With-Type – Poor man's dependent types

Dominique Unruh
RWTH Aachen, University of Tartu

September 5, 2024

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

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*›
  by (*smt* (*verit*, *best*) *UNIV-I bij-betw-def bij-betw-iff-bijections inj-on-def inv-f-eq type-definition.Rep-inject type-definition.Rep-range type-definition.intro*)

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*›
  by (*simp add*: *Domainp-iff rel-unit-itself.simps*)
lemma *rel-unit-itself-iff*[*simp*]: ‹*rel-unit-itself x y* ⟷ (*y* = *TYPE*(*'a*))›
  by (*simp add*: *rel-unit-itself.simps*)

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* ⟷ (∀ *r rp. bi-unique r* ⟶ *right-total r* ⟶ *S* = *Collect* (*Domainp r*) ⟶ *C S rp* ⟶ *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* ∧ (∀ *rep abs-ops. bij-betw rep UNIV S* ⟶ (*R* (λ*x y. x* = *rep y*) *rep-ops abs-ops*) ⟶ *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*›
by (*simp add*: *WITH-TYPE-CLASS-type-def*)

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*›
by (*simp add*: *WITH-TYPE-REL-type-def WITH-TYPE-CLASS-type-def with-type-wellformed-def Domainp-iff*)

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.
by (*auto simp*: *with-type-def WITH-TYPE-REL-type-def WITH-TYPE-CLASS-type-def with-type-wellformed-def*)

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*›
using *assms*
by (*auto intro*!: *simp*: *with-type-def*)

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*›
using *assms* **by** (*auto simp add*: *with-type-def case-prod-beta type-definition-bij-betw-iff*)

lemma *with-type-nonempty*: ‹*with-type C R S p P* ⟹ *S* ≠ {}›
by (*simp add*: *with-type-def case-prod-beta*)

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 –
from ex
obtain rep :: ‹′abs ⇒ ′rep› **and** abs **where** td: ‹type-definition rep abs S›
by auto
then have bij: ‹bij-betw rep UNIV S›
by (simp add: bij-betw-def inj-on-def type-definition.Rep-inject type-definition.Rep-range)
define r **where** ‹r = (λx y. x = rep y)›
have [simp]: ‹bi-unique r› ‹right-total r›
using r-def td typedef-bi-unique **apply** blast
by (simp add: r-def right-totalI)
have aux1: ‹(⋀x. rep x ∈ S) ⟹ x ∈ S ⟹ x = rep (abs x)› **for** x b
using td type-definition.Abs-inverse **by** fastforce
have Sr: ‹S = Collect (Domainp r)›
using type-definition.Rep[OF td]
by (auto simp: r-def intro!: DomainPI aux1)
from wt **have** nice: ‹with-type-wellformed C S R› **and** ‹C S p›
by (simp-all add: with-type-def case-prod-beta)
from nice[unfolded with-type-wellformed-def, rule-format, OF ‹bi-unique r› ‹right-total r› Sr ‹C S p›]
obtain abs-ops **where** abs-ops: ‹R (λx y. x = rep y) p abs-ops›
apply atomize-elim **by** (auto simp: r-def)
from bij abs-ops wt
show P
by (auto simp: with-type-def case-prod-beta)
qed

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 –
from ex **obtain** Rep :: ‹′abs2⇒′rep› **and** Abs **where** td: ‹type-definition Rep Abs S›
by auto
define r **where** ‹r x y = (x = Rep y)› **for** x y
have bi-unique-r: ‹bi-unique r›
using bi-unique-def td type-definition.Rep-inject r-def **by** fastforce
have right-total-r: ‹right-total r›
by (simp add: right-totalI r-def)
have right-total-R[transfer-rule]: ‹right-total (r ===> r ===> r)›
by (meson bi-unique-r right-total-r bi-unique-alt-def right-total-fun)

note *trans* = *trans*[*OF bi-unique-r*, *OF right-total-r*, *transfer-rule*]

from *td*
have *rS*: ‹*Collect* (*Domainp r*) = *S*›
by (*auto simp*: *r-def Domainp-iff type-definition.Rep elim*!: *type-definition.Rep-cases*[**where** *P*=‹*Ex* -›])

from *wt* **have** *sg*: ‹*C S p*›
by (*simp-all add*: *with-type-def case-prod-beta*)

with *nice* **have** ‹*Domainp* (*R2 r*) *p*›
by (*simp add*: *bi-unique-r with-type-wellformed-def rS right-total-r*)

with *sg*
have ‹∃ *x* :: *'abs-ops2*. *axioms x*›
apply (*transfer' fixing*: *R2 S p*)
using *apply-rsp' local.trans rS* **by** *fastforce*

then obtain *abs-plus* :: *'abs-ops2*
where *abs-plus*: ‹*axioms abs-plus*›
apply *atomize-elim* **by** *auto*

then show *?thesis*
by *auto*
qed

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 −
from *ex* **obtain** *Rep* :: ‹*'abs2*⇒*'rep*› **and** *Abs* **where** *td*: ‹*type-definition Rep Abs S*›
by *auto*
define *r* **where** ‹*r x y* = (*x* = *Rep y*)› **for** *x y*
have *bi-unique-r*: ‹*bi-unique r*›
using *bi-unique-def td type-definition.Rep-inject r-def* **by** *fastforce*
have *right-total-r*: ‹*right-total r*›
by (*simp add*: *right-totalI r-def*)

have *right-total-R*[*transfer-rule*]: ‹*right-total* (*r* ===> *r* ===> *r*)›
  by (*meson bi-unique-r right-total-r bi-unique-alt-def right-total-fun*)

from *td*
have *rS*: ‹*Collect* (*Domainp r*) = *S*›
  by (*auto simp*: *r-def Domainp-iff type-definition.Rep elim*!: *type-definition.Rep-cases*[**where** *P*=‹*Ex* -›])

note *trans* = *trans*[*OF bi-unique-r*, *OF right-total-r*, *unfolded rS*, *transfer-rule*]

note *rel-itself* = *rel-itself*[*OF bi-unique-r*, *OF right-total-r*, *of p*, *transfer-rule*]

from *wt* **have** *sg*: ‹*C S p*›
  by (*simp-all add*: *with-type-def case-prod-beta*)
then show ‹*axioms TYPE*(*'abs2*)›
  by *transfer*
qed

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-file *with-type.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.

setup ‹
With-Type.add-with-type-info-global {
  class = ***class*** ‹*type*›,
  rep-class = ***const-name*** ‹*WITH-TYPE-CLASS-type*›,
  rep-rel = ***const-name*** ‹*WITH-TYPE-REL-type*›,
  with-type-wellformed = @{*thm with-type-wellformed-type*},
  param-names = [],
  transfer = *NONE*,
  rep-rel-itself = *NONE*
}›

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*]].

parse-translation ‹[
  (***syntax-const*** ‹*-with-type*›, *With-Type.with-type-parse-translation*),
  (***syntax-const*** ‹*-with-type-with*›, *With-Type.with-type-parse-translation*)

]›
typed-print-translation ‹[(***const-syntax***‹*with-type*›, *With-Type.with-type-print-translation*)
]›

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*.

attribute-setup *cancel-with-type* =
‹*Thm.rule-attribute* [] (*With-Type.cancel-with-type o Context.proof-of*) |> *Scan.succeed*›
‹*Transforms* (*let* ′*t*=... *in P*) *into P*›

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.

```
method-setup with-type-mp = ‹Scan.succeed () >>
  (fn (-) => fn ctxt => CONTEXT-METHOD (fn facts =>
      Method.RUNTIME (With-Type.with-type-mp-tac here facts)))›
```

```
ML ‹
val - =
  Outer-Syntax.command command-keyword‹with-type-case› Sets up local proof after the method‹with-type-mp› method (for the main goal).
   (Scan.repeat (Parse.maybe Parse.binding) >> (fn args => Toplevel.proof (With-Type.with-type-case-cmd args)))
›
```

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*
no-notation *m-inv* (*inv*ı - [*81*] *80*)

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*›
by (*simp add*: *with-type-wellformed-def WITH-TYPE-CLASS-semigroup-add-def WITH-TYPE-REL-semigroup-add*
fun.Domainp-rel Domainp-pred-fun-eq bi-unique-alt-def)

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 −
define *f* **where** ‹*f y* = (*SOME x. r x y*)› **for** *y*
have *rf*: ‹*r x y* ⟷ *x* = *f y*› **for** *x y*
unfolding *f-def*
apply (*rule someI2-ex*)
using *assms*
by (*auto intro*!: *simp*: *right-total-def bi-unique-def*)
have *inj*: ‹*inj f*›
using ‹*bi-unique r*›
by (*auto intro*!: *injI simp*: *bi-unique-def rf*)
have *aux1*: ‹∀ *ya yb. x* (*f ya*) (*f yb*) = *f* (*y ya yb*) ⟹
∀ *a*. (∃ *y*. *a* = *f y*) ⟶ (∀ *b*. (∃ *y*. *b* = *f y*) ⟶ (∀ *c*. (∃ *y*. *c* = *f y*) ⟶ *x* (*x a b*) *c* = *x a* (*x b c*))) ⟹
y (*y a b*) *c* = *y a* (*y b c*)› **for** *x y a b c*
by (*metis inj injD*)
show *?thesis*
unfolding *WITH-TYPE-REL-semigroup-add-def rel-fun-def*
unfolding *WITH-TYPE-CLASS-semigroup-add-def Domainp-iff rf*
by (*auto intro*!: *simp*: *class.semigroup-add-def aux1*)
qed

setup ‹
With-Type.add-with-type-info-global {
class = ***class*** ‹*semigroup-add*›,
param-names = [*plus*],
rep-class = ***const-name*** ‹*WITH-TYPE-CLASS-semigroup-add*›,
rep-rel = ***const-name*** ‹*WITH-TYPE-REL-semigroup-add*›,
with-type-wellformed = @{*thm with-type-wellformed-semigroup-add*},
transfer = *SOME* @{*thm with-type-transfer-semigroup-add*},
rep-rel-itself = *NONE*
}
›

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 ≠ {}›
  by (simp add: carrier-def)

lemma carrier-semigroup[with-type-intros]: ‹WITH-TYPE-CLASS-semigroup-add carrier carrier-plus›
  by (auto simp: WITH-TYPE-CLASS-semigroup-add-def
        carrier-def carrier-plus-def)
```

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 (with-type-intro)
  show ‹carrier ≠ {}› by simp
  fix Rep :: ‹'t ⇒ int› and T and plus-t
  assume ‹bij-betw Rep UNIV carrier›
  then interpret type-definition Rep ‹inv Rep› carrier
    using type-definition-bij-betw-iff by blast
  define r where ‹r = (λx y. x = Rep y)›
  have [transfer-rule]: ‹bi-unique r›
    by (simp add: Rep-inject bi-unique-def r-def)
  have [transfer-rule]: ‹right-total r›
    by (simp add: r-def right-total-def)
  assume ‹WITH-TYPE-REL-semigroup-add (λx y. x = Rep y) carrier-plus plus-t›
  then have transfer-carrier[transfer-rule]: ‹(r ===> r ===> r) carrier-plus plus-t›
    by (simp add: r-def WITH-TYPE-REL-semigroup-add-def)
  have [transfer-rule]: ‹((r ===> r ===> r) ===> (⟷)) (WITH-TYPE-CLASS-semigroup-add (Collect (Domainp r))) class.semigroup-add›
  proof (intro rel-funI)
    fix x y
    assume xy[transfer-rule]: ‹(r ===> r ===> r) x y›
    have aux1: ‹(∀ a. Domainp r a ⟶ (∀ b. Domainp r b ⟶ (∀ c. Domainp r c ⟶ x (x a b) c = x a (x b c))) ⟹
        r a b ⟹ r ba bb ⟹ r c bc ⟹ x (x a ba) c = x a (x ba c)› for a b ba bb c bc
      by blast
    have aux2: ‹r a b ⟹ r ba bb ⟹ Domainp r (x a ba)› for a b ba bb
      by (metis DomainPI rel-funD xy)
    show ‹WITH-TYPE-CLASS-semigroup-add (Collect (Domainp r)) x ⟷ class.semigroup-add y›
      unfolding class.semigroup-add-def
      apply transfer
      by (auto intro!: aux1 aux2 simp: WITH-TYPE-CLASS-semigroup-add-def)
  qed
```

```
  have dom-r: ‹Collect (Domainp r) = carrier›
    by (auto elim!: Rep-cases simp: Rep r-def Domainp-iff)
  interpret semigroup-add plus-t
    apply transfer
    using carrier-semigroup dom-r by auto

  have 1: ‹plus-t x y = plus-t y x› for x y
    apply transfer
    apply (simp add: carrier-plus-def)
    by presburger

  have 2: ‹plus-t x (plus-t x x) = plus-t (plus-t x x) x› for x
    by (simp add: add-assoc)

  from 1 2
  show ‹∀ x y. plus-t x y = plus-t y x ∧ plus-t x (plus-t x x) = plus-t (plus-t x x) x›
    by simp
qed
```

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 — — This proof can be ignored. We just didn't want to have a "sorry" in the theory file
  have prime (2 ^ 31 − 1 :: nat)
    by (subst lucas-lehmer-correct') eval
  also have ‹... = 2147483647›
    by eval
  finally show ‹prime (2147483647 :: nat)›
    by —
qed

lemma prime-2147483647: ‹prime (2147483647 :: nat)›
proof —
  from example-semigroup
  have ‹let 't::semigroup-add = carrier with carrier-plus in
        prime (2147483647 :: nat)›
  proof with-type-mp
    with-type-case
    show ‹prime (2147483647 :: nat)›
      apply (rule artificial-lemma)
      using with-type-mp.premise by auto
  qed
  from this[cancel-with-type]
  show ?thesis
    by —
qed
```

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*›
  by (*simp add*: *with-type-wellformed-def WITH-TYPE-CLASS-ab-group-add-def WITH-TYPE-REL-ab-group-add-de*
    fun.Domainp-rel Domainp-pred-fun-eq bi-unique-alt-def prod.Domainp-rel DomainPI)

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 −
  define *f* **where** ‹*f y* = (*SOME x*. *r x y*)› **for** *y*
  have *rf*: ‹*r x y* ⟷ *x* = *f y*› **for** *x y*
    unfolding *f-def*
    apply (*rule someI2-ex*)
    using *assms*
    by (*auto intro*!: *simp*: *right-total-def bi-unique-def*)
  have *inj*: ‹*inj f*›
    using ‹*bi-unique r*›
    by (*auto intro*!: *injI simp*: *bi-unique-def rf*)
  show *?thesis*
    unfolding *WITH-TYPE-REL-ab-group-add-def rel-fun-def*
    unfolding *WITH-TYPE-CLASS-ab-group-add-def*
    unfolding *Domainp-iff rf*
    using *inj*
    apply (*simp add*: *class.ab-group-add-def class.comm-monoid-add-def*
      class.ab-semigroup-add-def class.semigroup-add-def class.ab-semigroup-add-axioms-def
      class.comm-monoid-add-axioms-def class.ab-group-add-axioms-def inj-def)
    apply *safe*

by *metis+*
qed

setup ‹
With-Type.add-with-type-info-global {
class = ***class*** ‹*ab-group-add*›,
param-names = [*plus*, *zero*, *minus*, *uminus*],
rep-class = ***const-name*** ‹*WITH-TYPE-CLASS-ab-group-add*›,
rep-rel = ***const-name*** ‹*WITH-TYPE-REL-ab-group-add*›,
with-type-wellformed = @{*thm with-type-wellformed-ab-group-add*},
transfer = *SOME* @{*thm with-type-transfer-ab-group-add*},
rep-rel-itself = *NONE*
}
›

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*›
by (*auto simp*: *WITH-TYPE-CLASS-ab-group-add-def carrier-plus-def*
carrier-def carrier-group-def)

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 *with-type-intro*
show ‹*carrier* ≠ {}› **by** *simp*
fix *Rep* :: ‹′*t* ⇒ *int*› **and** *t-ops*
assume *wt*: ‹*WITH-TYPE-REL-ab-group-add* (λ*x y*. *x* = *Rep y*) *carrier-group t-ops*›
define *plus zero minus uminus* **where** ‹*plus* = *fst t-ops*›
and ‹*zero* = *fst* (*snd t-ops*)›
and ‹*minus* = *fst* (*snd* (*snd t-ops*))›
and ‹*uminus* = *snd* (*snd* (*snd t-ops*))›

assume ‹*bij-betw Rep UNIV carrier*›
then interpret *type-definition Rep* ‹*inv Rep*› *carrier*
by (*simp add*: *type-definition-bij-betw-iff*)

define *r* **where** ‹*r* = (λ*x y*. *x* = *Rep y*)›
have [*transfer-rule*]: ‹*bi-unique r*›
by (*simp add*: *Rep-inject bi-unique-def r-def*)
have [*transfer-rule*]: ‹*right-total r*›
by (*simp add*: *r-def right-total-def*)
from *wt* **have** *transfer-carrier*[*transfer-rule*]: ‹((*r* ===> *r* ===> *r*) *** *r* *** (*r* ===> *r*

===> r) *** (r ===> r)) *carrier-group t-ops*›
by (*simp add*: *r-def WITH-TYPE-REL-ab-group-add-def*)
have *transfer-plus*[*transfer-rule*]: ‹(r ===> r ===> r) *carrier-plus plus*›
apply (*subst asm-rl*[*of* ‹*carrier-plus* = *fst* (*carrier-group*)›])
apply (*simp add*: *carrier-group-def*)
unfolding *plus-def*
by *transfer-prover*
have *dom-r*: ‹*Collect* (*Domainp r*) = *carrier*›
by (*auto elim*!: *Rep-cases simp*: *Rep r-def Domainp-iff*)

from *with-type-transfer-ab-group-add*[*OF* ‹*bi-unique r*› ‹*right-total r*›]
have [*transfer-rule*]: ‹((r ===> r ===> r) ===> r ===> (r ===> r ===> r) ===> (r ===> r) ===> (⟷))
(λp z m u. *WITH-TYPE-CLASS-ab-group-add carrier* (p,z,m,u)) *class.ab-group-add*›
unfolding *dom-r WITH-TYPE-REL-ab-group-add-def*
by (*auto intro*!: *simp*: *rel-fun-def*)

interpret *ab-group-add plus zero minus uminus*
unfolding *zero-def plus-def minus-def uminus-def*
apply *transfer*
using *carrier-ab-group-add dom-r*
by (*auto intro*!: *simp*: *Let-def case-prod-beta*)

have *1*: ‹*plus x y* = *plus y x*› **for** *x y*
— We could prove this simply with *by* (*simp add*: *add-commute*), but we use the approach of going to the concrete type for demonstration.
apply *transfer*
apply (*simp add*: *carrier-plus-def*)
by *presburger*

have *2*: ‹*plus x* (*plus x x*) = *plus* (*plus x x*) *x*› **for** *x*
by (*simp add*: *add-assoc*)

from *1 2*
show ‹∀ x y. *plus x y* = *plus y x* ∧ *plus x* (*plus x x*) = *plus* (*plus x x*) *x*›
by (*simp add*: *plus-def case-prod-beta*)
qed

lemma *artificial-lemma'*: ‹(∃ p (x::-::*group-add*) y. p x y = p y x) ⟹ *prime* (*2305843009213693951* :: *nat*)›
proof − — This proof can be ignored. We just didn't want to have a "sorry" in the theory file
have *prime* (*2* ^ *61* − *1* :: *nat*)
by (*subst lucas-lehmer-correct'*) *eval*
also have ‹. . . = *2305843009213693951*›
by *eval*
finally show ‹*prime* (*2305843009213693951* :: *nat*)›
by −
qed

lemma *prime-2305843009213693951*: ‹*prime* (*2305843009213693951* :: *nat*)›
proof −
  from *example-ab-group*
  have ‹*let* ′*t*::*ab-group-add* = *carrier with carrier-group in prime* (*2305843009213693951* :: *nat*)›
  proof *with-type-mp*
    with-type-case
    show ‹*prime* (*2305843009213693951* :: *nat*)›
      apply (*rule artificial-lemma*′)
      using *with-type-mp.premise* **by** *auto*
  qed
  from *this*[*cancel-with-type*]
  show *?thesis*
    by −
qed

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*›
  using *WITH-TYPE-CLASS-finite-def* **by** *blast*

lemma *with-type-wellformed-finite*[*with-type-intros*]:
  ‹*with-type-wellformed WITH-TYPE-CLASS-finite S WITH-TYPE-REL-finite*›
  by (*simp add*: *with-type-wellformed-def WITH-TYPE-REL-finite-def*)

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*›
  unfolding *WITH-TYPE-REL-finite-def*

```
proof (rule rel-funI)
  fix x :: unit and y :: ‹'abs itself›
  assume ‹rel-unit-itself x y›
  then have [simp]: ‹y = TYPE('abs)›
    by simp
  note right-total-UNIV-transfer[transfer-rule]
  show ‹WITH-TYPE-CLASS-finite (Collect (Domainp r)) x ⟷ class.finite y›
    apply (simp add: WITH-TYPE-CLASS-finite-def class.finite-def)
    apply transfer
    by simp
qed

setup ‹
With-Type.add-with-type-info-global {
  class = class ‹finite›,
  param-names = [],
  rep-class = const-name ‹WITH-TYPE-CLASS-finite›,
  rep-rel = const-name ‹WITH-TYPE-REL-finite›,
  with-type-wellformed = @{thm with-type-wellformed-finite},
  transfer = SOME @{thm with-type-transfer-finite},
  rep-rel-itself = SOME @{lemma ‹bi-unique r ⟹ right-total r ⟹ (WITH-TYPE-REL-finite
r) p TYPE('abs2)›
      by (simp add: WITH-TYPE-REL-finite-def rel-unit-itself.simps Transfer.Rel-def)}
}
›
```

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›
  by (rule exI[of - ‹λ-. 0›], auto)
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 ‹λx y a. x a + y a›.
lift-definition minus-vs-over :: ‹'a vs-over ⇒ 'a vs-over ⇒ 'a vs-over› is ‹λx y a. x a − y a›.
lift-definition uminus-vs-over :: ‹'a vs-over ⇒ 'a vs-over› is ‹λx a. − x a›.
lift-definition zero-vs-over :: ‹'a vs-over› is ‹λ-. 0›.
lift-definition scaleR-vs-over :: ‹real ⇒ 'a vs-over ⇒ 'a vs-over› is ‹λr x a. r ∗ x a›.
instance
  apply (intro-classes; transfer)
  by (auto intro!: ext simp: distrib-right distrib-left)
end

instantiation vs-over :: (finite) real-normed-vector begin
lift-definition norm-vs-over :: ‹'a vs-over ⇒ real› is ‹λx. L2-set x UNIV›.
```

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 *intro-classes*
  fix *x y* :: ‹′*a vs-over*›
  show ‹*dist x y* = *norm* (*x* − *y*)›
  using *dist-vs-over-def* **by** *presburger*
  show ‹*sgn x* = *x* $/_R$ *norm x*›
  using *sgn-vs-over-def* **by** *blast*
  show ‹(*uniformity* :: (′*a vs-over* × ′*a vs-over*) *filter*) = (*INF e*∈{*0*<..}. *principal* {(*x*, *y*). *dist x y* < *e*})›
  using *uniformity-vs-over-def* **by** *blast*
  show ‹(*norm x* = *0*) = (*x* = *0*)›
    apply *transfer*
    by (*auto simp*: *L2-set-eq-0-iff*)
  show ‹*norm* (*x* + *y*) ≤ *norm x* + *norm y*›
    apply *transfer*
    by (*rule L2-set-triangle-ineq*)
  show ‹*norm* (*a* $*_R$ *x*) = |*a*| ∗ *norm x*› **for** *a*
    apply *transfer*
    by (*simp add*: *L2-set-def power-mult-distrib real-sqrt-mult flip*: *sum-distrib-left*)
  show ‹*open U* = (∀ *x*∈*U*. $\forall_F$ (*x*′, *y*) *in uniformity*. *x*′ = *x* ⟶ *y* ∈ *U*)› **for** *U* :: ‹′*a vs-over set*›
    by (*simp add*: *open-vs-over-def*)
qed
end

instantiation *vs-over* :: (*finite*) *real-inner* **begin**
lift-definition *inner-vs-over* :: ‹′*a vs-over* ⇒ ′*a vs-over* ⇒ *real*› **is** ‹λ*x y*. ∑ *a*∈*UNIV*. *x a* ∗ *y a*›**.**
instance
  apply (*intro-classes*; *transfer*)
  by (*auto intro*!: *sum-nonneg sum.cong simp*: *mult.commute sum-nonneg-eq-0-iff L2-set-def power2-eq-square sum-distrib-left mult-ac distrib-left simp flip*: *sum.distrib*)
end

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

Returns the basis vector corresponding to ′*a*.

lift-definition *basis-vec* :: ‹′*a* ⇒ ′*a vs-over*› **is** ‹λ*a*::′*a*. *indicator* {*a*}›**.**
definition *Basis-vs-over* :: ‹′*a vs-over set*› **where** ‹*Basis* = *range basis-vec*›
instance
  apply (*intro-classes*; *unfold Basis-vs-over-def*; *transfer*)
  by (*auto intro*!: *simp*: *indicator-def*)

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 (cases ‹B = {}›)
  case True
  with assms have all-0: ‹(x :: 'a) = 0› for x
    by auto
  then have ‹sphere (0::'a) r = {0} ∨ sphere (0::'a) r = {}›
    by (auto simp add: sphere-def)
  then show ?thesis
    by fastforce
next
  case False
  have ‹let 't::finite = B in compact (sphere (0::'t vs-over) r)›
  proof with-type-intro
    from False show ‹B ≠ {}› by –
    from assms show ‹finite B› by –
    fix rep :: ‹'t ⇒ -›
    assume ‹bij-betw rep UNIV B›
    from compact-sphere[where 'a=‹'t vs-over›]
    show ‹compact (sphere (0::'t vs-over) r)›
      by simp
  qed
  then have ‹let 't::finite = B in compact (sphere (0::'a) r)›
  proof with-type-mp
    with-type-case

    define f :: ‹'t vs-over ⇒ 'a› where ‹f x = (∑ t∈UNIV. Rep-vs-over x t *_R rep-t t)› for x
    have ‹linear f›
      by (auto intro!: linearI sum.distrib simp: f-def plus-vs-over.rep-eq scaleR-vs-over.rep-eq
          scaleR-add-left scaleR-right.sum simp flip: scaleR-scaleR)
    then have ‹continuous-on X f› for X
      using linear-continuous-on linear-linear by blast
    moreover from with-type-mp.premise have ‹compact (sphere (0::'t vs-over) r)›
      by –
    ultimately have compact-fsphere: ‹compact (f ‘ sphere 0 r)›
      using compact-continuous-image by blast
```

```
    have ‹surj f›
    proof (unfold surj-def, rule allI)
      fix y :: 'a
      from assms have ‹y ∈ span B›
        by auto
      then show ‹∃ x. y = f x›
      proof (induction rule: span-induct-alt)
        case base
        then show ?case
          by (auto intro!: exI[of - 0] simp: f-def zero-vs-over.rep-eq)
      next
        case (step c b y)
        from step.IH
        obtain x where yfx: ‹y = f x›
          by auto
        have ‹b = f (basis-vec (inv rep-t b))›
          by (simp add: f-def basis-vec.rep-eq step.hyps type-definition-t.Abs-inverse)
        then have ‹c *R b + y = f (c *R basis-vec (inv rep-t b) + x)›
          using ‹linear f›
          by (simp add: linear-add linear-scale yfx)
        then show ?case
          by auto
      qed
    qed
    have ‹norm (f x) = norm x› for x
    proof −
      have aux1: ‹(a, b) ∉ range (λt. (t, t)) ⟹ rep-t a • rep-t b ≠ 0 ⟹ Rep-vs-over x b = 0›
for a b
      by (metis (mono-tags, lifting) of-bool-eq(1) onb range-eqI type-definition-t.Rep type-definition-t.Rep-inverse)
      have rep-inner: ‹inner (rep-t t) (rep-t u) = of-bool (t=u)› for t u
        by (simp add: onb type-definition-t.Rep type-definition-t.Rep-inject)
      have ‹(norm (f x))^2 = inner (f x) (f x)›
        by (simp add: dot-square-norm)
      also have ‹... = (∑ (t,t')∈UNIV. (Rep-vs-over x t * Rep-vs-over x t') * inner (rep-t t)
(rep-t t'))›
      by (auto intro!: simp: f-def inner-sum-right inner-sum-left sum-distrib-left sum.cartesian-product
          case-prod-beta inner-commute mult-ac)
      also have ‹... = (∑ (t,t')∈(λt. (t,t))`UNIV. (Rep-vs-over x t * Rep-vs-over x t') * inner
(rep-t t) (rep-t t'))›
        by (auto intro!: sum.mono-neutral-cong-right simp: aux1)
      also have ‹... = (∑ t∈UNIV. (Rep-vs-over x t)^2)›
        apply (subst sum.reindex)
        by (auto intro!: injI simp: rep-inner power2-eq-square)
      also have ‹... = (norm x)^2›
        by (simp add: norm-vs-over-def L2-set-def sum-nonneg)
      finally show ?thesis
        by simp
    qed
    then have ‹f ` sphere 0 r = sphere 0 r›
```

```
      using ‹surj f›
      by (fastforce simp: sphere-def)
    with compact-fsphere
    show ‹compact (sphere (0::'a) r)›
      by simp
  qed
  from this[cancel-with-type]
  show ‹compact (sphere (0::'a) r)›
    by −
qed

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.