Haskell-style type classes with Isabelle/Isar

Florian Haftmann

25 May 2015

Abstract

This tutorial introduces Isar type classes, which are a convenient mechanism for organizing specifications. Essentially, they combine an operational aspect (in the manner of Haskell) with a logical aspect, both managed uniformly.
1 Introduction

Type classes were introduced by Wadler and Blott [9] into the Haskell language to allow for a reasonable implementation of overloading1. As a canonical example, a polymorphic equality function \( eq :: \alpha \Rightarrow \alpha \Rightarrow \text{bool} \) which is overloaded on different types for \( \alpha \), which is achieved by splitting introduction of the \( eq \) function from its overloaded definitions by means of \textit{class} and \textit{instance} declarations: 2

\[
\begin{align*}
\text{class} & \quad \text{eq where} \\
& \quad \text{eq} :: \alpha \Rightarrow \alpha \Rightarrow \text{bool} \\
\text{instance} & \quad \text{n} \quad \text{eq where} \\
& \quad \text{eq} 0 0 = \text{True} \\
& \quad \text{eq} 0 - = \text{False} \\
& \quad \text{eq} - 0 = \text{False} \\
& \quad \text{eq} (\text{Suc} \ n) (\text{Suc} \ m) = \text{eq} \ n \ m \\
\text{instance} & \quad (\alpha::\text{eq}, \beta::\text{eq}) \text{ pair} :: \text{eq} \ where \\
& \quad \text{eq} (x1, y1) (x2, y2) = \text{eq} \ x1 \ x2 \land \text{eq} \ y1 \ y2 \\
\text{class} & \quad \text{ord extends eq where} \\
& \quad \text{less-eq} :: \alpha \Rightarrow \alpha \Rightarrow \text{bool} \\
& \quad \text{less} :: \alpha \Rightarrow \alpha \Rightarrow \text{bool}
\end{align*}
\]

Type variables are annotated with (finitely 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 [6, 7, 10].

From a software engineering point of view, type classes roughly 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 \textit{class eq} above could be given the following specification, demanding that \textit{class eq} is an equivalence relation obeying reflexivity, symmetry and transitivity:

\[
\begin{align*}
\text{class} & \quad \text{eq where} \\
& \quad \text{eq} :: \alpha \Rightarrow \alpha \Rightarrow \text{bool} \\
& \quad \text{satisfying}
\end{align*}
\]

\footnote{1throughout this tutorial, we are referring to classical Haskell 1.0 type classes, not considering later additions in expressiveness}

\footnote{2syntax here is a kind of isabellized Haskell}
2 A SIMPLE ALGEBRA EXAMPLE

2.1 Class definition

Depending on an arbitrary type $\alpha$, class \textit{semigroup} introduces a binary operator ($\otimes$) that is assumed to be associative:

\begin{verbatim}
class semigroup =
  fixes mult :: $\alpha \Rightarrow \alpha \Rightarrow \alpha$  (infixl $\otimes$ 70)
  assumes assoc: $(x \otimes y) \otimes z = x \otimes (y \otimes z)$
\end{verbatim}

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

From a theoretical point of view, type classes are lightweight 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. instantiating 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: locales [4].

Isar type classes also directly support code generation in a Haskell like fashion. Internally, they are mapped to more primitive Isabelle concepts [3].

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.
2.2 Class instantiation

The concrete type \texttt{int} is made a \textit{semigroup} instance by providing a suitable definition for the class parameter (\(\otimes\)) and a proof for the specification of \textit{assoc}. This is accomplished by the \texttt{instantiation} target:

\begin{verbatim}
instantiation int :: semigroup
begin

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

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

end
\end{verbatim}

\texttt{instantiation} defines class parameters at a particular instance using common specification tools (here, \texttt{definition}). The concluding \texttt{instance} opens a proof that the given parameters actually conform to the class specification. Note that the first proof step is the \textit{default} method, which for such instance proofs maps to the \texttt{intro-classes} method. This reduces an instance judgement to the relevant primitive proof goals; typically it is the first method applied in an instantiation proof.

From now on, the type-checker will consider \texttt{int} as a \textit{semigroup} automatically, i.e. any general results are immediately available on concrete instances.

Another instance of \textit{semigroup} yields the natural numbers:

\begin{verbatim}
instantiation nat :: semigroup
begin

primrec mult-nat where
  (0::nat) \otimes n = n
| Suc m \otimes n = Suc (m \otimes n)

instance proof
  fix \(m, n, q ::\) nat
  show \(m \otimes n \otimes q = m \otimes (n \otimes q)\)
    by (induct m) auto

end
\end{verbatim}
Note the occurrence of the name \textit{mult-nat} in the \texttt{primrec} declaration; by default, the local name of a class operation $f$ to be instantiated on type constructor $\kappa$ is mangled as $f\cdot\kappa$. In case of uncertainty, these names may be inspected using the \texttt{print-context} command.

\subsection{Lifting and parametric types}

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

\begin{verbatim}
instantiation prod :: (semigroup, semigroup) semigroup
begin

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

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

end
\end{verbatim}

Associativity of product semigroups is established using the definition of $\otimes$ on products and the hypothetical associativity of the type components; these hypotheses are legitimate due to the \texttt{semigroup} constraints imposed on the type components by the \texttt{instance} proposition. Indeed, this pattern often occurs with parametric types and type classes.

\subsection{Subclassing}

We define a subclass \texttt{monoidl} (a semigroup with a left-hand neutral) by extending \texttt{semigroup} with one additional parameter \texttt{neutral} together with its characteristic property:
class monoidl = semigroup +
  fixes neutral :: α (1)
  assumes neutl: 1 ⊗ x = x

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

instance proof
  fix n :: nat
  show 1 ⊗ n = n
    unfolding neutral-nat-def by simp

next
  fix k :: int
  show 1 ⊗ k = k
    unfolding neutral-int-def mult-int-def by simp

qed

end

instantiation prod :: (monoidl, monoidl) monoidl
begin

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

instance proof
  fix p :: α::monoidl × β::monoidl
  show 1 ⊗ 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:

```plaintext
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 prod :: (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:

```plaintext
class group = monoidl +
  fixes inverse :: α ⇒ α ((÷) [1000] 999)
  assumes invl: x÷ ⊗ x = 1

instantiation int :: group
begin

definition
```
inverse-int-def: \( i \div = - (i::\text{int}) \)

instance proof
   fix \( i :: \text{int} \)
   have \(-i + i = 0\) by simp
   then show \( i \div \otimes i = 1\)
      unfolding mult-int-def neutral-int-def inverse-int-def .
qed

end

3 Type classes as locales

3.1 A look behind the scenes

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 other than a locale:

```plaintext
class idem = 
  fixes \( f :: \alpha \Rightarrow \alpha \)
  assumes idem: \( f (f x) = f x \)
```

equivalent to the locale:

```plaintext
locale idem = 
  fixes \( f :: \alpha \Rightarrow \alpha \)
  assumes idem: \( f (f x) = f x \)
```

together with corresponding constant(s):

```plaintext
consts \( f :: \alpha \Rightarrow \alpha \)
```

The connection to the type system is done by means of a primitive type class `idem`, together with a corresponding interpretation:

```plaintext
interpretation idem-class:
  idem \( f :: (\alpha :: \text{idem}) \Rightarrow \alpha \)
```

This gives you the full power of the Isabelle module system; conclusions in locale `idem` are implicitly propagated to class `idem`. 
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 \texttt{left-cancel} lemma for groups, which states that the function \((x \otimes)\) is injective:

\begin{verbatim}
lemma (in group) left-cancel: \(x \otimes y = x \otimes z \iff y = z\)
proof
  assume \(x \otimes y = x \otimes z\)
  then have \(x \div (x \otimes y) = x \div (x \otimes z)\) by simp
  then have \((x \div x) \otimes y = (x \div x) \otimes z\) using assoc by simp
  then show \(y = z\) using neutl and invl by simp
next
  assume \(y = z\)
  then show \(x \otimes y = x \otimes z\) by simp
qed
\end{verbatim}

Here the \texttt{“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 \texttt{group.left-cancel: \(\forall x y z :: \alpha::group. x \otimes y = x \otimes z \iff y = z\).} Since type \texttt{int} has been made an instance of \texttt{group} before, we may refer to that fact as well: \(\forall x y z :: \texttt{int}. x \otimes y = x \otimes z \iff y = z\).

3.3 Derived definitions

Isabelle locales are targets which support local definitions:

\begin{verbatim}
primrec (in monoid) pow-nat :: nat \Rightarrow \alpha \Rightarrow \alpha where
  pow-nat 0 x = 1
| pow-nat (Suc n) x = x \otimes pow-nat n x
\end{verbatim}

If the locale \texttt{group} is also a class, this local definition is propagated onto a global definition of \texttt{pow-nat :: nat \Rightarrow \alpha::monoid \Rightarrow \alpha::monoid} with corresponding theorems

\begin{verbatim}
pow-nat 0 x = 1
pow-nat (Suc n) x = x \otimes pow-nat n x.
\end{verbatim}

As you can see from this example, for local definitions you may use any specification tool which works together with locales, such as Krauss’s recursive function package [5].
3.4 A functor analogy

We introduced Isar classes by analogy to type classes in 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 that have a canonical interpretation as type classes. There is also the possibility of other interpretations. For example, lists also form a monoid with append and \([\ ]\) as operations, but it seems inappropriate to apply to lists the same operations as for genuinely algebraic types. In such a case, we can simply make a particular interpretation of monoids for lists:

\[
\text{interpretation list-monoid: monoid append } []\
\text{proof qed auto}
\]

This enables us to apply facts on monoids to lists, e.g. \([\ ] \otimes x = x\).

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

\[
\begin{align*}
\text{primrec} \ \text{replicate} :: \ & \text{nat} \Rightarrow \alpha \ \text{list} \Rightarrow \alpha \ \text{list} \ \text{where} \\
& \text{replicate} \ 0 \ - = [] \\
& | \ \text{replicate} \ (\text{Suc} \ n) \ xs = xs \ @ \ \text{replicate} \ n \ xs
\end{align*}
\]

\[
\text{interpretation list-monoid: monoid append } [] \ \text{where} \\
\text{monoid.pow-nat append } [] = \text{replicate}
\]

\[
\text{proof} -
\text{interpret} \text{ monoid append } [] . .
\text{show} \text{ monoid.pow-nat append } [] = \text{replicate}
\text{proof}
\text{fix} \ n
\text{show} \text{ monoid.pow-nat append } [] \ n = \text{replicate} \ n
\text{by (induct} n \text{) auto}
\text{qed}
\text{qed intro-locales}
\]

This pattern is also helpful to reuse abstract specifications on the same type. For example, think of a class preorder; for type nat, there are at least two possible instances: the natural order or the order induced by the divides relation. But only one of these instances can be used for interpretation; using the locale behind the class preorder, it is still possible to utilise the same abstract specification again using interpretation.
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
  fix x
  from invl have x ÷ ⊗ x = 1 by simp
  with assoc [symmetric] neutl invl have x ÷ ⊗ (x ⊗ 1) = x ÷ ⊗ x by simp
  with left-cancel show x ⊗ 1 = x by simp
qed
```

The logical proof is carried out on the locale level. 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 (figure 1).

For illustration, a derived definition in group using pow-nat

```
definition (in group) pow-int :: int ⇒ α ⇒ α where
  pow-int k x = (if k >= 0 then pow-nat (nat k) x
                   else (pow-nat (nat (- k)) x ÷ )

yields the global definition of pow-int :: int ⇒ α::group ⇒ α::group with the corresponding theorem pow-int k x = (if 0 ≤ k then pow-nat (nat k) x else (pow-nat (nat (- k)) x ÷ ).
```
3.6 A note on syntax

As a convenience, class context syntax allows references to local class operations and their global counterparts uniformly; type inference resolves ambiguities. For example:

```haskell
context semigroup
begin
  term x ⊗ y — example 1
  term (x::nat) ⊗ y — example 2
end

term x ⊗ y — example 3
```

Here in example 1, the term refers to the local class operation \( \text{mult} [\alpha] \), whereas in example 2 the type constraint enforces the global class operation \( \text{mult} [\text{nat}] \). In the global context in example 3, the reference is to the polymorphic global class operation \( \text{mult} [?\alpha :: \text{semigroup}] \).

4 Further issues

4.1 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. If the target language (e.g. SML) lacks type classes, then they are implemented by an explicit dictionary construction. As example, let’s go back to the power function:

```haskell
definition example :: int where
  example = pow-int 10 (-2)
```

This maps to Haskell as follows:

```haskell
module Example(Num, Int, example) where {
import Prelude ((==), (/=), (<), (<=), (>), (>=), (+), (-), (*), (/), (**), (>>=), (>>), (<<=), (&&), (||), (^), (^^), (.), ($), ($!), (++), (!!), Eq, error, id, return, not, fst, snd, map, filter, concat, concatMap, reverse, zip, null, takeWhile, dropWhile, all, any, Integer, negate, abs, divMod, String, Bool(True, False), Maybe(Nothing, Just));
```
import qualified Prelude;

data Num = One | Bit0 Num | Bit1 Num;

data Int = Zero_int | Pos Num | Neg Num;

neutral_int :: Int;
neutral_int = Zero_int;

plus_num :: Num -> Num -> Num;
plus_num (Bit1 m) (Bit1 n) = Bit0 (plus_num (plus_num m n) One);
plus_num (Bit1 m) (Bit0 n) = Bit1 (plus_num m n);
plus_num (Bit1 m) One = Bit0 (plus_num m One);
plus_num (Bit0 m) (Bit1 n) = Bit1 (plus_num m n);
plus_num (Bit0 m) (Bit0 n) = Bit0 (plus_num m n);
plus_num (Bit0 m) One = Bit1 m;
plus_num One (Bit1 n) = Bit0 (plus_num n One);
plus_num One (Bit0 n) = Bit1 n;
plus_num One One = Bit0 One;

uminus_int :: Int -> Int;
uminus_int (Neg m) = Pos m;
uminus_int (Pos m) = Neg m;
uminus_int Zero_int = Zero_int;

bitM :: Num -> Num;
bitM One = One;
bitM (Bit0 n) = Bit1 (bitM n);
bitM (Bit1 n) = Bit1 (Bit0 n);

dup :: Int -> Int;
dup (Neg n) = Neg (Bit0 n);
dup (Pos n) = Pos (Bit0 n);
dup Zero_int = Zero_int;

plus_int :: Int -> Int -> Int;
plus_int (Neg m) (Neg n) = Neg (plus_num m n);
plus_int (Neg m) (Pos n) = sub n m;
plus_int (Pos m) (Neg n) = Pos (plus_num m n);
plus_int (Pos m) (Pos n) = Pos (plus_num m n);
plus_int Zero_int l = l;
plus_int k Zero_int = k;

sub :: Num -> Num -> Int;
sub (Bit0 m) (Bit1 n) = minus_int (dup (sub m n)) (Pos One);
sub (Bit1 m) (Bit0 n) = plus_int (dup (sub m n)) (Pos One);
sub (Bit1 m) (Bit1 n) = dup (sub m n);
sub (Bit0 m) (Bit0 n) = dup (sub m n);
sub One (Bit1 n) = Neg (Bit0 n);
sub One (Bit0 n) = Neg (bitM n);
sub (Bit1 m) One = Pos (Bit0 m);
sub (Bit0 m) One = Pos (bitM m);
sub One One = Zero_int;

minus_int :: Int -> Int -> Int;
minus_int (Neg m) (Neg n) = sub n m;
minus_int (Neg m) (Pos n) = Neg (plus_num m n);
minus_int (Pos m) (Neg n) = Pos (plus_num m n);
minus_int (Pos m) (Pos n) = sub m n;
minus_int Zero_int l = uminus_int l;
minus_int k Zero_int = k;

mult_int :: Int -> Int -> Int;
mult_int i j = plus_int i j;
inverse_int :: Int -> Int;
inverse_int i = uminus_int i;

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;
};

instance Semigroup Int where {
  mult = mult_int;
};

instance Monoidl Int where {
  neutral = neutral_int;
};

instance Monoid Int where {
};

instance Group Int where {
  inverse = inverse_int;
};

data Nat = Zero_nat | Suc Nat;

plus_nat :: Nat -> Nat -> Nat;
plus_nat (Suc m) n = plus_nat m (Suc n);
plus_nat Zero_nat n = n;

one_nat :: Nat;
one_nat = Suc Zero_nat;

nat_of_num :: Num -> Nat;
nat_of_num (Bit1 n) = let {
  m = nat_of_num n;
  } in Suc (plus_nat m m);
nat_of_num (Bit0 n) = let {
  m = nat_of_num n;
  } in plus_nat m m;
nat_of_num One = one_nat;

nat :: Int -> Nat;
nat (Pos k) = nat_of_num k;
nat Zero_int = Zero_nat;
nat (Neg k) = Zero_nat;

less_eq_num :: Num -> Num -> Bool;
less_eq_num (Bit1 m) (Bit0 n) = less_num m n;
less_eq_num (Bit1 m) (Biti n) = less_eq_num m n;
less_eq_num (Bit0 m) (Bit1 n) = less_eq_num m n;
less_eq_num (Bit0 m) (Bit0 n) = less_eq_num m n;
less_eq_num (Bit1 m) One = False;
less_eq_num (Bit0 m) One = False;
less_eq_num One n = True;
The code in SML has explicit dictionary passing:

```sml
structure Example : sig
  type num
  type int
  val example : int
end = struct
  datatype num = One | Bit0 of num | Bit1 of num;
  datatype int = Zero_int | Pos of num | Neg of num;
  val neutral_int : int = Zero_int;
  fun plus_num (Bit1 m) (Bit0 n) = Bit0 (plus_num (plus_num m n) One)
  | plus_num (Bit1 m) (Bit0 n) = Bit1 (plus_num m n)
  | plus_num (Bit1 m) (One) = Bit0 (plus_num m One)
  | plus_num (Bit0 m) (Bit1 n) = Bit1 (plus_num m n)
  | plus_num (Bit0 m) (Bit0 n) = Bit0 (plus_num m n)
  | plus_num (Bit0 m) (One) = Bit1 m
  | plus_num (One) (Bit1 n) = Bit0 (plus_num n One)
  | plus_num (One) (Bit0 n) = Bit1 n
  | plus_num (One) (One) = Bit0 One;
  fun uminus_int (Neg m) = Pos m
  | uminus_int (Pos m) = Neg m
  | uminus_int Zero_int = Zero_int;
end
```

4 FURTHER ISSUES

```
less_num :: Num -> Num -> Bool;
less_num (Bit1 m) (Bit0 n) = less_num m n;
less_num (Bit1 m) (Bit1 n) = less_num m n;
less_num (Bit0 m) (Bit1 n) = less_eq_num m n;
less_num (Bit0 m) (Bit0 n) = less_num m n;
less_num One (Bit1 n) = True;
less_num One (Bit0 n) = True;
less_num m One = False;

less_eq_int :: Int -> Int -> Bool;
less_eq_int (Neg k) (Neg l) = less_eq_num l k;
less_eq_int (Neg k) (Pos l) = True;
less_eq_int (Neg k) Zero_int = True;
less_eq_int (Pos k) (Neg l) = False;
less_eq_int (Pos k) (Pos l) = less_eq_num k l;
less_eq_int (Pos k) Zero_int = False;
less_eq_int Zero_int (Neg l) = False;
less_eq_int Zero_int (Pos l) = True;
less_eq_int Zero_int Zero_int = True;

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

pow_int :: forall a. (Group a) => Int -> a -> a;
pow_int k x =
  (if less_eq_int Zero_int k then pow_nat (nat k) x
   else inverse (pow_nat (nat (uminus_int k)) x));

example :: Int;
example = pow_int (Pos (Bit0 (Bit1 (Bit0 One)))) (Neg (Bit0 One));
```
fun bitM One = One
  | bitM (Bit0 n) = Bit1 (bitM n)
  | bitM (Bit1 n) = Bit1 (Bit0 n);

fun dup (Neg n) = Neg (Bit0 n)
  | dup (Pos n) = Pos (Bit0 n)
  | dup Zero_int = Zero_int;

fun plus_int (Neg m) (Neg n) = Neg (plus_num m n)
  | plus_int (Neg m) (Pos n) = sub n m
  | plus_int (Pos m) (Neg n) = sub m n
  | plus_int (Pos m) (Pos n) = Pos (plus_num m n)
  | plus_int Zero_int l = l
  | plus_int k Zero_int = k
and sub (Bit0 m) (Bit1 n) = minus_int (dup (sub m n)) (Pos One)
  | sub (Bit1 m) (Bit0 n) = plus_int (dup (sub m n)) (Pos One)
  | sub (Bit1 m) (Bit1 n) = dup (sub m n)
  | sub (Bit0 m) (Bit0 n) = dup (sub m n)
  | sub One (Bit1 n) = Neg (Bit0 n)
  | sub One (Bit0 n) = Neg (bitM n)
  | sub (Bit1 m) One = Pos (Bit0 m)
  | sub (Bit0 m) One = Pos (bitM m)
  | sub One One = Zero_int
and minus_int (Neg m) (Neg n) = sub n m
  | minus_int (Neg m) (Pos n) = Neg (plus_num m n)
  | minus_int (Pos m) (Neg n) = Pos (plus_num m n)
  | minus_int (Pos m) (Pos n) = sub m n
  | minus_int Zero_int l = uminus_int l
  | minus_int k Zero_int = k;

fun mult_int i j = plus_int i j;
fun inverse_int i = uminus_int i;

val one_nat : nat = Suc Zero_nat;
fun nat_of_num (Bit1 n) = 
    let
    val m = nat_of_num n;
    in
    Suc (plus_nat m m)
    end
| nat_of_num (Bit0 n) = let
    val m = nat_of_num n;
    in
    plus_nat m m
    end
| nat_of_num One = one_nat;

fun nat (Pos k) = nat_of_num k
| nat Zero_int = Zero_nat
| nat (Neg k) = Zero_nat;

fun less_eq_num (Bit1 m) (Bit0 n) = less_num m n
| less_eq_num (Bit1 m) (Bit1 n) = less_eq_num m n
| less_eq_num (Bit0 m) (Bit1 n) = less_eq_num m n
| less_eq_num (Bit0 m) (Bit0 n) = less_eq_num m n
| less_eq_num (Bit1 m) One = false
| less_eq_num (Bit0 m) One = false
| less_eq_num One n = true
and less_num (Bit1 m) (Bit0 n) = less_num m n
| less_num (Bit1 m) (Bit1 n) = less_num m n
| less_num (Bit0 m) (Bit1 n) = less_eq_num m n
| less_num (Bit0 m) (Bit0 n) = less_num m n
| less_num One (Bit1 n) = true
| less_num One (Bit0 n) = true
| less_num m One = false;

fun less_eq_int (Neg k) (Neg l) = less_eq_num l k
| less_eq_int (Neg k) (Pos l) = true
| less_eq_int (Neg k) Zero_int = true
| less_eq_int (Pos k) (Neg l) = false
| less_eq_int (Pos k) (Pos l) = less_eq_num k l
| less_eq_int (Pos k) Zero_int = false
| less_eq_int Zero_int (Neg l) = false
| less_eq_int Zero_int Zero_int = true
| less_eq_int Zero_int Zero_int = true;

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

fun pow_int A_ k x = 
    if less_eq_int Zero_int k then pow_nat (monoid_group A_) (nat k) x
    else inverse A_ (pow_nat (monoid_group A_) (nat (uminus_int k)) x));

val example : int = 
    pow_int group_int (Pos (Bit0 (Bit1 (Bit0 One)))) (Neg (Bit0 One));
final case class Bit0(a: num) extends num
final case class Bit1(a: num) extends num

abstract sealed class int
final case class zero_int() extends int
final case class Pos(a: num) extends int
final case class Neg(a: num) extends int

def neutral_int: int = zero_int()
def plus_num(x0: num, x1: num): num = (x0, x1) match {
case (Bit1(m), Bit1(n)) => Bit0(plus_num(plus_num(m, n), One()))
case (Bit1(m), Bit0(n)) => Bit1(plus_num(m, n))
case (Bit1(m), One()) => Bit0(plus_num(m, One()))
case (Bit0(m), Bit1(n)) => Bit1(plus_num(m, n))
case (Bit0(m), Bit0(n)) => Bit0(plus_num(m, n))
case (Bit0(m), One()) => Bit1(m)
case (One(), Bit1(n)) => Bit0(plus_num(n, One()))
case (One(), Bit0(n)) => Bit1(n)
case (One(), One()) => Bit0(One())
}
def uminus_int(x0: int): int = x0 match {
case Neg(m) => Pos(m)
case Pos(m) => Neg(m)
case zero_int() => zero_int()
}
def BitM(x0: num): num = x0 match {
case One() => One()
case Bit0(n) => Bit1(BitM(n))
case Bit1(n) => Bit1(Bit0(n))
}
def dup(x0: int): int = x0 match {
case Neg(n) => Neg(Bit0(n))
case Pos(n) => Pos(Bit0(n))
case zero_int() => zero_int()
}
def minus_int(k: int, l: int): int = (k, l) match {
case (Neg(m), Neg(n)) => sub(n, m)
case (Neg(m), Pos(n)) => Neg(plus_num(m, n))
case (Pos(m), Neg(n)) => Pos(plus_num(m, n))
case (Pos(m), Pos(n)) => sub(m, n)
case (zero_int(), l) => uminus_int(l)
case (k, zero_int()) => k
}
def sub(x0: num, x1: num): int = (x0, x1) match {
case (Bit0(m), Bit1(n)) => minus_int(dup(sub(m, n)), Pos(One()))
case (Bit1(m), Bit0(n)) => plus_int(dup(sub(m, n)), Pos(One()))
case (Bit1(m), Bit1(n)) => dup(sub(m, n))
case (Bit0(m), Bit0(n)) => dup(sub(m, n))
case (One(), Bit1(n)) => Neg(Bit0(n))
case (One(), Bit0(n)) => Neg(BitM(n))
case (Bit1(m), One()) => Pos(Bit0(m))
case (Bit0(m), One()) => Pos(BitM(m))
case (One(), One()) => zero_int()
}
def plus_int(k: int, l: int): int = (k, l) match {
case (Neg(m), Neg(n)) => Neg(plus_num(m, n))
case (Neg(m), Pos(n)) => Neg(sub(m, n))
case (Pos(m), Neg(n)) => sub(m, n)
case (Pos(m), Pos(n)) => sub(m, n)
}
case (Pos(m), Pos(n)) => Pos(plus_num(m, n))
case (zero_int(), l) => l
case (k, zero_int()) => k
}
def mult_int(i: int, j: int): int = plus_int(i, j)
def inverse_int(i: int): int = uminus_int(i)

trait semigroup[A] {
  val 'Example.mult': (A, A) => A
}
def mult[A](a: A, b: A)(implicit A: semigroup[A]): A = A.'Example.mult'(a, b)

trait monoidl[A] extends semigroup[A] {
  val 'Example.neutral': A
}
def neutral[A](implicit A: monoidl[A]): A = A.'Example.neutral'

trait monoid[A] extends monoidl[A] {
  val 'Example.neutral': A
  val 'Example.mult': (a: int, b: int) => mult_int(a, b)
}

trait group[A] extends monoid[A] {
  val 'Example.inverse': A => A
  def inverse[a: A](implicit A: group[A]): A = A.'Example.inverse'(a)
}

abstract sealed class nat
final case class zero_nat() extends nat
final case class Suc(a: nat) extends nat

def plus_nat(x0: nat, n: nat): nat = (x0, n) match {
  case (Suc(m), n) => plus_nat(m, Suc(n))
  case (zero_nat(), n) => n
}
def one_nat: nat = Suc(zero_nat())

def nat_of_num(x0: num): nat = x0 match {
  case Bit1(n) => {
    val m: nat = nat_of_num(n);
    Suc(plus_nat(m, m))
  }
  case Bit0(n) => {
...
val m: nat = nat_of_num(n);
  plus_nat(m, m)
)
  case One() => one_nat}

def nat(x0: int): nat = x0 match {
  case Pos(k) => nat_of_num(k)
  case Neg() => zero_nat()
  case One() => zero_nat()
}

def less_num(m: num, x1: num): Boolean = (m, x1) match {
  case (Bit1(m), Bit0(n)) => less_num(m, n)
  case (Bit1(m), Bit1(n)) => less_num(m, n)
  case (Bit0(m), Bit1(n)) => less_eq_num(m, n)
  case (Bit0(m), Bit0(n)) => less_num(m, n)
  case (One(), Bit1(n)) => true
  case (One(), Bit0(n)) => true
  case (m, One()) => false
}

def less_eq_num(x0: num, n: num): Boolean = (x0, n) match {
  case (Bit1(m), Bit0(n)) => less_num(m, n)
  case (Bit1(m), Bit1(n)) => less_eq_num(m, n)
  case (Bit0(m), Bit1(n)) => less_eq_num(m, n)
  case (Bit0(m), Bit0(n)) => less_eq_num(m, n)
  case (Bit1(m), One()) => false
  case (Bit0(m), One()) => false
  case (One(), n) => true
}

def less_eq_int(x0: int, x1: int): Boolean = (x0, x1) match {
  case (Neg(k), Neg(l)) => less_eq_num(l, k)
  case (Neg(k), Pos(l)) => true
  case (Neg(k), zero_int()) => true
  case (Pos(k), Neg(l)) => false
  case (Pos(k), Pos(l)) => less_eq_num(k, l)
  case (Pos(k), zero_int()) => false
  case (zero_int(), Neg(l)) => false
  case (zero_int(), Pos(l)) => true
  case (zero_int(), zero_int()) => true
}

def pow_nat[A : monoid](xa0: nat, x: A): A = (xa0, x) match {
  case (zero_nat(), x) => neutral[A]
  case (Suc(n), x) => mult[A](x, pow_nat[A](n, x))
}

def pow_int[A : group](k: int, x: A): A =
  if (less_eq_int(zero_int(), k)) pow_nat[A](nat(k), x)
  else inverse[A](pow_nat[A](nat(uminus_int(k)), x))

def example: int =
  pow_int[int](Pos(Bit0(Bit1(Bit0(One())))), Neg(Bit0(One())))
} /* object Example */
4.2 Inspecting the type class universe

To facilitate orientation in complex subclass structures, two diagnostics commands are provided:

**print-classes** print a list of all classes together with associated operations etc.

**class-deps** visualizes the subclass relation between all classes as a Hasse diagram. An optional first sort argument constrains the set of classes to all subclasses of this sort, an optional second sort argument to all superclasses of this sort.

References

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


