Project: Mathematical Knowledge Management

Participants: imode, stoopkid, dmiles, aindilis, b3nszy

Abstract:

MKM seeks to formalize mathematical knowledge with CYC.

Use Cases:

Automated Theorem Proving, Intelligent Tutoring Systems,
Generalized Program Synthesis and Analysis.

- with security/complexity guarantees
- hardware (hypothetically)
- machine languages
- higher level languages
- MKM language itself (i.e. Godelization)

Routing over transitive relations, like equality, implication

- data-format translation as a networking problem

Crowd-sourcing code with guarantees
Semantics for semantic web & semantic wiki
Open-source governance (representing the semantics of it)
“World-building”

Resources:
https://pastebin.com/A8Rbm7Cv
https://frdcsa.org/frdcsa/internal/perform/

stoopkid’s current stack of choice:
some of these layers are potentially overlapping
some of these layers may be able to be smashed together entirely

1. machines
2. machine languages
3. generic rewriting engine
4. generic deductive engine (i.e. prolog, CYC, etc..)
    a. write the rules for your foundation of math
    b. untyped lambda calculus as theorem-proving framework
        i. inconsistent
    c. simple type theory
    d. polymorphic type theory
    e. dependent type theory

   f. homotopy type theory
   g. modal logic
   h. pure type systems
5. consistent foundation for math, as database (ideally constructive; i.e. dependent type theory, homotopy type theory)
   a. fixed & finite / finitely-expressible rule/axiom set / schema-set
   b. good feature to have: has executable constructive proof, via proof-simplification derived from logical harmony conditions on introduction/elimination rule-set pairs
      i. note: non-termination of execution via proof-simplification is problematic if allowed without restriction, but this does not necessarily mean it cannot be allowed. the problems it causes are:
         1. causing dependent type-checking to fail to terminate
         2. conflicting with some assumed finiteness of the terms implied by the rest of the rules, example:
            a. if you can prove that there’s no Nat x such that x + 1 = x, then if you can define a term badNat = badNat + 1, then you’re gonna have a problem
         3. confer: data vs. codata
   c. necessary feature to have: has proofs about executables
      i. possibly via proofs about its proofs, as in the case of (a), as they are executable
      ii. proofs about models of other execution environments; can model and prove things about Turing-complete computing systems
         1. note: if it has executable proofs as in the case of (a), they do not need to be Turing-complete or even allow any non-termination in order to achieve this
      iii. self-representation, i.e. Godelization
   d. proof-checking must terminate, ideally with low complexity except for when the user specifically directs the proof-checking to run some function of potentially arbitrary complexity
6. mathematical proof
   a. results about pure structure
   b. 2 + 2 = 4
   c. addition is commutative
   d. “exists injection/surjection” is transitive
   e. function composition is associative
   f. functions respect equality
   g. functions distribute over the components of an if-then-else statement
   h. every regular language corresponds to a regular expression and a finite state machine
   i. abstract algebra
7. “world-building” as “rules under hypothesis”
   a. basically defining your world as abstract algebraic structures

i. Groups/Rings/Fields/Monoids can be seen as being types of worlds
ii. in dependent type theory their structure and properties can be expressed as the field-types in a record type
iii. you ask whether there actually exist such worlds, i.e. whether the laws defining the abstract algebraic structure are satisfiable or not
iv. in constructive logic, any such proof of satisfiability would essentially be an instance of the abstract algebraic structure in question, i.e. a proof of satisfiability of the Group laws would be an actual Group (i.e. a set with a binary operation) paired with everything necessary to interface with the properties specified by the Group laws (i.e. having identity element, inverses, associativity, totality)
v. in other words, the proofs are (or contain) machine-interpretable models in the sense of model theory
    1. universal algebra + logic = **model theory**.
    2. https://en.wikipedia.org/wiki/Model_theory
vi. now generalize this to essentially arbitrary kinds of constraints, i.e. be extremely liberal with what kind of abstract algebraic structures you're considering:
    1. Example:
       https://github.com/sto0pkid/CategoryTheory/blob/master/Agatha2.agda#L96

       Yes this is an abstract algebraic datatype
       Any instance of it happens to satisfy this property:
       https://github.com/sto0pkid/CategoryTheory/blob/master/Agatha2.agda#L158

    2. "prolog in record types" / "prolog under hypothesis"

b. real-world knowledge modeling
c. machine modeling
    i. hardware logic verification
d. language modeling
    i. C/C++ language specifications
    ii. RDF specifications
    iii. English
e. scientific modeling; perfect example of why we would place the world-building layer under hypothesis of the actual mathematical framework rather than allowing proliferation of rules attempting to express real-world concepts being taken as "truth":
    i. Newtonian mechanics?
    ii. Relativity?
    iii. Some kind of quantum gravity string theory?

        iv. Biocentric universe theory??
        v. Flat-earth theory?!?!?
    f. constraint specification / “specification specification”
        i. “this program must satisfy such and such properties”
    g. self-representation i.e. Godelization
        i. note: the fact that a Godelization actually corresponds to the formal system we’re working in, and the results based on this, exist in the metatheory, which from the perspective of the formal system might as well be out in the inaccessible “real world”
        ii. self-compilation

8. personal knowledge-base
9. networking
    a. semantic web
    b. crowd-sourced logic & code
        i. with guarantees up to whatever constraints are specified in the types it’s annotated with
    c. packaging and distribution management
10. inferencing engines
    a. automate reasoning over the knowledge-base below this layer
    b. can apply essentially arbitrary methods; proof-checking guarantees you will never put anything into the system that breaks consistency
        i. neural net that learns how to prove things?
            1. using proof-checker as a free “trainer”/validator?
            2. derive training sets from problems we actually want to solve?
11. interface
    a. accessible proof-tactics framework
    b. immersive environment (like a semantic wiki)
    c. LATEX support
12. execution of general non-terminating algorithms / Turing-completeness recovered (again) as infinite sequence of user-interactions, doing one terminating step at a time
    a. where a “user” is any physical thing that can perform the interactions
    b. can describe the individual steps of an arbitrary Turing-machine
        i. just need something to keep pushing the “step” button

+ Andrew’s use cases:

I am interested in the A.I. as theorem proving paradigm. I proved a (trivial?) result that shows that in order to increase the deductive closure of theorem proving systems it is ultimately necessary that the theorem proving program increase in size:

https://frdcsa.org/frdcsa/introduction/writeup.txt

So my goal is to create a substratum upon which we can have a lot of solvers for different theorem proving and related computational tasks. Due to Curry Howard we have that theorem provers are programs. So I collect programs in the hopes that some of them will increase the deductive closure. (Collecting software is the quickest means to augment your system with most likely meaningful complexity). For instance, I recently started packaging for Debian GNU+Linux about 180 more software systems found here:

https://github.com/johnyf/tool_lists/blob/master/verification_synthesis.md

So far I've made 256 packages of various AI systems, but my intention is to scale that to the hundreds of thousands, noting that the AI effect means that AI really is a designation for the unsolved problems in fields, and that what is and isn't AI software forms a spectrum.

I take the approach that whenever a person has a problem of any sort, they enter that into the intake part of the system. The system then works to solve as many of these problems as it can, with emphasis on those problems whose solutions go on to solve other problems for more people.

One of my projects is the Textbook Knowledge Formation project, which seeks to translate textbooks semi-automatically into CYC or Prolog. (see https://github.com/aindilis/nlu-mf). It is similar to RKF and a project from AI2, probably Vulcan or something. The goal here is to translate all literature, not just mathematical, but I'm using mathematical literature to bootstrap the process because of the affinity between mathematics and CS KBS systems.

I have over 600 internal projects that serve to glue the external code together, Glue AI I think I've heard it termed by dmiles. (see https://frdcsa.org/frdcsa/internal and https://frdcsa.org/frdcsa/minor).

So some more specific use cases: program synthesis. It should be possible to create an intelligent agent which debates how to implement solutions to various algorithmic problems. For instance, sorting has implementations like quicksort, etc. So for a given problem definition, the software should be able to use its mathematical knowledge to rule out certain approaches to the problem, and to help guide the search for the algorithmic solution.

I believe that the totality of mathematics/logic/CS (as well as the rest of human knowledge) cannot fit into one brain, but it can fit into a digital mind that lacks conventional mortality, and that creating such a digital mind is the paramount occupation of our time.

MKM can not only aid in theorem proving but also with pedagogy, and help to answer questions directly wrt mathematical knowledge for people who are learning or relearning respective areas of mathematics/CS.