the terminology of [19]).

It is advisable study the implementations of existing proof commands for suitable modes to be asserted.

`Proof.simple_goal` *state* returns the structured Isar goal (if available) in the form seen by "simple" methods (like *simp* or *blast*). The Isar goal facts are already inserted as premises into the subgoals, which are presented individually as in `Proof.goal`.

`Proof.goal` *state* returns the structured Isar goal (if available) in the form seen by regular methods (like *rule*). The auxiliary internal encoding of Pure conjunctions is split into individual subgoals as usual.

`Proof.raw_goal` *state* returns the structured Isar goal (if available) in the raw internal form seen by "raw" methods (like *induct*). This form is rarely appropriate for diagnostic tools; `Proof.simple_goal` or `Proof.goal` should be used in most situations.

`Proof.theorem` *before_qed after_qed statement ctxt* initializes a toplevel Isar proof state within a given context.

The optional *before_qed* method is applied at the end of the proof, just before extracting the result (this feature is rarely used).

The *after_qed* continuation receives the extracted result in order to apply it to the final context in a suitable way (e.g. storing named facts). Note that at this generic level the target context is specified as `Proof.context`, but the usual wrapping of toplevel proofs into command transactions will provide a `local_theory` here (chapter 8). This affects the way how results are stored.

The *statement* is given as a nested list of terms, each associated with optional **is** patterns as usual in the Isar source language. The original nested list structure over terms is turned into one over theorems when *after_qed* is invoked.

ML Antiquotations

Isar.goal : *ML_antiquotation*

@{*Isar.goal*} refers to the regular goal state (if available) of the current proof state managed by the Isar toplevel — as abstract value.

This only works for diagnostic ML commands, such as **ML_val** or **ML_command**.

ML Examples

The following example peeks at a certain goal configuration.

notepad

```
begin
  have A and B and C
    ML_val
      ‹val n = Thm.nprems_of (#goal @{Isar.goal});
       assert (n = 3);›
    sorry
end
```

Here we see 3 individual subgoals in the same way as regular proof methods would do.

7.2 Proof methods

A *method* is a function $\mathit{thm}^* \to \mathit{context} * \mathit{goal} \to (\mathit{context} \times \mathit{goal})^{**}$ that operates on the full Isar goal configuration with context, goal facts, and tactical goal state and enumerates possible follow-up goal states. Under normal circumstances, the goal context remains unchanged, but it is also possible to declare named extensions of the proof context (*cases*).

This means a proof method is like a structurally enhanced tactic (cf. §4.2). The well-formedness conditions for tactics need to hold for methods accordingly, with the following additions.

- Goal addressing is further limited either to operate uniformly on *all* subgoals, or specifically on the *first* subgoal.

  Exception: old-style tactic emulations that are embedded into the method space, e.g. *rule_tac*.

- A non-trivial method always needs to make progress: an identical follow-up goal state has to be avoided.[1]

  Exception: trivial stuttering steps, such as "$-$" or *succeed*.

- Goal facts passed to the method must not be ignored. If there is no sensible use of facts outside the goal state, facts should be inserted into the subgoals that are addressed by the method.

Syntactically, the language of proof methods appears as arguments to Isar commands like **by** or **apply**. User-space additions are reasonably easy by

[1] This enables the user to write method expressions like meth^+ without looping, while the trivial do-nothing case can be recovered via $\mathit{meth}^?$.

plugging suitable method-valued parser functions into the framework, using the **method_setup** command, for example.

To get a better idea about the range of possibilities, consider the following Isar proof schemes. This is the general form of structured proof text:

from $facts_1$ **have** *props* **using** $facts_2$
proof (*initial_method*)
  body
qed (*terminal_method*)

The goal configuration consists of $facts_1$ and $facts_2$ appended in that order, and various *props* being claimed. The *initial_method* is invoked with facts and goals together and refines the problem to something that is handled recursively in the proof *body*. The *terminal_method* has another chance to finish any remaining subgoals, but it does not see the facts of the initial step.

This pattern illustrates unstructured proof scripts:

have *props*
  using $facts_1$ **apply** $method_1$
  apply $method_2$
  using $facts_3$ **apply** $method_3$
  done

The $method_1$ operates on the original claim while using $facts_1$. Since the **apply** command structurally resets the facts, the $method_2$ will operate on the remaining goal state without facts. The $method_3$ will see again a collection of $facts_3$ that has been inserted into the script explicitly.

Empirically, any Isar proof method can be categorized as follows.

1. *Special method with cases* with named context additions associated with the follow-up goal state.

   Example: *induct*, which is also a "raw" method since it operates on the internal representation of simultaneous claims as Pure conjunction (§2.3.2), instead of separate subgoals (§4.1).

2. *Structured method* with strong emphasis on facts outside the goal state.

   Example: *rule*, which captures the key ideas behind structured reasoning in Isar in its purest form.

3. *Simple method* with weaker emphasis on facts, which are inserted into subgoals to emulate old-style tactical "premises".

   Examples: *simp*, *blast*, *auto*.

4. *Old-style tactic emulation* with detailed numeric goal addressing and explicit references to entities of the internal goal state (which are otherwise invisible from proper Isar proof text). The naming convention *foo_tac* makes this special non-standard status clear.

   Example: *rule_tac*.

When implementing proof methods, it is advisable to study existing implementations carefully and imitate the typical "boiler plate" for context-sensitive parsing and further combinators to wrap-up tactic expressions as methods.[2]

ML Reference

```
type Proof.method
CONTEXT_METHOD: (thm list -> context_tactic) -> Proof.method
METHOD: (thm list -> tactic) -> Proof.method
SIMPLE_METHOD: tactic -> Proof.method
SIMPLE_METHOD': (int -> tactic) -> Proof.method
Method.insert_tac: Proof.context -> thm list -> int -> tactic
Method.setup: binding -> (Proof.context -> Proof.method) context_parser ->
  string -> theory -> theory
```

Type `Proof.method` represents proof methods as abstract type.

`CONTEXT_METHOD` (*fn facts => context_tactic*) wraps *context_tactic* depending on goal facts as a general proof method that may change the proof context dynamically. A typical operation is `Proof_Context.update_cases`, which is wrapped up as combinator `CONTEXT_CASES` for convenience.

`METHOD` (*fn facts => tactic*) wraps *tactic* depending on goal facts as regular proof method; the goal context is passed via method syntax.

`SIMPLE_METHOD` *tactic* wraps a tactic that addresses all subgoals uniformly as simple proof method. Goal facts are already inserted into all subgoals before *tactic* is applied.

`SIMPLE_METHOD'` *tactic* wraps a tactic that addresses a specific subgoal as simple proof method that operates on subgoal 1. Goal facts are inserted into the subgoal then the *tactic* is applied.

[2] Aliases or abbreviations of the standard method combinators should be avoided. Note that from Isabelle99 until Isabelle2009 the system did provide various odd combinations of method syntax wrappers that made applications more complicated than necessary.

`Method.insert_tac` *ctxt facts i* inserts *facts* into subgoal *i*. This is convenient to reproduce part of the `SIMPLE_METHOD` or `SIMPLE_METHOD'` wrapping within regular `METHOD`, for example.

`Method.setup` *name parser description* provides the functionality of the Isar command **method_setup** as ML function.

ML Examples

See also **method_setup** in [19] which includes some abstract examples.

The following toy examples illustrate how the goal facts and state are passed to proof methods. The predefined proof method called "*tactic*" wraps ML source of type `tactic` (abstracted over `facts`). This allows immediate experimentation without parsing of concrete syntax.

```
notepad
begin
  fix A B :: bool
  assume a: A and b: B

  have A ∧ B
    apply (tactic ‹resolve_tac context @{thms conjI} 1›)
    using a apply (tactic ‹resolve_tac context facts 1›)
    using b apply (tactic ‹resolve_tac context facts 1›)
    done

  have A ∧ B
    using a and b
    ML_val ‹@{Isar.goal}›
    apply (tactic ‹Method.insert_tac context facts 1›)
    apply (tactic ‹(resolve_tac context @{thms conjI} THEN_ALL_NEW assume_tac context) 1›)
    done
end
```

The next example implements a method that simplifies the first subgoal by rewrite rules that are given as arguments.

```
method_setup my_simp =
  ‹Attrib.thms >> (fn thms => fn ctxt =>
    SIMPLE_METHOD' (fn i =>
      CHANGED (asm_full_simp_tac
      (put_simpset HOL_basic_ss ctxt addsimps thms) i)))›
  "rewrite subgoal by given rules"
```

The concrete syntax wrapping of **method_setup** always passes-through the proof context at the end of parsing, but it is not used in this example.

The `Attrib.thms` parser produces a list of theorems from the usual Isar syntax involving attribute expressions etc. (syntax category *thms*) [19]. The resulting `thms` are added to `HOL_basic_ss` which already contains the basic Simplifier setup for HOL.

The tactic `asm_full_simp_tac` is the one that is also used in method *simp* by default. The extra wrapping by the `CHANGED` tactical ensures progress of simplification: identical goal states are filtered out explicitly to make the raw tactic conform to standard Isar method behaviour.

Method *my_simp* can be used in Isar proofs like this:

```
notepad
begin
  fix a b c :: 'a
  assume a: a = b
  assume b: b = c
  have a = c by (my_simp a b)
end
```

Here is a similar method that operates on all subgoals, instead of just the first one.

```
method_setup my_simp_all =
  ‹Attrib.thms >> (fn thms => fn ctxt =>
    SIMPLE_METHOD
      (CHANGED
        (ALLGOALS (asm_full_simp_tac
          (put_simpset HOL_basic_ss ctxt addsimps thms)))))›
  "rewrite all subgoals by given rules"

notepad
begin
  fix a b c :: 'a
  assume a: a = b
  assume b: b = c
  have a = c and c = b by (my_simp_all a b)
end
```

Apart from explicit arguments, common proof methods typically work with a default configuration provided by the context. As a shortcut to rule management we use a cheap solution via the **named_theorems** command to declare a dynamic fact in the context.

named_theorems *my_simp*

The proof method is now defined as before, but we append the explicit arguments and the rules from the context.

```
method_setup my_simp' =
  ‹Attrib.thms >> (fn thms => fn ctxt =>
    let
      val my_simps = Named_Theorems.get ctxt named_theorems‹my_simp›
    in
      SIMPLE_METHOD' (fn i =>
        CHANGED (asm_full_simp_tac
          (put_simpset HOL_basic_ss ctxt
            addsimps (thms @ my_simps)) i))
    end)›
  "rewrite subgoal by given rules and my_simp rules from the context"
```

Method *my_simp′* can be used in Isar proofs like this:

```
notepad
begin
  fix a b c :: 'a
  assume [my_simp]: a ≡ b
  assume [my_simp]: b ≡ c
  have a ≡ c by my_simp'
end
```

The *my_simp* variants defined above are “simple” methods, i.e. the goal facts are merely inserted as goal premises by the `SIMPLE_METHOD'` or `SIMPLE_METHOD` wrapper. For proof methods that are similar to the standard collection of *simp*, *blast*, *fast*, *auto* there is little more that can be done.

Note that using the primary goal facts in the same manner as the method arguments obtained via concrete syntax or the context does not meet the requirement of “strong emphasis on facts” of regular proof methods, because rewrite rules as used above can be easily ignored. A proof text “**using** *foo* **by** *my_simp*” where *foo* is not used would deceive the reader.

The technical treatment of rules from the context requires further attention. Above we rebuild a fresh `simpset` from the arguments and *all* rules retrieved from the context on every invocation of the method. This does not scale to really large collections of rules, which easily emerges in the context of a big theory library, for example.

This is an inherent limitation of the simplistic rule management via **named_theorems**, because it lacks tool-specific storage and retrieval. More realistic applications require efficient index-structures that organize theorems in a customized manner, such as a discrimination net that is indexed by the left-hand sides of rewrite rules. For variations on the Simplifier, re-use of the existing type `simpset` is adequate, but scalability would require it be maintained statically within the context data, not dynamically on each tool invocation.

7.3 Attributes

An *attribute* is a function *context* × *thm* → *context* × *thm*, which means both a (generic) context and a theorem can be modified simultaneously. In practice this mixed form is very rare, instead attributes are presented either as *declaration attribute: thm* → *context* → *context* or *rule attribute: context* → *thm* → *thm*.

Attributes can have additional arguments via concrete syntax. There is a collection of context-sensitive parsers for various logical entities (types, terms, theorems). These already take care of applying morphisms to the arguments when attribute expressions are moved into a different context (see also §8.2).

When implementing declaration attributes, it is important to operate exactly on the variant of the generic context that is provided by the system, which is either global theory context or local proof context. In particular, the background theory of a local context must not be modified in this situation!

ML Reference

```
type attribute
Thm.rule_attribute: thm list ->
  (Context.generic -> thm -> thm) -> attribute
Thm.declaration_attribute:
  (thm -> Context.generic -> Context.generic) -> attribute
Attrib.setup: binding -> attribute context_parser ->
  string -> theory -> theory
```

Type `attribute` represents attributes as concrete type alias.

`Thm.rule_attribute` *thms* (*fn context* => *rule*) wraps a context-dependent rule (mapping on `thm`) as attribute.

The *thms* are additional parameters: when forming an abstract closure, the system may provide dummy facts that are propagated according to strict evaluation discipline. In that case, *rule* is bypassed.

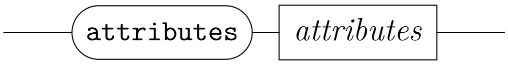
`Thm.declaration_attribute` (*fn thm => decl*) wraps a theorem-dependent declaration (mapping on `Context.generic`) as attribute.

When forming an abstract closure, the system may provide a dummy fact as *thm*. In that case, *decl* is bypassed.

`Attrib.setup` *name parser description* provides the functionality of the Isar command **attribute_setup** as ML function.

ML Antiquotations

attributes : *ML_antiquotation*



@{*attributes* [. . .]} embeds attribute source representation into the ML text, which is particularly useful with declarations like `Local_Theory.note`. Attribute names are internalized at compile time, but the source is unevaluated. This means attributes with formal arguments (types, terms, theorems) may be subject to odd effects of dynamic scoping!

ML Examples

See also **attribute_setup** in [19] which includes some abstract examples.

Chapter 8

Local theory specifications

A *local theory* combines aspects of both theory and proof context (cf. §1.1), such that definitional specifications may be given relatively to parameters and assumptions. A local theory is represented as a regular proof context, augmented by administrative data about the *target context*.

The target is usually derived from the background theory by adding local **fix** and **assume** elements, plus suitable modifications of non-logical context data (e.g. a special type-checking discipline). Once initialized, the target is ready to absorb definitional primitives: **define** for terms and **note** for theorems. Such definitions may get transformed in a target-specific way, but the programming interface hides such details.

Isabelle/Pure provides target mechanisms for locales, type-classes, type-class instantiations, and general overloading. In principle, users can implement new targets as well, but this rather arcane discipline is beyond the scope of this manual. In contrast, implementing derived definitional packages to be used within a local theory context is quite easy: the interfaces are even simpler and more abstract than the underlying primitives for raw theories.

Many definitional packages for local theories are available in Isabelle. Although a few old packages only work for global theories, the standard way of implementing definitional packages in Isabelle is via the local theory interface.

8.1 Definitional elements

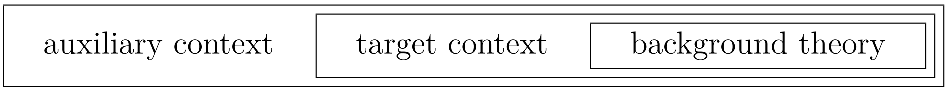
There are separate elements **define** $c \equiv t$ for terms, and **note** $b = thm$ for theorems. Types are treated implicitly, according to Hindley-Milner discipline (cf. §6.1). These definitional primitives essentially act like *let*-bindings within a local context that may already contain earlier *let*-bindings and some initial λ-bindings. Thus we gain *dependent definitions* that are relative to an initial axiomatic context. The following diagram illustrates this idea of axiomatic elements versus definitional elements:

	λ-binding	*let*-binding
types	fixed α	arbitrary β
terms	**fix** $x :: \tau$	**define** $c \equiv t$
theorems	**assume** a: A	**note** b = ‘B‘

A user package merely needs to produce suitable **define** and **note** elements according to the application. For example, a package for inductive definitions might first **define** a certain predicate as some fixed-point construction, then **note** a proven result about monotonicity of the functor involved here, and then produce further derived concepts via additional **define** and **note** elements.

The cumulative sequence of **define** and **note** produced at package runtime is managed by the local theory infrastructure by means of an *auxiliary context*. Thus the system holds up the impression of working within a fully abstract situation with hypothetical entities: **define** $c \equiv t$ always results in a literal fact ‘$c \equiv t$‘, where c is a fixed variable c. The details about global constants, name spaces etc. are handled internally.

So the general structure of a local theory is a sandwich of three layers:



When a definitional package is finished, the auxiliary context is reset to the target context. The target now holds definitions for terms and theorems that stem from the hypothetical **define** and **note** elements, transformed by the particular target policy (see [9, §4–5] for details).

ML Reference

```
type local_theory = Proof.context
Named_Target.init: string list -> string -> theory -> local_theory
Local_Theory.define: (binding * mixfix) * (Attrib.binding * term) ->
    local_theory -> (term * (string * thm)) * local_theory
Local_Theory.note: Attrib.binding * thm list ->
    local_theory -> (string * thm list) * local_theory
```

Type `local_theory` represents local theories. Although this is merely an alias for `Proof.context`, it is semantically a subtype of the same: a `local_theory` holds target information as special context data. Subtyping means that any value *lthy*: `local_theory` can be also used with operations on expecting a regular *ctxt*: `Proof.context`.

`Named_Target.init` *includes name thy* initializes a local theory derived from the given background theory. An empty name refers to a *global theory* context, and a non-empty name refers to a **locale** or **class** context (a fully-qualified internal name is expected here). This is useful for experimentation — normally the Isar toplevel already takes care to initialize the local theory context.

`Local_Theory.define` ((*b*, *mx*), (*a*, *rhs*)) *lthy* defines a local entity according to the specification that is given relatively to the current *lthy* context. In particular the term of the RHS may refer to earlier local entities from the auxiliary context, or hypothetical parameters from the target context. The result is the newly defined term (which is always a fixed variable with exactly the same name as specified for the LHS), together with an equational theorem that states the definition as a hypothetical fact.

Unless an explicit name binding is given for the RHS, the resulting fact will be called *b_def*. Any given attributes are applied to that same fact — immediately in the auxiliary context *and* in any transformed versions stemming from target-specific policies or any later interpretations of results from the target context (think of **locale** and **interpretation**, for example). This means that attributes should be usually plain declarations such as *simp*, while non-trivial rules like *simplified* are better avoided.

`Local_Theory.note` (*a*, *ths*) *lthy* is analogous to `Local_Theory.define`, but defines facts instead of terms. There is also a slightly more general variant `Local_Theory.notes` that defines several facts (with attribute expressions) simultaneously.

This is essentially the internal version of the **lemmas** command, or **declare** if an empty name binding is given.

8.2 Morphisms and declarations

See also [3].

Chapter 9

System integration

9.1 Isar toplevel

The Isar *toplevel state* represents the outermost configuration that is transformed by a sequence of transitions (commands) within a theory body. This is a pure value with pure functions acting on it in a timeless and stateless manner. Historically, the sequence of transitions was wrapped up as sequential command loop, such that commands are applied one-by-one. In contemporary Isabelle/Isar, processing toplevel commands usually works in parallel in multi-threaded Isabelle/ML [21, 22].

9.1.1 Toplevel state

The toplevel state is a disjoint sum of empty *toplevel*, or *theory*, or *proof*. The initial toplevel is empty; a theory is commenced by a **theory** header; within a theory we may use theory commands such as **definition**, or state a **theorem** to be proven. A proof state accepts a rich collection of Isar proof commands for structured proof composition, or unstructured proof scripts. When the proof is concluded we get back to the (local) theory, which is then updated by defining the resulting fact. Further theory declarations or theorem statements with proofs may follow, until we eventually conclude the theory development by issuing **end** to get back to the empty toplevel.

ML **Reference**

```
type Toplevel.state
exception Toplevel.UNDEF
Toplevel.is_toplevel: Toplevel.state -> bool
Toplevel.theory_of: Toplevel.state -> theory
Toplevel.proof_of: Toplevel.state -> Proof.state
```

Type `Toplevel.state` represents Isar toplevel states, which are normally manipulated through the concept of toplevel transitions only (§9.1.2).

`Toplevel.UNDEF` is raised for undefined toplevel operations. Many operations work only partially for certain cases, since `Toplevel.state` is a sum type.

`Toplevel.is_toplevel` *state* checks for an empty toplevel state.

`Toplevel.theory_of` *state* selects the background theory of *state*, it raises `Toplevel.UNDEF` for an empty toplevel state.

`Toplevel.proof_of` *state* selects the Isar proof state if available, otherwise it raises an error.

ML Antiquotations

Isar.state : *ML_antiquotation*

@{*Isar.state*} refers to Isar toplevel state at that point — as abstract value.

This only works for diagnostic ML commands, such as **ML_val** or **ML_command**.

9.1.2 Toplevel transitions

An Isar toplevel transition consists of a partial function on the toplevel state, with additional information for diagnostics and error reporting: there are fields for command name, source position, and other meta-data.

The operational part is represented as the sequential union of a list of partial functions, which are tried in turn until the first one succeeds. This acts like an outer case-expression for various alternative state transitions. For example, **qed** works differently for a local proofs vs. the global ending of an outermost proof.

Transitions are composed via transition transformers. Internally, Isar commands are put together from an empty transition extended by name and source position. It is then left to the individual command parser to turn the given concrete syntax into a suitable transition transformer that adjoins actual operations on a theory or proof state.

ML Reference

```
Toplevel.keep: (Toplevel.state -> unit) ->
  Toplevel.transition -> Toplevel.transition
Toplevel.theory: (theory -> theory) ->
  Toplevel.transition -> Toplevel.transition
Toplevel.theory_to_proof: (theory -> Proof.state) ->
  Toplevel.transition -> Toplevel.transition
Toplevel.proof: (Proof.state -> Proof.state) ->
  Toplevel.transition -> Toplevel.transition
Toplevel.proofs: (Proof.state -> Proof.state Seq.result Seq.seq) ->
  Toplevel.transition -> Toplevel.transition
Toplevel.end_proof: (bool -> Proof.state -> Proof.context) ->
  Toplevel.transition -> Toplevel.transition
```

`Toplevel.keep` *tr* adjoins a diagnostic function.

`Toplevel.theory` *tr* adjoins a theory transformer.

`Toplevel.theory_to_proof` *tr* adjoins a global goal function, which turns a theory into a proof state. The theory may be changed before entering the proof; the generic Isar goal setup includes an `after_qed` argument that specifies how to apply the proven result to the enclosing context, when the proof is finished.

`Toplevel.proof` *tr* adjoins a deterministic proof command, with a singleton result.

`Toplevel.proofs` *tr* adjoins a general proof command, with zero or more result states (represented as a lazy list).

`Toplevel.end_proof` *tr* adjoins a concluding proof command, that returns the resulting theory, after applying the resulting facts to the target context.

ML Examples

The file `~~/src/HOL/Examples/Commands.thy` shows some example Isar command definitions, with the all-important theory header declarations for outer syntax keywords.

9.2 Theory loader database

In batch mode and within dumped logic images, the theory database maintains a collection of theories as a directed acyclic graph. A theory may refer to other theories as **imports**, or to auxiliary files via special *load commands* (e.g. **ML_file**). For each theory, the base directory of its own theory file is called *master directory*: this is used as the relative location to refer to other files from that theory.

ML Reference

```
use_thy: string -> unit
Thy_Info.get_theory: string -> theory
Thy_Info.remove_thy: string -> unit
Thy_Info.register_thy: theory -> unit
```

`use_thy` *A* ensures that theory *A* is fully up-to-date wrt. the external file store; outdated ancestors are reloaded on demand.

`Thy_Info.get_theory` *A* retrieves the theory value presently associated with name *A*. Note that the result might be outdated wrt. the file-system content.

`Thy_Info.remove_thy` *A* deletes theory *A* and all descendants from the theory database.

`Thy_Info.register_thy` *text thy* registers an existing theory value with the theory loader database and updates source version information according to the file store.

Bibliography

[1] H. Barendregt and H. Geuvers. Proof assistants using dependent type systems. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier, 2001.

[2] S. Berghofer and T. Nipkow. Proof terms for simply typed higher order logic. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 38–52. Springer-Verlag, 2000.

[3] A. Chaieb and M. Wenzel. Context aware calculation and deduction — ring equalities via Gröbner Bases in Isabelle. In M. Kauers, M. Kerber, R. Miner, and W. Windsteiger, editors, *Towards Mechanized Mathematical Assistants (CALCULEMUS 2007)*, volume 4573 of *LNAI*. Springer-Verlag, 2007.

[4] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[5] L. Damas and H. Milner. Principal type schemes for functional programs. In *ACM Symp. Principles of Programming Languages*, 1982.

[6] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indag. Math.*, 34:381–392, 1972.

[7] G. Gentzen. Untersuchungen über das logische Schließen. *Math. Zeitschrift*, 1935.

[8] F. Haftmann and M. Wenzel. Constructive type classes in Isabelle. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs, TYPES 2006*, volume 4502 of *LNCS*. Springer, 2007.

[9] F. Haftmann and M. Wenzel. Local theory specifications in Isabelle/Isar. In S. Berardi, F. Damiani, and U. de Liguoro, editors, *Types for Proofs and Programs, TYPES 2008*, volume 5497 of *LNCS*. Springer, 2009.

[10] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4), 1991.

[11] T. Nipkow. Functional unification of higher-order patterns. In M. Vardi, editor, *Eighth Annual Symposium on Logic in Computer Science*, pages 64–74. IEEE Computer Society Press, 1993.

[12] T. Nipkow and C. Prehofer. Type reconstruction for type classes. *Journal of Functional Programming*, 5(2):201–224, 1995.

[13] L. C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983. http://www.cl.cam.ac.uk/~lp15/papers/Reports/TR035-lcp-rewriting.pdf.

[14] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[15] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996. https://www.cl.cam.ac.uk/~lp15/MLbook.

[16] P. Schroeder-Heister. A natural extension of natural deduction. *Journal of Symbolic Logic*, 49(4), 1984.

[17] H. Sutter. The free lunch is over — a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[18] M. Wenzel. *The Isabelle System Manual*. https://isabelle.in.tum.de/doc/system.pdf.

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

[20] M. Wenzel. *Isabelle/jEdit*. https://isabelle.in.tum.de/doc/jedit.pdf.

[21] M. Wenzel. Parallel proof checking in Isabelle/Isar. In G. Dos Reis and L. Théry, editors, *ACM SIGSAM Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS 2009)*. ACM Digital Library, 2009.

[22] M. Wenzel. Shared-memory multiprocessing for interactive theorem proving. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving — 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*. Springer, 2013.

Index