

Fixed-length vectors

Lars Hupel

May 26, 2024

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 <i>Fixed-Length-Vector</i>	
	imports <i>HOL-Library.Numeral-Type</i> <i>HOL-Library.Code-Cardinality</i>	
	begin	
	lemma <i>zip-map-same</i> : $\langle \text{zip } (\text{map } f \text{ } xs) (\text{map } g \text{ } xs) = \text{map } (\lambda x. (f \text{ } x, g \text{ } x)) \text{ } xs \rangle$	
	<i><proof></i>	

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]: $\langle \text{indexes} = [1 :: 1] \rangle$
 $\langle \text{proof} \rangle$

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

definition *from-index-bit0* :: $\langle 'a \text{ bit0} \Rightarrow \text{nat} \rangle$ **where** $\langle \text{from-index-bit0 } x = \text{nat} (\text{Rep-bit0 } x) \rangle$

definition *to-index-bit0* :: $\langle \text{nat} \Rightarrow 'a \text{ bit0} \rangle$ **where** $\langle \text{to-index-bit0} \equiv \text{of-nat} \rangle$

instance
 $\langle \text{proof} \rangle$

end

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

definition *from-index-bit1* :: $\langle 'a \text{ bit1} \Rightarrow \text{nat} \rangle$ **where** $\langle \text{from-index-bit1 } x = \text{nat} (\text{Rep-bit1 } x) \rangle$

definition *to-index-bit1* :: $\langle \text{nat} \Rightarrow 'a \text{ bit1} \rangle$ **where** $\langle \text{to-index-bit1} \equiv \text{of-nat} \rangle$

instance
 $\langle \text{proof} \rangle$

end

lemma *indexes-bit-simps*:

$\langle (\text{indexes} :: 'a :: \text{finite bit0 list}) = \text{map of-nat } [0..<2 * \text{CARD}('a)] \rangle$
 $\langle (\text{indexes} :: 'b :: \text{finite bit1 list}) = \text{map of-nat } [0..<2 * \text{CARD}('b) + 1] \rangle$
 $\langle \text{proof} \rangle$

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

class *index-enum* = *index1* + *enum* +
assumes *indexes-eq-enum*: $\langle \text{indexes} = \text{enum-class.enum} \rangle$

instance *num1* :: *index-enum*
 $\langle \text{proof} \rangle$

instance *bit0* :: (finite) *index-enum*
 $\langle \text{proof} \rangle$

instance *bit1* :: (finite) *index-enum*
 $\langle \text{proof} \rangle$

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 :: ⟨('a ⇒ 'b) ⇒ 'a ^ 'c ⇒ 'b ^ 'c⟩ is map ⟨proof⟩
```

```
lift-definition set-vec :: ⟨'a ^ 'b ⇒ 'a set⟩ is set ⟨proof⟩
```

```
lift-definition rel-vec :: ⟨('a ⇒ 'b ⇒ bool) ⇒ 'a ^ 'c ⇒ 'b ^ 'c ⇒ bool⟩ is list-all2
  ⟨proof⟩
```

```
lift-definition pred-vec :: ⟨('a ⇒ bool) ⇒ 'a ^ 'b ⇒ bool⟩ is list-all ⟨proof⟩
```

```
lift-definition zip-vec :: ⟨'a ^ 'c ⇒ 'b ^ 'c ⇒ ('a × 'b) ^ 'c⟩ is zip ⟨proof⟩
```

```
lift-definition replicate-vec :: ⟨'a ⇒ 'a ^ 'b⟩ is ⟨replicate CARD('b)⟩ ⟨proof⟩
```

```
bnf ⟨('a, 'b) vec⟩
  map: map-vec
  sets: set-vec
  bd: natLeq
  wits: replicate-vec
  rel: rel-vec
  pred: pred-vec
  ⟨proof⟩
```

3 Indexing

```
lift-definition nth-vec' :: ⟨'a ^ 'b ⇒ nat ⇒ 'a⟩ is nth ⟨proof⟩
```

```
lift-definition nth-vec :: ⟨'a ^ 'b ⇒ 'b :: index1 ⇒ 'a⟩ (infixl $ 90)
```

— We fix this to *index1* because indexing a unit vector makes no sense.
is ⟨λ*xs*. *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 ⟨λf. 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* \implies *f = g*
<proof>

lemma *vec-cong*: $(\bigwedge i. f \$ i = g \$ i) \implies f = g$
<proof>

lemma *finite-set-vec*[*intro, simp*]: *<finite (set-vec v)>*
<proof>

lemma *set-vec-in*[*intro, simp*]: *<v \$ i \in set-vec v>*
<proof>

lemma *set-vecE*[*elim*]:
assumes *<x \in 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 = (\chi i. v \$ i)>*
<proof>

lemma *vec-explode1*:
fixes *v :: 'a ^ 1*
obtains *a where <v = (\chi -. 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) = (\chi i. (f i, g i))>*
<proof>

lemma *zip-map-vec*: $\langle \text{zip-vec } (\text{map-vec } f \ u) \ (\text{map-vec } g \ v) = \text{map-vec } (\lambda(x, y). (f \ x, \ g \ y)) \ (\text{zip-vec } u \ v) \rangle$
 $\langle \text{proof} \rangle$

lemma *list-of-vec-length[simp]*: $\langle \text{length } (\text{list-of-vec } (v :: 'a \wedge 'b)) = \text{CARD}('b) \rangle$
 $\langle \text{proof} \rangle$

lemma *list-vec-list*: $\langle \text{length } xs = \text{CARD}('n) \implies \text{list-of-vec } (\text{vec-of-list } xs :: 'a \wedge 'n) = xs \rangle$
 $\langle \text{proof} \rangle$

lemma *map-vec-list*: $\langle \text{length } xs = \text{CARD}('n) \implies \text{map-vec } f \ (\text{vec-of-list } xs :: 'a \wedge 'n) = \text{vec-of-list } (\text{map } f \ xs) \rangle$
 $\langle \text{proof} \rangle$

lemma *set-vec-list*: $\langle \text{length } xs = \text{CARD}('n) \implies \text{set-vec } (\text{vec-of-list } xs :: 'a \wedge 'n) = \text{set } xs \rangle$
 $\langle \text{proof} \rangle$

lemma *list-all-zip*: $\langle \text{length } xs = \text{length } ys \implies \text{list-all } P \ (\text{zip } xs \ ys) \longleftrightarrow \text{list-all2 } (\lambda x \ y. P \ (x, \ y)) \ xs \ ys \rangle$
 $\langle \text{proof} \rangle$

lemma *pred-vec-zip*: $\langle \text{pred-vec } P \ (\text{zip-vec } xs \ ys) \longleftrightarrow \text{rel-vec } (\lambda x \ y. P \ (x, \ y)) \ xs \ ys \rangle$
 $\langle \text{proof} \rangle$

lemma *list-all2-left*: $\langle \text{length } xs = \text{length } ys \implies \text{list-all2 } (\lambda x \ y. P \ x) \ xs \ ys \longleftrightarrow \text{list-all } P \ xs \rangle$
 $\langle \text{proof} \rangle$

lemma *list-all2-right*: $\langle \text{length } xs = \text{length } ys \implies \text{list-all2 } (\lambda -. P) \ xs \ ys \longleftrightarrow \text{list-all } P \ ys \rangle$
 $\langle \text{proof} \rangle$

lemma *rel-vec-left*: $\langle \text{rel-vec } (\lambda x \ y. P \ x) \ xs \ ys \longleftrightarrow \text{pred-vec } P \ xs \rangle$
 $\langle \text{proof} \rangle$

lemma *rel-vec-right*: $\langle \text{rel-vec } (\lambda -. P) \ xs \ ys \longleftrightarrow \text{pred-vec } P \ ys \rangle$
 $\langle \text{proof} \rangle$

6 Instances

definition *bounded-lists* :: $\langle \text{nat} \implies 'a \ \text{set} \implies 'a \ \text{list set} \rangle$ **where**
 $\langle \text{bounded-lists } n \ A = \{xs. \text{length } xs = n \wedge \text{list-all } (\lambda x. x \in A) \ xs\} \rangle$

lemma *bounded-lists-finite*:
assumes $\langle \text{finite } A \rangle$
shows $\langle \text{finite } (\text{bounded-lists } n \ A) \rangle$

⟨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 ⟨ $\bigwedge i j. i \neq j \implies v \$ i \neq v \$ j$ ⟩
shows ⟨*distinct-vec v*⟩
⟨proof⟩

lemma *distinct-vec-mapI*: $\langle \text{distinct-vec } xs \implies \text{inj-on } f \text{ (set-vec } xs) \implies \text{distinct-vec (map-vec } f \text{ } xs) \rangle$
 $\langle \text{proof} \rangle$

lemma *distinct-vec-zip-fst*: $\langle \text{distinct-vec } u \implies \text{distinct-vec (zip-vec } u \text{ } v) \rangle$
 $\langle \text{proof} \rangle$

lemma *distinct-vec-zip-snd*: $\langle \text{distinct-vec } v \implies \text{distinct-vec (zip-vec } u \text{ } v) \rangle$
 $\langle \text{proof} \rangle$

lemma *inj-set-of-vec*: $\langle \text{distinct-vec (map-vec } f \text{ } v) \implies \text{inj-on } f \text{ (set-vec } v) \rangle$
 $\langle \text{proof} \rangle$

lemma *distinct-vec-list*: $\langle \text{length } xs = \text{CARD('n)} \implies \text{distinct-vec (vec-of-list } xs \text{ :: 'a } ^ \wedge \text{'n)} \longleftrightarrow \text{distinct } xs \rangle$
 $\langle \text{proof} \rangle$

7.2 Summing

lift-definition *sum-vec* :: $\langle 'b :: \text{comm-monoid-add } ^ \wedge \text{'a} \Rightarrow 'b \rangle$ **is** *sum-list* $\langle \text{proof} \rangle$

lemma *sum-vec-lambda*: $\langle \text{sum-vec (vec-lambda } v) = \text{sum-list (map } v \text{ indexes)} \rangle$
 $\langle \text{proof} \rangle$

lemma *elem-le-sum-vec*:

fixes *f* :: $\langle 'a :: \text{canonically-ordered-monoid-add } ^ \wedge \text{'b} :: \text{index1} \rangle$

shows $f \ \$ \ i \leq \text{sum-vec } f$

$\langle \text{proof} \rangle$

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** $\langle \text{replicate}' \ n = \text{replicate } n \ \text{undefined} \rangle$

declare $[[\text{code abort: replicate}']]$

lift-definition *vec-of-list'* :: $\langle 'a \ \text{list} \Rightarrow 'a \ ^ \wedge \text{'n} \rangle$

is $\langle \lambda xs. \ \text{if } \text{length } xs \neq \text{CARD('n)} \ \text{then } \text{replicate}' \ \text{CARD('n)} \ \text{else } xs \rangle$

$\langle \text{proof} \rangle$

experiment begin

proposition

$\langle \text{sum-vec } (\chi \ (i::2). \ (3::\text{nat})) = 6 \rangle$

$\langle \text{distinct-vec (vec-of-list}' \ [1::\text{nat}, 2] \text{ :: } \text{nat} \ ^ \wedge \ 2) \rangle$

$\langle \neg \ \text{distinct-vec (vec-of-list}' \ [1::\text{nat}, 1] \text{ :: } \text{nat} \ ^ \wedge \ 2) \rangle$

```

    <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  $\chi$  10)
end

bundle no-vec-syntax begin
no-type-notation
  vec (infixl ^ 15)
no-notation
  nth-vec (infixl $ 90) and
  vec-lambda (binder  $\chi$  10)
end

unbundle no-vec-syntax

end

```