2

# Julia - Fast symbolic computing?

Student Name: Andrew Monteith

Supervisor Name: Lawrence Mitchell

Submitted as part of the degree of BSc Computer Science to the

Board of Examiners in the Department of Computer Sciences, Durham University

***Abstract —***

**Background** - Symbolic computing is a prevalent area of research in many communities, for example machine learning and scientific computing, that deals with algorithms that manipulate mathematical expressions; applications that perform symbolic computing are called computer algebra systems (CAS). Julia is a recently emerging programming language that has been designed for numerical programming who's suitability for symbolic computing is yet to be fully explored.

**Aims** - The aim of this project is to implement a CAS tailored for tensor algebra and to compare it against a preexisting well used CAS for tensor algebra to establish the suitability of Julia for symbolic computing.

**Method** - We implement our CAS in Julia and compare our solution against the Unified Form Language, a well used CAS with equivalent domain functionality implemented in Python. We then perform a range of benchmarks on common symbolic operations such as differentiation using both synthesized and real world input expressions to compare the runtime performance .

**Results** - Julia is shown to be faster in every benchmark performed with a minimal speedup of 1.5 achieved in all operations benchmarked. We also show Julia's features to aid maintainability particularly for symbolic computing packages.

**Conclusions** - Julia is shown to be a suitable candidate for symbolic computing by outperforming an existing widely-used package and by providing features that aid maintainability.

***Keywords —*** Julia, Domain Specific Languages, Symbolic Computing

## I INTRODUCTION

High level interfaces specialized for a specific domain (domain specific languages) are gaining popularity in scientific computing since they offer high expressiveness and also if designed effectively good performance. Within the domain of finite element analysis, an area concerned with physical simulations, the Unified Form Language (UFL) [Alnæs et al. [2014]] is a popular high level interface implemented in Python that can expressing and symbolically manipulate partial differential equations, UFL is used in various packages like FEniCS Logg et al. [2012]. As the complexity of equations written in UFL increase so does the runtime cost of symbolic operations. In this report we implement a DSL and symbolic operations, guided by UFL, in a newly emerging language called Julia which offers feature not present in Python such as a stronger type system, just in time compilation and metaprogramming. We then discuss whether whether these features make Julia suitable for implementing such symbolic operations and DSLs.

## *A Background Concepts*

### A.1 Domain Specific Languages

A domain specific language (DSL) is a computer language specialised to a particular application domain [Fowler and Parsons [2011]]. DSLs allow for expressing a problem using terminology and concepts tightly coupled to a particular domain independent of any detail on how to solve the problem. For example the language Cascading Style Sheet (CSS) is a DSL used to specify the visual properties and layout of a webpage. Listing 1 shows that the code snippet is able to capture the details of the problem, any label needs to be centered and red, however it includes nothing about how to in practice apply the style such as how to compute the center of any label. Often DSLs are examples of declarative languages because of not specifying precisely how to solve the problem. The key benefit of a DSL is that no underlying technical knowledge is required which can allow domain experts with little technical knowledge to design high quality solutions to problems in that domain.

```
label {
    position: center;
    color:red;
}
```

Listing 1: Example CSS Snippet

DSLs come in a variety of flavours each with their respective benefits and weaknesses that one must consider when creating one. The first major decision one must make is whether to make the DSL embedded or external. The difference is that embedded DSLs (eDSL) are implemented as an application programming interface (API) in a host programming language whereas an external DSL is implemented independent of a host language. For an eDSL one does not have to write a parser since the host language has it's own syntax however one will also be constrained by any characteristics of the host language such as it's type system or runtime performance whereas an external DSL one is free to choose any characteristics of their DSL language, such as syntax, however must then implement that in the compiler/interpreter for ones lanuguage. CSS is an example of an external DSL since there are a variety of interpreters for it such as in different web browsers, whereas UFL is an example of an eDSL since it is implemented as an API in Python. Listing 2 shows an equation and it's UFL equivalent, as one can see whilst it's constrained by the host language the DSL would be understandable by a domain expert with no knowledge of Python. One might choose an eDSL when existing software packages in their target domain are implemented in a common language so that ones eDSL is trivially compatible, whereas external DSLs are good for domains which one meets in many contexts, such as CSS for the range of devices(phones, desktops, tablets) that need to describe layouts where each vendor could design their own web browser.

$$F := \int u \cdot v \, dx$$

Domain Snippet

```
u, v = Function(), Function()
F = dot(u, v)*dx
```

UFL

Listing 2: Embedded DSL Snippet

If one chooses an eDSL then one must decide whether to make it deep or shallow. A Shallow eDSL is where the API eagerly evaluates to the target language, so is just syntactic sugar, whereas a deep eDSL offers an API that constructs an intermediate representation (IR), typically some tree, semantically equivalent to the original DSL fragment. For example consider the domain called symbolic computing, also called computer algebra, which is the study of algorithms that manipulate and analyze mathematical expressions [Cohen [2003]]. A computer algebra system (CAS) is a computer program that performs symbolic operations, a CAS will commonly be integrated with other systems and so will use an eDSL to represent it's expressions. Assume the CAS represented the expression $-8 - \sqrt{x} + (3 + 5)$ using an eDSL snippet `-8-sqrt(x)+(3+5)`. The CAS will want to store and manipulate the expression therefore will want a deep embedding so that it can pass around the IR, a shallow embedding would try to immediately evaluate the `sqrt(x)` term but we may not have a value of `x` yet. A deep embedding allows for the greatest number of optimisations since a shallow embedding only has a local view of the expression so would not spot the $3 + 5$ and $-8$ terms cancel whereas the IR of the deep embedding could be inspected and simplified as such.

When a CAS performs symbolic operations, such as simplifying $3+5$ or cancelling appropriate terms these are often implemented as transformation passes over the tree (IR). A transformation pass is a typically bottom-up traversal of the IR with a function being applied to each node and it's children, an example of a transformation pass can be seen in fig. 1 which shows the transformation pass called common subexpression elimination (CSE) which ensures every subtree is unique. The performance of the symbolic operations is heavily influenced by the performance of the transformation passes which is influenced by the runtime performance of the implementation language. Directly implementing the symbolic operations in Python can lead to insufficient performance as discussed in section II therefore we look at a language which offers a large number of features different to Python, for example being compiled instead of being interpreted, to see if it's more suitable.

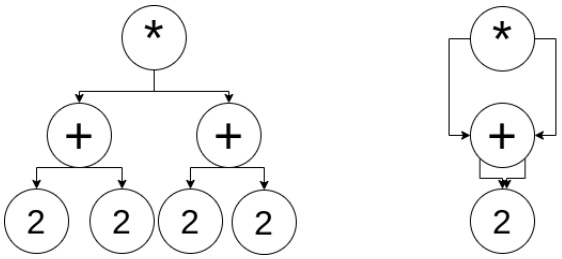


Figure 1: Common Subexpression Elimination

### A.2 Julia Programming Language

Julia [Bezanson et al. [2017]] is a programming language designed for numerical computing. It's syntax is heavily inspired by Python and uses just-in-time compilation (JIT), compiling to machine code at runtime, in order to maximise performance. We introduce the three core features we primarily exploit in our solution: the type system, multiple dispatch and metaprogramming, but this is by no means an exhaustive and full primer on the language.

Julia has two classes of types concrete and abstract. Abstract types (AT) are non-instantiable, only serving as nodes in the type tree and describe sets of related concrete types; those concrete types that are there descendants. Concrete types are structs which behave akin to how structs

behave in C, ie are simple composite types. In Julia one uses abstract types to build up a type hierarchy describing groups of related concrete types, however the hierarchy is flat in that no concrete type can inherit from another concrete type. The key difference between Python and Julia's approach is the object hierarchy one create using inheritance in Python guarantees certain properties about children in the hierarchy, such as existence of member variables or functions whereas in Julia being a subtype of an abstract type guarantees nothing semantically about the type. Listing 3 shows an example type hierarchy, `<:` is the subtyping operator, the Animal type is the root node of the type hierarchy in that it is a subtype of nothing, there are 3 concrete types Cat, Dog and Bird with Cat and Dog being a subtype of Animal and Bird being a subtype of CanFly which is a subtype of Animal.

```
1 abstract type Animal end
2 abstract type CanFly <: Animal end
3
4 struct Cat <: Animal end
5 struct Dog <: Animal end
6 struct Bird <: CanFly end
```

Listing 3: Example Julia Type Tree

Another difference between Julia and Python is that Julia uses multiple dispatch whereas Python uses single dispatch. Dispatch is the process for, at runtime, a function call being matched with an implementation depending on the types of the parameters. Dispatch is a necessary process since not all matchings can be performed at compile time as not all parameter types are fully known. Consider the expression `foo(x, y)` in the context of listing 4 where `x` and `y` are known to have type Animal. One might ask why we would not know `x`'s actual type if we cannot create abstract types, but if we had just read x from an array of Animals then we could not guarantee it's concrete type. Since `Animal` is an abstract type, `x` and `y` must have type `Cat`, `Dog` or `Bird`. Suppose `x` and `y` have true types `Cat` and `Bird` respectively, but at compile time we only know they are an `Animal`. Since Julia supports multiple dispatch it will further inspect the type of every parameter, this means at runtime it will discover `x` is an Animal and `y` is a subtype of `CanFly` therefore will choose the method on line 1, formally we say multiple dispatch means the method definition chosen is polymorphic in every parameter type. Single dispatch is where the method definition chosen is polymorphic only in the first parameter type, this means that Python would not query whether `y` is a subtype of `CanFly` since it is not the first parameter in the method and will choose the method on line 2, but Python would be able to choose appropriately a method definition defined for `x` being a subtype of CanFly.

```
1 foo(x::Animal, y::CanFly) = print("second parameter can fly")
2 foo(x::Animal, y::Animal) = print("nobody can fly")
```

Listing 4: Julia Example Method Definitions

Julia's type system and multiple dispatch allows for better generalization of methods for groups of types than single dispatch since you can place firmer restrictions on the combination of types for which a method applies for. For example single dispatch is unable to accommodate the definition of foo on line 1 of listing 4 as single dispatch can only specialize methods on the first type of the parameter. By defining a method with abstract types you are able to describe a function for a large number of types but having that description decoupled from any details of

the concrete types in the family. Julia's runtime will attempt to minimise the number of dispatch calls required at runtime by resolving method calls at compile time if possible. For example if at compile time it knows `x` and `y` to have types `Animal` and `Bird` respectively then it will specialize that method call to the method on line 1 and not need to perform dispatch at compile time. By minimising the cost of multiple dispatch and using abstract types to describe large collection of types Julia has both expressiveness and performance.

Metaprogramming is the ability to treat code as data. A language that supports metaprogramming will expose it's abstract syntax tree (AST) as a data type for functions to manipulate, the abstract syntax tree is a mutable equivalent of the program source code. Julia exposes macros which are a special function executed at compiletime that manipulate the AST. Listing 5 shows a macro that can gets expanded into some structs. Whilst the example is trivial, it is trivial to see metaprogramming is able to reduce boilerplate and could be used as a useful technique for specifying a DSL.

```
1  macro gen_struct(name)
2    :(struct $name$ end)
3  end
4
5  @gen_struct Struct1
6  @gen_struct Struct2
```

Unexpanded Macro

```
1  struct Struct1 end
2  struct Struct2 end
```

Expended Macro

Listing 5: Julia Metaprogramming

### *B Project Goals*

To investigate the suitability of Julia for symbolic computing we implement a subset of the functionality offered by UFL in Julia. Symbolic computing is used in a variety of domains who's main concern is performance, such as computational simulations, therefore we primarily discuss the runtime of any solution when talking about suitability. The timescale of the project means it is infeasible to recreate all of the functionality offered by UFL so we base our goals around the amount of functionality implemented. We now list our desired goals for functionality ranked from most important to least important for making conclusions and so we will only implement later items if prior items are implemented.

- We aim to implement the tensor algebra functionality from UFL. Performing tensor algebra efficiently from UFL requires very simple symbolic manipulation and few transformation passes. If Julia is unable to do the simpler functionality effectively then it would be unsuitable for implementing symbolic manipulation in DSLs.
- We then aim to implement some simple transformation passes from UFL such as CSE and Automatic Differentiation. Whilst the minimum objectives covered the simplest functionality these objectives would allow a fuller evaluation of the scope of operations a symbolic language would perform albeit on a minimal DSL.
- Finally we will aim to implement multiple transformation passes and vector calculus. This goal serves to expand the scope of operations that can be used to judge the suitability of Ju-

lia and would provide enough functionality to allow effective comparisons of benchmarks from the Python UFL to the Julia UFL.

## II RELATED WORK

Here we present a brief discussion of implementation approaches for DSLs in other languages and the current state of Julia with respect to symbolic manipulation and relevant eDSLs in it's ecosystem.

### *A Implementation Approaches*

SciPy [Meurer et al. [2017]] is a Computer Algebra System written in Python. The entire implementation was originally written in Python but the authors found for programs of $10^4$-$10^6$ symbols the symbolic manipulation performance proved insufficient. The lack of performance was because Python is interpreted meaning the program was not run close enough to the metal. The symbolic manipulation was reimplemented in a C++ library called symengine [Meurer et al. [2020]] which was shown through benchmarks to be quicker than the former Python implementation. One reason the C++ was quicker was because it is a compiled language therefore is executed directly by the CPU and not an interpreter. Secondly Python is a dynamically typed language therefore checks for type errors at runtime, C++ is a statically typed language which means the types must be known and correct at runtime meaning no overhead from checking types and type specific optimisations can be performed such as constant folding. Choosing Python as the host language for the eDSL allowed SciPy to integrate into a large package ecosystem and it has become a key package for mathematical packages being cited over 183 times at the time of writing. However it shows that a dynamically typed language would be insufficient for maximal performance on processing-intense operations.

Haskell [HaskellWiki [2013]] is a functional language known for a rich type system. Haskells type system allows one to embed the domain semantics into the eDSL, that is you are able to capture errors in the domain semantics as type errors in the Haskell eDSL equivalent. For example Thiemann [2001] describes an implementation of an eDSL in Haskell for generating HTML files; all one needs to know about HTML is it is a hierarchical tree that describes a webpage. If the title node contained any non-invalid data it would be invalid as this would contradict the domain semantics, ie a non text title would make no sense for a webpage! Listing 6 shows a snippet of the Haskell eDSL attempting to add an image node into a title node; make creates a new instance of the node. Haskell is able to produce a compile time error for this invalid operation however notice that the error message gives very little information that a domain expert could understand, ie a webpage designer, and would require some Haskell knowledge to interpret. This shows one of the shortcomings of this approach, an eDSL should attempt to minimise the amount of knowledge outside of that domain required to use it which Haskell cannot accommodate for as the error messages require knowledge of it's non-trivial type system. Using Haskell's type system ensures one cover all edge cases of a DSL whoever also generates non-trivial error message. If one were to choose a dynamically typed language one would need to check the domain semantics at runtime which means one could miss edge cases but could also give much friendlier error messages.

```
> add (make TITLE) (make IMG)
ERROR - Unresolved overloading
*** Type : AddTo TITLE IMG => ELT TITLE
*** Expression : add (make TITLE) (make IMG)
```

Listing 6: Haskell Typed HTML eDSL

Language Orientated Programming [Pickering [2010]] is a software development paradigm where a language is given the same status as modules, objects and components. The core concept is to group sets of similar user requirements and to design a eDSL such that the solution is as isomorphic to the requirements as possible. For example if one were designing a game, then one would create a eDSL for the soundscapes, a eDSL for the rendering, and so on, but in all the same host language for compatibility reasons. Benefits of this practice is that maintenance and extention of the software can be done by domain experts who have little technical knowledge. Racket [Felleisen et al. [2018]] is a programming language designed to support Language Orientated Programming. Racket is based on Lisp and so has strong metaprogramming facilities and has a core API that contains many helper libraries for defining custom syntax and semantics for creating languages. To create a language in Racket one must as a minimum write a parser, ie defining the syntax, that reads in your custom language and outputs a valid AST in Racket, for custom semantics one can check whilst parsing, inspect the AST once parsed or not at all. MiniJava Feltey et al. [2016] is a language written in Racket with the syntax and semantics of Java. Listing 7 shows an example Racket program for the MiniJava language, Line 1 instructs the Racket runtime that this file contains mini-java and to load the parser, the file contents are then parsed into valid Racket code and then executed. Racket's runtime means that one is actually creating an External DSL, rather than an Embedded, however one is forced to use Racket as the underlying technology. Racket's ability to easily create External DSLs with a custom syntax is powerful in that domain experts can fully tailor their DSL to their own knowledge such that they're comfortable, however having a piece of software comprised of multiple languages with many syntaxes could determiment the maintability as one would need learn a new syntax each DSL.

```
#lang mini-java

class Main {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

Listing 7: MiniJava DSL

## *B Julia Ecosystem*

In this section we discuss three existing Julia libraries that are relevant to our chosen domain and discuss why we did not base our implementation on them.

### B.1 Existing Packages

DifferentialEquations.jl [Rackauckas and Nie [2017]] is a ecosystem for solving differential equations (DE) in Julia. The ecosystem is designed to be loosely coupled together and such

is formed of many packages each with a unique purpose, such as 1 DE solver per package. The ecosystem uses the eDSL ModellingToolkit.jl noa [2020] as a standardised way to store information across packages. The eDSL makes heavy use of metaprogramming for the API which allows for more natural ways of expressing DEs as it's not confined to the Julia's syntax. Whilst Julia allows for static typing, the eDSL chooses to keep most of it's structures weakly typed, ie omitting the type declarations for variables. This is because there is such a wide range of information that can be stored alongisde a DE that no one data structure could represent it all so we cannot specify a type. Whilst weakly typed structures can be optimised less than strongly typed structures as discussed in section III, this package does not contain any non-trivial symbolic operations and so was not designed with performance in mind.

Rewrite.jl [Grodin [2020a]] is a popular term rewriting library in Julia. It allows one to define a system of symbols and transformation rules available on those symbols. Listing 8 describes a very simple system based around boolean algebra. The theory macro defines all the symbols for which we can define rules on, a Free Theory is a special symbol that can be matched in rules, an AC theory describes that any rule which includes the symbol will be commutative and associative which means the rule on line 6 implicitly generates the rule $F \& x := F$. Using metaprogramming to entirely define the system allows it to be optimised as fully as possible by the JIT however not supporting any runtime API, alongside the lack of features such as conditional matching, restricts the function of the library and such would make it unsuitable to base a our implementation on. Simplify.jl [Grodin [2020b]] is a library belonging to the DifferentialEquations.jl ecosystem used for algebraic simplification. The library provides support for simplification of polynomials containing algebraic terms and simple functions like trigonometry and logarithms.

```
1 @theory! begin
2     F, T => FreeTheory(), FreeTheory()
3     (&) => ACTheory()
4 end
5 @rules! Prop [x, y] begin
6     x & F := F
7 end
8 @rewrite(Prop, x & F) # Evalutes to F
```

Listing 8: Rewrite.jl Example Snippet

### B.2 Feasibility for our solution

The DifferentialEquations.jl ecosystem does not yet support matrices and tensors as first class datatypes. Significant changes would be needed to be made in order for the semantics of tensor algebra to be encoded into the eDSL, such as encoding the shape of a tensor into the datatype to validate whether 2 tensors can be multiplied, and then tensor operations would need be implemented into Rewrite.jl and Simplify.jl accordingly. The total work for this is out of the scope of this project and even if achieved we would only then support tensor algebra, further transformation passes would also need to be implemented to achieve the advanced objectives which once again would be difficult under the timescale available.

# III SOLUTION

Having chosen not to use the existing packages from the Julia ecosystem we now discuss the decisions we made whilst designing and implementing our solution. The decisions made and discussed in this section are being made in order to try and maximise the runtime performance of our solution whilst also trying to match the effectiveness of the Python implementation.

## *A Common performance patterns*

Here we present a brief overview of some common patterns and points recommended by the authors of Julia [Bezanson et al.] to get the best performance from ones program.

Since Julia allows duck typing then a variables type could change during runtime. If a variables type does not change during runtime we call it type stable. Type stability is important for performance as if a variables type changes the JIT cannot apply many optimisations at compile time as it's harder to reason about correctness, since different types have different properties. If a type is known at compiletime then the JIT is also able to reduce the number of dispatch calls at runtime.

When defining a struct it is recommended that as many members as possible have concrete types. If a variable has an abstract type, which describes a family of types, then one cannot deduce the true size of the variable since each type in the family could be a different size which means if the true type is needed a small dispatch cost will be incurred. Therefore these variables must be stored on the heap at runtime which is considered a slow operation compared to if the type, and therefore size, was known by the JIT which could then store the variable on the stack which is quicker to allocate memory. If member variables are stored on the heap this means all the data associated with the struct cannot be guaranteed to be stored in contiguous memory which could lead to performance problems such as thrashing the cache or cache misses.

Julia's type system supports parametric types, akin to generics in Java, where one can have type variables as part of a type. For example listing 9 shows how to define a product node using parametric types; the variables `L` and `R` are type variables you can substitute with any type for example `Product{Int, Int}` represents the type of the Product of two integers. An obvious observation is that if two Product nodes have different types for the type variables they are considered different types overall, for example `Product{Int, Float}` is different to `Product{Int, Complex}`. One must be careful when introducing parametric types as if the number of possible distinct types arising from the parameterisation is large, it will cause the number of types to grow exponentially which causes processes like multiple dispatch and compilation to be slow. This combinatorial type explosion can be seen by the type of the simple expression $(2+2)*(2+2)$ being the distinct type `Product{Sum{Int,Int},Sum{Int}}`, as the expressions become non-trivial it is easy to see, so too will the number of types grow exponentially. Therefore one should only introduce type variables if the number of types possible for them is low and the same information cannot be encoded just as effectively as a non-type variable.

```
1 struct Product{L, R}
2     children::Tuple{L, R}
3 end
```

Listing 9: Julia Parametric Types

## *B Solution Design Choices*

In this section we discuss the finer technical choices we made in order to implement our solution, where appropriate we draw comparisons with the technical choices made by the Python implementation and whether the Python choice is suitable for Julia.

### B.1 Representing the IR

The most important decision to make is how to represent the IR in Julia. The Python UFL uses a tree data structure as the IR where each node is a class that's part of a single inheritance hierarchy. The hierarchy has been designed to optimise groupings of similar domain elements, for example all nodes representing a function are a subclass of the abstract class MathFunction. One would want to keep these groupings as they can be exploited for benefits such as code duplication. To understand the rest of the discussions one should be aware of the three most general node types in UFL: Expr, Operator and Terminal which describe the root node type, non-terminal expressions (functions, products, ...) and terminal expressions (constants, ...) respectively.

Julia does not support object orientated programming so we cannot use inheritance and so resort to using abstract types to describing groups of domain similar elements. A decision decision made by the Python implementation is that any internal node in the inheritance hierarchy are abstract classes, ie not instantiable, which means structure wise they are equivalent to abstract types in Julia. This means there is a direct mapping from the Python inheritance hierarchy to the Julia type hierarchy, as shown in listing 10, where any python class which is an internal node in the inheritance hierarchy becomes an abstract type in the Julia type hierarchy and any leaf nodes become Julia concrete types. We choose to copy this hierarchy as it means we can best exploit the domain groupings.

```python
class Expr: ...

class Operator(Expr): ...
class Terminal(Expr): ...

class MathFunction(Operator): ...
class Sin(MathFunction): ...
```

Python IR

```julia
abstract type Expr end

abstract type Operator <: Expr end
abstract type Terminal <: Expr end

abstract type MathFunction <: Operator end
struct Sin <: MathFunction end
```

Julia IR

Listing 10: IR Equivalence

Since structs represent instantiable nodes in our IR they must store their children, whilst we could use parametric variables to store the type of the children we choose not to for the reasons described in section A. This means we must store our children with the abstract type `Expr`, the detriments of this have been discussed earlier however this is a necessary decision to allow the IR to scale with the number of node types.

### B.2 Describing behaviour for families of types

The Python implementation uses class-based inheritance heavily, every UFL node type (class) is in the same type tree (inheritance hierarchy) which allows one to remove code duplication and

generalise behaviour on families of types of UFL node. For example if you add the property `is_constant_value=false` on the root type then every node then has that property, you can then override that property in the nodes that are constant. Julia does not support the inheritance of structs from structs, here we discuss some techniques to achieve the same behaviour as in Python and the solutions we adopted in our implementation. For our discussions we describe different methods of how to specify the behaviour `is_constant_value` for families of types.

The first method is to use metaprogramming to generate the required code (defining the behaviour) for each struct definition. This technique is not optimal since there would be alot of code duplication and bloating of structs with duplicated logic. For example consider adding the property `is_constant_value=false`, the macro would add this as a member variable for each struct with an appropriate value, since only the minority of types need it defined true you are wasting alot of data on other types defining it as false.

An abstract type describes a family of types, using Multiple Dispatch one can use functions to describe behaviour on abstract types. Listing 11 shows how we can define the constant value property on a family of types. The expression `is_constant_value(x)` would evaluate to true if, and only if, the type of x is a subtype of AbstractCV. This method is effective since there is little code duplication since the behaviour is defined in once place. The main limitation of this solution is no type can subtype two distinct abstract types, therefore this method could not support a node that had two properties as this would require subtyping two distinct abstract types.

```
1 abstract type AbstractCV <: Expr end
2 is_constant_value(x::Expr) = false
3 is_constant_value(x::AbstractCV) = true
```

Listing 11: Julia Shared Behaviour

The final technique I investigated is called Traits. Traits are a way of generalising the second method to multiple properties whilst not forcing any restrictions on the type hierarchy, ie having to inherit from specific abstract types. A trait is formed of 3 parts: the trait types, trait functions and trait dispatch. We now present a brief example of a trait described in Listing 12. The type traits describe the distinct categories a type can be classified as relative to the property, for example lines 1 and 2 describe a type can either be constant or varying. The trait functions give a mapping from types to the corresponding type traits, for example line 4 describes any subtype of `AbstractCV` has the type trait `IsConst`. Finally the trait dispatch describe give the actual behaviour for a trait type, for example line 7 describes the trait type `IsConst` maps to true. Consider the expression `is_constant_value(x)` where x subtypes `AbstractCV`. The where clause on line 10 sets the variable T to the type of x. Multiple dispatch will then match T with line 4, since T represents a subtype of `AbstractCV`, so `trait_dispatch` will return an `IsConst` therefore `trait_dispatch` will then be matched to line 7 and return true. It is easy to see we can generalise this technique to any number of type traits and trait functions. Since trait functions are merely a description of an existing type tree they do not force any constraints on the type hierarchy. When using this technique one should attempt to minimise the number of trait functions used as too many will cause a performance bottleneck since multiple dispatch must check every candidate definition for the closest match. One difficulty I encountered during my implementation is the more complex the type tree, the more complex the trait functions, this complexity could manifest itself as a greater number of trait functions or with a more complex reliance on multi dispatch which made debugging and maintainability more difficult. In our

implementation we use this method for describing behaviour for families of types where the second method isn't applicable as it does not constrain the type hierarchy and so does not hinder the natural mapping from the Python IR to Julia IR as described earlier.

```
struct IsConst end
struct IsVarying end

trait_func(::Type{<:AbstractCV}) = IsConst()
trait_func(::Type{<:AbstractExpr}) = IsVarying()

trait_dispatch(::IsConst, x) = true
trait_dispatch(::IsVarying, x) = false

is_constant_value(x::T) where T = trait_dispatch(trait_func(T), x)
```

Listing 12: Julia Type Traits

### B.3 Macros and Hashing

Metaprogramming can be used to programatically generate Julia code and therefore is a common technique for reducing boilerplate. Since Julia does not allow inheritance we use metaprogramming to generate the boilerplate code for UFL nodes. Using macros to succinctly define types and features for a DSL improves the maintainability by reducing the API surface exposed and encapsulating internal DSL specifics which aren't relevant to the specific type/feature, for example the member variables like children for nodes. Listing 13 shows the skeleton of the Sin function that uses a macro to generate the boilerplate code. The ufl_fields tag describes the properties the ufl_macro should add in this case operands are the children in the UFL tree. We would want to add this property to any non-terminal node however a limitation of macros is they're run before any semantic information is added therefore we cannot query in the macro whether the Sin is a subtype of Terminal so we must specify the property manually. The ufl_tags property is used to pass additional information for the macro, whilst this could be done as part of the method signature it is merely a preference of style to do it this way. The num_of_children is an optional property specifying ahead of time the number of children for the node, the macro respects this by changing the type signature of the tuple in the expanded macro respectively. The default type for the ufl_operands property is a parameterised length tuple, ie the length of the tuple is determined at runtime, by using the macro we can save a small cost by specifying it at compile time if it is known.

```
@ufl_type struct Sin <: MathFunction
    ufl_fields=(operands,)
    ufl_tags=(num_of_children=1,)

    ...
end
```

Unexpanded Macro

```
struct Sin <: MathFunction
    ufl_operands::Tuple{AbstractExpr}

    ...
end
```

Expanded Macro

Listing 13: Metaprogramming Snippets

In order to implement alot of optimisations, such as common subexpression elimination, we need to be able to compare equality on UFL nodes. UFL defines two nodes to be equal if, and

only if, the hash values of the two nodes are equal, the hash value of a UFL node depends on the hash value of all nodes in the subtree, for example for two plus nodes with children 2, 2 and 2,3 respectively require different hash values as they represent different expressions. The Python UFL employs a lazy hashing model meaning the hash is computed when it's needed, in this case it is computed by hashing a list of all the hashes in the subtree which are accumulated via a traversal; to avoid repeated computation the hash is cached. In Julia structs are immutable by default, one can make them mutable however this can limit optimisations and introduce bugs such as accidentally changing a node and then the nodes hash is incorrect, therefore we aim to maintain immutability. This means a lazy hashing method where the hash is lazily computed then set on the node is infeasible since the node is immutable, to overcome this one might think to use a global dictionary that stores the hash of each node but a lookup into that dictionary would need the hash of the node, which is stored in the dictionary, causing a circular dependence. The solution we adopt is to compute the hash when we create the node (eager hashing), this means we may compute the hash of nodes that are never needed however the number of these nodes is likely small due to how ubiquitously hash values are used. Listing 14 shows a simplified UFL Sum node with eager hashing added, where LHS and RHS are UFL nodes (we omit the types to save space). The sig macro is used to describe which variables to include in computing the hash. The ufl_type macro will automatically insert the member variable ufl_hash_code, which stores the hash, line 4 then computes the hash where the 1 is a unique ID for the Sum type and then the second 2 elements are the hash of the LHS and the RHS variables which is then passed to the struct constructor on line 5 initialising ufl_hash_code. Since trees are constructed bottom up a parent will always know the hash values of each of it's children therefore hashing is $O(1)$ during initialisation.

```
1  @ufl_type struct Sum <: Operator
2    ...
3    function Sum(LHS, RHS)
4      new(@sig((LHS, RHS)))
5    end
6  end
```

Unexpanded Macro

```
1  struct Sum <: MathFunction
2    ufl_hash_code::UInt32
3    ...
4    function Sum(LHS, RHS)
5      sum_hash = hash((1, hash(LHS), hash(RHS))
6      new(sum_hash, ...)
7    end
8  end
```

Expanded Macro

Listing 14: Hashing

### B.4 Tree Traversals and Transformation Passes

UFL uses two flavours of tree traversal which are pre-order and post-order; pre-order traversals visit the root then recursively traverse each subtree, post-order traversals recursively traverse each subtree then visit the root. Traversals are the building block for performing transformation passes therefore need to be as performant as possible. Initially we implemented traversals as iterators however Julia's iterators, by design, aren't type stable which meant we could not get optimal performance for reasons described in section section A. Our two remaining options are implementing traversals as either a function or macro. Listing 15 shows a stripped down version of the pre-order traversal implemented with each technique. Since a node does not know the true type of it's children, on line 4 of the functional style there will be a dispatch cost incurred

by calling $f$ if the JIT is unable to inline the call, such as if f is defined differently for different types. The macro style takes the loop AST as an input and can extract the loop body code and directly inline it at compile time which means there is no dispatch cost and so will be as quick or quicker than the functional style. Using a macro will cause some code bloat since the same code is being added at every macro expansion however this is negligible. The same logic can be applied for implementing post-order traversals therefore we use macros for traversals in our implementation.

```
function pot(root, f::Function)
  ...
  while unvisited node v
      f(v)
  end
end
# Usage:
pot(root, x -> println(x))
```

Function Traversal

```
macro pot(loop_ast)
    ...
    while unvisited node $iterator_var$
       $body$
    end
end
# Usage:
@pot for x in root
  println(x)
end
```

Macro Traversal

Listing 15: Julia Traversal Styles

The function `map_expr_dag(root, func)` is used heavily in transformation passes. The function performs a unique post order traversal from the root, unique in that it will skip any node it has already seen, on each node it will call `func` with two arguments: the node it's on and the results of the already processed children. Once a node has been processed we cache the value for later use, we are able to skip nodes because as the tree is immutable if we see a node we have already seen then the processed value will already be in the cache, assuming func is deterministic. The result returned is the processed value of the root node. Listing 16 shows how to implement a transformation pass that counts all the nodes in a tree. Line 2 matches a Terminal node, ie has no children, therefore contributes 1 to the tree total. Line 3 matches nodes that have children therefore contribute 1 plus the total number of nodes in each of the subtrees beneath it, since we are passed the processed value of each child then `children[i]` will be the number of nodes in the subtree of the $i^{th}$ child.

```
function count_nodes(root::AbstractExpr)
    func(::Terminal) = 1
    func(::Operator, children) = 1 + sum(children)
    return map_expr_dag(root, func)
end
```

Listing 16: Example Transformation Pass

Whilst the example transformation reduces the size of the tree, often transformations would increase the size of the tree, for example during differentiation the term $sin(x^2)$ which is 4 nodes becomes $(2*x)*sin(x^2)$ which is 8 nodes. By treating the tree as a directed acyclic graph (DAG), and since nodes are immutable, then we are able to reuse the cached nodes are reduce the rate at which the trees grow in the number of nodes. For example Figure 2 shows hows how treating the differentiation example as a DAG and reusing duplicate nodes can lower the number to 6 rather

than 8. The larger the trees get, the more likely we are to see repeated subtrees therefore the greater saving in size of the tree we can get from this optimisation.

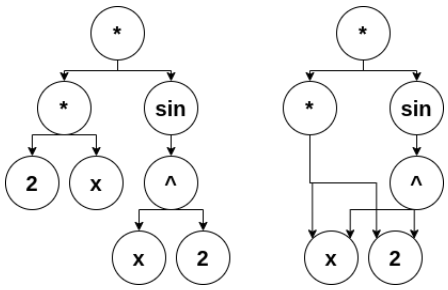


Figure 2: Julia DAG IR

### *C Testing*

Julia has an inbuilt test harness for running unit tests. To check whether a particular operation, or series of, has been performed correctly we perform the same operation in the Python implementation and then check whether to see the two resultant DSL expressions describe the same expression. We use a bottom-up testing approach by first writing unit tests for simple operations like constructing nodes and simple property querying. We then have tests for more complex operations like transformation passes such as whether differentiation works for all the UFL node types implemented. We finally then test whether series of transformations applied to large expressions work correctly. By testing at these three levels we are able to have a large test coverage and check a large number of scenarios for correctness.

## IV RESULTS

In this section we present the results of our benchmarks for the Julia and Python implementations. All benchmarking was performed on a Intel® Core™ i5-4690K Processor. Whilst we achieved all our goals we must be careful to exercise minimal logic implemented in the Python UFL that is not implemented in Julia as to make comparisons fair.

### *A Tree Construction*

Constructing a tree is the most fundamental operation in UFL, in this section we benchmark constructing trees of various sizes. Since an operations performance varies depending on the type of the node it's processing, a randomly generated tree should include a wide variety of node types. However to make results comparable, the process to generate a random tree of $n$ nodes should produces trees with the distribution of node types independent of $n$. To randomly generate a tree $t$ of size $n$ we iteratively add four atoms (listing 17) as leaves on t until t has $n$ nodes. Since each iteration contributes a fixed number of nodes to $n$, one may have to add or remove a few leaves if this number is not a multiple of $n$. By adding the atoms as leaves we produce a tree of expected depth $O(log\ n)$, and by using the same atoms the number of distinct types in any tree is the independent of $n$, in our case thirteen.

```
# u, v are Functions. x represents a scalar variable
atom_1 = tr(grad(u))
atom_2 = dot(u, v)
atom_3 = x^2
atom_4 = ln(x^2)
```

Listing 17: Atoms to build tree

Figure 3 shows the time taken to construct a tree of varying sizes $n$ constructed using the method described above. Note that whilst the scale is logarithmic the average speedup of using Julia was $5.4$ with the execution time growing linearly with the input size which means any transformation pass that expands or creates a new tree, such as differentiation and CSE, it's runtime will be $\Omega(n)$. Since the Julia implementation eagerly hashes the node as the tree is constructed, and hashing in Python has a non-zero cost, then the graph shows that for any $n$ the cost of creating and hashing a tree in Julia is less than Python.

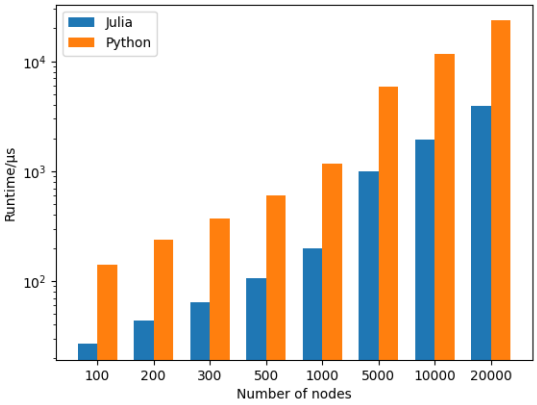


Figure 3: Execution time for constructing a tree

## *B Traversals*

Any transformation pass on the tree will require a traversal on either the entire tree or a subtree therefore it is an important consideration when determining the suitability of Julia for UFL. The JIT will optimise away any traversal with no side effect so we benchmark a traversal that computes the size of the tree as this is the traversal with the smallest side effect possible, simply incrementing a single counter.We generate our trees using the same method as described earlier. Figure 4 shows the runtime for pre-order and post-order traversals on trees of varying size. The average speedup for the pre-order and post-order traversals are $2.1$ and $5.3$ respectively. In both implementations the algorithmic aspects of the traversals, such as the data structures used, are identical therefore the speedup achieved by Julia shows it's type system and runtime model (JIT) compared to Pythons allows for quicker code. For trees where the number of children vary per node, post-order traversals are more complicated to implement than pre-order traversals. The larger increase in speedup for post-order traversals compared to pre-order implies that the speedup benefit received by using Julia over Python will only get better as the complexity and size of your programs grow.

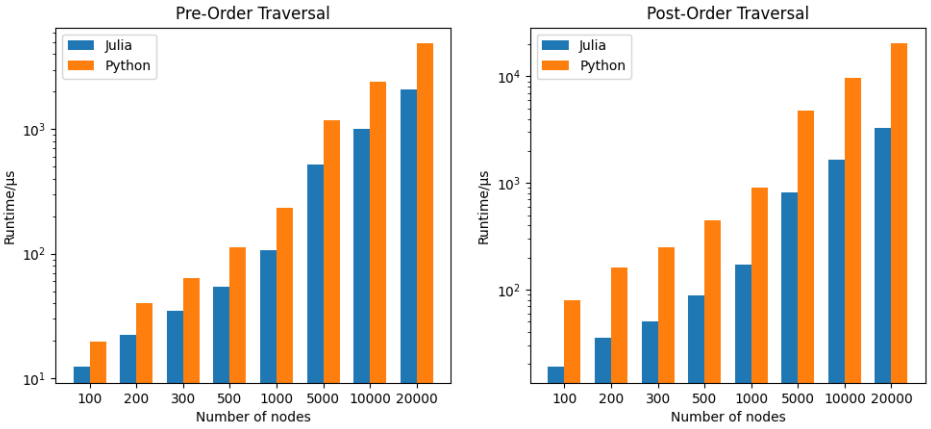


Figure 4: Execution time for counting the nodes in a tree

## C *Differentiation*

Symbolic Differentiation is a fundamental transformation pass for UFL. UFL supports two forms of differentiation classical and gateaux. Our current implementation of gateaux differentiation is minimal therefore we only benchmark the classical derivative. The pass replaces the subtree of any grad node with it's derivative, for example grad($x^2$) is replaced with $2x$. Since the runtime of differentiation is affected by the types of nodes in the tree we choose to define 5 custom instances, table 1, which vary in difficulty to differentiate. Expressions 1 to 3 represent simple algebraic expressions of increasing complexity designed to test the runtime of simple differential rules. Expressions 4 and 5 are designed to exercise the tensor algebra specific differential rules. Table 1 shows the time taken to compute the derivative on each example, once again Julia is quicker in every example. The average speedup for examples 1-3 and 4-5 are $2.15$ and $1.8$ respectively. This shows that the data structures used for handling tensor specific operations are slower to differentiate than the simpler algebraic operators and there could be future effort into optimising the tensor algebraic differentiation further. Examples 2 and 3 have near identical runtimes even though example 3 was constructed to be more complex being a larger tree and having a larger range of types of nodes. The runtime is not dramatically higher because `max_expr_dag` caches previous computed results therefore the pass will not recompute the derivative of $x^i * ln(x)$, however the equality in the runtimes is coincidental.

Table 1: Differentiation pass runtimes

| Id | Expression | Julia/s | Python/s |
|---|---|---|---|
| 1 | $\sum_{i=1}^{10} x^i$ | 0.45 | 1.21 |
| 2 | $\sum_{i=1}^{10} ix^i - ix^{-i}$ | 1.82 | 3.51 |
| 3 | $\sum_{i=1}^{10} x^i \log(x) + \frac{1}{x^i \log(x)}$ | 1.85 | 3.45 |
| 4 | $\sum_{i=1}^{10} x^i u \cdot v + \log(x^{-i})(u \cdot v)$ | 2.96 | 5.05 |
| 5 | $\sum_{i=1}^{10} x^i \det(\nabla u) + \log(x^{-i}) \operatorname{Tr}(\nabla u)$ | 2.98 | 5.74 |

### D Example Form

So far all UFL snippets we've benchmarked are artificially constructed, for this section we benchmark a UFL snippet $f$, describing a hyperelastic material defined in Section 5.4 of [Alnæs et al. [2014]], that was described by the author of UFL as a major motivator during the development of the language. We benchmark mimicking a small portion of a typical pipeline applied to a UFL snippet by applying a series of transformation passes to $f$: algebraic lowering, differentiation, function pullback then differentiation. Algebraic lowering replaces any tensor algebra operators, eg det, trace, dot with a lower level representation that is easier to manipulate and simplify. Function pullback is a transformation pass that wraps a form argument (function, argument, constant, ...) in a reference value node, why this is useful is out of the scope of this section. Table 2 shows a breakdown of the runtime of each transformation pass, with Julia giving an overall speedup of 1.7. As more transformation passes were applied the size of the tree will have increased which is why the second differentiation pass took longer than the first. The Function Pullback pass is a simple substitution pass but the speedup benefit from it was lower compared to the other passes, this is because during the traversal performed in a pass if a node has had it's children changed we need to reconstruct it with it's new children, the Python runtime makes it is very easy to query the appropriate method for reconstruction since the concrete type of the node is always known however in Julia one needs to deduce the concrete type first, since all nodes read from the tree are a common type, the cost of doing so affects the speedup.

Table 2: Transformation Pass Runtimes

| Pass Number | Pass Name | Julia Runtime ($\mu$s) | Python Runtime ($\mu$s) |
|---|---|---|---|
| 1 | Algebraic Lowering | 364 | 680 |
| 2 | Differentiation | 697 | 1300 |
| 3 | Function Pullback | 609 | 913 |
| 4 | Differentiation | 775 | 1250 |

## V EVALUATION

In this section, we evaluate the strengths and limitations of our solution framed in the context of the feasibility of Julia for symbolic computing.

### A Solution Strengths

Our benchmarking showed that Julia was quicker for all functionality we implemented. Any transformation pass not implemented would likely be a combination of traversals, tree construction and pass specific logic. We have shown traversals and tree constructions are substantially quicker in Julia and that pass specific logic is quicker, typically due the difference is runtime environments. This means any transformation pass one would want to implement would likely be quicker in Julia implying it is suitable for symbolic computing relative to our main definition of suitability. Julia's metaprogramming was effective at reducing boilerplate as we were able to add new nodes in roughly the same number of lines as the Python implementation. The type system was able to preserve the domain groupings from UFL and also offered techniques like traits to effectively describe complex behaviours.

### *B Solution Limitations*

The main limitation of our approach is the fact we store our IR as a homogeneous tree. One possible solution to this is to move information that is particularly pertinent to the type from a member variable to a parametric variable, for example in the Julia stdlib the dimension of the array is stored as part of the type rather than a member variable. Another limitations with our solution is that all computations are currently performed at runtime however there are some operations that could be deferred to compiletime which would make the runtime faster.

### *C Approach*

We chose UFL as our symbolic language because UFL is well known to the project supervisor and is implemented in a language the student is confident with. The domain was not fully known to the student however the functionality from UFL we explored was easy to pick up with the students background. Our approach of assessing the suitability of Julia by mirroring operations defined in UFL meant we had a large suite of functionality we could implement and compare against. By adopting an agile approach we were able to quickly change the direction of the project by choosing different areas from symbolic manipulation that exist in UFL depending on what seemed interesting or would particularly exercise and explore a particular feature of Julia. Python does not claim to be performant and since we define performance as our main criteria for suitability, it may weaken Julia's claim to suitability compared to other languages who do claim to be performant like C++.

## VI CONCLUSIONS

In this project we show Julia's to be a good candidate for symbolic computing. We define suitability to be primarily performance alongside maintainability. To assess the suitability we implemented a subset of a well-used symbolic computing package called the Unified Form Language in Julia. To assess the performance we compared common symbolic operations like Differentiation on both synthesized examples and a real life example and found that Julia gave you a speedup of $1.5$ and above compared to same operations in the Python implementation. We also present a brief overview of features in Julia and give brief arguments of how they improve the maintainability of your project.

## References


Martin S. Alnæs, Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Trans. Math. Softw.*, 40(2):9:1–9:37, 2014. ISSN 0098-3500. doi: 10.1145/2566630.

Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012. ISBN 978-3-642-23098-1. doi: 10.1007/978-3-642-23099-8.

Martin Fowler and Rebecca Parsons. *Domain-specific languages*. Addison-Wesley, Upper Saddle River, NJ, 2011. ISBN 9780321712943.

Joel S. Cohen. *Computer alegebra and symbolic computation: mathematical methods*. AK Peters, Natick, Mass, 2003. ISBN 9781568811598.

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, 59(1):65–98, January 2017. ISSN 0036-1445, 1095-7200. doi: 10.1137/141000671.

Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, and Sergey B. Kirpichev. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017. ISSN 2376-5992. doi: 10.7717/peerj-cs.103.

Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. symengine/symengine, March 2020. URL `https://github.com/symengine/symengine`. original-date: 2008-08-13T18:40:14Z.

HaskellWiki. Haskell — haskellwiki,, 2013. URL `https://wiki.haskell.org/index.php?title=Haskell&oldid=56799`. [Online; accessed 27-March-2020].

Peter Thiemann. A typed representation for html and xml documents in haskell. *Journal of Functional Programming*, 12, 03 2001. doi: 10.1017/S0956796802004392.

Robert Pickering. *Language-Oriented Programming*, pages 327–349. Apress, Berkeley, CA, 2010. ISBN 978-1-4302-2390-0. doi: 10.1007/978-1-4302-2390-0_12.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. A programmable programming language. *Communications of the ACM*, 61(3):62–71, February 2018. ISSN 0001-0782, 1557-7317. doi: 10.1145/3127323.

D Feltey, P. S Florence, T Knutson, V St-Amour, R Culpepper, M Flatt, B. R Findler, and M Felleisen. Languages the racket way. 2016. URL `https://users.cs.northwestern.edu/~stamourv/papers/languages-the-racket-way.pdf`.

Chris Rackauckas and Qing Nie. Differentialequations.jl – a performant and feature-rich ecosystem for solving differential equations in julia. *Journal of Open Research Software*, 5, 05 2017. doi: 10.5334/jors.151.

SciML/ModelingToolkit.jl, March 2020. URL `https://github.com/SciML/ModelingToolkit.jl`. original-date: 2018-02-27T01:36:26Z.

Harrison Grodin. HarrisonGrodin/Rewrite.jl, March 2020a. URL `https://github.com/HarrisonGrodin/Rewrite.jl`. original-date: 2019-01-05T03:25:37Z.

Harrison Grodin. HarrisonGrodin/Simplify.jl, February 2020b. URL `https://github.com/HarrisonGrodin/Simplify.jl`. original-date: 2018-04-28T20:08:29Z.

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Performance Tips · The Julia Language. URL `https://docs.julialang.org/en/v1/manual/performance-tips/`.