An Implementation of the Heine-Borel Covering Theorem in Type Theory

Jan Cederquist

Department of Computing Science
University of Göteborg
S-412 96 Göteborg, Sweden
e-mail: ceder@cs.chalmers.se

Abstract. We describe an implementation, in type theory, of a proof of a pointfree formulation of the Heine-Borel covering theorem for intervals with rational endpoints.

1 Introduction

The proof presented here is a complete formalisation of the proof presented in "*A constructive proof of the Heine-Borel covering theorem for formal reals*" [CN]. We describe an implementation, in type theory, of a proof of a pointfree formulation of the Heine-Borel covering theorem for intervals with rational endpoints. The implementations also contain a definition of formal spaces as a type, and definitions of the continuum and the closed rational interval as instances of that type.

The paper is organised as follows: in section 2 we describe the proof-checker Half, in which the implementation has been done, and the type theory it is based on. The rest of the paper is devoted to formal definitions and the proof of the Heine-Borel covering theorem. In section 3 some general definitions are given. In section 4 we define a general formal topology, we also define the notion of compact space and Stone space. Then the rational numbers are defined as an object of an abstract data type. In section 6 the continuum is defined as a formal space and some of its properties are proved. Then the closed rational interval [a,b] is defined as a formal space and compactness of this space is proved.

In order to make this paper readable, we concentrate on the definitions and many proofs (or lemmas and definitions used in these proofs) are left out. In the code these omitted proofs are replaced by the ellipsis ....[1] However, all identifiers used in the proofs presented are defined and the main theorem is given with all details.

2 Description of the Proof-Checker Half

The implementation has been done in the proof-checker Half, developed by

[1] The complete proofs are obtainable from the URL: `ftp://ftp.cs.chalmers.se/pub/users/ceder/heineb/hb.tar`.

Thierry Coquand, using a type-checker and an `emacs`-interface implemented by Dan Synek.

The Half system is a successor to ALF [M]. It is a logical framework based on Martin-Löf's polymorphic type theory with one universe [ML], extended by a *theory* mechanism (similar to the theory mechanism in PVS [OSR]) and *let-expressions* (cf. [C, Br, Ba]).

The system has three levels; **Set**, **Type** and **Kind**. **Set** is an element and a subset of **Type**. Elements can be formed in both **Set** and **Type**; both **Set** and **Type** are closed under function types (Π-types) and disjoint union (Σ-types) and allow recursive definitions. There is also a type **Theory** for theories. **Kind** consists of the types **Set**, **Type** and **Theory**, and function types.

A proof (program) in Half consists of a list of definitions and lemmas, having the form $f(x_1 : T_1, \ldots, x_n : T_n) = e : T$, where the type T_i may depend on the parameters $x_1, \ldots, x_{i-1}$ and e is an expression of type T.

The Π-type is used for expressing dependent function spaces. Given two types A and B, the Π-type of functions from A to B is written $(x : A) \to B$. Elements of $(x : A) \to B$ are functions $\lambda x \to e$, where the abstracted variable x has type A and e is an expression of type B. The elimination form for elements of Π-types is application.

A recursive data type is defined using the reserved word **data**:

$$\begin{array}{l} \mathbf{data}\{ \\ \quad c_1(a_{11} : A_{11}, \ldots, a_{1m} : A_{1m}), \\ \qquad\qquad \vdots \\ \quad c_n(a_{n1} : A_{n1}, \ldots, a_{nk} : A_{nk})\}, \end{array}$$

where A_{ij} is an arbitrary type. Elements are introduced using the constructors c_i

$$c_i \ a_{i1} \cdots a_{ij}$$

and the elimination form, for objects of a recursively defined data type, is the *case-expression*

$$\begin{array}{l} \mathbf{case}\ x\ \mathbf{of}\ \{ \\ \quad c_1 \ a_{11} \cdots a_{1m} \to e_1, \\ \qquad\qquad \vdots \\ \quad c_n \ a_{n1} \cdots a_{nk} \to e_n\}, \end{array}$$

where $e_1, \ldots, e_n$ are expressions of the same type (the type of the case-expression). For example, the set of finite lists may be defined by

$$\mathit{list}(A : \mathbf{Set}) = \mathbf{data}\{\mathit{Nil}, \mathit{Cons}(x : A,\ xs : \mathit{list}\ A)\} : \mathbf{Set}$$

and a list can then be analysed using a case-expression as in the following definition of append:

$$\begin{array}{l} \mathit{append}(A : \mathbf{Set},\ l_1 : \mathit{list}\ A,\ l_2 : \mathit{list}\ A) = \\ \quad \mathbf{case}\ l_1\ of\ \{ \\ \quad\quad \mathit{Nil} \to l_2, \\ \quad\quad \mathit{Cons}\ x\ xs \to \mathit{Cons}\ x\ (\mathit{append}\ A\ xs\ l_2)\} : \mathit{list}\ A. \end{array}$$

Note that, using these recursive definitions on functional form, non-linear inductive types cannot be defined, i.e. dependencies between the parameters cannot be introduced. It turned out that pattern matching together with non-linear inductive definitions is a non-conservative extension of Martin-Löf's type theory (see [H]). The approach taken in Half is to allow only linear inductive definitions. As a consequense, the *Id*-type

$$\frac{a \in A}{id(A, a) \in Id(A, a, a)}$$

is not definable: without dependencies between the parameters there is no way of saying that the two elements are the same. Therefore, for abstract sets, instead of working with sets and the *Id*-type, we work in a more general setting using *setoids*, i.e. sets with equivalence relations. For concrete sets, equalities are explicitly defined. This is also closer to the usual mathematical approach where a set comes together with an equality relation.

A Σ-type is a dependent record $\mathbf{sig}\{t_1 : T_1, \ldots, t_n : T_n\}$, where the type T_i may depend on $t_1, \ldots, t_{i-1}$. An object of a Σ-type is formed by constructing objects of the types T_i, $\mathbf{struct}\{t_1 = e_1, \ldots, t_n = e_n\}$, where e_i is an expression of type T_i. The elimination rule for Σ-types is projection; if M is of type $\mathbf{sig}\{t_1 : T_1, \ldots, t_n : T_n\}$, the value of its i'th component is accessed by $M.t_i$.

Adding Σ-types to the system is a conservative extension of the system; it does not affect the strength of the theory, equivalent definitions can always be obtained using recursive definitions with one constructor. However, to analyse objects of a recursively defined set, case-analysis is required, even if there is only one case to consider.

Theories are lists of definitions and proofs:

$$\begin{array}{l} th = \mathbf{theory}\{ \\ \qquad f_1(a_{11} : A_{11}, \ldots, a_{1m} : A_{1m}) = e_1 : T_1, \\ \qquad\qquad\qquad \vdots \\ \qquad f_n(a_{n1} : A_{n1}, \ldots, a_{nk} : A_{nk}) = e_n : T_n\} \\ \quad : \mathbf{Theory} \end{array}$$

Theories are used to collect definitions and lemmas that logically belong together. Identifiers defined in a theory can be accessed from outside: if th is a theory and f_i an identifier defined in th, then the value of f_i is reached by $th.f_i$.

By defining functions giving theories as result, a notion of parametrised theory is obtained. Identifiers defined in a parametrised theory can then be accessed from outside, provided they are given proper parameters. Also the notion of (parametrised) theory is a conservative extension of the system: functions occuring in a parametrised theory can always be parametrised themselves and defined outside the theory.

The *let-expressions* are used for local lemmas and abbreviations:

$$\mathbf{let}\ x = e_1 : T\ \mathbf{in}\ e_2$$

In the environment ρ, the expression above computes to $e_2(\rho, x = e_1\rho)$, i.e. the value of e_2 in the environment ρ extended with $x = e_1\rho$.

Expressions of this language are thus formed by

sorts	**Set**, **Type** and **Theory**
Π-types	$(x : A) \to B$
abstractions	$\lambda x \to e$
applications	$a\ b$
Σ-types	$\mathbf{sig}\{a_1 : A_1, \ldots, a_n : A_n\}$
structures	$\mathbf{struct}\{a_1 = e_1, \ldots, a_n = e_n\}$
projections	$b.a_i$
rec. def. types	$\mathbf{data}\{$ $c_1(a_{11} : A_{11}, \ldots, a_{1m} : A_{1m}),$ $\vdots$ $c_n(a_{n1} : A_{n1}, \ldots, a_{nk} : A_{nk})\}$
constructors	c_i
case expressions	$\mathbf{case}\ x\ \mathbf{of}\ \{$ $c_1\ a_{11} \cdots a_{1m} \to e_1,$ $\vdots$ $c_n\ a_{n1} \cdots a_{nk} \to e_n\}$
let expressions	$\mathbf{let}\ x = e_1 : T\ \mathbf{in}\ e_2$
theories	$\mathbf{theory}\{$ $f_1(a_{11} : A_{11}, \ldots, a_{1m} : A_{1m}) = e_1 : T_1,$ $\vdots$ $f_n(a_{n1} : A_{n1}, \ldots, a_{nk} : A_{nk}) = e_n : T_n\}$
projections	$th.f_i$
variables	x

The system also allows mutual recursive definitions. But this has not been used in the proofs in this paper, and we have also avoided mutual recursion between a function f and functions locally defined in f.

There is a "size check" for inductively defined types. The type

$$\begin{array}{l}\mathbf{data}\{\\ \quad c_1(a_{11} : A_{11}, \ldots, a_{1m} : A_{1m}),\\ \qquad\qquad \vdots \\ \quad c_n(a_{n1} : A_{n1}, \ldots, a_{nk} : A_{nk})\}\end{array}$$

lives in **Set** or **Type** if all A_{ij}'s live in **Set** or **Type**, respectively.

The definitional equality is a combination of structural equality and equal by name; for checking equality of "complex" structures, i.e. **data**, **sig**, **struct** and **case**, comparision "by name" is used. This means for instance that in

$$\begin{array}{l} Bool = \mathbf{data}\{False, True\} : \mathbf{Set},\\ Bool' = \mathbf{data}\{False, True\} : \mathbf{Set},\\ Bool'' = Bool : \mathbf{Set}\end{array}$$

Bool and *Bool'* are not equal, but *Bool* and *Bool"* are. This is the approach taken for several strongly typed languages.

The presence of both **Set** and **Type** in Half, where **Set** corresponds to a universe, allows a more abstract reasoning than is possible in a system without a universe. We show this by a small example with subsets of a set represented as propositional functions. First we give a name for the type of predicates over a type A:

$$pred(A : \mathbf{Type}) = (x : A) \to \mathbf{Set} : \mathbf{Type}.$$

The predicates over A are objects in the function space from A to **Set**. This function space does not form a set in predicative type theory (it has the type **Type**). In the same way, given a type A, we form the type for relations on A:

$$rel(A : \mathbf{Type}) = (x : A,\ y : A) \to \mathbf{Set} : \mathbf{Type}.$$

Now we represent subsets of a set A as predicates over A. We say that U is a subset of A if U is a propositional function ranging over A and that an element a of A is a member of U iff $U(a)$ holds. A propositional function U is then a subset of another propositional function V provided that Ux implies Vx for all x of type A:

$$subset(A : \mathbf{Set}) = \lambda U\ V \to (x : A,\ h : Ux) \to Vx : rel\ (pred\ A).$$

Note that in the type we can see that, given a set A, *subset* A is a relation on predicates of A. Also note that, in the last definition, A must be a set, since by the definition of *rel*, $(x : A,\ h : Ux) \to Vx$ has to be a set. The system checks this for us.

3 Preliminary Definitions

From now on, Half-code in `typewriter font` is mixed with comments, motivations and less formal definitions and proofs.

First some definitions about relations, predicates and operations:

```
rel(A:Type) = (x:A,y:A) -> Set : Type,
pred(A:Type) = (x:A) -> Set : Type,
bin(A:Type) = (x:A,y:A) -> A : Type,
op(A:Type) = (x:A) -> A : Type,
ref(A:Set,R:rel A) = (x:A) -> R x x : Set,
sym(A:Set,R:rel A) = (x:A,y:A,p:R x y) -> R y x : Set,
trans(A:Set,R:rel A) = (x:A,y:A,z:A,p:R x y,q:R y z) -> R x z : Set,
```

Note that `rel`, `pred`, `bin` and `op` also are applicable to elements of type `Set`.

The propositions *false* and *true* are defined as the empty set and a singleton set, respectively:

```
n0 = data{} : Set,
n1 = data{$True} : Set,
```

(In Half-code the constructors start with `$`). The connectives are defined by

```
not (A:Set) = (x:A) -> n0 : Set,
and (A:Set,B:Set) = sig{fst:A,snd:B} : Set,
or (A:Set,B:Set) = data{$Inl (x:A),$Inr (y:B)} : Set,
```

The definition of `and` should be compared to the equivalent definition using a *pair-constructor*,

```
and2(A:Set,B:Set) = data{$pair(a:A,b:B)} : Set.
```

However, a case-expression is required in order to analyse an object of type `and2 A B`, whereas for `and`, the proofs of `A` and `B` are obtained directly using the names `fst` and `snd`, respectively.

The existential quantifier is here defined using a *Σ-set* and, as was the case with `and`, the elimination rules are first and second projection:

```
exists(A:Set,B:pred A) = sig{fst:A,snd:B fst} : Set,
```

A set A is dense with respect to a relation R, if for all related x and y in A, there exists a z in A such that $R\ x\ z$ and $R\ z\ y$. (If `e` is an expression then `\x -> e` is the notation for a lambda abstraction.)

```
dense(A:Set,R:rel A) =
  (x:A,y:A,h:R x y) -> exists A (\z -> and (R x z) (R z y)) : Set,
```

A set A is decidable if $A \vee \neg A$ holds, and a relation R on A is decidable, if for all x and y, $R\ x\ y$ is decidable:

```
dec(A:Set) = or A (not A) : Set,
dec_rel(A:Set,R:rel A) = (x:A,y:A) -> dec (R x y) : Set,
```

A setoid is a set with an equivalence relation:

```
setoid(A:Set,R:rel A) =
  sig{isref:ref A R,issym:sym A R,istrans:trans A R} : Set,
```

A monoid is a setoid with a binary operation satisfying congruence, commutativity and associativity:

```
monoid(A:Set,eq:rel A,add:bin A) =
  sig{issetoid:setoid A eq,
      iscong:(x:A,y:A,z:A,t:A,h1:eq x z,h2:eq y t) ->
               eq (add x y) (add z t),
      iscom:(x:A,y:A) -> eq (add x y) (add y x),
      isassoc:(x:A,y:A,z:A) -> eq (add x (add y z)) (add (add x y) z)}
 : Set,
```

In the definition of formal space (section 4), propositional functions are used as subsets. Below we define what it means for a propositional function to be a subset of another propositional function. In general predicates do not respect equivalence relations, therefore the second (weaker) definition, that takes the equality relation as parameter, is used at some places. To justify the second subset relation, consider the following example: let $=$ be an equality defined on a set containing the elements x and y, and let U and V be predicates over that set; then using the first definition we do not in general have $x = y \mathbin{\&} Ux \mathbin{\&} U \subseteq V \Rightarrow Vy$. Moreover, in a formal topology the cover relation respects the equality relation. So, the second definition below is just as strong as it needs to be.

```
Subset(A:Set) = \U V -> (x:A,h:U x) -> V x : rel (pred A),

Subset2(A:Set,eq:rel A) =
  \U V -> (x:A,h:U x) -> exists A (\y -> and (eq x y) (V y))
  : rel (pred A),
```

Binary union for subsets as predicates is defined using disjunction:

```
Union(A:Set) = \U V -> \x -> or (U x) (V x) : bin (pred A),
```

We now define finite lists and use the concept of parametrised theory to collect some definitions and lemmas for lists.

```
list(A:Set) = data{$Nil,$Cons (x:A,xs:list A)} : Set,

theory_fin_list(A:Set,R:rel A) = theory{
  mem_list(x:A) = \xs -> case xs of {
                          $Nil -> n0,
                          $Cons y xs1 -> or (R x y) (mem_list x xs1)}
                : pred (list A),
   ...
```

(Just observe here that several definitions and lemmas are left out. We take the liberty of freely writing text inside theories like this.)

```
} : Theory {- end of theory_fin_list -}
```

An easy way to handle finite subsets is to use lists. But since lists of a type A and predicates over A have different types, a method for converting lists into predicates is needed when mixing the two notions. To transform a list into a predicate we simply abstract a variable belonging to the list (see `finset` below). The meaning of `finsubset l U` is "`l` is a finite subset of `U`". Finally, `findpart` takes a list `X` and a proof that `X` is a subset of a union, and finds the sublist `Y` of `X` belonging to the first subset in the union.

```
theory_subsets(A:Set,eq:rel A,issetoid:setoid A eq) = theory {
  th_fin_list = theory_fin_list A eq : Theory,

  finset(l:list A) = \x -> th_fin_list.mem_list x l : pred A,

  finsubset(l:list A,U:pred A) =
    case l of {
     $Nil -> n1,
     $Cons x xs -> and (U x) (finsubset xs U)} : Set,

  subset2 = Subset2 A eq:rel (pred A),

  findpart(X:list A,U:pred A,V:pred A,h:finsubset X (union U V)) = ...
    :exists
     (list A)
     (\Y->and (finsubset Y U) (subset2 (finset X) (union (finset Y) V)))
   ...
} : Theory,
```

We conclude this section by giving a type for intervals and a theory for intervals. Given a set, an interval is simply the pair of its endpoints:

```
interval(A:Set) = sig{lp:A,rp:A} : Set,
```

Given a set `A` and a relation `R` on `A`, we define the corresponding relation `S`, for intervals of `A`. This is used in the definition of the continuum (section 6), where $=$, $<$ and $\leq$ for rational intervals are defined from the corresponding relations on the rational numbers.

```
theory_interval(A:Set,R:rel A) = theory{
  B = interval A : Set,
  S = \I J -> and (R J.lp I.lp) (R I.rp J.rp) : rel B,
   ...
} : Theory
```

4 Formal Spaces

We recall the definition of *formal topology* given by Giovanni Sambin [S]. A formal topology over a set A is a structure

$$\langle A, =, \cdot, \lhd \rangle$$

where $\langle A, =, \cdot \rangle$ is a commutative monoid, $\lhd$ is a relation, called *cover*, between elements and subsets of A such that, for any $x, y \in A$ and $U, V \subseteq A$, the following conditions hold:

$$\text{(substitutivity)} \quad \frac{x = y \qquad y \lhd U}{x \lhd U}$$

$$\text{(reflexivity)} \quad \frac{x \in U}{x \lhd U}$$

$$\text{(transitivity)} \quad \frac{x \lhd U \qquad U \lhd V}{x \lhd V} \quad \text{where} \quad U \lhd V \equiv (\forall u \in U)(u \lhd V)$$

$$\text{(dot - left)} \quad \frac{x \lhd U}{x \cdot y \lhd U}$$

$$\text{(dot - right)} \quad \frac{x \lhd U \qquad x \lhd V}{x \lhd U \cdot V} \quad \text{where} \quad U \cdot V \equiv \{u \cdot v \mid u \in U, v \in V\}.$$

Subsets of the base A are represented by propositional functions ranging over A (see the previous sections).

We point out that, in contrast to the definition of formal topology given in [S], we do not require the base monoid to have a unit, nor do we have the positivity predicate used in [S]. The equality relation on the base set is also explicit here.

The formal topologies are here defined as a *Σ-type*: The set A with the relation $=$, the binary operation $\cdot$ and the relation $\lhd$ form a formal space, if A, $=$, $\cdot$ form a monoid and the rules of a formal topology (substitutivity, reflexivity, transitivity, dot-left, dot-right) are satisfied. In the implementation `eq`, `dot` and `cov` are used for $=$, $\cdot$ and $\lhd$, respectively. `DOT` and `COV` are used for the generalisations of $\cdot$ and $\lhd$, respectively, to subsets. Since `DOT` and `COV` are used both in the definition of the formal space and in the theory for formal spaces, they are defined globally; and, since $\cdot$ and $\lhd$ for subsets, depend on $\cdot$ and $\lhd$ for elements, `DOT` and `COV` have `A`, `eq`, `dot` and `cov` as parameters.

```
COV(A:Set,cov:(x:A,U:pred A)->Set)=
  \U V -> (x:A,p:U x) -> cov x V:rel (pred A),
DOT(A:Set,eq:rel A,dot:bin A)=
  \U V z -> sig{x:A,y:A,px:U x,py:V y,iseq:eq z (dot x y)}
  :bin (pred A),
```

If x and y are elements in U and V, respectively, then, immediate by the definition of $\cdot$ for subsets, $x \cdot y$ is an element in $U \cdot V$.

```
lemDOT(A:Set,eq:rel A,isref:ref A eq,dot:bin A,
       x:A,y:A,U:pred A,V:pred A,p:U x,q:V y)=
  struct{x=x,y=y,px=p,py=q,iseq=isref (dot x y)}
  :DOT A eq dot U V (dot x y),

space(A:Set,eq:rel A,dot:bin A,cov:(x:A,U:pred A)->Set)=
  sig{ismonoid:monoid A eq dot,
```

```
      ax0:(x:A,y:A,h2:eq x y,U:pred A,h3:cov y U)->cov x U,
      ax1:(x:A,U:pred A,h2:U x)->cov x U,
      ax2:(x:A,U:pred A,V:pred A,h2:cov x U,h3:COV A cov U V)->cov x V,
      ax3:(x:A,y:A,U:pred A,h2:cov x U)->cov (dot x y) U,
      ax4:(x:A,U:pred A,V:pred A,h2:cov x U,h3:cov x V)->
            cov x (DOT A eq dot U V)} : Type,
```

In the theory below some general facts of formal spaces are proved and some definitions are given. Later on we will define concrete formal spaces as instances of this theory.

```
 theory_space (A:Set,eq:rel A,dot:bin A,cov:(x:A,U:pred A)->Set,
               s:space A eq dot cov) =
 theory{
   union=Union A:bin (pred A),
   subset2=Subset2 A eq:rel (pred A),
   Cov=COV A cov:rel (pred A),
   Dot=DOT A eq dot:bin (pred A),

   lemDot=lemDOT A eq isref dot
         :(x:A,y:A,U:pred A,V:pred A,p:U x,q:V y)->Dot U V (dot x y),
```

The following lemmas say that $U \subseteq V \Rightarrow U \lhd V$ and $(U \cup V) \cdot (W \cup V) \lhd (U \cdot W) \cup V$, respectively.

```
   lem12(U:pred A,V:pred A)=
     \h->\x p->let {h1=h x p:exists A (\y ->and (eq x y) (V y))}
               in s.ax0 x h1.fst V h1.snd.fst (s.ax1 h1.fst V h1.snd.snd)
     :(h:subset2 U V)->Cov U V,

   lem7(U:pred A,V:pred A,W:pred A)=
     \x p->s.ax0
           x
           (dot p.x p.y)
           (union (Dot U W) V)
           p.iseq
           (case p.px of {
             $Inl x1->
               case p.py of {
                $Inl x2->
                  s.ax1 (dot p.x p.y) (union (Dot U W) V)
                     ($Inl (lemDot p.x p.y U W x1 x2)),
                $Inr y->
                  ax31 p.x p.y (union (Dot U W) V)
                     (s.ax1 p.y (union (Dot U W) V) ($Inr y))},
             $Inr y->
               s.ax3 p.x p.y (union (Dot U W) V)
                  (s.ax1 p.x (union (Dot U W) V) ($Inr y))})
     :Cov (Dot (union U V) (union W V)) (union (Dot U W) V),
```

Given a formal space (note that we still are inside the theory `theory_space`), we now define the space induced by a subset. In our implementation that is achieved by a nested theory, in which the induced cover is defined and the cover-rules are proved.

```
theory_indspace(V:pred A) = theory{
```

Let $\lhd$ be a cover. The cover induced by the subset V is defined by $a \lhd_V U \equiv a \lhd U \cup V$:

```
  covind=\x U->cov x (union U V):(x:A,U:pred A)->Set,
```

In order to prove that the space induced by V really is a formal space, an object of `space A eq dot covind` is constructed:

```
  indspace=
    struct{
     ismonoid=s.ismonoid,
     ax0=\x y h2 U h3->s.ax0 x y h2 (union U V) h3,
     ax1=\x U h2->s.ax1 x (union U V) ($Inl h2),
     ax2=\x U V1 h2 h3->
           s.ax2
           x
           (union U V)
           (union V1 V)
           h2
           (\x1 p->case p of {
                    $Inl x2->h3 x1 x2,
                    $Inr y->s.ax1 x1 (union V1 V) ($Inr y)}),
     ax3=\x y U h2->s.ax3 x y (union U V) h2,
     ax4=\x U V1 h2 h3->
           s.ax2
           x
           (Dot (union U V) (union V1 V))
           (union (Dot U V1) V)
           (s.ax4 x (union U V) (union V1 V) h2 h3)
           (lem7 U V V1)}
    :space A eq dot covind
}:Theory,  {- end of theory_indspace -}
```

In order to define compactness and Stone spaces, a notion of finite subset is needed. For that purpose finite lists are used. Given a list, the function `finset` returns the corresponding subset, and the meaning of `finsubset l U` is "`l` is a finite subset of `U`".

```
th_subs=theory_subsets A eq issetoid:Theory,
finset=th_subs.finset:(l:list A)->pred A,
finsubset=th_subs.finsubset:(l:list A,U:pred A)->Set,
```

The following predicate says that, given a subset `U` and predicate `P` for subsets, there exists a finite subset of `U` for which `P` holds.

```
existsFin(U:pred A,P:pred (pred A))=
  exists (list A) (\l->and (finsubset l U) (P (finset l))):Set,
```

`isCover U` is an abbreviation for "`U` covers the whole space":

```
    isCover=\U->(x:A)->cov x U:pred (pred A),
```

Now compactness, saying that if a subset U covers the whole space then there exists a finite subset of U that covers the whole space, and Stone cover (see [S]), saying that if the element x is covered by U then there exists a finite subset of U that covers x, are easily defined:

```
    compact=(U:pred A)->(h:isCover U)->existsFin U isCover:Type,
    stone=(x:A,U:pred A,h:cov x U)->existsFin U (\U0->cov x U0):Type
  }:Theory,  {- end of theory_space -}
```

We conclude this section by "covering" the definitions above with one more level of abstraction.

```
  SPACE = sig{A:Set,eq:rel A,dot:bin A,cov:(x:A,U:pred A)->Set,
              is_a_space:space A eq dot cov} : Type,
```

Given a space `s`, `Token s` returns the base of `s`

```
  Token(s:SPACE) = s.A : Set,
```

and `Open s` returns the subsets of the space `s`.

```
  Open(s:SPACE) = pred (Token s) : Type,
```

Given a space `s` and a subset `U` in `s`, `indSpace U s` forms the space where the cover in `s` is induced by `U`.

```
  indSpace(s:SPACE,U:Open s) =
    let {th1=theory_space s.A s.eq s.dot s.cov s.is_a_space:Theory,
         th2=th1.theory_indspace U:Theory}
    in struct{
        A=s.A,
        eq=s.eq,
        dot=s.dot,
        cov=th2.covind,
        is_a_space=th2.indspace}:SPACE,
```

`CompactSPACE` is a predicate of over all spaces (`SPACE`), saying that the space is compact.

```
  CompactSPACE(s:SPACE)=
    let {th=theory_space s.A s.eq s.dot s.cov s.is_a_space:Theory}
    in th.compact:Type,
```

Using a Σ-type, a compact space is a space which is compact.

```
  COMPACTSPACE=sig{s:SPACE,iscompact:CompactSPACE s}:Type,
```

We also define the corresponding for Stone spaces:

```
  StoneSPACE(s:SPACE)=
    let {th=theory_space s.A s.eq s.dot s.cov s.is_a_space:Theory}
    in th.stone:Type,

  STONESPACE=sig{s:SPACE,isstone:StoneSPACE s}:Type
```

5 Linear Ordering

The rational numbers are formed abstractly as an unbounded, dense, decidable linear ordering. Following von Plato [vP], we start with the order relation $<$ satisfying the axioms $\neg(x < y \ \& \ y < x)$ and $x < y \Rightarrow (x < z \vee z < y)$.

```
islinear(A:Set,lt:rel A) =
 sig{LO1:(x:A,y:A,p:lt x y,q:lt y x) -> n0,
     LO2:(x:A,y:A,z:A,p:lt x y) -> or (lt x z) (lt z y)}
 : Set,
```

Less-then-or-equal (or rather *not-greater-than*) is defined as $x \leq y \equiv \neg(y < x)$. The equality $x = y \equiv x \leq y \ \& \ y \leq x$ then satisfying reflexivity, symmetry and transitivity.

```
leq = \x y -> not (lt y x) : rel A,
eq = \x y -> and (leq x y) (leq y x) : rel A,
```

To this ordering decidability $(x < y \vee y \leq x)$ is added:

```
isdeclinear(A:Set,lt:rel A)=sig{DLO1:islinear A lt,
                                DLO2:dec_rel A lt}:Set,
```

Then **max** and **min** can be defined by analysing the proof of $x < y \vee y \leq x$.

The rationals also form an unbounded $((\forall a)(\exists x)(x < a)$ and $(\forall a)(\exists x)(a < x))$ and dense $(x < y \Rightarrow (\exists z)(x < z < y))$ set.

```
isdenseunbdeclinear(A:Set,lt:rel A)=
  sig{isdeclin:isdeclinear A lt,
      nolb:(a:A)->exists A (\x->lt x a),
      noub:(a:A)->exists A (\x->lt a x),
      isdense:dense A lt}:Set,
```

Now we collect the definitions above in the following theory:

```
theoryUnboundedDenseDecidableLinear(
    A:Set,lt:rel A,isdudl:isdenseunbdeclinear A lt)=theory{
  leq=...:rel A,
  eq=...:rel A,
  min=...:bin A,
  max=...bin A,
   ...
} : Theory
```

6 The Continuum as a Formal Space

The *topology of formal reals* is the structure

$$\langle Q \times Q, =_{Q \times Q}, \cdot, \lhd \rangle \ ,$$

where Q is the set of rational numbers. The monoid operation is defined by $(p, q) \cdot (r, s) \equiv (max(p, r), min(q, s))$; the cover $\lhd$ is defined by

$$(p, q) \lhd U \equiv (\forall p', q')(p < p' \ \& \ q' < q \Rightarrow (p', q') \lhd_f U) \ ,$$

where the relation $\lhd_f$ is inductively defined by

1. $\dfrac{q \leq p}{(p,q) \lhd_f U}$

2. $\dfrac{(p,q) \in U}{(p,q) \lhd_f U}$

3. $\dfrac{r < s \quad (p,s) \lhd_f U \quad (r,q) \lhd_f U}{(p,q) \lhd_f U}$

4. $\dfrac{(p',q') \lhd_f U \quad p' \leq p \quad q \leq q'}{(p,q) \lhd_f U}$.

For properties of this formal space we refer to [CN, NS].

Starting from the linear ordering of the previous section, the continuum is here to be defined as a formal space. In the following theory, $\cdot$, $\lhd_f$ and $\lhd$ are defined (having the names `dot`, `covf` and `cov`, respectively). We also prove that $\lhd_f$ is a Stone cover and that $\lhd$ is a cover relation.

```
theory_continuum(Q:Set,ltQ:rel Q,
                 isdudl:isdenseunbdeclinear Q ltQ)=theory{
  th_dudl=theoryUnboundedDenseDecidableLinear Q ltQ isdudl
         :Theory,
  leqQ=th_dudl.leq:rel Q,
  max=th_dudl.max:bin Q,
  min=th_dudl.min:bin Q,
  eqQ=th_dudl.eq:rel Q,
  eqQsym=...:sym Q eqQ,
```

The base consists of the rational intervals:

```
  QxQ=interval Q:Set,
  int(p:Q,q:Q)=struct{lp=p,rp=q}:QxQ,
```

By instantiating the theory `theory_interval` with `Q` and a relation on `Q`, the corresponding relation on intervals is obtained.

```
  th_int_eqQ=theory_interval Q eqQ:Theory,
  eqQxQ=th_int_eqQ.S:rel QxQ,
  th_int_ltQ=theory_interval Q ltQ:Theory,
  ltQxQ=th_int_ltQ.S:rel QxQ,
  th_int_leqQ=theory_interval Q leqQ:Theory,
  leqQxQ=th_int_leqQ.S:rel QxQ,
  eqQxQref=...:ref QxQ eqQxQ,
```

The dot-operation is defined as intersection:

```
  dot=\x y -> int (max x.lp y.lp) (min x.rp y.rp):bin QxQ,
```

The formalised version of $\lhd_f$ is recursively defined by the following:

```
  covf(I:QxQ,U:pred QxQ)=
    data{$C1(h:leqQ I.rp I.lp),
         $C2(h:U I),
         $C3(J:QxQ,h1:ltQ J.lp J.rp,
```

```
             h2:covf (int I.lp J.rp) U,
             h3:covf (int J.lp I.rp) U),
         $C4(J:QxQ,h1:leqQxQ I J,h2:covf J U)}:Set,
```

`(QxQ,eqQxQ,dot)` forms a commutative monoid:

```
ismonoid=struct{
          issetoid=struct{
                    isref=...,
                    issym=...,
                    istrans=...},
          iscong=...,
          iscom=...,
          isassoc=...}:monoid QxQ eqQxQ dot,
```

`(QxQ,eqQxQ,dot,covf)` is a formal space:

```
Rf=struct{
    ismonoid=ismonoid,
    ax0=...,
    ax1=\x U h2->$C2 h2,
    ax2=...,
    ax3=...,
    ax4=...}:space QxQ eqQxQ dot covf,
th_Rf=theory_space QxQ eqQxQ dot covf Rf:Theory,
```

The formalised version of $\lhd$ is explicitly defined by the following:

```
cov(I:QxQ,U:pred QxQ)=(J:QxQ,h:ltQxQ J I)->covf J U:Set,
```

`(QxQ,eqQxQ,dot,cov)` forms a formal space:

```
R=struct{
   ismonoid=ismonoid,
   ax0=...,
   ax1=...,
   ax2=...,
   ax3=...,
   ax4=...}
  :space QxQ eqQxQ dot cov,
th_R=theory_space QxQ eqQxQ dot cov R:Theory,
```

```
th_subs=theory_subsets QxQ eqQxQ (ismonoid.issetoid):Theory,
finset=th_subs.finset:(l:list QxQ) -> pred QxQ,
finsubset=th_subs.finsubset:(l:list QxQ,U:pred QxQ)->Set,
```

`covf` is a Stone cover:

```
covfSc(I:QxQ,U:pred QxQ)=...
  :(h:covf I U)->
     exists
     (list QxQ)
     (\U0->and (finsubset U0 U) (covf I (finset U0))),

isstone=covfSc:th_Rf.stone,
```

and (QxQ,eqQxQ,dot,covf) form a Stone space:

```
    isStoneSpace=struct{
                  s=struct{
                     A=QxQ,
                     eq=eqQxQ,
                     dot=dot,
                     cov=covf,
                     is_a_space=Rf},
                  isstone=covfSc}:STONESPACE
  } :Theory  {- end of theory_continuum -}
```

7 The Heine-Borel Covering Theorem

We now define the closed rational interval $[a,b]$ as a formal space and prove the Heine-Borel covering theorem, saying that if U is a subset that covers $[a,b]$ then there exists a finite subset of U that covers $[a,b]$.

Let $\mathcal{R} \equiv \langle Q \times Q, =_{Q\times Q}, \cdot, \lhd_{\mathcal{R}} \rangle$ be the formal topology of formal reals and let

$$[a,b] \equiv \langle Q \times Q, =_{Q\times Q}, \cdot, \lhd_{[a,b]} \rangle$$

where the relation $\lhd_{[a,b]}$ is defined by

$$(p,q) \lhd_{[a,b]} U \equiv (p,q) \lhd_{\mathcal{R}} U \cup \{(r,a) \mid r \in Q\} \cup \{(b,s) \mid s \in Q\}.$$

Intuitively, the interval (p,q) is covered by U in the space $[a,b]$, if (p,q) intersected with *the closed interval* $[a,b]$ is covered by U in the space $\mathcal{R}$. In the sequel we will use the notation $\mathcal{C}[a,b]$ for $\{(r,a) \mid r \in Q\} \cup \{(b,s) \mid s \in Q\}$ and we understand $\mathcal{C}[a,b]$ as the complement of $[a,b]$.

```
  theory_heineborel(Q:Set,ltQ:rel Q,
                    isdudl:isdenseunbdeclinear Q ltQ,a:Q,b:Q)=
  theory{
    th_c=theory_continuum Q ltQ isdudl:Theory,
    th_R=th_c.th_R:Theory,
    th_Rf=th_c.th_Rf:Theory,
    th_subs=th_c.th_subs:Theory,

    eqQ=th_c.eqQ:rel Q,

    QxQ=th_c.QxQ:Set,
    eqQxQ=th_c.eqQxQ:rel QxQ,
    int=th_c.int:(p:Q,q:Q)->QxQ,
    dot=th_c.dot:bin QxQ,
    covR=th_c.cov:(I:QxQ,U:pred QxQ)->Set,
    covRf=th_c.covf:(I:QxQ,U:pred QxQ)->Set,

    Rf=th_c.Rf:space QxQ eqQxQ dot (th_c.covf),
```

```
union=th_R.union:bin (pred QxQ),
finset=th_subs.finset:(l:list QxQ)->pred QxQ,

ltQxQ=th_c.ltQxQ:rel QxQ,
```

The cover on $[a,b]$ is defined in the following way: I is covered by U in $[a,b]$ if I is covered by the union of U and the the complement of $[a,b]$ in $\mathcal{R}$. The fact that $[a,b]$ really is a formal space is immediate, since the cover is an instance of a cover induced by a subset.

```
Cab=union (\x->(eqQ a x.rp)) (\x->(eqQ b x.lp)):pred QxQ,

covab(I:QxQ,U:pred QxQ)=covR I (union U Cab):Set,

th_ind=th_R.theory_indspace Cab:Theory,
ab=th_ind.indspace:space QxQ eqQxQ dot covab,
th_ab=theory_space QxQ eqQxQ dot covab ab:Theory,
```

`hblem1,2` below prove the equivalence

$$(\forall I)(I \lhd_{[a,b]} U) \Leftrightarrow (\exists r,s)(r < a \;\&\; b < s \;\&\; (r,s) \lhd_{\mathcal{R}_f} U \cup \mathcal{C}[a,b]).$$

$\Rightarrow$: By the axiomatisation of the rational numbers, there exist p and q such that $p < a$ and $b < q$. So given $(\forall I)(I \lhd_{[a,b]} U) \equiv (\forall I)(I \lhd_{\mathcal{R}} U \cup \mathcal{C}[a,b])$, then in particular $(p,q) \lhd_{\mathcal{R}} U \cup \mathcal{C}[a,b]$. Again by the axioms, there exists r and s such that $p < r < a$ and $b < s < q$. Then by the definition of $\lhd_{\mathcal{R}}$, $(r,s) \lhd_{\mathcal{R}_f} U \cup \mathcal{C}[a,b]$.

```
hblem1(U: pred QxQ)=
  \h->let {p=isdudl.no_lb a:exists Q (\x->ltQ x a),
           q=isdudl.no_ub b:exists Q (\x->ltQ b x),
           r=isdudl.isdense p.fst a p.snd
            :exists Q (\x->and (ltQ p.fst x) (ltQ x a)),
           s=isdudl.isdense b q.fst q.snd
            :exists Q (\x->and (ltQ b x) (ltQ x q.fst))}
      in struct{
          fst=int r.fst s.fst,
          snd=struct{
               fst=struct{
                    fst=r.snd.snd,
                    snd=s.snd.fst},
               snd=h (int p.fst q.fst) (int r.fst s.fst)
                    (struct{
                      fst=r.snd.fst,
                      snd=s.snd.snd})}}
  :(h:(I:QxQ)->covab I U)->
     exists QxQ (\x->and (ltQxQ (int a b) x) (covRf x (union U Cab))),
```

$\Leftarrow$: It is enough to prove $(p,q) \lhd_{\mathcal{R}_f} U \cup \mathcal{C}[a,b]$, for arbitrary p and q. Since $r < a$ and $b < s$, $(p,q) \lhd_{\mathcal{R}_f} \{(p,a),(r,s),(b,q)\}$. Since $(p,a) \in \mathcal{C}[a,b]$, $(r,s) \lhd_{\mathcal{R}_f} U \cup \mathcal{C}[a,b]$ and $(b,q) \in \mathcal{C}[a,b]$, $\{(p,a),(r,s),(b,q)\} \lhd_{\mathcal{R}_f} U \cup \mathcal{C}[a,b]$. The claim now follows by transitivity.

```
hblem2(U:pred QxQ)=
  \h->\I->\J->\h1->
    let {rs=h.fst:QxQ,
         U1=finset
            ($Cons rs ($Cons (int J.lp a) ($Cons (int b J.rp) $Nil)))
           :pred QxQ,
         h1=$C3
            (int rs.lp a)
            h.snd.fst.fst
            (Rf.ax1 (int J.lp a) U1
               ($Inr ($Inl (th_c.eqQxQref (int J.lp a)))))
            ($C3
             (int b rs.rp)
             h.snd.fst.snd
             (Rf.ax1 (int h.fst.lp h.fst.rp) U1
                ($Inl (th_c.eqQxQref rs)))
             (Rf.ax1 (int b J.rp) U1
                ($Inr ($Inr ($Inl (th_c.eqQxQref (int b J.rp)))))))
           :th_c.covf J U1,
         h2=\J1->\p->
              case p of {
               $Inl x -> Rf.ax0 J1 rs (union U Cab) x h.snd.snd,
               $Inr y ->
                 case y of {
                  $Inl x ->Rf.ax1
                           J1
                           (union U Cab)
                           ($Inr ($Inl (th_c.eqQsym J1.rp a x.snd))),
                  $Inr y ->
                    case y of {
                     $Inl x -> Rf.ax1 J1 (union U Cab)
                                 ($Inr ($Inr x.fst)),
                     $Inr y -> case y of {}}}}
           :th_c.Covf U1 (union U Cab)}
    in (Rf.ax2 J U1 (union U Cab) h1 h2)
  :(h:exists QxQ (\rs->and
                     (ltQxQ (int a b) rs)
                     (covRf rs (union U Cab))))->
     (I:QxQ)->covab I U,
```

The Heine-Borel covering theorem: $[a,b]$ is compact, i.e.

$$(\forall I)(I \lhd_{[a,b]} U) \Rightarrow (\exists U_0 \subseteq_f U)(\forall I)(I \lhd_{[a,b]} U_0),$$

where $U_0 \subseteq_f U$ means that U_0 is a finite subset of U.

The proof goes as follows. Given $(\forall I)(I \lhd_{[a,b]} U)$, by **hblem1**, $(\exists r,s)(r < a \;\&\; b < s \;\&\; (r,s) \lhd_{\mathcal{R}_f} U \cup \mathcal{C}[a,b])$. Since $\lhd_{\mathcal{R}_f}$ is a Stone cover, there exists $W_0 \subseteq_f U \cup \mathcal{C}[a,b]$ such that $(r,s) \lhd_{\mathcal{R}_f} W_0$. By **findpart**, in **theory_subsets**, the part of W_0 belonging to U can be extracted, thus $(\exists U_0 \subseteq_f U)(\exists r,s)(r < a \;\&\; b < s \;\&\; (r,s) \lhd_{\mathcal{R}_f} U_0 \cup \mathcal{C}[a,b])$. The claim then follows by **hblem2**.

```
  subset2=Subset2 QxQ eqQxQ:rel (pred QxQ),

  heine_borel=\U h->
    let {rs=hblem1 U h
           :exists
            QxQ
            (\rs->and (ltQxQ (int a b) rs) (covRf rs (union U Cab))),
         W0=th_c.covfSc (rs.fst) (union U Cab) rs.snd.snd
           :exists (list QxQ) (\W0->and
                                 (th_subs.finsubset W0 (union U Cab))
                                 (covRf rs.fst (finset W0))),
         U0=th_subs.findpart W0.fst U Cab W0.snd.fst
           :exists
            (list QxQ)
            (\U0->and
                  (th_subs.finsubset U0 U)
                  (subset2 (finset W0.fst) (union (finset U0) Cab))),
         h1=Rf.ax2
            rs.fst
            (finset W0.fst)
            (union (finset U0.fst) Cab)
            W0.snd.snd
            (th_Rf.lem12
             (finset W0.fst)
             (union (finset U0.fst) Cab)
             U0.snd.snd)
           :covRf rs.fst (union (finset U0.fst) Cab)}
    in struct{
        fst=U0.fst,
        snd=struct{
             fst=U0.snd.fst,
             snd=hblem2 (finset U0.fst) (struct{
                                           fst=rs.fst,
                                           snd=struct{
                                                fst=rs.snd.fst,
                                                snd=h1}})}}
    :th_ab.compact
}:Theory  {- end of theory_heineborel -}
```

8 Acknowledgement

Parts of the formalisation in section 4 are due to Thierry Coquand. Sara Negri, Jan von Plato and Jan Smith have also contributed with useful remarks to this paper.

References

[Ba] H. Barendregt. *Lambda calculi with types*, In S. Abramsky, D.M. Gabbay and

T.S.E. Maibaum eds., "Handbook of Logic in Computer Science, Vol. 2", Oxford University Press, Oxford, 1992.

[Br] N.G. de Bruijn. *A plea for weaker frameworks*, In G. Huet and G. Plotkin eds., "Logical Frameworks", pp. 40-68, Cambridge University Press, Cambridge, 1991.

[CN] J. Cederquist, S. Negri. *A constructive proof of the Heine-Borel covering theorem for formal reals*, In S. Berardi and M. Coppo eds., "Types for Proofs and Programs", Lecture Notes in Computer Science 1158, pp. 62–75, Springer-Verlag, 1996.

[C] T. Coquand. *An algorithm for type-checking dependent types*, Science of Computer Programming 26, pp. 167-177, Elsevier, 1996.

[H] M. Hofmann. *A model of intensional Martin-Löf type theory in which unicity of identity proofs does not hold*, Technical report, Dept. of Computer Science, University of Edinburgh, 1993.

[M] L. Magnusson. "The Implementation of ALF - a Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution", Chalmers University of Technology and University of Göteborg, PhD Thesis, 1995.

[ML] P. Martin-Löf. *An Intuitionistic Theory of Types* (1972), To be published in the proceedings of Twenty-five years of Constructive Type Theory, G. Sambin and J. Smith eds., Oxford University Press.

[NS] S. Negri, D. Soravia. *The continuum as a formal space*, Archive for Mathematical Logic, to appear.

[OSR] S. Owre, N. Shankar, J. M. Rushby. *The PVS Specification Language (Beta Release)*, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA, 1993.

[vP] J. von Plato. *A memorandum on the constructive axioms of linear order*, Dept. of Philosophy, University of Helsinki, 1995.

[S] G. Sambin. *Intuitionistic formal spaces – a first communication*, In D. Skordev ed., "Mathematical logic and its applications", pp. 187-204, Plenum Press, 1987.