Haskell-style type classes with Isabelle/Isar

Florian Haftmann

30 January 2011

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 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: [2]

```
class eq where
  eq :: α ⇒ α ⇒ bool

instance nat :: eq where
  eq 0 0 = True
  eq 0 - = False
  eq - 0 = False
  eq (Suc n) (Suc m) = eq n m

instance (α::eq, β::eq) pair :: eq where
  eq (x1, y1) (x2, y2) = eq x1 x2 ∧ eq y1 y2

class ord extends eq where
  less-eq :: α ⇒ α ⇒ bool
  less :: α ⇒ α ⇒ bool
```

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 *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 :: α ⇒ α ⇒ bool
satisfying
```

[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

refl: $\mathit{eq}\ x\ x$
sym: $\mathit{eq}\ x\ y \longleftrightarrow \mathit{eq}\ x\ y$
trans: $\mathit{eq}\ x\ y \wedge \mathit{eq}\ y\ z \longrightarrow \mathit{eq}\ x\ z$

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 A simple algebra example

2.1 Class definition

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

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

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*: $\bigwedge x\ y\ z :: \alpha{::}\mathit{semigroup}.\ (x \otimes y) \otimes z = x \otimes (y \otimes z)$.

2.2 Class instantiation

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

```
instantiation int :: semigroup
begin

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

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

end
```

instantiation defines 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 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 *int* as a *semigroup* automatically, i.e. any general results are immediately available on concrete instances.

Another instance of *semigroup* yields the natural numbers:

```
instantiation nat :: semigroup
begin

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

instance proof
  fix m n q :: nat
  show m ⊗ n ⊗ q = m ⊗ (n ⊗ 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 be instantiated on type constructor κ is mangled as f-κ. In case of uncertainty, these names may be inspected using the **print-context** command or the corresponding Proof-General button.

2.3 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:

```
instantiation prod :: (semigroup, semigroup) semigroup
begin

definition
  mult-prod-def: p₁ ⊗ p₂ = (fst p₁ ⊗ fst p₂, snd p₁ ⊗ snd p₂)

instance proof
  fix p₁ p₂ p₃ :: α::semigroup × β::semigroup
  show p₁ ⊗ p₂ ⊗ p₃ = p₁ ⊗ (p₂ ⊗ p₃)
    unfolding mult-prod-def by (simp add: assoc)
qed

end
```

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 *semigroup* constraints imposed on the type components by the **instance** proposition. Indeed, this pattern often occurs with parametric types and type classes.

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 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:

```
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 ⊗ 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:

```
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:

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

instantiation int :: group
begin

definition
```

inverse-int-def: $i\div = -\ (i{::}int)$

instance proof
  fix i :: *int*
  have $-i + i = 0$ **by** *simp*
  then show $i\div \otimes i = \mathbf{1}$
    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:

class *idem* =
  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 type class

classes *idem* < *type*

together with a corresponding interpretation:

interpretation *idem-class*:
  idem f :: $(\alpha{::}idem) \Rightarrow \alpha$

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 *left-cancel* lemma for groups, which states that the function $(x \otimes)$ is injective:

```
lemma (in group) left-cancel: x ⊗ y = x ⊗ z ⟷ y = z
proof
  assume x ⊗ y = x ⊗ z
  then have x÷ ⊗ (x ⊗ y) = x÷ ⊗ (x ⊗ z) by simp
  then have (x÷ ⊗ x) ⊗ y = (x÷ ⊗ x) ⊗ z using assoc by simp
  then show y = z using neutl and invl by simp
next
  assume y = z
  then show x ⊗ y = x ⊗ 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*: $\bigwedge x\ y\ z :: \alpha{::}group.\ x \otimes y = x \otimes z \longleftrightarrow y = z$. Since type *int* has been made an instance of *group* before, we may refer to that fact as well: $\bigwedge x\ y\ z :: int.\ x \otimes y = x \otimes z \longleftrightarrow y = z$.

3.3 Derived definitions

Isabelle locales are targets which support local definitions:

```
primrec (in monoid) pow-nat :: nat ⇒ α ⇒ α where
  pow-nat 0 x = 1
  | pow-nat (Suc n) x = x ⊗ pow-nat n x
```

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

$pow\text{-}nat\ 0\ x = \mathbf{1}$

$pow\text{-}nat\ (Suc\ n)\ x = x \otimes pow\text{-}nat\ n\ x.$

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, *list*s 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:

```
interpretation list-monoid: monoid append []
  proof qed 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:

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

interpretation list-monoid: monoid append [] where
  monoid.pow-nat append [] = replicate
proof -
  interpret monoid append [] ..
  show monoid.pow-nat append [] = replicate
  proof
    fix n
    show monoid.pow-nat append [] n = replicate n
      by (induct n) auto
  qed
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 **instantiation**; 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).

$$\begin{tikzcd}
& \mathit{semigroup} \arrow[d] & \\
& \mathit{monoidl} \arrow[dl] \arrow[ddr] & \\
\mathit{monoid} & & \\
& & \mathit{group}
\end{tikzcd}
\qquad
\begin{tikzcd}
& \mathit{semigroup} \arrow[d] & \\
& \mathit{monoidl} \arrow[dl] & \\
\mathit{monoid} \arrow[drr] & & \\
& & \mathit{group}
\end{tikzcd}$$

Figure 1: Subclass relationship of monoids and groups: before and after establishing the relationship *group* ⊆ *monoid*; transitive edges are left out.

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:

context *semigroup*
begin

term $x \otimes y$ — example 1
term $(x{::}nat) \otimes y$ — example 2

end

term $x \otimes y$ — example 3

Here in example 1, the term refers to the local class operation *mult* $[\alpha]$, 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* $[?\alpha :: semigroup]$.

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:

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

This maps to Haskell as follows:

```
module Example where {

data Nat = Zero_nat | Suc 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;
};

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

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

instance Semigroup Integer where {
  mult = mult_int;
};

neutral_int :: Integer;
neutral_int = 0;

instance Monoidl Integer where {
  neutral = neutral_int;
};

instance Monoid Integer where {
};

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

instance Group Integer where {
  inverse = inverse_int;
};

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

}
```

The code in SML has explicit dictionary passing:

```
structure Example : sig
  datatype nat = Zero_nat | Suc of nat
  val nat_aux : IntInf.int -> nat -> nat
  val nat : IntInf.int -> nat
  type 'a semigroup
  val mult : 'a semigroup -> 'a -> 'a -> 'a
  type 'a monoidl
  val semigroup_monoidl : 'a monoidl -> 'a semigroup
  val neutral : 'a monoidl -> 'a
  type 'a monoid
  val monoidl_monoid : 'a monoid -> 'a monoidl
  type 'a group
  val monoid_group : 'a group -> 'a monoid
  val inverse : 'a group -> 'a -> 'a
  val pow_nat : 'a monoid -> nat -> 'a -> 'a
  val pow_int : 'a group -> IntInf.int -> 'a -> 'a
  val mult_int : IntInf.int -> IntInf.int -> IntInf.int
  val semigroup_int : IntInf.int semigroup
  val neutral_int : IntInf.int
  val monoidl_int : IntInf.int monoidl
  val monoid_int : IntInf.int monoid
  val inverse_int : IntInf.int -> IntInf.int
  val group_int : IntInf.int group
  val example : IntInf.int
end = struct

datatype nat = Zero_nat | Suc of 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};
val mult = #mult : 'a semigroup -> 'a -> 'a -> 'a;

type 'a monoidl = {semigroup_monoidl : 'a semigroup, neutral : 'a};
val semigroup_monoidl = #semigroup_monoidl : 'a monoidl -> 'a semigroup;
val neutral = #neutral : 'a monoidl -> 'a;

type 'a monoid = {monoidl_monoid : 'a monoidl};
val monoidl_monoid = #monoidl_monoid : 'a monoid -> 'a monoidl;

type 'a group = {monoid_group : 'a monoid, inverse : 'a -> 'a};
val monoid_group = #monoid_group : 'a group -> 'a monoid;
val inverse = #inverse : 'a group -> 'a -> 'a;

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 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));

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

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

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

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

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

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

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

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

end; (*struct Example*)
```

In Scala, implicts are used as dictionaries:

```
object Example {

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

def nat_aux(i: BigInt, n: nat): nat =
  (if (i <= BigInt(0)) n else nat_aux(i - BigInt(1), Suc(n)))

def nat(i: BigInt): nat = nat_aux(i, Zero_nat)

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] {
}

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

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: BigInt, x: A): A =
  (if (BigInt(0) <= k) pow_nat[A](nat(k), x)
    else inverse[A](pow_nat[A](nat((- k)), x)))

def mult_int(i: BigInt, j: BigInt): BigInt = i + j

implicit def semigroup_int: semigroup[BigInt] = new semigroup[BigInt] {
  val `Example.mult` = (a: BigInt, b: BigInt) => mult_int(a, b)
}

def neutral_int: BigInt = BigInt(0)

implicit def monoidl_int: monoidl[BigInt] = new monoidl[BigInt] {
  val `Example.neutral` = neutral_int
  val `Example.mult` = (a: BigInt, b: BigInt) => mult_int(a, b)
}

implicit def monoid_int: monoid[BigInt] = new monoid[BigInt] {
  val ‘Example.neutral‘ = neutral_int
  val ‘Example.mult‘ = (a: BigInt, b: BigInt) => mult_int(a, b)
}

def inverse_int(i: BigInt): BigInt = (- i)

implicit def group_int: group[BigInt] = new group[BigInt] {
  val ‘Example.inverse‘ = (a: BigInt) => inverse_int(a)
  val ‘Example.neutral‘ = neutral_int
  val ‘Example.mult‘ = (a: BigInt, b: BigInt) => mult_int(a, b)
}

def example: BigInt = pow_int[BigInt](BigInt(10), BigInt(- 2))

} /* 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.

References

[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*, volume 2283 of *Lecture Notes in Computer Science.* Springer-Verlag, 2002.

[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.