Using Axiomatic Type Classes in Isabelle
a tutorial

Markus Wenzel

28th August 1998

ISABELLE features a Haskell-like type system with ordered type classes already since 1991 (see [Nipkow, 1993b]). Initially, classes mainly served as a *syntactic tool* to formulate polymorphic object logics in a clean way (like many-sorted FOL, see [Paulson, 1994, §1.1.2–1.1.3]).

Applying classes at the *logical level* to provide a simple notion of abstract theories and instantiations to concrete ones, has also been long proposed (see [Nipkow, 1993a] and [Nipkow, 1993b, §4]). 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.

How simple things really are has been shown in [Wenzel, 1994, esp. §4] which also provided an implementation of the axclass package that was eventually released with ISABELLE94. Unfortunately there was a snag: That version of ISABELLE still contained an old conceptual bug in the core machinery which caused theories to become inconsistent after introducing empty sorts. Note that empty intersections of axiomatic type classes easily occur, unless all basic classes are very trivial.

These problems prevented the axclass package to be used seriously — they have been fixed in ISABELLE94-2.

1 Some simple examples

Axiomatic type classes are a concept of ISABELLE's meta-logic. They may be applied to any object-logic that directly uses the meta type system. Subsequently, we present various examples that are formulated within ISABELLE/HOL[1], except the one of §1.4 which is in ISABELLE/FOL[2].

1.1 Semigroups

An axiomatic type class is simply a class of types that all meet certain *axioms*. Thus, type classes may be also understood as type predicates — i.e. abstractions over a single type argument α. Class axioms typically contain polymorphic

[1]See also directory `HOL/AxClasses/Tutorial`.

[2]See also files `FOL/ex/NatClass.thy` and `FOL/ex/NatClass.ML`.

constants that depend on this type α. These *characteristic constants* behave like operations associated with the "carrier" α.

We illustrate these basic concepts with the following theory `Semigroup`:

```
Semigroup = HOL +

consts
  "<*>"         :: "['a, 'a] => 'a"             (infixl 70)

axclass
  semigroup < term
  assoc         "(x <*> y) <*> z = x <*> (y <*> z)"

end
```

`Semigroup` first declares a polymorphic constant $\langle * \rangle :: [\alpha, \alpha] \Rightarrow \alpha$ and then defines the class `semigroup` of all those types τ such that $\langle * \rangle :: [\tau, \tau] \Rightarrow \tau$ is an associative operator[3]. The axiom `assoc` 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 α and a fixed *signature*. Different signatures require different classes. In the following theory `Semigroups`, class `plus_sg` represents semigroups of the form $(\tau, \langle + \rangle^\tau)$ while `times_sg` represents semigroups $(\tau, \langle * \rangle^\tau)$:

```
Semigroups = HOL +

consts
  "<+>"         :: "['a, 'a] => 'a"             (infixl 65)
  "<*>"         :: "['a, 'a] => 'a"             (infixl 70)
  assoc         :: "(['a, 'a] => 'a) => bool"

defs
  assoc_def     "assoc f == ALL x y z. f (f x y) z = f x (f y z)"

axclass
  plus_sg < term
  plus_assoc    "assoc (op <+>)"

axclass
  times_sg < term
  times_assoc   "assoc (op <*>)"

end
```

Even if both classes `plus_sg` and `times_sg` represent semigroups in a sense, they are not the same!

[3] Note that $\langle * \rangle$ is used here instead of $*$, because the latter is already declared in theory `HOL` in a slightly different way.

1.2 Basic group theory

The meta 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.

1.2.1 Monoids and Groups

First a small theory that provides some polymorphic constants to be used later for the signature parts:

```
Sigs = HOL +

consts
  "<*>"     :: "['a, 'a] => 'a"       (infixl 70)
  inverse   :: "'a => 'a"
  "1"       :: "'a"                   ("1")

end
```

Next comes the theory `Monoid` which defines class `monoid`[4]:

```
Monoid = Sigs +

axclass
  monoid < term
  assoc          "(x <*> y) <*> z = x <*> (y <*> z)"
  left_unit      "1 <*> x = x"
  right_unit     "x <*> 1 = x"

end
```

So class `monoid` contains exactly those types τ where $\langle * \rangle :: [\tau, \tau] \Rightarrow \tau$ and $\mathtt{1} :: \tau$ are specified appropriately, such that $\langle * \rangle$ is associative and `1` is a left and right unit for $\langle * \rangle$.

Independently of `Monoid`, we now define theory `Group` which introduces a linear hierarchy of semigroups, general groups and abelian groups:

```
Group = Sigs +

axclass
  semigroup < term
  assoc          "(x <*> y) <*> z = x <*> (y <*> z)"
```

[4]Note that multiple class axioms are allowed for user convenience — they simply represent the conjunction of their respective universal closures.

```
axclass
  group < semigroup
  left_unit     "1 <*> x = x"
  left_inverse  "inverse x <*> x = 1"

axclass
  agroup < group
  commut        "x <*> y = y <*> x"

end
```

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

1.2.2 Abstract reasoning

In a sense, axiomatic type classes may be viewed as *abstract theories*. Above class definitions internally generate the following abstract axioms:

```
assoc:        (?x::?'a::semigroup) <*> (?y::?'a::semigroup)
                <*> (?z::?'a::semigroup) = ?x <*> (?y <*> ?z)

left_unit:    1 <*> (?x::?'a::group) = ?x
left_inverse: inverse (?x::?'a::group) <*> ?x = 1

commut:       (?x::?'a::agroup) <*> (?y::?'a::agroup) = ?y <*> ?x
```

All of these contain type variables $\alpha :: c$ that are restricted to types of some class c. These *sort constraints* express a logical precondition for the whole formula. For example, `assoc` states that for all τ, provided that $\tau ::$ `semigroup`, the operation $\langle * \rangle :: [\tau, \tau] \Rightarrow \tau$ is associative.

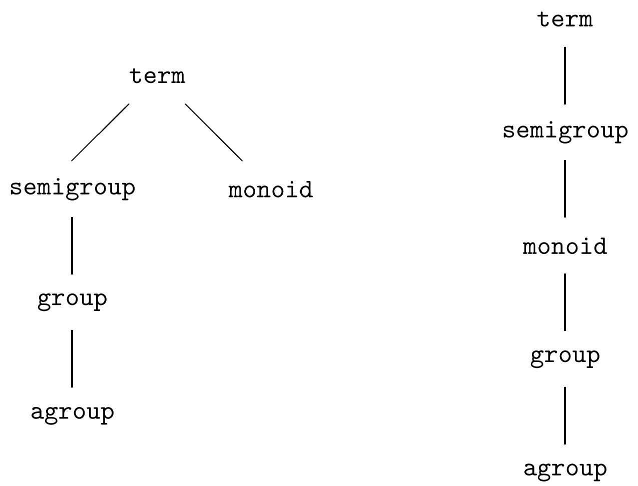
From a purely technical point of view, abstract axioms are just ordinary Isabelle-theorems (of ML-type `thm`). They may be used for Isabelle-proofs without special treatment. Such "abstract proofs" usually yield new abstract theorems. For example, in theory `Group` we may derive:

```
right_unit:      (?x::?'a::group) <*> 1 = ?x
right_inverse:   (?x::?'a::group) <*> inverse ?x = 1
inverse_product: inverse ((?x::?'a::group) <*> (?y::?'a::group)) =
                   inverse ?y <*> inverse ?x
inverse_inv:     inverse (inverse (?x::?'a::group)) = ?x
ex1_inverse:     ALL x::?'a::group. EX! y::?'a::group. y <*> x = 1
```

Abstract theorems (which include abstract axioms, of course) 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`.

1.2.3 Abstract instantiation

Until now, theories `Monoid` and `Group` were independent. Forming their union `Monoid` + `Group` we get the following class hierarchy at the type signature level (left hand side):



We know by definition that $\langle * \rangle$ is associative for monoids, i.e. `monoid` $\subseteq$ `semigroup`. Furthermore we have `assoc`, `left_unit` and `right_unit` for groups (where `right_unit` is derivable from the group axioms), that is `group` $\subseteq$ `monoid`. This way we get the refined class hierarchy shown above at the right hand side.

The additional inclusions `monoid` $\subseteq$ `semigroup` and `group` $\subseteq$ `monoid` are established by logical means and have to be explicitly made known at the type signature level. This is what the theory section `instance` does. So we define `MonoidGroupInsts` as follows:

```
MonoidGroupInsts = Monoid + Group +

instance
  monoid < semigroup            (Monoid.assoc)

instance
  group < monoid                (assoc, left_unit, right_unit)

end
```

The `instance` section does really check that the class inclusions (or type arities) to be added are derivable. To this end, it sets up a suitable goal and attempts a proof with the help of some internal axioms and user supplied theorems. These latter *witnesses* usually should be appropriate type instances of the class axioms (as are `Monoid.assoc` and `assoc`, `left_unit`, `right_unit` above).

The most important internal tool for instantiation proofs is the class introduction rule that is automatically generated by `axclass`. For class `group` this is axiom `groupI`:

```
groupI:    [| OFCLASS(?'a::term, semigroup_class);
              !!x::?'a::term. 1 <*> x = x;
              !!x::?'a::term. inverse x <*> x = 1 |]
           ==> OFCLASS(?'a::term, group_class)
```

There are also axioms `monoidI`, `semigroupI` and `agroupI` of a similar form. Note that `OFCLASS`(τ, c`_class`) expresses class membership $\tau :: c$ as a proposition of the meta-logic.

1.2.4 Concrete instantiation

So far we have covered the case of `instance` c_1 `<` c_2 that could be described as *abstract instantiation* — c_1 is more special than c_2 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 transferred to ISABELLE's type signature level. This form of `instance` is a "safe" variant of the old-style `arities` theory section.

As an example, we show that type `bool` with exclusive-or as operation $\langle * \rangle$, identity as `inverse`, and `False` as `1` forms an abelian group. We first define theory `Xor`:

```
Xor = Group +

defs
  prod_bool_def     "x <*> y   == x ~= (y::bool)"
  inverse_bool_def  "inverse x == x::bool"
  unit_bool_def     "1         == False"

end
```

It is important to note that above `defs` are just overloaded meta-level constant definitions. In particular, type classes are not yet involved at all! This form of constant definition with overloading (and optional primitive recursion over types) would be also possible in traditional versions of HOL that lack type classes. Nonetheless, such definitions are best applied in the context of classes.

Since we chose the `defs` of theory `Xor` suitably, the class membership `bool` :: `agroup` is now derivable as follows:

```
open AxClass;
goal_arity Xor.thy ("bool", [], "agroup");
  Level 0
  OFCLASS(bool, agroup_class)
   1. OFCLASS(bool, agroup_class)
```

```
by (axclass_tac Xor.thy []);
  Level 1
  OFCLASS(bool, agroup_class)
   1. !!(x::bool) (y::bool) z::bool. x <*> y <*> z = x <*> (y <*> z)
   2. !!x::bool. 1 <*> x = x
   3. !!x::bool. inverse x <*> x = 1
   4. !!(x::bool) y::bool. x <*> y = y <*> x
```

The tactic axclass_tac has applied agroupI internally to expand the class definition. It now remains to be shown that the agroup axioms hold for bool. To this end, we expand the definitions of Xor and solve the new subgoals by fast_tac of HOL:

```
by (rewrite_goals_tac [Xor.prod_bool_def, Xor.inverse_bool_def,
        Xor.unit_bool_def]);
  Level 2
  OFCLASS(bool, agroup_class)
   1. !!(x::bool) (y::bool) z::bool. x ~= y ~= z = (x ~= (y ~= z))
   2. !!x::bool. False ~= x = x
   3. !!x::bool. x ~= x = False
   4. !!(x::bool) y::bool. x ~= y = (y ~= x)
by (ALLGOALS (fast_tac HOL_cs));
  Level 3
  OFCLASS(bool, agroup_class)
  No subgoals!
qed "bool_in_agroup";
  val bool_in_agroup = "OFCLASS(bool, agroup_class)"
```

The result is theorem OFCLASS(bool, agroup_class) which expresses bool :: agroup as a meta-proposition. This is not yet known at the type signature level, though.

What we have done here by hand, can be also performed via instance in a similar way behind the scenes. This may look as follows[5]:

```
BoolGroupInsts = Group +

defs
  prod_bool_def     "x <*> y   == x ~= (y::bool)"
  inverse_bool_def  "inverse x == x::bool"
  unit_bool_def     "1         == False"

instance
  bool :: agroup                {| ALLGOALS (fast_tac HOL_cs) |}

end
```

This way, we have bool :: agroup in the type signature of BoolGroupInsts, and all abstract group theorems are transferred to bool for free.

Similarly, we could now also instantiate our group theory classes to many

[5]Subsequently, theory Xor is no longer required.

other concrete types. For example, `int` :: `agroup` (by defining $\langle * \rangle$ as addition, `inverse` as negation and `1` as zero, say) or `list` :: (`term`)`semigroup` (e.g. if $\langle * \rangle$ is list append). Thus, the characteristic constants $\langle * \rangle$, `inverse`, `1` really become overloaded, i.e. have different meanings on different types.

1.2.5 Lifting and Functors

As already mentioned above, HOL-like systems not only support overloaded definitions of polymorphic constants (without requiring type classes), but even primitive recursion over types. That is, definitional equations $c^\tau \equiv t$ may also contain constants of name c on the RHS — provided that these have types that are strictly simpler (structurally) than τ.

This feature enables us to *lift operations*, e.g. to Cartesian products, direct sums or function spaces. Below, theory `ProdGroupInsts` lifts $\langle * \rangle$ componentwise to two-place Cartesian products $\alpha \times \beta$:

```
ProdGroupInsts = Prod + Group +

defs
  prod_prod_def "p <*> q == (fst p <*> fst q, snd p <*> snd q)"

instance
  "*" :: (semigroup, semigroup) semigroup
    {| simp_tac (prod_ss addsimps [assoc]) 1 |}
end
```

Note that `prod_prod_def` basically has the form $\vdash \langle * \rangle^{\alpha \times \beta} \equiv \ldots \langle * \rangle^{\alpha} \ldots \langle * \rangle^{\beta} \ldots$, i.e. the recursive occurrences $\langle * \rangle^{\alpha}$ and $\langle * \rangle^{\beta}$ are really at "simpler" types.

It is very easy to see that associativity of $\langle * \rangle^{\alpha}$, $\langle * \rangle^{\beta}$ transfers to $\langle * \rangle^{\alpha \times \beta}$. Hence the two-place type constructor $\times$ maps semigroups into semigroups. This fact is proven and put into ISABELLE's type signature by above `instance` section.

Thus, if we view class instances as "structures", overloaded constant definitions with primitive recursion over types indirectly provide some kind of "functors" (mappings between abstract theories, that is).

1.3 Syntactic classes

There is still a feature of ISABELLE's type system left that we have not yet used: When declaring polymorphic constants $c :: \tau$, the type variables occurring in τ may be constrained by type classes (or even general sorts). Note that by default, in ISABELLE/HOL the declaration:

```
<*> :: ['a, 'a] => 'a
```

is actually an abbreviation for:

```
<*> :: ['a::term, 'a::term] => 'a::term
```

Since class term is the universal class of HOL, this is not really a restriction at all. A less trivial example is the following theory Product:

```
Product = HOL +

axclass
  product < term

consts
  "<*>" :: "['a::product, 'a] => 'a"      (infixl 70)

end
```

Here class product is defined as subclass of term, but without any additional axioms. This effects in logical equivalence of product and term, since productI is as follows:

```
productI: OFCLASS(?'a::logic, term_class) ==>
            OFCLASS(?'a::logic, product_class)
```

I.e. term $\subseteq$ product. The converse inclusion is already contained in the type signature of theory Product.

Now, what is the difference of declaring $\langle * \rangle :: [\alpha :: \texttt{product}, \alpha] \Rightarrow \alpha$ vs. declaring $\langle * \rangle :: [\alpha :: \texttt{term}, \alpha] \Rightarrow \alpha$? In this special case (where product $\equiv$ term), it should be obvious that both declarations are the same from the logic's point of view. It is even most sensible in the general case not to attach any *logical* meaning to sort constraints occurring in constant *declarations* (see [Wenzel, 1994, page 79] for more details).

On the other hand there are syntactic differences, of course. Constants $\langle * \rangle^{\tau}$ are rejected by the type checker, unless τ :: product is part of the type signature. In our example, this arity may be always added when required by means of a trivial instance.

Thus, we can follow this discipline: 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 $\tau :: c$.

This is done for class product and type bool in theory ProductInsts below:

```
ProductInsts = Product +

instance
  bool :: product

defs
  prod_bool_def "x <*> y  == x & y::bool"

end
```

Note that instance bool :: product does not require any witnesses, since this statement is logically trivial. Nonetheless, instance really performs a proof.

Only after `bool` :: `product` is made known to the type checker, does `prod_bool_def` become syntactically well-formed.

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 $\langle * \rangle$ on `bool` after having instantiated `bool` :: `product`. ISABELLE does not require these definitions, which is in contrast to programming languages like Haskell.

While ISABELLE type classes and those of Haskell are almost the same as far as type checking and type inference are concerned, there are major 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` won't make much sense in ISABELLE, because its meta-logic has no corresponding notion of "providing operations" or "names".

1.4 Defining natural numbers in FOL

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:

```
NatClass = FOL +

consts
  "0"           :: "'a"                                  ("0")
  Suc           :: "'a => 'a"
  rec           :: "['a, 'a, ['a, 'a] => 'a] => 'a"

axclass
  nat < term
  induct        "[| P(0); !!x. P(x) ==> P(Suc(x)) |] ==> P(n)"
  Suc_inject    "Suc(m) = Suc(n) ==> m = n"
  Suc_neq_0     "Suc(m) = 0 ==> R"
  rec_0         "rec(0, a, f) = a"
  rec_Suc       "rec(Suc(m), a, f) = f(m, rec(m, a, f))"

consts
  "+"           :: "['a::nat, 'a] => 'a"                 (infixl 60)

defs
  add_def       "m + n == rec(m, n, %x y. Suc(y))"

end
```

`NatClass` is an abstract version of `Nat`[6]. Basically, we have just replaced all occurrences of *type* `nat` by α and used the natural number axioms to define *class* `nat`[7]. Thus class `nat` contains exactly those types τ that are isomorphic

[6] See directory `FOL/ex`.

[7] There's a snag: The original recursion operator `rec` had to be made monomorphic, in a sense.

to "the" natural numbers (with signature `0`, `Suc`, `rec`).
Furthermore, theory `NatClass` defines an "abstract constant" + based on class `nat`.

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 reasoning is always possible — independent of satisfiability. The meta-logic won't break, even if some class (or general sort) turns out to be empty ("inconsistent") later[8].
For example, we may derive the following abstract natural numbers theorems:

```
add_0:    0 + (?n::?'a::nat) = ?n
add_Suc:  Suc(?m::?'a::nat) + (?n::?'a::nat) = Suc(?m + ?n)
```

See also file `FOL/ex/NatClass.ML`. Note that this required only some trivial modifications of the original `Nat.ML`.

2 The user interface of ISABELLE's axclass package

The actual axiomatic type class package of ISABELLE/PURE mainly consists of two new theory sections: `axclass` and `instance`. Some typical applications of these have already been demonstrated in §1, below their syntax and semantics are presented more completely.

2.1 The `axclass` section

Within theory files, `axclass` introduces an axiomatic type class definition. Its concrete syntax is:

`axclass`
  c `<` c_1`, ... ,`c_n
  id_1 axm_1
  $\vdots$
  id_m axm_m

Where $c, c_1, \ldots, c_n$ are classes (category *id* or *string*) and $axm_1, \ldots, axm_m$ (with $m \geq 0$) are formulas (category *string*).
Class c has to be new, and sort $\{c_1, \ldots, c_n\}$ a subsort of `logic`. Each class axiom axm_j may contain any term variables, but at most one type variable (which need not be the same for all axioms). The sort of this type variable has to be a supersort of $\{c_1, \ldots, c_n\}$.

The `axclass` section declares c as subclass of $c_1, \ldots, c_n$ to the type signature.
Furthermore, $axm_1, \ldots, axm_m$ are turned into the "abstract axioms" of c with names $id_1, \ldots, id_m$. This is done by replacing all occurring type variables by

[8] As a consequence of an old bug, this is *not* true for pre-ISABELLE94-2 versions.

$\alpha :: c$. Original axioms that do not contain any type variable will be prefixed by the logical precondition $\texttt{OFCLASS}(\alpha :: \texttt{logic}, c\texttt{\_class})$.

Another axiom of name c`I` — the "class c introduction rule" — is built from the respective universal closures of $axm_1, \ldots, axm_m$ appropriately.

2.2 The `instance` section

Section `instance` proves class inclusions or type arities at the logical level and then transfers these into the type signature.

Its concrete syntax is:

$$
\begin{array}{l}
\texttt{instance} \\
\quad [\ c_1\ \texttt{<}\ c_2 \mid t\ \texttt{::}\ \texttt{(}sort_1\texttt{,}\ \ldots\ \texttt{,}sort_n\texttt{)}\, sort\ ] \\
\quad [\ \texttt{(}name_1\texttt{,}\ \ldots\ \texttt{,}name_m\texttt{)}\ ] \\
\quad [\ \{ \mid text \mid \}\ ]
\end{array}
$$

Where c_1, c_2 are classes and t is an n-place type constructor (all of category *id* or *string*). Furthermore, $sort_i$ are sorts in the usual ISABELLE-syntax.

Internally, `instance` first sets up an appropriate goal that expresses the class inclusion or type arity as a meta-proposition. Then tactic `AxClass.axclass_tac` is applied with all preceding meta-definitions of the current theory file and the user-supplied witnesses. The latter are $name_1, \ldots, name_m$, where *id* refers to an ML-name of a theorem, and *string* to an axiom of the current theory node[9].

Tactic `AxClass.axclass_tac` first unfolds the class definition by resolving with rule c`I`, and then applies the witnesses according to their form: Meta-definitions are unfolded, all other formulas are repeatedly resolved[10] with.

The final optional argument *text* is ML-code of an arbitrary user tactic which is applied last to any remaining goals.

Because of the complexity of `instance`'s witnessing mechanisms, new users of the axclass package are advised to only use the simple form `instance` $\ldots$ $(id_1, \ldots, id_m)$, where the identifiers refer to theorems that are appropriate type instances of the class axioms. This typically requires an auxiliary theory, though, which defines some constants and then proves these witnesses.

References

[Nipkow, 1993a] T. Nipkow. Axiomatic Type Classes (in Isabelle). Presentation at the workshop *Types for Proof and Programs*, Nijmegen, 1993.

[Nipkow, 1993b] T. Nipkow. Order-Sorted Polymorphism in Isabelle. In G. Huet, G. Plotkin, editors, *Logical Environments*, pp. 164–188, Cambridge University Press, 1993.

[9] Thus, the user may reference axioms from above this `instance` in the theory file. Note that new axioms appear at the ML-toplevel only after the file is processed completely.

[10] This is done in a way that enables proper object-*rules* to be used as witnesses for corresponding class axioms.

[Paulson, 1994] L.C. Paulson. *Isabelle — A Generic Theorem Prover*. LNCS 828, 1994.

[Wenzel, 1994] M. Wenzel. *Axiomatische Typ-Klassen in Isabelle*. Diplomarbeit, TU Mnchen, 1994.