The Isabelle Programming Tutorial (draft)

by Christian Urban with contributions from:

Stefan Berghofer
Sascha Böhme
Jeremy Dawson
Alexander Krauss

March 23, 2009

Contents

Chapter 1

Introduction

If your next project requires you to program on the ML-level of Isabelle, then this tutorial is for you. It will guide you through the first steps of Isabelle programming, and also explain tricks of the trade. The best way to get to know the ML-level of Isabelle is by experimenting with the many code examples included in the tutorial. The code 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 tutorial, please feel free—you are most welcome! The tutorial is meant to be gentle and comprehensive. To achieve this we need your feedback.

1.1 Intended Audience and Prior Knowledge

This tutorial 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 Isabelle/Isar 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 Many parts of this manual are outdated now, but some parts, particularly the chapters on tactics, are still useful.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 provides specification material (like grammars, examples and so on) about Isar and its implementation. It is currently in the process of being updated.

Then of course there is:

The code. Which is 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 and learn from it. The UNIX command *grep -R* is often your best friend while programming with Isabelle.

1.3 Typographic Conventions

All ML-code in this tutorial is typeset in shaded boxes, like the following ML-expression:

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

These boxes correspond to how code can be processed inside the interactive environment of Isabelle. It is therefore easy to experiment with what is displayed. However, for better readability we will drop the enclosing **ML** *{* ... *}* and just write:

```
3 + 4
```

Whenever appropriate we also show the response the code generates when evaluated. This response is prefixed with a *">"*, like:

```
3 + 4
> 7
```

The user-level commands of Isabelle (i.e., the non-ML code) are written in **bold face** (e.g., **lemma**, **apply**, **foobar** and so on). We use *$ ...* to indicate that a command needs to be run in a Unix-shell, for example:

```
$ grep -R ThyOutput *
```

Pointers to further information and Isabelle files are typeset in *italic* and highlighted as follows:

> ***Read More***
> *Further information or pointers to files.*

The pointers to Isabelle files are hyperlinked to the tip of the Mercurial repository of Isabelle at http://isabelle.in.tum.de/repos/isabelle/.

A few exercises are scattered around the text. Their solutions are given in Appendix B. Of course, you learn most, if you first try to solve the exercises on your own, and then look at the solutions.

1.4 Acknowledgements

Financial support for this tutorial was provided by the German Research Council (DFG) under grant number URB 165/5-1. The following people contributed to the text:

- **Stefan Berghofer** wrote nearly all of the ML-code of the **simple_inductive**-package and the code for the *chunk*-antiquotation. He also wrote the first version of the chapter describing the package and has been helpful *beyond measure* with answering questions about Isabelle.
- **Sascha Böhme** contributed the recipes in A.2, A.4, A.5, A.6 and A.7. He also wrote section 4.6 and helped with recipe A.3.
- **Jeremy Dawson** wrote the first version of the chapter about parsing.
- **Armin Heller** helped with recipe A.8.
- **Alexander Krauss** wrote the first version of the "first-steps" chapter and also contributed the material on `NamedThmsFun`.

Please let me know of any omissions. Responsibility for any remaining errors lies with me.

This document is still in the process of being written! All of the text is still under construction. Sections and chapters that are under heavy construction are marked with TBD.

This document was compiled with:
Isabelle repository version

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 given in this chapter, we assume you are working inside the theory starting with

theory FirstSteps
imports Main
begin
...

We also generally assume you are working with HOL. The given examples might need to be adapted slightly if you work in a different logic.

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 in the Introduction, we will drop the **ML** `{* ... *}` scaffolding whenever we show code. The lines prefixed with `">"` are not part of the code, rather they indicate what the response is when the code is evaluated.

Once a portion of code is relatively stable, you usually want to export it to a separate ML-file. Such files can then be included in a theory by using the **uses**-command 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 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 with 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*.

You can print out error messages with the function *error*; for example:

```
if 0=1 then true else (error "foo")
> Exception- ERROR "foo" raised
> At command "ML".
```

(FIXME *Toplevel.debug Toplevel.profiling*)

Most often you want to inspect data of type *term*, *cterm* or *thm*. Isabelle contains elaborate pretty-printing functions for printing them, but for quick-and-dirty solutions they are far too unwieldy. A simple way to transform a term into a string is to use the function *Syntax.string_of_term*.

```
Syntax.string_of_term @{context} @{term "1::nat"}
> "\^E\^Fterm\^E\^E\^Fconst\^Fname=HOL.one_class.one\^E1\^E\^F\^E\^E\^F\^E"
```

This produces a string with some additional information encoded in it. The string can be properly printed by using the function *warning*.

```
warning (Syntax.string_of_term @{context} @{term "1::nat"})
> "1"
```

A *cterm* can be transformed into a string by the following function.

```
fun str_of_cterm ctxt t =
   Syntax.string_of_term ctxt (term_of t)
```

In this example the function *term_of* extracts the *term* from a *cterm*. If there are more than one *cterm*s to be printed, you can use the function *commas* to separate them.

```
fun str_of_cterms ctxt ts =
   commas (map (str_of_cterm ctxt) ts)
```

The easiest way to get the string of a theorem is to transform it into a *cterm* using the function *crep_thm*.

```
fun str_of_thm_raw ctxt thm =
  str_of_cterm ctxt (#prop (crep_thm thm))
```

Theorems also include schematic variables, such as *?P*, *?Q* and so on.

```
warning (str_of_thm_raw @{context} @{thm conjI})
> ⟦?P; ?Q⟧ ⟹ ?P ∧ ?Q
```

In order to improve the readability of theorems we convert these schematic variables into free variables using the function *Variable.import_thms*.

```
fun no_vars ctxt thm =
let
  val ((_, [thm']), _) = Variable.import_thms true [thm] ctxt
in
  thm'
end

fun str_of_thm ctxt thm =
  str_of_cterm ctxt (#prop (crep_thm (no_vars ctxt thm)))
```

Theorem *conjI* looks now as follows

```
warning (str_of_thm @{context} @{thm conjI})
> ⟦P; Q⟧ ⟹ P ∧ Q
```

Again the function *commas* helps with printing more than one theorem.

```
fun str_of_thms ctxt thms =
  commas (map (str_of_thm ctxt) thms)

fun str_of_thms_raw ctxt thms =
  commas (map (str_of_thm_raw ctxt) thms)
```

2.3 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 simplest combinator is *I*, which is just the identity function defined as

```
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. As a result, `K` defines a constant function always 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 its argument `x` by 5. It proceeds 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). The reverse application allows you to read what happens in a top-down manner. This kind of coding should also be familiar, if you have been exposed to 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!

In the context of Isabelle, a "real world" example for a function written in the waterfall fashion might be the following code:

```
1 fun apply_fresh_args pred ctxt =
2   pred |> fastype_of
3        |> binder_types
4        |> map (pair "z")
5        |> Variable.variant_frees ctxt [pred]
6        |> map Free
7        |> (curry list_comb) pred
```

This code extracts the argument types of a given predicate and then generates for each argument type a distinct variable; finally it applies the generated variables to the predicate. For example

```
apply_fresh_args @{term "P::nat ⇒ int ⇒ unit ⇒ bool"} @{context}
 |> Syntax.string_of_term @{context}
 |> warning
> P z za zb
```

You can read off this behaviour from how `apply_fresh_args` is coded: in Line 2, the function `fastype_of` calculates the type of the predicate; `binder_types` in the next line produces the list of argument types (in the case above the list `[nat, int, unit]`); Line 4 pairs up each type with the string `z`; the function `variant_frees` generates for each `z` a unique name avoiding the given `pred`; the list of name-type pairs is turned into a list of variable terms in Line 6, which in the last line is applied by the function `list_comb` to the predicate. We have to use the function `curry`, because `list_comb` expects the predicate and the variables list as a pair.

The combinator `#>` is the reverse function composition. It can be used to define the following function

```
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 you 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 you 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 `1` and then by `2`. 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, you can use the next combinator.

The combinator `` ` `` (a backtick) 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 increments `x`, but also keeps `x`. The intermediate result is therefore the pair `(x + 1, x)`. After that, the function 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 roundabout 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 application. Recall also that the related reverse function composition is `#>`. In fact all the combinators `|->`, `|>>` and `||>` described above have related combinators for function composition, namely `#->`, `#>>` and `##>`. Using `#->`, for example, 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)

> ***Read More***
> *The most frequently used combinators are defined in the files* `Pure/library.ML` *and* `Pure/General/basics.ML`*. Also [Impl. Man., Sec. B.1] contains further information about combinators.*

2.4 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 using the function `Context.theory_name`.

Note, however, that antiquotations are statically linked, 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 *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 ...}` for a single theorem

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

and `@{thms ...}` for more than one

```
@{thms conj_ac}
> (?P ∧ ?Q) = (?Q ∧ ?P)
> (?P ∧ ?Q ∧ ?R) = (?Q ∧ ?P ∧ ?R)
> ((?P ∧ ?Q) ∧ ?R) = (?P ∧ ?Q ∧ ?R)
```

You can also refer to the current simpset. To illustrate this we implement the function that extracts the theorem names stored in a simpset.

```
fun get_thm_names_from_ss simpset =
let
  val {simps,...} = MetaSimplifier.dest_ss simpset
in
  map #1 simps
end
```

The function `dest_ss` returns a record containing all information stored in the simpset. We are only interested in the names of the simp-rules. So now you can feed in the current simpset into this function. The simpset can be referred to using the antiquotation `@{simpset}`.

```
get_thm_names_from_ss @{simpset}
> ["Nat.of_nat_eq_id", "Int.of_int_eq_id", "Nat.One_nat_def", ...]
```

Again, this way of referencing simpsets makes you independent from additions of lemmas to the simpset by the user that potentially cause loops.

On the ML-level of Isabelle, you often have to work with qualified names; these are strings with some additional information, such as positional information and qualifiers. Such bindings can be generated with the antiquotation `@{binding ...}`.

```
@{binding "name"}
> name
```

An example where a binding is needed is the function `define`. Below, this function is used to define the constant `TrueConj` as the conjunction `True ∧ True`.

```
local_setup {*
  snd o LocalTheory.define Thm.internalK
    ((@{binding "TrueConj"}, NoSyn),
     (Attrib.empty_binding, @{term "True ∧ True"})) *}
```

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

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

Such bindings are difficult to maintain and can be overwritten by the user accidentally. This often broke Isabelle 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 chapter you will learn more about antiquotations: they can simplify Isabelle programming since one can directly access all kinds of logical elements from the ML-level.

2.5 Terms and Types

One way to construct Isabelle terms, is by using the antiquotation `@{term ...}`. For example:

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

will show the term `a + b = c`, but printed using an internal representation corresponding to the data type `term`.

This internal representation 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. Note that the names of bound variables are kept at abstractions for printing purposes, and so should be treated only as "comments". Application in Isabelle is realised with the term-constructor `$`.

> ***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 2.5.1. *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 printing depth to a higher value:

```
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 a 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
```

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

2.6 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). However, you often need to construct terms dynamically. For example, a function that returns the implication `⋀(x::nat). P x ⟹ Q x` taking `P` and `Q` as arguments can only be written as:

```
fun make_imp P Q =
let
  val x = Free ("x", @{typ nat})
in
  Logic.all x (Logic.mk_implies (P $ x, Q $ x))
end
```

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

```
fun make_wrong_imp P Q = @{prop "⋀(x::nat). P x ⟹ Q x"}
```

To see this apply `@{term S}` and `@{term T}` to both functions. With `make_imp` we obtain the intended term involving the given arguments

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

There are a number of handy functions that are frequently used for constructing terms. One is the function `list_comb` which a term and a list of terms as argument, and produces as output the term list applied to the term. For example

```
list_comb (@{term "P::nat"}, [@{term "True"}, @{term "False"}])
> Free ("P", "nat") $ Const ("True", "bool") $ Const ("False", "bool")
```

(FIXME: handy functions for constructing terms: `lambda`, `subst_free`)

```
  lambda @{term "x::nat"} @{term "P x"}
```

As can be seen this function does not take the typing annotation into account.

> ***Read More***
> *There are many functions in `Pure/term.ML`, `Pure/logic.ML` and `HOL/Tools/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 2.6.1. *Write a function `rev_sum : term -> term` that takes a term of the form t_1 + t_2 + ... + t_n (whereby n 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 2.6.2. *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.*

There are a few subtle issues with constants. They usually crop up when pattern matching terms or types, or when constructing them. While it is perfectly ok to write the function *is_true* as follows

```
fun is_true @{term True} = true
  | is_true _ = false
```

this does not work for picking out ∀-quantified terms. Because the function

```
fun is_all (@{term All} $ _) = true
  | is_all _ = false
```

will not correctly match the formula ∀ *x. P x*:

```
is_all @{term "∀x::nat. P x"}
> false
```

The problem is that the *@term*-antiquotation in the pattern fixes the type of the constant *All* to be *('a ⇒ bool) ⇒ bool* for an arbitrary, but fixed type *'a*. A properly working alternative for this function is

```
fun is_all (Const ("All", _) $ _) = true
  | is_all _ = false
```

because now

```
is_all @{term "∀x::nat. P x"}
> true
```

matches correctly (the first wildcard in the pattern matches any type and the second any term).

However there is still a problem: consider the similar function that attempts to pick out *Nil*-terms:

```
fun is_nil (Const ("Nil", _)) = true
  | is_nil _ = false
```

Unfortunately, also this function does *not* work as expected, since

```
is_nil @{term "Nil"}
> false
```

The problem is that on the ML-level the name of a constant is more subtle than you might expect. The function *is_all* worked correctly, because *All* is such a fundamental constant, which can be referenced by *Const ("All", some_type)*. However, if you look at

```
@{term "Nil"}
> Const ("List.list.Nil", ...)
```

the name of the constant *Nil* depends on the theory in which the term constructor is defined (*List*) and also in which data type (*list*). Even worse, some constants have a name involving type-classes. Consider for example the constants for *zero* and *(op *)*:

```
(@{term "0::nat"}, @{term "op *"})
> (Const ("HOL.zero_class.zero", ...),
>  Const ("HOL.times_class.times", ...))
```

While you could use the complete name, for example *Const ("List.list.Nil", some_type)*, for referring to or matching against *Nil*, this would make the code rather brittle. The reason is that the theory and the name of the data type can easily change. To make the code more robust, it is better to use the antiquotation *@{const_name ...}*. With this antiquotation you can harness the variable parts of the constant's name. Therefore a functions for matching against constants that have a polymorphic type should be written as follows.

```
fun is_nil_or_all (Const (@{const_name "Nil"}, _)) = true
  | is_nil_or_all (Const (@{const_name "All"}, _) $ _) = true
  | is_nil_or_all _ = false
```

Occasional you have to calculate what the "base" name of a given constant is. For this you can use the function *Sign.extern_const* or *Long_Name.base_name*. For example:

```
Sign.extern_const @{theory} "List.list.Nil"
> "Nil"
```

The difference between both functions is that *extern_const* returns the smallest name that is still unique, whereas *base_name* always strips off all qualifiers.

> ***Read More***
> *Functions about naming are implemented in* Pure/General/name_space.ML*; functions about signatures in* Pure/sign.ML.

Although types of terms can often be inferred, there are many situations where you need to construct types manually, especially when defining constants. 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 with the combinator --> as:

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

A handy function for manipulating terms is `map_types`: it takes a function and applies it to every type in a term. You can, for example, change every `nat` in a term into an `int` using the function:

```
fun nat_to_int t =
  (case t of
     @{typ nat} => @{typ int}
   | Type (s, ts) => Type (s, map nat_to_int ts)
   | _ => t)
```

An example as follows:

```
map_types nat_to_int @{term "a = (1::nat)"}
> Const ("op =", "int ⇒ int ⇒ bool")
>            $ Free ("a", "int") $ Const ("HOL.one_class.one", "int")
```

(FIXME: readmore about types)

2.7 Type-Checking

You can freely construct and manipulate `terms` and `types`, since they are just arbitrary unchecked trees. However, you 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 `terms`, which are just trees, `cterms` 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 you can write:

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

This can also be written 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
```

In Isabelle not just terms need to be certified, but also types. For example, you obtain the certified type for the Isabelle type `nat ⇒ bool` on the ML-level as follows:

```
ctyp_of @{theory} (@{typ nat} --> @{typ bool})
> nat ⇒ bool
```

> ***Read More***
> *For functions related to* `cterm`*s and* `ctyp`*s see the file* `Pure/thm.ML`*.*

Exercise 2.7.1. *Check that the function defined in Exercise 2.6.1 returns a result that type-checks.*

Remember Isabelle follows the Church-style typing for terms, i.e. a term contains enough typing information (constants, free variables and abstractions all have typing information) so that it is always clear what the type of a term is. Given a well-typed term, the function `type_of` returns the type of a term. Consider for example:

```
type_of (@{term "f::nat ⇒ bool"} $ @{term "x::nat"})
> bool
```

To calculate the type, this function traverses the whole term and will detect any typing inconsistency. For example changing the type of the variable `x` from `nat` to `int` will result in the error message:

```
type_of (@{term "f::nat ⇒ bool"} $ @{term "x::int"})
> *** Exception- TYPE ("type_of: type mismatch in application" ...
```

Since the complete traversal might sometimes be too costly and not necessary, there is the function `fastype_of`, which also returns the type of a term.

```
fastype_of (@{term "f::nat ⇒ bool"} $ @{term "x::nat"})
> bool
```

However, efficiency is gained on the expense of skipping some tests. You can see this in the following example

```
fastype_of (@{term "f::nat ⇒ bool"} $ @{term "x::int"})
> bool
```

where no error is detected.

Sometimes it is a bit inconvenient to construct a term with complete typing annotations, especially in cases where the typing information is redundant. A short-cut is to use the "place-holder" type *dummyT* and then let type-inference figure out the complete type. An example is as follows:

```
let
  val c = Const (@{const_name "plus"}, dummyT)
  val o = @{term "1::nat"}
  val v = Free ("x", dummyT)
in
  Syntax.check_term @{context} (c $ o $ v)
end
> Const ("HOL.plus_class.plus", "nat ⇒ nat ⇒ nat") $
>   Const ("HOL.one_class.one", "nat") $ Free ("x", "nat")
```

Instead of giving explicitly the type for the constant `plus` and the free variable *x*, the type-inference fills in the missing information.

> ***Read More***
> *See `Pure/Syntax/syntax.ML` where more functions about reading, checking and pretty-printing of terms are defined. Functions related to the type inference are implemented in `Pure/type.ML` and `Pure/type_infer.ML`.*

(FIXME: say something about sorts)

2.8 Theorems

Just like *cterm*s, theorems are abstract objects of type *thm* that can only be build by going through interfaces. As a consequence, every proof in Isabelle is correct by construction. This follows the tradition of the LCF approach [2].

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:

```
let
  val assm1 = @{cprop "⋀(x::nat). P x ⟹ Q x"}
  val assm2 = @{cprop "(P::nat⇒bool) t"}
```

```
  val Pt_implies_Qt =
        assume assm1
        |> forall_elim @{cterm "t::nat"};

  val Qt = implies_elim Pt_implies_Qt (assume assm2);
in
  Qt
  |> implies_intr assm2
  |> implies_intr assm1
end
> ⟦⋀x. P x ⟹ Q x; P t⟧ ⟹ Q t
```

This code-snippet constructs the following proof:

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

However, while we obtained a theorem as result, this theorem is not yet stored in Isabelle's theorem database. So it cannot be referenced later on. How to store theorems will be explained in Section 2.12.

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

(FIXME: how to add case-names to goal states - maybe in the next section)

2.9 Theorem Attributes

Theorem attributes are *[simp]*, *[OF ...]*, *[symmetric]* and so on. Such attributes are *neither* tags *nor* flags annotated to theorems, but functions that do further processing once a theorem is proven. In particular, it is not possible to find out what are all theorems that have a given attribute in common, unless of course the function behind the attribute stores the theorems in a retrievable data structure.

If you want to print out all currently known attributes a theorem can have, you can use the function:

```
Attrib.print_attributes @{theory}
> COMP:  direct composition with rules (no lifting)
> HOL.dest:  declaration of Classical destruction rule
> HOL.elim:  declaration of Classical elimination rule
> ...
```

To explain how to write your own attribute, let us start with an extremely simple version of the attribute *[symmetric]*. The purpose of this attribute is to produce the "symmetric" version of an equation. The main function behind this attribute is

```
val my_symmetric = Thm.rule_attribute (fn _ => fn thm => thm RS @{thm sym})
```

where the function *Thm.rule_attribute* expects a function taking a context (which we ignore in the code above) and a theorem (*thm*), and returns another theorem (namely *thm* resolved with the lemma *sym*: $s = t \Longrightarrow t = s$). The function *Thm.rule_attribute* then returns an attribute.

Before we can use the attribute, we need to set it up. This can be done using the function *Attrib.setup* as follows.

```
setup {* Attrib.setup @{binding "my_sym"}
  (Scan.succeed my_symmetric) "applying the sym rule"*}
```

The attribute does not expect any further arguments (unlike *[OF ...]*, for example, which can take a list of theorems as argument). Therefore we use the parser *Scan.succeed*. Later on we will also consider attributes taking further arguments. An example for the attribute *[my_sym]* is the proof

lemma *test[my_sym]: "2 = Suc (Suc 0)"* **by** *simp*

which stores the theorem *Suc (Suc 0) = 2* under the name *test*. We can also use the attribute when referring to this theorem.

```
thm test[my_sym]
> 2 = Suc (Suc 0)
```

The purpose of *Thm.rule_attribute* is to directly manipulate theorems. Another usage of attributes is to add and delete theorems from stored data. For example the attribute *[simp]* adds or deletes a theorem from the current simpset. For these applications, you can use *Thm.declaration_attribute*. To illustrate this function, let us introduce a reference containing a list of theorems.

```
val my_thms = ref ([]:thm list)
```

A word of warning: such references must not be used in any code that is meant to be more than just for testing purposes! Here it is only used to illustrate matters. We will show later how to store data properly without using references. The function *Thm.declaration_attribute* expects us to provide two functions that add and delete theorems from this list. For this we use the two functions:

```
fun my_thms_add thm ctxt =
  (my_thms := Thm.add_thm thm (!my_thms); ctxt)

fun my_thms_del thm ctxt =
  (my_thms := Thm.del_thm thm (!my_thms); ctxt)
```

These functions take a theorem and a context and, for what we are explaining here it is sufficient that they just return the context unchanged. They change however the reference *my_thms*, whereby the function *Thm.add_thm* adds a theorem if it is not already included in the list, and *Thm.del_thm* deletes one. Both functions use the predicate *Thm.eq_thm_prop* which compares theorems according to their proved propositions (modulo alpha-equivalence).

You can turn both functions into attributes using

```
val my_add = Thm.declaration_attribute my_thms_add
val my_del = Thm.declaration_attribute my_thms_del
```

and set up the attributes as follows

```
setup {* Attrib.setup @{binding "my_thms"}
  (Attrib.add_del my_add my_del) "maintaining a list of my_thms" *}
```

Now if you prove the lemma attaching the attribute *[my_thms]*

lemma *trueI_2[my_thms]: "True"* **by** *simp*

then you can see the lemma is added to the initially empty list.

```
!my_thms
> ["True"]
```

You can also add theorems using the command **declare**.

declare *test[my_thms] trueI_2[my_thms add]*

The *add* is the default operation and does not need to be given. This declaration will cause the theorem list to be updated as follows.

```
!my_thms
> ["True", "Suc (Suc 0) = 2"]
```

The theorem *trueI_2* only appears once, since the function *Thm.add_thm* tests for duplicates, before extending the list. Deletion from the list works as follows:

declare *test[my_thms del]*

After this, the theorem list is again:

```
!my_thms
> ["True"]
```

We used in this example two functions declared as *Thm.declaration_attribute*, but there can be any number of them. We just have to change the parser for reading the arguments accordingly.

However, as said at the beginning of this example, using references for storing theorems is *not* the received way of doing such things. The received way is to start a "data slot" in a context by using the functor *GenericDataFun*:

```
structure Data = GenericDataFun
(type T = thm list
 val empty = []
 val extend = I
 fun merge _ = Thm.merge_thms)
```

To use this data slot, you only have to change *my_thms_add* and *my_thms_del* to:

```
val thm_add = Data.map o Thm.add_thm
val thm_del = Data.map o Thm.del_thm
```

where *Data.map* updates the data appropriately in the context. Since storing theorems in a special list is such a common task, there is the functor *NamedThmsFun*, which does most of the work for you. To obtain such a named theorem lists, you just declare

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

and set up the *FooRules* with the command

```
setup {* FooRules.setup *}
```

This code declares a data slot where the theorems are stored, an attribute *foo* (with the *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 you 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* can be retrieved using the function *FooRules.get*:

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

> ***Read More***
> *For more information see `Pure/Tools/named_thms.ML` and also the recipe in Section* **??** *about storing arbitrary data.*

(FIXME What are: `theory_attributes`, `proof_attributes`?)

> ***Read More***
> *FIXME:* `Pure/more_thm.ML` `Pure/Isar/attrib.ML`

2.10 Setups (TBD)

In the previous section we used **setup** in order to make a theorem attribute known to Isabelle. What happens behind the scenes is that **setup** expects a functions of type `theory -> theory`: the input theory is the current theory and the output the theory where the theory attribute has been stored.

This is a fundamental principle in Isabelle. A similar situation occurs for example with declaring constants. The function that declares a constant on the ML-level is `Sign.add_consts_i`. If you write[1]

```
Sign.add_consts_i [(@{binding "BAR"}, @{typ "nat"}, NoSyn)] @{theory}
```

for declaring the constant `BAR` with type `nat` and run the code, then you indeed obtain a theory as result. But if you query the constant on the Isabelle level using the command **term**

```
term "BAR"
> "BAR" :: "'a"
```

you do not obtain a constant of type `nat`, but a free variable (printed in blue) of polymorphic type. The problem is that the ML-expression above did not register the declaration with the current theory. This is what the command **setup** is for. The constant is properly declared with

```
setup {* Sign.add_consts_i [(@{binding "BAR"}, @{typ "nat"}, NoSyn)] *}
```

Now

```
term "BAR"
> "BAR" :: "nat"
```

[1]Recall that ML-code needs to be enclosed in **ML** `{* ... *}`.

returns a (black) constant with the type *nat*.

A similar command is **local_setup**, which expects a function of type *local_theory -> local_theory*. Later on we will also use the commands **method_setup** for installing methods in the current theory and **simproc_setup** for adding new simprocs to the current simpset.

2.11 Theories, Contexts and Local Theories (TBD)

There are theories, proof contexts and local theories (in this order, if you want to order them).

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.

2.12 Storing Theorems (TBD)

PureThy.add_thms_dynamic

2.13 Pretty-Printing (TBD)

Pretty.big_list, *Pretty.brk*, *Pretty.block*, *Pretty.chunks*

2.14 Misc (TBD)

```
DatatypePackage.get_datatype @{theory} "List.list"
```

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, 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 `$$` 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"])
```

The 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 `!!` (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 `$$` 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 `--`. For example parsing `h`, `e` and `l` in this sequence you can achieve 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, you want to parse a particular string, then you 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 `||`. 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 `|--` and `--|` 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 `!!` helps to produce appropriate error messages during parsing. For example if you want to parse that `p` is immediately followed by `q`, or start a completely different parser `r`, you 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 you lose 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 `!!`. This function aborts the whole process of parsing in case of a failure and prints an error message. For example if you invoke the parser

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

on `"hello"`, the parsing succeeds

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

but if you invoke it on `"world"`

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

then the parsing aborts and the error message `foo` is printed. In order to see the error message properly, you 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` (see Section 3.5 which explains this function in more detail).

Let us now return to our example of parsing `(p -- q) || r`. If you want to generate the correct error message for p-followed-by-q, then you 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 you 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 you 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, for example, tokens. 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, you nearly always want to apply a function on the parsed items. One way to do this is the function `(p >> f)`, 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)
   val input = (explode "foo bar foo")
in
   Scan.finite Symbol.stopper (p >> implode) input
end
> ("foo bar foo",[])
```

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

The function *Scan.ahead* parses some input, but leaves the original input unchanged. For example:

```
Scan.ahead (Scan.this_string "foo") (explode "foo")
> ("foo", ["f", "o", "o"])
```

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.)

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`*.*

3.2 Parsing Theory Syntax

(FIXME: context parser)

Most of the time, however, Isabelle developers have to deal with parsing tokens, not strings. These token parsers have the type:

```
type 'a parser = OuterLex.token list -> 'a * OuterLex.token list
```

The reason for using token parsers is that theory syntax, as well as the parsers for the arguments of proof methods, use the type *OuterLex.token* (which is identical to the type *OuterParse.token*). However, there are also handy 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.

The first example shows how to generate a token list out of a string using the function *OuterSyntax.scan*. It is given the argument *Position.none* 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:

```
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 you now parse

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

you obtain a list consisting of only a command and two keyword tokens. If you want to see which keywords and commands are currently known to Isabelle, 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
> (["}", "{", ...], ["⇌", "↽", ...])
```

The parser `OuterParse.$$$` parses a single keyword. 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 input1 = filtered_input "where for"
  val input2 = filtered_input "| in"
in
  (OuterParse.$$$ "where" input1, OuterParse.$$$ "|" input2)
end
> (("where",...), ("|",...))
```

Like before, you can sequentially connect parsers with `--`. 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`*.*

(FIXME: or give parser for numbers)

Whenever there is a possibility that the processing of user input can fail, it is a good idea to give as much information about where the error occured. For this Isabelle can attach positional information to tokens and then thread this information up the processing chain. To see this, modify the function *filtered_input* described earlier to

```
fun filtered_input' str =
       filter OuterLex.is_proper (OuterSyntax.scan (Position.line 7) str)
```

where we pretend the parsed string starts on line 7. An example is

```
filtered_input' "foo \n bar"
> [Token (("foo", ({line=7, end_line=7}, {line=7})), (Ident, "foo"), ...),
>  Token (("bar", ({line=8, end_line=8}, {line=8})), (Ident, "bar"), ...)]
```

in which the *"\n"* causes the second token to be in line 8.

By using the parser *OuterParse.position* you can decode the positional information and return it as part of the parsed input. For example

```
let
  val input = (filtered_input' "where")
in
  parse (OuterParse.position (OuterParse.$$$ "where")) input
end
> (("where", {line=7, end_line=7}), [])
```

> ***Read More***
> *The functions related to positions are implemented in the file* `Pure/General/position.ML`.

3.3 Parsing Inner Syntax

There is usually no need to write your own parser for parsing inner syntax, that is for terms and types: you can just call the pre-defined parsers. Terms can be parsed using the function `OuterParse.term`. For example:

```
let
  val input = OuterSyntax.scan Position.none "foo"
in
  OuterParse.term input
end
> ("\^E\^Ftoken\^Efoo\^E\^F\^E", [])
```

The function `OuterParse.prop` is similar, except that it gives a different error message, when parsing fails. As you can see, the parser not just returns the parsed string, but also some encoded information. You can decode the information with the function `YXML.parse`. For example

```
YXML.parse "\^E\^Ftoken\^Efoo\^E\^F\^E"
> XML.Elem ("token", [], [XML.Text "foo"])
```

The result of the decoding is an XML-tree. You can see better what is going on if you replace `Position.none` by `Position.line 42`, say:

```
let
  val input = OuterSyntax.scan (Position.line 42) "foo"
in
  YXML.parse (fst (OuterParse.term input))
end
> XML.Elem ("token", [("line", "42"), ("end_line", "42")], [XML.Text "foo"])
```

The positional information is stored as part of an XML-tree so that code called later on will be able to give more precise error messages.

> ***Read More***
> *The functions to do with input and output of XML and YXML are defined in* `Pure/General/xml.ML` *and* `Pure/General/yxml.ML`.

3.4 Parsing Specifications

There are a number of special purpose parsers that help with parsing specifications of function definitions, inductive predicates and so on. In Capter 5, for example, we will need to parse specifications for inductive predicates of the form:

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

For this we are going to use the parser:

```
1 val spec_parser =
2      OuterParse.fixes --
3      Scan.optional
4        (OuterParse.$$$ "where" |--
5           OuterParse.!!!
6             (OuterParse.enum1 "|"
7                (SpecParse.opt_thm_name ":" -- OuterParse.prop))) []
```

Note that the parser does not parse the keyword **simple_inductive**, even if it is meant to process definitions as shown above. The parser of the keyword will be given by the infrastructure that will eventually call `spec_parser`.

To see what the parser returns, let us parse the string corresponding to the definition of `even` and `odd`:

```
let
  val input = filtered_input
     ("even and odd " ^
      "where " ^
      "  even0[intro]: \"even 0\" " ^
      "| evenS[intro]: \"odd n ⟹ even (Suc n)\" " ^
      "| oddS[intro]:  \"even n ⟹ odd (Suc n)\"")
in
  parse spec_parser input
end
> (([(even, NONE, NoSyn), (odd, NONE, NoSyn)],
>      [((even0,...), "\^E\^Ftoken\^Eeven 0\^E\^F\^E"),
>       ((evenS,...), "\^E\^Ftoken\^Eodd n ⟹ even (Suc n)\^E\^F\^E"),
>       ((oddS,...), "\^E\^Ftoken\^Eeven n ⟹ odd (Suc n)\^E\^F\^E")]), [])
```

As you see, the result is a pair consisting of a list of variables with optional type-annotation and syntax-annotation, and a list of rules where every rule has optionally a name and an attribute.

The function `OuterParse.fixes` in Line 2 of the parser reads an **and**-separated list of variables that can include optional type annotations and syntax translations. For example:[2]

```
let
  val input = filtered_input
        "foo::\"int ⇒ bool\" and bar::nat (\"BAR\" 100) and blonk"
in
   parse OuterParse.fixes input
end
> ([(foo, SOME "\^E\^Ftoken\^Eint ⇒ bool\^E\^F\^E", NoSyn),
>   (bar, SOME "\^E\^Ftoken\^Enat\^E\^F\^E", Mixfix ("BAR", [], 100)),
>   (blonk, NONE, NoSyn)],[])
```

Whenever types are given, they are stored in the `SOME`s. The types are not yet used to type the variables: this must be done by type-inference later on. Since types are part of the inner syntax they are strings with some encoded information (see previous section). If a syntax translation is present for a variable, then it is stored in the `Mixfix` datastructure; no syntax translation is indicated by `NoSyn`.

> ***Read More***
> *The datastructre for sytax annotations is defined in* `Pure/Syntax/mixfix.ML`.

Lines 3 to 7 in the function `spec_parser` implement the parser for a list of introduction rules, that is propositions with theorem annotations. The introduction rules are propositions parsed by `OuterParse.prop`. However, they can include an optional theorem name plus some attributes. For example

```
let
  val input = filtered_input "foo_lemma[intro,dest!]:"
  val ((name, attrib), _) = parse (SpecParse.thm_name ":") input
in
  (name, map Args.dest_src attrib)
end
> (foo_lemma, [(("intro", []), ...), (("dest", [...]), ...)])
```

The function `opt_thm_name` is the "optional" variant of `thm_name`. Theorem names can contain attributes. The name has to end with `":"`—see the argument of the function `SpecParse.opt_thm_name` in Line 7.

> ***Read More***
> *Attributes and arguments are implemented in the files* `Pure/Isar/attrib.ML` *and* `Pure/Isar/args.ML`.

[2]Note that in the code we need to write `\"int ⇒ bool\"` in order to properly escape the double quotes in the compound type.

3.5 New Commands and Keyword Files

(FIXME: update to the right command setup)

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
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 you have to do on the ML-level, you 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*. This command requires log files to be present (in order to extract the keywords from them). To generate these log files, you first need to 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"
```

```
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**.

For the theory *Command.thy*, you first need to 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
```

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

```
use_thy "Command";
```

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

```
$ isabelle make
```

If the compilation succeeds, you have finally created 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. If you now type `ls $ISABELLE_LOGS`, then the directory should include the files:

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

From them you can 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.[3] This keyword file needs to be copied into the directory `~/.isabelle/etc`. To make Isabelle aware of this keyword file, you have to start Isabelle with the option `-k foobar`, that is:

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

If you now build a theory on top of `Command.thy`, then the command **foobar** can be used. Similarly with any other new command.

At the moment **foobar** is not very useful. Let us refine it a bit next by letting it take 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 you 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 you can type

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

[3]To see whether things are fine, check that `grep foobar` on this file returns something non-empty.

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 **function**). To achieve this kind of behaviour, you have to use the kind indicator *thy_goal*. Note, however, once you change the "kind" of a command from *thy_decl* to *thy_goal* then the keyword file needs to be re-created!

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, 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
14 in
15   OuterSyntax.command "foobar" "proving a proposition" kind prove_prop
16 end
```

The function *set_up_thm* in Lines 2 to 7 takes a string (the proposition to be proved) and a context as argument. 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). Lines 9 to 11 contain the parser for the proposition.

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

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

and you can build the proof

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

(FIXME What do *Toplevel.theory* *Toplevel.print* *Toplevel.local_theory* do?)
(FIXME read a name and show how to store theorems)

3.6 Methods

Methods are a central concept in Isabelle. They are the ones you use for example in **apply**. To print out all currently known methods you can use the Isabelle command:

print_methods

An example of a very simple method is the following code.

```
method_setup foobar_meth =
 {* Scan.succeed
      (K (SIMPLE_METHOD ((etac @{thm conjE} THEN' rtac @{thm conjI}) 1))) *}
         "foobar method"
```

It defines the method *foobar_meth*, which takes no arguments (therefore the parser *Scan.succeed*) and only applies the tactic *conjE* and then *conjI*. This method can be used in the following proof

lemma shows "$A \wedge B \Longrightarrow C \wedge D$"
apply(foobar_meth)

where it results in the goal state

goal (2 subgoals):
 1. $[\![A;\ B]\!] \Longrightarrow C$
 2. $[\![A;\ B]\!] \Longrightarrow D$

(FIXME: explain a version of rule-tac)

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 definitions. 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. However, there are also more structured operations available on the ML-level that help with the handling of variables and assumptions.

4.1 Basics of Reasoning with Tactics

To see how tactics work, let us first transcribe a simple **apply**-style proof into ML. Suppose 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` (that is `P ∨ Q ⟹ Q ∨ P` in the proof at hand) under the assumptions `As` (happens to be empty) 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` in the code above correspond to `erule`, `rule` and `assumption`, respectively. The operator `THEN` strings the tactics together.

> ***Read More***
> *To learn more about the function `Goal.prove` see [Impl. Man., Sec. 4.3] and the file `Pure/goal.ML`. See `Pure/tactic.ML` and `Pure/tctical.ML` for the code of basic tactics and tactic combinators; see also Chapters 3 and 4 in the old Isabelle Reference Manual, and Chapter 3 in the Isabelle/Isar Implementation Manual.*

Note that in the code above we use antiquotations for referencing the theorems. Many theorems also have ML-bindings with the same name. Therefore, we could also just have written `etac disjE 1`, or in case where there are no ML-binding obtain the theorem dynamically using the function `thm`; for example `etac (thm "disjE") 1`. Both ways however are 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, you will often find it 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

By using `tactic {* ... *}` you can call from the user-level of Isabelle the tactic `foo_tac` or any other function that returns a tactic.

The tactic `foo_tac` is just a sequence of simple tactics stringed together by `THEN`. As can be seen, each simple tactic in `foo_tac` has a hard-coded number that stands for the subgoal analysed by the tactic (`1` stands for the first, or top-most, subgoal). This hard-coding of goals is sometimes wanted, but usually it is not. To avoid the explicit numbering, 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 you can first discharge the second subgoal, and subsequently the first.

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

This kind of addressing is more difficult to achieve when the goal is hard-coded inside the tactic. For most operators that combine tactics (*THEN* is only one such operator) a "primed" version exists.

The tactics *foo_tac* and *foo_tac'* are very specific for analysing goals being only of the form $P \vee Q \Longrightarrow Q \vee P$. If the goal is not of this form, then these tactics return the error message:

```
*** empty result sequence -- proof command failed
*** At command "apply".
```

This means they failed. The reason for this error message is that tactics are functions mapping a goal state to a (lazy) sequence of successor states. Hence the type of a tactic is:

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

By convention, if a tactic fails, then it should return the empty sequence. Therefore, if you write your own tactics, they should not raise exceptions willy-nilly; only in very grave failure situations should a tactic raise the exception *THM*.

The simplest tactics are *no_tac* and *all_tac*. The first returns the empty sequence and is defined as

```
fun no_tac thm = Seq.empty
```

which means *no_tac* always fails. The second returns the given theorem wrapped in a single member sequence; it is defined as

```
fun all_tac thm = Seq.single thm
```

which means *all_tac* always succeeds, but also does not make any progress with the proof.

The lazy list of possible successor goal states shows through at the user-level of Isabelle when using the command **back**. For instance in the following proof there are two possibilities for how to apply *foo_tac'*: either using the first assumption or the second.

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

By using **back**, we construct the proof that uses the second assumption. While in the proof above, it does not really matter which assumption is used, in more interesting cases provability might depend on exploring different possibilities.

> ***Read More***
> *See Pure/General/seq.ML for the implementation of lazy sequences. In day-to-day Isabelle programming, however, one rarely constructs sequences explicitly, but uses the predefined tactics and tactic combinators instead.*

It might be surprising that tactics, which transform one goal state to the next, are functions from theorems to theorem (sequences). The surprise resolves by knowing that every goal state is indeed a theorem. To shed more light on this, let us modify the code of *all_tac* to obtain the following tactic

```
fun my_print_tac ctxt thm =
let
  val _ = warning (str_of_thm ctxt thm)
in
  Seq.single thm
end
```

which prints out the given theorem (using the string-function defined in Section 2.2) and then behaves like *all_tac*. With this tactic we are in the position to inspect every goal state in a proof. Consider now the proof in Figure 4.1: as can be seen, internally every goal state 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 subgoals. So after setting up the lemma, the goal state is always of the form $C \Longrightarrow (C)$; when the proof is finished we are left with (C). Since the goal C can potentially be an implication, there is a “protector” wrapped around it (in from of an outermost constant *Const ("prop", bool ⇒ bool)*; however this constant is invisible in the figure). This wrapper prevents that premises of C are mis-interpreted as open subgoals. 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. If you use the predefined tactics, which we describe in the next section, this will always be the case.

```
lemma shows "⟦A; B⟧ ⟹ A ∧ B"
apply(tactic {* my_print_tac @{context} *})

goal (1 subgoal):
 1. ⟦A; B⟧ ⟹ A ∧ B

internal goal state:
(⟦A; B⟧ ⟹ A ∧ B) ⟹ (⟦A; B⟧ ⟹ A ∧ B)

apply(rule conjI)
apply(tactic {* my_print_tac @{context} *})

goal (2 subgoals):
 1. ⟦A; B⟧ ⟹ A
 2. ⟦A; B⟧ ⟹ B

internal goal state:
(⟦A; B⟧ ⟹ A) ⟹ (⟦A; B⟧ ⟹ B) ⟹ (⟦A; B⟧ ⟹ A ∧ B)

apply(assumption)
apply(tactic {* my_print_tac @{context} *})

goal (1 subgoal):
 1. ⟦A; B⟧ ⟹ B

internal goal state:
(⟦A; B⟧ ⟹ B) ⟹ (⟦A; B⟧ ⟹ A ∧ B)

apply(assumption)
apply(tactic {* my_print_tac @{context} *})

No subgoals!

internal goal state:
⟦A; B⟧ ⟹ A ∧ B

done
```

Figure 4.1: The figure shows a proof where each intermediate goal state is printed by the Isabelle system and by *my_print_tac*. The latter shows the goal state as represented internally (highlighted boxes). This tactic shows that every goal state in Isabelle is represented by a theorem: when you start the proof of ⟦A; B⟧ ⟹ A ∧ B the theorem is (⟦A; B⟧ ⟹ A ∧ B) ⟹ (⟦A; B⟧ ⟹ A ∧ B); when you finish the proof the theorem is ⟦A; B⟧ ⟹ A ∧ B.

> ***Read More***
> *For more information about the internals of goals see [Impl. Man., Sec. 3.1].*

4.2 Simple Tactics

Let us start with explaining the simple tactic `print_tac`, which is quite useful for low-level debugging of tactics. It just prints out a message and the current goal state. Unlike `my_print_tac` shown earlier, it prints the goal state as the user would see it. For example, processing the proof

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

gives:

```
foo message

False ⟹ True
1. False ⟹ True
```

A simple tactic making theorem proving particularly simple is `SkipProof.cheat_tac`. This tactic corresponds to the Isabelle command **sorry** and is sometimes useful during the development of tactics.

```
lemma shows "False" and "something_else_obviously_false"
apply(tactic {* SkipProof.cheat_tac @{theory} *})
```

```
No subgoals!
```

Another simple tactic is the function `atac`, which, as shown in the previous section, corresponds to the assumption command.

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

```
No subgoals!
```

Similarly, `rtac`, `dtac`, `etac` and `ftac` correspond to `rule`, `drule`, `erule` and `frule`, respectively. Each of them takes a theorem as argument and attempts to apply it to a goal. Below are three self-explanatory examples.

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

```
goal (2 subgoals):
 1. P
 2. Q
```

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

```
goal (1 subgoal):
 1. ⟦P; Q⟧ ⟹ False
```

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

goal (1 subgoal):
1. True ⟹ False

Note the number in each tactic call. Also as mentioned in the previous section, most basic tactics take such a number as argument: this argument addresses the subgoal the tacics are analysing. In the proof below, we first split up the conjunction in the second subgoal by focusing on this subgoal first.

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

goal (3 subgoals):
1. Foo
2. P
3. Q

The function `resolve_tac` is similar to `rtac`, except that it expects a list of theorems as arguments. From this list it will apply the first applicable theorem (later theorems that are also applicable can be explored via the lazy sequences mechanism). Given the code

```
val resolve_tac_xmp = resolve_tac [@{thm impI}, @{thm conjI}]
```

an example for `resolve_tac` is the following proof where first an outermost implication is analysed and then an outermost conjunction.

lemma shows *"C ⟶ (A ∧ B)"* **and** *"(A ⟶ B) ∧ C"*
apply*(tactic {* resolve_tac_xmp 1 *})*
apply*(tactic {* resolve_tac_xmp 2 *})*

goal (3 subgoals):
1. C ⟹ A ∧ B
2. A ⟶ B
3. C

Similar versions taking a list of theorems exist for the tactics `dtac` (`dresolve_tac`), `etac` (`eresolve_tac`) and so on.

Another simple tactic is `cut_facts_tac`. It inserts a list of theorems into the assumptions of the current goal state. For example

lemma shows *"True ≠ False"*
apply*(tactic {* cut_facts_tac [@{thm True_def}, @{thm False_def}] 1 *})*

produces the goal state

goal (1 subgoal):
1. ⟦True ≡ (λx. x) = (λx. x); False ≡ ∀P. P⟧ ⟹ True ≠ False

Since rules are applied using higher-order unification, an automatic proof procedure might become too fragile, if it just applies inference rules as shown above. The reason is that a number of rules introduce meta-variables into the goal state. Consider for example the proof

lemma shows *"∀x∈A. P x ⟹ Q x"*
apply(*tactic {* dtac @{thm bspec} 1 *}*)

```
goal (2 subgoals):
 1. ?x ∈ A
 2. P ?x ⟹ Q x
```

where the application of rule *bspec* generates two subgoals involving the meta-variable *?x*. Now, if you are not careful, tactics applied to the first subgoal might instantiate this meta-variable in such a way that the second subgoal becomes unprovable. If it is clear what the *?x* should be, then this situation can be avoided by introducing a more constraint version of the *bspec*-rule. Such constraints can be given by pre-instantiating theorems with other theorems. One function to do this is *RS*

```
@{thm disjI1} RS @{thm conjI}
> ⟦?P1; ?Q⟧ ⟹ (?P1 ∨ ?Q1) ∧ ?Q
```

which in the example instantiates the first premise of the *conjI*-rule with the rule *disjI1*. If the instantiation is impossible, as in the case of

```
@{thm conjI} RS @{thm mp}
> *** Exception- THM ("RSN: no unifiers", 1,
> ["⟦?P; ?Q⟧ ⟹ ?P ∧ ?Q", "⟦?P ⟶ ?Q; ?P⟧ ⟹ ?Q"]) raised
```

then the function raises an exception. The function *RSN* is similar to *RS*, but takes an additional number as argument that makes explicit which premise should be instantiated.

To improve readability of the theorems we produce below, we shall use the function *no_vars* from Section 2.2, which transforms schematic variables into free ones. Using this function for the first *RS*-expression above produces the more readable result:

```
no_vars @{context} (@{thm disjI1} RS @{thm conjI})
> ⟦P; Q⟧ ⟹ (P ∨ Qa) ∧ Q
```

If you want to instantiate more than one premise of a theorem, you can use the function *MRS*:

```
no_vars @{context} ([@{thm disjI1}, @{thm disjI2}] MRS @{thm conjI})
> ⟦P; Q⟧ ⟹ (P ∨ Qa) ∧ (Pa ∨ Q)
```

If you need to instantiate lists of theorems, you can use the functions *RL* and *MRL*. For example in the code below, every theorem in the second list is instantiated with every theorem in the first.

```
fun sp_tac {prems, params, asms, concl, context, schematics} =
let
  val str_of_params = str_of_cterms context params
  val str_of_asms = str_of_cterms context asms
  val str_of_concl = str_of_cterm context concl
  val str_of_prems = str_of_thms context prems
  val str_of_schms = str_of_cterms context (snd schematics)

  val _ = (warning ("params: " ^ str_of_params);
           warning ("schematics: " ^ str_of_schms);
           warning ("assumptions: " ^ str_of_asms);
           warning ("conclusion: " ^ str_of_concl);
           warning ("premises: " ^ str_of_prems))
in
  no_tac
end
```

Figure 4.2: A function that prints out the various parameters provided by the tactic *SUBPROOF*. It uses the functions defined in Section 2.2 for extracting strings from `cterms` and `thms`.

```
[@{thm impI}, @{thm disjI2}] RL [@{thm conjI}, @{thm disjI1}]
> [⟦P ⟹ Q; Qa⟧ ⟹ (P ⟶ Q) ∧ Qa,
>  ⟦Q; Qa⟧ ⟹ (P ∨ Q) ∧ Qa,
>  (P ⟹ Q) ⟹ (P ⟶ Q) ∨ Qa,
>  Q ⟹ (P ∨ Q) ∨ Qa]
```

> ***Read More***
> *The combinators for instantiating theorems are defined in* `Pure/drule.ML`.

Often proofs on the ML-level involve elaborate operations on assumptions and $\bigwedge$-quantified variables. To do such operations using the basic tactics shown so far is very unwieldy and brittle. Some convenience and safety is provided by the tactic *SUBPROOF*. This tactic fixes the parameters and binds the various components of a goal state to a record. To see what happens, assume the function defined in Figure 4.2, which takes a record and just prints out the content of this record (using the string transformation functions from in Section 2.2). Consider now the proof:

lemma shows *"⋀x y. A x y ⟹ B y x ⟶ C (?z y) x"*
apply(*tactic {* SUBPROOF sp_tac @{context} 1 *}*)?

The tactic produces the following printout:

params:	x, y
schematics:	z
assumptions:	`A x y`
conclusion:	`B y x ⟶ C (z y) x`
premises:	`A x y`

Notice in the actual output the brown colour of the variables x and y. Although they are parameters in the original goal, they are fixed inside the subproof. By convention these fixed variables are printed in brown colour. Similarly the schematic variable *z*. The assumption, or premise, *A x y* is bound as *cterm* to the record-variable *asms*, but also as *thm* to *prems*.

Notice also that we had to append *"?"* to the **apply**-command. The reason is that *SUBPROOF* normally expects that the subgoal is solved completely. Since in the function *sp_tac* we returned the tactic *no_tac*, the subproof obviously fails. The question-mark allows us to recover from this failure in a graceful manner so that the warning messages are not overwritten by an "empty sequence" error message.

If we continue the proof script by applying the *impI*-rule

apply*(rule impI)*
apply*(tactic {* SUBPROOF sp_tac @{context} 1 *})?*

then the tactic prints out:

params:	*x*, *y*
schematics:	*z*
assumptions:	*A x y*, *B y x*
conclusion:	*C (z y) x*
premises:	*A x y*, *B y x*

Now also *B y x* is an assumption bound to *asms* and *prems*.

One convenience of *SUBPROOF* is that we can apply the assumptions using the usual tactics, because the parameter *prems* contains them as theorems. With this you can easily implement a tactic that behaves almost like *atac*:

```
val atac' = SUBPROOF (fn {prems, ...} => resolve_tac prems 1)
```

If you apply *atac'* to the next lemma

lemma shows *"⟦B x y; A x y; C x y⟧ ⟹ A x y"*
apply*(tactic {* atac' @{context} 1 *})*

it will produce

No subgoals!

The restriction in this tactic which is not present in *atac* is that it cannot instantiate any schematic variable. This might be seen as a defect, but it is actually an advantage in the situations for which *SUBPROOF* was designed: the reason is that, as mentioned before, instantiation of schematic variables can affect several goals and can render them unprovable. *SUBPROOF* is meant to avoid this.

Notice that *atac'* inside *SUBPROOF* calls *resolve_tac* with the subgoal number *1* and also the outer call to *SUBPROOF* in the **apply**-step uses *1*. This is another advantage of *SUBPROOF*: the addressing inside it is completely local to the tactic inside the subproof. It is therefore possible to also apply *atac'* to the second goal by just writing:

lemma shows *"True"* **and** *"⟦B x y; A x y; C x y⟧ ⟹ A x y"*

```
apply(tactic {* atac' @{context} 2 *})
apply(rule TrueI)
done
```

> ***Read More***
> *The function* `SUBPROOF` *is defined in* `Pure/subgoal.ML` *and also described in [Impl. Man., Sec. 4.3].*

Similar but less powerful functions than `SUBPROOF` are `SUBGOAL` and `CSUBGOAL`. They allow you to inspect a given subgoal (the former presents the subgoal as a `term`, while the latter as a `cterm`). With this you can implement a tactic that applies a rule according to the topmost logic connective in the subgoal (to illustrate this we only analyse a few connectives). The code of the tactic is as follows.

```
1 fun select_tac (t, i) =
2   case t of
3      @{term "Trueprop"} $ t' => select_tac (t', i)
4    | @{term "op ⟹"} $ _ $ t' => select_tac (t', i)
5    | @{term "op ∧"} $ _ $ _ => rtac @{thm conjI} i
6    | @{term "op ⟶"} $ _ $ _ => rtac @{thm impI} i
7    | @{term "Not"} $ _ => rtac @{thm notI} i
8    | Const (@{const_name "All"}, _) $ _ => rtac @{thm allI} i
9    | _ => all_tac
```

The input of the function is a term representing the subgoal and a number specifying the subgoal of interest. In Line 3 you need to descend under the outermost `Trueprop` in order to get to the connective you like to analyse. Otherwise goals like `A ∧ B` are not properly analysed. Similarly with meta-implications in the next line. While for the first five patterns we can use the `@term`-antiquotation to construct the patterns, the pattern in Line 8 cannot be constructed in this way. The reason is that an antiquotation would fix the type of the quantified variable. So you really have to construct the pattern using the basic term-constructors. This is not necessary in other cases, because their type is always fixed to function types involving only the type `bool`. (See Section 2.6 about constructing terms manually.) For the catch-all pattern, we chose to just return `all_tac`. Consequently, `select_tac` never fails.

Let us now see how to apply this tactic. Consider the four goals:

```
lemma shows "A ∧ B" and "A ⟶ B ⟶C" and "∀x. D x" and "E ⟹ F"
apply(tactic {* SUBGOAL select_tac 4 *})
apply(tactic {* SUBGOAL select_tac 3 *})
apply(tactic {* SUBGOAL select_tac 2 *})
apply(tactic {* SUBGOAL select_tac 1 *})
```

```
goal (5 subgoals):
 1. A
 2. B
 3. A ⟹ B ⟶ C
 4. ⋀x. D x
 5. E ⟹ F
```

where in all but the last the tactic applied an introduction rule. Note that we applied the tactic to the goals in "reverse" order. This is a trick in order to be independent from the subgoals that are produced by the rule. If we had applied it in the other order

```
lemma shows "A ∧ B" and "A ⟶ B ⟶C" and "∀x. D x" and "E ⟹ F"
apply(tactic {* SUBGOAL select_tac 1 *})
apply(tactic {* SUBGOAL select_tac 3 *})
apply(tactic {* SUBGOAL select_tac 4 *})
apply(tactic {* SUBGOAL select_tac 5 *})
```

then we have to be careful to not apply the tactic to the two subgoals produced by the first goal. To do this can result in quite messy code. In contrast, the "reverse application" is easy to implement.

Of course, this example is contrived: there are much simpler methods available in Isabelle for implementing a proof procedure analysing a goal according to its topmost connective. These simpler methods use tactic combinators, which we will explain in the next section.

4.3 Tactic Combinators

The purpose of tactic combinators is to build compound tactics out of smaller tactics. In the previous section we already used *THEN*, which just strings together two tactics in a sequence. For example:

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

```
goal (3 subgoals):
 1. Foo
 2. Bar
 3. False
```

If you want to avoid the hard-coded subgoal addressing, then you can use the "primed" version of *THEN*. For example:

```
lemma shows "(Foo ∧ Bar) ∧ False"
apply(tactic {* (rtac @{thm conjI} THEN' rtac @{thm conjI}) 1 *})
```

```
goal (3 subgoals):
 1. Foo
 2. Bar
 3. False
```

Here you only have to specify the subgoal of interest only once and it is consistently applied to the component tactics. For most tactic combinators such a "primed" version exists and in what follows we will usually prefer it over the "unprimed" one.

If there is a list of tactics that should all be tried out in sequence, you can use the combinator `EVERY'`. For example the function `foo_tac'` from page 48 can also be written as:

```
val foo_tac'' = EVERY' [etac @{thm disjE}, rtac @{thm disjI2},
                        atac, rtac @{thm disjI1}, atac]
```

There is even another way of implementing this tactic: in automatic proof procedures (in contrast to tactics that might be called by the user) there are often long lists of tactics that are applied to the first subgoal. Instead of writing the code above and then calling `foo_tac'' 1`, you can also just write

```
val foo_tac1 = EVERY1 [etac @{thm disjE}, rtac @{thm disjI2},
                       atac, rtac @{thm disjI1}, atac]
```

and call `foo_tac1`.

With the combinators `THEN'`, `EVERY'` and `EVERY1` it must be guaranteed that all component tactics successfully apply; otherwise the whole tactic will fail. If you rather want to try out a number of tactics, then you can use the combinator `ORELSE'` for two tactics, and `FIRST'` (or `FIRST1`) for a list of tactics. For example, the tactic

```
val orelse_xmp = rtac @{thm disjI1} ORELSE' rtac @{thm conjI}
```

will first try out whether rule `disjI` applies and after that `conjI`. To see this consider the proof

```
lemma shows "True ∧ False" and "Foo ∨ Bar"
apply(tactic {* orelse_xmp 2 *})
apply(tactic {* orelse_xmp 1 *})
```

which results in the goal state

```
goal (3 subgoals):
 1. True
 2. False
 3. Foo
```

Using `FIRST'` we can simplify our `select_tac` from Page 57 as follows:

```
val select_tac' = FIRST' [rtac @{thm conjI}, rtac @{thm impI},
                          rtac @{thm notI}, rtac @{thm allI}, K all_tac]
```

Since we like to mimic the behaviour of `select_tac` as closely as possible, we must include `all_tac` at the end of the list, otherwise the tactic will fail if no rule applies (we also have to wrap `all_tac` using the `K`-combinator, because it does not take a subgoal number as argument). You can test the tactic on the same goals:

```
lemma shows "A ∧ B" and "A ⟶ B ⟶C" and "∀x. D x" and "E ⟹ F"
apply(tactic {* select_tac' 4 *})
apply(tactic {* select_tac' 3 *})
apply(tactic {* select_tac' 2 *})
apply(tactic {* select_tac' 1 *})
```

```
goal (5 subgoals):
 1. A
 2. B
 3. A ⟹ B ⟶ C
 4. ⋀x. D x
 5. E ⟹ F
```

Since such repeated applications of a tactic to the reverse order of *all* subgoals is quite common, there is the tactic combinator *ALLGOALS* that simplifies this. Using this combinator you can simply write:

lemma shows "$A \wedge B$" **and** "$A \longrightarrow B \longrightarrow C$" **and** "$\forall x.\ D\ x$" **and** "$E \Longrightarrow F$"
apply(tactic {* ALLGOALS select_tac' *})

```
goal (5 subgoals):
 1. A
 2. B
 3. A ⟹ B ⟶ C
 4. ⋀x. D x
 5. E ⟹ F
```

Remember that we chose to implement `select_tac'` so that it always succeeds. This can be potentially very confusing for the user, for example, in cases where the goal is the form

lemma shows "$E \Longrightarrow F$"
apply(tactic {* select_tac' 1 *})

```
goal (1 subgoal):
 1. E ⟹ F
```

In this case no rule applies. The problem for the user is that there is little chance to see whether or not progress in the proof has been made. By convention therefore, tactics visible to the user should either change something or fail.

To comply with this convention, we could simply delete the `K all_tac` from the end of the theorem list. As a result `select_tac'` would only succeed on goals where it can make progress. But for the sake of argument, let us suppose that this deletion is *not* an option. In such cases, you can use the combinator *CHANGED* to make sure the subgoal has been changed by the tactic. Because now

lemma shows "$E \Longrightarrow F$"
apply(tactic {* CHANGED (select_tac' 1) *})

gives the error message:

```
*** empty result sequence -- proof command failed
*** At command "apply".
```

We can further extend `select_tac'` so that it not just applies to the topmost connective, but also to the ones immediately "underneath", i.e. analyse the goal completely. For this you can use the tactic combinator *REPEAT*. As an example suppose the following tactic

```
val repeat_xmp = REPEAT (CHANGED (select_tac' 1))
```

which applied to the proof

lemma shows "$((\neg A) \wedge (\forall x.\ B\ x)) \wedge (C \longrightarrow D)$"
apply(tactic {* repeat_xmp *})

produces

```
goal (3 subgoals):
 1. A ⟹ False
 2. ∀x. B x
 3. C ⟶ D
```

Here it is crucial that *select_tac'* is prefixed with *CHANGED*, because otherwise *REPEAT* runs into an infinite loop (it applies the tactic as long as it succeeds). The function *REPEAT1* is similar, but runs the tactic at least once (failing if this is not possible).

If you are after the "primed" version of *repeat_xmp*, then you need to implement it as

```
val repeat_xmp' = REPEAT o CHANGED o select_tac'
```

since there are no "primed" versions of *REPEAT* and *CHANGED*.

If you look closely at the goal state above, the tactics *repeat_xmp* and *repeat_xmp'* are not yet quite what we are after: the problem is that goals 2 and 3 are not analysed. This is because the tactic is applied repeatedly only to the first subgoal. To analyse also all resulting subgoals, you can use the tactic combinator *REPEAT_ALL_NEW*. Suppose the tactic

```
val repeat_all_new_xmp = REPEAT_ALL_NEW (CHANGED o select_tac')
```

you see that the following goal

lemma shows *"((¬A) ∧ (∀x. B x)) ∧ (C ⟶ D)"*
apply*(tactic {* repeat_all_new_xmp 1 *})*

```
goal (3 subgoals):
 1. A ⟹ False
 2. ⋀x. B x
 3. C ⟹ D
```

is completely analysed according to the theorems we chose to include in *select_tac'*.

Recall that tactics produce a lazy sequence of successor goal states. These states can be explored using the command **back**. For example

lemma *"⟦P1 ∨ Q1; P2 ∨ Q2⟧ ⟹ R"*
apply*(tactic {* etac @{thm disjE} 1 *})*

applies the rule to the first assumption yielding the goal state:

```
goal (2 subgoals):
 1. ⟦P2 ∨ Q2; P1⟧ ⟹ R
 2. ⟦P2 ∨ Q2; Q1⟧ ⟹ R
```

After typing

back

the rule now applies to the second assumption.

```
goal (2 subgoals):
 1. ⟦P1 ∨ Q1; P2⟧ ⟹ R
 2. ⟦P1 ∨ Q1; Q2⟧ ⟹ R
```

Sometimes this leads to confusing behaviour of tactics and also has the potential to explode the search space for tactics. These problems can be avoided by prefixing the tactic with the tactic combinator *DETERM*.

```
lemma "⟦P1 ∨ Q1; P2 ∨ Q2⟧ ⟹ R"
apply(tactic {* DETERM (etac @{thm disjE} 1) *})
```

```
goal (2 subgoals):
 1. ⟦P2 ∨ Q2; P1⟧ ⟹ R
 2. ⟦P2 ∨ Q2; Q1⟧ ⟹ R
```

This combinator will prune the search space to just the first successful application. Attempting to apply **back** in this goal states gives the error message:

```
*** back: no alternatives
*** At command "back".
```

> ***Read More***
> *Most tactic combinators described in this section are defined in* `Pure/tctical.ML`.

4.4 Simplifier Tactics

A lot of convenience in the reasoning with Isabelle derives from its powerful simplifier. The power of simplifier is a strength and a weakness at the same time, because you can easily make the simplifier to run into a loop and its behaviour can be difficult to predict. There is also a multitude of options that you can configurate to control the behaviour of the simplifier. We describe some of them in this and the next section.

There are the following five main tactics behind the simplifier (in parentheses is their user-level counterpart):

simp_tac	(simp (no_asm))
asm_simp_tac	(simp (no_asm_simp))
full_simp_tac	(simp (no_asm_use))
asm_lr_simp_tac	(simp (asm_lr))
asm_full_simp_tac	(simp)

All of the tactics take a simpset and an interger as argument (the latter as usual to specify the goal to be analysed). So the proof

```
lemma "Suc (1 + 2) < 3 + 2"
apply(simp)
done
```

corresponds on the ML-level to the tactic

```
lemma "Suc (1 + 2) < 3 + 2"
apply(tactic {* asm_full_simp_tac @{simpset} 1 *})
done
```

If the simplifier cannot make any progress, then it leaves the goal unchanged, i.e. does not raise any error message. That means if you use it to unfold a definition for a constant and this constant is not present in the goal state, you can still safely apply the simplifier.

When using the simplifier, the crucial information you have to provide is the simpset. If this information is not handled with care, then the simplifier can easily run into a loop. Therefore a good rule of thumb is to use simpsets that are as minimal as possible. It might be surprising that a simpset is more complex than just a simple collection of theorems used as simplification rules. One reason for the complexity is that the simplifier must be able to rewrite inside terms and should also be able to rewrite according to rules that have preconditions.

The rewriting inside terms requires congruence rules, which are meta-equalities typical of the form

$$\frac{t_1 \equiv s_1 \;\ldots\; t_n \equiv s_n}{constr\ t_1 \ldots t_n \equiv constr\ s_1 \ldots s_n}$$

with *constr* being a term-constructor, like *If* or *Let*. Every simpset contains only one concruence rule for each term-constructor, which however can be overwritten. The user can declare lemmas to be congruence rules using the attribute *[cong]*. In HOL, the user usually states these lemmas as equations, which are then internally transformed into meta-equations.

The rewriting with rules involving preconditions requires what is in Isabelle called a subgoaler, a solver and a looper. These can be arbitrary tactics that can be installed in a simpset and which are called during various stages during simplification. However, simpsets also include simprocs, which can produce rewrite rules on demand (see next section). Another component are split-rules, which can simplify for example the "then" and "else" branches of if-statements under the corresponding preconditions.

> ***Read More***
> *For more information about the simplifier see* `Pure/meta_simplifier.ML` *and* `Pure/simplifier.ML`*. The simplifier for HOL is set up in* `HOL/Tools/simpdata.ML`*. Generic splitters are implemented in* `Provers/splitter.ML`*.*

> ***Read More***
> *FIXME: Find the right place Discrimination nets are implemented in* `Pure/net.ML`*.*

The most common combinators to modify simpsets are

```
addsimps      delsimps
addcongs      delcongs
addsimprocs   delsimprocs
```

(FIXME: What about splitters? `addsplits`, `delsplits`)

To see how they work, consider the function in Figure 4.3, which prints out some parts of a simpset. If you use it to print out the components of the empty simpset, i.e. `empty_ss`

```
fun print_ss ctxt ss =
let
  val {simps, congs, procs, ...} = MetaSimplifier.dest_ss ss

  fun name_thm (nm, thm) =
      "  " ^ nm ^ ": " ^ (str_of_thm ctxt thm)
  fun name_ctrm (nm, ctrm) =
      "  " ^ nm ^ ": " ^ (str_of_cterms ctxt ctrm)

  val s1 = ["Simplification rules:"]
  val s2 = map name_thm simps
  val s3 = ["Congruences rules:"]
  val s4 = map name_thm congs
  val s5 = ["Simproc patterns:"]
  val s6 = map name_ctrm procs
in
  (s1 @ s2 @ s3 @ s4 @ s5 @ s6)
     |> separate "\n"
     |> implode
     |> warning
end
```

Figure 4.3: The function `MetaSimplifier.dest_ss` returns a record containing all printable information stored in a simpset. We are here only interested in the simplification rules, congruence rules and simprocs.

```
print_ss @{context} empty_ss
> Simplification rules:
> Congruences rules:
> Simproc patterns:
```

you can see it contains nothing. This simpset is usually not useful, except as a building block to build bigger simpsets. For example you can add to `empty_ss` the simplification rule `Diff_Int` as follows:

```
val ss1 = empty_ss addsimps [@{thm Diff_Int} RS @{thm eq_reflection}]
```

Printing then out the components of the simpset gives:

```
print_ss @{context} ss1
> Simplification rules:
>   ??.unknown: A - B ∩ C ≡ A - B ∪ (A - C)
> Congruences rules:
> Simproc patterns:
```

(FIXME: Why does it print out ??.unknown)

Adding also the congruence rule `UN_cong`

```
val ss2 = ss1 addcongs [@{thm UN_cong} RS @{thm eq_reflection}]
```

gives

```
print_ss @{context} ss2
> Simplification rules:
>   ??.unknown: A - B ∩ C ≡ A - B ∪ (A - C)
> Congruences rules:
>   UNION: ⟦A = B; ⋀x. x ∈ B ⟹ C x = D x⟧ ⟹ ⋃x∈A. C x ≡ ⋃x∈B. D x
> Simproc patterns:
```

Notice that we had to add these lemmas as meta-equations. The *empty_ss* expects this form of the simplification and congruence rules. However, even when adding these lemmas to *empty_ss* we do not end up with anything useful yet.

In the context of HOL, the first really useful simpset is *HOL_basic_ss*. While printing out the components of this simpset

```
print_ss @{context} HOL_basic_ss
> Simplification rules:
> Congruences rules:
> Simproc patterns:
```

also produces "nothing", the printout is misleading. In fact the *HOL_basic_ss* is setup so that it can solve goals of the form

```
True, t = t, t ≡ t and False ⟹ P;
```

and also resolve with assumptions. For example:

lemma
 "True" **and** *"t = t"* **and** *"t ≡ t"* **and** *"False ⟹ Foo"* **and** *"⟦A; B; C⟧ ⟹ A"*
apply(*tactic {* ALLGOALS (simp_tac HOL_basic_ss) *}*)
done

This behaviour is not because of simplification rules, but how the subgoaler, solver and looper are set up in *HOL_basic_ss*.

The simpset *HOL_ss* is an extention of *HOL_basic_ss* containing already many useful simplification and congruence rules for the logical connectives in HOL.

```
print_ss @{context} HOL_ss
> Simplification rules:
>   Pure.triv_forall_equality: (⋀x. PROP V) ≡ PROP V
>   HOL.the_eq_trivial: THE x. x = y ≡ y
>   HOL.the_sym_eq_trivial: THE ya. y = ya ≡ y
>   ...
> Congruences rules:
>   HOL.simp_implies: ...
```

```
>     ⟹ (PROP P =simp=> PROP Q) ≡ (PROP P' =simp=> PROP Q')
>   op -->: ⟦P ≡ P'; P' ⟹ Q ≡ Q'⟧ ⟹ P ⟶ Q ≡ P' ⟶ Q'
> Simproc patterns:
>   ...
```

The simplifier is often used to unfold definitions in a proof. For this the simplifier contains the *rewrite_goals_tac*. Suppose for example the definition

```
definition "MyTrue ≡ True"
```

then in the following proof we can unfold this constant

```
lemma shows "MyTrue ⟹ True ∧ True"
apply(rule conjI)
apply(tactic {* rewrite_goals_tac [@{thm MyTrue_def}] *})
```

producing the goal state

```
goal (2 subgoals):
 1. True ⟹ True
 2. True ⟹ True
```

As you can see, the tactic unfolds the definitions in all subgoals.

The simplifier is often used in order to bring terms into a normal form. Unfortunately, often the situation arises that the corresponding simplification rules will cause the simplifier to run into an infinite loop. Consider for example the simple theory about permutations over natural numbers shown in Figure 4.4. The purpose of the lemmas is to push permutations as far inside as possible, where they might disappear by Lemma *perm_rev*. However, to fully normalise all instances, it would be desirable to add also the lemma *perm_compose* to the simplifier for pushing permutations over other permutations. Unfortunately, the right-hand side of this lemma is again an instance of the left-hand side and so causes an infinite loop. The seems to be no easy way to reformulate this rule and so one ends up with clunky proofs like:

```
lemma
  fixes c d::"nat" and pi1 pi2::"prm"
  shows "pi1·(c, pi2·((rev pi1)·d)) = (pi1·c, (pi1·pi2)·d)"
apply(simp)
apply(rule trans)
apply(rule perm_compose)
apply(simp)
done
```

It is however possible to create a single simplifier tactic that solves such proofs. The trick is to introduce an auxiliary constant for permutations and split the simplification into two phases (below actually three). Let assume the auxiliary constant is

```
definition
  perm_aux :: "prm ⇒ 'a ⇒ 'a" ("_ ·aux _" [80,80] 80)
where
  "pi ·aux c ≡ pi · c"
```

Now the two lemmas

```
lemma perm_aux_expand:
```

```
types  prm = "(nat × nat) list"
consts perm :: "prm ⇒ 'a ⇒ 'a"  ("_ • _" [80,80] 80)

primrec (perm_nat)
 "[]•c = c"
 "(ab#pi)•c = (if (pi•c)=fst ab then snd ab
               else (if (pi•c)=snd ab then fst ab else (pi•c)))"

primrec (perm_prod)
 "pi•(x, y) = (pi•x, pi•y)"

primrec (perm_list)
 "pi•[] = []"
 "pi•(x#xs) = (pi•x)#(pi•xs)"

lemma perm_append[simp]:
  fixes c::"nat" and pi1 pi2::"prm"
  shows "((pi1@pi2)•c) = (pi1•(pi2•c))"
by (induct pi1) (auto)

lemma perm_eq[simp]:
  fixes c::"nat" and pi::"prm"
  shows "(pi•c = pi•d) = (c = d)"
by (induct pi) (auto)

lemma perm_rev[simp]:
  fixes c::"nat" and pi::"prm"
  shows "pi•((rev pi)•c) = c"
by (induct pi arbitrary: c) (auto)

lemma perm_compose:
  fixes c::"nat" and pi1 pi2::"prm"
  shows "pi1•(pi2•c) = (pi1•pi2)•(pi1•c)"
by (induct pi2) (auto)
```

Figure 4.4: A simple theory about permutations over `nat`. The point is that the lemma *perm_compose* cannot be directly added to the simplifier, as it would cause the simplifier to loop. It can still be used as a simplification rule if the permutation is sufficiently protected. (FIXME: Uses old primrec.)

```
  fixes c::"nat" and pi1 pi2::"prm"
  shows "pi1·(pi2·c) = pi1 ·aux (pi2·c)"
unfolding perm_aux_def by (rule refl)

lemma perm_compose_aux:
  fixes c::"nat" and pi1 pi2::"prm"
  shows "pi1·(pi2·aux c) = (pi1·pi2) ·aux (pi1·c)"
unfolding perm_aux_def by (rule perm_compose)
```

are simple consequence of the definition and *perm_compose*. More importantly, the lemma *perm_compose_aux* can be safely added to the simplifier, because now the right-hand side is not anymore an instance of the left-hand side. In a sense it freezes all redexes of permutation compositions after one step. In this way, we can split simplification of permutations into three phases without the user not noticing anything about the auxiliary contant. We first freeze any instance of permutation compositions in the term using lemma *"perm_aux_expand"* (Line 9); then simplifiy all other permutations including pusing permutations over other permutations by rule *perm_compose_aux* (Line 10); and finally “unfreeze” all instances of permutation compositions by unfolding the definition of the auxiliary constant.

```
1  val perm_simp_tac =
2  let
3    val thms1 = [@{thm perm_aux_expand}]
4    val thms2 = [@{thm perm_append}, @{thm perm_eq}, @{thm perm_rev},
5                 @{thm perm_compose_aux}] @ @{thms perm_prod.simps} @
6                 @{thms perm_list.simps} @ @{thms perm_nat.simps}
7    val thms3 = [@{thm perm_aux_def}]
8  in
9    simp_tac (HOL_basic_ss addsimps thms1)
10   THEN' simp_tac (HOL_basic_ss addsimps thms2)
11   THEN' simp_tac (HOL_basic_ss addsimps thms3)
12 end
```

For all three phases we have to build simpsets addig specific lemmas. As is sufficient for our purposes here, we can add these lemma to *HOL_basic_ss* in order to obtain the desired results. Now we can solve the following lemma

```
lemma
  fixes c d::"nat" and pi1 pi2::"prm"
  shows "pi1·(c, pi2·((rev pi1)·d)) = (pi1·c, (pi1·pi2)·d)"
apply(tactic {* perm_simp_tac 1 *})
done
```

in one step. This tactic can deal with most instances of normalising permutations; in order to solve all cases we have to deal with corner-cases such as the lemma being an exact instance of the permutation composition lemma. This can often be done easier by implementing a simproc or a conversion. Both will be explained in the next two chapters.

(FIXME: Is it interesting to say something about *op =simp=>*?)

(FIXME: What are the second components of the congruence rules—something to do with weak congruence constants?)

(FIXME: Anything interesting to say about *Simplifier.clear_ss*?)

(FIXME: *ObjectLogic.full_atomize_tac*, *ObjectLogic.rulify_tac*)

4.5 Simprocs

In Isabelle you can also implement custom simplification procedures, called *simprocs*. Simprocs can be triggered by the simplifier on a specified term-pattern and rewrite a term according to a theorem. They are useful in cases where a rewriting rule must be produced on "demand" or when rewriting by simplification is too unpredictable and potentially loops.

To see how simprocs work, let us first write a simproc that just prints out the pattern which triggers it and otherwise does nothing. For this you can use the function:

```
1 fun fail_sp_aux simpset redex =
2 let
3   val ctxt = Simplifier.the_context simpset
4   val _ = warning ("The redex: " ^ (str_of_cterm ctxt redex))
5 in
6   NONE
7 end
```

This function takes a simpset and a redex (a *cterm*) as arguments. In Lines 3 and 4, we first extract the context from the given simpset and then print out a message containing the redex. The function returns *NONE* (standing for an optional *thm*) since at the moment we are *not* interested in actually rewriting anything. We want that the simproc is triggered by the pattern *Suc n*. This can be done by adding the simproc to the current simpset as follows

simproc_setup *fail_sp ("Suc n") = {* K fail_sp_aux *}*

where the second argument specifies the pattern and the right-hand side contains the code of the simproc (we have to use *K* since we ignoring an argument about morphisms[1]). After this, the simplifier is aware of the simproc and you can test whether it fires on the lemma:

lemma shows *"Suc 0 = 1"*
apply*(simp)*

This will print out the message twice: once for the left-hand side and once for the right-hand side. The reason is that during simplification the simplifier will at some point reduce the term *1* to *Suc 0*, and then the simproc "fires" also on that term.

We can add or delete the simproc from the current simpset by the usual **declare**-statement. For example the simproc will be deleted with the declaration

declare *[[simproc del: fail_sp]]*

[1] FIXME: what does the morphism do?

If you want to see what happens with just *this* simproc, without any interference from other rewrite rules, you can call `fail_sp` as follows:

lemma shows *"Suc 0 = 1"*
apply*(tactic {* simp_tac (HOL_basic_ss addsimprocs [@{simproc fail_sp}]) 1*})*

Now the message shows up only once since the term *1* is left unchanged.

Setting up a simproc using the command **simproc_setup** will always add automatically the simproc to the current simpset. If you do not want this, then you have to use a slightly different method for setting up the simproc. First the function `fail_sp_aux` needs to be modified to

```
fun fail_sp_aux' simpset redex =
let
  val ctxt = Simplifier.the_context simpset
  val _ = warning ("The redex: " ^ (Syntax.string_of_term ctxt redex))
in
  NONE
end
```

Here the redex is given as a `term`, instead of a `cterm` (therefore we printing it out using the function `string_of_term`). We can turn this function into a proper simproc using the function `Simplifier.simproc_i`:

```
val fail_sp' =
let
  val thy = @{theory}
  val pat = [@{term "Suc n"}]
in
  Simplifier.simproc_i thy "fail_sp'" pat (K fail_sp_aux')
end
```

Here the pattern is given as `term` (instead of `cterm`). The function also takes a list of patterns that can trigger the simproc. Now the simproc is set up and can be explicitly added using `addsimprocs` to a simpset whenerver needed.

Simprocs are applied from inside to outside and from left to right. You can see this in the proof

lemma shows *"Suc (Suc 0) = (Suc 1)"*
apply*(tactic {* simp_tac (HOL_basic_ss addsimprocs [fail_sp']) 1*})*

The simproc `fail_sp'` prints out the sequence

```
> Suc 0
> Suc (Suc 0)
> Suc 1
```

To see how a simproc applies a theorem, let us implement a simproc that rewrites terms according to the equation:

lemma *plus_one:*

shows *"Suc n ≡ n + 1"* **by** *simp*

Simprocs expect that the given equation is a meta-equation, however the equation can contain preconditions (the simproc then will only fire if the preconditions can be solved). To see that one has relatively precise control over the rewriting with simprocs, let us further assume we want that the simproc only rewrites terms "greater" than *Suc 0*. For this we can write

```
fun plus_one_sp_aux ss redex =
  case redex of
    @{term "Suc 0"} => NONE
  | _ => SOME @{thm plus_one}
```

and set up the simproc as follows.

```
val plus_one_sp =
let
  val thy = @{theory}
  val pat = [@{term "Suc n"}]
in
  Simplifier.simproc_i thy "sproc +1" pat (K plus_one_sp_aux)
end
```

Now the simproc is set up so that it is triggered by terms of the form *Suc n*, but inside the simproc we only produce a theorem if the term is not *Suc 0*. The result you can see in the following proof

lemma shows *"P (Suc (Suc (Suc 0))) (Suc 0)"*
apply*(tactic {* simp_tac (HOL_basic_ss addsimprocs [plus_one_sp]) 1*})*

where the simproc produces the goal state

```
goal (1 subgoal):
 1. P (Suc 0 + 1 + 1) (Suc 0)
```

As usual with rewriting you have to worry about looping: you already have a loop with *plus_one_sp*, if you apply it with the default simpset (because the default simpset contains a rule which just does the opposite of *plus_one_sp*, namely rewriting *"+ 1"* to a successor). So you have to be careful in choosing the right simpset to which you add a simproc.

Next let us implement a simproc that replaces terms of the form *Suc n* with the number *n* increase by one. First we implement a function that takes a term and produces the corresponding integer value.

```
fun dest_suc_trm ((Const (@{const_name "Suc"}, _)) $ t) = 1 + dest_suc_trm t
  | dest_suc_trm t = snd (HOLogic.dest_number t)
```

It uses the library function *dest_number* that transforms (Isabelle) terms, like *0*, *1*, *2* and so on, into integer values. This function raises the exception *TERM*, if the term is not a number. The next function expects a pair consisting of a term *t* (containing *Suc*s) and the corresponding integer value *n*.

```
1 fun get_thm ctxt (t, n) =
2 let
3   val num = HOLogic.mk_number @{typ "nat"} n
4   val goal = Logic.mk_equals (t, num)
5 in
6   Goal.prove ctxt [] [] goal (K (arith_tac ctxt 1))
7 end
```

From the integer value it generates the corresponding number term, called *num* (Line 3), and then generates the meta-equation $t \equiv num$ (Line 4), which it proves by the arithmetic tactic in Line 6.

For our purpose at the moment, proving the meta-equation using *arith_tac* is fine, but there is also an alternative employing the simplifier with a very restricted simpset. For the kind of lemmas we want to prove, the simpset *num_ss* in the code will suffice.

```
fun get_thm_alt ctxt (t, n) =
let
  val num = HOLogic.mk_number @{typ "nat"} n
  val goal = Logic.mk_equals (t, num)
  val num_ss = HOL_ss addsimps [@{thm One_nat_def}, @{thm Let_def}] @
          @{thms nat_number} @ @{thms neg_simps} @ @{thms plus_nat.simps}
in
  Goal.prove ctxt [] [] goal (K (simp_tac num_ss 1))
end
```

The advantage of *get_thm_alt* is that it leaves very little room for something to go wrong; in contrast it is much more difficult to predict what happens with *arith_tac*, especially in more complicated circumstances. The disatvantage of *get_thm_alt* is to find a simpset that is sufficiently powerful to solve every instance of the lemmas we like to prove. This requires careful tuning, but is often necessary in "production code".[2]

Anyway, either version can be used in the function that produces the actual theorem for the simproc.

```
fun nat_number_sp_aux ss t =
let
  val ctxt = Simplifier.the_context ss
in
  SOME (get_thm ctxt (t, dest_suc_trm t))
  handle TERM _ => NONE
end
```

This function uses the fact that *dest_suc_trm* might throw an exception *TERM*. In this case there is nothing that can be rewritten and therefore no theorem is produced

[2] It would be of great help if there is another way than tracing the simplifier to obtain the lemmas that are successfully applied during simplification. Alas, there is none.

(i.e. the function returns *NONE*). To try out the simproc on an example, you can set it up as follows:

```
val nat_number_sp =
let
  val thy = @{theory}
  val pat = [@{term "Suc n"}]
in
  Simplifier.simproc_i thy "nat_number" pat (K nat_number_sp_aux)
end
```

Now in the lemma

lemma *"P (Suc (Suc 2)) (Suc 99) (0::nat) (Suc 4 + Suc 0) (Suc (0 + 0))"*
apply*(tactic {* simp_tac (HOL_ss addsimprocs [nat_number_sp]) 1*})*

you obtain the more legible goal state

```
goal (1 subgoal):
 1. P 4 100 0 (5 + 1) (Suc (0 + 0))
```

where the simproc rewrites all *Suc*s except in the last argument. There it cannot rewrite anything, because it does not know how to transform the term *Suc (0 + 0)* into a number. To solve this problem have a look at the next exercise.

Exercise 4.5.1. *Write a simproc that replaces terms of the form t_1 + t_2 by their result. You can assume the terms are "proper" numbers, that is of the form 0, 1, 2 and so on.*

(FIXME: We did not do anything with morphisms. Anything interesting one can say about them?)

4.6 Conversions

Conversions are a thin layer on top of Isabelle's inference kernel, and can be viewed as a controllable, bare-bone version of Isabelle's simplifier. One difference between conversions and the simplifier is that the former act on *cterm*s while the latter acts on *thm*s. However, we will also show in this section how conversions can be applied to theorems via tactics. The type for conversions is

```
type conv = cterm -> thm
```

whereby the produced theorem is always a meta-equality. A simple conversion is the function *Conv.all_conv*, which maps a *cterm* to an instance of the (meta)reflexivity theorem. For example:

```
Conv.all_conv @{cterm "Foo ∨ Bar"}
> Foo ∨ Bar ≡ Foo ∨ Bar
```

Another simple conversion is *Conv.no_conv* which always raises the exception *CTERM*.

```
Conv.no_conv @{cterm True}
> *** Exception- CTERM ("no conversion", []) raised
```

A more interesting conversion is the function *Thm.beta_conversion*: it produces a meta-equation between a term and its beta-normal form. For example

```
let
  val add = @{cterm "λx y. x + (y::nat)"}
  val two = @{cterm "2::nat"}
  val ten = @{cterm "10::nat"}
in
  Thm.beta_conversion true (Thm.capply (Thm.capply add two) ten)
end
> ((λx y. x + y) 2) 10 ≡ 2 + 10
```

Note that the actual response in this example is *2 + 10 ≡ 2 + 10*, since the pretty-printer for *cterm*s eta-normalises terms. But how we constructed the term (using the function *Thm.capply*, which is the application *$* for *cterm*s) ensures that the left-hand side must contain beta-redexes. Indeed if we obtain the "raw" representation of the produced theorem, we can see the difference:

```
let
  val add = @{cterm "λx y. x + (y::nat)"}
  val two = @{cterm "2::nat"}
  val ten = @{cterm "10::nat"}
  val thm = Thm.beta_conversion true (Thm.capply (Thm.capply add two) ten)
in
  #prop (rep_thm thm)
end
> Const ("==",...) $
>   (Abs ("x",...,Abs ("y",...,...)) $...$...) $
>     (Const ("HOL.plus_class.plus",...) $ ... $ ...)
```

The argument *true* in *Thm.beta_conversion* indicates that the right-hand side will be fully beta-normalised. If instead *false* is given, then only a single beta-reduction is performed on the outer-most level. For example

```
let
  val add = @{cterm "λx y. x + (y::nat)"}
  val two = @{cterm "2::nat"}
in
  Thm.beta_conversion false (Thm.capply add two)
end
> ((λx y. x + y) 2) ≡ λy. 2 + y
```

Again, we actually see as output only the fully eta-normalised term.

The main point of conversions is that they can be used for rewriting *cterms*. To do this you can use the function *Conv.rewr_conv*, which expects a meta-equation as an argument. Suppose we want to rewrite a *cterm* according to the meta-equation:

lemma *true_conj1: "True ∧ P ≡ P"* **by** *simp*

You can see how this function works in the example rewriting *True ∧ (Foo ⟶ Bar)* to *Foo ⟶ Bar*.

```
let
  val ctrm = @{cterm "True ∧ (Foo ⟶ Bar)"}
in
  Conv.rewr_conv @{thm true_conj1} ctrm
end
> True ∧ (Foo ⟶ Bar) ≡ Foo ⟶ Bar
```

Note, however, that the function *Conv.rewr_conv* only rewrites the outer-most level of the *cterm*. If the given *cterm* does not match exactly the left-hand side of the theorem, then *Conv.rewr_conv* raises the exception *CTERM*.

This very primitive way of rewriting can be made more powerful by combining several conversions into one. For this you can use conversion combinators. The simplest conversion combinator is *then_conv*, which applies one conversion after another. For example

```
let
  val conv1 = Thm.beta_conversion false
  val conv2 = Conv.rewr_conv @{thm true_conj1}
  val ctrm = Thm.capply @{cterm "λx. x ∧ False"} @{cterm "True"}
in
  (conv1 then_conv conv2) ctrm
end
> (λx. x ∧ False) True ≡ False
```

where we first beta-reduce the term and then rewrite according to *true_conj1*. (Recall the problem with the pretty-printer normalising all terms.)

The conversion combinator *else_conv* tries out the first one, and if it does not apply, tries the second. For example

```
let
  val conv = Conv.rewr_conv @{thm true_conj1} else_conv Conv.all_conv
  val ctrm1 = @{cterm "True ∧ Q"}
  val ctrm2 = @{cterm "P ∨ (True ∧ Q)"}
in
  (conv ctrm1, conv ctrm2)
end
> (True ∧ Q ≡ Q, P ∨ True ∧ Q ≡ P ∨ True ∧ Q)
```

Here the conversion of *true_conj1* only applies in the first case, but fails in the second. The whole conversion does not fail, however, because the combinator *Conv.else_conv* will then try out *Conv.all_conv*, which always succeeds.

The conversion combinator `Conv.try_conv` constructs a conversion which is tried out on a term, but in case of failure just does nothing. For example

```
Conv.try_conv (Conv.rewr_conv @{thm true_conj1}) @{cterm "True ∨ P"}
> True ∨ P ≡ True ∨ P
```

Apart from the function `beta_conversion`, which is able to fully beta-normalise a term, the conversions so far are restricted in that they only apply to the outermost level of a `cterm`. In what follows we will lift this restriction. The combinator `Conv.arg_conv` will apply the conversion to the first argument of an application, that is the term must be of the form `t1 $ t2` and the conversion is applied to `t2`. For example

```
let
  val conv = Conv.rewr_conv @{thm true_conj1}
  val ctrm = @{cterm "P ∨ (True ∧ Q)"}
in
  Conv.arg_conv conv ctrm
end
> P ∨ (True ∧ Q) ≡ P ∨ Q
```

The reason for this behaviour is that `(op ∨)` expects two arguments. Therefore the term must be of the form `(Const ... $ t1) $ t2`. The conversion is then applied to `t2` which in the example above stands for `True ∧ Q`. The function `Conv.fun_conv` applies the conversion to the first argument of an application.

The function `Conv.abs_conv` applies a conversion under an abstractions. For example:

```
let
  val conv = K (Conv.rewr_conv @{thm true_conj1})
  val ctrm = @{cterm "λP. True ∧ P ∧ Foo"}
in
  Conv.abs_conv conv @{context} ctrm
end
> λP. True ∧ P ∧ Foo ≡ λP. P ∧ Foo
```

Note that this conversion needs a context as an argument. The conversion that goes under an application is `Conv.combination_conv`. It expects two conversions as arguments, each of which is applied to the corresponding "branch" of the application.

We can now apply all these functions in a conversion that recursively descends a term and applies a "`true_conj1`"-conversion in all possible positions.

```
fun all_true1_conv ctxt ctrm =
  case (Thm.term_of ctrm) of
    @{term "op ∧"} $ @{term True} $ _ =>
      (Conv.arg_conv (all_true1_conv ctxt) then_conv
```

```
5         Conv.rewr_conv @{thm true_conj1}) ctrm
6   | _ $ _ => Conv.combination_conv
7                (all_true1_conv ctxt) (all_true1_conv ctxt) ctrm
8   | Abs _ => Conv.abs_conv (fn (_, ctxt) => all_true1_conv ctxt) ctxt ctrm
9   | _ => Conv.all_conv ctrm
```

This function "fires" if the terms is of the form `True ∧ ...`; it descends under applications (Line 6 and 7) and abstractions (Line 8); otherwise it leaves the term unchanged (Line 9). In Line 2 we need to transform the `cterm` into a `term` in order to be able to pattern-match the term. To see this conversion in action, consider the following example:

```
let
  val ctxt = @{context}
  val ctrm = @{cterm "distinct [1, x] ⟶ True ∧ 1 ≠ x"}
in
  all_true1_conv ctxt ctrm
end
> distinct [1, x] ⟶ True ∧ 1 ≠ x ≡ distinct [1, x] ⟶ 1 ≠ x
```

To see how much control you have about rewriting by using conversions, let us make the task a bit more complicated by rewriting according to the rule `true_conj1`, but only in the first arguments of `If`s. Then the conversion should be as follows.

```
fun if_true1_conv ctxt ctrm =
  case Thm.term_of ctrm of
    Const (@{const_name If}, _) $ _ =>
      Conv.arg_conv (all_true1_conv ctxt) ctrm
  | _ $ _ => Conv.combination_conv
                (if_true1_conv ctxt) (if_true1_conv ctxt) ctrm
  | Abs _ => Conv.abs_conv (fn (_, ctxt) => if_true1_conv ctxt) ctxt ctrm
  | _ => Conv.all_conv ctrm
```

Here is an example for this conversion:

```
let
  val ctxt = @{context}
  val ctrm =
       @{cterm "if P (True ∧ 1 ≠ 2) then True ∧ True else True ∧ False"}
in
  if_true1_conv ctxt ctrm
end
> if P (True ∧ 1 ≠ 2) then True ∧ True else True ∧ False
> ≡ if P (1 ≠ 2) then True ∧ True else True ∧ False
```

So far we only applied conversions to `cterm`s. Conversions can, however, also work on theorems using the function `Conv.fconv_rule`. As an example, consider the conversion `all_true1_conv` and the lemma:

lemma *foo_test: "P ∨ (True ∧ ¬P)"* **by** *simp*

Using the conversion you can transform this theorem into a new theorem as follows

```
Conv.fconv_rule (all_true1_conv @{context}) @{thm foo_test}
> ?P ∨ ¬ ?P
```

Finally, conversions can also be turned into tactics and then applied to goal states. This can be done with the help of the function *CONVERSION*, and also some predefined conversion combinators that traverse a goal state. The combinators for the goal state are: *Conv.params_conv* for converting under parameters (i.e. where goals are of the form $\bigwedge x.\ P \Longrightarrow Q$); the function *Conv.prems_conv* for applying a conversion to all premises of a goal, and *Conv.concl_conv* for applying a conversion to the conclusion of a goal.

Assume we want to apply *all_true1_conv* only in the conclusion of the goal, and *if_true1_conv* should only apply to the premises. Here is a tactic doing exactly that:

```
fun true1_tac ctxt = CSUBGOAL (fn (goal, i) =>
  CONVERSION
    (Conv.params_conv ~1 (fn ctxt =>
       (Conv.prems_conv ~1 (if_true1_conv ctxt) then_conv
          Conv.concl_conv ~1 (all_true1_conv ctxt))) ctxt) i)
```

We call the conversions with the argument *~1*. This is to analyse all parameters, premises and conclusions. If we call them with a non-negative number, say *n*, then these conversions will only be called on *n* premises (similar for parameters and conclusions). To test the tactic, consider the proof

```
lemma
  "if True ∧ P then P else True ∧ False ⟹
  (if True ∧ Q then True ∧ Q else P) ⟶ True ∧ (True ∧ Q)"
apply(tactic {* true1_tac @{context} 1 *})
```

where the tactic yields the goal state

```
goal (1 subgoal):
 1. if P then P else True ∧ False ⟹ (if Q then Q else P) ⟶ Q
```

As you can see, the premises are rewritten according to *if_true1_conv*, while the conclusion according to *all_true1_conv*.

To sum up this section, conversions are not as powerful as the simplifier and simprocs; the advantage of conversions, however, is that you never have to worry about non-termination.

Exercise 4.6.1. *Write a tactic that does the same as the simproc in exercise 4.5.1, but is based in conversions. That means replace terms of the form* t_1 + t_2 *by their result. You can make the same assumptions as in 4.5.1.*

Exercise 4.6.2. *Compare your solutions of Exercises 4.5.1 and 4.6.1, and try to determine which way of rewriting such terms is faster. For this you might have to construct quite large terms. Also see Recipe A.3 for information about timing.*

Read More
See `Pure/conv.ML` *for more information about conversion combinators. Further conversions are defined in* `Pure/thm.ML`*,* `Pure/drule.ML` *and* `Pure/meta_simplifier.ML`*.*

(FIXME: check whether `Pattern.match_rew` and `Pattern.rewrite_term` are of any use/efficient)

4.7 Structured Proofs (TBD)

TBD

```
lemma True
proof

  {
    fix A B C
    assume r: "A & B ⟹ C"
    assume A B
    then have "A & B" ..
    then have C by (rule r)
  }

  {
    fix A B C
    assume r: "A & B ⟹ C"
    assume A B
    note conjI [OF this]
    note r [OF this]
  }
oops
```

```
  fun prop ctxt s =
  Thm.cterm_of (ProofContext.theory_of ctxt) (Syntax.read_prop ctxt s)


  val ctxt0 = @{context};
  val ctxt = ctxt0;
  val (_, ctxt) = Variable.add_fixes ["A", "B", "C"] ctxt;
  val ([r], ctxt) = Assumption.add_assumes [prop ctxt "A & B ⟹ C"] ctxt;
  val (this, ctxt) = Assumption.add_assumes [prop ctxt "A", prop ctxt "B"]
ctxt;
  val this = [@{thm conjI} OF this];
  val this = r OF this;
  val this = Assumption.export false ctxt ctxt0 this
  val this = Variable.export ctxt ctxt0 [this]
```

Chapter 5

How to Write a Definitional Package (TBD)

"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 [1]

HOL is based on just a few primitive constants, like equality and implication, whose properties are described by axioms. All other concepts, such as inductive predicates, datatypes, or recursive functions have to be defined in terms of those constants, and the desired properties, for example induction theorems, or recursion equations have to be derived from the definitions by a formal proof. Since it would be very tedious for a user to define complex inductive predicates or datatypes "by hand" just using the primitive operators of higher order logic, *definitional packages* have been implemented automating such work. Thanks to those packages, the user can give a high-level specification, for example a list of introduction rules or constructors, and the package then does all the low-level definitions and proofs behind the scenes. In this chapter we explain how such a package can be implemented.

As a running example, we have chosen a rather simple package for defining inductive predicates. To keep things really 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. Instead, we will use a simpler *impredicative* (i.e. involving quantification on predicate variables) encoding of inductive predicates. 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 induction principle for inductive predicates.

5.1 Preliminaries

The user will just give a specification of an inductive predicate and expects from the package to produce a convenient reasoning infrastructure. This infrastructure needs to be derived from the definition that correspond to the specified predicate. This will roughly mean that the package has three main parts, namely:

- parsing the specification and typing the parsed input,
- making the definitions and deriving the reasoning infrastructure, and
- storing the results in the theory.

Before we start with explaining all parts, let us first give three examples showing how to define inductive predicates by hand and then also how to prove by hand important properties about them. From these examples, we will figure out a general method for defining inductive predicates. The aim in this section is *not* to write proofs that are as beautiful as possible, but as close as possible to the ML-code we will develop in later sections.

We first consider the transitive closure of a relation *R*. It is an inductive predicate characterised by the two introduction rules:

$$\frac{}{\mathit{trcl}\ R\ x\ x} \qquad \frac{R\ x\ y \qquad \mathit{trcl}\ R\ y\ z}{\mathit{trcl}\ R\ x\ z}$$

In Isabelle, the user will state for *trcl* the specification:

simple_inductive
  trcl :: "('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool"
where
  base: "trcl R x x"
| step: "trcl R x y ⟹ R y z ⟹ trcl R x z"

As said above the package has to make an appropriate definition and provide lemmas to reason about the predicate *trcl*. Since an inductively defined predicate is the least predicate closed under a collection of introduction rules, the predicate *trcl R x y* can be defined so that it holds if and only if *P x y* holds for every predicate *P* closed under the rules above. This gives rise to the definition

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"

where we quantify over the predicate *P*. We have to use the object implication ⟶ and object quantification ∀ for stating this definition (there is no other way for definitions in HOL). However, the introduction rules and induction principles should use the meta-connectives since they simplify the reasoning for the user.

With this definition, the proof of the induction principle for *trcl* is almost immediate. It suffices to convert all the meta-level connectives in the lemma to object-level connectives using the proof method *atomize* (Line 4), expand the definition of *trcl*

(Line 5 and 6), eliminate the universal quantifier contained in it (Line 7), and then solve the goal by assumption (Line 8).

```
1 lemma trcl_induct:
2   assumes "trcl R x y"
3   shows "(⋀x. P x x) ⟹ (⋀x y z. R x y ⟹ P y z ⟹ P x z) ⟹ P x y"
4 apply(atomize (full))
5 apply(cut_tac prems)
6 apply(unfold trcl_def)
7 apply(drule spec[where x=P])
8 apply(assumption)
9 done
```

The proofs for the introduction rules are slightly more complicated. For the first one, we need to prove the following lemma:

```
1 lemma trcl_base:
2   shows "trcl R x x"
3 apply(unfold trcl_def)
4 apply(rule allI impI)+
5 apply(drule spec)
6 apply(assumption)
7 done
```

We again unfold first the definition and apply introduction rules for $\forall$ and $\longrightarrow$ as often as possible (Lines 3 and 4). We then end up in the goal state:

```
goal (1 subgoal):
 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. Thus, all we have to do is to eliminate the universal quantifier in front of the first assumption (Line 5), and then solve the goal by assumption (Line 6).

Next we have to show that the second introduction rule also follows from the definition. Since this rule has premises, the proof is a bit more involved. After unfolding the definitions and applying the introduction rules for $\forall$ and $\longrightarrow$

```
lemma trcl_step:
  shows "R x y ⟹ trcl R y z ⟹ trcl R x z"
apply (unfold trcl_def)
apply (rule allI impI)+
```

we obtain the goal state

```
goal (1 subgoal):
 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
```

To see better where we are, let us explicitly name the assumptions by starting a subproof.

```
proof -
  case (goal1 P)
  have p1: "R x y" by fact
  have p2: "∀P. (∀x. P x x)
                 ⟶ (∀x y z. R x y ⟶ P y z ⟶ P x z) ⟶ P y z" by fact
  have r1: "∀x. P x x" by fact
  have r2: "∀x y z. R x y ⟶ P y z ⟶ P x z" by fact
  show "P x z"
```

The assumptions *p1* and *p2* correspond to the premises of the second introduction rule; the assumptions *r1* and *r2* correspond to the introduction rules. We apply *r2* to the goal *P x z*. In order for the assumption to be applicable as a rule, we have to eliminate the universal quantifier and turn the object-level implications into meta-level ones. This can be accomplished using the *rule_format* attribute. So we continue the proof with:

```
    apply (rule r2[rule_format])
```

This gives us two new subgoals

```
goal (2 subgoals):
 1. R x ?y
 2. P ?y z
```

which can be solved using assumptions *p1* and *p2*. The latter involves a quantifier and implications that have to be eliminated before it can be applied. To avoid potential problems with higher-order unification, we explicitly instantiate the quantifier to *P* and also match explicitly the implications with *r1* and *r2*. This gives the proof:

```
    apply(rule p1)
    apply(rule p2[THEN spec[where x=P], THEN mp, THEN mp, OF r1, OF r2])
    done
qed
```

Now we are done. It might be surprising that we are not using the automatic tactics available in Isabelle for proving this lemmas. After all *blast* would easily dispense of it.

```
lemma trcl_step_blast:
  shows "R x y ⟹ trcl R y z ⟹ trcl R x z"
apply(unfold trcl_def)
apply(blast)
done
```

Experience has shown that it is generally a bad idea to rely heavily on *blast*, *auto* and the like in automated proofs. The reason is that you do not have precise control over them (the user can, for example, declare new intro- or simplification rules that can throw automatic tactics off course) and also it is very hard to debug proofs involving automatic tactics whenever something goes wrong. Therefore if possible, automatic tactics should be avoided or sufficiently constrained.

The method of defining inductive predicates by impredicative quantification also generalises to mutually inductive predicates. The next example defines the predicates *even* and *odd* characterised by the following rules:

$$\frac{}{\textit{even } 0} \qquad \frac{\textit{odd } n}{\textit{even } (\textit{Suc } n)} \qquad \frac{\textit{even } n}{\textit{odd } (\textit{Suc } n)}$$

The user will state for this inductive definition the specification:

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

Since the predicates `even` and `odd` are mutually inductive, each corresponding definition must quantify over both predicates (we name them below `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 principles, we use exactly the same technique as in the transitive closure example, namely:

```
lemma even_induct:
  assumes "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 prems)
apply(unfold even_def)
apply(drule spec[where x=P])
apply(drule spec[where x=Q])
apply(assumption)
done
```

The only difference with the proof `trcl_induct` is that we have to instantiate here two universal quantifiers. We omit the other induction principle that has `Q n` as conclusion. The proofs of the introduction rules are also very similar to the ones in the `trcl`-example. We only show the proof of the second introduction rule.

```
lemma evenS:
  shows "odd m ⟹ even (Suc m)"
apply (unfold odd_def even_def)
apply (rule allI impI)+
proof -
  case (goal1 P Q)
  have p1: "∀P Q. P 0 ⟶ (∀m. Q m ⟶ P (Suc m))
                         ⟶ (∀m. P m ⟶ Q (Suc m)) ⟶ Q m" by fact
  have r1: "P 0" by fact
  have r2: "∀m. Q m ⟶ P (Suc m)" by fact
  have r3: "∀m. P m ⟶ Q (Suc m)" by fact
  show "P (Suc m)"
```

```
13      apply(rule r2[rule_format])
14      apply(rule p1[THEN spec[where x=P], THEN spec[where x=Q],
15                    THEN mp, THEN mp, THEN mp, OF r1, OF r2, OF r3])
16      done
17  qed
```

In Line 13, we apply the assumption *r2* (since we prove the second introduction rule). In Lines 14 and 15 we apply assumption *p1* (if the second introduction rule had more premises we have to do that for all of them). In order for this assumption to be applicable, the quantifiers need to be instantiated and then also the implications need to be resolved with the other rules.

As a final example, we define the accessible part of a relation *R* characterised by the introduction rule

$$\frac{\bigwedge y.\ R\ y\ x \Longrightarrow accpart\ R\ y}{accpart\ R\ x}$$

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

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

The proof of the induction principle is again straightforward.

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

Proving the introduction rule is a little more complicated, because the quantifier and the implication in the premise. The proof is as follows.

```
lemma accpartI:
  shows "(⋀y. R y x ⟹ accpart R y) ⟹ accpart R x"
apply (unfold accpart_def)
apply (rule allI impI)+
proof -
  case (goal1 P)
  have p1: "⋀y. R y x ⟹
                 (∀P. (∀x. (∀y. R y x ⟶ P y) ⟶ P x) ⟶ P y)" by fact
  have r1: "∀x. (∀y. R y x ⟶ P y) ⟶ P x" by fact
  show "P x"
    apply(rule r1[rule_format])
    proof -
      case (goal1 y)
      have r1_prem: "R y x" by fact
      show "P y"
        apply(rule p1[OF r1_prem, THEN spec[where x=P], THEN mp, OF r1])
      done
```

```
18   qed
19 qed
```

In Line 11, applying the assumption *r1* generates a goal state with the new local assumption *R y x*, named *r1_prem* in the proof above (Line 14). This local assumption is used to solve the goal *P y* with the help of assumption *p1*.

The point of these examples is to get a feeling what the automatic proofs should do in order to solve all inductive definitions we throw at them. This is usually the first step in writing a package. We next explain the parsing and typing part of the package.

5.2 Parsing and Typing the Specification

To be able to write down the specification in Isabelle, we have to introduce a new command (see Section 3.5). As the keyword for the new command we chose **simple_inductive**. In the package we want to support some "advanced" features: First, we want that the package can cope with specifications inside locales. For example it should be possible to declare

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

and then define the transitive closure and the accessible part as follows:

```
simple_inductive (in rel)
  trcl'
where
  base: "trcl' x x"
| step: "trcl' x y ⟹ R y z ⟹ trcl' x z"
```

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

Second, we want that the user can specify fixed parameters. Remember in the previous section we stated that the user can give the specification for the transitive closure of a relation *R* as

```
simple_inductive
  trcl :: "('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool"
where
  base: "trcl R x x"
| step: "trcl R x y ⟹ R y z ⟹ trcl R x z"
```

Note that there is no locale given in this specification—the parameter *R* therefore needs to be included explicitly in *trcl*, but stays fixed throughout the specification. The problem with this way of stating the specification for the transitive closure is that it derives the following induction principle.

$$\frac{\begin{array}{l} \mathit{trcl}\ R\ x\ y \\ \bigwedge R\ x.\ P\ R\ x\ x \\ \bigwedge R\ x\ y\ z.\ [\![P\ R\ x\ y;\ R\ y\ z]\!] \Longrightarrow P\ R\ x\ z \end{array}}{P\ R\ x\ y}$$

But this does not correspond to the induction principle we derived by hand, which was

$$\frac{\begin{array}{l} \mathit{trcl}\ R\ x\ y \\ \bigwedge x.\ P\ x\ x \\ \bigwedge x\ y\ z.\ [\![R\ x\ y;\ P\ y\ z]\!] \Longrightarrow P\ x\ z \end{array}}{P\ x\ y}$$

The difference is that in the one derived by hand the relation *R* is not a parameter of the proposition *P* to be proved and it is also not universally qunatified in the second and third premise. The point is that the parameter *R* stays fixed thoughout the definition and we do not want to regard it as an "ordinary" argument of the transitive closure, but one that can be freely instantiated. In order to recognise such parameters, we have to extend the specification to include a mechanism to state fixed parameters. The user should be able to write

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"

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

This leads directly to the railroad diagram shown in Figure 5.1 for the syntax of **simple_inductive**. This diagram more or less translates directly into the parser:

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

which we described in Section 3.4. If we feed into the parser the string (which corresponds to our definition of *Ind_Prelims.even* and *Ind_Prelims.odd*):

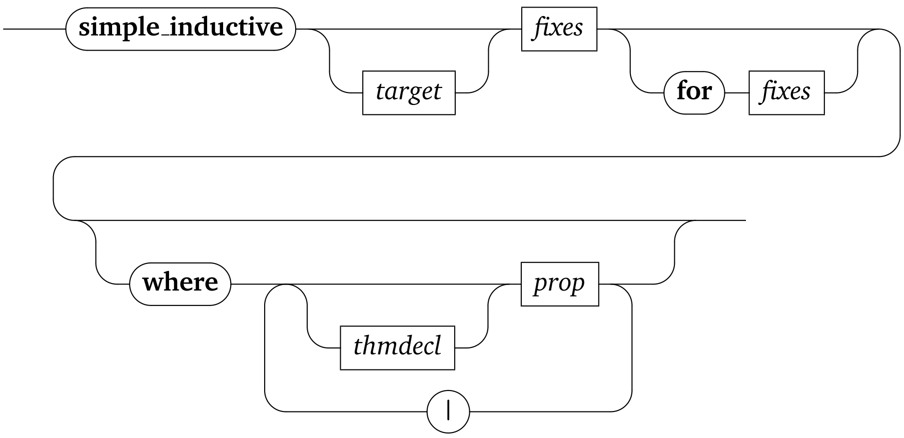

Figure 5.1: A railroad diagram describing the syntax of **simple_inductive**. The *target* indicates an optional locale; the *fixes* are an **and**-separated list of names for the inductive predicates (they can also contain typing- and syntax anotations); similarly the *fixes* after **for** to indicate fixed parameters; *prop* stands for a introduction rule with an optional theorem declaration (*thmdecl*).

```
let
  val input = filtered_input
     ("even and odd " ^
      "where " ^
      "  even0[intro]: \"even 0\" " ^
      "| evenS[intro]: \"odd n ⟹ even (Suc n)\" " ^
      "| oddS[intro]:  \"even n ⟹ odd (Suc n)\"")
in
  parse spec_parser input
end
> (([(even, NONE, NoSyn), (odd, NONE, NoSyn)],
>      [((even0,...), "\^E\^Ftoken\^Eeven 0\^E\^F\^E"),
>       ((evenS,...), "\^E\^Ftoken\^Eodd n ⟹ even (Suc n)\^E\^F\^E"),
>       ((oddS,...), "\^E\^Ftoken\^Eeven n ⟹ odd (Suc n)\^E\^F\^E")]), [])
```

then we get back a locale (in this case `NONE`), the predicates (with type and syntax annotations), the parameters (similar as the predicates) and the specifications of the introduction rules.

This is all the information we need for calling the package and setting up the keyword. The latter is done in Lines 6 and 7 in the code below.

```
1 val specification =
2   spec_parser >>
3     (fn (((loc, preds), params), specs) =>
4       Toplevel.local_theory loc (add_inductive preds params specs))
```

```
5
6 val _ = OuterSyntax.command "simple_inductive" "define inductive predicates"
7           OuterKeyword.thy_decl specification
```

We call *OuterSyntax.command* with the kind-indicator *OuterKeyword.thy_decl* since the package does not need to open up any goal state (see Section 3.5). Note that the predicates and parameters are at the moment only some “naked” variables: they have no type yet (even if we annotate them with types) and they are also no defined constants yet (which the predicates will eventually be). In Lines 1 to 4 we gather the information from the parser to be processed further. The locale is passed as argument to the function *Toplevel.local_theory*.[1] The other arguments, i.e. the predicates, parameters and intro rule specifications, are passed to the function *add_inductive* (Line 4).

We now come to the second subtask of the package, namely transforming the parser output into some internal datastructures that can be processed further. Remember that at the moment the introduction rules are just strings, and even if the predicates and parameters can contain some typing annotations, they are not yet in any way reflected in the introduction rules. So the task of *add_inductive* is to transform the strings into properly typed terms. For this it can use the function *read_spec*. This function takes some constants with possible typing annotations and some rule specifications and attempts to find a type according to the given type constraints and the type constraints by the surrounding (local theory). However this function is a bit too general for our purposes: we want that each introduction rule has only name (for example *even0* or *evenS*), if a name is given at all. The function *read_spec* however allows more than one rule. Since it is quite convenient to rely on this function (instead of building your own) we just quick ly write a wrapper function that translates between our specific format and the general format expected by *read_spec*. The code of this wrapper is as follows:

```
1 fun read_specification’ vars specs lthy =
2 let
3   val specs’ = map (fn (a, s) => (a, [s])) specs
4   val ((varst, specst), _) =
5                  Specification.read_specification vars specs’ lthy
6   val specst’ = map (apsnd the_single) specst
7 in
8   (varst, specst’)
9 end
```

It takes a list of constants, a list of rule specifications and a local theory as input. Does the transformation of the rule specifications in Line 3; calls the function and transforms the now typed rule specifications back into our format and returns the type parameter and typed rule specifications.

[1] FIXME Is this already described?

```
fun add_inductive preds params specs lthy =
let
  val (vars, specs') = read_specification' (preds @ params) specs lthy;
  val (preds', params') = chop (length preds) vars;
  val params'' = map fst params'
in
  add_inductive_i preds' params'' specs' lthy
end;
```

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 specification of the inductive predicate, including type annotations and 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 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 -> 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 -> local_theory
end;
```

(FIXME: explain Binding.binding; Attrib.binding somewhere else)

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 and the induction principle as well introduction rules.

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.[2] 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 params specs lthy =
let
  val (vars, specs') = read_specification' (preds @ params) specs lthy;
  val (preds', params') = chop (length preds) vars;
  val params'' = map fst params'
in
  add_inductive_i preds' params'' specs' 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 ->                    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_spec*.

[2] FIXME: why ist it there then?

Note that the format for rules supported by *read_spec* 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_spec*, 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_spec* 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_spec* 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.

```
opt_thm_name:
  string -> Attrib.binding parser
```

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:

```
SpecParse.opt_thm_name
```

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:

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 [3], 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.

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". This will become important later on when we deal with fixed parameters and locales.

The purpose of the package we show next is that the user just specifies the inductive predicate by stating some introduction rules and then the packages makes the equivalent definition and derives from it the needed properties.

From a high-level perspective the package consists of 6 subtasks:

5.3 The General Construction Principle

The point of these examples is to get a feeling what the automatic proofs should do in order to solve all inductive definitions we throw at them. For this it is instructive to look at the general construction principle of inductive definitions, which we shall do in the next section.

Before we start with the implementation, it is useful to describe the general form of inductive definitions that our package should accept. Suppose $R_1, \ldots, R_n$ be mutually inductive predicates and $\vec{p}$ be some fixed parameters. Then the introduction rules for $R_1, \ldots, R_n$ may have the form

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

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

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

where

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

The induction principles for the inductive predicates $R_1, \ldots, R_n$ are

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

where

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

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

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

where $\vec{K}$ abbreviates $K_1, \ldots, K_r$. Applying the introduction rules for $\forall$ and $\longrightarrow$ yields a goal 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 the 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

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

What remains is to implement these proofs generically.

```
datatype trm =
  Var "string"
| App "trm" "trm"
| Lam "string" "trm"

simple_inductive
  fresh :: "string ⇒ trm ⇒ bool" ("_ ♯ _" [100,100] 100)
where
  "a≠b ⟹ a♯Var b"
| "⟦a♯t; a♯s⟧ ⟹ a♯App t s"
| "a♯Lam a t"
| "⟦a≠b; a♯t⟧ ⟹ a♯Lam b t"
```

5.4 Code

$$\mathtt{rule ::=} \bigwedge \mathtt{xs.\ As} \Longrightarrow (\bigwedge \mathtt{ys.\ Bs} \Longrightarrow \mathtt{pred\ ss})^* \Longrightarrow \mathtt{pred\ ts}$$

$$\mathtt{orule ::=} \forall \mathtt{xs.\ As} \longrightarrow (\forall \mathtt{ys.\ Bs} \longrightarrow \mathtt{pred\ ss})^* \longrightarrow \mathtt{pred\ ts}$$

$$\mathtt{def ::= pred} \equiv \lambda \mathtt{zs.}\ \forall \mathtt{preds.\ orules} \longrightarrow \mathtt{pred\ zs}$$

$$\mathtt{ind ::=} \bigwedge \mathtt{zs.\ pred\ zs} \Longrightarrow \mathtt{rules[preds::=Ps]} \Longrightarrow \mathtt{P\ zs}$$

$$\mathtt{oind ::=} \forall \mathtt{zs.\ pred\ zs} \longrightarrow \mathtt{orules[preds::=Ps]} \longrightarrow \mathtt{P\ zs}$$

Induction proof

After "objectivication" we have $\mathtt{pred\ zs}$ and $\mathtt{orules[preds::=Ps]}$; and have to show $\mathtt{P\ zs}$. Expanding $\mathtt{pred\ zs}$ gives $\forall \mathtt{preds.\ orules} \longrightarrow \mathtt{pred\ zs}$. Instantiating the $\mathtt{preds}$ with $\mathtt{Ps}$ gives $\mathtt{orules[preds::=Ps]} \longrightarrow \mathtt{P\ zs}$. So we can conclude with $\mathtt{P\ zs}$.

Intro proof

Assume we want to prove the ith intro rule.

We have to show $\forall \mathtt{xs.\ As} \longrightarrow (\forall \mathtt{ys.\ Bs} \longrightarrow \mathtt{pred\ ss})^* \longrightarrow \mathtt{pred\ ts}$; expanding the defs, gives

$$\forall \mathtt{xs.\ As} \longrightarrow (\forall \mathtt{ys.\ Bs} \longrightarrow (\forall \mathtt{preds.\ orules} \longrightarrow \mathtt{pred\ ss}))^* \longrightarrow (\forall \mathtt{preds.\ orules} \longrightarrow \mathtt{pred\ ts}$$

By applying as many allI and impI as possible, we have

$\mathtt{As}$, $(\forall \mathtt{ys.\ Bs} \longrightarrow (\forall \mathtt{preds.\ orules} \longrightarrow \mathtt{pred\ ss}))^*$, $\mathtt{orules}$; and have to show $\mathtt{pred\ ts}$

the ith $\mathtt{orule}$ is of the form $\forall \mathtt{xs.\ As} \longrightarrow (\forall \mathtt{ys.\ Bs} \longrightarrow \mathtt{pred\ ss})^* \longrightarrow \mathtt{pred\ ts}$.

So we apply the ith $\mathtt{orule}$, but we have to show the $\mathtt{As}$ (by assumption) and all $(\forall \mathtt{ys.\ Bs} \longrightarrow \mathtt{pred\ ss})^*$. For the latter we use the assumptions $(\forall \mathtt{ys.\ Bs} \longrightarrow (\forall \mathtt{preds.\ orules} \longrightarrow \mathtt{pred\ ss}))^*$ and $\mathtt{orules}$.

First we have to produce for each predicate the definition of the form

$$\mathtt{pred} \equiv \lambda \mathtt{zs.}\ \forall \mathtt{preds.\ orules} \longrightarrow \mathtt{pred\ zs}$$

and then "register" the definitions with Isabelle. To do the latter, we use the following wrapper for `LocalTheory.define`. The wrapper takes a predicate name, a syntax annotation and a term representing the right-hand side of the definition.

```
fun make_defs ((predname, syn), trm) lthy =
let
  val arg = ((predname, syn), (Attrib.empty_binding, trm))
  val ((_, (_ , thm)), lthy') = LocalTheory.define Thm.internalK arg lthy
in
  (thm, lthy')
end
```

It returns the definition (as a theorem) and the local theory in which this definition has been made. In Line 4, `internalK` is a flag attached to the theorem (others possibilities are `definitionK` and `axiomK`). These flags just classify theorems and have no significant meaning, except for tools that, for example, find theorems in the theorem database. We also use `empty_binding` in Line 3, since the definition does not need to have any theorem attributes. A testcase for this function is

```
local_setup {* fn lthy =>
let
  val arg =  ((@{binding "MyTrue"}, NoSyn), @{term True})
  val (def, lthy') = make_defs arg lthy
in
  warning (str_of_thm lthy' def); lthy'
end *}
```

which makes the definition `MyTrue ≡ True` and then prints it out. Since we are testing the function inside **local_setup**, i.e. make changes to the ambient theory, we can query the definition using the usual command **thm**:

```
thm MyTrue_def
> MyTrue ≡ True
```

The next two functions construct the right-hand sides of the definitions, which are of the form

$$\lambda zs.\ \forall preds.\ orules \longrightarrow pred\ zs$$

The variables `zs` need to be chosen so that they do not occur in the `orules` and also be distinct from the `preds`.

The first function constructs the term for one particular predicate, say `pred`; the number of arguments of this predicate is determined by the number of argument types given in `arg_tys`. The other arguments are all `preds` and the `orules`.

```
fun defs_aux lthy orules preds (pred, arg_tys) =
let
  fun mk_all x P = HOLogic.all_const (fastype_of x) $ lambda x P

  val fresh_args =
        arg_tys
        |> map (pair "z")
        |> Variable.variant_frees lthy (preds @ orules)
        |> map Free
in
  list_comb (pred, fresh_args)
  |> fold_rev (curry HOLogic.mk_imp) orules
  |> fold_rev mk_all preds
  |> fold_rev lambda fresh_args
end
```

The function in Line 3 is just a helper function for constructing universal quantifications. The code in Lines 5 to 9 produces the fresh *zs*. For this it pairs every argument type with the string *"z"* (Line 7); then generates variants for all these strings so that they are unique w.r.t. to the predicates and *orules* (Line 8); in Line 9 it generates the corresponding variable terms for the unique strings.

The unique free variables are applied to the predicate (Line 11) using the function *list_comb*; then the *orules* are prefixed (Line 12); in Line 13 we quantify over all predicates; and in line 14 we just abstract over all the *zs*, i.e. the fresh arguments of the predicate. A testcase for this function is

```
local_setup{* fn lthy =>
let
  val orules = [@{prop "even 0"},
                @{prop "∀n::nat. odd n ⟶ even (Suc n)"},
                @{prop "∀n::nat. even n ⟶ odd (Suc n)"}]
  val preds = [@{term "even::nat⇒bool"}, @{term "odd::nat⇒bool"}]
  val pred = @{term "even::nat⇒bool"}
  val arg_tys = [@{typ "nat"}]
  val def = defs_aux lthy orules preds (pred, arg_tys)
in
  warning (Syntax.string_of_term lthy def); lthy
end *}
```

It calls *defs_aux* for the definition of *even* and prints out the definition. So we obtain as printout

```
λz. ∀even odd. (even 0) ⟶ (∀n. odd n ⟶ even (Suc n))
                        ⟶ (∀n. even n ⟶ odd (Suc n)) ⟶ even z
```

The second function for the definitions has to just iterate the function *defs_aux* over all predicates. The argument *preds* is again the the list of predicates as *terms*; the argument *prednames* is the list of names of the predicates; *arg_tyss* is the list of argument-type-lists for each predicate.

```
1 fun definitions rules preds prednames syns arg_typss lthy =
2 let
3   val thy = ProofContext.theory_of lthy
4   val orules = map (ObjectLogic.atomize_term thy) rules
5   val defs = map (defs_aux lthy orules preds) (preds ~~ arg_typss)
6 in
7   fold_map make_defs (prednames ~~ syns ~~ defs) lthy
8 end
```

The user will state the introduction rules using meta-implications and meta-quantifications. In Line 4, we transform these introduction rules into the object logic (since definitions cannot be stated with meta-connectives). To do this transformation we have to obtain the theory behind the local theory (Line 3); with this theory we can use the function *ObjectLogic.atomize_term* to make the transformation (Line 4).

The call to *defs_aux* in Line 5 produces all left-hand sides of the definitions. The actual definitions are then made in Line 7. The result of the function is a list of theorems and a local theory. A testcase for this function is

```
local_setup {* fn lthy =>
let
  val rules = [@{prop "even 0"},
               @{prop "⋀n::nat. odd n ⟹ even (Suc n)"},
               @{prop "⋀n::nat. even n ⟹ odd (Suc n)"}]
  val preds = [@{term "even::nat⇒bool"}, @{term "odd::nat⇒bool"}]
  val prednames = [@{binding "even"}, @{binding "odd"}]
  val syns = [NoSyn, NoSyn]
  val arg_tyss = [[@{typ "nat"}], [@{typ "nat"}]]
  val (defs, lthy') = definitions rules preds prednames syns arg_tyss lthy
in
  warning (str_of_thms lthy' defs); lthy'
end *}
```

where we feed into the functions all parameters corresponding to the *even-odd* example. The definitions we obtain are:

```
thm even_def odd_def
> even ≡ λz. ∀even odd. (even 0) ⟶ (∀n. odd n ⟶ even (Suc n))
>                                  ⟶ (∀n. even n ⟶ odd (Suc n)) ⟶ even z,
> odd ≡ λz. ∀even odd. (even 0) ⟶ (∀n. odd n ⟶ even (Suc n))
>                                 ⟶ (∀n. even n ⟶ odd (Suc n)) ⟶ odd z
```

This completes the code for making the definitions. Next we deal with the induction principles. Recall that the proof of the induction principle for *even* was:

```
lemma man_ind_principle:
assumes prems: "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 prems)
apply(unfold even_def)
apply(drule spec[where x=P])
apply(drule spec[where x=Q])
apply(assumption)
done
```

The code for automating such induction principles has to accomplish two tasks: constructing the induction principles from the given introduction rules and then automatically generating proofs for them using a tactic.

The tactic will use the following helper function for instantiating universal quantifiers.

```
fun inst_spec ctrm =
 Drule.instantiate' [SOME (ctyp_of_term ctrm)] [NONE, SOME ctrm] @{thm spec}
```

This helper function instantiates the *?x* in the theorem ∀ *x. ?P x* ⟹ *?P ?x* with a given *cterm*. Together with the tactic

```
fun inst_spec_tac ctrms =
  EVERY' (map (dtac o inst_spec) ctrms)
```

we can use *inst_spec_tac* in the following proof to instantiate the three quantifiers in the assumption.

lemma
  fixes *P::"nat ⇒ nat ⇒ nat ⇒ bool"*
  shows *"∀x y z. P x y z ⟹ True"*
apply *(tactic {*
  inst_spec_tac [@{cterm "a::nat"},@{cterm "b::nat"},@{cterm "c::nat"}] 1 *})*

We obtain the goal state

```
goal (1 subgoal):
 1. P a b c ⟹ True
```

Now the complete tactic for proving the induction principles can be implemented as follows:

```
fun induction_tac defs prems insts =
  EVERY1 [ObjectLogic.full_atomize_tac,
          cut_facts_tac prems,
          K (rewrite_goals_tac defs),
          inst_spec_tac insts,
          assume_tac]
```

We only have to give it the definitions, the premise (like *even n*) and the instantiations as arguments. Compare this with the manual proof given for the lemma *man_ind_principle*: there is almos a one-to-one correspondence between the **apply**-script and the *induction_tac*. A testcase for this tactic is the function

```
fun test_tac prems =
let
  val defs = [@{thm even_def}, @{thm odd_def}]
  val insts = [@{cterm "P::nat⇒bool"}, @{cterm "Q::nat⇒bool"}]
in
  induction_tac defs prems insts
end
```

which indeed proves the induction principle:

lemma *auto_ind_principle:*
assumes *prems: "even n"*
shows *"P 0 ⟹ (⋀m. Q m ⟹ P (Suc m)) ⟹ (⋀m. P m ⟹ Q (Suc m)) ⟹ P n"*
apply*(tactic {* test_tac @{thms prems} *})*
done

While the tactic for the induction principle is relatively simple, it is a bit harder to construct the goals from the introduction rules the user provides. In general we have to construct for each predicate *pred* a goal of the form *pred ?zs* ⟹ *rules[preds := ?Ps]* ⟹ *?P$?zs*

where the predicates *preds* are replaced in the introduction rules by new distinct variables written *Ps*. We also need to generate fresh arguments for the predicate *pred* in the premise and the *?P* in the conclusion. Note that the *?Ps* and *?zs* need to be schematic variables that can be instantiated. This corresponds to what the *auto_ind_principle* looks like:

thm *auto_ind_principle*
> ⟦*even ?n; ?P 0;* ⋀*m. ?Q m* ⟹ *?P (Suc m);* ⋀*m. ?P m* ⟹ *?Q (Suc m)*⟧ ⟹ *?P ?n*

We achieve that in two steps.

The function below expects that the introduction rules are already appropriately substituted. The argument *srules* stands for these substituted rules; *cnewpreds* are the certified terms coresponding to the variables *Ps*; *pred* is the predicate for which we prove the introduction principle; *newpred* is its replacement and *tys* are the argument types of this predicate.

```
1  fun prove_induction lthy defs srules cnewpreds ((pred, newpred), arg_tys)  =
2  let
3    val zs = replicate (length arg_tys) "z"
4    val (newargnames, lthy') = Variable.variant_fixes zs lthy;
5    val newargs = map Free (newargnames ~~ arg_tys)
6
7    val prem = HOLogic.mk_Trueprop (list_comb (pred, newargs))
8    val goal = Logic.list_implies
9           (srules, HOLogic.mk_Trueprop (list_comb (newpred, newargs)))
10 in
11   Goal.prove lthy' [] [prem] goal
12   (fn {prems, ...} => induction_tac defs prems cnewpreds)
13   |> singleton (ProofContext.export lthy' lthy)
14 end
```

In Line 3 we produce names *zs* for each type in the argument type list. Line 4 makes these names unique and declares them as *free* (but fixed) variables in the local theory *lthy'*. In Line 5 we just construct the terms corresponding to these variables. The term variables are applied to the predicate in Line 7 (this corresponds to the first premise *pred zs* of the induction principle). In Line 8 and 9, we first construct the term *P zs* and then add the (substituded) introduction rules as premises. In case that no introduction rules are given, the conclusion of this implication needs to be wrapped inside a *Trueprop*, otherwise the Isabelle's goal mechanism will fail.

In Line 11 we set up the goal to be proved; in the next line we call the tactic for proving the induction principle. This tactic expects the definitions, the premise and the (certified) predicates with which the introduction rules have been substituted. The code in these two lines will return a theorem. However, it is a theorem proved inside the local theory *lthy'*, where the variables *zs* are fixed, but free. By exporting this theorem from *lthy'* (which contains the *zs* as free) to *lthy* (which does not), we obtain the desired schematic variables.

```
local_setup{* fn lthy =>
let
  val defs = [@{thm even_def}, @{thm odd_def}]
  val srules = [@{prop "P (0::nat)"},
                @{prop "⋀n::nat. Q n ⟹ P (Suc n)"},
                @{prop "⋀n::nat. P n ⟹ Q (Suc n)"}]
  val cnewpreds = [@{cterm "P::nat⇒bool"}, @{cterm "Q::nat⇒bool"}]
  val pred = @{term "even::nat⇒bool"}
  val newpred = @{term "P::nat⇒bool"}
  val arg_tys = [@{typ "nat"}]
  val intro =
    prove_induction lthy defs srules cnewpreds ((pred, newpred), arg_tys)
in
  warning (str_of_thm_raw lthy intro); lthy
end *}
```

This prints out:

```
⟦even ?z; P 0; ⋀n. Q n ⟹ P (Suc n); ⋀n. P n ⟹ Q (Suc n)⟧ ⟹ P ?z
```

Note that the export from `lthy'` to `lthy` in Line 13 above has turned the free, but fixed, `z` into a schematic variable `?z`.

We still have to produce the new predicates with which the introduction rules are substituted and iterate `prove_induction` over all predicates. This is what the next function does.

```
1  fun inductions rules defs preds arg_tyss lthy  =
2  let
3    val Ps = replicate (length preds) "P"
4    val (newprednames, lthy') = Variable.variant_fixes Ps lthy
5
6    val thy = ProofContext.theory_of lthy'
7
8    val tyss' = map (fn tys => tys ---> HOLogic.boolT) arg_tyss
9    val newpreds = map Free (newprednames ~~ tyss')
10   val cnewpreds = map (cterm_of thy) newpreds
11   val srules = map (subst_free (preds ~~ newpreds)) rules
12
13 in
14   map (prove_induction lthy' defs srules cnewpreds)
15         (preds ~~ newpreds ~~ arg_tyss)
16           |> ProofContext.export lthy' lthy
17 end
```

In Line 3 we generate a string `"P"` for each predicate. In Line 4, we use the same trick as in the previous function, that is making the `Ps` fresh and declaring them as fixed, but free, in the new local theory `lthy'`. From the local theory we extract the ambient theory in Line 6. We need this theory in order to certify the new predicates. In Line 8 we calculate the types of these new predicates using the given argument types. Next we turn them into terms and subsequently certify them (Line 9 and 10).

We can now produce the substituted introduction rules (Line 11) using the function *subst_free*. Line 14 and 15 just iterate the proofs for all predicates. From this we obtain a list of theorems. Finally we need to export the fixed variables *Ps* to obtain the schematic variables (Line 16).

A testcase for this function is

```
local_setup {* fn lthy =>
let
  val rules = [@{prop "even (0::nat)"},
               @{prop "⋀n::nat. odd n ⟹ even (Suc n)"},
               @{prop "⋀n::nat. even n ⟹ odd (Suc n)"}]
  val defs = [@{thm even_def}, @{thm odd_def}]
  val preds = [@{term "even::nat⇒bool"}, @{term "odd::nat⇒bool"}]
  val tyss = [[@{typ "nat"}], [@{typ "nat"}]]
  val ind_thms = inductions rules defs preds tyss lthy
in
  warning (str_of_thms_raw lthy ind_thms); lthy
end *}
```

which prints out

```
> even ?z ⟹ ?P1 0 ⟹
> (⋀m. ?Pa1 m ⟹ ?P1 (Suc m)) ⟹ (⋀m. ?P1 m ⟹ ?Pa1 (Suc m)) ⟹ ?P1 ?z,
> odd ?z ⟹ ?P1 0
> ⟹ (⋀m. ?Pa1 m ⟹ ?P1 (Suc m)) ⟹ (⋀m. ?P1 m ⟹ ?Pa1 (Suc m)) ⟹ ?Pa1
?z
```

Note that now both, the *Ps* and the *zs*, are schematic variables. The numbers have been introduced by the pretty-printer and are not significant.

This completes the code for the induction principles. Finally we can prove the introduction rules. Their proofs are quite a bit more involved. To ease them somewhat we use the following two helper function.

```
val all_elims = fold (fn ct => fn th => th RS inst_spec ct)
val imp_elims = fold (fn th => fn th' => [th', th] MRS @{thm mp})
```

To see what they do, let us suppose whe have the follwoing three theorems.

```
lemma all_elims_test:
  fixes P::"nat ⇒ nat ⇒ nat ⇒ bool"
  shows "∀x y z. P x y z" sorry

lemma imp_elims_test:
  fixes A B C::"bool"
  shows "A ⟶ B ⟶ C" sorry

lemma imp_elims_test':
  fixes A::"bool"
  shows "A" "B" sorry
```

The function *all_elims* takes a list of (certified) terms and instantiates theorems of the form *all_elims_test*. For example we can instantiate the quantifiers in this theorem with *a*, *b* and *c* as follows

```
let
  val ctrms = [@{cterm "a::nat"}, @{cterm "b::nat"}, @{cterm "c::nat"}]
  val new_thm = all_elims ctrms @{thm all_elims_test}
in
  warning (str_of_thm @{context} new_thm)
end
> P a b c
```

Similarly, the function *imp_elims* eliminates preconditions from implications. For example

```
warning (str_of_thm @{context}
            (imp_elims @{thms imp_elims_test'} @{thm imp_elims_test}))
> C
```

```
  prems_of
  Logic.strip_params
  Logic.strip_assums_hyp


fun chop_print_tac m n ctxt thm =
let
  val [trm] = prems_of thm
  val params = map fst (Logic.strip_params trm)
  val prems  = Logic.strip_assums_hyp trm
  val (prems1, prems2) = chop (length prems - m) prems;
  val (params1, params2) = chop (length params - n) params;
  val _ = warning (Syntax.string_of_term ctxt trm)
  val _ = warning (commas params)
  val _ = warning (commas (map (Syntax.string_of_term ctxt) prems))
  val _ = warning ((commas params1) ^ " | " ^ (commas params2))
  val _ = warning ((commas (map (Syntax.string_of_term ctxt) prems1)) ^ " |
" ^
                  (commas (map (Syntax.string_of_term ctxt) prems2)))
in
   Seq.single thm
end


  METAHYPS


fun chop_print_tac2 ctxt prems =
let
  val _ = warning (commas (map (str_of_thm ctxt) prems))
```

```
in
   all_tac
end
```

lemma *intro1:*
  shows "$\bigwedge$m. odd m $\Longrightarrow$ even (Suc m)"
apply(tactic {* ObjectLogic.rulify_tac 1 *})
apply(tactic {* rewrite_goals_tac [@{thm even_def}, @{thm odd_def}] *})
apply(tactic {* REPEAT (resolve_tac [@{thm allI}, @{thm impI}] 1) *})
apply(tactic {* chop_print_tac 3 2 @{context} *})
oops

```
fun SUBPROOF_test tac ctxt =
  SUBPROOF (fn {params, prems, context,...} =>
    let
      val thy = ProofContext.theory_of context
    in
      tac (params, prems, context)
      THEN  Method.insert_tac prems 1
      THEN  print_tac "SUBPROOF Test"
      THEN  SkipProof.cheat_tac thy
    end) ctxt 1
```

lemma *fresh_App:*
  shows "$\bigwedge$a t s. ⟦a♯t; a♯s⟧ $\Longrightarrow$ a♯App t s"
apply(tactic {* ObjectLogic.rulify_tac 1 *})
apply(tactic {* rewrite_goals_tac [@{thm fresh_def}] *})
apply(tactic {* REPEAT (resolve_tac [@{thm allI}, @{thm impI}] 1) *})
oops

```
fun subproof2 prem params2 prems2 =
 SUBPROOF (fn {prems, ...} =>
   let
     val prem' = prems MRS prem;
     val prem'' =
       case prop_of prem' of
           _ $ (Const (@{const_name All}, _) $ _) =>
             prem' |> all_elims params2
                   |> imp_elims prems2
```

```
        | _ => prem';
  in
    rtac prem'' 1
  end)
```

```
fun subproof1 rules preds i =
 SUBPROOF (fn {params, prems, context = ctxt', ...} =>
  let
    val (prems1, prems2) = chop (length prems - length rules) prems;
    val (params1, params2) = chop (length params - length preds) params;
  in
    rtac (ObjectLogic.rulify (all_elims params1 (nth prems2 i))) 1
    (* applicateion of the i-ith intro rule *)
    THEN
    EVERY1 (map (fn prem => subproof2 prem params2 prems2 ctxt') prems1)
  end)
```

params1 are the variables of the rules; *params2* is the variables corresponding to the *preds*.

prems1 are the assumption corresponding to the rules; *prems2* are the assumptions coming from the allIs/impIs

you instantiate the parameters i-th introduction rule with the parameters that come from the rule; and you apply it to the goal

this now generates subgoals corresponding to the premisses of this intro rule

```
fun intros_tac defs rules preds i ctxt =
  EVERY1 [ObjectLogic.rulify_tac,
          K (rewrite_goals_tac defs),
          REPEAT o (resolve_tac [@{thm allI}, @{thm impI}]),
          subproof1 rules preds i ctxt]
```

A test case

```
fun intros_tac_test ctxt i =
let
  val rules = [@{prop "even (0::nat)"},
               @{prop "⋀n::nat. odd n ⟹ even (Suc n)"},
               @{prop "⋀n::nat. even n ⟹ odd (Suc n)"}]
  val defs = [@{thm even_def}, @{thm odd_def}]
  val preds = [@{term "even::nat⇒bool"}, @{term "odd::nat⇒bool"}]
in
  intros_tac defs rules preds i ctxt
end
```

lemma *intro0:*

shows *"even 0"*
apply(*tactic {* intros_tac_test @{context} 0 *}*)
done

lemma *intro1:*
shows *"$\bigwedge m$. odd m $\Longrightarrow$ even (Suc m)"*
apply(*tactic {* intros_tac_test @{context} 1 *}*)
done

lemma *intro2:*
shows *"$\bigwedge m$. even m $\Longrightarrow$ odd (Suc m)"*
apply(*tactic {* intros_tac_test @{context} 2 *}*)
done

```
fun introductions rules preds defs lthy =
let
  fun prove_intro (i, goal) =
    Goal.prove lthy [] [] goal
      (fn {context, ...} => intros_tac defs rules preds i context)
in
  map_index prove_intro rules
end
```

main internal function

```
fun add_inductive pred_specs rule_specs lthy =
let
  val syns = map snd pred_specs
  val pred_specs' = map fst pred_specs
  val prednames = map fst pred_specs'
  val preds = map (fn (p, ty) => Free (Binding.name_of p, ty)) pred_specs'

  val tyss = map (binder_types o fastype_of) preds
  val (attrs, rules) = split_list rule_specs

  val (defs, lthy') = definitions rules preds prednames syns tyss lthy
  val ind_rules = inductions rules defs preds tyss lthy'
  val intro_rules = introductions rules preds defs lthy'

  val mut_name = space_implode "_" (map Binding.name_of prednames)
  val case_names = map (Binding.name_of o fst) attrs
in
    lthy'
    |> LocalTheory.notes Thm.theoremK (map (fn (((a, atts), _), th) =>
        ((Binding.qualify false mut_name a, atts), [([th], [])]))
(rule_specs ~~ intro_rules))
    |-> (fn intross => LocalTheory.note Thm.theoremK
         ((Binding.qualify false mut_name (@{binding "intros"}), []), maps
snd intross))
    |>> snd
```

```
    ||>> (LocalTheory.notes Thm.theoremK (map (fn (((R, _), _), th) =>
          ((Binding.qualify false (Binding.name_of R) (@{binding "induct"}),
           [Attrib.internal (K (RuleCases.case_names case_names)),
            Attrib.internal (K (RuleCases.consumes 1)),
            Attrib.internal (K (Induct.induct_pred ""))]), [([th], [])]))
           (pred_specs ~~ ind_rules)) #>> maps snd)
    |> snd
end
```

```
fun add_inductive_cmd pred_specs rule_specs lthy =
let
  val ((pred_specs', rule_specs'), _) =
         Specification.read_spec pred_specs rule_specs lthy
in
  add_inductive pred_specs' rule_specs' lthy
end

val spec_parser =
   OuterParse.fixes --
   Scan.optional
     (OuterParse.$$$ "where" |--
        OuterParse.!!!
          (OuterParse.enum1 "|"
             (SpecParse.opt_thm_name ":" -- OuterParse.prop))) []

val specification =
  spec_parser >>
    (fn ((pred_specs), rule_specs) => add_inductive_cmd pred_specs
rule_specs)

val _ = OuterSyntax.local_theory "simple_inductive"
              "define inductive predicates"
                 OuterKeyword.thy_decl specification
```

Things to include at the end:

- say something about add-inductive-i to return the rules
- say that the induction principle is weaker (weaker than what the standard inductive package generates)
- say that no conformity test is done

simple_inductive
  Even **and** *Odd*
where
  Even0: "Even 0"
| *EvenS: "Odd n ⟹ Even (Suc n)"*
| *OddS: "Even n ⟹ Odd (Suc n)"*

Appendix A

Recipes

Possible topics:

- translations/print translations; *ProofContext.print_syntax*
- user space type systems (in the form that already exists)
- unification and typing algorithms (*Pure/pattern.ML* implements HOPU)
- useful datastructures: discrimination nets, association lists

A.1 Useful Document Antiquotations

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

Solution: This can be achieved with 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 on the Isabelle level

print_antiquotations

you obtain a list of all currently available document antiquotations and their options.

Below we will give the code for two additional document antiquotations both of which are intended to typeset ML-code. The crucial point of these document antiquotations is that they not just print the ML-code, but also check whether it compiles. This will provide a sanity check for the code and also allows you 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 document antiquotation is as follows:

```
1 fun ml_val code_txt = "val _ = " ^ code_txt
2
3 fun output_ml {context = ctxt, ...} code_txt =
4   (ML_Context.eval_in (SOME ctxt) false Position.none (ml_val code_txt);
5    ThyOutput.output (map Pretty.str (space_explode "\n" code_txt)))
6
7 val _ = ThyOutput.antiquotation "ML_checked" (Scan.lift Args.name) output_ml
```

The parser `(Scan.lift Args.name)` in Line 7 parses a string, in this case the code, and then calls the function `output_ml`. As mentioned before, the parsed code is sent to the ML-compiler in Line 4 using the function `ml_val`, which constructs the appropriate ML-expression, and using `eval_in`, which calls the compiler. If the code is "approved" by the compiler, then the output function `output` in the next line pretty prints the code. This function expects that the code is a list of (pretty)strings where each string correspond to a line in the output. Therefore the use of `(space_explode "\n" txt)` which produces such a list according to linebreaks. There are a number of options for antiquotations that are observed by the function `output` when printing the code (including `[display]` and `[quotes]`). The function `antiquotation` in Line 7 sets up the new document antiquotation.

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

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 {context = ctxt, ...} (code_txt, pos) =
2   (ML_Context.eval_in (SOME ctxt) false pos (ml_val code_txt);
3    ThyOutput.output (map Pretty.str (space_explode "\n" code_txt)))
4
5 val _ = ThyOutput.antiquotation "ML_checked"
6          (Scan.lift (OuterParse.position Args.name)) output_ml
```

where in Lines 1 and 2 the positional information is properly treated. The parser `OuterParse.position` encodes the positional information in the result.

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

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

```
@{ML_resp "a_piece_of_code" "a_pattern"}
```

To add some convenience and also to deal with large outputs, the user can give a partial specification by using ellipses. For example `(..., ...)` for specifying a pair.

In order to check consistency between the pattern and the output of the code, we have to change the ML-expression that is sent to the compiler: in *ML_checked* we sent the expression *"val _ = a_piece_of_code"* to the compiler; now the wildcard _ must be be replaced by the given pattern. However, we have to remove all ellipses from it and replace them by *"_"*. 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 add a response indicator to the result using:

```
fun add_resp pat = map (fn s => "> " ^ s) pat
```

The rest of the code of *ML_resp* is:

```
1  fun output_ml_resp {context = ctxt, ...} ((code_txt, pat), pos) =
2    (ML_Context.eval_in (SOME ctxt) false pos (ml_pat (code_txt, pat));
3     let
4       val code_output = space_explode "\n" code_txt
5       val resp_output = add_resp (space_explode "\n" pat)
6     in
7       ThyOutput.output (map Pretty.str (code_output @ resp_output))
8     end)
9
10 val _ = ThyOutput.antiquotation "ML_resp"
11          (Scan.lift (OuterParse.position (Args.name -- Args.name)))
12             output_ml_resp
```

In comparison with *ML_checked*, we only changed the line about the compiler (Line 2), the lines about the output (Lines 4 to 7) and the parser in the setup (Line 11). Now you can 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 document antiquotation, however, is that the pattern can only be given for values that can be constructed. This excludes values that are abstract datatypes, like *thm*s and *cterm*s.

A.2 Restricting the Runtime of a Function

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

Solution: In PolyML 5.2.1 and later, this can be achieved using the function *timeLimit*.

Assume you defined the Ackermann function on the ML-level.

```
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 TimeLimit.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 5.2.1 or later, because this version of PolyML has the infrastructure for multithreaded 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.3 Measuring Time

Problem: You want to measure the running time of a tactic or function.

Solution: Time can be measured using the function *start_timing* and *end_timing*.

Suppose you defined the Ackermann function on the Isabelle level.

```
fun
 ackermann:: "(nat × nat) ⇒ nat"
where
    "ackermann (0, n) = n + 1"
  | "ackermann (m, 0) = ackermann (m - 1, 1)"
  | "ackermann (m, n) = ackermann (m - 1, ackermann (m, n - 1))"
```

You can measure how long the simplifier takes to verify a datapoint of this function. The actual timing is done inside the wrapper function:

```
1 fun timing_wrapper tac st =
2 let
3   val t_start = start_timing ();
4   val res = tac st;
5   val t_end = end_timing t_start;
6 in
7   (warning (#message t_end); res)
8 end
```

Note that this function, in addition to a tactic, also takes a state `st` as argument and applies this state to the tactic (Line 4). The reason is that tactics are lazy functions and you need to force them to run, otherwise the timing will be meaningless. The simplifier tactic, amongst others, can be forced to run by just applying the state to it. But “fully” lazy tactics, such as `resolve_tac`, need even more “standing-on-ones-head” to force them to run.

The time between start and finish of the simplifier will be calculated as the end time minus the start time. An example of the wrapper is the proof

```
lemma "ackermann (3, 4) = 125"
apply(tactic {*
  timing_wrapper (simp_tac (@{simpset} addsimps @{thms "nat_number"}) 1) *})
done
```

where it returns something on the scale of 3 seconds. We chose to return this information as a string, but the timing information is also accessible in number format.

> ***Read More***
> *Basic functions regarding timing are defined in* `Pure/ML-Systems/polyml_common.ML` *(for the PolyML compiler). Some more advanced functions are defined in* `Pure/General/output.ML`.

A.4 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, say `bval` containing a boolean, `ival` containing an integer and `sval` containing a string. These values can be declared on the ML-level by

```
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 with

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

or on the ML-level with

```
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 achieved using the command **setup**.

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

Now the retrieval of this value yields:

```
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
```

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

There are many good reasons to control parameters in this way. One is that no global reference is needed, which would cause many headaches with the multithreaded execution of Isabelle.

A.5 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.6 Executing an External Application

Problem: You want to use an external application.

Solution: The function *system_out* might be the right thing for you.

This function executes an external command as if printed in a shell. It returns the output of the program and its return value.

For example, consider running an ordinary shell commands:

```
system_out "echo Hello world!"
> ("Hello world!\n", 0)
```

Note that it works also fine with timeouts (see Recipe A.2 on Page 112), i.e. external applications are killed properly. For example, the following expression takes only approximately one second:

```
TimeLimit.timeLimit (Time.fromSeconds 1) system_out "sleep 30"
     handle TimeLimit.TimeOut => ("timeout", ~1)
> ("timeout", ~1)
```

The function *system_out* can also be used for more reasonable applications, e.g. coupling external solvers with Isabelle. In that case, one has to make sure that Isabelle can find the particular executable. One way to ensure this is by adding a Bash-like variable binding into one of Isabelle's settings file (prefer the user settings file usually to be found at *$HOME/.isabelle/etc/settings*).

For example, assume you want to use the application *foo* which is here supposed to be located at */usr/local/bin/*. The following line has to be added to one of Isabelle's settings file:

FOO=/usr/local/bin/foo

In Isabelle, this application may now be executed by

```
system_out "$FOO"
> ...
```

A.7 Writing an Oracle

Problem: You want to use a fast, new decision procedure not based one Isabelle's tactics, and you do not care whether it is sound.

Solution: Isabelle provides the oracle mechanisms to bypass the inference kernel. Note that theorems proven by an oracle carry a special mark to inform the user of their potential incorrectness.

> ***Read More***
> *A short introduction to oracles can be found in [isar-ref: no suitable label for section 3.11]. A simple example, which we will slightly extend here, is given in `FOL/ex/Iff_Oracle.thy`. The raw interface for adding oracles is `add_oracle` in `Pure/thm.ML`.*

For our explanation here, we restrict ourselves to decide propositional formulae which consist only of equivalences between propositional variables, i.e. we want to decide whether `(P = (Q = P)) = Q` is a tautology.

Assume, that we have a decision procedure for such formulae, implemented in ML. Since we do not care how it works, we will use it here as an "external solver":

use *"external_solver.ML"*

We do, however, know that the solver provides a function `IffSolver.decide`. It takes a string representation of a formula and returns either `true` if the formula is a tautology or `false` otherwise. The input syntax is specified as follows:

formula ::= atom | (formula <=> formula)

and all token are separated by at least one space.

(FIXME: is there a better way for describing the syntax?)

We will proceed in the following way. We start by translating a HOL formula into the string representation expected by the solver. The solver's result is then used to build an oracle, which we will subsequently use as a core for an Isar method to be able to apply the oracle in proving theorems.

Let us start with the translation function from Isabelle propositions into the solver's string representation. To increase efficiency while building the string, we use functions from the `Buffer` module.

```
fun translate t =
 let
   fun trans t =
     (case t of
       @{term "op = :: bool ⇒ bool ⇒ bool"} $ t $ u =>
         Buffer.add " (" #>
         trans t #>
         Buffer.add "<=>" #>
         trans u #>
         Buffer.add ") "
     | Free (n, @{typ bool}) =>
        Buffer.add " " #>
        Buffer.add n #>
        Buffer.add " "
     | _ => error "inacceptable term")
 in Buffer.content (trans t Buffer.empty) end
```

Here is the string representation of the term `p = (q = p)`:

```
translate @{term "p = (q = p)"}
> " ( p <=> ( q <=> p ) ) "
```

Let us check, what the solver returns when given a tautology:

```
IffSolver.decide (translate @{term "p = (q = p) = q"})
> true
```

And here is what it returns for a formula which is not valid:

```
IffSolver.decide (translate @{term "p = (q = p)"})
> false
```

Now, we combine these functions into an oracle. In general, an oracle may be given any input, but it has to return a certified proposition (a special term which is type-checked), out of which Isabelle's inference kernel "magically" makes a theorem.

Here, we take the proposition to be show as input. Note that we have to first extract the term which is then passed to the translation and decision procedure. If the solver finds this term to be valid, we return the given proposition unchanged to be turned then into a theorem:

```
oracle iff_oracle = {* fn ct =>
  if IffSolver.decide (translate (HOLogic.dest_Trueprop (Thm.term_of ct)))
  then ct
  else error "Proof failed."*}
```

Here is what we get when applying the oracle:

```
iff_oracle @{cprop "p = (p::bool)"}
> p = p
```

(FIXME: is there a better way to present the theorem?)

To be able to use our oracle for Isar proofs, we wrap it into a tactic:

```
val iff_oracle_tac =
  CSUBGOAL (fn (goal, i) =>
    (case try iff_oracle goal of
      NONE => no_tac
    | SOME thm => rtac thm i))
```

and create a new method solely based on this tactic:

```
method_setup iff_oracle = {*
   Scan.succeed (K (Method.SIMPLE_METHOD' iff_oracle_tac))
*} "Oracle-based decision procedure for chains of equivalences"
```

Finally, we can test our oracle to prove some theorems:

```
lemma "p = (p::bool)"
   by iff_oracle

lemma "p = (q = p) = q"
   by iff_oracle
```

(FIXME: say something about what the proof of the oracle is ... what do you mean?)

A.8 SAT Solvers

Problem: You like to use a SAT solver to find out whether an Isabelle formula is satisfiable or not.

Solution: Isabelle contains a general interface for a number of external SAT solvers (including ZChaff and Minisat) and also contains a simple internal SAT solver that is based on the DPLL algorithm.

The SAT solvers expect a propositional formula as input and produce a result indicating that the formula is either satisfiable, unsatisfiable or unknown. The type of the propositional formula is *PropLogic.prop_formula* with the usual constructors such as *And*, *Or* and so on.

The function *PropLogic.prop_formula_of_term* translates an Isabelle term into a propositional formula. Let us illustrate this function by translating $A \wedge \neg A \vee B$. The function will return a propositional formula and a table. Suppose

```
val (pform, table) =
      PropLogic.prop_formula_of_term @{term "A ∧ ¬A ∨ B"}  Termtab.empty
```

then the resulting propositional formula *pform* is

```
Or (And (BoolVar 1, Not (BoolVar 1)), BoolVar 2)
```

where indices are assigned for the variables *A* and *B*, respectively. This assignment is recorded in the table that is given to the translation function and also returned (appropriately updated) in the result. In the case above the input table is empty (i.e. *Termtab.empty*) and the output table is

```
Termtab.dest table
> [(Free ("A", "bool"), 1), (Free ("B", "bool"), 2)]
```

An index is also produced whenever the translation function cannot find an appropriate propositional formula for a term. Attempting to translate $\forall x.\ P\ x$

```
val (pform', table') =
      PropLogic.prop_formula_of_term @{term "∀x::nat. P x"}  Termtab.empty
```

returns *BoolVar 1* for *pform'* and the table *table'* is:

```
map (apfst (Syntax.string_of_term @{context})) (Termtab.dest table')
> (∀x. P x, 1)
```

In the print out of the tabel, we used some pretty printing scaffolding to see better which assignment the table contains.

Having produced a propositional formula, you can now call the SAT solvers with the function *SatSolver.invoke_solver*. For example

```
SatSolver.invoke_solver "dpll" pform
> SatSolver.SATISFIABLE assg
```

determines that the formula *pform* is satisfiable. If we inspect the returned function *assg*

```
let
  val SatSolver.SATISFIABLE assg = SatSolver.invoke_solver "dpll" pform
in
  (assg 1, assg 2, assg 3)
end
> (SOME true, SOME true, NONE)
```

we obtain a possible assignment for the variables *A* and *B* that makes the formula satisfiable.

Note that we invoked the SAT solver with the string *"dpll"*. This string specifies which SAT solver is invoked (in this case the internal one). If instead you invoke the SAT solver with the string *"auto"*

```
SatSolver.invoke_solver "auto" pform
```

several external SAT solvers will be tried (assuming they are installed). If no external SAT solver is installed, then the default is *"dpll"*.

There are also two tactics that make use of SAT solvers. One is the tactic *sat_tac*. For example

lemma *"True"*
apply(*tactic {* sat.sat_tac 1 *}*)
done

However, for proving anything more exciting using *sat_tac* you have to use a SAT solver that can produce a proof. The internal one is not usuable for this.

> ***Read More***
> *The interface for the external SAT solvers is implemented in HOL/Tools/sat_solver.ML. This file contains also a simple SAT solver based on the DPLL algorithm. The tactics for SAT solvers are implemented in HOL/Tools/sat_funcs.ML. Functions concerning propositional formulas are implemented in HOL/Tools/prop_logic.ML. The tables used in the translation function are implemented in Pure/General/table.ML.*

A.9 User Space Type-Systems

Appendix B

Solutions to Most Exercises

Solution for Exercise 2.6.1.

```
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 2.6.2.

```
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 parser = Scan.repeat (scan_cmt || any)

val scan_all =
      Scan.finite Symbol.stopper parser >> 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**)")
```

Solution for Exercise 4.5.1.

```
fun dest_sum term =
  case term of
    (@{term "(op +):: nat ⇒ nat ⇒ nat"} $ t1 $ t2) =>
        (snd (HOLogic.dest_number t1), snd (HOLogic.dest_number t2))
  | _ => raise TERM ("dest_sum", [term])

fun get_sum_thm ctxt t (n1, n2) =
let
  val sum = HOLogic.mk_number @{typ "nat"} (n1 + n2)
  val goal = Logic.mk_equals (t, sum)
in
  Goal.prove ctxt [] [] goal (K (arith_tac ctxt 1))
end

fun add_sp_aux ss t =
let
  val ctxt = Simplifier.the_context ss
  val t' = term_of t
in
  SOME (get_sum_thm ctxt t' (dest_sum t'))
  handle TERM _ => NONE
end
```

The setup for the simproc is

```
simproc_setup add_sp ("t1 + t2") = {* K add_sp_aux *}
```

and a test case is the lemma

```
lemma "P (Suc (99 + 1)) ((0 + 0)::nat) (Suc (3 + 3 + 3)) (4 + 1)"
  apply(tactic {* simp_tac (HOL_basic_ss addsimprocs [@{simproc add_sp}]) 1 *})
```

where the simproc produces the goal state

```
goal (1 subgoal):
 1. P (Suc 100) 0 (Suc 9) ((4::'a) + (1::'a))
```

Solution for Exercise 4.6.1.

The following code assumes the function `dest_sum` from the previous exercise.

```
fun add_simple_conv ctxt ctrm =
let
  val trm =  Thm.term_of ctrm
in
  get_sum_thm ctxt trm (dest_sum trm)
end

fun add_conv ctxt ctrm =
  (case Thm.term_of ctrm of
     @{term "(op +)::nat ⇒ nat ⇒ nat"} $ _ $ _ =>
         (Conv.binop_conv (add_conv ctxt)
          then_conv (Conv.try_conv (add_simple_conv ctxt))) ctrm
    | _ $ _ => Conv.combination_conv
                  (add_conv ctxt) (add_conv ctxt) ctrm
    | Abs _ => Conv.abs_conv (fn (_, ctxt) => add_conv ctxt) ctxt ctrm
    | _ => Conv.all_conv ctrm)

fun add_tac ctxt = CSUBGOAL (fn (goal, i) =>
  CONVERSION
    (Conv.params_conv ~1 (fn ctxt =>
       (Conv.prems_conv ~1 (add_conv ctxt) then_conv
          Conv.concl_conv ~1 (add_conv ctxt))) ctxt) i)
```

A test case for this conversion is as follows

```
lemma "P (Suc (99 + 1)) ((0 + 0)::nat) (Suc (3 + 3 + 3)) (4 + 1)"
  apply(tactic {* add_tac @{context} 1 *})?
```

where it produces the goal state

```
goal (1 subgoal):
 1. P (Suc 100) 0 (Suc 9) ((4::'a) + (1::'a))
```

Solution for Exercise 4.6.1.

We use the timing function *timing_wrapper* from Recipe A.3. To measure any difference between the simproc and conversion, we will create mechanically terms involving additions and then set up a goal to be simplified. We have to be careful to set up the goal so that other parts of the simplifier do not interfere. For this we construct an unprovable goal which, after simplification, we are going to "prove" with the help of "**sorry**", that is the method *SkipProof.cheat_tac*

For constructing test cases, we first define a function that returns a complete binary tree whose leaves are numbers and the nodes are additions.

```
fun term_tree n =
let
  val count = ref 0;

  fun term_tree_aux n =
    case n of
      0 => (count := !count + 1; HOLogic.mk_number @{typ nat} (!count))
    | _ => Const (@{const_name "plus"}, @{typ "nat⇒nat⇒nat"})
```

```
            $ (term_tree_aux (n - 1)) $ (term_tree_aux (n - 1))
in
  term_tree_aux n
end
```

This function generates for example:

```
warning (Syntax.string_of_term @{context} (term_tree 2))
> (1 + 2) + (3 + 4)
```

The next function constructs a goal of the form `P ...` with a term produced by `term_tree` filled in.

```
fun goal n = HOLogic.mk_Trueprop (@{term "P::nat⇒ bool"} $ (term_tree n))
```

Note that the goal needs to be wrapped in a `Trueprop`. Next we define two tactics, `c_tac` and `s_tac`, for the conversion and simproc, respectively. The idea is to first apply the conversion (respectively simproc) and then prove the remaining goal using `cheat_tac`.

```
local
  fun mk_tac tac = timing_wrapper (EVERY1 [tac, K (SkipProof.cheat_tac
@{theory})])
in
val c_tac = mk_tac (add_tac @{context})
val s_tac = mk_tac (simp_tac (HOL_basic_ss addsimprocs [@{simproc add_sp}]))
end
```

This is all we need to let the conversion run against the simproc:

```
val _ = Goal.prove @{context} [] [] (goal 8) (K c_tac)
val _ = Goal.prove @{context} [] [] (goal 8) (K s_tac)
```

If you do the exercise, you can see that both ways of simplifying additions perform relatively similar with perhaps some advantages for the simproc. That means the simplifier, even if much more complicated than conversions, is quite efficient for tasks it is designed for. It usually does not make sense to implement general-purpose rewriting using conversions. Conversions only have clear advantages in special situations: for example if you need to have control over innermost or outermost rewriting, or when rewriting rules are lead to non-termination.

Appendix C

Comments for Authors

- This tutorial can be compiled on the command-line with:

  ```
  $ isabelle make
  ```

  You very likely need a recent snapshot of Isabelle in order to compile the tutorial. Some parts of the tutorial 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 tutorial. 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 tutorial, however, ensures that the environment markers are not printed.

- Line numbers can be printed using **ML** *%linenos {* ... *}* for ML-code or **lemma** *%linenos ...* for proofs. The tag is *%linenosgray* when the numbered text should be gray.

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] M. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[3] 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.

[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.