Using Axiomatic Type Classes in Isabelle

Markus Wenzel
TU München

22 November 2007

Abstract

Isabelle offers order-sorted type classes on top of the simple types of plain Higher-Order Logic. The resulting type system is similar to that of the programming language Haskell. Its interpretation within the logic enables further application, though, apart from restricting polymorphism syntactically. In particular, the concept of *Axiomatic Type Classes* provides a useful light-weight mechanism for hierarchically-structured abstract theories. Subsequently, we demonstrate typical uses of Isabelle's axiomatic type classes to model basic algebraic structures.

This document describes axiomatic type classes using Isabelle/Isar theories, with proofs expressed via Isar proof language elements. The new theory format greatly simplifies the arrangement of the overall development, since definitions and proofs may be freely intermixed. Users who prefer tactic scripts over structured proofs do not need to fall back on separate ML scripts, though, but may refer to Isar's tactic emulation commands.

Contents

Chapter 1

Introduction

A Haskell-style type-system [1] with ordered type-classes has been present in Isabelle since 1991 already [2]. Initially, classes have mainly served as a *purely syntactic* tool to formulate polymorphic object-logics in a clean way, such as the standard Isabelle formulation of many-sorted FOL [5].

Applying classes at the *logical level* to provide a simple notion of abstract theories and instantiations to concrete ones, has been long proposed as well [3, 2]. At that time, Isabelle still lacked built-in support for these *axiomatic type classes*. More importantly, their semantics was not yet fully fleshed out (and unnecessarily complicated, too).

Since Isabelle94, actual axiomatic type classes have been an integral part of Isabelle's meta-logic. This very simple implementation is based on a straight-forward extension of traditional simply-typed Higher-Order Logic, by including types qualified by logical predicates and overloaded constant definitions (see [7] for further details).

Yet even until Isabelle99, there used to be still a fundamental methodological problem in using axiomatic type classes conveniently, due to the traditional distinction of Isabelle theory files vs. ML proof scripts. This has been finally overcome with the advent of Isabelle/Isar theories [6]: now definitions and proofs may be freely intermixed. This nicely accommodates the usual procedure of defining axiomatic type classes, proving abstract properties, defining operations on concrete types, proving concrete properties for instantiation of classes etc.

So to cut a long story short, the present version of axiomatic type classes now provides an even more useful and convenient mechanism for light-weight abstract theories, without any special technical provisions to be observed by the user.

Chapter 2

Examples

Axiomatic type classes are a concept of Isabelle's meta-logic [5, 7]. They may be applied to any object-logic that directly uses the meta type system, such as Isabelle/HOL [4]. Subsequently, we present various examples that are all formulated within HOL, except the one of §2.4 which is in FOL. See also http://isabelle.in.tum.de/library/HOL/AxClasses/ and http://isabelle.in.tum.de/library/FOL/ex/NatClass.html.

2.1 Semigroups

theory *Semigroups* **imports** *Main* **begin**

An axiomatic type class is simply a class of types that all meet certain properties, which are also called *class axioms*. Thus, type classes may be also understood as type predicates — i.e. abstractions over a single type argument $'a$. Class axioms typically contain polymorphic constants that depend on this type $'a$. These *characteristic constants* behave like operations associated with the "carrier" type $'a$.

We illustrate these basic concepts by the following formulation of semigroups.

consts
  times :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\odot$ *70*)
axclass *semigroup* $\subseteq$ *type*
  assoc: $(x \odot y) \odot z = x \odot (y \odot z)$

Above we have first declared a polymorphic constant $\odot :: 'a \Rightarrow 'a \Rightarrow 'a$ and then defined the class *semigroup* of all types τ such that $\odot :: \tau \Rightarrow \tau \Rightarrow \tau$ is indeed an associative operator. The *assoc* axiom contains exactly one type variable, which is invisible in the above presentation, though. Also note that free term variables (like x, y, z) are allowed for user convenience — conceptually all of these are bound by outermost universal quantifiers.

In general, type classes may be used to describe *structures* with exactly one carrier $'a$ and a fixed *signature*. Different signatures require different

classes. Below, class *plus-semigroup* represents semigroups $(\tau, \oplus^\tau)$, while the original *semigroup* would correspond to semigroups of the form $(\tau, \odot^\tau)$.

consts
plus :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\oplus$ *70*)
axclass *plus-semigroup* $\subseteq$ *type*
assoc: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$

Even if classes *plus-semigroup* and *semigroup* both represent semigroups in a sense, they are certainly not quite the same.

end

2.2 Basic group theory

theory *Group* **imports** *Main* **begin**

The meta-level type system of Isabelle supports *intersections* and *inclusions* of type classes. These directly correspond to intersections and inclusions of type predicates in a purely set theoretic sense. This is sufficient as a means to describe simple hierarchies of structures. As an illustration, we use the well-known example of semigroups, monoids, general groups and Abelian groups.

2.2.1 Monoids and Groups

First we declare some polymorphic constants required later for the signature parts of our structures.

consts
times :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\odot$ *70*)
invers :: $'a \Rightarrow 'a$ $((\text{-}^{-1})\ [1000]\ 999)$
one :: $'a$ (1)

Next we define class *monoid* of monoids with operations $\odot$ and 1. Note that multiple class axioms are allowed for user convenience — they simply represent the conjunction of their respective universal closures.

axclass *monoid* $\subseteq$ *type*
assoc: $(x \odot y) \odot z = x \odot (y \odot z)$
left-unit: $1 \odot x = x$
right-unit: $x \odot 1 = x$

So class *monoid* contains exactly those types τ where $\odot :: \tau \Rightarrow \tau \Rightarrow \tau$ and $1 :: \tau$ are specified appropriately, such that $\odot$ is associative and 1 is a left and right unit element for the $\odot$ operation.

Independently of *monoid*, we now define a linear hierarchy of semigroups, general groups and Abelian groups. Note that the names of class axioms are automatically qualified with each class name, so we may re-use common names such as *assoc*.

axclass *semigroup* $\subseteq$ *type*
  assoc: $(x \odot y) \odot z = x \odot (y \odot z)$

axclass *group* $\subseteq$ *semigroup*
  left-unit: $1 \odot x = x$
  left-inverse: $x^{-1} \odot x = 1$

axclass *agroup* $\subseteq$ *group*
  commute: $x \odot y = y \odot x$

Class *group* inherits associativity of $\odot$ from *semigroup* and adds two further group axioms. Similarly, *agroup* is defined as the subset of *group* such that for all of its elements τ, the operation $\odot :: \tau \Rightarrow \tau \Rightarrow \tau$ is even commutative.

2.2.2 Abstract reasoning

In a sense, axiomatic type classes may be viewed as *abstract theories*. Above class definitions gives rise to abstract axioms *assoc*, *left-unit*, *left-inverse*, *commute*, where any of these contain a type variable $'a :: c$ that is restricted to types of the corresponding class *c*. *Sort constraints* like this express a logical precondition for the whole formula. For example, *assoc* states that for all τ, provided that $\tau ::$ *semigroup*, the operation $\odot :: \tau \Rightarrow \tau \Rightarrow \tau$ is associative.

From a technical point of view, abstract axioms are just ordinary Isabelle theorems, which may be used in proofs without special treatment. Such "abstract proofs" usually yield new "abstract theorems". For example, we may now derive the following well-known laws of general groups.

theorem *group-right-inverse*: $x \odot x^{-1} = (1::'a::group)$
proof −
  have $x \odot x^{-1} = 1 \odot (x \odot x^{-1})$
    by (*simp only*: *group-class.left-unit*)
  also have $\ldots = 1 \odot x \odot x^{-1}$
    by (*simp only*: *semigroup-class.assoc*)
  also have $\ldots = (x^{-1})^{-1} \odot x^{-1} \odot x \odot x^{-1}$
    by (*simp only*: *group-class.left-inverse*)
  also have $\ldots = (x^{-1})^{-1} \odot (x^{-1} \odot x) \odot x^{-1}$
    by (*simp only*: *semigroup-class.assoc*)

```
  also have ... = (x⁻¹)⁻¹ ⊙ 1 ⊙ x⁻¹
    by (simp only: group-class.left-inverse)
  also have ... = (x⁻¹)⁻¹ ⊙ (1 ⊙ x⁻¹)
    by (simp only: semigroup-class.assoc)
  also have ... = (x⁻¹)⁻¹ ⊙ x⁻¹
    by (simp only: group-class.left-unit)
  also have ... = 1
    by (simp only: group-class.left-inverse)
  finally show ?thesis .
qed
```

With *group-right-inverse* already available, *group-right-unit* is now established much easier.

```
theorem group-right-unit: x ⊙ 1 = (x::'a::group)
proof −
  have x ⊙ 1 = x ⊙ (x⁻¹ ⊙ x)
    by (simp only: group-class.left-inverse)
  also have ... = x ⊙ x⁻¹ ⊙ x
    by (simp only: semigroup-class.assoc)
  also have ... = 1 ⊙ x
    by (simp only: group-right-inverse)
  also have ... = x
    by (simp only: group-class.left-unit)
  finally show ?thesis .
qed
```

Abstract theorems may be instantiated to only those types τ where the appropriate class membership $\tau :: c$ is known at Isabelle's type signature level. Since we have *agroup* $\subseteq$ *group* $\subseteq$ *semigroup* by definition, all theorems of *semigroup* and *group* are automatically inherited by *group* and *agroup*.

2.2.3 Abstract instantiation

From the definition, the *monoid* and *group* classes have been independent. Note that for monoids, *right-unit* had to be included as an axiom, but for groups both *right-unit* and *right-inverse* are derivable from the other axioms. With *group-right-unit* derived as a theorem of group theory (see page 5), we may now instantiate *monoid* $\subseteq$ *semigroup* and *group* $\subseteq$ *monoid* properly as follows (cf. figure 2.1).

```
instance monoid ⊆ semigroup
proof
  fix x y z :: 'a::monoid
```

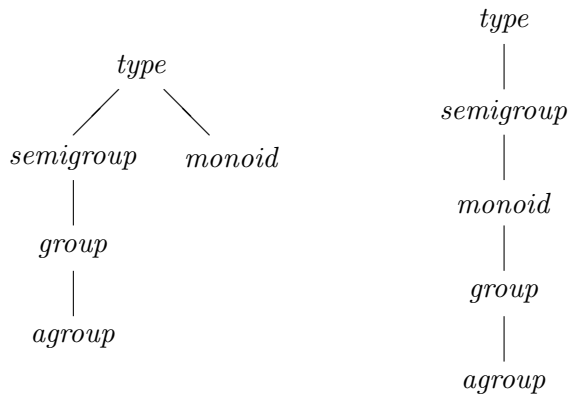

Figure 2.1: Monoids and groups: according to definition, and by proof

```
  show x ⊙ y ⊙ z = x ⊙ (y ⊙ z)
    by (rule monoid-class.assoc)
qed

instance group ⊆ monoid
proof
  fix x y z :: 'a::group
  show x ⊙ y ⊙ z = x ⊙ (y ⊙ z)
    by (rule semigroup-class.assoc)
  show 1 ⊙ x = x
    by (rule group-class.left-unit)
  show x ⊙ 1 = x
    by (rule group-right-unit)
qed
```

The **instance** command sets up an appropriate goal that represents the class inclusion (or type arity, see §2.2.4) to be proven (see also [6]). The initial proof step causes back-chaining of class membership statements wrt. the hierarchy of any classes defined in the current theory; the effect is to reduce to the initial statement to a number of goals that directly correspond to any class axioms encountered on the path upwards through the class hierarchy.

2.2.4 Concrete instantiation

So far we have covered the case of the form **instance** $c_1 \subseteq c_2$, namely *abstract instantiation* — c_1 is more special than c_1 and thus an instance of c_2. Even more interesting for practical applications are *concrete instantiations* of

axiomatic type classes. That is, certain simple schemes $(\alpha_1, \ldots, \alpha_n)\ t :: c$ of class membership may be established at the logical level and then transferred to Isabelle's type signature level.

As a typical example, we show that type *bool* with exclusive-or as $\odot$ operation, identity as $^{-1}$, and *False* as 1 forms an Abelian group.

defs (**overloaded**)
times-bool-def: $x \odot y \equiv x \neq (y::bool)$
inverse-bool-def: $x^{-1} \equiv x::bool$
unit-bool-def: $1 \equiv False$

It is important to note that above **defs** are just overloaded meta-level constant definitions, where type classes are not yet involved at all. This form of constant definition with overloading (and optional recursion over the syntactic structure of simple types) are admissible as definitional extensions of plain HOL [7]. The Haskell-style type system is not required for overloading. Nevertheless, overloaded definitions are best applied in the context of type classes.

Since we have chosen above **defs** of the generic group operations on type *bool* appropriately, the class membership *bool* :: *agroup* may be now derived as follows.

instance *bool* :: *agroup*
proof (*intro-classes*,
  unfold times-bool-def inverse-bool-def unit-bool-def)
 fix *x y z*
 show $((x \neq y) \neq z) = (x \neq (y \neq z))$ **by** *blast*
 show $(False \neq x) = x$ **by** *blast*
 show $(x \neq x) = False$ **by** *blast*
 show $(x \neq y) = (y \neq x)$ **by** *blast*
qed

The result of an **instance** statement is both expressed as a theorem of Isabelle's meta-logic, and as a type arity of the type signature. The latter enables type-inference system to take care of this new instance automatically.

We could now also instantiate our group theory classes to many other concrete types. For example, *int* :: *agroup* (e.g. by defining $\odot$ as addition, $^{-1}$ as negation and 1 as zero) or *list* :: (*type*) *semigroup* (e.g. if $\odot$ is defined as list append). Thus, the characteristic constants $\odot$, $^{-1}$, 1 really become overloaded, i.e. have different meanings on different types.

2.2.5 Lifting and Functors

As already mentioned above, overloading in the simply-typed HOL systems may include recursion over the syntactic structure of types. That is, definitional equations $c^\tau \equiv t$ may also contain constants of name c on the right-hand side — if these have types that are structurally simpler than τ.

This feature enables us to *lift operations*, say to Cartesian products, direct sums or function spaces. Subsequently we lift $\odot$ component-wise to binary products $'a \times 'b$.

defs (**overloaded**)
times-prod-def: $p \odot q \equiv (fst\ p \odot fst\ q,\ snd\ p \odot snd\ q)$

It is very easy to see that associativity of $\odot$ on $'a$ and $\odot$ on $'b$ transfers to $\odot$ on $'a \times 'b$. Hence the binary type constructor $\odot$ maps semigroups to semigroups. This may be established formally as follows.

instance $*$ :: (*semigroup*, *semigroup*) *semigroup*
proof (*intro-classes*, *unfold times-prod-def*)
fix $p\ q\ r$ :: $'a$::*semigroup* $\times$ $'b$::*semigroup*
show
$(fst\ (fst\ p \odot fst\ q,\ snd\ p \odot snd\ q) \odot fst\ r,$
$snd\ (fst\ p \odot fst\ q,\ snd\ p \odot snd\ q) \odot snd\ r) =$
$(fst\ p \odot fst\ (fst\ q \odot fst\ r,\ snd\ q \odot snd\ r),$
$snd\ p \odot snd\ (fst\ q \odot fst\ r,\ snd\ q \odot snd\ r))$
by (*simp add*: *semigroup-class.assoc*)
qed

Thus, if we view class instances as "structures", then overloaded constant definitions with recursion over types indirectly provide some kind of "functors" — i.e. mappings between abstract theories.

end

2.3 Syntactic classes

theory *Product* **imports** *Main* **begin**

There is still a feature of Isabelle's type system left that we have not yet discussed. When declaring polymorphic constants $c :: \sigma$, the type variables occurring in σ may be constrained by type classes (or even general sorts) in an arbitrary way. Note that by default, in Isabelle/HOL the declaration $\odot :: 'a \Rightarrow 'a \Rightarrow 'a$ is actually an abbreviation for $\odot :: 'a$::*type* $\Rightarrow 'a \Rightarrow 'a$ Since class *type* is the universal class of HOL, this is not really a constraint at all.

The *product* class below provides a less degenerate example of syntactic type classes.

axclass
product $\subseteq$ *type*
consts
product :: $'a$::*product* $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ (**infixl** $\odot$ *70*)

Here class *product* is defined as subclass of *type* without any additional axioms. This effects in logical equivalence of *product* and *type*, as is reflected by the trivial introduction rule generated for this definition.

So what is the difference of declaring $\odot$ :: $'a$::*product* $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ vs. declaring $\odot$ :: $'a$::*type* $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ anyway? In this particular case where *product* $\equiv$ *type*, it should be obvious that both declarations are the same from the logic's point of view. It even makes the most sense to remove sort constraints from constant declarations, as far as the purely logical meaning is concerned [7].

On the other hand there are syntactic differences, of course. Constants $\odot$ on some type τ are rejected by the type-checker, unless the arity τ :: *product* is part of the type signature. In our example, this arity may be always added when required by means of an **instance** with the default proof **..**.

Thus, we may observe the following discipline of using syntactic classes. Overloaded polymorphic constants have their type arguments restricted to an associated (logically trivial) class *c*. Only immediately before *specifying* these constants on a certain type τ do we instantiate τ :: *c*.

This is done for class *product* and type *bool* as follows.

instance *bool* :: *product* **..**
defs (**overloaded**)
product-bool-def: $x \odot y \equiv x \wedge y$

The definition *prod-bool-def* becomes syntactically well-formed only after the arity *bool* :: *product* is made known to the type checker.

It is very important to see that above **defs** are not directly connected with **instance** at all! We were just following our convention to specify $\odot$ on *bool* after having instantiated *bool* :: *product*. Isabelle does not require these definitions, which is in contrast to programming languages like Haskell [1].

While Isabelle type classes and those of Haskell are almost the same as far as type-checking and type inference are concerned, there are important semantic differences. Haskell classes require their instances to *provide operations* of certain *names*. Therefore, its `instance` has a `where` part that tells the system what these "member functions" should be.

This style of `instance` would not make much sense in Isabelle's meta-logic, because there is no internal notion of "providing operations" or even "names of functions".

end

2.4 Defining natural numbers in FOL

theory *NatClass* **imports** *FOL* **begin**

Axiomatic type classes abstract over exactly one type argument. Thus, any *axiomatic* theory extension where each axiom refers to at most one type variable, may be trivially turned into a *definitional* one.

We illustrate this with the natural numbers in Isabelle/FOL.[1]

consts
$zero :: 'a \quad (0)$
$Suc :: 'a \Rightarrow 'a$
$rec :: 'a \Rightarrow 'a \Rightarrow ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a$

axclass $nat \subseteq term$
induct: $P(0) \Longrightarrow (\bigwedge x.\ P(x) \Longrightarrow P(Suc(x))) \Longrightarrow P(n)$
Suc-inject: $Suc(m) = Suc(n) \Longrightarrow m = n$
Suc-neq-0: $Suc(m) = 0 \Longrightarrow R$
rec-0: $rec(0,\ a,\ f) = a$
rec-Suc: $rec(Suc(m),\ a,\ f) = f(m,\ rec(m,\ a,\ f))$

constdefs
$add :: 'a::nat \Rightarrow 'a \Rightarrow 'a \quad (\textbf{infixl}\ +\ 60)$
$m + n \equiv rec(m,\ n,\ \lambda x\ y.\ Suc(y))$

This is an abstract version of the plain *Nat* theory in FOL.[2] Basically, we have just replaced all occurrences of type *nat* by $'a$ and used the natural number axioms to define class *nat*. There is only a minor snag, that the original recursion operator *rec* had to be made monomorphic.

Thus class *nat* contains exactly those types τ that are isomorphic to "the" natural numbers (with signature 0, *Suc*, *rec*).

What we have done here can be also viewed as *type specification*. Of course, it still remains open if there is some type at all that meets the class axioms. Now a very nice property of axiomatic type classes is that abstract

[1] See also http://isabelle.in.tum.de/library/FOL/ex/NatClass.html
[2] See http://isabelle.in.tum.de/library/FOL/ex/Nat.html

reasoning is always possible — independent of satisfiability. The meta-logic won't break, even if some classes (or general sorts) turns out to be empty later — "inconsistent" class definitions may be useless, but do not cause any harm.

Theorems of the abstract natural numbers may be derived in the same way as for the concrete version. The original proof scripts may be re-used with some trivial changes only (mostly adding some type constraints).

end

Bibliography

[1] Paul Hudak, Simon Peyton Jones, and Philip Wadler. Report on the programming language Haskell: A non-strict, purely functional language. *SIGPLAN Notices*, 27(5), May 1992. Version 1.2.

[2] T. Nipkow. Order-sorted polymorphism in Isabelle. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 164–188. Cambridge University Press, 1993.

[3] Tobias Nipkow. Axiomatic type classes (in Isabelle). Presentation at the workshop *Types for Proof and Programs*, Nijmegen, 1993.

[4] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle's Logics: HOL*. http://isabelle.in.tum.de/doc/logics-HOL.pdf.

[5] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer, 1994. LNCS 828.

[6] Markus Wenzel. *The Isabelle/Isar Reference Manual*. http://isabelle.in.tum.de/doc/isar-ref.pdf.

[7] Markus Wenzel. Type classes and overloading in higher-order logic. In Elsa L. Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics: TPHOLs '97*, volume 1275 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[8] Markus Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics: TPHOLs '99*, volume 1690 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.