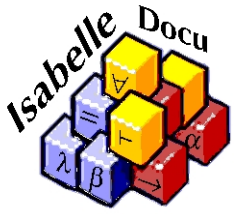


The Isabelle Programming Tutorial (fragment)

with contributions by:

Stefan Berghofer
Sascha Böhme
Jeremy Dawson
Alexander Krauss

February 13, 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, feel free—you are most welcome!

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 Implementation Manual describes Isabelle from a high-level perspective, documenting both the underlying concepts and some of the interfaces.

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 Many parts of this manual are outdated now, but some parts, particularly the chapters on tactics, are still useful.

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

Then of course there is:

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

1.3 Typographic Conventions

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

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

These boxes corresponds 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 usual user-level commands of Isabelle are written in bold, for example **lemma**, **foobar** and so on. We use `$` to indicate that a command needs to be run in a Unix-shell, for example:

```
$ ls -la
```

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

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

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
...

2.1 Including ML-Code

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

```
ML {*
   3 + 4
*}
> 7
```

Like normal Isabelle proof scripts, **ML**-commands can be evaluated by using the advance and undo buttons of your Isabelle environment. The code inside the **ML**-command can also contain value and function bindings, and even those can be undone when the proof script is retracted. As mentioned earlier, we will drop the **ML** *{** ... **}* 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 1 else (error "foo")
> "foo"
```

Section 2.10 will give more information about printing the main data structures of Isabelle, namely *term*, *cterm* and *thm*.

2.3 Antiquotations

The main advantage of embedding all code in a theory is that the code can contain references to entities defined on the logical level of Isabelle. By this we mean definitions, theorems, terms and so on. This kind of reference is realised with antiquotations. For example, one can print out the name of the current theory by typing

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

where *@{theory}* is an antiquotation that is substituted with the current theory (remember that we assumed we are inside the theory *FirstSteps*). The name of this theory can be extracted 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 allI}
> (⋀x. ?P x) ⟹ ∀x. ?P x
```

and simpsets:

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

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

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

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

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

These bindings are difficult to maintain and also can be accidentally overwritten by the user. This often breakes 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 introduction, we will learn more about these antiquotations: they greatly simplify Isabelle programming since one can directly access all kinds of logical elements from th ML-level.

2.4 Terms and Types

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

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

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

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

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

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

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

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

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

```
print_depth 50
```

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

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

where 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`.

2.5 Constructing Terms and Types Manually

While antiquotations are very convenient for constructing terms, they can only construct fixed terms (remember they are "linked" at compile-time). However, you often need to construct terms dynamically. For example, a function that returns the implication `⋀(x::τ). P x ⟹ Q x` taking `P`, `Q` and the type τ as arguments can only be written as:

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

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

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

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

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

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

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

(FIXME: expand the following point)

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

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

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

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

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

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 as:

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

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

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

A handy function for manipulating terms is `map_types`: it takes a function and applies it to every type in the 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 then apply it 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")
```

2.6 Type-Checking

You can freely construct and manipulate *terms*, 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 + 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
```

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

(FIXME: *ctyp_of*, *fastype_of*, *dummyT*)

2.7 Theorems

Just like *cterms*, theorems are abstract objects of type *thm* that can only be built by going through interfaces. As a consequence, every proof in Isabelle is correct by construction.

(FIXME reference LCF-philosophy)

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

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

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

```
let
  val thy = @{theory}

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

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

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

This code-snippet constructs the following proof:

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

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 the next section.

> ***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`.

[1] Note that `|>` is reverse application. See Section 2.12.

2.8 Storing Theorems

PureThy.add_thms_dynamic

2.9 Theorem Attributes

2.10 Printing Terms and Theorems

During development, you often want to inspect date 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 printing directions 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)
```

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 ctxt thm =
  let
    val {prop, ...} = crep_thm thm
  in
    str_of_cterm ctxt prop
  end
```

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

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

2.11 Operations on Constants (Names)

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

authentic syntax?

```
  @{const_name lfp}
```

constants in case-patterns?

In the meantime, lfp has been moved to the Inductive theory, so it is no longer called Lfp.lfp. If a `@{const_name}` antiquotation had been used, we would have gotten an error for this. Another advantage of the antiquotation is that we can then just write `@{const_name lfp}` rather than `@{const_name Lfp.lfp}` or whatever, and it expands to the correct name.

2.12 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.

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

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

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

or

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

and typographically more economical than

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

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

(FIXME: give a real world example involving theories)

Similarly, the combinator `#>` is the reverse function composition. It can be used to define 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 ` 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 applications. 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)

2.13 Misc

```
 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 due to Earley, which is driven by priority grammars. Parsers for outer syntax are built up by functional parsing combinators. These combinators are a well-established technique for parsing, which has, for example, been described in Paulson's classic ML-book [5]. Isabelle developers are usually concerned with writing these outer syntax parsers, either for new definitional packages or for calling tactics with specific arguments.

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

3.1 Building Generic Parsers

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

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

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

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

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

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

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

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

Parsers that explore alternatives can be constructed using the function *||*. 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, we need to prefix the parser with the function `Scan.error`. For example:

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

This "prefixing" is usually done by wrappers such as `OuterSyntax.command` (see Section 3.6 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 tokens, for example. However, this kind of manually wrapping is often already done by the surrounding infrastructure.

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

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

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

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

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

fails, while

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

succeeds.

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

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

After parsing is done, 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.

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

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

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

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

3.2 Parsing Theory Syntax

Most of the time, however, Isabelle developers have to deal with parsing tokens, not strings. This is because the parsers for the theory syntax, as well as the parsers for the arguments of proof methods the type `OuterLex.token` (which is identical to the type `OuterParse.token`). 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 below `Position.none` as argument since, at the moment, we are not interested in generating precise error messages. The following code

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

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

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

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

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

For convenience we define the function:

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

If 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:

```
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)

3.3 Positional Information

OuterParse.position

3.4 Parsing Inner Syntax

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

(FIXME funny output for a proposition)

3.5 Parsing Specifications

There are a number of special purpose parsers that help with parsing specifications of functions, inductive definitions and so on. For example the *OuterParse.target* reads a target in order to indicate a locale.

```
let
  val input = filtered_input "(in test)"
in
  parse OuterParse.target input
end
> ("test",[])
```

The function *OuterParse.opt_target* makes this parser "optional", that is wrapping the result into an option type and returning *NONE* if no target is present.

The function *OuterParse.fixes* reads an **and**-separated list of constants that can include type annotations and syntax translations. For example:[2]

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

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

Whenever types are given, then they are stored in the *SOME*s. If a syntax translation is present for a constant, then it is stored in the *Mixfix* data structure; no syntax translation is indicated by *NoSyn*.

(FIXME: should for-fixes take any syntax annotation?)

OuterParse.for_fixes is an "optional" that prefixes *OuterParse.fixes* with the command **for**. (FIXME give an example and explain more)

```
let
  val input = filtered_input
        "for foo::\"int ⇒ bool\" (\"FOO\" [100] 100) and bar::nat and
blonk"
in
   parse OuterParse.for_fixes input
end
> ([(foo, SOME ..., Mixfix ("FOO",[100],100)),
>   (bar, SOME ..., NoSyn),
>   (blonk, NONE, NoSyn)],[])
```

```
let
  val input = filtered_input "test_lemma[intro,foo,elim,dest!,bar]:"
in
  parse (SpecParse.thm_name ":") input
     |> fst |> snd |> (Attrib.pretty_attribs @{context}) |> (map
Pretty.string_of)
end
```

(FIXME: why is intro, elim and dest treated differently from bar?)

```
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)\"")

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

```
> ((((NONE, [(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")]), [])
```

The (outer?) parser for the package: returns optionally a locale; a list of predicate constants with optional type-annotation and optional syntax-annotation; a list of for-fixes (fixed parameters); and a list of rules where each rule has optionally a name and an attribute.

3.6 New Commands and Keyword Files

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

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

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

```
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**.

Pure and *HOL* are usually compiled during the installation of Isabelle. So log files for them should be already available. If not, then they can be conveniently compiled with the help of the build-script from the Isabelle distribution.

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

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

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

For the theory *Command.thy*, 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. 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))
```

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

```
  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"

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*.

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.

(FIXME: explain *Toplevel.print* etc)

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

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` 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.*

Note that in the code above we used 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 there are no ML-binding obtained 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 ∨ Q ⟹ Q ∨ P*. If the goal is not of this form, then they return the error message:

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

This means the tactics 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 up 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 2.10) 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)` applied to each goal; however this constant is invisible in the figure). This 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: A proof where we show the goal state as printed by the Isabelle system and as represented internally (highlighted boxes).

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

4.2 Simple Tactics

Let us start with the tactic *print_tac*, which is quite useful for low-level debugging of tactics. It just prints out a message and the current goal state. Processing the proof

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

gives:

```
foo message

False ⟹ True
1. False ⟹ True
```

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 an argument; it addresses the subgoal they 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
```

(FIXME: is it important to get the number of subgoals?)

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
```

Similarly 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(*drule bspec*)

```
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 following function

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

that transform the schematic variables of a theorem into free variables. Using this function for the first `RS`-expression above would produce 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 2.10 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 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 2.10). 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*

Note 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 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].*

A similar but less powerful function than *SUBPROOF* is *SUBGOAL*. It allows you to inspect a given subgoal. 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*. The final 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 36 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 just 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 45 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 laso 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 ∧ B"* **and** *"A ⟶ B ⟶C"* **and** *"∀x. D x"* **and** *"E ⟹ 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 ⟹ 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 ⟹ 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 *"((¬A) ∧ (∀x. B x)) ∧ (C ⟶ 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 will combinator 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 Rewriting and Simplifier Tactics

rewrite_goals_tac ObjectLogic.full_atomize_tac ObjectLogic.rulify_tac

4.5 Structured Proofs

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

"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 a few axioms. All other concepts, such as inductive predicates, datatypes, or recursive functions are defined in terms of those constants, and the desired properties, for example induction theorems, or recursion equations are derived from the definitions by a formal proof. Since it would be very tedious for a user to define complex inductive predicates or datatypes "by hand" just using the primitive operators of higher order logic, packages have been implemented automating such work. Thanks to those packages, the user can give a high-level specification, like a list of introduction rules or constructors, and the package then does all the low-level definitions and proofs behind the scenes. 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 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 suggested by Melham [3]. Due to its simplicity, this package will necessarily have a reduced functionality. It does neither support introduction rules involving arbitrary monotone operators, nor does it prove case analysis (or inversion) rules. Moreover, it only proves a weaker form of the rule induction theorem.

5.1 Examples of Inductive Definitions

Let us first give three examples showing how to define inductive predicates by hand and prove characteristic properties such as introduction rules and an induction rule. From these examples, we will then figure out a general method for defining inductive predicates. 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 producing the proofs that we will develop later.

As a first example, we consider the transitive closure of a relation *R*. It is an inductive predicate characterized by the two introduction rules

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

(FIXME first rule should be an "axiom")

Note that the *trcl* predicate has two different kinds of parameters: the first parameter *R* stays *fixed* throughout the definition, whereas the second and third parameter changes in the "recursive call".

Since an inductively defined predicate is the least predicate closed under a collection of introduction rules, we define the predicate *trcl R x y* in such a way that it holds if and only if *P x y* holds for every predicate *P* closed under the rules above. This gives rise to the definition

definition *"trcl R 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*. Note that we have to use the object implication ⟶ and object quantification ∀ for stating this definition.

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

(FIXME: add linenumbers to the proof below and the text above)

```
1  trcl_induct:
2   assumes asm: "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 (atomize (full))
5 (cut_tac asm)
6 (unfold trcl_def)
7 (drule spec[where x=P])
8 (assumption)
9
```

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

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

```
lemma trcl_base: "trcl R x x"
apply(unfold trcl_def)
apply(rule allI impI)+
apply(drule spec)
apply(assumption)
done
```

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

```
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
```

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

```
lemma trcl_step: "R x y ⟹ trcl R y z ⟹ trcl R x z"
  apply (unfold trcl_def)
  apply (rule allI impI)+
  proof -
    case (goal1 P)
    have g1: "R x y" by fact
    have g2: "∀P. (∀x. P x x) ⟶ (∀x y z. R x y ⟶ P y z ⟶ P x z) ⟶ P
y z" by fact
    have g3: "∀x. P x x" by fact
    have g4: "∀x y z. R x y ⟶ P y z ⟶ P x z" by fact
    show ?case
      apply (rule g4 [rule_format])
      apply (rule g1)
      apply (rule g2 [THEN spec [where x=P], THEN mp, THEN mp, OF g3, OF g4])
      done
```

```
  qed
```

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

$$\mathit{even}\ 0 \qquad \frac{\mathit{odd}\ m}{\mathit{even}\ (\mathit{Suc}\ m)} \qquad \frac{\mathit{even}\ m}{\mathit{odd}\ (\mathit{Suc}\ m)}$$

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

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

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

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

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

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

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

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

$$\frac{\forall y.\ R\ y\ x \longrightarrow accpart\ R\ y}{accpart\ R\ x}$$

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

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

The proof of the induction theorem is again straightforward:

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

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

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

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

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

5.2 The General Construction Principle

Before we start with the implementation, it is useful to describe the general form of inductive definitions that our package should accept. We closely follow the notation for inductive definitions introduced by Schwichtenberg [6] for the Minlog system. Let $R_1, \ldots, R_n$ be mutually inductive predicates and $\vec{p}$ be parameters. Then the introduction rules for $R_1, \ldots, R_n$ may have the form

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

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

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

where

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

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

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

where

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

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

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

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

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

that can be solved using the assumptions

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

5.3 The Interface

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

```
signature SIMPLE_INDUCTIVE_PACKAGE =
sig
  val add_inductive_i:
    ((Binding.binding * typ) * mixfix) list ->                    predicates
    (Binding.binding * typ) list ->                               parameters
    ((Binding.binding * Attrib.src list) * 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;
```

The function for external invocation of the package is called `add_inductive`, whereas the one for internal invocation is called `add_inductive_i`. Both of these functions take as arguments the names and types of the inductive predicates, the names and types of their parameters, the actual introduction rules and a *local theory*. They return a local theory containing the definition, together with a tuple containing the introduction and induction rules, which are stored in the local theory, too. In contrast to an ordinary theory, which simply consists of a type signature, as well as tables for constants, axioms and theorems, a local theory also contains additional

context information, such as locally fixed variables and local assumptions that may be used by the package. The type `local_theory` is identical to the type of *proof contexts* `Proof.context`, although not every proof context constitutes a valid local theory. Note that `add_inductive_i` expects the types of the predicates and parameters to be specified using the datatype `typ` of Isabelle's logical framework, whereas `add_inductive` expects them to be given as optional strings. If no string is given for a particular predicate or parameter, this means that the type should be inferred by the package. Additional *mixfix syntax* may be associated with the predicates and parameters as well. Note that `add_inductive_i` does not allow mixfix syntax to be associated with parameters, since it can only be used for parsing. The names of the predicates, parameters and rules are represented by the type `Binding.binding`. Strings can be turned into elements of the type `Binding.binding` using the function

```
Binding.name : string ->
  Binding.binding
```

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

```
fun read_specification' vars specs lthy =
let
  val specs' = map (fn (a, s) => [(a, [s])]) specs
  val ((varst, specst), _) = Specification.read_specification vars specs' lthy
  val specst' = map (apsnd the_single) specst
in
  (varst, specst')
end

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

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

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

to the list `specs` containing the parsed introduction rules. The function *`read_specification`* also returns the list `vars` of predicates and parameters that contains the inferred types as well. This list has to be chopped into the two lists `preds_syn'` and `params_syn'` for predicates and parameters, respectively. All variables occurring in a rule but not in the list of variables passed to *`read_specification`* will be bound by a meta-level universal quantifier.

Finally, *`read_specification`* also returns another local theory, but we can safely discard it. As an example, let us look at how we can use this function to parse the introduction rules of the *`trcl`* predicate:

```
Specification.read_specification
  [(Binding.name "trcl", NONE, NoSyn),
   (Binding.name "r", SOME "'a ⇒ 'a ⇒ bool", NoSyn)]
  [[((Binding.name "base", []), ["trcl r x x"])],
   [((Binding.name "step", []), ["trcl r x y ⟹ r y z ⟹ trcl r x z"])]]
  @{context}
> ((...,
>   [(...,
>     [Const ("all", ...) $ Abs ("x", TFree ("'a", ...),
>        Const ("Trueprop", ...) $
>          (Free ("trcl", ...) $ Free ("r", ...) $ Bound 0 $ Bound 0))]),
>    (...,
>     [Const ("all", ...) $ Abs ("x", TFree ("'a", ...),
>        Const ("all", ...) $ Abs ("y", TFree ("'a", ...),
>          Const ("all", ...) $ Abs ("z", TFree ("'a", ...),
>            Const ("==>", ...) $
>              (Const ("Trueprop", ...) $
>                (Free ("trcl", ...) $ Free ("r", ...) $ Bound 2 $ Bound 1)) $
>              (Const ("==>", ...) $ ... $ ...)))])]),
>  ...)
> : (((Binding.binding * typ) * mixfix) list *
>    (Attrib.binding * term list) list) * local_theory
```

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

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

The definition of the transitive closure should look as follows:

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

Even and odd numbers can be defined by

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

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

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

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

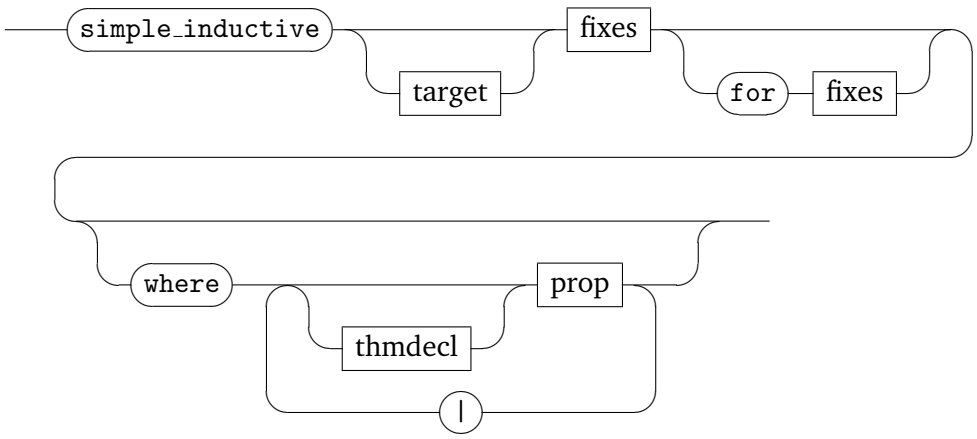
```
locale rel =
  fixes r :: "'a ⇒ 'a ⇒ bool"

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

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

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

the possible syntax of theory commands, and the parser for theory commands need not be concerned about the syntactic structure of terms and types.

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



Functional parsers For parsing terms and types, Isabelle uses a rather general and sophisticated algorithm due to Earley, which is driven by *priority grammars*. In contrast, parsers for theory syntax are built up using a set of combinators. Functional parsing using combinators is a well-established technique, which has been described by many authors, including Paulson [**?**] and Wadler [7]. The central idea is that a parser is a function of type `'a list -> 'b * 'a list`, where `'a` is a type of *tokens*, and `'b` is a type for encoding items that the parser has recognized. When a parser is applied to a list of tokens whose prefix it can recognize, it returns an encoding of the prefix as an element of type `'b`, together with the suffix of the list containing the remaining tokens. Otherwise, the parser raises an exception indicating a syntax error. The library for writing functional parsers in Isabelle can roughly be split up into two parts. The first part consists of a collection of generic parser combinators that are contained in the structure `Scan` defined in the file `Pure/General/scan.ML` in the Isabelle sources. While these combinators do not make any assumptions about the concrete structure of the tokens used, the second part of the library consists of combinators for dealing with specific token types. The following is an excerpt from the signature of `Scan`:

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

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

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

will try the alternative parser r. By writing

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

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

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

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

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

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

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

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

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

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

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

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

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

val ind_decl =
  parser >>
    (fn (((loc, preds), params), specs) =>
      Toplevel.local_theory loc (add_inductive preds params specs))
```

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

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

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

rather than with the more instructive message

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

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

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

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

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

which, apart from the local theory transformer, takes an optional name of a locale to be used as a basis for the local theory.

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

The whole parser for our command has type

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

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

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

which imperatively updates the parser table behind the scenes.

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

```
fun INDUCTION rules preds' Tss defs lthy1 lthy2 =
let
    val (Pnames, lthy3) = Variable.variant_fixes (replicate (length preds')
"P") lthy2;
    val Ps = map (fn (s, Ts) => Free (s, Ts ---> HOLogic.boolT)) (Pnames ~~
Tss);
    val cPs = map (cterm_of (ProofContext.theory_of lthy3)) Ps;
    val rules'' = map (subst_free (preds' ~~ Ps)) rules;

    fun prove_indrule ((R, P), Ts)  =
      let
        val (znames, lthy4) = Variable.variant_fixes (replicate (length Ts)
"z") lthy3;
        val zs = map Free (znames ~~ Ts)

        val prem = HOLogic.mk_Trueprop (list_comb (R, zs))
        val goal = Logic.list_implies (rules'', HOLogic.mk_Trueprop
(list_comb (P, zs)))
      in
        Goal.prove lthy4 [] [prem] goal
          (fn {prems, ...} => EVERY1
             ([ObjectLogic.full_atomize_tac,
               cut_facts_tac prems,
               K (rewrite_goals_tac defs)] @
              map (fn ct => dtac (inst_spec ct)) cPs @
              [assume_tac])) |>
          singleton (ProofContext.export lthy4 lthy1)
      end;
in
  map prove_indrule (preds' ~~ Ps ~~ Tss)
end
```

```
fun INTROS rules preds' defs lthy1 lthy2 =
let
```

```
  fun prove_intro (i, r) =
      Goal.prove lthy2 [] [] r
        (fn {prems, context = ctxt} => EVERY
           [ObjectLogic.rulify_tac 1,
            rewrite_goals_tac defs,
            REPEAT (resolve_tac [@{thm allI},@{thm impI}] 1),
            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
                THEN
                EVERY1 (map (fn prem =>
                  SUBPROOF (fn {prems = prems', concl, ...} =>
                    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) ctxt') prems1)
              end) ctxt 1]) |>
      singleton (ProofContext.export lthy2 lthy1)
in
  map_index prove_intro rules
end
```

Appendix A

Recipes

A.1 Accumulate a List of Theorems under a Name

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

Solution: This can be achieved using named theorem lists.

Named theorem lists can be set up using the code

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

and the command

setup *{* FooRules.setup *}*

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

Furthermore, the facts are made available on the user level under the dynamic fact name *foo*. For example we can declare three lemmas to be of the kind *foo* by:

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

and undeclare the first one by:

declare *rule1[foo del]*

and query the remaining ones with:

thm *foo*
> ?C
> ?B

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

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

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

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

A.2 Ad-hoc Transformations of Theorems

A.3 Useful Document Antiquotations

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

Solution: This can be achieved using document antiquotations.

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

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

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

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

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

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

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

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

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

(FIXME: say something about OuterParse.position)

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

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

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

To add some convenience and also to deal with large outputs, the user can give a partial specification by giving the abbreviation `"..."`. For example `(...,...)` for a pair.

Whereas in the antiquotation `@{ML_checked "piece_of_code"}` above, we have sent the expression `"val _ = piece_of_code"` to the compiler, in the second the wildcard `_` we will be replaced by a proper pattern. To do this we need to replace the `"..."` by `"_"` before sending the code to the compiler. The following function will do this:

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

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

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

The rest of the code of the antiquotation is

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

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

This extended antiquotation allows us to write

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

to obtain

```
true andalso false
> false
```

or

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

to obtain

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

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

A.4 Restricting the Runtime of a Function

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

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

Assume you defined the Ackermann function:

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

Now the call

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

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

```
TimeLimit.timeLimit (Time.fromSeconds 5) ackermann (4, 12)
  handle 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, because PolyML has a rich infrastructure for multithreading programming on which *timeLimit* relies.

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

A.5 Configuration Options

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

Solution: This can be achieved using configuration values.

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

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

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

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

or on the ML-level

```
setup_sval @{theory}
```

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

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

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

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

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

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

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

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

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

The retrival of this value yields now

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

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

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

> ***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 it avoid global references, which cause many headaches with the multithreaded execution of Isabelle.

A.6 Storing Data

Problem: Your tool needs to manage data.

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

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

```
local

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

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

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

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

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

A.7 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.4 on Page 72), 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.8 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/IffOracle.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 = {*
   Method.no_args (Method.SIMPLE_METHOD' iff_oracle_tac)
*} "Oracle-based decision procedure for chains of equivalences"
```

(FIXME: what does *Method.SIMPLE_METHOD'* do? ... what do you mean?)

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?)

Appendix B

Solutions to Most Exercises

Solution for Exercise 2.5.1.

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

Solution for Exercise 2.5.2.

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

Solution for Exercise 3.1.1.

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

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

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

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

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

Appendix C

Comments for Authors

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

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

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

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

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

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