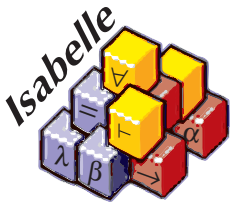

Old Isabelle Reference Manual

Lawrence C. Paulson
Computer Laboratory
University of Cambridge
lcp@cl.cam.ac.uk

With Contributions by Tobias Nipkow and Markus Wenzel

12 February 2013

Note: this document is part of the earlier Isabelle documentation and is mostly outdated. Fully obsolete parts of the original text have already been removed. The remaining material covers some aspects that did not make it into the newer manuals yet.

Acknowledgements

Tobias Nipkow, of T. U. Munich, wrote most of Chapters **??** and **??**. Markus Wenzel contributed to Chapter 2. Jeremy Dawson, Sara Kalvala, Martin Simons and others suggested changes and corrections. The research has been funded by the EPSRC (grants GR/G53279, GR/H40570, GR/K57381, GR/K77051, GR/M75440) and by ESPRIT (projects 3245: Logical Frameworks, and 6453: Types), and by the DFG Schwerpunktprogramm *Deduktion*.

Contents

Chapter 1

Theorems and Forward Proof

1.1 *Sort hypotheses

```
strip_shyps         : thm -> thm
strip_shyps_warning : thm -> thm
```

Isabelle's type variables are decorated with sorts, constraining them to certain ranges of types. This has little impact when sorts only serve for syntactic classification of types — for example, FOL distinguishes between terms and other types. But when type classes are introduced through axioms, this may result in some sorts becoming *empty*: where one cannot exhibit a type belonging to it because certain sets of axioms are unsatisfiable.

If a theorem contains a type variable that is constrained by an empty sort, then that theorem has no instances. It is basically an instance of *ex falso quodlibet*. But what if it is used to prove another theorem that no longer involves that sort? The latter theorem holds only if under an additional non-emptiness assumption.

Therefore, Isabelle's theorems carry around sort hypotheses. The `shyps` field is a list of sorts occurring in type variables in the current `prop` and `hyps` fields. It may also includes sorts used in the theorem's proof that no longer appear in the `prop` or `hyps` fields — so-called *dangling* sort constraints. These are the critical ones, asserting non-emptiness of the corresponding sorts.

Isabelle automatically removes extraneous sorts from the `shyps` field at the end of a proof, provided that non-emptiness can be established by looking at the theorem's signature: from the `classes` and `arities` information. This operation is performed by `strip_shyps` and `strip_shyps_warning`.

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

`strip_shyps_warning` is like `strip_shyps`, but issues a warning message of any pending sort hypotheses that do not have a (syntactic) witness.

1.2 Proof terms

Isabelle can record the full meta-level proof of each theorem. The proof term contains all logical inferences in detail. Resolution and rewriting steps are broken down to primitive rules of the meta-logic. The proof term can be inspected by a separate proof-checker, for example.

According to the well-known *Curry-Howard isomorphism*, a proof can be viewed as a λ-term. Following this idea, proofs in Isabelle are internally represented by a datatype similar to the one for terms described in §**??**.

```
infix 8 % %%;

datatype proof =
   PBound of int
 | Abst of string * typ option * proof
 | AbsP of string * term option * proof
 | op % of proof * term option
 | op %% of proof * proof
 | Hyp of term
 | PThm of (string * (string * string list) list) *
           proof * term * typ list option
 | PAxm of string * term * typ list option
 | Oracle of string * term * typ list option
 | MinProof of proof list;
```

`Abst` (*a*, τ, *prf*) is the abstraction over a *term variable* of type τ in the body *prf*. Logically, this corresponds to $\bigwedge$ introduction. The name *a* is used only for parsing and printing.

`AbsP` (*a*, φ, *prf*) is the abstraction over a *proof variable* standing for a proof of proposition φ in the body *prf*. This corresponds to $\Longrightarrow$ introduction.

prf `%` *t* is the application of proof *prf* to term *t* which corresponds to $\bigwedge$ elimination.

prf_1 `%%` prf_2 is the application of proof prf_1 to proof prf_2 which corresponds to $\Longrightarrow$ elimination.

`PBound` *i* is a *proof variable* with de Bruijn [2] index *i*.

`Hyp` φ corresponds to the use of a meta level hypothesis φ.

`PThm` ((*name*, *tags*), *prf*, φ, $\overline{\tau}$) stands for a pre-proved theorem, where *name* is the name of the theorem, *prf* is its actual proof, φ is the proven proposition, and $\overline{\tau}$ is a type assignment for the type variables occurring in the proposition.

`PAxm` (*name*, φ, $\overline{\tau}$) corresponds to the use of an axiom with name *name* and proposition φ, where $\overline{\tau}$ is a type assignment for the type variables occurring in the proposition.

`Oracle` (*name*, φ, $\overline{\tau}$) denotes the invocation of an oracle with name *name* which produced a proposition φ, where $\overline{\tau}$ is a type assignment for the type variables occurring in the proposition.

`MinProof` *prfs* represents a *minimal proof* where *prfs* is a list of theorems, axioms or oracles.

Note that there are no separate constructors for abstraction and application on the level of *types*, since instantiation of type variables is accomplished via the type assignments attached to `Thm`, `Axm` and `Oracle`.

Each theorem's derivation is stored as the `der` field of its internal record:

```
#2 (#der (rep_thm conjI));
  PThm (("HOL.conjI", []),
    AbsP ("H", None, AbsP ("H", None, ...)), ..., None) %
      None % None : Proofterm.proof
```

This proof term identifies a labelled theorem, `conjI` of theory `HOL`, whose underlying proof is `AbsP ("H", None, AbsP ("H", None, ...))`. The theorem is applied to two (implicit) term arguments, which correspond to the two variables occurring in its proposition.

Isabelle's inference kernel can produce proof objects with different levels of detail. This is controlled via the global reference variable `proofs`:

`proofs := 0;` only record uses of oracles

`proofs := 1;` record uses of oracles as well as dependencies on other theorems and axioms

`proofs := 2;` record inferences in full detail

Reconstruction and checking of proofs as described in §1.2.1 will not work for proofs constructed with `proofs` set to `0` or `1`. Theorems involving oracles will be printed with a suffixed `[!]` to point out the different quality of confidence achieved.

The dependencies of theorems can be viewed using the function `thm_deps`:

`thm_deps [`thm_1`, ..., `thm_n`];`

generates the dependency graph of the theorems thm_1, ..., thm_n and displays it using Isabelle's graph browser. For this to work properly, the theorems in question have to be proved with `proofs` set to a value greater than `0`. You can use

```
ThmDeps.enable : unit -> unit
ThmDeps.disable : unit -> unit
```

to set `proofs` appropriately.

1.2.1 Reconstructing and checking proof terms

When looking at the above datatype of proofs more closely, one notices that some arguments of constructors are *optional*. The reason for this is that keeping a full proof term for each theorem would result in enormous memory requirements. Fortunately, typical proof terms usually contain quite a lot of redundant information that can be reconstructed from the context. Therefore, Isabelle's inference kernel creates only *partial* (or *implicit*) proof terms, in which all typing information in terms, all term and type labels of abstractions `AbsP` and `Abst`, and (if possible) some argument terms of `%` are omitted. The following functions are available for reconstructing and checking proof terms:

```
Reconstruct.reconstruct_proof :
  Sign.sg -> term -> Proofterm.proof -> Proofterm.proof
Reconstruct.expand_proof :
  Sign.sg -> string list -> Proofterm.proof -> Proofterm.proof
ProofChecker.thm_of_proof : theory -> Proofterm.proof -> thm
```

`Reconstruct.reconstruct_proof` *sg t prf* turns the partial proof *prf* into a full proof of the proposition denoted by t, with respect to signature *sg*. Reconstruction will fail with an error message if *prf* is not a proof of t, is ill-formed, or does not contain sufficient information for reconstruction by *higher order pattern unification* [3, 1]. The latter may only happen for proofs built up "by hand" but not for those produced automatically by Isabelle's inference kernel.

`Reconstruct.expand_proof` *sg* `[`$name_1$`, ..., `$name_n$`]` *prf* expands and reconstructs the proofs of all theorems with names $name_1$, ..., $name_n$ in the (full) proof *prf*.

`ProofChecker.thm_of_proof` *thy prf* turns the (full) proof *prf* into a theorem with respect to theory *thy* by replaying it using only primitive rules from Isabelle's inference kernel.

1.2.2 Parsing and printing proof terms

Isabelle offers several functions for parsing and printing proof terms. The concrete syntax for proof terms is described in Fig. 1.1. Implicit term arguments in partial proofs are indicated by "`_`". Type arguments for theorems

$$
\begin{array}{rcl}
\mathit{proof} & = & \texttt{Lam}\ \mathit{params}\,.\ \mathit{proof} \quad | \quad \Lambda\mathit{params}\,.\ \mathit{proof} \\
& | & \mathit{proof}\ \texttt{\%}\ \mathit{any} \quad | \quad \mathit{proof} \cdot \mathit{any} \\
& | & \mathit{proof}\ \texttt{\%\%}\ \mathit{proof} \quad | \quad \mathit{proof} \bullet \mathit{proof} \\
& | & \mathit{id} \quad | \quad \mathit{longid} \\
\\
\mathit{param} & = & \mathit{idt} \quad | \quad \mathit{idt}\ \texttt{:}\ \mathit{prop} \quad | \quad \texttt{(}\ \mathit{param}\ \texttt{)} \\
\\
\mathit{params} & = & \mathit{param} \quad | \quad \mathit{param}\ \mathit{params}
\end{array}
$$

Figure 1.1: Proof term syntax

and axioms may be specified using `%` or "$\cdot$" with an argument of the form `TYPE(`*type*`)` (see §??). They must appear before any other term argument of a theorem or axiom. In contrast to term arguments, type arguments may be completely omitted.

```
ProofSyntax.read_proof : theory -> bool -> string -> Proofterm.proof
ProofSyntax.pretty_proof : Sign.sg -> Proofterm.proof -> Pretty.T
ProofSyntax.pretty_proof_of : bool -> thm -> Pretty.T
ProofSyntax.print_proof_of : bool -> thm -> unit
```

The function `read_proof` reads in a proof term with respect to a given theory. The boolean flag indicates whether the proof term to be parsed contains explicit typing information to be taken into account. Usually, typing information is left implicit and is inferred during proof reconstruction. The pretty printing functions operating on theorems take a boolean flag as an argument which indicates whether the proof term should be reconstructed before printing.

The following example (based on Isabelle/HOL) illustrates how to parse and check proof terms. We start by parsing a partial proof term

```
val prf = ProofSyntax.read_proof Main.thy false
  "impI % _ % _ %% (Lam H : _. conjE % _ % _ % _ %% H %%
     (Lam (H1 : _) H2 : _. conjI % _ % _ %% H2 %% H1))";
  val prf = PThm (("HOL.impI", []), ..., ..., None) % None % None %%
    AbsP ("H", None, PThm (("HOL.conjE", []), ..., ..., None) %
      None % None % None %% PBound 0 %%
      AbsP ("H1", None, AbsP ("H2", None, ...))) : Proofterm.proof
```

The statement to be established by this proof is

```
val t = term_of
  (read_cterm (sign_of Main.thy) ("A & B --> B & A", propT));
  val t = Const ("Trueprop", "bool => prop") $
    (Const ("op -->", "[bool, bool] => bool") $
      ... $ ... : Term.term
```

Using `t` we can reconstruct the full proof

```
val prf' = Reconstruct.reconstruct_proof (sign_of Main.thy) t prf;
  val prf' = PThm (("HOL.impI", []), ..., ..., Some []) %
    Some (Const ("op &", ...) $ Free ("A", ...) $ Free ("B", ...)) %
    Some (Const ("op &", ...) $ Free ("B", ...) $ Free ("A", ...)) %%
    AbsP ("H", Some (Const ("Trueprop", ...) $ ...), ...)
      : Proofterm.proof
```

This proof can finally be turned into a theorem

```
val thm = ProofChecker.thm_of_proof Main.thy prf';
  val thm = "A & B --> B & A" : Thm.thm
```

Chapter 2

Syntax Transformations

2.1 Transforming parse trees to ASTs

The parse tree is the raw output of the parser. Translation functions, called **parse AST translations**, transform the parse tree into an abstract syntax tree.

The parse tree is constructed by nesting the right-hand sides of the productions used to recognize the input. Such parse trees are simply lists of tokens and constituent parse trees, the latter representing the nonterminals of the productions. Let us refer to the actual productions in the form displayed by `print_syntax` (see §**??** for an example).

Ignoring parse AST translations, parse trees are transformed to ASTs by stripping out delimiters and copy productions. More precisely, the mapping $[\![-]\!]$ is derived from the productions as follows:

- Name tokens: $[\![t]\!] = \texttt{Variable}\, s$, where t is an `id`, `var`, `tid`, `tvar`, `num`, `xnum` or `xstr` token, and s its associated string. Note that for `xstr` this does not include the quotes.

- Copy productions: $[\![\ldots P \ldots]\!] = [\![P]\!]$. Here $\ldots$ stands for strings of delimiters, which are discarded. P stands for the single constituent that is not a delimiter; it is either a nonterminal symbol or a name token.

- 0-ary productions: $[\![\ldots \texttt{=>} c]\!] = \texttt{Constant}\, c$. Here there are no constituents other than delimiters, which are discarded.

- n-ary productions, where $n \geq 1$: delimiters are discarded and the remaining constituents $P_1, \ldots, P_n$ are built into an application whose head constant is c:

$$[\![\ldots P_1 \ldots P_n \ldots \texttt{=>} c]\!] = \texttt{Appl}\,[\texttt{Constant}\, c, [\![P_1]\!], \ldots, [\![P_n]\!]]$$

Figure 2.1 presents some simple examples, where `==`, `_appl`, `_args`, and so forth name productions of the Pure syntax. These examples illustrate the

input string	AST
`"f"`	`f`
`"'a"`	`'a`
`"t == u"`	`("==" t u)`
`"f(x)"`	`("_appl" f x)`
`"f(x, y)"`	`("_appl" f ("_args" x y))`
`"f(x, y, z)"`	`("_appl" f ("_args" x ("_args" y z)))`
`"%x y. t"`	`("_lambda" ("_idts" x y) t)`

Figure 2.1: Parsing examples using the Pure syntax

input string	AST
`"f(x, y, z)"`	`(f x y z)`
`"'a ty"`	`(ty 'a)`
`"('a, 'b) ty"`	`(ty 'a 'b)`
`"%x y z. t"`	`("_abs" x ("_abs" y ("_abs" z t)))`
`"%x :: 'a. t"`	`("_abs" ("_constrain" x 'a) t)`
`"[| P; Q; R |] => S"`	`("==>" P ("==>" Q ("==>" R S)))`
`"['a, 'b, 'c] => 'd"`	`("fun" 'a ("fun" 'b ("fun" 'c 'd)))`

Figure 2.2: Built-in parse AST translations

need for further translations to make ASTs closer to the typed λ-calculus. The Pure syntax provides predefined parse AST translations for ordinary applications, type applications, nested abstractions, meta implications and function types. Figure 2.2 shows their effect on some representative input strings.

The names of constant heads in the AST control the translation process. The list of constants invoking parse AST translations appears in the output of `print_syntax` under `parse_ast_translation`.

2.2 Transforming ASTs to terms

The AST, after application of macros (see §??), is transformed into a term. This term is probably ill-typed since type inference has not occurred yet. The term may contain type constraints consisting of applications with head `"_constrain"`; the second argument is a type encoded as a term. Type inference later introduces correct types or rejects the input.

Another set of translation functions, namely parse translations, may affect

this process. If we ignore parse translations for the time being, then ASTs are transformed to terms by mapping AST constants to constants, AST variables to schematic or free variables and AST applications to applications.

More precisely, the mapping $[\![-]\!]$ is defined by

- Constants: $[\![\texttt{Constant}\, x]\!] = \texttt{Const}(x, \texttt{dummyT})$.
- Schematic variables: $[\![\texttt{Variable "?}xi\texttt{"}]\!] = \texttt{Var}((x, i), \texttt{dummyT})$, where x is the base name and i the index extracted from xi.
- Free variables: $[\![\texttt{Variable}\, x]\!] = \texttt{Free}(x, \texttt{dummyT})$.
- Function applications with n arguments:

$$[\![\texttt{Appl}\, [f, x_1, \ldots, x_n]]\!] = [\![f]\!]\ \texttt{\$}\ [\![x_1]\!]\ \texttt{\$} \ldots \texttt{\$}\ [\![x_n]\!]$$

Here `Const`, `Var`, `Free` and `$` are constructors of the datatype `term`, while `dummyT` stands for some dummy type that is ignored during type inference.

So far the outcome is still a first-order term. Abstractions and bound variables (constructors `Abs` and `Bound`) are introduced by parse translations. Such translations are attached to `"_abs"`, `"!!"` and user-defined binders.

2.3 Printing of terms

The output phase is essentially the inverse of the input phase. Terms are translated via abstract syntax trees into strings. Finally the strings are pretty printed.

Print translations (§**??**) may affect the transformation of terms into ASTs. Ignoring those, the transformation maps term constants, variables and applications to the corresponding constructs on ASTs. Abstractions are mapped to applications of the special constant `_abs`.

More precisely, the mapping $[\![-]\!]$ is defined as follows:

- $[\![\texttt{Const}(x, \tau)]\!] = \texttt{Constant}\, x$.
- $[\![\texttt{Free}(x, \tau)]\!] = \mathit{constrain}(\texttt{Variable}\, x, \tau)$.
- $[\![\texttt{Var}((x, i), \tau)]\!] = \mathit{constrain}(\texttt{Variable "?}xi\texttt{"}, \tau)$, where `?`$xi$ is the string representation of the `indexname` (x, i).
- For the abstraction $\lambda x :: \tau\ .\ t$, let x' be a variant of x renamed to differ from all names occurring in t, and let t' be obtained from t by replacing all bound occurrences of x by the free variable x'. This

replaces corresponding occurrences of the constructor `Bound` by the term `Free`$(x', \texttt{dummyT})$:

$$[\![\texttt{Abs}(x, \tau, t)]\!] = \texttt{Appl}\,[\texttt{Constant "\_abs"}, \mathit{constrain}(\texttt{Variable}\,x', \tau), [\![t']\!]]$$

- $[\![\texttt{Bound}\,i]\!]$ = `Variable "B.`i`"`. The occurrence of constructor `Bound` should never happen when printing well-typed terms; it indicates a de Bruijn index with no matching abstraction.
- Where f is not an application,

$$[\![f\ \texttt{\$}\ x_1\ \texttt{\$} \ldots \texttt{\$}\ x_n]\!] = \texttt{Appl}\,[[\![f]\!], [\![x_1]\!], \ldots, [\![x_n]\!]]$$

Type constraints are inserted to allow the printing of types. This is governed by the boolean variable `show_types`:

- $\mathit{constrain}(x, \tau) = x$ if $\tau =$ `dummyT` or `show_types` is set to `false`.
- $\mathit{constrain}(x, \tau) = \texttt{Appl}\,[\texttt{Constant "\_constrain"}, x, [\![\tau]\!]]$ otherwise.

  Here, $[\![\tau]\!]$ is the AST encoding of τ: type constructors go to `Constant`s; type identifiers go to `Variable`s; type applications go to `Appl`s with the type constructor as the first element. If `show_sorts` is set to `true`, some type variables are decorated with an AST encoding of their sort.

The AST, after application of macros (see §**??**), is transformed into the final output string. The built-in **print AST translations** reverse the parse AST translations of Fig. 2.2.

For the actual printing process, the names attached to productions of the form $\ldots A_1^{(p_1)} \ldots A_n^{(p_n)} \ldots$ `=>`c play a vital role. Each AST with constant head c, namely `"`c`"` or (`"`c`"` $x_1 \ldots x_n$), is printed according to the production for c. Each argument x_i is converted to a string, and put in parentheses if its priority (p_i) requires this. The resulting strings and their syntactic sugar (denoted by $\ldots$ above) are joined to make a single string.

If an application (`"`c`"` $x_1 \ldots x_m$) has more arguments than the corresponding production, it is first split into ((`"`c`"` $x_1 \ldots x_n$) $x_{n+1} \ldots x_m$). Applications with too few arguments or with non-constant head or without a corresponding production are printed as $f(x_1, \ldots, x_l)$ or $(\alpha_1, \ldots, \alpha_l)ty$. Multiple productions associated with some name c are tried in order of appearance. An occurrence of `Variable`$\,x$ is simply printed as x.

Blanks are *not* inserted automatically. If blanks are required to separate tokens, specify them in the mixfix declaration, possibly preceded by a slash (`/`) to allow a line break.

Bibliography

[1] Stefan Berghofer and Tobias Nipkow. Proof terms for simply typed higher order logic. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 38–52. Springer-Verlag, 2000.

[2] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indag. Math.*, 34:381–392, 1972.

[3] Tobias Nipkow. Functional unification of higher-order patterns. In M. Vardi, editor, *Eighth Annual Symposium on Logic in Computer Science*, pages 64–74. IEEE Computer Society Press, 1993.