GSoC 2021 Proposal

Symbolic Integration

By Miguel Raz Guzmán Macedo

Summary

This project aims to contribute an indefinite symbolic integrator written entirely in the Julia language. It is based on a heuristic approach, taken wholecloth from the RUBI project. RUBI is written in Mathematica and seeks to employ the patter-matching programming model to its fullest extent: They separate over 7000 (mutually exclusive) integration rules and traverse a binary tree to find the correct rules to apply. It's main advantages are a 10x speedup over the basic Mathematica integrator and a larger test problem coverage. **Concretely,** the goals for this project are to reach reasonable parity coverage with the RUBI test suite first, and second, a performance improvement over RUBI of at least 10x.

The project

- At the end of this project I aim to deliver a package that can calculate symbolic indefinite integrals at the level of freshman calculus.
- I think that the Julia community, pure and applied fields at large are interested in this project because it's a long awaited feature in the Julia ecosystem.
    - Symbolic integration is a fundamental research and educational tool - verifying calculations within Julia is extremely useful without having to pay a Mathematica license.

Previous hurdles

- There are quite a few hurdles to encounter, and I can gladly say I've overcome already a few of them.
    i. RUBI is an ongoing project, so automating the translation is necessary. The main (and mostly single) author of RUBI does not follow SemVer, and thus has made replication of work more difficult.
    ii. Setting up a Mathematica to Julia pipeline is easier said than done, and it's worse when the public facing code is in the form of Mathematica `.nb` files.
        - Several approaches that I tried, for days weeks, before coming to the last (moderately succesful) approach.

a. Try to parse the `.nb` notebooks as given on the site. Doomed to failure - the notebooks are in a nested format already, bloated, and would have needed to write a Mathematica + notebook parser wholecloth, whilst stripping metadata. A few days of this and you realize no extensions are made to easily port `.nb` files to Jupyter to Julia and you're just solving a different problem.
b. Try to parse the `.pdf` files, hope those don't have as much cruft. Succesful for a bit with `PDFIO.jl`, but very brittle. Many of the assumptions metadata needed for integrating (like, say, stating that `n` is a positive real number) got completely broken through several different lines. This also proved unsusteinable.
c. Ask myself if someone hasn't already written them in a nice format, perhaps we can still it off somebody else with MIT licensing. The SymJava project did this, but I don't speak Java and they didn't seem to share a single file that could be easily reused. This would also set a dependency on SymJava for any RUBI updates, which is unfortunate.
d. Finally, find a couple of abandoned repos that **do** happen to have the 7540 rules in a `.m` format. and parse *those* with some handy regex and `FileTrees.jl`.
e. Poke around for a few more days until you find the one forum the author answers design questions and current state of the project.

- At this point, Rubin.jl now contains 7540 parsed rules (with some hiccups.) in a huge JSON file. This will aid in sharing with other projects for benchmarking and easier portability.

iii. Now that we have all the information in a huge JSON, begin a Mathematica to Julia parser.

a. Cry and gnash teeth for 15 minutes, since I've never done parsing before.
b. Start poking around for how to handle parsing.
c. Realize that using regex does not necessarily solve all our problems, since we have to change the format because of parsing ambiguities and how assumption metadata is handled. Concretely, Mathematica can pick up a new pattern matcher with `lhs := rhs ;/ conditions`, and unfortunately not all of those conditions are handled appropriately. (Bonus points - some conditionals use chaining `/;`, others use `&&`, just for fun.
d. Oh right, I forgot the cardinal rule of software - google if someone hasn't solved my problem before! In a sense, un/fortunately. `Sympy` has a `sympy.parsing.mathematica.parse` function, but it breaks with our examples because they don't have as many special functions baked into their allowed cases, they ignore inert trig functions to avoid extending the pattern matching tree depth, and they can't handle local scope `With[...]`. The first point is patchable with a list of all the function names that you can regex in `src/rubirules.json` and stash into the custom dictionary that `sympy.(...).parse` allows, but the non-uppercase inert trig functions blow up the function so a Julia rewrite has to be done anyways. The third point however is not as

simply patcheable - `With[...]` introduces local scope renaming, and that backtracking logic is not so easily handled by a lone regex attempt, so that attempt also meets a deadend.

e. Final design: parse the `lhs := rhs /; conditions` into 3 separate strings, and then use a `WolframKernel` to run `FullForm[Hold[...]]` on that string alone. That gives us a full Mathematica style S-expression that can be much more easily transpiled to Julia.

f. For now, skip the 962 functions with local scope `With[...]` and the verbose-by-default `$LoadShowSteps` 26 functions to handle separately.

In summary: these coding adventrues involved learning Mathematica basics and its parsing rules, as well as pattern matching internals and the regex to handle it. I had to communicate with the author on the current state of their project, technical design decisions that affect performance, read the documentation and code internals of Sympy and SymJava projects, and leverage different parts of the Julia ecosystem to try and solve my problem. On the Julia side, I've used FileTrees.jl to parse in parallel the entire notebook and test suite in under 5 seconds both, JSON3.jl to transform the structs into reusable data, and external processes within Julia to invoke the Wolfram kernel and interoperate with Julia. This is all work that I've done in the Rubin.jl repo which I believe presents an honest rendition of my efforts so far along with this document. Additionally, the use of `ArtifactUtils.jl` to setup a reproducible build for other maintainers is a quality of life addition to help onboard other developers. A `Pkg.instantiate()` should resolve downloading both `Rubi-4.16.1.0` and the `MathematicaSyntaxTestSuite`.

Future hurdles

- Handling assumptions at the type level is still an ongoing discussion with the expers in Zulip. My plan to get around it is to encode that metadata in a JSON format so that translating to a Symbolics.jl or Methatheory backend involves no more parsing but only accessing a struct and formatting.
- Simplification: The integration routines rely on being able to simplify in different steps of the process, and this can get expensive. Instrumentation to figure out worst and best case approaches is a potential idea to measure when it's worth full simplification. Additionally, this project will defer simplification to a symbolic backend. I think this is a legitimate separation of concerns and should not be the focus of Rubin.jl until proven otherwise.
- Performance: instrumenting the code base will be necessary to find performance pitfalls. This includes counting the rule pattern matches and the allocations during simplification. Potentially, mulitthreading with atomics could help cut down on allocations in Metatheory.jl's egraph approach and I am in contact with the author to investigate different ideas there.

Milestones

1. A JSON file of RUBI rules with `lhs` , `rhs` , and `conditions` .
2. A JSON file of RUBI rules with `full_line_capture` verbatim, `lhs` , `rhs` , and `conditions` fully parsed in mathematica.
3. An implementation of the symbolic integrator based on Symbolics.jl and/or Metatheory.jl. Since neither has fully fleshed out how to handle assumptions fully, it's best to stay agnostic as to which will be the first candidate, although targetting both is expected.
4. A JSON file of RUBI test suite with `integrand` , `optimal_steps` , `answer` , and `level` (answers may be equivalent, but involve higher level functions than needed, eg using hypergeometric functions which are redundant when simplified).
5. A full benchmark run of Rubin.jl vs RUBI - (and include per test case timings in an aggregate format, unlike RUBI.)

Deliverables

- [X] JSON files of RUBI Rules and RUBI tests
- [ ] Mathematica to Julia transpiler, ignoring local scope
- [ ] Implement 1 Julia symbolic backend
- [ ] Maximize test coverage, benchmark against 12000.org
- [ ] Documentation, tutorial videos.

About me

I am a final year physics undergraduate student from Mexico. I've been coding in Julia for a few years as a hobby, and I wish to take onto myself a more focused and professional development of my Julia skills. I love the Julia community and have definitely contributed to open source before. I find it a great place to grow and meet people - but it can be a lot starting out. That's why I've made an effort to document some cool things about the language (like it's awesome REPL, in a tutorial video). I really like helping others learn about Julia and I think I've gotten better at it over time in the community.

The code that I would most proudly present for this application, not because of its elegance or prettiness, but because of the grit, is the `utils.jl` file in Rubin.jl I've documented before on all the previous hurdles I had to overcome in writing and rewriting that code, and I'm proud of what's recorded (in literate programming style) there.

I'm also involved in the Julia D+I efforts, and I've made several spanish speaking outreaching presentations to spread the use of Julia in Mexico and LatinAmerica (a recent recorded talk is here).

Logistics

I wish to be contacted via Slack (I'm Miguel Raz there as well.) if possible, or my email with this application is fine.

I have no other commitments and would be absolutely thrilled to finally complete a GSoC. I owe it to the community.