The Isabelle/Isar Implementation

Makarius Wenzel

With Contributions by Florian Haftmann and Larry Paulson

21 June 2010

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

Contents

List of Figures

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 [11]).

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.

In order to avoid the full-scale overhead of explicit sub-theory identification of arbitrary intermediate stages, a theory is switched into *draft* mode under certain circumstances. A draft theory acts like a linear type, where updates invalidate earlier versions. An invalidated draft is called *stale*.

The *checkpoint* operation produces a safe stepping stone that will survive the next update without becoming stale: both the old and the new theory remain valid and are related by the sub-theory relation. Checkpointing essentially recovers purely functional theory values, at the expense of some extra internal bookkeeping.

The *copy* operation produces an auxiliary version that has the same data content, but is unrelated to the original: updates of the copy do not affect the original, neither does the sub-theory relation hold.

The *merge* operation produces the least upper bound of two theories, which actually degenerates into absorption of one theory into the other (according to the nominal sub-theory relation).

The *begin* operation starts a new theory by importing several parent theories 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, working mostly on drafts internally, while transaction boundaries of Isar top-level commands (§8.1) are guaranteed to be safe checkpoints.

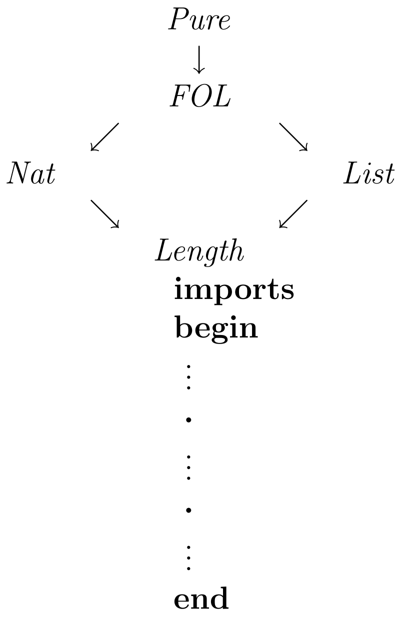


Figure 1.1: A theory definition depending on ancestors

There is a separate notion of *theory reference* for maintaining a live link to an evolving theory context: updates on drafts are propagated automatically. Dynamic updating stops after an explicit *end* only.

Derived entities may store a theory reference in order to indicate the context they belong to. This implicitly assumes monotonic reasoning, because the referenced context may become larger without further notice.

ML

Reference

```
type theory
Theory.subthy: theory * theory -> bool
Theory.checkpoint: theory -> theory
Theory.copy: theory -> theory
Theory.merge: theory * theory -> theory
Theory.begin_theory: string -> theory list -> theory
```

```
type theory_ref
Theory.deref: theory_ref -> theory
Theory.check_thy: theory -> theory_ref
```

`theory` represents theory contexts. This is essentially a linear type, with explicit runtime checking! Most internal theory operations destroy the original version, which then becomes "stale".

`Theory.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.checkpoint` *thy* produces a safe stepping stone in the linear development of *thy*. This changes the old theory, but the next update will result in two related, valid theories.

`Theory.copy` *thy* produces a variant of *thy* with the same data. The copy is not related to the original, but the original is unchanged.

`Theory.merge` (thy_1, thy_2) absorbs one theory into the other, without changing thy_1 or thy_2. This version of ad-hoc theory merge fails for unrelated theories!

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

`theory_ref` represents a sliding reference to an always valid theory; updates on the original are propagated automatically.

`Theory.deref` *thy_ref* turns a `theory_ref` into an `theory` value. As the referenced theory evolves monotonically over time, later invocations of `Theory.deref` may refer to a larger context.

`Theory.check_thy` *thy* produces a `theory_ref` from a valid `theory` value.

1.1.2 Proof context

A proof context is a container for pure data with a back-reference to the theory it belongs to. The *init* operation creates a proof context from a given theory. Modifications to draft theories are propagated to the proof context as usual, but there is also an explicit *transfer* operation to force resynchronization with more substantial updates to the underlying theory.

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
ProofContext.init_global: theory -> Proof.context
ProofContext.theory_of: Proof.context -> theory
ProofContext.transfer: theory -> Proof.context -> Proof.context
```

`Proof.context` represents proof contexts. Elements of this type are essentially pure values, with a sliding reference to the background theory.

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

`ProofContext.theory_of` *ctxt* selects the background theory from *ctxt*, dereferencing its internal `theory_ref`.

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

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
```

`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 `ProofContext.theory_of` as required.

`Context.proof_of` *context* always produces a proof context from the generic *context*, using `ProofContext.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 SML signature:

type T	representing type
val *empty*: T	empty default value
val *extend*: $T \to T$	re-initialize on import
val *merge*: $T \times T \to T$	join on import

The *empty* value acts as initial default for *any* theory that does not declare actual data content; *extend* is acts like a unitary version of *merge*.

Implementing *merge* can be tricky. The general idea is that *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 is like a chain of diamonds would cause an exponential blowup!

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

type *T*	representing type
val *init*: *theory* → *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.

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 these data declaration over type *T* result in an ML structure with the following signature:

get: *context* → *T*
put: *T* → *context* → *context*
map: (*T* → *T*) → *context* → *context*

These other operations provide exclusive access for the particular kind of context (theory, proof, or generic context). This interface fully 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 OrdList.T;
    val empty = [];
    val extend = I;
    fun merge (ts1, ts2) =
      OrdList.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 (OrdList.insert Term_Ord.fast_term_ord t) thy end;

  end;
*}
```

We use `term OrdList.T` for reasonably efficient representation of a set of terms: all operations are linear in the number of stored elements. Here we assume that our users 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: `OrdList.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. §5.2) is not necessarily monotonic wrt. the background theory, since constraints of term constants can be strengthened 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.

1.2 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.

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

- Basic names (§1.2.2): free and bound variables.
- Indexed names (§1.2.3): schematic variables.
- Long names (§1.2.4): constants of any kind (type constructors, term constants, other concepts defined in user space). Such entities are typically managed via name spaces (§1.2.5).

1.2.1 Strings of symbols

A *symbol* constitutes the smallest textual unit in Isabelle — 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*” or raw byte in the range of 128. . . 255, for example “`a`”,

2. a regular symbol “`\<`*ident*`>`”, for example “`\<alpha>`”,

3. a control symbol “`\<^`*ident*`>`”, for example “`\<^bold>`”,

4. a raw symbol “`\<^raw:`*text*`>`” where *text* consists of printable characters excluding “`.`” and “`>`”, for example “`\<^raw:$\sum_{i = 1}^n$>`”,

5. a numbered raw control symbol “`\<^raw`*n*`>` where *n* consists of digits, for example “`\<^raw42>`”.

The *ident* syntax for symbol names is $letter$ $(letter \mid digit)^*$, 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.

Since the character set underlying Isabelle symbols is 7-bit ASCII and 8-bit characters are passed through transparently, Isabelle can also process Unicode/UCS data in UTF-8 encoding.[1] 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
```

[1] When counting precise source positions internally, bytes in the range of 128. . . 191 are ignored. In UTF-8 encoding, this interval covers the additional trailer bytes, so Isabelle happens to count Unicode characters here, not bytes in memory. In ISO-Latin encoding, the ignored range merely includes some extra punctuation characters that even have replacements within the standard collection of Isabelle symbols; the accented letters range is counted properly.

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

`Symbol.symbol` represents individual Isabelle symbols.

`Symbol.explode` *str* produces a symbol list from the packed form. This function supercedes `String.explode` for virtually all purposes of manipulating text in Isabelle![2]

`Symbol.is_letter`, `Symbol.is_digit`, `Symbol.is_quasi`, `Symbol.is_blank` classify standard symbols according to fixed syntactic conventions of Isabelle, cf. [11].

`Symbol.sym` is a concrete datatype that represents the different kinds of symbols explicitly, with constructors `Symbol.Char`, `Symbol.Sym`, `Symbol.Ctrl`, `Symbol.Raw`.

`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 the basic `explode: string -> string list` operation would produce a list of singleton strings as in Isabelle/ML today. When SML97 came out, Isabelle did not adopt its slightly anachronistic 8-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.

1.2.2 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.

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

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.invents: Name.context -> string -> int -> string list
Name.variants: string list -> Name.context -> string list * 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.

`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.invents` *context name n* produces *n* fresh names derived from *name*.

`Name.variants` *names context* produces fresh variants of *names*; the result is entered into 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. §4.1).

1.2.3 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 $?x$, $?xa$, $?xb$.

$\boxed{\text{ML}}$ Reference

```
type indexname
```

`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.

1.2.4 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.

1.2.5 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:

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

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 are better suffixed in addition to the usual qualification, e.g. *c_type.intro* and *c_class.intro* for theorems related to type *c* and class *c*, respectively.

ML

Reference

```
type binding
Binding.empty: binding
Binding.name: string -> binding
Binding.qualify: bool -> string -> binding -> binding
Binding.prefix: bool -> string -> binding -> binding
Binding.conceal: binding -> binding
Binding.str_of: binding -> string
```

```
type Name_Space.naming
Name_Space.default_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: bool -> Name_Space.naming -> binding -> Name_Space.T ->
  string * Name_Space.T
Name_Space.intern: Name_Space.T -> string -> string
Name_Space.extern: Name_Space.T -> string -> string
Name_Space.is_concealed: Name_Space.T -> string -> bool
```

`binding` represents the abstract concept of name bindings.

`Binding.empty` is the empty binding.

`Binding.name` *name* produces a binding with base name *name*.

`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 7).

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

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

`Name_Space.default_naming` is the default naming policy. In a theory context, this is usually 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.

`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` *strict naming bindings space* enters a name binding as fully qualified internal name into the name space, with external accesses determined by the naming policy.

`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` *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!

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 [8]. 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 the universal sort, which is the largest element wrt. the sort order.

[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!

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 (α)*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 [7].

ML Reference

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

```
Sign.subsort: theory -> sort * sort -> bool
Sign.of_sort: theory -> typ * sort -> bool
Sign.add_types: (binding * int * mixfix) list -> theory -> theory
Sign.add_type_abbrev: 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
```

`class` represents type classes.

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

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

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

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

`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_types` $[(\kappa, k, mx), \ldots]$ declares a new type constructors κ with k arguments and optional mixfix syntax.

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

`Sign.primitive_class` $(c, [c_1, \ldots, c_n])$ 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 $\kappa :: (\vec{s})s$.

2.2 Terms

The language of terms is that of simply-typed λ-calculus with de-Bruijn indices for bound variables (cf. [3] or [9]), 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 applicatins:

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

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 $t\vartheta :: \sigma$ and $t\vartheta' :: \sigma$ 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
op aconv: term * term -> bool
map_types: (typ -> typ) -> term -> term
fold_types: (typ -> 'a -> 'a) -> term -> 'a -> 'a
map_aterms: (term -> term) -> term -> term
fold_aterms: (term -> 'a -> 'a) -> term -> 'a -> 'a
```

```
fastype_of: term -> typ
lambda: term -> term -> term
betapply: term * term -> term
Sign.declare_const: (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
```

`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`, `op $`.

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!

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

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

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

`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.

`Sign.declare_const` $((c,\ \sigma),\ mx)$ 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* $(c,\ \tau)$ and `Sign.const_instance` *thy* $(c,\ [\tau_1,\ \ldots,\ \tau_n])$
convert between two representations of polymorphic constants: full type instance vs. compact type arguments form.

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

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. 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

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

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

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

Figure 2.2: Primitive inferences of Pure

$\vdash (\lambda x.\ b[x])\ a \equiv b[a]$	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

inhabitable. "Assumptions" $x :: \tau$ for type-membership are only present as long as some x_τ occurs in the statement body.[2]

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.

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

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

Figure 2.4: Admissible substitution rules

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

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

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 system always records oracle invocations within derivations of theorems by a unique tag.

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.

ML

Reference

```
type ctyp
type cterm
Thm.ctyp_of: theory -> typ -> ctyp
Thm.cterm_of: theory -> term -> cterm
```

```
type thm
proofs: int Unsynchronized.ref
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: string list * string list -> int -> thm -> thm
Thm.instantiate: (ctyp * ctyp) list * (cterm * cterm) list -> thm -> thm
Thm.add_axiom: binding * term -> theory -> (string * thm) * theory
Thm.add_oracle: binding * (’a -> cterm) -> theory
  -> (string * (’a -> thm)) * theory
Thm.add_def: bool -> bool -> binding * term -> theory -> (string * thm) * theory
```

```
Theory.add_deps: string -> string * typ -> (string * typ) list -> theory -> theory
```

`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 theory.

`Thm.ctyp_of` *thy* τ and `Thm.cterm_of` *thy* t explicitly checks types and terms, respectively. This also involves some basic normalizations, such expansion of type and term abbreviations from the theory context.

Re-certification is relatively slow and should be avoided in tight reasoning loops. There are separate operations to decompose certified entities (including actual theorems).

`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 contains a sliding back-reference to the enclosing theory, cf. §1.1.1.

`proofs` specifies the detail of proof recording within `thm` values: `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.

`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 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` (*name*, A) *thy* 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` *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` *name* c_τ $\vec{d}_\sigma$ declares dependencies of a named specification for constant c_τ, relative to existing specifications for constants $\vec{d}_\sigma$.

2.3.2 Auxiliary definitions

Theory *Pure* provides a few auxiliary definitions, see figure 2.5. These special constants are normally not exposed to the user, but appear in internal encodings.

$\mathit{conjunction} :: \mathit{prop} \Rightarrow \mathit{prop} \Rightarrow \mathit{prop}$ (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. §3.1).

The *prop* marker (#) makes arbitrarily complex propositions appear as atomic, without changing the meaning: $\Gamma \vdash A$ and $\Gamma \vdash \#A$ are interchangeable. See §3.1 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 $\mathit{TYPE}(\tau)$ for $\mathit{TYPE}_{\tau\ \mathit{itself}}$. Although being devoid of any particular meaning, the term $\mathit{TYPE}(\tau)$ accounts for the type τ within the term language. In particular, $\mathit{TYPE}(\alpha)$ may be used as formal argument in primitive definitions, in order to circumvent hidden polymorphism (cf. §2.2). For example, $c\ \mathit{TYPE}(\alpha) \equiv A[\alpha]$ defines $c :: \alpha\ \mathit{itself} \Rightarrow \mathit{prop}$ in terms of a proposition A that depends on an additional type argument, which is essentially a predicate on types.

$\boxed{\text{ML}}$

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 $\mathit{TERM}\ t$.

`Drule.dest_term` recovers term t from $\mathit{TERM}\ t$.

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

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

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 [4], but we allow arbitrary nesting similar to [10]. 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 [6]. 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: thm -> thm
```

`Simplifier.norm_hhf` *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:

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

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

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

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

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):

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

FIXME *elim_resolution*, *dest_resolution*

ML Reference

```
op RS: thm * thm -> thm
op OF: thm * thm list -> thm
```

rule$_1$ *RS* *rule*$_2$ resolves *rule*$_1$ with *rule*$_2$ according to the *resolution* principle explained above. Note that the corresponding rule attribute in the Isar language is called *THEN*.

rule OF rules resolves a list of rules with the first rule, addressing its premises 1, …, *length rules* (operating from last to first). This means the newly emerging premises are all concatenated, without interfering. Also note that compared to *RS*, the rule argument order is swapped: *rule*$_1$ *RS* *rule*$_2$ = *rule*$_2$ *OF* [*rule*$_1$].

Chapter 3

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.

3.1 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:

$$\frac{C}{\#C}\ (\mathit{protect}) \qquad \frac{A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow \#C}{A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow C}\ (\mathit{conclude})$$

ML

Reference

```
Goal.init: cterm -> thm
Goal.finish: Proof.context -> thm -> thm
Goal.protect: 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` *thm* protects the full statement of theorem *thm*.

`Goal.conclude` *thm* removes the goal protection, even if there are pending subgoals.

3.2 Tactics

A *tactic* is a function $goal \to goal^{**}$ that maps a given goal state (represented as a theorem, cf. §3.1) 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 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.

[2] The lack of memoing and the strict nature of SML 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.

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 §3.3.

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* → *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 (§4.3); 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).

ML Reference

```
type tactic = thm -> thm Seq.seq
no_tac: tactic
all_tac: tactic
print_tac: string -> tactic
PRIMITIVE: (thm -> thm) -> tactic
SUBGOAL: (term * int -> tactic) -> int -> tactic
CSUBGOAL: (cterm * int -> tactic) -> int -> tactic
```

`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.

`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` *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.

3.2.1 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: thm list -> int -> tactic
eresolve_tac: thm list -> int -> tactic
dresolve_tac: thm list -> int -> tactic
forward_tac: thm list -> int -> tactic

assume_tac: int -> tactic
eq_assume_tac: int -> tactic

match_tac: thm list -> int -> tactic
ematch_tac: thm list -> int -> tactic
dmatch_tac: thm list -> int -> tactic
```

`resolve_tac` *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` *thms i* performs elim-resolution with the given theorems, which should normally be elimination rules.

`dresolve_tac` *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.

`assume_tac` *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`, and `dmatch_tac` are similar to `resolve_tac`, `eresolve_tac`, and `dresolve_tac`, respectively, but do not instantiate schematic variables in the goal state.

Flexible subgoals are not updated at will, but are left alone. Strictly speaking, matching means to treat the unknowns in the goal state as constants; these tactics merely discard unifiers that would update the goal state.

3.2.2 Explicit instantiation within a subgoal context

The main resolution tactics (§3.2.1) 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 §4.1.

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 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*, τ). 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

```
res_inst_tac: Proof.context -> (indexname * string) list -> thm -> int -> tactic
eres_inst_tac: Proof.context -> (indexname * string) list -> thm -> int -> tactic
dres_inst_tac: Proof.context -> (indexname * string) list -> thm -> int -> tactic
forw_inst_tac: Proof.context -> (indexname * string) list -> thm -> int -> tactic
rename_tac: string list -> int -> tactic
```

`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*.

`eres_inst_tac` is like `res_inst_tac`, but performs elim-resolution.

`dres_inst_tac` is like `res_inst_tac`, but performs destruct-resolution.

`forw_inst_tac` is like `dres_inst_tac` except that the selected assumption is not deleted.

`rename_tac` *names i* renames the innermost parameters of subgoal *i* according to the provided *names* (which need to be distinct indentifiers).

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 §4.3 allows to refer to internal goal structure with explicit context management.

3.3 Tacticals

A *tactical* is a functional combinator for building up complex tactics from simpler ones. Typical tactical perform sequential composition, disjunction (choice), iteration, or goal addressing. Various search strategies may be expressed via tacticals.

FIXME

Chapter 4

Structured proofs

4.1 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: $\Gamma = \alpha$: *type*, x: α, a: $A(x_\alpha)$.

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 $\Gamma = x$: *term*, α: *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 $x\colon \mathit{term}, \alpha\colon \mathit{type} \vdash x_\alpha = x_\alpha$ produces in the first step $x\colon \mathit{term} \vdash x_\alpha = x_\alpha$ for fixed α, and only in the second step $\vdash ?x_{?\alpha} = ?x_{?\alpha}$ for schematic $?x$ and $?\alpha$.

The Isabelle/Isar proof context manages the gory details of term vs. type variables, with high-level principles for moving the frontier between fixed and schematic variables.

The *add_fixes* operation explictly 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 * ctyp) list * (cterm * cterm) list) * thm list) * Proof.context
Variable.focus: cterm -> Proof.context ->
  ((string * cterm) list * cterm) * 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` *B* decomposes the outermost $\bigwedge$ prefix of proposition *B*.

ML Examples

The following example (in theory *Pure*) shows how to work with fixed term and type parameters work with type-inference.

typedecl *foo* — some basic type for testing purposes

```
ML {*
  (*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::foo) ≡ y";
  (*official declaration of u -- propagates constraints etc.*)
  val ctxt2 = ctxt1 |> Variable.declare_term u;
  val t2 = Syntax.read_term ctxt2 "x";  (*x::foo is enforced*)
*}
```

In the above example, the starting context had been 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 ctxt0 = @{context};
  val ([x1, x2, x3], ctxt1) =
    ctxt0 |> Variable.variant_fixes ["x", "x", "x"];
*}
```

Subsequent 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:

```
lemma PROP XXX
proof –
  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"];
  *}
  oops
```

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.

4.2 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:

$$\frac{\Gamma \vdash B}{\Gamma - A \vdash A \Longrightarrow B}\ (\Longrightarrow\text{-}intro)$$

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:

$$\frac{\Gamma \vdash B}{\Gamma - A \vdash \#A \Longrightarrow B}\ (\#\!\Longrightarrow\text{-}intro)$$

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:

$$\frac{\Gamma \vdash B\ x}{\Gamma - (x \equiv t) \vdash B\ t}\ (\equiv\text{-}expand)$$

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: 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
```

`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` 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 `ProofContext.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 {*
  (*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 `ProofContext.export`).

4.3 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. §4.1) and assumptions (cf. §4.2), 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 [11] for the user-level **obtain** and **guess** elements. 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
```

```
Goal.prove: Proof.context -> string list -> term list -> term ->
  ({prems: thm list, context: Proof.context} -> tactic) -> thm
Goal.prove_multi: Proof.context -> 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.

`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_multi` is simular to `Goal.prove`, but states several conclusions simultaneously. The goal is encoded by means of Pure conjunction; `Goal.conjunction_tac` will turn this into a collection of individual subgoals.

`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.

Chapter 5

Concrete syntax and type-checking

FIXME

5.1 Parsing and printing

FIXME

5.2 Checking and unchecking

Chapter 6

Isar language elements

The primary Isar language consists of three main categories of language elements:

1. Proof commands
2. Proof methods
3. Attributes

6.1 Proof commands

FIXME

6.2 Proof methods

FIXME

6.3 Attributes

FIXME

Chapter 7

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 local theory interface is already the standard way of implementing definitional packages in Isabelle.

7.1 Definitional elements

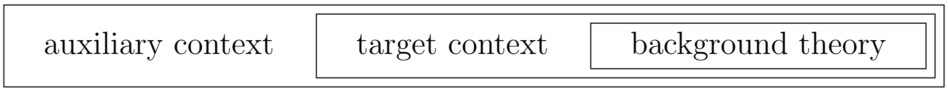
There are separate elements **define** $c \equiv t$ for terms, and **note** $b = thm$ for theorems. Types are treated implicitly, according to Hindley-Milner discipline (cf. §4.1). These definitional primitives essentially act like *let*-bindings within a local context that may already contain earlier *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** $b = \langle B \rangle$

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 $\langle c \equiv t \rangle$, 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 [5, §4–5] for details).

ML Reference

```
type local_theory = Proof.context
Theory_Target.init: string option -> 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
```

`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`.

`Theory_Target.init` *NONE thy* initializes a trivial local theory from the given background theory. Alternatively, *SOME name* may be given to initialize 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.

7.2 Morphisms and declarations

FIXME

Chapter 8

System integration

8.1 Isar toplevel

The Isar toplevel may be considered the centeral hub of the Isabelle/Isar system, where all key components and sub-systems are integrated into a single read-eval-print loop of Isar commands. We shall even incorporate the existing ML toplevel of the compiler and run-time system (cf. §8.2).

Isabelle/Isar departs from the original "LCF system architecture" where ML was really The Meta Language for defining theories and conducting proofs. Instead, ML now only serves as the implementation language for the system (and user extensions), while the specific Isar toplevel supports the concepts of theory and proof development natively. This includes the graph structure of theories and the block structure of proofs, support for unlimited undo, facilities for tracing, debugging, timing, profiling etc.

The toplevel maintains an implicit state, which is transformed by a sequence of transitions – either interactively or in batch-mode. In interactive mode, Isar state transitions are encapsulated as safe transactions, such that both failure and undo are handled conveniently without destroying the underlying draft theory (cf. §1.1.1). In batch mode, transitions operate in a linear (destructive) fashion, such that error conditions abort the present attempt to construct a theory or proof altogether.

The toplevel state is a disjoint sum of empty *toplevel*, or *theory*, or *proof*. On entering the main Isar loop we start with an empty toplevel. A theory is commenced by giving a **theory** header; within a theory we may issue theory commands such as **definition**, or state a **theorem** to be proven. Now we are within a proof state, with 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 theory, which is then updated by storing the resulting fact. Further theory declarations or theorem statements with proofs may follow, until we eventually conclude the theory development by issuing **end**. The resulting theory is then stored within the theory database and we are back to the empty toplevel.

In addition to these proper state transformations, there are also some diagnostic commands for peeking at the toplevel state without modifying it (e.g. **thm**, **term**, **print-cases**).

ML Reference

```
type Toplevel.state
Toplevel.UNDEF: exn
Toplevel.is_toplevel: Toplevel.state -> bool
Toplevel.theory_of: Toplevel.state -> theory
Toplevel.proof_of: Toplevel.state -> Proof.state
Toplevel.debug: bool Unsynchronized.ref
Toplevel.timing: bool Unsynchronized.ref
Toplevel.profiling: int Unsynchronized.ref
```

`Toplevel.state` represents Isar toplevel states, which are normally manipulated through the concept of toplevel transitions only (§8.1.1). Also note that a raw toplevel state is subject to the same linearity restrictions as a theory context (cf. §1.1.1).

`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*, raises `Toplevel.UNDEF` for an empty toplevel state.

`Toplevel.proof_of` *state* selects the Isar proof state if available, otherwise raises `Toplevel.UNDEF`.

`Toplevel.debug := true` makes the toplevel print further details about internal error conditions, exceptions being raised etc.

`Toplevel.timing := true` makes the toplevel print timing information for each Isar command being executed.

`Toplevel.profiling :=` n controls low-level profiling of the underlying ML runtime system. For Poly/ML, $n = 1$ means time and $n = 2$ space profiling.

8.1.1 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, optional source text, as well as flags for interactive-only commands (which issue a warning in batch-mode), printing of result state, etc.

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 the main proof.

Toplevel 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 etc.

ML Reference

```
Toplevel.print: Toplevel.transition -> Toplevel.transition
Toplevel.no_timing: Toplevel.transition -> Toplevel.transition
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.seq) ->
  Toplevel.transition -> Toplevel.transition
Toplevel.end_proof: (bool -> Proof.state -> Proof.context) ->
  Toplevel.transition -> Toplevel.transition
```

`Toplevel.print` *tr* sets the print flag, which causes the toplevel loop to echo the result state (in interactive mode).

`Toplevel.no_timing` *tr* indicates that the transition should never show timing information, e.g. because it is a diagnostic command.

`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 argument that specifies how to apply the proven result to the theory, 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 storing the resulting facts in the context etc.

8.1.2 Toplevel control

There are a few special control commands that modify the behavior the toplevel itself, and only make sense in interactive mode. Under normal circumstances, the user encounters these only implicitly as part of the protocol between the Isabelle/Isar system and a user-interface such as Proof General.

undo follows the three-level hierarchy of empty toplevel vs. theory vs. proof: undo within a proof reverts to the previous proof context, undo after a proof reverts to the theory before the initial goal statement, undo of a theory command reverts to the previous theory value, undo of a theory header discontinues the current theory development and removes it from the theory database (§8.3).

kill aborts the current level of development: kill in a proof context reverts to the theory before the initial goal statement, kill in a theory context aborts the current theory development, removing it from the database.

exit drops out of the Isar toplevel into the underlying ML toplevel (§8.2). The Isar toplevel state is preserved and may be continued later.

quit terminates the Isabelle/Isar process without saving.

8.2 ML toplevel

The ML toplevel provides a read-compile-eval-print loop for ML values, types, structures, and functors. ML declarations operate on the global system state, which consists of the compiler environment plus the values of ML reference variables. There is no clean way to undo ML declarations, except for reverting to a previously saved state of the whole Isabelle process. ML input is either read interactively from a TTY, or from a string (usually within a theory text), or from a source file (usually loaded from a theory).

Whenever the ML toplevel is active, the current Isabelle theory context is passed as an internal reference variable. Thus ML code may access the theory context during compilation, it may even change the value of a theory being under construction — while observing the usual linearity restrictions (cf. §1.1.1).

ML Reference

```
ML_Context.the_generic_context: unit -> Context.generic
Context.>> : (Context.generic -> Context.generic) -> unit
```

`ML_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 `ML_Context.the_generic_context ()` correctly. Recall that evaluation of a function body is delayed until actual runtime. Moreover, persistent ML toplevel bindings to an unfinished theory should be avoided: code should either project out the desired information immediately, or produce an explicit `theory_ref` (cf. §1.1.1).

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

It is very important to note that the above functions are really restricted to the compile time, even though the ML compiler is invoked at runtime! The majority of ML code uses explicit functional arguments of a theory or proof context instead. Thus it may be invoked for an arbitrary context later on, without having to worry about any operational details.

```
Isar.main: unit -> unit
Isar.loop: unit -> unit
Isar.state: unit -> Toplevel.state
Isar.exn: unit -> (exn * string) option
Isar.goal: unit ->
  {context: Proof.context, facts: thm list, goal: thm}
```

`Isar.main ()` invokes the Isar toplevel from ML, initializing an empty toplevel state.

`Isar.loop ()` continues the Isar toplevel with the current state, after having dropped out of the Isar toplevel loop.

`Isar.state ()` and `Isar.exn ()` get current toplevel state and error condition, respectively. This only works after having dropped out of the Isar toplevel loop.

`Isar.goal ()` produces the full Isar goal state, consisting of proof context, facts that have been indicated for immediate use, and the tactical goal according to §3.1.

8.3 Theory database

The theory database maintains a collection of theories, together with some administrative information about their original sources, which are held in an external store (i.e. some directory within the regular file system).

The theory database is organized as a directed acyclic graph; entries are referenced by theory name. Although some additional interfaces allow to include a directory specification as well, this is only a hint to the underlying theory loader. The internal theory name space is flat!

Theory A is associated with the main theory file A`.thy`, which needs to be accessible through the theory loader path. Any number of additional ML source files may be associated with each theory, by declaring these dependencies in the theory header as **uses**, and loading them consecutively within the theory context. The system keeps track of incoming ML sources and associates them with the current theory.

The basic internal actions of the theory database are *update*, *outdate*, and *remove*:

- *update* A introduces a link of A with a *theory* value of the same name; it asserts that the theory sources are now consistent with that value;
- *outdate* A invalidates the link of a theory database entry to its sources, but retains the present theory value;
- *remove* A deletes entry A from the theory database.

These actions are propagated to sub- or super-graphs of a theory entry as expected, in order to preserve global consistency of the state of all loaded theories with the sources of the external store. This implies certain causalities between actions: *update* or *outdate* of an entry will *outdate* all descendants; *remove* will *remove* all descendants.

There are separate user-level interfaces to operate on the theory database directly or indirectly. The primitive actions then just happen automatically while working with the system. In particular, processing a theory header **theory** A **imports** $B_1 \ldots B_n$ **begin** ensures that the sub-graph of the collective imports $B_1 \ldots B_n$ is up-to-date, too. Earlier theories are reloaded as required, with *update* actions proceeding in topological order according to theory dependencies. There may be also a wave of implied *outdate* actions for derived theory nodes until a stable situation is achieved eventually.

ML Reference

```
theory: string -> theory
use_thy: string -> unit
use_thys: string list -> unit
Thy_Info.touch_thy: string -> unit
Thy_Info.remove_thy: string -> unit

Thy_Info.begin_theory: ... -> bool -> theory
Thy_Info.end_theory: theory -> unit
Thy_Info.register_theory: theory -> unit

datatype action = Update | Outdate | Remove
Thy_Info.add_hook: (Thy_Info.action -> string -> unit) -> unit
```

`theory` *A* retrieves the theory value presently associated with name *A*. Note that the result might be outdated.

`use_thy` *A* ensures that theory *A* is fully up-to-date wrt. the external file store, reloading outdated ancestors as required. In batch mode, the simultaneous `use_thys` should be used exclusively.

`use_thys` is similar to `use_thy`, but handles several theories simultaneously. Thus it acts like processing the import header of a theory, without performing the merge of the result. By loading a whole sub-graph of theories like that, the intrinsic parallelism can be exploited by the system, to speedup loading.

`Thy_Info.touch_thy` *A* performs and *outdate* action on theory *A* and all descendants.

`Thy_Info.remove_thy` *A* deletes theory *A* and all descendants from the theory database.

`Thy_Info.begin_theory` is the basic operation behind a **theory** header declaration. This ML function is normally not invoked directly.

`Thy_Info.end_theory` concludes the loading of a theory proper and stores the result in the theory database.

`Thy_Info.register_theory` *text thy* registers an existing theory value with the theory loader database. There is no management of associated sources.

`Thy_Info.add_hook` *f* registers function *f* as a hook for theory database actions. The function will be invoked with the action and theory name being involved;

thus derived actions may be performed in associated system components, e.g. maintaining the state of an editor for the theory sources.

The kind and order of actions occurring in practice depends both on user interactions and the internal process of resolving theory imports. Hooks should not rely on a particular policy here! Any exceptions raised by the hook are ignored.

Appendix A

Advanced ML programming

A.1 Style

Like any style guide, also this one should not be interpreted dogmatically, but with care and discernment. It consists of a collection of recommendations which have been turned out useful over long years of Isabelle system development and are supposed to support writing of readable and managable code. Special cases encourage derivations, as far as you know what you are doing. [1]

fundamental law of programming Whenever writing code, keep in mind: A program is written once, modified ten times, and read hundred times. So simplify its writing, always keep future modifications in mind, and never jeopardize readability. Every second you hesitate to spend on making your code more clear you will have to spend ten times understanding what you have written later on.

white space matters Treat white space in your code as if it determines the meaning of code.

- The space bar is the easiest key to find on the keyboard, press it as often as necessary. `2 + 2` is better than `2+2`, likewise `f (x, y)` is better than `f(x,y)`.
- Restrict your lines to 80 characters. This will allow you to keep the beginning of a line in view while watching its end.[2]
- Ban tabulators; they are a context-sensitive formatting feature and likely to confuse anyone not using your favorite editor.[3]

[1] This style guide is loosely based on http://caml.inria.fr/resources/doc/guides/guidelines.en.html

[2] To acknowledge the lax practice of text editing these days, we tolerate as much as 100 characters per line, but anything beyond 120 is not considered proper source text.

[3] Some modern programming language even forbid tabulators altogether according to the formal syntax definition.

- Get rid of trailing whitespace. Instead, do not suppress a trailing newline at the end of your files.
- Choose a generally accepted style of indentation, then use it systematically throughout the whole application. An indentation of two spaces is appropriate. Avoid dangling indentation.

cut-and-paste succeeds over copy-and-paste *Never* copy-and-paste code when programming. If you need the same piece of code twice, introduce a reasonable auxiliary function (if there is no such function, very likely you got something wrong). Any copy-and-paste will turn out to be painful when something has to be changed later on.

comments are a device which requires careful thinking before using it. The best comment for your code should be the code itself. Prefer efforts to write clear, understandable code over efforts to explain nasty code.

functional programming is based on functions Model things as abstract as appropriate. Avoid unnecessarrily concrete datatype representations. For example, consider a function taking some table data structure as argument and performing lookups on it. Instead of taking a table, it could likewise take just a lookup function. Accustom your way of coding to the level of expressiveness a functional programming language is giving onto you.

tuples are often in the way. When there is no striking argument to tuple function arguments, just write your function curried.

telling names Any name should tell its purpose as exactly as possible, while keeping its length to the absolutely necessary minimum. Always give the same name to function arguments which have the same meaning. Separate words by underscores (`int_of_string`, not `intOfString`).[4]

A.2 Thread-safe programming

Recent versions of Poly/ML (5.2.1 or later) support robust multithreaded execution, based on native operating system threads of the underlying platform. Thus threads will actually be executed in parallel on multi-core systems. A

[4] Some recent tools for Emacs include special precautions to cope with bumpy names in *camelCase*, e.g. for improved on-screen readability. It is easier to abstain from using such names in the first place.

speedup-factor of approximately 1.5–3 can be expected on a regular 4-core machine.[5] Threads also help to organize advanced operations of the system, with explicit communication between sub-components, real-time conditions, time-outs etc.

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 immediately available to other threads, without requiring untyped character streams, awkward serialization etc.

On the other hand, some programming guidelines need to be observed in order to make unprotected parallelism work out smoothly. While the ML system implementation is responsible to maintain basic integrity of the representation of ML values in memory, the application programmer needs to ensure that multithreaded execution does not break the intended semantics.

Critical shared resources. Actually only those parts outside the purely functional world of ML are critical. In particular, this covers

- global references (or arrays), i.e. those that persist over several invocations of associated operations,[6]
- direct I/O on shared channels, notably *stdin*, *stdout*, *stderr*.

The majority of tools implemented within the Isabelle/Isar framework will not require any of these critical elements: nothing special needs to be observed when staying in the purely functional fragment of ML. Note that output via the official Isabelle channels does not count as direct I/O, so the operations `writeln`, `warning`, `tracing` etc. are safe.

Moreover, ML bindings within the toplevel environment (`type`, `val`, `structure` etc.) due to run-time invocation of the compiler are also safe, because Isabelle/Isar manages this as part of the theory or proof context.

Multithreading in Isabelle/Isar. The theory loader automatically exploits the overall parallelism of independent nodes in the development graph, as well as the inherent irrelevance of proofs for goals being fully specified in advance. This means, checking of individual Isar proofs is parallelized by default. Beyond that, very sophisticated proof tools may use local parallelism

[5] There is some inherent limitation of the speedup factor due to garbage collection, which is still sequential. It helps to provide initial heap space generously, using the `-H` option of Poly/ML.

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

internally, via the general programming model of "future values" (see also ~~/src/Pure/Concurrent/future.ML).

Any ML code that works relatively to the present background theory is already safe. Contextual data may be easily stored within the theory or proof context, thanks to the generic data concept of Isabelle/Isar (see §1.1.4). This greatly diminishes the demand for global state information in the first place.

In rare situations where actual mutable content needs to be manipulated, Isabelle provides a single *critical section* that may be entered while preventing any other thread from doing the same. Entering the critical section without contention is very fast, and several basic system operations do so frequently. This also means that each thread should leave the critical section quickly, otherwise parallel execution performance may degrade significantly.

Despite this potential bottle-neck, centralized locking is convenient, because it prevents deadlocks without demanding special precautions. Explicit communication demands other means, though. The high-level abstraction of synchronized variables ~~/src/Pure/Concurrent/synchronized.ML enables parallel components to communicate via shared state; see also ~~/src/Pure/Concurrent/mailbox.ML as canonical example.

Good conduct of impure programs. The following guidelines enable non-functional programs to participate in multithreading.

- Minimize global state information. Using proper theory and proof context data will actually return to functional update of values, without any special precautions for multithreading. Apart from the fully general functors for theory and proof data (see §1.1.4) there are drop-in replacements that emulate primitive references for common cases of *configuration options* for type `bool/int/string` (see structure `Config` and `Attrib.config_bool` etc.), and lists of theorems (see functor `Named_Thms`).

- Keep components with local state information *re-entrant*. Instead of poking initial values into (private) global references, create a new state record on each invocation, and pass that through any auxiliary functions of the component. The state record may well contain mutable references, without requiring any special synchronizations, as long as each invocation sees its own copy. Occasionally, one might even return to plain functional updates on non-mutable record values here.

- Isolate process configuration flags. The main legitimate application of global references is to configure the whole process in a certain way, essentially affecting all threads. A typical example is the `show_types` flag, which tells the pretty printer to output explicit type information for terms. Such flags usually do not affect the functionality of the core system, but only the view being presented to the user.

Occasionally, such global process flags are treated like implicit arguments to certain operations, by using the `setmp_CRITICAL` combinator for safe temporary assignment. Its traditional purpose was to ensure proper recovery of the original value when exceptions are raised in the body, now the functionality is extended to enter the *critical section* (with its usual potential of degrading parallelism).

Note that recovery of plain value passing semantics via `setmp_CRITICAL` *ref value* assumes that this *ref* is exclusively manipulated within the critical section. In particular, any persistent global assignment of *ref* := *value* needs to be marked critical as well, to prevent intruding another threads local view, and a lost-update in the global scope, too.

Recall that in an open "LCF-style" system like Isabelle/Isar, the user participates in constructing the overall environment. This means that state-based facilities offered by one component will require special caution later on. So minimizing critical elements, by staying within the plain value-oriented view relative to theory or proof contexts most of the time, will also reduce the chance of mishaps occurring to end-users.

ML Reference

```
NAMED_CRITICAL: string -> (unit -> 'a) -> 'a
CRITICAL: (unit -> 'a) -> 'a
setmp_CRITICAL: 'a Unsynchronized.ref -> 'a -> ('b -> 'c) -> 'b -> 'c
```

`NAMED_CRITICAL` *name f* evaluates *f* () while staying within the critical section of Isabelle/Isar. No other thread may do so at the same time, but non-critical parallel execution will continue. The *name* argument serves for diagnostic purposes and might help to spot sources of congestion.

`CRITICAL` is the same as `NAMED_CRITICAL` with empty name argument.

`setmp_CRITICAL` *ref value f x* evaluates *f x* while staying within the critical section and having *ref* := *value* assigned temporarily. This recovers a value-passing semantics involving global references, regardless of exceptions or concurrency.

Appendix B

Basic library functions

Beyond the proposal of the SML/NJ basis library, Isabelle comes with its own library, from which selected parts are given here. These should encourage a study of the Isabelle sources, in particular files *Pure/library.ML* and *Pure/General/*.ML*.

B.1 Linear transformations

ML

Reference

```
op |> : 'a * ('a -> 'b) -> 'b
```

Many problems in functional programming can be thought of as linear transformations, i.e. a caluclation starts with a particular value `x : foo` which is then transformed by application of a function `f : foo -> foo`, continued by an application of a function `g : foo -> bar`, and so on. As a canoncial example, take functions enriching a theory by constant declararion and primitive definitions:

```
Sign.declare_const: (binding * typ) * mixfix
  -> theory -> term * theory
Thm.add_def: bool -> bool -> binding * term -> theory -> (string * thm) * theory
```

Written with naive application, an addition of constant *bar* with type $foo \Rightarrow foo$ and a corresponding definition $bar \equiv \lambda x.\ x$ would look like:

```
(fn (t, thy) => Thm.add_def false false
  (Binding.name "bar_def", Logic.mk_equals (t, @{term "%x. x"})) thy)
    (Sign.declare_const
      ((Binding.name "bar", @{typ "foo => foo"}), NoSyn) thy)
```

With increasing numbers of applications, this code gets quite unreadable. Further, it is unintuitive that things are written down in the opposite order as they actually “happen”.

At this stage, Isabelle offers some combinators which allow for more convenient notation, most notably reverse application:

```
thy
|> Sign.declare_const ((Binding.name "bar", @{typ "foo => foo"}), NoSyn)
|> (fn (t, thy) => thy
|> Thm.add_def false false
     (Binding.name "bar_def", Logic.mk_equals (t, @{term "%x. x"})))
```

ML Reference

```
op |-> : (’c * ’a) * (’c -> ’a -> ’b) -> ’b
op |>> : (’a * ’c) * (’a -> ’b) -> ’b * ’c
op ||> : (’c * ’a) * (’a -> ’b) -> ’c * ’b
op ||>> : (’c * ’a) * (’a -> ’d * ’b) -> (’c * ’d) * ’b
```

Usually, functions involving theory updates also return side results; to use these conveniently, yet another set of combinators is at hand, most notably `op |->` which allows curried access to side results:

```
thy
|> Sign.declare_const ((Binding.name "bar", @{typ "foo => foo"}), NoSyn)
|-> (fn t => Thm.add_def false false
      (Binding.name "bar_def", Logic.mk_equals (t, @{term "%x. x"})))
```

`op |>>` allows for processing side results on their own:

```
thy
|> Sign.declare_const ((Binding.name "bar", @{typ "foo => foo"}), NoSyn)
|>> (fn t => Logic.mk_equals (t, @{term "%x. x"}))
|-> (fn def => Thm.add_def false false (Binding.name "bar_def", def))
```

Orthogonally, `op ||>` applies a transformation in the presence of side results which are left unchanged:

```
thy
|> Sign.declare_const ((Binding.name "bar", @{typ "foo => foo"}), NoSyn)
||> Sign.add_path "foobar"
|-> (fn t => Thm.add_def false false
      (Binding.name "bar_def", Logic.mk_equals (t, @{term "%x. x"})))
||> Sign.restore_naming thy
```

`op ||>>` accumulates side results:

```
thy
|> Sign.declare_const ((Binding.name "bar", @{typ "foo => foo"}), NoSyn)
||>> Sign.declare_const ((Binding.name "foobar", @{typ "foo => foo"}), NoSyn)
|-> (fn (t1, t2) => Thm.add_def false false
      (Binding.name "bar_def", Logic.mk_equals (t1, t2)))
```

ML Reference

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

This principles naturally lift to *lists* using the `fold` and `fold_map` combinators. The first lifts a single function

`f : 'a -> 'b -> 'b` to `'a list -> 'b -> 'b`

such that

```
y |> fold f [x1, x2, ..., x_n]
  ⇝ y |> f x1 |> f x2 |> ... |> f x_n
```

The second accumulates side results in a list by lifting a single function

`f : 'a -> 'b -> 'c * 'b` to `'a list -> 'b -> 'c list * 'b`

such that

```
y |> fold_map f [x1, x2, ..., x_n]
  ⇝ y |> f x1 ||>> f x2 ||>> ... ||>> f x_n
      ||> (fn ((z1, z2), ..., z_n) => [z1, z2, ..., z_n])
```

Example:

```
let
  val consts = ["foo", "bar"];
in
  thy
  |> fold_map (fn const => Sign.declare_const
       ((Binding.name const, @{typ "foo => foo"}), NoSyn)) consts
  |>> map (fn t => Logic.mk_equals (t, @{term "%x. x"}))
  |-> (fn defs => fold_map (fn def =>
       Thm.add_def false false (Binding.empty, def)) defs)
end
```

ML Reference

```
op #> : ('a -> 'b) * ('b -> 'c) -> 'a -> 'c
op #-> : ('a -> 'c * 'b) * ('c -> 'b -> 'd) -> 'a -> 'd
op #>> : ('a -> 'c * 'b) * ('c -> 'd) -> 'a -> 'd * 'b
op ##> : ('a -> 'c * 'b) * ('b -> 'd) -> 'a -> 'c * 'd
op ##>> : ('a -> 'c * 'b) * ('b -> 'e * 'd) -> 'a -> ('c * 'e) * 'd
```

All those linear combinators also exist in higher-order variants which do not expect a value on the left hand side but a function.

ML Reference

```
op ` : ('b -> 'a) -> 'b -> 'a * 'b
tap: ('b -> 'a) -> 'b -> 'b
```

These operators allow to "query" a context in a series of context transformations:

```
thy
|> tap (fn _ => writeln "now adding constant")
|> Sign.declare_const ((Binding.name "bar", @{typ "foo => foo"}), NoSyn)
||>> ‘(fn thy => Sign.declared_const thy
         (Sign.full_name thy (Binding.name "foobar")))
|-> (fn (t, b) => if b then I
       else Sign.declare_const
         ((Binding.name "foobar", @{typ "foo => foo"}), NoSyn) #> snd)
```

B.2 Options and partiality

ML

Reference

```
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
try: ('a -> 'b) -> 'a -> 'b option
can: ('a -> 'b) -> 'a -> bool
```

Standard selector functions on *option*s are provided. The `try` and `can` functions provide a convenient interface for handling exceptions – both take as arguments a function `f` together with a parameter `x` and handle any exception during the evaluation of the application of `f` to `x`, either return a lifted result (`NONE` on failure) or a boolean value (`false` on failure).

B.3 Common data structures

B.3.1 Lists (as set-like data structures)

```
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
merge: ('a * 'a -> bool) -> 'a list * 'a list -> 'a list
```

Lists are often used as set-like data structures – set-like in the sense that they support a notion of `member`-ship, `insert`-ing and `remove`-ing, but are order-sensitive. This is convenient when implementing a history-like mechanism: `insert` adds an element *to the front* of a list, if not yet present; `remove` removes *all* occurences of a particular element. Correspondingly `merge` implements a a merge on two lists suitable for merges of context data (§1.1.1).

Functions are parametrized by an explicit equality function to accomplish overloaded equality; in most cases of monomorphic equality, writing `op =` should suffice.

B.3.2 Association lists

```
exception AList.DUP
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.default: ('a * 'a -> bool) -> ('a * 'b) -> ('a * 'b) list -> ('a * 'b) list
AList.delete: ('a * 'b -> bool) -> 'a -> ('b * 'c) list -> ('b * 'c) list
AList.map_entry: ('a * 'b -> bool) -> 'a
    -> ('c -> 'c) -> ('b * 'c) list -> ('b * 'c) list
AList.map_default: ('a * 'a -> bool) -> 'a * 'b -> ('b -> 'b)
    -> ('a * 'b) list -> ('a * 'b) list
AList.join: ('a * 'a -> bool) -> ('a -> 'b * 'b -> 'b) (*exception DUP*)
    -> ('a * 'b) list * ('a * 'b) list -> ('a * 'b) list (*exception AList.DUP*)
AList.merge: ('a * 'a -> bool) -> ('b * 'b -> bool)
    -> ('a * 'b) list * ('a * 'b) list -> ('a * 'b) list (*exception AList.DUP*)
```

Association lists can be seens as an extension of set-like lists: on the one hand, they may be used to implement finite mappings, on the other hand, they remain order-sensitive and allow for multiple key-value-pair with the same key: `AList.lookup` returns the *first* value corresponding to a particular key, if present. `AList.update` updates the *first* occurence of a particular key; if no such key exists yet, the key-value-pair is added *to the front*. `AList.delete` only deletes the *first* occurence of a key. `AList.merge` provides an operation suitable for merges of context data (§1.1.1), where an equality parameter on values determines whether a merge should be considered a conflict. A slightly generalized operation if implementend by the `AList.join` function which allows for explicit conflict resolution.

B.3.3 Tables

```
type 'a Symtab.table
exception Symtab.DUP of string
exception Symtab.SAME
exception Symtab.UNDEF of string
Symtab.empty: 'a Symtab.table
Symtab.lookup: 'a Symtab.table -> string -> 'a option
Symtab.defined: 'a Symtab.table -> string -> bool
Symtab.update: (string * 'a) -> 'a Symtab.table -> 'a Symtab.table
Symtab.default: string * 'a -> 'a Symtab.table -> 'a Symtab.table
Symtab.delete: string
    -> 'a Symtab.table -> 'a Symtab.table (*exception Symtab.UNDEF*)
Symtab.map_entry: string -> ('a -> 'a)
    -> 'a Symtab.table -> 'a Symtab.table
Symtab.map_default: (string * 'a) -> ('a -> 'a)
    -> 'a Symtab.table -> 'a Symtab.table
Symtab.join: (string -> 'a * 'a -> 'a) (*exception Symtab.DUP/Symtab.SAME*)
    -> 'a Symtab.table * 'a Symtab.table
    -> 'a Symtab.table (*exception Symtab.DUP*)
Symtab.merge: ('a * 'a -> bool)
    -> 'a Symtab.table * 'a Symtab.table
    -> 'a Symtab.table (*exception Symtab.DUP*)
```

Tables are an efficient representation of finite mappings without any notion of order; due to their efficiency they should be used whenever such pure finite mappings are neccessary.

The key type of tables must be given explicitly by instantiating the `Table` functor which takes the key type together with its `order`; for convience, we restrict here to the `Symtab` instance with `string` as key type.

Most table functions correspond to those of association lists.

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] 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.

[4] G. Gentzen. Untersuchungen über das logische Schließen. *Math. Zeitschrift*, 1935.

[5] 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.

[6] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4), 1991.

[7] T. Nipkow and C. Prehofer. Type reconstruction for type classes. *Journal of Functional Programming*, 5(2):201–224, 1995.

[8] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[9] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

[10] P. Schroeder-Heister, editor. *Extensions of Logic Programming*, LNAI 475. Springer, 1991.

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

Index