RANK-POLYMORPHIC ARRAYS IN DEPENDENTLY-TYPED LANGUAGES

Artjoms Šinkarovs

7 May 2022

INTRODUCTION

(Demo: rank polymorphism in APL)

In the talk I showed the program generating prime numbers:

```
      a ← ⍳ 30
      a/⍨2=+⌿0=a∘.|a
2 3 5 7 11 13 17 19 23 29
```

TYPE SYSTEM FOR RANK-POLYMORPHIC ARRAYS

- Guarantee lack of out-of-bound access
- Support APL-like combinators
- Inevitable necessity to use dependent types

WHAT IS AN ARRAY TYPE?

Let us attempt to use the `Vec` type.

```
data Vec (X : Set) : ℕ → Set where
  []    : Vec X 0
  _::_  : ∀ {n} → X → Vec X n → Vec X (1 + n)
```

EXPRESS MULTIPLE DIMENSIONS

If the number of dimensions is statically known, we can nest `Vec`s as follows:

```
Mat : Set → ℕ → ℕ → Set
Mat X m n = Vec (Vec X n) m

_⊕_ : ∀ {m n} → (a b : Mat ℕ m n) → Mat ℕ m n
a ⊕ b = …
```

EXPRESS MULTIPLE DIMENSIONS

If the number of dimensions is statically known, we can nest `Vec`s as follows:

```
Mat : Set → ℕ → ℕ → Set
Mat X m n = Vec (Vec X n) m

_⊕_ : ∀ {m n} → (a b : Mat ℕ m n) → Mat ℕ m n
a ⊕ b = …
```

How do we handle rank-polymorphic functions?

RANK POLYMOPRHISM THROUGH NESTING

Let us define a function that computes the n-fold nesting of `Vec`s based on the array shape:

```
Tensor : Set → List ℕ → Set
Tensor X [] = X
Tensor X (x ∷ s) = Vec (Tensor X s) x
```

RANK-POLYMORPHIC EXAMPLE

We can use `Tensor` to specify as follows:

```
_⊕'_ : ∀ {s} → (a b : Tensor ℕ s) → Tensor ℕ s
a ⊕' b = …

ten-2×3 : Tensor ℕ (2 ∷ 3 ∷ [])
ten-2×3 = ((1 ∷ 2 ∷ 3 ∷ []) ∷
           (4 ∷ 5 ∷ 6 ∷ []) ∷ [])

test-⊕' : Tensor ℕ (2 ∷ 3 ∷ [])
test-⊕' = ten-2×3 ⊕' ten-2×3
```

TRY TO WRITE A PROGRAM

(Demo with `Tensor`)

In the talk I showed how to write a matrix-multiply program using `Tensor` encoding:

```
postulate
  mtrans : ∀ {m n} → Mat ℕ m n → Mat  ℕ n m

matmul : ∀ {m n k} → Mat ℕ m n → Mat ℕ n k
       → Mat ℕ m k
matmul a b = map (λ r → map (λ c →
  sum (zipWith _*_ r c)) (mtrans b)) a
```

REPRESENTABILITY

We know that `Vec` is represented by `Fin n`:

```
data Fin : ℕ → Set where
  zero : ∀ {n} → Fin (1 + n)
  suc  : ∀ {n} → Fin n → Fin (1 + n)
```

which means that `(Fin n → X) ≅ Vec X n`

REPRESENTABILITY

We know that `Vec` is represented by `Fin n`:

```
data Fin : ℕ → Set where
  zero : ∀ {n} → Fin (1 + n)
  suc  : ∀ {n} → Fin n → Fin (1 + n)
```

which means that `(Fin n → X) ≅ Vec X n`

Can we find a representation for `Tensor X n`?

ARRAYS AS FUNCTIONS

```
data Ix : List ℕ → Set where
  []  : Ix []
  _::_ : ∀ {n s} → Fin n → Ix s → Ix (n :: s)

Ar : Set → List ℕ → Set
Ar X s = Ix s → X
```

EXAMPLES (1)

```
sum : ∀ {n} → Ar ℕ (n ∷ []) → ℕ
sum = …

matmul : ∀ {m n k}
       → Ar ℕ (m ∷ n ∷ [])
       → Ar ℕ (n ∷ k ∷ []) → Ar ℕ (m ∷ k ∷ [])
matmul a b (i ∷ j ∷ []) = sum λ {
  (k ∷ []) → a (i ∷ k ∷ []) * b (k ∷ j ∷ [])
  }
```

EXAMPLES (2)

```
raise : ∀ {m} n → Fin m → Fin (n + m)
raise zero    i = i
raise (suc n) i = suc (raise n i)

drop : ∀ {m s X} → (n : ℕ)
     → Ar X ((n + m) ∷ s) → Ar X (m ∷ s)
drop n a (i ∷ ix) = a (raise n i ∷ ix)
```

TRANSPOSE

```
rev∘rev : ∀ {s : List ℕ} → reverse (reverse s) ≡ s
rev∘rev = …

rev-ix : ∀ {s} → Ix s → Ix (reverse s)
rev-ix ix = …

transpose : ∀ {X s} → Ar X s → Ar X (reverse s)
transpose a ix = a (subst Ix rev∘rev (rev-ix ix))
```

TRANSPORTING OVER EQUALITIES

If we have p ≡ q, we can transport Ix (or any other type over it):

```
transp-ix : ∀ {s p} → s ≡ p → Ix s → Ix p
transp-ix refl ix = ix

transp-ar : ∀ {X s p} → s ≡ p → Ar X s → Ar X p
transp-ar refl a = a
```

However, what happens if we don't have equality, but we have an isomorphism? Given s ≅ p, can we derive

GENERALISATION

The key point of this construction is a monoidal universe:

```
record MonoidalUniv : Set₁ where
  field
    U : Set
    El : U → Set

    ι : U
    _⊗_ : U → U → U

    el-ι : El ι ↔ ⊤
    el-⊗ : ∀ {a b} → El (a ⊗ b) ↔ (El a × El b)
```

MULTIPLE DIMENSIONS

```
module _ (M : MonoidalUniv) where
  open MonoidalUniv M

  split : ∀ {a b} → All El (a ++ b)
          → All El a × All El b
  join  : ∀ {a b} → All El a × All El b
          → All El (a ++ b)

  ranked : MonoidalUniv
  MonoidalUniv.U ranked = List U
  MonoidalUniv.El ranked = All El
  MonoidalUniv.ι ranked = []
  MonoidalUniv._⊗_ ranked = _++_
  MonoidalUniv.el-ι ranked = …
  MonoidalUniv.el-⊗ ranked = …
```

RESHAPES (1)

```
module _ (M : MonoidalUniv) where
  module M = MonoidalUniv M
  module R = MonoidalUniv (ranked M)

  data Reshape : R.U → R.U → Set where
    eq : ∀ {s} → Reshape s s
    _⊙_ : ∀ {s p q} → Reshape p q → Reshape s p
      → Reshape s q
    rflat : ∀ {m n s p} → Reshape s p
      → Reshape (m ∷ n ∷ s) (m M.⊗ n ∷ p)
    rjoin : ∀ {m n s p} → Reshape s p
      → Reshape (m M.⊗ n ∷ s) (m ∷ n ∷ p)
    rswap  : ∀ {m n s p} → Reshape s p
      → Reshape (m ∷ n ∷ s) (n ∷ m ∷ p)
    prep : ∀ {m s p} → Reshape s p
      → Reshape (m ∷ s) (m ∷ p)
```

RESHAPES (2)

```
rev : ∀ {a b} → Reshape a b → Reshape b a
rev = …

_⟨_⟩ : ∀ {a b}
     → R.El a → Reshape b a → R.El b
ix ⟨ eq ⟩ = ix
ix ⟨ r ⊙ r₁ ⟩ = ix ⟨ r ⟩ ⟨ r₁ ⟩
(i ∷ ix) ⟨ rflat r ⟩ =
  let p , q = Inverse.f M.el-⊗ i
  in p ∷ q ∷ (ix ⟨ r ⟩)
(i ∷ j ∷ ix) ⟨ rjoin r ⟩ =
  Inverse.f⁻¹ M.el-⊗ (i , j) ∷ (ix ⟨ r ⟩)
(i ∷ j ∷ ix) ⟨ rswap r ⟩ = j ∷ i ∷ (ix ⟨ r ⟩)
(i ∷ ix) ⟨ prep r ⟩ = i ∷ (ix ⟨ r ⟩)
```

GENERALISATION

We can play the same game with Σ-universes:

```
record SigmaUniv : Set₁ where
  field
    U : Set
    El : U → Set

    ι : U
    σ : (a : U) → (El a → U) → U

    el-ι : El ι ↔ ⊤
    el-σ : ∀ {a b} → El (σ a b)
         ↔ (Σ[ x ∈ El a ] (El (b x)))
```

MULTIPLE DIMENSIONS

Multiple dimensions is given by the following data structure:

```
module _ (S : SigmaUniv) where
  open SigmaUniv S

  data Σn : Set
  AllΣn : Σn → Set

  data Σn where
    1d : U → Σn
    nd : (x : Σn) → (AllΣn x → Σn) → Σn

  AllΣn (1d x) = El x
  AllΣn (nd x f) = Σ (AllΣn x) (AllΣn ∘ f)
```

CONCLUSIONS

- Typed arrays operations with safe indexing are possible;
- Container-like approach preserves “natural” reasoning in terms of indices and do not commit to any representation;
- Monoidal universes give rise to many nice properties including flattening and element-preserving reshapes;
- Mappable to parallel architectures;
- Generalisation to non-homogeneous arrays.

THANK YOU