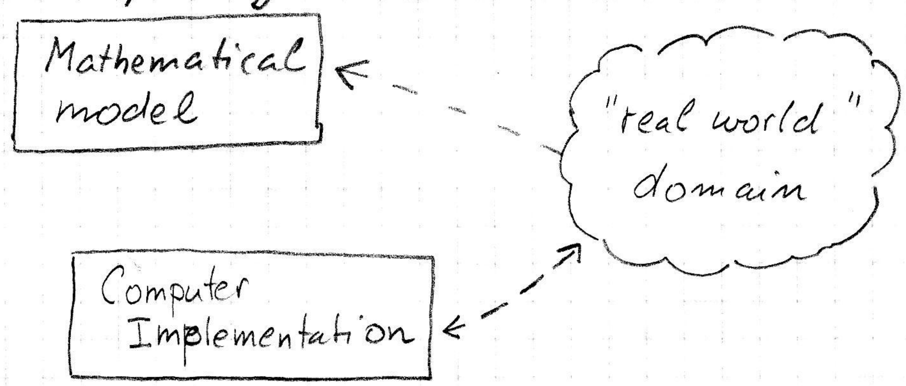
SPECIFICATIONS VIA REALIZABILITY

Andrej Bauer & Chris Stone

Dagstuhl, January 2006

(1) MOTIVATION

Typical situation in real number computing (or scientific computing in general):



Can we give a connection between the two? Goals:

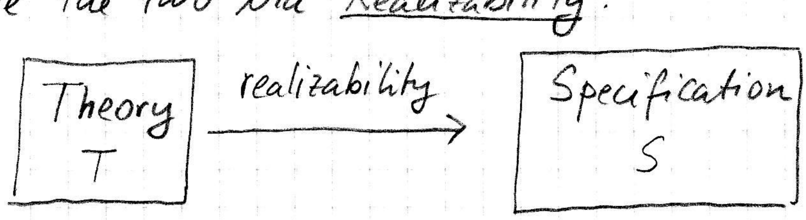
1) Such a connection may help develop data-structures & algorithms
2) May be a guarantee of correct design
3) May expose weaknesses and over-idealizations in the mathematical model.

One approach:

View "mathematical model" as a definition of one or more mathematical structures, given in the language of first-order logic.

View "computer implementation" as an implementation of one or more data structures (modules, classes, ...) in a given programming language.

Relate the two via <u>Realizability</u>:



An implementation may be checked against S

Note: We cannot expect that a mere description of a theory T will already give us an *implementation* satisfying S. This would amount to having a procedure which shows an arbitrary T to be consistent (or not).

An implementation *could* be extracted automatically from a formal construction of a *model* of T (cf. "program extraction from proofs"). However, this would typically not give one very efficient algorithms.

(2) REALIZABILITY IN 10 MINUTES

Realizability is a particular interpretation of logic in terms of a model of computation.

Originally invented by S.C. Kleene (1945).

Possible model of computation:

1) Turing Machines ⟶ Recursive Analysis
2) Type II Turing Machines ⟶ Type II Effectivity
3) Domain Theory ⟶ Domain Representations
4) Programming Language P ⋮

Realizability opens the door to a systematic study of models of computable/effective analysis.

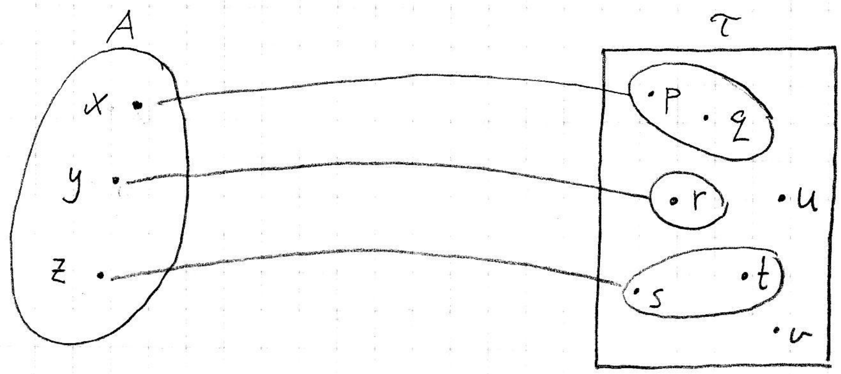
Today: use a programming language P to obtain Specifications that *actually* compile on a computer.

P must be a reasonable language (e.g. ML). It should support higher-order functions, but concievably this is not an essential requirement.

The essential idea of realizability is implicitly known to every programmer:

In order to compute with the elements of a set, they must first be represented (<u>realized</u>) by suitable values of a suitable datatype.

A set (e.g. "real numbers") Datatype $|A| = \tau$



x is realized by p (and also q)

Three equivalent views:

$p \Vdash_A x$ as a realizability relation $\Vdash_A$

$\delta_A(p) = x$ as a representation $\delta_A : \tau \to A$

$p =_A q$ as a partial equivalence relation $=_A$ on τ (A is isomorphic as a set to the set of equivalence classes of $=_A$)

We call such a triple $(A, |A|, \Vdash_A)$ a <u>modest set</u>, equivalently a <u>representation</u> $(\delta_A : |A| \to A)$, equivalently a <u>PER</u> (partial equivalence relation) $(|A|, =_A)$.

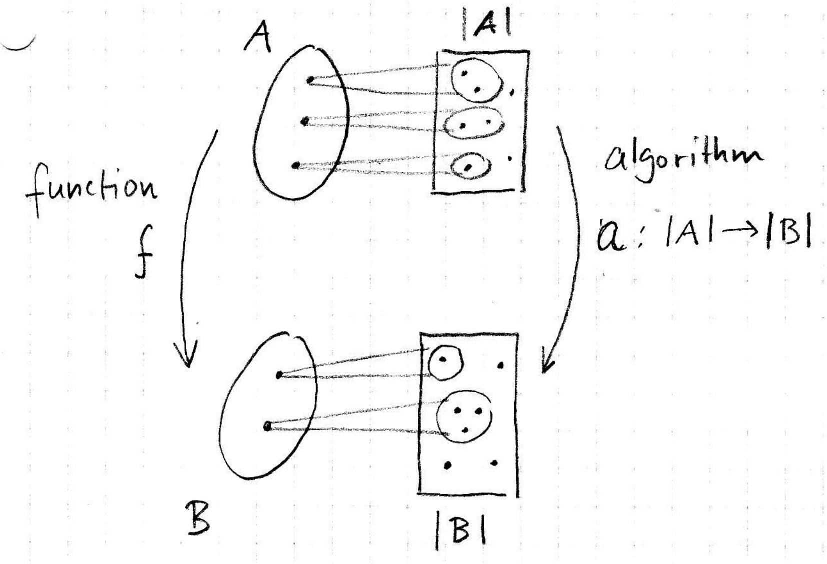
<u>Note</u>:

1) Every $x \in A$ must have at least one realizer

2) Not every $u \in \tau$ need realize something. Define:

$$\|A\| = \{ u \in |A| \;;\; \exists x \in A.\, u \Vdash_A x \}$$
$$= \operatorname{dom} \delta_A$$
$$= \{ u \in |A| \;;\; u =_A u \}.$$

Realized maps:



Must satisfy:

$$p \Vdash_A x \Rightarrow a(p) \Vdash_B f(x)$$

(equivalently:

$$f(\delta_A(p)) = \delta_B(a(p))$$

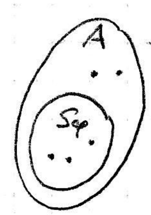
$$p =_A q \Rightarrow a(p) =_B a(q)$$

Modest sets and realized maps form a category. Mod(P)
This category supports an interpretation of first-order logic.

Realizability logic

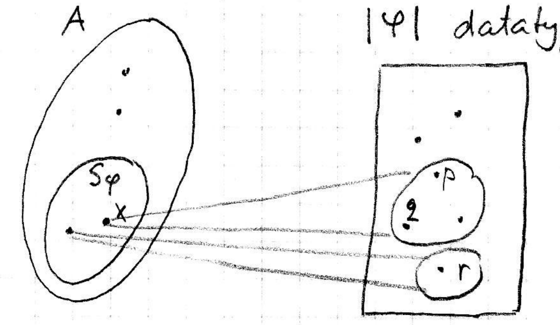
Classical view: a predicate $\varphi(x)$ on a set A is a subset of A,

$$S_\varphi = \{x \in A \mid \varphi(x)\}$$



But in computing it may be important to "know computationally" why a given $x \in A$ satisfies $\varphi(x)$.

Realizability predicate:



$$p \Vdash \varphi(x)$$

"p is a computational witness (reason, realizer) which shows that $\varphi(x)$ holds"

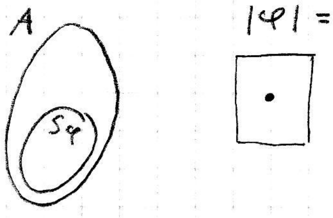
Example:

1) "point x is inside solid S because it belongs to mesh cell $c \in S$"

$$c \Vdash \text{"x is inside } S\text{"}$$

A trivial case of realizability predicate:

$|\varphi| = \text{unit}$ (one-point datatype)



The computational reason does not say anything interesting

A predicate φ for which $|\varphi| = \text{unit}$ is <u>stable</u> (<u>or classical</u>). It has no computational content.

In general: $|\varphi|$ the datatype of values which represent the <u>computational content</u> of φ.

(3) THEORIES AND SPECIFICATIONS

A <u>first-order theory</u> (e.g. the theory of real numbers) consists of:

- basic sets (sorts)
- basic constants and operations
- basic relations
- axioms expressed in first-order logic

We may also list some theorems which, although they follow from the axioms, are important to us and deserve to be mentioned.

A <u>specification</u> (e.g. for exact real arithmetic) consists of:

- abstract datatypes <u>with</u> PER's
- value declarations
- <u>assertions</u> about values

} An implementation provides concrete datatypes and values which satisfy the assertions.

The realizability interpretation translates a theory to a specification.

- The logic of the theory is <u>constructive</u> (because the Law of Excluded Middle does not have a realizer!)
- The logic of assertions is <u>classical</u> (because realizability relation extracts the computational content and puts it in the datatype).

set A ⟶ type A (with a PER $=_A$)
constant $x \in A$ ⟶ val x : A ($x \in \|A\|$)
function $f : A \to B$ ⟶ val f : A → B
predicate φ on A ⟶ type $|\varphi|$ (with a realizability relation $p \Vdash \varphi(x)$)
Stable predicate φ on A ⟶ a subset of equivalence classes of $|A|$.
axiom φ ⟶ val a : $|\varphi|$ + assertion "$a \Vdash \varphi$"

<u>Examples</u>:

Theory of real numbers ⟶ Exact Real Arithmetic
Free join-semilattice ⟶ Finite Sets
Natural Numbers ⟶ Big natural number arithmetic
⋮ ⋮

<u>Note</u>: The formulation of the theory influences the ingridients of the specification, but it <u>does not</u> enforce any restrictions on how the basic types and constants are implemented.

⇒ Even if we use mathematically pleasing and natural axiomatizations, we still can choose <u>efficient</u> datatypes and algorithms to implement them.

E.g. even if natural numbers are axiomatized in terms of a successor function, we may still implement them in binary notation (and <u>not</u>, say as an inductive datatype nat = Zero | Succ of nat).

(other examples are given by live demo)

```
# Natural numbers as the initial successor algebra

theory Natural =
thy
  set nat
  const zero : nat
  const succ : nat -> nat

  # The axiom of recursion states that nat is the inital
  # successor algebra. Thus it takes as a parameter an
  # arbitrary successor algebra A = (s, x, f)

  axiom recursion (A : thy
                         set s
                         const x : s
                         const f : s -> s
                       end) =
  some1 (g : nat -> A.s) . (
    g zero = A.x and
    all (n : nat) . (g (succ n) = A.f (g n))
  )
end
```

```
module type Natural =
sig
  type nat
  (**  Assertion per_nat =  PER(=nat=)
  *)

  val zero : nat
  (**  Assertion zero_total =  zero : ||nat||
  *)

  val succ : nat -> nat
  (**  Assertion succ_total =
         all (x:nat, y:nat).  x =nat= y -> succ x =nat= succ y
  *)

  module Recursion : functor (A : sig
                                    type s
                                    (**  Assertion per_s =  PER(=s=)
                                    *)

                                    val x : s
                                    (**  Assertion x_total =  x : ||s||
                                    *)

                                    val f : s -> s
                                    (**  Assertion f_total =
                                           all (y:s, v:s).  y =s= v ->
                                             f y =s= f v
                                    *)
                                  end) ->
                      sig
                        val recursion : nat -> A.s
                        (**  Assertion recursion =
                               (all (x:nat, y:nat).  x =nat= y ->
                                  recursion x =A.s= recursion y) and
                               recursion zero =A.s= A.x and
                               (all (n:||nat||).
                                  recursion (succ n) =A.s= A.f (recursion n)) and
                               (all (u:nat -> A.s).
                                  (all (x:nat, y:nat).  x =nat= y ->
                                     u x =A.s= u y) -> u zero =A.s= A.x and
                                  (all (n:||nat||).
                                     u (succ n) =A.s= A.f (u n)) ->
                                  all (x:nat, y:nat).  x =nat= y ->
                                    recursion x =A.s= u y)
                        *)
                      end
end
```

```
# We axiomatize the real numbers as the ordered Archimedean Cauchy
# complete field. The theory of reals is here built from natural
# numbers. It would presumably be better to start with a suitable
# dense linearly ordered field, such as the rationals or the dyadic
# rationals.

theory Real (N : Natural) =
thy
  set r
  constant zero : r
  constant one : r
  constant ( + ) : r -> r -> r      # addition
  constant ( * ) : r -> r -> r      # multiplication
  constant ( - ) : r -> r -> r      # subtraction
  constant ( ~- ): r -> r           # opposite value

  # for division we need the set of non-zero reals r'

  set r' = { x : r | not (x = zero) }

  constant inv : r' -> r'           # division

  ##################################################
  # Basic algebraic structure

  #### (r, zero, +, ~-) is a commutative group

  # Assume variables x, y, z range over r
  implicit x, y, z : r

  axiom assoc_plus x y z =
        (x + y) + z = x + (y + z)

  axiom comm_plus x y =
        x + y = y + x

  axiom zero_plus x =
        x + zero = zero

  axiom inverse_plus x =
        x + (~- x) = zero

  #### (r, zero, one, +, ~-, *) is a commutative ring with unit

  axiom assoc_mult x y z =
        (x * y) * z = x * (y * z)

  axiom comm_mult x y =
        x * y = y * x

  axiom one_mult x =
        x * one = x

  axiom distributivity x y z =
        (x + y) * z = (x * z) + (y * z)

  #### r is a field

  axiom field x =
        not (x = zero) -> x * (inv x) = one

  ##################################################
  # linear order on R

  #### we take into account the fact that < and <= are stable

  stable relation ( < ) : r * r
```

```
stable relation ( <= ) x y = not (y < x)

axiom assymetry x y =
      not (x < y and y < x)

axiom linearity x y z =
      x < y -> x < z or z < y

axiom not_apart x y =
      not (x < y or y < x) -> x = y

axiom order_plus x y z =
      x < y -> x + z < y + z

axiom order_mult x y z =
      x < y and zero < z -> x * z < y * z

axiom order_positive x y =
      zero < x and zero < y -> zero < x * y

axiom order_inverse x =
      zero < x -> zero < (inv (x :> r') :< r)

#### The Archimedean axiom

# First we need an embedding of natural numbers in r
const i : N.nat -> r

axiom i_injects =
  i N.zero = zero and
  all (n : N.nat) . (i (N.succ n) = one + i n)

axiom archimedean x =
      zero < x -> some (n : N.nat) . x < i n

####################################################
# the lattice structure on R

# To express Cahuchy completion, we want absolute values,
# which are easiest to get via min and max.

const max : r -> r -> r
const min : r -> r -> r

axiom max_is_lub x y =
      x <= max x y and y <= max x y and
      all z. ((x <= z and y <= z) -> max x y <= z)

axiom min_is_glb x y =
      min x y <= x and min x y <= y and
      all z. ((z <= x and z <= y) -> z <= min x y)

# Now the absolute value may be defined. Note that
# this does NOT force us to _implement_ abs in this
# way. We may do it however we like it, as long as it
# is equivalent to this definition.

const (abs : r -> r) = lam x . (max x (~- x))

####################################################
# Cauchy completeness

# We will postulate Cauchy completness for rapdily converging
# Cauchy sequences. The ordinary definition would work just as
# well but would be slightly more verbose.

# We want powers of 2^-n. The quickest way to do this is to
# have a function halfTo: n |--> (1/2)^n.
```

```
  const halfTo : N.nat -> r

  axiom halfTo_is_what_you_think_it_is =
    halfTo N.zero = one and
    all (n : N.nat) . (halfTo (N.succ n) + halfTo (N.succ n) = halfTo n)


  # The axiom of Cauchy completness

  axiom cauchy (a : N.nat -> r) =
  begin
    (all (n : N.nat) . (abs (a (N.succ n) - a n) < halfTo n))
    ->
    some (x : r) . all (n : N.nat) . (
      abs (x - a (N.succ n)) < halfTo n
    )
  end

end
```

```
module type Real =
functor (N : Natural) ->
 sig
   type r
   (**  Assertion per_r =  PER(=r=)
   *)

   val zero : r
   (**  Assertion zero_total =  zero : ||r||
   *)

   val one : r
   (**  Assertion one_total =  one : ||r||
   *)

   val (+) : r -> r -> r
   (**  Assertion (+)_total =
          all (x:r, y:r).  x =r= y ->
            all (x':r, y':r).  x' =r= y' -> x + x' =r= y + y'
   *)

   val ( * ) : r -> r -> r
   (**  Assertion ( * )_total =
          all (x:r, y:r).  x =r= y ->
            all (x':r, y':r).  x' =r= y' -> x * x' =r= y * y'
   *)

   val (-) : r -> r -> r
   (**  Assertion (-)_total =
          all (x:r, y:r).  x =r= y ->
            all (x':r, y':r).  x' =r= y' -> x - x' =r= y - y'
   *)

   val (~-) : r -> r
   (**  Assertion (~-)_total =  all (x:r, y:r).  x =r= y -> (~-) x =r= (~-) y
   *)

   type r' = r
   (**  Assertion r'_def_total (k:||r||) =  k : ||r'|| <-> k : ||r|| and
                not (k =r= zero)

        Assertion r'_def_per (x:||r||, y:||r||) =  x =r'= y <-> x =r= y
   *)

   val inv : r -> r
   (**  Assertion inv_total =  all (x:r, y:r).  x =r'= y -> inv x =r'= inv y
   *)


   (**  Assertion assoc_plus (x:||r||, y:||r||, z:||r||) =
                (x + y) + z =r= x + (y + z)
   *)


   (**  Assertion comm_plus (x:||r||, y:||r||) =  x + y =r= y + x
   *)


   (**  Assertion zero_plus (x:||r||) =  x + zero =r= zero
   *)


   (**  Assertion inverse_plus (x:||r||) =  x + (~-) x =r= zero
   *)


   (**  Assertion assoc_mult (x:||r||, y:||r||, z:||r||) =
                (x * y) * z =r= x * (y * z)
```

```
 *)


(**  Assertion comm_mult (x:||r||, y:||r||) =  x * y =r= y * x
 *)


(**  Assertion one_mult (x:||r||) =  x * one =r= x
 *)


(**  Assertion distributivity (x:||r||, y:||r||, z:||r||) =
            (x + y) * z =r= x * z + y * z
 *)


(**  Assertion field (x:||r||) =  not (x =r= zero) ->
            x * inv (assure (not (x =r= zero)) in x) =r= one
 *)


(**  Assertion predicate_(<) =
      PREDICATE((<), r * r, lam t u.(pi0 t =r= pi0 u and pi1 t =r= pi1 u))
 *)

type _leq = top
(**  Assertion (<=)_def (x:||r||, y:||r||) =  x <= y <-> not (y < x)
 *)


(**  Assertion assymetry (x:||r||, y:||r||) =  not (x < y and y < x)
 *)

val linearity : r -> r -> r -> ['or0 | 'or1]
(**  Assertion linearity (x:||r||, y:||r||, z:||r||) =  x < y ->
            linearity x y z = 'or0 and x < z cor
            linearity x y z = 'or1 and z < y
 *)


(**  Assertion not_apart (x:||r||, y:||r||) =
            (all (u:['or0 | 'or1]).
               not (u = 'or0 and x < y cor u = 'or1 and y < x)) -> x =r= y
 *)


(**  Assertion order_plus (x:||r||, y:||r||, z:||r||) =  x < y ->
            x + z < y + z
 *)


(**  Assertion order_mult (x:||r||, y:||r||, z:||r||) =  x < y and
            zero < z -> x * z < y * z
 *)


(**  Assertion order_positive (x:||r||, y:||r||) =  zero < x and
            zero < y -> zero < x * y
 *)


(**  Assertion order_inverse (x:||r||) =  zero < x ->
            zero < inv (assure (not (x =r= zero)) in x)
 *)

val i : N.nat -> r
(**  Assertion i_total =
      all (x:N.nat, y:N.nat).  x =N.nat= y -> i x =r= i y
```

```
  *)


  (**  Assertion i_injects =  i N.zero =r= zero and
         (all (n:||N.nat||).  i (N.succ n) =r= one + i n)
  *)

  val archimedean : r -> N.nat
  (**  Assertion archimedean (x:||r||) =  zero < x ->
              archimedean x : ||N.nat|| and x < i (archimedean x)
  *)

  val max : r -> r -> r
  (**  Assertion max_total =
         all (x:r, y:r).  x =r= y ->
           all (x':r, y':r).  x' =r= y' -> max x x' =r= max y y'
  *)

  val min : r -> r -> r
  (**  Assertion min_total =
         all (x:r, y:r).  x =r= y ->
           all (x':r, y':r).  x' =r= y' -> min x x' =r= min y y'
  *)


  (**  Assertion max_is_lub (x:||r||, y:||r||) =  x <= max x y and
              y <= max x y and
              (all (z:||r||).  x <= z and y <= z -> max x y <= z)
  *)


  (**  Assertion min_is_glb (x:||r||, y:||r||) =  min x y <= x and
              min x y <= y and
              (all (z:||r||).  z <= x and z <= y -> z <= min x y)
  *)

  val abs : r -> r
  (**  Assertion abs_def =
         all (x:r, y:r).  x =r= y -> abs x =r= max y ((~-) y)
  *)

  val halfTo : N.nat -> r
  (**  Assertion halfTo_total =
         all (x:N.nat, y:N.nat).  x =N.nat= y -> halfTo x =r= halfTo y
  *)


  (**  Assertion halfTo_is_what_you_think_it_is =
         halfTo N.zero =r= one and
         (all (n:||N.nat||).
            halfTo (N.succ n) + halfTo (N.succ n) =r= halfTo n)
  *)

  val cauchy : (N.nat -> r) -> r
  (**  Assertion cauchy (a:||N.nat -> r||) =
              (all (n:||N.nat||).  abs (a (N.succ n) - a n) < halfTo n) ->
              cauchy a : ||r|| and
              (all (n:||N.nat||).  abs (cauchy a - a (N.succ n)) < halfTo n)
  *)
end
```

```
# The theory of a group

theory Group =
thy
  set g

  const e : g

  const ( * ) : g -> g -> g

  const i : g -> g

  implicit x, y, z : g

  axiom unit x =
    e * x = x and x * e = x

  axiom associative x y z =
    (x * y) * z = x * (y * z)

  axiom inverse x =
    x * (i x) =  e and (i x) * x = e

end
```

```
module type Group =
sig
  type g
  (**  Assertion per_g =  PER(=g=)
  *)

  val e : g
  (**  Assertion e_total =  e : ||g||
  *)

  val ( * ) : g -> g -> g
  (**  Assertion ( * )_total =
         all (x:g, y:g).  x =g= y ->
           all (x':g, y':g).  x' =g= y' -> x * x' =g= y * y'
  *)

  val i : g -> g
  (**  Assertion i_total =  all (x:g, y:g).  x =g= y -> i x =g= i y
  *)


  (**  Assertion unit (x:||g||) =  e * x =g= x and x * e =g= x
  *)


  (**  Assertion associative (x:||g||, y:||g||, z:||g||) =
              (x * y) * z =g= x * (y * z)
  *)


  (**  Assertion inverse (x:||g||) =  x * i x =g= e and i x * x =g= e
  *)
end
```

```
theory Category =
thy
  set ob    # objects
  set mor   # morphisms

  const id : ob -> mor
  const dom : mor -> ob
  const cod : mor -> ob

  # The set of composable morphisms
  set comp = { p : mor * mor | cod p.0 = dom p.1 }

  # Composition
  const cmp : comp -> mor

  implicit a,b,c : ob
  implicit f,g,h : mor

  axiom id_dom a = dom (id a) = a

  axiom id_cod a = cod (id a) = a

  axiom dom_comp f g =
    dom (cmp (f,g)) = dom f

  axiom cod_comp f g =
    cod (cmp(f,g)) = cod g

  axiom assoc f g h =
    cmp(cmp (f,g),h) = cmp(f, cmp(g,h))

  axiom id_neutral f =
    f = cmp(id(dom f),f) and f = cmp(f,id(cod f))
end
```

```
module type Category =
sig
  type ob
  (**  Assertion per_ob =  PER(=ob=)
  *)

  type mor
  (**  Assertion per_mor =  PER(=mor=)
  *)

  val id : ob -> mor
  (**  Assertion id_total =  all (x:ob, y:ob).  x =ob= y -> id x =mor= id y
  *)

  val dom : mor -> ob
  (**  Assertion dom_total =
         all (x:mor, y:mor).  x =mor= y -> dom x =ob= dom y
  *)

  val cod : mor -> ob
  (**  Assertion cod_total =
         all (x:mor, y:mor).  x =mor= y -> cod x =ob= cod y
  *)

  type comp = mor * mor
  (**  Assertion comp_def_total (k:||mor * mor||) =  k : ||comp|| <->
                pi0 k : ||mor|| and pi1 k : ||mor|| and
                cod (pi0 k) =ob= dom (pi1 k)

       Assertion comp_def_per (t:||mor * mor||, u:||mor * mor||) =
                t =comp= u <-> pi0 t =mor= pi0 u and pi1 t =mor= pi1 u
  *)

  val cmp : mor * mor -> mor
  (**  Assertion cmp_total =
         all (x:mor * mor, y:mor * mor).  x =comp= y -> cmp x =mor= cmp y
  *)


  (**  Assertion id_dom (a:||ob||) =  dom (id a) =ob= a
  *)


  (**  Assertion id_cod (a:||ob||) =  cod (id a) =ob= a
  *)


  (**  Assertion dom_comp (f:||mor||, g:||mor||) =
                dom (cmp (assure (cod f =ob= dom g) in (f,g))) =ob= dom f
  *)


  (**  Assertion cod_comp (f:||mor||, g:||mor||) =
                cod (cmp (assure (cod f =ob= dom g) in (f,g))) =ob= cod g
  *)


  (**  Assertion assoc (f:||mor||, g:||mor||, h:||mor||) =
                cmp
                  (assure (cod (cmp (assure (cod f =ob= dom g) in (f,g))) =ob=
                            dom h)
                   in (cmp (assure (cod f =ob= dom g) in (f,g)),h)) =mor=
                cmp
                  (assure (cod f =ob= dom
                                     (cmp
                                        (assure (cod g =ob= dom h) in (g,h))))
                   in (f,cmp (assure (cod g =ob= dom h) in (g,h))))
  *)
```

```
  (**  Assertion id_neutral (f:||mor||) =
              f -mor- cmp
                         (assure (cod (id (dom f)) -ob- dom f)
                          in (id (dom f),f)) end
              f -mor- cmp
                         (assure (cod f -ob- dom (id (cod f)))
                          in (f,id (cod f)))
  *)
end
```

```
# The theory of a constructive field with an appartness relation
# (a relation which expresses how "unequal" two elements are).

theory Field =
thy

  set s

  relation ( <> ) : s * s  # apartness

  const zero : s
  const one : s
  const ( + ) : s -> s -> s
  const ( * ) : s -> s -> s
  const ( - ) : s -> s -> s
  const ( ~- ) : s -> s

  set s' = { x : s | x <> zero }

  const inv : s' -> s

  implicit x, y, z : s

  #### Apartness axioms

  axiom apart1 x y =
    not (x <> y) <-> x = y

  axiom apart2 x y =
    x <> y -> y <> x

  axiom apart3 x y z =
    x <> y -> (x <> z or y <> z)

  #### (s, zero, +, ~-) is a commutative group

  axiom assoc_plus x y z =
        (x + y) + z = x + (y + z)

  axiom comm_plus x y =
        x + y = y + x

  axiom zero_plus x =
        x + zero = zero

  axiom inverse_plus x =
        x + (~- x) = zero

  #### (s, zero, one, +, ~-, *) is a commutative ring with unit

  axiom assoc_mult x y z =
        (x * y) * z = x * (y * z)

  axiom comm_mult x y =
        x * y = y * x

  axiom one_mult x =
        x * one = x

  axiom distributivity x y z =
        (x + y) * z = (x * z) + (y * z)

  #### s is a field

  axiom field x =
        x <> zero -> x * (inv x) = one

end
```

```
module type Field =
sig
  type s
  (**  Assertion per_s =  PER(=s=)
  *)

  type _neq
  (**  Assertion predicate_(<>) =
         PREDICATE((<>), s * s, lam t u.(pi0 t =s= pi0 u and pi1 t =s= pi1 u))
  *)

  val zero : s
  (**  Assertion zero_total =  zero : ||s||
  *)

  val one : s
  (**  Assertion one_total =  one : ||s||
  *)

  val (+) : s -> s -> s
  (**  Assertion (+)_total =
         all (x:s, y:s).  x =s= y ->
           all (x':s, y':s).  x' =s= y' -> x + x' =s= y + y'
  *)

  val ( * ) : s -> s -> s
  (**  Assertion ( * )_total =
         all (x:s, y:s).  x =s= y ->
           all (x':s, y':s).  x' =s= y' -> x * x' =s= y * y'
  *)

  val (-) : s -> s -> s
  (**  Assertion (-)_total =
         all (x:s, y:s).  x =s= y ->
           all (x':s, y':s).  x' =s= y' -> x - x' =s= y - y'
  *)

  val (~-) : s -> s
  (**  Assertion (~-)_total =  all (x:s, y:s).  x =s= y -> (~-) x =s= (~-) y
  *)

  type s' = s * _neq
  (**  Assertion s'_def_total (k:||s * _neq||) =  k : ||s'|| <->
             pi0 k : ||s|| and pi1 k |= pi0 k <> zero

       Assertion s'_def_per (x:||s * _neq||, y:||s * _neq||) =  x =s'= y <->
             pi0 x =s= pi0 y
  *)

  val inv : s * _neq -> s
  (**  Assertion inv_total =
         all (x:s * _neq, y:s * _neq).  x =s'= y -> inv x =s= inv y
  *)


  (**  Assertion apart1 (x:||s||, y:||s||) =
             ((all (r:_neq).  not (r |= x <> y)) -> x =s= y) and
             (x =s= y -> all (r:_neq).  not (r |= x <> y))
  *)

  val apart2 : s -> s -> _neq -> _neq
  (**  Assertion apart2 (x:||s||, y:||s||) =
             all (r:_neq).  r |= x <> y -> apart2 x y r |= y <> x
  *)

  val apart3 : s -> s -> s -> _neq -> ['or0 of _neq | 'or1 of _neq]
  (**  Assertion apart3 (x:||s||, y:||s||, z:||s||) =
             all (r:_neq).  r |= x <> y ->
```

```
                (some (r':_neq).  apart3 x y z r - 'or0 r' and r' |- x <> z) cor
                (some (r':_neq).  apart3 x y z r - 'or1 r' and r' |- y <> z)
  *)


  (**  Assertion assoc_plus (x:||s||, y:||s||, z:||s||) -
              (x + y) + z -s- x + (y + z)
  *)


  (**  Assertion comm_plus (x:||s||, y:||s||) -  x + y -s- y + x
  *)


  (**  Assertion zero_plus (x:||s||) -  x + zero -s- zero
  *)


  (**  Assertion inverse_plus (x:||s||) -  x + (~-) x -s- zero
  *)


  (**  Assertion assoc_mult (x:||s||, y:||s||, z:||s||) -
              (x * y) * z -s- x * (y * z)
  *)


  (**  Assertion comm_mult (x:||s||, y:||s||) -  x * y -s- y * x
  *)


  (**  Assertion one_mult (x:||s||) -  x * one -s- x
  *)


  (**  Assertion distributivity (x:||s||, y:||s||, z:||s||) -
              (x + y) * z -s- x * z + y * z
  *)


  (**  Assertion field (x:||s||) -
              all (r:_neq).  r |- x <> zero ->
                x * inv (assure r' : _neq . r' |- x <> zero in (x,r')) -s- one
  *)
end
```