ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Dipl.-Inf. Achim D. Brucker
Dr. Burkhart Wolff

Computer-supported Modeling and Reasoning

http://www.infsec.ethz.ch/education/permanent/csmr/

(rev. 16802)

Submission date: –

HOL: Axiomatic Classes and Typed Set Theory

In this exercise, we will deepen our knowledge on a specific concept of theory structuring in Isabelle, namely axiomatic classes. We will extend conservative library constructions in typed set theory, and will lay the groundwork for inductive definitions.

Technically, we will apply automated proof procedures, be it on the level of rewriting or tableaux based procedures and combined methods such as auto.

1 Isabelle

1.1 Axiomatic Classes

Languages like Haskell have popularized the notion of type classes. In its simplest form, a type class is a set of types with a common interface: all types in that class must provide the functions in the interface. Isabelle offers a similar concept, called *axiomatic type classes*. An axiomatic type classes is something like a type class with axioms, i.e., an axiomatic specification of a class of types, thus a type 'a being in a class C (written 'a::C) must satisfy all axioms of C. Furthermore, type classes can be organized in a hierarchy. Thus there is the

notion of a class D being a sub class of a class C, written D < C. This is the case if all axioms of C are also provable in D. D < C

ord Isabelle/HOL already has a built-in type class ord that among others defines <= the <= symbol for orders. On top of ord we can introduce a type class reford axclass which requires reflexivity for the order relation:

```
axclass reford < ord
  reford_refl : "x <= y"
```

For types being in the type class reford we now have an antisymmetric order and should be able to proof:

```
lemma "(x::'a::reford) <= x"
```

But for now, there are no concrete types in the type class reford.

1.2 Instances

To bring life in our new type class reford we have to declare that a concrete instance type is an **instance** of our type class and we also have to define the meaning of <= over bool.

But first we prove that bool is an instance of the type class ord:

```
instance bool :: ord
  apply( intro_classes )
  done
```

intro_classes Where intro_classes is a special method for doing "instance-proofs", i.e., every proof of a type being a instance of a type class should start with applying this method. Further, we define the meaning of our order <= over bool as implication (⟶):[1]

```
defs (overloaded)
  leq_bool_def : "p <= q ≡ p ⟶ q"
```

and prove that bool is a instance of the type class reford:

```
instance bool :: reford
  apply( intro_classes )
  apply(unfold leq_bool_def )
  apply(rule imp_refl )
  done
```

[1] The (overloaded) keyword is used here because the syntax of <= is used in many different contexts and we "overload" it with our definition.

1.3 Using the Simplifier

The simplifier uses a "current simplifier set" available in a proof context. This can be modified in the ISAR-language by adding new rules (that must have the format the simplifier may process; i.e. it must be a higher-order pattern rule), deleting rules or by adding rules of a special format, e.g. splitter rules or congruence rules, which we will discuss in the future.

Examples for the syntax of the simplifier method are:

apply(simp add: A B C)
apply(simp_all del : B)
apply(simp only : A)
apply(simp addsplit : E)
apply(simp addcong: F)

2 Exercises

2.1 Exercise 33

1. Define an axiomatic class "qorder" of quasi-orderings (these are structures with an ordering symbol op <= which are reflexive and transitive).

2. Define an axiomatic subclass " linqorder " of *linear quasi-orderings* which enjoy the additional property A <= B ∨B <= A

   Define the relation:

   A ~=~ B == A <= B ∧B <= A

   on it.

3. Show that linear quasi-orderings are equivalence relations and prove the following properties (min is inherited from class ord):

   lemma min_cong: "A ~=~ B ⟹min A B ~=~ B"
   lemma linear_order_CE [dest !]:
   "¬ (A::'a::linqorder) <= B ⟹B <= A"
   lemma min_com: "min (A::'a::linqorder) B ~=~ min B A"
   lemma min_sym "min (A::'a::linqorder) B ~=~ min B A"
   lemma le_split :
   "(A::'a::linqorder) <= B ⟹¬(B <= A) ∨(A ~=~ B)"
   lemma quasi_refl : "A ~=~ A"
   lemma quasi_sym: "A ~=~ B ⟹B ~=~ A"
   lemma "⟦ A ~=~ B; B ~=~ C ⟧⟹A ~=~ C"

4. Define the ordering op <= over pairs by conjoining the ordering on components of the pairs and prove

   lemma "(a::('a::qorder * 'b::qorder)) ~=~ b ==>b ~=~ a"

Hint: Lookup the definition of the axiomatic class order in the HOL theory (`http://isabelle.in.tum.de/library/HOL/HOL.html`) and modify it!

Hint: Use simp, fast, auto!

2.2 Exercise 34

1. Prove the following set-theoretic properties only using the simplifier (not fast, not blast, not auto):

   A ∪ (B ∪ A) ⊆ A ∪ B
   A = D ⟹ A ∪ (C ∪ B) ∪ D = C ∪ B ∪ A
   F = B ⟹ A ∩ (B ∪ C) = (C ∩ A) ∪ (B ∩ A ∩ F)

2. Prove the following set-theoretic properties with methods of your choice:

   Domain r = UNIV ⟹ Id ⊆ r\={} O r
   Domain r ≠ UNIV ⟹ ∃x. (x, x) ∉ r\={} O r
   ∃ x∈ A. X ⊆ B x ⟹ X ⊆ UNION A B

Hint: For the first task, set up the simplifier such that it computes ACI normal forms.

2.3 Exercise 35

We define a (tiny) fragment of the specification language Z.[2] Begin by defining the type of relations as sets of products using the type synonym:

types ('a,'b) "<=>" = "('a*'b) set" (infixr 20)

Define the Z constructs notational equivalent:

syntax
dom :: "('a <=> 'b) => 'a set"
ran :: "('a <=> 'b) => 'b set"

[2] You can find more information about Z on the "Z Notation Website": `http://archive.comlab.ox.ac.uk/z.html`.

translations
"dom r" == "Domain r"
"ran r" == "Range r"

1. Define the following operators over sets A and B:

A <--> B	relation
A -|-> B	partial function
A ---> B	total function
A >-|-> B	partial injection
A >--> B	total injection
A -|->> B	partial surjection
A >-->> B	bijection

2. Define the operator *override* A (+) B that takes two relations and "combines" them as follows:

   a) any (x,y):A is in the override, iff x ~: dom B,

   b) any (x,y):B is in the override, iff x ~: dom B.

3. Prove:

   f : A -|-> B ⟹ f : A <--> B
   f : A ---> B ⟹ f : A -|-> B
   f : A >-|-> B ⟹ f : A -|-> B
   f : A >--> B ⟹ f : A -|->> B ⟹ f : A >-->> B
   A (+) A = A
   (A (+) B) (+) C = A (+) (B (+) C)

Hint: Use simp, fast, auto as you like.

Hint: It might be useful to define a concept like "domain restriction" S <: A (cutting down a relation A by erasing all pairs, whose first component is in a given set S).