MoA core Magnolia API

Marius Kleppe Larnøy

February 15, 2022

Here we outline the core MoA Magnolia API, including padding operations. For now all components are `implementation` modules, reason being the convenience of producing working code faster. The implementations can easily be enhanced with corresponding `concept`s when the core structure is in place.
Rather than going really abstract top-level when designing the API, this specification will be developed more with a concrete implementation in mind.
The core API will be built upon a small set of external operations for accessing and modifying arrays.

1 External API

- A struct `Array` for representing an Array. Stores the elements of the array and its shape.
- A struct `PaddedArray` inheriting `Array`. In addition to storing the contents, `PaddedArray` stores both the unpadded and the padded shape
- MoA concepts of shape and index is represented by their own typedefs, aliasing `std::vector<size_t>`
- Getters/setters with direct array access
- Utility functions, like wrappers/unwrappers, (safe) conversion between types
- Constructors for arrays, shapes, indices
- IO, (pretty)printing, for debugging
- templates for looping
- structs for Element types (e.g. Int32, Float64)

2 Core concepts/signatures/implementations

2.1 `external implementation ExtOps`

The Magnolia side of the external C++ API.
Types:

- Array: unpadded array type
- PaddedArray: padded array type
- Element (required): Generic element type (assumed to be either an integer type or a floating point type)
- Index: Index type
- UInt32 (might change): unsigned int type
- Shape: the shape type

Getters/setters

- `function get(a: Array, i: Index): Array`
- `procedure set(upd a: Array, obs i: Index, obs e: Element`

Unary operations

- `function dim(a: Array): UInt32`
- `function shape(a: Array): Shape`
- `function total(a: Array): UInt32`
- `function total(s: Shape): UInt32`

Various util, array creation, and IO functions

2.2 `implementation Concatenation`

Defines catenation on both 1-dim arrays (vectors) and n-dim arrays

- `funcion cat_vec(vec1:Array, vec2:Array):Array guard dim(vec1) == one() && dim(vec2) == one();`
- `function cat(a1: Array, a2: Array):Array guard drop(one(), shape(a1)) == drop(one(), shape(a2));`

2.3 `implementation Padding`

Defines operations for circular padding on n-dim arrays

- `function circular_padr(a: Array, ix: UInt32):PaddedArray`
- `function circular_padl(a: Array, ix: UInt32):PaddedArray`
- **TODO: finish**

2.4 Implementations for more required operations...

2.5 `concept DNF`

DNF transformation rules

2.6 `concept ONF`

Transforming DNF expressions with specific hardware in mind

3 General notes, thoughts

- Question: overload functions like total on argument type, or define separate functions for different types. E.g. total of a shape neccessary, or just design shape signature with a function that yields the total
- Question: specify Util concepts like Int and abstract structures like Semigroup/Monoid. Not part of core moa, but useful
- Question: Would it be any benefit to further split up the concepts? e.g. some functions in one concept, padding separated maybe? not sure
- Follow-up: Might be best to define operations closer to their intended arguments. The upside to doing it like this is that we keep the signatures lean and clean, and dont have to import redundant signatures for defining operations. Also avoid circular dependencies
- Follow-up2: separate DNF and ONF level operations?
- some sort of Map concept? more of a underlying concept, how does the operations actually apply to arrays
- Note: Separating the types, while bringing some much needed distinctions between elements, brings with it its own set of issues. Neccessarily it leads to many of the functions requiring overloading, bringing redundancy into the code. Might need some design rethinking to avoid it, or maybe it is a neccessary evil.
- Finding it a bit hard to distinguish between which functions are staying in C++ and those that have to be implemented in Magnolia. Why even leave implementation to Mg when it can be done in Cpp
-
  - **Benjamin:** For example, we can not apply rewrite rules on backend code. For axiom-based rewriting, we need to write in Magnolia. Also, our language is more restricted, which allows for different kinds of transformations that would not be possible in C++.
- Only template types have keyword required? Makes sense as other types are provided

- – **Benjamin:** Required = this external block is well-defined if this type is provided. Not required (in external blocks) = this external block, once well-defined, provides you with this type

- Design choice, limit C++ implemented functions to basic access, define the rest via moa in Magnolia

- Loops in Magnolia