The Isabelle/HOL type-class hierarchy

Florian Haftmann

12 December 2016

Abstract

This primer introduces corner stones of the Isabelle/HOL type-class hierarchy and gives some insights into its internal organization.

1 Introduction

The Isabelle/HOL type-class hierarchy entered the stage in a quite ancient era – first related *NEWS* entries date back to release Isabelle99-1. Since then, there have been ongoing modifications and additions, leading to (Isabelle2016) more than 180 type-classes. This sheer complexity makes access and understanding of that type-class hierarchy difficult and involved, let alone maintenance.

The purpose of this primer is to shed some light on this, not only on the mere ingredients, but also on the design principles which have evolved and proven useful over time.

2 Foundations

2.1 Locales and type classes

Some familiarity with the Isabelle module system is assumed: defining locales and type-classes, interpreting locales and instantiating type-classes, adding relationships between locales (**sublocale**) and type-classes (**subclass**). Handy introductions are the respective tutorials [1] [2].

2.2 Strengths and restrictions of type classes

The primary motivation for using type classes in Isabelle/HOL always have been numerical types, which form an inclusion chain:

$$nat \sqsubset int \sqsubset rat \sqsubset real \sqsubset complex$$

The inclusion $\sqsubset$ means that any value of the numerical type to the left hand side mathematically can be transferred to the numerical type on the right hand side.

How to accomplish this given the quite restrictive type system of Isabelle/HOL? Paulson [4] explains that each numerical type has some characteristic properties which define an characteristic algebraic structure; $\sqsubset$ then corresponds to specialization of the corresponding characteristic algebraic structures. These algebraic structures are expressed using algebraic type classes and embeddings of numerical types into them:

$$\begin{array}{lccc}
\textit{of-nat} :: & \textit{nat} & \Rightarrow & {}'a::\textit{semiring-}1 \\
 & \sqcap & & \uparrow \\
\textit{of-int} :: & \textit{int} & \Rightarrow & {}'a::\textit{ring-}1 \\
 & \sqcap & & \uparrow \\
\textit{of-rat} :: & \textit{rat} & \Rightarrow & {}'a::\textit{field-char-}0 \\
 & \sqcap & & \uparrow \\
\textit{of-real} :: & \textit{real} & \Rightarrow & {}'a::\textit{real-algebra-}1 \\
 & \sqcap & & \\
 & \textit{complex} & &
\end{array}$$

$d \leftarrow c$ means that c is subclass of d. Hence each characteristic embedding *of-num* can transform any value of type *num* to any numerical type further up in the inclusion chain.

This canonical example exhibits key strengths of type classes:

- Sharing of operations and facts among different types, hence also sharing of notation and names: there is only one plus operation using infix syntax $+$, only one zero written 0, and neutrality ($\bigwedge a::{}'a::\textit{monoid-add}.\ 0 + a = a$ and $\bigwedge a::{}'a::\textit{monoid-add}.\ a + 0 = a$) is referred to uniformly by names *add*-0-*left* and *add*-0-*right*.

- Proof tool setups are shared implicitly: *add*-0 and *add*-0-*right* are simplification rules by default.

- Hence existing proofs about particular numerical types are often easy to generalize to algebraic structures, given that they do not depend on specific properties of those numerical types.

Considerable restrictions include:

- Type class operations are restricted to one type parameter; this is insufficient e.g. for expressing a unified power operation adequately (see §**??**).

- Parameters are fixed over the whole type class hierarchy and cannot be refined in specific situations: think of integral domains with a predicate *is-unit*; for natural numbers, this degenerates to the much simpler *HOL.equal* (1::*nat*) but facts refer to *is-unit* nonetheless.

- Type classes are not apt for meta-theory. There is no practically usable way to express that the units of an integral domain form a multiplicative group using type classes. But see session *HOL−Algebra* which provides locales with an explicit carrier.

2.3 Navigating the hierarchy

An indispensable tool to inspect the class hierarchy is **class-deps** which displays the graph of classes, optionally showing the logical content for each class also. Optional parameters restrict the graph to a particular segment which is useful to get a graspable view. See the Isar reference manual [5] for details.

3 The hierarchy

3.1 Syntactic type classes

At the top of the hierarchy there are a couple of syntactic type classes, ie. classes with operations but with no axioms, most notably:

- **class** *plus* with $(a::'a::plus) + b$
- **class** *zero* with $0::'a::zero$
- **class** *times* with $(a::'a::times) * b$
- **class** *one* with $1::'a::one$

Before the introduction of the **class** statement in Isabelle [3] it was impossible to define operations with associated axioms in the same class, hence there were always pairs of syntactic and logical type classes. This restriction is lifted nowadays, but there are still reasons to maintain syntactic type classes:

- Syntactic type classes allow generic notation to be used regardless of a particular logical interpretation; e.g. although multiplication $*$ is usually associative, there are examples where it is not (e.g. octonions), and leaving $*$ without axioms allows to re-use this syntax by means of type class instantiation also for such exotic examples.
- Type classes might share operations but not necessarily axioms on them, e.g. *gcd* (see §**??**). Hence their common base is a syntactic type class.

However syntactic type classes should only be used with striking cause. Otherwise there is risk for confusion if the notation suggests properties which do not hold without particular constraints. This can be illustrated using numerals (see §**??**): $2 + 2 = 4$ is provable without further ado, and this also meets the typical expectation towards a numeral notation; in more ancient releases numerals were purely syntactic and $2 + 2 = 4$ was not provable without particular type constraints.

3.2 Additive and multiplicative semigroups and monoids

In common literature, notation for semigroups and monoids is either multiplicative ($*$, 1) or additive ($+$, 0) with underlying properties isomorphic. In Isabelle/HOL, this is accomplished using the following abstract setup:

- A *semigroup* introduces an abstract binary associative operation.
- A *monoid* is an extension of *semigroup* with a neutral element.
- Both *semigroup* and *monoid* provide dedicated syntax for their operations ($\boldsymbol{*}$, $\mathbf{1}$). This syntax is not visible on the global theory level but only for abstract reasoning inside the respective locale.
- Concrete global syntax is added building on existing syntactic type classes §3.1 using the following classes:
  - **class** *semigroup-mult* = *times*
  - **class** *monoid-mult* = *one* + *semigroup-mult*
  - **class** *semigroup-add* = *plus*
  - **class** *monoid-add* = *zero* + *semigroup-add*

  Locales *semigroup* and *monoid* are interpreted (using **sublocale**) into their corresponding type classes, with prefixes *add* and *mult*; hence facts derived in *semigroup* and *monoid* are propagated simultaneously to *both* using a consistent naming policy, ie.

  - *semigroup.assoc*: $\bigwedge f\ (a::'a::type)\ b\ c.\ semigroup\ f \Longrightarrow f\ (f\ a\ b)\ c = f\ a\ (f\ b\ c)$
  - *mult.assoc*: $\bigwedge(a::'a::semigroup\text{-}mult)\ b\ c.\ a * b * c = a * (b * c)$
  - *add.assoc*: $\bigwedge(a::'a::semigroup\text{-}add)\ b\ c.\ a + b + c = a + (b + c)$
  - *monoid.right-neutral*: $\bigwedge f\ (z::'a::type)\ a.\ monoid\ f\ z \Longrightarrow f\ a\ z = a$
  - *mult.right-neutral*: $\bigwedge a::'a::monoid\text{-}mult.\ a * 1 = a$
  - *add.right-neutral*: $\bigwedge a::'a::monoid\text{-}add.\ a + 0 = a$
- Note that the syntax in *semigroup* and *monoid* is bold; this avoids clashes when writing properties inside one of these locales in presence of that global concrete type class syntax.

That hierarchy extends in a straightforward manner to abelian semigroups and commutative monoids[1]:

- Locales *abel-semigroup* and *comm-monoid* add commutativity as property.
- Concrete syntax emerges through
  - **class** *ab-semigroup-add* = *semigroup-add*
  - **class** *ab-semigroup-mult* = *semigroup-mult*
  - **class** *comm-monoid-add* = *zero* + *ab-semigroup-add*
  - **class** *comm-monoid-mult* = *one* + *ab-semigroup-mult*

  and corresponding interpretation of the locales above, yielding

  - *abel-semigroup.commute*: $\bigwedge f\ (a::'a::type)\ b.\ abel\text{-}semigroup\ f \Longrightarrow f\ a\ b = f\ b\ a$
  - *mult.commute*: $\bigwedge(a::'a::ab\text{-}semigroup\text{-}mult)\ b.\ a * b = b * a$
  - *add.commute*: $\bigwedge(a::'a::ab\text{-}semigroup\text{-}add)\ b.\ a + b = b + a$

Named collection of theorems Locale interpretation interacts smoothly with named collections of theorems as introduced by command **named-theorems**. In our example, rules concerning associativity and commutativity are no simplification rules by default since they desired orientation may vary depending on the situation. However, there is a collection *ac-simps* where facts *abel-semigroup.assoc*, *abel-semigroup.commute* and *abel-semigroup.left-commute* are declared as members. Due to interpretation, also *mult.assoc*, *mult.commute* and *mult.left-commute* are also members of *ac-simps*, as any corresponding facts stemming from interpretation of *abel-semigroup*. Hence adding *ac-simps* to the simplification rules for a single method call uses all associativity and commutativity rules known by means of interpretation.

[1]The designation *abelian* is quite standard concerning (semi)groups, but not for monoids

3.3 Additive and multiplicative groups

The hierarchy for inverse group operations takes into account that there are weaker algebraic structures with only a partially inverse operation. E. g. the natural numbers have bounded subtraction $(m::nat) - (n::nat)$ which is only an inverse operation if $(n::nat) \leq (m::nat)$; unary minus $-$ is pointless on the natural numbers.

Hence for both additive and multiplicative notation there are syntactic classes for inverse operations, both unary and binary:

- **class** *minus* with $(a::'a::minus) - b$
- **class** *uminus* with $- a::'a::uminus$
- **class** *divide* with $(a::'a::divide)$ *div* b
- **class** *inverse* = *divide* with *inverse* $a::'a::inverse$ and $(a::'a::inverse) / b$

Here *inverse* specializes the "partial" syntax a *div* b to the more specific a / b.

Semantic properties are added by

- **class** *cancel-ab-semigroup-add* = *ab-semigroup-add* + *minus*
- **class** *cancel-comm-monoid-add* = *cancel-ab-semigroup-add* + *comm-monoid-add*

which specify a minimal binary partially inverse operation as

- *add-diff-cancel-left'*: $\bigwedge(a::'a::cancel\text{-}ab\text{-}semigroup\text{-}add)\ b.\ a + b - a = b$
- *diff-diff-add*: $\bigwedge(a::'a::cancel\text{-}ab\text{-}semigroup\text{-}add)\ b\ c.\ a - b - c = a - (b + c)$

which in turn allow to derive facts like

- *add-left-imp-eq*: $\bigwedge(a::'a::cancel\text{-}semigroup\text{-}add)\ b\ c.\ a + b = a + c \Longrightarrow b = c$

The total inverse operation is established as follows:

- Locale *group* extends the abstract hierarchy with the inverse operation.
- The concrete additive inverse operation emerges through

- **class** *group-add* = *minus* + *uminus* + *monoid-add* (in *Groups*)

- **class** *ab-group-add* = *minus* + *uminus* + *comm-monoid-add* (in *Groups*)

  and corresponding interpretation of locale *group*, yielding e.g.

- *group.left-inverse*: $\bigwedge f\ (z{::}'a{::}type)\ inverse\ a.\ group\ f\ z\ inverse \Longrightarrow f\ (inverse\ a)\ a = z$

- *add.left-inverse*: $\bigwedge a{::}'a{::}group\text{-}add.\ -\ a + a = 0$

There is no multiplicative counterpart. Why? In rings, the multiplicative group excludes the zero element, hence the inverse operation is not total. See further §**??**.

Mitigating against redundancy by default simplification rules Inverse operations imposes some redundancy on the type class hierarchy: in a group with a total inverse operation, the unary operation is simpler and more primitive than the binary one; but we cannot eliminate the binary one in favour of a mere syntactic abbreviation since the binary one is vital to express a partial inverse operation.

This is mitigated by providing suitable default simplification rules: expression involving the unary inverse operation are simplified to binary inverse operation whenever appropriate. The rationale is that simplification is a central device in explorative proving, where proof obligation remaining after certain default proof steps including simplification are inspected to get an idea what is missing to finish a proof. When preferable normal forms are encoded into default simplification rules, proof obligations after simplification are normalized and hence more proof-friendly.

References

[1] Clemens Ballarin. *Tutorial to Locales and Locale Interpretation.* http://isabelle.in.tum.de/doc/locales.pdf.

[2] Florian Haftmann. *Haskell-style type classes with Isabelle/Isar.* http://isabelle.in.tum.de/doc/classes.pdf.

[3] Florian Haftmann and Makarius Wenzel. Constructive type classes in Isabelle. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs, TYPES 2006*, volume 4502 of *LNCS*. Springer, 2007.

[4] Lawrence C. Paulson. Organizing numerical theories using axiomatic type classes. *Journal of Automated Reasoning*, 33(1):29–49, 2004.

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