Two Dimensional Numerical Representations An Adventure with Block Matrices

Michael Youssef[1]

Rhineland Palatinate Technical University Kaiserslautern-Landau, Kaiserslautern, Germany
youssef@rptu.de

Abstract. Numerical representation is a technique for modelling datastructures after number systems. Typically and historically, the method was primarily used to derive one-dimensional datastructures. This paper explores utilizing the technique for modelling two-dimensional datastructures instead, with insights into multi-dimensional ones. The aim of this paper is to devise data structures for modelling block matrices. We look into the different derivations and highlight some roadblocks that would otherwise not be an issue in the one-dimensional case. The metrics at considerations include dimensional-bias, ease of defining functions and proving properties about them as well as the easiness of understanding of the datastructure. Keypoint : Numerical Representations, Block Matrices, Agda

Keywords: Block Matrices · Numerical Representations · Agda.

1 Introduction

Numerical representations are a well-established concept that has been around for quite a while. Most notably, Okasaki [3] introduced a number of datastructures that are based on various number systems. Over the last three decades, there has been a lot of work done to derive one-dimensional datastructures using this method. What has not been explored much is using the same techniques for two-dimensional datastructures. In fact, and as we show in some of the sections of this paper, some of these datastructures are subtly hidden in recent works and well-established ones alike. The aim of this paper is to highlight the different challenges faced when transitioning to two-dimensional structures that would otherwise not be an issue when dealing with the one dimensional structures, as well as deriving a suitable datastructure for modelling block matrices which is ideally easy to understand, and easy to work with and reason about.

Contributions This paper makes the following contributions.

- Explores the design space of two-dimensional numerical representation in the context of block matrices.

- Highlights the relevant design decisions when it comes to transitioning from one-dimensional to two-dimensional structures.
- Reasons about a suitable implementation of block matrices with regards to difficulty of proving and programming with the corresponding structure in the context of semiring frameworks.

2 Numerical Representations

To start of, we recap the one-dimensional case. Deriving the datastructures can be done in multiple ways. For example, Okasaki [3] does it in an ad-hoc fashion, while some of the works that followed had done it more rigorously. Hinze [1] used functor equations to derive numerical representation. While more recently, Hinze and Swierstra [2] used isomorphism equational reasoning to derive such representations using the laws of exponents. They also proposed that this method can be used for deriving two or multi-dimensional representations. This paper focuses on extending their work in that direction. For the purpose of this paper, we use their framework.

To that end, we recap the main idea of the framework. The starting point is defining a finite map from some finite set to an arbitrary type A.

```
sequence : Index n → A
```

Index is some representation of a finite set and the entire function type represents a finite sequence type of length n. The main idea is to vary the representation of the *Index* implementation such that different number systems can yield different representation of the sequence type itself.

For example, if we use the standard library's finite sets *Fin*, we get vanilla lists as the corresponding sequence type by solving the isomorphism equation $(Index\ n \to elem) \cong Array\ n\ elem$ between the function type and the corresponding datastructure to be derived.

```
data Fin : ℕ → Set where
  zero :                Fin (suc n)
  suc  : (i : Fin n) → Fin (suc n)
```

```
data Array : Peano → Set → Set where
  nil  :                          Array zero     elem
  cons : elem → Array n elem → Array (succ n) elem
```

2.1 From One to Two Dimensional Representations

The idea behind a two or multi-dimensional derivation is to use a multi dimensional finite map instead. For example, if we solve the equation
$(Index\ n \to Index\ n \to elem) \cong PMatrix\ n\ elem$. We can obtain a two-dimensional

numerical representation based on the same index type modelling square matrices.

```
data PMatrix : Peano → Set → Set where
  empty : PMatrix zero A
  block : A → Array n A → Array n A
                → PMatrix n A
                → PMatrix (succ n) A
```

3 Exploration of The Design Space

In this section we explore some of the common and likewise, less common number systems with regards to two-dimensional representations. We highlight some considerations and how they affect the resulting datastructures. Since we intend to use the matrices within semiring frameworks, the main focus will be about square matrices, however, sometimes it is necessary to drop that restriction.

3.1 Row vs Column Major vs Neutral

One immediate method of approaching the two-dimensional datastructures is to model them as one-dimensional ones with their element types again being one-dimensional datastructures. In other words, we can solve the equations as follows

$$
\begin{aligned}
&(\mathit{Index}\ m \to (\mathit{Index}\ n \to \mathit{elem}))\\
\cong\ &\\
&(\mathit{Index}\ m \to \mathit{Array}\ n\ \mathit{elem})\\
\cong\ &\\
&\mathit{Array}\ m\ (\mathit{Array}\ n\ \mathit{elem})
\end{aligned}
$$

Although this approach always works for any number system, This is essentially the naive approach which still exists in some main stream languages that models matrices as sequences of sequences. The fundamental issue with this approach is the dimensional-bias. In essence, row or column major operations could be substantially more difficult depending on which dimension we favor. One such operation which would be non-trivial under both schemes, row or column-major, would be the matrix *transpose*. Moreover, if we are to respect dimensional constraints, such approach is only possible in a dependent setting, since we can control the size of the inner sequence w.r.t to the outer one using size indices.

3.2 Empty row or column size

Yet another roadblock is the empty sequence. In the one dimensional case, that is not an issue, since there is a single numerical value representing the size of the datastructure, however, For the two dimensions case, considering some type of matrices indexed by its row and column dimensions, the types *Matrix* 0 *n elem*, *Matrix* 0 *n elem* and *Matrix* 0 *n elem* are different, yet model the same thing, the empty matrix. We can opt to use nested datatypes to do away with the indices while respecting the dimensional constraints similar to how Okasaki describes top down and bottom up matrices [4], however, nested datatypes are not always easy to comprehend and work with, specially if we move to a two-dimensional setting.

Furthermore, this is not always an issue as we have briefly mentioned before that we are primarily interested in square matrices, however, as we demonstrate later on, some representations of square block matrices require rectangular sub-matrices.

One way out of that is to use number systems that don't admit a zero representation. This would typically work in most applications since the base case of a block matrix is usually assumed to be a scalar in many applications.

For example, an adjusted index based on the Peano numerals would be the following

```
data AtMost : Peano → Set where
  izero :                AtMost n
  isucc : AtMost n → AtMost (succ n)
```

We will stick to non-emptiness for the remainder of the paper unless otherwise stated.

3.3 Common Number Systems

Peano First of, we start with Peano numerals. The one-dimensional structure we get using the index scheme described in Section 3.2 is simply, a non-empty vanilla list. Similarly, the two-dimensional counterpart is not so different, we essentially get two constructors for representing scalars as well as non-trivial $2 + n \times 2 + n$ matrices.

```
data AMatrix : Peano → Set → Set where
  scalar : elem → AMatrix 0  elem
  block  : elem → Array   m elem → Array    m        elem
              → AMatrix m elem → AMatrix (succ m) elem
```

This simple and easy to work with datastructure suffices for a lot of applications, however, it is not necessarily the easiest when it comes to proofs.

Leibniz Leibniz numerals are essentially a unique binary representation as opposed to standard binary numerals that allow for leading zeros and suffer from a non-unique representation.

```
data Leibniz : Set where
  𝟘b  : Leibniz
```

```
_𝟙 : Leibniz → Leibniz
_𝟚 : Leibniz → Leibniz
```

The one-dimensional derivations of Leibniz numerals give us Braun trees or one-two trees (a.k.a. lazy Braun trees) As Hinze and Swierstra have shown [2]. In principle, Leibniz numerals distinguish between even and odd lengths, this translates to a duplicity of constructors in the corresponding datastructure. For example, for Braun trees, we get three constructors. Likewise, if we derive Braun matrices, we get nine constructors...

```
data BM  : Leibniz → Leibniz → Set → Set where
  scalar    : e → BM 𝟘b 𝟘b e
  column₁   : BM l          𝟘b e → BM l 𝟘b e → BM (l 𝟙)  𝟘b     e
  column₂   : BM (lsucc l) 𝟘b e → BM l 𝟘b e → BM (l 𝟚)  𝟘b     e
  row₁      : BM 𝟘b l          e → BM 𝟘b l e → BM 𝟘b     (l 𝟙) e
  row₂      : BM 𝟘b (lsucc l) e → BM 𝟘b l e → BM 𝟘b     (l 𝟚) e
  block₂₁   : BM (lsucc l₁) l₂          e → BM (lsucc l₁) l₂ e
    →         BM l₁          l₂          e → BM l₁          l₂ e
    →         BM (l₁ 𝟚) (l₂ 𝟙) e
  block₁₂   : BM l₁          (lsucc l₂) e → BM l₁          l₂ e
    →         BM l₁          (lsucc l₂) e → BM l₁          l₂ e
    →         BM (l₁ 𝟙) (l₂ 𝟚) e
  block₂₂   : BM (lsucc l₁) (lsucc l₂) e → BM (lsucc l₁) l₂ e
    →         BM l₁          (lsucc l₂) e → BM l₁          l₂ e
    →         BM (l₁ 𝟚) (l₂ 𝟚) e
  block₁₁   : BM l₁  l₂ e   → BM l₁ l₂ e
    →         BM l₁  l₂ e   → BM l₁ l₂ e
    →         BM (l₁ 𝟙) (l₂ 𝟙) e
```

In this case of Braun matrices, we can't opt to have square matrices only since some of the sub-matrices have dimensions differing by one.

Unfortunately, when it comes to Leibniz numerals, the duplicity scales up as we add more dimensions. For example, scaling up to a third dimension or more would cause an explosion in the number of cases with this implementation.

Moreover, implementing matrix multiplication for Braun matrices, which is arguably the most important matrix operation, would require a whopping 27 cases.

One way to tackle this issue is instead of having a single datastructure with all possibilities of splitting rows and columns, we can have two mutually recursive data structures instead. This approach typically yields more constructors in total, but less per each datastructure. We will do demonstrate the approach with one of the examples that follow.

3.4 Implementations Based on Leaf Trees

Perhaps a less obvious numerical representation is the usage of tree number systems, particularly, leaf trees. A basic implementation of such datastructure is described using two constructors, one for the leaf itself and another for a binary split. We model such datastructure as follows.

```
data Shape : Set where
  O : Shape
  B : Shape → Shape → Shape


fromShape : Shape → Peano
fromShape O       = succ zero
fromShape (B x y) = fromShape x + fromShape y
```

This datastructure suffers from non-uniqueness. This is however not necessary a major issue as the point here is to have flexibility when splitting a matrix into blocks. A non-unique representation allows for such flexibility. On the other hand, the resulting data structures are typically not easy to work with or reason about.

The index type corresponding to this number system is the following.

```
data SIndex : Shape → Set where
  one   : SIndex O
  left  : SIndex l → SIndex (B l r)
  right : SIndex r → SIndex (B l r)
```

Surprisingly, this is the most common two-dimensional numerical base for already existing block matrix datastructures. We cover some possible derivations below.

Q Trees Eriksson and Jansson [5] used a representation of block matrices based on leaf trees to model the transitive closure operation using semirings and other similar algebraic structures. The datastructure they used is the following.

```
data M : Shape → Shape → Set → Set where
  scalar : elem                          → M O          O          elem
  row    : M O c₁ elem → M O c₂ elem → M O          (B c₁ c₂) elem
  column : M r₁ O elem → M r₂ O elem → M (B r₁ r₂) O          elem
  block  : M r₁ c₁ elem → M r₁ c₂ elem →
             M r₂ c₁ elem → M r₂ c₂ elem → M (B r₁ r₂) (B c₁ c₂) elem
```

This implementation is quite flexible, row-column neutral and easy to understand, however, doing any non-trivial proof requires substantial effort.

KD-trees We have previously noted before that in the two-dimensional case, the number of constructors can increase substantially and pointed out that one way to tackle this is to use mutual recursion. If we apply this idea to leaf trees, we get a datastructure similar to KD-trees, or 2D-trees in the two-dimension case.

```
data KDTreeR : Shape → Shape → Set → Set
data KDTreeC : Shape → Shape → Set → Set
data KDTreeR where
  single    : elem                          → KDTreeR O O elem
  split-col : KDTreeR O c₁         elem → KDTreeR O c₂ elem
    →         KDTreeR O (B c₁ c₂) elem
  split-row : KDTreeC r₁        c  elem → KDTreeC r₂ c  elem
    →         KDTreeR (B r₁ r₂) c  elem
data KDTreeC where
  single    : elem                          → KDTreeC O O elem
  split-row : KDTreeC r₁        O elem → KDTreeC r₂ O elem
    →         KDTreeC (B r₁ r₂) O elem
  split-col : KDTreeR r c₁         elem → KDTreeR r  c₂ elem
    →         KDTreeC r (B c₁ c₂) elem
```

Of course, the main idea of KD-Trees is to have sparsity. We can solve this by adding additional constructors. For example, we can augment the row-biased KD-Tree by a sparse constructor as a constructor *sparse* : *KDTreeR m n elem* and similarly, *sparse* : *KDTreeC m n elem* for the column-biased one. We can additionally mark the sparse constructors with additional information. For example, we could wish to have an identity sparse matrix as well as a zero sparse matrix *zero* : *KDTreeR m n elem* and *id* : *KDTreeR m m elem* and similarly, *zero* : *KDTreeC m n elem* and *id* : *KDTreeC m m elem*.

This scheme for modelling sparsity in a datastructure is simple, easy to understand and always works, however, we quickly stumble upon a major issue... Propositional equality no longer works. At this point we may consider sacrificing the clarity of this simple method over a clever way of representing sparsity, nevertheless, we will address this problem using a different approach at a later point.

Multiway Split The leaf trees we have chosen are actually binary leaf trees. Generally speaking, nothing is preventing us from having ternary or even a multi-way split instead. If we opt for a multi-way splitm, we can derive a datastructure which allows for an arbitrary splitting scheme. This is perhaps as flexible as it gets. We can split a matrix into an arbitrary number of sub-matrices with arbitrary sizes.

For example, we can model a number system using multiway leaf trees using vanilla lists and derive a corresponding index type as follows.

```
data Array (A : Set) : Peano → Set where
  nil  : Array A zero
  cons : A → Array A n → Array A (succ n)
data Shape : Set where
  M : Array Shape n → Shape
```

```
data SIndex : Shape → Set where
  leaf : SIndex (M nil)
  here : SIndex x        → SIndex (M (cons x xs))
  next : SIndex (M xs) → SIndex (M (cons x xs))
```

The corresponding one-dimensional datastructure would be the following

```
data SArr : Shape → Set → Set where
  leaf : elem                                → SArr (M nil) elem
  cons : SArr x elem → SArr (M xs) elem → SArr (M (cons x xs)) elem
```

In general, the leaf tree scheme allows us to model the datastructure by its structural shape. The multiway leaf tree scheme is the least restrictive where the shape itself it some sequence type based on how we instantiate the *Array* type used in the multi-way split.

2D derivation WIP

3.5 Skew Binary

The motivation behind using Skew binary is simple, for most implementations of Block matrices, the proofs are typically quite technically involved without much insight into the core logic of the proof itself. To clarify, the vast majority of the proof effort goes towards navigating the structure of the matrices themselves rather than making any profound logical arguments. One implementation of square block matrices where proofs are free of structural hurdles is the perfectly balanced variant. The core idea is that we can embed any squad matrix into a perfectly balanced square matrix of a larger size, then, we can simply restrict any equational reasoning to perfectly balanced square matrices. This setting requires a type of block matrices which has two properties.

– Can easily model perfectly balanced matrix subtypes.
– Has an easy implementation of padding.

In terms of number systems, the first requirement begs for a binary number system based approach. As for the second, we first elaborate that padding is just the two-dimensional variant of the *cons* operation in lists. In terms of number systems, this would be the *increment* operation. A numerical representation based on Peano naturals would be a good choice here, however, a skew binary implementation is the obvious choice which satisfies both requirements.

To start of, we need a datastructure for skew binary numbers. Ideally, canonical skew binary numbers. Myers REFERENCE proved that a variant of skew binary numbers that only admit the first non-zero numeral to be a 2 is unique. The following datastructure captures that.

```
data Skew¹ : Set where
  0 b      : Skew¹
  _𝟙_𝟘s    : Skew¹ → Peano → Skew¹
```

```
data Skew  : Set where
  0 b𝟘     : Skew
  _𝟙_𝟘s    : Skew¹ → Peano → Skew
  _𝟚_𝟘s    : Skew¹ → Peano → Skew
⟦_,_⟧¹ : Skew¹ → Peano → Peano
⟦ 0 b          , n  ⟧¹   = zero
⟦ x 𝟙 m 𝟘s     , n  ⟧¹   = 2 ↑ (n + m) ∸ 1 + ⟦ x , succ (n + m) ⟧¹
⟦_⟧ : Skew → Peano
⟦ 0 b𝟘        ⟧ = zero
⟦ x 𝟙 m 𝟘s    ⟧ = 2 ↑ (succ m)          ∸ 1 + ⟦ x , succ (succ m) ⟧¹
⟦ x 𝟚 m 𝟘s    ⟧ = 2 ↑ (succ (succ m)) ∸ 2 + ⟦ x , succ (succ m) ⟧¹
```

The corresponding index type we get is the following.

```
data TreeIndex  : Peano → Set where
  Node  : TreeIndex h
  L     : TreeIndex h →  TreeIndex (succ h)
  R     : TreeIndex h →  TreeIndex (succ h)
data SkewIndex¹  : Skew¹ → Peano → Set where
  single  :                                  SkewIndex¹ 0 b        n
  node    : TreeIndex    (succ (n + x))   → SkewIndex¹ (l 𝟙 x 𝟘s) n
  rest    : SkewIndex¹ l (succ (n + x))   → SkewIndex¹ (l 𝟙 x 𝟘s) n
data SkewIndex  : Skew → Set where
  scalar  : SkewIndex 0 b𝟘
  T₁   :    TreeIndex n      → SkewIndex (l 𝟙 n 𝟘s)
  Ts₁  :    SkewIndex¹ l n → SkewIndex (l 𝟙 n 𝟘s)
  T₂   :    TreeIndex n      → SkewIndex (l 𝟚 n 𝟘s)
  T₃   :    TreeIndex n      → SkewIndex (l 𝟚 n 𝟘s)
  Ts₂  :    SkewIndex¹ l n → SkewIndex (l 𝟚 n 𝟘s)
```

This is quite a lengthy index type, however, it clearly highlights Okasaki's description of a one-dimensional sequence type: a sequence of trees where the possible sole 2 numeral might introduce two trees instead of 1, hence the duplicity in *SkewIndex* and *SkewIndex*^{1}.

The one-dimensional variant we get is the following.

```
data Tree  : Peano → Set → Set where
  scalar   :    elem                                    → Tree zero       elem
  Node     :    Tree h elem → elem → Tree h elem → Tree (succ h) elem
data SRAL¹  : Skew¹ → Peano → Set → Set where
  scalar   :    elem
    →           SRAL¹ 0 b          n elem
  Node-𝟙   :    Tree (succ (n + m)) elem → SRAL¹ l (succ (n + m)) elem
    →           SRAL¹ (l 𝟙 m 𝟘s) n elem
data SRAL  : Skew → Set → Set where
  scalar   :    elem                                              → SRAL 0 b𝟘         elem
```

```
Node-𝟙 : Tree n elem →             SRAL¹ l n elem → SRAL (l 𝟙 n 𝟘s) elem
Node-𝟚 : Tree n elem → Tree n elem → SRAL¹ l n elem → SRAL (l 𝟚 n 𝟘s) elem
```

The two-dimensional square representation is the following.

```
data QT : Peano → Set → Set where
  scalar : e → QT zero e
  block : e
    → Tree m e → Tree m e → Tree m e → Tree m e
    → QT m e → QT m e → QT m e → QT m e
      → QT (succ m) e
data SM¹ : Skew¹ → Peano → Set → Set where
  scalar : e → SM¹ 0 b m e
  block : QT (succ (m + x)) e
          → SRAL¹ l (succ (m + x)) (Tree (succ (m + x)) e)
          → SRAL¹ l (succ (m + x)) (Tree (succ (m + x)) e)
          → SM¹ l (succ (m + x)) e → SM¹ (l 𝟙 x 𝟘s) m e
data SM : Skew → Set → Set where
  scalar  : e → SM 0 b𝟘 e
  block𝟙  : QT m e
        → SRAL¹ l m (Tree m e) → SRAL¹ l m (Tree m e)
        → SM¹ l m e → SM (l 𝟙 m 𝟘s) e
  block𝟚  : QT m e → QT m e → QT m e → QT m e
        → SRAL¹ l m (Tree m e) → SRAL¹ l m (Tree m e)
        → SRAL¹ l m (Tree m e) → SRAL¹ l m (Tree m e)
        → SM¹ l m e → SM (l 𝟚 m 𝟘s) e
```

For some representations, we require the non-matching dimensions even if the end goal is to model square matrices. This is due to the fact that matrix multiplication sometimes recursively requires multiplying two matrices of non-matching dimensions. This is not a problem that occurs with the skew binary representation.

4 Minimizing Proof Effort

We have demonstrated a few representations of block matrices. Furthermore, as we have already pointed out, doing proofs with block matrices are usually very tedious in terms of the technical detail related to the structure of the matrix, rather than the core logic of the proof. For example, even proving something as profound as associativity of matrix multiplication would require dozens, if not a few hundred lines in Agda with some of the implementations mentioned above.

4.1 Isolating Proofs

We have pointed out the perfectly balanced block matrices are free of the structural nuances. Intuitively, if we are using a semiring algebra or any of its extensions. We can interpret the matrix as a graph, or perhaps an automaton. Most properties that we would want to prove in this setting would still hold regardless of the fact that we could add a vertex without any edges in graph terms, or a dead node in automata terms.

What we are really trying to demonstrate here, in part, is that padding preserves matrix addition and multiplication.

```
pad m₁ + pad m₂ ≡ pad (m₁ + m₂)
pad m₁ ∗ pad m₂ ≡ pad (m₁ ∗ m₂)
```

To achieve embedding of matrices, we require a few lemmas

```
B→A            : B → A
A→B            : A → B
iso-fro        : A→B (B→A a) ≈ᴮ a
≈-compatible   : a ≈ b   → (A→B a) ≈ᴮ (A→B b)
≈ᴮ-compatible  : a ≈ᴮ b → (B→A a) ≈   (B→A b)
embed-I        : B→A Iᴮ ≈ I
embed-+        : B→A (a +ᴮ b) ≈ (B→A a + B→A b)
embed-•        : B→A (a •ᴮ b) ≈ (B→A a • B→A b)
```

A and B can be any type in general as long as they satisfy the requirements, however, for the task at hand, A is the type of square matrices of size $1 + n$, and B is the type of matrices of size n. $\approx^B$ and $\approx$ are equivalence relations over both types. Similarly, for the task at hand, they would be the same equivalence relation on matrices.

We can then prove that the reverse morphism $B{\to}A$ is an algebra homomorphism over the following expression signature.

```
data KExpr (Carrier : Set) : Set where
  _ᵃ  : (a : Carrier) → KExpr Carrier
  Iˢ  : KExpr Carrier
  _+ˢ_ : KExpr Carrier → KExpr Carrier → KExpr Carrier
  _•ˢ_ : KExpr Carrier → KExpr Carrier → KExpr Carrier
```

```
B→A-homo : ∀ {x} → B→A (algB x) ≈ algA (fmap B→A x)
B→A-homo {atom a  } =  -reflexive
B→A-homo {Iˢ       } =  embed-I
B→A-homo {e₁ +ˢ e₂} =
   -transitive embed-+ (+-congruent
     (B→A-homo {x = e₁})
     (B→A-homo {x = e₂}))
B→A-homo {e₁ •ˢ e₂ } =
   -transitive embed-• (•-congruent
     (B→A-homo {x = e₁})
     (B→A-homo {x = e₂}))
```

It can be shown that *Expr* is a functor and *algA* and *algB* are evaluation functions over *Expr*. The following proposition transfers equational proofs over the two types.

```
↑^AB : ∀ {s_1 s_2 : Expr B}
  → algA (fmap B→A s_1) ≈ algA (fmap B→A s_2)
  → algB s_1 ≈^B algB s_2
↑^AB {a} {b} algA[fmap__B → A__a] ≈ algA[fmap__B → A__a] =
     algB a
  ≈ ⟨ ≈^B −symmetric iso-fro                                     ⟩
     A→B (B→A (algB a))
  ≈ ⟨ ≈-compatible (B→A-homo {x = a})                             ⟩
     A→B (algA (fmap B→A a))
  ≈ ⟨ ≈-compatible algA[fmap__B → A__a] ≈ algA[fmap__B → A__a] ⟩
     A→B (algA (fmap B→A b))
  ≈ ⟨ ≈-compatible ( -symmetric (B→A-homo {x = b}))               ⟩
     A→B (B→A (algB b))
  ≈ ⟨ iso-fro                                                     ⟩
     algB b
  ∎
```

Obtaining a proof of matrix associativity for example would be the following expression

```
↑^AB {(a ^a •^s b ^a) •^s c ^a} {a ^a •^s (b ^a •^s c ^a)} •-associative
```

What remains is to instantiate this framework accordingly. The equivalence relation can be instantiated by propositional equality or another custom equivalence relation depending on our requirements. We instantiate A, B and the two morphisms as follows.

```
A    = SM (inc s) elem
B    = SM s       elem
B→A  = pad
A→B  = trim
```

pad and *trim* are the two-dimensional variants of *cons* and *tail* For cons, we pad the matrix by the identity of multiplication precisely at M_{00} and the identity of addition at M_{0i} and M_{i0}.

What remains is to define *pad*, *trim* and prove that they satisfy the requirements mentioned before.

WIP

4.2 Sparse Matrices

TODO: Sparse matrices quotient types.

5 Conclusion

We have demonstrated various two-dimensional numerical representations and argued about them in the context of semiring frameworks. We set out to find an implementation that is ideally easy to understand and work with, as well as being easy to reason about. Furthermore, we minimized proof effort by exploiting the convenient properties of a perfectly balanced block matrix. Additionally, we have shown how to tackle sparsity in an easy and convenient way.

References

1. Hinze, R.: Manufacturing datatypes. J. Funct. Program. **11**, 493–524 (09 2001). https://doi.org/10.1017/S095679680100404X
2. Hinze, R., Swierstra, W.: Calculating datastructures. In: Komendantskaya, E. (ed.) Mathematics of Program Construction. pp. 62–101. Springer International Publishing, Cham (2022)
3. Okasaki, C.: Purely functional data structures. Cambridge University Press, USA (1998)
4. Okasaki, C.: From fast exponentiation to square matrices: an adventure in types. In: Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming. p. 28–35. ICFP '99, Association for Computing Machinery, New York, NY, USA (1999). https://doi.org/10.1145/317636.317781, https://doi.org/10.1145/317636.317781
5. Sandberg Eriksson, A., Jansson, P.: An agda formalisation of the transitive closure of block matrices (extended abstract). In: Proceedings of the 1st International Workshop on Type-Driven Development. p. 60–61. TyDe 2016, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2976022.2976025, https://doi.org/10.1145/2976022.2976025