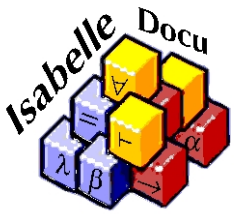

The Isabelle Cookbook

A Gentle Tutorial for Programming on the ML-Level of Isabelle (draft)

by Christian Urban with contributions from:

Stefan Berghofer
Jasmin Blanchette
Sascha Böhme
Lukas Bulwahn
Jeremy Dawson
Rafal Kolanski
Alexander Krauss
Tobias Nipkow
Andreas Schropp
Christian Sternagel

August 20, 2014

Contents

Chapter 1

Introduction

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

Richard Bornat, In *Defence of Programming*. [1]

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". We also hope the tutorial will encourage students and researchers to play with Isabelle and implement new ideas. The source code of Isabelle can look intimidating, but beginners can get by with knowledge of only a handful of concepts, a small number of functions and a few basic coding conventions. There is also a considerable amount of code written in Scala that allows Isabelle interface with the Jedit GUI. Explanation of this part is beyond this tutorial.

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 the Isabelle distribution.If something does not work, then please let us know. It is impossible for us to know every environment, operating system or editor in which Isabelle is used. If you have comments, criticism or like to add to the tutorial, please feel free—you are most welcome!! The tutorial is meant to be gentle and comprehensive. To achieve this we need your help and feedback.

1.1 Intended Audience and Prior Knowledge

This tutorial targets readers who already know how to use Isabelle for writing theories and proofs. We also assume that readers are familiar with the functional programming language ML, the language in which most of Isabelle is implemented. If you are unfamiliar with either of these two subjects, then you should first work through the Isabelle/HOL tutorial [4] or Paulson's book on ML [5]. Recently, Isabelle has adopted a sizable amount of Scala code for a slick GUI based on jEdit. This part of the code is beyond the interest of this tutorial, since it mostly does not concern the regular Isabelle developer.

1.2 Existing Documentation

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

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

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

The Isar Reference Manual provides specification material (like grammars, examples and so on) about Isar and its implementation.

Then of course there are:

The Isabelle sources. They are the ultimate reference for how things really work. Therefore you should not hesitate to look at the way things are actually implemented. While much of the Isabelle code is uncommented, some parts have very helpful comments—particularly the code about theorems and terms. Despite the lack of comments in most parts, it is often good to look at code that does similar things as you want to do and learn from it. This tutorial contains frequently pointers to the Isabelle sources. Still, the UNIX command `grep -R` is often your best friend while programming with Isabelle.[1] To understand the sources, it is often also necessary to track the change history of a file or files. The Mercurial repository[2] for Isabelle provides convenient interfaces to query the history of files and "change sets".

1.3 Typographic Conventions

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

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

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

[1] Or hypersearch if you work with jEdit.
[2] http://isabelle.in.tum.de/repos/isabelle/

```
3 + 4
```

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

```
3 + 4
> 7
```

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

```
$ grep -R Thy_Output *
```

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

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

Note that pointers to Isabelle files are hyperlinked to the tip of the Mercurial repository at http://isabelle.in.tum.de/repos/isabelle/, not the latest stable release of Isabelle.

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

1.4 How To Understand Isabelle Code

One of the more difficult aspects of any kind of programming is to understand code written by somebody else. This is aggravated in Isabelle by the fact that many parts of the code contain none or only few comments. There is one strategy that might be helpful to navigate your way: ML is an interactive programming environment, which means you can evaluate code on the fly (for example inside an **ML** `{*...*}` section). So you can copy (self-contained) chunks of existing code into a separate theory file and then study it alongside with examples. You can also install "probes" inside the copied code without having to recompile the whole Isabelle distribution. Such probes might be messages or printouts of variables (see chapter 2). Although PolyML also contains a debugger, it seems probing the code with explicit print statements is the most effective method for understanding what some piece of code is doing. However do not expect quick results with this! It is painful. Depending on the size of the code you are looking at, you will spend the better part of a quiet afternoon with it. And there seems to be no better way for understanding code in Isabelle.

1.5 Aaaaargh! My Code Does not Work Anymore

One unpleasant aspect of any code development inside a larger system is that one has to aim at a "moving target". Isabelle is no exception of this. Every update lets potentially all hell break loose, because other developers have changed code you are relying on. Cursing is somewhat helpful in such situations, but taking the view that incompatible code changes are a fact of life might be more gratifying. Isabelle is a research project. In most circumstances it is just impossible to make research backward compatible (imagine Darwin attempting to make the Theory of Evolution backward compatible).

However, there are a few steps you can take to mitigate unwanted interferences with code changes from other developers. First, you can base your code on the latest stable release of Isabelle (it is aimed to have one such release at least once every year). This might cut you off from the latest feature implemented in Isabelle, but at least you do not have to track side-steps or dead-ends in the Isabelle development. Of course this means also you have to synchronise your code at the next stable release. If you do not synchronise, be warned that code seems to "rot" very quickly. Another possibility is to get your code into the Isabelle distribution. For this you have to convince other developers that your code or project is of general interest. If you managed to do this, then the problem of the moving target goes away, because when checking in new code, developers are strongly urged to test it against Isabelle's code base. If your project is part of that code base, then maintenance is done by others. Unfortunately, this might not be a helpful advice for all types of projects. A lower threshold for inclusion has the Archive of Formal Proofs, short AFP.[3] This archive has been created mainly for formalisations that are interesting but not necessarily of general interest. If you have ML-code as part of a formalisation, then this might be the right place for you. There is no problem with updating your code after submission. At the moment developers are not as diligent with checking their code against the AFP than with checking agains the distribution, but generally problems will be caught and the developer, who caused them, is expected to fix them. So also in this case code maintenance is done for you.

1.6 Serious Isabelle ML-Programming

As already pointed out in the previous section, Isabelle is a joint effort of many developers. Therefore, disruptions that break the work of others are generally frowned upon. "Accidents" however do happen and everybody knows this. Still to keep them to a minimum, you can submit your changes first to a rather sophisticated *testboard*, which will perform checks of your changes against the Isabelle repository and against the AFP. The advantage of the testboard is that the testing is performed by rather powerful machines saving you lengthy tests on, for example, your own laptop. You can see the results of the testboard at

http://isabelle.in.tum.de/testboard/Isabelle/

[3] http://afp.sourceforge.net/

which is organised like a Mercurial repository. A green point next to a change indicates that the change passes the corresponding tests (for this of course you have to allow some time). You can summit any changes to the testboard using the command

```
$ hg push -f ssh://...@hgbroy.informatik.tu-muenchen.de\
   //home/isabelle-repository/repos/testboard
```

where the dots need to be replaced by your login name. Note that for pushing changes to the testboard you need to add the option *-f*, which should *never* be used with the main Isabelle repository. While the testboard is a great system for supporting Isabelle developers, its disadvantage is that it needs login permissions for the computers in Munich. So in order to use it, you might have to ask other developers to obtain one.

1.7 Some Naming Conventions in the Isabelle Sources

There are a few naming conventions in the Isabelle code that might aid reading and writing code. (Remember that code is written once, but read many times.) The most important conventions are:

- *t*, *u*, *trm* for (raw) terms; ML-type: *term*
- *ct*, *cu* for certified terms; ML-type: *cterm*
- *ty*, *T*, *U* for (raw) types; ML-type: *typ*
- *S* for sorts; ML-type: *sort*
- *th*, *thm* for theorems; ML-type: *thm*
- *foo_tac* for tactics; ML-type: *tactic*
- *thy* for theories; ML-type: *theory*
- *ctxt* for proof contexts; ML-type: *Proof.context*
- *lthy* for local theories; ML-type: *local_theory*
- *context* for generic contexts; ML-type *Context.generic*
- *mx* for mixfix syntax annotations; ML-type *mixfix*
- *prt* for pretty printing; ML-type *Pretty.T*
- *phi* for morphisms; ML-type *morphism*

1.8 Acknowledgements

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

- **Stefan Berghofer** wrote nearly all of the ML-code of the **simple_inductive**-package and the code for the *chunk*-antiquotation. He also wrote the first version of chapter 7 describing this package and has been helpful *beyond measure* with answering questions about Isabelle.
- **Jasmin Blanchette** helped greatly with section 3.9 and exercise 3.2.3.
- **Sascha Böhme** contributed the recipes in A.2, A.4 and A.5. He also wrote section 6.6 and helped with recipe A.3. Parts of section 2.5 are by him.
- **Lukas Bulwahn** made me aware of a problem with recursive parsers, contributed exercise 5.2.2 and contributed to the "introspection" of theorems in section 3.7.
- **Jeremy Dawson** wrote the first version of chapter 5 about parsing.
- **Armin Heller** helped with recipe A.6.
- **Rafal Kolanski** contributed to the "introspection" of theorems in section 3.7.
- **Alexander Krauss** wrote a very early version of the "first-steps" chapter and also contributed the material on *Named_Thms*.
- **Michael Norrish** proofread parts of the text.
- **Andreas Schropp** improved and corrected section 3.3 and contributed towards section 3.4.
- **Christian Sternagel** proofread the tutorial and made many improvemets to the text.
- **Dmitriy Traytel** suggested to use the ML-antiquotation *command_spec* in section 5.8, which simplified the code.

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

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

Chapter 2

First Steps

"The most effective debugging tool is still careful thought, coupled with judiciously placed print statements."

Brian Kernighan, in *Unix for Beginners*, 1979

Isabelle programming is done in ML. Just like lemmas and proofs, ML-code for Isabelle must be 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 First_Steps
imports Main
begin
...

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

2.1 Including ML-Code

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

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

If you work with ProofGeneral then like normal Isabelle scripts **ML**-commands can be evaluated by using the advance and undo buttons of your Isabelle environment. If you work with the Jedit GUI, then you just have to hover the cursor over the code and you see the evaluated result in the "Output" window.

As mentioned in the Introduction, we will drop the **ML** `{* ... *}` scaffolding whenever we show code. The lines prefixed with `">"` are not part of the code, rather

they indicate what the response is when the code is evaluated. There are also the commands **ML_val** and **ML_prf** for including ML-code. The first evaluates the given code, but any effect on the theory, in which the code is embedded, is suppressed. The second needs to be used if ML-code is defined inside a proof. For example

```
lemma test:
shows "True"
ML_prf {* writeln "Trivial!" *}
oops
```

However, both commands will only play minor roles in this tutorial (we most of the time make sure that the ML-code is defined outside proofs).

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 somewhere inside a theory by using the command **ML_file**. For example

```
theory First_Steps
imports Main
begin
...
ML_file "file_to_be_included.ML"
...
```

2.2 Printing and Debugging

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

```
writeln "any string"
> "any string"
```

will print out `"any string"`. 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 antiquotation `@{make_string}`:

```
writeln (@{make_string} 1)
> "1"
```

However, `@{makes_tring}` only works if the type of what is converted is monomorphic and not a function.

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

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

This function raises the exception *ERROR*, which will then be displayed by the infrastructure indicating that it is an error by painting the output red. Note that this exception is meant for "user-level" error messages seen by the "end-user". For messages where you want to indicate a genuine program error, then use the exception *Fail*.

Most often you want to inspect data of Isabelle's basic data structures, namely *term*, *typ*, *cterm*, *ctyp* and *thm*. Isabelle contains elaborate pretty-printing functions, which we will explain in more detail in Section 3.9. For now we just use the functions *writeln* from the structure *Pretty* and *pretty_term* from the structure *Syntax*. For more convenience, we bind them to the toplevel.

```
val pretty_term = Syntax.pretty_term
val pwriteln = Pretty.writeln
```

They can now be used as follows

```
pwriteln (pretty_term @{context} @{term "1::nat"})
> "1"
```

If there is more than one term to be printed, you can use the function *commas* and *block* to separate them.

```
fun pretty_terms ctxt trms =
  Pretty.block (Pretty.commas (map (pretty_term ctxt) trms))
```

You can also print out terms together with their typing information. For this you need to set the configuration value *show_types* to *true*.

```
val show_types_ctxt = Config.put show_types true @{context}
```

Now by using this context *pretty_term* prints out

```
pwriteln (pretty_term show_types_ctxt @{term "(1::nat, x)"})
> (1::nat, x::'a)
```

where *1* and *x* are displayed with their inferred type. Other configuration values that influence printing of terms include

- *show_brackets*
- *show_sorts*
- *eta_contract*

A *cterm* can be printed with the following function.

```
fun pretty_cterm ctxt ctrm =
  pretty_term ctxt (term_of ctrm)
```

Here the function *term_of* extracts the *term* from a *cterm*. More than one *cterm*s can be printed again with *commas*.

```
fun pretty_cterms ctxt ctrms =
  Pretty.block (Pretty.commas (map (pretty_cterm ctxt) ctrms))
```

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

```
fun pretty_thm ctxt thm =
  pretty_term ctxt (prop_of thm)
```

Theorems include schematic variables, such as *?P*, *?Q* and so on. They are needed in Isabelle in order to able to instantiate theorems when they are applied. For example the theorem *conjI* shown below can be used for any (typable) instantiation of *?P* and *?Q*.

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

However, in order to improve the readability when printing theorems, we can switch off the question marks as follows:

```
fun pretty_thm_no_vars ctxt thm =
let
  val ctxt' = Config.put show_question_marks false ctxt
in
  pretty_term ctxt' (prop_of thm)
end
```

With this function, theorem *conjI* is now printed as follows:

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

Again the functions *commas* and *block* help with printing more than one theorem.

```
fun pretty_thms ctxt thms =
  Pretty.block (Pretty.commas (map (pretty_thm ctxt) thms))

fun pretty_thms_no_vars ctxt thms =
  Pretty.block (Pretty.commas (map (pretty_thm_no_vars ctxt) thms))
```

Printing functions for *typ* are

```
fun pretty_typ ctxt ty = Syntax.pretty_typ ctxt ty
fun pretty_typs ctxt tys =
  Pretty.block (Pretty.commas (map (pretty_typ ctxt) tys))
```

respectively *ctyp*

```
fun pretty_ctyp ctxt cty = pretty_typ ctxt (typ_of cty)
fun pretty_ctyps ctxt ctys =
  Pretty.block (Pretty.commas (map (pretty_ctyp ctxt) ctys))
```

> ***Read More***
> *The simple conversion functions from Isabelle's main datatypes to* `strings` *are implemented in* `Pure/Syntax/syntax.ML`*. The configuration values that change the printing information are declared in* `Pure/Syntax/printer.ML`*.*

Note that for printing out several "parcels" of information that belong together, like a warning message consisting of a term and its type, you should try to print these parcels together in a single string. Therefore do *not* print out information as

```
pwriteln (Pretty.str "First half,");
pwriteln (Pretty.str "and second half.")
> First half,
> and second half.
```

but as a single string with appropriate formatting. For example

```
pwriteln (Pretty.str ("First half," ^ "\n" ^ "and second half."))
> First half,
> and second half.
```

To ease this kind of string manipulations, there are a number of library functions in Isabelle. For example, the function *cat_lines* concatenates a list of strings and inserts newlines in between each element.

```
pwriteln (Pretty.str (cat_lines ["foo", "bar"]))
> foo
> bar
```

Section 3.9 will explain the infrastructure that Isabelle provides for more elaborate pretty printing.

> ***Read More***
> *Most of the basic string functions of Isabelle are defined in* `Pure/library.ML`*.*

2.3 Combinators

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

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

```
fun I x = x
```

Another simple combinator is `K`, defined as

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

`K` "wraps" a function around `x` that 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. The reverse application allows you to read what happens in a top-down manner. This kind of coding should be familiar, if you have been exposed to Haskell's *do*-notation. Writing the function `inc_by_five` using the reverse application is much clearer than writing

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

or

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

and typographically more economical than

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

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

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

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

This function takes a term and a context as argument. If the term is of function type, then `apply_fresh_args` returns the term with distinct variables applied to it. For example below three variables are applied to the term `P`::`nat` ⇒ `int` ⇒ `unit` ⇒ `bool`:

```
let
  val trm = @{term "P::nat ⇒ int ⇒ unit ⇒ bool"}
  val ctxt = @{context}
in
  apply_fresh_args trm ctxt
   |> pretty_term ctxt
   |> pwriteln
end
> P z za zb
```

You can read off this behaviour from how `apply_fresh_args` is coded: in Line 2, the function `fastype_of` calculates the type of the term; `binder_types` in the next line produces the list of argument types (in the case above the list `[nat, int, unit]`); Line 4 pairs up each type with the string `z`; the function `variant_frees` generates for each `z` a unique name avoiding the given `f`; the list of name-type pairs is turned into a list of variable terms in Line 6, which in the last line is applied by the function

list_comb to the original term. In this last step we have to use the function *curry*, because *list_comb* expects the function and the variables list as a pair.

Functions like *apply_fresh_args* are often needed when constructing terms involving fresh variables. For this the infrastructure helps tremendously to avoid any name clashes. Consider for example:

```
let
  val trm = @{term "za::'a ⇒ 'b ⇒ 'c"}
  val ctxt = @{context}
in
  apply_fresh_args trm ctxt
   |> pretty_term ctxt
   |> pwriteln
end
> za z zb
```

where the *za* is correctly avoided.

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. This combinator is often used for setup functions inside the **setup**- or **local_setup**-command. These functions have to be of type *theory -> theory*, respectively *local_theory -> local_theory*. More than one such setup function can be composed with *#>*. Consider for example the following code, where we store the theorems *conjI*, *conjunct1* and *conjunct2* under alternative names.

```
1 local_setup {*
2 let
3   fun my_note name thm = Local_Theory.note ((name, []), [thm]) #> snd
4 in
5   my_note @{binding "foo_conjI"} @{thm conjI} #>
6   my_note @{binding "bar_conjunct1"} @{thm conjunct1} #>
7   my_note @{binding "bar_conjunct2"} @{thm conjunct2}
8 end *}
```

The function `my_note` in line 3 is just a wrapper for the function *note* in the structure *Local_Theory*; its purpose is to store a theorem under a name. In lines 5 to 6 we call this function to give alternative names for the three theorems. The point of *#>* is that you can sequence such function calls.

The remaining combinators we describe 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 or print out an intermediate result, for instance). The function

```
1 fun inc_by_three x =
2       x |> (fn x => x + 1)
3         |> tap (fn x => pwriteln (Pretty.str (@{make_string} 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. The function `tap` can only be used for side-calculations, because any value that is computed cannot be merged back into the "main waterfall". To do this, you can use the next combinator.

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

```
fun `f = fn x => (f x, x)
```

An example for this combinator is the function

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

which 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)
```

These two functions can, for example, be used to avoid explicit `lets` for intermediate values in functions that return pairs. As an example, suppose you want to separate a list of integers into two lists according to a threshold. If the threshold is `5`, the list `[1,6,2,5,3,4]` should be separated as `([1,2,3,4], [6,5])`. Such a function can be implemented as

```
fun separate i [] = ([], [])
  | separate i (x::xs) =
      let
        val (los, grs) = separate i xs
      in
        if i <= x then (los, x::grs) else (x::los, grs)
      end
```

where the return value of the recursive call is bound explicitly to the pair `(los, grs)`. However, this function can be implemented more concisely as

```
fun separate _ [] = ([], [])
  | separate i (x::xs) =
      if i <= x
      then separate i xs |!> cons x
      else separate i xs |>> cons x
```

avoiding the explicit `let`. While in this example the gain in conciseness is only small, in more complicated situations the benefit of avoiding `lets` can be substantial.

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)
```

The combinator `|!>>` plays a central rôle whenever your task is to update a theory and the update also produces a side-result (for example a theorem). Functions for such tasks return a pair whose second component is the theory and the fist component is the side-result. Using `|!>>`, you can do conveniently the update and also accumulate the side-results. Consider the following simple function.

```
1 fun acc_incs x =
2     x |> (fn x => ("", x))
3       |!>> (fn x => (x, x + 1))
4       |!>> (fn x => (x, x + 1))
5       |!>> (fn x => (x, x + 1))
```

The purpose of Line 2 is to just pair up the argument with a dummy value (since `|!>>` operates on pairs). Each of the next three lines just increment the value by one, but also nest the intermediate results to the left. For example

```
acc_incs 1
> ((((("", 1), 2), 3), 4)
```

You can continue this chain with:

```
acc_incs 1 |>> (fn x => (x, x + 2))
> (((((("", 1), 2), 3), 4), 6)
```

An example where this combinator is useful is as follows

```
let
  val ((names1, names2), _) =
    @{context}
    |> Variable.variant_fixes (replicate 4 "x")
    |>> Variable.variant_fixes (replicate 5 "x")
in
  (names1, names2)
end
> (["x", "xa", "xb", "xc"], ["xd", "xe", "xf", "xg", "xh"])
```

Its purpose is to create nine variants of the string `"x"` so that no variant will clash with another. Suppose for some reason we want to bind four variants to the lists `name1` and the rest to `name2`. In order to obtain non-clashing variants we have to thread the context through the function calls (the context records which variants have been previously created). For the first call we can use `|>`, but in the second and any further call to `variant_fixes` we have to use `|>>` in order to account for the result(s) obtained by previous calls.

A more realistic example for this combinator is the following code

```
val (((one_def, two_def), three_def), ctxt') =
  @{context}
  |> Local_Defs.add_def ((@{binding "One"}, NoSyn), @{term "1::nat"})
  |>> Local_Defs.add_def ((@{binding "Two"}, NoSyn), @{term "2::nat"})
  |>> Local_Defs.add_def ((@{binding "Three"}, NoSyn), @{term "3::nat"})
```

where we make three definitions, namely `One` $\equiv$ `1`, `Two` $\equiv$ `2` and `Three` $\equiv$ `3`. The point of this code is that we augment the initial context with the definitions. The result we are interested in is the augmented context, that is `ctxt'`, but also the side-results containing information about the definitions—the function `add_def` returns both as pairs. We can use this information for example to print out the definiens and the theorem corresponding to the definitions. For example for the first definition:

```
let
  val (one_trm, one_thm) = one_def
in
  pwriteln (pretty_term ctxt' one_trm);
```

```
  pwriteln (pretty_thm ctxt' one_thm)
end
> One
> One ≡ 1
```

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

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

When using combinators for writing functions in waterfall fashion, it is sometimes necessary to do some "plumbing" in order to fit functions together. We have already seen such plumbing in the function `apply_fresh_args`, where `curry` is needed for making the function `list_comb`, which works over pairs, to fit with the combinator `|>`. Such plumbing is also needed in situations where a function operates over lists, but one calculates only with a single element. An example is the function `check_terms`, whose purpose is to simultaneously type-check a list of terms. Consider the code:

```
let
  val ctxt = @{context}
in
  map (Syntax.parse_term ctxt) ["m + n", "m * n", "m - (n::nat)"]
  |> Syntax.check_terms ctxt
  |> pretty_terms ctxt
  |> pwriteln
end
> m + n, m * n, m - n
```

In this example we obtain three terms (using the function `parse_term`) whose variables `m` and `n` are of type `nat`. If you have only a single term, then `check_terms` needs plumbing. This can be done with the function `singleton`.[1] For example

```
1 let
2   val ctxt = @{context}
3 in
4   Syntax.parse_term ctxt "m - (n::nat)"
5   |> singleton (Syntax.check_terms ctxt)
6   |> pretty_term ctxt
7   |> pwriteln
8 end
9 > m - n
```

[1] There is already a function `check_term` in the file `Pure/Syntax/syntax.ML` that is implemented in terms of `singleton` and `check_terms`.

where in Line 5, the function operating over lists fits with the single term generated in Line 4.

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

Exercise 2.3.1: *Find out what the combinator* `K I` *does.*

2.4 ML-Antiquotations

Recall from Section 2.1 that code in Isabelle is always embedded in a theory. The main advantage of this is that the code can contain references to entities defined on the logical level of Isabelle. By this we mean references to definitions, theorems, terms and so on. These reference are realised in Isabelle with ML-antiquotations, often just called antiquotations.[2] Syntactically antiquotations are indicated by the `@`-sign followed by text wrapped in `{...}`. For example, one can print out the name of the current theory with the code

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

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

Note, however, that antiquotations are statically linked, that is their value is determined at "compile-time", not at "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 `"First_Steps"`, 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.

Another important antiquotation is `@{context}`. (What the difference between a theory and a context is will be described in Chapter 4.) A context is for example needed in order to use the function `print_abbrevs` that list of all currently defined abbreviations. For example

[2] Note that there are two kinds of antiquotations in Isabelle, which have very different purposes and infrastructures. The first kind, described in this section, are *ML-antiquotation*. They are used to refer to entities (like terms, types etc) from Isabelle's logic layer inside ML-code. The other kind of antiquotations are *document* antiquotations. They are used only in the text parts of Isabelle and their purpose is to print logical entities inside LaTeX-documents. Document antiquotations are part of the user level and therefore we are not interested in them in this Tutorial, except in Appendix A.1 where we show how to implement your own document antiquotations.

```
Proof_Context.print_abbrevs @{context}
> ...
> INTER ≡ INFI
> Inter ≡ Inf
> ...
```

The precise output of course depends on the abbreviations that are currently defined; this can change over time. You can also use antiquotations to refer to proved theorems: `@{thm ...}` for a single theorem

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

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

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

The thm-antiquotations can also be used for manipulating theorems. For example, if you need the version of the theorem `refl` that has a meta-equality instead of an equality, you can write

```
@{thm refl[THEN eq_reflection]}
> ?x ≡ ?x
```

The point of these antiquotations is that referring to theorems in this way makes your code independent from what theorems the user might have stored under this name (this becomes especially important when you deal with theorem lists; see Section 2.5).

It is also possible to prove lemmas with the antiquotation `@{lemma ... by ...}` whose first argument is a statement (possibly many of them separated by `and`) and the second is a proof. For example

```
val foo_thms = @{lemma "True" and "False ⟹ P" by simp_all}
```

The result can be printed out as follows.

```
foo_thms |> pretty_thms_no_vars @{context}
         |> pwriteln
> True, False ⟹ P
```

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

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

The function *dest_ss* returns a record containing all information stored in the simpset, but here we are only interested in the names of the simp-rules. Now you can feed in the current simpset into this function. The current simpset can be referred to using *simpset_of*.

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

Again, this way of referencing simpsets makes you independent from additions of lemmas to the simpset by the user, which can potentially cause loops in your code.

It is also possible to define your own antiquotations. But you should exercise care when introducing new ones, as they can also make your code also difficult to read. In the next chapter we describe how to construct terms with the (build in) antiquotation *@{term ...}*. A restriction of this antiquotation is that it does not allow you to use schematic variables in terms. If you want to have an antiquotation that does not have this restriction, you can implement your own using the function *inline* from the structure *ML_Antiquotation*. The code for the antiquotation *term_pat* is as follows.

```
1  val term_pat_setup =
2  let
3    val parser = Args.context -- Scan.lift Args.name_inner_syntax
4
5    fun term_pat (ctxt, str) =
6       str |> Proof_Context.read_term_pattern ctxt
7           |> ML_Syntax.print_term
8           |> ML_Syntax.atomic
9  in
10   ML_Antiquotation.inline @{binding "term_pat"} (parser >> term_pat)
11 end
```

To use it you also have to install it using **setup** like so

```
setup {* term_pat_setup *}
```

The parser in Line 2 provides us with a context and a string; this string is transformed into a term using the function *read_term_pattern* (Line 5); the next two lines transform the term into a string so that the ML-system can understand it. (All these functions will be explained in more detail in later sections.) An example for this antiquotation is:

```
@{term_pat "Suc (?x::nat)"}
> Const ("Suc", "nat ⇒ nat") $ Var (("x", 0), "nat")
```

which shows the internal representation of the term *Suc ?x*. Similarly we can write an antiquotation for type patterns. Its code is

```
val type_pat_setup =
let
  val parser = Args.context -- Scan.lift Args.name_inner_syntax

  fun typ_pat (ctxt, str) =
    let
      val ctxt' = Proof_Context.set_mode Proof_Context.mode_schematic ctxt
    in
      str |> Syntax.read_typ ctxt'
          |> ML_Syntax.print_typ
          |> ML_Syntax.atomic
      end
in
  ML_Antiquotation.inline @{binding "typ_pat"} (parser >> typ_pat)
end
```

which can be installed with

```
setup {* type_pat_setup *}
```

However, a word of warning is in order: Introducing new antiquotations should be done only after careful deliberations. They can potentially make your code harder to read, than making it easier.

> ***Read More***
> *The files* Pure/ML/ml_antiquotation.ML *and* Pure/ML/ml_antiquotations.ML *contain the infrastructure and definitions for most antiquotations. Most of the basic operations on ML-syntax are implemented in* Pure/ML/ml_syntax.ML.

2.5 Storing Data in Isabelle

Isabelle provides mechanisms for storing (and retrieving) arbitrary data. Before we delve into the details, let us digress a bit. Conventional wisdom has it that the type-system of ML ensures that an *'a list*, say, can only hold elements of the same type, namely *'a* (or whatever is substitued for it). Despite this common wisdom, however, it is possible to implement a universal type in ML, although by some arguably accidental features of ML. This universal type can be used to store data of different type into a single list. In fact, it allows one to inject and to project data of *arbitrary* type. This is in contrast to datatypes, which only allow injection and projection of data for some *fixed* collection of types. In light of the conventional wisdom cited above it

is important to keep in mind that the universal type does not destroy type-safety of ML: storing and accessing the data can only be done in a type-safe manner...though run-time checks are needed for that.

> ***Read More***
> *In Isabelle the universal type is implemented as the type* `Universal.universal` *in the file* `Pure/ML-Systems/universal.ML`.

We will show the usage of the universal type by storing an integer and a boolean into a single list. Let us first define injection and projection functions for booleans and integers into and from the type `Universal.universal`.

```
local
  val fn_int  = Universal.tag () : int  Universal.tag
  val fn_bool = Universal.tag () : bool Universal.tag
in
  val inject_int   = Universal.tagInject fn_int;
  val inject_bool  = Universal.tagInject fn_bool;
  val project_int  = Universal.tagProject fn_int;
  val project_bool = Universal.tagProject fn_bool
end
```

Using the injection functions, we can inject the integer `13` and the boolean value `true` into `Universal.universal`, and then store them in a `Universal.universal list` as follows:

```
val foo_list =
let
  val thirteen = inject_int 13
  val truth_val = inject_bool true
in
  [thirteen, truth_val]
end
```

The data can be retrieved with the projection functions defined above.

```
project_int (nth foo_list 0);
project_bool (nth foo_list 1)
> 13
> true
```

Notice that we access the integer as an integer and the boolean as a boolean. If we attempt to access the integer as a boolean, then we get a runtime error.

```
project_bool (nth foo_list 0)
> *** exception Match raised
```

This runtime error is the reason why ML is still type-sound despite containing a universal type.

Now, Isabelle heavily uses this mechanism for storing all sorts of data: theorem lists, simpsets, facts etc. Roughly speaking, there are two places where data can be stored in Isabelle: in *theories* and in *proof contexts*. Data such as simpsets are "global" and therefore need to be stored in a theory (simpsets need to be maintained across proofs and even across theories). On the other hand, data such as facts change inside a proof and are only relevant to the proof at hand. Therefore such data needs to be maintained inside a proof context, which represents "local" data. You can think of a theory as the "longterm memory" of Isabelle (nothing will be deleted from it), and a proof-context as a "shortterm memory" (it dynamically changes according to what is needed at the time).

For theories and proof contexts there are, respectively, the functors `Theory_Data` and `Proof_Data` that help with the data storage. Below we show how to implement a table in which you can store theorems and look them up according to a string key. The intention in this example is to be able to look up introduction rules for logical connectives. Such a table might be useful in an automatic proof procedure and therefore it makes sense to store this data inside a theory. Consequently we use the functor `Theory_Data`. The code for the table is:

```
1 structure Data = Theory_Data
2   (type T = thm Symtab.table
3    val empty = Symtab.empty
4    val extend = I
5    val merge = Symtab.merge (K true))
```

In order to store data in a theory, we have to specify the type of the data (Line 2). In this case we specify the type `thm Symtab.table`, which stands for a table in which `string`s can be looked up producing an associated `thm`. We also have to specify four functions to use this functor: namely how to initialise the data storage (Line 3), how to extend it (Line 4) and how two tables should be merged (Line 5). These functions correspond roughly to the operations performed on theories and we just give some sensible defaults.[3] The result structure `Data` contains functions for accessing the table (`Data.get`) and for updating it (`Data.map`). There is also the functions `Data.put`, which however is not relevant here. Below we define two auxiliary functions, which help us with accessing the table.

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

Since we want to store introduction rules associated with their logical connective, we can fill the table as follows.

[3] **FIXME: Say more about the assumptions of these operations.**

```
setup {*
  update "conj" @{thm conjI} #>
  update "imp"  @{thm impI}  #>
  update "all"  @{thm allI}
*}
```

The use of the command **setup** makes sure the table in the *current* theory is updated (this is explained further in section 4.1). The lookup can now be performed as follows.

```
lookup @{theory} "conj"
> SOME "⟦?P; ?Q⟧ ⟹ ?P ∧ ?Q"
```

An important point to note is that these tables (and data in general) need to be treated in a purely functional fashion. Although we can update the table as follows

```
setup {* update "conj" @{thm TrueI} *}
```

and accordingly, `lookup` now produces the introduction rule for `True`

```
lookup @{theory} "conj"
> SOME "True"
```

there are no references involved. This is one of the most fundamental coding conventions for programming in Isabelle. References interfere with the multithreaded execution model of Isabelle and also defeat its undo-mechanism. To see the latter, consider the following data container where we maintain a reference to a list of integers.

```
structure WrongRefData = Theory_Data
  (type T = (int list) Unsynchronized.ref
   val empty = Unsynchronized.ref []
   val extend = I
   val merge = fst)
```

We initialise the reference with the empty list. Consequently a first lookup produces `ref []`.

```
WrongRefData.get @{theory}
> ref []
```

For updating the reference we use the following function

```
fun ref_update n = WrongRefData.map
      (fn r => let val _ = r := n::(!r) in r end)
```

which takes an integer and adds it to the content of the reference. As before, we update the reference with the command **setup**.

```
setup {* ref_update 1 *}
```

A lookup in the current theory gives then the expected list `ref [1]`.

```
WrongRefData.get @{theory}
> ref [1]
```

So far everything is as expected. But, the trouble starts if we attempt to backtrack to the "point" before the **setup**-command. There, we would expect that the list is empty again. But since it is stored in a reference, Isabelle has no control over it. So it is not empty, but still `ref [1]`. Adding to the trouble, if we execute the **setup**-command again, we do not obtain `ref [1]`, but

```
WrongRefData.get @{theory}
> ref [1, 1]
```

Now imagine how often you go backwards and forwards in your proof scripts.[4] By using references in Isabelle code, you are bound to cause all hell to break loose. Therefore observe the coding convention: Do not use references for storing data!

> ***Read More***
> *The functors for data storage are defined in* `Pure/context.ML`*. Isabelle contains implementations of several container data structures, including association lists in* `Pure/General/alist.ML`*, directed graphs in* `Pure/General/graph.ML`*, and tables and symtables in* `Pure/General/table.ML`*.*

Storing data in a proof context is done in a similar fashion. As mentioned before, the corresponding functor is `Proof_Data`. With the following code we can store a list of terms in a proof context.

```
structure Data = Proof_Data
  (type T = term list
   fun init _ = [])
```

The init-function we have to specify must produce a list for when a context is initialised (possibly taking the theory into account from which the context is derived). We choose here to just return the empty list. Next we define two auxiliary functions for updating the list with a given term and printing the list.

[4] The same problem can be triggered in the Jedit GUI by making the parser to go over and over again over the **setup** command.

```
fun update trm = Data.map (fn trms => trm::trms)

fun print ctxt =
  case (Data.get ctxt) of
    [] => pwriteln (Pretty.str "Empty!")
  | trms => pwriteln (pretty_terms ctxt trms)
```

Next we start with the context generated by the antiquotation `@{context}` and update it in various ways.

```
let
  val ctxt0 = @{context}
  val ctxt1 = ctxt0 |> update @{term "False"}
                    |> update @{term "True ∧ True"}
  val ctxt2 = ctxt0 |> update @{term "1::nat"}
  val ctxt3 = ctxt2 |> update @{term "2::nat"}
in
  print ctxt0;
  print ctxt1;
  print ctxt2;
  print ctxt3
end
> Empty!
> True ∧ True, False
> 1
> 2, 1
```

Many functions in Isabelle manage and update data in a similar fashion. Consequently, such calculations with contexts occur frequently in Isabelle code, although the "context flow" is usually only linear. Note also that the calculation above has no effect on the underlying theory. Once we throw away the contexts, we have no access to their associated data. This is different for theories, where the command **setup** registers the data with the current and future theories, and therefore one can access the data potentially indefinitely.

Move elsewhere

For convenience there is an abstract layer, namely the type `Context.generic`, for treating theories and proof contexts more uniformly. This type is defined as follows

```
datatype generic =
  Theory of theory
| Proof of proof
```

5

There are two special instances of the data storage mechanism described above. The first instance implements named theorem lists using the functor `Named_Thms`. This

[5] **FIXME: say more about generic contexts.**

is because storing theorems in a list is such a common task. To obtain a named theorem list, you just declare

```
structure FooRules = Named_Thms
  (val name = @{binding "foo"}
   val description = "Theorems for foo")
```

and set up the *FooRules* with the command

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

This code declares a data container where the theorems are stored, an attribute *foo* (with the *add* and *del* options for adding and deleting theorems) and an internal ML-interface for retrieving and modifying the theorems. Furthermore, the theorems are made available on the user-level under the name *foo*. For example you can declare three lemmas to be a member of the theorem list *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]*

You can query the remaining ones with:

thm *foo*
> ?C
> ?B

On the ML-level, we can add theorems to the list with *FooRules.add_thm*:

```
setup {* Context.theory_map (FooRules.add_thm @{thm TrueI}) *}
```

The rules in the list can be retrieved using the function *FooRules.get*:

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

Note that this function takes a proof context as argument. This might be confusing, since the theorem list is stored as theory data. It becomes clear by knowing that the proof context contains the information about the current theory and so the function can access the theorem list in the theory via the context.

> ***Read More***
> *For more information about named theorem lists see* Pure/Tools/named_thms.ML.

The second special instance of the data storage mechanism are configuration values. They are used to enable users to configure tools without having to resort to the ML-level (and also to avoid references). Assume you want the user to control three values, say *bval* containing a boolean, *ival* containing an integer and *sval* containing a string. These values can be declared by

```
val bval = Attrib.setup_config_bool @{binding "bval"} (K false)
val ival = Attrib.setup_config_int @{binding "ival"} (K 0)
val sval = Attrib.setup_config_string @{binding "sval"} (K "some string")
```

where each value needs to be given a default. The user can now manipulate the values from the user-level of Isabelle with the command

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

On the ML-level these values can be retrieved using the function *get* from a proof context

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

or directly from a theory using the function *get_global*

```
Config.get_global @{theory} bval
> true
```

It is also possible to manipulate the configuration values from the ML-level with the functions *put* and *put_global*. For example

```
let
  val ctxt = @{context}
  val ctxt' = Config.put sval "foo" ctxt
  val ctxt'' = Config.put sval "bar" ctxt'
in
  (Config.get ctxt sval,
   Config.get ctxt' sval,
   Config.get ctxt'' sval)
end
> ("some string", "foo", "bar")
```

A concrete example for a configuration value is *simp_trace*, which switches on trace information in the simplifier. This can be used for example in the following proof

lemma
  shows *"(False ∨ True) ∧ True"*
proof *(rule conjI)*
  show *"False ∨ True"* **using** *[[simp_trace = true]]* **by** *simp*
next

```
  show "True" by simp
qed
```

in order to inspect how the simplifier solves the first subgoal.

> ***Read More***
> *For more information about configuration values see the files* `Pure/Isar/attrib.ML` *and* `Pure/config.ML`.

2.6 Summary

This chapter describes the combinators that are used in Isabelle, as well as a simple printing infrastructure for `term`, `cterm` and `thm`. The section on ML-antiquotations shows how to refer statically to entities from the logic level of Isabelle. Isabelle also contains mechanisms for storing arbitrary data in theory and proof contexts.

> ***Coding Conventions / Rules of Thumb***
>
> - *Print messages that belong together in a single string.*
> - *Do not use references in Isabelle code.*

Chapter 3

Isabelle Essentials

One man's obfuscation is another man's abstraction.

Frank Ch. Eigler on the Linux Kernel Mailing List,
24 Nov. 2009

Isabelle is built around a few central ideas. One central idea is the LCF-approach to theorem proving [3] where there is a small trusted core and everything else is built on top of this trusted core. The fundamental data structures involved in this core are certified terms and certified types, as well as theorems.

3.1 Terms and Types

In Isabelle, there are certified terms and uncertified terms (respectively types). Uncertified terms are often just called terms. One way to construct them is by using the antiquotation `@{term ...}`. For example

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

constructs the term `a + b = c`. The resulting term is printed using the internal representation corresponding to the datatype *term*, which is defined as follows:

```
1 datatype term =
2   Const of string * typ
3 | Free of string * typ
4 | Var of indexname * typ
5 | Bound of int
6 | Abs of string * typ * term
7 | $ of term * term
```

This datatype implements Church-style lambda-terms, where types are explicitly recorded in variables, constants and abstractions. The important point of having

terms is that you can pattern-match against them; this cannot be done with certified terms. As can be seen in Line 5, terms use the usual de Bruijn index mechanism for representing bound variables. For example in

```
@{term "λx y. x y"}
> Abs ("x", "'a ⇒ 'b", Abs ("y", "'a", Bound 1 $ Bound 0))
```

the indices refer to the number of Abstractions (*Abs*) that we need to skip until we hit the *Abs* that binds the corresponding variable. Constructing a term with dangling de Bruijn indices is possible, but will be flagged as ill-formed when you try to typecheck or certify it (see Section 3.5). Note that the names of bound variables are kept at abstractions for printing purposes, and so should be treated only as "comments". Application in Isabelle is realised with the term-constructor *$*.

Be careful if you pretty-print terms. Consider pretty-printing the abstraction term shown above:

```
@{term "λx y. x y"}
  |> pretty_term @{context}
  |> pwriteln
> λx. x
```

This is one common source for puzzlement in Isabelle, which has tacitly eta-contracted the output. You obtain a similar result with beta-redexes

```
let
  val redex = @{term "(λ(x::int) (y::int). x)"}
  val arg1 = @{term "1::int"}
  val arg2 = @{term "2::int"}
in
  pretty_term @{context} (redex $ arg1 $ arg2)
  |> pwriteln
end
> 1
```

There is a dedicated configuration value for switching off tacit eta-contractions, namely *eta_contract* (see Section 2.2), but none for beta-contractions. However you can avoid the beta-contractions by switching off abbreviations using the configuration value *show_abbrevs*. For example

```
let
  val redex = @{term "(λ(x::int) (y::int). x)"}
  val arg1 = @{term "1::int"}
  val arg2 = @{term "2::int"}
  val ctxt = Config.put show_abbrevs false @{context}
in
  pretty_term ctxt (redex $ arg1 $ arg2)
  |> pwriteln
end
> (λx y. x) 1 2
```

Isabelle makes a distinction between *free* variables (term-constructor `Free` and written on the user level in blue colour) and *schematic* variables (term-constructor `Var` and written with a leading question mark). Consider the following two examples

```
let
  val v1 = Var (("x", 3), @{typ bool})
  val v2 = Var (("x1", 3), @{typ bool})
  val v3 = Free ("x", @{typ bool})
in
  pretty_terms @{context} [v1, v2, v3]
  |> pwriteln
end
> ?x3, ?x1.3, x
```

When constructing terms, you are usually concerned with free variables (as mentioned earlier, you cannot construct schematic variables using the built in antiquotation `@{term ...}`). If you deal with theorems, you have to, however, observe the distinction. The reason is that only schematic variables can be instantiated with terms when a theorem is applied. A similar distinction between free and schematic variables holds for types (see below).

> ***Read More***
> *Terms and types are described in detail in [Impl. Man., Sec. 2.2]. Their definition and many useful operations are implemented in `Pure/term.ML`. For constructing terms involving HOL constants, many helper functions are defined in `HOL/Tools/hologic.ML`.*

Constructing terms via antiquotations has the advantage that only typable terms can be constructed. For example

```
@{term "x x"}
> Type unification failed: Occurs check!
```

raises a typing error, while it perfectly ok to construct the term with the raw ML-constructors:

```
let
  val omega = Free ("x", @{typ "nat ⇒ nat"}) $ Free ("x", @{typ nat})
in
  pwriteln (pretty_term @{context} omega)
end
> x x
```

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 3.1.1: *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 ≤ - 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 need to set the printing depth to a higher value by

declare `[[ML_print_depth = 50]]`

The antiquotation `@{prop ...}` constructs terms by inserting the usually invisible `Trueprop`-coercions whenever necessary. Consider for example the pairs

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

where it is not (since it is already constructed by a meta-implication). The purpose of the `Trueprop`-coercion is to embed formulae of an object logic, for example HOL, into the meta-logic of Isabelle. The coercion is needed whenever a term is constructed that will be proved as a theorem.

As already seen above, types can be constructed using the antiquotation `@{typ ...}`. For example:

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

The corresponding datatype is

```
datatype typ =
  Type  of string * typ list
| TFree of string * sort
| TVar  of indexname * sort
```

Like with terms, there is the distinction between free type variables (term-constructor `TFree`) and schematic type variables (term-constructor `TVar` and printed with a leading question mark). A type constant, like `int` or `bool`, are types with an empty list of argument types. However, it needs a bit of effort to show an example, because Isabelle always pretty prints types (unlike terms). Using just the antiquotation `@{typ "bool"}` we only see

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

which is the pretty printed version of `bool`. However, in PolyML (version $\geq$5.3) it is easy to install your own pretty printer. With the function below we mimic the behaviour of the usual pretty printer for datatypes (it uses pretty-printing functions which will be explained in more detail in Section 3.9).

```
local
  fun pp_pair (x, y) = Pretty.list "(" ")" [x, y]
  fun pp_list xs = Pretty.list "[" "]" xs
  fun pp_str s   = Pretty.str s
  fun pp_qstr s  = Pretty.quote (pp_str s)
  fun pp_int i   = pp_str (string_of_int i)
  fun pp_sort S  = pp_list (map pp_qstr S)
  fun pp_constr a args = Pretty.block [pp_str a, Pretty.brk 1, args]
in
fun raw_pp_typ (TVar ((a, i), S)) =
      pp_constr "TVar" (pp_pair (pp_pair (pp_qstr a, pp_int i), pp_sort S))
  | raw_pp_typ (TFree (a, S)) =
      pp_constr "TFree" (pp_pair (pp_qstr a, pp_sort S))
  | raw_pp_typ (Type (a, tys)) =
      pp_constr "Type" (pp_pair (pp_qstr a, pp_list (map raw_pp_typ tys)))
end
```

We can install this pretty printer with the function `addPrettyPrinter` as follows.

```
PolyML.addPrettyPrinter
  (fn _ => fn _ => ml_pretty o Pretty.to_ML o raw_pp_typ)
```

Now the type bool is printed out in full detail.

```
@{typ "bool"}
> Type ("HOL.bool", [])
```

When printing out a list-type

```
@{typ "'a list"}
> Type ("List.list", [TFree ("'a", ["HOL.type"])])
```

we can see the full name of the type is actually `List.list`, indicating that it is defined in the theory `List`. However, one has to be careful with names of types, because even if `fun` is defined in the theory `HOL`, it is still represented by their simple name.

```
@{typ "bool ⇒ nat"}
> Type ("fun", [Type ("HOL.bool", []), Type ("Nat.nat", [])])
```

We can restore the usual behaviour of Isabelle's pretty printer with the code

```
PolyML.addPrettyPrinter
  (fn _ => fn _ => ml_pretty o Pretty.to_ML o Proof_Display.pp_typ Pure.thy)
```

After that the types for booleans, lists and so on are printed out again the standard Isabelle way.

```
@{typ "bool"};
@{typ "'a list"}
> "bool"
> "'a List.list"
```

> ***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`.

3.2 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 manually. For example, a function that returns the implication $\bigwedge x::nat.\ P\ x \Longrightarrow Q\ x$ taking `P` and `Q` as arguments can only be written as:

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

The reason is that you cannot pass the arguments `P` and `Q` into an antiquotation.[1] For example the following does *not* work.

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

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

[1] At least not at the moment.

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

whereas with *make_wrong_imp* you obtain a term involving the *P* and *Q* from the antiquotation.

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

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

```
let
  val trm = @{term "P::bool ⇒ bool ⇒ bool"}
  val args = [@{term "True"}, @{term "False"}]
in
  list_comb (trm, args)
end
> Free ("P", "bool ⇒ bool ⇒ bool")
>    $ Const ("True", "bool") $ Const ("False", "bool")
```

Another handy function is *lambda*, which abstracts a variable in a term. For example

```
let
  val x_nat = @{term "x::nat"}
  val trm = @{term "(P::nat ⇒ bool) x"}
in
  lambda x_nat trm
end
> Abs ("x", "Nat.nat", Free ("P", "bool ⇒ bool") $ Bound 0)
```

In this example, *lambda* produces a de Bruijn index (i.e. *Bound 0*), and an abstraction, where it also records the type of the abstracted variable and for printing purposes also its name. Note that because of the typing annotation on *P*, the variable *x* in *P x* is of the same type as the abstracted variable. If it is of different type, as in

```
let
  val x_int = @{term "x::int"}
  val trm = @{term "(P::nat ⇒ bool) x"}
in
  lambda x_int trm
end
> Abs ("x", "int", Free ("P", "nat ⇒ bool") $ Free ("x", "nat"))
```

then the variable `Free ("x", "nat")` is *not* abstracted. This is a fundamental principle of Church-style typing, where variables with the same name still differ, if they have different type.

There is also the function `subst_free` with which terms can be replaced by other terms. For example below, we will replace in `f 0 x` the subterm `f 0` by `y`, and `x` by `True`.

```
let
  val sub1 = (@{term "(f::nat ⇒ nat ⇒ nat) 0"}, @{term "y::nat ⇒ nat"})
  val sub2 = (@{term "x::nat"}, @{term "True"})
  val trm = @{term "((f::nat ⇒ nat ⇒ nat) 0) x"}
in
  subst_free [sub1, sub2] trm
end
> Free ("y", "nat ⇒ nat") $ Const ("True", "bool")
```

As can be seen, `subst_free` does not take typability into account. However it takes alpha-equivalence into account:

```
let
  val sub = (@{term "(λy::nat. y)"}, @{term "x::nat"})
  val trm = @{term "(λx::nat. x)"}
in
  subst_free [sub] trm
end
> Free ("x", "nat")
```

Similarly the function `subst_bounds`, replaces lose bound variables with terms. To see how this function works, let us implement a function that strips off the outermost forall quantifiers in a term.

```
fun strip_alls t =
let
  fun aux (x, T, t) = strip_alls t |>> cons (Free (x, T))
in
  case t of
    Const (@{const_name All}, _) $ Abs body => aux body
  | _ => ([], t)
end
```

The function returns a pair consisting of the stripped off variables and the body of the universal quantification. For example

```
strip_alls @{term "∀x y. x = (y::bool)"}
> ([Free ("x", "bool"), Free ("y", "bool")],
>   Const ("op =", ...) $ Bound 1 $ Bound 0)
```

Note that we produced in the body two dangling de Bruijn indices. Later on we will also use the inverse function that builds the quantification from a body and a list of (free) variables.

```
fun build_alls ([], t) = t
  | build_alls (Free (x, T) :: vs, t) =
      Const (@{const_name "All"}, (T --> @{typ bool}) --> @{typ bool})
        $ Abs (x, T, build_alls (vs, t))
```

As said above, after calling `strip_alls`, you obtain a term with loose bound variables. With the function `subst_bounds`, you can replace these loose `Bound`s with the stripped off variables.

```
1 let
2   val (vrs, trm) = strip_alls @{term "∀x y. x = (y::bool)"}
3 in
4   subst_bounds (rev vrs, trm)
5   |> pretty_term @{context}
6   |> pwriteln
7 end
8 > x = y
```

Note that in Line 4 we had to reverse the list of variables that `strip_alls` returned. The reason is that the head of the list the function `subst_bounds` takes is the replacement for `Bound 0`, the next element for `Bound 1` and so on.

Notice also that this function might introduce name clashes, since we substitute just a variable with the name recorded in an abstraction. This name is by no means unique. If clashes need to be avoided, then we should use the function `dest_abs`, which returns the body where the loose de Bruijn index is replaced by a unique free variable. For example

```
let
  val body = Bound 0 $ Free ("x", @{typ nat})
in
  Term.dest_abs ("x", @{typ "nat ⇒ bool"}, body)
end
> ("xa", Free ("xa", "Nat.nat ⇒ bool") $ Free ("x", "Nat.nat"))
```

Sometimes it is necessary to manipulate de Bruijn indices in terms directly. There are many functions to do this. We describe only two. The first, `incr_boundvars`, increases by an integer the indices of the loose bound variables in a term. In the code below

```
@{term "∀x y z u. z = u"}
  |> strip_alls
  ||> incr_boundvars 2
  |> build_alls
```

```
  |> pretty_term @{context}
  |> pwriteln
> ∀x y z u. x = y
```

we first strip off the forall-quantified variables (thus creating two loose bound variables in the body); then we increase the indices of the loose bound variables by *2* and finally re-quantify the variables. As a result of *incr_boundvars*, we obtain now a term that has the equation between the first two quantified variables.

The second function, *loose_bvar1*, tests whether a term contains a loose bound of a certain index. For example

```
let
  val body = snd (strip_alls @{term "∀x y. x = (y::bool)"})
in
  [loose_bvar1 (body, 0),
   loose_bvar1 (body, 1),
   loose_bvar1 (body, 2)]
end
> [true, true, false]
```

There are also many convenient functions that construct specific HOL-terms in the structure *HOLogic*. For example *mk_eq* constructs an equality out of two terms. The types needed in this equality are calculated from the type of the arguments. For example

```
let
  val eq = HOLogic.mk_eq (@{term "True"}, @{term "False"})
in
  eq |> pretty_term @{context}
     |> pwriteln
end
> True = False
```

> ***Read More***
> *There are many functions in* Pure/term.ML, Pure/logic.ML *and* HOL/Tools/hologic.ML *that make manual constructions of terms and types easier.*

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

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

this does not work for picking out $\forall$-quantified terms. Because the function

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

will not correctly match the formula `"∀x::nat. P x"`:

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

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

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

because now

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

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

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

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

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

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

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

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

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

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

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

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

The antiquotation for properly referencing type constants is `@{type_name ...}`. For example

```
@{type_name "list"}
> "List.list"
```

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 ty1 ty2 = Type ("fun", [ty1, ty2])
```

This can be equally written with the combinator `-->` as:

```
fun make_fun_type ty1 ty2 = ty1 --> ty2
```

If you want to construct a function type with more than one argument type, then you can use `--->`.

```
fun make_fun_types tys ty = tys ---> ty
```

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

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

Here is an example:

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

If you want to obtain the list of free type-variables of a term, you can use the function `add_tfrees` (similarly `add_tvars` for the schematic type-variables). One would expect that such functions take a term as input and return a list of types. But their type is actually

```
Term.term -> (string * Term.sort) list -> (string * Term.sort) list
```

that is they take, besides a term, also a list of type-variables as input. So in order to obtain the list of type-variables of a term you have to call them as follows

```
Term.add_tfrees @{term "(a, b)"} []
> [("'b", ["HOL.type"]), ("'a", ["HOL.type"])]
```

The reason for this definition is that `add_tfrees` can be easily folded over a list of terms. Similarly for all functions named `add_*` in *Pure/term.ML*.

Exercise 3.2.1: *Write a function* `rev_sum : term -> term` *that takes a term of the form* t_1 `+` t_2 `+ ... +` t_n *(whereby* `n` *might be one) 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 3.2.2: *Write a function that takes two terms representing natural numbers in unary notation (like* `Suc (Suc (Suc 0))`*, and produces the number representing their sum.*

Exercise 3.2.3: *Write a function that removes trivial forall and exists quantifiers that do not quantify over any variables. For example the term* $\forall$`x y z. P x = P z` *should be transformed to* $\forall$`x z. P x = P z`*, deleting the quantification* `y`*. Hint: use the functions* `incr_boundvars` *and* `loose_bvar1`*.*

Exercise 3.2.4: *Write a function that takes an integer* `i` *and produces an Isabelle integer list from* `1` *upto* `i`*, and then builds the reverse of this list using* `rev`*. The relevant helper functions are* `upto`*,* `HOLogic.mk_number` *and* `HOLogic.mk_list`*.*

Exercise 3.2.5: *Implement the function, which we below name deBruijn, that depends on a natural number n>0 and constructs terms of the form:*

$$\begin{aligned} \mathit{rhs}\ n &\stackrel{\mathit{def}}{=} \bigwedge i{=}1\ldots n.\ P\,i \\ \mathit{lhs}\ n &\stackrel{\mathit{def}}{=} \bigwedge i{=}1\ldots n.\ P\,i = P\ (i + 1\ \mathit{mod}\ n) \longrightarrow \mathit{rhs}\ n \\ \mathit{deBruijn}\ n &\stackrel{\mathit{def}}{=} \mathit{lhs}\ n \longrightarrow \mathit{rhs}\ n \end{aligned}$$

This function returns for n=3 the term

$$\begin{aligned} &(P\ 1 = P\ 2 \longrightarrow P\ 1 \wedge P\ 2 \wedge P\ 3) \wedge \\ &(P\ 2 = P\ 3 \longrightarrow P\ 1 \wedge P\ 2 \wedge P\ 3) \wedge \\ &(P\ 3 = P\ 1 \longrightarrow P\ 1 \wedge P\ 2 \wedge P\ 3) \longrightarrow P\ 1 \wedge P\ 2 \wedge P\ 3 \end{aligned}$$

Make sure you use the functions defined in `HOL/Tools/hologic.ML` *for constructing the terms for the logical connectives.*[2]

3.3 Unification and Matching

As seen earlier, Isabelle's terms and types may contain schematic term variables (term-constructor `Var`) and schematic type variables (term-constructor `TVar`). These variables stand for unknown entities, which can be made more concrete by instantiations. Such instantiations might be a result of unification or matching. While in case of types, unification and matching is relatively straightforward, in case of terms the algorithms are substantially more complicated, because terms need higher-order versions of the unification and matching algorithms. Below we shall use the antiquotations `@{typ_pat ...}` and `@{term_pat ...}` from Section 2.4 in order to construct examples involving schematic variables.

Let us begin with describing the unification and matching functions for types. Both return type environments (ML-type `Type.tyenv`) which map schematic type variables to types and sorts. Below we use the function `typ_unify` from the structure `Sign` for unifying the types `?'a * ?'b` and `?'b list * nat`. This will produce the mapping, or type environment, `[?'a := ?'b list, ?'b := nat]`.

```
1 val (tyenv_unif, _) = let
2   val ty1 = @{typ_pat "?'a * ?'b"}
3   val ty2 = @{typ_pat "?'b list * nat"}
4 in
5   Sign.typ_unify @{theory} (ty1, ty2) (Vartab.empty, 0)
6 end
```

The environment `Vartab.empty` in line 5 stands for the empty type environment, which is needed for starting the unification without any (pre)instantiations. The `0` is an integer index that will be explained below. In case of failure, `typ_unify` will throw the exception `TUNIFY`. We can print out the resulting type environment bound to `tyenv_unif` with the built-in function `dest` from the structure `Vartab`.

```
Vartab.dest tyenv_unif
> [(("'a", 0), (["HOL.type"], "?'b List.list")),
>  (("'b", 0), (["HOL.type"], "nat"))]
```

The first components in this list stand for the schematic type variables and the second are the associated sorts and types. In this example the sort is the default sort `HOL.type`. Instead of `Vartab.dest`, we will use in what follows our own pretty-printing function from Figure 3.1 for `Type.tyenv`s. For the type environment in the example this function prints out the more legible:

[2] Thanks to Roy Dyckhoff for suggesting this exercise and working out the details.

```
fun pretty_helper aux env =
  env |> Vartab.dest
      |> map aux
      |> map (fn (s1, s2) => Pretty.block [s1, Pretty.str " := ", s2])
      |> Pretty.enum "," "[" "]"
      |> pwriteln

fun pretty_tyenv ctxt tyenv =
let
  fun get_typs (v, (s, T)) = (TVar (v, s), T)
  val print = pairself (pretty_typ ctxt)
in
  pretty_helper (print o get_typs) tyenv
end
```

Figure 3.1: A pretty printing function for type environments, which are produced by unification and matching.

```
pretty_tyenv @{context} tyenv_unif
> [?'a := ?'b list, ?'b := nat]
```

The way the unification function *typ_unify* is implemented using an initial type environment and initial index makes it easy to unify more than two terms. For example

```
1 val (tyenvs, _) = let
2   val tys1 = (@{typ_pat "?'a"}, @{typ_pat "?'b list"})
3   val tys2 = (@{typ_pat "?'b"}, @{typ_pat "nat"})
4 in
5   fold (Sign.typ_unify @{theory}) [tys1, tys2] (Vartab.empty, 0)
6 end
```

The index *0* in Line 5 is the maximal index of the schematic type variables occurring in *tys1* and *tys2*. This index will be increased whenever a new schematic type variable is introduced during unification. This is for example the case when two schematic type variables have different, incomparable sorts. Then a new schematic type variable is introduced with the combined sorts. To show this let us assume two sorts, say *s1* and *s2*, which we attach to the schematic type variables *?'a* and *?'b*. Since we do not make any assumption about the sorts, they are incomparable.

class *s1*
class *s2*

```
val (tyenv, index) = let
  val ty1 = @{typ_pat "?'a::s1"}
  val ty2 = @{typ_pat "?'b::s2"}
```

```
in
  Sign.typ_unify @{theory} (ty1, ty2) (Vartab.empty, 0)
end
```

To print out the result type environment we switch on the printing of sort information by setting `show_sorts` to true. This allows us to inspect the typing environment.

```
pretty_tyenv @{context} tyenv
> [?'a::s1 := ?'a1::{s1, s2}, ?'b::s2 := ?'a1::{s1, s2}]
```

As can be seen, the type variables `?'a` and `?'b` are instantiated with a new type variable `?'a1` with sort `{s1, s2}`. Since a new type variable has been introduced the `index`, originally being `0`, has been increased to `1`.

```
index
> 1
```

Let us now return to the unification problem `?'a * ?'b` and `?'b list * nat` from the beginning of this section, and the calculated type environment `tyenv_unif`:

```
pretty_tyenv @{context} tyenv_unif
> [?'a := ?'b list, ?'b := nat]
```

Observe that the type environment which the function `typ_unify` returns is *not* an instantiation in fully solved form: while `?'b` is instantiated to `nat`, this is not propagated to the instantiation for `?'a`. In unification theory, this is often called an instantiation in *triangular form*. These triangular instantiations, or triangular type environments, are used because of performance reasons. To apply such a type environment to a type, say `?'a * ?'b`, you should use the function `norm_type`:

```
Envir.norm_type tyenv_unif @{typ_pat "?'a * ?'b"}
> nat list * nat
```

Matching of types can be done with the function `typ_match` also from the structure `Sign`. This function returns a `Type.tyenv` as well, but might raise the exception `TYPE_MATCH` in case of failure. For example

```
val tyenv_match = let
  val pat = @{typ_pat "?'a * ?'b"}
  and ty  = @{typ_pat "bool list * nat"}
in
  Sign.typ_match @{theory} (pat, ty) Vartab.empty
end
```

Printing out the calculated matcher gives

```
pretty_tyenv @{context} tyenv_match
> [?'a := bool list, ?'b := nat]
```

Unlike unification, which uses the function `norm_type`, applying the matcher to a type needs to be done with the function `subst_type`. For example

```
Envir.subst_type tyenv_match @{typ_pat "?'a * ?'b"}
> bool list * nat
```

Be careful to observe the difference: always use `subst_type` for matchers and `norm_type` for unifiers. To show the difference, let us calculate the following matcher:

```
val tyenv_match' = let
  val pat = @{typ_pat "?'a * ?'b"}
  and ty  = @{typ_pat "?'b list * nat"}
in
  Sign.typ_match @{theory} (pat, ty) Vartab.empty
end
```

Now `tyenv_unif` is equal to `tyenv_match'`. If we apply `norm_type` to the type `?'a * ?'b` we obtain

```
Envir.norm_type tyenv_match' @{typ_pat "?'a * ?'b"}
> nat list * nat
```

which does not solve the matching problem, and if we apply `subst_type` to the same type we obtain

```
Envir.subst_type tyenv_unif @{typ_pat "?'a * ?'b"}
> ?'b list * nat
```

which does not solve the unification problem.

> ***Read More***
> *Unification and matching for types is implemented in* `Pure/type.ML`*. The "interface" functions for them are in* `Pure/sign.ML`*. Matching and unification produce type environments as results. These are implemented in* `Pure/envir.ML`*. This file also includes the substitution and normalisation functions, which apply a type environment to a type. Type environments are lookup tables which are implemented in* `Pure/term_ord.ML`*.*

Unification and matching of terms is substantially more complicated than the type-case. The reason is that terms have abstractions and, in this context, unification or matching modulo plain equality is often not meaningful. Nevertheless, Isabelle implements the function `first_order_match` for terms. This matching function returns a type environment and a term environment. To pretty print the latter we use the function `pretty_env`:

```
fun pretty_env ctxt env =
let
  fun get_trms (v, (T, t)) = (Var (v, T), t)
  val print = pairself (pretty_term ctxt)
in
  pretty_helper (print o get_trms) env
end
```

As can be seen from the *get_trms*-function, a term environment associates a schematic term variable with a type and a term. An example of a first-order matching problem is the term *P (λa b. Q b a)* and the pattern *?X ?Y*.

```
val (_, fo_env) = let
  val fo_pat = @{term_pat "(?X::(nat⇒nat⇒nat)⇒bool) ?Y"}
  val trm_a = @{term "P::(nat⇒nat⇒nat)⇒bool"}
  val trm_b = @{term "λa b. (Q::nat⇒nat⇒nat) b a"}
  val init = (Vartab.empty, Vartab.empty)
in
  Pattern.first_order_match @{theory} (fo_pat, trm_a $ trm_b) init
end
```

In this example we annotated types explicitly because then the type environment is empty and can be ignored. The resulting term environment is

```
pretty_env @{context} fo_env
> [?X := P, ?Y := λa b. Q b a]
```

The matcher can be applied to a term using the function *subst_term* (remember the same convention for types applies to terms: *subst_term* is for matchers and *norm_term* for unifiers). The function *subst_term* expects a type environment, which is set to empty in the example below, and a term environment.

```
let
  val trm = @{term_pat "(?X::(nat⇒nat⇒nat)⇒bool) ?Y"}
in
  Envir.subst_term (Vartab.empty, fo_env) trm
  |> pretty_term @{context}
  |> pwriteln
end
> P (λa b. Q b a)
```

First-order matching is useful for matching against applications and variables. It can also deal with abstractions and a limited form of alpha-equivalence, but this kind of matching should be used with care, since it is not clear whether the result is meaningful. A meaningful example is matching *λx. P x* against the pattern *λy. ?X y*. In this case, first-order matching produces *[?X := P]*.

```
let
  val fo_pat = @{term_pat "λy. (?X::nat⇒bool) y"}
  val trm = @{term "λx. (P::nat⇒bool) x"}
  val init = (Vartab.empty, Vartab.empty)
in
  Pattern.first_order_match @{theory} (fo_pat, trm) init
  |> snd
  |> pretty_env @{context}
end
> [?X := P]
```

Unification of abstractions is more thoroughly studied in the context of higher-order pattern unification and higher-order pattern matching. A *pattern* is a well-formed term in which the arguments to every schematic variable are distinct bounds. In particular this excludes terms where a schematic variable is an argument of another one and where a schematic variable is applied twice with the same bound variable. The function `pattern` in the structure `Pattern` tests whether a term satisfies these restrictions under the assumptions that it is beta-normal, well-typed and has no loose bound variables.

```
let
  val trm_list =
        [@{term_pat "?X"}, @{term_pat "a"},
         @{term_pat "f (λa b. ?X a b) c"},
         @{term_pat "λa b. (op +) a b"},
         @{term_pat "λa. (op +) a ?Y"}, @{term_pat "?X ?Y"},
         @{term_pat "λa b. ?X a b ?Y"}, @{term_pat "λa. ?X a a"},
         @{term_pat "?X a"}]
in
  map Pattern.pattern trm_list
end
> [true, true, true, true, true, false, false, false, false]
```

The point of the restriction to patterns is that unification and matching are decidable and produce most general unifiers, respectively matchers. Note that *both* terms to be unified have to be higher-order patterns for this to work. The exception `Pattern` indicates failure in this regard. In this way, matching and unification can be implemented as functions that produce a type and term environment (unification actually returns a record of type `Envir.env` containing a max-index, a type environment and a term environment). The corresponding functions are `match` and `unify`, both implemented in the structure `Pattern`. An example for higher-order pattern unification is

```
let
  val trm1 = @{term_pat "λx y. g (?X y x) (f (?Y x))"}
  val trm2 = @{term_pat "λu v. g u (f u)"}
  val init = Envir.empty 0
  val env = Pattern.unify @{theory} (trm1, trm2) init
```

```
in
  pretty_env @{context} (Envir.term_env env)
end
> [?X := λy x. x, ?Y := λx. x]
```

The function *Envir.empty* generates a record with a specified max-index for the schematic variables (in the example the index is *0*) and empty type and term environments. The function *Envir.term_env* pulls out the term environment from the result record. The corresponding function for type environment is *Envir.type_env*. An assumption of this function is that the terms to be unified have already the same type. In case of failure, the exceptions that are raised are either *Pattern*, *MATCH* or *Unif*.

As mentioned before, unrestricted higher-order unification, respectively unrestricted higher-order matching, is in general undecidable and might also not posses a single most general solution. Therefore Isabelle implements the unification function *unifiers* so that it returns a lazy list of potentially infinite unifiers. An example is as follows

```
val uni_seq =
let
  val trm1 = @{term_pat "?X ?Y"}
  val trm2 = @{term "f a"}
  val init = Envir.empty 0
in
  Unify.unifiers (@{theory}, init, [(trm1, trm2)])
end
```

The unifiers can be extracted from the lazy sequence using the function *Seq.pull*. In the example we obtain three unifiers *un1* ... *un3*.

```
val SOME ((un1, _), next1) = Seq.pull uni_seq;
val SOME ((un2, _), next2) = Seq.pull next1;
val SOME ((un3, _), next3) = Seq.pull next2;
val NONE = Seq.pull next3
```

[3]

We can print them out as follows.

```
pretty_env @{context} (Envir.term_env un1);
pretty_env @{context} (Envir.term_env un2);
pretty_env @{context} (Envir.term_env un3)
> [?X := λa. a, ?Y := f a]
> [?X := f, ?Y := a]
> [?X := λb. f a]
```

[3] **FIXME: what is the list of term pairs in the unifier: flex-flex pairs?**

In case of failure the function *unifiers* does not raise an exception, rather returns the empty sequence. For example

```
let
  val trm1 = @{term "a"}
  val trm2 = @{term "b"}
  val init = Envir.empty 0
in
  Unify.unifiers (@{theory}, init, [(trm1, trm2)])
  |> Seq.pull
end
> NONE
```

In order to find a reasonable solution for a unification problem, Isabelle also tries first to solve the problem by higher-order pattern unification. Only in case of failure full higher-order unification is called. This function has a built-in bound, which can be accessed and manipulated as a configuration value. For example

```
Config.get_global @{theory} (Unify.search_bound)
> Int 60
```

If this bound is reached during unification, Isabelle prints out the warning message *"Unification bound exceeded"* and plenty of diagnostic information (sometimes annoyingly plenty of information).

For higher-order matching the function is called *matchers* implemented in the structure *Unify*. Also this function returns sequences with possibly more than one matcher. Like *unifiers*, this function does not raise an exception in case of failure, but returns an empty sequence. It also first tries out whether the matching problem can be solved by first-order matching.

Higher-order matching might be necessary for instantiating a theorem appropriately. More on this will be given in Sections 3.7. Here we only have a look at a simple case, namely the theorem *spec*:

thm *spec*
> $\forall x.\ ?P\ x \Longrightarrow ?P\ ?x$

as an introduction rule. Applying it directly can lead to unexpected behaviour since the unification has more than one solution. One way round this problem is to instantiate the schematic variables *?P* and *?x*. instantiation function for theorems is *instantiate_normalize* from the structure *Drule*. One problem, however, is that this function expects the instantiations as lists of *ctyp* and *cterm* pairs:

```
instantiate_normalize: (ctyp * ctyp) list * (cterm * cterm) list -> thm -> thm
```

This means we have to transform the environment the higher-order matching function returns into such an instantiation. For this we use the functions *term_env* and *type_env*, which extract from an environment the corresponding variable mappings

for schematic type and term variables. These mappings can be turned into proper *ctyp*-pairs with the function

```
fun prep_trm thy (x, (T, t)) =
  (cterm_of thy (Var (x, T)), cterm_of thy t)
```

and into proper *cterm*-pairs with

```
fun prep_ty thy (x, (S, ty)) =
  (ctyp_of thy (TVar (x, S)), ctyp_of thy ty)
```

We can now calculate the instantiations from the matching function.

```
1 fun matcher_inst thy pat trm i =
2 let
3   val univ = Unify.matchers thy [(pat, trm)]
4   val env = nth (Seq.list_of univ) i
5   val tenv = Vartab.dest (Envir.term_env env)
6   val tyenv = Vartab.dest (Envir.type_env env)
7 in
8   (map (prep_ty thy) tyenv, map (prep_trm thy) tenv)
9 end
```

In Line 3 we obtain the higher-order matcher. We assume there is a finite number of them and select the one we are interested in via the parameter *i* in the next line. In Lines 5 and 6 we destruct the resulting environments using the function *dest*. Finally, we need to map the functions *prep_trm* and *prep_ty* over the respective environments (Line 8). As a simple example we instantiate the *spec* rule so that its conclusion is of the form *Q True*.

```
1 let
2   val pat = Logic.strip_imp_concl (prop_of @{thm spec})
3   val trm = @{term "Trueprop (Q True)"}
4   val inst = matcher_inst @{theory} pat trm 1
5 in
6   Drule.instantiate_normalize inst @{thm spec}
7 end
8 > ∀x. Q x ⟹ Q True
```

Note that we had to insert a *Trueprop*-coercion in Line 3 since the conclusion of *spec* contains one.

> ***Read More***
> *Unification and matching of higher-order patterns is implemented in* Pure/pattern.ML*. This file also contains a first-order matcher for terms. Full higher-order unification is implemented in* Pure/unify.ML*. It uses lazy sequences which are implemented in* Pure/General/seq.ML*.*

3.4 Sorts (TBD)

Type classes are formal names in the type system which are linked to predicates of one type variable (via the axclass mechanism) and thereby express extra properties on types, to be propagated by the type system. The type-in-class judgement is defined via a simple logic over types, with inferences solely based on modus ponens, instantiation and axiom use. The declared axioms of this logic are called an order-sorted algebra (see Schmidt-Schauss). It consists of an acyclic subclass relation and a set of image containment declarations for type constructors, called arities.

A well-behaved high-level view on type classes has long been established (cite Haftmann-Wenzel): the predicate behind a type class is the foundation of a locale (for context-management reasons) and may use so-called type class parameters. These are type-indexed constants dependent on the sole type variable and are implemented via overloading. Overloading a constant means specifying its value on a type based on a well-founded reduction towards other values of constants on types. When instantiating type classes (i.e. proving arities) you are specifying overloading via primitive recursion.

Sorts are finite intersections of type classes and are implemented as lists of type class names. The empty intersection, i.e. the empty list, is therefore inhabited by all types and is called the topsort.

Free and schematic type variables are always annotated with sorts, thereby restricting the domain of types they quantify over and corresponding to an implicit hypothesis about the type variable.

ML `{* Sign.classes_of @{theory} *}`

ML `{* Sign.of_sort @{theory} *}`

> ***Read More***
> *Classes, sorts and arities are defined in* `Pure/term.ML`.
>
> `Pure/sorts.ML` *contains comparison and normalization functionality for sorts, manages the order sorted algebra and offers an interface for reinterpreting derivations of type in class judgements* `Pure/defs.ML` *manages the constant dependency graph and keeps it well-founded (its define function doesn't terminate for complex non-well-founded dependencies)* `Pure/axclass.ML` *manages the theorems that back up subclass and arity relations and provides basic infrastructure for establishing the high-level view on type classes* `Pure/sign.ML` *is a common interface to all the type-theory-like declarations (especially names, constants, paths, type classes) a theory acquires by theory extension mechanisms and manages associated certification functionality. It also provides the most needed functionality from individual underlying modules.*

3.5 Type-Checking

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

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

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

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

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

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

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

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

where no error is detected.

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

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

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

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

Exercise 3.5.1: *Check that the function defined in Exercise 3.2.1 returns a result that type-checks. See what happens to the solutions of this exercise given in Appendix B when they receive an ill-typed term as input.*

3.6 Certified Terms and Certified Types

You can freely construct and manipulate *term*s and *typ*es, 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 *term*s, which are just trees, *cterm*s are abstract objects that are guaranteed to be type-correct, and they can only be constructed via "official interfaces".

Certification is always relative to a theory context. For example you can write:

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

This can also be written with an antiquotation:

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

Attempting to obtain the certified term for

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

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

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

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

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

or with the antiquotation:

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

Since certified terms are, unlike terms, abstract objects, we cannot pattern-match against them. However, we can construct them. For example the function *apply* produces a certified application.

```
Thm.apply @{cterm "P::nat ⇒ bool"} @{cterm "3::nat"}
> P 3
```

Similarly the function *list_comb* from the structure *Drule* applies a list of *cterms*.

```
let
  val chead = @{cterm "P::unit ⇒ nat ⇒ bool"}
  val cargs = [@{cterm "()"}, @{cterm "3::nat"}]
in
  Drule.list_comb (chead, cargs)
end
> P () 3
```

> ***Read More***
> *For functions related to* cterms *and* ctyps *see the files* Pure/thm.ML, Pure/more_thm.ML *and* Pure/drule.ML.

3.7 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. This follows the tradition of the LCF-approach.

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  val my_thm =
2  let
3    val assm1 = @{cprop "⋀(x::nat). P x ⟹ Q x"}
4    val assm2 = @{cprop "(P::nat ⇒ bool) t"}
5
6    val Pt_implies_Qt =
7      Thm.assume assm1
8      |> Thm.forall_elim @{cterm "t::nat"}
9
10   val Qt = Thm.implies_elim Pt_implies_Qt (Thm.assume assm2)
11 in
12   Qt
13   |> Thm.implies_intr assm2
14   |> Thm.implies_intr assm1
15 end
```

Note that in Line 3 and 4 we use the antiquotation *@{cprop ...}*, which inserts necessary *Trueprop*s.

If we print out the value of *my_thm* then we see only the final statement of the theorem.

```
pwriteln (pretty_thm @{context} my_thm)
> ⟦⋀x. P x ⟹ Q x; P t⟧ ⟹ Q t
```

However, internally the code-snippet constructs the following proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\bigwedge x.\ P\ x \Longrightarrow Q\ x \vdash \bigwedge x.\ P\ x \Longrightarrow Q\ x}\,(assume)}{\bigwedge x.\ P\ x \Longrightarrow Q\ x \vdash P\ t \Longrightarrow Q\ t}\,(\textstyle\bigwedge\text{-}elim) \quad \dfrac{}{P\ t \vdash P\ t}\,(assume)}{\bigwedge x.\ P\ x \Longrightarrow Q\ x, P\ t \vdash Q\ t}\,(\Longrightarrow\text{-}elim)}{\bigwedge x.\ P\ x \Longrightarrow Q\ x \vdash P\ t \Longrightarrow Q\ t}\,(\Longrightarrow\text{-}intro)}{\vdash \llbracket \bigwedge x.\ P\ x \Longrightarrow Q\ x;\ P\ t \rrbracket \Longrightarrow Q\ t}\,(\Longrightarrow\text{-}intro)$$

While we obtained a theorem as result, this theorem is not yet stored in Isabelle's theorem database. Consequently, it cannot be referenced on the user level. One way to store it in the theorem database is by using the function *note* from the structure *Local_Theory* (the Isabelle command **local_setup** will be explained in more detail in Section 4.3).

```
local_setup {*
  Local_Theory.note ((@{binding "my_thm"}, []), [my_thm]) #> snd *}
```

The third argument of *note* is the list of theorems we want to store under a name. We can store more than one under a single name. The first argument of *note* is the name under which we store the theorem or theorems. The second argument can contain a list of theorem attributes, which we will explain in detail in Section 3.8. Below we just use one such attribute, *simp_add*, for adding the theorem to the simpset:

```
local_setup {*
  Local_Theory.note ((@{binding "my_thm_simp"},
       [Attrib.internal (K Simplifier.simp_add)]), [my_thm]) #> snd *}
```

Note that we have to use another name under which the theorem is stored, since Isabelle does not allow us to call `note` twice with the same name. The attribute needs to be wrapped inside the function `internal` from the structure `Attrib`. If we use the function `get_thm_names_from_ss` from the previous chapter, we can check whether the theorem has actually been added.

```
let
  fun pred s = match_string "my_thm_simp" s
in
  exists pred (get_thm_names_from_ss @{context})
end
> true
```

The main point of storing the theorems `my_thm` and `my_thm_simp` is that they can now also be referenced with the **thm**-command on the user-level of Isabelle

```
thm my_thm my_thm_simp
> ⟦⋀x. P x ⟹ Q x; P t⟧ ⟹ Q t
> ⟦⋀x. P x ⟹ Q x; P t⟧ ⟹ Q t
```

or with the `@{thm ...}`-antiquotation on the ML-level. Otherwise the user has no access to these theorems.

Recall that Isabelle does not let you call `note` twice with the same theorem name. In effect, once a theorem is stored under a name, this association is fixed. While this is a "safety-net" to make sure a theorem name refers to a particular theorem or collection of theorems, it is also a bit too restrictive in cases where a theorem name should refer to a dynamically expanding list of theorems (like a simpset). Therefore Isabelle also implements a mechanism where a theorem name can refer to a custom theorem list. For this you can use the function `add_thms_dynamic`. To see how it works let us assume we defined our own theorem list `MyThmList`.

```
structure MyThmList = Generic_Data
  (type T = thm list
   val empty = []
   val extend = I
   val merge = merge Thm.eq_thm_prop)

fun update thm = Context.theory_map (MyThmList.map (Thm.add_thm thm))
```

The function `update` allows us to update the theorem list, for example by adding the theorem `TrueI`.

```
setup {* update @{thm TrueI} *}
```

We can now install the theorem list so that it is visible to the user and can be refered to by a theorem name. For this need to call *add_thms_dynamic*

```
setup {*
  Global_Theory.add_thms_dynamic (@{binding "mythmlist"}, MyThmList.get)
*}
```

with a name and a function that accesses the theorem list. Now if the user issues the command

```
thm mythmlist
> True
```

the current content of the theorem list is displayed. If more theorems are stored in the list, say

```
setup {* update @{thm FalseE} *}
```

then the same command produces

```
thm mythmlist
> False ⟹ ?P
> True
```

Note that if we add the theorem *FalseE* again to the list

```
setup {* update @{thm FalseE} *}
```

we still obtain the same list. The reason is that we used the function *add_thm* in our update function. This is a dedicated function which tests whether the theorem is already in the list. This test is done according to alpha-equivalence of the proposition of the theorem. The corresponding testing function is *eq_thm_prop*. Suppose you proved the following three theorems.

```
lemma
  shows thm1: "∀x. P x"
  and   thm2: "∀y. P y"
  and   thm3: "∀y. Q y" sorry
```

Testing them for alpha equality produces:

```
(Thm.eq_thm_prop (@{thm thm1}, @{thm thm2}),
 Thm.eq_thm_prop (@{thm thm2}, @{thm thm3}))
> (true, false)
```

Many functions destruct theorems into *cterm*s. For example the functions *lhs_of* and *rhs_of* return the left and right-hand side, respectively, of a meta-equality.

```
let
  val eq = @{thm True_def}
in
  (Thm.lhs_of eq, Thm.rhs_of eq)
  |> pairself (Pretty.string_of o (pretty_cterm @{context}))
end
> (True, (λx. x) = (λx. x))
```

Other function produce terms that can be pattern-matched. Suppose the following two theorems.

lemma
  shows *foo_test1: "A ⟹ B ⟹ C"*
  and *foo_test2: "A ⟶ B ⟶ C"* **sorry**

We can destruct them into premises and conclusions as follows.

```
let
  val ctxt = @{context}
  fun prems_and_concl thm =
     [[Pretty.str "Premises:", pretty_terms ctxt (Thm.prems_of thm)],
      [Pretty.str "Conclusion:", pretty_term ctxt (Thm.concl_of thm)]]
     |> map Pretty.block
     |> Pretty.chunks
     |> pwriteln
in
  prems_and_concl @{thm foo_test1};
  prems_and_concl @{thm foo_test2}
end
> Premises: ?A, ?B
> Conclusion: ?C
> Premises:
> Conclusion: ?A ⟶ ?B ⟶ ?C
```

Note that in the second case, there is no premise. The reason is that $\Longrightarrow$ separates premises and conclusion, while $\longrightarrow$ is the object implication from HOL, which just constructs a formula.

> ***Read More***
> *The basic functions for theorems are defined in* Pure/thm.ML, Pure/more_thm.ML *and* Pure/drule.ML.

Although we will explain the simplifier in more detail as tactic in Section 6.4, the simplifier can be used to work directly over theorems, for example to unfold definitions. To show this, we build the theorem *True* $\equiv$ *True* (Line 1) and then unfold the constant *True* according to its definition (Line 2).

```
1 Thm.reflexive @{cterm "True"}
2   |> Simplifier.rewrite_rule @{context} [@{thm True_def}]
3   |> pretty_thm @{context}
4   |> pwriteln
5 > (λx. x) = (λx. x) ≡ (λx. x) = (λx. x)
```

Often it is necessary to transform theorems to and from the object logic, that is replacing all $\longrightarrow$ and $\forall$ by $\Longrightarrow$ and $\bigwedge$, or the other way around. A reason for such a transformation might be stating a definition. The reason is that definitions can only be stated using object logic connectives, while theorems using the connectives from the meta logic are more convenient for reasoning. Therefore there are some build in functions which help with these transformations. The function `rulify` replaces all object connectives by equivalents in the meta logic. For example

```
Object_Logic.rulify @{context} @{thm foo_test2}
> ⟦?A; ?B⟧ ⟹ ?C
```

The transformation in the other direction can be achieved with function `atomize` and the following code.

```
let
  val thm = @{thm foo_test1}
  val meta_eq = Object_Logic.atomize @{context} (cprop_of thm)
in
  Raw_Simplifier.rewrite_rule @{context} [meta_eq] thm
end
> ?A ⟶ ?B ⟶ ?C
```

In this code the function `atomize` produces a meta-equation between the given theorem and the theorem transformed into the object logic. The result is the theorem with object logic connectives. However, in order to completely transform a theorem involving meta variables, such as `list.induct`, which is of the form

```
⟦?P []; ⋀x1 x2. ?P x2 ⟹ ?P (x1 # x2)⟧ ⟹ ?P ?list
```

we have to first abstract over the meta variables `?P` and `?list`. For this we can use the function `forall_intr_vars`. This allows us to implement the following function for atomizing a theorem.

```
fun atomize_thm ctxt thm =
let
  val thm' = forall_intr_vars thm
  val thm'' = Object_Logic.atomize ctxt (cprop_of thm')
in
  Raw_Simplifier.rewrite_rule ctxt [thm''] thm'
end
```

This function produces for the theorem `list.induct`

```
atomize_thm @{context} @{thm list.induct}
> ∀P list. P [] ⟶ (∀a list. P list ⟶ P (a # list)) ⟶ P list
```

Theorems can also be produced from terms by giving an explicit proof. One way to achieve this is by using the function *prove* in the structure *Goal*. For example below we use this function to prove the term $P \Longrightarrow P$.

```
let
  val trm = @{term "P ⟹ P::bool"}
  val tac = K (atac 1)
in
  Goal.prove @{context} ["P"] [] trm tac
end
> ?P ⟹ ?P
```

This function takes first a context and second a list of strings. This list specifies which variables should be turned into schematic variables once the term is proved (in this case only *"P"*). The fourth argument is the term to be proved. The fifth is a corresponding proof given in form of a tactic (we explain tactics in Chapter 6). In the code above, the tactic proves the theorem by assumption.

There is also the possibility of proving multiple goals at the same time using the function *prove_multi*. For this let us define the following three helper functions.

```
fun rep_goals i = replicate i @{prop "f x = f x"}
fun rep_tacs i = replicate i (rtac @{thm refl})

fun multi_test ctxt i =
  Goal.prove_multi ctxt ["f", "x"] [] (rep_goals i)
    (K ((Goal.conjunction_tac THEN' RANGE (rep_tacs i)) 1))
```

With them we can now produce three theorem instances of the proposition.

```
multi_test @{context} 3
> ["?f ?x = ?f ?x", "?f ?x = ?f ?x", "?f ?x = ?f ?x"]
```

However you should be careful with *prove_multi* and very large goals. If you increase the counter in the code above to *3000*, you will notice that takes approximately ten(!) times longer than using *map* and *prove*.

```
let
  fun test_prove ctxt thm =
    Goal.prove ctxt ["P", "x"] [] thm (K (rtac @{thm refl} 1))
in
  map (test_prove @{context}) (rep_goals 3000)
end
```

While the LCF-approach of going through interfaces ensures soundness in Isabelle, there is the function *make_thm* in the structure *Skip_Proof* that allows us to turn any proposition into a theorem. Potentially making the system unsound. This is sometimes useful for developing purposes, or when explicit proof construction should be omitted due to performace reasons. An example of this function is as follows:

```
Skip_Proof.make_thm @{theory} @{prop "True = False"}
> True = False
```

> ***Read More***
> *Functions that setup goal states and prove theorems are implemented in* `Pure/goal.ML`. *A function and a tactic that allow one to skip proofs of theorems are implemented in* `Pure/skip_proof.ML`.

Theorems also contain auxiliary data, such as the name of the theorem, its kind, the names for cases and so on. This data is stored in a string-string list and can be retrieved with the function `get_tags`. Assume you prove the following lemma.

```
lemma foo_data:
  shows "P ⟹ P ⟹ P" by assumption
```

The auxiliary data of this lemma can be retrieved using the function `get_tags`. So far the the auxiliary data of this lemma is

```
Thm.get_tags @{thm foo_data}
> [("name", "General.foo_data"), ("kind", "lemma")]
```

consisting of a name and a kind. When we store lemmas in the theorem database, we might want to explicitly extend this data by attaching case names to the two premises of the lemma. This can be done with the function `name` from the structure `Rule_Cases`.

```
local_setup {*
  Local_Theory.note ((@{binding "foo_data'"}, []),
    [(Rule_Cases.name ["foo_case_one", "foo_case_two"]
       @{thm foo_data})]) #> snd *}
```

The data of the theorem `foo_data'` is then as follows:

```
Thm.get_tags @{thm foo_data'}
> [("name", "General.foo_data'"), ("kind", "lemma"),
>  ("case_names", "foo_case_one;foo_case_two")]
```

You can observe the case names of this lemma on the user level when using the proof methods `cases` and `induct`. In the proof below

```
lemma
shows "Q ⟹ Q ⟹ Q"
proof (cases rule: foo_data')
print_cases
> cases:
>   foo_case_one:
>     let "?case" = "?P"
>   foo_case_two:
>     let "?case" = "?P"
```

we can proceed by analysing the cases *foo_case_one* and *foo_case_two*. While if the theorem has no names, then the cases have standard names *1*, *2* and so on. This can be seen in the proof below.

```
lemma
shows "Q ⟹ Q ⟹ Q"
proof (cases rule: foo_data)
print_cases
> cases:
>   1:
>     let "?case" = "?P"
>   2:
>     let "?case" = "?P"
```

Sometimes one wants to know the theorems that are involved in proving a theorem, especially when the proof is by *auto*. These theorems can be obtained by introspecting the proved theorem. To introspect a theorem, let us define the following three functions that analyse the *proof_body* data-structure from the structure *Proofterm*.

```
fun pthms_of (PBody {thms, ...}) = map #2 thms
val get_names = map #1 o pthms_of
val get_pbodies = map (Future.join o #3) o pthms_of
```

To see what their purpose is, consider the following three short proofs.

```
lemma my_conjIa:
shows "A ∧ B ⟹ A ∧ B"
apply(rule conjI)
apply(drule conjunct1)
apply(assumption)
apply(drule conjunct2)
apply(assumption)
done

lemma my_conjIb:
shows "A ∧ B ⟹ A ∧ B"
apply(assumption)
done

lemma my_conjIc:
shows "A ∧ B ⟹ A ∧ B"
apply(auto)
done
```

While the information about which theorems are used is obvious in the first two proofs, it is not obvious in the third, because of the *auto*-step. Fortunately, "behind" every proved theorem is a proof-tree that records all theorems that are employed for establishing this theorem. We can traverse this proof-tree extracting this information. Let us first extract the name of the established theorem. This can be done with

```
@{thm my_conjIa}
  |> Thm.proof_body_of
  |> get_names
> ["Essential.my_conjIa"]
```

whereby *Essential* refers to the theory name in which we established the theorem *my_conjIa*. The function *proof_body_of* returns a part of the data that is stored in a theorem. Notice that the first proof above references three theorems, namely *conjI*, *conjunct1* and *conjunct2*. We can obtain them by descending into the first level of the proof-tree, as follows.

```
@{thm my_conjIa}
  |> Thm.proof_body_of
  |> get_pbodies
  |> map get_names
  |> List.concat
> ["HOL.conjunct2", "HOL.conjunct1", "HOL.conjI", "Pure.protectD",
>   "Pure.protectI"]
```

The theorems *Pure.protectD* and *Pure.protectI* that are internal theorems are always part of a proof in Isabelle. Note also that applications of *assumption* do not count as a separate theorem, as you can see in the following code.

```
@{thm my_conjIb}
  |> Thm.proof_body_of
  |> get_pbodies
  |> map get_names
  |> List.concat
> ["Pure.protectD", "Pure.protectI"]
```

Interestingly, but not surprisingly, the proof of *my_conjIc* procceeds quite differently from *my_conjIa* and *my_conjIb*, as can be seen by the theorems that are returned for *my_conjIc*.

```
@{thm my_conjIc}
  |> Thm.proof_body_of
  |> get_pbodies
  |> map get_names
  |> List.concat
> ["HOL.Eq_TrueI", "HOL.simp_thms_25", "HOL.eq_reflection",
>   "HOL.conjunct2", "HOL.conjunct1", "HOL.TrueI", "Pure.protectD",
>   "Pure.protectI"]
```

Of course we can also descend into the second level of the tree “behind” *my_conjIa* say, which means we obtain the theorems that are used in order to prove *conjunct1*, *conjunct2* and *conjI*.

```
@{thm my_conjIa}
  |> Thm.proof_body_of
  |> get_pbodies
  |> map get_pbodies
  |> (map o map) get_names
  |> List.concat
  |> List.concat
  |> duplicates (op=)
> ["HOL.spec", "HOL.and_def", "HOL.mp", "HOL.impI", "Pure.protectD",
>   "Pure.protectI"]
```

Exercise 3.7.1: *Have a look at the theorems that are used when a lemma is "proved" by* **sorry***?*

> ***Read More***
> *The data-structure* `proof_body` *is implemented in the file* `Pure/proofterm.ML`*. The implementation of theorems and related functions are in* `Pure/thm.ML`*.*

One great feature of Isabelle is its document preparation system, where proved theorems can be quoted in documents referencing directly their formalisation. This helps tremendously to minimise cut-and-paste errors. However, sometimes the verbatim quoting is not what is wanted or what can be shown to readers. For such situations Isabelle allows the installation of *theorem styles*. These are, roughly speaking, functions from terms to terms. The input term stands for the theorem to be presented; the output can be constructed to ones wishes. Let us, for example, assume we want to quote theorems without leading $\forall$-quantifiers. For this we can implement the following function that strips off $\forall$s.

```
1 fun strip_allq (Const (@{const_name "All"}, _) $ Abs body) =
2       Term.dest_abs body |> snd |> strip_allq
3   | strip_allq (Const (@{const_name "Trueprop"}, _) $ t) =
4       strip_allq t
5   | strip_allq t = t
```

We use in Line 2 the function `dest_abs` for deconstructing abstractions, since this function deals correctly with potential name clashes. This function produces a pair consisting of the variable and the body of the abstraction. We are only interested in the body, which we feed into the recursive call. In Line 3 and 4, we also have to explicitly strip of the outermost `Trueprop`-coercion. Now we can install this function as the theorem style named `my_strip_allq`.

```
setup{*
  Term_Style.setup @{binding "my_strip_allq"} (Scan.succeed (K strip_allq))
*}
```

We can test this theorem style with the following theorem

```
theorem style_test:
shows "∀x y z. (x, x) = (y, z)" sorry
```

Now printing out in a document the theorem `style_test` normally using `@{thm ...}` produces

```
@{thm style_test}
> ∀x y z. (x, x) = (y, z)
```

as expected. But with the theorem style *@{thm (my_strip_allq) ...}* we obtain

```
@{thm (my_strip_allq) style_test}
> (x, x) = (y, z)
```

without the leading quantifiers. We can improve this theorem style by explicitly giving a list of strings that should be used for the replacement of the variables. For this we implement the function which takes a list of strings and uses them as name in the outermost abstractions.

```
fun rename_allq [] t = t
  | rename_allq (x::xs) (Const (@{const_name "All"}, U) $ Abs (_, T, t)) =
      Const (@{const_name "All"}, U) $ Abs (x, T, rename_allq xs t)
  | rename_allq xs (Const (@{const_name "Trueprop"}, U) $ t) =
      rename_allq xs t
  | rename_allq _ t = t
```

We can now install a the modified theorem style as follows

```
setup {* let
  val parser = Scan.repeat Args.name
  fun action xs = K (rename_allq xs #> strip_allq)
in
  Term_Style.setup @{binding "my_strip_allq2"} (parser >> action)
end *}
```

The parser reads a list of names. In the function *action* we first call *rename_allq* with the parsed list, then we call *strip_allq* on the resulting term. We can now suggest, for example, two variables for stripping off the first two ∀-quantifiers.

```
@{thm (my_strip_allq2 x' x'') style_test}
> (x', x') = (x'', z)
```

Such styles allow one to print out theorems in documents formatted to ones heart content. The styles can also be used in the document antiquotations *@{prop ...}*, *@{term_type ...}* and *@{typeof ...}*.

Next we explain theorem attributes, which is another mechanism for dealing with theorems.

> ***Read More***
> *Theorem styles are implemented in* Pure/Thy/term_style.ML.

3.8 Theorem Attributes

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

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

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

The theorem attributes fall roughly into two categories: the first category manipulates theorems (for example *[symmetric]* and *[THEN ...]*), and the second stores theorems somewhere as data (for example *[simp]*, which adds theorems to the current simpset).

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

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

where the function *rule_attribute* expects a function taking a context (which we ignore in the code above) and a theorem (*thm*), and returns another theorem (namely *thm* resolved with the theorem *sym*: $s = t \Longrightarrow t = s$; the function *RS* is explained in Section 6.2). The function *rule_attribute* then returns an attribute.

Before we can use the attribute, we need to set it up. This can be done using the Isabelle command **attribute_setup** as follows:

```
attribute_setup my_sym =
  {* Scan.succeed my_symmetric *} "applying the sym rule"
```

Inside the *{* ... *}*, we have to specify a parser for the theorem attribute. Since the attribute does not expect any further arguments (unlike *[THEN ...]*, for instance), we use the parser *Scan.succeed*. An example for the attribute *[my_sym]* is the proof

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

which stores the theorem *Suc (Suc 0) = 2* under the name *test*. You can see this, if you query the lemma:

thm *test*
```
> Suc (Suc 0) = 2
```

We can also use the attribute when referring to this theorem:

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

An alternative for setting up an attribute is the function *setup*. So instead of using **attribute_setup**, you can also set up the attribute as follows:

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

This gives a function from *theory -> theory*, which can be used for example with **setup** or with *Context.>> o Context.map_theory*.[4]

As an example of a slightly more complicated theorem attribute, we implement our own version of *[THEN ...]*. This attribute will take a list of theorems as argument and resolve the theorem with this list (one theorem after another). The code for this attribute is

```
fun MY_THEN thms =
let
  fun RS_rev thm1 thm2 = thm2 RS thm1
in
  Thm.rule_attribute (fn _ => fn thm => fold RS_rev thms thm)
end
```

where for convenience we define the reverse and curried version of *RS*. The setup of this theorem attribute uses the parser *thms*, which parses a list of theorems.

```
attribute_setup MY_THEN = {* Attrib.thms >> MY_THEN *}
  "resolving the list of theorems with the theorem"
```

You can, for example, use this theorem attribute to turn an equation into a meta-equation:

thm *test[MY_THEN eq_reflection]*
> Suc (Suc 0) ≡ 2

If you need the symmetric version as a meta-equation, you can write

thm *test[MY_THEN sym eq_reflection]*
> 2 ≡ Suc (Suc 0)

It is also possible to combine different theorem attributes, as in:

thm *test[my_sym, MY_THEN eq_reflection]*
> 2 ≡ Suc (Suc 0)

[4] **FIXME: explain what happens here.**

However, here also a weakness of the concept of theorem attributes shows through: since theorem attributes can be arbitrary functions, they do not commute in general. If you try

```
thm test[MY_THEN eq_reflection, my_sym]
> exception THM 1 raised: RSN: no unifiers
```

you get an exception indicating that the theorem `sym` does not resolve with meta-equations.

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

```
structure MyThms = Named_Thms
  (val name = @{binding "attr_thms"}
   val description = "Theorems for an Attribute")
```

We are going to modify this list by adding and deleting theorems. For this we use the two functions `MyThms.add_thm` and `MyThms.del_thm`. You can turn them into attributes with the code

```
val my_add = Thm.declaration_attribute MyThms.add_thm
val my_del = Thm.declaration_attribute MyThms.del_thm
```

and set up the attributes as follows

```
attribute_setup my_thms = {* Attrib.add_del my_add my_del *}
  "maintaining a list of my_thms"
```

The parser `add_del` is a predefined parser for adding and deleting lemmas. Now if you prove the next lemma and attach to it the attribute `[my_thms]`

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

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

```
MyThms.get @{context}
> ["True"]
```

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

```
declare test[my_thms] trueI_2[my_thms add]
```

With this attribute, the `add` operation is the default and does not need to be explicitly given. These three declarations will cause the theorem list to be updated as:

```
MyThms.get @{context}
> ["True", "Suc (Suc 0) = 2"]
```

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

declare *test[my_thms del]*

After this, the theorem list is again:

```
MyThms.get @{context}
> ["True"]
```

We used in this example two functions declared as *declaration_attribute*, but there can be any number of them. We just have to change the parser for reading the arguments accordingly.
[5] [6]

> ***Read More***
> *FIXME: Pure/more_thm.ML; parsers for attributes is in Pure/Isar/attrib.ML...also explained in the chapter about parsing.*

3.9 Pretty-Printing

So far we printed out only plain strings without any formatting except for occasional explicit line breaks using *"\n"*. This is sufficient for "quick-and-dirty" printouts. For something more sophisticated, Isabelle includes an infrastructure for properly formatting text. Most of its functions do not operate on *string*s, but on instances of the pretty type:

```
Pretty.T
```

The function *str* transforms a (plain) string into such a pretty type. For example

```
Pretty.str "test"
> String ("test", 4)
```

where the result indicates that we transformed a string with length 4. Once you have a pretty type, you can, for example, control where linebreaks may occur in case the text wraps over a line, or with how much indentation a text should be printed. However, if you want to actually output the formatted text, you have to transform the pretty type back into a *string*. This can be done with the function *string_of*. In what follows we will use the following wrapper function for printing a pretty type:

[5] **FIXME What are:** *theory_attributes*, *proof_attributes***?**
[6] **FIXME readmore**

```
fun pprint prt = tracing (Pretty.string_of prt)
```

The point of the pretty-printing infrastructure is to give hints about how to layout text and let Isabelle do the actual layout. Let us first explain how you can insert places where a line break can occur. For this assume the following function that replicates a string n times:

```
fun rep n str = implode (replicate n str)
```

and suppose we want to print out the string:

```
val test_str = rep 8 "fooooooooooooooobaaaaaaaaaaaar "
```

We deliberately chose a large string so that it spans over more than one line. If we print out the string using the usual "quick-and-dirty" method, then we obtain the ugly output:

```
tracing test_str
> fooooooooooooooobaaaaaaaaaaaar fooooooooooooooobaaaaaaaaaaaar fooooooooo
> ooooobaaaaaaaaaaaar fooooooooooooooobaaaaaaaaaaaar fooooooooooooooobaaaaa
> aaaaaaar fooooooooooooooobaaaaaaaaaaaar fooooooooooooooobaaaaaaaaaaaar fo
> ooooooooooooobaaaaaaaaaaaar
```

We obtain the same if we just use the function *pprint*.

```
pprint (Pretty.str test_str)
> fooooooooooooooobaaaaaaaaaaaar fooooooooooooooobaaaaaaaaaaaar fooooooooo
> ooooobaaaaaaaaaaaar fooooooooooooooobaaaaaaaaaaaar fooooooooooooooobaaaaa
> aaaaaaar fooooooooooooooobaaaaaaaaaaaar fooooooooooooooobaaaaaaaaaaaar fo
> ooooooooooooobaaaaaaaaaaaar
```

However by using pretty types you have the ability to indicate possible linebreaks for example at each whitespace. You can achieve this with the function *breaks*, which expects a list of pretty types and inserts a possible line break in between every two elements in this list. To print this list of pretty types as a single string, we concatenate them with the function *blk* as follows:

```
let
  val ptrs = map Pretty.str (space_explode " " test_str)
in
  pprint (Pretty.blk (0, Pretty.breaks ptrs))
end
> fooooooooooooooobaaaaaaaaaaaar fooooooooooooooobaaaaaaaaaaaar
> fooooooooooooooobaaaaaaaaaaaar fooooooooooooooobaaaaaaaaaaaar
> fooooooooooooooobaaaaaaaaaaaar fooooooooooooooobaaaaaaaaaaaar
> fooooooooooooooobaaaaaaaaaaaar fooooooooooooooobaaaaaaaaaaaar
```

Here the layout of *test_str* is much more pleasing to the eye. The *0* in *blk* stands for no hanging indentation of the printed string. You can increase the indentation and obtain

```
let
  val ptrs = map Pretty.str (space_explode " " test_str)
in
  pprint (Pretty.blk (3, Pretty.breaks ptrs))
end
> fooooooooooooooobaaaaaaaaaaaar fooooooooooooooobaaaaaaaaaaaar
>    fooooooooooooooobaaaaaaaaaaaar fooooooooooooooobaaaaaaaaaaaar
>    fooooooooooooooobaaaaaaaaaaaar fooooooooooooooobaaaaaaaaaaaar
>    fooooooooooooooobaaaaaaaaaaaar fooooooooooooooobaaaaaaaaaaaar
```

where starting from the second line the indent is 3. If you want that every line starts with the same indent, you can use the function *indent* as follows:

```
let
  val ptrs = map Pretty.str (space_explode " " test_str)
in
  pprint (Pretty.indent 10 (Pretty.blk (0, Pretty.breaks ptrs)))
end
>           fooooooooooooooobaaaaaaaaaaaar fooooooooooooooobaaaaaaaaaaaar
>           fooooooooooooooobaaaaaaaaaaaar fooooooooooooooobaaaaaaaaaaaar
>           fooooooooooooooobaaaaaaaaaaaar fooooooooooooooobaaaaaaaaaaaar
>           fooooooooooooooobaaaaaaaaaaaar fooooooooooooooobaaaaaaaaaaaar
```

If you want to print out a list of items separated by commas and have the linebreaks handled properly, you can use the function *commas*. For example

```
let
  val ptrs = map (Pretty.str o string_of_int) (99998 upto 100020)
in
  pprint (Pretty.blk (0, Pretty.commas ptrs))
end
> 99998, 99999, 100000, 100001, 100002, 100003, 100004, 100005, 100006,
> 100007, 100008, 100009, 100010, 100011, 100012, 100013, 100014, 100015,
> 100016, 100017, 100018, 100019, 100020
```

where *upto* generates a list of integers. You can print out this list as a "set", that means enclosed inside *"{"* and *"}"*, and separated by commas using the function *enum*. For example

```
let
  val ptrs = map (Pretty.str o string_of_int) (99998 upto 100020)
in
  pprint (Pretty.enum "," "{" "}" ptrs)
end
```

```
> {99998, 99999, 100000, 100001, 100002, 100003, 100004, 100005, 100006,
>   100007, 100008, 100009, 100010, 100011, 100012, 100013, 100014, 100015,
>   100016, 100017, 100018, 100019, 100020}
```

As can be seen, this function prints out the "set" so that starting from the second, each new line has an indentation of 2.

If you print out something that goes beyond the capabilities of the standard functions, you can do relatively easily the formatting yourself. Assume you want to print out a list of items where like in "English" the last two items are separated by `"and"`. For this you can write the function

```
1 fun and_list [] = []
2   | and_list [x] = [x]
3   | and_list xs =
4       let
5         val (front, last) = split_last xs
6       in
7         (Pretty.commas front) @
8         [Pretty.brk 1, Pretty.str "and", Pretty.brk 1, last]
9       end
```

where Line 7 prints the beginning of the list and Line 8 the last item. We have to use `Pretty.brk 1` in order to insert a space (of length 1) before the `"and"`. This space is also a place where a line break can occur. We do the same after the `"and"`. This gives you for example

```
let
  val ptrs1 = map (Pretty.str o string_of_int) (1 upto 22)
  val ptrs2 = map (Pretty.str o string_of_int) (10 upto 28)
in
  pprint (Pretty.blk (0, and_list ptrs1));
  pprint (Pretty.blk (0, and_list ptrs2))
end
> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21
> and 22
> 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27 and
> 28
```

Like `blk`, the function `chunks` prints out a list of items, but automatically inserts forced breaks between each item. Compare

```
let
  val a_and_b = [Pretty.str "a", Pretty.str "b"]
in
  pprint (Pretty.blk (0, a_and_b));
  pprint (Pretty.chunks a_and_b)
end
> ab
> a
> b
```

The function *big_list* helps with printing long lists. It is used for example in the command **print_theorems**. An example is as follows.

```
let
  val pstrs = map (Pretty.str o string_of_int) (4 upto 10)
in
  pprint (Pretty.big_list "header" pstrs)
end
> header
>   4
>   5
>   6
>   7
>   8
>   9
>   10
```

The point of the pretty-printing functions is to conveninetly obtain a lay-out of terms and types that is pleasing to the eye. If we print out the the terms produced by the the function *de_bruijn* from exercise 3.2.5 we obtain the following:

```
pprint (Syntax.pretty_term  @{context} (de_bruijn 4))
> (P 3 = P 4 ⟶ P 4 ∧ P 3 ∧ P 2 ∧ P 1) ∧
> (P 2 = P 3 ⟶ P 4 ∧ P 3 ∧ P 2 ∧ P 1) ∧
> (P 1 = P 2 ⟶ P 4 ∧ P 3 ∧ P 2 ∧ P 1) ∧
> (P 1 = P 4 ⟶ P 4 ∧ P 3 ∧ P 2 ∧ P 1) ⟶
> P 4 ∧ P 3 ∧ P 2 ∧ P 1
```

We use the function *pretty_term* for pretty-printing terms. Next we like to pretty-print a term and its type. For this we use the function *tell_type*.

```
1  fun tell_type ctxt trm =
2  let
3    fun pstr s = Pretty.breaks (map Pretty.str (space_explode " " s))
4    val ptrm = Pretty.quote (Syntax.pretty_term ctxt trm)
5    val pty  = Pretty.quote (Syntax.pretty_typ ctxt (fastype_of trm))
6  in
7    pprint (Pretty.blk (0,
8      (pstr "The term " @ [ptrm] @ pstr " has type "
9        @ [pty, Pretty.str "."])))
10 end
```

In Line 3 we define a function that inserts possible linebreaks in places where a space is. In Lines 4 and 5 we pretty-print the term and its type using the functions *pretty_term* and *pretty_typ*. We also use the function *quote* in order to enclose the term and type inside quotation marks. In Line 9 we add a period right after the type without the possibility of a line break, because we do not want that a line break occurs there. Now you can write

```
tell_type @{context} @{term "min (Suc 0)"}
> The term "min (Suc 0)" has type "nat ⇒ nat".
```

To see the proper line breaking, you can try out the function on a bigger term and type. For example:

```
tell_type @{context} @{term "op = (op = (op = (op = (op = op =))))"}
> The term "op = (op = (op = (op = (op = op =))))" has type
> "(((((’a ⇒ ’a ⇒ bool) ⇒ bool) ⇒ bool) ⇒ bool) ⇒ bool) ⇒ bool".
```

> ***Read More***
> *The general infrastructure for pretty-printing is implemented in the file* `Pure/General/pretty.ML`*. The file* `Pure/Syntax/syntax.ML` *contains pretty-printing functions for terms, types, theorems and so on.*
>
> `Pure/PIDE/markup.ML`

3.10 Summary

> ***Coding Conventions / Rules of Thumb***
>
> - *Start with a proper context and then pass it around through all your functions.*

Chapter 4

Advanced Isabelle

All things are difficult before they are easy.

proverb

Programming is not just an act of telling a computer what to do: it is also an act of telling other programmers what you wished the computer to do. Both are important, and the latter deserves care.

—Andrew Morton, Linux Kernel mailinglist, 13 March 2012

While terms, types and theorems are the most basic data structures in Isabelle, there are a number of layers built on top of them. Most of these layers are concerned with storing and manipulating data. Handling them properly is an essential skill for programming on the ML-level of Isabelle. The most basic layer are theories. They contain global data and can be seen as the "long-term memory" of Isabelle. There is usually only one theory active at each moment. Proof contexts and local theories, on the other hand, store local data for a task at hand. They act like the "short-term memory" and there can be many of them that are active in parallel.

4.1 Theories

Theories, as said above, are the most basic layer of abstraction in Isabelle. They record information about definitions, syntax declarations, axioms, theorems and much more. For example, if a definition is made, it must be stored in a theory in order to be usable later on. Similar with proofs: once a proof is finished, the proved theorem needs to be stored in the theorem database of the theory in order to be usable. All relevant data of a theory can be queried with the Isabelle command **print_theory**.

print_theory

```
> names: Pure Code_Generator HOL ...
> classes: Inf < type ...
> default sort: type
> syntactic types: #prop ...
> logical types: 'a × 'b ...
```

```
> type arities: * :: (random, random) random ...
> logical constants: == :: 'a ⇒ 'a ⇒ prop ...
> abbreviations: ...
> axioms: ...
> oracles: ...
> definitions: ...
> theorems: ...
```

Functions acting on theories often end with the suffix *_global*, for example the function *read_term_global* in the structure *Syntax*. The reason is to set them syntactically apart from functions acting on contexts or local theories, which will be discussed in the next sections. There is a tendency amongst Isabelle developers to prefer "non-global" operations, because they have some advantages, as we will also discuss later. However, some basic understanding of theories is still necessary for effective Isabelle programming.

An important Isabelle command with theories is **setup**. In the previous chapters we used it already to make a theorem attribute known to Isabelle and to register a theorem under a name. What happens behind the scenes is that **setup** expects a function of type *theory -> theory*: the input theory is the current theory and the output the theory where the attribute has been registered or the theorem has been stored. This is a fundamental principle in Isabelle. A similar situation arises with declaring a constant, which can be done on the ML-level with function *declare_const* from the structure *Sign*. To see how **setup** works, consider the following code:

```
let
  val thy = @{theory}
  val bar_const = ((@{binding "BAR"}, @{typ "nat"}), NoSyn)
in
  Sign.declare_const @{context} bar_const thy
end
```

If you simply run this code[1] with the intention of declaring a constant *BAR* having type *nat*, then indeed you obtain a theory as result. But if you query the constant on the Isabelle level using the command **term**

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

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

```
setup {* fn thy =>
let
  val bar_const = ((@{binding "BAR"}, @{typ "nat"}), NoSyn)
```

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

```
  val (_, thy') = Sign.declare_const @{context} bar_const thy
in
  thy'
end *}
```

where the declaration is actually applied to the current theory and

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

now returns a (black) constant with the type *nat*, as expected.

In a sense, **setup** can be seen as a transaction that takes the current theory *thy*, applies an operation, and produces a new current theory *thy'*. This means that we have to be careful to apply operations always to the most current theory, not to a *stale* one. Consider again the function inside the **setup**-command:

```
setup {* fn thy =>
let
  val bar_const = ((@{binding "BAR"}, @{typ "nat"}), NoSyn)
  val (_, thy') = Sign.declare_const @{context} bar_const thy
in
  thy
end *}
> ERROR: "Stale theory encountered"
```

This time we erroneously return the original theory *thy*, instead of the modified one *thy'*. Such buggy code will always result into a runtime error message about stale theories.

> ***Read More***
> *Most of the functions about theories are implemented in* Pure/theory.ML *and* Pure/global_theory.ML.

4.2 Contexts

Contexts are arguably more important than theories, even though they only contain "short-term memory data". The reason is that a vast number of functions in Isabelle depend in one way or another on contexts. Even such mundane operations like printing out a term make essential use of contexts. For this consider the following contrived proof-snippet whose purpose is to fix two variables:

```
lemma "True"
proof -
  ML_prf {* Variable.dest_fixes @{context} *}
  fix x y
  ML_prf {* Variable.dest_fixes @{context} *}
```

The interesting point is that we injected ML-code before and after the variables are fixed. For this remember that ML-code inside a proof needs to be enclosed inside

ML_prf *{* ... *}*, not **ML** *{* ... *}*. The function *dest_fixes* from the structure *Variable* takes a context and returns all its currently fixed variable (names). That means a context has a dataslot containing information about fixed variables. The ML-antiquotation *@{context}* points to the context that is active at that point of the theory. Consequently, in the first call to *dest_fixes* this dataslot is empty; in the second it is filled with *x* and *y*. What is interesting is that contexts can be stacked. For this consider the following proof fragment:

```
lemma "True"
proof -
  fix x y
  { fix z w
    ML_prf {* Variable.dest_fixes @{context} *}
  }
  ML_prf {* Variable.dest_fixes @{context} *}
```

Here the first time we call *dest_fixes* we have four fixes variables; the second time we get only the fixes variables *x* and *y* as answer, since *z* and *w* are not anymore in the scope of the context. This means the curly-braces act on the Isabelle level as opening and closing statements for a context. The above proof fragment corresponds roughly to the following ML-code

```
val ctxt0 = @{context};
val ([x, y], ctxt1) = Variable.add_fixes ["x", "y"] ctxt0;
val ([z, w], ctxt2) = Variable.add_fixes ["z", "w"] ctxt1
```

where the function *add_fixes* fixes a list of variables specified by strings. Let us come back to the point about printing terms. Consider printing out the term *(x, y, z, w)* using our function *pretty_term*. This function takes a term and a context as argument. Notice how the printing of the term changes according to which context is used.

```
let
  val trm = @{term "(x, y, z, w)"}
in
  pwriteln (Pretty.chunks
    [ pretty_term ctxt0 trm,
      pretty_term ctxt1 trm,
      pretty_term ctxt2 trm ])
end
> (x, y, z, w)
> (x, y, z, w)
> (x, y, z, w)
```

The term we are printing is in all three cases the same—a tuple containing four free variables. As you can see, however, in case of *ctxt0* all variables are highlighted indicating a problem, while in case of *ctxt1 x* and *y* are printed as normal (blue) free variables, but not *z* and *w*. In the last case all variables are printed as expected. The point of this example is that the context contains the information which variables

are fixed, and designates all other free variables as being alien or faulty. Therefore the highlighting. While this seems like a minor detail, the concept of making the context aware of fixed variables is actually quite useful. For example it prevents us from fixing a variable twice

```
@{context}
|> Variable.add_fixes ["x", "x"]
> ERROR: Duplicate fixed variable(s): "x"
```

More importantly it also allows us to easily create fresh names for fixed variables. For this you have to use the function *variant_fixes* from the structure *Variable*.

```
@{context}
|> Variable.variant_fixes ["y", "y", "z"]
> (["y", "ya", "z"], ...)
```

Now a fresh variant for the second occurence of *y* is created avoiding any clash. In this way we can also create fresh free variables that avoid any clashes with fixed variables. In Line 3 below we fix the variable *x* in the context *ctxt1*. Next we want to create two fresh variables of type *nat* as variants of the string *"x"* (Lines 6 and 7).

```
1  let
2    val ctxt0 = @{context}
3    val (_, ctxt1) = Variable.add_fixes ["x"] ctxt0
4    val frees = replicate 2 ("x", @{typ nat})
5  in
6    (Variable.variant_frees ctxt0 [] frees,
7     Variable.variant_frees ctxt1 [] frees)
8  end
9  > ([("x", "nat"), ("xa", "nat")],
10 >  [("xa", "nat"), ("xb", "nat")])
```

As you can see, if we create the fresh variables with the context *ctxt0*, then we obtain *x* and *xa*; but in *ctxt1* we obtain *xa* and *xb* avoiding *x*, which was fixed in this context.

Often one has the problem that given some terms, create fresh variables avoiding any variable occurring in those terms. For this you can use the function *declare_term* from the structure *Variable*.

```
let
  val ctxt1 = Variable.declare_term @{term "(x, xa)"} @{context}
  val frees = replicate 2 ("x", @{typ nat})
in
  Variable.variant_frees ctxt1 [] frees
end
> [("xb", "nat"), ("xc", "nat")]
```

The result is *xb* and *xc* for the names of the fresh variables, since *x* and *xa* occur in the term we declared. Note that *declare_term* does not fix the variables; it just makes them "known" to the context. You can see that if you print out a declared term.

```
let
  val trm = @{term "P x y z"}
  val ctxt1 = Variable.declare_term trm @{context}
in
  pwriteln (pretty_term ctxt1 trm)
end
> P x y z
```

All variables are highligted, indicating that they are not fixed. However, declaring a term is helpful when parsing terms using the function *read_term* from the structure *Syntax*. Consider the following code:

```
let
  val ctxt0 = @{context}
  val ctxt1 = Variable.declare_term @{term "x::nat"} ctxt0
in
  (Syntax.read_term ctxt0 "x",
   Syntax.read_term ctxt1 "x")
end
> (Free ("x", "'a"), Free ("x", "nat"))
```

Parsing the string in the context *ctxt0* results in a free variable with a default polymorphic type, but in case of *ctxt1* we obtain a free variable of type *nat* as previously declared. Which type the parsed term receives depends upon the last declaration that is made, as the next example illustrates.

```
let
  val ctxt1 = Variable.declare_term @{term "x::nat"} @{context}
  val ctxt2 = Variable.declare_term @{term "x::int"} ctxt1
in
  (Syntax.read_term ctxt1 "x",
   Syntax.read_term ctxt2 "x")
end
> (Free ("x", "nat"), Free ("x", "int"))
```

The most useful feature of contexts is that one can export, or transfer, terms and theorems between them. We show this first for terms.

```
1 let
2   val ctxt0 = @{context}
3   val (_, ctxt1) = Variable.add_fixes ["x", "y", "z"] ctxt0
4   val foo_trm = @{term "P x y z"}
5 in
```

```
6   singleton (Variable.export_terms ctxt1 ctxt0) foo_trm
7   |> pretty_term ctxt0
8   |> pwriteln
9 end
> P ?x ?y ?z
```

In Line 3 we fix the variables *x*, *y* and *z* in context *ctxt1*. The function *export_terms* from the structure *Variable* can be used to transfer terms between contexts. Transferring means to turn all (free) variables that are fixed in one context, but not in the other, into schematic variables. In our example, we are transferring the term *P x y z* from context *ctxt1* to *ctxt0*, which means *x*, *y* and *z* become schematic variables (as can be seen by the leading question marks in the result). Note that the variable *P* stays a free variable, since it not fixed in *ctxt1*; it is even highlighed, because *ctxt0* does not know about it. Note also that in Line 6 we had to use the function *singleton*, because the function *export_terms* normally works over lists of terms.

The case of transferring theorems is even more useful. The reason is that the generalisation of fixed variables to schematic variables is not trivial if done manually. For illustration purposes we use in the following code the function *make_thm* from the structure *Skip_Proof*. This function will turn an arbitray term, in our case *P x y z x y z*, into a theorem (disregarding whether it is actually provable).

```
let
  val thy = @{theory}
  val ctxt0 = @{context}
  val (_, ctxt1) = Variable.add_fixes ["P", "x", "y", "z"] ctxt0
  val foo_thm = Skip_Proof.make_thm thy @{prop "P x y z x y z"}
in
  singleton (Proof_Context.export ctxt1 ctxt0) foo_thm
end
> ?P ?x ?y ?z ?x ?y ?z
```

Since we fixed all variables in *ctxt1*, in the exported result all of them are schematic. The great point of contexts is that exporting from one to another is not just restricted to variables, but also works with assumptions. For this we can use the function *export* from the structure *Assumption*. Consider the following code.

```
1 let
2   val ctxt0 = @{context}
3   val ([eq], ctxt1) = Assumption.add_assumes [@{cprop "x ≡ y"}] ctxt0
4   val eq' = Thm.symmetric eq
5 in
6   Assumption.export false ctxt1 ctxt0 eq'
7 end
8 > x ≡ y ⟹ y ≡ x
```

The function *add_assumes* from the structure *Assumption* adds the assumption $x \equiv y$ to the context *ctxt1* (Line 3). This function expects a list of *cterm*s and

returns them as theorems, together with the new context in which they are "assumed". In Line 4 we use the function *symmetric* from the structure *Thm* in order to obtain the symmetric version of the assumed meta-equality. Now exporting the theorem *eq'* from *ctxt1* to *ctxt0* means $y \equiv x$ will be prefixed with the assumed theorem. The boolean flag in *export* indicates whether the assumptions should be marked with the goal marker (see Section 6.1). In normal circumstances this is not necessary and so should be set to *false*. The result of the export is then the theorem $x \equiv y \Longrightarrow y \equiv x$. As can be seen this is an easy way for obtaing simple theorems. We will explain this in more detail in Section 6.8.

The function *export* from the structure *Proof_Context* combines both export functions from *Variable* and *Assumption*. This can be seen in the following example.

```
let
  val ctxt0 = @{context}
  val ((fvs, [eq]), ctxt1) = ctxt0
    |> Variable.add_fixes ["x", "y"]
    ||>> Assumption.add_assumes [@{cprop "x ≡ y"}]
  val eq' = Thm.symmetric eq
in
  Proof_Context.export ctxt1 ctxt0 [eq']
end
> [?x ≡ ?y ⟹ ?y ≡ ?x]
```

4.3 Local Theories and Local Setups (TBD)

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 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.

Context.>> o Context.map_theory Local_Theory.declaration

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

4.4 Morphisms (TBD)

Morphisms are arbitrary transformations over terms, types, theorems and bindings. They can be constructed using the function *morphism*, which expects a record with functions of type

```
binding:  binding -> binding
    typ:  typ -> typ
   term:  term -> term
   fact:  thm list -> thm list
```

The simplest morphism is the *identity*-morphism defined as

```
val identity = Morphism.morphism "" {binding = [], typ = [], term = [], fact
= []}
```

Morphisms can be composed with the function *$>*

```
fun trm_phi (Free (x, T)) = Var ((x, 0), T)
  | trm_phi (Abs (x, T, t)) = Abs (x, T, trm_phi t)
  | trm_phi (t $ s) = (trm_phi t) $ (trm_phi s)
  | trm_phi t = t

val phi = Morphism.term_morphism "foo" trm_phi

Morphism.term phi @{term "P x y"}
```

term_morphism

term, *thm*

> ***Read More***
> *Morphisms are implemented in the file* Pure/morphism.ML.

4.5 Misc (TBD)

ML *{*Datatype.get_info @{theory} "List.list"*}*

FIXME: association lists: *Pure/General/alist.ML*

FIXME: calling the ML-compiler

4.6 What Is In an Isabelle Name? (TBD)

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

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

An example where a qualified name is needed is the function *define*. This function is used below to define the constant *TrueConj* as the conjunction *True* $\wedge$ *True*.

```
local_setup {*
  Local_Theory.define ((@{binding "TrueConj"}, NoSyn),
      ((@{binding "TrueConj_def"}, []), @{term "True ∧ True"})) #> snd *}
```

Now querying the definition you obtain:

thm *TrueConj_def*
> TrueConj ≡ True ∧ True

> ***Read More***
> *The basic operations on bindings are implemented in Pure/General/binding.ML.*

2 3 4

ML *{* Sign.intern_type @{theory} "list" *}*
ML *{* Sign.intern_const @{theory} "prod_fun" *}*

5

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

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

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

Binding.name_of returns the string without markup

Binding.conceal

4.7 Concurrency (TBD)

prove_future future_result

4.8 Parse and Print Translations (TBD)

4.9 Summary

TBD

[2] **FIXME give a better example why bindings are important**
[3] **FIXME give a pointer to local_setup; if not, then explain why *snd* is needed.**
[4] **FIXME: There should probably a separate section on binding, long-names and sign.**
[5] **FIXME: Explain the following better; maybe put in a separate section and link with the comment in the antiquotation section.**

Chapter 5

Parsing

An important principle underlying the success and popularity of Unix is the philosophy of building on the work of others.

Tony Travis in an email about the "LINUX is obsolete" debate

Document your code as if someone else might have to take it over at any moment and know what to do with it. That person might actually be you – how often have you had to revisit your own code and thought to yourself, what was I trying to do here?

Phil Chu in "Seven Habits of Highly Effective Programmers"

Isabelle distinguishes between *outer* and *inner* syntax. Commands, such as **definition**, **inductive** and so on, belong to the outer syntax, whereas terms, types and so on belong to the inner syntax. For parsing inner syntax, Isabelle uses a rather general and sophisticated algorithm, which is driven by priority grammars. Parsers for outer syntax are built up by functional parsing combinators. These combinators are a well-established technique for parsing, which has, for example, been described in Paulson's classic ML-book [5]. Isabelle developers are usually concerned with writing these outer syntax parsers, either for new definitional packages or for calling methods 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* `Parse` *in the file* `Pure/Isar/parse.ML`*. In addition specific parsers for packages are defined in* `Pure/Isar/parse_spec.ML`*. Parsers for method arguments are defined in* `Pure/Isar/args.ML`*.*

5.1 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") (Symbol.explode "hello")
> ("h", ["e", "l", "l", "o"])
```

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

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

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

will raise the exception *FAIL*. The function *$$* will also fail if you try to consume more than a single character.

There are three exceptions that are raised by 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).

In the examples above we use the function *explode* from the structure *Symbol*, instead of the more standard library function *explode*, for obtaining an input list for the parser. The reason is that *explode* is aware of character sequences, for example *\<foo>*, that have a special meaning in Isabelle. To see the difference consider

```
let
  val input = "\<foo> bar"
in
  (String.explode input, Symbol.explode input)
end
> (["\", "<", "f", "o", "o", ">", " ", "b", "a", "r"],
>  ["\<foo>", " ", "b", "a", "r"])
```

Slightly more general than the parser *$$* is the function *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 = Symbol.explode "hello"
  val input2 = Symbol.explode "world"
in
  (hw input1, hw input2)
end
> (("h", ["e", "l", "l", "o"]),("w", ["o", "r", "l", "d"]))
```

Two parsers can be connected in sequence by using the function `--`. For example parsing `h`, `e` and `l` (in this order) you can achieve by:

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

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

If, as in the previous example, you want to parse a particular string, then you can use the function `this_string`.

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

Parsers that explore alternatives can be constructed using the function `||`. 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 = Symbol.explode "hello"
  val input2 = Symbol.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. That means the item being dropped is the one that `|--` and `--|` "point" away. For example:

```
let
  val just_e = $$ "h" |-- $$ "e"
  val just_h = $$ "h" --| $$ "e"
  val input = Symbol.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 = Symbol.explode "hello"
  val input2 = Symbol.explode "world"
in
  (p input1, p input2)
end
> (("h", ["e", "l", "l", "o"]), ("x", ["w", "o", "r", "l", "d"]))
```

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

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

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

The function `!!` helps with producing appropriate error messages during parsing. For example if you want to parse `p` immediately followed by `q`, or start a completely different parser `r`, you might write:

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

However, this parser is problematic for producing a useful error message, if the parsing of `(p -- q)` fails. Because with the parser above you lose the information that `p` should be followed by `q`. To see this assume that `p` is present in the input, but it is not followed by `q`. That means `(p -- q)` will fail and hence the alternative parser `r` will be tried. However, in many circumstances this will be the wrong parser for the input "`p`-followed-by-something" 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 _ => fn _ =>"foo") ($$ "h")
```

on `"hello"`, the parsing succeeds

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

but if you invoke it on `"world"`

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

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

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

This kind of "prefixing" to see the correct error message is usually done by wrappers such as `local_theory` (see Section 5.8 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 failure of parsing 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
  ($$ p -- (!! (fn _ => err_msg) ($$ q))) || ($$ r -- $$ r)
end
```

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

```
Scan.error (p_followed_by_q "h" "e" "w") (Symbol.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") (Symbol.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") (Symbol.explode "hhhhello")
> (["h", "h", "h", "h"], ["e", "l", "l", "o"])
```

Note that `repeat` stores the parsed items in a list. The function `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 it run with the string `"hhhh"`. This can be avoided by using the wrapper `finite` and the "stopper-token" `stopper`. With them you can write:

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

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

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

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

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

The function `unless` 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") (Symbol.explode "hello")
> Exception FAIL raised
```

fails, while

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

succeeds.

The functions `repeat` and `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 = Symbol.explode "fooooo"
  val input2 = Symbol.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 almost always want to apply a function to the parsed items. One way to do this is the function `>>` where `(p >> f)` 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)
  val parser = $$ "h" -- $$ "e"
in
  (parser >> double) (Symbol.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 = Symbol.explode "foo bar foo"
in
   Scan.finite Symbol.stopper (p >> implode) input
end
> ("foo bar foo",[])
```

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

The function `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, Symbol.explode "hello")
> (("h", "e"), (1, ["l", "l", "o"]))
```

[1]

Be aware of recursive parsers. Suppose you want to read strings separated by commas and by parentheses into a tree datastructure; for example, generating the tree

[1] **FIXME: In which situations is `lift` useful? Give examples.**

corresponding to the string `"(A, A), (A, A)"` where the `A`s will be the leaves. We assume the trees are represented by the datatype:

```
datatype tree =
    Lf of string
  | Br of tree * tree
```

Since nested parentheses should be treated in a meaningful way—for example the string `"((A))"` should be read into a single leaf—you might implement the following parser.

```
fun parse_basic s =
  $$ s >> Lf || $$ "(" |-- parse_tree s --| $$ ")"

and parse_tree s =
  parse_basic s --| $$ "," -- parse_tree s >> Br || parse_basic s
```

This parser corresponds to the grammar:

```
<Basic>   ::=  <String> | (<Tree>)
<Tree>    ::=  <Basic>, <Tree> | <Basic>
```

The parameter `s` is the string over which the tree is parsed. The parser `parse_basic` reads either a leaf or a tree enclosed in parentheses. The parser `parse_tree` reads either a pair of trees separated by a comma, or acts like `parse_basic`. Unfortunately, because of the mutual recursion, this parser will immediately run into a loop, even if it is called without any input. For example

```
parse_tree "A"
> *** Exception- TOPLEVEL_ERROR raised
```

raises an exception indicating that the stack limit is reached. Such looping parser are not useful, because of ML's strict evaluation of arguments. Therefore we need to delay the execution of the parser until an input is given. This can be done by adding the parsed string as an explicit argument. So the parser above should be implemented as follows.

```
fun parse_basic s xs =
  ($$ s >> Lf || $$ "(" |-- parse_tree s --| $$ ")") xs

and parse_tree s xs =
  (parse_basic s --| $$ "," -- parse_tree s >> Br || parse_basic s) xs
```

While the type of the parser is unchanged by the addition, its behaviour changed: with this version of the parser the execution is delayed until some string is applied for the argument `xs`. This gives us exactly the parser what we wanted. An example is as follows:

```
let
  val input = Symbol.explode "(A,((A))),A"
in
  Scan.finite Symbol.stopper (parse_tree "A") input
end
> (Br (Br (Lf "A", Lf "A"), Lf "A"), [])
```

Exercise 5.1.1: *Write a parser that parses an input string so that any comment enclosed within `(*...*)` is replaced by the same comment but enclosed within `(**...**)` in the output string. To enclose a string, you can use the function `enclose s1 s2 s` which produces the string `s1 ^ s ^ s2`. Hint: To simplify the task ignore the proper nesting of comments.*

5.2 Parsing Theory Syntax

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

```
type 'a parser = Token.T list -> 'a * Token.T list
```

REDO!!

The reason for using token parsers is that theory syntax, as well as the parsers for the arguments of proof methods, use the type `Token.T`.

> ***Read More***
> *The parser functions for the theory syntax are contained in the structure `Parse` defined in the file `Pure/Isar/parse.ML`. The definition for tokens is in the file `Pure/Isar/token.ML`.*

The structure `Token` 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 `scan`. It is given the argument `Position.none` since, at the moment, we are not interested in generating precise error messages. The following code

```
Outer_Syntax.scan
  (Keyword.get_lexicons ()) 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. The second indicates a space.

We can easily change what is recognised as a keyword with the function `define`. For example calling it with

```
val _ = Keyword.define ("hello", NONE)
```

then lexing *"hello world"* will produce

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

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

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

For convenience we define the function:

```
fun filtered_input str =
  filter Token.is_proper (Outer_Syntax.scan (Keyword.get_lexicons ())
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 one command and two keyword tokens. If you want to see which keywords and commands are currently known to Isabelle, use the function *get_lexicons*:

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

You might have to adjust the `default_print_depth` in order to see the complete list.

The parser `$$$` parses a single keyword. For example:

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

Any non-keyword string can be parsed with the function `reserved`. For example:

```
let
  val p = Parse.reserved "bar"
  val input = filtered_input "bar"
in
  p input
end
> ("bar",[])
```

Like before, you can sequentially connect parsers with `--`. For example:

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

The parser `Parse.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
  (Parse.enum "|" (Parse.$$$ "in")) input
end
> (["in", "in", "in"], [...])
```

The function `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" `Token.stopper`. We can write:

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

The following function will help to run examples.

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

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

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

Exercise 5.2.1: *(FIXME) A type-identifier, for example `'a`, is a token of kind `Keyword`. It can be parsed using the function `type_ident`.*

(FIXME: or give parser for numbers)

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

```
fun filtered_input' str =
       filter Token.is_proper (Outer_Syntax.scan (Keyword.get_lexicons ())
(Position.line 7) str)
```

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

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

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

By using the parser *position* you can access the token position and return it as part of the parser result. For example

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

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

Exercise 5.2.2: *Write a parser for the context-free grammar representing arithmetic expressions with addition and multiplication. As usual, multiplication binds stronger than addition, and both of them nest to the right. The context-free grammar is defined as:*

```
<Basic>   ::=  <Number> | (<Expr>)
<Factor>  ::=  <Basic> * <Factor> | <Basic>
<Expr>    ::=  <Factor> + <Expr>  | <Factor>
```

Hint: Be careful with recursive parsers.

5.3 Parsers for ML-Code (TBD)

ML_source

5.4 Context Parser (TBD)

Args.context

Args.context

Used for example in **attribute_setup** and **method_setup**.

5.5 Argument and Attribute Parsers (TBD)

5.6 Parsing Inner Syntax

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

```
let
  val input = Outer_Syntax.scan (Keyword.get_lexicons ()) Position.none
"foo"
in
  Parse.term input
end
> ("<markup>", [])
```

The function *prop* is similar, except that it gives a different error message, when parsing fails. As you can see, the parser not just returns the parsed string, but also some markup information. You can decode the information with the function *parse* in *YXML*. The result of the decoding is an XML-tree. You can see better what is going on if you replace *Position.none* by *Position.line 42*, say:

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

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

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

FIXME: *parse_term check_term parse_typ check_typ read_term read_term*

5.7 Parsing Specifications

There are a number of special purpose parsers that help with parsing specifications of function definitions, inductive predicates and so on. In Chapter 7, for example, we will need to parse specifications for inductive predicates of the form:

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

For this we are going to use the parser:

```
1 val spec_parser =
2     Parse.fixes --
```

```
3       Scan.optional
4         (Parse.$$$ "where" |--
5            Parse.!!!
6              (Parse.enum1 "|"
7                 (Parse_Spec.opt_thm_name ":" -- Parse.prop))) []
```

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

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

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

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

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

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

Whenever types are given, they are stored in the *SOME*s. The types are not yet used to type the variables: this must be done by type-inference later on. Since types are part

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

of the inner syntax they are strings with some encoded information (see previous section). If a mixfix-syntax is present for a variable, then it is stored in the `Mixfix` data structure; no syntax translation is indicated by `NoSyn`.

> ***Read More***
> *The data structure for mixfix annotations are implemented in* `Pure/Syntax/mixfix.ML` *and* `Pure/Syntax/syntax.ML`.

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

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

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

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

Exercise 5.7.1: *Have a look at how the parser* `Parse_Spec.where_alt_specs` *is implemented in file* `Pure/Isar/parse_spec.ML`. *This parser corresponds to the "where-part" of the introduction rules given above. Below we paraphrase the code of* `where_alt_specs` *adapted to our purposes.*

```
1  val spec_parser' =
2       Parse.fixes --
3       Scan.optional
4       (Parse.$$$ "where" |--
5          Parse.!!!
6            (Parse.enum1 "|"
7               ((Parse_Spec.opt_thm_name ":" -- Parse.prop) --|
8                    Scan.option (Scan.ahead (Parse.name ||
9                    Parse.$$$ "[") --
10                   Parse.!!! (Parse.$$$ "|"))))) []
```

Both parsers accept the same input, but if you look closely, you can notice an additional "tail" (Lines 8 to 10) in `spec_parser'`. *What is the purpose of this additional "tail"?*

(FIXME: `Parse.type_args`, `Parse.typ`, `Parse.opt_mixfix`)

5.8 New Commands

Often new commands, for example for providing new definitional principles, need to be implemented. While this is not difficult on the ML-level and for jEdit, in order to be backwards compatible, new commands need also to be recognised by Proof-General. This results in some subtle configuration issues, which we will explain in the next section. Here we just describe how to define new commands to work with jEdit.

Let us start with a "silly" command that does nothing at all. We shall name this command **foobar**. Before you can implement any new command, you have to "announce" it in the **keywords**-section of your theory header. For **foobar** we need to write something like

```
theory Foo
imports Main
keywords "foobar" :: thy_decl
...
```

whereby *thy_decl* indicates the kind of the command. Another possible kind is *thy_goal*, or you can also omit the kind entirely, in which case you declare a keyword (something that is part of a command).

Now you can implement **foobar** as follows.

```
let
  val do_nothing = Scan.succeed (Local_Theory.background_theory I)
in
  Outer_Syntax.local_theory @{command_spec "foobar"}
    "description of foobar"
      do_nothing
end
```

The crucial function *local_theory* expects the name for the command, a kind indicator, a short description and a parser producing a local theory transition (explained later). For the name and the kind, you can use the ML-antiquotation *@{command_spec ...}*. You can now write in your theory

foobar

but of course you will not see anything since **foobar** is not intended to do anything. Remember, however, that this only works in jEdit. In order to enable also Proof-General recognise this command, a keyword file needs to be generated (see next section).

As it stands, the command **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. We announce the command **foobar_trace** in the theory header as

keywords *"foobar_trace" :: thy_decl*

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 the parser *succeed* does not parse any argument, but immediately returns the simple function *Local_Theory.background_theory I*. We can replace this code by a function that first parses a proposition (using the parser *Parse.prop*), then prints out some tracing information (using the function *trace_prop*) and finally does nothing. For this you can write:

```
let
  fun trace_prop str =
    Local_Theory.background_theory (fn ctxt => (tracing str; ctxt))
in
  Outer_Syntax.local_theory @{command_spec "foobar_trace"}
    "traces a proposition"
      (Parse.prop >> trace_prop)
end
```

This command can now be used to see the proposition in the tracing buffer.

foobar_trace *"True ∧ False"*

Note that so far we used *thy_decl* as the kind indicator for the new 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* and the function *local_theory_to_proof* to set up the command. Below we show the command **foobar_goal** which takes a proposition as argument and then starts a proof in order to prove it. Therefore, we need to announce this command in the header as *thy_goal*.

keywords *"foobar_goal" :: thy_goal*

Then we can write:

```
let
  fun goal_prop str ctxt =
    let
      val prop = Syntax.read_prop ctxt str
    in
      Proof.theorem NONE (K I) [[(prop, [])]] ctxt
    end
in
  Outer_Syntax.local_theory_to_proof @{command_spec "foobar_goal"}
    "proves a proposition"
      (Parse.prop >> goal_prop)
end
```

The function *goal_prop* 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 *read_prop*, which converts a string into a proper proposition. In Line 6 the function *theorem* 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).

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

foobar_goal *"True ∧ True"*

```
goal (1 subgoal):
 1. True ∧ True
```

and can prove the proposition as follows.

apply*(rule conjI)*
apply*(rule TrueI)+*
done

The last command we describe here is **foobar_proof**. Like **foobar_goal**, its purpose is to take a proposition and open a corresponding proof-state that allows us to give a proof for it. However, unlike **foobar_goal**, the proposition will be given as a ML-value. Such a command is quite useful during development when you generate a goal on the ML-level and want to see whether it is provable. In addition we want to allow the proved proposition to have a name that can be referenced later on.

The first problem for **foobar_proof** is to parse some text as ML-source and then interpret it as an Isabelle term using the ML-runtime. For the parsing part, we can use the function *ML_source* in the structure *Parse*. For running the ML-interpreter we need the following scaffolding code.

```
structure Result = Proof_Data
  (type T = unit -> term
   fun init thy () = error "Result")

val result_cookie = (Result.get, Result.put, "Result.put")
```

With this in place, we can implement the code for **foobar_prove** as follows.

```
1  let
2     fun after_qed thm_name thms lthy =
3           Local_Theory.note (thm_name, (flat thms)) lthy |> snd
4
5     fun setup_proof (thm_name, {text, ...}) lthy =
6       let
7         val trm = Code_Runtime.value lthy result_cookie ("", text)
8       in
9         Proof.theorem NONE (after_qed thm_name) [[(trm, [])]] lthy
10      end
11
12    val parser = Parse_Spec.opt_thm_name ":" -- Parse.ML_source
```

```
13 in
14    Outer_Syntax.local_theory_to_proof @{command_spec "foobar_prove"}
15      "proving a proposition"
16        (parser >> setup_proof)
17 end
```

In Line 12, we implement a parser that first reads in an optional lemma name (terminated by ":") and then some ML-code. The function in Lines 5 to 10 takes the ML-text and lets the ML-runtime evaluate it using the function *value* in the structure *Code_Runtime*. Once the ML-text has been turned into a term, the function *theorem* opens a corresponding proof-state. This function takes the function *after_qed* as argument, whose purpose is to store the theorem (once it is proven) under the given name *thm_name*.

You can now define a term, for example

```
val prop_true = @{prop "True"}
```

and give it a proof using **foobar_prove**:

foobar_prove *test: prop_true*
apply*(rule TrueI)*
done

Finally you can test whether the lemma has been stored under the given name.

thm *test*
> True

While this is everything you have to do for a new command when using jEdit, things are not as simple when using Emacs and ProofGeneral. We explain the details next.

5.9 Proof-General and Keyword Files

In order to use a new command in Emacs and Proof-General, you need a keyword file that can be loaded by ProofGeneral. To keep things simple we take as running example the command **foobar** from the previous section.

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 5.1 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
keywords "foobar" :: thy_decl
begin
ML {*
let
  val do_nothing = Scan.succeed (Local_Theory.background_theory I)
in
  Outer_Syntax.local_theory @{command_spec "foobar"}
    "description of foobar"
       do_nothing
end
*}
end
```

Figure 5.1: This file can be used to generate a log file. This log file in turn can be used 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

```
no_document 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/Isabelle2012/polyml-5.2.1_x86-linux/log
```

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

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

on the Unix prompt. If you now type `ls $ISABELLE_LOGS`, then the directory should include the files:

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

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

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

The result is the file `isar-keywords-foobar.el`. It should contain the string `foobar` twice.[3] This keyword file needs to be copied into the directory `~/.isabelle/etc`. To make ProofGeneral aware of it, 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 you can now use the command **foobar** in Proof-General

A similar procedure has to be done with any other new command, and also any new keyword that is introduced with the function `define`. For example:

```
val _ = Keyword.define ("blink", NONE)
```

Also if the kind of a command changes, from `thy_decl` to `thy_goal` say, you need to recreate the keyword file.

TBD below

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

5.10 Methods (TBD)

(FIXME: maybe move to after the tactic section)

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

print_methods
```
> methods:
>   -:  do nothing (insert current facts only)
>   HOL.default:  apply some intro/elim rule (potentially classical)
>   ...
```

An example of a very simple method is:

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

It defines the method *foo*, which takes no arguments (therefore the parser *Scan.succeed*) and only applies a single tactic, namely the tactic which applies *conjE* and then *conjI*. The function *SIMPLE_METHOD* turns such a tactic into a method. The method *foo* can be used as follows

lemma shows *"A ∧ B ⟹ C ∧ D"*
apply(*foo*)

where it results in the goal state
goal (2 subgoals):
1. ⟦A; B⟧ ⟹ C 2. ⟦A; B⟧ ⟹ D

method_setup *test = {*
*Scan.lift (Scan.succeed (K Method.succeed)) *} {* bla *}*

lemma *"True"*
apply(*test*)
oops

method_setup *joker = {*
*Scan.lift (Scan.succeed (fn ctxt => Method.cheating ctxt true)) *} {* bla *}*

lemma *"False"*
apply(*joker*)
oops

if true is set then always works

ML *{* atac *}*

method_setup *first_atac = {* Scan.lift (Scan.succeed (K (SIMPLE_METHOD (atac 1))))
} { bla *}*

```
ML {* HEADGOAL *}

lemma "A ⟹ A"
apply(first_atac)
oops

method_setup my_atac = {* Scan.lift (Scan.succeed (K (SIMPLE_METHOD' atac))) *}
{* bla *}

lemma "A ⟹ A"
apply(my_atac)
oops
```

```
ML {* resolve_tac *}

method_setup myrule =
  {* Scan.lift (Scan.succeed (K (METHOD (fn thms => resolve_tac thms 1)))) *}
  {* bla *}

lemma
  assumes a: "A ⟹ B ⟹ C"
  shows "C"
using a
apply(myrule)
oops
```

(**********) (FIXME: explain a version of rule-tac)

Chapter 6

Tactical Reasoning

"The first thing I would say is that when you write a program, think of it primarily as a work of literature. You're trying to write something that human beings are going to read. Don't think of it primarily as something a computer is going to follow. The more effective you are at making your program readable, the more effective it's going to be: You'll understand it today, you'll understand it next week, and your successors who are going to maintain and modify it will understand it."

Donald E. Knuth, from an interview in Dr. Dobb's Journal, 1996.

One of the main reason for descending to the ML-level of Isabelle is to be able to implement automatic proof procedures. Such proof procedures can considerably lessen the burden of manual reasoning. They are centred around the idea of 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 discharged. In this chapter we will explain simple tactics and how to combine them using tactic combinators. We also describe the simplifier, simprocs and conversions.

6.1 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:
  shows "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 `prove` sets up a goal state for proving the goal `P ∨ Q ⟹ Q ∨ P`. We can give this function some assumptions in the third argument (there are no assumption in the proof at hand). The second argument stands for a list of variables (given as strings). This list contains the variables that will be turned into schematic variables once the goal is proved (in our case `P` and `Q`). The last argument is the tactic that proves the goal. This tactic can make use of the local assumptions (there are none in this example). The tactics `etac`, `rtac` and `atac` in the code above correspond roughly to `erule`, `rule` and `assumption`, respectively. The combinator `THEN` strings the tactics together.

TBD: Write something about `prove_multi`.

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

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
  shows `"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. There are some goal transformation that are performed by `tactic`. They can be avoided by using `raw_tactic` instead.

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)
```

where *THEN'* is used instead of *THEN*. (For most combinators that combine tactics—*THEN* is only one such combinator—a "primed" version exists.) With *foo_tac'* you can 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
  shows *"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.

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 these tactics return the error message:[1]

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

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

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

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

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

[1] To be precise, tactics do not produce this error message; the message originates from the **apply** wrapper that calls the tactic.

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

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

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

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

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

```
lemma
  shows "⟦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 _ = tracing (Pretty.string_of (pretty_thm_no_vars ctxt thm))
in
  Seq.single thm
end
```

which prints out the given theorem (using the string-function defined in Section 2.2) and then behaves like `all_tac`. With this tactic we are in the position to inspect every goal state in a proof. For this consider the proof in Figure 6.1: 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 (the wrapper is the outermost constant *Const ("Pure.prop", bool ⇒ bool)*; in the figure we make it visible as a *#*). This wrapper prevents that premises of *C* are misinterpreted 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. This instantiations of schematic variables can be observed on the user level. Have a look at the following definition and proof.

```
definition
  EQ_TRUE
where
  "EQ_TRUE X ≡ (X = True)"

schematic_lemma test:
  shows "EQ_TRUE ?X"
unfolding EQ_TRUE_def
by (rule refl)
```

By using **schematic_lemma** it is possible to prove lemmas including meta-variables on the user level. However, the proved theorem is not *EQ_TRUE ?X*, as one might expect, but *EQ_TRUE True*. We can test this with:

```
thm test
> EQ_TRUE True
```

The reason for this result is that the schematic variable *?X* is instantiated inside the proof to be *True* and then the instantiation propagated to the "outside".

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

6.2 Simple Tactics

In this section we will introduce more of the simple tactics in Isabelle. The first one is *print_tac*, which is quite useful for low-level debugging of tactics. It just prints out a message and the current goal state. Unlike *my_print_tac* shown earlier, it prints the goal state as the user would see it. For example, processing the proof

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

gives:

```
foo message

False ⟹ True
1. False ⟹ True
```

notation (**output**) *"Pure.prop"* (*"#_" [1000] 1000*)

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)

Figure 6.1: The figure shows an Isabelle snippet of a proof where each intermediate goal state is printed by the Isabelle system and by *my_print_tac*. The latter shows the goal state as represented internally (highlighted boxes). This tactic shows that every goal state in Isabelle is represented by a theorem: when you start the proof of *⟦A; B⟧ ⟹ A ∧ B* the theorem is *(⟦A; B⟧ ⟹ A ∧ B) ⟹ #(⟦A; B⟧ ⟹ A ∧ B)*; when you finish the proof the theorem is *#(⟦A; B⟧ ⟹ A ∧ B)*.

A simple tactic for easy discharge of any proof obligations, even difficult ones, is *cheat_tac* in the structure *Skip_Proof*. This tactic corresponds to the Isabelle command **sorry** and is sometimes useful during the development of tactics.

lemma
  shows *"False"* **and** *"Goldbach_conjecture"*
apply(*tactic {* Skip_Proof.cheat_tac 1 *}*)

```
goal (1 subgoal):
 1. Goldbach_conjecture
```

Another simple tactic is the function *atac*, which, as shown earlier, corresponds to the assumption method.

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

```
No subgoals!
```

Similarly, *rtac*, *dtac*, *etac* and *ftac* correspond (roughly) 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
```

The function *resolve_tac* is similar to *rtac*, except that it expects a list of theorems as argument. 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_xmp_tac = 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_xmp_tac 1 *})
apply(tactic {* resolve_xmp_tac 2 *})

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

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

Another simple tactic is `cut_facts_tac`. It inserts a list of theorems into the assumptions of the current goal state. Below we will insert the definitions for the constants `True` and `False`. So

```
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
```

Often proofs on the ML-level involve elaborate operations on assumptions and $\bigwedge$-quantified variables. To do such operations using the basic tactics shown so far is very unwieldy and brittle. Some convenience and safety is provided by the functions `FOCUS` and `SUBPROOF`. These tactics fix the parameters and bind the various components of a goal state to a record. To see what happens, suppose the function defined in Figure 6.2, which takes a record and just prints out the contents of this record. Then consider the proof:

```
schematic_lemma
  shows "⋀x y. A x y ⟹ B y x ⟶ C (?z y) x"
apply(tactic {* Subgoal.FOCUS foc_tac @{context} 1 *})
```

The tactic produces the following printout:

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

The `params` and `schematics` stand for list of pairs where the left-hand side of `:=` is replaced by the right-hand side inside the tactic. Notice that in the actual output the variables `x` and `y` have a brown colour. Although they are parameters in the original goal, they are fixed inside the tactic. 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`.

```
fun foc_tac {prems, params, asms, concl, context, schematics} =
let
  fun assgn1 f1 f2 xs =
    let
      val out = map (fn (x, y) => Pretty.enum ":=" "" "" [f1 x, f2 y]) xs
    in
      Pretty.list "" "" out
    end

  fun assgn2 f xs = assgn1 f f xs

  val pps = map (fn (s, pp) => Pretty.block [Pretty.str s, pp])
    [("params: ", assgn1 Pretty.str (pretty_cterm context) params),
     ("assumptions: ", pretty_cterms context asms),
     ("conclusion: ", pretty_cterm context concl),
     ("premises: ", pretty_thms_no_vars context prems),
     ("schematics: ", assgn2 (pretty_cterm context) (snd schematics))]
in
  tracing (Pretty.string_of (Pretty.chunks pps)); all_tac
end
```

Figure 6.2: A function that prints out the various parameters provided by *FOCUS* and *SUBPROOF*. It uses the functions defined in Section 2.2 for extracting strings from *cterm*s and *thm*s.

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

apply(*rule impI*)
apply(*tactic {* Subgoal.FOCUS foc_tac @{context} 1 *}*)

then the tactic prints out:

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

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

One difference between the tactics *SUBPROOF* and *FOCUS* is that the former expects that the goal is solved completely, which the latter does not. Another is that *SUBPROOF* cannot instantiate any schematic variables.

Observe that inside *FOCUS* and *SUBPROOF*, we are in a goal state where there is only a conclusion. This means the tactics *dtac* and the like are of no use for manipulating the goal state. The assumptions inside *FOCUS* and *SUBPROOF* are given as cterms and theorems in the arguments *asms* and *prems*, respectively. This means we can apply them using the usual assumption tactics. With this you can for example easily implement a tactic that behaves almost like *atac*:

```
val atac' = Subgoal.FOCUS (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!
```

Notice that `atac'` inside `FOCUS` calls `resolve_tac` with the subgoal number `1` and also the outer call to `FOCUS` in the **apply**-step uses `1`. This is another advantage of `FOCUS` and `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
```

To sum up, both `FOCUS` and `SUBPROOF` are rather convenient, but can impose a considerable run-time penalty in automatic proofs. If speed is of the essence, then maybe the two lower level combinators described next are more appropriate.

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

Similar but less powerful functions than `FOCUS`, respectively `SUBPROOF`, are `SUBGOAL` and `CSUBGOAL`. They allow you to inspect a given subgoal (the former presents the subgoal as a `term`, while the latter as a `cterm`). With them 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 this pattern using the basic term-constructors. This is not necessary in the other cases, because their type is always fixed to function types involving only the type *bool*. (See Section 3.2 about constructing terms manually.) For the catch-all pattern, we chose to just return *all_tac*. Consequently, *select_tac* never fails.

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

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

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

where in all but the last the tactic applies 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 tactic analysing a goal according to its topmost connective. These simpler methods use tactic combinators, which we will explain in the next section. But before that we will show how tactic application can be constrained.

> ***Read More***
> *The functions* SUBGOAL *and* CSUBGOAL *are defined in* Pure/tactical.ML.

Since *rtac* and the like use higher-order unification, an automatic proof procedure based on them might become too fragile, if it just applies theorems as shown above.

The reason is that a number of theorems introduce schematic variables into the goal state. Consider for example the proof

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

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

where the application of theorem `bspec` generates two subgoals involving the new schematic variable `?x`. Now, if you are not careful, tactics applied to the first subgoal might instantiate this schematic 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 constrained version of the `bspec`-theorem. One way to give such constraints is by pre-instantiating theorems with other theorems. The function `RS`, for example, does this.

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

In this example it instantiates the first premise of the `conjI`-theorem with the theorem `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. This number makes explicit which premise should be instantiated.

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

```
[@{thm disjI1}, @{thm disjI2}] MRS @{thm conjI}
> ⟦?P2; ?Q1⟧ ⟹ (?P2 ∨ ?Q2) ∧ (?P1 ∨ ?Q1)
```

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

```
let
  val list1 = [@{thm impI}, @{thm disjI2}]
  val list2 = [@{thm conjI}, @{thm disjI1}]
in
  list1 RL list2
end
```

```
> [⟦?P1 ⟹ ?Q1; ?Q⟧ ⟹ (?P1 ⟶ ?Q1) ∧ ?Q,
>  ⟦?Q1; ?Q⟧ ⟹ (?P1 ∨ ?Q1) ∧ ?Q,
>  (?P1 ⟹ ?Q1) ⟹ (?P1 ⟶ ?Q1) ∨ ?Q,
>  ?Q1 ⟹ (?P1 ∨ ?Q1) ∨ ?Q]
```

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

Higher-order unification is also often in the way when applying certain congruence theorems. For example we would expect that the theorem *cong*

thm *cong*
```
> ⟦?f = ?g; ?x = ?y⟧ ⟹ ?f ?x = ?g ?y
```

is applicable in the following proof producing the subgoals `t x = s u` and `y = w`.

lemma
  fixes `x y u w::"'a"`
  shows `"t x y = s u w"`
apply`(rule cong)`

```
goal (2 subgoals):
 1. (λa. a) = (λa. a)
 2. t x y = s u w
```

As you can see this is unfortunately *not* the case if we apply *cong* with the method *rule*. The problem is that higher-order unification produces an instantiation that is not the intended one. While we can use **back** to interactively find the intended instantiation, this is not an option for an automatic proof procedure. Fortunately, the tactic *cong_tac* helps with applying congruence theorems and finding the intended instantiation. For example

lemma
  fixes `x y u w::"'a"`
  shows `"t x y = s u w"`
apply`(tactic {* Cong_Tac.cong_tac @{thm cong} 1 *})`

```
goal (2 subgoals):
 1. t x = s u
 2. y = w
```

However, frequently it is necessary to explicitly match a theorem against a goal state and in doing so construct manually an appropriate instantiation. Imagine you have the theorem

lemma *rule:*
  shows `"⟦f = g; x = y⟧ ⟹ R (f x) (g y)"`
sorry

and you want to apply it to the goal $t_1\ t_2 \leq s_1\ s_2$. Since in the theorem all variables are schematic, we have a nasty higher-order unification problem and *rtac* will not be able to apply this rule in the way we want. For the goal at hand we want

to use it so that *R* is instantiated to the constant $\leq$ and similarly "obvious" instantiations for the other variables. To achieve this we need to match the conclusion of *rule* against the goal reading off an instantiation for *rule*. For this the function *first_order_match* matches two *cterm*s and produces, in the successful case, a matcher that can be used to instantiate the theorem. The instantiation can be done with the function *instantiate_normalize*. To automate this we implement the following function.

```
1 fun fo_rtac thm = Subgoal.FOCUS (fn {concl, ...} =>
2   let
3     val concl_pat = Drule.strip_imp_concl (cprop_of thm)
4     val insts = Thm.first_order_match (concl_pat, concl)
5   in
6     rtac (Drule.instantiate_normalize insts thm) 1
7   end)
```

Note that we use *FOCUS* because it gives us directly access to the conclusion of the goal state, but also because this function takes care of correctly handling parameters that might be present in the goal state. In Line 3 we use the function *strip_imp_concl* for calculating the conclusion of a theorem (produced as *cterm*). An example of *fo_rtac* is as follows.

lemma
  shows "$\bigwedge t_1\ s_1\ (t_2::'a)\ (s_2::'a).\ (t_1\ t_2) \leq (s_1\ s_2)$"
apply(*tactic {* fo_rtac @{thm rule} @{context} 1 *}*)

goal (2 subgoals):
1. $\bigwedge t_1\ s_1\ t_2\ s_2.\ t_1 = s_1$
2. $\bigwedge t_1\ s_1\ t_2\ s_2.\ t_2 = s_2$

We obtain the intended subgoals and also the parameters are correctly introduced in both of them. Such manual instantiations are quite frequently necessary in order to appropriately constrain the application of theorems. Otherwise one can end up with "wild" higher-order unification problems that make automatic proofs fail.

> ***Read More***
> *Functions for matching* cterms *are defined in* Pure/thm.ML*. Functions for instantiating schematic variables in theorems are defined in* Pure/drule.ML*.*

6.3 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* $\wedge$ *Bar*) $\wedge$ *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 in each component, then, as seen earlier, 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 have to specify the subgoal of interest only once and it is consistently applied to the component. For most tactic combinators such a "primed" version exists and in what follows we will usually prefer it over the "unprimed" one.

The tactic combinator *RANGE* takes a list of *n* tactics, say, and applies them to each of the first *n* subgoals. For example below we first apply the introduction rule for conjunctions and then apply a tactic to each of the two subgoals.

```
lemma
  shows "A ⟹ True ∧ A"
apply(tactic {* (rtac @{thm conjI}
                 THEN' RANGE [rtac @{thm TrueI}, atac]) 1 *})

No subgoals!
```

If there is a list of tactics that should all be tried out in sequence on one specified subgoal, you can use the combinator *EVERY'*. For example the function *foo_tac'* from page 115 can also be written as:

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

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

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

and call *foo_tac1*.

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

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

will first try out whether theorem *disjI* applies and in case of failure will try *conjI*. To see this consider the proof

lemma
  shows "True ∧ False" **and** "Foo ∨ Bar"
apply(tactic {* orelse_xmp_tac 2 *})
apply(tactic {* orelse_xmp_tac 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 123 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 theorem applies (we also have to wrap *all_tac* using the *K*-combinator, because it does not take a subgoal number as argument). You can test the tactic on the same goals:

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

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

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

lemma
  shows "A ∧ 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 by fact of having `all_tac` at the end of the tactic list. The same can be achieved with the tactic combinator `TRY`. For example:

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

This tactic behaves in the same way as `select_tac'`: it tries out one of the given tactics and if none applies leaves the goal state unchanged. This, however, can be potentially very confusing when visible to 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 theorem applies. But because we wrapped the tactic in a `TRY`, it does not fail anymore. The problem is that for the user there is little chance to see whether progress in the proof has been made, or not. 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` in `select_tac'` or delete `TRY` from `select_tac''`. 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 the `select_tac`s so that they not just apply 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_tac = REPEAT (CHANGED (select_tac' 1))
```

which applied to the proof

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

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_tac*, then you can implement it as

```
val repeat_xmp_tac' = 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, then you see the tactics *repeat_xmp_tac* and *repeat_xmp_tac'* 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*. Supposing the tactic

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

you can see that the following goal

lemma
  shows *"((¬A) ∧ (∀x. B x)) ∧ (C ⟶ D)"*
apply*(tactic {* repeat_all_new_xmp_tac 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
  shows *"⟦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
  shows "⟦P1 ∨ Q1; P2 ∨ Q2⟧ ⟹ R"
apply(tactic {* DETERM (etac @{thm disjE} 1) *})

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

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

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

2 3

> ***Read More***
> *Most tactic combinators described in this section are defined in* `Pure/tactical.ML`*. Some combinators for the purpose of proof search are implemented in* `Pure/search.ML`*.*

Exercise 6.3.1: *Dyckhoff presents in [2] inference rules of a sequent calculus, called G4ip, for intuitionistic propositional logic. The interesting feature of this calculus is that no contraction rule is needed in order to be complete. As a result the rules can be applied exhaustively, which in turn leads to simple decision procedure for propositional intuitionistic logic. The task is to implement this decision procedure as a tactic. His rules are*

$$\frac{}{A, \Gamma \Rightarrow A}\ Ax \qquad \frac{}{False, \Gamma \Rightarrow G}\ False$$

$$\frac{A, B, \Gamma \Rightarrow G}{A \wedge B, \Gamma \Rightarrow G}\ \wedge_L \qquad \frac{\Gamma \Rightarrow A \quad \Gamma \Rightarrow B}{\Gamma \Rightarrow A \wedge B}\ \wedge_R$$

$$\frac{A, \Gamma \Rightarrow G \quad B, \Gamma \Rightarrow G}{A \vee B, \Gamma \Rightarrow G}\ \vee_L \qquad \frac{\Gamma \Rightarrow A}{\Gamma \Rightarrow A \vee B}\ \vee_{R_1} \qquad \frac{\Gamma \Rightarrow B}{\Gamma \Rightarrow A \vee B}\ \vee_{R_2}$$

$$\frac{B, A, \Gamma \Rightarrow G}{A \longrightarrow B, A, \Gamma \Rightarrow G}\ \longrightarrow_{L_1} \qquad \frac{A, \Gamma \Rightarrow B}{\Gamma \Rightarrow A \longrightarrow B}\ \longrightarrow_R$$

$$\frac{C \longrightarrow (D \longrightarrow B), \Gamma \Rightarrow G}{(C \wedge D) \longrightarrow B, \Gamma \Rightarrow G}\ \longrightarrow_{L_2} \qquad \frac{C \longrightarrow B, D \longrightarrow B, \Gamma \Rightarrow G}{(C \vee D) \longrightarrow B, \Gamma \Rightarrow G}\ \longrightarrow_{L_3}$$

$$\frac{D \longrightarrow B, \Gamma \Rightarrow C \longrightarrow D \quad B, \Gamma \Rightarrow G}{(C \longrightarrow D) \longrightarrow B, \Gamma \Rightarrow G}\ \longrightarrow_{L_4}$$

[2] **FIXME: say something about *COND* and COND'**
[3] **FIXME: PARALLEL-CHOICE PARALLEL-GOALS**

Note that in Isabelle right rules need to be implemented as introduction rule, the left rules as elimination rules. You have to to prove separate theorems corresponding to $\longrightarrow_{L_{1..4}}$. The tactic should explore all possibilites of applying these rules to a propositional formula until a goal state is reached in which all subgoals are discharged. For this you can use the tactic combinator `DEPTH_SOLVE` *in the structure* `Search`*.*

Exercise 6.3.2: *Add to the sequent calculus from the previous exercise also rules for equality and run your tactic on the de Bruijn formulae discussed in Exercise 3.2.5.*

6.4 Simplifier Tactics

A lot of convenience in reasoning with Isabelle derives from its powerful simplifier. We will describe it in this section. However, due to its complexity, we can mostly only scratch the surface.

The power of the simplifier is a strength and a weakness at the same time, because you can easily make the simplifier run into a loop and in general its behaviour can be difficult to predict. There is also a multitude of options that you can configure to change the behaviour of the simplifier. There are the following five main tactics behind the simplifier (in parentheses is their user-level counterpart):

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

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

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

corresponds on the ML-level to the tactic

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

If the simplifier cannot make any progress, then it leaves the goal unchanged, i.e., does not raise any error message. That means if you use it to unfold a definition for a constant and this constant is not present in the goal state, you can still safely apply the simplifier.
[4]

There is one restriction you have to keep in mind when using the simplifier: it can only deal with rewriting rules whose left-hand sides are higher order pattern (see

[4] **FIXME: show rewriting of cterms**

Section 3.3 on unification). Whether a term is a pattern or not can be tested with the function *pattern* from the structure *Pattern*. If a rule is not a pattern and you want to use it for rewriting, then you have to use simprocs or conversions, both of which we shall describe in the next section.

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

The rewriting inside terms requires congruence theorems, which are typically meta-equalities of the form

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

with *constr* being a constant, like *If*, *Let* and so on. Every simpset contains only one congruence rule for each term-constructor, which however can be overwritten. The user can declare lemmas to be congruence rules using the attribute *[cong]*. Note that in HOL these congruence theorems are usually stated as equations, which are then internally transformed into meta-equations.

The rewriting with theorems involving premises requires what is in Isabelle called a subgoaler, a solver and a looper. These can be arbitrary tactics that can be installed in a simpset and which are executed at various stages during simplification.

Simpsets can also include simprocs, which produce rewrite rules on demand according to a pattern (see next section for a detailed description of simpsets). Another component are split-rules, which can simplify for example the "then" and "else" branches of if-statements under the corresponding preconditions.

> ***Read More***
> *For more information about the simplifier see* Pure/raw_simplifier.ML *and* Pure/simplifier.ML*. The generic splitter is implemented in* Provers/splitter.ML*.*

[5]

The most common combinators for modifying simpsets are:

```
addsimps        delsimps
addsimprocs     delsimprocs
add_cong        del_cong
```

To see how they work, consider the function in Figure 6.3, which prints out some parts of a simpset. If you use it to print out the components of the empty simpset, i.e., *empty_ss*

[5] **FIXME: Find the right place to mention this: Discrimination nets are implemented in** Pure/net.ML**.**

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

  fun name_sthm (nm, thm) =
    Pretty.enclose (nm ^ ": ") "" [pretty_thm_no_vars ctxt thm]
  fun name_cthm ((_, nm), thm) =
    Pretty.enclose (nm ^ ": ") "" [pretty_thm_no_vars ctxt thm]
  fun name_ctrm (nm, ctrm) =
    Pretty.enclose (nm ^ ": ") "" [pretty_cterms ctxt ctrm]

  val pps = [Pretty.big_list "Simplification rules:" (map name_sthm simps),
             Pretty.big_list "Congruences rules:" (map name_cthm congs),
             Pretty.big_list "Simproc patterns:" (map name_ctrm procs)]
in
  pps |> Pretty.chunks
      |> pwriteln
end
```

Figure 6.3: The function *dest_ss* returns a record containing all printable information stored in a simpset. We are here only interested in the simplification rules, congruence rules and simprocs.

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

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

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

Printing then out the components of the simpset gives:

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

6

Adding also the congruence rule *strong_INF_cong*

[6] **FIXME: Why does it print out ??.unknown**

```
val ss2 = ss1 |> Simplifier.add_cong (@{thm strong_INF_cong} RS @{thm
eq_reflection})
```

gives

```
print_ss @{context} (Raw_Simplifier.simpset_of ss2)
> Simplification rules:
>   ??.unknown: A - B ∩ C ≡ A - B ∪ (A - C)
> Congruences rules:
>   Complete_Lattices.Inf_class.INFIMUM:
>     ⟦A1 = B1; ⋀x. x ∈ B1 =simp=> C1 x = D1 x⟧ ⟹ INFIMUM A1 C1 ≡
INFIMUM B1 D1
> Simproc patterns:
```

Notice that we had to add these lemmas as meta-equations. The *empty_ss* expects this form of the simplification and congruence rules. This is different, if we use for example the simpset *HOL_basic_ss* (see below), where rules are usually added as equation. However, even when adding these lemmas to *empty_ss* we do not end up with anything useful yet. In the context of HOL, the first really useful simpset is *HOL_basic_ss*. While printing out the components of this simpset

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

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

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

and also resolve with assumptions. For example:

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

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

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

```
print_ss @{context} HOL_ss
> Simplification rules:
>   Pure.triv_forall_equality: (⋀x. PROP V) ≡ PROP V
>   HOL.the_eq_trivial: THE x. x = y ≡ y
>   HOL.the_sym_eq_trivial: THE ya. y = ya ≡ y
>   ...
> Congruences rules:
>   HOL.simp_implies: ...
>     ⟹ (PROP P =simp=> PROP Q) ≡ (PROP P' =simp=> PROP Q')
>   op -->: ⟦P ≡ P'; P' ⟹ Q ≡ Q'⟧ ⟹ P ⟶ Q ≡ P' ⟶ Q'
> Simproc patterns:
>   ...
```

> ***Read More***
> *The simplifier for HOL is set up in* `HOL/Tools/simpdata.ML`*. The simpset* `HOL_ss` *is implemented in* `HOL/HOL.thy`*.*

The simplifier is often used to unfold definitions in a proof. For this the simplifier implements the tactic `rewrite_goal_tac`.[7] Suppose for example the definition

definition *"MyTrue ≡ True"*

then we can use this tactic to unfold the definition of this constant.

lemma
  shows *"MyTrue ⟹ True"*
apply(*tactic {* rewrite_goal_tac @{context} @{thms MyTrue_def} 1 *}*)

producing the goal state

```
goal (1 subgoal):
 1. True ⟹ True
```

If you want to unfold definitions in *all* subgoals, not just one, then use the the tactic `rewrite_goals_tac`.

The simplifier is often used in order to bring terms into a normal form. Unfortunately, often the situation arises that the corresponding simplification rules will cause the simplifier to run into an infinite loop. Consider for example the simple theory about permutations over natural numbers shown in Figure 6.4. The purpose of the lemmas is to push permutations as far inside as possible, where they might disappear by Lemma `perm_rev`. However, to fully normalise all instances, it would be desirable to add also the lemma `perm_compose` to the simplifier for pushing permutations over other permutations. Unfortunately, the right-hand side of this lemma is again an instance of the left-hand side and so causes an infinite loop. There seems to be no easy way to reformulate this rule and so one ends up with clunky proofs like:

lemma
fixes *c d::"nat"* **and** *pi$_1$ pi$_2$::"prm"*
  shows *"pi$_1$·(c, pi$_2$·((rev pi$_1$)·d)) = (pi$_1$·c, (pi$_1$·pi$_2$)·d)"*

[7] **FIXME: see LocalDefs infrastructure.**

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

overloading
  perm_nat ≡ "perm :: prm ⇒ nat ⇒ nat"
  perm_prod ≡ "perm :: prm ⇒ ('a×'b) ⇒ ('a×'b)"
  perm_list ≡ "perm :: prm ⇒ 'a list ⇒ 'a list"
begin

fun swap::"nat ⇒ nat ⇒ nat ⇒ nat"
where
 "swap a b c = (if c=a then b else (if c=b then a else c))"

primrec perm_nat
where
  "perm_nat [] c = c"
| "perm_nat (ab#pi) c = swap (fst ab) (snd ab) (perm_nat pi c)"

fun perm_prod
where
 "perm_prod pi (x, y) = (pi·x, pi·y)"

primrec perm_list
where
  "perm_list pi [] = []"
| "perm_list pi (x#xs) = (pi·x)#(perm_list pi xs)"

end

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

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

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

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

Figure 6.4: A simple theory about permutations over *nat*s. The point is that the lemma *perm_compose* cannot be directly added to the simplifier, as it would cause the simplifier to loop. It can still be used as a simplification rule if the permutation in the right-hand side is sufficiently protected.

apply(*simp*)
apply(*rule trans*)
apply(*rule perm_compose*)
apply(*simp*)
done

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

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

Now the two lemmas

lemma *perm_aux_expand:*
  fixes *c::"nat"* **and** pi_1 pi_2*::"prm"*
  shows *"*pi_1*·(*pi_2*·c) =* pi_1 *·aux (*pi_2*·c)"*
unfolding *perm_aux_def* **by** *(rule refl)*

lemma *perm_compose_aux:*
  fixes *c::"nat"* **and** pi_1 pi_2*::"prm"*
  shows *"*pi_1*·(*pi_2*·aux c) = (*pi_1*·*pi_2*) ·aux (*pi_1*·c)"*
unfolding *perm_aux_def* **by** *(rule perm_compose)*

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

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

For all three phases we have to build simpsets adding specific lemmas. As is sufficient

for our purposes here, we can add these lemmas to *HOL_basic_ss* in order to obtain the desired results. Now we can solve the following lemma

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

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

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

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

(FIXME: what are *mksimps_pairs*; used in Nominal.thy)

(FIXME: explain *simplify* and *Simplifier.rewrite_rule* etc.)

6.5 Simprocs

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

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

```
1 fun fail_simproc ctxt redex =
2 let
3   val _ = pwriteln (Pretty.block
4     [Pretty.str "The redex: ", pretty_cterm ctxt redex])
5 in
6   NONE
7 end
```

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

```
simproc_setup fail ("Suc n") = {* K fail_simproc *}
```

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

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

```
> The redex: Suc 0
> The redex: Suc 0
```

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

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

```
declare [[simproc del: fail]]
```

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

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

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

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

```
fun fail_simproc' ctxt redex =
let
  val _ = pwriteln (Pretty.block
    [Pretty.str "The redex:", pretty_term ctxt redex])
in
  NONE
end
```

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

```
fun fail' ctxt =
let
  val thy = @{theory}
  val pat = [@{term "Suc n"}]
in
  Simplifier.simproc_global_i thy "fail_simproc'" pat (K fail_simproc' ctxt)
end
```

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

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

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

The simproc *fail'* prints out the sequence

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

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

lemma *plus_one:*
  shows *"Suc n ≡ n + 1"* **by** *simp*

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

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

and set up the simproc as follows.

```
fun plus_one ctxt =
let
  val thy = @{theory}
  val pat = [@{term "Suc n"}]
in
  Simplifier.simproc_global_i thy "sproc +1" pat (K plus_one_simproc ctxt)
end
```

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

lemma
  shows *"P (Suc (Suc (Suc 0))) (Suc 0)"*
apply*(tactic {* simp_tac (put_simpset HOL_basic_ss @{context} addsimprocs [plus_one @{context}]) 1*})*

where the simproc produces the goal state

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

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

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

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

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

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

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

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

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

The advantage of *get_thm_alt* is that it leaves very little room for something to go wrong; in contrast it is much more difficult to predict what happens with *arith_tac*, especially in more complicated circumstances. The disadvantage of *get_thm_alt* is

to find a simpset that is sufficiently powerful to solve every instance of the lemmas we like to prove. This requires careful tuning, but is often necessary in "production code".[8]

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

```
fun nat_number_simproc ctxt t =
  SOME (get_thm_alt ctxt (t, dest_suc_trm t))
  handle TERM _ => NONE
```

This function uses the fact that *dest_suc_trm* might raise an exception *TERM*. In this case there is nothing that can be rewritten and therefore no theorem is produced (i.e. the function returns *NONE*). To try out the simproc on an example, you can set it up as follows:

```
fun nat_number ctxt =
let
  val thy = @{theory}
  val pat = [@{term "Suc n"}]
in
  Simplifier.simproc_global_i thy "nat_number" pat (K nat_number_simproc
ctxt)
end
```

Now in the lemma

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

you obtain the more legible goal state

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

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

Exercise 6.5.1: *Write a simproc that replaces terms of the form* t_1 + t_2 *by their result. You can assume the terms are "proper" numbers, that is of the form 0, 1, 2 and so on.*

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

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

6.6 Conversions

Conversions are a thin layer on top of Isabelle's inference kernel, and can be viewed as a controllable, bare-bone version of Isabelle's simplifier. The purpose of conversions is to manipulate *cterm*s. However, we will also show in this section how conversions can be applied to theorems and to goal states. The type of conversions is

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

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

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

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

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

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

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

If you run this example, you will notice that the actual response is the seemingly nonsensical *2 + 10 ≡ 2 + 10*. The reason is that the pretty-printer for *cterm*s eta-normalises (sic) terms and therefore produces this output. If we get hold of the "raw" representation of the produced theorem, we obtain the expected result.

```
let
  val add = @{cterm "λx y. x + (y::nat)"}
  val two = @{cterm "2::nat"}
  val ten = @{cterm "10::nat"}
  val ctrm = Thm.apply (Thm.apply add two) ten
in
```

```
  Thm.prop_of (Thm.beta_conversion true ctrm)
end
> Const ("Pure.eq",...) $
>   (Abs ("x",...,Abs ("y",...,...)) $...$...) $
>     (Const ("Groups.plus_class.plus",...) $ ... $ ...)
```

or in the pretty-printed form

```
let
   val add = @{cterm "λx y. x + (y::nat)"}
   val two = @{cterm "2::nat"}
   val ten = @{cterm "10::nat"}
   val ctrm = Thm.apply (Thm.apply add two) ten
   val ctxt = @{context}
     |> Config.put eta_contract false
     |> Config.put show_abbrevs false
in
  Thm.prop_of (Thm.beta_conversion true ctrm)
  |> pretty_term ctxt
  |> pwriteln
end
> (λx y. x + y) 2 10 ≡ 2 + 10
```

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

The main point of conversions is that they can be used for rewriting *cterm*s. One example is the function *rewr_conv*, which expects a meta-equation as an argument. Suppose the following meta-equation.

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

It can be used for example to rewrite *True ∧ (Foo ⟶ Bar)* to *Foo ⟶ Bar*. The code is as follows.

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

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

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

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

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

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

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

Here the conversion *true_conj1* only applies in the first case, but fails in the second. The whole conversion does not fail, however, because the combinator *else_conv* will then try out *all_conv*, which always succeeds. The same behaviour can also be achieved with conversion combinator *try_conv*. For example

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

Rewriting with more than one theorem can be done using the function *rewrs_conv* from the structure *Conv*.

Apart from the function *beta_conversion*, which is able to fully beta-normalise a term, the conversions so far are restricted in that they only apply to the outer-most level of a *cterm*. In what follows we will lift this restriction. The combinators *fun_conv* and *arg_conv* will apply a conversion to the first, respectively second, argument of an application. For example

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

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

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

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

Note that this conversion needs a context as an argument. We also give the conversion as `(K conv)`, which is a function that ignores its argument (the argument being a sufficiently freshened version of the variable that is abstracted and a context). The conversion that goes under an application is `combination_conv`. It expects two conversions as arguments, each of which is applied to the corresponding "branch" of the application. An abbreviation for this conversion is the function `comb_conv`, which applies the same conversion to both branches.

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

```
1 fun true_conj1_conv ctxt ctrm =
2   case (Thm.term_of ctrm) of
3     @{term "op ∧"} $ @{term True} $ _ =>
4       (Conv.arg_conv (true_conj1_conv ctxt) then_conv
5          Conv.rewr_conv @{thm true_conj1}) ctrm
6   | _ $ _ => Conv.comb_conv (true_conj1_conv ctxt) ctrm
7   | Abs _ => Conv.abs_conv (fn (_, ctxt) => true_conj1_conv ctxt) ctxt ctrm
8   | _ => Conv.all_conv ctrm
```

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

```
let
  val conv = true_conj1_conv @{context}
  val ctrm = @{cterm "distinct [1::nat, x] ⟶ True ∧ 1 ≠ x"}
in
  conv ctrm
end
> distinct [1, x] ⟶ True ∧ 1 ≠ x ≡ distinct [1, x] ⟶ 1 ≠ x
```

Conversions that traverse terms, like *true_conj1_conv* above, can be implemented more succinctly with the combinators *bottom_conv* and *top_conv*. For example:

```
fun true_conj1_conv ctxt =
let
  val conv = Conv.try_conv (Conv.rewr_conv @{thm true_conj1})
in
  Conv.bottom_conv (K conv) ctxt
end
```

If we regard terms as trees with variables and constants on the top, then *bottom_conv* traverses first the the term and on the "way down" applies the conversion, whereas *top_conv* applies the conversion on the "way up". To see the difference, assume the following two meta-equations

```
lemma conj_assoc:
  fixes A B C::bool
  shows "A ∧ (B ∧ C) ≡ (A ∧ B) ∧ C"
  and   "(A ∧ B) ∧ C ≡ A ∧ (B ∧ C)"
by simp_all
```

and the *cterm (a ∧ (b ∧ c)) ∧ d*. Here you should pause for a moment to be convinced that rewriting top-down and bottom-up according to the two meta-equations produces two results. Below these two results are calculated.

```
let
  val ctxt = @{context}
  val conv = Conv.try_conv (Conv.rewrs_conv @{thms conj_assoc})
  val conv_top = Conv.top_conv (K conv) ctxt
  val conv_bot = Conv.bottom_conv (K conv) ctxt
  val ctrm = @{cterm "(a ∧ (b ∧ c)) ∧ d"}
in
  (conv_top ctrm, conv_bot ctrm)
end
> ("(a ∧ (b ∧ c)) ∧ d ≡ a ∧ (b ∧ (c ∧ d))",
>  "(a ∧ (b ∧ c)) ∧ d ≡ (a ∧ b) ∧ (c ∧ d)")
```

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

```
fun if_true1_simple_conv ctxt ctrm =
  case Thm.term_of ctrm of
    Const (@{const_name If}, _) $ _ =>
      Conv.arg_conv (true_conj1_conv ctxt) ctrm
  | _ => Conv.no_conv ctrm

val if_true1_conv = Conv.top_sweep_conv if_true1_simple_conv
```

In the first function we only treat the top-most redex and also only the success case. As default we return *no_conv*. To apply this "simple" conversion inside a term, we use in the last line the combinator *top_sweep_conv*, which traverses the term starting from the root and applies it to all the redexes on the way, but stops in each branch as soon as it found one redex. Here is an example for this conversion:

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

So far we only applied conversions to *cterm*s. Conversions can, however, also work on theorems using the function *fconv_rule*. As an example, consider again the conversion *true_conj1_conv* and the lemma:

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

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

```
let
  val conv = Conv.fconv_rule (true_conj1_conv @{context})
  val thm = @{thm foo_test}
in
  conv thm
end
> ?P ∨ ¬ ?P
```

Finally, Isabelle provides function *CONVERSION* for turning conversions into tactics. This needs some predefined conversion combinators that traverse a goal state and can selectively apply the conversion. The combinators for the goal state are:

- *params_conv* for converting under parameters (i.e. where a goal state is of the form $\bigwedge x.\ P\ x \Longrightarrow Q\ x$)
- *prems_conv* for applying a conversion to premises of a goal state, and

- *concl_conv* for applying a conversion to the conclusion of a goal state.

Assume we want to apply *true_conj1_conv* only in the conclusion of the goal, and *if_true1_conv* should only apply to the premises. Here is a tactic doing exactly that:

```
fun true1_tac ctxt =
  CONVERSION
    (Conv.params_conv ~1 (fn ctxt =>
       (Conv.prems_conv ~1 (if_true1_conv ctxt) then_conv
          Conv.concl_conv ~1 (true_conj1_conv ctxt))) ctxt)
```

We call the conversions with the argument *~1*. By this we analyse all parameters, all premises and the conclusion of a goal state. If we call *concl_conv* with a non-negative number, say *n*, then this conversions will skip the first *n* premises and applies the conversion to the "rest" (similar for parameters and conclusions). To test the tactic, consider the proof

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

where the tactic yields the goal state

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

As you can see, the premises are rewritten according to *if_true1_conv*, while the conclusion according to *true_conj1_conv*. If we only have one conversion that should be uniformly applied to the whole goal state, we can simplify *true1_tac* using *top_conv*.

Conversions are also be helpful for constructing meta-equality theorems. Such theorems are often definitions. As an example consider the following two ways of defining the identity function in Isabelle.

```
definition id1::"'a ⇒ 'a"
where "id1 x ≡ x"

definition id2::"'a ⇒ 'a"
where "id2 ≡ λx. x"
```

Although both definitions define the same function, the theorems *id1_def* and *id2_def* are different meta-equations. However it is easy to transform one into the other. The function *abs_def* from the structure *Drule* can automatically transform theorem *id1_def* into *id2_def* by abstracting all variables on the left-hand side in the right-hand side.

```
Drule.abs_def @{thm id1_def}
> id1 ≡ λx. x
```

Unfortunately, Isabelle has no built-in function that transforms the theorems in the other direction. We can implement one using conversions. The feature of conversions we are using is that if we apply a `cterm` to a conversion we obtain a meta-equality theorem (recall that the type of conversions is an abbreviation for `cterm -> thm`). The code of the transformation is below.

```
1  fun unabs_def ctxt def =
2  let
3    val (lhs, rhs) = Thm.dest_equals (cprop_of def)
4    val xs = strip_abs_vars (term_of rhs)
5    val (_, ctxt') = Variable.add_fixes (map fst xs) ctxt
6
7    val thy = Proof_Context.theory_of ctxt'
8    val cxs = map (cterm_of thy o Free) xs
9    val new_lhs = Drule.list_comb (lhs, cxs)
10
11   fun get_conv [] = Conv.rewr_conv def
12     | get_conv (_::xs) = Conv.fun_conv (get_conv xs)
13 in
14   get_conv xs new_lhs |>
15   singleton (Proof_Context.export ctxt' ctxt)
16 end
```

In Line 3 we destruct the meta-equality into the `cterm`s corresponding to the left-hand and right-hand side of the meta-equality. The assumption in `unabs_def` is that the right-hand side is an abstraction. We compute the possibly empty list of abstracted variables in Line 4 using the function `strip_abs_vars`. For this we have to first transform the `cterm` into a `term`. In Line 5 we import these variables into the context `ctxt'`, in order to export them again in Line 15. In Line 8 we certify the list of variables, which in turn we apply to the `lhs` of the definition using the function `list_comb`. In Line 11 and 12 we generate the conversion according to the length of the list of (abstracted) variables. If there are none, then we just return the conversion corresponding to the original definition. If there are variables, then we have to prefix this conversion with `fun_conv`. Now in Line 14 we only have to apply the new left-hand side to the generated conversion and obtain the the theorem we want to construct. As mentioned above, in Line 15 we only have to export the context `ctxt'` back to `ctxt` in order to produce meta-variables for the theorem. An example is as follows.

```
unabs_def @{context} @{thm id2_def}
> id2 ?x1 ≡ ?x1
```

To sum up this section, conversions are more general than the simplifier or simprocs, but you have to do more work yourself. Also conversions are often much less efficient

than the simplifier. The advantage of conversions, however, is that they provide much less room for non-termination.

Exercise 6.6.1: *Write a tactic that does the same as the simproc in exercise 6.5.1, but is based on conversions. You can make the same assumptions as in 6.5.1.*

Exercise 6.6.2: *Compare your solutions of Exercises 6.5.1 and 6.6.1, and try to determine which way of rewriting such terms is faster. For this you might have to construct quite large terms. Also see Recipe A.3 for information about timing.*

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

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

6.7 Declarations (TBD)

6.8 Structured Proofs (TBD)

TBD

```
lemma True
proof

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

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

ML {*
  val ctxt0 = @{context};
  val ctxt = ctxt0;
  val (_, ctxt) = Variable.add_fixes ["A", "B", "C"] ctxt;
  val ([r], ctxt) = Assumption.add_assumes [@{cprop "A & B ⟹ C"}] ctxt
  val (this, ctxt) = Assumption.add_assumes [@{cprop "A"}, @{cprop "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]
*}
```

6.9 Summary

Coding Conventions / Rules of Thumb

- *Reference theorems with the antiquotation* `@{thm ...}`.
- *If a tactic is supposed to fail, return the empty sequence.*
- *If you apply a tactic to a sequence of subgoals, apply it in reverse order (i.e. start with the last subgoal).*
- *Use simpsets that are as small as possible.*

Chapter 7

How to Write a Definitional Package

"We will most likely never realize the full importance of painting the Tower, that it is the essential element in the conservation of metal works and the more meticulous the paint job, the longer the tower shall endure."

Gustave Eiffel, in his book *The 300-Meter Tower*.[1]

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

As the running example we have chosen a rather simple package for defining inductive predicates. To keep things really simple, we will not use the general Knaster-Tarski fixpoint theorem on complete lattices, which forms the basis of Isabelle/HOL's standard inductive definition package. Instead, we will describe a simpler *impredicative* (i.e. involving quantification on predicate variables) encoding of inductive predicates. Due to its simplicity, this package will necessarily have a reduced functionality. It does neither support introduction rules involving arbitrary monotonic operators, nor does it prove case analysis rules (also called inversion rules). Moreover, it only proves a weaker form of the induction principle for inductive predicates. But it illustrates the implementation pf a typical package in Isabelle.

[1] The Eiffel Tower has been re-painted 18 times since its initial construction, an average of once every seven years. It takes more than one year for a team of 25 painters to paint the tower from top to bottom.

7.1 Preliminaries

The user will just give a specification of inductive predicate(s) and expects from the package to produce a convenient reasoning infrastructure. This infrastructure needs to be derived from the definition that correspond to the specified predicate(s). Before we start with explaining all parts of the package, let us first give some examples showing how to define inductive predicates and then also how to generate a reasoning infrastructure for them. From the examples we will figure out a general method for defining inductive predicates. This is usually the first step in writing a package for Isabelle. The aim in this section is *not* to write proofs that are as beautiful as possible, but as close as possible to the ML-implementation we will develop in later sections.

We first consider the transitive closure of a relation *R*. The "pencil-and-paper" specification for the transitive closure is:

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

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

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

Note we have to use the object implication $\longrightarrow$ and object quantification $\forall$ for stating this definition (there is no other way for definitions in HOL). However, the introduction rules and induction principles associated with the transitive closure should use the meta-connectives, since they simplify the reasoning for the user.

With this definition, the proof of the induction principle for *trcl* is almost immediate. It suffices to convert all the meta-level connectives in the lemma to object-level connectives using the proof method *atomize* (Line 4 below), expand the definition of *trcl* (Line 5 and 6), eliminate the universal quantifier contained in it (Line 7), and then solve the goal by *assumption* (Line 8).

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

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

```
1 lemma trcl_base:
```

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

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

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

The two assumptions come from the definition of *trcl* and correspond to the introduction rules. Thus, all we have to do is to eliminate the universal quantifier in front of the first assumption (Line 5), and then solve the goal by *assumption* (Line 6).

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

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

we obtain the goal state

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

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

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

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

apply *(rule r2[rule_format])*

This gives us two new subgoals

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

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

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

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

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

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

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

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

Since the predicates *even* and *odd* are mutually inductive, each corresponding definition must quantify over both predicates (we name them below *P* and *Q*).

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

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

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

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

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

```
1  lemma evenS:
2  shows "odd m ⟹ even (Suc m)"
3  apply (unfold odd_def even_def)
4  apply (rule allI impI)+
5  proof -
6    case (goal1 P Q)
7    have p1: "∀P Q. P 0 ⟶ (∀m. Q m ⟶ P (Suc m))
8                              ⟶ (∀m. P m ⟶ Q (Suc m)) ⟶ Q m" by fact
9    have r1: "P 0" by fact
10   have r2: "∀m. Q m ⟶ P (Suc m)" by fact
11   have r3: "∀m. P m ⟶ Q (Suc m)" by fact
12   show "P (Suc m)"
13     apply(rule r2[rule_format])
14     apply(rule p1[THEN spec[where x=P], THEN spec[where x=Q],
15             THEN mp, THEN mp, THEN mp, OF r1, OF r2, OF r3])
16     done
17 qed
```

The interesting lines are 7 to 15. Again, the assumptions fall into two categories: `p1` corresponds to the premise of the introduction rule; `r1` to `r3` come from the definition of `even`. In Line 13, we apply the assumption `r2` (since we prove the second introduction rule). In Lines 14 and 15 we apply assumption `p1` (if the second introduction rule had more premises we have to do that for all of them). In order for this assumption to be applicable, the quantifiers need to be instantiated and then also the implications need to be resolved with the other rules.

Next we define the accessible part of a relation `R` given by the single rule:

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

The interesting point of this definition is that it contains a quantification that ranges only over the premise and the premise has also a precondition. The definition of `accpart` is:

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

The proof of the induction principle is again straightforward and omitted. Proving the introduction rule is a little more complicated, because the quantifier and the implication in the premise. The proof is as follows.

```
1  lemma accpartI:
2  shows "(⋀y. R y x ⟹ accpart R y) ⟹ accpart R x"
3  apply (unfold accpart_def)
4  apply (rule allI impI)+
5  proof -
6    case (goal1 P)
7    have p1: "⋀y. R y x ⟹
8                    (∀P. (∀x. (∀y. R y x ⟶ P y) ⟶ P x) ⟶ P y)" by fact
9    have r1: "∀x. (∀y. R y x ⟶ P y) ⟶ P x" by fact
10   show "P x"
11     apply(rule r1[rule_format])
12     proof -
13       case (goal1 y)
14       have r1_prem: "R y x" by fact
15       show "P y"
16   apply(rule p1[OF r1_prem, THEN spec[where x=P], THEN mp, OF r1])
17       done
18   qed
19 qed
```

As you can see, there are now two subproofs. The assumptions fall as usual into two categories (Lines 7 to 9). In Line 11, applying the assumption *r1* generates a goal state with the new local assumption *R y x*, named *r1_prem* in the second subproof (Line 14). This local assumption is used to solve the goal *P y* with the help of assumption *p1*.

Exercise 7.1.1: *Give the definition for the freshness predicate for lambda-terms. The rules for this predicate are:*

$$\frac{a \neq b}{fresh\ a\ (Var\ b)} \qquad \frac{fresh\ a\ t \qquad fresh\ a\ s}{fresh\ a\ (App\ t\ s)}$$

$$\frac{}{fresh\ a\ (Lam\ a\ t)} \qquad \frac{a \neq b \qquad fresh\ a\ t}{fresh\ a\ (Lam\ b\ t)}$$

From the definition derive the induction principle and the introduction rules.

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

7.2 Parsing and Typing the Specification

To be able to write down the specifications of inductive predicates, we have to introduce a new command (see Section 5.8). As the keyword for the new command

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

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

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

simple_inductive
  fresh :: "string ⇒ trm ⇒ bool"
where
  fresh_var: "a≠b ⟹ fresh a (Var b)"
| fresh_app: "⟦fresh a t; fresh a s⟧ ⟹ fresh a (App t s)"
| fresh_lam1: "fresh a (Lam a t)"
| fresh_lam2: "⟦a≠b; fresh a t⟧ ⟹ fresh a (Lam b t)"
```

Figure 7.1: Specification given by the user for the inductive predicates *trcl*, *even* and *odd*, *accpart* and *fresh*.

we chose **simple_inductive**. Examples of specifications from the previous section are shown in Figure 7.1. The syntax used in these examples more or less translates directly into the parser:

```
val spec_parser =
   Parse.fixes --
   Scan.optional
     (Parse.$$$ "where" |--
        Parse.!!!
          (Parse.enum1 "|"
             (Parse_Spec.opt_thm_name ":" -- Parse.prop))) []
```

which we explained in Section 5.7. There is no code included for parsing the keyword and what is called a *target*. The latter can be given optionally after the keyword. The target is an "advanced" feature which we will inherit for "free" from the infrastructure on which we shall build the package. The target stands for a locale and allows us to specify

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

and then define the transitive closure and the accessible part of this locale as follows:

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

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

Note that in these definitions the parameter `R`, standing for the relation, is left implicit. For the moment we will ignore this kind of implicit parameters and rely on the fact that the infrastructure will deal with them. Later, however, we will come back to them.

If we feed into the parser the string that corresponds to our definition of `even` and `odd`

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

then we get back the specifications of the predicates (with type and syntax annotations), and specifications of the introduction rules. This is all the information we need for calling the package and setting up the keyword. The latter is done in Lines 5 to 7 in the code below.

```
1 val specification : (local_theory -> local_theory) parser =
2   spec_parser >>
3     (fn (pred_specs, rule_specs) => add_inductive_cmd pred_specs rule_specs)
4
5 val _ = Outer_Syntax.local_theory @{command_spec "simple_inductive2"}
6           "definition of simple inductive predicates"
7              specification
```

We call `local_theory` with the kind-indicator `thy_decl` since the package does not need to open up any proof (see Section 5.8). The auxiliary function `specification`

in Lines 1 to 3 gathers the information from the parser to be processed further by the function `add_inductive_cmd`, which we describe below.

Note that the predicates when they come out of the parser are just some "naked" strings: they have no type yet (even if we annotate them with types) and they are also not defined constants yet (which the predicates eventually will be). Also the introduction rules are just strings. What we have to do first is to transform the parser's output into some internal datastructures that can be processed further. For this we can use the function `read_spec`. This function takes some strings (with possible typing annotations) and some rule specifications, and attempts to find a typing according to the given type constraints given by the user and the type constraints by the "ambient" theory. It returns the type for the predicates and also returns typed terms for the introduction rules. So at the heart of the function `add_inductive_cmd` is a call to `read_spec`.

```
fun add_inductive_cmd pred_specs rule_specs lthy =
let
  val ((pred_specs', rule_specs'), _) =
        Specification.read_spec pred_specs rule_specs lthy
in
  add_inductive pred_specs' rule_specs' lthy
end
```

Once we have the input data as some internal datastructure, we call the function `add_inductive`. This function does the heavy duty lifting in the package: it generates definitions for the predicates and derives from them corresponding induction principles and introduction rules. The description of this function will span over the next two sections.

7.3 The Code in a Nutshell

The inductive package will generate the reasoning infrastructure for mutually recursive predicates, say $pred_1 \ldots pred_n$. In what follows we will have the convention that various, possibly empty collections of "things" (lists, terms, nested implications and so on) are indicated either by adding an `"s"` or by adding a superscript $"^*"$. The shorthand for the predicates will therefore be `preds` or $pred^*$. In the case of the predicates there must be, of course, at least a single one in order to obtain a meaningful definition.

The input for the inductive package will be some `preds` with possible typing and syntax annotations, and also some introduction rules. We call below the introduction rules short as `rules`. Borrowing some idealised Isabelle notation, one such `rule` is assumed to be of the form

$$rule ::= \bigwedge xs.\ \underbrace{As}_{\text{non-recursive premises}} \Longrightarrow \underbrace{(\bigwedge ys.\ Bs \Longrightarrow pred\ ss)^*}_{\text{recursive premises}} \Longrightarrow pred\ ts$$

For the purposes here, we will assume the `rules` have this format and omit any code that actually tests this. Therefore "things" can go horribly wrong, if the `rules`

are not of this form. The *As* and *Bs* in a *rule* stand for formulae not involving the inductive predicates *preds*; the instances *pred ss* and *pred ts* can stand for different predicates, like $pred_1$ *ss* and $pred_2$ *ts*, in case mutual recursive predicates are defined; the terms *ss* and *ts* are the arguments of these predicates. Every formula left of "$\Longrightarrow$ *pred ts*" is a premise of the rule. The outermost quantified variables *xs* are usually omitted in the user's input. The quantification for the variables *ys* is local with respect to one recursive premise and must be given. Some examples of *rules* are

```
a ≠ b ⟹ fresh a (Var b)
```

which has only a single non-recursive premise, whereas

```
odd n ⟹ even (Suc n)
```

has a single recursive premise; the rule

```
(⋀y. R y x ⟹ accpart R y) ⟹ accpart R x
```

has a single recursive premise that has a precondition. As is custom all rules are stated without the leading meta-quantification $\bigwedge xs$.

The output of the inductive package will be definitions for the predicates, induction principles and introduction rules. For the definitions we need to have the *rules* in a form where the meta-quantifiers and meta-implications are replaced by their object logic equivalents. Therefore an *orule* is of the form

```
orule ::= ∀xs. As ⟶ (∀ys. Bs ⟶ pred ss)* ⟶ pred ts
```

A definition for the predicate *pred* has then the form

```
def ::= pred ≡ λzs. ∀preds. orules ⟶ pred zs
```

The induction principles for every predicate *pred* are of the form

```
ind ::= pred ?zs ⟹ rules[preds := ?Ps] ⟹ ?P ?zs
```

where in the *rules*-part every *pred* is replaced by a fresh schematic variable *?P*.

In order to derive an induction principle for the predicate *pred*, we first transform *ind* into the object logic and fix the schematic variables. Hence we have to prove a formula of the form

```
pred zs ⟶ orules[preds := Ps] ⟶ P zs
```

If we assume *pred zs* and unfold its definition, then we have an assumption

```
∀preds. orules ⟶ pred zs
```

and must prove the goal

```
orules[preds := Ps] ⟶ P zs
```

This can be done by instantiating the $\forall$ *preds*-quantification with the *Ps*. Then we are done since we are left with a simple identity.

Although the user declares the introduction rules *rules*, they must also be derived from the *defs*. These derivations are a bit involved. Assuming we want to prove the introduction rule

```
⋀xs. As ⟹ (⋀ys. Bs ⟹ pred ss)* ⟹ pred ts
```

then we have assumptions of the form

```
(i)  As
(ii) (⋀ys. Bs ⟹ pred ss)*
```

and must show the goal

```
pred ts
```

If we now unfold the definitions for the *preds*, we have assumptions

```
(i)   As
(ii)  (⋀ys. Bs ⟹ ∀preds. orules ⟶ pred ss)*
(iii) orules
```

and need to show

```
pred ts
```

In the last step we removed some quantifiers and moved the precondition *orules* into the assumption. The *orules* stand for all introduction rules that are given by the user. We apply the *orule* that corresponds to introduction rule we are proving. After transforming the object connectives into meta-connectives, this introduction rule must necessarily be of the form

```
As ⟹ (⋀ys. Bs ⟹ pred ss)* ⟹ pred ts
```

When we apply this rule we end up in the goal state where we have to prove goals of the form

```
(a) As
(b) (⋀ys. Bs ⟹ pred ss)*
```

We can immediately discharge the goals *As* using the assumptions in *(i)*. The goals in *(b)* can be discharged as follows: we assume the *Bs* and prove *pred ss*. For this we resolve the *Bs* with the assumptions in *(ii)*. This gives us the assumptions

$(\forall preds.\ orules \longrightarrow pred\ ss)^*$

Instantiating the universal quantifiers and then resolving with the assumptions in *(iii)* gives us $pred\ ss$, which is the goal we are after. This completes the proof for introduction rules.

What remains is to implement in Isabelle the reasoning outlined in this section. We will describe the code in the next section. For building testcases, we use the shorthands for *even/odd*, *fresh* and *accpart* defined in Figure 7.2.

7.4 The Gory Details

As mentioned before the code falls roughly into three parts: the code that deals with the definitions, with the induction principles and with the introduction rules. In addition there are some administrative functions that string everything together.

Definitions

We first have to produce for each predicate the user specifies an appropriate definition, whose general form is

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

and then "register" the definition inside a local theory. To do the latter, we use the following wrapper for the function *define*. The wrapper takes a predicate name, a syntax annotation and a term representing the right-hand side of the definition.

```
1 fun make_defn ((predname, mx), trm) lthy =
2 let
3   val arg = ((predname, mx), (Attrib.empty_binding, trm))
4   val ((_, (_ , thm)), lthy') = Local_Theory.define arg lthy
5 in
6   (thm, lthy')
7 end
```

It returns the definition (as a theorem) and the local theory in which the definition has been made. We use *empty_binding* in Line 3, since the definitions for our inductive predicates are not meant to be seen by the user and therefore do not need to have any theorem attributes.

The next two functions construct the right-hand sides of the definitions, which are terms whose general form is:

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

When constructing these terms, the variables *zs* need to be chosen so that they do not occur in the *orules* and also be distinct from the *preds*.

```
(* even-odd example *)
val eo_defs = [@{thm even_def}, @{thm odd_def}]

val eo_rules =
  [@{prop "even 0"},
   @{prop "⋀n. odd n ⟹ even (Suc n)"},
   @{prop "⋀n. even n ⟹ odd (Suc n)"}]

val eo_orules =
  [@{prop "even 0"},
   @{prop "∀n. odd n ⟶ even (Suc n)"},
   @{prop "∀n. even n ⟶ odd (Suc n)"}]

val eo_preds =  [@{term "even::nat ⇒ bool"}, @{term "odd::nat ⇒ bool"}]
val eo_prednames = [@{binding "even"}, @{binding "odd"}]
val eo_mxs = [NoSyn, NoSyn]
val eo_arg_tyss = [[@{typ "nat"}], [@{typ "nat"}]]
val e_pred = @{term "even::nat ⇒ bool"}
val e_arg_tys = [@{typ "nat"}]



(* freshness example *)
val fresh_rules =
  [@{prop "⋀a b. a ≠ b ⟹ fresh a (Var b)"},
   @{prop "⋀a s t. fresh a t ⟹ fresh a s ⟹ fresh a (App t s)"},
   @{prop "⋀a t. fresh a (Lam a t)"},
   @{prop "⋀a b t. a ≠ b ⟹ fresh a t ⟹ fresh a (Lam b t)"}]

val fresh_orules =
  [@{prop "∀a b. a ≠ b ⟶ fresh a (Var b)"},
   @{prop "∀a s t. fresh a t ⟶ fresh a s ⟶ fresh a (App t s)"},
   @{prop "∀a t. fresh a (Lam a t)"},
   @{prop "∀a b t. a ≠ b ⟶ fresh a t ⟶ fresh a (Lam b t)"}]

val fresh_pred =  @{term "fresh::string ⇒ trm ⇒ bool"}
val fresh_arg_tys = [@{typ "string"}, @{typ "trm"}]



(* accessible-part example *)
val acc_rules =
     [@{prop "⋀R x. (⋀y. R y x ⟹ accpart R y) ⟹ accpart R x"}]
val acc_pred = @{term "accpart::('a ⇒ 'a ⇒ bool) ⇒'a ⇒ bool"}
```

Figure 7.2: Shorthands for the inductive predicates *even/odd*, *fresh* and *accpart*. The names of these shorthands follow the convention *rules*, *orules*, *preds* and so on. The purpose of these shorthands is to simplify the construction of testcases in Section 7.4.

The first function, named *defn_aux*, constructs the term for one particular predicate (the argument *pred* in the code below). The number of arguments of this predicate is determined by the number of argument types given in *arg_tys*. The other arguments of the function are the *orules* and all the *preds*.

```
1  fun defn_aux lthy orules preds (pred, arg_tys) =
2  let
3    fun mk_all x P = HOLogic.all_const (fastype_of x) $ lambda x P
4
5    val fresh_args =
6          arg_tys
7          |> map (pair "z")
8          |> Variable.variant_frees lthy (preds @ orules)
9          |> map Free
10 in
11   list_comb (pred, fresh_args)
12   |> fold_rev (curry HOLogic.mk_imp) orules
13   |> fold_rev mk_all preds
14   |> fold_rev lambda fresh_args
15 end
```

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

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

```
local_setup {* fn lthy =>
let
  val def = defn_aux lthy eo_orules eo_preds (e_pred, e_arg_tys)
in
  pwriteln (pretty_term lthy def); lthy
end *}
```

where we use the shorthands defined in Figure 7.2. The testcase calls *defn_aux* for the predicate *even* and prints out the generated definition. So we obtain as printout

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

If we try out the function with the rules for freshness

```
local_setup {* fn lthy =>
let
```

```
  val arg = (fresh_pred, fresh_arg_tys)
  val def = defn_aux lthy fresh_orules [fresh_pred] arg
in
  pwriteln (pretty_term lthy def); lthy
end *}
```

we obtain

```
λz za.
   ∀fresh.
      (∀a b. a ≠ b ⟶ fresh a (Var b)) ⟶
      (∀a s t. fresh a t ⟶ fresh a s ⟶ fresh a (App t s)) ⟶
      (∀a t. fresh a (Lam a t)) ⟶
      (∀a b t. a ≠ b ⟶ fresh a t ⟶ fresh a (Lam b t)) ⟶ fresh z za
```

The second function, named *defns*, has to iterate the function *defn_aux* over all predicates. The argument *preds* is again the list of predicates as *terms*; the argument *prednames* is the list of binding names of the predicates; *mxs* are the list of syntax, or mixfix, annotations for the predicates; *arg_tyss* is the list of argument-type-lists.

```
1 fun defns rules preds prednames mxs arg_typss lthy =
2 let
3   val thy = Proof_Context.theory_of lthy
4   val orules = map (Object_Logic.atomize_term thy) rules
5   val defs = map (defn_aux lthy orules preds) (preds ~~ arg_typss)
6 in
7   fold_map make_defn (prednames ~~ mxs ~~ defs) lthy
8 end
```

The user will state the introduction rules using meta-implications and meta-quantifications. In Line 4, we transform these introduction rules into the object logic (since definitions cannot be stated with meta-connectives). To do this transformation we have to obtain the theory behind the local theory using the function *theory_of* (Line 3); with this theory we can use the function *atomize_term* to make the transformation (Line 4). The call to *defn_aux* in Line 5 produces all right-hand sides of the definitions. The actual definitions are then made in Line 7. The result of the function is a list of theorems and a local theory (the theorems are registered with the local theory). A testcase for this function is

```
local_setup {* fn lthy =>
let
  val (defs, lthy') =
    defns eo_rules eo_preds eo_prednames eo_mxs eo_arg_tyss lthy
in
  pwriteln (pretty_thms_no_vars lthy' defs); lthy
end *}
```

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

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

Note that in the testcase we return the local theory `lthy` (not the modified `lthy'`). As a result the test case has no effect on the ambient theory. The reason is that if we introduce the definition again, we pollute the name space with two versions of `even` and `odd`. We want to avoid this here.

This completes the code for introducing the definitions. Next we deal with the induction principles.

Induction Principles

Recall that the manual proof for the induction principle of `even` was:

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

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

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

```
fun inst_spec ctrm =
let
  val cty = ctyp_of_term ctrm
in
  Drule.instantiate' [SOME cty] [NONE, SOME ctrm] @{thm spec}
end
```

This helper function uses the function `instantiate'` and instantiates the `?x` in the theorem $\forall x.\ ?P\ x \Longrightarrow ?P\ ?x$ with a given `cterm`. We call this helper function in the following tactic..

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

This tactic expects a list of `cterm`s. It allows us in the proof below to instantiate the three quantifiers in the assumption.

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

We obtain the goal state
goal (1 subgoal):
1. P a b c ⟹ True

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

```
1 fun ind_tac ctxt defs prem insts =
2   EVERY1 [Object_Logic.full_atomize_tac ctxt,
3           cut_facts_tac prem,
4           rewrite_goal_tac ctxt defs,
5           inst_spec_tac insts,
6           assume_tac]
```

We have to give it as arguments the definitions, the premise (a list of formulae) and the instantiations. The premise is *even n* in lemma *manual_ind_prin_even* shown above; in our code it will always be a list consisting of a single formula. Compare this tactic with the manual proof for the lemma *manual_ind_prin_even*: as you can see there is almost a one-to-one correspondence between the **apply**-script and the *ind_tac*. We first rewrite the goal to use only object connectives (Line 2), "cut in" the premise (Line 3), unfold the definitions (Line 4), instantiate the assumptions of the goal (Line 5) and then conclude with *assume_tac*.

Two testcases for this tactic are:

lemma *automatic_ind_prin_even:*
assumes *prem: "even z"*
shows *"P 0 ⟹ (⋀m. Q m ⟹ P (Suc m)) ⟹ (⋀m. P m ⟹ Q (Suc m)) ⟹ P z"*
by *(tactic {* ind_tac @{context} eo_defs @{thms prem}*}*
*[@{cterm "P::nat⇒bool"}, @{cterm "Q::nat⇒bool"}] *})*

lemma *automatic_ind_prin_fresh:*
assumes *prem: "fresh z za"*
shows *"(⋀a b. a ≠ b ⟹ P a (Var b)) ⟹*
(⋀a t s. ⟦P a t; P a s⟧ ⟹ P a (App t s)) ⟹
(⋀a t. P a (Lam a t)) ⟹
(⋀a b t. ⟦a ≠ b; P a t⟧ ⟹ P a (Lam b t)) ⟹ P z za"
by *(tactic {* ind_tac @{context} @{thms fresh_def} @{thms prem}*}*
*[@{cterm "P::string⇒trm⇒bool"}] *})*

While the tactic for proving the induction principles is relatively simple, it will be a bit more work to construct the goals from the introduction rules the user provides. Therefore let us have a closer look at the first proved theorem:

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

The variables *z*, *P* and *Q* are schematic variables (since they are not quantified in the lemma). These variables must be schematic, otherwise they cannot be instantiated by the user. To generate these schematic variables we use a common trick in Isabelle programming: we first declare them as *free*, *but fixed*, and then use the infrastructure to turn them into schematic variables.

In general we have to construct for each predicate *pred* a goal of the form

pred ?zs $\Longrightarrow$ *rules[preds := ?Ps]* $\Longrightarrow$ *?P ?zs*

where the predicates *preds* are replaced in *rules* by new distinct variables *?Ps*. We also need to generate fresh arguments *?zs* for the predicate *pred* and the *?P* in the conclusion.

We generate these goals in two steps. The first function, named *prove_ind*, expects that the introduction rules are already appropriately substituted. The argument *srules* stands for these substituted rules; *cnewpreds* are the certified terms coresponding to the variables *?Ps*; *pred* is the predicate for which we prove the induction principle; *newpred* is its replacement and *arg_tys* are the argument types of this predicate.

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

In Line 3 we produce names *zs* for each type in the argument type list. Line 4 makes these names unique and declares them as free, but fixed, variables in the local theory *lthy'*. That means they are not schematic variables (yet). In Line 5 we construct the terms corresponding to these variables. The variables are applied to the predicate in Line 7 (this corresponds to the first premise *pred zs* of the induction principle). In Line 8 and 9, we first construct the term *P zs* and then add the (substituted) introduction rules as preconditions. In case that no introduction rules are given, the conclusion of this implication needs to be wrapped inside a *Trueprop*, otherwise the Isabelle's goal mechanism will fail.[2]

In Line 11 we set up the goal to be proved using the function *prove*; in the next line we call the tactic for proving the induction principle. As mentioned before, this tactic expects the definitions, the premise and the (certified) predicates with which

[2] FIXME: check with Stefan...is this so?

the introduction rules have been substituted. The code in these two lines will return a theorem. However, it is a theorem proved inside the local theory `lthy'`, where the variables `zs` are free, but fixed (see Line 4). By exporting this theorem from `lthy'` (which contains the `zs` as free variables) to `lthy` (which does not), we obtain the desired schematic variables `?zs`. A testcase for this function is

```
local_setup {* fn lthy =>
let
  val newpreds = [@{term "P::nat ⇒ bool"}, @{term "Q::nat ⇒ bool"}]
  val cnewpreds = [@{cterm "P::nat ⇒ bool"}, @{cterm "Q::nat ⇒ bool"}]
  val newpred = @{term "P::nat ⇒ bool"}
  val srules =  map (subst_free (eo_preds ~~ newpreds)) eo_rules
  val intro =
      prove_ind lthy eo_defs srules cnewpreds ((e_pred, newpred), e_arg_tys)
in
  pwriteln (pretty_thm lthy intro); lthy
end *}
```

This prints out the theorem:

$$\llbracket even\ ?z;\ P\ 0;\ \bigwedge n.\ Q\ n \Longrightarrow P\ (Suc\ n);\ \bigwedge n.\ P\ n \Longrightarrow Q\ (Suc\ n) \rrbracket \Longrightarrow P\ ?z$$

The export from `lthy'` to `lthy` in Line 13 above has correctly turned the free, but fixed, `z` into a schematic variable `?z`; the variables `P` and `Q` are not yet schematic.

We still have to produce the new predicates with which the introduction rules are substituted and iterate `prove_ind` over all predicates. This is what the second function, named `inds` does.

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

In Line 3, we generate a string `"P"` for each predicate. In Line 4, we use the same trick as in the previous function, that is making the `Ps` fresh and declaring them as free, but fixed, in the new local theory `lthy'`. From the local theory we extract the

ambient theory in Line 6. We need this theory in order to certify the new predicates. In Line 8, we construct the types of these new predicates using the given argument types. Next we turn them into terms and subsequently certify them (Line 9 and 10). We can now produce the substituted introduction rules (Line 11) using the function *subst_free*. Line 14 and 15 just iterate the proofs for all predicates. From this we obtain a list of theorems. Finally we need to export the fixed variables *Ps* to obtain the schematic variables *?Ps* (Line 16).

A testcase for this function is

```
local_setup {* fn lthy =>
let
  val ind_thms = inds eo_rules eo_defs eo_preds eo_arg_tyss lthy
in
  pwriteln (pretty_thms lthy ind_thms); lthy
end *}
```

which prints out

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

Note that now both, the *?Ps* and the *?zs*, are schematic variables. The numbers attached to these variables have been introduced by the pretty-printer and are *not* important for the user.

This completes the code for the induction principles. The final peice of reasoning infrastructure we need are the introduction rules.

Introduction Rules

Constructing the goals for the introduction rules is easy: they are just the rules given by the user. However, their proofs are quite a bit more involved than the ones for the induction principles. To explain the general method, our running example will be the introduction rule

```
⋀a b t. ⟦a ≠ b; fresh a t⟧ ⟹ fresh a (Lam b t)
```

about freshness for lambdas. In order to ease somewhat our work here, we use the following two helper functions.

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

To see what these functions do, let us suppose we have the following three theorems.

lemma *all_elims_test:*

fixes *P::"nat ⇒ nat ⇒ nat ⇒ bool"*
shows *"∀x y z. P x y z"* **sorry**

lemma *imp_elims_test:*
shows *"A ⟶ B ⟶ C"* **sorry**

lemma *imp_elims_test':*
shows *"A" "B"* **sorry**

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

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

Note the difference with *inst_spec_tac* from Page 170: *inst_spec_tac* is a tactic which operates on a goal state; in contrast *all_elims* operates on theorems.

Similarly, the function *imp_elims* eliminates preconditions from implications. For example we can eliminate the preconditions *A* and *B* from *imp_elims_test*:

```
let
  val res = imp_elims @{thms imp_elims_test'} @{thm imp_elims_test}
in
  pwriteln (pretty_thm_no_vars @{context} res)
end
> C
```

Now we set up the proof for the introduction rule as follows:

lemma *fresh_Lam:*
shows *"⋀a b t. ⟦a ≠ b; fresh a t⟧ ⟹ fresh a (Lam b t)"*

The first step in the proof will be to expand the definitions of freshness and then introduce quantifiers and implications. For this we will use the tactic

```
1 fun expand_tac ctxt defs =
2   Object_Logic.rulify_tac ctxt 1
3   THEN rewrite_goal_tac ctxt defs 1
4   THEN (REPEAT (resolve_tac [@{thm allI}, @{thm impI}] 1))
```

The function in Line 2 "rulifies" the lemma.[3] This will turn out to be important later on. Applying this tactic in our proof of *fresh_Lem*

[3] FIXME: explain this better

apply`(tactic {* expand_tac @{context} @{thms fresh_def} *})`

gives us the goal state

```
goal (1 subgoal):
 1. ⋀a b t fresh.
       ⟦a ≠ b;
        ∀fresh.
           (∀a b. a ≠ b ⟶ fresh a (Var b)) ⟶
           (∀a t s. fresh a t ⟶ fresh a s ⟶ fresh a (App t s)) ⟶
           (∀a t. fresh a (Lam a t)) ⟶
           (∀a b t. a ≠ b ⟶ fresh a t ⟶ fresh a (Lam b t)) ⟶ fresh a t;
        ∀a b. a ≠ b ⟶ fresh a (Var b);
        ∀a t s. fresh a t ⟶ fresh a s ⟶ fresh a (App t s);
        ∀a t. fresh a (Lam a t);
        ∀a b t. a ≠ b ⟶ fresh a t ⟶ fresh a (Lam b t)⟧
       ⟹ fresh a (Lam b t)
```

As you can see, there are parameters (namely `a`, `b` and `t`) which come from the introduction rule and parameters (in the case above only *fresh*) which come from the universal quantification in the definition *fresh a (App t s)*. Similarly, there are assumptions that come from the premises of the rule (namely the first two) and assumptions from the definition of the predicate (assumption three to six). We need to treat these parameters and assumptions differently. In the code below we will therefore separate them into *params1* and *params2*, respectively *prems1* and *prems2*. To do this separation, it is best to open a subproof with the tactic *SUBPROOF*, since this tactic provides us with the parameters (as list of *cterms*) and the assumptions (as list of *thms*). The problem with *SUBPROOF*, however, is that it always expects us to completely discharge the goal (see Section 6.2). This is a bit inconvenient for our gradual explanation of the proof here. Therefore we use first the function *FOCUS*, which does s ame as *SUBPROOF* but does not require us to completely discharge the goal.

First we calculate the values for *params1/2* and *prems1/2* from *params* and *prems*, respectively. To better see what is going in our example, we will print out these values using the printing function in Figure 7.3. Since *FOCUS* will supply us the *params* and *prems* as lists, we can separate them using the function *chop*.

```
1 fun chop_test_tac preds rules =
2   Subgoal.FOCUS (fn {params, prems, context, ...} =>
3   let
4     val cparams = map snd params
5     val (params1, params2) = chop (length cparams - length preds) cparams
6     val (prems1, prems2) = chop (length prems - length rules) prems
7   in
8     chop_print params1 params2 prems1 prems2 context; all_tac
9   end)
```

For the separation we can rely on the fact that Isabelle deterministically produces parameters and premises in a goal state. The last parameters that were introduced

```
fun chop_print params1 params2 prems1 prems2 ctxt =
let
 val pps = [Pretty.big_list "Params1 from the rule:" (map (pretty_cterm
ctxt) params1),
            Pretty.big_list "Params2 from the predicate:" (map (pretty_cterm
ctxt) params2),
            Pretty.big_list "Prems1 from the rule:" (map (pretty_thm ctxt)
prems1),
            Pretty.big_list "Prems2 from the predicate:" (map (pretty_thm
ctxt) prems2)]
in
  pps |> Pretty.chunks
      |> Pretty.string_of
      |> tracing
end
```

Figure 7.3: A helper function that prints out the parameters and premises that need to be treated differently.

come from the quantifications in the definitions (see the tactic `expand_tac`). Therefore we only have to subtract in Line 5 the number of predicates (in this case only `1`) from the lenghts of all parameters. Similarly with the `prems` in line 6: the last premises in the goal state come from unfolding the definition of the predicate in the conclusion. So we can just subtract the number of rules from the number of all premises. To check our calculations we print them out in Line 8 using the function `chop_print` from Figure 7.3 and then just do nothing, that is `all_tac`. Applying this tactic in our example

apply`(tactic {* chop_test_tac [fresh_pred] fresh_rules @{context} 1 *})`

gives

```
Params1 from the rule:
a, b, t
Params2 from the predicate:
fresh
Prems1 from the rule:
a ≠ b
∀fresh.
(∀a b. a ≠ b ⟶ fresh a (Var b)) ⟶
(∀a t s. fresh a t ⟶ fresh a s ⟶ fresh a (App t s)) ⟶
(∀a t. fresh a (Lam a t)) ⟶
(∀a b t. a ≠ b ⟶ fresh a t ⟶ fresh a (Lam b t)) ⟶ fresh a t
Prems2 from the predicate:
∀a b. a ≠ b ⟶ fresh a (Var b)
∀a t s. fresh a t ⟶ fresh a s ⟶ fresh a (App t s)
∀a t. fresh a (Lam a t)
∀a b t. a ≠ b ⟶ fresh a t ⟶ fresh a (Lam b t)
```

We now have to select from `prems2` the premise that corresponds to the introduction rule we prove, namely:

```
∀a b t. a ≠ b ⟶ fresh a t ⟶ fresh a (Lam a t)
```

To use this premise with *rtac*, we need to instantiate its quantifiers (with *params1*) and transform it into rule format (using *rulify*). So we can modify the code as follows:

```
fun apply_prem_tac i preds rules =
  Subgoal.FOCUS (fn {params, prems, context, ...} =>
  let
    val cparams = map snd params
    val (params1, params2) = chop (length cparams - length preds) cparams
    val (prems1, prems2) = chop (length prems - length rules) prems
  in
    rtac (Object_Logic.rulify context (all_elims params1 (nth prems2 i))) 1
  end)
```

The argument *i* corresponds to the number of the introduction we want to prove. We will later on let it range from *0* to the number of *rules - 1*. Below we apply this function with *3*, since we are proving the fourth introduction rule.

apply*(tactic {* apply_prem_tac 3 [fresh_pred] fresh_rules @{context} 1 *})*

The goal state we obtain is:

```
1. ... ⟹ a ≠ b
2. ... ⟹ fresh a t
```

As expected there are two subgoals, where the first comes from the non-recursive premise of the introduction rule and the second comes from the recursive one. The first goal can be solved immediately by *prems1*. The second needs more work. It can be solved with the other premise in *prems1*, namely

```
∀fresh.
   (∀a b. a ≠ b ⟶ fresh a (Var b)) ⟶
   (∀a t s. fresh a t ⟶ fresh a s ⟶ fresh a (App t s)) ⟶
   (∀a t. fresh a (Lam a t)) ⟶
   (∀a b t. a ≠ b ⟶ fresh a t ⟶ fresh a (Lam b t)) ⟶ fresh a t
```

but we have to instantiate it appropriately. These instantiations come from *params1* and *prems2*. We can determine whether we are in the simple or complicated case by checking whether the topmost connective is an ∀. The premises in the simple case cannot have such a quantification, since the first step of *expand_tac* was to "rulify" the lemma. The premise of the complicated case must have at least one ∀ coming from the quantification over the *preds*. So we can implement the following function

```
fun prepare_prem params2 prems2 prem =
  rtac (case prop_of prem of
           _ $ (Const (@{const_name All}, _) $ _) =>
                 prem |> all_elims params2
                      |> imp_elims prems2
         | _ => prem)
```

which either applies the premise outright (the default case) or if it has an outermost universial quantification, instantiates it first with *params1* and then *prems1*. The following tactic will therefore prove the lemma completely.

```
fun prove_intro_tac i preds rules =
  SUBPROOF (fn {params, prems, context, ...} =>
  let
    val cparams = map snd params
    val (params1, params2) = chop (length cparams - length preds) cparams
    val (prems1, prems2) = chop (length prems - length rules) prems
  in
    rtac (Object_Logic.rulify context (all_elims params1 (nth prems2 i))) 1
    THEN EVERY1 (map (prepare_prem params2 prems2) prems1)
  end)
```

Note that the tactic is now *SUBPROOF*, not *FOCUS* anymore. The full proof of the introduction rule is as follows:

lemma *fresh_Lam:*
shows *"⋀a b t. ⟦a ≠ b; fresh a t⟧ ⟹ fresh a (Lam b t)"*
apply*(tactic {* expand_tac @{context} @{thms fresh_def} *})*
apply*(tactic {* prove_intro_tac 3 [fresh_pred] fresh_rules @{context} 1 *})*
done

Phew!...

Unfortunately, not everything is done yet. If you look closely at the general principle outlined for the introduction rules in Section 7.3, we have not yet dealt with the case where recursive premises have preconditions. The introduction rule of the accessible part is such a rule.

lemma *accpartI:*
shows *"⋀R x. (⋀y. R y x ⟹ accpart R y) ⟹ accpart R x"*
apply*(tactic {* expand_tac @{context} @{thms accpart_def} *})*
apply*(tactic {* chop_test_tac [acc_pred] acc_rules @{context} 1 *})*
apply*(tactic {* apply_prem_tac 0 [acc_pred] acc_rules @{context} 1 *})*

Here *chop_test_tac* prints out the following values for *params1/2* and *prems1/2*

```
Params1 from the rule:
x
Params2 from the predicate:
P
Prems1 from the rule:
R ?y x ⟹ ∀P. (∀x. (∀y. R y x ⟶ P y) ⟶ P x) ⟶ P ?y
Prems2 from the predicate:
∀x. (∀y. R y x ⟶ P y) ⟶ P x
```

and after application of the introduction rule using *apply_prem_tac*, we are in the goal state

```
1. ⋀y. R y x ⟹ P y
```

In order to make progress, we have to use the precondition `R y x` (in general there can be many of them). The best way to get a handle on these preconditions is to open up another subproof, since the preconditions will then be bound to `prems`. Therfore we modify the function `prepare_prem` as follows

```
1  fun prepare_prem params2 prems2 ctxt prem =
2    SUBPROOF (fn {prems, ...} =>
3    let
4      val prem' = prems MRS prem
5    in
6      rtac (case prop_of prem' of
7             _ $ (Const (@{const_name All}, _) $ _) =>
8                   prem' |> all_elims params2
9                         |> imp_elims prems2
10          | _ => prem') 1
11   end) ctxt
```

In Line 4 we use the `prems` from the `SUBPROOF` and resolve them with `prem`. In the simple cases, that is where the `prem` comes from a non-recursive premise of the rule, `prems` will be just the empty list and the function `MRS` does nothing. Similarly, in the cases where the recursive premises of the rule do not have preconditions. In case there are preconditions, then Line 4 discharges them. After that we can proceed as before, i.e., check whether the outermost connective is $\forall$.

The function `prove_intro_tac` only needs to be changed so that it gives the context to `prepare_prem` (Line 8). The modified version is below.

```
1  fun prove_intro_tac i preds rules =
2    SUBPROOF (fn {params, prems, context, ...} =>
3    let
4      val cparams = map snd params
5      val (params1, params2) = chop (length cparams - length preds) cparams
6      val (prems1, prems2) = chop (length prems - length rules) prems
7    in
8      rtac (Object_Logic.rulify context (all_elims params1 (nth prems2 i))) 1
9      THEN EVERY1 (map (prepare_prem params2 prems2 context) prems1)
10   end)
```

With these two functions we can now also prove the introduction rule for the accessible part.

lemma `accpartI:`
shows `"⋀R x. (⋀y. R y x ⟹ accpart R y) ⟹ accpart R x"`
apply`(tactic {* expand_tac @{context} @{thms accpart_def} *})`
apply`(tactic {* prove_intro_tac 0 [acc_pred] acc_rules @{context} 1 *})`
done

Finally we need two functions that string everything together. The first function is the tactic that performs the proofs.

```
1 fun intro_tac defs rules preds i ctxt =
2   EVERY1 [Object_Logic.rulify_tac ctxt,
3           rewrite_goal_tac ctxt defs,
4           REPEAT o (resolve_tac [@{thm allI}, @{thm impI}]),
5           prove_intro_tac i preds rules ctxt]
```

Lines 2 to 4 in this tactic correspond to the function *expand_tac*. Some testcases for this tactic are:

lemma *even0_intro:*
shows *"even 0"*
by *(tactic {* intro_tac eo_defs eo_rules eo_preds 0 @{context} *})*

lemma *evenS_intro:*
shows *"⋀m. odd m ⟹ even (Suc m)"*
by *(tactic {* intro_tac eo_defs eo_rules eo_preds 1 @{context} *})*

lemma *fresh_App:*
shows *"⋀a t s. ⟦fresh a t; fresh a s⟧ ⟹ fresh a (App t s)"*
by *(tactic {*
  intro_tac @{thms fresh_def} fresh_rules [fresh_pred] 1 @{context} *})*

The second function sets up in Line 4 the goals to be proved (this is easy for the introduction rules since they are exactly the rules given by the user) and iterates *intro_tac* over all introduction rules.

```
1 fun intros rules preds defs lthy =
2 let
3   fun intros_aux (i, goal) =
4     Goal.prove lthy [] [] goal
5       (fn {context, ...} => intro_tac defs rules preds i context)
6 in
7   map_index intros_aux rules
8 end
```

The iteration is done with the function *map_index* since we need the introduction rule together with its number (counted from *0*). This completes the code for the functions deriving the reasoning infrastructure. It remains to implement some administrative code that strings everything together.

Administrative Functions

We have produced various theorems (definitions, induction principles and introduction rules), but apart from the definitions, we have not yet registered them with the theorem database. This is what the functions *note* does.

For convenience, we use the following three wrappers this function:

```
fun note_many qname ((name, attrs), thms) =
  Local_Theory.note ((Binding.qualify false qname name, attrs), thms)
```

```
fun note_single1 qname ((name, attrs), thm) =
  note_many qname ((name, attrs), [thm])

fun note_single2 name attrs (qname, thm) =
  note_many (Binding.name_of qname) ((name, attrs), [thm])
```

The function that "holds everything together" is `add_inductive`. Its arguments are the specification of the predicates `pred_specs` and the introduction rules `rule_spec`.

```
1  fun add_inductive pred_specs rule_specs lthy =
2  let
3    val mxs = map snd pred_specs
4    val pred_specs' = map fst pred_specs
5    val prednames = map fst pred_specs'
6    val preds = map (fn (p, ty) => Free (Binding.name_of p, ty)) pred_specs'
7    val tyss = map (binder_types o fastype_of) preds
8
9    val (namesattrs, rules) = split_list rule_specs
10
11   val (defs, lthy') = defns rules preds prednames mxs tyss lthy
12   val ind_prins = inds rules defs preds tyss lthy'
13   val intro_rules = intros rules preds defs lthy'
14
15   val mut_name = space_implode "_" (map Binding.name_of prednames)
16   val case_names = map (Binding.name_of o fst) namesattrs
17 in
18   lthy' |> note_many mut_name ((@{binding "intros"}, []), intro_rules)
19         ||>> note_many mut_name ((@{binding "inducts"}, []), ind_prins)
20         ||>> fold_map (note_single1 mut_name) (namesattrs ~~ intro_rules)
21         ||>> fold_map (note_single2 @{binding "induct"}
22               [Attrib.internal (K (Rule_Cases.case_names case_names)),
23                Attrib.internal (K (Rule_Cases.consumes 1)),
24                Attrib.internal (K (Induct.induct_pred ""))])
25              (prednames ~~ ind_prins)
26         |> snd
27 end
```

In Line 3 the function extracts the syntax annotations from the predicates. Lines 4 to 6 extract the names of the predicates and generate the variables terms (with types) corresponding to the predicates. Line 7 produces the argument types for each predicate.

Line 9 extracts the introduction rules from the specifications and stores also in `namesattrs` the names and attributes the user may have attached to these rules.

Line 11 produces the definitions and also registers the definitions in the local theory `lthy'`. The next two lines produce the induction principles and the introduction rules (all of them as theorems). Both need the local theory `lthy'` in which the definitions have been registered.

Lines 15 produces the name that is used to register the introduction rules. It is costum to collect all introduction rules under `string.intros`, whereby `string` stands

for the `"_"`-separated list of predicate names (for example `even_odd`. Also by custom, the case names in intuction proofs correspond to the names of the introduction rules. These are generated in Line 16.

Lines 18 and 19 now add to `lthy'` all the introduction rules und induction principles under the name `mut_name.intros` and `mut_name.inducts`, respectively (see previous paragraph).

Line 20 add further every introduction rule under its own name (given by the user).[4] Line 21 registers the induction principles. For this we have to use some specific attributes. The first `case_names` corresponds to the case names that are used by Isar to reference the proof obligations in the induction. The second `consumes 1` indicates that the first premise of the induction principle (namely the predicate over which the induction proceeds) is eliminated.

This completes all the code and fits in with the "front end" described in Section 7.2.[5]

7.5 Extensions of the Package (TBD)

Things to include at the end:

- include the code for the parameters
- say something about add-inductive to return the rules
- say something about the two interfaces for calling packages

Exercise 7.5.1: *In Section 7.3 we required that introduction rules must be of the form*

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

where the `As` and `Bs` can be any collection of formulae not containing the `preds`. This requirement is important, because if violated, the theory behind the inductive package does not work and also the proofs break. Write code that tests whether the introduction rules given by the user fit into the scheme described above. Hint: It is not important in which order the premises ar given; the `As` and $(\bigwedge ys.\ Bs \Longrightarrow pred\ ss)$ premises can occur in any order.

Exercise 7.5.2: *If you define `even` and `odd` with the standard inductive package*

```
inductive
  even_2 and odd_2
where
  even0_2: "even_2 0"
| evenS_2: "odd_2 m ⟹ even_2 (Suc m)"
| oddS_2:  "even_2 m ⟹ odd_2 (Suc m)"
```

[4] FIXME: what happens if the user did not give any name.

[5] FIXME: Describe `Induct.induct_pred`. Why the mut-name? What does `Binding.qualify` do?

you will see that the generated induction principle for `even'` *(namely* `even_2_odd_2.inducts` *has the additional assumptions* `odd_2 m` *and* `even_2 m` *in the recursive cases. These additional assumptions can sometimes make "life easier" in proofs. Since more assumptions can be made when proving properties, these induction principles are called strong inductions principles. However, it is the case that the "weak" induction principles imply the "strong" ones. Hint: Prove a property taking a pair (or tuple in case of more than one predicate) as argument: the property that you originally want to prove and the predicate(s) over which the induction proceeds.*
Write code that automates the derivation of the strong induction principles from the weak ones.

> ***Read More***
> *The standard inductive package is based on least fix-points. It allows more general introduction rules that can include any monotone operators and also provides a richer reasoning infrastructure. The code of this package can be found in* `HOL/Tools/inductive.ML`.

7.6 Definitional Packages

Type declarations

```
fun pat_completeness_auto ctxt =
  Pat_Completeness.pat_completeness_tac ctxt 1
    THEN auto_tac ctxt
```

ML *{*
  val conf = Function_Common.default_config
}

datatype *foo = Foo nat*

local_setup*{*Primrec.add_primrec [(@{binding "bar"}, NONE, NoSyn)]*
*   [(Attrib.empty_binding, @{term "⋀x. bar (Foo x) = x"})]*
* #> snd *}*

local_setup*{*Function.add_function [(@{binding "baz"}, NONE, NoSyn)]*
*   [(Attrib.empty_binding, @{term "⋀x. baz (Foo x) = x"})]*
*   conf pat_completeness_auto #> snd*}*

Appendix A

Recipes

Possible topics:

- translations/print translations; *Proof_Context.print_syntax*
- user space type systems (in the form that already exists)
- useful datastructures: discrimination nets, graphs, association lists
- Brief history of Isabelle

A.1 Useful Document Antiquotations

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

Solution: This can be achieved with document antiquotations.

Document antiquotations can be used for ensuring consistent type-setting of various entities in a document. They can also be used for sophisticated LaTeX-hacking. If you type on the Isabelle level

print_antiquotations

you obtain a list of all currently available document antiquotations and their options.

Below we will give the code for two additional document antiquotations both of which are intended to typeset ML-code. The crucial point of these document antiquotations is that they not just print the ML-code, but also check whether it compiles. This will provide a sanity check for the code and also allows you to keep documents in sync with other code, for example Isabelle.

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

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

The code is checked by sending the ML-expression *"val _ = a_piece_of_code"* to the ML-compiler (i.e. the function *ML_Context.eval_source_in* in Line 7 below). The complete code of the document antiquotation is as follows:

```
1  fun ml_val code_txt = "val _ = " ^ code_txt
2
3  fun output_ml {context = ctxt, ...} code_txt =
4  let
5    val srcpos = {delimited = false, text = (ml_val code_txt), pos =
6  Position.none}
7  in
8    (ML_Context.eval_source_in (SOME ctxt) ML_Compiler.flags srcpos;
9     Thy_Output.output ctxt (map Pretty.str (space_explode "\n" code_txt)))
10 end
11
12 val ml_checked_setup = Thy_Output.antiquotation @{binding "ML_checked"}
13 (Scan.lift Args.name) output_ml
```

setup *{* ml_checked_setup *}*

The parser *(Scan.lift Args.name)* in Line 7 parses a string, in this case the code, and then calls the function *output_ml*. As mentioned before, the parsed code is sent to the ML-compiler in Line 4 using the function *ml_val*, which constructs the appropriate ML-expression, and using *eval_in*, which calls the compiler. If the code is "approved" by the compiler, then the output function *output* in the next line pretty prints the code. This function expects that the code is a list of (pretty)strings where each string correspond to a line in the output. Therefore the use of *(space_explode "\n" txt)* which produces such a list according to linebreaks. There are a number of options for antiquotations that are observed by the function *output* when printing the code (including *[display]* and *[quotes]*). The function *antiquotation* in Line 7 sets up the new document antiquotation.

> ***Read More***
> *For more information about options of document antiquotations see [Isar Ref. Man., Sec. 4.2]).*

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

```
1  fun output_ml {context = ctxt, ...} (code_txt, pos) =
2  let
3    val srcpos = {delimited = false, pos = pos, text = ml_val code_txt}
4  in
5    (ML_Context.eval_source_in (SOME ctxt) ML_Compiler.flags srcpos;
6     Thy_Output.output ctxt (map Pretty.str (space_explode "\n" code_txt)))
7  end
8
9  val ml_checked_setup2 = Thy_Output.antiquotation @{binding "ML_checked2"}
10          (Scan.lift (Parse.position Args.name)) output_ml
```

setup *{* ml_checked_setup2 *}*

where in Lines 1 and 2 the positional information is properly treated. The parser *Parse.position* encodes the positional information in the result.

We can now write *@{ML_checked2 "2 + 3"}* in a document in order to obtain *2 + 3* and be sure that this code compiles until somebody changes the definition of addition.

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

```
@{ML_resp "a_piece_of_code" "a_pattern"}
```

To add some convenience and also to deal with large outputs, the user can give a partial specification by using ellipses. For example *(..., ...)* for specifying a pair. In order to check consistency between the pattern and the output of the code, we have to change the ML-expression that is sent to the compiler: in *ML_checked2* we sent the expression *"val _ = a_piece_of_code"* to the compiler; now the wildcard _ must be be replaced by the given pattern. However, we have to remove all ellipses from it and replace them by *"_"*. The following function will do this:

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

Next we add a response indicator to the result using:

```
fun add_resp pat = map (fn s => "> " ^ s) pat
```

The rest of the code of *ML_resp* is:

```
1  fun output_ml_resp {context = ctxt, ...} ((code_txt, pat), pos) =
2    (let
3       val srcpos = {delimited = false, text = ml_pat (code_txt, pat), pos =
4  pos}
5     in
6       ML_Context.eval_source_in (SOME ctxt) ML_Compiler.flags srcpos
7     end;
8     let
9       val code_output = space_explode "\n" code_txt
10      val resp_output = add_resp (space_explode "\n" pat)
11    in
12      Thy_Output.output ctxt (map Pretty.str (code_output @ resp_output))
```

```
13    end)
14
15
16  val ml_resp_setup = Thy_Output.antiquotation @{binding "ML_resp"}
17            (Scan.lift (Parse.position (Args.name -- Args.name)))
18               output_ml_resp
```

setup *{* ml_resp_setup *}*

In comparison with *ML_checked2*, we only changed the line about the compiler (Line 2), the lines about the output (Lines 4 to 7) and the parser in the setup (Line 11). Now you can write

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

to obtain

```
true andalso false
> false
```

or

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

to obtain

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

In both cases, the check by the compiler ensures that code and result match. A limitation of this document antiquotation, however, is that the pattern can only be given for values that can be constructed. This excludes values that are abstract datatypes, like *thm*s and *cterm*s.

A.2 Restricting the Runtime of a Function

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

Solution: In PolyML 5.2.1 and later, this can be achieved using the function *timeLimit*.

Assume you defined the Ackermann function on the ML-level.

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

Now the call

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

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

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

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

Note that *timeLimit* is only meaningful when you use PolyML 5.2.1 or later, because this version of PolyML has the infrastructure for multithreaded programming on which *timeLimit* relies.

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

A.3 Measuring Time

Problem: You want to measure the running time of a tactic or function.

Solution: Time can be measured using the function *start* and *result*.

Suppose you defined the Ackermann function on the Isabelle level.

```
fun
 ackermann:: "(nat × nat) ⇒ nat"
where
    "ackermann (0, n) = n + 1"
  | "ackermann (m, 0) = ackermann (m - 1, 1)"
  | "ackermann (m, n) = ackermann (m - 1, ackermann (m, n - 1))"
```

You can measure how long the simplifier takes to verify a datapoint of this function. The actual timing is done inside the wrapper function:

```
1 fun timing_wrapper tac st =
2 let
3   val t_start = Timing.start ();
4   val res = tac st;
5   val t_end = Timing.result t_start;
6 in
7   (writeln (Timing.message t_end); res)
8 end
```

Note that this function, in addition to a tactic, also takes a state *st* as argument and applies this state to the tactic (Line 4). The reason is that tactics are lazy functions

and you need to force them to run, otherwise the timing will be meaningless. The simplifier tactic, amongst others, can be forced to run by just applying the state to it. But "fully" lazy tactics, such as `resolve_tac`, need even more "standing-on-ones-head" to force them to run.

The time between start and finish of the simplifier will be calculated as the end time minus the start time. An example of the wrapper is the proof

```
lemma "ackermann (3, 4) = 125"
apply(tactic {*
  timing_wrapper (simp_tac (@{context} addsimps @{thms "eval_nat_numeral"}) 1)
*})
done
```

where it returns something on the scale of 3 seconds. We chose to return this information as a string, but the timing information is also accessible in number format.

> ***Read More***
> *Basic functions regarding timing are defined in* `Pure/General/timing.ML` *(for the PolyML compiler). Some more advanced functions are defined in* `Pure/General/output.ML`.

A.4 Executing an External Application (TBD)

Problem: You want to use an external application.

Solution: The function `bash_output` 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:

```
Isabelle_System.bash_output "echo Hello world!"
> ("Hello world!\n", 0)
```

Note that it works also fine with timeouts (see Recipe A.2 on Page 188), i.e. external applications are killed properly. For example, the following expression takes only approximately one second:

```
TimeLimit.timeLimit (Time.fromSeconds 1) Isabelle_System.bash_output "sleep
30"
     handle TimeLimit.TimeOut => ("timeout", ~1)
> ("timeout", ~1)
```

The function `bash_output` 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

```
Isabelle_System.bash_output "$FOO"
> ...
```

A.5 Writing an Oracle (TBD)

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 HOL/ex/Iff_Oracle.thy. The raw interface for adding oracles is* `add_oracle` *in Pure/thm.ML.*

For our explanation here, we restrict ourselves to decide propositional formulae which consist only of equivalences between propositional variables, i.e. we want to decide whether *(P = (Q = P)) = Q* is a tautology.

Assume, that we have a decision procedure for such formulae, implemented in ML. Since we do not care how it works, we will use it here as an "external solver":

ML_file *"external_solver.ML"*

We do, however, know that the solver provides a function *IffSolver.decide*. It takes a string representation of a formula and returns either *true* if the formula is a tautology or *false* otherwise. The input syntax is specified as follows:

formula ::= atom | (formula <=> formula)

and all token are separated by at least one space.

(FIXME: is there a better way for describing the syntax?)

We will proceed in the following way. We start by translating a HOL formula into the string representation expected by the solver. The solver's result is then used to build an oracle, which we will subsequently use as a core for an Isar method to be able to apply the oracle in proving theorems.

Let us start with the translation function from Isabelle propositions into the solver's string representation. To increase efficiency while building the string, we use functions from the *Buffer* module.

```
fun translate t =
  let
    fun trans t =
      (case t of
        @{term "op = :: bool ⇒ bool ⇒ bool"} $ t $ u =>
          Buffer.add " (" #>
          trans t #>
          Buffer.add "<=>" #>
          trans u #>
          Buffer.add ") "
      | Free (n, @{typ bool}) =>
         Buffer.add " " #>
         Buffer.add n #>
         Buffer.add " "
      | _ => error "inacceptable term")
  in Buffer.content (trans t Buffer.empty) end
```

Here is the string representation of the term `p = (q = p)`:

```
translate @{term "p = (q = p)"}
> " ( p <=> ( q <=> p ) ) "
```

Let us check, what the solver returns when given a tautology:

```
IffSolver.decide (translate @{term "p = (q = p) = q"})
> true
```

And here is what it returns for a formula which is not valid:

```
IffSolver.decide (translate @{term "p = (q = p)"})
> false
```

Now, we combine these functions into an oracle. In general, an oracle may be given any input, but it has to return a certified proposition (a special term which is type-checked), out of which Isabelle's inference kernel "magically" makes a theorem.

Here, we take the proposition to be show as input. Note that we have to first extract the term which is then passed to the translation and decision procedure. If the solver finds this term to be valid, we return the given proposition unchanged to be turned then into a theorem:

```
oracle iff_oracle = {* fn ct =>
  if IffSolver.decide (translate (HOLogic.dest_Trueprop (Thm.term_of ct)))
  then ct
  else error "Proof failed."*}
```

Here is what we get when applying the oracle:

```
iff_oracle @{cprop "p = (p::bool)"}
> p = p
```

(FIXME: is there a better way to present the theorem?)

To be able to use our oracle for Isar proofs, we wrap it into a tactic:

```
val iff_oracle_tac =
  CSUBGOAL (fn (goal, i) =>
```

```
    (case try iff_oracle goal of
      NONE => no_tac
    | SOME thm => rtac thm i))
```

and create a new method solely based on this tactic:

```
method_setup iff_oracle = {*
   Scan.succeed (K (Method.SIMPLE_METHOD' iff_oracle_tac))
*} "Oracle-based decision procedure for chains of equivalences"
```

Finally, we can test our oracle to prove some theorems:

```
lemma "p = (p::bool)"
   by iff_oracle

lemma "p = (q = p) = q"
   by iff_oracle
```

(FIXME: say something about what the proof of the oracle is ... what do you mean?)

A.6 SAT Solvers

Problem: You like to use a SAT solver to find out whether an Isabelle formula is satisfiable or not.

Solution: Isabelle contains a general interface for a number of external SAT solvers (including ZChaff and Minisat) and also contains a simple internal SAT solver that is based on the DPLL algorithm.

The SAT solvers expect a propositional formula as input and produce a result indicating that the formula is either satisfiable, unsatisfiable or unknown. The type of the propositional formula is `Prop_Logic.prop_formula` with the usual constructors such as `And`, `Or` and so on.

The function `Prop_Logic.prop_formula_of_term` translates an Isabelle term into a propositional formula. Let us illustrate this function by translating $A \wedge \neg A \vee B$. The function will return a propositional formula and a table. Suppose

```
val (pform, table) =
      Prop_Logic.prop_formula_of_term @{term "A ∧ ¬A ∨ B"}  Termtab.empty
```

then the resulting propositional formula `pform` is

```
Or (And (BoolVar 1, Not (BoolVar 1)), BoolVar 2)
```

where indices are assigned for the variables A and B, respectively. This assignment is recorded in the table that is given to the translation function and also returned (appropriately updated) in the result. In the case above the input table is empty (i.e. `Termtab.empty`) and the output table is

```
Termtab.dest table
> [(Free ("A", "bool"), 1), (Free ("B", "bool"), 2)]
```

An index is also produced whenever the translation function cannot find an appropriate propositional formula for a term. Attempting to translate $\forall x.\ P\ x$

```
val (pform', table') =
      Prop_Logic.prop_formula_of_term @{term "∀x::nat. P x"}
Termtab.empty
```

returns `BoolVar 1` for `pform'` and the table `table'` is:

```
map (apfst (Syntax.string_of_term @{context})) (Termtab.dest table')
> (∀x. P x, 1)
```

In the print out of the tabel, we used some pretty printing scaffolding to see better which assignment the table contains.

Having produced a propositional formula, you can now call the SAT solvers with the function `SAT_Solver.invoke_solver`. For example

```
SAT_Solver.invoke_solver "minisat" pform
> SAT_Solver.SATISFIABLE assg
```

determines that the formula `pform` is satisfiable. If we inspect the returned function `assg`

```
let
  val SAT_Solver.SATISFIABLE assg = SAT_Solver.invoke_solver "auto" pform
in
  (assg 1, assg 2, assg 3)
end
> (SOME true, SOME true, NONE)
```

we obtain a possible assignment for the variables `A` an `B` that makes the formula satisfiable.

Note that we invoked the SAT solver with the string `"auto"`. This string specifies which specific SAT solver is invoked. If instead called with `"auto"` several external SAT solvers will be tried (assuming they are installed).

There are also two tactics that make use of SAT solvers. One is the tactic `sat_tac`. For example

lemma `"True"`
apply`(tactic {* SAT.sat_tac @{context} 1 *})`
done

However, for proving anything more exciting using `sat_tac` you have to use a SAT solver that can produce a proof. The internal one is not usuable for this.

Read More
The interface for the external SAT solvers is implemented in `HOL/Tools/sat_solver.ML`*. This file contains also a simple SAT solver based on the DPLL algorithm. The tactics for SAT solvers are implemented in* `HOL/Tools/sat.ML`*. Functions concerning propositional formulas are implemented in* `HOL/Tools/prop_logic.ML`*. The tables used in the translation function are implemented in* `Pure/General/table.ML`*.*

A.7 User Space Type-Systems (TBD)

Appendix B

Solutions to Most Exercises

Solution for Exercise 3.2.1.

```
fun rev_sum
  ((p as Const (@{const_name plus}, _)) $ t $ u) = p $ u $ rev_sum t
  | rev_sum t = t
```

An alternative solution using the function *mk_binop* is:

```
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 3.2.2.

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

Solution for Exercise 3.2.3.

```
1  val quantifiers = [@{const_name All}, @{const_name Ex}]
2
3  fun kill_trivial_quantifiers trm =
4  let
5    fun aux t =
6      case t of
7        Const (s1, T1) $ Abs (x, T2, t2) =>
8          if member (op =) quantifiers s1 andalso not (loose_bvar1 (t2, 0))
9          then incr_boundvars ~1 (aux t2)
10         else Const (s1, T1) $ Abs (x, T2, aux t2)
11     | t1 $ t2 => aux t1 $ aux t2
```

```
12      | Abs (s, T, t') => Abs (s, T, aux t')
13      | _ => t
14  in
15    aux trm
16  end
```

In line 7 we traverse the term, by first checking whether a term is an application of a constant with an abstraction. If the constant stands for a listed quantifier (see Line 1) and the bound variable does not occur as a loose bound variable in the body, then we delete the quantifier. For this we have to increase all other dangling de Bruijn indices by `-1` to account for the deleted quantifier. An example is as follows:

```
@{prop "∀x y z. P x = P z"}
  |> kill_trivial_quantifiers
  |> pretty_term @{context}
  |> pwriteln
> ∀x z. P x = P z
```

Solution for Exercise 3.2.4.

```
fun mk_rev_upto i =
  1 upto i
  |> map (HOLogic.mk_number @{typ int})
  |> HOLogic.mk_list @{typ int}
  |> curry (op $) @{term "rev :: int list ⇒ int list"}
```

Solution for Exercise 3.2.5.

```
fun P n = @{term "P::nat ⇒ bool"} $ (HOLogic.mk_number @{typ "nat"} n)

fun rhs 1 = P 1
  | rhs n = HOLogic.mk_conj (P n, rhs (n - 1))

fun lhs 1 n = HOLogic.mk_imp (HOLogic.mk_eq (P 1, P n), rhs n)
  | lhs m n = HOLogic.mk_conj (HOLogic.mk_imp
                 (HOLogic.mk_eq (P (m - 1), P m), rhs n), lhs (m - 1) n)

fun de_bruijn n =
  HOLogic.mk_Trueprop (HOLogic.mk_imp (lhs n n, rhs n))
```

Solution for Exercise 5.1.1.

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

val scan_cmt =
let
  val begin_cmt = Scan.this_string "(*"
```

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

val parser = Scan.repeat (scan_cmt || any)

val scan_all =
      Scan.finite Symbol.stopper parser >> implode #> fst
```

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

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

Solution for Exercise 5.2.2.

```
datatype expr =
   Number of int
 | Mult of expr * expr
 | Add of expr * expr

fun parse_basic xs =
  (Parse.nat >> Number
   || Parse.$$$ "(" |-- parse_expr --| Parse.$$$ ")") xs
and parse_factor xs =
  (parse_basic --| Parse.$$$ "*" -- parse_factor >> Mult
   || parse_basic) xs
and parse_expr xs =
  (parse_factor --| Parse.$$$ "+" -- parse_expr >> Add
   || parse_factor) xs
```

Solution for Exercise 6.3.1.

The axiom rule can be implemented with the function `atac`. The other rules correspond to the theorems:

$\wedge_R$	*conjI*	$False$	*FalseE*
$\vee_{R_1}$	*disjI1*	$\wedge_L$	*conjE*
$\vee_{R_2}$	*disjI2*	$\vee_L$	*disjE*
$\longrightarrow_R$	*impI*	$=_L$	*iffE*
$=_R$	*iffI*		

For the other rules we need to prove the following lemmas.

```
lemma impE1:
  shows "⟦A ⟶ B; A; B ⟹ R⟧ ⟹ R"
  by iprover

lemma impE2:
  shows "⟦(C ∧ D) ⟶ B; C ⟶ (D ⟶B) ⟹ R⟧ ⟹ R"
  and   "⟦(C ∨ D) ⟶ B; ⟦C ⟶ B; D ⟶ B⟧ ⟹ R⟧ ⟹ R"
  and   "⟦(C ⟶ D) ⟶ B; D ⟶ B ⟹ C ⟶ D; B ⟹ R⟧ ⟹ R"
  and   "⟦(C = D) ⟶ B; (C ⟶ D) ⟶ ((D ⟶ C) ⟶ B) ⟹ R⟧ ⟹ R"
  by iprover+
```

Now the tactic which applies a single rule can be implemented as follows.

```
1  val apply_tac =
2  let
3    val intros = @{thms conjI disjI1 disjI2 impI iffI}
4    val elims = @{thms FalseE conjE disjE iffE impE2}
5  in
6    atac
7    ORELSE' resolve_tac intros
8    ORELSE' eresolve_tac elims
9    ORELSE' (etac @{thm impE1} THEN' atac)
10 end
```

In Line 11 we apply the rule *impE1* in concjunction with *atac* in order to reduce the number of possibilities that need to be explored. You can use the tactic as follows.

```
lemma
  shows "((((P ⟶ Q) ⟶ P) ⟶ P) ⟶ Q) ⟶ Q"
apply(tactic {* (DEPTH_SOLVE o apply_tac) 1 *})
done
```

We can use the tactic to prove or disprove automatically the de Bruijn formulae from Exercise 3.2.5.

```
fun de_bruijn_prove ctxt n =
let
  val goal = HOLogic.mk_Trueprop (HOLogic.mk_imp (lhs n n, rhs n))
in
  Goal.prove ctxt ["P"] [] goal
   (fn _ => (DEPTH_SOLVE o apply_tac) 1)
end
```

You can use this function to prove de Bruijn formulae.

```
de_bruijn_prove @{context} 3
```

Solution for Exercise 6.5.1.

```
fun dest_sum term =
  case term of
    (@{term "(op +):: nat ⇒ nat ⇒ nat"} $ t1 $ t2) =>
        (snd (HOLogic.dest_number t1), snd (HOLogic.dest_number t2))
  | _ => raise TERM ("dest_sum", [term])

fun get_sum_thm ctxt t (n1, n2) =
let
  val sum = HOLogic.mk_number @{typ "nat"} (n1 + n2)
  val goal = Logic.mk_equals (t, sum)
in
  Goal.prove ctxt [] [] goal (K (Arith_Data.arith_tac ctxt 1))
end

fun add_sp_aux ctxt t =
let
  val t' = term_of t
in
  SOME (get_sum_thm ctxt t' (dest_sum t'))
  handle TERM _ => NONE
end
```

The setup for the simproc is

```
simproc_setup add_sp ("t1 + t2") = {* K add_sp_aux *}
```

and a test case is the lemma

```
lemma "P (Suc (99 + 1)) ((0 + 0)::nat) (Suc (3 + 3 + 3)) ((4 + 1)::nat)"
  apply(tactic {* simp_tac (put_simpset HOL_basic_ss @{context} addsimprocs [@{simproc
add_sp}]) 1 *})
```

where the simproc produces the goal state

```
goal (1 subgoal):
 1. P (Suc 100) 0 (Suc 9) 5
```

Solution for Exercise 6.6.1.

The following code assumes the function *dest_sum* from the previous exercise.

```
fun add_simple_conv ctxt ctrm =
let
  val trm = Thm.term_of ctrm
in
  case trm of
     @{term "(op +)::nat ⇒ nat ⇒ nat"} $ _ $ _ =>
        get_sum_thm ctxt trm (dest_sum trm)
    | _ => Conv.all_conv ctrm
end

val add_conv = Conv.bottom_conv add_simple_conv

fun add_tac ctxt = CONVERSION (add_conv ctxt)
```

A test case for this conversion is as follows

```
lemma "P (Suc (99 + 1)) ((0 + 0)::nat) (Suc (3 + 3 + 3)) ((4 + 1)::nat)"
  apply(tactic {* add_tac @{context} 1 *})?
```

where it produces the goal state

```
goal (1 subgoal):
 1. P (Suc 100) 0 (Suc 9) 5
```

Solution for Exercise 6.6.2.

We use the timing function `timing_wrapper` from Recipe A.3. To measure any difference between the simproc and conversion, we will create mechanically terms involving additions and then set up a goal to be simplified. We have to be careful to set up the goal so that other parts of the simplifier do not interfere. For this we construct an unprovable goal which, after simplification, we are going to "prove" with the help of "**sorry**", that is the method `Skip_Proof.cheat_tac`

For constructing test cases, we first define a function that returns a complete binary tree whose leaves are numbers and the nodes are additions.

```
fun term_tree n =
let
  val count = Unsynchronized.ref 0;

  fun term_tree_aux n =
    case n of
      0 => (count := !count + 1; HOLogic.mk_number @{typ nat} (!count))
    | _ => Const (@{const_name "plus"}, @{typ "nat⇒nat⇒nat"})
             $ (term_tree_aux (n - 1)) $ (term_tree_aux (n - 1))
in
  term_tree_aux n
end
```

This function generates for example:

```
pwriteln (pretty_term @{context} (term_tree 2))
> (1 + 2) + (3 + 4)
```

The next function constructs a goal of the form `P ...` with a term produced by `term_tree` filled in.

```
fun goal n = HOLogic.mk_Trueprop (@{term "P::nat⇒ bool"} $ (term_tree n))
```

Note that the goal needs to be wrapped in a `Trueprop`. Next we define two tactics, `c_tac` and `s_tac`, for the conversion and simproc, respectively. The idea is to first apply the conversion (respectively simproc) and then prove the remaining goal using `cheat_tac`.

ML `Skip_Proof.cheat_tac`

```
local
  fun mk_tac tac =
        timing_wrapper (EVERY1 [tac, Skip_Proof.cheat_tac])
in
  fun c_tac ctxt = mk_tac (add_tac ctxt)
  fun s_tac ctxt = mk_tac (simp_tac
    (put_simpset HOL_basic_ss ctxt addsimprocs [@{simproc add_sp}]))
end
```

This is all we need to let the conversion run against the simproc:

```
val _ = Goal.prove @{context} [] [] (goal 8) (fn {context, ...} => c_tac
context)
val _ = Goal.prove @{context} [] [] (goal 8) (fn {context, ...} => s_tac
context)
```

If you do the exercise, you can see that both ways of simplifying additions perform relatively similar with perhaps some advantages for the simproc. That means the simplifier, even if much more complicated than conversions, is quite efficient for tasks it is designed for. It usually does not make sense to implement general-purpose rewriting using conversions. Conversions only have clear advantages in special situations: for example if you need to have control over innermost or outermost rewriting, or when rewriting rules are lead to non-termination.

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] R. Dyckhoff. Contraction-Free Sequent Calculi for Intuitionistic Logic. *The Journal of Symbolic Logic*, 57(3):795–807, 1992.

[3] M. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[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.

Structure Index