Software for indefinite integration

Corrundum.red – Integration for all

A. C. Norman[1], D. J. Jeffrey[2]

[1]Trinity College, Cambridge, U.K.
[2]Dept. Mathematics, The University of Western Ontario, London, Ontario, Canada

Abstract

The Rubi system for the evaluation of indefinite integrals was developed by Albert Rich over a period of more than 15 years. His unexpected death has led to a number of discussions on what can be done to develop the system further. One possible direction addresses the fact that Albert Rich's development was built entirely on the Mathematica system. A number of people have considered the porting of Rubi to a variety of alternative computer algebra systems. The requirements for porting are discussed.

Keywords

Indefinite integration, Anti-derivatives, Rule-based Integration, RUBI database, Reduce, Mathematica

1. Introduction

For over 80 years, a key resource regarding indefinite integration has been Gradshteyn and Ryzhik[1]. It has also been maintained and updated over the years, the 8th edition [2] being released in 2015. The modern alternative is one of the computer algebra systems. These have made major progress since the 1960s, when Slagle's SAINT[3] and Moses's SIN [4] were released. The web site 12000.org[5] reports testing 9 currently available systems[1] (commercial and public domain) against a test suite of 106812 integrals. As with any test suite, it can be criticized for showing biases, but it covers all the basic integrals and is of a scale that makes it hard to ignore. To a good approximation it is a concatenation of all the other significant sets of test cases that its authors could find.

The commercial systems have the highest success rates, with the open source systems significantly lower, save that the permissively licensed Rubi shows outstanding capability. We have therefore been studying Rubi to investigate the feasibility of its use with systems other than Mathematica, which is currently the foundation on which it is built.

We view this as being potentially valuable for four main reasons:

1. Many systems are still unable to find integrals for a significant fraction of the examples in the above torture test. This is likely to apply to almost every case that requires advanced special functions to express the result. Perhaps for many users this will not be a serious limitation, because their problem will not involve say the Fresnel S function. But measured against Gradshtein and Ryzhik it is a limitation.
2. Where results are generated they may often be bulkier than would be ideal. This can go as far as returning a result expressed using elliptic integrals or incorporating imaginary numbers when something more elementary would serve better.
3. Something that may count as a special case of the above is the treatment of multi-valued functions and delivering an integral in a form that handles them and their principal values consistently. This can be of extreme important in integration when users substitute in endpoints to find a definite integral.

SCSS 2024: 10th International Symposium on Symbolic Computation in Software Science, August 28–30, 2024, Tokyo, Japan
acn1@cam.ac.uk (A. C. Norman); djeffrey@uwo.ca (D. J. Jeffrey)
0000-0002-2161-6803 (D. J. Jeffrey)

[1]Mathematica, Maple, Mupad, Maxima, FriCas,Giac/Xcas, SymPy, Reduce, Rubi.

4. With reasonable generality, the existing schemes return integrals with no commentary available as to how they were derived, and the techniques used are embedded in large bodies of code which will be unavailable when one of the commercial systems is used, but opaque even if an open source system is selected. With Rubi a commentary on how results were obtained is almost automatically available.

2. What is Rubi?

The RUle-Based-Integration system Rubi is a public domain project addressing indefinite integration [6]. It consists of two parts. The first part is a database consisting of transformation rules which convert an integral expression into a simpler form, allowing an iterative application of the rules to evaluate the initially given integral. The second part is a collection of utility functions, at present coded in the Mathematica language, which massages mathematical expressions into forms suitable for the application of one of the transformation rules contained in the database.

Readers not familiar with Rubi may appreciate a little background concerning the project. Rubi was started by Albert Rich (1949–2023) circa 2007, after the development of the Derive computer algebra system was discontinued [7]. It is based on a term-rewriting rule-based paradigm, which was expected to bring the following benefits.

- Speed and compactness.
- An ability to display the steps taken during an evaluation.
- Results which are correct in the complex plane.
- Results which are in the most compact and æsthetically pleasing form possible.
- A system which is readily modified and extended.

Although Albert Rich corresponded with many others, and received their suggestions for integration rules, he largely worked alone, and as a consequence, only lightly documented his program code.

2.1. The current state of Rubi

There are presently two versions of Rubi available. The first is 4.16.1; this is the last version whose release was overseen by Albert Rich [6]. The other is 4.17.3, which has been retrieved from material left by Albert Rich and implemented by supporters of Rubi [8]. The versions are very similar and can be summarized as follows.

- There is a database containing over 7000 rules, recorded in Mathematica syntax. It is in the public domain. They rules can be read in mathematical notation in PDFs from the Rubi website [6].
- Rubi includes several test suites of integration problems. These form part of the test suite used by 12000.org.
- The order in which the rules are placed can have an effect on the performance.
- The system consists of more than the database of rules. There are large support files, also written in Mathematica syntax, defining many auxiliary functions. These functions serve a variety of roles.
  - Testing the properties of a parameter. For most integration problems, the parameters in the rules are given numerical values. Thus, it is assumed that a parameter can be tested to see whether it is positive, or an integer, and so on.
  - Applying an algebraic transformation, such as extracting the content of a polynomial, making a partial fraction expansion, etc.

  Unfortunately, there are very few comment lines in the files containing the auxiliary functions. This makes working with them difficult.

2.2. The way forward

It is unlikely that a person or persons will come forward and take over the development of Rubi. The attitude of the *de facto* custodians of Albert Rich's legacy is that Rubi is frozen in its current state. A person wishing to continue Rubi is allowed to use the database as they see fit, but any development will be a new project. The analogy being made is that anyone can read a scientific paper and then write their own paper with whatever new thoughts the author has. The author references and acknowledges the earlier work, but is *au fond* working on a separate project.

There are several ways in which Rubi can be developed further.

- More integration rules can be developed.
- There are a number of integrals in the test files which cause Rubi to disappear into a loop. A modification of the data base could reduce the occurrence of loops.
- Rubi could be transported to other languages, such as Maple, muPad, etc.

For those of us who do not have Mathematica installed on our computer (perhaps because of cost), or who really value being able to inspect the workings of software whose results we rely on, or who are developers of other algebra systems (present and future), Rubi looks a very attractive starting point. As already mentioned, at present Rubi 4.x.x exists in the form of a large set of Mathematica rewrite rules and a significant but not-too-large collection of utility code to back them up. Albert Rich, however, had been planning a transition to Rubi 5 that would transform the rule-set into a decision tree that would not rely to the same extent on pattern matching in the system it was installed on. That was expected to make it much easier to migrate Rubi for use with other systems. Unfortunately this part of the project remains incomplete.

The work explained here is called "Corrundum.red" and is a step towards making full Rubi available outside Mathematica. While it is being developed using the Reduce system[9][10], the intent is to keep reliance on the algebraic facilities in Reduce fairly low and reasonable well explained, so that adaptation elsewhere might be more readily possible. In particular the rule-rewrite engine does not rely on any such engine already built into Reduce; thus exactly what it does can be made explicitly visible.

The name "corrundum.red" is of course something of a pun, in that corrundum is crystalline aluminium oxide with hardness 9 on the Mohs scale, and when tainted with chromium so that it shows up as a vivid red colour you obtain the gemstone ruby. Of course ".red" is the file suffix used by the Reduce algebra system. So this name references Rubi (which is a sometimes-used alternate spelling for ruby); it links to Reduce and is intended to suggest both toughness and high value.

There have been multiple exercises to leverage the Rubi rule set in the past, so we are not the first to look into this. In fact, a previously abandoned investigation, using Reduce, was conducted by one of us (ACN).

We of course note the existence of "symja" or $S\Psi MJ\Lambda$ which aims mainly at support for Android and is all coded in Java. That has at its core an implementation of a language rather closely modelled on the one used by Mathematica, and as a result they have been able to import Rubi wholesale leading to very impressive capabilities. For our purposes this provides excellent confirmation that emulating enough of Mathematica to make Rubi work is feasible, and it provides a non closed-source version. But the Mathematica-style support there pervades so much of Symja that it does not help to guide those who want only enough of it for use in some alternative system that does not start off in Java and does not start off with Mathematica-like syntax and semantics baked in.

Especially with Albert Rich's work migrating Rubi from a rule-based system to one making transitions based on a decision tree, significant parts of Rubi were made to work on Maple, however again that project seems never to have reached a stage of full support for all the rules. https://de.maplesoft.com/company/casestudies/stories/maple-for-accuracy-and-support.aspx?L=G

Similarly Julia has the adoption of Rubi on its 2021 roadmap: https://juliasymbolics.org/roadmap/ but the github repository that we found https://github.com/ufechner7/Rubi.jl has been inactive for some years.

With all this background our work has many precedents. We hope that this paper will explain, in more detail than we were able to find elsewhere, some of the particular challenges in this activity, while we attempt to extricate Rubi from Mathematica syntax, semantics and support, and thus be of interest to others who want to revive previous attempts or start new ones.

Corrundum.red has a number of components and these will be explained in the following sections.

3. Capturing the Rubi rules

A parser for the subset of Mathematica syntax used in the transformation rules was coded for Reduce. Happily a rather naive recursive descent parser proved sufficient. Anyone at all familiar with this style of parser would observe that it is basically merely one procedure to correspond to each entry in the BNF rendition of the syntax involved. This naive parser was first created a decade ago and used to start an investigation of a version of Rubi from back then: that project stalled and the revived parser now had to have some extensions to cope with the Mathematica syntax that has meanwhile crept into Rubi and its utilities. Over that decade Reduce has been provided with a parser generator using the standard LALR scheme which could have made this scheme for reading Mathematica style text easier and much more compact, but re-using the existing almost-working if cruder scheme seemed the simpler path. And of course since this is just parsing expressions and not a full language the naive approach is not seriously offensive. We believe it would also have been possible to make Mathematica export files in a very similar format.

The result of a parse is the full set of rules expressed as Lisp-style data structures with a collection of operators that denote the various things that Mathematica provides. So for instance the very first Rubi rewrite starts as

```
(* ::Code:: *)
Int[u_.*(a_+b_.*x_^n_.)^p_.,x_Symbol] :=
  Int[u*(b*x^n)^p,x] /;
FreeQ[{a,b,n,p},x] && EqQ[a,0]
```

and becomes

```
(/;
  (:=
    (Int
      (times
        (_. u)
        (expt (plus (_ a) (times (_. b) (expt (_ x) (_. n)))) (_. p)))
      (_ x Symbol))
    (Int (times u (expt (times b (expt x n)) p)) x))
  (and (FreeQ (bracelist a b n p) x) (EqQ a 0)))
```

Actually in the file as generated all punctuation characters in the parenthesised form are preceded by escape characters so that for instance "/;" and "_." are seen as merely simple symbol names.

This adjusted format explicitly represents a parse tree and it is likely that internally Mathematica turns the raw input into something equivalent – but what we have here is totally unambiguous and entirely suitable for processing with any scheme that is happy to work with tree-like data structures. At the data structure level Reduce works with Lisp-style data and so it is especially comfortable with this, but it would be close to trivial to write code to read this tree notation in almost any language. Given that a simple tree-walk could re-display the material in the syntax most suitable for some other system, be it re-creating Mathematica syntax or converting for Maple, Maxima, Axiom or whatever.

It can be seen that a rule generally has three components. A template or pattern which here is basically $\int (u * (a + bx^n)^p, x)$ and where the various annotations are to mark various things as parameters or wildcards. Then there is the transformed version, which in this case purely serves to remove the a that

is originally present. Finally there is a set of conditions. These indicate that the various parameters must all be independent of x and that this transformation that gets rid of a is only applicable when $a = 0$, in this case reminding us that the actual expression may contain a sub-expression equivalent to 0 but starting off looking more complicated. Of course we all know that identifying when a general expression is zero is undecidable!

There are almost 7500 of these rules! Using them involves facing up to (at least) two significant challenges. One is that of gaining a sufficiently full understanding of Mathematica pattern matching so that its behaviour against this rule-set can be re-created. The other is that throughout the Rubi rules there are function calls in both conditions and replacements that invoke a range of Mathematica primitives. Some (such as FreeQ here) have behavior that is simple to understand, but the precise expectations and limitations of even something as obvious looking as `EqQ[a,0]` have to cause some pause for thought.

4. Pattern matching and the Utilities

When one first admires Rubi, what most catches the eye is the set of simple-looking tree rewrite rules. These basically look at the structural form of integrands and on that basis perform transformations. Furthermore it is on record that Albert Rich's plans (for a Rubi 5) involved transforming the identification of which rules to apply into a decision tree – in effect a giant nest of "if" statements.

Thus when we started work, in a spirit of optimism we expected that, with the rule-set expressed as data structured such that Reduce could manipulate it easily, our main task would be to get simple tree-rewrites involving 7000+ rules to run reasonably efficiently.

After some while it became clear that things were not quite that straightforward.

1. When one writes `Int[expression, x]` in Mathematica, the expression delivered by Mathematica to pattern matching definitions of `Int` is neither the unaltered input nor a fully simplified version of it. It is somewhere in between. If Rubi is ported to any other world, this pre-processing is not likely to be replicated precisely, and so behaviour will differ. As a tiny example consider the Mathematica definition `f[a_*b_] := {"product", a, b}` which decomposes a product. The input `f[x+x]` reveals that what Mathematica sees for matching is the product of 2 and x, while `f[x*x]` is not seen as a product at all but as a power. `f[(1+x)*(x+1)]` is again seen as a power, while the more elaborate `f[x*((x+1)^2-1-x^2)]` is not seen as having an argument that is equivalent to x^2 and so is reported as being product. All of this is perfectly proper to the extent that it is Mathematica doing what it chooses to do, but it does make it harder to think about testing Rubi, since the exact form of expressions it sees have been subject to this not very clearly documented adjustment.
2. Mathematica pattern matching is powerful and it is aware that addition and multiplication are both commutative. Thus a pattern of the form `Int[a*x^2 + b*x + c, x]` (here the underscores are omitted to make the presentation neater) will match a quadratic regardless of the order in which the terms are presented. So in some sense a rule with that pattern as its left-hand side represents 6 cases corresponding to different term orderings. At one stage we considered pre-conditioning the rule set so that all of those cases were explicitly shown and hence the actual matching would not need to worry about this issue and hence might be simpler and faster: it became apparent that doing so caused the size of the rule set to explode beyond reason.
3. A further cleverness of the Mathematica matcher is that in certain cases some parameters can have default values. This is indicated by using `"_."` where they are mentioned in a pattern. With this scheme the pattern `a_. + b_.*x_^n_.` will match not just `p+q*z^k` but also variants on that with `p=0` and/or `q` and `k=1`, all the way down to just `z`. If one combines the consequence of this with commutativity this one pattern can match a dozen different input forms including say `z*q+p`. One rewrite rule can contain multiple instances of these two shorthands and we found that if everything was expanded out to show the full range of cases to be accepted that we could

end up with well over a thousand cases following from what was presented as a single rule. This astonished us!

4. The matching that is made has to be subject to some conditions - which are given at the end of the rule following the symbol `/;`. A first thought is to perform structural matching first and after that check if the constraints apply. However consider a case where the pattern is (without the underscores here) `a + b*x^n + c*x^m` and the constraints include `m=2*n`. Now commutativity shows that the two sub-patterns `b*x^n` and `c*x^m` could match input in either order, but only one is will suit the constraint. The most naive approach would be to insist that the syntactic matcher returned not just a single match but all possible ones that arise based on commutativity and default values and that the constraints are then used to determine which (if any) are viable. In some cases this would return very many matches. It feels more reasonable to make the structural matcher able to deliver its first match and then backtrack to look for another if that becomes necessary. Of course in the worst case just as many variants could have to be tried.

There are ways to address all of these issues. The pre-conditioning‘of input for instance to sort constant terms in sums and products either to the start or the end helps substantially with commutativity. Where we have arrived at is that we index constraints by the parameters they depend on. Then when our pattern matcher is about to ascribe a value to a parameter it checks if that completes the information about everything that the relevant constraints will use. It can thus test the conditions during matching and as soon as possible, and this of course saves us from investigating parts of an expression once we have discovered that that will be futile. With this we then accept the first viable ordering for any commuting argument list.

We are aware that our early-check scheme could be unsatisfactory for a fully general set of rules. Consider a complete pattern that contains both `(x^n+x^m)` and `(x^p+x^q)` and then pathologically a constraint on `n`, `m`, `p` and `q` that is not symmetric in them. The orderings for matching the two commuting sums would need to coordinate. We believe there are no such situations in the Rubi rule-set, so our matcher is not fully general but should cope with requirements here. This illustrates that an implementation of Mathematica-like matching that just has to cope with a fixed set of rules such as those in Rubi may be able to be simpler and hence potentially faster than one that needs to come with the most general case.

The matcher we have coded is essentially a modest size body of Lisp code and of itself it is independent of Reduce. It is intended to be reasonably clear and as concise as we can manage, so for instance to cope with a commutative operator it forms a list of all permutations and then scans the list reporting the first occasion it finds a match. It would clearly be possible to fold the enumeration of permutations with checking them but for a first version and for “work in progress” we preferred simplicity over potential performance gains. In a similar way and although we have thought of many ways that will allow us to do better, we have started with checking input expressions against each of the 7000+ patterns in turn and we have avoided exploring the interesting rabbit-hole of indexing, expression signatures and hashing, “trie” structures and expansion of the matching process into executable code. Those are all for further investigation at a later stage.

5. Predicate and Replacement material

On starting this project we expected that pattern matching would be the core of the work. Well, the conditions applied to rules contain many predicates that are obvious and easy to support. For instance for a very large proportion of parameters there is a check that the parameter does not depend on the independent variable. For exponents there can be checks such as that n is not -1. The test $m = 2 * n$ is slightly less obvious but once encountered is easy to handle. So at first we felt we were making good progress as we provided more and more of those. Within the formulae that appear as results there are cases where n is numeric and $n + 1$ is used, and it makes obvious sense to perform the arithmetic. Again that was easy. Perhaps especially so as it plays to Lisp’s strengths and is all code that could reasonably have appeared in 1960s attempts at integration! Following this path and by providing a tiny

set of our own integration rules we were able to perform some first integrations really rather early in our work. At that time we felt greatly encouraged.

Beyond the trivial there are then a collection of Mathematica operations that do "genuine" algebra, but where the intent is clear and where any other host algebra system will have something equivalent. An example arises when a result will be of the form `Log[expression]` because if the expression has a constant factor (i.e. a non-trivial content with respect to the variable of integration) then that can be taken out and its only contribution to the result would be to merge it in with the constant of integration. So when Rubi uses a function `RemoveContent[]` that is liable to involve chaining to the underpinning algebra system and the exact details of how that is done can not be portable but the expected returned value is unambiguous.

Some Mathematica primitives as called here can of course represent special portability challenges and in the end the balance as to how much of Rubi's success flows from its own rule-set and how much from the power of (say) the Mathematica `Simplify[]` function (which will apply multiple transformations on its input and return the result that Mathematica things is nicest) is something that can not be answered until our work is complete.

However we then found that in both predicates and especially in replacements there were instances of function calls with significant depth. These are defined in a file `IntegrationUtilityFunctions.m`. The around 8000 lines of Mathematica code there include a great many small wrappers for simple operations that are not problematic at all, however there are other sections that gave us pause. An example is the function `Subst` which is fairly cryptically explained there as "Subst[u,x,v] returns u with all nondummy occurrences of x replaced by v and resulting constant terms replaced by 0." but where inspection reveals that it together with its immediate sub-functions represents around 800 lines of code. For a while that felt like a really severe road-block.

Our eventual response has been to recognize that it is necessary to view the Utilities file as a further set of rewrites to be handled alongside and using basically the same mechanisms as the main set of integration rules. This is possible because much of Mathematica's notation is close to being functional – all the "procedure definitions" in the Utilities file can, from only a slightly different perspective, be seen as "rewrites".

The Utilities use a distinctly broader range of Mathematica syntax and facilities than the main set of integration rules. This includes the various scoping constructs that interact with just how Mathematica interprets the meaning of symbols, catch and throw, mapping over lists with what are in effect lambda expressions and explicit in-line invocations of the pattern matcher. We have taken a pragmatic approach – when some utility function uses "fancy" Mathematica features but is in fact fairly small or simple we have just implemented a replacement in Lisp/Reduce so only large and messy functions need to be interpreted from their Mathematica form. We end up with what amounts to an interpreter for just enough of the Mathematica language to cope with Rubi, and a scheme where any particular function might end up either recursing in the interpreter to continue processing Mathematica forms or dropping out into our own independent code.

As we were working on that we had arranged that any attempt to use a function that had not yet been provided would lead to a clear message but that processing could continue (with Rubi typically just considering the rewrite concerned inapplicable and going on to try the next). This meant that by running the full almost 80K test cases through our code we could count which functions were blocking progress most often. At all stages this provided us with a form of priority list. For finer grain investigation we obviously also maintained scripts to test just a sample from the full test suite and also to try our own hand-picked cases so that they could be run with extra tracing.

For longer than we had imagined would be the case our corrundum.red could only deliver results for around 1% of the examples in the test suite and we perhaps started to feel a little despondent. On implementing some more Mathematica functions to get the Utilities going at one stage boosted that so we could solve almost 7% of the first 10000 test cases. The current state is that our code now does not report any dangling unimplemented functions but clearly the emulation of some of Mathematica is still buggy and for most examples the code yields a result that is a malformed formula. This is liable to be "just a bug" and so we hope it will soon be fixed! But this paper is explicitly about "work in progress"

and the comments here show that that is exactly where we are.

6. Current Results

On testing cases from the Rubi test suite our performance so far is quite lamentable, with the very best try to date managing to get proper answers for just under 7% of the cases! But this mainly reflects some residual Mathematica primitives that have not yet been coded and debugged enough to let all examples run to completion. However our code can show the transformations that it makes and so one can observe the rewrite engine both when behaving well and when not.

Here is a case of mixed gladness, where there is some degree of mixture between Mathamatica botation (eg Log) and Reduce (eg log):

```
Test case 57
Use rule 2778 on Int(Log(x),x)
transforms to x*Log(x) + (-1)*x
Use rule 2779 on Int(Log(x)^2,x)
transforms to x*Log(x)^2 + (-1)*xtimes(2,x*Log(x) + (-1)*x)
(~~~ test case 57 My leafcount 19 Theirs 15)
Input:

int(log(x)^2,x)
My result:

x*log(x)^2 + (-1)*xtimes(2,x*log(x) + (-1)*x)
Reference result from Rubi test file:

2*x + (-2)*x*log(x) + x*log(x)^2
```

"xtimes" is our rendering of the symbol \[Star] that Rubi introduces in some places where we suspect it is trying to prevent Mathematica from transforming products under its feet. You can see that at present we have not turned it back into ordinary multiplication on the way out. But you can also see that along the way Corrundum has used two of the Rubi rules (which we call numbers 2778 and 2779, those numbers being easily decoded in our source files) and that despite its ugliness our result here is not much bulkier than the "official" Rubi one and is basically correct. This and many other cases illustrates that the exact form of the official Rubi result depends on what amounts to post-Rubi term reordering and cleanup done by Mathematica, because in those cases our results can be equally correct but just (for instance) order terms differently.

Various of the more severe problem cases we face are going to be easy to track and trace from the sort of output we generate: first we show the input expression, then for each pattern that matches we document both the identity of the rule triggered and the resulting transformed expression. Given that this is all basically Lisp code it is then simple to set tracing on the functions that implement each step.

The above statement is correct in that debugging steps that are taken can be straightforward – however there may also be instances in which Corrundum fails to make a match that Rubi expects and perhaps relies on. This could either be because of limitations in our pattern matching code or because intermediate transforms have not left an expression in exactly the shape that Rubi expected - for instance because reaching that shape was a consequence of fine details of Mathematica processing.

To help us track issues of that sort we are taking a second track. We have taken the full set of Rubi rules in Mathematica format and added annotations that cause Mathematica to report on each rewrite it makes, as in:

```
Test case 2: Int[x Sqrt[1 + 3 x], x]
```

```
Rule 120
Rule 125
...
Rule 105
Rule 104

                      3/2               5/2
         -2 (1 + 3 x)       2 (1 + 3 x)
Result: --------------- + --------------
                27                 45
```

and by cross referencing this against output from our system we will be able to track rewrites that we do not make as well as debug ones that we do. Again the numbers attached to rules are ones that have to be interpreted by indexing into our versions of the source files – they are not absolutely fixed to the reference version of the Rubi sources. Also the rule-tracing is not the one that Rubi has as one of its nicer features - it is one that provides yet finer grain information about the rewrites that are applied.

This sort of trace also makes it possible to identify the modest number of cases where the current Rubi rule-set can lead to unending cycles of transformation, so that attention can be given to tidying up there. It also has the potential to let us identify both which Rubi rules are used most (so we can improve performance by checking for those ones first) and perhaps spotting ones that are never used at all.

As has been notes several times already this is a project still in full swing, but we believe it has already hit a number of valuable goals:

1. In understanding the current state of Rubi;
2. Understanding much more about the way in which it intertwines with Mathematica than any experiment that remained within the Mathematica world could;
3. Provide a somewhat freestanding rewrite engine to use the Rubi rules. The dependence we have on Reduce for algebra support is fairly modest (at least in our opinion) and (maybe when we get round to it) easy to document. The main engine we have is expressed as Lisp code (albeit presented in Reduce syntax that sugars it and may make it easier for outsiders to come to grips with it;
4. Get a system that can perform some integration using the Rubi rule-set, with a real prospect that getting a lot further is now "just debugging". We have a "proof of concept".
5. As we have developed this we have stashed away multiple ideas for performance engineering to make our system rather fast.

So this has been and remains an entertaining project to have been working on, and we have now reached a state where we would be willing to get others with suitable coding competence to join in to push the work further, either until we have something fit to merge into Reduce or to build on what we have done to consider support for say Maxima, Maple or some other system.

7. Some lessons mainly about Mathematica that we have learned

The characterisation here comes from the perspective of one who is unfamiliar with Mathematica and only confronts it through the prism of Rubi. This will of course provide a tinted and distorted impression, but it is nevertheless relevant while trying to reproduce Rubi's behaviour. So here we collect points, some of which have been alluded to earlier but are still things we know now that we did not at the start of this work.

To a first approximation Mathematica has two schemes for processing algebraic expressions. There seems to be a range of transformations that are automatically and universally made. These include cases like $1+1 \rightarrow 2$ but also $a*z*a \rightarrow a^2*z$ where terms in products are sorted and consolidated into powers when they are sufficiently identical, but not if they are equivalent but sufficiently textually different. There are then a wide range of functions that can explicitly perform operations such as expanding

products, arranging a common denominator and so on. Finally there are generic simplification functions that try out a larger or smaller number of the explicit transformations and return what they judge to be the nicest variant on their input that they come across.

We explain this here because a study of the Rubi sources shows clearly that Albert Rich found the need to work around some of the details of all this. Two cases in particular illustrate this by showing transformations only reasonably comprehensible as work-arounds to detailed Mathematica behaviour,

In Mathematica the standard trigonometric functions are named Sin, Cos, Tan and the like - with upper case initial letters. The Rubi rules and utilities in places introduce a parallel set called sin, cos and tan in lower case that are referred to as "inert". Rules first convert from the standard Mathematica versions to these ones and apply their own set of simplifications, but also at least potentially pass the inert expressions through more general Mathematica transformations where presumably otherwise the system would have "simplified" things in unwanted ways.

Somewhat similarly \[Star] is used for a variant on multiplication which is processed a little differently from the regular Mathematica one.

Finally the utility functions file contains a definition

```
FixIntRules[rulelist_] := Block[{Int, Subst, Simp, Star},
  SetAttributes[{Int, Subst, Simp, Star},HoldAll];
  Map[Function[FixIntRule[#,#[[1,1,2,1]]]], rulelist]]
```

where `SetAttributes` seems to be being used to make fairly broad adjustments to Mathematica processing in a way that anybody not familiar with both all the Rubi rules and their intended interactions and just what Mathematica will do with or without that directive may find challenging to decode.

A feeling that arises from all of this is that to support Rubi *fully* outside Mathematica it is perhaps necessary to emulate some of the corners of Mathematica behaviour where Rubi is then carefully taking steps to allow for the fact that it did not want them to apply!

None of this is to be viewed as a criticism of either Mathematica or of that state of the final Rubi snapshot, but it may suggest that the entanglement between them is non-trivial. A different view is that perhaps in reality the worries expressed here will end up impacting only a small fraction of the examples in the test suite, and in some future Rubi some adaptation or adjustment of rules may sort that out easily. Until our work is complete we can not tell!

References

[1] I. S. Gradshteyn, I. M. Ryzhik, Table of Integrals, Series, and Products, Gosudarstvennoe Izdatel'stvo Tehniko-Teoretieskoj Literatury, 1943.

[2] I. S. Gradshteyn, I. M. Ryzhik, D. Zwillinger, V. Moll, Table of integrals, series, and products; 8th ed., Academic Press, Amsterdam, 2015. URL: https://cds.cern.ch/record/1702455. doi:0123849330.

[3] J. Slagle, A heuristic program that solves symbolic integration problems in freshman calculus : symbolic automatic integrator (SAINT), Ph.D. thesis, MIT, 1961.

[4] J. Moses, Symbolic Integration, Technical Report AC-TR-47, MIT, 1967.

[5] N. M. Abbas, Cas integration tests, 2024. https://www.12000.org/my_notes/CAS_integration_tests/index.htm.

[6] A. D. Rich, P. Scheibe, Rubi (Rule-based Integrator), 2024. https://rulebasedintegration.org/.

[7] A. D. Rich, D. J. Jeffrey, A knowledge repository for indefinite integration based on transformation rules, in: Intelligent Computer Mathematics – LNCS 5625, Springer, 2009, pp. 480–485.

[8] A. Rich, P. Scheibe, Rubi – rule-based integration, 2024. https://github.com/RuleBasedIntegration/Rubi/releases/tag/4.17.3.0.

[9] A. C. Hearn, REDUCE: A user-oriented interactive system for algebraic simplification, in: M. Klerer, J. Reinfelds (Eds.), Interactive Systems for Experimental Applied Mathematics, Academic Press, New York, 1968, pp. 79–90.

[10] A. C. Hearn, many contributors, Reduce – a portable general-purpose computer algebra system, 2024. https://sourceforge.net/projects/reduce-algebra.