Fixed-length vectors

Lars Hupel

February 6, 2026

Abstract

This theory introduces a type constructor for lists with known length, also known as *vectors*. Those vectors are indexed with a numeral type that represent their length. This can be employed to avoid carrying around length constraints on lists. Instead, those constraints are discharged by the type checker. As compared to the vectors defined in the distribution, this definition can easily work with unit vectors. We exploit the fact that the cardinality of an infinite type is defined to be 0: thus any infinite length index type represents a unit vector. Furthermore, we set up automation and BNF support.

Contents

1 Type class for indexing **1**

2 Type definition and BNF setup **4**

3 Indexing **5**

4 Unit vector **6**

5 General lemmas **6**

6 Instances **8**

7 Further operations **9**
7.1 Distinctness 9
7.2 Summing 10

8 Code setup **10**

theory *Fixed-Length-Vector*
imports *HOL−Library.Numeral-Type HOL−Library.Code-Cardinality*
begin

lemma *zip-map-same*: ‹*zip* (*map f xs*) (*map g xs*) = *map* (λ*x*. (*f x*, *g x*)) *xs*›
⟨*proof*⟩

1 Type class for indexing

The *index* class is used to define an isomorphism between some index types with fixed cardinality and a subset of the natural numbers. Crucially, infinite types can be made instance of this class too, since Isabelle defines infinite cardinality to be equal to zero.

The *index1* class then adds more properties, such as injectivity, which can only be satisfied for finite index types.

This class is roughly similar to the *enum* class defined in the Isabelle library, which is proved at a later point. However, *enum* does not admit infinite types.

class *index* =
  fixes *from-index* :: ‹'*a* ⇒ *nat*›
    and *to-index* :: ‹*nat* ⇒ '*a*›
  assumes *to-from-index*: ‹*n* < *CARD*('*a*) ⟹ *from-index* (*to-index n*) = *n*›
  assumes *from-index-surj*: ‹*n* < *CARD*('*a*) ⟹ ∃ *a*. *from-index a* = *n*›
begin

A list of all possible indexes.

definition *indexes* :: ‹'*a list*› **where**
‹*indexes* = *map to-index* [*0*..<*CARD*('*a*)]›

There are as many indexes as the cardinality of type '*a*.

lemma *indexes-length*[*simp*]: ‹*length indexes* = *CARD*('*a*)›
  ⟨*proof*⟩

lemma *list-cong-index*:
  assumes ‹*length xs* = *CARD*('*a*)› ‹*length ys* = *CARD*('*a*)›
  assumes ‹⋀*i*. *xs* ! *from-index i* = *ys* ! *from-index i*›
  shows ‹*xs* = *ys*›
  ⟨*proof*⟩

lemma *from-indexE*:
  assumes ‹*n* < *CARD*('*a*)›
  obtains *a* **where** ‹*from-index a* = *n*›
  ⟨*proof*⟩

end

Restrict *index* to only admit finite types.

class *index1* = *index* +
  assumes *from-index-bound*[*simp*]: ‹*from-index a* < *CARD*('*a*)›
  assumes *from-index-inj*: ‹*inj from-index*›
begin

Finiteness follows from the class assumptions.

lemma *card-nonzero*[*simp*]: ‹*0* < *CARD*('*a*)›

⟨*proof*⟩

lemma *finite-type*[*simp*]: ‹*finite* (*UNIV* :: *'a set*)›
⟨*proof*⟩

sublocale *finite*
⟨*proof*⟩

to-index and *from-index* form a bijection.

lemma *from-to-index*[*simp*]: ‹*to-index* (*from-index i*) = *i*›
⟨*proof*⟩

lemma *indexes-from-index*[*simp*]: ‹*indexes* ! *from-index i* = *i*›
⟨*proof*⟩

lemma *to-index-inj-on*: ‹*inj-on to-index* {*0*..<*CARD*('*a*)}›
⟨*proof*⟩

end

Finally, we instantiate the class for the pre-defined numeral types.

instantiation *num0* :: *index*
begin

definition *from-index-num0* :: ‹*num0* ⇒ *nat*› **where** ‹*from-index-num0 - = undefined*›
definition *to-index-num0* :: ‹*nat* ⇒ *num0*› **where** ‹*to-index-num0 - = undefined*›

instance
⟨*proof*⟩

end

lemma *indexes-zero*[*simp*]: ‹*indexes* = ([] :: *0 list*)›
⟨*proof*⟩

instantiation *num1* :: *index1*
begin

definition *from-index-num1* :: ‹*num1* ⇒ *nat*› **where** [*simp*]: ‹*from-index-num1 - = 0*›
definition *to-index-num1* :: ‹*nat* ⇒ *num1*› **where** [*simp*]: ‹*to-index-num1 - = 1*›

instance
⟨*proof*⟩

end

lemma *indexes-one*[*simp*]: ‹*indexes* = [*1* :: *1*]›
⟨*proof*⟩

instantiation *bit0* :: (*finite*) *index1*
begin

definition *from-index-bit0* :: ‹*'a bit0* ⇒ *nat*› **where** ‹*from-index-bit0 x* = *nat* (*Rep-bit0 x*)›

definition *to-index-bit0* :: ‹*nat* ⇒ *'a bit0*› **where** ‹*to-index-bit0* ≡ *of-nat*›

instance
⟨*proof*⟩

end

instantiation *bit1* :: (*finite*) *index1*
begin

definition *from-index-bit1* :: ‹*'a bit1* ⇒ *nat*› **where** ‹*from-index-bit1 x* = *nat* (*Rep-bit1 x*)›

definition *to-index-bit1* :: ‹*nat* ⇒ *'a bit1*› **where** ‹*to-index-bit1* ≡ *of-nat*›

instance
⟨*proof*⟩

end

lemma *indexes-bit-simps*:
‹(*indexes* :: *'a* :: *finite bit0 list*) = *map of-nat* [*0*..<*2* ∗ *CARD*(*'a*)]›
‹(*indexes* :: *'b* :: *finite bit1 list*) = *map of-nat* [*0*..<*2* ∗ *CARD*(*'b*) + *1*]›
⟨*proof*⟩

The following class and instance definitions connect *indexes* to *enum-class.enum*.

class *index-enum* = *index1* + *enum* +
assumes *indexes-eq-enum*: ‹*indexes* = *enum-class.enum*›

instance *num1* :: *index-enum*
⟨*proof*⟩

instance *bit0* :: (*finite*) *index-enum*
⟨*proof*⟩

instance *bit1* :: (*finite*) *index-enum*
⟨*proof*⟩

2 Type definition and BNF setup

A vector is a list with a fixed length, where the length is given by the cardinality of the second type parameter. To obtain the unit vector, we can choose an infinite type. There is no reason to restrict the index type to a particular sort constraint at this point, even though later on, *index* is frequently used.

typedef $('a, 'b)$ *vec* $= \{xs.\ length\ xs = CARD('b)\} :: 'a\ list\ set$
  morphisms *list-of-vec vec-of-list*
  ⟨*proof*⟩

declare *vec.list-of-vec-inverse*[*simp*]

type-notation *vec* (**infixl** ‹^› *15*)

setup-lifting *type-definition-vec*

lift-definition *map-vec* :: $\langle('a \Rightarrow 'b) \Rightarrow 'a\ \hat{}\ 'c \Rightarrow 'b\ \hat{}\ 'c\rangle$ **is** *map* ⟨*proof*⟩

lift-definition *set-vec* :: $\langle 'a\ \hat{}\ 'b \Rightarrow 'a\ set\rangle$ **is** *set* ⟨*proof*⟩

lift-definition *rel-vec* :: $\langle('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ \hat{}\ 'c \Rightarrow 'b\ \hat{}\ 'c \Rightarrow bool\rangle$ **is** *list-all2* ⟨*proof*⟩

lift-definition *pred-vec* :: $\langle('a \Rightarrow bool) \Rightarrow 'a\ \hat{}\ 'b \Rightarrow bool\rangle$ **is** *list-all* ⟨*proof*⟩

lift-definition *zip-vec* :: $\langle 'a\ \hat{}\ 'c \Rightarrow 'b\ \hat{}\ 'c \Rightarrow ('a \times 'b)\ \hat{}\ 'c\rangle$ **is** *zip* ⟨*proof*⟩

lift-definition *replicate-vec* :: $\langle 'a \Rightarrow 'a\ \hat{}\ 'b\rangle$ **is** ‹*replicate CARD('b)*› ⟨*proof*⟩

bnf $\langle('a, 'b)\ vec\rangle$
  map: *map-vec*
  sets: *set-vec*
  bd: *natLeq*
  wits: *replicate-vec*
  rel: *rel-vec*
  pred: *pred-vec*
  ⟨*proof*⟩

3 Indexing

lift-definition *nth-vec'* :: $\langle 'a\ \hat{}\ 'b \Rightarrow nat \Rightarrow 'a\rangle$ **is** *nth* ⟨*proof*⟩

lift-definition *nth-vec* :: $\langle 'a\ \hat{}\ 'b \Rightarrow 'b :: index1 \Rightarrow 'a\rangle$ (**infixl** ‹$› *90*)
  — We fix this to *index1* because indexing a unit vector makes no sense.
  is ‹*λxs. nth xs ∘ from-index*› ⟨*proof*⟩

lemma *nth-vec-alt-def*: ‹*nth-vec v = nth-vec' v ∘ from-index*›

⟨*proof*⟩

We additionally define a notion of converting a function into a vector, by mapping over all *indexes*.

lift-definition *vec-lambda* :: ‹('*b* :: *index* ⇒ '*a*) ⇒ '*a* ^ '*b*› (**binder** ‹χ› *10*)
  is ‹λ*f*. *map f indexes*› ⟨*proof*⟩

lemma *vec-lambda-nth*[*simp*]: ‹*vec-lambda f* $ *i* = *f i*›
  ⟨*proof*⟩

4 Unit vector

The *unit vector* is the unique vector of length zero. We use *0* as index type, but *nat* or any other infinite type would work just as well.

lift-definition *unit-vec* :: ‹'*a* ^ *0*› **is** ‹[]› ⟨*proof*⟩

lemma *unit-vec-unique*: ‹*v* = *unit-vec*›
  ⟨*proof*⟩

lemma *unit-vec-unique-simp*[*simp*]: ‹*NO-MATCH v unit-vec* ⟹ *v* = *unit-vec*›
  ⟨*proof*⟩

lemma *set-unit-vec*[*simp*]: ‹*set-vec* (*v* :: '*a* ^ *0*) = {}›
  ⟨*proof*⟩

lemma *map-unit-vec*[*simp*]: ‹*map-vec f v* = *unit-vec*›
  ⟨*proof*⟩

lemma *zip-unit-vec*[*simp*]: ‹*zip-vec u v* = *unit-vec*›
  ⟨*proof*⟩

lemma *rel-unit-vec*[*simp*]: ‹*rel-vec R* (*u* :: '*a* ^ *0*) *v* ⟷ *True*›
  ⟨*proof*⟩

lemma *pred-unit-vec*[*simp*]: ‹*pred-vec P* (*v* :: '*a* ^ *0*)›
  ⟨*proof*⟩

5 General lemmas

lemmas *vec-simps*[*simp*] =
  map-vec.rep-eq
  zip-vec.rep-eq
  replicate-vec.rep-eq

lemmas *map-vec-cong*[*fundef-cong*] = *map-cong*[*Transfer.transferred*]

lemmas *rel-vec-cong* = *list.rel-cong*[*Transfer.transferred*]

lemmas *pred-vec-cong* = *list.pred-cong*[*Transfer.transferred*]

lemma *vec-eq-if*: *list-of-vec f* = *list-of-vec g* ⟹ *f* = *g*
⟨*proof*⟩

lemma *vec-cong*: (⋀*i*. *f* $ *i* = *g* $ *i*) ⟹ *f* = *g*
⟨*proof*⟩

lemma *finite-set-vec*[*intro*, *simp*]: ‹*finite* (*set-vec v*)›
⟨*proof*⟩

lemma *set-vec-in*[*intro*, *simp*]: ‹*v* $ *i* ∈ *set-vec v*›
⟨*proof*⟩

lemma *set-vecE*[*elim*]:
assumes ‹*x* ∈ *set-vec v*›
obtains *i* **where** ‹*x* = *v* $ *i*›
⟨*proof*⟩

lemma *map-nth-vec*[*simp*]: ‹*map-vec f v* $ *i* = *f* (*v* $ *i*)›
⟨*proof*⟩

lemma *replicate-nth-vec*[*simp*]: ‹*replicate-vec a* $ *i* = *a*›
⟨*proof*⟩

lemma *replicate-set-vec*[*simp*]: ‹*set-vec* (*replicate-vec a* :: *'a* ^ *'b* :: *index1*) = {*a*}›
⟨*proof*⟩

lemma *vec-explode*: ‹*v* = (χ *i*. *v* $ *i*)›
⟨*proof*⟩

lemma *vec-explode1*:
fixes *v* :: ‹*'a* ^ *1*›
obtains *a* **where** ‹*v* = (χ -. *a*)›
⟨*proof*⟩

lemma *zip-nth-vec*[*simp*]: ‹*zip-vec u v* $ *i* = (*u* $ *i*, *v* $ *i*)›
⟨*proof*⟩

lemma *zip-vec-fst*[*simp*]: ‹*map-vec fst* (*zip-vec u v*) = *u*›
⟨*proof*⟩

lemma *zip-vec-snd*[*simp*]: ‹*map-vec snd* (*zip-vec u v*) = *v*›
⟨*proof*⟩

lemma *zip-lambda-vec*[*simp*]: ‹*zip-vec* (*vec-lambda f*) (*vec-lambda g*) = (χ *i*. (*f i*, *g i*))›
⟨*proof*⟩

lemma *zip-map-vec*: ‹*zip-vec* (*map-vec f u*) (*map-vec g v*) = *map-vec* (λ(*x*, *y*). (*f x*, *g y*)) (*zip-vec u v*)›
⟨*proof*⟩

lemma *list-of-vec-length*[*simp*]: ‹*length* (*list-of-vec* (*v* :: *'a* ^ *'b*)) = *CARD*(*'b*)›
⟨*proof*⟩

lemma *list-vec-list*: ‹*length xs* = *CARD*(*'n*) ⟹ *list-of-vec* (*vec-of-list xs* :: *'a* ^ *'n*) = *xs*›
⟨*proof*⟩

lemma *map-vec-list*: ‹*length xs* = *CARD*(*'n*) ⟹ *map-vec f* (*vec-of-list xs* :: *'a* ^ *'n*) = *vec-of-list* (*map f xs*)›
⟨*proof*⟩

lemma *set-vec-list*: ‹*length xs* = *CARD*(*'n*) ⟹ *set-vec* (*vec-of-list xs* :: *'a* ^ *'n*) = *set xs*›
⟨*proof*⟩

lemma *list-all-zip*: ‹*length xs* = *length ys* ⟹ *list-all P* (*zip xs ys*) ⟷ *list-all2* (λ*x y*. *P* (*x*, *y*)) *xs ys*›
⟨*proof*⟩

lemma *pred-vec-zip*: ‹*pred-vec P* (*zip-vec xs ys*) ⟷ *rel-vec* (λ*x y*. *P* (*x*, *y*)) *xs ys*›
⟨*proof*⟩

lemma *list-all2-left*: ‹*length xs* = *length ys* ⟹ *list-all2* (λ*x y*. *P x*) *xs ys* ⟷ *list-all P xs*›
⟨*proof*⟩

lemma *list-all2-right*: ‹*length xs* = *length ys* ⟹ *list-all2* (λ-. *P*) *xs ys* ⟷ *list-all P ys*›
⟨*proof*⟩

lemma *rel-vec-left*: ‹*rel-vec* (λ*x y*. *P x*) *xs ys* ⟷ *pred-vec P xs*›
⟨*proof*⟩

lemma *rel-vec-right*: ‹*rel-vec* (λ-. *P*) *xs ys* ⟷ *pred-vec P ys*›
⟨*proof*⟩

6 Instances

definition *bounded-lists* :: ‹*nat* ⇒ *'a set* ⇒ *'a list set*› **where**
‹*bounded-lists n A* = {*xs*. *length xs* = *n* ∧ *list-all* (λ*x*. *x* ∈ *A*) *xs*}›

lemma *bounded-lists-finite*:
assumes ‹*finite A*›
shows ‹*finite* (*bounded-lists n A*)›

⟨*proof*⟩

lemma *bounded-lists-bij*: ‹*bij-betw list-of-vec* (*UNIV* :: (*'a* ^ *'b*) *set*) (*bounded-lists CARD*(*'b*) *UNIV*)›
⟨*proof*⟩

If the base type is *finite*, so is the vector type.

instance *vec* :: (*finite*, *type*) *finite*
⟨*proof*⟩

The *size* of the vector is the *length* of the underlying list.

instantiation *vec* :: (*type*, *type*) *size*
begin

lift-definition *size-vec* :: ‹*'a* ^ *'b* ⇒ *nat*› **is** *length* ⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *size-vec-alt-def*[*simp*]: ‹*size* (*v* :: *'a* ^ *'b*) = *CARD*(*'b*)›
⟨*proof*⟩

Vectors can be compared for equality.

instantiation *vec* :: (*equal*, *type*) *equal*
begin

lift-definition *equal-vec* :: ‹*'a* ^ *'b* ⇒ *'a* ^ *'b* ⇒ *bool*› **is** ‹*equal-class.equal*› ⟨*proof*⟩

instance
⟨*proof*⟩

end

7 Further operations

7.1 Distinctness

lift-definition *distinct-vec* :: ‹*'a* ^ *'n* ⇒ *bool*› **is** ‹*distinct*› ⟨*proof*⟩

lemma *distinct-vec-alt-def*: ‹*distinct-vec v* ⟷ (∀ *i j*. *i* ≠ *j* ⟶ *v* $ *i* ≠ *v* $ *j*)›
⟨*proof*⟩

lemma *distinct-vecI*:
assumes ‹⋀*i j*. *i* ≠ *j* ⟹ *v* $ *i* ≠ *v* $ *j*›
shows ‹*distinct-vec v*›
⟨*proof*⟩

lemma *distinct-vec-mapI*: ‹*distinct-vec xs* ⟹ *inj-on f* (*set-vec xs*) ⟹ *distinct-vec* (*map-vec f xs*)›
⟨*proof*⟩

lemma *distinct-vec-zip-fst*: ‹*distinct-vec u* ⟹ *distinct-vec* (*zip-vec u v*)›
⟨*proof*⟩

lemma *distinct-vec-zip-snd*: ‹*distinct-vec v* ⟹ *distinct-vec* (*zip-vec u v*)›
⟨*proof*⟩

lemma *inj-set-of-vec*: ‹*distinct-vec* (*map-vec f v*) ⟹ *inj-on f* (*set-vec v*)›
⟨*proof*⟩

lemma *distinct-vec-list*: ‹*length xs* = *CARD*('*n*) ⟹ *distinct-vec* (*vec-of-list xs* :: '*a* ^ '*n*) ⟷ *distinct xs*›
⟨*proof*⟩

7.2 Summing

lift-definition *sum-vec* :: ‹'*b*::*comm-monoid-add* ^ '*a* ⇒ '*b*› **is** *sum-list* ⟨*proof*⟩

lemma *sum-vec-lambda*: ‹*sum-vec* (*vec-lambda v*) = *sum-list* (*map v indexes*)›
⟨*proof*⟩

lemma *elem-le-sum-vec*:
fixes *f* :: ‹'*a* :: *canonically-ordered-monoid-add* ^ '*b* :: *index1*›
shows *f* $ *i* ≤ *sum-vec f*
⟨*proof*⟩

8 Code setup

Since *vec-of-list* cannot be directly used in code generation, we defined a convenience wrapper that checks the length and aborts if necessary.

definition *replicate'* **where** ‹*replicate' n* = *replicate n undefined*›

declare [[*code abort*: *replicate'*]]

lift-definition *vec-of-list'* :: ‹'*a list* ⇒ '*a* ^ '*n*›
is ‹λ*xs*. *if length xs* ≠ *CARD*('*n*) *then replicate' CARD*('*n*) *else xs*›
⟨*proof*⟩

experiment begin

proposition
‹*sum-vec* (χ (*i*::*2*). (*3*::*nat*)) = *6*›
‹*distinct-vec* (*vec-of-list'* [*1*::*nat*, *2*] :: *nat* ^ *2*)›
‹¬ *distinct-vec* (*vec-of-list'* [*1*::*nat*, *1*] :: *nat* ^ *2*)›

⟨*proof*⟩

end

export-code
sum-vec
map-vec
rel-vec
pred-vec
set-vec
zip-vec
distinct-vec
list-of-vec
vec-of-list′
checking *SML*

lifting-update *vec.lifting*
lifting-forget *vec.lifting*

bundle *vec-syntax*
begin
type-notation *vec* (**infixl** ‹⌢› *15*)
notation *nth-vec* (**infixl** ‹$› *90*) **and** *vec-lambda* (**binder** ‹χ› *10*)
end

unbundle *no vec-syntax*

end