Haskell-style type classes with Isabelle/Isar

Florian Haftmann

8 June 2008

Abstract

This tutorial introduces the look-and-feel of Isar type classes to the end-user; Isar type classes are a convenient mechanism for organizing specifications, overcoming some drawbacks of raw axiomatic type classes. Essentially, they combine an operational aspect (in the manner of Haskell) with a logical aspect, both managed uniformly.

Chapter 1

Haskell-style classes with Isabelle/Isar

1.1 Introduction

Type classes were introduces by Wadler and Blott [9] into the Haskell language, to allow for a reasonable implementation of overloading[1]. As a canonical example, a polymorphic equality function $eq :: \alpha \Rightarrow \alpha \Rightarrow bool$ which is overloaded on different types for α, which is achieved by splitting introduction of the *eq* function from its overloaded definitions by means of *class* and *instance* declarations:

class eq where[2]
  $eq :: \alpha \Rightarrow \alpha \Rightarrow bool$

instance nat :: *eq where*
  $eq\ 0\ 0 = True$
  $eq\ 0\ \text{-} = False$
  $eq\ \text{-}\ 0 = False$
  $eq\ (Suc\ n)\ (Suc\ m) = eq\ n\ m$

instance $(\alpha{::}eq,\ \beta{::}eq)$ *pair* :: *eq where*
  $eq\ (x1,\ y1)\ (x2,\ y2) = eq\ x1\ x2 \wedge eq\ y1\ y2$

class ord extends eq where
  $less\text{-}eq :: \alpha \Rightarrow \alpha \Rightarrow bool$
  $less :: \alpha \Rightarrow \alpha \Rightarrow bool$

Type variables are annotated with (finitly many) classes; these annotations are assertions that a particular polymorphic type provides definitions for overloaded functions.

Indeed, type classes not only allow for simple overloading but form a generic calculus, an instance of order-sorted algebra [7, 6, 10].

[1]throughout this tutorial, we are referring to classical Haskell 1.0 type classes, not considering later additions in expressiveness

[2]syntax here is a kind of isabellized Haskell

From a software enigineering point of view, type classes correspond to interfaces in object-oriented languages like Java; so, it is naturally desirable that type classes do not only provide functions (class parameters) but also state specifications implementations must obey. For example, the *class eq* above could be given the following specification, demanding that *class eq* is an equivalence relation obeying reflexivity, symmetry and transitivity:

class eq where
  $eq :: \alpha \Rightarrow \alpha \Rightarrow bool$
satisfying
  refl: $eq\ x\ x$
  sym: $eq\ x\ y \leftrightarrow eq\ x\ y$
  trans: $eq\ x\ y \wedge eq\ y\ z \longrightarrow eq\ x\ z$

From a theoretic point of view, type classes are leightweight modules; Haskell type classes may be emulated by SML functors [1]. Isabelle/Isar offers a discipline of type classes which brings all those aspects together:

1. specifying abstract parameters together with corresponding specifications,
2. instantating those abstract parameters by a particular type
3. in connection with a "less ad-hoc" approach to overloading,
4. with a direct link to the Isabelle module system (aka locales [4]).

Isar type classes also directly support code generation in a Haskell like fashion.

This tutorial demonstrates common elements of structured specifications and abstract reasoning with type classes by the algebraic hierarchy of semigroups, monoids and groups. Our background theory is that of Isabelle/HOL [8], for which some familiarity is assumed.

Here we merely present the look-and-feel for end users. Internally, those are mapped to more primitive Isabelle concepts. See [3] for more detail.

1.2 A simple algebra example

1.2.1 Class definition

Depending on an arbitrary type α, class *semigroup* introduces a binary operator $\circ$ that is assumed to be associative:

class *semigroup* = *type* +

fixes *mult* :: $\alpha \Rightarrow \alpha \Rightarrow \alpha$ (**infixl** $\circ$ 70)
assumes *assoc*: $(x \circ y) \circ z = x \circ (y \circ z)$

This **class** specification consists of two parts: the *operational* part names the class parameter (**fixes**), the *logical* part specifies properties on them (**assumes**). The local **fixes** and **assumes** are lifted to the theory toplevel, yielding the global parameter *mult* :: α::*semigroup* $\Rightarrow \alpha \Rightarrow \alpha$ and the global theorem *semigroup.assoc*: $\bigwedge x\ y\ z$ :: α::*semigroup*. $(x \circ y) \circ z = x \circ (y \circ z)$.

1.2.2 Class instantiation

The concrete type *int* is made a *semigroup* instance by providing a suitable definition for the class parameter *mult* and a proof for the specification of *assoc*. This is accomplished by the **instantiation** target:

instantiation *int* :: *semigroup*
begin

definition
mult-int-def: $i \circ j = i + (j::int)$

instance proof
fix *i j k* :: *int* **have** $(i + j) + k = i + (j + k)$ **by** *simp*
then show $(i \circ j) \circ k = i \circ (j \circ k)$
unfolding *mult-int-def* **.**
qed

end

instantiation allows to define class parameters at a particular instance using common specification tools (here, **definition**). The concluding **instance** opens a proof that the given parameters actually conform to the class specification. Note that the first proof step is the *default* method, which for such instance proofs maps to the *intro-classes* method. This boils down an instance judgement to the relevant primitive proof goals and should conveniently always be the first method applied in an instantiation proof.

From now on, the type-checker will consider *int* as a *semigroup* automatically, i.e. any general results are immediately available on concrete instances. Another instance of *semigroup* are the natural numbers:

instantiation *nat* :: *semigroup*
begin

primrec *mult-nat* **where**

$(0{::}nat) \circ n = n$
$\mid Suc\ m \circ n = Suc\ (m \circ n)$

instance proof
fix $m\ n\ q :: nat$
show $m \circ n \circ q = m \circ (n \circ q)$
by $(induct\ m)\ auto$
qed

end

Note the occurence of the name *mult-nat* in the primrec declaration; by default, the local name of a class operation f to instantiate on type constructor κ are mangled as f-κ. In case of uncertainty, these names may be inspected using the **print-context** command or the corresponding ProofGeneral button.

1.2.3 Lifting and parametric types

Overloaded definitions giving on class instantiation may include recursion over the syntactic structure of types. As a canonical example, we model product semigroups using our simple algebra:

instantiation $* :: (semigroup, semigroup)\ semigroup$
begin

definition
mult-prod-def: $p_1 \circ p_2 = (fst\ p_1 \circ fst\ p_2,\ snd\ p_1 \circ snd\ p_2)$

instance proof
fix $p_1\ p_2\ p_3 :: \alpha{::}semigroup \times \beta{::}semigroup$
show $p_1 \circ p_2 \circ p_3 = p_1 \circ (p_2 \circ p_3)$
unfolding *mult-prod-def* **by** $(simp\ add{:}\ assoc)$
qed

end

Associativity from product semigroups is established using the definition of $\circ$ on products and the hypothetical associativety of the type components; these hypothesis are facts due to the *semigroup* constraints imposed on the type components by the *instance* proposition. Indeed, this pattern often occurs with parametric types and type classes.

1.2.4 Subclassing

We define a subclass *monoidl* (a semigroup with a left-hand neutral) by extending *semigroup* with one additional parameter *neutral* together with its property:

class *monoidl* = *semigroup* +
  fixes *neutral* :: α (**1**)
  assumes *neutl*: **1** $\circ$ $x = x$

Again, we prove some instances, by providing suitable parameter definitions and proofs for the additional specifications. Obverve that instantiations for types with the same arity may be simultaneous:

instantiation *nat* **and** *int* :: *monoidl*
begin

definition
  neutral-nat-def: **1** = (0::*nat*)

definition
  neutral-int-def: **1** = (0::*int*)

instance proof
  fix *n* :: *nat*
  show **1** $\circ$ $n = n$
    unfolding *neutral-nat-def* **by** *simp*
next
  fix *k* :: *int*
  show **1** $\circ$ $k = k$
    unfolding *neutral-int-def mult-int-def* **by** *simp*
qed

end

instantiation * :: (*monoidl*, *monoidl*) *monoidl*
begin

definition
  neutral-prod-def: **1** = (**1**, **1**)

instance proof
  fix *p* :: α::*monoidl* $\times$ β::*monoidl*
  show **1** $\circ$ $p = p$
    unfolding *neutral-prod-def mult-prod-def* **by** (*simp add*: *neutl*)

```
  qed

end
```

Fully-fledged monoids are modelled by another subclass which does not add new parameters but tightens the specification:

```
  class monoid = monoidl +
    assumes neutr: x ∘ 1 = x

  instantiation nat and int :: monoid
  begin

  instance proof
    fix n :: nat
    show n ∘ 1 = n
      unfolding neutral-nat-def by (induct n) simp-all
  next
    fix k :: int
    show k ∘ 1 = k
      unfolding neutral-int-def mult-int-def by simp
  qed

  end

  instantiation ∗ :: (monoid, monoid) monoid
  begin

  instance proof
    fix p :: α::monoid × β::monoid
    show p ∘ 1 = p
      unfolding neutral-prod-def mult-prod-def by (simp add: neutr)
  qed

  end
```

To finish our small algebra example, we add a *group* class with a corresponding instance:

```
  class group = monoidl +
    fixes inverse :: α ⇒ α    ((-⁻¹) [1000] 999)
    assumes invl: x⁻¹ ∘ x = 1

  instantiation int :: group
  begin
```

definition
  inverse-int-def: $i^{-1} = - (i::int)$

instance proof
  fix $i :: int$
  have $-i + i = 0$ **by** *simp*
  then show $i^{-1} \circ i = \mathbf{1}$
    unfolding *mult-int-def neutral-int-def inverse-int-def* **.**
qed

end

1.3 Type classes as locales

1.3.1 A look behind the scene

The example above gives an impression how Isar type classes work in practice. As stated in the introduction, classes also provide a link to Isar's locale system. Indeed, the logical core of a class is nothing else than a locale:

class *idem* = *type* +
  fixes $f :: \alpha \Rightarrow \alpha$
  assumes *idem*: $f\ (f\ x) = f\ x$

essentially introduces the locale

locale *idem* =
  fixes $f :: \alpha \Rightarrow \alpha$
  assumes *idem*: $f\ (f\ x) = f\ x$

together with corresponding constant(s):

consts $f :: \alpha \Rightarrow \alpha$

The connection to the type system is done by means of a primitive axclass

axclass *idem* < *type*
  idem: $f\ (f\ x) = f\ x$

together with a corresponding interpretation:

interpretation *idem-class*:
  idem $[f :: (\alpha::idem) \Rightarrow \alpha]$
by *unfold-locales* (*rule idem*)

This give you at hand the full power of the Isabelle module system; conclusions in locale *idem* are implicitly propagated to class *idem*.

1.3.2 Abstract reasoning

Isabelle locales enable reasoning at a general level, while results are implicitly transferred to all instances. For example, we can now establish the *left-cancel* lemma for groups, which states that the function $(x \circ)$ is injective:

lemma (**in** *group*) *left-cancel*: $x \circ y = x \circ z \leftrightarrow y = z$
proof
  assume $x \circ y = x \circ z$
  then have $x^{-1} \circ (x \circ y) = x^{-1} \circ (x \circ z)$ **by** *simp*
  then have $(x^{-1} \circ x) \circ y = (x^{-1} \circ x) \circ z$ **using** *assoc* **by** *simp*
  then show $y = z$ **using** *neutl* **and** *invl* **by** *simp*
next
  assume $y = z$
  then show $x \circ y = x \circ z$ **by** *simp*
qed

Here the "**in** *group*" target specification indicates that the result is recorded within that context for later use. This local theorem is also lifted to the global one *group.left-cancel*: $\bigwedge x\ y\ z :: \alpha::group.\ x \circ y = x \circ z \leftrightarrow y = z$. Since type *int* has been made an instance of *group* before, we may refer to that fact as well: $\bigwedge x\ y\ z :: int.\ x \circ y = x \circ z \leftrightarrow y = z$.

1.3.3 Derived definitions

Isabelle locales support a concept of local definitions in locales:

primrec (**in** *monoid*)
  pow-nat :: $nat \Rightarrow \alpha \Rightarrow \alpha$ **where**
  pow-nat $0\ x = \mathbf{1}$
  | *pow-nat* $(Suc\ n)\ x = x \circ$ *pow-nat* $n\ x$

If the locale *group* is also a class, this local definition is propagated onto a global definition of *pow-nat* :: $nat \Rightarrow \alpha::monoid \Rightarrow \alpha::monoid$ with corresponding theorems
  pow-nat $0\ x = \mathbf{1}$
  pow-nat $(Suc\ n)\ x = x \circ$ *pow-nat* $n\ x$.
As you can see from this example, for local definitions you may use any specification tool which works together with locales (e.g. [5]).

1.3.4 A functor analogy

We introduced Isar classes by analogy to type classes functional programming; if we reconsider this in the context of what has been said about type classes and locales, we can drive this analogy further by stating that type classes essentially correspond to functors which have a canonical interpretation as type classes. Anyway, there is also the possibility of other interpretations. For example, also *list*s form a monoid with *op* @ and [] as operations, but it seems inappropriate to apply to lists the same operations as for genuinly algebraic types. In such a case, we simply can do a particular interpretation of monoids for lists:

```
interpretation list-monoid: monoid [op @ []]
  by unfold-locales auto
```

This enables us to apply facts on monoids to lists, e.g. [] @ $x = x$.

When using this interpretation pattern, it may also be appropriate to map derived definitions accordingly:

```
fun
  replicate :: nat ⇒ α list ⇒ α list
where
  replicate 0 - = []
  | replicate (Suc n) xs = xs @ replicate n xs

interpretation list-monoid: monoid [op @ []] where
  monoid.pow-nat (op @) [] = replicate
proof
  fix n :: nat
  show monoid.pow-nat (op @) [] n = replicate n
    by (induct n) auto
qed
```

1.3.5 Additional subclass relations

Any *group* is also a *monoid*; this can be made explicit by claiming an additional subclass relation, together with a proof of the logical difference:

```
subclass (in group) monoid
proof unfold-locales
  fix x
  from invl have x⁻¹ ∘ x = 1 by simp
  with assoc [symmetric] neutl invl have x⁻¹ ∘ (x ∘ 1) = x⁻¹ ∘ x by simp
  with left-cancel show x ∘ 1 = x by simp
qed
```

The logical proof is carried out on the locale level and thus conveniently is opened using the *unfold-locales* method which only leaves the logical differences still open to proof to the user. Afterwards it is propagated to the type system, making *group* an instance of *monoid* by adding an additional edge to the graph of subclass relations (cf. figure 1.1).

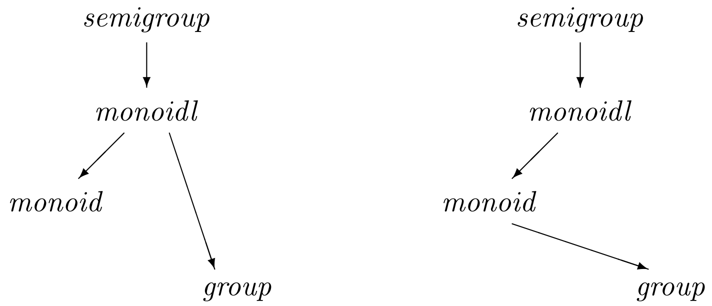

Figure 1.1: Subclass relationship of monoids and groups: before and after establishing the relationship *group* $\subseteq$ *monoid*; transitive edges left out.

For illustration, a derived definition in *group* which uses *pow-nat*:

definition (**in** *group*)
pow-int :: *int* $\Rightarrow \alpha \Rightarrow \alpha$ **where**
pow-int k x = (*if k* >= 0
then pow-nat (*nat k*) *x*
else (*pow-nat* (*nat* (− *k*)) *x*)$^{-1}$)

yields the global definition of *pow-int* :: *int* $\Rightarrow$ α::*group* $\Rightarrow$ α::*group* with the corresponding theorem *pow-int k x* = (*if* $0 \leq k$ *then pow-nat* (*nat k*) *x else* (*pow-nat* (*nat* (− *k*)) *x*)$^{-1}$).

1.3.6 A note on syntax

As a commodity, class context syntax allows to refer to local class operations and their global conuterparts uniformly; type inference resolves ambiguities. For example:

context *semigroup*
begin

term $x \circ y$ — example 1
term (x::*nat*) $\circ$ y — example 2

end

term $x \circ y$ — example 3

Here in example 1, the term refers to the local class operation *mult* [α], whereas in example 2 the type constraint enforces the global class operation *mult* [*nat*]. In the global context in example 3, the reference is to the polymorphic global class operation *mult* [*?α* :: *semigroup*].

1.4 Type classes and code generation

Turning back to the first motivation for type classes, namely overloading, it is obvious that overloading stemming from **class** statements and **instantiation** targets naturally maps to Haskell type classes. The code generator framework [2] takes this into account. Concerning target languages lacking type classes (e.g. SML), type classes are implemented by explicit dictionary construction. For example, lets go back to the power function:

definition
example :: *int* **where**
example = *pow-int* 10 (−2)

This maps to Haskell as:

export-code *example* **in** *Haskell* **module-name** *Classes* **file** *code-examples/*

```
module Classes where {


data Nat = Suc Nat | Zero_nat;

nat_aux :: Integer -> Nat -> Nat;
nat_aux i n = (if i <= 0 then n else nat_aux (i - 1) (Suc n));

nat :: Integer -> Nat;
nat i = nat_aux i Zero_nat;

class Semigroup a where {
  mult :: a -> a -> a;
};

class (Semigroup a) => Monoidl a where {
  neutral :: a;
};

class (Monoidl a) => Monoid a where {
};

class (Monoid a) => Group a where {
  inverse :: a -> a;
};

inverse_int :: Integer -> Integer;
inverse_int i = negate i;

neutral_int :: Integer;
neutral_int = 0;

mult_int :: Integer -> Integer -> Integer;
mult_int i j = i + j;
```

```
instance Semigroup Integer where {
  mult = mult_int;
};

instance Monoidl Integer where {
  neutral = neutral_int;
};

instance Monoid Integer where {
};

instance Group Integer where {
  inverse = inverse_int;
};

pow_nat :: forall a. (Monoid a) => Nat -> a -> a;
pow_nat (Suc n) x = mult x (pow_nat n x);
pow_nat Zero_nat x = neutral;

pow_int :: forall a. (Group a) => Integer -> a -> a;
pow_int k x =
  (if 0 <= k then pow_nat (nat k) x
    else inverse (pow_nat (nat (negate k)) x));

example :: Integer;
example = pow_int 10 (-2);

}
```

The whole code in SML with explicit dictionary passing:

export-code *example* **in** *SML* **module-name** *Classes* **file** *code-examples/classes.ML*

```
structure Classes =
struct

datatype nat = Suc of nat | Zero_nat;

fun nat_aux i n =
  (if IntInf.<= (i, (0 : IntInf.int)) then n
    else nat_aux (IntInf.- (i, (1 : IntInf.int))) (Suc n));

fun nat i = nat_aux i Zero_nat;

type 'a semigroup = {mult : 'a -> 'a -> 'a};
fun mult (A_:'a semigroup) = #mult A_;

type 'a monoidl =
  {Classes__semigroup_monoidl : 'a semigroup, neutral : 'a};
fun semigroup_monoidl (A_:'a monoidl) = #Classes__semigroup_monoidl A_;
fun neutral (A_:'a monoidl) = #neutral A_;

type 'a monoid = {Classes__monoidl_monoid : 'a monoidl};
fun monoidl_monoid (A_:'a monoid) = #Classes__monoidl_monoid A_;

type 'a group = {Classes__monoid_group : 'a monoid, inverse : 'a -> 'a};
fun monoid_group (A_:'a group) = #Classes__monoid_group A_;
fun inverse (A_:'a group) = #inverse A_;

fun inverse_int i = IntInf.~ i;
```

```
val neutral_int : IntInf.int = (0 : IntInf.int);

fun mult_int i j = IntInf.+ (i, j);

val semigroup_int = {mult = mult_int} : IntInf.int semigroup;

val monoidl_int =
  {Classes__semigroup_monoidl = semigroup_int, neutral = neutral_int} :
  IntInf.int monoidl;

val monoid_int = {Classes__monoidl_monoid = monoidl_int} :
  IntInf.int monoid;

val group_int =
  {Classes__monoid_group = monoid_int, inverse = inverse_int} :
  IntInf.int group;

fun pow_nat A_ (Suc n) x =
  mult ((semigroup_monoidl o monoidl_monoid) A_) x (pow_nat A_ n x)
  | pow_nat A_ Zero_nat x = neutral (monoidl_monoid A_);

fun pow_int A_ k x =
  (if IntInf.<= ((0 : IntInf.int), k)
    then pow_nat (monoid_group A_) (nat k) x
    else inverse A_ (pow_nat (monoid_group A_) (nat (IntInf.~ k)) x));

val example : IntInf.int =
  pow_int group_int (10 : IntInf.int) (~2 : IntInf.int);

end; (*struct Classes*)
```

Bibliography

[1] Stefan Wehr et. al. ML modules and Haskell type classes: A constructive comparison.

[2] Florian Haftmann. *Code generation from Isabelle theories*. http://isabelle.in.tum.de/doc/codegen.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] Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. Locales: A sectioning concept for Isabelle. 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.

[5] Alexander Krauss. Partial recursive functions in Higher-Order Logic. In U. Furbach and N. Shankar, editors, *Automated Reasoning: IJCAR 2006*, volume 4130 of *Lecture Notes in Computer Science*, pages 589–603. Springer-Verlag, 2006.

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

[7] T. Nipkow and C. Prehofer. Type checking type classes. In *ACM Symp. Principles of Programming Languages*, 1993.

[8] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002. LNCS Tutorial 2283.

[9] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *ACM Symp. Principles of Programming Languages*, 1989.

[10] 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.