# My Personal, Ecclectic Isabelle Programming Manual

**Version : Isabelle 2019**

Burkhart Wolff

August 19, 2019

# Contents


**1 Introduction** **7**

**2 SML and Fundamental SML libraries** **9**
- 2.1 ML, Text and Antiquotations . . . 9
- 2.2 The Isabelle/Pure bootstrap . . . 10
- 2.3 Elements of the SML library . . . 12

**3 Prover Architecture** **15**
- 3.1 The Nano-Kernel: Contexts, (Theory)-Contexts, (Proof)-Contexts . . . 15
  - 3.1.1 Mechanism 1 : Core Interface. . . . 16
  - 3.1.2 Mechanism 2 : Extending the Global Context $\theta$ by User Data . . . 16
- 3.2 The LCF-Kernel: terms, types, theories, proof_contexts, thms . . . 17
  - 3.2.1 Terms and Types . . . 17
  - 3.2.2 Type-Certification (=checking that a type annotation is consistent) 19
  - 3.2.3 Type-Inference (= inferring consistent type information if possible) 21
  - 3.2.4 Constructing Terms without Type-Inference . . . 21
  - 3.2.5 Theories and the Signature API . . . 22
  - 3.2.6 Thm's and the LCF-Style, "Mikro"-Kernel . . . 22
  - 3.2.7 Theories . . . 24
- 3.3 Advanced Specification Constructs . . . 25
  - 3.3.1 Example . . . 26
- 3.4 Backward Proofs: Tactics, Tacticals and Goal-States . . . 26
- 3.5 The classical goal package . . . 29
  - 3.5.1 Proof Example . . . 29
- 3.6 The Isar Engine . . . 29
  - 3.6.1 Transaction Management in the Isar-Engine : The Toplevel . . . 30
  - 3.6.2 Configuration flags of fixed type in the Isar-engine. . . . 32

**4 Front-End** **33**
- 4.1 Basics: string, bstring and xstring . . . 36
- 4.2 Positions . . . 36
- 4.3 Markup and Low-level Markup Reporting . . . 36
  - 4.3.1 A simple Example . . . 37
  - 4.3.2 A Slightly more Complex Example . . . 38
  - 4.3.3 Environment Structured Reporting . . . 40
- 4.4 The System Lexer and Token Issues . . . 40
  - 4.4.1 Tokens . . . 40

4.4.2 A Lexer Configuration Example . . . 41
4.5 Combinator Parsing . . . 41
4.5.1 Advanced Parser Library . . . 42
4.5.2 Bindings . . . 44
4.5.3 Input streams. . . . 45
4.6 Term Parsing . . . 45
4.6.1 Example . . . 47
4.7 Output: Very Low Level . . . 47
4.8 Output: LaTeX . . . 48
4.9 Inner Syntax Cartouches . . . 50

**5 Conclusion** **53**

## Abstract

While Isabelle is mostly known as part of Isabelle/HOL (an interactive theorem prover), it actually provides a system framework for developing a wide spectrum of applications. A particular strength of the Isabelle framework is the combination of text editing, formal verification, and code generation.

This is a programming-tutorial of Isabelle and Isabelle/HOL, a complementary text to the unfortunately somewhat outdated "The Isabelle Cookbook" in https://nms.kcl.ac.uk/christian.urban/Cookbook/. The reader is encouraged not only to consider the generated .pdf, but also consult the loadable version in Isabelle/jEdit [5]in order to make experiments on the running code.

This text is written itself in Isabelle/Isar using a specific document ontology for technical reports. It is intended to be a "living document", i.e. it is not only used for generating a static, conventional .pdf, but also for direct interactive exploration in Isabelle/jedit. This way, types, intermediate results of computations and checks can be repeated by the reader who is invited to interact with this document. Moreover, the textual parts have been enriched with a maximum of formal content which makes this text re-checkable at each load and easier maintainable.

**Keywords:** LCF-Architecture, Isabelle, SML, PIDE, Programming Guide, Tactic Programming

# 1 Introduction

While Isabelle [3] is mostly known as part of Isabelle/HOL (an interactive theorem prover), it actually provides a system framework for developing a wide spectrum of applications. A particular strength of the Isabelle framework is the combination of text editing, formal verification, and code generation. This is a programming-tutorial of Isabelle and Isabelle/HOL, a complementary text to the unfortunately somewhat outdated "The Isabelle Cookbook" in https://nms.kcl.ac.uk/christian.urban/Cookbook/. The reader is encouraged not only to consider the generated .pdf, but also consult the loadable version in Isabelle/jedit in order to make experiments on the running code. This text is written itself in Isabelle/Isar using a specific document ontology for technical reports. It is intended to be a "living document", i.e. it is not only used for generating a static, conventional .pdf, but also for direct interactive exploration in Isabelle/jedit. This way, types, intermediate results of computations and checks can be repeated by the reader who is invited to interact with this document. Moreover, the textual parts have been enriched with a maximum of formal content which makes this text re-checkable at each load and easier maintainable.

This cookbook roughly follows the Isabelle system architecture shown in Figure 1.1, and, to be more precise, more or less in the bottom-up order.

We start from the basic underlying SML platform over the Kernels to the tactical layer and end with a discussion over the front-end technologies.

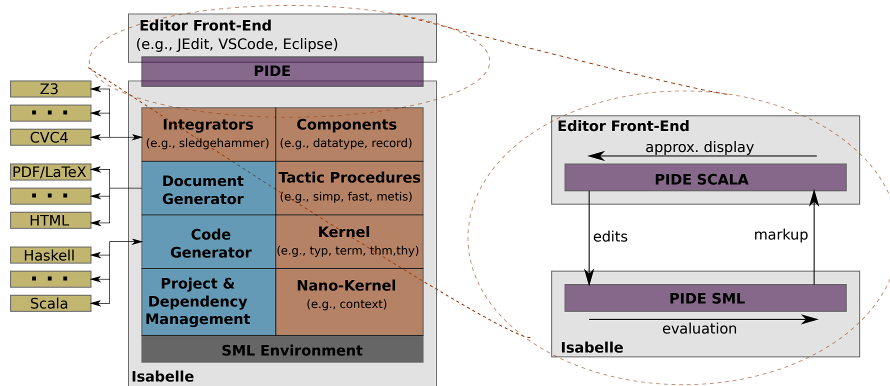


Figure 1.1: The system architecture of Isabelle (left-hand side) and the asynchronous communication between the Isabelle system and the IDE (right-hand side).

# 2 SML and Fundamental SML libraries

## 2.1 ML, Text and Antiquotations

Isabelle is written in SML, the "Standard Meta-Language", which is is an impure functional programming language allowing, in principle, mutable variables and side-effects. The following Isabelle/Isar commands allow for accessing the underlying SML interpreter of Isabelle directly. In the example, a mutable variable X is declared, defined to 0 and updated; and finally re-evaluated leading to output:

```
ML‹ val X = Unsynchronized.ref 0;
    X:= !X + 1;
    X
  ›
```

However, since Isabelle is a platform involving parallel execution, concurrent computing, and, as an interactive environment, involves backtracking / re-evaluation as a consequence of user- interaction, imperative programming is discouraged and nearly never used in the entire Isabelle code-base [1]. The preferred programming style is purely functional:

```
ML‹ fun fac x = if x = 0 then 1 else x * fac(x−1) ;
    fac 10;
  ›
```

```
ML‹ type state = {  a : int,   b : string}
    fun incr-state ({a, b}:state) =  {a=a+1, b=b}
  ›
```

The faculty function is defined and executed; the (sub)-interpreter in Isar works in the conventional read-execute-print loop for each statement separated by a ";". Functions, types, data-types etc. can be grouped to modules (called *structures*) which can be constrained to interfaces (called *signatures*) and even be parametric structures (called *functors*).

The Isabelle/Isar interpreter (called *toplevel* ) is extensible; by a mixture of SML and Isar-commands, domain-specific components can be developed and integrated into the system on the fly. Actually, the Isabelle system code-base consists mainly of SML and `.thy`-files containing such mixtures of Isar commands and SML.

Besides the ML-command used in the above examples, there are a number of commands representing text-elements in Isabelle/Isar; text commands can be interleaved arbitraryly with other commands. Text in text-commands may use LaTeX and is used for type-setting documentations in a kind of literate programming style.

So: the text command for:

*This is a text.*

... is represented in an `.thy` file by:

```
text\isa{\isactrlemph {\isasymopen}This\ is\ a\ text{\isachardot}{\isasymclose}}
```

and desplayed in the Isabelle/jedit front-end at the sceen by:

text-commands, ML- commands (and in principle any other command) can be seen as *quotations* over the underlying SML environment (similar to OCaml or Haskell). Linking these different sorts of quotations with each other and the underlying SML-envirnment is supported via *antiquotations*'s. Generally speaking, antiquotations are a programming technique to specify expressions or patterns in a quotation, parsed in the context of the enclosing expression or pattern where the quotation is. Another way to understand this concept: anti-quotations are "semantic macros" that produce a value for the surrounding context (ML, text, HOL, whatever) depending on local arguments and the underlying logical context.

The way an antiquotation is specified depends on the quotation expander: typically a specific character and an identifier, or specific parentheses and a complete expression or pattern.

In Isabelle documentations, antiquotation's were heavily used to enrich literate explanations and documentations by "formal content", i.e. machine-checked, typed references to all sorts of entities in the context of the interpreting environment. Formal content allows for coping with sources that rapidly evolve and were developed by distributed teams as is typical in open-source developments. A paradigmatic example for antiquotation in Texts and Program snippets is here:

[1], `$ISABELLE_HOME/src/Pure/ROOT.ML`

**ML**‹ *val x = @{thm refl};*
*val y = @{term A ⟶ B}*
*val y = @{typ 'a list}*
›

... which we will describe in more detail later.

In a way, literate specification attempting to maximize its formal content is a way to ensure "Agile Development" in a (theory)-document development, at least for its objectives, albeit not for its popular methods and processes like SCRUM.

**A maximum of formal content inside text documentation also ensures the consistency of this present text with the underlying Isabelle version.**

## 2.2 The Isabelle/Pure bootstrap

It is instructive to study the fundamental bootstrapping sequence of the Isabelle system; it is written in the Isar format and gives an idea of the global module dependencies: `$ISABELLE_HOME/src/Pure/ROOT.ML`. Loading this file (for example by hovering over

[1]sdf

Figure 2.1: A text-element as presented in Isabelle/jedit.

this hyperlink in the antiquotation holding control or command key in Isabelle/jedit and activating it) allows the Isabelle IDE to support hyperlinking *inside* the Isabelle source.

The bootstrapping sequence is also reflected in the following diagram Figure 1.1.

## 2.3 Elements of the SML library

;

Elements of the `$ISABELLE_HOME/src/Pure/General/basics.ML` SML library are basic exceptions. Note that exceptions should be catched individually, uncatched exceptions except those generated by the specific "error" function are discouraged in Isabelle source programming since they might produce races. Finally, a number of commonly used "squigglish" combinators is listed:

```
ML‹ Bind      : exn;
    Chr       : exn;
    Div       : exn;
    Domain    : exn;
    Fail      : string -> exn;
    Match     : exn;
    Overflow  : exn;
    Size      : exn;
    Span      : exn;
    Subscript : exn;

    exnName : exn -> string ; (* -- very interesting to query an unknown exception  *)
    exnMessage : exn -> string ;›
```

```
ML‹
op ! : 'a Unsynchronized.ref -> 'a;
op := : ('a Unsynchronized.ref * 'a) -> unit;

op #> : ('a -> 'b) * ('b -> 'c) -> 'a -> 'c; (* reversed function composition *)
op o : (('b -> 'c) * ('a -> 'b)) -> 'a -> 'c;
op |-- : ('a -> 'b * 'c) * ('c -> 'd * 'e) -> 'a -> 'd * 'e;
op --| : ('a -> 'b * 'c) * ('c -> 'd * 'e) -> 'a -> 'b * 'e;
op -- : ('a -> 'b * 'c) * ('c -> 'd * 'e) -> 'a -> ('b * 'd) * 'e;
op ? : bool * ('a -> 'a) -> 'a -> 'a;
ignore: 'a -> unit;
op before : ('a * unit) -> 'a;
I: 'a -> 'a;
K: 'a -> 'b -> 'a
›
```

Some basic examples for the programming style using these combinators can be found in the "The Isabelle Cookbook" section 2.3.

An omnipresent data-structure in the Isabelle SML sources are tables implemented in `$ISABELLE_HOME/src/Pure/General/table.ML`. These generic tables are presented in

an efficient purely functional implementation using balanced 2-3 trees. Key operations are:

**ML**‹
```
signature TABLE-reduced =
sig
  type key
  type 'a table
  exception DUP of key
  exception SAME
  exception UNDEF of key
  val empty: 'a table
  val dest: 'a table -> (key * 'a) list
  val keys: 'a table -> key list
  val lookup-key: 'a table -> key -> (key * 'a) option
  val lookup: 'a table -> key -> 'a option
  val defined: 'a table -> key -> bool
  val update: key * 'a -> 'a table -> 'a table
  (* ... *)
end
```
›

... where `key` is usually just a synonym for string.

# 3 Prover Architecture

## 3.1 The Nano-Kernel: Contexts, (Theory)-Contexts, (Proof)-Contexts

What I call the 'Nano-Kernel' in Isabelle can also be seen as an acyclic theory graph. The meat of it can be found in the file `$ISABELLE_HOME/src/Pure/context.ML`. My notion is a bit criticisable since this component, which provides the type of `theory` and `Proof.context`, is not that Nano after all. However, these type are pretty empty place-holders at that level and the content of `$ISABELLE_HOME/src/Pure/theory.ML` is registered much later. The sources themselves mention it as "Fundamental Structure". In principle, theories and proof contexts could be REGISTERED as user data inside contexts. The chosen specialization is therefore an acceptable meddling of the abstraction "Nano-Kernel" and its application context: Isabelle.

Makarius himself says about this structure:

"Generic theory contexts with unique identity, arbitrarily typed data, monotonic development graph and history support. Generic proof contexts with arbitrarily typed data."

In my words: a context is essentially a container with

- an id
- a list of parents (so: the graph structure)
- a time stamp and
- a sub-context relation (which uses a combination of the id and the time-stamp to establish this relation very fast whenever needed; it plays a crucial role for the context transfer in the kernel.

  A context comes in form of three 'flavours':

- theories : `theory`'s, containing a syntax and axioms, but also user-defined data and configuration information.
- Proof-Contexts: `Proof.context` containing theories but also additional information if Isar goes into prove-mode. In general a richer structure than theories coping also with fixes, facts, goals, in order to support the structured Isar proof-style.
- Generic: `Context.generic`, i.e. the sum of both.

  All context have to be seen as mutable; so there are usually transformations defined on them which are possible as long as a particular protocol (`begin_thy` - `end_thy` etc) are respected.

Contexts come with type user-defined data which is mutable through the entire lifetime of a context.

### 3.1.1 Mechanism 1 : Core Interface.

To be found in `$ISABELLE_HOME/src/Pure/context.ML`:

```
ML‹
Context.parents-of: theory −> theory list;
Context.ancestors-of: theory −> theory list;
Context.proper-subthy : theory ∗ theory −> bool;
Context.Proof: Proof.context −> Context.generic; (∗constructor∗)
Context.proof-of : Context.generic −> Proof.context;
Context.certificate-theory-id : Context.certificate −> Context.theory-id;
Context.theory-name : theory −> string;
Context.map-theory: (theory −> theory) −> Context.generic −> Context.generic;
›
```

ML structure `Proof_Context`:

```
ML‹
 Proof-Context.init-global: theory −> Proof.context;
 Context.Proof: Proof.context −> Context.generic; (∗ the path to a generic Context ∗)
 Proof-Context.get-global: theory −> string −> Proof.context
›
```

### 3.1.2 Mechanism 2 : Extending the Global Context $\theta$ by User Data

A central mechanism for constructing user-defined data is by the `Generic_Data` SML functor. A plugin needing some data `T` and providing it with implementations for an `empty`, and operations `merge` and `extend`, can construct a lense with operations `get` and `put` that attach this data into the generic system context. Rather than using unsynchronized SML mutable variables, this is the mechanism to introduce component local data in Isabelle, which allows to manage this data for the necessary backtrack- and synchronization features in the pervasively parallel evaluation framework that Isabelle as a system represents.

```
ML ‹
datatype X = mt
val init = mt;
val ext = I
fun merge (X,Y) = mt

structure Data = Generic-Data
(
  type T = X
  val empty  = init
  val extend = ext
  val merge  = merge
);
```

```
Data.get : Context.generic -> Data.T;
Data.put : Data.T -> Context.generic -> Context.generic;
Data.map : (Data.T -> Data.T) -> Context.generic -> Context.generic;
(* there are variants to do this on theories ... *)
›
```

## 3.2 The LCF-Kernel: terms, types, theories, proof_contexts, thms

The classical LCF-style *kernel* is about

1. Types and terms of a typed $\lambda$-Calculus including constant symbols, free variables, $\lambda$-binder and application,
2. An infrastructure to define types and terms, a *signature*, that also assigns to constant symbols types
3. An abstract type of *theorem* and logical operations on them
4. (Isabelle specific): a notion of *theory*, i.e. a container providing a signature and set (list) of theorems.

### 3.2.1 Terms and Types

A basic data-structure of the kernel is `$ISABELLE_HOME/src/Pure/term.ML`

```
ML‹ (* open Term; *)
signature TERM' = sig
  (* ... *)
  type indexname = string * int
  type class = string
  type sort = class list
  type arity = string * sort list * sort
  datatype typ =
    Type  of string * typ list |
    TFree of string * sort |
    TVar  of indexname * sort
  datatype term =
    Const of string * typ |
    Free of string * typ |
    Var of indexname * typ |
    Bound of int |
    Abs of string * typ * term |
    $ of term * term
  exception TYPE of string * typ list * term list
  exception TERM of string * term list
  (* ... *)
```

*end*
⟩

This core-data structure of the Isabelle Kernel is accessible in the Isabelle/ML environment and serves as basis for programmed extensions concerning syntax, type-checking, and advanced tactic programming over kernel primitives and higher API's. There are a number of anti-quotations giving support for this task; since `Const`-names are long-names revealing information of the potentially evolving library structure, the use of anti-quotations leads to a safer programming style of tactics and became therefore standard in the entire Isabelle code-base.

The following examples show how term- and type-level antiquotations are used and that they can both be used for term-construction as well as term-destruction (pattern-matching):

```
ML⟨ val Const (HOL.implies, @{typ bool ⇒ bool ⇒ bool})
           $ Free (A, @{typ bool})
           $ Free (B, @{typ bool})
      = @{term A ⟶ B};

 val HOL.bool = @{type-name bool};

 (* three ways to write type bool:@ *)
 val Type(fun,[s,Type(fun,[@{typ bool},Type(@{type-name bool},[])])]) = @{typ bool ⇒ bool ⇒
bool};

⟩
```

Note that the SML interpreter is configured that he will actually print a type `Type("HOL.bool",[])` just as `"bool": typ`, so a compact notation looking pretty much like a string. This can be confusing at times.

Note, furthermore, that there is a programming API for the HOL-instance of Isabelle: it is contained in `$ISABELLE_HOME/src/HOL/Tools/hologic.ML`. It offers for many operators of the HOL logic specific constructors and destructors:

```
ML⟨
  HOLogic.boolT         : typ;
  HOLogic.mk-Trueprop   : term −> term; (* the embedder of bool to prop fundamenyal for
HOL *)
  HOLogic.dest-Trueprop : term −> term;
  HOLogic.Trueprop-conv : conv −> conv;
  HOLogic.mk-setT       : typ −> typ;  (* ML level type constructor set *)
  HOLogic.dest-setT     : typ −> typ;
  HOLogic.Collect-const : typ −> term;
  HOLogic.mk-Collect    : string * typ * term −> term;
  HOLogic.mk-mem        : term * term −> term;
  HOLogic.dest-mem      : term −> term * term;
  HOLogic.mk-set        : typ −> term list −> term;
  HOLogic.conj-intr     : Proof.context −> thm −> thm −> thm; (* some HOL−level de-
rived−inferences *)
```

```
  HOLogic.conj-elim      : Proof.context -> thm -> thm * thm;
  HOLogic.conj-elims     : Proof.context -> thm -> thm list;
  HOLogic.conj           : term;          (* some ML level logical constructors *)
  HOLogic.disj           : term;
  HOLogic.imp            : term;
  HOLogic.Not            : term;
  HOLogic.mk-not         : term -> term;
  HOLogic.mk-conj        : term * term -> term;
  HOLogic.dest-conj      : term -> term list;
  HOLogic.conjuncts      : term -> term list;
  (* ... *)
⟩
```

### 3.2.2 Type-Certification (=checking that a type annotation is consistent)

```
ML⟨ Type.typ-instance: Type.tsig -> typ * typ -> bool (* raises  TYPE-MATCH *) ⟩
```

there is a joker type that can be added as place-holder during term construction. Jokers can be eliminated by the type inference.

```
ML⟨  Term.dummyT : typ ⟩
```

```
ML⟨
  Sign.typ-instance: theory -> typ * typ -> bool;
  Sign.typ-match: theory -> typ * typ -> Type.tyenv -> Type.tyenv;
  Sign.typ-unify: theory -> typ * typ -> Type.tyenv * int -> Type.tyenv * int;
  Sign.const-type: theory -> string -> typ option;
 Sign.certify-term: theory -> term -> term * typ * int;  (* core routine for CERTIFICATION
of types*)
  Sign.cert-term: theory -> term -> term;                  (* short-cut for the latter *)
  Sign.tsig-of: theory -> Type.tsig                      (* projects the type signature *)
⟩
```

`Sign.typ_match` etc. is actually an abstract wrapper on the structure `Type` which contains the heart of the type inference. It also contains the type substitution type `Type.tyenv` which is is actually a type synonym for `(sort * typ) Vartab.table` which in itself is a synonym for `'a Symtab.table`, so possesses the usual `Symtab.empty` and `Symtab.dest` operations.

Note that *polymorphic variables* are treated like constant symbols in the type inference; thus, the following test, that one type is an instance of the other, yields false:

```
ML⟨
val ty = @{typ 'a option};
val ty' = @{typ int option};

val Type(List.list,[S]) = @{typ ('a) list}; (* decomposition example *)

val false = Sign.typ-instance @{theory}(ty', ty);
⟩
```

In order to make the type inference work, one has to consider *schematic* type variables, which are more and more hidden from the Isar interface. Consequently, the typ

antiquotation above will not work for schematic type variables and we have to construct them by hand on the SML level:

```
ML‹ val t-schematic = Type(List.list,[TVar(('a,0),@{sort HOL.type})]); ›
```

MIND THE "'" !!!

On this basis, the following `Type.tyenv` is constructed:

```
ML‹
val tyenv = Sign.typ-match ( @{theory})
              (t-schematic, @{typ int list})
              (Vartab.empty);
val  [(('a, 0), ([HOL.type], @{typ int}))] = Vartab.dest tyenv;
›
```

Type generalization — the conversion between free type variables and schematic type variables — is apparently no longer part of the standard API (there is a slightly more general replacement in `Term_Subst.generalizeT_same`, however). Here is a way to overcome this by a self-baked generalization function:

```
ML‹
val generalize-typ = Term.map-type-tfree (fn (str,sort)=> Term.TVar((str,0),sort));
val generalize-term = Term.map-types generalize-typ;
val true = Sign.typ-instance @{theory} (ty', generalize-typ ty)
›
```

... or more general variants thereof that are parameterized by the indexes for schematic type variables instead of assuming just `0`.

Example:

```
ML‹val t = generalize-term @{term []}›
```

Now we turn to the crucial issue of type-instantiation and with a given type environment `tyenv`. For this purpose, one has to switch to the low-level interface `Term_Subst`.

```
ML‹
Term-Subst.map-types-same : (typ -> typ) -> term -> term;
Term-Subst.map-aterms-same : (term -> term) -> term -> term;
Term-Subst.instantiate: ((indexname * sort) * typ) list * ((indexname * typ) * term) list ->
term -> term;
Term-Subst.instantiateT: ((indexname * sort) * typ) list -> typ -> typ;
Term-Subst.generalizeT: string list -> int -> typ -> typ;
                        (* this is the standard type generalisation function !!!
                           only type-frees in the string-list were taken into account. *)
Term-Subst.generalize: string list * string list -> int -> term -> term
                        (* this is the standard term generalisation function !!!
                           only type-frees and frees in the string-lists were taken
                           into account. *)
›
```

Apparently, a bizarre conversion between the old-style interface and the new-style `tyenv` is necessary. See the following example.

```
ML‹
val S = Vartab.dest tyenv;
val S' = (map (fn (s,(t,u)) => ((s,t),u)) S) : ((indexname * sort) * typ) list;
        (* it took me quite some time to find out that these two type representations,
           obscured by a number of type-synonyms, where actually identical. *)
val ty = t-schematic;
val ty' = Term-Subst.instantiateT S' t-schematic;
val t = (generalize-term @{term []});

val t' = Term-Subst.map-types-same (Term-Subst.instantiateT S') (t)
(* or alternatively : *)
val t'' = Term.map-types (Term-Subst.instantiateT S') (t)
›
```

A more abstract env for variable management in tactic proofs. A bit difficult to use outside the very closed-up tracks of conventional use...

```
ML‹ Consts.the-const; (* T is a kind of signature ... *)
    Variable.import-terms;
    Vartab.update;›
```

### 3.2.3 Type-Inference (= inferring consistent type information if possible)

Type inference eliminates also joker-types such as `dummyT` and produces instances for schematic type variables where necessary. In the case of success, it produces a certifiable term.

```
ML‹ Type-Infer-Context.infer-types: Proof.context -> term list -> term list ›
```

### 3.2.4 Constructing Terms without Type-Inference

Using `Type_Infer_Context.infer_types` is not quite unproblematic: since the type inference can construct types for largely underspecified terms, it may happen that under some circumstances, tactics and proof-attempts fail since just some internal term representation was too general. A more defensive strategy is already sketched — but neither explicitely mentioned nor worked out in the interface in `HOLogic`. The idea is to have advanced term constructors that construct the right term from the leaves, which were by convention fully type-annotated (so: this does not work for terms with dangling @(ML Bound)'s).

Operations like `HOLogic.mk_prod` or `HOLogic.mk_fst` or `HOLogic.mk_eq` do exactly this by using an internal pure bottom-up type-inference `fastype_of`. The following routines are written in the same style complement the existing API `HOLogic`.

```
ML‹
fun mk-None ty = let val none = const-name ‹Option.option.None›
                     val none-ty = ty --> Type(type-name ‹option›,[ty])
                 in  Const(none, none-ty)
                 end;
fun mk-Some t = let val some = const-name ‹Option.option.Some›
                    val ty = fastype-of t
                    val some-ty = ty --> Type(type-name ‹option›,[ty])
```

```
            in  Const(some, some-ty) $  t
            end;

fun  mk-undefined (@{typ unit}) = Const (const-name ‹Product-Type.Unity›, typ ‹unit›)
   |mk-undefined t              = Const (const-name ‹HOL.undefined›, t)

fun meta-eq-const T = Const (const-name ‹Pure.eq›, T --> T --> propT);

fun mk-meta-eq (t, u) = meta-eq-const (fastype-of t) $ t $ u;

›
```

### 3.2.5 Theories and the Signature API

```
ML‹
Sign.tsig-of : theory -> Type.tsig;
Sign.syn-of  : theory -> Syntax.syntax;
Sign.of-sort : theory -> typ * sort -> bool ;
›
```

### 3.2.6 Thm's and the LCF-Style, "Mikro"-Kernel

The basic constructors and operations on theorems`$ISABELLE_HOME/src/Pure/thm.ML`, a set of derived (Pure) inferences can be found in `$ISABELLE_HOME/src/Pure/drule.ML`.

The main types provided by structure `thm` are certified types `ctyp`, certified terms `cterm`, `thm` as well as conversions `conv`.

```
ML‹
signature BASIC-THM =
sig
  type ctyp
  type cterm
  exception CTERM of string * cterm list
  type thm
  type conv = cterm -> thm
  exception THM of string * int * thm list
end;
›
```

Certification of types and terms on the kernel-level is done by the generators:

```
ML‹
  Thm.global-ctyp-of: theory -> typ -> ctyp;
  Thm.ctyp-of: Proof.context -> typ -> ctyp;
  Thm.global-cterm-of: theory -> term -> cterm;
  Thm.cterm-of: Proof.context -> term -> cterm;
›
```

... which perform type-checking in the given theory context in order to make a type or term "admissible" for the kernel.

We come now to the very heart of the LCF-Kernel of Isabelle, which provides the fundamental inference rules of Isabelle/Pure.

$$\frac{\Theta, \Pi, \Gamma \vdash q : B}{\Theta, \Pi, \Gamma - \{p : A\} \vdash (\lambda p : A.\ q) : (A \Longrightarrow B)} \ (\textit{imp-intro})$$

$$\frac{\Theta_1, \Pi_1, \Gamma_1 \vdash p : (A \Longrightarrow B) \quad \Theta_2, \Pi_2, \Gamma_2 \vdash q : A}{\Theta_1 \cup \Theta_2, \Pi_1 \cup \Pi_2, \Gamma_1 \cup \Gamma_2 \vdash p\ q : B} \ (\textit{imp-elim})$$

$$\frac{\Theta, \Pi, \Gamma \vdash p[x] : B[x] \quad x \notin \mathrm{FV}\,\Gamma}{\Theta, \Pi, \Gamma \vdash (\lambda x.\ p[x]) : (\bigwedge x.\ B[x])} \ (\textit{all-intro})$$

$$\frac{\Theta, \Pi, \Gamma \vdash p : (\bigwedge x.\ B[x])}{\Theta, \Pi, \Gamma \vdash p\ a : B[a]} \ (\textit{all-elim})$$

$$\frac{}{\Theta, \Pi, \{p : A\} \vdash p : A} \ (\textit{assm})$$

$$\frac{(c : A[?\overline{\alpha}]) \in \Theta}{\Theta, \emptyset, \emptyset \vdash c : A[\overline{\tau}]} \ (\textit{axiom})$$

(a) Pure Kernel Inference Rules I

$$\frac{\Theta, \Pi, \Gamma \vdash p[\alpha] : B[\alpha] \quad \alpha \notin \mathrm{TV}\,\Gamma}{\Theta, \Pi, \Gamma \vdash p[?\alpha] : B[?\alpha]} \ (\textit{type-gen})$$

$$\frac{\Theta, \Pi, \Gamma \vdash p[?\alpha] : B[?\alpha]}{\Theta, \Pi, \Gamma \vdash p[\tau] : B[\tau]} \ (\textit{type-inst})$$

(b) Pure Kernel Inference Rules II

Figure 3.1:

Besides a number of destructors on `thm`'s, the abstract data-type `thm` is used for logical objects of the form $\Gamma \vdash_\theta \phi$, where $\Gamma$ represents a set of local assumptions, $\theta$ the "theory" or more precisely the global context in which a formula $\phi$ has been constructed just by applying the following operations representing logical inference rules:

**ML**⟨
(*∗inference rules∗*)
*Thm.assume*: *cterm −> thm*;
*Thm.implies-intr*: *cterm −> thm −> thm*;
*Thm.implies-elim*: *thm −> thm −> thm*;
*Thm.forall-intr*: *cterm −> thm −> thm*;
*Thm.forall-elim*: *cterm −> thm −> thm*;

*Thm.transfer* : *theory −> thm −> thm*;

*Thm.generalize*: *string list ∗ string list −> int −> thm −> thm*;
*Thm.instantiate*: *((indexname∗sort)∗ctyp)list ∗ ((indexname∗typ)∗cterm) list −> thm −> thm*;
⟩

They reflect the Pure logic depicted in a number of presentations such as M. Wenzel, *Parallel Proof Checking in Isabelle/Isar*, PLMMS 2009, or simiular papers. Notated as logical inference rules, these operations were presented as follows:

Note that the transfer rule:

$$\frac{\Gamma \vdash_\theta \phi \qquad \theta \sqsubseteq \theta'}{\Gamma \vdash_{\theta'} \phi}$$

which is a consequence of explicit theories characteristic for Isabelle's LCF-kernel design and a remarkable difference to its sisters HOL-Light and HOL4; instead of transfer, these

systems reconstruct proofs in an enlarged global context instead of taking the result and converting it.

Besides the meta-logical (Pure) implication $_ \Longrightarrow _$, the Kernel axiomatizes also a Pure-Equality $_ \equiv _$ used for definitions of constant symbols:

```
ML‹
  Thm.reflexive: cterm -> thm;
  Thm.symmetric: thm -> thm;
  Thm.transitive: thm -> thm -> thm;
›
```

The operation:

```
ML‹  Thm.trivial: cterm -> thm; ›
```

... produces the elementary tautologies of the form $A \Longrightarrow A$, an operation used to start a backward-style proof.

The elementary conversions are:

```
ML‹
  Thm.beta-conversion: bool -> conv;
  Thm.eta-conversion: conv;
  Thm.eta-long-conversion: conv;
›
```

On the level of `Drule`, a number of higher-level operations is established, which is in part accessible by a number of forward-reasoning notations on Isar-level.

```
ML‹
  op RSN: thm * (int * thm) -> thm;
  op RS: thm * thm -> thm;
  op RLN: thm list * (int * thm list) -> thm list;
  op RL: thm list * thm list -> thm list;
  op MRS: thm list * thm -> thm;
  op OF: thm * thm list -> thm;
  op COMP: thm * thm -> thm;
›
```

### 3.2.7 Theories

This structure yields the datatype `thy` which becomes the content of `Context.theory`. In a way, the LCF-Kernel registers itself into the Nano-Kernel, which inspired me (bu) to this naming.

```
ML‹


(* intern Theory.Thy;

datatype thy = Thy of
 {pos: Position.T,
  id: serial,
  axioms: term Name-Space.table,
```

```
  defs: Defs.T,
  wrappers: wrapper list * wrapper list};

*)

Theory.check: {long: bool} -> Proof.context -> string * Position.T -> theory;

Theory.local-setup: (Proof.context -> Proof.context) -> unit;
Theory.setup: (theory -> theory) -> unit;  (* The thing to extend the table of commands with
parser - callbacks. *)
Theory.get-markup: theory -> Markup.T;
Theory.axiom-table: theory -> term Name-Space.table;
Theory.axiom-space: theory -> Name-Space.T;
Theory.axioms-of: theory -> (string * term) list;
Theory.all-axioms-of: theory -> (string * term) list;
Theory.defs-of: theory -> Defs.T;
Theory.at-begin: (theory -> theory option) -> theory -> theory;
Theory.at-end: (theory -> theory option) -> theory -> theory;
Theory.begin-theory: string * Position.T -> theory list -> theory;
Theory.end-theory: theory -> theory;›
```

## 3.3 Advanced Specification Constructs

Isabelle is built around the idea that system components were built on top of the kernel in order to give the user high-level specification constructs — rather than inside as in the Coq kernel that foresees, for example, data-types and primitive recursors already in the basic λ-term language. Therefore, records, definitions, type-definitions, recursive function definitions are supported by packages that belong to the *components* strata. With the exception of the `Specification.axiomatization` construct, they are all-together built as composition of conservative extensions.

The components are a bit scattered in the architecture. A relatively recent and high-level component (more low-level components such as `Global_Theory.add_defs` exist) for definitions and axiomatizations is here:

```
ML‹
local open Specification
  val -= definition: (binding * typ option * mixfix) option ->
        (binding * typ option * mixfix) list -> term list -> Attrib.binding * term ->
        local-theory -> (term * (string * thm)) * local-theory
  val -= definition': (binding * typ option * mixfix) option ->
        (binding * typ option * mixfix) list ->  term list -> Attrib.binding * term ->
        bool -> local-theory -> (term * (string * thm)) * local-theory
  val -= definition-cmd: (binding * string option * mixfix) option ->
        (binding * string option * mixfix) list -> string list -> Attrib.binding * string ->
         bool -> local-theory -> (term * (string * thm)) * local-theory
  val -= axiomatization: (binding * typ option * mixfix) list ->
        (binding * typ option * mixfix) list -> term list ->
        (Attrib.binding * term) list -> theory -> (term list * thm list) * theory
  val -= axiomatization-cmd: (binding * string option * mixfix) list ->
```

```
      (binding * string option * mixfix) list -> string list ->
      (Attrib.binding * string) list -> theory -> (term list * thm list) * theory
  val -= axiom: Attrib.binding * term -> theory -> thm * theory
  val -= abbreviation: Syntax.mode -> (binding * typ option * mixfix) option ->
      (binding * typ option * mixfix) list -> term -> bool -> local-theory -> local-theory
  val -= abbreviation-cmd: Syntax.mode -> (binding * string option * mixfix) option ->
     (binding * string option * mixfix) list -> string -> bool -> local-theory -> local-theory
in val - = () end
⟩
```

Note that the interface is mostly based on `local_theory`, which is a synonym to `Proof.context`. Need to lift this to a global system transition ? Don't worry, `Named_Target.theory_map: (local_theory -> local_theory) -> theory -> theory` does the trick.

### 3.3.1 Example

Suppose that we want do *definition* $I :: 'a \Rightarrow 'a$ *where* $I\ x = x$ at the ML-level. We construct our defining equation and embed it as a *prop* into Pure.

```
ML⟨ val ty = @{typ 'a}
   val term = HOLogic.mk-eq (Free(I,ty -->ty) $ Free(x, ty), Free(x, ty));
   val term-prop = HOLogic.mk-Trueprop term⟩
```

Recall the notes on defensive term construction wrt. typing in Section 3.2.4. Then the trick is done by:

```
setup⟨
let
fun mk-def name p  =
    let val nameb =  Binding.make(name,p)
        val ty-global = ty --> ty
      val args = (((SOME(nameb,SOME ty-global,NoSyn),(Binding.empty-atts,term-prop)),[]),[])
        val cmd = (fn (((decl, spec), prems), params) =>
                      #2 oo Specification.definition' decl params prems spec)
    in cmd args true
    end;
in  Named-Target.theory-map (mk-def I @{here} )
end⟩
```

**thm** *I-def*

Voilà.

## 3.4 Backward Proofs: Tactics, Tacticals and Goal-States

At this point, we leave the Pure-Kernel and start to describe the first layer on top of it, involving support for specific styles of reasoning and automation of reasoning.

`tactic`'s are in principle *relations* on theorems `thm`; this gives a natural way to represent the fact that HO-Unification (and therefore the mechanism of rule-instantiation) are non-deterministic in principle. Heuristics may choose particular preferences between

the theorems in the range of this relation, but the Isabelle Design accepts this fundamental fact reflected at this point in the prover architecture. This potentially infinite relation is implemented by a function of theorems to lazy lists over theorems, which gives both sufficient structure for heuristic considerations as well as a nice algebra, called `TACTICAL`'s, providing a bottom element `no_tac` (the function that always fails), the top-element `all_tac` (the function that never fails), sequential composition `op THEN`, (serialized) non-deterministic composition `op ORELSE`, conditionals, repetitions over lists, etc. The following is an excerpt of `~~/src/Pure/tactical.ML`:

```
ML‹
signature TACTICAL =
sig
  type tactic = thm -> thm Seq.seq

  val all_tac: tactic
  val no_tac: tactic
  val COND: (thm -> bool) -> tactic -> tactic -> tactic

  val THEN: tactic * tactic -> tactic
  val ORELSE: tactic * tactic -> tactic
  val THEN': ('a -> tactic) * ('a -> tactic) -> 'a -> tactic
  val ORELSE': ('a -> tactic) * ('a -> tactic) -> 'a -> tactic

  val TRY: tactic -> tactic
  val EVERY: tactic list -> tactic
  val EVERY': ('a -> tactic) list -> 'a -> tactic
  val FIRST: tactic list -> tactic
  (* ... *)
end
›
```

The next layer in the architecture describes `tactic`'s, i.e. basic operations on theorems in a backward reasoning style (bottom up development of proof-trees). An initial goal-state for some property $A$ — the *goal* — is constructed via the kernel `Thm.trivial`-operation into $A \Longrightarrow A$, and tactics either refine the premises — the *subgoals* the of this meta-implication — producing more and more of them or eliminate them in subsequent goal-states. Subgoals of the form $[\![B_1;\ B_2;\ A;\ B_3;\ B_4]\!] \Longrightarrow A$ can be eliminated via the `Tactic.assume_tac` - tactic, and a subgoal $C_m$ can be refined via the theorem $[\![E_1;\ E_2;\ E_3]\!] \Longrightarrow C_m$ the `Tactic.resolve_tac` - tactic to new subgoals $E_1$,$E_2$, $E_3$. In case that a theorem used for resolution has no premise $E_i$, the subgoal $C_m$ is also eliminated ("closed").

The following abstract of the most commonly used `tactic`'s drawn from `~~/src/Pure/tactic.ML` looks as follows:

```
ML‹
  (* ... *)
  assume_tac: Proof.context -> int -> tactic;
  compose_tac: Proof.context -> (bool * thm * int) -> int -> tactic;
  resolve_tac: Proof.context -> thm list -> int -> tactic;
```

```
eresolve-tac: Proof.context -> thm list -> int -> tactic;
forward-tac: Proof.context -> thm list -> int -> tactic;
dresolve-tac: Proof.context -> thm list -> int -> tactic;
rotate-tac: int -> int -> tactic;
defer-tac: int -> tactic;
prefer-tac: int -> tactic;
(* ... *)
⟩
```

Note that "applying a rule" is a fairly complex operation in the Extended Isabelle Kernel, i.e. the tactic layer. It involves at least four phases, interfacing a theorem coming from the global context $\theta$ (=theory), be it axiom or derived, into a given goal-state.

- *generalization*. All free variables in the theorem were replaced by schematic variables. For example, $x + y = y + x$ is converted into $?x + ?y = ?y + ?x$. By the way, type variables were treated equally.

- *lifting over assumptions*. If a subgoal is of the form: $[\![B_1;\ B_2]\!] \Longrightarrow A$ and we have a theorem $[\![D_1;\ D_2]\!] \Longrightarrow A$, then before applying the theorem, the premisses were *lifted* resulting in the logical refinement: $[\![[\![B_1;\ B_2]\!] \Longrightarrow D_1;\ [\![B_1;\ B_2]\!] \Longrightarrow D_2]\!] \Longrightarrow A$. Now, `resolve_tac`, for example, will replace the subgoal $[\![B_1;\ B_2]\!] \Longrightarrow A$ by the subgoals $[\![B_1;\ B_2]\!] \Longrightarrow D_1$ and $[\![B_1;\ B_2]\!] \Longrightarrow D_2$. Of course, if the theorem wouldn't have assumptions $D_1$ and $D_2$, the subgoal $A$ would be replaced by **nothing**, i.e. deleted.

- *lifting over parameters*. If a subgoal is meta-quantified like in: $\bigwedge x\ y\ z.\ A\ x\ y\ z$, then a theorem like $[\![D_1;\ D_2]\!] \Longrightarrow A$ is *lifted* to $\bigwedge x\ y\ z.\ [\![D_1';\ D_2']\!] \Longrightarrow A'$, too. Since free variables occurring in $D_1$, $D_2$ and $A$ have been replaced by schematic variables (see phase one), they must be replaced by parameterized schematic variables, i. e. a kind of skolem function. For example, $?x + ?y = ?y + ?x$ would be lifted to $!!\ x\ y\ z.\ ?x\ x\ y\ z + ?y\ x\ y\ z = ?y\ x\ y\ z + ?x\ x\ y\ z$. This way, the lifted theorem can be instantiated by the parameters $x\ y\ z$ representing "fresh free variables" used for this sub-proof. This mechanism implements their logically correct bookkeeping via kernel primitives.

- *Higher-order unification (of schematic type and term variables)*. Finally, for all these schematic variables, a solution must be found. In the case of `resolve_tac`, the conclusion of the (doubly lifted) theorem must be equal to the conclusion of the subgoal, so $A$ must be $\alpha/\beta$-equivalent to $A'$ in the example above, which is established by a higher-order unification process. It is a bit unfortunate that for implementation efficiency reasons, a very substantial part of the code for HO-unification is in the kernel module `thm`, which makes this critical component of the architecture larger than necessary.

In a way, the two lifting processes represent an implementation of the conversion between Gentzen Natural Deduction (to which Isabelle/Pure is geared) reasoning and Gentzen Sequent Deduction.

## 3.5 The classical goal package

The main mechanism in Isabelle as an LCF-style system is to produce `thm`'s in backward-style via tactics as described in Section 3.4. Given a context — be it global as `theory` or be it inside a proof-context as `Proof.context`, user-programmed verification of (type-checked) terms or just strings can be done via the operations:

```
ML‹
Goal.prove-internal : Proof.context −> cterm list −> cterm −> (thm list −> tactic) −> thm;

Goal.prove-global :  theory −> string list −> term list −> term −>
                   ({context: Proof.context, prems: thm list} −> tactic) −> thm;
(∗ ... and many more variants. ∗)
›
```

### 3.5.1 Proof Example

The proof:

**lemma** *(10::int) + 2 = 12* **by** *simp*

... represents itself at the SML interface as follows:

```
ML‹val tt = HOLogic.mk-Trueprop (Syntax.read-term @{context} (10::int) + 2 = 12);
        (∗ read-term parses and type−checks its string argument;
           HOLogic.mk-Trueprop wraps the embedder from @{ML-type bool} to
           @{ML-type prop} from Pure. ∗)

val thm1 = Goal.prove-global @{theory}                      (∗ global context ∗)
                             []                             (∗ name ? ∗)
                             []                             (∗ local assumption context ∗)
                             (tt)                           (∗ parsed goal ∗)
                             (fn - => simp-tac  @{context} 1)  (∗ proof tactic ∗)›
```

## 3.6 The Isar Engine

The main structure of the Isar-engine is `Toplevel` and provides and internal `state` with the necessary infrastructure — i.e. the operations to pack and unpack theories and Proof.contexts — on it:

```
ML‹
   Toplevel.theory-toplevel: theory −> Toplevel.state;
   Toplevel.init-toplevel: unit −> Toplevel.state;
   Toplevel.is-toplevel: Toplevel.state −> bool;
   Toplevel.is-theory: Toplevel.state −> bool;
   Toplevel.is-proof: Toplevel.state −> bool;
   Toplevel.is-skipped-proof: Toplevel.state −> bool;
   Toplevel.level: Toplevel.state −> int;
   Toplevel.context-of: Toplevel.state −> Proof.context;
   Toplevel.generic-theory-of: Toplevel.state −> generic-theory;
   Toplevel.theory-of: Toplevel.state −> theory;
   Toplevel.proof-of: Toplevel.state −> Proof.state;
   Toplevel.presentation-context: Toplevel.state −> Proof.context;
```

(* ... *) ›

The extensibility of Isabelle as a system framework depends on a number of tables, into which various concepts commands, ML-antiquotations, text-antiquotations, cartouches, ... can be entered via a late-binding on the fly.

A paradigmatic example is the `Outer_Syntax.command`-operation, which — representing in itself a toplevel system transition — allows to define a new command section and bind its syntax and semantics at a specific keyword. Calling `Outer_Syntax.command` creates an implicit `Theory.setup` with an entry for a call-back function, which happens to be a parser that must have as side-effect a Toplevel-transition-transition. Registers `Toplevel.transition -> Toplevel.transition` parsers to the Isar interpreter.

```
ML‹ Outer-Syntax.command : Outer-Syntax.command-keyword -> string ->
                  (Toplevel.transition -> Toplevel.transition) parser -> unit;›
```

A paradigmatic example: "text" is defined by:

Here are a few queries relevant for the global config of the isar engine:

```
ML‹ Document.state();›
ML‹ Session.get-keywords(); (* this looks to be really session global. *) ›
ML‹ Thy-Header.get-keywords @{theory};(* this looks to be really theory global. *) ›
```

### 3.6.1 Transaction Management in the Isar-Engine : The Toplevel

```
ML‹Toplevel.exit: Toplevel.transition -> Toplevel.transition;
  Toplevel.keep: (Toplevel.state -> unit) -> Toplevel.transition -> Toplevel.transition;
  Toplevel.keep': (bool -> Toplevel.state -> unit) -> Toplevel.transition -> Toplevel.transition;
  Toplevel.ignored: Position.T -> Toplevel.transition;
  Toplevel.generic-theory: (generic-theory -> generic-theory) -> Toplevel.transition -> Toplevel.transition;
  Toplevel.theory': (bool -> theory -> theory) -> Toplevel.transition -> Toplevel.transition;
  Toplevel.theory: (theory -> theory) -> Toplevel.transition -> Toplevel.transition;

  Toplevel.present-local-theory:
  (xstring * Position.T) option ->
     (Toplevel.state -> unit) -> Toplevel.transition -> Toplevel.transition;
  (* where text treatment and antiquotation parsing happens *)


  (*fun document-command markdown (loc, txt) =
    Toplevel.keep (fn state =>
      (case loc of
        NONE => ignore (output-text state markdown txt)
      | SOME (-, pos) =>
          error (Illegal target specification -- not a theory context ^ Position.here pos))) o
    Toplevel.present-local-theory loc (fn state => ignore (output-text state markdown txt)); *)

  (* Isar Toplevel Steuerung *)
  Toplevel.keep : (Toplevel.state -> unit) -> Toplevel.transition -> Toplevel.transition;
```

```
(* val keep' = add-trans o Keep;
   fun keep f = add-trans (Keep (fn - => f));
 *)

Toplevel.present-local-theory : (xstring * Position.T) option -> (Toplevel.state -> unit) ->
    Toplevel.transition -> Toplevel.transition;
    (* fun present-local-theory target = present-transaction (fn int =>
          (fn Theory (gthy, -) =>
                  let val (finish, lthy) = Named-Target.switch target gthy;
                  in Theory (finish lthy, SOME lthy) end
          | - => raise UNDEF));

       fun present-transaction f g = add-trans (Transaction (f, g));
       fun transaction f = present-transaction f (K ());
    *)

Toplevel.theory : (theory -> theory) -> Toplevel.transition -> Toplevel.transition;
   (* fun theory' f = transaction (fn int =>
                (fn Theory (Context.Theory thy, -) =>
                      let val thy' = thy
                                     |> Sign.new-group
                                     |> f int
                                     |> Sign.reset-group;
                      in Theory (Context.Theory thy', NONE) end
                | - => raise UNDEF));

      fun theory f = theory' (K f); *)›
```

**ML**‹

```
(* Isar Toplevel Control *)
Toplevel.keep :  (Toplevel.state -> unit) -> Toplevel.transition -> Toplevel.transition;
    (* val keep' = add-trans o Keep;
       fun keep f = add-trans (Keep (fn - => f));
     *)

Toplevel.present-local-theory : (xstring * Position.T) option -> (Toplevel.state -> unit) ->
    Toplevel.transition -> Toplevel.transition;
    (* fun present-local-theory target = present-transaction (fn int =>
          (fn Theory (gthy, -) =>
                  let val (finish, lthy) = Named-Target.switch target gthy;
                  in Theory (finish lthy, SOME lthy) end
          | - => raise UNDEF));

       fun present-transaction f g = add-trans (Transaction (f, g));
       fun transaction f = present-transaction f (K ());
    *)
```

```
Toplevel.theory : (theory -> theory) -> Toplevel.transition -> Toplevel.transition;
  (* fun theory' f = transaction (fn int =>
              (fn Theory (Context.Theory thy, _) =>
                  let val thy' = thy
                                 |> Sign.new_group
                                 |> f int
                                 |> Sign.reset_group;
                  in Theory (Context.Theory thy', NONE) end
              | _ => raise UNDEF));

     fun theory f = theory' (K f); *)
›
```

### 3.6.2 Configuration flags of fixed type in the Isar-engine.

The toplevel also provides an infrastructure for managing configuration options for system components. Based on a sum-type `Config.value` with the alternatives `Bool of bool | Int of int | Real of real | String of string` and building the parametric configuration types `'a Config.T` and the instance `type raw = value T`, for all registered configurations the protocol:

```
ML‹
  Config.get: Proof.context -> 'a Config.T -> 'a;
  Config.map: 'a Config.T -> ('a -> 'a) -> Proof.context -> Proof.context;
  Config.put: 'a Config.T -> 'a -> Proof.context -> Proof.context;
  Config.get_global: theory -> 'a Config.T -> 'a;
  Config.map_global: 'a Config.T -> ('a -> 'a) -> theory -> theory;
  Config.put_global: 'a Config.T -> 'a -> theory -> theory;›
```

... etc. is defined.

Example registration of an config attribute XS232:

```
ML‹
val (XS232, XS232_setup)
     = Attrib.config_bool binding‹XS232› (K false);

val _ = Theory.setup XS232_setup;›
```

Another mechanism are global synchronised variables:

```
ML‹ (* For example *)

  val C = Synchronized.var Pretty.modes latEEex;
  (* Synchronized: a mechanism to bookkeep global
     variables with synchronization mechanism included *)
  Synchronized.value C;›
```

# 4 Front-End

In the following chapter, we turn to the right part of the system architecture shown in Figure 1.1: The PIDE ("Prover-IDE") layer [4] consisting of a part written in SML and another in SCALA. Roughly speaking, PIDE implements "continuous build - continuous check" - functionality over a textual, albeit generic document model. It transforms user modifications of text elements in an instance of this model into increments (edits) and communicates them to the Isabelle system. The latter reacts by the creation of a multitude of light-weight reevaluation threads resulting in an asynchronous stream of markup that is used to annotate text elements. Such markup is used to highlight, e.g., variables or keywords with specific colors, to hyper-linking bound variables to their defining occurrences, or to annotate type-information to terms which becomes displayed by specific user-gestures on demand (hovering), etc. Note that PIDE is not an editor, it is the framework that coordinates these asynchronous information streams and optimizes it to a certain extent (outdated markup referring to modified text is dropped, and corresponding re-calculations are oriented to the user focus, for example). Four concrete editors — also called PIDE applications — have been implemented:

1. an Eclipse plugin (developped by an Edinburg-group, based on an very old PIDE version),
2. a Visual-Studio Code plugin (developed by Makarius Wenzel), currently based on a fairly old PIDE version,
3. clide, a web-client supporting javascript and HTML5 (developed by a group at University Bremen, based on a very old PIDE version), and
4. the most commonly used: the plugin in JEdit - Editor, (developed by Makarius Wenzel, current PIDE version.)

The document model forsees a number of text files, which are organized in form of an acyclic graph. Such graphs can be grouped into *sessions* and "frozen" to binaries in order to avoid long compilation times. Text files have an abstract name serving as identity (the mapping to file-paths in an underlying file-system is done in an own build management). The primary format of the text files is `.thy` (historically for: theory), secondary formats can be `.sty`,`.tex`, `.png`, `.pdf`, or other files processed by Isabelle and listed in a configuration processed by the build system.

A `.thy` file consists of a *header*, a *context-definition* and a *body* consisting of a sequence of *commands*. Even the header consists of a sequence of commands used for introductory text elements not depending on any context information (so: practically excluding any form of text antiquotation (see above)). The context-definition contains an *import* and a *keyword* section; for example:

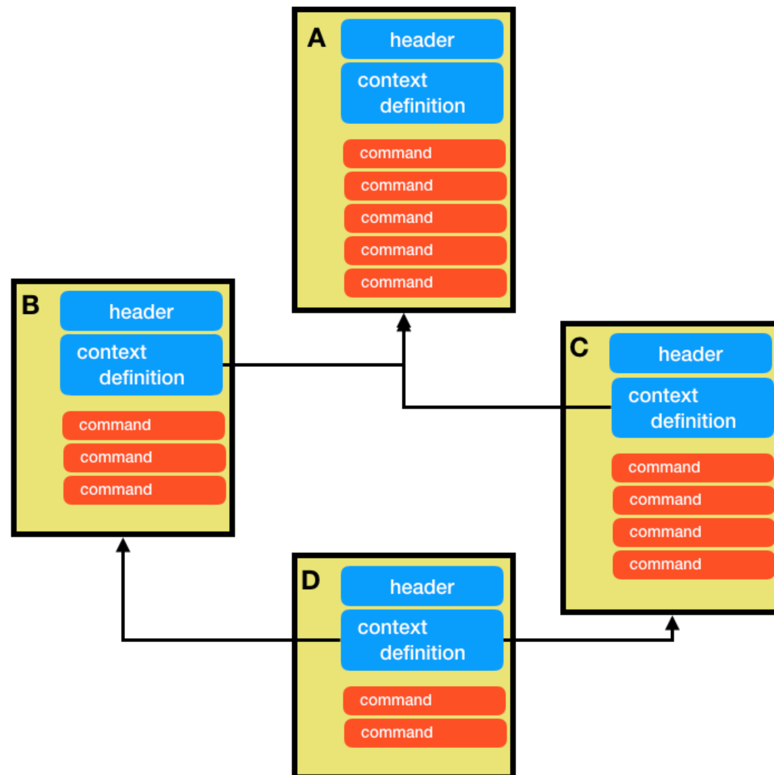


Figure 4.1: A Theory-Graph in the Document Model

```
theory Isa_DOF                (* Isabelle Document Ontology Framework *)
  imports  Main
           RegExpInterface    (* Interface to functional regular automata for monitoring *)
           Assert

  keywords "+=" ":=" "accepts" "rejects"
```

where `Isa_DOF` is the abstract name of the text-file, `Main` etc. refer to imported text files (recall that the import relation must be acyclic). *keywords* are used to separate commands form each other; predefined commands allow for the dynamic creation of new commands similarly to the definition of new functions in an interpreter shell (or: toplevel, see above.). A command starts with a pre-declared keyword and specific syntax of this command; the declaration of a keyword is only allowed in the same `.thy`-file where the the corresponding new command is defined. The semantics of the command is expressed in ML and consists of a `Toplevel.transition -> Toplevel.transition` function. Thus, the Isar-toplevel supports the generic document model and allows for user-programmed extensions.

In the traditional literature, Isabelle `.thy`-files were were said to be processed by processed by two types of parsers:

1. the "outer-syntax" (i.e. the syntax for commands) is processed by a lexer-library and parser combinators built on top, and
2. the "inner-syntax" (i.e. the syntax for $\Lambda$ - terms) with an evolved, eight-layer parsing and pretty-printing process based on an Early-algorithm.

This picture is less and less true for a number of reasons:

1. With the advent of (‹)... (›), a mechanism for *cascade-syntax* came to the Isabelle platform that introduce a flexible means to change parsing contexts *and* parsing technologies.
2. Inside the term-parser levels, the concept of *cartouche* can be used to escape the parser and its underlying parsing technology.
3. Outside, in the traditional toplevel-parsers, the (‹)... (›) is becoming more and more enforced (some years ago, syntax like *term*{* ... *} was replaced by syntax *term*(‹)... (›). This makes technical support of cascade syntax more and more easy.
4. The Lexer infra-structure is already rather generic; nothing prevents to add beside the lexer - configurations for ML-Parsers, Toplevel Command Syntax parsers, mathematical notation parsers for $\lambda$-terms new pillars of parsing technologies, say, for parsing C or Rust or JavaScript inside Isabelle.

Figure 4.2: Output with hyperlinked position.

## 4.1 Basics: string, bstring and xstring

`string` is the basic library type from the SML library in structure `String`. Many Isabelle operations produce or require formats thereof introduced as type synonyms `bstring` (defined in structure `Binding` and `xstring` (defined in structure `Name_Space`. Unfortunately, the abstraction is not tight and combinations with elementary routines might produce quite crappy results.

**ML**‹*val b = Binding.name-of* @{*binding here*}›

... produces the system output `val it = "here": bstring`, but note that it is misleading to believe it is just a string.

**ML**‹*String.explode b*›
**ML**‹*String.explode*(*Binding.name-of*
(*Binding.conglomerate*[*Binding.qualified-name X.x*,
@{*binding here*}] ))›

However, there is an own XML parser for this format. See Section Markup.

**ML**‹ *fun dark-matter x = XML.content-of* (*YXML.parse-body x*)›

## 4.2 Positions

A basic data-structure relevant for PIDE are *positions*; beyond the usual line- and column information they can represent ranges, list of ranges, and the name of the atomic sub-document in which they are contained. In the command:

**ML**‹
*val pos* = @{*here*};
*val markup* = *Position.here pos*;
*writeln* (*And a link to the declaration of 'here' is ^markup*)›

... uses the antiquotation `@{here}` to infer from the system lexer the actual position of itself in the global document, converts it to markup (a string-representation of it) and sends it via the usual `writeln` to the interface.

Figure 4.2 shows the produced output where the little house-like symbol in the display is hyperlinked to the position of `@{here}` in the ML sample above.

## 4.3 Markup and Low-level Markup Reporting

The structures `Markup` and `Properties` represent the basic annotation data which is part of the protocol sent from Isabelle to the front-end. They are qualified as "quasi-abstract",

which means they are intended to be an abstraction of the serialized, textual presentation of the protocol. Markups are structurally a pair of a key and properties; `Markup` provides a number of of such *keys* for annotation classes such as "constant", "fixed", "cartouche", some of them quite obscure. Here is a code sample from *Isabelle-DOF*. A markup must be tagged with an id; this is done by the `serial`-function discussed earlier. Markup Operations, were used for hyperlinking applications to binding occurrences, info for hovering, infors for type ...

```
ML‹
(* Position.report is also a type consisting of a pair of a position and markup. *)
(* It would solve all my problems if I find a way to infer the defining Position.report
   from a type definition occurence ... *)

Position.report: Position.T -> Markup.T -> unit;
Position.reports: Position.report list -> unit;
     (* ? ? ? I think this is the magic thing that sends reports to the GUI. *)
Markup.entity : string -> string -> Markup.T;
Markup.properties : Properties.T -> Markup.T -> Markup.T ;
Properties.get : Properties.T -> string -> string option;
Markup.enclose : Markup.T -> string -> string;

(* example for setting a link, the def flag controls if it is a defining or a binding
occurence of an item *)
fun theory-markup (def:bool) (name:string) (id:serial) (pos:Position.T) =
  if id = 0 then Markup.empty
  else
    Markup.properties (Position.entity-properties-of def id pos)
      (Markup.entity Markup.theoryN name);
Markup.theoryN : string;

serial();   (* A global, lock-guarded seriel counter used to produce unique identifiers,
              be it on the level of thy-internal states or as reference in markup in
              PIDE *)
›
```

### 4.3.1 A simple Example

```
ML‹
local

val docclassN = doc-class;

(* derived from: theory-markup; def for defining occurrence (true) in contrast to
   referring occurence (false). *)
fun docclass-markup def name id pos =
  if id = 0 then Markup.empty
  else          Markup.properties (Position.entity-properties-of def id pos)
                                  (Markup.entity docclassN name);
```

```
in

fun report-defining-occurrence pos cid =
          let val id = serial ()
              val markup-of-cid = docclass-markup true cid id pos
          in  Position.report pos markup-of-cid end;

end
›
```

The @*ML* *report-defining-occurrence*-function above takes a position and a "cid" parsed in the Front-End, converts this into markup together with a unique number identifying this markup, and sends this as a report to the Front-End.

### 4.3.2 A Slightly more Complex Example

**ML** ‹

```
fun markup-tvar def-name ps (name, id) =
  let
    fun markup-elem name = (name, (name, []): Markup.T);
    val (tvarN, tvar) = markup-elem ((case def-name of SOME name => name | - => ) ^ 's
nickname is);
    val entity = Markup.entity tvarN name
    val def = def-name = NONE
  in
    tvar ::
    (if def then I else cons (Markup.keyword-properties Markup.ML-keyword3))
      (map (fn pos => Markup.properties (Position.entity-properties-of def id pos) entity) ps)
  end

fun report [] - - = I
  | report ps markup x =
      let val ms = markup x
      in fold (fn p => fold (fn m => cons ((p, m), )) ms) ps end
›
```

**ML** ‹

```
local
val data = — Derived from Yakoub's example ;-)

  [ (‹Frédéric 1er›,  ‹King of Naples›)
  , (‹Frédéric II›,   ‹King of Sicily›)
  , (‹Frédéric III›,  ‹the Handsome›)
  , (‹Frédéric IV›,   ‹of the Empty Pockets›)
  , (‹Frédéric V›,    ‹King of Denmark–Norway›)
  , (‹Frédéric VI›,   ‹the Knight›)
  , (‹Frédéric VII›,  ‹Count of Toggenburg›)
  , (‹Frédéric VIII›, ‹Count of Zollern›)
```

```
  , (‹Frédéric IX›,   ‹the Old›)
  , (‹Frédéric X›,    ‹the Younger›) ]

val (tab0, markup) =
  fold-map (fn (name, msg) => fn reports =>
             let val id = serial ()
                 val pos = [Input.pos-of name]
             in
                ( (fst(Input.source-content msg), (name, pos, id))
               , report pos (markup-tvar NONE pos) (fst(Input.source-content name), id) reports)
             end)
           data
           []

val () = Position.reports-text markup
in
val tab = Symtab.make tab0
end
›

ML ‹
val - =
  fold (fn input =>
        let
          val pos1' = Input.pos-of input
          fun ctnt name0  = fst(Input.source-content name0)
          val pos1  = [pos1']
          val msg1  = fst(Input.source-content input)
          val msg2  = No persons were found to have such nickname
        in
          case Symtab.lookup tab (fst(Input.source-content input)) of
            NONE => tap (fn - => Output.information (msg2 ^ Position.here-list pos1))
                        (cons ((pos1', Markup.bad ()), ))
          | SOME (name0, pos0, id) => report pos1 (markup-tvar (SOME (ctnt name0)) pos0)
(msg1, id)
        end)
       [ ‹the Knight›   — Example of a correct retrieval (CTRL + Hovering shows what we are
expecting)
      , ‹the Handsome› — Example of a correct retrieval (CTRL + Hovering shows what we are
expecting)
      , ‹the Spy›       — Example of a failure to retrieve the person in tab
 ]
      []
  |> Position.reports-text
›
```

The pudding comes with the eating:

### 4.3.3 Environment Structured Reporting

The structure `Name_Space` offers an own infra-structure for names and manages the markup accordingly. MORE TO COME

```
'a Name_Space.table
```

## 4.4 The System Lexer and Token Issues

Four syntactic contexts are predefined in Isabelle (others can be added): the ML context, the text context, the Isar-command context and the teerm-context, referring to different needs of the Isabelle Framework as an extensible framework supporting incremental, partially programmable extensions and as a Framework geared towards Formal Proofs and therefore mathematical notations. The basic data-structure for the lexical treatment of these elemens are `Token`'s.

### 4.4.1 Tokens

The basic entity that lexers treat are *tokens.* defined in `Token` It provides a classification infrastructure, the references to positions and Markup as well as way's to annotate tokens with (some) values they denote:

```
ML‹
local
  open Token

  type dummy = Token.T
  type src = Token.T list
  type file = {src-path: Path.T, lines: string list, digest: SHA1.digest, pos: Position.T}

  type name-value = {name: string, kind: string, print: Proof.context -> Markup.T * xstring}


  val - = Token.is-command : Token.T -> bool;
  val - = Token.content-of : Token.T -> string; (* textueller kern eines Tokens. *)


  val - = pos-of: T -> Position.T

(*
datatype kind =
   (*immediate source*)
   Command | Keyword | Ident | Long-Ident | Sym-Ident | Var | Type-Ident | Type-Var | Nat |
   Float | Space |
   (*delimited content*)
   String | Alt-String | Verbatim | Cartouche | Comment of Comment.kind option |
   (*special content*)
   Error of string | EOF
```

```
datatype value =
  Source of src |
  Literal of bool * Markup.T |
  Name of name-value * morphism |
  Typ of typ |
  Term of term |
  Fact of string option * thm list |
  Attribute of morphism -> attribute |
  Declaration of declaration |
  Files of file Exn.result list


*)
in val - = ()
end
›
```

### 4.4.2 A Lexer Configuration Example

```
ML‹
(* MORE TO COME *)
›
```

## 4.5 Combinator Parsing

Parsing Combinators go back to monadic programming as advocated by Wadler et. al, and has been worked out [2]. Parsing combinators are one of the two major parsing technologies of the Isabelle front-end, in particular for the outer-syntax used for the parsing of toplevel-commands. The core of the combinator library is `Scan` providing the `'a parser` which is a synonym for `Token.T list -> 'a * Token.T list`.

"parsers" are actually interpreters; an 'a parser is a function that parses an input stream and computes(=evaluates, computes) it into 'a. Since the semantics of an Isabelle command is a transition => transition or theory ⇒ theory function, i.e. a global system transition. parsers of that type can be constructed and be bound as call-back functions to a table in the Toplevel-structure of Isar.

The library also provides a bunch of infix parsing combinators, notably:

```
ML‹
 val - = op !! : ('a * message option -> message) -> ('a -> 'b) -> 'a -> 'b
             (*apply function*)
 val - = op >> : ('a -> 'b * 'c) * ('b -> 'd) -> 'a -> 'd * 'c
 (*alternative*)
 val - = op || : ('a -> 'b) * ('a -> 'b) -> 'a -> 'b
 (*sequential pairing*)
 val - = op -- : ('a -> 'b * 'c) * ('c -> 'd * 'e) -> 'a -> ('b * 'd) * 'e
 (*dependent pairing*)
 val - = op :-- : ('a -> 'b * 'c) * ('b -> 'c -> 'd * 'e) -> 'a -> ('b * 'd) * 'e
```

```
(*projections*)
val - = op :|-- : ('a -> 'b * 'c) * ('b -> 'c -> 'd * 'e) -> 'a -> 'd * 'e
val - = op |-- : ('a -> 'b * 'c) * ('c -> 'd * 'e) -> 'a -> 'd * 'e
val - = op --| : ('a -> 'b * 'c) * ('c -> 'd * 'e) -> 'a -> 'b * 'e
(*concatenation*)
val - = op ^^ : ('a -> string * 'b) * ('b -> string * 'c) -> 'a -> string * 'c
val - = op ::: : ('a -> 'b * 'c) * ('c -> 'b list * 'd) -> 'a -> 'b list * 'd
val - = op @@@ : ('a -> 'b list * 'c) * ('c -> 'b list * 'd) -> 'a -> 'b list * 'd
(*one element literal*)
val - = op $$ : string -> string list -> string * string list
val - = op ~$$ : string -> string list -> string * string list
›
```

Usually, combinators were used in one of these following instances:

```
ML‹

  type 'a parser = Token.T list -> 'a * Token.T list
  type 'a context-parser = Context.generic * Token.T list -> 'a * (Context.generic * Token.T
list)


(* conversion between these two : *)

fun parser2contextparser pars (ctxt, toks) = let val (a, toks') = pars toks
                                             in (a,(ctxt, toks')) end;
val - = parser2contextparser : 'a parser -> 'a context-parser;

(* bah, is the same as Scan.lift *)
val - = Scan.lift Args.cartouche-input : Input.source context-parser;›
```

### 4.5.1 Advanced Parser Library

There are two parts. A general multi-purpose parsing combinator library is found under `Parse`, providing basic functionality for parsing strings or integers. There is also an important combinator that reads the current position information out of the input stream:

```
ML‹
Parse.nat: int parser;
Parse.int: int parser;
Parse.enum-positions: string -> 'a parser -> ('a list * Position.T list) parser;
Parse.enum: string -> 'a parser -> 'a list parser;
Parse.input: 'a parser -> Input.source parser;

Parse.enum : string -> 'a parser -> 'a list parser;
Parse.!!! : (Token.T list -> 'a) -> Token.T list -> 'a;
Parse.position: 'a parser -> ('a * Position.T) parser;

(* Examples *)
Parse.position Args.cartouche-input;
›
```

The second part is much more high-level, and can be found under `Args`. In parts, these combinators are again based on more low-level combinators, in parts they serve as an an interface to the underlying Earley-parser for mathematical notation used in types and terms. This is perhaps meant with the fairly cryptic comment: "Quasi-inner syntax based on outer tokens: concrete argument syntax of attributes, methods etc." at the beginning of this structure.

**ML**‹

```
local

  open Args

  (* some more combinaotrs *)
  val - =  symbolic: Token.T parser
  val - =  $$$ : string -> string parser
  val - =  maybe: 'a parser -> 'a option parser
  val - =  name-token: Token.T parser


  (* common isar syntax *)
  val - =  colon: string parser
  val - =  query: string parser
  val - =  bang: string parser
  val - =  query-colon: string parser
  val - =  bang-colon: string parser
  val - =  parens: 'a parser -> 'a parser
  val - =  bracks: 'a parser -> 'a parser
  val - =  mode: string -> bool parser
  val - =  name: string parser
  val - =  name-position: (string * Position.T) parser
  val - =  cartouche-inner-syntax: string parser
  val - =  cartouche-input: Input.source parser
  val - =  text-token: Token.T parser

  (* advanced string stuff *)
  val - =  embedded-token: Token.T parser
  val - =  embedded-inner-syntax: string parser
  val - =  embedded-input: Input.source parser
  val - =  embedded: string parser
  val - =  embedded-position: (string * Position.T) parser
  val - =  text-input: Input.source parser
  val - =  text: string parser
  val - =  binding: binding parser

  (* stuff related to INNER SYNTAX PARSING *)
  val - =  alt-name: string parser
  val - =  liberal-name: string parser
  val - =  var: indexname parser
```

```
val _ = internal_source: Token.src parser
val _ = internal_name: Token.name_value parser
val _ = internal_typ: typ parser
val _ = internal_term: term parser
val _ = internal_fact: thm list parser
val _ = internal_attribute: (morphism -> attribute) parser
val _ = internal_declaration: declaration parser


val _ = named_source: (Token.T -> Token.src) -> Token.src parser
val _ = named_typ: (string -> typ) -> typ parser
val _ = named_term: (string -> term) -> term parser

val _ = text_declaration: (Input.source -> declaration) -> declaration parser
val _ = cartouche_declaration: (Input.source -> declaration) -> declaration parser
val _ = typ_abbrev: typ context_parser

val _ = typ: typ context_parser
val _ = term: term context_parser
val _ = term_pattern: term context_parser
val _ = term_abbrev: term context_parser

(* syntax for some major Pure commands in Isar *)
val _ = prop: term context_parser
val _ = type_name: {proper: bool, strict: bool} -> string context_parser
val _ = const: {proper: bool, strict: bool} -> string context_parser
val _ = goal_spec: ((int -> tactic) -> tactic) context_parser
val _ = context: Proof.context context_parser
val _ = theory: theory context_parser

in val _ = () end
```

⟩

### 4.5.2 Bindings

The structure `Binding` serves as *structured name bindings*, as says the description, i.e. a mechanism to basically associate an input string-fragment to its position. This concept is vital in all parsing processes and the interaction with PIDE.

Key are two things:

1. the type-synonym `bstring` which is synonym to `string` and intended for "primitive names to be bound"

2. the projection `Binding.pos_of : binding -> Position.T`

3. the constructor establishing a binding `Binding.make: bstring * Position.T -> binding`

Since this is so common in interface programming, there are a number of antiquotations

```
ML‹
val H = @{binding here}; (* There are bindings consisting of a text−span and a position,
                            where ¨positions¨ are absolute references to a file *)

Binding.pos-of H;  (* clicking on H activates the hyperlink to the defining occ of H above *)
(*  {offset=23, end-offset=27, id=−17214}: Position.T *)

(* a modern way to construct a binding is by the following code antiquotation : *)
binding ‹theory›

›
```

### 4.5.3 Input streams.

Reads as : Generic input with position information.

```
ML‹
  Input.source-explode : Input.source −> Symbol-Pos.T list;
  Input.source-explode (Input.string  f  @{thm refl});

   (* If stemming from the input window, this can be s th like:

        [( , {offset=14, id=−2769}), (f, {offset=15, id=−2769}), ( , {offset=16, id=−2769}),
        (@, {offset=17, id=−2769}), ({, {offset=18, id=−2769}), (t, {offset=19, id=−2769}),
        (h, {offset=20, id=−2769}), (m, {offset=21, id=−2769}), ( , {offset=22, id=−2769}),
        (r, {offset=23, id=−2769}), (e, {offset=24, id=−2769}), (f, {offset=25, id=−2769}),
         (l, {offset=26, id=−2769}), (}, {offset=27, id=−2769})]
   *)

›
```

## 4.6 Term Parsing

The heart of the parsers for mathematical notation, based on an Earley parser that can cope with incremental changes of the grammar as required for sophisticated mathematical output, is hidden behind the API described in this section.

Note that the naming underlies the following convention : there are:

1. "parser"s and type-"checker"s
2. "reader"s which do both together with pretty-printing

   This is encapsulated the data structure `Syntax` — the table with const symbols, print and ast translations, ... The latter is accessible, e.g. from a Proof context via `Proof_Context.syn_of`.

Inner Syntax Parsing combinators for elementary Isabelle Lexems

**ML**‹
*Syntax.parse-sort* : *Proof.context* −> *string* −> *sort*;
*Syntax.parse-typ* : *Proof.context* −> *string* −> *typ*;
*Syntax.parse-term* : *Proof.context* −> *string* −> *term*;
*Syntax.parse-prop* : *Proof.context* −> *string* −> *term*;
*Syntax.check-term* : *Proof.context* −> *term* −> *term*;
*Syntax.check-props*: *Proof.context* −> *term list* −> *term list*;
*Syntax.uncheck-sort*: *Proof.context* −> *sort* −> *sort*;
*Syntax.uncheck-typs*: *Proof.context* −> *typ list* −> *typ list*;
*Syntax.uncheck-terms*: *Proof.context* −> *term list* −> *term list*;›

In contrast to mere parsing, the following operators provide also type-checking and internal reporting to PIDE — see below. I did not find a mechanism to address the internal serial-numbers used for the PIDE protocol, however, rumours have it that such a thing exists. The variants `_global` work on theories instead on `Proof.context`s.

**ML**‹
*Syntax.read-sort*: *Proof.context* −> *string* −> *sort*;
*Syntax.read-typ* : *Proof.context* −> *string* −> *typ*;
*Syntax.read-term*: *Proof.context* −> *string* −> *term*;
*Syntax.read-typs*: *Proof.context* −> *string list* −> *typ list*;
*Syntax.read-sort-global*: *theory* −> *string* −> *sort*;
*Syntax.read-typ-global*: *theory* −> *string* −> *typ*;
*Syntax.read-term-global*: *theory* −> *string* −> *term*;
*Syntax.read-prop-global*: *theory* −> *string* −> *term*;
›

The following operations are concerned with the conversion of pretty-prints and, from there, the generation of (non-layouted) strings.

**ML**‹
*Syntax.pretty-term*:*Proof.context* −> *term* −> *Pretty.T*;
*Syntax.pretty-typ*:*Proof.context* −> *typ* −> *Pretty.T*;
*Syntax.pretty-sort*:*Proof.context* −> *sort* −> *Pretty.T*;
*Syntax.pretty-classrel*: *Proof.context* −> *class list* −> *Pretty.T*;
*Syntax.pretty-arity*: *Proof.context* −> *arity* −> *Pretty.T*;
*Syntax.string-of-term*: *Proof.context* −> *term* −> *string*;
*Syntax.string-of-typ*: *Proof.context* −> *typ* −> *string*;
*Syntax.lookup-const* : *Syntax.syntax* −> *string* −> *string option*;
›

Note that `Syntax.install_operations` is a late-binding interface, i.e. a collection of "hooks" used to resolve an apparent architectural cycle. The real work is done in `~~/src/Pure/Syntax/syntax_phases.ML`

Even the parsers and type checkers stemming from the theory-structure are registered via hooks (this can be confusing at times). Main phases of inner syntax processing, with standard implementations of parse/unparse operations were treated this way. At the very very end in , it sets up the entire syntax engine (the hooks) via:

### 4.6.1 Example

```
ML‹
(* Recall the Arg−interface to the more high−level, more Isar−specific parsers: *)
Args.name            : string parser ;
Args.const           : {proper: bool, strict: bool} −> string context-parser;
Args.cartouche-input: Input.source parser;
Args.text-token      : Token.T parser;


(* here follows the definition of the attribute parser : *)
val Z = let val attribute = Parse.position Parse.name −−
                            Scan.optional (Parse.$$$ = |−− Parse.!!! Parse.name) ;
        in  (Scan.optional(Parse.$$$ , |−− (Parse.enum , attribute))) end ;


(* Here is the code to register the above parsers as text antiquotations into the Isabelle
   Framework: *)
Thy-Output.antiquotation-pretty-source  binding ‹theory›  (Scan.lift  (Parse.position
Args.embedded));

Thy-Output.antiquotation-raw binding ‹file› (Scan.lift (Parse.position Parse.path)) ;


(* where we have the registration of the action

        (Scan.lift (Parse.position Args.cartouche-input))))

   to be bound to the

        name

   as a whole is a system transaction that, of course, has the type

        theory −> theory : *)
(fn name => (Thy-Output.antiquotation-pretty-source name
                                       (Scan.lift (Parse.position Args.cartouche-input))))
  : binding −> (Proof.context −> Input.source * Position.T −> Pretty.T) −> theory −>
theory;
›
```

## 4.7 Output: Very Low Level

For re-directing the output channels, the structure `Output` may be relevant:

```
ML‹
Output.output; (* output is the  structure for the hooks with the target devices. *)
Output.output bla-1:;
›
```

It provides a number of hooks that can be used for redirection hacks ...

## 4.8 Output: LaTeX

The heart of the LaTeX generator is to be found in the structure `Thy_Output`. This is an own parsing and writing process, with the risc that a parsed file in the IDE parsing process can not be parsed for the LaTeX Generator. The reason is twofold:

1. The LaTeX Generator makes a rough attempt to mimic the LayOut if the thy-file; thus, its spacing is relevant.
2. there is a special bracket (*<*) ... (*>*) that allows to specify input that is checked by Isabelle, but excluded from the LaTeX generator (this is handled in an own sub-parser called `Document_Source.improper` where also other forms of comment parsers are provided.

   Since Isabelle2018, an own AST is provided for the LaTeX syntax, analogously to `Pretty`. Key functions of this structure `Latex` are:

```
ML‹
local

  open Latex

  type dummy = text

  val - = string: string -> text;
  val - = text: string * Position.T -> text

  val - = output-text: text list -> string
  val - = output-positions: Position.T -> text list -> string
  val - = output-name: string -> string
  val - = output-ascii: string -> string
  val - = output-symbols: Symbol.symbol list -> string

  val - = begin-delim: string -> string
  val - = end-delim: string -> string
  val - = begin-tag: string -> string
  val - = end-tag: string -> string
  val - = environment-block: string -> text list -> text
  val - = environment: string -> string -> string

  val - = block: text list -> text
  val - = enclose-body: string -> string -> text list -> text list
  val - = enclose-block: string -> string -> text list -> text

in val - = ()
end;

Latex.output-ascii;
Latex.environment isa bg;
```

```
Latex.output-ascii a-b:c'é;
(* Note: *)
space-implode sd &e sf dfg [qs,er,alpa];

›
```

Here is an abstract of the main interface to `Thy_Output`:

```
ML‹
 output-document: Proof.context -> {markdown: bool} -> Input.source -> Latex.text list;
 output-token: Proof.context -> Token.T -> Latex.text list;
 output-source: Proof.context -> string -> Latex.text list;
 present-thy: Options.T -> theory -> segment list -> Latex.text list;

 isabelle: Proof.context -> Latex.text list -> Latex.text;

 isabelle-typewriter: Proof.context -> Latex.text list -> Latex.text;

 typewriter: Proof.context -> string -> Latex.text;

 verbatim: Proof.context -> string -> Latex.text;

 source: Proof.context -> {embedded: bool} -> Token.src -> Latex.text;

 pretty: Proof.context -> Pretty.T -> Latex.text;
 pretty-source: Proof.context -> {embedded: bool} -> Token.src -> Pretty.T -> Latex.text;
 pretty-items: Proof.context -> Pretty.T list -> Latex.text;
 pretty-items-source: Proof.context -> {embedded: bool} -> Token.src -> Pretty.T list ->
Latex.text;

(* finally a number of antiquotation registries : *)
 antiquotation-pretty:
       binding -> 'a context-parser -> (Proof.context -> 'a -> Pretty.T) -> theory ->
theory;
 antiquotation-pretty-source:
   binding -> 'a context-parser -> (Proof.context -> 'a -> Pretty.T) -> theory -> theory;
 antiquotation-raw:
     binding -> 'a context-parser -> (Proof.context -> 'a -> Latex.text) -> theory ->
theory;
 antiquotation-verbatim:
     binding -> 'a context-parser -> (Proof.context -> 'a -> string) -> theory -> theory;

›
```

Thus, `Syntax_Phases` does the actual work of markup generation, including markup generation and generation of reports. Look at the following snippet:

```
ML‹
(*
fun check-typs ctxt raw-tys =
  let
```

```
    val (sorting-report, tys) = Proof-Context.prepare-sortsT ctxt raw-tys;
    val - = if Context-Position.is-visible ctxt then Output.report sorting-report else ();
  in
    tys
    |> apply-typ-check ctxt
    |> Term-Sharing.typs (Proof-Context.theory-of ctxt)
  end;
```

*which is the real implementation behind Syntax.check-typ*

*or*:

```
fun check-terms ctxt raw-ts =
  let
    val (sorting-report, raw-ts') = Proof-Context.prepare-sorts ctxt raw-ts;
    val (ts, ps) = Type-Infer-Context.prepare-positions ctxt raw-ts';

    val tys = map (Logic.mk-type o snd) ps;
    val (ts', tys') = ts @ tys
      |> apply-term-check ctxt
      |> chop (length ts);
    val typing-report =
      fold2 (fn (pos, -) => fn ty =>
        if Position.is-reported pos then
          cons (Position.reported-text pos Markup.typing
            (Syntax.string-of-typ ctxt (Logic.dest-type ty)))
        else I) ps tys' [];

    val - =
      if Context-Position.is-visible ctxt then Output.report (sorting-report @ typing-report)
      else ();
  in Term-Sharing.terms (Proof-Context.theory-of ctxt) ts' end;
```

*which is the real implementation behind Syntax.check-term*

*As one can see, check−routines internally generate the markup.*

*)
›

## 4.9 Inner Syntax Cartouches

The cascade-syntax principle underlying recent isabelle versions requires a particular mechanism, called "cartouche" by Makarius who was influenced by French Wine and French culture when designing this.

When parsing terms or types (via the Earley Parser), a standard mechanism for calling another parser inside the current process is needed that is bound to the (‹)› ... ‹(›) paranthesis'.

The following example — drawn from the Isabelle/DOF implementation — allows to parse UTF8 - Unicode strings as alternative to ''*abc*'' HOL-strings.

```
ML‹— Dynamic setup of inner syntax cartouche


(* Author:     Frédéric Tuong, Université Paris−Saclay *)
(* Title:     HOL/ex/Cartouche-Examples.thy
   Author:     Makarius *)
  local
    fun mk-char (f-char, f-cons, -) (s, -) accu =
        fold
          (fn c => fn (accu, l) =>
            (f-char c accu, f-cons c l))
          (rev (map Char.ord (String.explode s)))
          accu;

    fun mk-string (-, -, f-nil) accu [] = (accu, f-nil)
      | mk-string f accu (s :: ss) = mk-char f s (mk-string f accu ss);
  in
    fun string-tr f f-mk accu content args =
      let fun err () = raise TERM (string-tr, args) in
        (case args of
          [(c as Const (@{syntax-const -constrain}, -)) $ Free (s, -) $ p] =>
            (case Term-Position.decode-position p of
              SOME (pos, -) => c $ f (mk-string f-mk accu (content (s, pos))) $ p
            | NONE => err ())
        | - => err ())
      end;
  end;
›

syntax -cartouche-string :: cartouche-position ⇒ -  (-)

ML‹
structure Cartouche-Grammar = struct
  fun list-comb-mk cst n c = list-comb (Syntax.const cst, String-Syntax.mk-bits-syntax n c)
  val nil1 = Syntax.const @{const-syntax String.empty-literal}
  fun cons1 c l = list-comb-mk @{const-syntax String.Literal} 7 c $ l

  val default =
    [ ( char list
      , ( Const (@{const-syntax Nil}, @{typ char list})
        , fn c => fn l => Syntax.const @{const-syntax Cons} $ list-comb-mk @{const-syntax
Char} 8 c $ l
        , snd))
    , ( String.literal, (nil1, cons1, snd))]
end
›
```

```
ML‹
fun parse-translation-cartouche binding l f-integer accu =
  let val cartouche-type = Attrib.setup-config-string binding (K (fst (hd l)))
      (* if there is no type specified, by default we set the first element
         to be the default type of cartouches *) in
  fn ctxt =>
    let val cart-type = Config.get ctxt cartouche-type in
    case List.find (fn (s, -) => s = cart-type) l of
      NONE => error (Unregistered return type for the cartouche: \ ^ cart-type ^ \)
    | SOME (-, (nil0, cons, f)) =>
               string-tr f (f-integer, cons, nil0) accu (Symbol-Pos.cartouche-content o
Symbol-Pos.explode)
    end
  end
›
```

The following registration of this cartouche for strings is fails because it has already been done in the surrounding Isabelle/DOF environment...

**term** ‹$A \Longrightarrow B$› = ''''

# 5 Conclusion

This interactive Isabelle Programming Cook-Book represents my current way to view and explain Isabelle programming API's to students and collaborators. It differs from the reference manual in some places on purpose, since I believe that a lot of internal Isabelle API's need a more conceptual view on what is happening (even if this conceptual view is at times over-abstracting a little). It is written in Isabelle/DOF and conceived as "living document" (a term that I owe Simon Foster), i.e. as hypertext-heavy text making direct references to the Isabelle API's which were checked whenever this document is re-visited in Isabelle/jEdit.

All hints and contributions of collegues and collaborators are greatly welcomed; all errors and the roughness of this presentation is entirely my fault.

# Bibliography


[1] B. Barras, L. D. C. González-Huesca, H. Herbelin, Y. Régis-Gianas, E. Tassi, M. Wenzel, and B. Wolff. Pervasive parallelism in highly-trustable interactive theorem proving systems. volume 7961 of *Lecture Notes in Computer Science*, pages 359–363. Springer, 2013. ISBN 978-3-642-39319-8. URL https://doi.org/10.1007/978-3-642-39320-4_29.

[2] G. Hutton. Higher-order functions for parsing. *J. Funct. Program.*, 2(3):323–343, 1992. URL https://doi.org/10.1017/S0956796800000411.

[3] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. ISBN 3-540-43376-7. URL https://doi.org/10.1007/3-540-45949-9.

[4] M. Wenzel. Asynchronous user interaction and tool integration in isabelle/pide. volume 8558 of *Lecture Notes in Computer Science*, pages 515–530. Springer, 2014. ISBN 978-3-319-08969-0. URL https://doi.org/10.1007/978-3-319-08970-6_33.

[5] M. Wenzel. System description: Isabelle/jedit in 2014. volume 167 of *EPTCS*, pages 84–94, 2014. URL https://doi.org/10.4204/EPTCS.167.10.