The Isabelle Programmer's Cookbook (fragment)

with contributions by:

Alexander Krauss
Jeremy Dawson
Stefan Berghofer

September 30, 2008

Contents

Chapter 1

Introduction

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

1.1 Intended Audience and Prior Knowledge

This cookbook targets an audience who already knows how to use Isabelle for writing theories and proofs. We also assume that readers are familiar with the Standard ML, the programming language in which most of Isabelle is implemented. If you are unfamiliar with either of these two subjects, you should first work through the Isabelle/HOL tutorial [1] and Paulson's book on Standard ML [2].

1.2 Existing Documentation

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

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

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 Many parts of this manual are outdated now, but some parts, mainly the chapters on tactics, are still useful.

Then of ourse there is:

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

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

Chapter 2

First Steps

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

```
theory CookBook
imports Main
begin
...
```

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

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

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

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

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

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

```
ML {*
  val _ = warning "any string"
*}
```

will print out the string inside the response buffer of Isabelle.

2.1 Antiquotations

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

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

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

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

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

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

In a similar way you can use antiquotations to refer to types and theorems:

```
ML {* @{typ "(int * nat) list"} *}
ML {* @{thm allI} *}
```

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

2.2 Terms

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

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

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

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

Read More

Terms are described in detail in [Impl. Man., ch. 2]. Their definition and many useful operations can be found in `Pure/term.ML`.

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

Exercise 2.2.1. *Look at the internal term representation of the following terms, and find out why they are represented like this.*

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

Hint: The third term is already quite big, and the pretty printer may omit parts of it by default. If you want to see all of it, you can use `print_depth 50` *to set the limit to a value high enough.*

```
ML {*
  @{const_name plus}
*}
```

```
ML {*
  @{term "{ [x::int] | x. x ≤ -2 }"}
*}
```

The internal names of constants like `zero` or `+` are often more complex than one first expects. Here, the extra prefixes `zero_class` and `plus_class` are present because the constants are defined within a type class. Guessing such internal names can be extremely hard, which is why the system provides another antiquotation: `@{const_name plus}` gives just this name.

FIXME: maybe explain `@{prop ...}` as a special kind of terms

```
ML {* @{prop "True"} *}
```

2.3 Possible Section on Construting Explicitly Terms

There is a disadvantage of using the `@{term ...}` antiquotation directly in order to construct terms.

```
ML {*

  val nat = HOLogic.natT
  val x = Free ("x", nat)
  val t = Free ("t", nat)
  val P = Free ("P", nat --> HOLogic.boolT)
  val Q = Free ("Q", nat --> HOLogic.boolT)

  val A1 = Logic.all x
           (Logic.mk_implies (HOLogic.mk_Trueprop (P $ x),
                              HOLogic.mk_Trueprop (Q $ x)))


  val A2 = HOLogic.mk_Trueprop (P $ t)

*}
```

Exercise 2.3.1. *Write a function* `rev_sum : term -> term` *that takes a term of the form* t_1 + t_2 + ... + t_n *(whereby* `i` *might be zero) and returns the reversed sum* t_n + ... + t_2 + t_1. *Assume the* t_i *can be arbitrary expressions and also note that + associates to the left. Try your function on some examples, and see if the result typechecks. (FIXME: clash with the type-checking section later)*

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

Exercise 2.3.2. *Write a function which takes two terms representing natural numbers in unary (like* `Suc (Suc (Suc 0))`*), and produce the unary number representing their sum.*

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

Exercise 2.3.3. *Look at the functions defined in* `Pure/logic.ML` *and* `HOL/hologic.ML` *and see if they can make your life easier.*

2.4 Type checking

We can freely construct and manipulate terms, since they are just arbitrary unchecked trees. However, we eventually want to see if a term is well-formed, or type checks, relative to a theory.

Type checking is done via the function `cterm_of`, which turns a `term` into a `cterm`, a *certified* term. Unlike `terms`, which are just trees, `cterms` are abstract objects that are guaranteed to be type-correct, and can only be constructed via the official interfaces.

(FIXME: Alex what do you mean concretely by "official interfaces")

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

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

2.5 Theorems

Just like `cterms`, theorems (of type `thm`) are abstract objects that can only be built by going through the kernel interfaces, which means that all your proofs will be checked. The basic rules of the Isabelle/Pure logical framework are defined in *Pure/thm.ML*.

Using these rules, which are just ML functions, you can do simple natural deduction proofs on the ML level. For example, the statement

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

can be proved in ML like this[1]:

```
ML {*

let
  val thy = @{theory}

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

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

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

```
  val Qt = implies_elim Pt_implies_Qt (assume assm2);
in

  Qt
  |> implies_intr assm2
  |> implies_intr assm1
end

*}
```

FIXME Explain this program more carefully (`assume`, `forall_elim` ...)

2.6 Tactical reasoning

The goal-oriented tactical style is similar to the *apply* style at the user level. Reasoning is centered around a *goal*, which is modified in a sequence of proof steps until it is solved.

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

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

Since the final result C could again be an implication, there is the `#` around the final result, which protects its premises from being misinterpreted as open subgoals. The protection $\# :: prop \Rightarrow prop$ is just the identity and used as a syntactic marker.

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

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

See *Pure/General/seq.ML* for the implementation of lazy sequences.

Of course, tactics are expected to behave nicely and leave the final conclusion C intact. In order to start a tactical proof for A, we just set up the trivial goal $A \Longrightarrow \#(A)$ and run the tactic on it. When the subgoal is solved, we have just $\#(A)$ and can remove the protection.

The operations in *Pure/goal.ML* do just that and we can use them.

Let us transcribe a simple apply style proof from the tutorial[1] into ML:

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

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

Tactics that affect only a certain subgoal, take a subgoal number as an integer parameter. Here we always work on the first subgoal, following exactly the `apply` script.

(Fixme: would it make sense to explain THEN' here)

2.7 Case Study: Relation Composition

Note: This is completely unfinished. I hoped to have a section with a nontrivial example, but I ran into several problems.

Recall that HOL has special syntax for set comprehensions: `{f x y |x y. P x y}` abbreviates `"{u. ∃x y. u = f x y ∧ P x y}"`.

We will automatically prove statements of the following form:

$$\{(l_1\ x,\ r_1\ x)\ |x.\ P_1\ x\}\ O\ \{(l_2\ x,\ r_2\ x)\ |x.\ P_2\ x\} =$$
$$\{(l_2\ x,\ r_1\ y)\ |x\ y.\ r_2\ x = l_1\ y \land P_2\ x \land P_1\ y\}$$

In Isabelle, relation composition is defined to be consistent with function composition, that is, the relation applied “first” is written on the right hand side. This different from what many textbooks do.

The above statement about composition is not proved automatically by `simp`, and it cannot be solved by a fixed set of rewrite rules, since the number of (implicit) quantifiers may vary. Here, we only have one bound variable in each comprehension, but in general there can be more. On the other hand, `auto` proves the above statement quickly, by breaking the equality into two parts and proving them separately. However, if e.g. P_1 is a complicated expression, the automated tools may get confused.

Our goal is now to develop a small procedure that can compute (with proof) the composition of two relation comprehensions, which can be used to extend the simplifier.

2.8 A tactic

Let's start with a step-by-step proof of the above statement

```
lemma "{($l_1$ x, $r_1$ x) |x. $P_1$ x} O {($l_2$
  x, $r_2$ x) |x. $P_2$ x}
  = {($l_2$ x, $r_1$ y) |x y. $r_2$ x = $l_1$ y ∧ $P_2$ x ∧ $P_1$ y}"
apply (rule set_ext)
apply (rule iffI)
 apply (erule rel_compE)  — ⊆
 apply (erule CollectE)      — eliminate Collect, ∃ , ∧, and pairs
 apply (erule CollectE)
 apply (erule exE)
 apply (erule exE)
 apply (erule conjE)
 apply (erule conjE)
 apply (erule Pair_inject)
 apply (erule Pair_inject)
 apply (simp only:)

 apply (rule CollectI)     — introduce them again
 apply (rule exI)
 apply (rule exI)
 apply (rule conjI)
  apply (rule refl)
 apply (rule conjI)
  apply (rule sym)
  apply (assumption)
 apply (rule conjI)
  apply assumption
 apply assumption

apply (erule CollectE)   — ⊆
apply (erule exE)+
apply (erule conjE)+
apply (simp only:)
apply (rule rel_compI)
 apply (rule CollectI)
 apply (rule exI)
 apply (rule conjI)
  apply (rule refl)
 apply assumption

apply (rule CollectI)
apply (rule exI)
apply (rule conjI)
apply (subst Pair_eq)
apply (rule conjI)
 apply assumption
apply (rule refl)
apply assumption
done
```

The reader will probably need to step through the proof and verify that there is nothing spectacular going on here. The *apply* script just applies the usual elimination and introduction rules in the right order.

This script is of course totally unreadable. But we are not trying to produce pretty Isar proofs here. We just want to find out which rules are needed and how they must be applied to complete the proof. And a detailed apply-style proof can often be turned into a tactic quite easily. Of course we must resist the temptation to use `auto`, `blast` and friends, since their behaviour is not predictable enough. But the simple `rule` and `erule` methods are fine.

Notice that this proof depends only in one detail on the concrete equation that we want to prove: The number of bound variables in the comprehension corresponds to the number of existential quantifiers that we have to eliminate and introduce again. In fact this is the only reason why the equations that we want to prove are not just instances of a single rule.

Here is the ML equivalent of the tactic script above:

```
ML {*
val compr_compose_tac =
  rtac @{thm set_ext}
  THEN' rtac @{thm iffI}
  THEN' etac @{thm rel_compE}
  THEN' etac @{thm CollectE}
  THEN' etac @{thm CollectE}
  THEN' (fn i => REPEAT (etac @{thm exE} i))
  THEN' etac @{thm conjE}
  THEN' etac @{thm conjE}
  THEN' etac @{thm Pair_inject}
  THEN' etac @{thm Pair_inject}
  THEN' asm_full_simp_tac HOL_basic_ss
  THEN' rtac @{thm CollectI}
  THEN' (fn i => REPEAT (rtac @{thm exI} i))
  THEN' rtac @{thm conjI}
  THEN' rtac @{thm refl}
  THEN' rtac @{thm conjI}
  THEN' rtac @{thm sym}
  THEN' assume_tac
  THEN' rtac @{thm conjI}
  THEN' assume_tac
  THEN' assume_tac

  THEN' etac @{thm CollectE}
  THEN' (fn i => REPEAT (etac @{thm exE} i))
  THEN' etac @{thm conjE}
  THEN' etac @{thm conjE}
  THEN' etac @{thm conjE}
  THEN' asm_full_simp_tac HOL_basic_ss
  THEN' rtac @{thm rel_compI}
  THEN' rtac @{thm CollectI}
  THEN' (fn i => REPEAT (rtac @{thm exI} i))
  THEN' rtac @{thm conjI}
  THEN' rtac @{thm refl}
  THEN' assume_tac
  THEN' rtac @{thm CollectI}
  THEN' (fn i => REPEAT (rtac @{thm exI} i))
  THEN' rtac @{thm conjI}
  THEN' simp_tac (HOL_basic_ss addsimps [@{thm Pair_eq}])
  THEN' rtac @{thm conjI}
  THEN' assume_tac
  THEN' rtac @{thm refl}
  THEN' assume_tac
*}
```

```
lemma test1: "{(l_1 x, r_1 x) |x. P_1 x} O {(l_2
  x, r_2 x) |x. P_2 x}
  = {(l_2 x, r_1 y) |x y. r_2 x = l_1 y ∧ P_2 x ∧ P_1 y}"
by (tactic "compr_compose_tac 1")
```

```
lemma test3: "{(l_1 x, r_1 x) |x. P_1 x} O {(l_2 x z, r_2 x z) |x z. P_2 x
z}
  = {(l_2 x z, r_1 y) |x y z. r_2 x z = l_1 y ∧ P_2 x z ∧ P_1 y}"
by (tactic "compr_compose_tac 1")
```

So we have a tactic that works on at least two examples. Getting it really right requires some more effort. Consider the goal

```
lemma "{(n, Suc n) |n. n > 0} O {(n, Suc n) |n. P n}
  = {(n, Suc m)|n m. Suc n = m ∧ P n ∧ m > 0}"
```

This is exactly an instance of *test1*, but our tactic fails on it with the usual uninformative *empty result requence*.

We are now in the frequent situation that we need to debug. One simple instrument for this is `print_tac`, which is the same as `all_tac` (the identity for `THEN`), i.e. it does nothing, but it prints the current goal state as a side effect. Another debugging option is of course to step through the interactive apply script.

Finding the problem could be taken as an exercise for the patient reader, and we will go ahead with the solution.

The problem is that in this instance the simplifier does more than it did in the general version of lemma *test1*. Since l_1 and l_2 are just the identity function, the equation corresponding to $l_1\ y = r_2\ x$ becomes $m = Suc\ n$. Then the simplifier eagerly replaces all occurences of m by $Suc\ n$ which destroys the structure of the proof.

This is perhaps the most important lesson to learn, when writing tactics:

Avoid automation at all cost!!!.

Let us look at the proof state at the point where the simplifier is invoked:

```
 1. ⋀x xa y z n na.
       ⟦x = (xa, z); P n; 0 < na; xa = n; y = Suc n; y = na; z = Suc na⟧
       ⟹ x ∈ {(n, Suc m) |n m. Suc n = m ∧ P n ∧ 0 < m}
 2. ⋀x. x ∈ {(n, Suc m) |n m. Suc n = m ∧ P n ∧ 0 < m} ⟹
         x ∈ {(n, Suc n) |n. 0 < n} O {(n, Suc n) |n. P n}
```

Like in the apply proof, we now want to eliminate the equations that "define" `x`, `xa` and `z`. The other equations are just there by coincidence, and we must not touch them.

For such purposes, there is the internal tactic *hyp_subst_single*. Its job is to take exactly one premise of the form `v = t`, where `v` is a variable, and replace `v` in the whole subgoal. The hypothesis to eliminate is given by its position.

We can use this tactic to eliminate `x`:

```
apply (tactic "single_hyp_subst_tac 0 1")
```

```
 1. ⋀x xa y z n na.
       ⟦P n; 0 < na; xa = n; y = Suc n; y = na; z = Suc na⟧
       ⟹ (xa, z) ∈ {(n, Suc m) |n m. Suc n = m ∧ P n ∧ 0 < m}
 2. ⋀x. x ∈ {(n, Suc m) |n m. Suc n = m ∧ P n ∧ 0 < m} ⟹
         x ∈ {(n, Suc n) |n. 0 < n} O {(n, Suc n) |n. P n}
```

```
apply (tactic "single_hyp_subst_tac 2 1")
apply (tactic "single_hyp_subst_tac 2 1")
apply (tactic "single_hyp_subst_tac 3 1")
 apply (rule CollectI)      — introduce them again
 apply (rule exI)
 apply (rule exI)
 apply (rule conjI)
  apply (rule refl)
 apply (rule conjI)
  apply (assumption)
 apply (rule conjI)
  apply assumption
 apply assumption

apply (erule CollectE)    — ⊆
apply (erule exE)+
apply (erule conjE)+
apply (tactic "single_hyp_subst_tac 0 1")
apply (rule rel_compI)
 apply (rule CollectI)
 apply (rule exI)
 apply (rule conjI)
  apply (rule refl)
 apply assumption

apply (rule CollectI)
apply (rule exI)
apply (rule conjI)
apply (subst Pair_eq)
apply (rule conjI)
 apply assumption
apply (rule refl)
apply assumption
done

ML {*
val compr_compose_tac =
  rtac @{thm set_ext}
  THEN’ rtac @{thm iffI}
  THEN’ etac @{thm rel_compE}
  THEN’ etac @{thm CollectE}
  THEN’ etac @{thm CollectE}
  THEN’ (fn i => REPEAT (etac @{thm exE} i))
  THEN’ etac @{thm conjE}
  THEN’ etac @{thm conjE}
  THEN’ etac @{thm Pair_inject}
  THEN’ etac @{thm Pair_inject}
  THEN’ single_hyp_subst_tac 0
  THEN’ single_hyp_subst_tac 2
  THEN’ single_hyp_subst_tac 2
  THEN’ single_hyp_subst_tac 3
  THEN’ rtac @{thm CollectI}
  THEN’ (fn i => REPEAT (rtac @{thm exI} i))
  THEN’ rtac @{thm conjI}
  THEN’ rtac @{thm refl}
  THEN’ rtac @{thm conjI}
  THEN’ assume_tac
  THEN’ rtac @{thm conjI}
  THEN’ assume_tac
  THEN’ assume_tac

  THEN’ etac @{thm CollectE}
  THEN’ (fn i => REPEAT (etac @{thm exE} i))
  THEN’ etac @{thm conjE}
  THEN’ etac @{thm conjE}
  THEN’ etac @{thm conjE}
  THEN’ single_hyp_subst_tac 0
  THEN’ rtac @{thm rel_compI}
  THEN’ rtac @{thm CollectI}
  THEN’ (fn i => REPEAT (rtac @{thm exI} i))
  THEN’ rtac @{thm conjI}
  THEN’ rtac @{thm refl}
  THEN’ assume_tac
  THEN’ rtac @{thm CollectI}
  THEN’ (fn i => REPEAT (rtac @{thm exI} i))
  THEN’ rtac @{thm conjI}
  THEN’ stac @{thm Pair_eq}
  THEN’ rtac @{thm conjI}
  THEN’ assume_tac
  THEN’ rtac @{thm refl}
  THEN’ assume_tac
*}
```

```
lemma "{(n, Suc n) |n. n > 0 ∧ A} O {(n, Suc n) |n m. P m n}
  = {(n, Suc m)|n m’ m. Suc n = m ∧ P m’ n ∧ (m > 0 ∧ A)}"
apply (tactic "compr_compose_tac 1")
done
```

The next step is now to turn this tactic into a simplification procedure. This just means that we need some code that builds the term of the composed relation.

```
use "comp_simproc"
```

Chapter 3

Parsing

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

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

3.1 Parsing Isar input

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

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

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

```
open StringCvt ;
type ('res, 'src) ex_reader = 'src -> 'res * 'src
(* ex_reader : ('res, 'src) reader -> ('res, 'src) ex_reader *)
fun ex_reader rdr src = Option.valOf (rdr src) ;
(* reader : ('res, 'src) ex_reader -> ('res, 'src) reader *)
fun reader exrdr src = SOME (exrdr src) handle _ => NONE ;
```

3.2 The Scan structure

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

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

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

Some functions from that structure are

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

These functions parse a result off the input source twice.

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

-- returns both.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3 The `OuterLex` structure

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

```
structure T = OuterLex;
```

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

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

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

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

but here are some runnable examples for viewing tokens:

FIXME

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

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

There is a special end-of-file token:

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

3.4 The `OuterParse` structure

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

```
structure P = OuterParse;
```

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

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

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

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

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

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

A few examples:

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

The following code helps run examples:

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

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

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

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

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

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

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

3.5 The SpecParse structure

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

```
open SpecParse ;
attrib : Attrib.src tok_rdr ;
attribs : Attrib.src list tok_rdr ;
```

```
opt_attribs : Attrib.src list tok_rdr ;
xthm : (thmref * Attrib.src list) tok_rdr ;
xthms1 : (thmref * Attrib.src list) list tok_rdr ;

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

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

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

3.6 The Args structure

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
typ : typ cgatlp ;

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

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

3.7 Attributes, and the Attrib structure

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

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

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

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

An attribute is stored in a theory as indicated by:

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

where the first and third arguments are name and description of the attribute, and the second is a function which parses the attribute input text (including the attribute name, which has necessarily already been parsed). Here, THEN_att is a function declared in the code for the structure Attrib, but not published in its signature. The source file src/Pure/Isar/attrib.ML shows the use of Attrib.add_attributes to add a number of attributes.

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

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

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

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

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

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

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

Attrib.thm : thm cgatlp ;

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

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

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

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

3.8 Methods, and the Method structure

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

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

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

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

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

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

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

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

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

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

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

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

A method is stored in a theory as indicated by:

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

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

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

Chapter 4

Recipes

Accumulate a list of theorems under a name

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

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

```
setup FooRules.setup
```

This declares a context data slot where the theorems are stored, an attribute *foo* (with the usual *add* and *del* options to declare new rules, and the internal ML interface to retrieve and modify the facts.

Furthermore, the facts are made available under the dynamic fact name *foo*:

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

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

```
thm foo
```

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

XXX

Read More

Bibliography

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

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

[3] M. Wenzel. *The Isabelle/Isar Implementation*. http://isabelle.in.tum.de/doc/implementation.pdf.