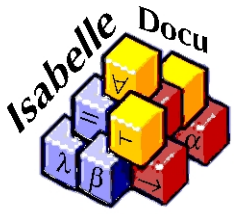


The Isabelle Programmer's Cookbook (fragment)

with contributions by:

Stefan Berghofer
Sascha Böhme
Jeremy Dawson
Alexander Krauss

January 27, 2009

Contents

Chapter 1

Introduction

The purpose of this Cookbook is to guide the reader through the first steps of Isabelle programming, and to explain tricks of the trade. The code provided in the Cookbook is as far as possible checked against recent versions of Isabelle. If something does not work, then please let us know. If you have comments, criticism or like to add to the Cookbook, feel free—you are most welcome!

1.1 Intended Audience and Prior Knowledge

This Cookbook targets readers who already know how to use Isabelle for writing theories and proofs. We also assume that readers are familiar with the functional programming language ML, the 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] or Paulson's book on ML [5].

1.2 Existing Documentation

The following documentation about Isabelle programming already exists (and is part of the distribution of Isabelle):

The Implementation Manual describes Isabelle from a high-level 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 of Isabelle 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 Many parts of this manual are outdated now, but some parts, particularly the chapters on tactics, are still useful.

The Isar Reference Manual is also an older document that provides material about Isar and its implementation. Some material in it is 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.

1.3 Typographic Conventions

All ML-code in this Cookbook is typeset in highlighed boxes, like the following ML-expression.

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

This corresponds to how code can be processed inside the interactive environment of Isabelle. It is therefore easy to experiment with the code (which by the way is highly recommended). However, for better readability we will drop the enclosing **ML** `{* ... *}` and just write

```
3 + 4
```

for the code above. Whenever appropriate we also show the response the code generates when evaluated. This response is prefixed with a `">"` like

```
3 + 4
> 7
```

The usual Isabelle commands are written in bold, for example **lemma**, **foobar** and so on. We use `$` to indicate that a command needs to be run on the Unix-command line, for example

```
$ ls -la
```

Pointers to further information and Isabelle files are given as follows:

Further information or pointer to file. **Read More**

Chapter 2

First Steps

Isabelle programming is done in ML. Just like lemmas and proofs, ML-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 starting with

theory FirstSteps
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
*}
> 7
```

Like "normal" Isabelle proof scripts, **ML**-commands can be evaluated by using the advance and undo buttons of your Isabelle environment. The code inside the **ML**-command can also contain value and function bindings, and even those can be undone when the proof script is retracted. As mentioned earlier, we will drop the **ML** *{** ... **}* whenever we show code.

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 FirstSteps
imports Main
uses *"file_to_be_included.ML"* ...
begin
...

2.2 Debugging and Printing

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

```
warning "any string"
> "any string"
```

will print out *"any string"* inside the response buffer of Isabelle. This function expects a string as argument. If you develop under PolyML, then there is a convenient, though again "quick-and-dirty", method for converting values into strings, namely using the function *makestring*:

```
warning (makestring 1)
> "1"
```

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

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

```
tracing "foo"
> "foo"
```

It is also possible to redirect the "channel" where the string *foo* is printed to a separate file, e.g. to prevent ProofGeneral from choking on massive amounts of trace output. This redirection can be achieved using the code

```
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*.

Error messages can be printed using the function *error*, as in

```
if 0=1 then 1 else (error "foo")
> "foo"
```

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. By this we mean definitions, theorems, terms and so on. This kind of reference is realised with antiquotations. For example, one can print out the name of the current theory by typing

```
Context.theory_name @{theory}
> "FirstSteps"
```

where `@{theory}` is an antiquotation that is substituted with the current theory (remember that we assumed we are inside the theory `FirstSteps`). The name of this theory can be extracted with the function `Context.theory_name`.

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

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

does, as its name suggest, *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 `"FirstSteps"`, no matter where the function is called. Operationally speaking, the antiquotation `@{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 proved theorems:

```
@{thm allI}
> (⋀x. ?P x) ⟹ ∀x. ?P x
```

or simpsets:

```
let
  val ({rules,...},_) = MetaSimplifier.rep_ss @{simpset}
in
  map #name (Net.entries rules)
end
> ["Nat.of_nat_eq_id", "Int.of_int_eq_id", "Nat.One_nat_def", ...]
```

The code about simpsets extracts the theorem names that are stored in the current simpset. We get hold of the current simpset with the antiquotation `@{simpset}`. The function `rep_ss` returns a record containing all information about the simpset. The rules of a simpset are stored in a *discrimination net* (a datastructure for fast indexing). From this net we can extract the entries using the function `Net.entries`.

Read More

The infrastructure for simpsets is implemented in `Pure/meta_simplifier.ML` and `Pure/simplifier.ML`. Discrimination nets are implemented in `Pure/net.ML`.

While antiquotations have many applications, they were originally introduced in order to avoid explicit bindings for theorems such as

```
val allI = thm "allI"
```

These bindings are difficult to maintain and also can be accidentally overwritten by the user. This often breakes definitional packages. Antiquotations solve this problem, since they are "linked" statically at compile-time. However, this static linkage also limits their usefulness in cases where data needs to be build up dynamically. 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 ...}`. For example

```
@{term "(a::nat) + b = c"}
> Const ("op =", ...) $
>                 (Const ("HOL.plus_class.plus", ...) $ ... $ ...) $ ...
```

This will show the term `a + b = c`, but printed 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.

Read More

Terms are described in detail in [Impl. Man., Sec. 2.2]. Their definition and many useful operations are implemented in `Pure/term.ML`.

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 function to set the limit to a value high enough:

```
print_depth 50
```

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

```
(@{term "P x"}, @{prop "P x"})
> (Free ("P", ...) $ Free ("x", ...),
>  Const ("Trueprop", ...) $ (Free ("P", ...) $ Free ("x", ...)))
```

where an coercion is inserted in the second component and

```
(@{term "P x ⟹ Q x"}, @{prop "P x ⟹ Q x"})
> (Const ("==>", ...) $ ... $ ..., Const ("==>", ...) $ ... $ ...)
```

where it is not (since it is already constructed by a meta-implication).
Types can be constructed using the antiquotation `@{typ ...}`. For example

```
@{typ "bool ⇒ nat"}
> bool ⇒ nat
```

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

2.5 Constructing Terms and Types Manually

While antiquotations are very convenient for constructing terms, they can only construct fixed terms (remember they are "linked" at compile-time). See Recipe A.7 on Page 58 for a function that pattern-matches over terms and where the pattern are constructed from antiquotations. However, one often needs to construct terms dynamically. For example, a function that returns the implication `⋀(x::τ). P x ⟹ Q x` taking `P`, `Q` and the type τ as arguments can only be written as

```
fun make_imp P Q tau =
  let
    val x = Free ("x",tau)
  in
    Logic.all x (Logic.mk_implies (P $ x, 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:

```
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. With `make_imp` we obtain the intended term involving `S`, `T` and `@{typ nat}`

```
make_imp @{term S} @{term T} @{typ nat}
> Const ... $
>     Abs ("x", Type ("nat",[]),
>       Const ... $ (Free ("S",...) $ ...) $
>                   (Free ("T",...) $ ...))
```

whereas with `make_wrong_imp` we obtain a term involving the `P` and `Q` from the antiquotation.

```
make_wrong_imp @{term S} @{term T} @{typ nat}
> Const ... $
>     Abs ("x", ...,
>       Const ... $ (Const ... $ (Free ("P",...) $ ...)) $
>                   (Const ... $ (Free ("Q",...) $ ...)))
```

(FIXME: expand the following point)

One tricky point in constructing terms by hand is to obtain the fully qualified name for constants. For example the names for `zero` and `+` 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 Isabelle provides the antiquotation `@{const_name ...}` which does the expansion automatically, for example:

```
@{const_name "Nil"}
> List.list.Nil
```

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

Similarly, one can construct types manually. For example the function returning a function type is as follows:

```
fun make_fun_type tau1 tau2 = Type ("fun",[tau1,tau2])
```

This can be equally written as

```
fun make_fun_type tau1 tau2 = tau1 --> tau2
```

Read More

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

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 `+ ... +` t_n (whereby `i` might be zero) and returns the reversed sum t_n `+ ... +` 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 notation (like `Suc (Suc (Suc 0))`), and produce the 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 converts a `term` into a `cterm`, a *certified* term. Unlike `term`s, which are just trees, `cterm`s are abstract objects that are guaranteed to be type-correct, and they can only be constructed via "official interfaces".

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

```
cterm_of @{theory} @{term "a + b = c"}
> a + b = c
```

This can also be wirtten with an antiquotation

```
@{cterm "(a::nat) + b = c"}
> a + b = c
```

Attempting to obtain the certified term for

```
@{cterm "1 + True"}
> Type unification failed ...
```

yields an error (since the term is not typable). A slightly more elaborate example that type-checks is

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

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 *cterm*s, theorems are abstract objects of type *thm* that can only be built by going through interfaces. As a consequence, every proof in Isabelle is correct by construction (FIXME reference LCF-philosophy)

To see theorems in "action", let us give a proof on the ML-level for the following statement:

lemma
  assumes $assm_1$: "⋀(x::nat). P x ⟹ Q x"
  and $assm_2$: "P t"
  shows "Q t"

The corresponding ML-code is as follows:[1]

```
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
```

[1] Note that |> is reverse application. See Section 2.11.

```
  Qt
  |> implies_intr assm2
  |> implies_intr assm1
end
> ⟦⋀x. P x ⟹ Q x; P t⟧ ⟹ Q t
```

This code-snippet constructs the following proof:

$$\dfrac{\dfrac{\dfrac{}{\bigwedge x.\ P\ x \Longrightarrow Q\ x \vdash \bigwedge x.\ P\ x \Longrightarrow Q\ x}\,(assume)}{\bigwedge x.\ P\ x \Longrightarrow Q\ x \vdash P\ t \Longrightarrow Q\ t}\,(\textstyle\bigwedge\text{-}elim) \qquad \dfrac{}{P\ t \vdash P\ t}\,(assume)}{\dfrac{\bigwedge x.\ P\ x \Longrightarrow Q\ x, P\ t \vdash Q\ t}{\bigwedge x.\ P\ x \Longrightarrow Q\ x \vdash P\ t \Longrightarrow Q\ t}\,(\Longrightarrow\text{-}intro)}{\vdash ⟦\bigwedge x.\ P\ x \Longrightarrow Q\ x; P\ t⟧ \Longrightarrow Q\ t}\,(\Longrightarrow\text{-}intro)}\,(\Longrightarrow\text{-}elim)$$

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

Read More

2.8 Storing Theorems

2.9 Theorem Attributes

2.10 Operations on Constants (Names)

```
Sign.base_name "List.list.Nil"
> "Nil"
```

2.11 Combinators

For beginners, perhaps the most puzzling parts in the existing code of Isabelle are the combinators. At first they seem to greatly obstruct the comprehension of the code, but after getting familiar with them, they actually ease the understanding and also the programming.

The most frequently used combinator are defined in the files `Pure/library.ML` *and* `Pure/General/basics.ML`. *The section ?? in the implementation manual contains also information about combinators.*

Read More

The simplest combinator is `I`, which is just the identity function.

```
fun I x = x
```

Another simple combinator is `K`, defined as

```
fun K x = fn _ => x
```

`K` "wraps" a function around the argument `x`. However, this function ignores its argument. So `K` defines a constant function returning `x`.

The next combinator is reverse application, `|>`, defined as

```
fun x |> f = f x
```

While just syntactic sugar for the usual function application, the purpose of this combinator is to implement functions in a "waterfall fashion". Consider for example the function

```
1 fun inc_by_five x =
2   x |> (fn x => x + 1)
3     |> (fn x => (x, x))
4     |> fst
5     |> (fn x => x + 4)
```

which increments the argument `x` by 5. It does this by first incrementing the argument by 1 (Line 2); then storing the result in a pair (Line 3); taking the first component of the pair (Line 4) and finally incrementing the first component by 4 (Line 5). This kind of cascading manipulations of values is quite common when dealing with theories (for example by adding a definition, followed by lemmas and so on). It should also be familiar to anyone who has used Haskell's do-notation. Writing the function `inc_by_five` using the reverse application is much clearer than writing

```
fun inc_by_five x = fst ((fn x => (x, x)) (x + 1)) + 4
```

or

```
fun inc_by_five x =
       ((fn x => x + 4) o fst o (fn x => (x, x)) o (fn x => x + 1)) x
```

and typographically more economical than

```
fun inc_by_five x =
   let val y1 = x + 1
       val y2 = (y1, y1)
       val y3 = fst y2
       val y4 = y3 + 4
   in y4 end
```

Another reason why the let-bindings in the code above are better to be avoided: it is more than easy to get the intermediate values wrong, not to mention the nightmares the maintenance of this code causes!

(FIXME: give a real world example involving theories)

Similarly, the combinator `#>` is the reverse function composition. It can be used to define functions as follows

```
val inc_by_six =
      (fn x => x + 1)
   #> (fn x => x + 2)
   #> (fn x => x + 3)
```

which is the function composed of first the increment-by-one function and then increment-by-two, followed by increment-by-three. Again, the reverse function composition allows one to read the code top-down.

The remaining combinators described in this section add convenience for the "waterfall method" of writing functions. The combinator `tap` allows one to get hold of an intermediate result (to do some side-calculations for instance). The function

```
1 fun inc_by_three x =
2      x |> (fn x => x + 1)
3        |> tap (fn x => tracing (makestring x))
4        |> (fn x => x + 2)
```

increments the argument first by one and then by two. In the middle (Line 3), however, it uses `tap` for printing the "plus-one" intermediate result inside the tracing buffer. The function `tap` can only be used for side-calculations, because any value that is computed cannot be merged back into the "main waterfall". To do this, the next combinator can be used.

The combinator `‘` is similar to `tap`, but applies a function to the value and returns the result together with the value (as a pair). For example the function

```
fun inc_as_pair x =
     x |> ‘(fn x => x + 1)
       |> (fn (x, y) => (x, y + 1))
```

takes `x` as argument, and then first increments `x`, but also keeps `x`. The intermediate result is therefore the pair `(x + 1, x)`. The function then increments the right-hand component of the pair. So finally the result will be `(x + 1, x + 1)`.

The combinators `|>>` and `||>` are defined for functions manipulating pairs. The first applies the function to the first component of the pair, defined as:

```
fun (x, y) |>> f = (f x, y)
```

and the second combinator to the second component, defined as

```
fun (x, y) ||> f = (x, f y)
```

With the combinator `|->` you can re-combine the elements from a pair. This combinator is defined as

```
fun (x, y) |-> f = f x y
```

and can be used to write the following version of the `double` function

```
fun double x =
      x |>  (fn x => (x, x))
        |-> (fn x => fn y => x + y)
```

Recall that `|>` is the reverse function applications. The related reverse function composition is `#>`. In fact all combinators `|->`, `|>>` and `||>` described above have related combinators for function composition, namely `#->`, `#>>` and `##>`. Using `|->`, the function `double` can also be written as

```
val double =
            (fn x => (x, x))
        #-> (fn x => fn y => x + y)
```

(FIXME: find a good exercise for combinators)

Chapter 3

Parsing

Isabelle distinguishes between *outer* and *inner* syntax. Theory commands, such as **definition**, **inductive** and so on, belong to the outer syntax, whereas items inside double quotation marks, such as terms, types and so on, belong to the inner syntax. For parsing inner syntax, Isabelle uses a rather general and sophisticated algorithm due to Earley, which is driven by priority grammars. Parsers for outer syntax are built up by functional parsing combinators. These combinators are a well-established technique for parsing, which has, for example, been described in Paulson's classic ML-book [5]. Isabelle developers are usually concerned with writing these outer syntax parsers, either for new definitional packages or for calling tactics with specific arguments.

Read More

The library for writing parser combinators is split up, roughly, into two parts. The first part consists of a collection of generic parser combinators defined in the structure `Scan` in the file `Pure/General/scan.ML`. The second part of the library consists of combinators for dealing with specific token types, which are defined in the structure `OuterParse` in the file `Pure/Isar/outer_parse.ML`.

3.1 Building Generic Parsers

Let us first have a look at parsing strings using generic parsing combinators. The function `(op $$)` takes a string as argument and will "consume" this string from a given input list of strings. "Consume" in this context means that it will return a pair consisting of this string and the rest of the input list. For example:

```
($$ "h") (explode "hello")
> ("h", ["e", "l", "l", "o"])
```

```
($$ "w") (explode "world")
> ("w", ["o", "r", "l", "d"])
```

This function will either succeed (as in the two examples above) or raise the exception `FAIL` if no string can be consumed. For example trying to parse

```
($$ "x") (explode "world")
> Exception FAIL raised
```

will raise the exception *FAIL*. There are three exceptions used in the parsing combinators:

- *FAIL* is used to indicate that alternative routes of parsing might be explored.
- *MORE* indicates that there is not enough input for the parser. For example in *($$ "h") []*.
- *ABORT* is the exception that is raised when a dead end is reached. It is used for example in the function *(op !!)* (see below).

However, note that these exceptions are private to the parser and cannot be accessed by the programmer (for example to handle them).

Slightly more general than the parser *(op $$)* is the function *Scan.one*, in that it takes a predicate as argument and then parses exactly one item from the input list satisfying this predicate. For example the following parser either consumes an *"h"* or a *"w"*:

```
let
  val hw = Scan.one (fn x => x = "h" orelse x = "w")
  val input1 = (explode "hello")
  val input2 = (explode "world")
in
    (hw input1, hw input2)
end
> (("h", ["e", "l", "l", "o"]),("w", ["o", "r", "l", "d"]))
```

Two parser can be connected in sequence by using the function *(op --)*. For example parsing *h*, *e* and *l* in this sequence can be achieved by

```
(($$ "h") -- ($$ "e") -- ($$ "l")) (explode "hello")
> ((("h", "e"), "l"), ["l", "o"])
```

Note how the result of consumed strings builds up on the left as nested pairs.

If, as in the previous example, one wants to parse a particular string, then one should use the function *Scan.this_string*:

```
Scan.this_string "hell" (explode "hello")
> ("hell", ["o"])
```

Parsers that explore alternatives can be constructed using the function *(op ||)*. For example, the parser *(p || q)* returns the result of *p*, in case it succeeds, otherwise it returns the result of *q*. For example

```
let
  val hw = ($$ "h") || ($$ "w")
  val input1 = (explode "hello")
  val input2 = (explode "world")
in
  (hw input1, hw input2)
end
> (("h", ["e", "l", "l", "o"]), ("w", ["o", "r", "l", "d"]))
```

The functions `(op |--)` and `(op --|)` work like the sequencing function for parsers, except that they discard the item being parsed by the first (respectively second) parser. For example

```
let
  val just_e = ($$ "h") |-- ($$ "e")
  val just_h = ($$ "h") --| ($$ "e")
  val input = (explode "hello")
in
  (just_e input, just_h input)
end
> (("e", ["l", "l", "o"]),("h", ["l", "l", "o"]))
```

The parser `Scan.optional p x` returns the result of the parser `p`, if it succeeds; otherwise it returns the default value `x`. For example

```
let
  val p = Scan.optional ($$ "h") "x"
  val input1 = (explode "hello")
  val input2 = (explode "world")
in
  (p input1, p input2)
end
> (("h", ["e", "l", "l", "o"]), ("x", ["w", "o", "r", "l", "d"]))
```

The function `Scan.option` works similarly, except no default value can be given. Instead, the result is wrapped as an `option`-type. For example:

```
let
  val p = Scan.option ($$ "h")
  val input1 = (explode "hello")
  val input2 = (explode "world")
in
  (p input1, p input2)
end
> ((SOME "h", ["e", "l", "l", "o"]), (NONE, ["w", "o", "r", "l", "d"]))
```

The function `(op !!)` helps to produce appropriate error messages during parsing. For example if one wants to parse that `p` is immediately followed by `q`, or start a completely different parser `r`, one might write

```
(p -- q) || r
```

However, this parser is problematic for producing an appropriate error message, in case the parsing of `(p -- q)` fails. Because in that case one loses the information that `p` should be followed by `q`. To see this consider the case in which `p` is present in the input, but not `q`. That means `(p -- q)` will fail and the alternative parser `r` will be tried. However in many circumstance this will be the wrong parser for the input "p-followed-by-q" and therefore will also fail. The error message is then caused by the failure of `r`, not by the absence of `q` in the input. This kind of situation can be avoided when using the function `(op !!)`. This function aborts the whole process of parsing in case of a failure and prints an error message. For example if we invoke the parser

```
(!! (fn _ => "foo") ($$ "h"))
```

on `"hello"`, the parsing succeeds

```
(!! (fn _ => "foo") ($$ "h")) (explode "hello")
> ("h", ["e", "l", "l", "o"])
```

but if we invoke it on `"world"`

```
(!! (fn _ => "foo") ($$ "h")) (explode "world")
> Exception ABORT raised
```

then the parsing aborts and the error message `foo` is printed out. In order to see the error message, we need to prefix the parser with the function `Scan.error`. For example

```
Scan.error (!! (fn _ => "foo") ($$ "h"))
> Exception Error "foo" raised
```

This "prefixing" is usually done by wrappers such as `OuterSyntax.command` (FIXME: give reference to later place).

Let us now return to our example of parsing `(p -- q) || r`. If we want to generate the correct error message for p-followed-by-q, then we have to write:

```
fun p_followed_by_q p q r =
  let
     val err_msg = (fn _ => p ^ " is not followed by " ^ q)
  in
    ($$ p -- (!! err_msg ($$ q))) || ($$ r -- $$ r)
  end
```

Running this parser with the `"h"` and `"e"`, and the input `"holle"`

```
Scan.error (p_followed_by_q "h" "e" "w") (explode "holle")
> Exception ERROR "h is not followed by e" raised
```

produces the correct error message. Running it with

```
Scan.error (p_followed_by_q "h" "e" "w") (explode "wworld")
> (("w", "w"), ["o", "r", "l", "d"])
```

yields the expected parsing.

The function `Scan.repeat p` will apply a parser `p` as often as it succeeds. For example

```
Scan.repeat ($$ "h") (explode "hhhhello")
> (["h", "h", "h", "h"], ["e", "l", "l", "o"])
```

Note that `Scan.repeat` stores the parsed items in a list. The function `Scan.repeat1` is similar, but requires that the parser `p` succeeds at least once.

Also note that the parser would have aborted with the exception `MORE`, if we had run it only on just `"hhhh"`. This can be avoided by using the wrapper `Scan.finite` and the "stopper-token" `Symbol.stopper`. With them we can write

```
Scan.finite Symbol.stopper (Scan.repeat ($$ "h")) (explode "hhhh")
> (["h", "h", "h", "h"], [])
```

`Symbol.stopper` is the "end-of-input" indicator for parsing strings; other stoppers need to be used when parsing token, for example. However, this kind of manually wrapping is often already done by the surrounding infrastructure.

The function `Scan.repeat` can be used with `Scan.one` to read any string as in

```
let
   val p = Scan.repeat (Scan.one Symbol.not_eof)
   val input = (explode "foo bar foo")
in
   Scan.finite Symbol.stopper p input
end
> (["f", "o", "o", " ", "b", "a", "r", " ", "f", "o", "o"], [])
```

where the function `Symbol.not_eof` ensures that we do not read beyond the end of the input string (i.e. stopper symbol).

The function `Scan.unless p q` takes two parsers: if the first one can parse the input, then the whole parser fails; if not, then the second is tried. Therefore

```
Scan.unless ($$ "h") ($$ "w") (explode "hello")
> Exception FAIL raised
```

fails, while

```
Scan.unless ($$ "h") ($$ "w") (explode "world")
> ("w",["o", "r", "l", "d"])
```

succeeds.

The functions `Scan.repeat` and `Scan.unless` can be combined to read any input until a certain marker symbol is reached. In the example below the marker symbol is a `"*"`.

```
let
  val p = Scan.repeat (Scan.unless ($$ "*") (Scan.one Symbol.not_eof))
  val input1 = (explode "fooooo")
  val input2 = (explode "foo*ooo")
in
  (Scan.finite Symbol.stopper p input1,
   Scan.finite Symbol.stopper p input2)
end
> ((["f", "o", "o", "o", "o", "o"], []),
>  (["f", "o", "o"], ["*", "o", "o", "o"]))
```

After parsing is done, one nearly always wants to apply a function on the parsed items. To do this the function `(p >> f)` can be employed, which runs first the parser `p` and upon successful completion applies the function `f` to the result. For example

```
let
  fun double (x,y) = (x^x,y^y)
in
  (($$ "h") -- ($$ "e") >> double) (explode "hello")
end
> (("hh", "ee"), ["l", "l", "o"])
```

doubles the two parsed input strings. Or

```
let
   val p = Scan.repeat (Scan.one Symbol.not_eof) >> implode
   val input = (explode "foo bar foo")
in
   Scan.finite Symbol.stopper p input
end
> ("foo bar foo",[])
```

where the single-character strings in the parsed output are transformed back into one string.

Exercise 3.1.1. *Write a parser that parses an input string so that any comment enclosed inside `(*...*)` is replaced by a the same comment but enclosed inside `(**...**)` in the output string. To enclose a string, you can use the function `enclose s1 s2 s` which produces the string `s1^s^s2`.*

The function `Scan.lift` takes a parser and a pair as arguments. This function applies the given parser to the second component of the pair and leaves the first component untouched. For example

```
Scan.lift (($$ "h") -- ($$ "e")) (1,(explode "hello"))
> (("h", "e"), (1, ["l", "l", "o"]))
```

(FIXME: In which situations is this useful? Give examples.)

3.2 Parsing Theory Syntax

Most of the time, however, Isabelle developers have to deal with parsing tokens, not strings. This is because the parsers for the theory syntax, as well as the parsers for the argument syntax of proof methods and attributes use the type `OuterLex.token` (which is identical to the type `OuterParse.token`). There are also parsers for ML-expressions and ML-files.

Read More

The parser functions for the theory syntax are contained in the structure `OuterParse` defined in the file `Pure/Isar/outer_parse.ML`. The definition for tokens is in the file `Pure/Isar/outer_lex.ML`.

The structure `OuterLex` defines several kinds of tokens (for example `Ident` for identifiers, `Keyword` for keywords and `Command` for commands). Some token parsers take into account the kind of tokens.

For the examples below, we can generate a token list out of a string using the function `OuterSyntax.scan`, which we give below `Position.none` as argument since, at the moment, we are not interested in generating precise error messages. The following code

```
OuterSyntax.scan Position.none "hello world"
> [Token (...,(Ident, "hello"),...),
>  Token (...,(Space, " "),...),
>  Token (...,(Ident, "world"),...)]
```

produces three tokens where the first and the last are identifiers, since `"hello"` and `"world"` do not match any other syntactic category.[1] The second indicates a space.

Many parsing functions later on will require spaces, comments and the like to have already been filtered out. So from now on we are going to use the functions `filter` and `OuterLex.is_proper` do this. For example

[1] Note that because of a possible a bug in the PolyML runtime system the result is printed as `?`, instead of the tokens.

```
let
   val input = OuterSyntax.scan Position.none "hello world"
in
   filter OuterLex.is_proper input
end
> [Token (...,(Ident, "hello"), ...), Token (...,(Ident, "world"), ...)]
```

For convenience we define the function

```
fun filtered_input str =
       filter OuterLex.is_proper (OuterSyntax.scan Position.none str)
```

If we parse

```
filtered_input "inductive | for"
> [Token (...,(Command, "inductive"),...),
>  Token (...,(Keyword, "|"),...),
>  Token (...,(Keyword, "for"),...)]
```

we obtain a list consisting of only a command and two keyword tokens. If you want to see which keywords and commands are currently known, type in the following code (you might have to adjust the `print_depth` in order to see the complete list):

```
let
  val (keywords, commands) = OuterKeyword.get_lexicons ()
in
  (Scan.dest_lexicon commands, Scan.dest_lexicon keywords)
end
> (["}","{",...],["⇌","↽",...])
```

Now the parser `OuterParse.$$$` parses a single keyword. For example

```
let
  val input1 = filtered_input "where for"
  val input2 = filtered_input "| in"
in
  (OuterParse.$$$ "where" input1, OuterParse.$$$ "|" input2)
end
> (("where",...),("|",...))
```

Like before, we can sequentially connect parsers with `(op --)`. For example

```
let
  val input = filtered_input "| in"
in
  (OuterParse.$$$ "|" -- OuterParse.$$$ "in") input
end
> (("|","in"),[])
```

The parser `OuterParse.enum s p` parses a possibly empty list of items recognised by the parser `p`, where the items being parsed are separated by the string `s`. For example

```
let
  val input = filtered_input "in | in | in foo"
in
  (OuterParse.enum "|" (OuterParse.$$$ "in")) input
end
> (["in","in","in"],[...])
```

`OuterParse.enum1` works similarly, except that the parsed list must be non-empty. Note that we had to add a string `"foo"` at the end of the parsed string, otherwise the parser would have consumed all tokens and then failed with the exception `MORE`. Like in the previous section, we can avoid this exception using the wrapper `Scan.finite`. This time, however, we have to use the "stopper-token" `OuterLex.stopper`. We can write

```
let
  val input = filtered_input "in | in | in"
in
  Scan.finite OuterLex.stopper
          (OuterParse.enum "|" (OuterParse.$$$ "in")) input
end
> (["in","in","in"],[])
```

The following function will help to run examples.

```
fun parse p input = Scan.finite OuterLex.stopper (Scan.error p) input
```

The function `OuterParse.!!!` can be used to force termination of the parser in case of a dead end, just like `Scan.!!` (see previous section), except that the error message is fixed to be `"Outer syntax error"` with a relatively precise description of the failure. For example:

```
let
  val input = filtered_input "in |"
  val parse_bar_then_in = OuterParse.$$$ "|" -- OuterParse.$$$ "in"
in
  parse (OuterParse.!!! parse_bar_then_in) input
end
> Exception ERROR "Outer syntax error: keyword "|" expected,
> but keyword in was found" raised
```

Exercise 3.2.1. *(FIXME) A type-identifier, for example* `'a`*, is a token of kind* `Keyword`*. It can be parsed using the function* `OuterParse.type_ident`*.*

3.3 Positional Information

OuterParse.position

```
  OuterParse.position
```

3.4 Parsing Inner Syntax

```
let
  val input = OuterSyntax.scan Position.none "0"
in
  OuterParse.prop input
end




  OuterParse.opt_target
```

(FIXME funny output for a proposition)

```
  OuterParse.opt_target --
  OuterParse.fixes --
  OuterParse.for_fixes --
  Scan.optional (OuterParse.$$$ "where" |--
    OuterParse.!!! (OuterParse.enum1 "|" (SpecParse.opt_thm_name ":" --
OuterParse.prop))) []



 OuterSyntax.command
```

3.5 New Commands and Keyword Files

Often new commands, for example for providing new definitional principles, need to be implemented. While this is not difficult on the ML-level, new commands, in order to be useful, need to be recognised by ProofGeneral. This results in some subtle configuration issues, which we will explain in this section.

To keep things simple, let us start with a "silly" command that does nothing at all. We shall name this command **foobar**. On the ML-level it can be defined as

```
let
  val do_nothing = Scan.succeed (Toplevel.theory I)
  val kind = OuterKeyword.thy_decl
```

```
theory Command
imports Main
begin
ML {*
let
  val do_nothing = Scan.succeed (Toplevel.theory I)
  val kind = OuterKeyword.thy_decl
in
  OuterSyntax.command "foobar" "description of foobar" kind do_nothing
end
*}
end
```

Figure 3.1: The file *Command.thy* is necessary for generating a log file. This log file enables Isabelle to generate a keyword file containing the command **foobar**.

```
in
  OuterSyntax.command "foobar" "description of foobar" kind do_nothing
end
```

The crucial function *OuterSyntax.command* expects a name for the command, a short description, a kind indicator (which we will explain later on more thoroughly) and a parser producing a top-level transition function (its purpose will also explained later).

While this is everything we have to do on the ML-level, we need a keyword file that can be loaded by ProofGeneral. This is to enable ProofGeneral to recognise **foobar** as a command. Such a keyword file can be generated with the command-line:

```
$ isabelle keywords -k foobar some_log_files
```

The option *-k foobar* indicates which postfix the name of the keyword file will be assigned. In the case above the file will be named *isar-keywords-foobar.el*. As can be seen, this command requires log files to be present (in order to extract the keywords from them). To generate these log files, we first package the code above into a separate theory file named *Command.thy*, say—see Figure 3.1 for the complete code.

For our purposes it is sufficient to use the log files of the theories *Pure*, *HOL* and *Pure-ProofGeneral*, as well as the log file for the theory *Command.thy*, which contains the new **foobar**-command. If you target other logics besides HOL, such as Nominal or ZF, then you need to adapt the log files appropriately. *Pure* and *HOL* are usually compiled during the installation of Isabelle. So log files for them should be already available. If not, then they can be conveniently compiled with the help of the build-script from the Isabelle distribution

```
$ ./build -m "Pure"
$ ./build -m "HOL"
```

The *Pure-ProofGeneral* theory needs to be compiled with

```
$ ./build -m "Pure-ProofGeneral" "Pure"
```

For the theory *Command.thy*, we first create a "managed" subdirectory with

```
$ isabelle mkdir FoobarCommand
```

This generates a directory containing the files

```
./IsaMakefile
./FoobarCommand/ROOT.ML
./FoobarCommand/document
./FoobarCommand/document/root.tex
```

We need to copy the file *Command.thy* into the directory *FoobarCommand* and add the line

```
use_thy "Command";
```

to the file *./FoobarCommand/ROOT.ML*. We can now compile the theory by just typing

```
$ isabelle make
```

We created finally all the necessary log files. They are stored in the directory

```
~/.isabelle/heaps/Isabelle2008/polyml-5.2.1_x86-linux/log
```

or something similar depending on your Isabelle distribution and architecture. One quick way to assign a shell variable to this directory is by typing

```
$ ISABELLE_LOGS="$(isabelle getenv -b ISABELLE_OUTPUT)"/log
```

on the Unix prompt. The directory should include the files

```
Pure.gz
HOL.gz
Pure-ProofGeneral.gz
HOL-FoobarCommand.gz
```

From them we create the keyword files. Assuming the name of the directory is in *$ISABELLE_LOGS*, then the Unix command for creating the keyword file is:

```
$ isabelle keywords -k foobar
   $ISABELLE_LOGS/{Pure.gz,HOL.gz,Pure-ProofGeneral.gz,HOL-FoobarCommand.gz}
```

The result is the file *isar-keywords-foobar.el*. It should contain the string *foobar* twice (to see whether things went wrong, check that *grep foobar* on this file returns something non-empty). This keyword file needs to be copied into the directory *~/.isabelle/etc*. To make Isabelle aware of this keyword file, we have to start Isabelle with the option *-k foobar*, i.e.

```
$ isabelle -k foobar a_theory_file
```

If we now build a theory on top of *Command.thy*, then we can make use of the command **foobar**. Similarly with any other new command.

At the moment **foobar** is not very useful. Let us refine it a bit next by taking a proposition as argument and printing this proposition inside the tracing buffer.

The crucial part of a command is the function that determines the behaviour of the command. In the code above we used a "do-nothing"-function, which because of *Scan.succeed* does not parse any argument, but immediately returns the simple toplevel function *Toplevel.theory I*. We can replace this code by a function that first parses a proposition (using the parser *OuterParse.prop*), then prints out the tracing information (using a new top-level function *trace_top_lvl*) and finally does nothing. For this we can write

```
let
  fun trace_top_lvl str =
     Toplevel.theory (fn thy => (tracing str; thy))

  val trace_prop = OuterParse.prop >> trace_top_lvl

  val kind = OuterKeyword.thy_decl
in
  OuterSyntax.command "foobar" "traces a proposition" kind trace_prop
end
```

Now we can type

```
foobar "True ∧ False"
> "True ∧ False"
```

and see the proposition in the tracing buffer.

Note that so far we used *thy_decl* as the kind indicator for the command. This means that the command finishes as soon as the arguments are processed. Examples of this kind of commands are **definition** and **declare**. In other cases, commands are expected to parse some arguments, for example a proposition, and then "open up" a proof in order to prove the proposition (for example **lemma**) or prove some other properties (for example in **function**). To achieve this kind of behaviour, we have to use the kind indicator *thy_goal*.

Below we change **foobar** so that it takes a proposition as argument and then starts a proof in order to prove it. Therefore in Line 13 below, we set the kind indicator to *thy_goal*.

```
1  let
2    fun set_up_thm str ctxt =
3      let
4        val prop = Syntax.read_prop ctxt str
5      in
```

```
6       Proof.theorem_i NONE (K I) [[(prop,[])]] ctxt
7     end;
8
9   val prove_prop = OuterParse.prop >>
10      (fn str => Toplevel.print o
11                   Toplevel.local_theory_to_proof NONE (set_up_thm str))
12
13  val kind = OuterKeyword.thy_goal
14in
15  OuterSyntax.command "foobar" "proving a proposition" kind prove_prop
16end
```

The function *set_up_thm* takes a string (the proposition to be proved) and a context. The context is necessary in order to be able to use *Syntax.read_prop*, which converts a string into a proper proposition. In Line 6 the function *Proof.theorem_i* starts the proof for the proposition. Its argument *NONE* stands for a locale (which we chose to omit); the argument *(K I)* stands for a function that determines what should be done with the theorem once it is proved (we chose to just forget about it). In Lines 9 to 11 contain the parser for the proposition.

If we now type **foobar** *"True ∧ True"*, we obtain the following proof state:

foobar *"True ∧ True"*
goal (1 subgoal)
1. True ∧ True

and we can build the proof

foobar *"True ∧ True"*
apply*(rule conjI)*
apply*(rule TrueI)+*
done

(FIXME What does *Toplevel.theory Toplevel.print*?)

(FIXME read a name and show how to store theorems)

(FIXME possibly also read a locale)

Chapter 4

Tactical Reasoning

The main reason for descending to the ML-level of Isabelle is to be able to implement automatic proof procedures. Such proof procedures usually lessen considerably the burden of manual reasoning, for example, when introducing new definition. These proof procedures are centred around refining a goal state using tactics. This is similar to the *apply*-style reasoning at the user level, where goals are modified in a sequence of proof steps until all of them are solved.

4.1 Tactical Reasoning

To see how tactics work, let us first transcribe a simple *apply*-style proof into ML. Consider the following proof.

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

This proof translates to the following ML-code.

```
let
  val ctxt = @{context}
  val goal = @{prop "P ∨ Q ⟹ Q ∨ P"}
in
  Goal.prove ctxt ["P", "Q"] [] goal
   (fn _ =>
      etac @{thm disjE} 1
      THEN rtac @{thm disjI2} 1
      THEN atac 1
      THEN rtac @{thm disjI1} 1
      THEN atac 1)
end
> ?P ∨ ?Q ⟹ ?Q ∨ ?P
```

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 (in our case `P` and `Q`). The `tac` is the tactic that proves the goal; it can make use of the local assumptions (there are none in this example). The functions `etac`, `rtac` and `atac` correspond to `erule`, `rule` and `assumption`, respectively. The operator `THEN` strings tactics together.

Read More

To learn more about the function `Goal.prove` see [Impl. Man., Sec. 4.3] and the file `Pure/goal.ML`. For more on the internals of goals see [Impl. Man., Sec. 3.1].

Note that we used antiquotations for referencing the theorems. We could also just have written `etac disjE 1` and so on, but this is considered bad style. The reason is that the binding for `disjE` can be re-assigned by the user and thus one does not have complete control over which theorem is actually applied. This problem is nicely prevented by using antiquotations, because then the theorems are fixed statically at compile time.

During the development of automatic proof procedures, it will often be necessary to test a tactic on examples. This can be conveniently done with the command **apply**`(tactic {* ... *})`. Consider the following sequence of tactics

```
val foo_tac =
      (etac @{thm disjE} 1
       THEN rtac @{thm disjI2} 1
       THEN atac 1
       THEN rtac @{thm disjI1} 1
       THEN atac 1)
```

and the Isabelle proof:

lemma `"P ∨ Q ⟹ Q ∨ P"`
apply`(tactic {* foo_tac *})`
done

The apply-step applies the `foo_tac`. Inside `tactic {* ... *}` we can write any function that returns a tactic.

As can be seen, each tactic in `foo_tac` has a hard-coded number that stands for the subgoal analysed by the tactic. In our case, we only focus on the first subgoal. This is sometimes wanted, but usually not. To avoid the explicit numbering in the tactic, you can write

```
val foo_tac' =
      (etac @{thm disjE}
       THEN' rtac @{thm disjI2}
       THEN' atac
       THEN' rtac @{thm disjI1}
       THEN' atac)
```

and then give the number for the subgoal explicitly when the tactic is called. So in the next proof we discharge first the second subgoal, and then the first.

```
lemma "P1 ∨ Q1 ⟹ Q1 ∨ P1"
  and "P2 ∨ Q2 ⟹ Q2 ∨ P2"
apply(tactic {* foo_tac' 2 *})
apply(tactic {* foo_tac' 1 *})
done
```

The tactic `foo_tac` is very specific for analysing goals of the form $P \vee Q \Longrightarrow Q \vee P$. If the goal is not of this form, then `foo_tac` throws the error message about an empty result sequence—meaning the tactic failed. The reason for this message is that tactics are functions that map a goal state to a (lazy) sequence of successor states, hence the type of a tactic is

```
type tactic = thm -> thm Seq.seq
```

Consequently, if a tactic fails, then it returns the empty sequence. This is by the way the default behaviour for a failing tactic; tactics should not raise exceptions.

In the following example there are two possibilities for how to apply the tactic.

```
lemma "⟦P ∨ Q; P ∨ Q⟧ ⟹ Q ∨ P"
apply(tactic {* foo_tac' 1 *})
back
done
```

The application of the tactic results in a sequence of two possible proofs. The Isabelle command **back** allows us to explore both possibilities.

Read More

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 functions instead. See `Pure/tactic.ML` and `Pure/tctical.ML` for the code; see also Chapters 3 and 4 in the old Isabelle Reference Manual.

4.2 Basic Tactics

```
lemma shows "False ⟹ False"
apply(tactic {* atac 1 *})
done

lemma shows "True ∧ True"
apply(tactic {* rtac @{thm conjI} 1 *})

 1. True
 2. True


lemma
  shows "Foo"
  and "True ∧ True"
apply(tactic {* rtac @{thm conjI} 2 *})

 1. Foo
 2. True
 3. True
```

lemma shows *"False ∧ False ⟹ False"*
apply*(tactic {* etac @{thm conjE} 1 *})*

```
1. ⟦False; False⟧ ⟹ False
```

lemma shows *"False ∧ True ⟹ False"*
apply*(tactic {* dtac @{thm conjunct2} 1 *})*

```
1. True ⟹ False
```

similarly *ftac*

diagnostics

lemma shows *"True ⟹ False"*
apply*(tactic {* print_tac "foo message" *})*

PRIMITIVE? *SUBGOAL* see page 32 in ref

all_tac no_tac

4.3 Operations on Tactics

THEN

lemma shows *"(True ∧ True) ∧ False"*
apply*(tactic {* (rtac @{thm conjI} 1) THEN (rtac @{thm conjI} 1) *})*

```
1. True
2. True
3. False
```

lemma shows *"True ∧ False"*
apply*(tactic {* (rtac @{thm disjI1} 1) ORELSE (rtac @{thm conjI} 1) *})*

```
1. True
2. False
```

EVERY REPEAT SUBPROOF

rewrite_goals_tac cut_facts_tac ObjectLogic.full_atomize_tac ObjectLogic.rulify_tac resolve_tac

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

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.

Chapter 5

How to write a definitional package

5.1 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 §5.2, we will illustrate the "manual" definition of inductive predicates using some examples. Starting from these examples, we will describe in §5.3 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 §5.4. 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 §**??**.

5.2 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 §5.3. This description will serve as a basis for the actual implementation to be developed in §5.4 – §**??**. 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 ⟹ in the above introduction rules have to be converted to object-level implications ⟶. Moreover, we use object-level universal quantifiers ∀ rather than meta-level universal quantifiers ⋀ 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 as-

sumption *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 $\forall$ and $\longrightarrow$, 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:

$$\begin{array}{l} even\ 0 \\ odd\ m \Longrightarrow even\ (Suc\ m) \\ even\ m \Longrightarrow odd\ (Suc\ m) \end{array}$$

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 ∀ and ⟶ 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
```

5.3 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

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

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

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

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

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

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

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

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

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

5.4 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:
    ((Binding.binding * typ) * mixfix) list ->                    predicates
    (Binding.binding * typ) list ->                               parameters
    (Attrib.binding * term) list ->                                    rules
    local_theory -> (thm list * thm list) * local_theory
  val add_inductive:
    (Binding.binding * string option * mixfix) list ->            predicates
    (Binding.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 `Binding.binding`. Strings can be turned into elements of the type `Binding.binding` using the function

```
Binding.name : string -> Binding.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 `Binding.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:
  (Binding.binding * string option * mixfix) list ->                variables
  (Attrib.binding * string list) list list ->                         rules
  local_theory ->
  (((Binding.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
  [(Binding.name "trcl", NONE, NoSyn),
   (Binding.name "r", SOME "'a ⇒ 'a ⇒ bool", NoSyn)]
  [[((Binding.name "base", []), ["trcl r x x"])],
   [((Binding.name "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 ("==>", ...) $ ... $ ...)))])]),
>  ...)
> : (((Binding.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`:

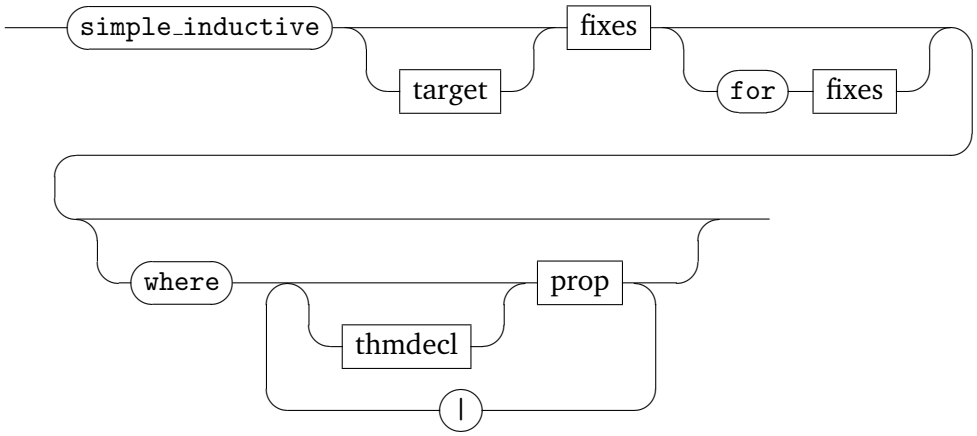
```
locale rel =
  fixes r :: "'a ⇒ 'a ⇒ bool"

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:

```
|| : ('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
```



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`:

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

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

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*:

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 ->
  (Binding.binding * string option * mixfix) list * token list
for_fixes: token list ->
  (Binding.binding * string option * mixfix) list * token list
!!! : (token list -> 'a) -> token list -> 'a

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

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:

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.

(FIXME : needs to be adjusted to new parser type)

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 in a method.

Solution: This can be achieved using named theorem lists.

Named theorem lists can be set up using the code

```
structure FooRules = NamedThmsFun (
  val name = "foo"
  val description = "Rules for foo");
```

and the command

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 for 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 we can declare three lemmas to be of the kind *foo* by:

lemma *rule1[foo]: "A"* **sorry**
lemma *rule2[foo]: "B"* **sorry**
lemma *rule3[foo]: "C"* **sorry**

and undeclare the first one by:

declare *rule1[foo del]*

and query the remaining ones with:

thm *foo*
> ?C
> ?B

On the ML-level the rules marked with *foo* an be retrieved using the function *FooRules.get*:

```
FooRules.get @{context}
> ["?C","?B"]
```

For more information see Pure/Tools/named_thms.ML and also the recipe in Section A.6 about storing arbitrary data. **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

A.3 Useful Document Antiquotations

Problem: How to keep your ML-code inside a document synchronised with the actual code?

Solution: This can be achieved using document antiquotations.

Document antiquotations can be used for ensuring consistent type-setting of various entities in a document. They can also be used for sophisticated LaTeX-hacking. If you type *Ctrl-c Ctrl-a h A* inside ProofGeneral, you obtain a list of all currently available document antiquotations and their options.

Below we give the code for two additional antiquotations that can be used to typeset ML-code and also to check whether the given code actually compiles. This provides a sanity check for the code and also allows one to keep documents in sync with other code, for example Isabelle.

We first describe the antiquotation *ML_checked* with the syntax:

```
@{ML_checked "a_piece_of_code"}
```

The code is checked by sending the ML-expression `"val _ = a_piece_of_code"` to the ML-compiler (i.e. the function `ML_Context.eval_in` in Line 4 below). The complete code of the antiquotation is as follows:

```
1 fun ml_val code_txt = "val _ = " ^ code_txt
2
3 fun output_ml src ctxt code_txt =
4   (ML_Context.eval_in (SOME ctxt) false Position.none (ml_val code_txt);
5   ThyOutput.output_list (fn _ => fn s => Pretty.str s) src ctxt
6                                             (space_explode "\n" code_txt))
7
8 val _ = ThyOutput.add_commands
9  [("ML_checked", ThyOutput.args (Scan.lift Args.name) output_ml)]
```

Note that the parser `(Scan.lift Args.name)` in line 9 parses a string, in this case the code given as argument. As mentioned before, this argument is sent to the ML-compiler in the line 4 using the function `ml_val`, which constructs the appropriate ML-expression. If the code is "approved" by the compiler, then the output function `ThyOutput.output_list (fn _ => fn s => Pretty.str s)` in the next line pretty prints the code. This function expects that the code is a list of strings where each string correspond to a line in the output. Therefore the use of `(space_explode "\n" txt)` which produces this list according to linebreaks. There are a number of options for antiquotations that are observed by `ThyOutput.output_list` when printing the code (including `[display]`, `[quotes]` and `[source]`).

For more information about options of antiquotations see [Isar Ref. Man., Sec. 5.2]). **Read More**

Since we used the argument `Position.none`, the compiler cannot give specific information about the line number, in case an error is detected. We can improve the code above slightly by writing

```
1 fun output_ml src ctxt (code_txt,pos) =
2   (ML_Context.eval_in (SOME ctxt) false pos (ml_val code_txt);
3   ThyOutput.output_list (fn _ => fn s => Pretty.str s) src ctxt
4                                             (space_explode "\n" code_txt))
5
6 val _ = ThyOutput.add_commands
7  [("ML_checked", ThyOutput.args
8        (Scan.lift (OuterParse.position Args.name)) output_ml)]
```

where in Lines 1 and 2 the positional information is properly treated.

(FIXME: say something about OuterParse.position)

We can now write in a document `@{ML_checked "2 + 3"}` in order to obtain `2 + 3` and be sure that this code compiles until somebody changes the definition of `(op +)`.

The second antiquotation we describe extends the first by allowing also to give a pattern that specifies what the result of the ML-code should be and to check the consistency of the actual result with the given pattern. For this we are going to implement the antiquotation

```
@{ML_resp "a_piece_of_code" "pattern"}
```

To add some convenience and also to deal with large outputs, the user can give a partial specification by giving the abbreviation `"..."`. For example `(...,...)` for a pair.
Whereas in the antiquotation `@{ML_checked "piece_of_code"}` above, we have sent the expression `"val _ = piece_of_code"` to the compiler, in the second the wildcard `_` we will be replaced by a proper pattern. To do this we need to replace the `"..."` by `"_"` before sending the code to the compiler. The following function will do this:

```
fun ml_pat (code_txt, pat) =
   let val pat' =
         implode (map (fn "..." => "_" | s => s) (Symbol.explode pat))
   in
     "val " ^ pat' ^ " = " ^ code_txt
   end
```

Next we like to add a response indicator to the result using:

```
fun add_resp_indicator pat =
  map (fn s => "> " ^ s) (space_explode "\n" pat)
```

The rest of the code of the antiquotation is

```
fun output_ml_resp src ctxt ((code_txt,pat),pos) =
  (ML_Context.eval_in (SOME ctxt) false pos (ml_pat (code_txt,pat));
   let
     val output = (space_explode "\n" code_txt) @ (add_resp_indicator pat)
   in
     ThyOutput.output_list (fn _ => fn s => Pretty.str s) src ctxt output
   end)

val _ = ThyOutput.add_commands
 [("ML_resp",
     ThyOutput.args
      (Scan.lift (OuterParse.position (Args.name -- Args.name)))
        output_ml_resp)]
```

This extended antiquotation allows us to write

```
@{ML_resp [display] "true andalso false" "false"}
```

to obtain

```
true andalso false
> false
```

or

```
@{ML_resp [display] "let val i = 3 in (i * i,"foo") end" "(9,...)"}
```

to obtain

```
let val i = 3 in (i * i,"foo") end
> (9,...)
```

In both cases, the check by the compiler ensures that code and result match. A limitation of this antiquotation, however, is that the hints can only be given in case they can be constructed as a pattern. This excludes values that are abstract datatypes, like theorems or cterms.

A.4 Restricting the Runtime of a Function

Problem: Your tool should run only a specified amount of time.

Solution: This can be achieved using the function `timeLimit`.

Assume you defined the Ackermann function:

```
fun ackermann (0, n) = n + 1
  | ackermann (m, 0) = ackermann (m - 1, 1)
  | ackermann (m, n) = ackermann (m - 1, ackermann (m, n - 1))
```

Now the call

```
ackermann (4, 12)
> ...
```

takes a bit of time before it finishes. To avoid this, the call can be encapsulated in a time limit of five seconds. For this you have to write:

```
TimeLimit.timeLimit (Time.fromSeconds 5) ackermann (4, 12)
  handle TimeOut => ~1
> ~1
```

where `TimeOut` is the exception raised when the time limit is reached.

Note that `timeLimit` is only meaningful when you use PolyML, because PolyML has a rich infrastructure for multithreading programming on which `timeLimit` relies.

Read More *The function* `timeLimit` *is defined in the structure* `TimeLimit` *which can be found in the file* `Pure/ML-Systems/multithreading_polyml.ML`.

A.5 Configuration Options

Problem: You would like to enhance your tool with options that can be changed by the user without having to resort to the ML-level.

Solution: This can be achieved using configuration values.

Assume you want to control three values, namely *bval* containing a boolean, *ival* containing an integer and *sval* containing a string. These values can be declared on the ML-level with

```
val (bval, setup_bval) = Attrib.config_bool "bval" false
val (ival, setup_ival) = Attrib.config_int "ival" 0
val (sval, setup_sval) = Attrib.config_string "sval" "some string"
```

where each value needs to be given a default. To enable these values, they need to be set up by

setup *{* setup_bval *}*
setup *{* setup_ival *}*

or on the ML-level

```
setup_sval @{theory}
```

The user can now manipulate the values from within Isabelle with the command

declare *[[bval = true, ival = 3]]*

On the ML-level these values can be retrieved using the function *Config.get*:

```
Config.get @{context} bval
> true
```

```
Config.get @{context} ival
> 3
```

The function *Config.put* manipulates the values. For example

```
Config.put sval "foo" @{context}; Config.get @{context} sval
> foo
```

The same can be achived using the command **setup**.

setup *{* Config.put_thy sval "bar" *}*

The retrival of this value yields now

```
Config.get @{context} sval
> "bar"
```

We can apply a function to a value using `Config.map`. For example incrementing `ival` can be done by

```
let
  val ctxt = Config.map ival (fn i => i + 1) @{context}
in
  Config.get ctxt ival
end
> 4
```

For more information see `Pure/Isar/attrib.ML` *and* `Pure/config.ML`. **Read More**

There are many good reasons to control parameters in this way. One is that it avoid global references, which cause many headaches with the multithreaded execution of Isabelle.

A.6 Storing Data

Problem: Your tool needs to manage data.

Solution: This can be achieved using a generic data slot.

Every generic data slot may keep data of any kind which is stored in the context.

```
local

structure Data = GenericDataFun
 ( type T = int Symtab.table
   val empty = Symtab.empty
   val extend = I
   fun merge _ = Symtab.merge (K true)
 )

in
 val lookup = Symtab.lookup o Data.get
 fun update k v = Data.map (Symtab.update (k, v))
end
```

setup `{* Context.theory_map (update "foo" 1) *}`

```
lookup (Context.Proof @{context}) "foo"
> SOME 1
```

alternatives: TheoryDataFun, ProofDataFun Code: Pure/context.ML

A.7 Using an External Solver

Problem: You want to use an external solver, because the solver might be more efficient for deciding a certain class of formulae than Isabelle tactics.

Solution: The easiest way to do this is by implementing an oracle. We will also construct proofs inside Isabelle by using the results produced by the oracle.

Read More

A short introduction to oracles can be found in [isar-ref: no suitable label for section 3.11]. A simple example is given in `FOL/ex/IffOracle`. (TODO: add more references to the code)

For our explanation here, we will use the *`metis`* prover for proving propositional formulae. The general method will be roughly as follows: Given a goal *`G`*, we transform it into the syntactical respresentation of *`metis`*, build the CNF of the negated formula and then let metis search for a refutation. *`Metis`* will either return the proved goal or raise an exception meaning that it was unable to prove the goal (FIXME: is this so?).

The translation function from Isabelle propositions into formulae of *`metis`* is as follows:

```
fun trans t =
  (case t of
    @{term Trueprop} $ t => trans t
  | @{term True} => Metis.Formula.True
  | @{term False} => Metis.Formula.False
  | @{term Not} $ t => Metis.Formula.Not (trans t)
  | @{term "op &"} $ t1 $ t2 => Metis.Formula.And (trans t1, trans t2)
  | @{term "op |"} $ t1 $ t2 => Metis.Formula.Or (trans t1, trans t2)
  | @{term "op -->"} $ t1 $ t2 => Metis.Formula.Imp (trans t1, trans t2)
  | @{term "op = :: bool => bool => bool"} $ t1 $ t2 =>
                                    Metis.Formula.Iff (trans t1, trans t2)
  | Free (n, @{typ bool}) => Metis.Formula.Atom (n, [])
  | _ => error "inacceptable term")
```

An example is as follows:

```
trans @{prop "A ∧ B"}
> Metis.Formula.And
>         (Metis.Formula.Atom ("A", []), Metis.Formula.Atom ("B", []))
```

The next function computes the conjunctive-normal-form.

```
1 fun make_cnfs fm =
2       fm |> Metis.Formula.Not
3          |> Metis.Normalize.cnf
4          |> map Metis.Formula.stripConj
5          |> map (map Metis.Formula.stripDisj)
6          |> map (map (map Metis.Literal.fromFormula))
7          |> map (map Metis.LiteralSet.fromList)
8          |> map (map Metis.Thm.axiom)
```

(FIXME: Is there a deep reason why Metis.Normalize.cnf returns a list?)
(FIXME: What does Line 8 do?)
(FIXME: Can this code be improved?)
Setting up the resolution.

```
fun refute cls =
 let val result =
       Metis.Resolution.loop
         (Metis.Resolution.new Metis.Resolution.default cls)
 in
   (case result of
      Metis.Resolution.Contradiction _ => true
    | Metis.Resolution.Satisfiable _ => false)
 end
```

Stringing the functions together.

```
fun solve f = List.all refute (make_cnfs f)
```

Setting up the oracle

```
fun prop_dp (thy, t) =
        if solve (trans t) then (Thm.cterm_of thy t)
        else error "Proof failed."
```

oracle *prop_oracle = prop_dp*

(FIXME: What does oracle do?)

```
fun prop_oracle_tac ctxt =
  SUBGOAL (fn (goal, i) =>
    (case (try prop_oracle (ProofContext.theory_of ctxt, goal)) of
      SOME thm => rtac thm i
    | NONE => no_tac))
```

(FIXME: The oracle returns a *cterm*. How is it possible that I can apply this cterm with *rtac*?)

method_setup *prop_oracle = {*
* Method.ctxt_args (fn ctxt => Method.SIMPLE_METHOD' (prop_oracle_tac ctxt))*
} "Oracle-based decision procedure for propositional logic"

(FIXME What does *Method.SIMPLE_METHOD'* do?)

lemma *test: "p ∨ ¬p"*
  by *prop_oracle*

(FIXME: say something about what the proof of the oracle is)

```
Thm.proof_of @{thm test}
> ???
```

lemma *"((p ⟶ q) ⟶ p) ⟶ p"*
  by *prop_oracle*

lemma *"∀x::nat. x ≥ 0"*
  sorry

(FIXME: proof reconstruction)

For communication with external programs, there are the primitives *system* and *system_out*, the latter of which captures the invoked program's output. For simplicity, here, we will use metis, an external solver included in the Isabelle destribution. Since it is written in ML, we can call it directly without the detour of invoking an external program.

Appendix B

Solutions to Most Exercises

Solution for Exercise 2.5.1.

```
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.

```
fun make_sum t1 t2 =
    HOLogic.mk_nat (HOLogic.dest_nat t1 + HOLogic.dest_nat t2)
```

Solution for Exercise 3.1.1.

```
val any = Scan.one (Symbol.not_eof);

val scan_cmt =
  let
    val begin_cmt = Scan.this_string "(*"
    val end_cmt = Scan.this_string "*)"
  in
   begin_cmt |-- Scan.repeat (Scan.unless end_cmt any) --| end_cmt
    >> (enclose "(**" "**)" o implode)
  end

val scan_all =
  Scan.finite Symbol.stopper (Scan.repeat (scan_cmt || any))
    >> implode #> fst
```

By using `#> fst` in the last line, the function `scan_all` retruns a string, instead of the pair a parser would normally return. For example:

```
let
   val input1 = (explode "foo bar")
   val input2 = (explode "foo (*test*) bar (*test*)")
 in
   (scan_all input1, scan_all input2)
 end
> ("foo bar","foo (**test**) bar (**test**)")
```

Appendix C

Comments for Authors

- The Cookbook can be compiled on the command-line with:

  ```
  $ isabelle make
  ```

  You very likely need a recent snapshot of Isabelle in order to compile the Cookbook. Some parts of the Cookbook also rely on compilation with PolyML.

- You can include references to other Isabelle manuals using the reference names from those manuals. To do this the following four LaTeX commands are defined:

	Chapters	Sections
Implementation Manual	`\ichcite{...}`	`\isccite{...}`
Isar Reference Manual	`\rchcite{...}`	`\rsccite{...}`

  So `\ichcite{ch:logic}` yields a reference for the chapter about logic in the implementation manual, namely [Impl. Man., Ch. 2].

- There are various document antiquotations defined for the Cookbook. They allow to check the written text against the current Isabelle code and also allow to show responses of the ML-compiler. Therefore authors are strongly encouraged to use antiquotations wherever appropriate.

  The following antiquotations are defined:

  - `@{ML "expr" for vars in structs}` should be used for displaying any ML-expression, because the antiquotation checks whether the expression is valid ML-code. The `for`- and `in`-arguments are optional. The former is used for evaluating open expressions by giving a list of free variables. The latter is used to indicate in which structure or structures the ML-expression should be evaluated. Examples are:

`@{ML "1 + 3"}`		`1 + 3`
`@{ML "a + b" for a b}`	produce	`a + b`
`@{ML Ident in OuterLex}`		`Ident`

- *@{ML_response "expr" "pat"}* should be used to display ML-expressions and their response. The first expression is checked like in the antiquotation *@{ML "expr"}*; the second is a pattern that specifies the result the first expression produces. This pattern can contain *"..."* for parts that you like to omit. The response of the first expression will be checked against this pattern. Examples are:

  ```
  @{ML_response "1+2" "3"}
  @{ML_response "(1+2,3)" "(3,...)"}
  ```

  which produce respectively

  ```
  1+2                 (1+2,3)
  > 3                 > (3,...)
  ```

  Note that this antiquotation can only be used when the result can be constructed: it does not work when the code produces an exception or returns an abstract datatype (like *thm* or *cterm*).

- *@{ML_response_fake "expr" "pat"}* works just like the antiquotation *@{ML_response "expr" "pat"}* above, except that the result-specification is not checked. Use this antiquotation when the result cannot be constructed or the code generates an exception. Examples are:

  ```
  @{ML_response_fake   "cterm_of @{theory} @{term \"a + b = c\"}"
                       "a + b = c"}
  @{ML_response_fake   "($$ \"x\") (explode \"world\")"
                       "Exception FAIL raised"}
  ```

  which produce respectively

  ```
  cterm_of @{theory} @{term "a + b = c"}
  > a + b = c
  ($$ "x") (explode "world")
  > Exception FAIL raised
  ```

  This output mimics to some extend what the user sees when running the code.

- *@{ML_response_fake_both "expr" "pat"}* can be used to show erroneous code. Neither the code nor the response will be checked. An example is:

  ```
  @{ML_response_fake_both   "@{cterm \"1 + True\"}"
                            "Type unification failed ..."}
  ```

- *@{ML_file "name"}* should be used when referring to a file. It checks whether the file exists. An example is

  ```
  @{ML_file "Pure/General/basics.ML"}
  ```

The listed antiquotations honour options including *[display]* and *[quotes]*. For example

`@{ML [quotes] "\"foo\" ^ \"bar\""}` produces `"foobar"`

whereas

`@{ML "\"foo\" ^ \"bar\""}` produces only `foobar`

- Functions and value bindings cannot be defined inside antiquotations; they need to be included inside **ML** `{* ... *}` environments. In this way they are also checked by the compiler. Some LaTeX-hack in the Cookbook, however, ensures that the environment markers are not printed.
- Line numbers can be printed using **ML** `%linenumbers {* ... *}` for ML-code or **lemma** `%linenumbers ...` for proofs.

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.