An Account with my Personal, Ecclectic Comments on the Isabelle Architecture

Version : Isabelle 2017

Burkhart Wolff

November 27, 2018

Contents

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This is a programming-tutorial of Isabelle and Isabelle/HOL, a complementary text to the unfortunately somewhat outdated "The Isabelle Cookbook" in https://nms.kcl.ac.uk/christian.urban/Cookbook/.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 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","Tactic Programming"]

1 SML and Fundamental SML libraries

1.1 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. The preferred programming style is purely functional:

ML⟨ *fun fac x = if x = 0 then 1 else x ∗ fac*(*x*−*1*) ;
fac 10;
⟩
— or
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.

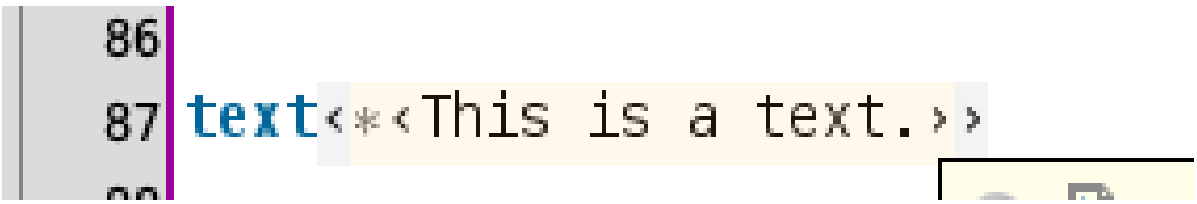

Figure 1.1: A text-element as presented in Isabelle/jedit.

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.

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.

1.2 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 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:

1.3 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:

[1] sdf

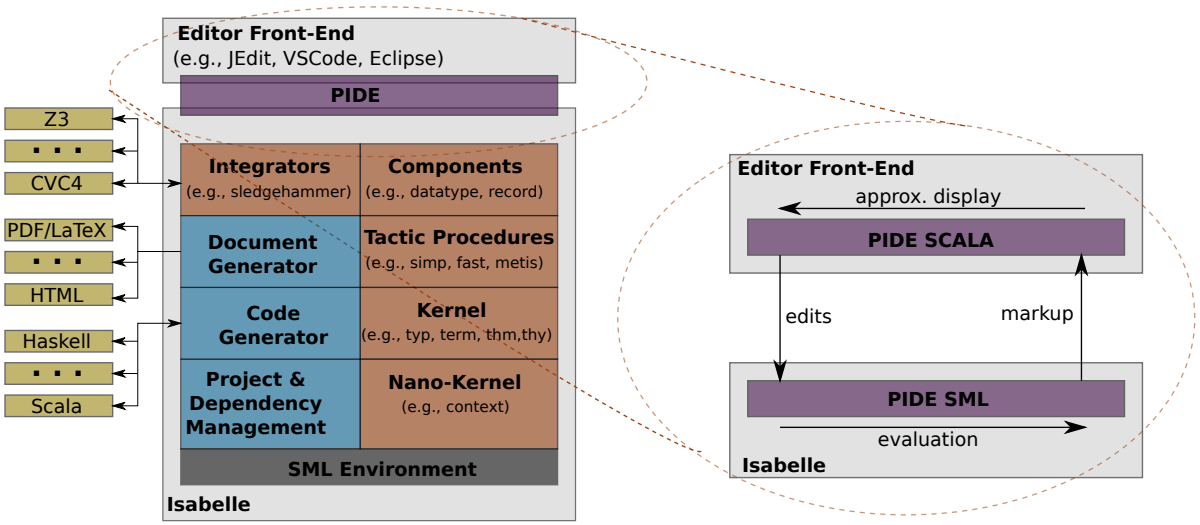

Figure 1.2: The system architecture of Isabelle (left-hand side) and the asynchronous communication between the Isabelle system and the IDE (right-hand side).

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.

2 Prover Architecture

2.1 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.

2.1.1 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*;
⟩⟩

2.1.2 Mechanism 2 : global arbitrary data structure that is attached to the global and local Isabelle context θ

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 ...* ∗)
⟩⟩

2.2 The LCF-Kernel: terms, types, theories, proof_contexts, thms

The classical LCF-style *kernel* is about

1. Types and terms of a typed λ-Calculus including constant symbols, free variables, λ-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.

2.2.1 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;
HOLogic.dest-Trueprop: term -> term;
HOLogic.Trueprop-conv: conv -> conv;
HOLogic.mk-setT: typ -> typ;
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*;
HOLogic.conj-elim: *Proof.context* −> *thm* −> *thm* ∗ *thm*;
HOLogic.conj-elims: *Proof.context* −> *thm* −> *thm list*;
HOLogic.conj: *term*;
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*;
(∗ ... ∗)
⟩

2.2.2 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)
›
```

2.2.3 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
```
⟫

2.2.4 thy and the signature interface

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

2.2.5 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.

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 Γ represents a set of local assumptions, θ the "theory" or more precisely the global context in which a formula ϕ has been constructed just by applying the following operations representing logical inference rules:

ML‹

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

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

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

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

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

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

(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 2.1:

```
(*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:

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

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;
```
⟩

2.2.6 Theories

This structure yields the datatype `hy*␣which␣becomes␣he` content of `ontext.theory*.␣In␣a␣way,␣the␣`LF-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: 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;
```
⟫

2.3 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 θ (=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 α/β-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.

2.4 The Isar Engine

ML⟪
```
Toplevel.theory;
Toplevel.presentation-context-of; (* Toplevel is a kind of table with call-back functions *)

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

fun control-antiquotation name s1 s2 =
  Thy-Output.antiquotation name (Scan.lift Args.cartouche-input)
    (fn {state, ...} => enclose s1 s2 o Thy-Output.output-text state {markdown = false});

Output.output;
```

```
Syntax.read-input ;
Input.source-content;

(*
  basic-entity @{binding const} (Args.const {proper = true, strict = false}) pretty-const #>
*)
⟩

ML⟨⟨
Config.get @{context} Thy-Output.display;
Config.get @{context} Thy-Output.source;
Config.get @{context} Thy-Output.modes;
Thy-Output.document-command;
(* is:
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));

end;

*)
⟩


ML⟨⟨ Thy-Output.document-command {markdown = true} ⟩⟩

ML⟨⟨  Latex.output-ascii;

      Latex.output-token
(* Hm, generierter output for
subsection*[Shaft-Encoder-characteristics]{ * Shaft Encoder Characteristics * } :

\begin{isamarkuptext}%
\isa{{\isacharbackslash}label{\isacharbraceleft}general{\isacharunderscore}hyps{\isacharbraceright}}%
\end{isamarkuptext}\isamarkuptrue%
\isacommand{subsection{\isacharasterisk}}\isamarkupfalse%
{\isacharbrackleft}Shaft{\isacharunderscore}Encoder{\isacharunderscore}characteristics{\isacharbrackright}{\isacharverbatimopen}\
Shaft\ Encoder\ Characteristics\ {\isacharverbatimclose}%

Generierter output for: text\<^cartouche>‹\label{sec:Shaft−Encoder−characteristics}›

\begin{isamarkuptext}%
\label{sec:Shaft−Encoder−characteristics}%
\end{isamarkuptext}\isamarkuptrue%


*)


⟩⟩

ML⟨⟨
```

```
Thy-Output.maybe-pretty-source :
    (Proof.context -> 'a -> Pretty.T) -> Proof.context -> Token.src -> 'a list -> Pretty.T list;

Thy-Output.output: Proof.context -> Pretty.T list -> string;

(* nuescht besonderes *)

fun document-antiq check-file ctxt (name, pos) =
  let
   (* val dir = master-directory (Proof-Context.theory-of ctxt); *)
   (* val - = check-path check-file ctxt dir (name, pos); *)
  in
    space-explode / name
   |> map Latex.output-ascii
   |> space-implode (Latex.output-ascii / ^ \\discretionary{}{}{})
   |> enclose \\isatt{ }
  end;

⟫
ML⟪ Type-Infer-Context.infer-types ⟫
ML⟪ Type-Infer-Context.prepare-positions ⟫
```

2.4.1 Transaction Management in the Isar-Engine : The Toplevel

```
ML⟪
Thy-Output.output-text: Toplevel.state -> {markdown: bool} -> Input.source -> string;
Thy-Output.document-command;

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)); *)
Thy-Output.document-command : {markdown: bool} -> (xstring * Position.T) option * Input.source ->
                              Toplevel.transition -> Toplevel.transition;

(* 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); *)

Thy-Output.document-command : {markdown: bool} -> (xstring * Position.T) option * Input.source ->
    Toplevel.transition -> Toplevel.transition;
    (*  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));

    *)

Thy-Output.output-text : Toplevel.state -> {markdown: bool} -> Input.source -> string ;
                    (* this is where antiquotation expansion happens : uses eval-antiquote *)


⟩⟩
```

ML⟨⟨

```
(* 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); *)

Thy-Output.document-command : {markdown: bool} -> (xstring * Position.T) option * Input.source ->
   Toplevel.transition -> Toplevel.transition;
   (*  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));

   *)

Thy-Output.output-text : Toplevel.state -> {markdown: bool} -> Input.source -> string ;
                  (* this is where antiquotation expansion happens : uses eval-antiquote *)


⟫
```

2.4.2 Configuration flags of fixed type in the Isar-engine.

ML⟪

```
Config.get @{context} Thy-Output.quotes;
Config.get @{context} Thy-Output.display;

val C = Synchronized.var Pretty.modes latEEex;
(* Synchronized: a mechanism to bookkeep global
   variables with synchronization mechanism included *)
Synchronized.value C;
(*
fun output ctxt prts =
   603   prts
   604   |> Config.get ctxt quotes ? map Pretty.quote
   605   |> (if Config.get ctxt display then
   606         map (Pretty.indent (Config.get ctxt indent) #> string-of-margin ctxt #> Output.output)
   607         #> space-implode \\isasep\\isanewline%\n
   608         #> Latex.environment isabelle
   609       else
```

```
610         map
611           ((if Config.get ctxt break then string-of-margin ctxt else Pretty.unformatted-string-of)
612             #> Output.output)
613         #> space-implode \\isasep\\isanewline%\n
614         #> enclose \\isa{ });
*)
⟩⟩
```

3 Front End

Introduction ... TODO

3.1 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 quire crappy results.

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

... produces the system output `val it = "here": bstring`, but note that it is trappy 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*)›

3.2 Parsing issues

Parsing combinators represent the ground building blocks of both generic input engines as well as the specific Isar framework. They are implemented in the structure `Token` providing core type `Token.T`.

ML‹‹ *open Token*››

ML‹‹

(∗ *Provided types* : ∗)
(∗
*type 'a parser = T list -> 'a * T list*
*type 'a context-parser = Context.generic * T list -> 'a * (Context.generic * 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;

Token.is-command;
Token.content-of; (∗ *textueller kern eines Tokens.* ∗)

⟫

Tokens and Bindings

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.make: bstring ∗ Position.T −> binding;
Binding.pos-of @{binding erzerzer};
Position.here: Position.T −> string;
(∗ Bindings ∗)
ML\<^cartouche>⟨*val X = @{here};*⟩

⟫

3.2.1 Input streams.

ML⟪
Input.source-explode : Input.source −> Symbol-Pos.T list;
(∗ conclusion: Input.source-explode converts f @{thm refl}
into:
[(, {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})]
∗)⟫

3.2.2 Scanning and combinator parsing.

Is used on two levels:

1. outer syntax, that is the syntax in which Isar-commands are written, and
2. inner-syntax, that is the syntax in which lambda-terms, and in particular HOL-terms were written.

A constant problem for newbies is this distinction, which makes it necessary that the " ... " quotes have to be used when switching to inner-syntax, except when only one literal is expected when an inner-syntax term is expected.

ML⟨
Scan.peek : ('a −> 'b −> 'c ∗ 'd) −> 'a ∗ 'b −> 'c ∗ ('a ∗ 'd);
Scan.optional: ('a −> 'b ∗ 'a) −> 'b −> 'a −> 'b ∗ 'a;
Scan.option: ('a −> 'b ∗ 'a) −> 'a −> 'b option ∗ 'a;
Scan.repeat: ('a −> 'b ∗ 'a) −> 'a −> 'b list ∗ 'a;
Scan.lift : ('a −> 'b ∗ 'c) −> 'd ∗ 'a −> 'b ∗ ('d ∗ 'c);
Scan.lift (Parse.position Args.cartouche-input);
⟩

"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 type 'a parser is already defined in the structure Token.

Syntax operations : Interface for parsing, type-checking, "reading" (both) and pretty-printing. Note that this is a late-binding interface, i.e. a collection of "hooks". The real work is done ... see below.

Encapsulates 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`.

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.!!! : *(T list −> 'a) −> T list −> 'a*;
Parse.position: *'a parser −> ('a ∗ Position.T) parser*;

(∗ *Examples* ∗)
Parse.position Args.cartouche-input;
⟩

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*;
⟩

ML⟪
fun read-terms ctxt =
grouped 10 Par-List.map-independent (Syntax.parse-term ctxt) #> Syntax.check-terms ctxt;
⟫

ML‹
(∗ More High−level, more Isar−specific Parsers ∗)
Args.name;
Args.const;
Args.cartouche-input: Input.source parser;
Args.text-token: Token.T parser;

val Z = let val attribute = Parse.position Parse.name −−
Scan.optional (Parse.$$$ = |−− Parse.!!! Parse.name) ;
in (Scan.optional(Parse.$$$, |−− (Parse.enum , attribute))) end ;
(∗ this leads to constructions like the following, where a parser for a ∗)
fn name => (Thy-Output.antiquotation name (Scan.lift (Parse.position Args.cartouche-input)));
›

3.3

ML‹*Sign.add-trrules*›

3.4 The PIDE Framework

3.4.1 Markup

Markup Operations, and reporting. Diag in Isa_DOF Foundations TR. Markup operation send via side-effect annotations to the GUI (precisely: to the PIDE Framework) that were used for hyperlinking applicating to binding occurrences, info for hovering, ...

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 ∗)

(∗ From Theory:
fun check ctxt (name, pos) =
  let
    val thy = Proof-Context.theory-of ctxt;
    val thy′ =
      Context.get-theory thy name
        handle ERROR msg =>
          let
            val completion =
              Completion.make (name, pos)
                (fn completed =>
                  map Context.theory-name (ancestors-of thy)
                  |> filter completed
                  |> sort-strings
                  |> map (fn a => (a, (Markup.theoryN, a))));
            val report = Markup.markup-report (Completion.reported-text completion);
          in error (msg ^ Position.here pos ^ report) end;
    val - = Context-Position.report ctxt pos (get-markup thy′);
  in thy′ end;

fun init-markup (name, pos) thy =
  let
    val id = serial ();
    val - = Position.report pos (theory-markup true name id pos);
  in map-thy (fn (-, -, axioms, defs, wrappers) => (pos, id, axioms, defs, wrappers)) thy end;

fun get-markup thy =
  let val {pos, id, ...} = rep-theory thy
  in theory-markup false (Context.theory-name thy) id pos end;


∗)

(∗
fun theory-markup def thy-name id pos =
  if id = 0 then Markup.empty
  else
    Markup.properties (Position.entity-properties-of def id pos)
      (Markup.entity Markup.theoryN thy-name);

fun get-markup thy =
  let val {pos, id, ...} = rep-theory thy
  in theory-markup false (Context.theory-name thy) id pos end;


fun init-markup (name, pos) thy =
  let
    val id = serial ();
    val - = Position.report pos (theory-markup true name id pos);
  in map-thy (fn (-, -, axioms, defs, wrappers) => (pos, id, axioms, defs, wrappers)) thy end;


fun check ctxt (name, pos) =
  let
```

```
  val thy = Proof-Context.theory-of ctxt;
  val thy' =
    Context.get-theory thy name
      handle ERROR msg =>
        let
          val completion =
            Completion.make (name, pos)
              (fn completed =>
                map Context.theory-name (ancestors-of thy)
                |> filter completed
                |> sort-strings
                |> map (fn a => (a, (Markup.theoryN, a))));
          val report = Markup.markup-report (Completion.reported-text completion);
        in error (msg ^ Position.here pos ^ report) end;
  val - = Context-Position.report ctxt pos (get-markup thy');
in thy' end;


*)

⟫
```

3.5 Output: Very Low Level

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

3.6 Output: LaTeX

```
ML‹
Thy-Output.verbatim-text;
Thy-Output.output-text: Toplevel.state -> {markdown: bool} -> Input.source -> string;
Thy-Output.antiquotation:
  binding ->
    'a context-parser ->
      ({context: Proof.context, source: Token.src, state: Toplevel.state} -> 'a -> string) ->
        theory -> theory;
Thy-Output.output: Proof.context -> Pretty.T list -> string;
Thy-Output.output-text: Toplevel.state -> {markdown: bool} -> Input.source -> string;

Thy-Output.output : Proof.context -> Pretty.T list -> string;
›


ML⟪
Syntax-Phases.reports-of-scope;
⟫


ML⟪
```

```
(* interesting piece for LaTeX Generation:
fun verbatim-text ctxt =
  if Config.get ctxt display then
    split-lines #> map (prefix (Symbol.spaces (Config.get ctxt indent))) #> cat-lines #>
    Latex.output-ascii #> Latex.environment isabellett
  else
    split-lines #>
    map (Latex.output-ascii #> enclose \\isatt{ }) #>
    space-implode \\isasep\\isanewline%\n;

(* From structure Thy-Output *)
fun pretty-const ctxt c =
  let
    val t = Const (c, Consts.type-scheme (Proof-Context.consts-of ctxt) c)
      handle TYPE (msg, -, -) => error msg;
    val ([t'], -) = Variable.import-terms true [t] ctxt;
  in pretty-term ctxt t' end;

  basic-entity @{binding const} (Args.const {proper = true, strict = false}) pretty-const #>

*)

Pretty.enclose : string -> string -> Pretty.T list -> Pretty.T; (* not to confuse with: String.enclose *)

(* At times, checks where attached to Pretty - functions and exceptions used to
   stop the execution/validation of a command *)
fun pretty-theory ctxt (name, pos) = (Theory.check ctxt (name, pos); Pretty.str name);
Pretty.enclose;
Pretty.str;
Pretty.mark-str;
Pretty.text bla-d;

Pretty.quote; (* Pretty.T transformation adding  *)
Pretty.unformatted-string-of : Pretty.T -> string ;

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


(*
fun pretty-command (cmd as (name, Command {comment, ...})) =
  Pretty.block
    (Pretty.marks-str
      ([Active.make-markup Markup.sendbackN {implicit = true, properties = [Markup.padding-line]},
        command-markup false cmd], name) :: Pretty.str : :: Pretty.brk 2 :: Pretty.text comment);
*)


⟩

ML⟨
Thy-Output.output-text;
(* is:
fun output-text state {markdown} source =
```

```
  let
    val is-reported =
      (case try Toplevel.context-of state of
        SOME ctxt => Context-Position.is-visible ctxt
      | NONE => true);

    val pos = Input.pos-of source;
    val syms = Input.source-explode source;

    val - =
      if is-reported then
        Position.report pos (Markup.language-document (Input.is-delimited source))
      else ();

    val output-antiquotes = map (eval-antiquote state) #> implode;

    fun output-line line =
      (if Markdown.line-is-item line then \\item  else ) ^
        output-antiquotes (Markdown.line-content line);

    fun output-blocks blocks = space-implode \n\n (map output-block blocks)
    and output-block (Markdown.Par lines) = cat-lines (map output-line lines)
      | output-block (Markdown.List {kind, body, ...}) =
          Latex.environment (Markdown.print-kind kind) (output-blocks body);
  in
    if Toplevel.is-skipped-proof state then
    else if markdown andalso exists (Markdown.is-control o Symbol-Pos.symbol) syms
    then
      let
        val ants = Antiquote.parse pos syms;
        val reports = Antiquote.antiq-reports ants;
        val blocks = Markdown.read-antiquotes ants;
        val - = if is-reported then Position.reports (reports @ Markdown.reports blocks) else ();
      in output-blocks blocks end
    else
      let
        val ants = Antiquote.parse pos (Symbol-Pos.trim-blanks syms);
        val reports = Antiquote.antiq-reports ants;
        val - = if is-reported then Position.reports (reports @ Markdown.text-reports ants) else ();
      in output-antiquotes ants end
  end;
*)

⟩⟩


ML⟨⟨
Outer-Syntax.print-commands @{theory};

Outer-Syntax.command : Outer-Syntax.command-keyword -> string ->
                       (Toplevel.transition -> Toplevel.transition) parser -> unit;
(* creates an implicit thy-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.
   *)

(*Example: text is :
```

val - =
Outer-Syntax.command (*text*, @{*here*}) *formal comment* (*primary style*)
(*Parse.opt-target* −− *Parse.document-source* >> *Thy-Output.document-command* {*markdown* = *true*});
∗)

(∗ *not exported*: *Thy-Output.output-token*; *Ich glaub*, *da passierts* ... ∗)
Thy-Output.present-thy;
⟩⟩

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 `~~/src/Pure/Syntax/syntax_phases.ML`, it sets up the entire syntax engine (the hooks) via:

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

end