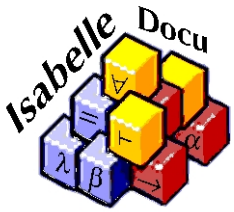

The Isabelle Programmer's Cookbook (fragment)

with contributions by:

Alexander Krauss
Jeremy Dawson
Stefan Berghofer

September 5, 2008

Contents

Chapter 1

Introduction

The purpose of this document is to guide the reader through the first steps in 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 the Isabelle system to write theories and proofs, but without using ML. You should also be familiar with the *Standard ML* programming language, which is used for Isabelle programming. If you are unfamiliar with any of these two subjects, you should first work through the Isabelle/HOL tutorial [**?**] and Paulson's book on Standard ML [**?**].

1.2 Primary Documentation

The Implementation Manual [?] describes Isabelle from a programmer's perspective, documenting both the underlying concepts and the concrete interfaces.

The Isabelle Reference Manual [?] is an older document that used to be the main reference, when all reasoning happened on the ML level. Many parts of it are outdated now, but some parts, mainly the chapters on tactics, are still useful.

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 dificult and frustrating at times, when interfaces are changing 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 happens in an enhanced dialect of Standard ML, which adds antiquotations containing references to the logical context.

Just like all lemmas or proofs, all ML code that you write lives in a theory, where it is embedded using the **ML** command:

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

The **ML** command takes an arbitrary ML expression, which is evaluated. It can also contain value or function bindings.

2.1 Antiquotations

The main advantage of embedding all code in a theory is that the code can contain references to entities that are defined in the theory. Let us for example, print out the name of the current theory:

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

The `@{theory}` antiquotation is substituted with the current theory, whose name can then be extracted using a the function `Context.theory_name`. Note that antiquotations are statically scoped. The function

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

does *not* return the name of the current theory. Instead, we have defined the constant function that always returns the string `"CookBook"`, which is the name of `@{theory}` at the point where the code is embedded. Operationally speaking, `@{theory}` is *not* replaced with code that will look up the current theory in some (destructive) data structure and return it. Instead, it is really replaced with the theory value.

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

2.2 Terms

We can simply quote Isabelle terms from ML using the `@{term ...}` antiquotation:

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

This shows the term `a + b = c` in the internal representation, with all gory details. Terms are just an ML datatype, and they are defined in `Pure/term.ML`.

The representation of terms uses deBruin indices: Bound variables are represented by the constructor `Bound`, and the index refers to the number of lambdas we have to skip until we hit the lambda that binds the variable. The names of bound variables are kept at the abstractions, but they are just comments. See [Impl. Man., ch. **??**] for more details.

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

In a similar way we can quote types and theorems:

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

In the default setup, types and theorems are printed as strings.

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 thick.

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.*

2.3 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 wellformed in a certain context.

Type checking is done via `cterm_of`, which turns 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 can only be constructed via the official interfaces.

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

```
ML {*
  let
    val natT = @{typ "nat"}
    val zero = @{term "0::nat"}(*Const ("HOL.zero_class.zero", natT)*)
  in
    cterm_of @{theory}
        (Const ("HOL.plus_class.plus", natT --> natT --> natT)
         $ zero $ zero)
  end
*}

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.

Exercise 2.3.1. *Write a function* `rev_sum : term -> term` *that takes a term of the form* t_1 `+` t_2 `+ ... +` t_n *and returns the reversed sum* t_n `+ ... +` t_2 `+` t_1. *Note that* `+` *associates to the left. Try your function on some examples, and see if the result typechecks.*

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.*

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 Theorems

Just like `cterm`s, 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 $\llbracket \bigwedge x.\ P\ x \Longrightarrow Q\ x;\ P\ t \rrbracket \Longrightarrow Q\ t$ can be proved like this[1]:

```
ML {*
let
  val thy = @{theory}
  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)))
           |> cterm_of thy

  val A2 = HOLogic.mk_Trueprop (P $ t)
           |> cterm_of thy

  val Pt_implies_Qt =
        assume A1
        |> forall_elim (cterm_of thy t)

  val Qt = implies_elim Pt_implies_Qt (assume A2)
in
  Qt
  |> implies_intr A2
  |> implies_intr A1
end
*}
```

2.5 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:

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

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

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 ⇒ 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[**?**] 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.

2.6 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:

```
{(l₁ x, r₁ x) |x. P₁ x} O {(l₂ x, r₂ x) |x. P₂ x} =
{(l₂ x, r₁ y) |x y. r₂ x = l₁ y ∧ P₂ x ∧ P₁ 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.

2.7 A tactic

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

```
lemma "{(l₁ x, r₁ x) |x. P₁ x} O {(l₂
  x, r₂ x) |x. P₂ x}
  = {(l₂ x, r₁ y) |x y. r₂ x = l₁ y ∧ P₂ x ∧ P₁ 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: "{(l1 x, r1 x) |x. P1 x} O {(l2
  x, r2 x) |x. P2 x}
```

```
  = {(l₂ x, r₁ y) |x y. r₂ x = l₁ y ∧ P₂ x ∧ P₁ y}"
by (tactic "compr_compose_tac 1")

lemma test3: "{(l₁ x, r₁ x) |x. P₁ x} O {(l₂ x z, r₂ x z) |x z. P₂ x
z}
  = {(l₂ x z, r₁ y) |x y z. r₂ x z = l₁ y ∧ P₂ x z ∧ P₁ 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} 0 {(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)* — $\subseteq$
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} 0 {(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

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};
*}
```

Read More *XXX*