Math. Struct. in Comp. Science (2015), *vol.* 25, *pp.* 1278–1294.
doi:10.1017/S0960129514000577 First published online 20 January 2015

An experimental library of formalized Mathematics based on the univalent foundations

VLADIMIR VOEVODSKY†

School of Mathematics, Institute for Advanced Study, Princeton, New Jersey, U.S.A.
Email: vladimir@ias.edu

Received 30 December 2013; revised 27 May 2014

1. Introduction

This is a short overview of an experimental library of Mathematics formalized in the Coq proof assistant using the univalent interpretation of the underlying type theory of Coq. I started to work on this library in February 2010 in order to gain experience with formalization of Mathematics in a constructive type theory based on the intuition gained from the univalent models (see Kapulkin *et al.* 2012).

Univalent models interpret types not as sets but as homotopy types. Their use in formalization of general Mathematics (as opposed to just homotopy theory) is based on the following consideration. First note that we can stratify mathematical constructions by their "level.' There is element-level Mathematics – the study of element-level objects such as numbers, polynomials or various series. Then one has set level Mathematics – the study of sets with structures such as groups, rings etc., which are invariant under isomorphisms. The next level is traditionally called category-level, but this is misleading. A collection of set-level objects naturally forms a groupoid since only isomorphisms are intrinsic to the objects one considers, while more general morphisms can often be defined in a variety of ways. Thus the next level after the set level is the groupoid-level – the study of properties of groupoids with structures which are invariant under the equivalences of groupoids. From this perspective a category is an example of a groupoid with structure which is rather similar to a partial ordering on a set.

Extending this stratification we may further consider 2-groupoids with structures, n-groupoids with structures and ∞-groupoids with structures. Thus a proper language for formalization of Mathematics should allow one to directly build and study groupoids of various levels and structures on them.

A major advantage of this point of view is that unlike ∞-categories, which can be defined in many substantially different ways the world of ∞-groupoids is determined by Grothendieck correspondence (see Grothendieck 1997) , which asserts that ∞-groupoids are 'the same' as homotopy types. Combining this correspondence with the previous considerations we come to the view that not only homotopy theory but *the whole of Mathematics* is the study of structures on homotopy types.

† Work on this paper was supported by NSF grant 1100938.

The univalent models of constructive type theories enable one to use such type theories to reason directly about homotopy types with structures. This is the main idea of the univalent foundations of Mathematics – to use constructive type theory together with the intuition coming from its univalent homotopy-theoretic semantics to write and to prove theorems about mathematical objects of all 'levels' formally.

Univalent Foundations can be seen as a realization of the vision of Michael Makkai whose paper Makkai (1995) was very important for me in my search for a formal language for contemporary Mathematics.

At the moment, there are two actively supported proof assistants based on constructive type theories – Coq and Agda. Both proof assistants continue to be developed by teams which consist mainly of computer scientists who are actively experimenting with new features which are introduced into the systems without a formal verification of their consistency. Of these two systems Coq has been in development longer and is more conservative. To further minimize the possibility of accidentally using a feature which may later be found to be inconsistent the library described here was written using a restricted subset of the type theory underlying Coq. For another approach in Coq, we suggest the reader to look at the HoTT project library at `https://github.com/HoTT/HoTT`. The version of the library which this text refers to was checked to compile with a patched version of `Coq 8.4pl3`. For instructions on how to get this version of Coq and how to patch it, see the file `Coq_patch/README`.

The type theory of Coq is, roughly speaking, a combination of three components. The first component is a version of the Thierry Coquand's calculus of constructions (CC) (see Coquand and Huet 1988). This is a type system with two universes, Prop and Type, dependent products and abstraction/application constructions satisfying β-reduction. The second component is a universe management system which replaces two universes of CC with an infinite hierarchy of universes which is due to Z. Luo (see Luo 1994). The third component is a machinery for creating strictly positive 'inductive types' described in Paulin-Mohring (1993).

In our library, we use a small subset of a modified version of the Coq type system. The modifications are introduced through a patch contained in the subdirectory Coq_patch. Some information on the content of this patch and on its history can be found in the README file of that subdirectory.

The main modification turns off the universe consistency verification system of Coq. This, of course, makes the type system inconsistent (any type, including the empty type, can be shown to have an object). The proper solution is instead to use universe polymorphism together with either resizing rules (see Voevodsky 2011) or higher inductive types (see Univalent Foundations Project 2013). However, these modifications are highly non-trivial and for the experimental purposes of the current library it seemed reasonable to rely on careful tracing of universe levels 'by hand.' This issue becomes important only starting with the file `hProp.v`. The first major file of the library, `uu0.v`, can be compiled without the patch.

The main restriction which we impose on the constructions of the library concerns the use of the inductive types machinery of Coq. In a rather ingenious way this machinery is normally used in Coq both to define standard ingredients of constructive type theories

https://doi.org/10.1017/S0960129514000577 Published online by Cambridge University Press

such as identity types, dependent sums, the one point type, disjoint unions, the empty type, Booleans and natural numbers and also to define a multitude of other constructs such as, for example, inequalities between natural numbers. In the current library we use this machinery only to introduce the standard constructions listed above. No further use of inductive types is made except in one place in the file `hnat.v` where we show that our approach to comparisons between natural numbers is equivalent to the approach taken in the standard library of Coq.

Another restriction is that we do not use the universe Prop. Associated with this universe there is a 'singleton elimination' rule which is inconsistent with the univalent model. To avoid accidental use of this rule by tactics the patch file modifies the way the universe level of inductive constructions (most notably of identity types) is computed. During the compilation of the first file of the library, `uuu.v`, the compiler should display '`paths 0 0:UUU.`' Without proper application of the patch the compiler would display '`paths 0 0:Prop.`'

The distribution of Coq includes an extensive 'standard library.' Our library uses only the first and most basic subdivision of the Coq's standard library, namely Coq.Init. In fact some of the files of the standard library may take very long to compile with the '-no-sharing' option which is introduced by the patch and which we use to overcome a bug in Coq's normalization algorithm. See the file `Coq_patch/README` for instructions of how to compile Coq without compiling most of the standard library.

2. File `uuu.v`

The first several lines of `uuu.v` introduce new notations for some of the constructions that are defined in Coq's standard library. The part of the library where these constructions are introduced is located in '`coqlocation/theories/Init/`' where '`coqlocation`' is the directory where the Coq distribution is. Files of this part of the library are automatically loaded by Coq while to load other parts of the standard library (located in other subdirectories of '`coqlocation/theories/`') requires an explicit instruction.

In the first new definition in '`uuu.v`' we introduce the version of *dependent sum* used in our library. It is called '`total2`' due on the one hand to its semantic meaning as the total space of a fibration and on the other to its function as a generic record of length 2.[†] Several important features of Coq formalization can be illustrated with this definition and the following definitions of '`pr1`' and '`pr2`.'

The first parameter of the construction, the type '`T`,' is shown in the definition in braces. This means that this is an *implicit* parameter, i.e., when '`total2`' is used one writes '`total2 P`' instead of '`total2 T P.`' Types of expressions are computable in Coq from expressions themselves and since the type of '`P`' must be '`T->Type`' the system can infer '`T`' from '`P`'.

The second parameter '`P`' is of the type '`T->Type.`' Here '`Type`' is a generic notation which Coq uses for universes. The universe management in Coq is rather baroque and well hidden from user control so for simplicity one may think that '`Type`' is synonymous

[†] In the first version of the library there was also '`total3`' corresponding to the generic record of length 3.

https://doi.org/10.1017/S0960129514000577 Published online by Cambridge University Press

with the name of some fixed universe 'UU.' A function 'T->UU' is intuitively a way to assign to any object of 'T' an object of 'UU,' i.e., a type which is contained in 'UU.' In other words, it is a family of types in 'UU' parametrized by 'T.' The semantics of this is as follows. If we use the univalent model with values in the category of simplicial sets then 'T' is mapped to a (Kan) simplicial set and 'UU' is mapped to the base of the universal Kan fibration which classifies Kan fibrations whose fibres belong to 'UU.' Thus 'P' corresponds to a Kan fibration over 'T' and 'total2 P' is the total space of this fibration.

In the informal semantics with values in ∞-groupoids 'T' is mapped to an ∞-groupoid while 'UU' is mapped to the ∞-groupoid of ∞-groupoids in 'UU' and their equivalences. The function 'P' then can be viewed as a functor and 'total2 P' is the ∞-groupoid of pairs (x, y) where x is an object of 'T' and y an object of $P(x)$.

The next definition is that of "pr1.' It takes three parameters and returns an object of 'T' where 'T' is the first parameter. In general, when one has a definition of some 'C' in Coq with parameters 'x1 x2... xn' one can write not only 'C a1... an' but also *partially applied* versions such as 'C a1... a(n-1)' or just 'C.' The type of such a partially applied definition will be a function type or more generally a *dependent product* type.

In the case of 'pr1' the first two parameters are implicit and supposed to be inferred from the third. If one wants to use a partially applied version of 'pr1' one has to provide the first two parameters explicitly. To tell Coq that a definition will be used as if all its parameters were explicit one uses prefix '@' and writes, for example, '@pr1 T P.' The type of this expression is the function type '(total2 T P) -> T' and its semantical meaning is the projection from the total space of a fibration to its base.

An extremely important feature of dependent type theories which is unavailable in the theories without dependent types and which at the first may seem confusing is that we also have '@pr2 T P.' Obviously not any fibration is trivial so we do not normally have a projection from the total space to a fibre *as a function*. However we always have it as a *dependent function*. By writing something like

```
Variable T:Type.
Variable P:T -> Type.
Check ( @pr2 T P ).
```

one will see that the type of '@pr2 T P' is 'forall tp:total2 P, P (pr1 tp).' The semantic meaning of the later expression is as follows. 'forall' is the name of the dependent product construction in Coq. Its general format is 'forall x:T1, T2' where 'T1' is a type expression and 'T2' is a type expression which may have a parameter 'x' of type 'T1.' Such an expression with a parameter semantically is the same as a function 'T1 -> UU,' i.e., the 'forall' construction has essentially the same parameters as 'total2' - a type and a family of types parametrized by this type. As was explained above such a pair corresponds in the univalent model in simplicial sets to a Kan fibration. The type 'forall x:T1, T2' is the (Kan) simplicial set of *sections* of this fibration.

If 'T2' does not actually depend on 'x' then one abbreviates the expression 'forall x:T1, T2' to 'T1 -> T2.' Semantically it corresponds to the case of a constant fibration whose sections are just functions from the base 'T1' to the fibre 'T2.'

https://doi.org/10.1017/S0960129514000577 Published online by Cambridge University Press

Returning to the case of '`@pr2 T P`' we see that semantically it is a section of the fibration over '`@total2 T P`' whose fibre over '`tp`' is the fibre of '`P`' over '`pr1 tp.`' In mathematical notation, if our fibration is $p : E \to B$ then '`@pr2 T P`' is the diagonal section of $E \times_B E$ over E.

3. File `uu0.v`

This file contains the results of the library which are applicable to *all* types.

The first three lines of the file are also repeated with some obvious changes in all the rest of the files of the library. These are commands to the Coq program.

The first one tells Coq not to do a certain type of steps automatically at the start of every proof but to leave the choice of whether or not to do these steps to the user.

The second and the third lines address the mechanism which loads other library files. They are discussed in more detail in the appendix.

Let me now use some of the first proofs given in '`uu0.v`' to illustrate how the proof system of Coq works. Note first that a line such as

'`Definition name1:expr1.`'

tells Coq that a constant called '`name1`' of type '`expr1`' will be provided by the user. In the case of '`Definition`' there are two ways to provide the value of this constant. One can write

'`Definition name1:expr1:= expr2.`'

in that case '`expr2`' should be an expression which has type '`expr1`' which will be the value of the constant '`name1.`' Alternatively, one can write '`Proof.`' after '`Definition name1:expr1.`' and then use various commands of Coq proof mode to construct the value of the constant. When Coq says '`Proof completed`' in the '`response`' window one writes either '`Qed.`' or '`Defined.`' The difference between the two is that when '`Qed.`' is used the actual structure of the constructed expression becomes hidden (opaque) while when '`Defined.`' is used the structure remains accessible.

The keywords '`Theorem,`' '`Lemma`' and '`Proposition`' are strictly equivalent and are equivalent to '`Definition`' except that one must use the proof mode to provide the value of the corresponding constant, i.e., one cannot simply provide the value after '`:=.`'

More generally Coq can be told that a constant with the name '`name1`' is going to be introduced by a line of the form

'`Definition name1 ( x1:texpr1 )... ( xn:texprn ):expr.`'

which is essentially equivalent to

'`Definition name1:forall x1:texpr1,..., forall xn:texprn, expr`'

with the only difference being that the first form allows one to say that some of the parameters will be implicit by using curly brackets.

The first proof of the library is that of '`Definition fromempty.`' The sentence which starts with the word '`Definition`' tells Coq that a constant with the name '`fromempty`' of type '`forall X:UU, empty -> X`' will be provided and that the type parameter '`X`' is implicit. The value for this constant is constructed inside the proof mode through the use of two tactics '`intros`' and '`destruct.`' We will not discuss here how the tactics language of Coq is working referring the reader instead to Coq reference manual.

https://doi.org/10.1017/S0960129514000577 Published online by Cambridge University Press

Detailed information about the mathematical content of the file `uu0.v` can be obtained from the comments in this file. We will only discuss here a few fundamental constructions the meaning of which might not be immediately obvious.

The first such construction is '`iscontr T`' where '`T`' is a type. It introduces the concept from which almost everything else is build – the concept of a contractible type. By definition, a proof of contractibility of a type '`T`' is an object of the type '`iscontr T`'. There are two ways to argue that this is a '`correct`' way to define contractibility. The first one is to point out that the more complex homotopy-theoretic notions defined with the use of this notion of contractibility are proved further in this file to satisfy a large number of expected properties.

Another is to analyse the univalent semantics of this construction. Consider for example a univalent model with values in Kan simplicial sets. Then '`T`' is a simplicial set. A point of '`iscontr T`' is a pair '`(cntr, s)`' where '`cntr`' is a point of '`T`' and '`s`' is an object of '`forall t:T, paths cntr t`.' The family of types '`t` $\mapsto$ `paths cntr t`' is the paths bundle corresponding to the point '`cntr`' and as explained above '`s`' is a section of this bundle. But the total space of the paths bundle is contractible and if it has a section then '`T`' is a retract of a contractible simplicial set and therefore it is contractible. In the opposite direction if '`T`' is contractible then it is in particular non-empty and we can choose a point '`cntr`' in '`T`.' Any fibration over a contractible s.s. is trivial and if it has a non-empty fibre it has a section. The fibre of the paths fibration defined by '`cntr`' over '`cntr`' is non-empty and therefore it has a section '`s`' which gives us a point in '`iscontr T`.'

The next fundamental definition is the property '`isweq`' of a function '`f`' to be a (weak) equivalence which is defined as the condition that all (homotopy) fibres of '`f`' are contractible. Along with '`isweq f`' we introduce '`weq X Y`' – the type of (weak) equivalences from '`X`' to '`Y`,' i.e., of pairs '`(f, is)`' where '`f:X -> Y`' and '`is:isweq f`.'

Theorem '`gradth`' shows that for a homotopy equivalence, i.e., a quadruple '`f:X -> Y`,' '`g:Y -> X`,' '`egf`,' '`efg`' where '`egf`' is a homotopy from '`funcomp f g`' to the identity of '`X`' and '`efg`' is a homotopy from '`funcomp g f`' to the identity of '`Y`,' the function '`f`' is a (weak) equivalence. The difference between the notions of a homotopy equivalence and a weak equivalence is somewhat subtle but important. Let '`X`' and '`Y`' be types and '`homeq X Y`' the type of quadruples '`(f, (g, (egf, efg)))`.' Theorem '`gradth`' (or rather definition '`weqgradth`') defines a function '`(homeq X Y) -> (weq X Y)`.' Using definitions '`homotweqinvweq`' and '`homotinvweqweq`' one gets a function '`(weq X Y) -> (homes X Y)`' and it is not difficult to show that these functions make '`weq X Y`' into a retract of '`homeq X Y`.' In general however this retraction is not an equivalence. The reason why '`weq`' is 'better' than '`homeq`' is related to the difference between *properties* and *structures* which is explained below.

Corollary '`iscontrweqf`' and definition '`wequnittocontr`' show that a type is contractible iff it is weakly equivalent to '`unit`' and in particular that up to weak equivalence there is only one contractible type. Corollary '`isweqmaponpaths`' shows that a weak equivalence defines a weak equivalence on '`paths`' types. Theorems '`twooutof3a`', '`twooutof3b`' and '`twooutof3c`' establish the 2-out-of-3 property of weak equivalences – if two out of three

https://doi.org/10.1017/S0960129514000577 Published online by Cambridge University Press

functions 'f', 'g', 'funcomp f g' are weak equivalences then so is the third. All these results are proved using 'gradth'.

Then there follows a series of simple results which assert that various natural functions such as the ones defining associativity and commutativity of direct products or distributivity of direct products and binary coproducts are weak equivalences.

The next tool-box which we introduce contains the type-theoretic versions of the results and definitions related to homotopy fibre sequences. Our approach to fibre sequences differs somewhat from the usual approaches. A fibre sequence structure 'fibseqstr' on a triple 'f:X->Y, g:Y->Z, z:Z' is defined as a homotopy from 'funcomp f g' to the constant function 'fun x : X => z' such that the associated function 'ezmap' from 'X' to the homotopy fibre 'hfiber f z' is a weak equivalence.

For any fibre sequence structure 'fs' on '(f,g,z)' and any object 'y:Y' we construct a function 'd1: paths (g y) z -> X' and the *derived* fibre sequence structure 'fibseq1' on the triple '(d1, f, y).' This construction can be iterated leading to a type theoretic construction of long homotopy exact sequences of fibrations.

We then investigate three standard situations where fibre sequences arise.

For any family of types 'P:Z->UU' over a type 'Z' and an object 'z:Z' we construct in 'fibseqpr1' a fibre sequence structure on the triple '(iz,pr1,z)' where 'iz' is the inclusion of the fibre 'P z' to 'total2 P' and 'pr1' the projection 'total2 P -> Z.' Applying to it the construction of the derived fibre sequence we get a family of weak equivalence 'ezweq1pr1' which connect the homotopy fibress of 'iz' with paths types on 'Z.'

For a function 'g:Y->Z' and an object 'z:Z' we define in 'fibseqg' the obvious structure of a fibre sequence on the triple '(hfiberpr1, g, z)' where 'hfiberpr1 : hfiber g z -> Y' is the standard function and give explicit descriptions of its first, second and third derived sequences.

Finally we construct a fibre sequence 'fibseqhf' of homotopy fibres of a composable pair of functions 'f:X->Y', 'g:Y->Z' for 'z:Z' and 'ye: hfiber g z' with the underlying sequence of morphisms of the form 'hfiber f (pr1 ye) -> hfiber (comp g f) z -> hfiber g z.'

The next fundamental notion which we introduce is the notion of h-levels. The definition 'isofhlevel n' uses the type 'nat' of natural numbers which is introduced in Coq.Init.Datatypes as the inductive type with two constructors 'O' of type 'nat' (corresponding to 0) and 'S' of type 'nat -> nat' (corresponding to the successor function $n \mapsto n+1$). Semantically we have that T is of h-level 0 iff it is contractible and of h-level $1+n$ iff for any x, y in T the paths space *paths x y* is of h-level n.

A function 'f:X->Y' is said to be of h-level 'n' if all its (homotopy) fibres are of h-level 'n.' In particular, a function is of h-level 'O' iff it is a weak equivalence.

Types of h-level 1 are called *propositions* and we write 'isaprop' instead of 'isofhlevel 1.' A homotopy type T is of h-level 1 iff for any $x, y \in T$ the paths space between x and y is contractible. In the world of classical homotopy types there are only two homotopy types with this property – the empty type and the contractible type. If 'T' is of h-level 1 and it is inhabited, i.e., there is an object 't:T' then, as 'iscontraprop1' shows 'T' is contractible. However, there are many non-equivalent types of h-level 1 which have no

https://doi.org/10.1017/S0960129514000577 Published online by Cambridge University Press

objects. This discrepancy between the model side and the syntactic side is the univalent form of the first Goedel's incompleteness theorem.

It is of a fundamental importance for the univalent approach to distinguish types which are propositions from more general types. In particular, if one wants to formalize univalently non-constructive proofs then one should add the axiom of excluded middle to the environment. Adding it in the form '`forall T:Type, coprod T (T-> empty)`' would be incompatible with the univalent models (and with the univalence axiom). This however does not mean that univalent semantics is incompatible with classical logic – the correct univalent formulation of the theorem of excluded middle is '`forall T:hProp, coprod T (T-> empty)`' where '`hProp := total2 (fun T:Type => isaprop T)`'.

A function between classical homotopy types $f : X \to Y$ is of h-level 1 iff it is homotopy equivalent to the inclusion of a union of connected components of Y into Y. On the type theoretic side we define inclusions as functions of h-level 1 ('`isincl`').

Inclusions correspond to *predicates* or *properties* – functions '`P:T->UU`' such that '`forall t:T, isaprop (P t).`' Given such '`P`' we can form the type '`total2 P`' whose objects are pairs '`(x,p)`' where '`x:T`' and '`p:P x.`' By '`isweqezmappr1`' the homotopy fibre of the projection '`pr1:total2 P -> T`' over '`x:T`' is weakly equivalent to '`P x.`' By '`isofhlevelweqf`' the h-levels are invariant under weak equivalences. We conclude that the projection '`total2 P -> T`' is a (homotopy) inclusion iff for all '`x:T`' the type '`P x`' is of h-level 1. If the h-level of '`P x`' is greater than 1 for some '`x:T`' then '`P`' defines a *structure* on objects of '`T.`'

One of the important naming conventions in our library is that any name which starts with 'is' such as '`isontr`' or '`isweq`' corresponds to a *property*. For further discussion of propositions and properties in the univalent approach see Section 4.

Types of h-level 2 are called sets (or, sometimes, h-sets) and we write '`isaset`' instead of '`isofhlevel 2.`' A classical homotopy type T is of h-level 2 iff the path space between any two points is either empty or contractible – one can easily see that this is equivalent to the condition that T is a disjoint union of contractible components, i.e., that it is homotopy equivalent to a set. On the type-theoretic side, due to the constructive nature of the theory, sets need not be disjoint unions of points. More precisely it is not necessarily true that for a set '`T`' and an object '`t:T`' there is an equivalence between '`T`' and '`coprod (compl T t) unit`' where '`compl T t`' is the complement to '`t`' in '`T.`' Types which satisfy the later property for all objects are called *types with decidable equality* (see '`isdeceq`'). We show that any type with decidable equality is an h-set in '`isasetifdeceq`' and use it to prove that Booleans ('`isasetbool`') and natural numbers ('`isasetnat`') are h-sets but not all h-sets can be proved to have decidable equality. A simple example of an h-set which does not have decidable equality is the type of functions '`nat -> Bool.`' Such types as Dedekind reals or p-adic numbers are also h-sets with undecidable equality.

Most of Mathematics as we know it deals with structures of h-level 2 on types of h-level 2. For example, a group is a pair '`(T,S)`' where '`T`' is an h-set and '`S`' is an object of the h-set of group structures on '`T`'. For further discussion of h-sets see Section 5.

For higher n the notion of h-level coincides with the well-known notion of n-types up to a shift of index by 2, i.e., a type T is of h-level $n+2$ iff for any x in T and $i > n$ one has $\pi_i(T, x) = 0$. The best known area of Mathematics whose univalent formalization

https://doi.org/10.1017/S0960129514000577 Published online by Cambridge University Press

requires types of h-level 3 is *category theory*. For a univalent approach to category theory (see Ahrens *et al.* 2014).

The file `uu0.v` contains three axioms – '`funextempty`,' '`etacorrection`' and '`funextfunax`' and the third one implies both the first and the second. Axioms are generally undesirable in constructive type theory even if, as is the case for these three axioms, they are semantically justified. The reason is that they tend to break a very important property of constructive type theories which is called *canonicity*. In its simplest form canonicity asserts that any object '`o`' of type '`nat`' (natural numbers) in the empty context which is in the *normal form* is of the form '`S ... S O`,' i.e., is a *numeral*. We will come back to this property in the discussion of the files `finitesets.v`, `hz.v` and `hq.v`.

For example, '`funextfunax`' can be used to define an object of type '`nat`' which is in the normal form but which is not a numeral as follows. Consider the transport along a path '`transportf.`' Let '`T`' be any type constant defined in the empty context (e.g. '`unit`' or even '`empty`'). Let '`f:=fun t:T => t`' be the identity function on '`T.`' Let '`e: paths f f`' be the path obtained by applying '`funextfunax`' to the homotopy '`fun t:T => idpath t t.`' Set '`x := transportf (fun g:T -> T => nat) e O.`' By doing this construction in Coq and typing '`Eval Compute in x`' – the command which displays the normal form of expression '`x`' – one immediately sees that '`x`' does not normalize to a numeral. For a further discussion of this phenomenon and its relation to the problem of constructive interpretation of the univalence axiom see Section 9.

Note that while an object of type '`nat`' defined with the use of axioms *may* happen not to normalize to a numeral, it is not necessarily so. In particular many of the test computations in files `finitesets.v`, `hz.v` and `hq.v` use theorems and definitions which include '`funextfunax.`'

Axiom '`funextfunax`' is known as the *functional extensionality* axiom. In its original form it is not even an 'axiom,' i.e., the type of '`funextfunax`' cannot be proved to be a proposition. More precisely, one can show that the homotopy type corresponding to the type of '`funextfunax`' under a univalent model has more than one connected component. To deal with this issue we use everywhere not '`funextfunax`' itself but its corollary '`funcontr`' which can be shown to be a proposition.

In the following parts of the library we use '`funcontr`' to show that the dependent product construction interacts in the expected way with weak equivalences (see '`isweqmaponsec`' and '`isweqmaponsec1`') and with h-levels (see '`impred`'). We also prove a number of results which justify our use of 'is' prefix in the names of constructions such as '`iscontr`', '`isweq`' and '`isofhlevel`' by showing that the types of corresponding constants are indeed of h-level 1.

4. File `hProp.v`

This is the only (so far) file in the folder '`hlevel1.`' It contains basic results related to types of h-level 1, i.e., to propositions. First we introduce the type '`hProp`' which relates to types of h-level 1 in the same way as the universe '`UU`' relates to all types. In fact we should consider the universe '`UU`' as a parameter of '`hProp`' writing '`hProp UU`' for the type of propositions in a universe '`UU.`' Unfortunately the universe management system

https://doi.org/10.1017/S0960129514000577 Published online by Cambridge University Press

does not allow universe parameters and we are forced to consider 'hProp' with respect to a fixed universe 'UU.'

In the univalent model a type is a proposition iff it is empty or contractible. Therefore, the model of 'hProp UU' is the simplicial subset of the model of 'UU' which consists of two connected components – the component of the empty type which is a 1-point simplicial set and the component of contractible types which is a large (relative to 'UU') contractible simplicial set.

This creates problems with the next construction in 'hProp' which we call 'ishinh_UU' and which is also known as the bracket type or squash type construction. The idea is that for a type 'T' there should be a proposition 'ishinh_UU T' which is true iff 'T' is inhabited. This is equivalent to saying that 'ishinh_UU T' should be defined together with a function 'hinhpr T : T -> ishinh_UU T' which is universal among functions from 'T' to propositions. Using the fact that h-levels are stable under the formation of dependent products ('impred' from uu0.v) we show in 'isapropisinh' that 'ishinh_UU T' is indeed a proposition and in 'hinhuniv' that the function 'hinhpr' it is universal.

However, there is an element of cheating here. In fact this part of hProp.v would not go through in un-patched Coq. The only reason it works in Coq is that we use the patched version which does not check universe consistency.

The problem is that 'ishinh_UU T' is a proposition in a bigger universe than 'UU' which is universal with respect to functions from 'T' to propositions in 'UU.'

How can this problem be fixed without introducing potential inconsistency? There are currently three ideas. The first two have to do with *resizing rules* and the third with *higher inductive types*. All three are associated with interesting unsolved problems. Note that the issue is particular to the constructive setting. If we did not care about computation and added the excluded middle axiom then we could use a double negation version 'isinhdneg' of 'ishinh' which does not lead to any issues with universe levels.

In the following part of the library we define an interpretation of intuitionistic logic on 'hProp.' The construction of 'ishinh_UU' is a necessary prerequisite for the construction of the disjunction – the disjoint union of two propositions considered as types is not in general a proposition and one has to apply 'ishinh_UU' to obtain disjunction as an operation on propositions.

In the last part of the file, we introduce the univalence axiom for 'hProp' and consider some of its corollaries.

5. File hSet.v

This file contains basic results related to sets, i.e., types of h-level 2. The first brief section discusses types which satisfy axiom of choice, i.e., which are 'projective objects.' It is later used in stnfsets.v and fintesets.v to show that the axiom of choice holds for families over finite sets.

Then we introduce a series of definitions and results about relations on types. Many of these results are later used to prove standard properties of comparisons on natural numbers and later on integers and rational numbers.

https://doi.org/10.1017/S0960129514000577 Published online by Cambridge University Press

The most important part of this file deals with set-quotients of types. The theory of quotients is well known to be one of the difficult points of the usual constructive type theory. The univalent model provides an explanation for this fact – since types are homotopy types rather than sets the quotients need to be understood as homotopy quotients which are often very complicated.

The set-quotients are, from homotopy-theoretical point of view, quotients with respect to 'homotopy-invariant equivalence relations.' The finest such relation is given by the condition 'a is path-connected to b' with the corresponding quotient being π_0. Quotients with respect to stronger equivalence relations on T are quotients of $\pi_0(T)$. The quotient with respect to the strongest relation, i.e., the one where any two points are equal is equivalent to '`ishinh_UU T.`'

While in classical setting such quotients create no problems in the constructive setting things are more complicated. One problem is the increase in the universe level when one passes to a quotient. It is similar to the problem which we discussed in the context of '`ishinh_UU.`'

Another problem can be seen in the way in which taking quotients interact with taking sub-objects. Let '`X`' be a type, '`R`' an equivalence relation on '`X`' and '`P:setquot R -> hProp`' a predicate on the quotient of '`X`' with respect to '`R`'. The composition '`Q`' of '`P`' with the projection '`setquotpr R:X -> setquot R`' is a predicate on '`X`'. Let '`U:= carrier P`' and '`X':= carrier Q`' be they sub-objects of '`setquot R`' and '`X`' respectively corresponding to '`P`' and '`Q.`' The restriction '`R'`' of '`R`' to '`X'`' is an equivalence relation and we may consider two types '`setquot R'`' and '`U`'. As is proved in '`weqsubquot`' these two types are equivalent. However the equivalence '`U -> setquot R',`' the obvious function '`X' -> U`' and the projection '`setquotpr R': X' -> setquot R'`' do not commute *computationally*.

This leads for example to the use of somewhat unnatural constructions to define the inverse on non-zero elements of fields of fractions since the straightforward definition 'does not compute.'

A possible way to deal with this issue by extending the type theory of Coq with a new component called tfc-terms (from the trivial fibration/cofibration axiom of model categories) is briefly discussed in the comments after '`weqsubquot.`'

At the end of the file `hSet.v` we describe another approach to set-quotients. Originally this part was written because I thought that the computational behaviour of this alternative construction will be better. However, it turned out to have very similar (and probably equivalent) problems as the first one.

6. Files algebra1*.v

These files introduce the standard notions of abstract algebra including the interaction between algebraic operations and partial orderings.

The file `algebra1a.v` introduces basic definitions related to binary operations and pairs of binary operations on h-sets. A few definitions where the generalizations from h-sets to all types are straightforward are given for arbitrary types.

https://doi.org/10.1017/S0960129514000577 Published online by Cambridge University Press

The file `algebra1b.v` is about monoids, abelian monoids, groups and abelian groups including the construction of monoids of fractions in the abelian case.

The file `algebra1c.v` is about rigs (semi-rings with a unit such that $0 \cdot 1 = 1 \cdot 0 = 0$), commutative rigs, rings and commutative rings. It includes the construction of the ring of differences from a rig and of localization of a commutative ring by a multiplicative system of elements. We also prove the basic results about the behaviour of partial orderings and equivalence relations with respect to these constructions.

The file `algebra1d.v` is the first one which contains material which is probably unusual for an average mathematician. It deals with the notions of an integral domain and of a field in constructive framework. Unlike the notions considered above the notions of an integral domain and of a field acquire additional distinctions in constructive Mathematics relative to the classical one. For example the condition 'every non-zero element is invertible' in the definition of a field has three non-equivalent constructive formulations – one can require that any element which is non-invertible is zero or that any element which is non-zero is invertible or that any element is either invertible or equals zero.

In `algebra1d.v` we consider the later definition (any element is either invertible or equals zero). It is the strongest (most restrictive) one and it immediately implies that the equality on a field is a decidable relation. This is clearly unsatisfactory for many purposes – for example real numbers or the 'field' of power series do not satisfy this condition. To deal with this problem one needs to introduce the notion of apartness relations and study their interactions with algebraic structures. Some information on the subject as well as further formalizations in the style of this library can be found in Pelayo *et al.* (2014).

In `algebra1d.v` we restrict ourselves to the case of decidable equality and give in that case a constructive definition of a field of functions of a (decidable) integral domain.

All constructions in the algebra files have non-trivial extensions from h-sets to arbitrary types. For example, the notion of a monoid generalizes to as yet undefined notion of an H-type which should include all the higher coherence structures associated with associativities. The notion of a partially ordered set generalizes to the notion of $(\infty, 1)$-category and the notion of a partially ordered monoid generalizes to the notion of a monodical $(\infty, 1)$-category. I do not know what is the classical name for the higher analogues of rigs and commutative rigs (going from rigs to rings is straightforward since the only axiom involved is the invertibility of addition which has a formulation common for types of all levels) and whether such objects been considered. None of these have as yet been defined in terms of type theory.

7. File `hnat.v`

In this file, we provide basic constructions and results related to the arithmetic operations and comparisons on natural numbers. The type '`nat`' is introduced in Coq.Init. We use this definition for natural numbers and also standard definitions for the addition, subtraction (which is defined such that for $n < m$ one has $n - m = 0$) and multiplication on '`nat.`'

Our approach to comparisons is different from the one used in Coq.Init. There the main comparison is '`le`' which is introduced through an inductive definition based on the principle that '`le`' is a family of types whose objects are either 'reflexivity' comparisons in

https://doi.org/10.1017/S0960129514000577 Published online by Cambridge University Press

'le n n' or successor comparisons obtained from constructor of the form 'le n m -> le n (S m).'

Since our library uses only those inductive constructions in Coq which are necessary for the definition of the standard ingredients of the Martin-Löf type theory we do not use 'le.' Instead we start with Boolean 'greater' which we call 'natgtb' defined by induction on 'nat' as a function 'nat -> nat -> Bool,' define 'natgth n m' as 'paths (natgtb n m) true' and then define the three other comparisons 'natlth,' 'natleh' and 'natgeh' in terms of 'natgth.' This has the advantage that the same definitions of 'less," 'less or equal' and 'greater or equal' in terms of 'greater' work for integers and rationals and the proofs of the main properties of these comparisons from the main properties of 'greater' can be directly copied from the 'nat' case to the cases of 'hz' and 'hq.'

After this choice of how to define the comparisons and prove their properties is made, the rest is rather straightforward.

At the end of the file, we analyse the Coq.Init construction of 'le'-types showing that 'le n m' is always a proposition (i.e., has h-level 1).

8. File stnfsets.v

This is the first of the two files where we introduce constructions related to finite sets. In this file, we deal only with 'standard' finite sets which are defined such that 'stn n' is the type of natural numbers which are less than 'n.'

Most of the file is occupied by constructions of various weak equivalences involving standard finite sets. For example, we construct a weak equivalence between 'weq (stn n) (stn n)' and 'stn (factorial n).'

At the end of the file, we use the notion of a standard finite set to formulate and prove results on bounded quantification and then to give a univalent proof of the accessibility theorem for natural numbers.

9. File finitesets.v

We define the structure of having n elements on a type 'T' as a weak equivalence from the standard set with n elements to 'T'. A type 'T' is called a finite set if there exists (or, in terminology of Univalent Foundations Project (2013), if there merely exists) a pair 'tpair _ n s' where 'n:nat' and 's' is a structure of having n elements on 'T.' We then use the results of stnfsets.v to show that various constructions on finite sets produce finite sets.

An important property of our approach is that despite the fact that we use 'mere' existence in the definition of what it means to be finite, there is a function 'fincard' which computes the cardinality of a finite set.

Related to this function are several examples of computation which are included at the end of the file finitesets.v. The property of Martin-Löf type theory which makes automatic computation possible is known as *canonicity*. In its simplest form, the *canonicity theorem* asserts that any object 'o' of type 'nat' defined in the empty context which is in the normal form is a numeral, i.e., a sequence 'S ... S O' (recall that 'O' is the notation for $0 \in \mathbf{N}$ and 'S' is the notation for the successor function $n \mapsto 1 + n$).

https://doi.org/10.1017/S0960129514000577 Published online by Cambridge University Press

The possibility of automatic terminating computation is a corollary of this property combined with *strong normalization* – the assertion that any sequence of reductions starting with a given well-formed expression is finite[†].

By definition, an expression is said to be in the normal form if there are no reduction steps starting with this expression. Therefore, in a theory with strong normalization for any well-formed expression there is a finite sequence of reductions which results in an expression in the normal form. If the expression in question is an object of type '`nat`' we conclude that applying any normalization algorithm to this expression, we will obtain after finitely many step a numeral, i.e., we will *compute* this expression.

Consider now Martin-Löf type theory together with an added axiom '`A:T_A.`' While in Martin-Löf type theory strong normalization holds over any context (i.e., after the addition of any number of axioms) the canonicity theorem usually fails over most non-empty contexts.

For example, if we obtain an object '`o`' of type '`nat`' using axiom '`funextempty`' then there is no guarantee that its normal form will be a numeral or, as we say, there is guarantee that 'it will compute.' However, many expressions which contain an axiom will compute since the subexpressions containing the axiom get eliminated at some stage of the normalization process.

In Coq, there are two main normalization algorithms which can be called by the commands '`Eval compute`' and '`Eval lazy`' respectively. Theoretically these algorithm are equivalent in the sense that both are supposed to always terminate and the answers produced should coincide. In practice, I have encountered many cases when '`Eval lazy`' terminates in a reasonable amount of time while '`Eval compute`' applied to the same expression takes too much time for me to wait it out.

The lines in the file `finitesets.v` which start with '`Eval compute`' or '`Eval lazy`' are tests to verify that the use of the axioms in various proofs of finiteness does not interfere with the computability of the cardinality function '`fincard.`'

Note that all of the axioms which we use in this library are corollaries of the general univalence axiom. So if or when the main conjecture on constructive interpretation of the univalence will be proved, we will have an algorithm which, when applied to any well-formed expression '`o`' of type '`nat`' which uses any of the axioms of the library will return an expression '`o''`' without any axioms in it and a proof that the new expression is pathsequal to '`o.`' This algorithm will however be of a different kind than the normalization algorithms[‡].

10. Files `hz.v` and `hq.v`

In these two files, we define first integers '`hz`' and then rational numbers '`hq.`' In both cases we follow Bourbaki approach. In the file `hnat.v` we have defined a commutative rig

[†] Strong normalization is a difficult theorem. In particular, using a variant of Goedel's argument, it can be shown that it cannot be proved unconditionally. In practice, all known proofs of strong normalization for Martin-Löf type theory require one to assume that a substantial portion of ZFC is consistent.

[‡] For a recent advance in solving this problem see `https://github.com/simhu/cubical`.

https://doi.org/10.1017/S0960129514000577 Published online by Cambridge University Press

of natural numbers. To get 'hz' we apply the general construction 'commrigtocommrng' of the ring of differences of a commutative rig from algebra1c.v. To get 'hq' we apply to the integral domain 'hz' the general construction 'fldfrac' of the field of fractions from algebra1d.v. Note that this construction requires the equality on the integral domain to be decidable. This is due to our definition 'isafield' of what a field is.

At the end of both files hz.v and hq.v are more test computations.

11. File funextfun.v

In this file, we introduce the univalence axiom and prove that it implies the functional extensionality axiom 'funextfunax' from uu0.v . The rest of the library does not depend on this file.

12. Appendix: On the Coq System for Naming and Loading Libraries

I am grateful to Dan Grayson for figuring out the answers to many questions which I had while writing this appendix.

At the top of the files of the foundations library (other than 'uuu.v') there are lines starting with 'Add LoadPath' and 'Require.' These are commands which tell Coq where to look for 'libraries' which are needed to compile the given file. For the purpose of this explanation I will use the file 'uu0.v.'

Understanding why a particular combination of these commands, the '-R' options in the 'Makefile,' and the '-R' options in the emacs variable 'coq-prog-args' used by the 'Proof General' when starting 'Coq' works, while a slightly different one does not, can be very confusing. Below, I will try to describe the minimum that I believe is sufficient to understand why the particular choices made in foundations library work as they do and to be able to predict the effect of possible small modifications of these choices.

The 'Require' command in the file 'uu0.v' tells Coq to load a 'library' that is called 'Foundations.Generalities.uuu.' The word 'Export,' as opposed to the word 'Import,' means that 'Foundations.Generalities.uuu' will also be loaded every time the 'Foundations.Generalities.uu0' (the name of the library in the file 'uu0.vo') is loaded.

Two issues contribute to the complexity of the behaviour of these commands. One is how the name of the library which is contained in a given '.vo' file is determined and another one is which files and directories Coq will look through when it tries to execute the 'Require' command and what will be the name of the library it will look for in each of these files (which will, as we will see below, be usually different from the name specified in the 'Require.')

The '.vo' files are created by 'coqc,' the non-interactive mode of Coq, using as the input a '.v' file, i.e., a file which contains the humanly readable Coq code. This is what the Makefile in the top directory of foundations library does: it calls the program 'coqc' for each of the '.v' files in the library to produce the corresponding '.vo' files. One cannot, for example, experiment with 'uu0.v' in 'Proof General' until 'uuu.vo' has been created by running 'coqc' on 'uuu.v.'

https://doi.org/10.1017/S0960129514000577 Published online by Cambridge University Press

If no, '`-R`' option is given when '`coqc`' command is called then the name of the library in the '`.vo`' file created by this call is the name of the (main part of) the '`.v`' file *as given to the* '`coqc.`' In the case of '`uu0.v`' the command '`coqc uu0,`' run in the directory '`Foundations/Generalities/,`' will put the name '`uu0`' to the library in the '`uu0.vo`' which it will produce. The command '`coqc Generalities/uu0`' ran in the directory '`Foundations`' will put into the '`uu0.vo`' a 'library' called '`Generalities.uu0.`'

What will happen if an arbitrary '`-R`' option is given to the '`coqc`' command I do not know. If the option is of the form '`-R "." "name"`' then the name of the library in the '`.vo`' file will be the name one would expect without the '`-R`' option with '`name.`' appended in front of it. The '`name`' may itself consist of several components, e.g., it can be '`Foundations.Generalities.`'

Suppose now that we want to run coq on the '`uu0.v`' file. When Coq reads the command '`Require Export Foundations.Generalities.uu0`' it will start looking for a file whose name is '`uu0.vo`' 'on the '`LoadPath.`''.

The latter expression means the following. '`LoadPath`' is represented by a list of pairs where the second component of the pair is the actual name of a directory and the first component is an expression of the form '`n1.n2.....nk`' (where '`ni`' are names) which Coq will use instead of the directory name internally.

One can find the content of this list from '`Proof General`' by running Coq over the command '`Print LoadPath.`'

The '`Add LoadPath`' command adds to this list the line which you would expect from the arguments of the '`Add LoadPath.`' A version of this command '`Add Rec LoadPath`' will also add the lines corresponding to all of the subdirectories of the directory mentioned in the arguments (except possibly some whose names contain symbols which are not permitted in identifiers).

If the Coq program was given '`-R name namedir`' as an argument it will have the same effect on the '`LoadPath`' as the command '`Add Rec LoadPath "namedir' name.`'

When Coq encounters the line '`Require Export n1.n2....nk.n`' it does the following. First it looks for the file '`n.vo`' in the directories '`dirname`' which appear in '`LoadPath`' in pair with '`n1.n2.....nk.`' It will take the first such file it finds and will check whether it contains library '`n1.....nk.n.`' If it does not it will not look for another possible match and will give an error message. If it does it will load the library.

The content of the '`LoadPath`' can also be modified by using '`-R`' option when calling Coq, e.g., by customizing the variable '`coq-prog-args`' in '`Proof General.`' One can experiment with the results of such modifications using the '`Print LoadPath`' command.

References

Ahrens, B., Kapulkin, C. and Shulman, M. (2014) Univalent categories and the Rezk completion. *Mathematical Structures in Computer Science* `http://dx.doi.org/10.1017/S0960129514000486`.

Coquand, T. and Huet, G. (1988) The calculus of constructions. *Information and Computation* **76** (2–3) 95–120.

https://doi.org/10.1017/S0960129514000577 Published online by Cambridge University Press

Grothendieck, A. (1997) Esquisse d'un programme. In: *Geometric Galois Actions, 1*, London Mathematical Society Lecture Note Series volume 242, Cambridge University Press, Cambridge 5–48. (With an English translation on pp. 243–283.)

Kapulkin, C., LeFanu Lumsdaine, P. and Voevodsky, V. (2012) The simplicial model of univalent foundations. Preprint, *arXiv:1211.2851*.

Luo, Z. (1994) *Computation and Reasoning. A Type Theory for Computer Science*, International Series of Monographs on Computer Science volume 11, The Clarendon Press. Oxford University Press, New York.

Makkai, M. (1995) First order logic with dependent sorts, with applications to category theory. Preprint, Available at: http://www.math.mcgill.ca/makkai/folds/foldsinpdf/FOLDS.pdf.

Paulin-Mohring, C. (1993) Inductive definitions in the system Coq: Rules and properties. In: *Typed Lambda Calculi and Applications (Utrecht, 1993)*, Springer Lecture Notes in Computer Science volume 664 Berlin 328–345.

Pelayo, A., Voevodsky, V. and Warren, M. A. (2014) A preliminary univalent formalization of the p-adic numbers. *Mathematical Structures in Computer Science* `http://dx.doi.org/10.1017/S0960129514000541`.

Univalent Foundations Project (2013) Homotopy type theory: Univalent foundations for mathematics. Available at: `http://homotopytypetheory.org/book` .

Voevodsky, V. (2011) Resizing rules, slides from a talk at TYPES2011. Available at: https://github.com/vladimirias/2011_Bergen.