Rank-polymorphic arrays within dependent types

Artjoms Šinkarovs[1]*and Sven-Bodo Scholz[2]

[1] Heriot-Watt University
a.sinkarovs@hw.ac.uk
[2] Radboud University
svenbodo.scholz@ru.nl

Many array languages such as APL [6], J [10], Futhark [4], or SaC [9] cater for multi-dimensional arrays as first class citizens. These languages have two main advantages. Firstly, they give rise to concise specifications of numerical algorithms that use array combinators rather than explicit indexing as often found in imperative languages such as Fortran or C. Secondly, as arrays have a very regular structure, many computations on arrays can be automatically parallelised, which leads to efficient executions on a range of parallel platforms [2, 3, 8, 5].

Rank polymorphism is the ability of functions to be applied to arrays of arbitrary ranks. Rank polymorphism is important for two reasons. Firstly, it gives rise to more general array combinators such as map, fold, take, transpose, *etc.* Secondly, the structure of the nesting can be used to enforce non-trivial traversals through sub-arrays which is often the basis for advanced parallel algorithms such as scan or blocked matrix multiply.

In this talk we present how rank-polymorphic arrays can be embedded within a dependently-typed language. On the one hand, our embedding offers the generality of the specifications found in array languages. On the other hand, we guarantee safe indexing and offer a way to reason about concurrency patterns within the given algorithm.

We present the key ingredients of the array framework in Agda. We start with the definition of an array theory.

```
record Array : Set₁ where
  field
    S : Set
    P : S → Set
    ι : ℕ → S
    _⊗_ : S → S → S
    ι-↔ : ∀ {n} → P (ι n) ↔ Fin n
    ⊗-↔ : ∀ {s p} → P (s ⊗ p) ↔ (P s × P p)
    Ar : S → Set → Set
    Ar-↔ : ∀ {s X} → (P s → X) ↔ Ar s X
```

Array shapes S are binary trees with natural numbers as leaves. Array indices P are indexed by shapes, representing trees of natural numbers of the same shape as the index, but where all leaves are component-wise smaller than the shape components. For example, for some shape $(\iota\ a \otimes (\iota\ b \otimes \iota\ c))$ the index is of the form $(i,(j,k))$ where $i < a$, $j < b$ and $k < c$. Array theory does not insist on a particular implementation of S and P but it requires the chosen implementation to be isomorphic to such trees (ι-↔ and ⊗-↔). Finally, arrays (Ar) are indexed by the shape and the element type, and we ask that arrays are representable functors (Ar-↔).

By expanding isomorphisms in the array theory, we get a number of useful array combinators as model constructions. For some $(A : \mathsf{Array})$, we have:

*This work is supported by the Engineering and Physical Sciences Research Council through the grant EP/N028201/1.

```
imap : ∀ {s} → (P s → X) → Ar s X
_[_]   : ∀ {s} → Ar s X → (P s → X)
nest : ∀ {s p} → Ar (s ⊗ p) X → Ar s (Ar p X)
unnest : ∀ {s p} → Ar s (Ar p X) → Ar (s ⊗ p) X
```

By noticing that for a fixed shape s Ar is an applicative functor [7], we obtain operations like:

```
map : ∀ {s} → (X → Y) → Ar s X → Ar s Y
zipWith : ∀ {s} → (X → Y → Z) → Ar s X → Ar s Y → Ar s Z
```

Next, we define an inductive reshape relation that gives rise to reversible permutations of array elements. There can be various reshaping relations allowing more or less liberal permutations.

```
data Reshape : S → S → Set
```

The Reshape relation gives rise to actions on array indices and arrays themselves:

```
_⟨_⟩ : ∀ {s p} → P p → Reshape s p → P s
reshape : ∀ {a b X} → Reshape a b → Ar a X → Ar b X
```

We finish this abstract by presenting the simplest example that demonstrates the power of the proposed framework — a blocked matrix-vector multiplication. We work in the initial model of our array theory, and we assume that we have two functions (_⊠_ : X → Y → Z) and (sum : ∀ {s} → Ar s Z → Z). Then, a straight-forward matrix-vector multiplication is given by mat-vec-canon. We define the blocked version mat-vec by running induction on s. When s is a singleton we use the canonical multiplication defined earlier, but when s is a product $(s \otimes p)$, we block the matrix into s matrices of p rows and apply *mat-vec* recursively on each block. All these recursive applications can be executed in parallel, as there are no dependencies between the parts.

```
mat-vec-canon : Ar (s ⊗ ι n) X → Ar (ι n) Y → Ar s Z
mat-vec-canon a v = imap λ i → sum $ imap λ k → a [ i ⊗ k ] ⊠ v [ k ]

mat-vec : Ar (s ⊗ ι n) X → Ar (ι n) Y → Ar s Z
mat-vec {s = ι m}     a v = mat-vec-canon a v
mat-vec {s = s ⊗ p} a v = unnest $ map (flip mat-vec v) (nest $ tile a)
```

We can demonstrate that our blocked algorithm computes the same results (using point-wise equality $\_\approx_a\_$); and that the blocked algorithm is stable under reshapes. That is, for all possible reshapes, computing blocked mat-vec on a reshaped array is the same as computing mat-vec on the original array and then performing the reshape.

```
mat-vec-ok : (a : Ar (s ⊗ ι n) X) → (v : Ar (ι n) Y) → mat-vec a v ≈ₐ mat-vec-canon a v
mat-vec-stable : (r : Reshape s p) → (a : Ar (s ⊗ ι n) X) → (v : Ar (ι n) Y)
                           → mat-vec (reshape (r ⊕ eq) a) v ≈ₐ reshape r (mat-vec a v)
```

In practice this means, that we can use array reshaping as a vehicle to control which sub-arrays will be executed in parallel. In the talk we will make this idea precise, explaining how exactly one can reason about parallel execution.

We conclude with the observation that the presented array theory is very similar to categories with families [1] which are often used to define type theories. In this analogy, contexts are shapes, substitutions are reshapes, and well-scoped terms are arrays.

References

[1] Marc Bezem, Thierry Coquand, Peter Dybjer, and Martín Escardó. On generalized algebraic theories and categories with families. *Math. Struct. Comput. Sci.*, 31(9):1006–1023, 2021.

[2] Clemens Grelck. Shared memory multiprocessor support for functional array processing in sac. *Journal of Functional Programming*, 15(3):353–401, 2005.

[3] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. Breaking the gpu programming barrier with the auto-parallelising sac compiler. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, page 15–24, New York, NY, USA, 2011. Association for Computing Machinery.

[4] Troels Henriksen. *Design and Implementation of the Futhark Programming Language*. PhD thesis, University of Copenhagen, Universitetsparken 5, 2100 København, 11 2017.

[5] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 556–571, New York, NY, USA, 2017. ACM.

[6] Kenneth E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., New York, NY, USA, 1962.

[7] CONOR MCBRIDE and ROSS PATERSON. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

[8] Trevor L. McDonell, Manuel M T Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising Purely Functional GPU Programs. In *ICFP '13: The 18th ACM SIGPLAN International Conference on Functional Programming*. ACM, September 2013.

[9] Sven-Bodo Scholz. Single assignment c: Efficient support for high-level array operations in a functional setting. *J. Funct. Program.*, 13(6):1005–1059, November 2003.

[10] Roger Stokes. Learning j. an introduction to the j programming language. `http://www.jsoftware.com/help/learning/contents.htm`, 2015. [Accessed 2023/02].