$SPAD/src/algebra carten.spad

Stephen M. Watt

May 20, 1991

Abstract

CartesianTensor(minix,dim,R) provides Cartesian tensors with components belonging to a commutative ring R. These tensors can have any number of indices. Each index takes values from *minix* to $minix+dim-1$.

GradedAlgebra(R,E) denotes "E-graded R-algebra". A graded algebra is a graded module together with a degree preserving R-linear map, called the *product*.

The name "product" is written out in full so inner and outer products with the same mapping type can be distinguished by name.

GradedModule(R,E) denotes "E-graded R-module", i.e. collection of R-modules indexed by an abelian monoid E. An element g of $G[s]$ for some specific s in E is said to be an element of G with *degree* s. Sums are defined in each module $G[s]$ so two elements of G have a sum if they have the same degree.

Morphisms can be defined and composed by degree to give the mathematical category of graded modules.

Contents

1 category GRMOD GradedModule

⟨*category GRMOD GradedModule*⟩≡

```
)abbrev category GRMOD GradedModule
++ Author: Stephen M. Watt
++ Date Created: May 20, 1991
++ Date Last Updated: May 20, 1991
++ Basic Operations: +, *, degree
++ Related Domains: CartesianTensor(n,dim,R)
++ Also See:
++ AMS Classifications:
++ Keywords: graded module, tensor, multi-linear algebra
++ Examples:
++ References: Algebra 2d Edition, MacLane and Birkhoff, MacMillan 1979
++ Description:
++  GradedModule(R,E) denotes ``E-graded R-module'', i.e. collection of
++  R-modules indexed by an abelian monoid E.
++  An element \spad{g} of \spad{G[s]} for some specific \spad{s} in \spad{E}
++  is said to be an element of \spad{G} with {\em degree} \spad{s}.
++  Sums are defined in each module \spad{G[s]} so two elements of \spad{G}
++  have a sum if they have the same degree.
++
++  Morphisms can be defined and composed by degree to give the
++  mathematical category of graded modules.

GradedModule(R: CommutativeRing, E: AbelianMonoid): Category ==
    SetCategory with
        degree: % -> E
            ++ degree(g) names the degree of g.  The set of all elements
            ++ of a given degree form an R-module.
        0: constant -> %
            ++ 0 denotes the zero of degree 0.
        _*: (R, %) -> %
            ++ r*g is left module multiplication.
        _*: (%, R) -> %
            ++ g*r is right module multiplication.

        _-: % -> %
            ++ -g is the additive inverse of g in the module of elements
            ++ of the same grade as g.
        _+: (%, %) -> %
            ++ g+h is the sum of g and h in the module of elements of
            ++ the same degree as g and h.  Error: if g and h
            ++ have different degrees.
        _-: (%, %) -> %
            ++ g-h is the difference of g and h in the module of elements of
```

```
            ++ the same degree as g and h.  Error: if g and h
            ++ have different degrees.
add
      (x: %) - (y: %) == x+(-y)
```

2 category GRALG GradedAlgebra

⟨*category GRALG GradedAlgebra*⟩≡

```
)abbrev category GRALG GradedAlgebra
++ Author: Stephen M. Watt
++ Date Created: May 20, 1991
++ Date Last Updated: May 20, 1991
++ Basic Operations: +, *, degree
++ Related Domains: CartesianTensor(n,dim,R)
++ Also See:
++ AMS Classifications:
++ Keywords: graded module, tensor, multi-linear algebra
++ Examples:
++ References: Encyclopedic Dictionary of Mathematics, MIT Press, 1977
++ Description:
++  GradedAlgebra(R,E) denotes ``E-graded R-algebra''.
++  A graded algebra is a graded module together with a degree preserving
++  R-linear map, called the {\em product}.
++
++  The name ``product'' is written out in full so inner and outer products
++  with the same mapping type can be distinguished by name.

GradedAlgebra(R: CommutativeRing, E: AbelianMonoid): Category ==
    Join(GradedModule(R, E),RetractableTo(R)) with
        1: constant -> %
            ++ 1 is the identity for \spad{product}.
        product: (%, %) -> %
            ++ product(a,b) is the degree-preserving R-linear product:
            ++
            ++   \spad{degree product(a,b) = degree a + degree b}
            ++   \spad{product(a1+a2,b) = product(a1,b) + product(a2,b)}
            ++   \spad{product(a,b1+b2) = product(a,b1) + product(a,b2)}
            ++   \spad{product(r*a,b) = product(a,r*b) = r*product(a,b)}
            ++   \spad{product(a,product(b,c)) = product(product(a,b),c)}
  add
        if not (R is %) then
            0: % == (0$R)::%
            1: % == 1$R::%
            (r: R)*(x: %) == product(r::%, x)
            (x: %)*(r: R) == product(x, r::%)
```

3 domain CARTEN CartesianTensor

⟨*domain CARTEN CartesianTensor*⟩≡
```
)abbrev domain CARTEN CartesianTensor
++ Author: Stephen M. Watt
++ Date Created: December 1986
++ Date Last Updated: May 15, 1991
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords: tensor, graded algebra
++ Examples:
++ References:
++ Description:
++   CartesianTensor(minix,dim,R) provides Cartesian tensors with
++   components belonging to a commutative ring R.  These tensors
++   can have any number of indices.  Each index takes values from
++   \spad{minix} to \spad{minix + dim - 1}.

CartesianTensor(minix, dim, R): Exports == Implementation where
    NNI ==> NonNegativeInteger
    I   ==> Integer
    DP  ==> DirectProduct
    SM  ==> SquareMatrix

    minix: Integer
    dim: NNI
    R: CommutativeRing

    Exports ==> Join(GradedAlgebra(R, NNI), GradedModule(I, NNI)) with

        coerce: DP(dim, R) -> %
            ++ coerce(v) views a vector as a rank 1 tensor.
        coerce: SM(dim, R)  -> %
            ++ coerce(m) views a matrix as a rank 2 tensor.

        coerce: List R -> %
            ++ coerce([r_1,...,r_dim]) allows tensors to be constructed
            ++ using lists.

        coerce: List % -> %
            ++ coerce([t_1,...,t_dim]) allows tensors to be constructed
            ++ using lists.

        rank: % -> NNI
```

```
    ++ rank(t) returns the tensorial rank of t (that is, the
    ++ number of indices).  This is the same as the graded module
    ++ degree.

elt: (%) -> R
    ++ elt(t) gives the component of a rank 0 tensor.
elt: (%, I) -> R
    ++ elt(t,i) gives a component of a rank 1 tensor.
elt: (%, I, I) -> R
    ++ elt(t,i,j) gives a component of a rank 2 tensor.
elt: (%, I, I, I) -> R
    ++ elt(t,i,j,k) gives a component of a rank 3 tensor.
elt: (%, I, I, I, I) -> R
    ++ elt(t,i,j,k,l) gives a component of a rank 4 tensor.

elt: (%, List I) -> R
    ++ elt(t,[i1,...,iN]) gives a component of a rank \spad{N} tensor.

-- This specializes the documentation from GradedAlgebra.
product: (%,%) -> %
    ++ product(s,t) is the outer product of the tensors s and t.
    ++ For example, if \spad{r = product(s,t)} for rank 2 tensors s and t,
    ++ then \spad{r} is a rank 4 tensor given by
    ++     \spad{r(i,j,k,l) = s(i,j)*t(k,l)}.

"*": (%, %) -> %
    ++ s*t is the inner product of the tensors s and t which contracts
    ++ the last index of s with the first index of t, i.e.
    ++     \spad{t*s = contract(t,rank t, s, 1)}
    ++     \spad{t*s = sum(k=1..N, t[i1,..,iN,k]*s[k,j1,..,jM])}
    ++ This is compatible with the use of \spad{M*v} to denote
    ++ the matrix-vector inner product.

contract:  (%, Integer, %, Integer) -> %
    ++ contract(t,i,s,j) is the inner product of tenors s and t
    ++ which sums along the \spad{k1}-th index of
    ++ t and the \spad{k2}-th index of s.
    ++ For example, if \spad{r = contract(s,2,t,1)} for rank 3 tensors
    ++ rank 3 tensors \spad{s} and \spad{t}, then \spad{r} is
    ++ the rank 4 \spad{(= 3 + 3 - 2)} tensor  given by
    ++     \spad{r(i,j,k,l) = sum(h=1..dim,s(i,h,j)*t(h,k,l))}.

contract:  (%, Integer, Integer)    -> %
    ++ contract(t,i,j) is the contraction of tensor t which
    ++ sums along the \spad{i}-th and \spad{j}-th indices.
    ++ For example,  if
```

```
    ++ \spad{r = contract(t,1,3)} for a rank 4 tensor t, then
    ++ \spad{r} is the rank 2 \spad{(= 4 - 2)} tensor given by
    ++     \spad{r(i,j) = sum(h=1..dim,t(h,i,h,j))}.

transpose: % -> %
    ++ transpose(t) exchanges the first and last indices of t.
    ++ For example, if \spad{r = transpose(t)} for a rank 4 tensor t, then
    ++ \spad{r} is the rank 4 tensor given by
    ++     \spad{r(i,j,k,l) = t(l,j,k,i)}.

transpose: (%, Integer, Integer) -> %
    ++ transpose(t,i,j) exchanges the \spad{i}-th and \spad{j}-th indices of t
    ++ For example, if \spad{r = transpose(t,2,3)} for a rank 4 tensor t, then
    ++ \spad{r} is the rank 4 tensor given by
    ++     \spad{r(i,j,k,l) = t(i,k,j,l)}.

reindex: (%, List Integer) -> %
    ++ reindex(t,[i1,...,idim]) permutes the indices of t.
    ++ For example, if \spad{r = reindex(t, [4,1,2,3])}
    ++ for a rank 4 tensor t,
    ++ then \spad{r} is the rank for tensor given by
    ++     \spad{r(i,j,k,l) = t(l,i,j,k)}.

kroneckerDelta:  () -> %
    ++ kroneckerDelta() is the rank 2 tensor defined by
    ++    \spad{kroneckerDelta()(i,j)}
    ++       \spad{= 1  if i = j}
    ++       \spad{= 0 if  i \~= j}

leviCivitaSymbol: () -> %
    ++ leviCivitaSymbol() is the rank \spad{dim} tensor defined by
    ++ \spad{leviCivitaSymbol()(i1,...idim) = +1/0/-1}
    ++ if \spad{i1,...,idim} is an even/is nota /is an odd permutation
    ++ of \spad{minix,...,minix+dim-1}.
ravel:     % -> List R
    ++ ravel(t) produces a list of components from a tensor such that
    ++   \spad{unravel(ravel(t)) = t}.

unravel:   List R -> %
    ++ unravel(t) produces a tensor from a list of
    ++ components such that
    ++   \spad{unravel(ravel(t)) = t}.

sample:    () -> %
    ++ sample() returns an object of type %.
```

```
Implementation ==> add

    PERM  ==> Vector Integer  -- 1-based entries from 1..n
    INDEX ==> Vector Integer  -- 1-based entries from minix..minix+dim-1


    get   ==> elt$Rep
    set_! ==> setelt$Rep

    -- Use row-major order:
    --   x[h,i,j] <-> x[(h-minix)*dim^2+(i-minix)*dim+(j-minix)]

    Rep := IndexedVector(R,0)

    n:     Integer
    r,s:   R
    x,y,z: %

    ---- Local stuff
    dim2: NNI := dim^2
    dim3: NNI := dim^3
    dim4: NNI := dim^4

    sample()==kroneckerDelta()$%
    int2index(n: Integer, indv: INDEX): INDEX ==
        n < 0 => error "Index error (too small)"
        rnk := #indv
        for i in 1..rnk repeat
            qr := divide(n, dim)
            n  := qr.quotient
            indv.((rnk-i+1) pretend NNI) := qr.remainder + minix
        n ~= 0 => error "Index error (too big)"
        indv

    index2int(indv: INDEX): Integer ==
        n: I := 0
        for i in 1..#indv repeat
            ix := indv.i - minix
            ix<0 or ix>dim-1 => error "Index error (out of range)"
            n := dim*n + ix
        n

    lengthRankOrElse(v: Integer): NNI ==
        v = 1    => 0
        v = dim  => 1
        v = dim2 => 2
```

```
    v = dim3 => 3
    v = dim4 => 4
    rx := 0
    while v ~= 0 repeat
        qr := divide(v, dim)
        v  := qr.quotient
        if v ~= 0 then
            qr.remainder ~= 0 => error "Rank is not a whole number"
            rx := rx + 1
    rx

-- l must be a list of the numbers 1..#l
mkPerm(n: NNI, l: List Integer): PERM ==
    #l ~= n =>
        error "The list is not a permutation."
    p:    PERM           := new(n, 0)
    seen: Vector Boolean := new(n, false)
    for i in 1..n for e in l repeat
        e < 1 or e > n => error "The list is not a permutation."
        p.i    := e
        seen.e := true
    for e in 1..n repeat
        not seen.e => error "The list is not a permutation."
    p

-- permute s according to p into result t.
permute_!(t: INDEX, s: INDEX, p: PERM): INDEX ==
    for i in 1..#p repeat t.i := s.(p.i)
    t

-- permsign!(v) = 1, 0, or -1  according as
-- v is an even, is not, or is an odd permutation of minix..minix+#v-1.
permsign_!(v: INDEX): Integer ==
    -- sum minix..minix+#v-1.
    maxix := minix+#v-1
    psum  := (((maxix+1)*maxix - minix*(minix-1)) exquo 2)::Integer
    -- +/v ~= psum => 0
    n := 0
    for i in 1..#v repeat n := n + v.i
    n ~= psum => 0
    -- Bubble sort!  This is pretty grotesque.
    totTrans: Integer := 0
    nTrans:   Integer := 1
    while nTrans ~= 0 repeat
        nTrans := 0
        for i in 1..#v-1 for j in 2..#v repeat
```

```
            if v.i > v.j then
                nTrans := nTrans + 1
                e := v.i; v.i := v.j; v.j := e
        totTrans := totTrans + nTrans
    for i in 1..dim repeat
        if v.i ~= minix+i-1 then return 0
    odd? totTrans => -1
    1


---- Exported functions
ravel x ==
    [get(x,i) for i in 0..#x-1]

unravel l ==
    -- lengthRankOrElse #l gives sytnax error
    nz: NNI := # l
    lengthRankOrElse nz
    z := new(nz, 0)
    for i in 0..nz-1 for r in l repeat set_!(z, i, r)
    z

kroneckerDelta() ==
    z := new(dim2, 0)
    for i in 1..dim for zi in 0.. by (dim+1) repeat set_!(z, zi, 1)
    z
leviCivitaSymbol() ==
    nz := dim^dim
    z  := new(nz, 0)
    indv: INDEX := new(dim, 0)
    for i in 0..nz-1 repeat
        set_!(z, i, permsign_!(int2index(i, indv))::R)
    z

-- from GradedModule
degree x ==
    rank x

rank x ==
    n := #x
    lengthRankOrElse n

elt(x) ==
    #x ~= 1    => error "Index error (the rank is not 0)"
    get(x,0)
elt(x, i: I) ==
```

```
    #x ~= dim  => error "Index error (the rank is not 1)"
    get(x,(i-minix))
elt(x, i: I, j: I) ==
    #x ~= dim2 => error "Index error (the rank is not 2)"
    get(x,(dim*(i-minix) + (j-minix)))
elt(x, i: I, j: I, k: I) ==
    #x ~= dim3 => error "Index error (the rank is not 3)"
    get(x,(dim2*(i-minix) + dim*(j-minix) + (k-minix)))
elt(x, i: I, j: I, k: I, l: I) ==
    #x ~= dim4 => error "Index error (the rank is not 4)"
    get(x,(dim3*(i-minix) + dim2*(j-minix) + dim*(k-minix) + (l-minix)))

elt(x, i: List I) ==
    #i ~= rank x => error "Index error (wrong rank)"
    n: I := 0
    for ii in i repeat
        ix := ii - minix
        ix<0 or ix>dim-1 => error "Index error (out of range)"
        n := dim*n + ix
    get(x,n)

coerce(lr: List R): % ==
    #lr ~= dim => error "Incorrect number of components"
    z := new(dim, 0)
    for r in lr for i in 0..dim-1 repeat set_!(z, i, r)
    z
coerce(lx: List %): % ==
    #lx ~= dim => error "Incorrect number of slices"
    rx := rank first lx
    for x in lx repeat
        rank x ~= rx => error "Inhomogeneous slice ranks"
    nx := # first lx
    z  := new(dim * nx, 0)
    for x in lx for offz in 0.. by nx repeat
        for i in 0..nx-1 repeat set_!(z, offz + i, get(x,i))
    z

retractIfCan(x:%):Union(R,"failed") ==
    zero? rank(x) => x()
    "failed"
Outf ==> OutputForm

mkOutf(x:%, i0:I, rnk:NNI): Outf ==
    odd? rnk =>
        rnk1  := (rnk-1) pretend NNI
        nskip := dim^rnk1
```

```
        [mkOutf(x, i0+nskip*i, rnk1) for i in 0..dim-1]::Outf
    rnk = 0 =>
        get(x,i0)::Outf
    rnk1  := (rnk-2) pretend NNI
    nskip := dim^rnk1
    matrix [[mkOutf(x, i0+nskip*(dim*i + j), rnk1)
                     for j in 0..dim-1] for i in 0..dim-1]
coerce(x): Outf ==
    mkOutf(x, 0, rank x)

0 == 0$R::Rep
1 == 1$R::Rep

--coerce(n: I): % == new(1, n::R)
coerce(r: R): % == new(1,r)

coerce(v: DP(dim,R)): % ==
    z := new(dim, 0)
    for i in 0..dim-1 for j in minIndex v .. maxIndex v repeat
        set_!(z, i, v.j)
    z
coerce(m: SM(dim,R)): % ==
    z := new(dim^2, 0)
    offz := 0
    for i in 0..dim-1 repeat
        for j in 0..dim-1 repeat
            set_!(z, offz + j, m(i+1,j+1))
        offz := offz + dim
    z

x = y ==
    #x ~= #y => false
    for i in 0..#x-1 repeat
       if get(x,i) ~= get(y,i) then return false
    true
x + y ==
    #x ~= #y => error "Rank mismatch"
    -- z := [xi + yi for xi in x for yi in y]
    z := new(#x, 0)
    for i in 0..#x-1 repeat set_!(z, i, get(x,i) + get(y,i))
    z
x - y ==
    #x ~= #y => error "Rank mismatch"
    -- [xi - yi for xi in x for yi in y]
    z := new(#x, 0)
    for i in 0..#x-1 repeat set_!(z, i, get(x,i) - get(y,i))
```

```
    z
- x ==
    -- [-xi for xi in x]
    z := new(#x, 0)
    for i in 0..#x-1 repeat set_!(z, i, -get(x,i))
    z
n * x ==
    -- [n * xi for xi in x]
    z := new(#x, 0)
    for i in 0..#x-1 repeat set_!(z, i, n * get(x,i))
    z
x * n ==
    -- [n * xi for xi in x]
    z := new(#x, 0)
    for i in 0..#x-1 repeat set_!(z, i, n* get(x,i))  -- Commutative!!
    z
r * x ==
    -- [r * xi for xi in x]
    z := new(#x, 0)
    for i in 0..#x-1 repeat set_!(z, i, r * get(x,i))
    z
x * r ==
    -- [xi*r for xi in x]
    z := new(#x, 0)
    for i in 0..#x-1 repeat set_!(z, i, r* get(x,i))  -- Commutative!!
    z
product(x, y) ==
    nx := #x; ny := #y
    z  := new(nx * ny, 0)
    for i in 0..nx-1 for ioff in 0.. by ny repeat
        for j in 0..ny-1 repeat
            set_!(z, ioff + j, get(x,i) * get(y,j))
    z
x * y ==
    rx := rank x
    ry := rank y
    rx = 0 => get(x,0) * y
    ry = 0 => x * get(y,0)
    contract(x, rx, y, 1)

contract(x, i, j) ==
    rx := rank x
    i < 1 or i > rx or j < 1 or j > rx or i = j =>
        error "Improper index for contraction"
    if i > j then (i,j) := (j,i)
```

```
    rl:= (rx- j) pretend NNI; nl:= dim^rl; zol:= 1;     xol:= zol
    rm:= (j-i-1) pretend NNI; nm:= dim^rm; zom:= nl;    xom:= zom*dim
    rh:= (i - 1) pretend NNI; nh:= dim^rh; zoh:= nl*nm
    xoh:= zoh*dim^2
    xok := nl*(1 + nm*dim)
    z   := new(nl*nm*nh, 0)
    for h in 1..nh _
    for xh in 0.. by xoh for zh in 0.. by zoh repeat
        for m in 1..nm _
        for xm in xh.. by xom for zm in zh.. by zom repeat
            for l in 1..nl _
            for xl in xm.. by xol for zl in zm.. by zol repeat
                set_!(z, zl, 0)
                for k in 1..dim for xk in xl.. by xok repeat
                    set_!(z, zl, get(z,zl) + get(x,xk))
    z

contract(x, i, y, j) ==
    rx := rank x
    ry := rank y

    i < 1 or i > rx or j < 1 or j > ry =>
        error "Improper index for contraction"

    rly:= (ry-j) pretend NNI;  nly:= dim^rly;  oly:= 1;    zoly:= 1
    rhy:= (j -1) pretend NNI; nhy:= dim^rhy
    ohy:= nly*dim; zohy:= zoly*nly
    rlx:= (rx-i) pretend NNI;  nlx:= dim^rlx
    olx:= 1;        zolx:= zohy*nhy
    rhx:= (i -1) pretend NNI;  nhx:= dim^rhx
    ohx:= nlx*dim;  zohx:= zolx*nlx

    z := new(nlx*nhx*nly*nhy, 0)

    for dxh in 1..nhx _
    for xh in 0.. by ohx for zhx in 0.. by zohx repeat
        for dxl in 1..nlx _
        for xl in xh.. by olx for zlx in zhx.. by zolx repeat
            for dyh in 1..nhy _
            for yh in 0.. by ohy for zhy in zlx.. by zohy repeat
                for dyl in 1..nly _
                for yl in yh.. by oly for zly in zhy.. by zoly repeat
                    set_!(z, zly, 0)
                    for k in 1..dim _
                    for xk in xl.. by nlx for yk in yl.. by nly repeat
                        set_!(z, zly, get(z,zly)+get(x,xk)*get(y,yk))
```

```
    z

transpose x ==
    transpose(x, 1, rank x)
transpose(x, i, j) ==
    rx := rank x
    i < 1 or i > rx or j < 1 or j > rx or i = j =>
        error "Improper indicies for transposition"
    if i > j then (i,j) := (j,i)

    rl:= (rx- j) pretend NNI; nl:= dim^rl; zol:= 1;      zoi := zol*nl
    rm:= (j-i-1) pretend NNI; nm:= dim^rm; zom:= nl*dim; zoj := zom*nm
    rh:= (i - 1) pretend NNI; nh:= dim^rh; zoh:= nl*nm*dim^2
    z   := new(#x, 0)
    for h in 1..nh for zh in 0..  by zoh repeat _
    for m in 1..nm for zm in zh.. by zom repeat _
    for l in 1..nl for zl in zm.. by zol repeat _
        for p in 1..dim _
        for zp in zl.. by zoi for xp in zl.. by zoj repeat
            for q in 1..dim _
            for zq in zp.. by zoj for xq in xp.. by zoi repeat
                set_!(z, zq, get(x,xq))
    z

reindex(x, l) ==
    nx := #x
    z: % := new(nx, 0)

    rx := rank x
    p  := mkPerm(rx, l)
    xiv: INDEX := new(rx, 0)
    ziv: INDEX := new(rx, 0)

    -- Use permutation
    for i in 0..#x-1 repeat
        pi := index2int(permute_!(ziv, int2index(i,xiv),p))
        set_!(z, pi, get(x,i))
    z
```

4 package CARTEN2 CartesianTensorFunctions2

⟨*package CARTEN2 CartesianTensorFunctions2*⟩≡

```
)abbrev package CARTEN2 CartesianTensorFunctions2
++ Author: Stephen M. Watt
++ Date Created: December 1986
++ Date Last Updated: May 30, 1991
++ Basic Operations:  reshape, map
++ Related Domains: CartesianTensor
++ Also See:
++ AMS Classifications:
++ Keywords: tensor
++ Examples:
++ References:
++ Description:
++   This package provides functions to enable conversion of tensors
++   given conversion of the components.

CartesianTensorFunctions2(minix, dim, S, T): CTPcat == CTPdef where
    minix:  Integer
    dim:    NonNegativeInteger
    S, T:   CommutativeRing
    CS  ==> CartesianTensor(minix, dim, S)
    CT  ==> CartesianTensor(minix, dim, T)

    CTPcat == with
        reshape: (List T, CS) -> CT
            ++ reshape(lt,ts) organizes the list of components lt into
            ++ a tensor with the same shape as ts.
        map: (S->T,   CS) -> CT
            ++ map(f,ts) does a componentwise conversion of the tensor ts
            ++ to a tensor with components of type T.
    CTPdef == add
        reshape(l, s) == unravel l
        map(f, s)     == unravel [f e for e in ravel s]
```

5 License

⟨*license*⟩≡

```
--Copyright (c) 1991-2002, The Numerical ALgorithms Group Ltd.
--All rights reserved.
--
--Redistribution and use in source and binary forms, with or without
--modification, are permitted provided that the following conditions are
--met:
--
--    - Redistributions of source code must retain the above copyright
--      notice, this list of conditions and the following disclaimer.
--
--    - Redistributions in binary form must reproduce the above copyright
--      notice, this list of conditions and the following disclaimer in
--      the documentation and/or other materials provided with the
--      distribution.
--
--    - Neither the name of The Numerical ALgorithms Group Ltd. nor the
--      names of its contributors may be used to endorse or promote products
--      derived from this software without specific prior written permission.
--
--THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
--IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
--TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
--PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER
--OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
--EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
--PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
--PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
--LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
--NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
--SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

⟨***⟩≡

⟨*license*⟩

⟨*category GRMOD GradedModule*⟩
⟨*category GRALG GradedAlgebra*⟩
⟨*domain CARTEN CartesianTensor*⟩
⟨*package CARTEN2 CartesianTensorFunctions2*⟩

References

[1] nothing