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 **5**

3 Indexing **6**

4 Unit vector **7**

5 General lemmas **7**

6 Instances **9**

7 Further operations **11**
7.1 Distinctness 11
7.2 Summing 11

8 Code setup **12**

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*›
by (*induction xs*) *auto*

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*]: ‹*0 < CARD('a)*›
  by (*metis less-nat-zero-code from-index-bound neq0-conv*)

lemma *finite-type*[*simp*]: ‹*finite* (*UNIV* :: *'a set*)›
  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*]: ‹*to-index* (*from-index i*) = *i*›
  by (*meson injD from-index-bound from-index-inj to-from-index*)

lemma *indexes-from-index*[*simp*]: ‹*indexes* ! *from-index i* = *i*›
  unfolding *indexes-def* **by** *auto*

lemma *to-index-inj-on*: ‹*inj-on to-index* {*0*..<*CARD('a)*}›
  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* :: ‹*num0* ⇒ *nat*› **where** ‹*from-index-num0 - = undefined*›
definition *to-index-num0* :: ‹*nat* ⇒ *num0*› **where** ‹*to-index-num0 - = undefined*›

instance
  by *standard auto*

end

lemma *indexes-zero*[*simp*]: ‹*indexes* = ([] :: *0 list*)›
  by (*auto simp*: *indexes-def*)

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

end

lemma *indexes-one*[*simp*]: ‹*indexes* = [*1* :: *1*]›
by (*auto simp*: *indexes-def*)

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
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* = ‹*Abs-bit0* (*int n*)›])
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* :: ‹'*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
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* = ‹*Abs-bit1* (*int n*)›])
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* ∗ *CARD*(*'a*)]›
‹(*indexes* :: *'b* :: *finite bit1 list*) = *map of-nat* [*0..<2* ∗ *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* = *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** ‹^› *15*)

setup-lifting *type-definition-vec*

lift-definition *map-vec* :: ‹(*'a* ⇒ *'b*) ⇒ *'a* ^ *'c* ⇒ *'b* ^ *'c*› **is** *map* **by** *auto*

lift-definition *set-vec* :: ‹*'a ^ 'b ⇒ 'a set*› **is** *set* **.**

lift-definition *rel-vec* :: ‹(*'a ⇒ 'b ⇒ bool*) ⇒ *'a ^ 'c ⇒ 'b ^ 'c ⇒ bool*› **is** *list-all2*
.

lift-definition *pred-vec* :: ‹(*'a ⇒ bool*) ⇒ *'a ^ 'b ⇒ bool*› **is** *list-all* **.**

lift-definition *zip-vec* :: ‹*'a ^ 'c ⇒ 'b ^ 'c ⇒ ('a × 'b) ^ 'c*› **is** *zip* **by** *auto*

lift-definition *replicate-vec* :: ‹*'a ⇒ 'a ^ 'b*› **is** ‹*replicate CARD('b)*› **by** *auto*

bnf ‹(*'a*, *'b*) *vec*›
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'* :: ‹*'a ^ 'b ⇒ nat ⇒ 'a*› **is** *nth* **.**

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 ‹*λxs. nth xs ∘ from-index*› **.**

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

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*› **by** *simp*

lemma *vec-lambda-nth*[*simp*]: ‹*vec-lambda f $ i = f i*›
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* :: ‹*'a ^ 0*› **is** ‹[]› **by** *simp*

lemma *unit-vec-unique*: ‹*v = unit-vec*›
by *transfer auto*

lemma *unit-vec-unique-simp*[*simp*]: ‹*NO-MATCH v unit-vec ⟹ v = unit-vec*›
by (*rule unit-vec-unique*)

lemma *set-unit-vec*[*simp*]: ‹*set-vec (v :: 'a ^ 0) = {}*›
by *transfer auto*

lemma *map-unit-vec*[*simp*]: ‹*map-vec f v = unit-vec*›
by *simp*

lemma *zip-unit-vec*[*simp*]: ‹*zip-vec u v = unit-vec*›
by *simp*

lemma *rel-unit-vec*[*simp*]: ‹*rel-vec R (u :: 'a ^ 0) v ⟷ True*›
by *transfer auto*

lemma *pred-unit-vec*[*simp*]: ‹*pred-vec P* (*v* :: *'a* ^ *0*)›
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*: *list-of-vec f* = *list-of-vec g* ⟹ *f* = *g*
by (*metis list-of-vec-inverse*)

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

lemma *finite-set-vec*[*intro*, *simp*]: ‹*finite* (*set-vec v*)›
by *transfer'* *auto*

lemma *set-vec-in*[*intro*, *simp*]: ‹*v* $ *i* ∈ *set-vec v*›
by *transfer auto*

lemma *set-vecE*[*elim*]:
assumes ‹*x* ∈ *set-vec v*›
obtains *i* **where** ‹*x* = *v* $ *i*›
using *assms*
by *transfer* (*auto simp*: *in-set-conv-nth elim*: *from-indexE*)

lemma *map-nth-vec*[*simp*]: ‹*map-vec f v* $ *i* = *f* (*v* $ *i*)›
by *transfer auto*

lemma *replicate-nth-vec*[*simp*]: ‹*replicate-vec a* $ *i* = *a*›
by *transfer auto*

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

lemma *vec-explode*: ‹*v* = (χ *i*. *v* $ *i*)›
by (*rule vec-cong*) *auto*

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

apply (*rule that*[*of* ‹*v* $ 1›])
apply (*subst vec-explode*[*of v*])
apply (*rule arg-cong*[**where** *f* = *vec-lambda*])
apply (*rule ext*)
apply (*subst num1-eq1*)
by (*rule refl*)

lemma *zip-nth-vec*[*simp*]: ‹*zip-vec u v* $ *i* = (*u* $ *i*, *v* $ *i*)›
by *transfer auto*

lemma *zip-vec-fst*[*simp*]: ‹*map-vec fst* (*zip-vec u v*) = *u*›
by *transfer auto*

lemma *zip-vec-snd*[*simp*]: ‹*map-vec snd* (*zip-vec u v*) = *v*›
by *transfer auto*

lemma *zip-lambda-vec*[*simp*]: ‹*zip-vec* (*vec-lambda f*) (*vec-lambda g*) = (χ *i*. (*f i*, *g i*))›
by *transfer'* (*simp add*: *zip-map-same*)

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*)›
by *transfer'* (*auto simp*: *zip-map1 zip-map2*)

lemma *list-of-vec-length*[*simp*]: ‹*length* (*list-of-vec* (*v* :: *'a* ^ *'b*)) = *CARD*(*'b*)›
using *list-of-vec* **by** *blast*

lemma *list-vec-list*: ‹*length xs* = *CARD*(*'n*) ⟹ *list-of-vec* (*vec-of-list xs* :: *'a* ^ *'n*) = *xs*›
by (*subst vec.vec-of-list-inverse*) *auto*

lemma *map-vec-list*: ‹*length xs* = *CARD*(*'n*) ⟹ *map-vec f* (*vec-of-list xs* :: *'a* ^ *'n*) = *vec-of-list* (*map f xs*)›
by (*rule map-vec.abs-eq*) (*auto simp*: *eq-onp-def*)

lemma *set-vec-list*: ‹*length xs* = *CARD*(*'n*) ⟹ *set-vec* (*vec-of-list xs* :: *'a* ^ *'n*) = *set xs*›
by (*rule set-vec.abs-eq*) (*auto simp*: *eq-onp-def*)

lemma *list-all-zip*: ‹*length xs* = *length ys* ⟹ *list-all P* (*zip xs ys*) ⟷ *list-all2* (λ*x y*. *P* (*x*, *y*)) *xs ys*›
by (*erule list-induct2*) *auto*

lemma *pred-vec-zip*: ‹*pred-vec P* (*zip-vec xs ys*) ⟷ *rel-vec* (λ*x y*. *P* (*x*, *y*)) *xs ys*›
by *transfer* (*simp add*: *list-all-zip*)

lemma *list-all2-left*: ‹*length xs* = *length ys* ⟹ *list-all2* (λ*x y*. *P x*) *xs ys* ⟷ *list-all P xs*›
by (*erule list-induct2*) *auto*

lemma *list-all2-right*: ‹*length xs* = *length ys* ⟹ *list-all2* (λ-. *P*) *xs ys* ⟷ *list-all P ys*›
by (*erule list-induct2*) *auto*

lemma *rel-vec-left*: ‹*rel-vec* (λ*x y*. *P x*) *xs ys* ⟷ *pred-vec P xs*›
by *transfer* (*simp add*: *list-all2-left*)

lemma *rel-vec-right*: ‹*rel-vec* (λ-. *P*) *xs ys* ⟷ *pred-vec P ys*›
by *transfer* (*simp add*: *list-all2-right*)

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 (*induction n*)
case (*Suc n*)
moreover have ‹*bounded-lists* (*Suc n*) *A* ⊆ (λ(*x*, *xs*). *x* # *xs*) ‘ (*A* × *bounded-lists n A*)›
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*: ‹*bij-betw list-of-vec* (*UNIV* :: (*'a* ^ *'b*) *set*) (*bounded-lists CARD*(*'b*) *UNIV*)›
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* :: (*finite*, *type*) *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* :: (*type*, *type*) *size*
begin

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

instance ..

end

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

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*› **.**

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

end

7 Further operations

7.1 Distinctness

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

lemma *distinct-vec-alt-def*: ‹*distinct-vec v* ⟷ (∀ *i j*. *i* ≠ *j* ⟶ *v* $ *i* ≠ *v* $ *j*)›
apply *transfer*
unfolding *distinct-conv-nth comp-apply*
by (*metis from-index-bound from-to-index to-from-index*)

lemma *distinct-vecI*:
assumes ‹⋀*i j*. *i* ≠ *j* ⟹ *v* $ *i* ≠ *v* $ *j*›
shows ‹*distinct-vec v*›
using *assms* **unfolding** *distinct-vec-alt-def* **by** *simp*

lemma *distinct-vec-mapI*: ‹*distinct-vec xs* ⟹ *inj-on f* (*set-vec xs*) ⟹ *distinct-vec* (*map-vec f xs*)›
by *transfer'* (*metis distinct-map*)

lemma *distinct-vec-zip-fst*: ‹*distinct-vec u* ⟹ *distinct-vec* (*zip-vec u v*)›
by *transfer'* (*metis distinct-zipI1*)

lemma *distinct-vec-zip-snd*: ‹*distinct-vec v* ⟹ *distinct-vec* (*zip-vec u v*)›
by *transfer'* (*metis distinct-zipI2*)

lemma *inj-set-of-vec*: ‹*distinct-vec* (*map-vec f v*) ⟹ *inj-on f* (*set-vec v*)›
by *transfer'* (*metis distinct-map*)

lemma *distinct-vec-list*: ‹*length xs* = *CARD*('*n*) ⟹ *distinct-vec* (*vec-of-list xs* :: '*a* ^ '*n*) ⟷ *distinct xs*›
by (*subst distinct-vec.rep-eq*) (*simp add*: *list-vec-list*)

7.2 Summing

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

lemma *sum-vec-lambda*: ‹*sum-vec* (*vec-lambda v*) = *sum-list* (*map v indexes*)›
by *transfer simp*

lemma *elem-le-sum-vec*:
fixes *f* :: ‹'*a* :: *canonically-ordered-monoid-add* ^ '*b* :: *index1*›
shows *f* $ *i* ≤ *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** ‹*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*›
by (*auto simp*: *replicate'-def*)

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*)›
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** ‹χ› *10*)
end

unbundle *no vec-syntax*

end