With-Type – Poor man's dependent types

Dominique Unruh
RWTH Aachen, University of Tartu

March 17, 2025

Abstract

The type system of Isabelle/HOL does not support dependent types or arbitrary quantification over types. We introduce a system to mimic dependent types and existential quantification over types *in limited circumstances* at the top level of theorems.

Contents

1 Introduction **1**

2 *Misc-With-Type* – Some auxiliary definitions and lemmas **3**

3 *With-Type* – Setting up the *with-type* mechanism **3**

4 *With-Type-Example* – Some contrieved simple examples **7**
4.1 Semigroups (class with one parameter) . . . 7
4.1.1 Example . . . 8
4.2 Abelian groups (class with several parameters) . . . 8
4.2.1 Example . . . 9

5 *Example-Euclidean-Space* – Example: compactness of the sphere **9**
5.1 Setting up type class *finite* for *with-type* . . . 10
5.2 Vector space over a given basis . . . 10
5.3 Compactness of the sphere. . . . 11

1 Introduction

The type system of Isabelle/HOL is relatively limited when it comes to the treatment of types (at least when compared with systems such as Coq or Lean). There is no support for arbitrary quantification over types, nor can types depend on other values. *Universal* quantification over types is implicitly possible at the top level of a theorem because type variables are treated as universally quantified.[1] In a very limited way, we can also mimic existential quantification on the top level: Instead of saying, e.g., `∃a. card (UNIV :: a set) = 3` ("there exists a type with three elements"), we can define a type with the desired property (`typedef witness = "1,2,3::int"`) and prove `card (UNIV :: witness set) = 3`. This achieves the same thing but it suffers from several drawbacks:

- We can only use this encoding at the top level of theorems. E.g., we cannot represent the claim `P (∃a. card (UNIV :: a set) = 3)` where `P` is an arbitrary predicate.
- It only works when we can explicitly construct the type that is claimed to exist (because we need to describe it in the `typedef` command).
- The witness we give cannot depend on variables local to the current theorem or proof because the `typedef` command can only be given on the top level of a theory, and can only depend on constants. E.g., it would not be possible to express something like:

```
∀n::nat.  (n >= 1 --> (∃a. card (UNIV :: a set) = n)).        (1)
```

In this work, we resolve the third limitation. Concretely, we will be able to define a set (not a type!) `witness n` that depends on a natural number `n`,[2] and write:

```
n >= 1 --> let 'a::type=witness n in (card (UNIV :: 'a set) = n)
```

This statement is read as:

> If `n >= 1`, and `'a` is defined to be the type described by the set `witness n` (imagine a *local* `typedef 'a = "witness n"`), then `card (UNIV :: a set) = n` holds.

This is nothing else than (1) with an explicitly specified witness.

We call the Isabelle constant implementing this construct `with_type`, because `let 'a::type=witness in P` can be read as "with type `'a` defined by `witness`, `P` holds".

Since in `let 'a::type=spec in ...`, the `spec` can depend on local variables, we essentially have encoded a limited version of dependent types. Limited because our encoding is not meaningful except at the top level of a theorem ("`premises ==> let 'a::type = ...`" is ok, "`P (let 'a::type = ...)`" for arbitrary `P` is not).

To be able to actually use this encoding in proofs, we implement three reasoning rules for introduction, elimination, and modus ponens. These are *roughly* the following:

[1] For example, a theorem such as `(1::?'a) + 1 = 2` can be interpreted as `∀a. (1::a) + 1 = 2`.

[2] In this example, `witness n` simply has to be an arbitrary `n`-element set, e.g., `witness n = {..<n}`.

$$\frac{\texttt{P}\ \textit{(given typedef)}}{\texttt{let 'a::type = w in P}}\ \textsc{Intro}$$

$$\frac{\texttt{let 'a::type = w in P} \qquad \texttt{'a}\ \textit{does not occur in}\ \texttt{P}}{\texttt{P}}\ \textsc{Elim}$$

$$\frac{\texttt{let 'a::type = w in P} \qquad \texttt{P ==> Q}\ \textit{(given typedef)}}{\texttt{let 'a::type = w in Q}}\ \textsc{ModusPonens}$$

Here "*(given typedef)*" means that the respective premise can be shown in a context where the local equivalent of a `typedef 'a = "w"` was declared. (In particular, there are morphisms `rep`, `abs` between `'a` and the set `w`.)

The elimination rule uses the `Types_To_Sets` extension [1] to get rid of the "unused" `let 'a::type`.

The `with_type` mechanism is not limited to types of type class `type` (the Isabelle/HOL type class containing all types). We can also write, e.g., `let 'a::ab_group_add = set with ops in P` which would say that `'a` is an abelian additive group (type class `ab_group_add`) defined via `typedef 'a = "set"` with group operations `ops` (which specifies the addition operation, the neutral element, etc.).

2 *Misc-With-Type* – Some auxiliary definitions and lemmas

theory *Misc-With-Type*
  imports *Main*
begin

lemma *type-definition-bij-betw-iff*: ‹*type-definition rep* (*inv rep*) *S* ⟷ *bij-betw rep UNIV S*›
  ⟨*proof*⟩

inductive *rel-unit-itself* :: ‹*unit* ⇒ *'a itself* ⇒ *bool*› **where**
— A canonical relation between *unit* and *'a itself*. Note that while the latter may not be a singleton type, in many situations we treat it as one by only using the element *TYPE*(*'a*).
  ‹*rel-unit-itself* () *TYPE*(*'a*)›

lemma *Domain-rel-unit-itself*[*simp*]: ‹*Domainp rel-unit-itself x*›
  ⟨*proof*⟩
lemma *rel-unit-itself-iff*[*simp*]: ‹*rel-unit-itself x y* ⟷ (*y* = *TYPE*(*'a*))›
  ⟨*proof*⟩

end

3 *With-Type* – Setting up the *with-type* mechanism

theory *With-Type*
  imports *HOL–Types-To-Sets.Types-To-Sets Misc-With-Type HOL–Eisbach.Eisbach*

keywords *with-type-case* :: *prf-asm* % *proof*
begin

definition *with-type-wellformed* **where**
— This states, roughly, that if operations *rp* satisfy the axioms of the class, then they are in the domain of the relation between abstract/concrete operations.
‹*with-type-wellformed C S R* $\longleftrightarrow$ ($\forall$ *r rp. bi-unique r* $\longrightarrow$ *right-total r* $\longrightarrow$ *S* = *Collect* (*Domainp r*) $\longrightarrow$ *C S rp* $\longrightarrow$ *Domainp* (*R r*) *rp*)›
for *C* :: ‹*'rep set* ⇒ *'rep-ops* ⇒ *bool*›
and *S* :: ‹*'rep set*›
and *R* :: ‹(*'rep* ⇒ *'abs* ⇒ *bool*) ⇒ (*'rep-ops* ⇒ *'abs-ops* ⇒ *bool*)›

In the following definition, roughly speaking, *with-type C R S rep-ops P* means that predicate *P* holds whenever type *'abs* (called the abstract type, and determined by the type of *P*) is an instance of the type class described by C,R, and is a stands in 1-1 correspondence to the subset *S* of some concrete type *'rep* (i.e., as if defined by *typedef 'abs = S*).

S – the carrier set of the representation of the type (concrete type)

rep-ops – operations on the concrete type (i.e., operations like addition or similar)

C – the properties that *S* and *rep-ops* are guaranteed to satisfy (basically, the type-class definition)

R – transfers a relation *r* between concrete/abstract type to a relation between concrete/abstract operations (*r* is always bi-unique and right-total)

P – the predicate that we claim holds. It can work on the type *'abs* (which is type-classed) but it also gets *rep* and *abs-ops* where *rep* is an embedding of the abstract into the concrete type, and *abs-ops* operations on the abstract type.

The intuitive meaning of *with-type C R S rep-ops P* is that *P* holds for any type *'t* that that can be represented by a concrete representation (*S*, *rep-ops*) and that has a type class matching the specification (*C*, *R*).

definition ‹*with-type* = (λ*C R S rep-ops P*. *S*≠{} ∧ *C S rep-ops* ∧ *with-type-wellformed C S R* ∧ ($\forall$ *rep abs-ops. bij-betw rep UNIV S* $\longrightarrow$ (*R* (λ*x y. x* = *rep y*) *rep-ops abs-ops*) $\longrightarrow$ *P rep abs-ops*))›
for *S* :: ‹*'rep set*› **and** *P* :: ‹(*'abs* ⇒ *'rep*) ⇒ *'abs-ops* ⇒ *bool*›
and *R* :: ‹(*'rep* ⇒ *'abs* ⇒ *bool*) ⇒ (*'rep-ops* ⇒ *'abs-ops* ⇒ *bool*)›
and *C* :: ‹*'rep set* ⇒ *'rep-ops* ⇒ *bool*›
and *rep-ops* :: ‹*'rep-ops*›

For every type class that we want to use with *with-type*, we need to define two constants specifying the axioms of the class (*WITH-TYPE-CLASS-classname*) and specifying how a relation between concrete/abstract type is lifted to a relation between concrete/abstract operations (*WITH-TYPE-REL-classname*). Here we give the trivial definitions for the default type class *type*

definition ‹*WITH-TYPE-CLASS-type S ops* = *True*› **for** *S* :: ‹*'rep set*› **and** *ops* :: *unit*
definition ‹*WITH-TYPE-REL-type r* = ((=) :: *unit* ⇒ - ⇒ -)› **for** *r* :: ‹*'rep* ⇒ *'abs* ⇒ *bool*›

named-theorems *with-type-intros*
— In this named fact collection, we collect introduction rules that are used to automatically discharge some simple premises in automated methods (currently only *with-type-intro*).

lemma [*with-type-intros*]: ‹*WITH-TYPE-CLASS-type S ops*›
⟨*proof*⟩

We need to show that *WITH-TYPE-CLASS-classname* and *WITH-TYPE-REL-classname* are wellbehaved. We do this here for class *type*. We will need this lemma also for registering the type class *type* later.

lemma *with-type-wellformed-type*[*with-type-intros*]:
‹*with-type-wellformed WITH-TYPE-CLASS-type S WITH-TYPE-REL-type*›
⟨*proof*⟩

lemma *with-type-simple*: ‹*with-type WITH-TYPE-CLASS-type WITH-TYPE-REL-type S* () *P* ⟷ *S*≠{} ∧ (∀ *rep. bij-betw rep UNIV S* ⟶ *P rep* ())›
— For class *type*, *with-type* can be rewritten in a much more compact and simpler way.
⟨*proof*⟩

lemma *with-typeI*:
assumes ‹*S* ≠ {}›
assumes ‹*C S p*›
assumes ‹*with-type-wellformed C S R*›
assumes *main*: ‹⋀(*rep* :: ′*abs* ⇒ ′*rep*) *abs-ops. bij-betw rep UNIV S* ⟹ *R* (λ*x y. x* = *rep y*) *p abs-ops* ⟹ *P rep abs-ops*›
shows ‹*with-type C R S p P*›
⟨*proof*⟩

lemma *with-type-mp*:
assumes ‹*with-type C R S p P*›
assumes ‹⋀*rep abs-ops. bij-betw rep UNIV S* ⟹ *C S p* ⟹ *P rep abs-ops* ⟹ *Q rep abs-ops*›
shows ‹*with-type C R S p Q*›
⟨*proof*⟩

lemma *with-type-nonempty*: ‹*with-type C R S p P* ⟹ *S* ≠ {}›
⟨*proof*⟩

lemma *with-type-prepare-cancel*:
— Auxiliary lemma used by the implementation of the *cancel-with-type*-mechanism (see below)
fixes *S* :: ‹′*rep set*› **and** *P* :: *bool*
and *R* :: ‹(′*rep* ⇒ ′*abs* ⇒ *bool*) ⇒ (′*rep-ops* ⇒ ′*abs-ops* ⇒ *bool*)›
and *C* :: ‹′*rep set* ⇒ ′*rep-ops* ⇒ *bool*›
and *p* :: ‹′*rep-ops*›
assumes *wt*: ‹*with-type C R S p* (λ(-::′*abs*⇒′*rep*) -. *P*)›
assumes *ex*: ‹(∃ (*rep*::′*abs*⇒′*rep*) *abs. type-definition rep abs S*)›
shows *P*
⟨*proof*⟩

lemma *with-type-transfer-class*:

— Auxiliary lemma used by ML function *cancel-with-type*
includes *lifting-syntax*
fixes *Rep* :: ‹*'abs* ⇒ *'rep*›
and *C S*
and *R* :: ‹(*'rep*⇒*'abs*⇒*bool*) ⇒ (*'rep-ops* ⇒ *'abs-ops* ⇒ *bool*)›
and *R2* :: ‹(*'rep*⇒*'abs2*⇒*bool*) ⇒ (*'rep-ops* ⇒ *'abs-ops2* ⇒ *bool*)›
assumes *trans*: ‹⋀*r* :: *'rep* ⇒ *'abs2* ⇒ *bool*. *bi-unique r* ⟹ *right-total r* ⟹ (*R2 r* ===> (⟷)) (*C* (*Collect* (*Domainp r*))) *axioms*›
assumes *nice*: ‹*with-type-wellformed C S R2*›
assumes *wt*: ‹*with-type C R S p P*›
assumes *ex*: ‹∃(*Rep* :: *'abs2*⇒*'rep*) *Abs*. *type-definition Rep Abs S*›
shows ‹∃ *x*::*'abs-ops2*. *axioms x*›
⟨*proof*⟩

lemma *with-type-transfer-class2*:
— Auxiliary lemma used by ML function *cancel-with-type*
includes *lifting-syntax*
fixes *Rep* :: ‹*'abs* ⇒ *'rep*›
and *C S*
and *R* :: ‹(*'rep*⇒*'abs*⇒*bool*) ⇒ (*'rep-ops* ⇒ *'abs itself* ⇒ *bool*)›
and *R2* :: ‹(*'rep*⇒*'abs2*⇒*bool*) ⇒ (*'rep-ops* ⇒ *'abs2 itself* ⇒ *bool*)›
assumes *trans*: ‹⋀*r* :: *'rep* ⇒ *'abs2* ⇒ *bool*. *bi-unique r* ⟹ *right-total r* ⟹ (*R2 r* ===> (⟷)) (*C* (*Collect* (*Domainp r*))) *axioms*›
assumes *nice*: ‹*with-type-wellformed C S R2*›
assumes *rel-itself*: ‹⋀(*r* :: *'rep* ⇒ *'abs2* ⇒ *bool*) *p*. *bi-unique r* ⟹ *right-total r* ⟹ (*R2 r*) *p TYPE*(*'abs2*)›
assumes *wt*: ‹*with-type C R S p P*›
assumes *ex*: ‹∃(*Rep* :: *'abs2*⇒*'rep*) *Abs*. *type-definition Rep Abs S*›
shows ‹*axioms TYPE*(*'abs2*)›
⟨*proof*⟩

Syntactic constants for rendering *with-type* nicely.

syntax *-with-type* :: *type* ⇒ *'a* => *'b* ⇒ *'c* (‹*let* - = - *in* -› [*0*,*0*,*10*] *10*)
syntax *-with-type-with* :: *type* ⇒ *'a* => *args* ⇒ *'b* ⇒ *'c* (‹*let* - = - *with* - *in* -› [*0*,*0*,*10*] *10*)
syntax (**output**) *-with-type-sort-annotation* :: *type* ⇒ *sort* ⇒ *type* (‹-::-›)
— An auxiliary syntactic constant used to enforce the printing of sort constraints in certain terms.

⟨*ML*⟩

Register the type class *type* with the *with-type*-mechanism. This enables readable syntax, and contains information needed by various tools such as the *cancel-with-type* attribute.

⟨*ML*⟩

Enabling input/output syntax for *with-type*. This allows to write, e.g., *let 't::type* = *S in P*, and the various relevant parameters such as *WITH-TYPE-CLASS-type* etc. are automatically looked up based on the indicated type class. This only works with type classes that have been registered beforehand.

Using the syntax when printing can be disabled by *declare* [[*with-type-syntax*=*false*]].

⟨*ML*⟩

Example of input syntax:

term ‹*let 't::type = N in rep-t = rep-t*›

Removes a toplevel *let 't=...* from a proposition *let 't=... in P*. This only works if *P* does not refer to the type *'t*.

⟨*ML*⟩

Convenience method for proving a theorem of the form *let 't=...*.

method *with-type-intro = rule with-typeI*; (*intro with-type-intros*)*?*

Method for doing a modus ponens inside *let 't=...*. Use as: *using PREMISE proof with-type-mp*. And inside the proof, use the command *with-type-case* before proving the main goal. Try *print-theorems* after *with-type-case* to see what it sets up.

⟨*ML*⟩

end

4 *With-Type-Example* – Some contrieved simple examples

theory *With-Type-Example*
imports *With-Type HOL−Computational-Algebra.Factorial-Ring Mersenne-Primes.Lucas-Lehmer-Code*
begin

unbundle *lifting-syntax* **and** *no m-inv-syntax*

4.1 Semigroups (class with one parameter)

definition ‹*WITH-TYPE-CLASS-semigroup-add S plus* ⟷ (∀ *a*∈*S*. ∀ *b*∈*S*. *plus a b* ∈ *S*) ∧ (∀ *a*∈*S*. ∀ *b*∈*S*. ∀ *c*∈*S*. *plus* (*plus a b*) *c* = *plus a* (*plus b c*))›
for *S* :: ‹*'rep set*› **and** *plus* :: ‹*'rep* ⇒ *'rep* ⇒ *'rep*›
definition ‹*WITH-TYPE-REL-semigroup-add r* = (*r* ===> *r* ===> *r*)›
for *r* :: ‹*'rep* ⇒ *'abs* ⇒ *bool*› **and** *rep-ops* :: ‹*'rep* ⇒ *'rep* ⇒ *'rep*› **and** *abs-ops* :: ‹*'abs* ⇒ *'abs* ⇒ *'abs*›

lemma *with-type-wellformed-semigroup-add*[*with-type-intros*]:
‹*with-type-wellformed WITH-TYPE-CLASS-semigroup-add S WITH-TYPE-REL-semigroup-add*›
⟨*proof*⟩

lemma *with-type-transfer-semigroup-add*:
assumes [*transfer-rule*]: ‹*bi-unique r*› ‹*right-total r*›
shows ‹(*WITH-TYPE-REL-semigroup-add r* ===> (⟷))
(*WITH-TYPE-CLASS-semigroup-add* (*Collect* (*Domainp r*))) *class.semigroup-add*›
⟨*proof*⟩

⟨*ML*⟩

4.1.1 Example

definition *carrier* :: ‹*int set*› **where** ‹*carrier* = {*0*,*1*,*2*}›
definition *carrier-plus* :: ‹*int* ⇒ *int* ⇒ *int*› **where** ‹*carrier-plus i j* = (*i* + *j*) *mod 3*›

lemma *carrier-nonempty*[*iff*]: ‹*carrier* ≠ {}›
⟨*proof*⟩

lemma *carrier-semigroup*[*with-type-intros*]: ‹*WITH-TYPE-CLASS-semigroup-add carrier carrier-plus*›
⟨*proof*⟩

This proof uses both properties of the specific carrier (existence of two different elements) and of semigroups in general (associativity)

lemma *example-semigroup*:
shows ‹*let* ′*t*::*semigroup-add* = *carrier with carrier-plus in* ∀ *x y*.
(*plus-t x y* = *plus-t y x* ∧ *plus-t x* (*plus-t x x*) = *plus-t* (*plus-t x x*) *x*)›
⟨*proof*⟩

Some hypothetical lemma where we use the existence of a commutative semigroup to derive that 2147483647 is prime. (The lemma is true since 2147483647 is prime, but otherwise this is completely fictional.)

lemma *artificial-lemma*: ‹(∃ *p* (*x*::-::*semigroup-add*) *y*. *p x y* = *p y x*) ⟹ *prime* (*2147483647* :: *nat*)›
⟨*proof*⟩

lemma *prime-2147483647*: ‹*prime* (*2147483647* :: *nat*)›
⟨*proof*⟩

4.2 Abelian groups (class with several parameters)

Here we do exactly the same as for semigroups, except that now we use an abelian group. This shows the additional subtleties that arise when a class has more than one parameter.

notation *rel-prod* (**infixr** ‹∗∗∗› *80*)

definition ‹*WITH-TYPE-CLASS-ab-group-add S* = (λ(*plus*,*zero*,*minus*,*uminus*). *zero* ∈ *S*
∧ (∀ *a*∈*S*. ∀ *b*∈*S*. *plus a b* ∈ *S*) ∧ (∀ *a*∈*S*. ∀ *b*∈*S*. *minus a b* ∈ *S*) ∧ (∀ *a*∈*S*. *uminus a* ∈ *S*)
∧ (∀ *a*∈*S*. ∀ *b*∈*S*. ∀ *c*∈*S*. *plus* (*plus a b*) *c*= *plus a* (*plus b c*)) ∧ (∀ *a*∈*S*. ∀ *b*∈*S*. *plus a b* = *plus b a*)
∧ (∀ *a*∈*S*. *plus zero a* = *a*) ∧ (∀ *a*∈*S*. *plus* (*uminus a*) *a* = *zero*) ∧ (∀ *a*∈*S*. ∀ *b*∈*S*. *minus a b* = *plus a* (*uminus b*)))›
for *S* :: ‹′*rep set*›
definition ‹*WITH-TYPE-REL-ab-group-add r* = (*r* ===> *r* ===> *r*) ∗∗∗ *r* ∗∗∗ (*r* ===> *r* ===> *r*) ∗∗∗ (*r* ===> *r*)›
for *r* :: ‹′*rep* ⇒ ′*abs* ⇒ *bool*› **and** *rep-ops* :: ‹′*rep* ⇒ ′*rep* ⇒ ′*rep*› **and** *abs-ops* :: ‹′*abs* ⇒ ′*abs* ⇒ ′*abs*›

lemma *with-type-wellformed-ab-group-add*[*with-type-intros*]:
‹*with-type-wellformed WITH-TYPE-CLASS-ab-group-add S WITH-TYPE-REL-ab-group-add*›
⟨*proof*⟩

lemma *with-type-transfer-ab-group-add*:
assumes [*transfer-rule*]: ‹*bi-unique r*› ‹*right-total r*›
shows ‹(*WITH-TYPE-REL-ab-group-add r* ===> (⟷))
(*WITH-TYPE-CLASS-ab-group-add* (*Collect* (*Domainp r*))) (λ(*p*,*z*,*m*,*u*). *class.ab-group-add p z m u*)›
⟨*proof*⟩

⟨*ML*⟩

4.2.1 Example

definition *carrier-group* **where** ‹*carrier-group* = (*carrier-plus*, *0*::*int*, (λ *i j*. (*i* − *j*) *mod 3*), (λ*i*. (− *i*) *mod 3*))›

lemma *carrier-ab-group-add*[*with-type-intros*]: ‹*WITH-TYPE-CLASS-ab-group-add carrier carrier-group*›
⟨*proof*⟩

declare [[*show-sorts*=*false*]]
lemma *example-ab-group*:
shows ‹*let 't*::*ab-group-add* = *carrier with carrier-group in* ∀ *x y*.
(*plus-t x y* = *plus-t y x* ∧ *plus-t x* (*plus-t x x*) = *plus-t* (*plus-t x x*) *x*)›
⟨*proof*⟩

lemma *artificial-lemma'*: ‹(∃ *p* (*x*::-::*group-add*) *y*. *p x y* = *p y x*) ⟹ *prime* (*2305843009213693951* :: *nat*)›
⟨*proof*⟩

lemma *prime-2305843009213693951*: ‹*prime* (*2305843009213693951* :: *nat*)›
⟨*proof*⟩

end

5 *Example-Euclidean-Space* – Example: compactness of the sphere

theory *Example-Euclidean-Space*
imports *With-Type HOL*−*Analysis.Euclidean-Space HOL*−*Analysis.Topology-Euclidean-Space*
begin

5.1 Setting up type class *finite* for *with-type*

definition ‹*WITH-TYPE-CLASS-finite S u* ⟷ *finite S*›
  for *S* :: ‹*'rep set*› **and** *u* :: *unit*
definition ‹*WITH-TYPE-REL-finite r* = (*rel-unit-itself* :: - ⇒ *'abs itself* ⇒ -)›
  for *r* :: ‹*'rep* ⇒ *'abs* ⇒ *bool*›

lemma [*with-type-intros*]: ‹*finite S* ⟹ *WITH-TYPE-CLASS-finite S x*›
  ⟨*proof*⟩

lemma *with-type-wellformed-finite*[*with-type-intros*]:
  ‹*with-type-wellformed WITH-TYPE-CLASS-finite S WITH-TYPE-REL-finite*›
  ⟨*proof*⟩

lemma *with-type-transfer-finite*:
  includes *lifting-syntax*
  fixes *r* :: ‹*'rep* ⇒ *'abs* ⇒ *bool*›
  assumes [*transfer-rule*]: ‹*bi-unique r*› ‹*right-total r*›
  shows ‹(*WITH-TYPE-REL-finite r* ===> (⟷))
      (*WITH-TYPE-CLASS-finite* (*Collect* (*Domainp r*))) *class.finite*›
  ⟨*proof*⟩

⟨*ML*⟩

5.2 Vector space over a given basis

'a vs-over is defined to be the vector space with an orthonormal basis enumerated by elements of *'a*, in other words $\mathbb{R}^{'a}$. We require *'a* to be finite.

typedef *'a vs-over* = ‹*UNIV* :: (*'a*::*finite*⇒*real*) *set*›
  ⟨*proof*⟩
setup-lifting *type-definition-vs-over*

instantiation *vs-over* :: (*finite*) *real-vector* **begin**
lift-definition *plus-vs-over* :: ‹*'a vs-over* ⇒ *'a vs-over* ⇒ *'a vs-over*› **is** ‹λ*x y a. x a* + *y a*›⟨*proof*⟩
lift-definition *minus-vs-over* :: ‹*'a vs-over* ⇒ *'a vs-over* ⇒ *'a vs-over*› **is** ‹λ*x y a. x a* − *y a*›⟨*proof*⟩
lift-definition *uminus-vs-over* :: ‹*'a vs-over* ⇒ *'a vs-over*› **is** ‹λ*x a.* − *x a*›⟨*proof*⟩
lift-definition *zero-vs-over* :: ‹*'a vs-over*› **is** ‹λ-. *0*›⟨*proof*⟩
lift-definition *scaleR-vs-over* :: ‹*real* ⇒ *'a vs-over* ⇒ *'a vs-over*› **is** ‹λ*r x a. r* ∗ *x a*›⟨*proof*⟩
instance
  ⟨*proof*⟩
end

instantiation *vs-over* :: (*finite*) *real-normed-vector* **begin**
lift-definition *norm-vs-over* :: ‹*'a vs-over* ⇒ *real*› **is** ‹λ*x. L2-set x UNIV*›⟨*proof*⟩
definition *dist-vs-over* :: ‹*'a vs-over* ⇒ *'a vs-over* ⇒ *real*› **where** ‹*dist-vs-over x y* = *norm* (*x* − *y*)›

definition *uniformity-vs-over* :: ‹('a vs-over × 'a vs-over) filter› **where** ‹*uniformity-vs-over* = (*INF* e∈{0<..}. *principal* {(x, y). *dist* x y < e})›
definition *sgn-vs-over* :: ‹'a vs-over ⇒ 'a vs-over› **where** ‹*sgn-vs-over* x = x $/_R$ *norm* x›
definition *open-vs-over* :: ‹'a vs-over set ⇒ bool› **where** ‹*open-vs-over* U = (∀ x∈U. $\forall_F$ (x', y) *in uniformity*. x' = x ⟶ y ∈ U)›
instance
⟨*proof*⟩
end

instantiation *vs-over* :: (*finite*) *real-inner* **begin**
lift-definition *inner-vs-over* :: ‹'a vs-over ⇒ 'a vs-over ⇒ real› **is** ‹λx y. ∑ a∈UNIV. x a ∗ y a›⟨*proof*⟩
instance
⟨*proof*⟩
end

instantiation *vs-over* :: (*finite*) *euclidean-space* **begin**

Returns the basis vector corresponding to *'a*.

lift-definition *basis-vec* :: ‹'a ⇒ 'a vs-over› **is** ‹λa::'a. *indicator* {a}›⟨*proof*⟩
definition *Basis-vs-over* :: ‹'a vs-over set› **where** ‹*Basis* = *range basis-vec*›
instance
⟨*proof*⟩
end

5.3 Compactness of the sphere.

compact (*sphere ?a ?r*) shows that a sphere in an Euclidean vector space (type class *euclidean-space*) is compact. We wish to transfer this result to any space with a finite orthonormal basis. Mathematically, this is the same statement, but the conversion between a statement based on type classes and one based on predicates about bases is non-trivial in Isabelle.

lemma *compact-sphere-onb*:
fixes B :: ‹'a::real-inner set›
assumes ‹*finite* B› **and** ‹*span* B = UNIV› **and** *onb*: ‹∀ b∈B. ∀ c∈B. *inner* b c = *of-bool* (b=c)›
shows ‹*compact* (*sphere* (0::'a) r)›
⟨*proof*⟩

end

References

[1] O. Kunar and A. Popescu. From types to sets by local type definition in higher-order logic. *Journal of Automated Reasoning*, 62(2):237260, June 2018.