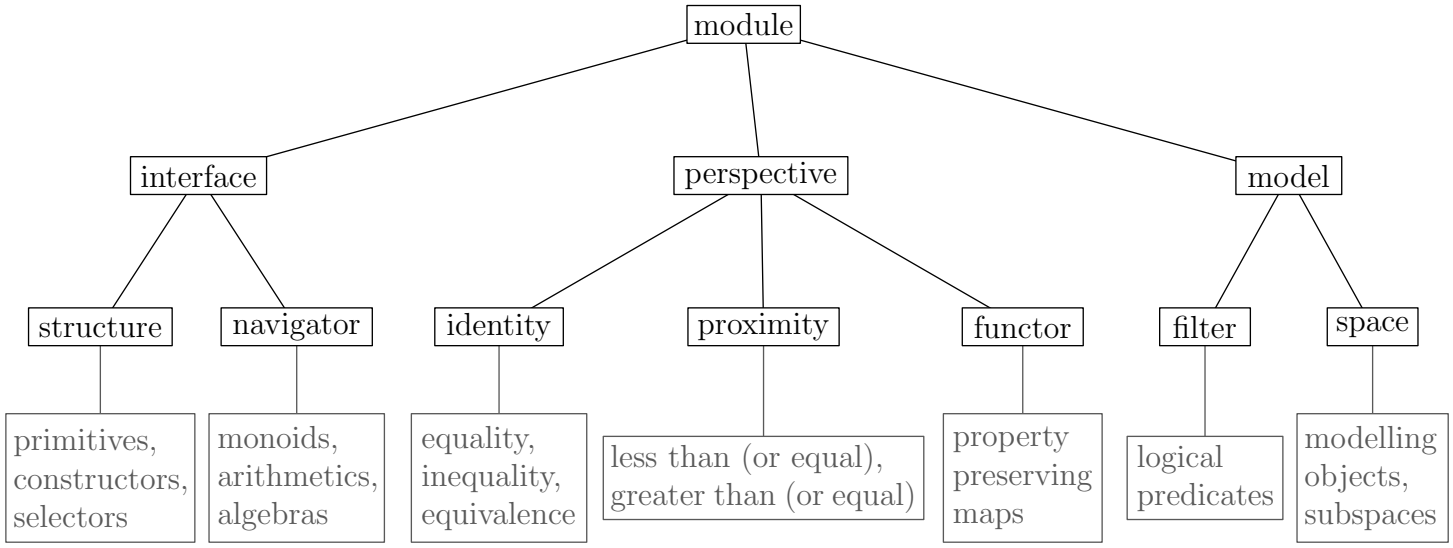
Module Theory

Daniel Nikpayuk

December 1, 2019

Module Theory is intended to offer a *schema* for organizing the tools used to build modelling spaces:



motivation

Motivation for this theory comes from the desire to standardize the ways in which we design and organize computer programming source code. This is not to say module theory is intended to be *the only way* to design and organize such code, rather it is meant to be *a* standard way.

As it stands, the module terminology already exists as a paradigm in many programming languages, the difference here is this theory takes its inspiration from mathematical modelling.

inspiration

What makes a space good as a model?

For example $\mathbb{R}^3$ the mathematical space that represents our 3 dimensional world is a privileged model in math literature, why then is it so successful? Why can we model spheres and toruses and cubes and dodecahedrons and Mobius strips and so many other spaces with this one single modelling space? The full answers to these questions are philosophical in nature and so are up for debate, but otherwise I feel it is safe to say they have everything to do with mathematical *methodologies*, which are the principles that guide a mathematician's value-systems.

I will claim here that the "subsetting" paradigm:

$$\{\, x \in A \mid P(x) \,\}$$

from Set Theory (a foundational language of mathematics) offers such a methodology, and is the initial inspiration for many privileged mathematical models. The idea is if a space, or structure, or (simply) set, is accessible such that one also has easy access to its substructures, subspaces, or subsets, this underlying *entropy*, or *expressivity*, is a sufficient condition when determining a successful model.

implementation

So what makes a space entropic/expressive when it comes to subspacing?

The short answer: Expressive tools that allow us to group and regroup the elements of a space with ease. After researching common patterns among the constructions of "successful" modelling spaces such as $\mathbb{R}$ or N, the module theory I have arrived at organizes its tools as follows:

- **interface**: The tools to interact with a given space.
  - **structure**: The tools to construct the space, and access its internal states.
  - **navigator**: The tools to navigate to individual elements of a space, usually at a lower cost than the construction tools otherwise allow for.
- **perspective**: The tools to compare elements of a given space.
  - **identity**: The tools to compare the identities of elements of a space.
  - **proximity**: The direct tools to compare nearness of elements and orderings of a space.
  - **functor**: The indirect tools to compare elements of a space, by means of some other similarly behaving space usually exterior to the module.
- **model**: The tools to create specific models from the given space.
  - **filter**: The tools constructed from **perspective** tools to group and regroup the elements of a space.
  - **space**: Unique descriptions of the implemented subspaces of the module's space.

The *interface, perspective, model* groupings are called **partitions**. Strictly speaking they're unnecessary in any module theory formalisms, but on a personal level I find they clarify the relationships between what are otherwise the *structure, navigator, identity, proximity, functor, filter*, and *space* **divisions**.

In practice if you're using this schema, I recommend you start by asking the following questions:

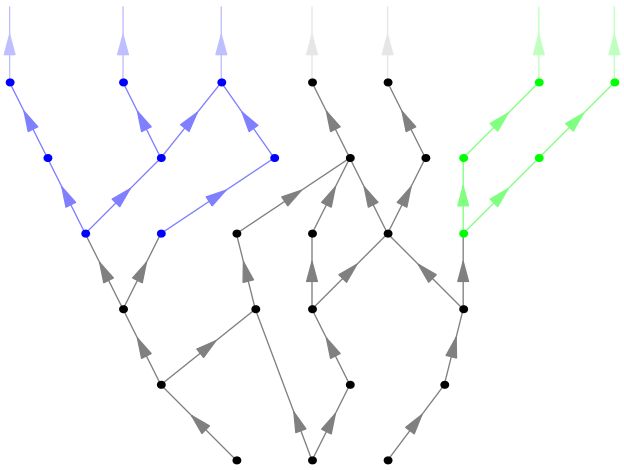
What space are you trying to model?
What tools are you using to do so?

Module theory then offers a standardized way to organize these tools, and to even clarify some of their relationships.

higher order modelling

The design of a module is intentionally *recursive*.

As the final division of a module is the modelling *space* (and its subspaces), one can then reinterpret the meaning of that space to be a space of other already existing modules:



Although simplistically unlabeled, this graphic can be interpreted as a module of modules. Each dot is a local module, while the arrows connecting are the navigational paths allowing us to access specific local modules. Within this global module, we would then classify the tools used to compare these dots, and use these comparisions to group and regroup these local modules.

In practice, this sort of thing could be interpreted as the basis of a code library. In this case, dot module constructors aren't strictly necessary as they already exist, while navigational paths represent module dependencies. After all, you can't access a dot module until you know you have access to the tools from other dot modules that it depends on.

In any case, as this reuse of "module" terminology can become confusing between local and global, the initial modules will instead be called 0-modules, modules of such modules will be called 1-modules, modules of 1-modules called 2-modules, and so on and so on. In this sense, you have higher order modules:

$$\vdots$$

module-3	library
module-2	lens
module-1	branch
module-0	module

Keep in mind the *library, lens, branch, module* terminology here are just examples, used as local aliases in what is otherwise the implementation of specific library of mine. In practice your own library might be made up of more or less n-modules with other alias names of your own as you see fit.

As for using module theory to assist in organizing code: It is most effective when the code you're writing is meant to be a model for many applications. Clear cut examples in real world programming languages are objects as part of the object oriented programming paradigm, or more generally the module paradigm.

conclusion

I will finish by saying this research is open and ongoing, so I will leave you with some thoughts to entice your further interest.

For starters, one of the main values in using this schema is that when the relationships are identified, any module adhering to this design is well suited for modular composition: Structure composes with structure, navigators compose with navigators, identity, proximity, functor perspectives compose respectively with their corresponding perspectives. When building a library, you can then focus on building your narrative design instead rather than how its tools will be organized.

Next, one important application is in modelling proof assistants. In this case, the space you're modelling is the space of "grammatical objects" of some other programming language. You can then talk about individual grammar points, compare when possible, group and regroup them, claim and prove theorems about them, or even use the subspaces as the basis to organize branches, lens, and libraries.

Finally, Not only does module theory offer a natural means of organizing modelling code, it offers other insights as well. For example within any programming language, the grammar tends to provide a few "functions" with side effects. Such functions are the bane of functional programming, considered necessary evils otherwise weakening an immutable scheme, but on the context of the tools used to build models, such functions are actually classified quite naturally as functors.