# Introduction

# Preparation

TODO: Summarise section

TODO: Make more discursive, story like.

TODO: Why am I doing all this?

## Ypnos

*Stencil computations* are an idiom used in parallel programming. They work by defining a *kernel* (or *stencil*) which is applied to each element of an array of data (which I will call a *grid*). The kernel computes a new value for the array location using the old value and the old values of its neighboring cells. In summary the opperation behaves similarly to convolution.

The idiom is particularly useful in the fields of graphical processing where some typical applications include: Gaussian blur, Laplacian of Gaussian, Canny edge detection and many other filter based methods. Stencils can also be used to computer approximations of differential systems of equations. As such they are useful in the simulation of physical systems via fluid, stress or heat dynamics.

Ypnos is an *embedded domain specific language*(EDSL) for stencil computations. Rather than build a language from scratch, it is embedded within the Haskell programming language. This allows Ypnos to share much of the syntax and implementation of its host language. Haskell is a particularly good fit for stencil computations as Ypnos syntax as it is a pure functional language. Its purity allow The idiom is particularly useful in the fields of graphical processing where some typical applications include: Gaussian blur, Laplacian of Gaussian, Canny edge detection and many other filter based methods. Stencils can also be used to computer approximations of differential systems of equations. As such they are useful in the simulation of physical systems via fluid, stress or heat dynamics.

Ypnos is an *embedded domain specific language*(EDSL) for stencil computations. Rather than build a language from scratch, it is embedded within the Haskell programming language. This allows Ypnos to share much of the syntax and implementation of its host language. Haskell is a particularly good fit for stencil computations as Ypnos syntax as it i The idiom is particularly useful in the fields of graphical processing where some typical applications include: Gaussian blur, Laplacian of Gaussian, Canny edge detection and many other filter based methods. Stencils can also be used to computer approximations of differential systems of equations. As such they are useful in the simulation of physical systems via fluid, stress or heat dynamics.

Ypnos is an *embedded domain specific language*(EDSL) for stencil computations. Rather than build a language from scratch, it is embedded within the Haskell programming language. This allows Ypnos to share much of the syntax and implementation of its host language. Haskell is a particularly good fit for stencil computations as Ypnos syntax as it is the programmer to write parallel programs without worrying about the interaction and sharing of state.

### Syntax

The Ypnos language provides a custom syntax for defining stencil functions as well as a collection of primative operations for their manipulation and use.

The syntax for a simple averaging stencil would look as follows:

```
avg2D = [fun| X*Y:|_  a _|
                  |b @c d|
                  |_  e _| -> (a + b + c + d + e )/5|]
```

Ypnos uses the *quasiquoting mechanism* in Haskell to provide its syntax. It essentially allows the programmer to provide a parser and enter the custom syntax in brackets `[parser| ... my custom syntaxt ...|]`. In the case of the Ypnos stencil functions the parser is called `fun`.

The next thing to be noted about the stencil is the syntax `X*Y:`. `X` and `Y` are both dimension variables (as is `Z`). They can be combined using the `*` operator. The syntax defines the dimensionality of the stencil and helps us parse the arguments.

The arguments are enclosed within pipe characters (`|`). Their arrangement in code is typically indented to reflect their grid shape. Arguments can either be named or "don't care" possitions denoted with either a name or an underscore respectively. The arguments annotated with `@` is the cursor, the central cell whose position is used for the result of the stencil.

The final section of the syntax comes after `->` and is the computation. This can be most Haskell syntax though recursion and function definition is not possible.

### Primitives

As well as the syntax for stencil functions, Ypnos provides a library of primative operations. The primatives allow the programmer to combine the stencils with grids to produce the computations they want. The main primative in Ypnos is the *run* primative which applies the stencil computation to a grid.

```
run :: (Grid D a -> b) -> Grid D a -> Grid D b
```

The application is done by moving the stencil cursor over each location in the grid. The arguments of the stencil are taken from positions in the grid relative to the cursor. The value is then computed using the specified computation and put into the same cursor location in a *new* grid.

TODO: Say something about the argument being Grid D a. n In some locations near the edge of the grid their may not be enough neighbors to satisfy a stencil. In this case Ypnos provides a special syntax for dealing with these *boundaries*. I have considered the implementation of boundaries beyond the scope of this project however I will include a brief description of their behaviour.

For each boundary of the grid outside of which the run primative may need to access we define a behaviour. We may compute a value for these locations using: the current location index, values from the grid (accessed via a specially bound variable). A common boundary is the *mirror* boundary which works by providing the closest value inside the grid when an outside access is made. This is the boundary that I have tacitly assumed in my implementation.

Another vital primative of the Ypnos language is the *reduce* primative whose purpose is to summarise the contents of a grid in one value. It may be used to compute functions such as the mean, sum or minimum/maximum.

```
reduce :: (a -> a -> a) -> Grid D a -> a
```

The primative uses an associative operator (of type `a -> a -> a`) to combine all the elements of the grid to one value. A more general version of this operator also exists which support an intermediary type.

```
reduceR :: Reducer a b -> Grid D a -> a
mkReducer :: exists b. (a -> b -> b)
                    -> (b -> b -> b)
                    -> b
                    -> (b -> c)
                    -> Reducer a
```

The *Reducer* data type takes parameters:

- a function reducing an element and a partial value to a partial value,
- a function reducing two partial values,
- a default partial value
- and a conversion from a partial value to the final value.

This the Reducer is passed into the *reduceR* primitive taking the place of our associative operator in the reduce primitive. Clearly, reduce can be implemented in terms of reduceR and so the later is the more general.

## Accelerate

Modern GPUs provide vast amounts of SIMD parallelism via general purpose interfaces (GPGPU). For most users these are hard to use as the interfaces are very low level and require the user to put a lot of effort into writing correct parallel programs.

Matrices are a mathematical model which embodies SIMD parallelism. They are operators over large amounts of data. It is possible to express many parallel calculations and operations as matrix equations.

*Accelerate* is an EDSL for the Haskell language which implements parallel array computations on the GPU. The primary target GPUs are those which support NVIDIA's CUDA extension for GPGPU programming. Accelerate uses algorithm skeletons for online CUDA code generation.

Accelerate uses the Haskell type system to differentiate between arrays on the CPU and GPU. It does this by way of a type encapsulating GPU operations. There is a further *stratification* of this type into scalar and array values with scalar computations being composed into array computations.

Accelerate contains built in support for stencil computations and so is an excellent intermediary target for this project. While the stencil semantics of Accelerate and Ypnos differ in some respects, the former is powerful enough to represent the later. As such, this project will concern itself not with the generation of CUDA code directly but through the Accelerate language.

### GPU computation

Haskell execution happens on the CPU and in main memory whereas GPU execution happens in parallel in a separate memory. In order for a process on the CPU to execute a CUDA program it must first send the program and the data to the GPU. When the result is ready it must be copied back into the main memory of the process concerned. I will call these two processes *copy-on* and *copy-off* respectively.

Accelerate chose to represent this difference in the type system. The `Acc` type denotes an operation on the GPU. For the purposes of Accelerate, the only operations allowed on the GPU are those over arrays. As such, `Array sh e` denotes an array of shape `sh` and element type `e` and `Acc (Array sh e)` denotes the same but in GPU memory and may also encapsulate an operation.

Arrays are signalled for use on the GPU via the `use` primitive. They are copied-on, executed and copied-off via the `run`. This primitive is responsible for the run-time compilation and actual data transfer. All other operations build an AST to be compiled by the run primitive. Together use and run form the constructors and destructors of the `Acc` data type.

```
use :: Array sh e -> Acc (Array sh e)
run :: Acc (Array sh e) -> Array sh e
```

### A stratified language

While the main type of operation in Accelerate is over arrays. However, we often want to compose arrays out of multiple scalar values or functions over scalars. A classic example of this is the map function to transform an entire array by a function over the individual values[1]. For this reason, in addition to the Acc type, Accelerate also provides the Exp type where the former represents collective operations and the later represents scalar computations.

The map function would then looks like this:

```
map :: (Exp a -> Exp b) -> Acc (Array sh a) -> Acc (Array sh b)
```

Scalar operations do not support any type of iteration or recursion in order to prevent the asymmetric operation run time. However, most other Haskell code is allowed. This is achieved by the Haskell class mechanism: Accelerate provides instances of Exp a for most common classes.

For example, in the support of addition, subtraction and other numerical operations, Accelerate provides an instance of the type class Num. This means that operations can be typed as follows:

```
(+) :: Num a => Exp a -> Exp a -> Exp a
1 + 2 + 3 :: Exp Integer
```

### Stencil support

I have already mentioned that function over scalars can be applied over a whole grid, the map function being an example of this. Accelerate provides support for stencil computations via the stencil function.

```
stencil :: Stencil sh a sten =>
           (sten -> Exp b) ->
           Boundary a ->
           Acc (Array sh a) ->
           Acc (Array sh b)

instance Stencil DIM2 a ((Exp a, Exp a, Exp a)
                        ,(Exp a, Exp a, Exp a)
                        ,(Exp a, Exp a, Exp a))
```

The first parameter is a function which represent the stencil. We see that sten, the stencil pattern, takes the form of a tuple grid of Exp a element type. This allows Accelerate to use Haskell's function syntax to define stencils.

The second parameter is the type of boundary. In Accelerate, the types of boundary allowed are fixed as opposed to Ypnos boundaries which can be fully specified. One of the types allowed is Mirror which deal with an out of bounds access by picking the nearest coordinate from within the array.

With these two parameters we have defined an operation performs the stencil convolution.

[1] In fact, the map function is conceptually similar to stencil application. The difference being that stencils also take into account the neighbourhood of a cell to compute the next value.

### Learning the libraries

In order to get familiar with the syntax of these two libraries I decided to implement some sample functions in both, the main one being the average function we have already seen. With this I was able to familiarise myself with the embedded languages and prepare for the task of implementation.

Furthermore, this allowed me to become familiar with the other tools I would be using in this project, namely:

- Haskell, the programming language. Having not used this before I had to get familiar with the syntax as well as some of the more advanced type system features and extensions such as: *type classes*, *type families* and *data families*.
- Cabal, a build system for Haskell with automatic dependencies resolution and fetching. The toy functions allowed me to test setting up a build system which I would later use for the rest of the project.
- Git, the version control system. I was already quite familiar with this system before this project but it allowed me to make use of some of Git's more advanced features such as *stashing*, *sub-projects* and *branching*.
- CUDA, the drivers and library. As I do not own a machine with a CUDA enabled graphics card, I was using a remote machine located in the Computer Laboratory. The sample functions allowed me to set up the machine with the drivers and configuration required in order to run the Accelerate library.

TODO: Requirements analysis plus other software engineering techniques

### Test driven development

The correctness of my implementation was a central goal from the beginning of the project. In order to achieve this I took a test driven approach to development. This meant that while writing the implementation I was simultaneously writing unit tests for that code. The approach allowed me to quickly and effectively find bugs which had not already been found by the Haskell type system.

*QuickCheck* is Haskell's defacto standard unit testing library. In most unit testing libraries for other platforms, the programmer has to provide sets of test data for the library to check against the program. The code for generating this data is left to the programmer. QuickCheck takes a different approach. Instead of specifying testing functions which include the test generation, we specify properties which take the data to be tested as an argument. We then leave the generation of this data up to the library.

QuickCheck is able to generate random testing data for most built in Haskell data types. For user defined types, the programmer must provide an instance of the class `Arbitrary` which allows QuickCheck to generate random samples for testing.

## Implementation

### Compilation of the stencils

Compilation of stencils was a central task in this project. The abstract Ypnos syntax allows much flexibility in the underlying implementation. Ypnos achieves this via Haskell's Quasi Quoting mechanism for compiling custom syntax to Haskell AST. Accelerate's implementation has overridden much of the Haskell operators required for this translation stage so the bulk of the effort went into producing the functions that contained the computation. These functions take the following form:

```
avg :: Exp a => Stencil3x3 a -> Exp a
avg (( _, a, _ )
    ,( b, c, d )
    ,( _, e, _ )) = (a + b + c + d + e) / 5

type Stencil3x3 = ((Exp a, Exp a, Exp a)
                  ,(Exp a, Exp a, Exp a)
                  ,(Exp a, Exp a, Exp a))
```

The arguments are formed as tuples of tuples. The rest of the stencil appears to be normal Haskell code. However, the return type, `Exp a`, insures that all the operations actually use Accelerates overridden methods to build an AST. The AST is then translated at run-time into CUDA code.

Haskell's quasiquoting mechanism is a compiler option which allows the library author to provide custom syntax for domain specific languages. As such, it is a perfect fit for Ypnos, which would like to hide the underling implementation from the user. It works by providing a parser object (refered to as a quasiquoter) and a syntax for applying it within normal Haskell code. The essential function of a quasiquoter is to provide an abbreviation for entering the AST manually.

Take for example the situation in which we want to write an embedded language to act as a calculator. We have the following AST for our simple calculator:

```
data Expr  =  IntExpr Integer
           |  BinopExpr (Integer -> Integer -> Integer) Expr Expr

e1 = BinopExpr (+) (IntExpr 1) (IntExpr 3)
e2 = [expr| 1 + 3 |]
```

We see that the quasiquoter `expr` allows us to appreviate the expression `e1` to the more obvious form of `e2`.

Clearly, if we were to swap out the quasiquoter this would be an effective way of producing multiple programs from the same syntax. This is what Ypnos achieves in its stencil syntax. The aim is to be able to change the quasiquoter and fully change the underlying implementation without any other modifications.

We could sensibly do the translation from Ypnos to Accelerate stencils in one of two ways: we (a) use Haskell's type system to mask the difference between the two types of stencil computation or (b) we use run-time conversion to mask the difference between the implementations and maintain the semblance of the types. I explored each of these approaches in the course of the project and I would like to now describe the benefits and drawbacks of both.

### A type system approach

As we saw in the previous section: the types of the Ypnos CPU stencil and the Accelerate library's stencil differ wildly. Let's a closer look at the precise differences between them in the types of our stencil `avg`[2]:

```
avgCPU :: (IArray UArray a, Fractional a) =>
          Grid (Dim X :* Dim Y) b dyn a -> a
avgGPU :: Floating (Exp a) =>
          Stencil3x3 a -> Exp a
```

In the GPU case we see that the type (once expanded) is tuples of tuples of `Exp a`. This allows Accelerate to make use of the built in Haskell syntax for functions. This is of little use to us as we have our own syntax already. On the other hand, in the CPU case we see that arguments take the form of a grid, which is exactly the same type as the grids it operates on.

This is no accident as Ypnos grid type is a comonad, the theoretic dual of the of the monad. This restrains the type of the run operation to be of the form:

[2] For the sake of simplicity I have excluded the type constraints relating to boundaries as these are very long and complicated.

```
cobind :: (D a -> b) -> D a -> D b
```

Where we let `D` be a grid of a certain dimension and `a` and `b` be the types of that grid.

Unfortunately, by translating directly to the Accelerate stencil type we lose the comonadic nature of the type. This is a shame because this type is both informative to the programmer yet flexible enough that by changing the instance of `D` we change the implementation.

The advantage of this method (as we will see more in detail when we discuss the alternative) is that all the translation effort is done at compile time allowing the running of the stencil to be more efficient.

Another way in which Accelerate and Ypnos stencils differ is that the former assumes that the cursor is centred whereas the later allows the user to program this. This can be translated by padding the stencil handed off to Accelerate such that the cursor has been centred.

This is perhaps best illustrated by example. Say that we have the following one dimensional stencil with the cursor at an off-centre location (denoted by `c`).

```
          b
  *---------------*
  | _ | c | _ | _ |
  *---*   *-------*
    a       b-a-1
```

We define the variables $a$ to be the position of the cursor and $b$ to be the length of the stencil. Now we must find how much we must pad at the beginning and end of the stencil to centre the cursor. This is given by the following two equations:

$$pad_{beginning} = max\{a, b\} - a$$

$$pad_{end} = max\{a, b\} - b + a + 1$$

This means that after centring we get the following:

```
         coffset roffset
          *---*   *---*
  | _ | _ | c | _ | _ |
```

### Implementing centring

In order to implement the centring I had to consider both the one and two dimensional cases. It would be quite easy to deal with this deal with this in two separate cases except that it would be nice to extend the approach to higher dimension eventually. I three principle approaches to doing this: using lists as an intermediary; using arrays as intermediaries; and operating on the grid patterns directly via type classes. Before addressing the approaches I will mention the types we were converting.

```
data GridPattern =
    GridPattern1D DimTag [VarP] |
    GridPattern2D DimTag DimTag [[VarP]]
```

`GridPattern` is the type in the Ypnos AST corresponding to the parsed pattern of arguments. We see that it takes both a 1D and 2D form where the variables (`VarP`) are a list and a list of lists respectively. We may also note that the dimensionality is expressed directly in the constructor and as such is not present in the type.

The pattern of arguments in Accelerate is a expressed as tuple in the 1D case and a tuple of tuples in the 2D case. This representation contains no information about which variables are cursors as we discussed in the previous section.

**Intermediate approaches** The first approach taken involved converting first from grid patterns into lists, balancing these lists then converting them into the centred tuples needed for the Accelerate functional representation. In order to do this I would have to define functions for measuring the location of the cursor, and padding the lists before and after. This approach proved difficult as lists did not explicitly incorporate their dimensionality in their type. This made it hard to treat the 1D and 2D cases differently.

The second approach attempted to use existing array code in order avoid writing such functions. The hope was that by converting to arrays, rather than lists, functions for appending and prepending rows and columns would already exist. However, this was not the case and I would have had to write these myself. This made the point of the intermediary stage of arrays altogether pointless.

**Direct approach** The third and final approach was to operate directly on the lists extracted from the `GridPattern` types. As I already mentioned: the problem with working with lists is that the dimensionality is lost in the type. To retain this information in the type system I designed a class `GridIx` to perform the basic operations - `addBefore`, `addAfter`, `find` and `size` - in a dimension sensitive way while still being polymorphic.

```
class (Ix i, Num i, ElMax i) => GridIx i where
    data GridPatt i :: * -> *
    addBefore :: i -> a -> GridPatt i a -> GridPatt i a
    addAfter :: i -> a -> GridPatt i a -> GridPatt i a
    find :: (a -> Bool) -> GridPatt i a -> i
    size :: GridPatt i a -> i
```

The associated data type `GridPatt` would take the type of the particular dimensionality of list that is appropriate for a given instance. In the case of the index type `Int` we would get `GridPatt Int a = [a]` and in the case of `(Int, Int)` we get `[[a]]`. This approach allows the algorithms for centring to be described at a general level without being concerned about the number of dimensions actually involved.

### A run-time approach

The second approach to the translation of stencils was to keep the types the same (or similar, as we will see) to Ypnos' original implementation. This is alluring as it allows us to both expose more information to the user through the programs type and maintain the theoretic underpinnings of Ypnos: the comonadic structure. In order to achieve this we need to do some run-time type translations. These have an overhead for (TODO: do they?) the performance of the stencil application but I will discuss at the end of this section what approaches can be taken to mitigate this overhead.

As already seen, we would like the `run` primitive to take the form:

```
run :: Comonad g => (g a -> b) -> g a -> g b
```

We have also seen that Accelerate does not accept stencils of this form. To solve this we previously broke the the comonadicity of the operation but we could attempt preserve it by introducing an *arrow* data constructor to absorb the differences in type between Accelerates notion of a function and Ypnos'. This changes the run function to:

run :: Comonad g => (g a `arr` b) -> g a -> g b

The data constructor is paramatrized on both `g a` and `b`. To build up an instance of `arr` we must pass in the stencil function to a special constructor. The constructor chosen decides the implementation used.

While previously we had to use different versions of the quasiquoter to produce different stencils at compile-time, we now use the same quasiquoter but convert the function at run-time. We achieve this by taking advantage of Haskell's polymorphism which allows a function over type `a` to generalise to a function of type `Exp a`. This generalisation in concert with the arrow data constructor allows our stencil functions to have the type:

stencil :: Comonad g => g (Exp a) -> Exp b stencil' :: Comonad g => g a `arr` b

Because of the arrow type, `stencil` and `stencil'` can actually have the same type.

However, we are only half-way there: the type of stencil accepted by Accelerate is still not of the form `g (Exp a) -> Exp b`. I achieve this stencil by a conversion function which builds an Accelerate stencil (call it *stencil A*) at run-time using the stencil encapsulated in the arrow data type (call it *stencil B*). Stencil A's arguments are used to build up a grid of type `g (Exp a)` then stencil B is used on this grid to produce the result of type `Exp b`.

While this run-time conversion creates an overhead it also, as we have seen, simplifies the types significantly. However TODO: mention deforestation.

## Implementation of the primitives

The primitives are the second central component of the translation. Without them we could not run our translated stencils on the GPU. Like the stencil translation the implementation of the primitives took two primary approaches. The first was to re-implement the primitives in a separate module. In this case the user would import whichever implementation they required. This approach had some fatal draw backs in that it required the user to change too much of their code between implementations. This led to the second approach of extracting the functionality of the primitive into a type class. This approach required the use of some complicated type features in order to make the types unify. However, the final result is much more usable.

### Non unifying approach

The initial of run was linked to the compile time implementation of the stencil function. At the highest level this meant that the function `run` had the following type:

```
run :: (Stencil sh x sten) =>
       (sten -> Exp y) -> Grid d Nil Static x -> Grid d Nil Static y
```

However, we see that the type of `sh` (required by Accelerate) and `d` (required by Ypnos) do not unify directly requiring another constraint to reconcile the two. Further, constraints need to then be added for the types of `x` and `y` to satisfy Accelerates `stencil` function. In the end the end this type becomes unwieldy meaning that it is not straight forward for the user to replace it in their code.

Similar problems would have plagued the implementation of the `reduce` primitive. However, having seen the first implementation of the `run` primitive I decided that a different approach was need so this incarnation of the `reduce` primitive never saw the light of day.

TODO: Reduce implementation and the trick to it.

TODO: Generalisation of functions

TODO: Type generalization approaches

TODO: Associated types vs type synonyms

TODO: run example

### Type class parameter

### Associated type families

### Associated data families

### Final implementation

TODO: run implementation

TODO: reduce implementation

## How would it be used

TODO: Example of usage and replacement

### Constructors and destructors

# Evaluation

## Approach

TODO: Measure of difficulty

TODO: measuring copy on/off

## Results

TODO: Speed up run

TODO: Reduce

## Deducing a model

TODO: Model of on/off

## Useability to the programmer

TODO: User having to write types

# Conclusion