

# 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

<b>1</b>	<b>Type class for indexing</b>	<b>1</b>
<b>2</b>	<b>Type definition and BNF setup</b>	<b>5</b>
<b>3</b>	<b>Indexing</b>	<b>6</b>
<b>4</b>	<b>Unit vector</b>	<b>7</b>
<b>5</b>	<b>General lemmas</b>	<b>7</b>
<b>6</b>	<b>Instances</b>	<b>9</b>
<b>7</b>	<b>Further operations</b>	<b>11</b>
	7.1 Distinctness . . . . .	11
	7.2 Summing . . . . .	11
<b>8</b>	<b>Code setup</b>	<b>12</b>
	<b>theory</b> <i>Fixed-Length-Vector</i>	
	<b>imports</b> <i>HOL-Library.Numeral-Type</i> <i>HOL-Library.Code-Cardinality</i>	
	<b>begin</b>	
	<b>lemma</b> <i>zip-map-same</i> : $\langle \text{zip } (\text{map } f \text{ } xs) (\text{map } g \text{ } xs) = \text{map } (\lambda x. (f \text{ } x, g \text{ } x)) \text{ } xs \rangle$	
	<b>by</b> ( <i>induction xs</i> ) <i>auto</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)⟩  
  unfolding indexes-def by auto
```

```
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⟩  
  apply (rule nth-equalityI)  
  using assms from-index-surj by auto
```

```
lemma from-indexE:  
  assumes ⟨n < CARD('a)⟩  
  obtains a where ⟨from-index a = n⟩  
  using assms by (metis from-index-surj)
```

**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]:  $\langle 0 < \text{CARD}(a) \rangle$   
**by** (*metis less-nat-zero-code from-index-bound neq0-conv*)

**lemma** *finite-type*[simp]:  $\langle \text{finite } (UNIV :: 'a \text{ set}) \rangle$   
**by** (*metis card-nonzero card.infinite less-irrefl*)

**sublocale** *finite*  
**by** *standard simp*

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

**lemma** *from-to-index*[simp]:  $\langle \text{to-index } (\text{from-index } i) = i \rangle$   
**by** (*meson injD from-index-bound from-index-inj to-from-index*)

**lemma** *indexes-from-index*[simp]:  $\langle \text{indexes ! from-index } i = i \rangle$   
**unfolding** *indexes-def* **by** *auto*

**lemma** *to-index-inj-on*:  $\langle \text{inj-on to-index } \{0..<\text{CARD}(a)\} \rangle$   
**by** (*rule inj-onI*) (*force elim: from-indexE*)

**end**

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

**instantiation** *num0* :: *index*  
**begin**

**definition** *from-index-num0* ::  $\langle \text{num0} \Rightarrow \text{nat} \rangle$  **where**  $\langle \text{from-index-num0 } - = \text{undefined} \rangle$

**definition** *to-index-num0* ::  $\langle \text{nat} \Rightarrow \text{num0} \rangle$  **where**  $\langle \text{to-index-num0 } - = \text{undefined} \rangle$

**instance**  
**by** *standard auto*

**end**

**lemma** *indexes-zero*[simp]:  $\langle \text{indexes} = ([ ] :: 0 \text{ list}) \rangle$   
**by** (*auto simp: indexes-def*)

**instantiation** *num1* :: *index1*  
**begin**

**definition** *from-index-num1* ::  $\langle \text{num1} \Rightarrow \text{nat} \rangle$  **where** [simp]:  $\langle \text{from-index-num1 } - = 0 \rangle$

**definition** *to-index-num1* ::  $\langle \text{nat} \Rightarrow \text{num1} \rangle$  **where** [simp]:  $\langle \text{to-index-num1 } - = 1 \rangle$

**instance**  
**by** *standard* (*auto simp: inj-on-def*)

**end**

**lemma** *indexes-one*[*simp*]:  $\langle \text{indexes} = [1 :: 1] \rangle$   
**by** (*auto simp: indexes-def*)

**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**

**apply** *standard*

**subgoal**

**by** (*simp add: to-index-bit0-def from-index-bit0-def bit0.of-nat-eq Abs-bit0-inverse*)

**subgoal for** *n*

**unfolding** *from-index-bit0-def* **by** (*auto simp: Abs-bit0-inverse intro!: exI*[**where**  $x = \langle \text{Abs-bit0 } (\text{int } n) \rangle$ ])

**subgoal for** *n*

**using** *Rep-bit0*[*of n*]

**by** (*simp add: from-index-bit0-def nat-less-iff*)

**subgoal**

**unfolding** *from-index-bit0-def inj-on-def*

**by** (*metis Rep-bit0 Rep-bit0-inverse atLeastLessThan-iff int-nat-eq*)

**done**

**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**

**apply** *standard*

**subgoal**

**by** (*simp add: to-index-bit1-def from-index-bit1-def bit1.of-nat-eq Abs-bit1-inverse*)

**subgoal for** *n*

**unfolding** *from-index-bit1-def* **by** (*auto simp: Abs-bit1-inverse intro!: exI*[**where**  $x = \langle \text{Abs-bit1 } (\text{int } n) \rangle$ ])

**subgoal for** *n*

**using** *Rep-bit1*[*of n*]

**by** (*simp add: from-index-bit1-def nat-less-iff*)

```

subgoal
  unfolding from-index-bit1-def inj-on-def
  by (metis Rep-bit1 Rep-bit1-inverse atLeastLessThan-iff eq-nat-nat-iff)
done

```

**end**

**lemma** *indexes-bit-simps*:

```

  ⟨(indexes :: 'a :: finite bit0 list) = map of-nat [0.. $2 * \text{CARD}('a)$ ⟩
  ⟨(indexes :: 'b :: finite bit1 list) = map of-nat [0.. $2 * \text{CARD}('b) + 1$ ⟩
  unfolding indexes-def to-index-bit0-def to-index-bit1-def
  by simp+

```

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
  by standard (auto simp: indexes-def enum-num1-def)

```

```

instance bit0 :: (finite) index-enum
  by standard (auto simp: indexes-def to-index-bit0-def enum-bit0-def Abs-bit0'-def bit0.of-nat-eq)

```

```

instance bit1 :: (finite) index-enum
  by standard (auto simp: indexes-def to-index-bit1-def enum-bit1-def Abs-bit1'-def bit1.of-nat-eq)

```

## 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 =  $\text{CARD}('b)$ } :: 'a list set
  morphisms list-of-vec vec-of-list
  by (rule exI[where x = ⟨replicate CARD('b) undefined⟩]) simp

```

```

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

```

```

type-notation vec (infixl  $\hat{\ }15$ )

```

```

setup-lifting type-definition-vec

```

```

lift-definition map-vec :: ⟨('a  $\Rightarrow$  'b)  $\Rightarrow$  'a  $\hat{\ }$  'c  $\Rightarrow$  'b  $\hat{\ }$  'c⟩ is map by auto

```

**lift-definition** *set-vec* ::  $\langle 'a \wedge 'b \Rightarrow 'a \text{ set} \rangle$  **is** *set* .

**lift-definition** *rel-vec* ::  $\langle ('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'a \wedge 'c \Rightarrow 'b \wedge 'c \Rightarrow \text{bool} \rangle$  **is** *list-all2* .

**lift-definition** *pred-vec* ::  $\langle ('a \Rightarrow \text{bool}) \Rightarrow 'a \wedge 'b \Rightarrow \text{bool} \rangle$  **is** *list-all* .

**lift-definition** *zip-vec* ::  $\langle 'a \wedge 'c \Rightarrow 'b \wedge 'c \Rightarrow ('a \times 'b) \wedge 'c \rangle$  **is** *zip by auto*

**lift-definition** *replicate-vec* ::  $\langle 'a \Rightarrow 'a \wedge 'b \rangle$  **is**  $\langle \text{replicate } \text{CARD}('b) \rangle$  **by** *auto*

**bnf**  $\langle ('a, 'b) \text{ vec} \rangle$   
*map*: *map-vec*  
*sets*: *set-vec*  
*bd*: *natLeq*  
*wits*: *replicate-vec*  
*rel*: *rel-vec*  
*pred*: *pred-vec*  
**subgoal**  
  **apply** (*rule ext*)  
  **by** *transfer'* *auto*  
**subgoal**  
  **apply** (*rule ext*)  
  **by** *transfer'* *auto*  
**subgoal**  
  **by** *transfer'* *auto*  
**subgoal**  
  **apply** (*rule ext*)  
  **by** *transfer'* *auto*  
**subgoal by** (*fact natLeq-card-order*)  
**subgoal by** (*fact natLeq-cinfinite*)  
**subgoal by** (*fact regularCard-natLeq*)  
**subgoal**  
  **apply** *transfer'*  
  **apply** (*simp flip: finite-iff-ordLess-natLeq*)  
  **done**  
**subgoal**  
  **apply** (*rule predicate2I*)  
  **apply** *transfer'*  
  **by** (*smt (verit) list-all2-trans relcompp.relcompI*)  
**subgoal**  
  **apply** (*rule ext*)+  
  **apply** *transfer*  
  **by** (*auto simp: list.in-rel*)  
**subgoal**  
  **apply** (*rule ext*)  
  **apply** *transfer'*  
  **by** (*auto simp: list-all-iff*)

**subgoal**  
 by *transfer' auto*  
**done**

### 3 Indexing

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

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

**lemma** *nth-vec-alt-def*:  $\langle \text{nth-vec } v = \text{nth-vec}' v \circ \text{from-index} \rangle$   
**by** *transfer' auto*

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

**lift-definition** *vec-lambda* ::  $\langle ('b :: \text{index} \Rightarrow 'a) \Rightarrow 'a \hat{ } 'b \rangle$  (**binder**  $\chi$  10)  
**is**  $\langle \lambda f. \text{map } f \text{ indexes} \rangle$  **by** *simp*

**lemma** *vec-lambda-nth[simp]*:  $\langle \text{vec-lambda } f \ \$ \ i = f \ i \rangle$   
**by** *transfer auto*

### 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* ::  $\langle 'a \hat{ } 0 \rangle$  **is**  $\langle [] \rangle$  **by** *simp*

**lemma** *unit-vec-unique*:  $\langle v = \text{unit-vec} \rangle$   
**by** *transfer auto*

**lemma** *unit-vec-unique-simp[simp]*:  $\langle \text{NO-MATCH } v \ \text{unit-vec} \Longrightarrow v = \text{unit-vec} \rangle$   
**by** (*rule unit-vec-unique*)

**lemma** *set-unit-vec[simp]*:  $\langle \text{set-vec } (v :: 'a \hat{ } 0) = \{\} \rangle$   
**by** *transfer auto*

**lemma** *map-unit-vec[simp]*:  $\langle \text{map-vec } f \ v = \text{unit-vec} \rangle$   
**by** *simp*

**lemma** *zip-unit-vec[simp]*:  $\langle \text{zip-vec } u \ v = \text{unit-vec} \rangle$   
**by** *simp*

**lemma** *rel-unit-vec[simp]*:  $\langle \text{rel-vec } R \ (u :: 'a \hat{ } 0) \ v \longleftrightarrow \text{True} \rangle$   
**by** *transfer auto*

**lemma** *pred-unit-vec*[*simp*]:  $\langle \text{pred-vec } P (v :: 'a \wedge 0) \rangle$   
**by** (*simp add: vec.pred-set*)

## 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*:  $\text{list-of-vec } f = \text{list-of-vec } g \implies f = g$   
**by** (*metis list-of-vec-inverse*)

**lemma** *vec-cong*:  $(\bigwedge i. f \$ i = g \$ i) \implies f = g$   
**by** *transfer (simp add: list-cong-index)*

**lemma** *finite-set-vec*[*intro, simp*]:  $\langle \text{finite } (\text{set-vec } v) \rangle$   
**by** *transfer' auto*

**lemma** *set-vec-in*[*intro, simp*]:  $\langle v \$ i \in \text{set-vec } v \rangle$   
**by** *transfer auto*

**lemma** *set-vecE*[*elim*]:  
**assumes**  $\langle x \in \text{set-vec } v \rangle$   
**obtains** *i where*  $\langle x = v \$ i \rangle$   
**using** *assms*  
**by** *transfer (auto simp: in-set-conv-nth elim: from-indexE)*

**lemma** *map-nth-vec*[*simp*]:  $\langle \text{map-vec } f v \$ i = f (v \$ i) \rangle$   
**by** *transfer auto*

**lemma** *replicate-nth-vec*[*simp*]:  $\langle \text{replicate-vec } a \$ i = a \rangle$   
**by** *transfer auto*

**lemma** *replicate-set-vec*[*simp*]:  $\langle \text{set-vec } (\text{replicate-vec } a :: 'a \wedge 'b :: \text{index1}) = \{a\} \rangle$   
**by** *transfer simp*

**lemma** *vec-explode*:  $\langle v = (\chi i. v \$ i) \rangle$   
**by** (*rule vec-cong*) *auto*

**lemma** *vec-explode1*:  
**fixes**  $v :: \langle 'a \wedge 1 \rangle$   
**obtains** *a where*  $\langle v = (\chi -. a) \rangle$



**apply** (*rule that*[*of*  $\langle v \ \$ \ 1 \rangle$ ])  
**apply** (*subst vec-explode*[*of*  $v$ ])  
**apply** (*rule arg-cong*[**where**  $f = \text{vec-lambda}$ ])  
**apply** (*rule ext*)  
**apply** (*subst num1-eq1*)  
**by** (*rule refl*)

**lemma** *zip-nth-vec*[*simp*]:  $\langle \text{zip-vec } u \ v \ \$ \ i = (u \ \$ \ i, v \ \$ \ i) \rangle$   
**by** *transfer auto*

**lemma** *zip-vec-fst*[*simp*]:  $\langle \text{map-vec fst } (\text{zip-vec } u \ v) = u \rangle$   
**by** *transfer auto*

**lemma** *zip-vec-snd*[*simp*]:  $\langle \text{map-vec snd } (\text{zip-vec } u \ v) = v \rangle$   
**by** *transfer auto*

**lemma** *zip-lambda-vec*[*simp*]:  $\langle \text{zip-vec } (\text{vec-lambda } f) \ (\text{vec-lambda } g) = (\chi \ i. (f \ i, g \ i)) \rangle$   
**by** *transfer' (simp add: zip-map-same)*

**lemma** *zip-map-vec*:  $\langle \text{zip-vec } (\text{map-vec } f \ u) \ (\text{map-vec } g \ v) = \text{map-vec } (\lambda(x, y). (f \ x, g \ y)) \ (\text{zip-vec } u \ v) \rangle$   
**by** *transfer' (auto simp: zip-map1 zip-map2)*

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

**lemma** *list-vec-list*:  $\langle \text{length } xs = \text{CARD}('n) \implies \text{list-of-vec } (\text{vec-of-list } xs :: 'a \ ^ \ 'n) = xs \rangle$   
**by** (*subst vec.vec-of-list-inverse*) *auto*

**lemma** *map-vec-list*:  $\langle \text{length } xs = \text{CARD}('n) \implies \text{map-vec } f \ (\text{vec-of-list } xs :: 'a \ ^ \ 'n) = \text{vec-of-list } (\text{map } f \ xs) \rangle$   
**by** (*rule map-vec.abs-eq*) (*auto simp: eq-onp-def*)

**lemma** *set-vec-list*:  $\langle \text{length } xs = \text{CARD}('n) \implies \text{set-vec } (\text{vec-of-list } xs :: 'a \ ^ \ 'n) = \text{set } xs \rangle$   
**by** (*rule set-vec.abs-eq*) (*auto simp: eq-onp-def*)

**lemma** *list-all-zip*:  $\langle \text{length } xs = \text{length } ys \implies \text{list-all } P \ (\text{zip } xs \ ys) \longleftrightarrow \text{list-all2 } (\lambda x y. P \ (x, y)) \ xs \ ys \rangle$   
**by** (*erule list-induct2*) *auto*

**lemma** *pred-vec-zip*:  $\langle \text{pred-vec } P \ (\text{zip-vec } xs \ ys) \longleftrightarrow \text{rel-vec } (\lambda x y. P \ (x, y)) \ xs \ ys \rangle$   
**by** *transfer (simp add: list-all-zip)*

**lemma** *list-all2-left*:  $\langle \text{length } xs = \text{length } ys \implies \text{list-all2 } (\lambda x y. P \ x) \ xs \ ys \longleftrightarrow \text{list-all } P \ xs \rangle$   
**by** (*erule list-induct2*) *auto*

**lemma** *list-all2-right*:  $\langle \text{length } xs = \text{length } ys \implies \text{list-all2 } (\lambda \cdot. P) \text{ } xs \text{ } ys \longleftrightarrow \text{list-all } P \text{ } ys \rangle$   
**by** (*erule list-induct2*) *auto*

**lemma** *rel-vec-left*:  $\langle \text{rel-vec } (\lambda x y. P \ x) \text{ } xs \text{ } ys \longleftrightarrow \text{pred-vec } P \text{ } xs \rangle$   
**by** *transfer (simp add: list-all2-left)*

**lemma** *rel-vec-right*:  $\langle \text{rel-vec } (\lambda \cdot. P) \text{ } xs \text{ } ys \longleftrightarrow \text{pred-vec } P \text{ } ys \rangle$   
**by** *transfer (simp add: list-all2-right)*

## 6 Instances

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

**lemma** *bounded-lists-finite*:  
**assumes**  $\langle \text{finite } A \rangle$   
**shows**  $\langle \text{finite } (\text{bounded-lists } n \ A) \rangle$   
**proof** (*induction n*)  
**case** (*Suc n*)  
**moreover have**  $\langle \text{bounded-lists } (\text{Suc } n) \ A \subseteq (\lambda(x, xs). x \# xs) \text{ } (A \times \text{bounded-lists } n \ A) \rangle$   
**unfolding** *bounded-lists-def*  
**by** (*force simp: length-Suc-conv split-beta*)  
**ultimately show** *?case*  
**using** *assms by (meson finite-SigmaI finite-imageI finite-subset)*  
**qed** (*simp add: bounded-lists-def*)

**lemma** *bounded-lists-bij*:  $\langle \text{bij-betw } \text{list-of-vec } (\text{UNIV} :: ('a \wedge 'b) \text{ set}) \text{ } (\text{bounded-lists } \text{CARD}('b) \ \text{UNIV}) \rangle$   
**unfolding** *bij-betw-def bounded-lists-def*  
**by** (*metis (no-types, lifting) Ball-set Collect-cong UNIV-I inj-def type-definition.Rep-range type-definition-vec vec-eq-if*)

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

**instance** *vec* ::  $(\text{finite}, \text{type}) \text{ finite}$   
**apply** *standard*  
**apply** (*subst bij-betw-finite[OF bounded-lists-bij]*)  
**apply** (*rule bounded-lists-finite*)  
**by** *simp*

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

**instantiation** *vec* ::  $(\text{type}, \text{type}) \text{ size}$   
**begin**

**lift-definition** *size-vec* ::  $\langle 'a \wedge 'b \Rightarrow \text{nat} \rangle$  **is** *length* .

**instance** ..

**end**

**lemma** *size-vec-alt-def*[*simp*]:  $\langle \text{size } (v :: 'a \wedge 'b) = \text{CARD}('b) \rangle$   
**by** *transfer simp*

Vectors can be compared for equality.

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

**lift-definition** *equal-vec* ::  $\langle 'a \wedge 'b \Rightarrow 'a \wedge 'b \Rightarrow \text{bool} \rangle$  **is**  $\langle \text{equal-class.equal} \rangle$  .

**instance**

**apply** *standard*  
**apply** *transfer'*  
**by** (*simp add: equal-list-def*)

**end**

## 7 Further operations

### 7.1 Distinctness

**lift-definition** *distinct-vec* ::  $\langle 'a \wedge 'n \Rightarrow \text{bool} \rangle$  **is**  $\langle \text{distinct} \rangle$  .

**lemma** *distinct-vec-alt-def*:  $\langle \text{distinct-vec } v \longleftrightarrow (\forall i j. i \neq j \longrightarrow v \$ i \neq v \$ j) \rangle$   
**apply** *transfer*  
**unfolding** *distinct-conv-nth comp-apply*  
**by** (*metis from-index-bound from-to-index to-from-index*)

**lemma** *distinct-vecI*:

**assumes**  $\langle \bigwedge i j. i \neq j \Longrightarrow v \$ i \neq v \$ j \rangle$   
**shows**  $\langle \text{distinct-vec } v \rangle$   
**using** *assms* **unfolding** *distinct-vec-alt-def* **by** *simp*

**lemma** *distinct-vec-mapI*:  $\langle \text{distinct-vec } xs \Longrightarrow \text{inj-on } f \text{ (set-vec } xs) \Longrightarrow \text{distinct-vec (map-vec } f \text{ } xs) \rangle$   
**by** *transfer'* (*metis distinct-map*)

**lemma** *distinct-vec-zip-fst*:  $\langle \text{distinct-vec } u \Longrightarrow \text{distinct-vec (zip-vec } u \text{ } v) \rangle$   
**by** *transfer'* (*metis distinct-zipI1*)

**lemma** *distinct-vec-zip-snd*:  $\langle \text{distinct-vec } v \Longrightarrow \text{distinct-vec (zip-vec } u \text{ } v) \rangle$   
**by** *transfer'* (*metis distinct-zipI2*)

**lemma** *inj-set-of-vec*:  $\langle \text{distinct-vec (map-vec } f \text{ } v) \Longrightarrow \text{inj-on } f \text{ (set-vec } v) \rangle$   
**by** *transfer'* (*metis distinct-map*)

**lemma** *distinct-vec-list*:  $\langle \text{length } xs = \text{CARD}('n) \implies \text{distinct-vec } (\text{vec-of-list } xs :: 'a \wedge 'n) \longleftrightarrow \text{distinct } xs \rangle$   
**by** (*subst distinct-vec.rep-eq*) (*simp add: list-vec-list*)

## 7.2 Summing

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

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

**lemma** *elem-le-sum-vec*:  
**fixes** *f* ::  $\langle 'a :: \text{canonically-ordered-monoid-add } \wedge 'b :: \text{index1} \rangle$   
**shows**  $f \$ i \leq \text{sum-vec } f$   
**by** *transfer (simp add: elem-le-sum-list)*

## 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'* ::  $\langle 'a \text{ list} \Rightarrow 'a \wedge 'n \rangle$   
**is**  $\langle \lambda xs. \text{if } \text{length } xs \neq \text{CARD}('n) \text{ then } \text{replicate}' \text{ CARD}('n) \text{ else } xs \rangle$   
**by** (*auto simp: replicate'-def*)

**experiment begin**

**proposition**  
 $\langle \text{sum-vec } (\chi (i::2). (3::\text{nat})) = 6 \rangle$   
 $\langle \text{distinct-vec } (\text{vec-of-list}' [1::\text{nat}, 2] :: \text{nat} \wedge 2) \rangle$   
 $\langle \neg \text{distinct-vec } (\text{vec-of-list}' [1::\text{nat}, 1] :: \text{nat} \wedge 2) \rangle$   
**by** *eval+*

**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

```