# The Isabelle Programmer's Cookbook (fragment)

with contributions by:

Alexander Krauss
Jeremy Dawson
Stefan Berghofer

October 13, 2008

# Contents

# Chapter 1

# Introduction

The purpose of this cookbook is to guide the reader through the first steps of Isabelle programming, and to provide recipes for solving common problems.

## 1.1 Intended Audience and Prior Knowledge

This cookbook targets an audience who already knows how to use Isabelle for writing theories and proofs. We also assume that readers are familiar with the Standard ML, the programming language in which most of Isabelle is implemented. If you are unfamiliar with either of these two subjects, you should first work through the Isabelle/HOL tutorial [4] and Paulson's book on Standard ML [5].

## 1.2 Existing Documentation

The following documentation about Isabelle programming already exist (they are included in the distribution of Isabelle):

**The Implementation Manual** describes Isabelle from a programmer's perspective, documenting both the underlying concepts and some of the interfaces.

**The Isabelle Reference Manual** is an older document that used to be the main reference at a time when all proof scripts were written on the ML level. Many parts of this manual are outdated now, but some parts, particularly the chapters on tactics, are still useful.

Then of course there is:

**The code** is of course the ultimate reference for how things really work. Therefore you should not hesitate to look at the way things are actually implemented. More importantly, it is often good to look at code that does similar things as you want to do, to learn from other people's code.

Since Isabelle is not a finished product, these manuals, just like the implementation itself, are always under construction. This can be difficult and frustrating at times, especially when interfaces changes occur frequently. But it is a reality that progress means changing things (FIXME: need some short and convincing comment that this is a strategy, not a problem that should be solved).

# Chapter 2

# First Steps

Isabelle programming is done in Standard ML. Just like lemmas and proofs, code in Isabelle is part of a theory. If you want to follow the code written in this chapter, we assume you are working inside the theory defined by

**theory** CookBook
**imports** Main
**begin**
...

## 2.1 Including ML-Code

The easiest and quickest way to include code in a theory is by using the **ML** command. For example

```
ML {*
  3 + 4
*}
```

Expressions inside **ML** commands are immediately evaluated, like "normal" Isabelle proof scripts, by using the advance and undo buttons of your Isabelle environment. The code inside the **ML** command can also contain value and function bindings. However on such **ML** commands the undo operation behaves slightly counter-intuitive, because if you define

```
ML {*
  val foo = true
*}
```

then Isabelle's undo operation has no effect on the definition of `foo`.

Once a portion of code is relatively stable, one usually wants to export it to a separate ML-file. Such files can then be included in a theory by using **uses** in the header of the theory, like

**theory** CookBook
**imports** Main
**uses** `"file_to_be_included.ML"` ...
**begin**
...

## 2.2 Debugging and Printing

During developments you might find it necessary to quickly inspect some data in your code. This can be done in a "quick-and-dirty" fashion using the function `warning`. For example

```
ML {* warning "any string" *}
```

will print out `"any string"` inside the response buffer of Isabelle. If you develop under PolyML, then there is a convenient, though again "quick-and-dirty", method for converting values into strings, for example:

```
ML {* warning (makestring 1) *}
```

However this only works if the type of what is converted is monomorphic and not a function.

The funtion warning should only be used for testing purposes, because any output this funtion generates will be overwritten, as soon as an error is raised. Therefore for printing anything more serious and elaborate, the function `tracing` should be used. This function writes all output into a separate buffer.

```
ML {* tracing "foo" *}
```

It is also possible to redirect the channel where the `foo` is printed to a separate file, e.g. to prevent Proof General from choking on massive amounts of trace output. This rediretion can be achieved using the code

```
ML {*
  val strip_specials =
  let
    fun strip ("\^A" :: _ :: cs) = strip cs
      | strip (c :: cs) = c :: strip cs
      | strip [] = [];
  in implode o strip o explode end;

  fun redirect_tracing stream =
  Output.tracing_fn := (fn s =>
    (TextIO.output (stream, (strip_specials s));
     TextIO.output (stream, "\n");
     TextIO.flushOut stream));
*}
```

Calling `redirect_tracing` with `(TextIO.openOut "foo.bar")` will cause that all tracing information is printed into the file `foo.bar`.

## 2.3 Antiquotations

The main advantage of embedding all code in a theory is that the code can contain references to entities defined on the logical level of Isabelle. This is done using antiquotations. For example, one can print out the name of the current theory by typing

```
ML {* Context.theory_name @{theory} *}
```

where *@{theory}* is an antiquotation that is substituted with the current theory (remember that we assumed we are inside the theory CookBook). The name of this theory can be extracted using the function `Context.theory_name`. So the code above returns the string `"CookBook"`.

Note, however, that antiquotations are statically scoped, that is the value is determined at "compile-time", not "run-time". For example the function

```
ML {*
  fun not_current_thyname () = Context.theory_name @{theory}
*}
```

does *not* return the name of the current theory, if it is run in a different theory. Instead, the code above defines the constant function that always returns the string "CookBook", no matter where the function is called. Operationally speaking, `@{theory}` is *not* replaced with code that will look up the current theory in some data structure and return it. Instead, it is literally replaced with the value representing the theory name.

In a similar way you can use antiquotations to refer to theorems or simpsets:

```
ML {* @{thm allI} *}
ML {* @{simpset} *}
```

In the course of this introduction, we will learn more about these antiquotations: they greatly simplify Isabelle programming since one can directly access all kinds of logical elements from ML.

## 2.4 Terms and Types

One way to construct terms of Isabelle on the ML level is by using the antiquotation `@{term ...}`:

```
ML {* @{term "(a::nat) + b = c"} *}
```

This will show the term `a + b = c`, but printed out using the internal representation of this term. This internal representation corresponds to the datatype `term`.

The internal representation of terms uses the usual de Bruijn index mechanism where bound variables are represented by the constructor `Bound`. The index in `Bound` refers to the number of Abstractions (`Abs`) we have to skip until we hit the `Abs` that binds the corresponding variable. However, in Isabelle the names of bound variables are kept at abstractions for printing purposes, and so should be treated only as comments.

*Terms are described in detail in [Impl. Man., Sec. 2.2]. Their definition and many useful operations can be found in* `Pure/term.ML`. **Read More**

Sometimes the internal representation of terms can be surprisingly different from what you see at the user level, because the layers of parsing/type checking/pretty printing can be quite elaborate.

**Exercise 2.4.1.** *Look at the internal term representation of the following terms, and find out why they are represented like this.*

- `case x of 0 ⇒ 0 | Suc y ⇒ y`
- `λ(x, y). P y x`
- `{[x] |x. x ≤ -2}`

*Hint: The third term is already quite big, and the pretty printer may omit parts of it by default. If you want to see all of it, you can use the following ML funtion to set the limit to a value high enough:*

```
ML {* print_depth 50 *}
```

The antiquotation `@{prop ...}` constructs terms of propositional type, inserting the invisible `Trueprop` coercions whenever necessary. Consider for example

```
ML {* @{term "P x"} ; @{prop "P x"} *}
```

which inserts the coercion in the latter case and

```
ML {* @{term "P x ⟹ Q x"} ; @{prop "P x ⟹ Q x"} *}
```

which does not.

Types can be constructed using the antiquotation `@{typ ...}`. For example

```
ML {* @{typ "bool ⇒ nat"} *}
```

(FIXME: Unlike the term antiquotation, `@{typ ...}` does not print the internal representation. Is there a reason for this, that needs to be explained here?)

*Types are described in detail in [Impl. Man., Sec. 2.1]. Their definition and many useful operations can be found in* `Pure/type.ML`*.* **Read More**

## 2.5 Constructing Terms and Types Manually

While antiquotations are very convenient for constructing terms and types, they can only construct fixed terms. Unfortunately, one often needs to construct them dynamically. For example, a function that returns the implication `⋀(x::τ). P x ⟹ Q x` taking `P`, `Q` and the typ `τ` as arguments can only be written as

```
ML {*
  fun make_imp P Q tau =
  let
    val x = Free ("x",tau)
  in Logic.all x (Logic.mk_implies (HOLogic.mk_Trueprop (P $ x),
                                    HOLogic.mk_Trueprop (Q $ x)))
  end
*}
```

The reason is that one cannot pass the arguments `P`, `Q` and `tau` into an antiquotation. For example the following does not work.

```
ML {*
  fun make_wrong_imp P Q tau = @{prop "⋀x. P x ⟹ Q x"}
*}
```

To see this apply `@{term S}`, `@{term T}` and `@{typ nat}` to both functions.

One tricky point in constructing terms by hand is to obtain the fully qualified name for constants. For example the names for `zero` or `+` are more complex than one first expects, namely

`HOL.zero_class.zero` and `HOL.plus_class.plus`.

The extra prefixes `zero_class` and `plus_class` are present because these constants are defined within type classes; the prefix `HOL` indicates in which theory they are defined. Guessing such internal names can sometimes be quite hard. Therefore Isabellle provides the antiquotation `@{const_name ...}` which does the expansion automatically, for example:

(FIXME: Is it useful to explain *@{const_syntax}*?)

(FIXME: how to construct types manually)

*There are many functions in `Pure/logic.ML` and `HOL/hologic.ML` that make such manual constructions of terms easier.* **Read More**

Have a look at these files and try to solve the following two exercises:

**Exercise 2.5.1.** *Write a function* `rev_sum : term -> term` *that takes a term of the form* $t_1 + t_2 + \ldots + t_n$ *(whereby i might be zero) and returns the reversed sum* $t_n + \ldots + t_2 + t_1$*. Assume the* $t_i$ *can be arbitrary expressions and also note that* `+` *associates to the left. Try your function on some examples.*

**Exercise 2.5.2.** *Write a function which takes two terms representing natural numbers in unary (like* `Suc (Suc (Suc 0))`*), and produce the unary number representing their sum.*

## 2.6 Type Checking

We can freely construct and manipulate terms, since they are just arbitrary unchecked trees. However, we eventually want to see if a term is well-formed, or type checks, relative to a theory. Type checking is done via the function `cterm_of`, which turns a `term` into a `cterm`, a *certified* term. Unlike `terms`, which are just trees, `cterms` are abstract objects that are guaranteed to be type-correct, and that can only be constructed via the official interfaces.

Type checking is always relative to a theory context. For now we can use the `@{theory}` antiquotation to get hold of the current theory. For example we can write:

```
ML {* cterm_of @{theory} @{term "(a::nat) + b = c"} *}
```

and

```
ML {*
  let
    val natT = @{typ "nat"}
    val zero = @{term "0::nat"}
  in
    cterm_of @{theory}
        (Const (@{const_name plus}, natT --> natT --> natT)
         $ zero $ zero)
  end
*}
```

**Exercise 2.6.1.** *Check that the function defined in Exercise 2.5.1 returns a result that type checks.*

## 2.7 Theorems

Just like `cterms`, theorems (of type `thm`) are abstract objects that can only be built by going through the kernel interfaces, which means that all your proofs will be checked.

To see theorems in "action", let us give a proof for the following statement

**lemma**

**assumes** *assm*$_1$: "⋀(x::nat). P x ⟹ Q x"
**and** *assm*$_2$: "P t"
**shows** "Q t"

on the ML level:[1]

**ML** {*
```
let
  val thy = @{theory}

  val assm1 = cterm_of thy @{prop "⋀(x::nat). P x ⟹ Q x"}
  val assm2 = cterm_of thy @{prop "((P::nat⇒bool) t)"}

  val Pt_implies_Qt =
        assume assm1
        |> forall_elim (cterm_of thy @{term "t::nat"});

  val Qt = implies_elim Pt_implies_Qt (assume assm2);
in

  Qt
  |> implies_intr assm2
  |> implies_intr assm1
end
*}
```

This code-snippet constructs the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\bigwedge x.\ P\ x \Longrightarrow Q\ x \vdash \bigwedge x.\ P\ x \Longrightarrow Q\ x}\,(\mathit{assume})}{\bigwedge x.\ P\ x \Longrightarrow Q\ x \vdash P\ t \Longrightarrow Q\ t}\,(\bigwedge\text{-}\mathit{elim}) \qquad \dfrac{}{P\ t \vdash P\ t}\,(\mathit{assume})}{\bigwedge x.\ P\ x \Longrightarrow Q\ x, P\ t \vdash Q\ t}\,(\Longrightarrow\text{-}\mathit{elim})}{\bigwedge x.\ P\ x \Longrightarrow Q\ x \vdash P\ t \Longrightarrow Q\ t}\,(\Longrightarrow\text{-}\mathit{intro})}{\vdash [\![\bigwedge x.\ P\ x \Longrightarrow Q\ x;\ P\ t]\!] \Longrightarrow Q\ t}\,(\Longrightarrow\text{-}\mathit{intro})$$

*For how the functions* `assume`, `forall_elim` *etc work see [Impl. Man., Sec. 2.3]. The basic functions for theorems are defined in* `Pure/thm.ML`. **Read More**

## 2.8 Tactical Reasoning

The goal-oriented tactical style reasoning of the ML level is similar to the `apply`-style at the user level, i.e. the reasoning is centred around a *goal*, which is modified in a sequence of proof steps until it is solved.

A goal (or goal state) is a special `thm`, which by convention is an implication of the form:

$$A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow \#(C)$$

where $C$ is the goal to be proved and the $A_i$ are the open subgoals. Since the goal $C$ can potentially be an implication, there is a `#` wrapped around it, which prevents that premises

[1]Note that `|>` is reverse application. This combinator, and several variants are defined in `Pure/General/basics.ML`.

are misinterpreted as open subgoals. The protection $\# :: prop \Rightarrow prop$ is just the identity function and used as a syntactic marker.

(FIXME: maybe show how this is printed on the screen)

*For more on goals see [Impl. Man., Sec. 3.1].* **Read More**

Tactics are functions that map a goal state to a (lazy) sequence of successor states, hence the type of a tactic is

```
thm -> thm Seq.seq
```

*See* `Pure/General/seq.ML` *for the implementation of lazy sequences. However in day-to-day Isabelle programming, one rarely constructs sequences explicitly, but uses the predefined tactic combinators (tacticals) instead (see* `Pure/tctical.ML`*). (FIXME: Pointer to the old reference manual)* **Read More**

While tactics can operate on the subgoals (the $A_i$ above), they are expected to leave the conclusion $C$ intact, with the exception of possibly instantiating schematic variables.

To see how tactics work, let us transcribe a simple `apply`-style proof from the tutorial [4] into ML:

```
lemma disj_swap: "P ∨ Q ⟹ Q ∨ P"
apply (erule disjE)
 apply (rule disjI2)
 apply assumption
apply (rule disjI1)
apply assumption
done
```

To start the proof, the function `Goal.prove ctxt xs As C tac` sets up a goal state for proving the goal $C$ under the assumptions $As$ (empty in the proof at hand) with the variables $xs$ that will be generalised once the goal is proved. The $tac$ is the tactic which proves the goal and which can make use of the local assumptions.

```
ML {*
let
  val ctxt = @{context}
  val goal = @{prop "P ∨ Q ⟹ Q ∨ P"}
in
  Goal.prove ctxt ["P", "Q"] [] goal (fn _ =>
    eresolve_tac [disjE] 1
    THEN resolve_tac [disjI2] 1
    THEN assume_tac 1
    THEN resolve_tac [disjI1] 1
    THEN assume_tac 1)
end
*}
```

*To learn more about the function* `Goal.prove` *see [Impl. Man., Sec. 4.3].* **Read More**

An alternative way to transcribe this proof is as follows

```
ML {*
let
  val ctxt = @{context}
  val goal = @{prop "P ∨ Q ⟹ Q ∨ P"}
```

```
in
  Goal.prove ctxt ["P", "Q"] [] goal (fn _ =>
    (eresolve_tac [disjE]
    THEN' resolve_tac [disjI2]
    THEN' assume_tac
    THEN' resolve_tac [disjI1]
    THEN' assume_tac) 1)
end
*}
```

(FIXME: are there any advantages/disadvantages about this way?)

## 2.9 Storing Theorems

## 2.10 Theorem Attributes

# Chapter 3

# Parsing

Lots of Standard ML code is given in this document, for various reasons, including:

- direct quotation of code found in the Isabelle source files, or simplified versions of such code
- identifiers found in the Isabelle source code, with their types (or specialisations of their types)
- code examples, which can be run by the reader, to help illustrate the behaviour of functions found in the Isabelle source code
- ancillary functions, not from the Isabelle source code, which enable the reader to run relevant code examples
- type abbreviations, which help explain the uses of certain functions

## 3.1 Parsing Isar input

The typical parsing function has the type `'src -> 'res * 'src`, with input of type `'src`, returning a result of type `'res`, which is (or is derived from) the first part of the input, and also returning the remainder of the input. (In the common case, when it is clear what the "remainder of the input" means, we will just say that the functions "returns" the value of type `'res`). An exception is raised if an appropriate value cannot be produced from the input. A range of exceptions can be used to identify different reasons for the failure of a parse.

This contrasts the standard parsing function in Standard ML, which is of type `type ('res, 'src) reader = 'src -> ('res * 'src) option`; (for example, `List.getItem` and `Substring.getc`). However, much of the discussion at FIX file:/home/jeremy/html/ml/SMLBasis/string-cvt.html is relevant.

Naturally one may convert between the two different sorts of parsing functions as follows:

```
open StringCvt ;
type ('res, 'src) ex_reader = 'src -> 'res * 'src
(* ex_reader : ('res, 'src) reader -> ('res, 'src) ex_reader *)
fun ex_reader rdr src = Option.valOf (rdr src) ;
```

```
(* reader : ('res, 'src) ex_reader -> ('res, 'src) reader *)
fun reader exrdr src = SOME (exrdr src) handle _ => NONE ;
```

## 3.2 The Scan structure

The source file is src/General/scan.ML. This structure provides functions for using and combining parsing functions of the type 'src -> 'res * 'src. Three exceptions are used:

```
exception MORE of string option;  (*need more input (prompt)*)
exception FAIL of string option;  (*try alternatives (reason of failure)*)
exception ABORT of string;        (*dead end*)
```

Many functions in this structure (generally those with names composed of symbols) are declared as infix.

Some functions from that structure are

```
|-- : ('src -> 'res1 * 'src') * ('src' -> 'res2 * 'src'') ->
'src -> 'res2 * 'src''
--| : ('src -> 'res1 * 'src') * ('src' -> 'res2 * 'src'') ->
'src -> 'res1 * 'src''
-- : ('src -> 'res1 * 'src') * ('src' -> 'res2 * 'src'') ->
'src -> ('res1 * 'res2) * 'src''
^^ : ('src -> string * 'src') * ('src' -> string * 'src'') ->
'src -> string * 'src''
```

These functions parse a result off the input source twice.

|-- and --| return the first result and the second result, respectively.

-- returns both.

^^ returns the result of concatenating the two results (which must be strings).

Note how, although the types 'src, 'src' and 'src'' will normally be the same, the types as shown help suggest the behaviour of the functions.

```
:-- : ('src -> 'res1 * 'src') * ('res1 -> 'src' -> 'res2 * 'src'') ->
'src -> ('res1 * 'res2) * 'src''
:|-- : ('src -> 'res1 * 'src') * ('res1 -> 'src' -> 'res2 * 'src'') ->
'src -> 'res2 * 'src''
```

These are similar to |-- and --|, except that the second parsing function can depend on the result of the first.

```
>> : ('src -> 'res1 * 'src') * ('res1 -> 'res2) -> 'src -> 'res2 * 'src'
|| : ('src -> 'res_src) * ('src -> 'res_src) -> 'src -> 'res_src
```

`p >> f` applies a function `f` to the result of a parse.

`||` tries a second parsing function if the first one fails by raising an exception of the form `FAIL _`.

```
succeed : 'res -> ('src -> 'res * 'src) ;
fail : ('src -> 'res_src) ;
!! : ('src * string option -> string) ->
('src -> 'res_src) -> ('src -> 'res_src) ;
```

`succeed r` returns `r`, with the input unchanged. `fail` always fails, raising exception `FAIL NONE`. `!! f` only affects the failure mode, turning a failure that raises `FAIL _` into a failure that raises `ABORT ...`. This is used to prevent recovery from the failure — thus, in `!! parse1 || parse2`, if `parse1` fails, it won't recover by trying `parse2`.

```
one : ('si -> bool) -> ('si list -> 'si * 'si list) ;
some : ('si -> 'res option) -> ('si list -> 'res * 'si list) ;
```

These require the input to be a list of items: they fail, raising `MORE NONE` if the list is empty. On other failures they raise `FAIL NONE`

`one p` takes the first item from the list if it satisfies `p`, otherwise fails.

`some f` takes the first item from the list and applies `f` to it, failing if this returns `NONE`.

```
many : ('si -> bool) -> 'si list -> 'si list * 'si list ;
```

`many p` takes items from the input until it encounters one which does not satisfy `p`. If it reaches the end of the input it fails, raising `MORE NONE`.

`many1` (with the same type) fails if the first item does not satisfy `p`.

```
option : ('src -> 'res * 'src) -> ('src -> 'res option * 'src)
optional : ('src -> 'res * 'src) -> 'res -> ('src -> 'res * 'src)
```

`option`: where the parser `f` succeeds with result `r` or raises `FAIL _`, `option f` gives the result `SOME r` or `NONE`.

`optional`: if parser `f` fails by raising `FAIL _`, `optional f default` provides the result `default`.

```
repeat : ('src -> 'res * 'src) -> 'src -> 'res list * 'src
repeat1 : ('src -> 'res * 'src) -> 'src -> 'res list * 'src
bulk : ('src -> 'res * 'src) -> 'src -> 'res list * 'src
```

`repeat f` repeatedly parses an item off the remaining input until `f` fails with `FAIL _`

`repeat1` is as for `repeat`, but requires at least one successful parse.

```
lift : ('src -> 'res * 'src) -> ('ex * 'src -> 'res * ('ex * 'src))
```

`lift` changes the source type of a parser by putting in an extra component `'ex`, which is ignored in the parsing.

The `Scan` structure also provides the type `lexicon`, HOW DO THEY WORK ?? TO BE COMPLETED

```
dest_lexicon: lexicon -> string list ;
make_lexicon: string list list -> lexicon ;
empty_lexicon: lexicon ;
extend_lexicon: string list list -> lexicon -> lexicon ;
merge_lexicons: lexicon -> lexicon -> lexicon ;
is_literal: lexicon -> string list -> bool ;
literal: lexicon -> string list -> string list * string list ;
```

Two lexicons, for the commands and keywords, are stored and can be retrieved by:

```
val (command_lexicon, keyword_lexicon) = OuterSyntax.get_lexicons () ;
val commands = Scan.dest_lexicon command_lexicon ;
val keywords = Scan.dest_lexicon keyword_lexicon ;
```

## 3.3 The `OuterLex` structure

The source file is *src/Pure/Isar/outer_lex.ML*. In some other source files its name is abbreviated:

```
structure T = OuterLex;
```

This structure defines the type `token`. (The types `OuterLex.token`, `OuterParse.token` and `SpecParse.token` are all the same).

Input text is split up into tokens, and the input source type for many parsing functions is `token list`.

The datatype definition (which is not published in the signature) is

```
datatype token = Token of Position.T * (token_kind * string);
```

but here are some runnable examples for viewing tokens:

FIXME

*begin{verbatim} type token = T.token ; val toks : token list = OuterSyntax.scan ‘‘theory,imports;be x.y.z apply ?v1 ?'a 'a -- || 44 simp (* xx *) { * fff * }’’ ; print_depth 20 ; List.map T.text_of toks ; val proper_toks = List.filter T.is_proper toks ; List.map T.kind_of proper_toks ; List.map T.unparse proper_toks ; List.map T.val_of proper_toks ; end{verbatim}*

The function `is_proper : token -> bool` identifies tokens which are not white space or comments: many parsing functions assume require spaces or comments to have been filtered out.

There is a special end-of-file token:

```
val (tok_eof : token, is_eof : token -> bool) = T.stopper ;
(* end of file token *)
```

## 3.4 The `OuterParse` structure

The source file is `src/Pure/Isar/outer_parse.ML`. In some other source files its name is abbreviated:

```
structure P = OuterParse;
```

Here the parsers use `token list` as the input source type.

Some of the parsers simply select the first token, provided that it is of the right kind (as returned by `T.kind_of`): these are `command, keyword, short_ident, long_ident, sym_ident, term_var, type_ident, type_var, number, string, alt_string, verbatim, sync, eof` Others select the first token, provided that it is one of several kinds, (eg, `name, xname, text, typ`).

```
type 'a tlp = token list -> 'a * token list ; (* token list parser *)
$$$ : string -> string tlp
nat : int tlp ;
maybe : 'a tlp -> 'a option tlp ;
```

`$$$ s` returns the first token, if it equals `s` *and* `s` is a keyword.

`nat` returns the first token, if it is a number, and evaluates it.

`maybe`: if `p` returns `r`, then `maybe p` returns `SOME r` ; if the first token is an underscore, it returns `NONE`.

A few examples:

```
P.list : 'a tlp -> 'a list tlp ; (* likewise P.list1 *)
P.and_list : 'a tlp -> 'a list tlp ; (* likewise P.and_list1 *)
val toks : token list = OuterSyntax.scan "44 ,_, 66,77" ;
val proper_toks = List.filter T.is_proper toks ;
P.list P.nat toks ; (* OK, doesn't recognize white space *)
P.list P.nat proper_toks ; (* fails, doesn't recognize what follows ',' *)
P.list (P.maybe P.nat) proper_toks ; (* fails, end of input *)
P.list (P.maybe P.nat) (proper_toks @ [tok_eof]) ; (* OK *)
val toks : token list = OuterSyntax.scan "44 and 55 and 66 and 77" ;
P.and_list P.nat (List.filter T.is_proper toks @ [tok_eof]) ; (* ??? *)
```

The following code helps run examples:

```
fun parse_str tlp str =
let val toks : token list = OuterSyntax.scan str ;
val proper_toks = List.filter T.is_proper toks @ [tok_eof] ;
val (res, rem_toks) = tlp proper_toks ;
val rem_str = String.concat
(Library.separate " " (List.map T.unparse rem_toks)) ;
in (res, rem_str) end ;
```

Some examples from `src/Pure/Isar/outer_parse.ML`

```
val type_args =
type_ident >> Library.single ||
$$$ "(" |-- !!! (list1 type_ident --| $$$ ")") ||
Scan.succeed [];
```

There are three ways parsing a list of type arguments can succeed. The first line reads a single type argument, and turns it into a singleton list. The second line reads "(", and then the remainder, ignoring the "(" ; the remainder consists of a list of type identifiers (at least one), and then a ")" which is also ignored. The `!!!` ensures that if the parsing proceeds this far and then fails, it won't try the third line (see the description of `Scan.!!`). The third line consumes no input and returns the empty list.

```
fun triple2 (x, (y, z)) = (x, y, z);
val arity = xname -- ($$$ "::" |-- !!! (
Scan.optional ($$$ "(" |-- !!! (list1 sort --| $$$ ")")) []
-- sort)) >> triple2;
```

The parser `arity` reads a typename $t$, then "`::`" (which is ignored), then optionally a list $ss$ of sorts and then another sort $s$. The result $(t, (ss, s))$ is transformed by `triple2` to $(t, ss, s)$. The second line reads the optional list of sorts: it reads first "`(`" and last "`)`", which are both ignored, and between them a comma-separated list of sorts. If this list is absent, the default `[]` provides the list of sorts.

```
parse_str P.type_args "('a, 'b) ntyp" ;
parse_str P.type_args "'a ntyp" ;
parse_str P.type_args "ntyp" ;
parse_str P.arity "ty :: tycl" ;
parse_str P.arity "ty :: (tycl1, tycl2) tycl" ;
```

## 3.5 The SpecParse structure

The source file is `src/Pure/Isar/spec_parse.ML`. This structure contains token list parsers for more complicated values. For example,

```
open SpecParse ;
attrib : Attrib.src tok_rdr ;
attribs : Attrib.src list tok_rdr ;
opt_attribs : Attrib.src list tok_rdr ;
xthm : (thmref * Attrib.src list) tok_rdr ;
xthms1 : (thmref * Attrib.src list) list tok_rdr ;

parse_str attrib "simp" ;
parse_str opt_attribs "hello" ;
val (ass, "") = parse_str attribs "[standard, xxxx, simp, intro, OF sym]" ;
map Args.dest_src ass ;
val (asrc, "") = parse_str attrib "THEN trans [THEN sym]" ;

parse_str xthm "mythm [attr]" ;
parse_str xthms1 "thm1 [attr] thms2" ;
```

As you can see, attributes are described using types of the `Args` structure, described below.

## 3.6 The Args structure

The source file is `src/Pure/Isar/args.ML`. The primary type of this structure is the `src` datatype; the single constructors not published in the signature, but `Args.src` and `Args.dest_src` are in fact the constructor and destructor functions. Note that the types `Attrib.src` and `Method.src` are in fact `Args.src`.

```
src : (string * Args.T list) * Position.T -> Args.src ;
dest_src : Args.src -> (string * Args.T list) * Position.T ;
Args.pretty_src : Proof.context -> Args.src -> Pretty.T ;
fun pr_src ctxt src = Pretty.string_of (Args.pretty_src ctxt src) ;

val thy = ML_Context.the_context () ;
val ctxt = ProofContext.init thy ;
map (pr_src ctxt) ass ;
```

So an `Args.src` consists of the first word, then a list of further "arguments", of type `Args.T`, with information about position in the input.

```
(* how an Args.src is parsed *)
P.position : 'a tlp -> ('a * Position.T) tlp ;
P.arguments : Args.T list tlp ;
```

```
val parse_src : Args.src tlp =
P.position (P.xname -- P.arguments) >> Args.src ;


val ((first_word, args), pos) = Args.dest_src asrc ;
map Args.string_of args ;
```

The Args structure contains more parsers and parser transformers for which the input source type is `Args.T list`. For example,

```
type 'a atlp = Args.T list -> 'a * Args.T list ;
open Args ;
nat : int atlp ; (* also Args.int *)
thm_sel : PureThy.interval list atlp ;
list : 'a atlp -> 'a list atlp ;
attribs : (string -> string) -> Args.src list atlp ;
opt_attribs : (string -> string) -> Args.src list atlp ;

(* parse_atl_str : 'a atlp -> (string -> 'a * string) ;
given an Args.T list parser, to get a string parser *)
fun parse_atl_str atlp str =
let val (ats, rem_str) = parse_str P.arguments str ;
val (res, rem_ats) = atlp ats ;
in (res, String.concat (Library.separate " "
(List.map Args.string_of rem_ats @ [rem_str]))) end ;

parse_atl_str Args.int "-1-," ;
parse_atl_str (Scan.option Args.int) "x1-," ;
parse_atl_str Args.thm_sel "(1-,4,13-22)" ;

val (ats as atsrc :: _, "") = parse_atl_str (Args.attribs I)
"[THEN trans [THEN sym], simp, OF sym]" ;
```

From here, an attribute is interpreted using `Attrib.attribute`.

`Args` has a large number of functions which parse an `Args.src` and also refer to a generic context. Note the use of `Scan.lift` for this. (as does `Attrib` - RETHINK THIS)

(`Args.syntax` shown below has type specialised)

```
type ('res, 'src) parse_fn = 'src -> 'res * 'src ;
type 'a cgatlp = ('a, Context.generic * Args.T list) parse_fn ;
Scan.lift : 'a atlp -> 'a cgatlp ;
term : term cgatlp ;
typ : typ cgatlp ;

Args.syntax : string -> 'res cgatlp -> src -> ('res, Context.generic) parse_fn ;
Attrib.thm : thm cgatlp ;
```

```
Attrib.thms : thm list cgatlp ;
Attrib.multi_thm : thm list cgatlp ;

(* parse_cgatl_str : 'a cgatlp -> (string -> 'a * string) ;
given a (Context.generic * Args.T list) parser, to get a string parser *)
fun parse_cgatl_str cgatlp str =
let
  (* use the current generic context *)
  val generic = Context.Theory thy ;
  val (ats, rem_str) = parse_str P.arguments str ;
  (* ignore any change to the generic context *)
  val (res, (_, rem_ats)) = cgatlp (generic, ats) ;
in (res, String.concat (Library.separate " "
    (List.map Args.string_of rem_ats @ [rem_str]))) end ;
```

### 3.7 Attributes, and the `Attrib` structure

The type `attribute` is declared in `src/Pure/thm.ML`. The source file for the `Attrib` structure is `src/Pure/Isar/attrib.ML`. Most attributes use a theorem to change a generic context (for example, by declaring that the theorem should be used, by default, in simplification), or change a theorem (which most often involves referring to the current theory). The functions `Thm.rule_attribute` and `Thm.declaration_attribute` create attributes of these kinds.

```
type attribute = Context.generic * thm -> Context.generic * thm;
type 'a trf = 'a -> 'a ; (* transformer of a given type *)
Thm.rule_attribute  : (Context.generic -> thm -> thm) -> attribute ;
Thm.declaration_attribute : (thm -> Context.generic trf) -> attribute ;

Attrib.print_attributes : theory -> unit ;
Attrib.pretty_attribs : Proof.context -> src list -> Pretty.T list ;

List.app Pretty.writeln (Attrib.pretty_attribs ctxt ass) ;
```

An attribute is stored in a theory as indicated by:

```
Attrib.add_attributes :
(bstring * (src -> attribute) * string) list -> theory trf ;
(*
Attrib.add_attributes [("THEN", THEN_att, "resolution with rule")] ;
*)
```

where the first and third arguments are name and description of the attribute, and the second is a function which parses the attribute input text (including the attribute name, which has necessarily already been parsed). Here, `THEN_att` is a function declared in the code for the structure

Attrib, but not published in its signature. The source file src/Pure/Isar/attrib.ML shows the use of Attrib.add_attributes to add a number of attributes.

```
FullAttrib.THEN_att : src -> attribute ;
FullAttrib.THEN_att atsrc (generic, ML_Context.thm "sym") ;
FullAttrib.THEN_att atsrc (generic, ML_Context.thm "all_comm") ;
```

```
Attrib.syntax : attribute cgatlp -> src -> attribute ;
Attrib.no_args : attribute -> src -> attribute ;
```

When this is called as syntax scan src (gc, th) the generic context gc is used (and potentially changed to gc') by scan in parsing to obtain an attribute attr which would then be applied to (gc', th). The source for parsing the attribute is the arguments part of src, which must all be consumed by the parse.

For example, for Attrib.no_args attr src, the attribute parser simply returns attr, requiring that the arguments part of src must be empty.

Some examples from src/Pure/Isar/attrib.ML, modified:

```
fun rot_att_n n (gc, th) = (gc, rotate_prems n th) ;
rot_att_n : int -> attribute ;
val rot_arg = Scan.lift (Scan.optional Args.int 1 : int atlp) : int cgatlp ;
val rotated_att : src -> attribute =
Attrib.syntax (rot_arg >> rot_att_n : attribute cgatlp) ;

val THEN_arg : int cgatlp = Scan.lift
(Scan.optional (Args.bracks Args.nat : int atlp) 1 : int atlp) ;

Attrib.thm : thm cgatlp ;

THEN_arg -- Attrib.thm : (int * thm) cgatlp ;

fun THEN_att_n (n, tht) (gc, th) = (gc, th RSN (n, tht)) ;
THEN_att_n : int * thm -> attribute ;

val THEN_att : src -> attribute = Attrib.syntax
(THEN_arg -- Attrib.thm >> THEN_att_n : attribute cgatlp);
```

The functions I've called rot_arg and THEN_arg read an optional argument, which for rotated is an integer, and for THEN is a natural enclosed in square brackets; the default, if the argument is absent, is 1 in each case. Functions rot_att_n and THEN_att_n turn these into attributes, where THEN_att_n also requires a theorem, which is parsed by Attrib.thm. Infix operators -- and >> are in the structure Scan.

## 3.8 Methods, and the `Method` structure

The source file is `src/Pure/Isar/method.ML`. The type `method` is defined by the datatype declaration

```
(* datatype method = Meth of thm list -> cases_tactic; *)
RuleCases.NO_CASES : tactic -> cases_tactic ;
```

In fact `RAW_METHOD_CASES` (below) is exactly the constructor `Meth`. A `cases_tactic` is an elaborated version of a tactic. `NO_CASES tac` is a `cases_tactic` which consists of a `cases_tactic` without any further case information. For further details see the description of structure `RuleCases` below. The list of theorems to be passed to a method consists of the current *facts* in the proof.

```
RAW_METHOD : (thm list -> tactic) -> method ;
METHOD : (thm list -> tactic) -> method ;

SIMPLE_METHOD : tactic -> method ;
SIMPLE_METHOD' : (int -> tactic) -> method ;
SIMPLE_METHOD'' : ((int -> tactic) -> tactic) -> (int -> tactic) -> method ;

RAW_METHOD_CASES : (thm list -> cases_tactic) -> method ;
METHOD_CASES : (thm list -> cases_tactic) -> method ;
```

A method is, in its simplest form, a tactic; applying the method is to apply the tactic to the current goal state.

Applying `RAW_METHOD tacf` creates a tactic by applying `tacf` to the current facts, and applying that tactic to the goal state.

`METHOD` is similar but also first applies `Goal.conjunction_tac` to all subgoals.

`SIMPLE_METHOD tac` inserts the facts into all subgoals and then applies `tacf`.

`SIMPLE_METHOD' tacf` inserts the facts and then applies `tacf` to subgoal 1.

`SIMPLE_METHOD'' quant tacf` does this for subgoal(s) selected by `quant`, which may be, for example, `ALLGOALS` (all subgoals), `TRYALL` (try all subgoals, failure is OK), `FIRSTGOAL` (try subgoals until it succeeds once), `(fn tacf => tacf 4)` (subgoal 4), etc (see the `Tactical` structure, FIXME)

A method is stored in a theory as indicated by:

```
Method.add_method :
(bstring * (src -> Proof.context -> method) * string) -> theory trf ;
( *
* )
```

where the first and third arguments are name and description of the method, and the second is a function which parses the method input text (including the method name, which has necessarily already been parsed).

Here, `xxx` is a function declared in the code for the structure `Method`, but not published in its signature. The source file `src/Pure/Isar/method.ML` shows the use of `Method.add_method` to add a number of methods.

# Chapter 4

# How to write a definitional package

## 4.1 Introduction

*"My thesis is that programming is not at the bottom of the intellectual pyramid, but at the top. It's creative design of the highest order. It isn't monkey or donkey work; rather, as Edsger Dijkstra famously claimed, it's amongst the hardest intellectual tasks ever attempted."*

Richard Bornat, In defence of programming

Higher order logic, as implemented in Isabelle/HOL, is based on just a few primitive constants, like equality, implication, and the description operator, whose properties are described as axioms. All other concepts, such as inductive predicates, datatypes, or recursive functions are *defined* using these constants, and the desired properties, for example induction theorems, or recursion equations are *derived* from the definitions by a *formal proof*. Since it would be very tedious for the average user to define complex inductive predicates or datatypes "by hand" just using the primitive operators of higher order logic, Isabelle/HOL already contains a number of *packages* automating such tedious work. Thanks to those packages, the user can give a high-level specification, like a list of introduction rules or constructors, and the package then does all the low-level definitions and proofs behind the scenes. The packages are written in Standard ML, the implementation language of Isabelle, and can be invoked by the user from within theory documents written in the Isabelle/Isar language via specific commands. Most of the time, when using Isabelle for applications, users do not have to worry about the inner workings of packages, since they can just use the packages that are already part of the Isabelle distribution. However, when developing a general theory that is intended to be applied by other users, one may need to write a new package from scratch. Recent examples of such packages include the verification environment for sequential imperative programs by Schirmer [7], the package for defining general recursive functions by Krauss [2], as well as the nominal datatype package by Berghofer and Urban [9].

The scientific value of implementing a package should not be underestimated: it is often more than just the automation of long-established scientific results. Of course, a carefully-developed theory on paper is indispensable as a basis. However, without an implementation, such a theory will only be of very limited practical use, since only an implementation enables other users to apply the theory on a larger scale without too much effort. Moreover, implementing a package is a bit like formalizing a paper proof in a theorem prover. In the literature, there are many examples of paper proofs that turned out to be incomplete or even faulty, and doing a formalization is a good way of uncovering such errors and ensuring that a proof is really correct. The same applies to the theory underlying definitional packages. For example, the general form of some complicated induction rules for nominal datatypes turned out to be quite hard to get right on the first try, so an implementation is an excellent way to find out whether the rules really work in practice.

Writing a package is a particularly difficult task for users that are new to Isabelle, since its programming interface consists of thousands of functions. Rather than just listing all those functions, we give a step-by-step tutorial for writing a package, using an example that is still simple enough to be easily understandable, but at the same time sufficiently complex to demonstrate enough of Isabelle's interesting features. As a running example, we have chosen a rather simple package for defining inductive predicates. To keep things simple, we will not use the general Knaster-Tarski fixpoint theorem on complete lattices, which forms the basis of Isabelle's standard inductive definition package originally due to Paulson [6]. Instead, we will use a simpler *impredicative* (i.e. involving quantification on predicate variables) encoding of inductive predicates suggested by Melham [3]. Due to its simplicity, this package will necessarily have a reduced functionality. It does neither support introduction rules involving arbitrary monotone operators, nor does it prove case analysis (or inversion) rules. Moreover, it only proves a weaker form of the rule induction theorem.

Reading this article does not require any prior knowledge of Isabelle's programming interface. However, we assume the reader to already be familiar with writing proofs in Isabelle/HOL using the Isar language. For further information on this topic, consult the book by Nipkow, Paulson, and Wenzel [4]. Moreover, in order to understand the pieces of code given in this tutorial, some familiarity with the basic concepts of the Standard ML programming language, as described for example in the textbook by Paulson [5], is required as well.

The rest of this article is structured as follows. In §4.2, we will illustrate the "manual" definition of inductive predicates using some examples. Starting from these examples, we will describe in §4.3 how the construction works in general. The following sections are then dedicated to the implementation of a package that carries out the construction of such inductive predicates. First of all, a parser for a high-level notation for specifying inductive predicates via a list of introduction rules is developed in §4.4. Having parsed the specification, a suitable primitive definition must be added to the theory, which will be explained in §**??**. Finally, §**??** will focus on methods for proving introduction and induction rules from the definitions introduced in §**??**.

## 4.2 Examples of inductive definitions

In this section, we will give three examples showing how to define inductive predicates by hand and prove characteristic properties such as introduction rules and an induction rule. From these examples, we will then figure out a general method for defining inductive predicates, which will be described in §4.3. This description will serve as a basis for the actual implementation to be developed in §4.4 – §**??**. It should be noted that our aim in this section is not to write proofs that are as beautiful as possible, but as close as possible to the ML code producing the proofs that we will develop later. As a first example, we consider the *transitive* *closure* `trcl R` of a relation `R`. It is characterized by the following two introduction rules

```
trcl R x x
R x y ⟹ trcl R y z ⟹ trcl R x z
```

Note that the `trcl` predicate has two different kinds of parameters: the first parameter `R` stays *fixed* throughout the definition, whereas the second and third parameter changes in the "recursive call". Since an inductively defined predicate is the *least* predicate closed under a collection of introduction rules, we define the predicate `trcl R x y` in such a way that it holds if and only if `P x y` holds for every predicate `P` closed under the above rules. This gives rise to a definition containing a universal quantifier over the predicate `P`:

```
definition "trcl ≡ λR x y.
  ∀P. (∀x. P x x) ⟶ (∀x y z. R x y ⟶ P y z ⟶ P x z) ⟶ P x y"
```

Since the predicate `trcl R x y` yields an element of the type of object level truth values `bool`, the meta-level implications $\Longrightarrow$ in the above introduction rules have to be converted to object-level implications $\longrightarrow$. Moreover, we use object-level universal quantifiers $\forall$ rather than meta-level universal quantifiers $\bigwedge$ for quantifying over the variable parameters of the introduction rules. Isabelle already offers some infrastructure for converting between meta-level and object-level connectives, which we will use later on.

With this definition of the transitive closure, the proof of the (weak) induction theorem is almost immediate. It suffices to convert all the meta-level connectives in the induction rule to object-level connectives using the `atomize` proof method, expand the definition of `trcl`, eliminate the universal quantifier contained in it, and then solve the goal by assumption.

```
lemma trcl_induct:
  assumes trcl: "trcl R x y"
  shows "(⋀x. P x x) ⟹ (⋀x y z. R x y ⟹ P y z ⟹ P x z) ⟹ P x y"
  apply (atomize (full))
  apply (cut_tac trcl)
  apply (unfold trcl_def)
  apply (drule spec [where x=P])
  apply assumption
  done
```

The above induction rule is *weak* in the sense that the induction step may only be proved using the assumptions `R x y` and `P y z`, but not using the additional assumption `trcl R y z`. A stronger induction rule containing this additional assumption can be derived from the weaker one with the help of the introduction rules for `trcl`.

We now turn to the proofs of the introduction rules, which are slightly more complicated. In order to prove the first introduction rule, we again unfold the definition and then apply the introdution rules for $\forall$ and $\longrightarrow$ as often as possible. We then end up in a proof state of the following form:

```
1. ⋀P. ⟦∀x. P x x; ∀x y z. R x y ⟶ P y z ⟶ P x z⟧ ⟹ P x x
```

The two assumptions correspond to the introduction rules, where `trcl R` has been replaced by `P`. Thus, all we have to do is to eliminate the universal quantifier in front of the first assumption, and then solve the goal by assumption:

```
lemma trcl_base: "trcl R x x"
```

```
  apply (unfold trcl_def)
  apply (rule allI impI)+
  apply (drule spec)
  apply assumption
  done
```

Since the second introduction rule has premises, its proof is not as easy as the previous one. After unfolding the definitions and applying the introduction rules for ∀ and ⟶, we get the proof state

```
 1. ⋀P. ⟦R x y;
         ∀P. (∀x. P x x) ⟶ (∀x y z. R x y ⟶ P y z ⟶ P x z) ⟶ P y z;
         ∀x. P x x; ∀x y z. R x y ⟶ P y z ⟶ P x z⟧
        ⟹ P x z
```

The third and fourth assumption corresponds to the first and second introduction rule, respectively, whereas the first and second assumption corresponds to the premises of the introduction rule. Since we want to prove the second introduction rule, we apply the fourth assumption to the goal `P x z`. In order for the assumption to be applicable, we have to eliminate the universal quantifiers and turn the object-level implications into meta-level ones. This can be accomplished using the `rule_format` attribute. Applying the assumption produces two new subgoals, which can be solved using the first and second assumption. The second assumption again involves a quantifier and implications that have to be eliminated before it can be applied. To avoid problems with higher order unification, it is advisable to provide an instantiation for the universally quantified predicate variable in the assumption.

```
lemma trcl_step: "R x y ⟹ trcl R y z ⟹ trcl R x z"
  apply (unfold trcl_def)
  apply (rule allI impI)+
  proof -
    case goal1
    show ?case
      apply (rule goal1(4) [rule_format])
      apply (rule goal1(1))
      apply (rule goal1(2) [THEN spec [where x=P], THEN mp, THEN mp,
        OF goal1(3-4)])
      done
  qed
```

This method of defining inductive predicates easily generalizes to mutually inductive predicates, like the predicates `even` and `odd` characterized by the following introduction rules:

```
even 0
odd m ⟹ even (Suc m)
even m ⟹ odd (Suc m)
```

Since the predicates are mutually inductive, each of the definitions contain two quantifiers over the predicates `P` and `Q`.

```
definition "even ≡ λn.
  ∀P Q. P 0 ⟶ (∀m. Q m ⟶ P (Suc m)) ⟶ (∀m. P m ⟶ Q (Suc m)) ⟶ P n"

definition "odd ≡ λn.
  ∀P Q. P 0 ⟶ (∀m. Q m ⟶ P (Suc m)) ⟶ (∀m. P m ⟶ Q (Suc m)) ⟶ Q n"
```

For proving the induction rule, we use exactly the same technique as in the transitive closure example:

```
lemma even_induct:
  assumes even: "even n"
  shows "P 0 ⟹ (⋀m. Q m ⟹ P (Suc m)) ⟹ (⋀m. P m ⟹ Q (Suc m)) ⟹ P n"
  apply (atomize (full))
  apply (cut_tac even)
  apply (unfold even_def)
  apply (drule spec [where x=P])
  apply (drule spec [where x=Q])
  apply assumption
  done
```

A similar induction rule having *Q n* as a conclusion can be proved for the *odd* predicate. The proofs of the introduction rules are also very similar to the ones in the previous example. We only show the proof of the second introduction rule, since it is almost the same as the one for the third introduction rule, and the proof of the first rule is trivial.

```
lemma evenS: "odd m ⟹ even (Suc m)"
  apply (unfold odd_def even_def)
  apply (rule allI impI)+
  proof -
    case goal1
    show ?case
      apply (rule goal1(3) [rule_format])
      apply (rule goal1(1) [THEN spec [where x=P], THEN spec [where x=Q],
        THEN mp, THEN mp, THEN mp, OF goal1(2-4)])
      done
  qed
```

As a final example, we will consider the definition of the accessible part of a relation *R* characterized by the introduction rule

$$(\bigwedge y.\ R\ y\ x \Longrightarrow accpart\ R\ y) \Longrightarrow accpart\ R\ x$$

whose premise involves a universal quantifier and an implication. The definition of *accpart* is as follows:

```
definition "accpart ≡ λR x. ∀P. (∀x. (∀y. R y x ⟶ P y) ⟶ P x) ⟶ P x"
```

The proof of the induction theorem is again straightforward:

```
lemma accpart_induct:
  assumes acc: "accpart R x"
  shows "(⋀x. (⋀y. R y x ⟹ P y) ⟹ P x) ⟹ P x"
  apply (atomize (full))
  apply (cut_tac acc)
  apply (unfold accpart_def)
  apply (drule spec [where x=P])
  apply assumption
  done
```

Proving the introduction rule is a little more complicated, due to the quantifier and the implication in the premise. We first convert the meta-level universal quantifier and implication to their object-level counterparts. Unfolding the definition of *accpart* and applying the introduction rules for $\forall$ and $\longrightarrow$ yields the following proof state:

```
1. ⋀P. ⟦⋀y. R y x ⟹ ∀P. (∀x. (∀y. R y x ⟶ P y) ⟶ P x) ⟶ P y;
       ∀x. (∀y. R y x ⟶ P y) ⟶ P x⟧
     ⟹ P x
```

Applying the second assumption produces a proof state with the new local assumption `R y x`, which will then be used to solve the goal `P y` using the first assumption.

```
lemma accpartI: "(⋀y. R y x ⟹ accpart R y) ⟹ accpart R x"
  apply (unfold accpart_def)
  apply (rule allI impI)+
  proof -
    case goal1
    note goal1' = this
    show ?case
      apply (rule goal1'(2) [rule_format])
      proof -
        case goal1
        show ?case
          apply (rule goal1'(1) [OF goal1, THEN spec [where x=P],
            THEN mp, OF goal1'(2)])
          done
      qed
    qed
```

## 4.3 The general construction principle

Before we start with the implementation, it is useful to describe the general form of inductive definitions that our package should accept. We closely follow the notation for inductive definitions introduced by Schwichtenberg [8] for the Minlog system. Let $R_1, \ldots, R_n$ be mutually inductive predicates and $\vec{p}$ be parameters. Then the introduction rules for $R_1, \ldots, R_n$ may have the form

$$\bigwedge \vec{x}_i.\ \vec{A}_i \Longrightarrow \left(\bigwedge \vec{y}_{ij}.\ \vec{B}_{ij} \Longrightarrow R_{k_{ij}}\ \vec{p}\ \vec{s}_{ij}\right)_{j=1,\ldots,m_i} \Longrightarrow R_{l_i}\ \vec{p}\ \vec{t}_i \qquad \text{for } i = 1, \ldots, r$$

where $\vec{A}_i$ and $\vec{B}_{ij}$ are formulae not containing $R_1, \ldots, R_n$. Note that by disallowing the inductive predicates to occur in $\vec{B}_{ij}$ we make sure that all occurrences of the predicates in the premises of the introduction rules are *strictly positive*. This condition guarantees the existence of predicates that are closed under the introduction rules shown above. The inductive predicates $R_1, \ldots, R_n$ can then be defined as follows:

$$R_i \equiv \lambda\vec{p}\ \vec{z}_i.\ \forall P_1 \ldots P_n.\ K_1 \longrightarrow \cdots \longrightarrow K_r \longrightarrow P_i\ \vec{z}_i \qquad \text{for } i = 1, \ldots, n$$

where

$$K_i \equiv \forall \vec{x}_i.\ \vec{A}_i \longrightarrow \left(\forall \vec{y}_{ij}.\ \vec{B}_{ij} \longrightarrow P_{k_{ij}}\ \vec{s}_{ij}\right)_{j=1,\ldots,m_i} \longrightarrow P_{l_i}\ \vec{t}_i \qquad \text{for } i = 1, \ldots, r$$

The (weak) induction rules for the inductive predicates $R_1, \ldots, R_n$ are

$$R_i\ \vec{p}\ \vec{z}_i \Longrightarrow I_1 \Longrightarrow \cdots \Longrightarrow I_r \Longrightarrow P_i\ \vec{z}_i \qquad \text{for } i = 1, \ldots, n$$

where

$$I_i \equiv \bigwedge \vec{x}_i.\ \vec{A}_i \Longrightarrow \left(\bigwedge \vec{y}_{ij}.\ \vec{B}_{ij} \Longrightarrow P_{k_{ij}}\ \vec{s}_{ij}\right)_{j=1,\ldots,m_i} \Longrightarrow P_{l_i}\ \vec{t}_i \qquad \text{for } i = 1, \ldots, r$$

Since $K_i$ and $I_i$ are equivalent modulo conversion between meta-level and object-level connectives, it is clear that the proof of the induction theorem is straightforward. We will therefore focus on the proof of the introduction rules. When proving the introduction rule shown above, we start by unfolding the definition of $R_1, \ldots, R_n$, which yields

$$\bigwedge \vec{x}_i.\ \vec{A}_i \Longrightarrow \left(\bigwedge \vec{y}_{ij}.\ \vec{B}_{ij} \Longrightarrow \forall P_1 \ldots P_n.\ \vec{K} \longrightarrow P_{k_{ij}}\ \vec{s}_{ij}\right)_{j=1,\ldots,m_i} \Longrightarrow \forall P_1 \ldots P_n.\ \vec{K} \longrightarrow P_{l_i}\ \vec{t}_i$$

where $\vec{K}$ abbreviates $K_1, \ldots, K_r$. Applying the introduction rules for $\forall$ and $\longrightarrow$ yields a proof state in which we have to prove $P_{l_i}\ \vec{t}_i$ from the additional assumptions $\vec{K}$. When using $K_{l_i}$ (converted to meta-logic format) to prove $P_{l_i}\ \vec{t}_i$, we get subgoals $\vec{A}_i$ that are trivially solvable by assumption, as well as subgoals of the form

$$\bigwedge \vec{y}_{ij}.\ \vec{B}_{ij} \Longrightarrow P_{k_{ij}}\ \vec{s}_{ij} \qquad \text{for } j = 1, \ldots, m_i$$

that can be solved using the assumptions

$$\bigwedge \vec{y}_{ij}.\ \vec{B}_{ij} \Longrightarrow \forall P_1 \ldots P_n.\ \vec{K} \longrightarrow P_{k_{ij}}\ \vec{s}_{ij} \qquad \text{and} \qquad \vec{K}$$

## 4.4 The interface

In order to add a new inductive predicate to a theory with the help of our package, the user must *invoke* it. For every package, there are essentially two different ways of invoking it, which we will refer to as *external* and *internal*. By external invocation we mean that the package is called from within a theory document. In this case, the type of the inductive predicate, as well as its introduction rules, are given as strings by the user. Before the package can actually make the definition, the type and introduction rules have to be parsed. In contrast, internal invocation means that the package is called by some other package. For example, the function definition package [2] calls the inductive definition package to define the graph of the function. However, it is not a good idea for the function definition package to pass the introduction rules for the function graph to the inductive definition package as strings. In this case, it is better to directly pass the rules to the package as a list of terms, which is more robust than handling strings that are lacking the additional structure of terms. These two ways of invoking the package are reflected in its ML programming interface, which consists of two functions:

```
signature SIMPLE_INDUCTIVE_PACKAGE =
sig
  val add_inductive_i:
    ((Name.binding * typ) * mixfix) list ->                    predicates
    (Name.binding * typ) list ->                               parameters
    (Attrib.binding * term) list ->                            rules
    local_theory -> (thm list * thm list) * local_theory
  val add_inductive:
    (Name.binding * string option * mixfix) list ->            predicates
    (Name.binding * string option * mixfix) list ->            parameters
    (Attrib.binding * string) list ->                          rules
    local_theory -> (thm list * thm list) * local_theory
end;
```

The function for external invocation of the package is called `add_inductive`, whereas the one for internal invocation is called `add_inductive_i`. Both of these functions take as arguments the names and types of the inductive predicates, the names and types of their parameters, the actual introduction rules and a *local theory*. They return a local theory containing the definition, together with a tuple containing the introduction and induction rules, which are stored in the local theory, too. In contrast to an ordinary theory, which simply consists of a type signature, as well as tables for constants, axioms and theorems, a local theory also contains additional context information, such as locally fixed variables and local assumptions that may be used by the package. The type `local_theory` is identical to the type of *proof contexts* `Proof.context`, although not every proof context constitutes a valid local theory. Note that `add_inductive_i` expects the types of the predicates and parameters to be specified using the datatype `typ` of Isabelle's logical framework, whereas `add_inductive` expects them to be given as optional strings. If no string is given for a particular predicate or parameter, this means that the type should be inferred by the package. Additional *mixfix syntax* may be associated with the predicates and parameters as well. Note that `add_inductive_i` does not allow mixfix syntax to be associated with parameters, since it can only be used for parsing. The names of the predicates, parameters and rules are represented by the type `Name.binding`. Strings can be turned into elements of the type `Name.binding` using the function

```
Name.binding : string -> Name.binding
```

Each introduction rule is given as a tuple containing its name, a list of *attributes* and a logical formula. Note that the type `Attrib.binding` used in the list of introduction rules is just a shorthand for the type `Name.binding * Attrib.src list`. The function `add_inductive_i` expects the formula to be specified using the datatype `term`, whereas `add_inductive` expects it to be given as a string. An attribute specifies additional actions and transformations that should be applied to a theorem, such as storing it in the rule databases used by automatic tactics like the simplifier. The code of the package, which will be described in the following section, will mostly treat attributes as a black box and just forward them to other functions for storing theorems in local theories. The implementation of the function `add_inductive` for external invocation of the package is quite simple. Essentially, it just parses the introduction rules and then passes them on to `add_inductive_i`:

```
fun add_inductive preds_syn params_syn intro_srcs lthy =
  let
    val ((vars, specs), _) = Specification.read_specification
      (preds_syn @ params_syn) (map (fn (a, s) => [(a, [s])]) intro_srcs)
      lthy;
    val (preds_syn', params_syn') = chop (length preds_syn) vars;
    val intrs = map (apsnd the_single) specs
  in
    add_inductive_i preds_syn' (map fst params_syn') intrs lthy
  end;
```

For parsing and type checking the introduction rules, we use the function

```
Specification.read_specification:
  (Name.binding * string option * mixfix) list ->          variables
  (Attrib.binding * string list) list list ->              rules
```

```
  local_theory ->
  (((Name.binding * typ) * mixfix) list *
   (Attrib.binding * term list) list) *
  local_theory
```

During parsing, both predicates and parameters are treated as variables, so the lists `preds_syn` and `params_syn` are just appended before being passed to `read_specification`. Note that the format for rules supported by `read_specification` is more general than what is required for our package. It allows several rules to be associated with one name, and the list of rules can be partitioned into several sublists. In order for the list `intro_srcs` of introduction rules to be acceptable as an input for `read_specification`, we first have to turn it into a list of singleton lists. This transformation has to be reversed later on by applying the function

```
the_single: 'a list -> 'a
```

to the list `specs` containing the parsed introduction rules. The function `read_specification` also returns the list `vars` of predicates and parameters that contains the inferred types as well. This list has to be chopped into the two lists `preds_syn'` and `params_syn'` for predicates and parameters, respectively. All variables occurring in a rule but not in the list of variables passed to `read_specification` will be bound by a meta-level universal quantifier. Finally, `read_specification` also returns another local theory, but we can safely discard it. As an example, let us look at how we can use this function to parse the introduction rules of the *trcl* predicate:

```
Specification.read_specification
  [(Name.binding "trcl", NONE, NoSyn),
   (Name.binding "r", SOME "'a ⇒ 'a ⇒ bool", NoSyn)]
  [[((Name.binding "base", []), ["trcl r x x"])],
   [((Name.binding "step", []), ["trcl r x y ⟹ r y z ⟹ trcl r x z"])]]
  @{context}

((...,
  [(...,
    [Const ("all", ...) $ Abs ("x", TFree ("'a", ...),
       Const ("Trueprop", ...) $
          (Free ("trcl", ...) $ Free ("r", ...) $ Bound 0 $ Bound 0))]),
   (...,
    [Const ("all", ...) $ Abs ("x", TFree ("'a", ...),
       Const ("all", ...) $ Abs ("y", TFree ("'a", ...),
          Const ("all", ...) $ Abs ("z", TFree ("'a", ...),
            Const ("==>", ...) $
               (Const ("Trueprop", ...) $
                  (Free ("trcl", ...) $ Free ("r", ...) $ Bound 2 $ Bound 1)) $
               (Const ("==>", ...) $ ... $ ...))))])]),
 ...)
: (((Name.binding * typ) * mixfix) list *
   (Attrib.binding * term list) list) * local_theory
```

In the list of variables passed to `read_specification`, we have used the mixfix annotation `NoSyn` to indicate that we do not want to associate any mixfix syntax with the variable. Moreover, we have only specified the type of `r`, whereas the type of `trcl` is computed using type inference. The local variables `x`, `y` and `z` of the introduction rules are turned into bound variables with the de Bruijn indices, whereas `trcl` and `r` remain free variables.

**Parsers for theory syntax** Although the function `add_inductive` parses terms and types, it still cannot be used to invoke the package directly from within a theory document. In order to do this, we have to write another parser. Before we describe the process of writing parsers for theory syntax in more detail, we first show some examples of how we would like to use the inductive definition package.

The definition of the transitive closure should look as follows:

```
simple_inductive
  trcl for r :: "'a ⇒ 'a ⇒ bool"
where
  base: "trcl r x x"
| step: "trcl r x y ⟹ r y z ⟹ trcl r x z"
```

Even and odd numbers can be defined by

```
simple_inductive
  even and odd
where
  even0: "even 0"
| evenS: "odd n ⟹ even (Suc n)"
| oddS: "even n ⟹ odd (Suc n)"
```

The accessible part of a relation can be introduced as follows:

```
simple_inductive
  accpart for r :: "'a ⇒ 'a ⇒ bool"
where
  accpartI: "(⋀y. r y x ⟹ accpart r y) ⟹ accpart r x"
```

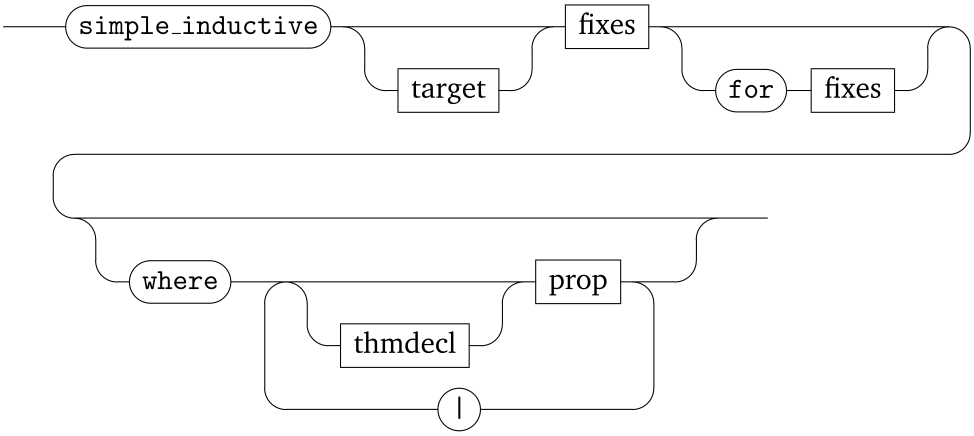
Moreover, it should also be possible to define the accessible part inside a locale fixing the relation *r*:

```
locale rel =
  fixes r :: "'a ⇒ 'a ⇒ bool"

simple_inductive (in rel) accpart'
where
  accpartI': "⋀x. (⋀y. r y x ⟹ accpart' y) ⟹ accpart' x"
```

In this context, it is important to note that Isabelle distinguishes between *outer* and *inner* syntax. Theory commands such as **simple_inductive** ... **for** ... **where** ... belong to the outer syntax, whereas items in quotation marks, in particular terms such as *"trcl r x x"* and types such as *"'a ⇒ 'a ⇒ bool"* belong to the inner syntax. Separating the two layers of outer and inner syntax greatly simplifies matters, because the parser for terms and types does not have to know anything about the possible syntax of theory commands, and the parser for theory commands need not be concerned about the syntactic structure of terms and types.

The syntax of the **simple_inductive** command can be described by the following railroad diagram:




**Functional parsers** For parsing terms and types, Isabelle uses a rather general and sophisticated algorithm due to Earley, which is driven by *priority grammars*. In contrast, parsers for theory syntax are built up using a set of combinators. Functional parsing using combinators is a well-established technique, which has been described by many authors, including Paulson [?] and Wadler [10]. The central idea is that a parser is a function of type `'a list -> 'b * 'a list`, where `'a` is a type of *tokens*, and `'b` is a type for encoding items that the parser has recognized. When a parser is applied to a list of tokens whose prefix it can recognize, it returns an encoding of the prefix as an element of type `'b`, together with the suffix of the list containing the remaining tokens. Otherwise, the parser raises an exception indicating a syntax error. The library for writing functional parsers in Isabelle can roughly be split up into two parts. The first part consists of a collection of generic parser combinators that are contained in the structure `Scan` defined in the file *Pure/General/scan.ML* in the Isabelle sources. While these combinators do not make any assumptions about the concrete structure of the tokens used, the second part of the library consists of combinators for dealing with specific token types. The following is an excerpt from the signature of `Scan`:

```
|| : ('a -> 'b) * ('a -> 'b) -> 'a -> 'b
-- : ('a -> 'b * 'c) * ('c -> 'd * 'e) -> 'a -> ('b * 'd) * 'e
|-- : ('a -> 'b * 'c) * ('c -> 'd * 'e) -> 'a -> 'd * 'e
--| : ('a -> 'b * 'c) * ('c -> 'd * 'e) -> 'a -> 'b * 'e
optional: ('a -> 'b * 'a) -> 'b -> 'a -> 'b * 'a
repeat: ('a -> 'b * 'a) -> 'a -> 'b list * 'a
repeat1: ('a -> 'b * 'a) -> 'a -> 'b list * 'a
>> : ('a -> 'b * 'c) * ('b -> 'd) -> 'a -> 'd * 'c
!! : ('a * string option -> string) -> ('a -> 'b) -> 'a -> 'b
```

Interestingly, the functions shown above are so generic that they do not even rely on the input and output of the parser being a list of tokens. If `p` succeeds, i.e. does not raise an exception, the parser `p || q` returns the result of `p`, otherwise it returns the result of `q`. The parser `p -- q` first parses an item of type `'b` using `p`, then passes the remaining tokens of type `'c` to `q`, which parses an item of type `'d` and returns the remaining tokens of type `'e`, which are finally returned together with a pair of type `'b * 'd` containing the two parsed items. The parsers `p |-- q` and `p --| q` work in a similar way as the previous one, with the difference that they discard the item parsed by the first and the second parser, respectively. If `p` succeeds, the parser `optional p x` returns the result of `p`, otherwise it returns the default value `x`. The parser `repeat p` applies `p` as often as it can, returning a possibly empty list of parsed items. The parser `repeat1 p` is similar, but requires `p` to succeed at least once. The parser `p >> f` uses `p` to parse an item of type `'b`, to which it applies the function `f` yielding a value of type `'d`, which is returned together with the remaining tokens of type `'c`. Finally, `!!` is used for transforming exceptions produced by parsers. If `p` raises an exception indicating that it cannot parse a given input, then an enclosing parser such as

```
q -- p || r
```

will try the alternative parser `r`. By writing

```
q -- !! err p || r
```

instead, one can achieve that a failure of `p` causes the whole parser to abort. The `!!` operator is similar to the *cut* operator in Prolog, which prevents the interpreter from backtracking. The `err` function supplied as an argument to `!!` can be used to produce an error message depending on the current state of the parser, as well as the optional error message returned by `p`.

So far, we have only looked at combinators that construct more complex parsers from simpler parsers. In order for these combinators to be useful, we also need some basic parsers. As an example, we consider the following two parsers defined in `Scan`:

```
  one: ('a -> bool) -> 'a list -> 'a * 'a list
  $$ : string -> string list -> string * string list
```

The parser `one pred` parses exactly one token that satisfies the predicate `pred`, whereas `$$ s` only accepts a token that equals the string `s`. Note that we can easily express `$$ s` using `one`:

```
one (fn s' => s' = s)
```

As an example, let us look at how we can use `$$` and `--` to parse the prefix "`hello`" of the character list "`hello world`":

```
($$ "h" -- $$ "e" -- $$ "l" -- $$ "l" -- $$ "o")
["h", "e", "l", "l", "o", " ", "w", "o", "r", "l", "d"]

((((("h", "e"), "l"), "l"), "o"), [" ", "w", "o", "r", "l", "d"])
: ((((string * string) * string) * string) * string) * string list
```

Most of the time, however, we will have to deal with tokens that are not just strings. The parsers for the theory syntax, as well as the parsers for the argument syntax of proof methods and attributes use the token type `OuterParse.token`, which is identical to `OuterLex.token`. The parser functions for the theory syntax are contained in the structure `OuterParse` defined in the file *Pure/Isar/outer_parse.ML*. In our parser, we will use the following functions:

```
$$$ : string -> token list -> string * token list
enum1: string -> (token list -> 'a * token list) -> token list ->
  'a list * token list
prop: token list -> string * token list
opt_target: token list -> string option * token list
fixes: token list ->
  (Name.binding * string option * mixfix) list * token list
for_fixes: token list ->
  (Name.binding * string option * mixfix) list * token list
!!! : (token list -> 'a) -> token list -> 'a
```

The parsers `$$$` and `!!!` are defined using the parsers `one` and `!!` from `Scan`. The parser `enum1 s p` parses a non-emtpy list of items recognized by the parser `p`, where the items are separated by `s`. A proposition can be parsed using the function `prop`. Essentially, a proposition is just a string or an identifier, but using the specific parser function `prop` leads to more instructive error messages, since the parser will complain that a proposition was expected when something else than a string or identifier is found. An optional locale target specification of the form *(***in** *...)* can be parsed using `opt_target`. The lists of names of the predicates and parameters, together with optional types and syntax, are parsed using the functions `fixes` and `for_fixes`, respectively. In addition, the following function from `SpecParse` for parsing an optional theorem name and attribute, followed by a delimiter, will be useful:

```
opt_thm_name:
  string -> token list -> Attrib.binding * token list
```

We now have all the necessary tools to write the parser for our **simple_inductive** command:

```
local structure P = OuterParse and K = OuterKeyword in

val ind_decl =
  P.opt_target --
  P.fixes -- P.for_fixes --
  Scan.optional (P.$$$ "where" |--
    P.!!! (P.enum1 "|" (SpecParse.opt_thm_name ":" -- P.prop))) [] >>
  (fn (((loc, preds), params), specs) =>
    Toplevel.local_theory loc (add_inductive preds params specs #> snd));

val _ = OuterSyntax.command "simple_inductive" "define inductive predicates"
  K.thy_decl ind_decl;

end;
```

The definition of the parser `ind_decl` closely follows the railroad diagram shown above. In order to make the code more readable, the structures `OuterParse` and `OuterKeyword` are abbreviated by `P` and `K`, respectively. Note how the parser combinator `!!!` is used: once the keyword `where` has been parsed, a non-empty list of introduction rules must follow. Had we not used the combinator `!!!`, a `where` not followed by a list of rules would have caused the parser to respond with the somewhat misleading error message

```
Outer syntax error: end of input expected, but keyword where was found
```

rather than with the more instructive message

```
Outer syntax error: proposition expected, but terminator was found
```

Once all arguments of the command have been parsed, we apply the function `add_inductive`, which yields a local theory transformer of type `local_theory -> local_theory`. Commands in Isabelle/Isar are realized by transition transformers of type

```
Toplevel.transition -> Toplevel.transition
```

We can turn a local theory transformer into a transition transformer by using the function

```
Toplevel.local_theory : string option ->
  (local_theory -> local_theory) ->
  Toplevel.transition -> Toplevel.transition
```

which, apart from the local theory transformer, takes an optional name of a locale to be used as a basis for the local theory. The whole parser for our command has type

```
OuterLex.token list ->
  (Toplevel.transition -> Toplevel.transition) * OuterLex.token list
```

which is abbreviated by `OuterSyntax.parser_fn`. The new command can be added to the system via the function

```
OuterSyntax.command :
  string -> string -> OuterKeyword.T -> OuterSyntax.parser_fn -> unit
```

which imperatively updates the parser table behind the scenes. In addition to the parser, this function takes two strings representing the name of the command and a short description, as well as an element of type `OuterKeyword.T` describing which *kind* of command we intend to add. Since we want to add a command for declaring new concepts, we choose the kind `OuterKeyword.thy_decl`. Other kinds include `OuterKeyword.thy_goal`, which is similar to `thy_decl`, but requires the user to prove a goal before making the declaration, or `OuterKeyword.diag`, which corresponds to a purely diagnostic command that does not change the context. For example, the `thy_goal` kind is used by the **function** command [2], which requires the user to prove that a given set of equations is non-overlapping and covers all cases. The kind of the command should be chosen with care, since selecting the wrong one can cause strange behaviour of the user interface, such as failure of the undo mechanism.

# Appendix A

# Recipes

## A.1 Accumulate a List of Theorems under a Name

**Problem:** Your tool *foo* works with special rules, called *foo*-rules. Users should be able to declare *foo*-rules in the theory, which are then used by some method.

**Solution:** This can be achieved using

```
ML {*
  structure FooRules = NamedThmsFun(
    val name = "foo"
    val description = "Rules for foo"
  );
*}
```

```
setup FooRules.setup
```

This code declares a context data slot where the theorems are stored, an attribute *foo* (with the usual *add* and *del* options to adding and deleting theorems) and an internal ML interface to retrieve and modify the theorems.

Furthermore, the facts are made available on the user level under the dynamic fact name *foo*. For example:

```
lemma rule1[foo]: "A" sorry
lemma rule2[foo]: "B" sorry
```

```
declare rule1[foo del]
```

```
thm foo
```

In an ML-context the rules marked with *foo* an be retrieved using

```
ML {* FooRules.get @{context} *}
```

*For more information see `Pure/Tools/named_thms.ML`.* **Read More**

(FIXME: maybe add a comment about the case when the theorems to be added need to satisfy certain properties)

## A.2 Ad-hoc Transformations of Theorems

# Appendix B

# Solutions to Most Exercises

**Solution for Exercise 2.5.1.**

```
ML {*
  fun rev_sum t =
  let
   fun dest_sum (Const (@{const_name plus}, _) $ u $ u') = u' :: dest_sum u
     | dest_sum u = [u]
   in
     foldl1 (HOLogic.mk_binop @{const_name plus}) (dest_sum t)
   end;
*}
```

**Solution for Exercise 2.5.2.**

```
ML {*
  fun make_sum t1 t2 =
      HOLogic.mk_nat (HOLogic.dest_nat t1 + HOLogic.dest_nat t2)
*}
```

# Bibliography

[1] R. Bornat. In defence of programming. Available online via http://www.cs.mdx.ac.uk/staffpages/r_bornat/lectures/revisedinauguraltext.pdf, April 2005. Corrected and revised version of inaugural lecture, delivered on 22nd January 2004 at the School of Computing Science, Middlesex University.

[2] A. Krauss. Partial Recursive Functions in Higher-Order Logic. In U. Furbach and N. Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 589–603. Springer-Verlag, 2006.

[3] T. F. Melham. A Package for Inductive Relation Definitions in HOL. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, Davis, California, August 28–30, 1991*, pages 350–357. IEEE Computer Society Press, 1992.

[4] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002. LNCS Tutorial 2283.

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

[6] L. C. Paulson. A fixedpoint approach to (co)inductive and (co)datatype definitions. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 187–211. MIT Press, 2000.

[7] N. Schirmer. A Verification Environment for Sequential Imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3452 of *Lecture Notes in Artificial Intelligence*, pages 398–414. Springer-Verlag, 2005.

[8] H. Schwichtenberg. Minimal Logic for Computable Functionals. Technical report, Mathematisches Institut, Ludwig-Maximilians-Universität München, December 2005. Available online at http://www.mathematik.uni-muenchen.de/~minlog/minlog/mlcf.pdf.

[9] C. Urban and S. Berghofer. A Recursion Combinator for Nominal Datatypes Implemented in Isabelle/HOL. In U. Furbach and N. Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 498–512. Springer-Verlag, 2006.

[10] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer-Verlag, 1995.