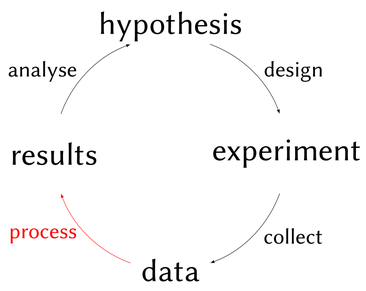
Universal representability for graphical processing units

Nick Hu
University of Oxford, Department of Computer Science
nick.hu@cs.ox.ac.uk

The essence of quantitative data

Quantitative data is numeric. The scientific method, in a nutshell, can be summarised as:

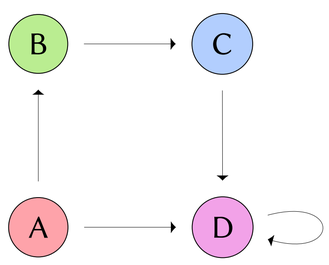


Processing data might mean regression analysis, noise filtering, or even reshaping. In the modern world, the processing is almost always performed by computers, which are instructed by programs.

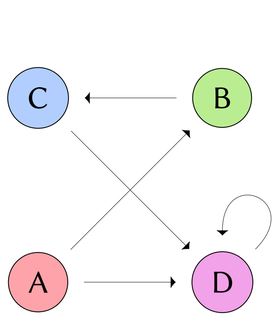
Thus, it is necessary to have a solid handle on what it might mean to represent and manipulate data on the machine.

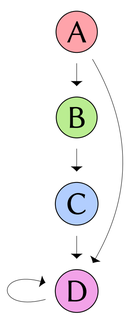
Isomorphism

Consider the following graph:



We have a notion of its *data*, and in a way the following graphs are the 'same' as they have the same data:





This notion of sameness is called *isomorphism*.

For the computer to understand a graph, it needs a *concrete representation* (in the sense that the mapping into computer memory is well-defined).

Adjacency list

For each node, a list of the nodes to which it is connected is tracked.

$$\begin{bmatrix} A \to [B, D], \\ B \to [C], \\ C \to [D], \\ D \to [D] \end{bmatrix}$$

Good for minimising memory usage.

Adjacency matrix

For every pair of nodes, there is either a connection from the first to the second (1) or there is not (0).

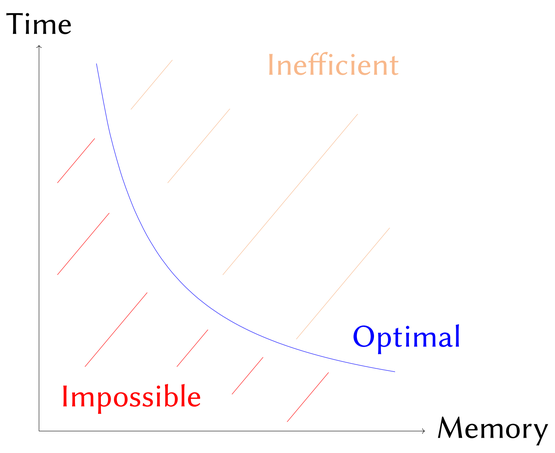
	A	B	C	D
A	0	1	0	1
B	0	0	1	0
C	0	0	0	1
D	0	0	0	1

Good for fast computations.

Concrete concerns

The two methods store identical data, but in different ways; neither is definitively better than the other, because different situations have differing *computational concerns*.

This is an example of the classic *time-space* trade-off principle:

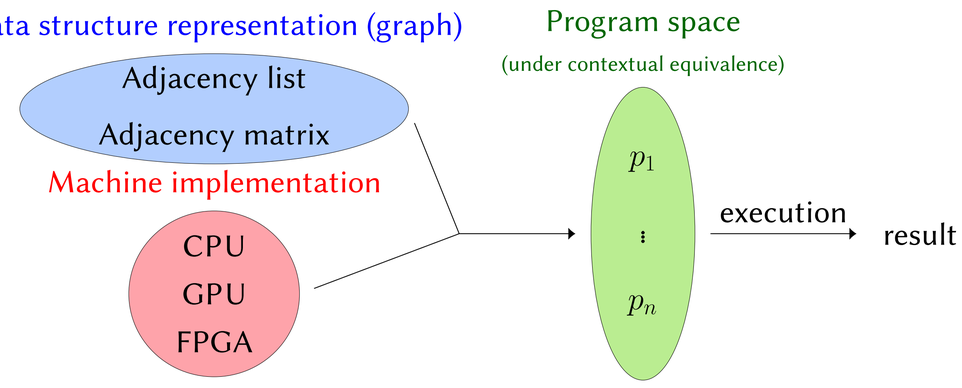


The adjacency list is *sparse* (compact but slow), whereas the adjacency matrix is *dense* (fast but large). The adjacency list is like tracking only the 1s of the adjacency matrix, and implicitly assuming everything else is 0 — both have the same information, and so they are isomorphic.

Quantitative data from the real world is often dominated by notional 0-values, so sparse representations are frequently used.

Abstraction

The principle of abstraction states that *we need not be concerned with concrete (implementation) details* when we program computers, and instead consider how each component interact together *abstractly*.



This separation of concerns facilitates compositionality, allowing for simple programs (which are much easier to verify for correctness) to be designed and built in isolation in order to compose them to create larger, more complex programs.

Representable functors

A *representable functor* can be considered a data structure which exists as a concrete tabulation in memory, and isomorphically as a function of index yielding the table entry at that index. The isomorphism part means that it is unnecessary to define both; from one representation, the computer can derive the other. In the programming language *Haskell*:

```
class Functor f => Naperian f where
  type Log f
  tabulate :: (Log f -> a) -> f a
  lookup :: f a -> (Log f -> a)
  positions :: f (Log f)
```

With this interface, the primitive array operations can be distilled:

- replication, to lift dimensions for rank polymorphism via alignment;
- zip, to combine arrays;
- traversal, to sequence array data;
- transposition, to select an index along a multi-dimensional array.

The atoms of these programs are *hypercuboids* (n-dimensional arrays), which are combined with these operations to process data.

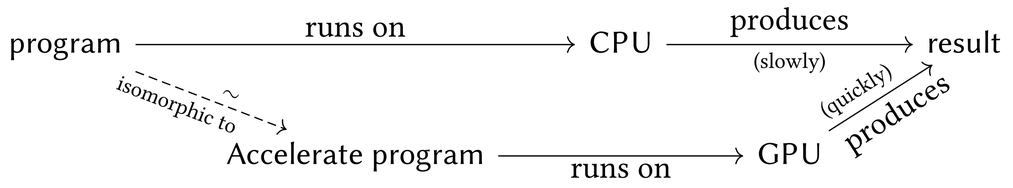
Programming in this way allows the use of *equational reasoning* to prove properties about programs (for example, identical behaviour on both the concrete tabulation and the function from indices) in the same way that one might prove theorems about algebra. Algebraic equalities lead to computational shortcuts, optimising algorithm efficiency.

Accelerate

Accelerate is a Haskell library which generates code to run on a graphical processing unit (GPU). Where a traditional computer processor might have up to a dozen cores each with a large amount of cache memory, a modern GPU might have thousands of compute cores which are individually simpler and slower, with less memory. This allows certain computational workloads, which are paralisable and often naturally numeric, to be sped up by several orders of magnitude.

However, programming for the GPU is traditionally very complicated, because it is a specialised hardware component as opposed to the general purpose CPU.

The goal is to be able to write the same programs for the *abstract machine*, which can automatically utilise the GPU if it is available.



Hypercuboidal Accelerate arrays

Representable functors provide semantics for writing array programs abstractly in terms of *hypercuboids*, which greatly increases programmer efficiency. My work is the interpretation of the hypercuboids into the Accelerate world, *providing witness to the isomorphism between Accelerate data structures and hypercuboids.* Such work facilitates the automated translation of hypercuboid programs into Accelerate programs which preserve correctness.

I was successful in laying the groundwork, establishing an effective mechanised translation between the base Accelerate `Array` type and the Naperian `Hyper` type:

$$\texttt{Hyper} \underset{\texttt{hyperise} \circ \lambda x.(\texttt{arrayShape}\ x)(\texttt{toList}\ x)}{\overset{\texttt{toArray}}{\underset{\sim}{\longrightarrow}}} \texttt{Array}$$

References

Jeremy Gibbons. 'APLicative Programming with Naperian Functors'. In: *European Symposium on Programming*. Ed. by Hongseok Yang. Vol. 10201. LNCS. Apr. 2017, pp. 568–583. DOI: 10.1007/978-3-662-54434-1_21. URL: http://www.cs.ox.ac.uk/people/jeremy.gibbons/publications/aplicative.pdf.