Modularity and composability in Mathematical Computation

Nicolas M. Thiéry et al.

Tetrapod: Modular Knowledge workshop

FLoC 2018, Oxford, July 13th of 2018

Abstract

Over the last decades, a huge amount of computational software was developed for pure mathematics, in particular to support research and education. As for any complex ecosystem of software components, the ability to compose them has a multiplier effect on the expressive power, flexibility, and range of applications.

The purpose of this session is to share experience on the many barriers to composability in computational mathematics, and how they are being tackled. Of particular interest will be the exploitation of knowledge to leverage some of the barriers. feedback from neighbor fields (proofs, data, knowledge) will be most welcome.

A collaborative document

This document, hosted on: https://hackmd.io/IZESWGzKSOCKJiC_tIP3pw, benefited from edits and comments from the participants of the workshop, including:

- Georges Gonthier (Inria Saclay)
- Florian Rabe (FAU Erlangen-Nürnberg, LRI, Univ. Paris Sud)
- Dennis Müller (FAU Erlangen-Nürnberg)
- Jeremy Gibbons (University of Oxford), local observer and gate-crasher
- Jacques Carette (McMaster University) System designer and developper

Composability for Math software?

For the sake of simplicity, we will mostly focus on two aspects of composability: the ability to transfer data and run procedures across "systems" (two different software, two instances of the same software on different machines, two libraries within a system, ...).

Data transfers

- **Goal**: Transfer an object O from system `A` to system `B`
- **Requirements**:
  - **A communication channel**: shared memory, disk, pipe, (web)socket, ...
  - **A communication protocol**
  - **Serialization / Deserialization**: conversion to/from a string of bytes
  - **A format specification**; e.g. XML (syntax) + OpenMath content dictionary (semantic)
  - **Format conversion**
    - **syntax**: e.g.: JSON <-> XML
    - **adaptation**: `DihedralGroup(4)` <-> `DihedralGroup(8)`
    - **change of representation (codec)**: recursive <-> sparse polynomial

  Idealy: applying a *theory morphism* to guarantee semantic preservation

Procedure calls

- **Goal**: From `B` request the computation `f(a,b,c)` in `A`
- **Requirements**:
  - **Data transfers**: send `a,b,c` to `A`, receive back the result
  - **Procedure call**: bindings, remote procedure calls, ...
  - **An API specification**: syntax and semantic of `f` in `A`?
  - **Adapting the API in B to the API in A**: `Size(A)` <-> `A.cardinality()`
    Idealy: applying a *theory morphism* to guarantee semantic preservation

Barriers faced by the computational pure maths community?

Semantic barriers

- **Many kinds of objects**:
  E.g. thousands in Sage, compared to "matrices of floats" in Matlab
- **Many data representations**, with very different algorithmic complexity
  E.g. for polynomials: sparse, dense, recursive, straight line program, evaluation, ...
- Objects at different levels of abstraction
- **Few core concepts**
  addition, multiplication commutativity
- ... but **many interesting ways to combine them**:
  Groups, Fields, graded commutative algebras, ...
- **Many interactions across different areas of mathematics**
  Hey, that's the point of maths!
- A **large variety of use cases**
  speed maniacs, flexibility fans, composability devouts

The good news

We can afford not to be completely formal; **best effort is usually ok**

Social barriers

- **Closed science** and misc **Montaigu-Capulet fights**
  - "don't want/dare to share my code"
  - "won't use code from XXX"
  - "won't fund contributions to a US-born project"

  Come on, we are in 2018!
- **Tight man power resources**
  - Hundreds of thousands use computer algebra
  - hundreds implement it;
  - a handful are paid full time for it (+ a hundred in commercial systems)
- **Mastering math and computer science**
- **High level of math specialization required when writing code**
  - thousands of people use Gröbner bases
  - a handful know how to implement them right
- Pieces of software that took decades to develop
  - We can't afford to reimplement them
  - We can't afford to not reimplement them:
    learn from one's mistakes, benefit from technology advances
- A fragmented community

Technical barriers

- **Systems written in different languages / idioms**
  C, C++, Python 2/3, Java, Javascript, Julia, GAP, Singular, Macaulay, Maple, Mathematica, MuPAD, Magma, Axiom/Aldor, Haskell, ML, Agda, ...
- **Systems written using different frameworks / API**
- **Packaging barriers**
- Tendency for **large old systems**: 10-30 years, 10^6 lines of code
  - lots of **technical debt**; **slow evolution**
  - **Lack of modularity**, **large dependencies** E.g. having to install SageMath to use just a few of its lines

Tendencies

- Specialized projects – often with a dedication to speed – make efforts to be developed as low-level libraries (e.g. C, C++).
- Large systems make efforts to be usable as **libraries** and not just **standalone systems**. Typically enabling low-level calls at the C level for fine-grained interactions.
- Modern languages make it easier to call libraries written in other languages (e.g. C, C++).
- Everybody makes efforts for better software development practices:
- **development workflows**, **publication**, **packaging**, **distribution**, ...
- There even is some funding for it (OpenDreamKit, OSCAR)
- **Examples**:
  - Integration of GAP, Singular, ... in SageMath
  - Symmetric integration Julia-GAP (OSCAR project)
  - Integration of linbox, libsemigroups, gmp, MPIR in GAP, SageMath, Singular
- **Caveat**: takes care of **binding** not yet of **adapting**

Knowledge for in-system modularity

- Many systems (e.g. Axiom, Sage, GAP, ...) embed mathematical knowledge (partial formalization) to:
  - structure their library
  - model closely the mathematics
  - define modular API's
  - enable composing its building blocks
- The few-concepts-many-combinations feature goes in the way of traditional computer-science best practices: **decoupling**, **separation of concerns**. This led to some **innovative designs**:
  - Bespoke method-resolution mechanism (GAP)
  - Object orientation using categories (Axiom, MuPAD)
  - Standard Python Object Orientation + mechanization of the generation of the large class hierarchy (SageMath)

Knowledge for cross-system modularity

- A not so successful attempt: OpenMath
- A tentative reboot with OpenDreamKit's Math-in-the-Middle approach
  - Uncovering the knowledge already embedded in each systems
  - Aligning it through a central ontology
  - Exploiting this for automatic generation of adapters

A tension when (attempting to) modularize large systems

- The "library of books" approach (e.g. GAP)
  - Fosters small independent packages
  - Credits package authors
  - Leads to duplication (analogy: overlapping books written by different authors, with different notations)
- The "wikipedia approach" (e.g. SageMath)
  - Fosters joint ownership, harmonization of API's
  - Fosters cross-subject interrelations; hey, it's math!
  - Leads to one huge tightly coupled blob

Open discussion

- How to foster a sustainable and agile ecosystem where (sub)components have their own life cycle, explore new approaches, compete, and happily die when no more relevant.
- What other barriers do you identify in Computational Math?
- How do the barriers differ from other areas
- Other case studies? solutions? approaches?

Appendix: some case studies and tentative solutions

OpenMath

- **Goal**: standardized serialization format to exchange data across systems
- **Syntax**: XML, json, binary
- **Semantic**: defined by content dictionaries (CD)
- **Hard problem**: agreeing on CD's for polynomials was already hard; scaling?

After twenty years, little adoption

The Math-in-the-Middle approach

- **Goals**:
  - Separate **serialization** from **adaptation**
  - Separate **binding** from **adaptation**
  - Separate the treatment of each system
- **Approach**:
  - A central math ontology
  - Formalize each systems by aligning to the central ontology

Categories in GAP, Axiom, MuPAD, SageMath

- **Barrier**: compose building blocks `A1` and `A2` into modular generic code `B`: `f(a1, a2)`
- **Requirements**: **uniform modular API + method resolution mechanism**
  Remember: few core concepts, hundreds of interesting combinations
- **Approaches**:
  - Axiom, MuPAD: OOP with large hierarchies of abstract classes (categories)
  - SageMath:
    - same, but even larger
    - based on standard OOP (Python) + infrastructure to scale
    - embed semantic knowledge at single point of truth; exploit it to automatically generate the class hierarchy
  - GAP: bespoke method resolution mechanism

Case Study: libsemigroups:, a library for computing with semigroups

Author: James B. Mitchel et al.

- **Problems**
  - Speeding GAP's semigroup package while increasing its availability
  - Use a low-level library `A` from a system `B` written in a different language
- **Approach**
  - Rewrite as a standalone templated C++ library
  - API: many iterations until converging to a natural API (no memory management, . . .)
  - Bindings: from Python: on-the-fly binding using cppyy (also tried Cython, Pybind11, . . .)
  - Adaptation: a thin layer for now; could use alignments

Case Study: the Sage / GAP interface

- **Problem**
  Use funtionalities from a system `A` in a system `B` written in a different language
- **Data exchange**
  - Objects returned as references (handles):
    - Few conversions
    - Reduced overhead
    - Manipulate from B objects of A with no native representation in B
- **Binding**
  - Originally: a pexpect interface
  - Now: ABI (direct calls at the C level)

- **Adapting**
  - Currently: monolithic adapters for some cases (groups, . . .)
  - In progress:
    - modular adapters, exploiting the category hierarchy and semantic alignments
    - objects returned as reference + semantic
    - alignment `B.cardinality() -> Size(A)` attached to the category of sets
    - alignment `B.conjugacy_classes() -> ConjugacyClasses(A)` attached to the category of groups