

# Symbolic-numeric programming in scientific computing

by

Shashi Gowda

Submitted to the Department of Mathematics and  
The Center for Computational Science and Engineering  
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024

© Shashi Gowda, 2024. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Author.....  
Shashi Gowda  
Department of Mathematics and  
The Center for Computational Science and Engineering  
May 10 2024

Certified by.....  
Alan Edelman  
Professor, Department of Mathematics  
Thesis Supervisor

Accepted by.....  
1) Jonathan Kelner  
Applied Mathematics Chair, Department of Mathematics  
2) Youssef Marzouk  
Co-Director, Center for Computational Science & Engineering



# Symbolic-numeric programming in scientific computing

by

Shashi Gowda

Submitted to the Department of Mathematics and  
The Center for Computational Science and Engineering  
on May 10 2024, in partial fulfillment of the  
requirements for the degree of  
Doctor of Philosophy

## Abstract

Scientific programming languages should pursue two goals: closeness to mathematical notation, and the ability to express efficient numerical algorithms. To meet these goals simultaneously, languages use imperative surface syntaxes that mimic mathematical notation. However, mimicking does not make them the same—mathematics is declarative, pliable, and caters to exploratory human nature; but algorithms are imperative and must cater to machines. Hence, there is a fundamental limit to this approach and we leave the expressive power of the symbolic representation on the table.

In this thesis, we ask the question: How can symbolic and numerical modes of computing co-exist, one informing the other? As an answer, we develop a symbolic-numeric system that can trace through numerical code to produce symbolic expressions, and turn symbolic expressions back into high-quality numerical code at staged compilation time. This allows the scientific user to generate code with the full power of algebraic manipulation and to treat numerical code as the symbolic artifact it is. We identified siloing of symbolic software into 3 categories which currently each reproduce similar forms of symbolic capabilities, but cannot share code between each other. Our work demonstrates that this siloing is not essential and an ecosystem of symbolic-numeric libraries can thrive in symbiosis.

Our system is adaptable to any domain: users can define 1) Symbolic variables of any type 2) the set of primitive (symbolically indivisible) functions in the domain, 3) the propagation of partial information, and 4) pattern-based rewrites and simplification rules. There is a tendency in scientific computing to create a “compiler for every problem” starting from scratch every time. Every such effort erects its own towers of symbolic and numerical capabilities. A system like ours eliminates this redundancy and lets every scientific user be a “compiler designer” without any prior knowledge of compiler development.

Thesis Supervisor: Alan Edelman

Title: Professor, Department of Mathematics



## Acknowledgments

I thank my advisor Alan Edelman for his encouragement over the course of the past 10 years. I kept an Alan in my head while writing this thesis and it was extremely helpful. Thanks to Jeff Bezanson for Julia and his excellent thesis on it which I derived copious inspiration from. I hope this thesis is a furthering of his vision (albeit in a direction that might scare a Julia compiler developer). I thank Gerry Sussman for his teachings, tea, and discussions which inspired this work. The ideas in this thesis were already present in his Scmutils system for the most part. His love for the craft of computing is infectious. I thank Yingbo Ma for the fun pair programming sessions and his contributions to Symbolics.

I thank my committee members Steven Johnson and John Urchel for their feedback and support.

Thanks go to my labmates in the Julia lab for their love, feedback, questions and problems. Especially, Torkel Loman, Flemming Holtorf, Frank Schaefer, Vaibhav Dixit, and Avik Pal. I thank Alexander Demin (GSoC 2021) and Xinghui Hu (UROOP 2023) for their excellent contributions to Symbolics.

I thank Chris Rackauckas for being the first customer of the code I wrote while procrastinating studying for my quals, which lead to this project, and what a customer to have! Some code that became Symbolics was first written by him and Yingbo Ma. It feels good to know that my work is used by hundreds of downstream projects, thanks to Chris's resourcefulness and leadership in the Julia SciML ecosystem. The example in 1.6.1 is by Chris. The ModelingToolkit and Catalyst sections of the Chapter 5 is on work done with the SciML team, especially Yingbo Ma, Torkel Loman, Chris, and Samuel Isaacson. The section on Convex programming is work done by Vaibhav Dixit. I thank Alex Jones for helping me with array-related examples. I thank David Sanders for his feedback and examples related to ReversePropagation.

I thank Alessandro Cheli for working on Metatheory.jl, and him and Willow Ahrens for discussions about the expression interface. I thank Willow for teaching me about Finch.

I thank Donna Vatnick, Sarah Greer, and Danny Sharp for their feedback on my writing.

I thank Viral Shah for putting me on as a full time Julia developer back in 2014, and his continued friendship and mentorship.

I thank my dear parents Sri Krishnegowda and Smt. Asharani, and my sister Spoorthi for their love, and their faith and pride in me and the things I do, despite always being scared by the things I do. I thank Acharya Virag Tripathi for teaching me love and affection for all things.



*to Śrī Rāma, the destroyer of doubt.*





# Contents

<b>1</b>	<b>Introduction</b>	<b>13</b>
1.1	Definitions . . . . .	16
1.2	A brief history of numerical and symbolic modes of computing . . . . .	17
1.3	Symbolic-numeric programming . . . . .	18
1.4	A Two-language problem in symbolic-numeric programming . . . . .	19
1.5	Quoting code by symbolic evaluation . . . . .	20
1.6	Vignettes of symbolic-numeric programming . . . . .	21
1.6.1	Eliminate memory overhead by symbolic reassembly . . . . .	21
1.6.2	Jacobian sparsity . . . . .	24
1.6.3	Pre-allocation . . . . .	25
1.6.4	Easy domain-specific optimization . . . . .	27
1.7	Contribution by chapter . . . . .	29
1.8	List of publications . . . . .	29
<b>2</b>	<b>Symbolic-numeric status quo</b>	<b>31</b>
2.1	The unfortunate siloing of symbolic software . . . . .	31
2.2	Symptoms of disharmony . . . . .	32
2.3	DSLs – Modelica example . . . . .	34
2.4	Optimizations and low-level compilers . . . . .	36

<b>3</b>	<b>Synthesis of a symbolic-numeric system</b>	<b>39</b>
3.1	Nature of scientific computing . . . . .	40
3.2	Contribution of Julia . . . . .	41
3.2.1	Dynamic multiple-dispatch . . . . .	41
3.2.2	Type inference, code selection and code specialization . . . . .	45
3.2.3	Macro programming . . . . .	47
3.2.4	Staged compilation . . . . .	48
3.3	Symbolic-numeric subsystem . . . . .	49
3.3.1	Expressions: A term interface . . . . .	49
3.3.2	Partial information . . . . .	54
3.3.3	Building on the term interface . . . . .	55
<b>4</b>	<b>The Symbolics.jl system</b>	<b>57</b>
4.1	Expression types and canonicalization . . . . .	57
4.1.1	Numeric expressions . . . . .	57
4.1.2	Implementation of the term interface . . . . .	59
4.1.3	Polynomial representation . . . . .	59
4.1.4	Default implicit assumptions, and how to avoid them . . . . .	61
4.2	Term rewriting . . . . .	62
4.2.1	Rule-based rewriting . . . . .	63
4.2.2	Associative-Commutative Rules . . . . .	64
4.3	Rewriter combinators . . . . .	65
4.4	Code generation . . . . .	67
4.4.1	<code>build_function</code> . . . . .	68
4.4.2	Staged Compilation . . . . .	68
4.5	Symbolic arrays . . . . .	68
4.5.1	ArrayMaker: nesting arrays . . . . .	71
4.6	Primitive registration . . . . .	73
4.7	Simplification . . . . .	73

4.8	Algebra and calculus functions . . . . .	75
4.9	Contributions . . . . .	77
4.10	Comparisons . . . . .	78
4.10.1	Modularity and extensibility . . . . .	82
4.10.2	Symbolic tracing of numerical code . . . . .	84
4.10.3	Numerical code generation . . . . .	85
4.10.4	Function registration . . . . .	86
4.10.5	Notes on JAX . . . . .	87
4.11	Limitations of Symbolics . . . . .	88
<b>5</b>	<b>Case studies</b>	<b>91</b>
5.1	SciML: Scientific Machine Learning . . . . .	91
5.1.1	Numerical building blocks . . . . .	92
5.1.2	Symbolic-numeric modeling language . . . . .	94
5.1.3	Moving the methodology forward in PDEs . . . . .	99
5.2	Convex optimization . . . . .	100
5.2.1	Symbolics vs. multiple-dispatch for implementation . . . . .	101
5.3	Catalyst . . . . .	103
5.3.1	Symbolic-numeric research advantage . . . . .	104
5.4	ReversePropagation.jl . . . . .	105
<b>6</b>	<b>Conclusion</b>	<b>109</b>
6.1	Advancing beyond the expressiveness of Julia . . . . .	109
	<b>Bibliography</b>	<b>111</b>



# Chapter 1

## Introduction

Every piece of numerical code was once a symbolic expression. For instance, the code for the Taylor approximation of  $e^x$  near 0 by a polynomial of degree 4 is a straightforward transcription of the polynomial:

```
function expl(x)
  1 + x + x^2 / 2 + x^3 / 6 + x^4 / 24
end
```

The approximation for different functions and of different degrees can be derived symbolically on a black board or, conveniently, on a computer<sup>1</sup>. However, turning symbolic results into numerical code is either not supported by symbolic computing environments, is inefficient, or is limited to and by a lower-level target language. For example, Mathematica [46], the poster child of symbolic computing, is great for symbolic manipulation, but cannot itself be compiled to machine code. The `CCodeGenerate` [47] function compiles symbolic expressions to C code, however, in an interactive application, one may not know which functions need to be generated beforehand, and need to generate them on a case-by-case basis. This is a particularly important problem in scientific computing where interactive computing is favored by users. The compiled code is limited by the constraints of the C language, such

---

<sup>1</sup>This example can be symbolically computed in Mathematica: `Sum[x^n/Factorial[n], {n, Range[0, 4]}`

as machine-precision numbers, whereas some computation may require arbitrary precision. Further, it can only call library code that is already somehow accessible from C.

The ability to analyze and transform numerical code as symbolic expressions is powerful as we will see in Chapter 5. We continue our example to illustrate this ability below. Let's say we have the Taylor approximation of a different function, say,  $e^{2x}$ :

```
function exp2(x)
    1 + 2x + 2x^2 + 4x^3 / 3 + 2x^4 / 3
end
```

Now,  $e^x + e^{2x}$  can be approximated by `exp1(x) + exp2(x)`<sup>2</sup> but, this is not efficient—the computation can be achieved with one polynomial evaluation whereas here it takes two, one each in `exp1` and `exp2`. The computational cost is at least double the necessary cost. A smart user may know that they just need to add the polynomials in the bodies of `exp1` and `exp2` to get a single polynomial that computes the same value but they don't have any means of adding the insides of a function programmatically. If these programs contain polynomial expressions, why can't we add the polynomial expressions?

Here is where we can use symbolic computation to overcome this limitation: In a sufficiently generic language, a symbolic unknown `y` can be passed into the functions in place of a number. If operations on unknowns produced symbolic expressions, the resulting two expressions will be polynomials. The polynomials can then be added and function can be generated automatically from that result:

```
@variables y # declare y to be a symbolic variable
# run the functions with symbols:
new_polynomial = exp1(y) + exp2(y)
# Result: 2 + 3y + (5y^2/2) + (3y^3/2) + (17y^4/24)
```

---

<sup>2</sup>This is also equal to `exp1(x) + exp1(2x)`

```
# make a function with new_polynomial as the body and y as the argument  
f = makefunction(new_polynomial, y)
```

Such symbolic treatment of numerical code is currently, at best, hidden away in compiler pipelines and not accessible to the average scientific user. Floating point optimizations [62] and automatic differentiation (AD) [39] are examples of symbolic transformations implemented as compiler passes. At the compiler level, we throw away the mathematical domain of variables and only deal with them as machine-precision numbers conforming to a machine-level standard. We discuss this with an example in Section 2.4. Designing compiler passes is usually not part of the expertise of a scientific programmer, and they are forced to live with or work around such limitations.

This thesis demonstrates that a sufficiently nice numerical language can be extended with a symbolic subsystem that can open up a “symbolic-numeric” mode of computing in which users can switch between the symbolic representation and machine executable code. This gives scientific users power that would previously only be available to language developers and compiler engineers. This power can be compounded by a general purpose computer algebra system. With rule-based term rewriting, users can optimize and transform programs as easily as they can symbolic expressions. This becomes a higher-level platform for building domain-specific languages than previously available. We present our implementation, `Symbolics.jl`, on top of the Julia [5] language.

Expressive power with performance is the name of the scientific computing game. The system we present here extends the expressive power of multi-methods where specialization occurs on the types of input, with a pattern-based rewriting and predicate-based dispatch. This leads to a system where the user can choose to overspecialize on input values and partial information when they know compiling a special case is beneficial. This thesis is a glimpse into what we believe is the future evolution of scientific computing languages.

## 1.1 Definitions

### **Symbolic computing**

Computing with named unknowns of different types such as real numbers, vectors and matrices. We use these terms interchangeably: symbolic computing, symbolic programming, computer algebra, symbolic manipulation. This is in line with the common use of these phrases to describe systems like Mathematica, SymPy [63], Maxima [40], etc.

### **Symbolic tracing, symbolic trace**

Executing a piece of code, originally intended to be run on numbers or number-like objects, with symbols to obtain a symbolic result instead of numerical result. The symbolic result is called a symbolic trace. Note that the symbolic trace can be an array of symbolic expressions, if the function being traced returns an array. In the example above,  $2 + 3y + (5y^2/2) + (3y^3/2) + (17y^4/24)$  is the symbolic trace of `exp1(y) + exp2(y)`.

### **Numerical computing and numerical language**

Computing with numbers and number-like objects such as vectors and matrices is numerical computing. We include both low-level numerical libraries like BLAS [6], TACO [54], FFTW [31] are implemented in low-level code, and high-level level productivity systems like MATLAB, NumPy and Julia in our definition. However we do not consider Python and MATLAB numerical languages. In our definition, numerical languages are those in which one can implement efficient low-level routines first and, optionally, also provide a high-level API. Examples are C, FORTRAN and Julia.

### **Symbolic expression, terms, algebraic expression, symbolic representation**

We use these terms interchangeably. Polynomials, trigonometric terms (even though we say algebraic expression), array expressions, are all included in our usage.



## 1.2 A brief history of numerical and symbolic modes of computing

In the early days of computing, numerical computing was the only kind of programming, and compilers were considered a luxury. FORTRAN [53] (short for Formula translation) was the first numerical computing language, and is still a staple in some fields of science. In recent years, emphasis has been given to productivity and interactive numerical computing by systems such as MATLAB [45], R [68], Mathematica [46], SymPy [63], Octave [24], etc. Recently the Julia language [5] has brought together the requirements of performance and productivity into one platform through elegant language design.

In 1960 the LISP (short for LISt Processor) system of McCarthy [61] was introduced for the IBM 704 computer. The LISP system would be used to build a “common sense” answering program that could take instructions in declarative or imperative form. This was perhaps the first practical form of symbolic computing.

Schoonschip was one of the first computer algebra systems, developed in 1963 by Martinus J. G. Veltman, for use in particle physics [86]. It was instrumental in his work leading to the Nobel prize in 1999 [88]. Symbolic computing (referred to also as symbolic programming, computer algebra, symbolic manipulation) has taken a few different forms: the first category, one can say, caters to the mathematical skills of the freshman science and engineering student – Mathematica, Sage [80], SymEngine [81] and SymPy [63] are in this camp; the second caters to abstract algebraists – GAP - Groups, Algorithms, Programming [33], Magma [9], Nemo [29] are in this category. These software are characterized by efficient implementations of limited kinds of algebraic objects: groups, fields and polynomials, etc.; a third category would be of packages such as Modelica [26], Simulink [23] which are domain specific languages for differential symbolic modeling to generate numerical computations. Symbolic languages are either implemented in Lisp, or in C/C++ (more in Table 4.6).

Numerical computing focuses on performance, while symbolic computing focuses on computer algebra, pattern matching and term rewriting.

### 1.3 Symbolic-numeric programming

Symbolic programming and numerical programming have been used in tandem for a long time in ad-hoc ways. For example, modeling and simulation packages need to turn symbolic equations into numerical functions given to numerical solvers. In this field symbolic algorithms are used to make problems feasible to solve, as well as to optimize numerical computation. Numerical software such as fast Fourier transforms [12] and numerical integrators require highly optimized kernels to handle specific kinds of input sizes and types. Symbolic programming is used to generate these variations.

The flip side of generating code from symbolic expressions is being able to symbolically analyze and transform numerical code. A numerical integrator needs to compute the jacobian of a user-provided function, automatic differentiation (AD), an essentially symbolic transformation, is performed for this purpose. Properties such as sparsity of the jacobian, hessian, or convexity which help guide program selection and optimization, and can be symbolically inferred by rewrite rules. AD is applied in machine learning where code denotes array operations rather than elementwise operations. A pattern-matching rule-based rewriter system would serve someone building a compiler for numerical purposes, for example the Finch sparse tensor compiler [2] that supports fast loop nests on sparse tensors with various storage datastructures.

FORMAC (FORMula MANipulation Compiler) [7] project started in 1962 at IBM was the first symbolic-numeric language. We reproduce an expert enumerating its purpose as presented in [73]:

- “...To provide a system which has basic FORTRAN capabilities and also the ability to do symbolic mathematics...”
- “...To provide a system which can be used for both production runs of problems involving both symbolic and numerical mathematics...”
- “...To provide a system which can be useful to application areas which have previously

used numerical approximations because the analytic methods were too time consuming and prone to errors...”

The FORMAC project shows that the intention to create a symbolic-numeric system has been around since the beginning of the genre of symbolic computing. We also see, in the retrospective [73], the difficulty of communicating the need for such a system to funding sources within IBM at the time which was a major road-block to its prevalence. We hope this thesis provides some compelling reasons for the necessity of such systems in the future as well as the technical know-how to construct it in a modern way.

## 1.4 A Two-language problem in symbolic-numeric programming

A two-language problem exists in scientific computing [5]: users write exploratory code in a high level productivity focused language, but the libraries they invoke from this code are actually written in a lower-level performance-focused language like C, or Fortran. Implicit is the assumption that users of the high-level language don't need to look at the lower-level code, and it's to be written by a different kind of expert.

Symbolic manipulation software actually exacerbate this split. Most symbolic softwares have their own language, and cannot describe efficient low-level numerical code, they do focus less on wrapping numerical libraries in comparison to a numerically focused dynamic language like, say, MATLAB would. The code generation from Symbolic expressions in these languages takes two directions which both have limitations:

- They compile to a low-level language such as C/C++/Fortran. Target programs must become limited to types and functions supported by the lower-level language. (e.g. Mathematica, Maple, etc. take this route – See Table 4.6 for a larger comparison).
- They compile to high-level functions wrapped in the high-level language. This limits the freedom to use lower-level constructs such as efficient loops when necessary, and

becomes essentially as fast as the host language. Sometimes, these languages also do not faithfully map from a symbolic form to a numerical form. See Section 4.10.2 for an example of this. MATLAB’s Symbolic Math Toolkit [78], SymPy, etc take this route.

The Julia language showed that this split between productivity and performance languages is not some law of nature, but it is accidental. With the right use of multiple-dispatch generic functions as an abstraction for code selection and code specialization at the same time, it achieves a single language that is dynamic, yet due to dynamic types and data flow type inference, can be compiled to efficient machine code just-in-time. Chapter 3 describes Julia’s contribution in detail.

A symbolic subsystem written with multiple-dispatch in a language like Julia can represent symbolic expressions which faithfully map to a subset of the language, and hence to meaningful numerical code. It can contain non-symbolic values like big integers, matrices, even strings, custom data types, calls to any library functions, and the generated code can simply use the same values rather than the ones obtained by their transcription via printing a file in a lower level language. Our system, Symbolics.jl described in Chapter 4, achieves this ideal and makes symbolic-numeric programming effective in interesting ways as demonstrated by the case studies in Chapter 5.

## 1.5 Quoting code by symbolic evaluation

A symbolic-numeric system needs to be able to represent code as symbolic expressions which have a parallel mathematical meaning. Ideally, these would be the expressions generated when a piece of code is evaluated with symbols. There are many ways one can represent code as such expressions:

1. **Abstract syntax trees (ASTs) as expressions:** To treat these expressions as values, one has to define arithmetic operators on expression trees. However this means the behavior of operators is fixed, and cannot vary based on the type of the expressions. For example, expressions representing numbers may commute, but expressions

representing matrices are not meant to. ASTs can also contain expressions that are code with control flow and side effects, now the operators become undefined. This is the method used for the most part in `scmutils` [34].

2. **Symbols and expressions as objects with types:** Here we define symbols and assign a “symbolic type”, and then define how operators on symbols and expressions of specific symbolic types interact with each other. This eliminates the problems caused by the trees lacking types as in the previous kind.
3. **A combination of normalized syntax tree, produced by partial evaluation:** basic blocks (a sequence of statements, with assignments), branches, loops, possibly in a normalized form like single-static-assignment (SSA). Coupled with the ability to use symbols within them, this representation has more power than the previous two.

Although the third kind is the most powerful, in the rest of this thesis we deal with the second kind. This was the most useful and intuitive representation in real world problems, and we leave the exploration of the third kind to future work.

Metaprogramming is the powerful idea of programmatically generating programs. It is facilitated by a natural representation of code as symbolic expression. It has the obvious benefit of relieving one of the tedium of manually writing and checking (hopefully) larger amount of code. It can make otherwise intractable software development tractable. One can think of it as an order increase in productivity.

## 1.6 Vignettes of symbolic-numeric programming

This section shows some simple illustrative examples of symbolic-numeric programming.

### 1.6.1 Eliminate memory overhead by symbolic reassembly

Following is a PDE discretization code. This code is a typical function that may be received by a numerical integrator. Although the code is easy to read, it’s not exactly efficient since

it allocates a bunch of intermediate arrays and invites garbage collector overhead.

```
using Symbolics, LinearAlgebra
function f(u, p, t)
    # all lines below lead to one allocation each
    A = u[:, :, 1]
    B = u[:, :, 2]
    C = u[:, :, 3]
    MyA = My*A
    AMx = A*Mx
    DA = @. _DD*(MyA + AMx)
    dA = @. DA +  $\alpha_1$  -  $\beta_1$ *A -  $r_1$ *A*B +  $r_2$ *C
    dB = @.  $\alpha_2$  -  $\beta_2$ *B -  $r_1$ *A*B +  $r_2$ *C
    dC = @.  $\alpha_3$  -  $\beta_3$ *C +  $r_1$ *A*B -  $r_2$ *C
    cat(dA, dB, dC, dims=3)
end
```

Here  $\alpha_i$ ,  $\beta_i$ , and  $r_i$  are scalar parameters,  $My$  and  $Mx$  are tridiagonal matrices of weights.

To optimize this code, we run this function with symbolic “placeholder” input. From the result of this symbolic run (we call this a symbolic trace), we generate a new function which is now in-place and modifies an existing block of memory, and completely unrolls the original function. By in-place, we mean it takes a pre-allocated array as the first argument and writes the result to it. The performance is measured by the ‘@b’ macro from the Chairmarks julia library.

```
@variables u[1:4,1:4,1:3]

f1! = makefunction(f, u, inplace=true) # make it in-place

U = rand(size(u)...);
```

```
out = zero(U)
```

```
julia> @b($f($U)) # benchmark `f` as-is
```

```
2.400 μs (41 allocs: 3.094 KiB)
```

```
julia> @b($f!($out, $U))] # benchmark optimized in-place f!
```

```
25.405 ns
```

```
julia> out ≈ f(U)
```

```
true
```

We see that the original code resulted in 41 allocations, and took 2.4 microseconds, but the symbolically reassembled function took only 25 nanoseconds and allocated nothing. This is a 94 times improvement in performance of this code which would be repeatedly run in the critical loop of a PDE solver.

There are certain further niceties here:

- this optimization generalizes to any size of input. For every size of the input, we get a specialized function.
- one may argue that using static arrays, i.e. arrays of static size that are backed by stack-allocated tuples can give you the same performance improvement without sacrificing performance:

```
julia> sU = SArray{Tuple{4,4,3}}(U);
```

```
julia> @b f($sU)
```

```
1.156 μs (23 allocs: 44.000 KiB)
```

in practice it does not work. Presumably, compiling such a series of expressions to work with static arrays is quite expensive

## 1.6.2 Jacobian sparsity

In automatic differentiation, sparsity of a Jacobian can be used to speed up the computation of the Jacobian itself.

Sparsity is nothing but the information on which output depends on which inputs. It's easy to conceive such an analysis: you simply take the symbolic trace, and see which inputs appear in each output of the trace.

```
brusselator_f(x, y, t) = ifelse((((x-0.3)^2 + (y-0.6)^2) <= 0.1^2) &&
                                (t >= 1.1), 5., 0.)

limit(a, N) = a == N+1 ? 1 : a == 0 ? N : a
function brusselator_2d_loop(du, u, p, t)
    A, B,  $\alpha$ , xyd, dx, N = p;  $\alpha$  =  $\alpha$ /dx^2
    @inbounds for I in CartesianIndices((N, N))
        i, j = Tuple(I)
        x, y = xyd[I[1]], xyd[I[2]]
        ip1, im1 = limit(i+1, N), limit(i-1, N)
        jp1, jm1 = limit(j+1, N), limit(j-1, N)
        du[i,j,1] =  $\alpha$ *(u[im1,j,1] + u[ip1,j,1] + u[i,jp1,1] +
                        u[i,jm1,1] - 4u[i,j,1]) + B + u[i,j,1]^2*u[i,j,2] -
                        (A + 1)*u[i,j,1] + brusselator_f(x, y, t)
        du[i,j,2] =  $\alpha$ *(u[im1,j,2] + u[ip1,j,2] + u[i,jp1,2] +
                        u[i,jm1,2] - 4u[i,j,2]) + A*u[i,j,1] - u[i,j,1]^2*u[i,j,2]
    end
end
```

Snippet 1.1: 2D brusselator loop

This functionality is already present in the Symbolics software in the form of the `jacobian_sparsity` function. Figure 1-1 shows the sparsity pattern for one instantiation of this function.



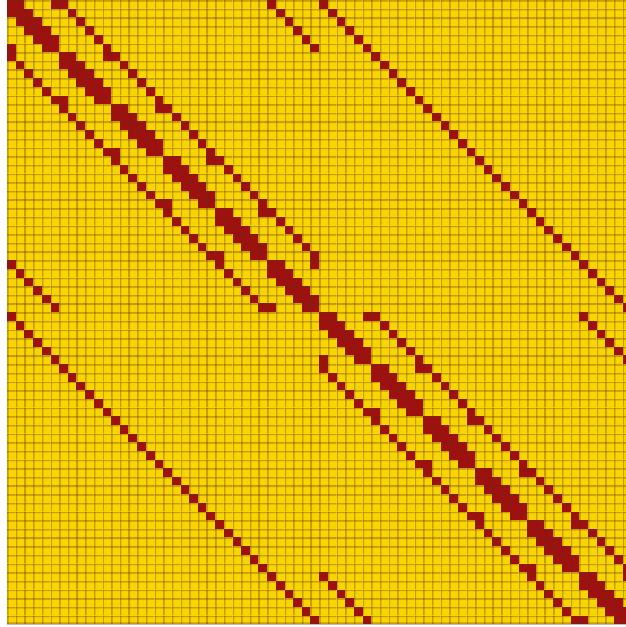


Figure 1-1: Sparsity pattern of the Jacobian of the Brusselator code in Listing ?? with input and output tensors of size  $6 \times 6 \times 2 = 72$ .

### 1.6.3 Pre-allocation

The commutator is a common operator in scientific computing:  $[B, H] = BH - HB$ . Its computation requires 3 allocations,  $BH$ ,  $HB$  and  $BH-HB$ . Now if you're applying this operator to thousands of  $B$  and  $H$  values of the same shape, you're going to be allocating 3 times as many intermediate arrays.

A symbolic transformation can turn the following code:

```
function commutator(B, H)
    B*H-H*B
end
```

into its equivalent, but in-place version:

```
function commutator!(X, Y, Z, B, H)
    mul!(X, B, H)
```

```
mul!(Y, H, B)
Z .= X .- Y
```

**end**

Given the sizes of **B** and **H**, it's possible to symbolically call `commutator` with *symbolic arrays* **B** and **H** of the same size. This results in the the symbolic expression  $\mathbf{B}*\mathbf{H}-\mathbf{H}*\mathbf{B}$ .

We can turn this expression symbolically into a completely in-place version by using the following algorithm:

- initialize a dictionary `inters` to store intermediate arrays
- walk the expression, for every call subexpression `f(args...)` do this:
  - allocate an intermediate array `tmp` of the inferred output size for that subexpression
  - generate a symbol `sym`, set `inters[sym] = tmp`
  - rewrite `f(args...)` as `inplace(f)(sym, args...)`
- assemble a new function expression which takes `inters` as an additional argument along with **B** and **H** with the resulting expression of the rewriting process in step 2 as the body.

This type of rewriting also comes in handy when optimizing neural inference.

The `inplace` function is defined simply as follows:

```
inplace_version(::typeof(*)) = mul!
inplace_version(::typeof(broadcast)) =
    @inline (out, f, args...) -> (broadcast!(f, out, args...); out)
# fallback. Take the extra argument but do the original operation
inplace_version(f) = @inline (out, args...) -> f(args...)
```

Here is the resulting code:

```
julia> @code_typed f!(intermediates, B, H)
```

Let's compare the performance:

```
julia> @b $f($B,$H)
```

```
107.881 ns (3 allocs: 384 bytes)
```

```
julia> @b $f!($intermediates, $B, $H)
```

```
33.644 ns
```

```
julia> f(B, H) == f!(intermediates, B, H)
```

```
true
```

## 1.6.4 Easy domain-specific optimization

In geometry the euclidean distance is a common operation. Users may implement this as  $\text{sqrt}(x^2 + y^2)$ , where  $x$  and  $y$  are the coordinates. There is a more numerically accurate way of computing it, called 'hypot' [8]. A geometry package can apply this domain-specific optimization by writing a simple rewrite rule in the symbolic language.

```
struct Point{T} x::T; y::T end
```

```
x(p::Point) = p.x
```

```
y(p::Point) = p.y
```

```
radius(p) = sqrt(x(p)^2 + y(p)^2)
```

```
f(p, q) = radius(p) - radius(q)
```

Note that to optimize  $f$ , a syntactic rewrite, such as that done by a hygienic macro, is not sufficient because  $f$  in turn calls the function `radius` which is where the expression we want

to rewrite is. The macro applied to `f` will not have access to the code of `radius`, especially if it resides in a different module. It would require type-propagation and reflection to obtain this code and rewrite it. However, one can easily trace through this function with a simple symbolic variable:

```
@variables a::Point b::Point

# "lift" existing `x` and `y` functions to be symbolic
@register_symbolic x(a::Point)::Real
@register_symbolic y(a::Point)::Real

f(a, b)
# => sqrt(x(p)^2 + y(p)^2) + sqrt(x(q)^2 + y(q)^2)
```

The trace can be rewritten with a pattern-based rule:

```
newexpr = replace(f(a, b), @rule sqrt(~a)^2 + (~b)^2 => hypot(~a, ~b))
# => hypot(x(a), y(a)) + hypot(x(b), y(b))

f' = makefunction(newexpr, a, b)
```

The function `f'` will look something like this:

```
function f'(p, q)
    hypot(x(p), y(p)) - hypot(x(p), y(p))
end
```

It is then possible to provide syntactic conveniences that abstract the above symbolic tracing followed by rewriting:

```
@symopt function f(p::Point, q::Point)
    radius(p) - radius(q)
```

```
end @rule sqrt((~a)^2 + (~b)^2) => hypot(~a, ~b)
```

Snippet 1.2: Replacing `sqrt(x^2 + y^2)` with `hypot(x,y)`

Notice the flexibility our system affords: the user was able to mark `x` and `y` as primitives in the symbolic tracing. The rule `@rule sqrt((~a)^2 + (~b)^2) => hypot(~a, ~b)` is easy enough for a Geometer to write down without knowing about how the Julia compiler works.

## 1.7 Contribution by chapter

- We identify software engineering problems in symbolic-numeric software, reflect on the cause, and analyse its symptoms. (Chapter 2)
- We reflect on the nature of the scientific computing, and the contributions of the Julia language, specifically multiple-dispatch, code generation, and staged programming. Then we design a symbolic subsystem on top of Julia. (Chapter 3)
- We introduce Symbolics, a free software symbolic-numeric system implemented in and for Julia (Chapter 4). Currently, Symbolics is actively utilized by 70 dependent projects and has a reach of 231 indirect dependencies, showcasing the practicality and impact of our approach. We discuss the technical contributions and compare Symbolics with other systems in Sections 4.9 and 4.10, respectively.
- By providing features for symbolic tracing and code generation, our system acts as a high-level basis for building domain-specific languages that can call numerical code from the whole Julia ecosystem of packages. We present case studies of Symbolics being used to build interesting scientific software. (Chapter 5)

## 1.8 List of publications

Chronological list of publications involving the author that are related to this thesis.

- “Sparsity Programming: Automated Sparsity-Aware Optimizations in Differentiable Programming” – Shashi Gowda, Yingbo Ma, Valentin Churavy, Alan Edelman, and Christopher Rackauckas – *Program Transformations for ML Workshop at NeurIPS 2019* [37]
- “High-performance Symbolic-numeric programming via multiple-dispatch” Shashi Gowda, Yingbo Ma, Alessandro Cheli, Maja Maja Gwózdź, Viral B. Shah, Alan Edelman, Christopher Rackauckas – *ACM Communications in Computer Algebra, Vol. 55, No. 3, Issue 217, September 2021* [36].
- “ModelingToolkit: A Composable Graph Transformation System For Equation-Based Modeling” – Yingbo Ma, Shashi Gowda, Ranjan Anantharaman, Chris Laughman, Viral B. Shah, Christopher Rackauckas – preprint Feb 2022. [59]
- “Catalyst: Fast and flexible modeling of reaction networks” – Torkel E. Loman, Yingbo Ma, Vasily Ilin, Shashi Gowda, Niklas Korsbo, Nikhil Yewale, Chris Rackauckas, Samuel A. Isaacson – *PLOS Computational Biology* 19, 10 (10 2023), 1–19. [57]
- “Groebner.jl: A package for Groebner bases computations in Julia” – Alexander Demin, Shashi Gowda – preprint 2023 [18]

# Chapter 2

## Symbolic-numeric status quo

In this chapter we consider the software engineering problems in today's symbolic-numeric programming systems. Generating numerical code is a feature that is available in most symbolic systems, however, they generate code of a different low-level language. We will reflect on some problems this creates. On the other hand, the only place numerical languages treat code as symbolic expressions is during compilation, but at this level mathematical meaning is thrown away, and expressions are dealt with according to floating point standards. Scientific users do not have any control over this phase of compilation, and cannot contribute domain expertise at this level of compilation.

### 2.1 The unfortunate siloing of symbolic software

Symbolic programming is seen as a sub-genre of scientific computing rather than an integral part of a system that is also numerically functional. We believe the main reason for this is that there has not been an implementation of a symbolic system in a language that is also productive and heavily used for numerical computing. The former attitude results in three categories of symbolic software:

1. Languages dedicated to symbolic computation, such as Mathematica and SymPy. Although they may be able to generate numerical (C/C++) code from symbolic code,

they are not also languages designed for writing efficient numerical code, so their vocabulary cannot contain arbitrary numerical methods.

2. The symbolic metaprogramming inside numerical systems such as FFTW, and NNLib. These systems don't use software from the first category, and would rather reproduce parts of it their implementation language.
3. Languages that use symbolic programming as the surface abstraction but have a hidden numerical workhorse built in a low-level language, examples in this category are JuMP, Convex optimizaiton libraries.

The main disadvantage is that, as of now, these categories do not share code. However, as we will demonstrate in this thesis, this division into three distinct categories is unnecessary. By fostering proper interaction between symbolic and numerical code, it becomes feasible to integrate these three types of software within a single system. The practical advantage is that software from each category can utilize functionalities from the others without the need to replicate certain aspects. For example, symbolic libraries can generate efficient algorithms that leverage an extensive array of existing numerical routines. Similarly, numerical functions can be "lifted" into the symbolic domain or transformed into "nodes" within symbolic expressions, significantly enhancing the power and utility of symbolic representations.

## 2.2 Symptoms of disharmony

Systems *without* properly interacting symbolic and numerical features exhibit one or more of these symptoms:

1. **They are inefficient at writing numerical algorithms** In systems in category 1, numerical routines are often implemented in low-level languages such as C. The host language focuses on symbolic programming, with features such as treating unbound names as symbolic variables, but these same features also preclude them from being a good implementation language for numerical code. However, features that support



robust numerical code without losing the flexibility required for symbolic programming can be efficiently implemented using macros in languages like Julia, which do not compromise on generality.

***Ideal:*** A language general enough for both numerical computing and symbolic computing.

2. **No support for user-defined symbolic expressions.** An ideal symbolic system should enable the creation of symbols that represent values of any type from the host language. These symbols often necessitate distinct algebraic and operator behaviors. Furthermore, expressions of the same type might require different data structures tailored to specific applications. Unfortunately, most existing packages are limited to providing a small variety of symbols, primarily numerical ones, and they standardize the data structure for expressions to be uniform and centrally defined. Additionally, a significant limitation is the lack of flexibility in defining which functions should be considered primitive when constructing symbolic expressions. To the best of our knowledge, most packages do not permit users to register new primitives dynamically at runtime.

***Ideal:*** A usefully generic definition of what constitutes a symbolic expression, and ways to define new kinds of symbolic expressions.

3. **They lack constant-space array symbols and expressions.** For a symbolic library to serve effectively as a metaprogramming tool in scientific software, it must be capable of representing vectors, matrices, and general  $n$ -dimensional arrays. We differentiate between an array of symbols—a straightforward concept—and a symbolic array, which is a symbol representing an array of known or unknown dimensions. Symbolic arrays and related expressions can be represented in constant  $O(1)$  space, regardless of the array's dimensions. This offers two advantages: first, it facilitates the application of simplification and transformation rules to these expressions; second, code derived from these expressions can seamlessly invoke routines that operate on

numerical arrays. Importantly, support for such array expressions could be built atop user-defined expression capabilities.

**Ideal:** the ability to represent arrays and operations on them symbolically, with well defined ways of turning the expressions back into numerical code.

4. **They only compile to numerical code in a different lower-level language.**

Although symbolic packages support numerical code generation, they typically offer a limited library of functions that the generated code can call. For instance, Mathematica uses an intermediate representation that directly maps to C. However, because Mathematica does not support efficient numerical code creation, it cannot incorporate existing numerical routines as nodes within the symbolic expression tree.

**Ideal:** a symbolic-numeric environment should allow the representation of calls to arbitrary library functions as symbolic nodes and compile these into executable numerical code.

As computer models become complex and larger in sheer size, generating numerical code from high-level symbolic representations becomes extremely important, and the limitations above will need to be overcome.

We recommend reading Section 4.10 for nuanced discussion of features of various symbolic-numeric systems. We tabulate the findings in Table 4.6.

## 2.3 DSLs – Modelica example

Domain-specific languages are useful in scientific computing. DSLs essentially mirror the tendency in mathematics to create new notation to suit a problem.

Modelica [32] is an object-oriented language for physical system simulation. Systems are written in a custom language, following is the example of an RC circuit:

Equations:

$$C \frac{V_c(t)}{dt} = i(t)$$
$$i(t) = \frac{E(t) - V_c(t)}{R}$$

Code:

```

model RC "RC model"

  Modelica.SIunits.Voltage E
  Modelica.SIunits.Voltage Vc
  Modelica.SIunits.Current i

  parameter Modelica.SIunits.Capacity C=1e-3
  parameter Modelica.SIunits.Resistance R=1e3

initial equation

  Vc = 0.0;

equation

  E = 10.0;
  C*der(Vc) = i;
  R*i = E - Vc;

end RC_v1;

```

The language is designed for description of systems that can be nested and connect with each other. This is essentially symbolic representation. The main task Modelica can do is transform this representation into one that can be executed by numerical differential equation and nonlinear equation solvers. A key part of this process is being able to compile to C code. Even though the language may not look like a traditional symbolic computing language such as Mathematica, it essentially needs to allow symbolic expressions and know how to deal with them. You can also notice that there is library of SI units.

Modia is an effort by the Modelica developers to rewrite it in Julia [64]. The motivation is that it is not easy to also have to maintain a language while developing symbolic and numerical algorithms for DAE simulations. Modia was developed before the work in this thesis using Julia expressions as its substrate symbolic expressions. In the Section ?? we discuss an alternative physical simulation system which uses the work in this thesis as the substrate.

We present SciML and the ModelingToolkit package in Section 5.1 which provides a counterpoint to the idea that one needs to invent a new language from scratch and create a

compiler for a DSL like Modelica.

## 2.4 Optimizations and low-level compilers

Low-level compilers (by default) allow only those optimizing transforms that result in the exactly equivalent numerical values. For examples, LLVM (Low-Level Virtual Machine) [55], which is what the Julia language uses, does not simplify the expression  $(a + b) - a$ .

```
julia> f(a,b) = a + b - a
f (generic function with 1 method)

julia> @code_llvm f(1.0,1.0)
define double @julia_f_133(double %0, double %1) #0 {
top:
    %2 = fadd double %0, %1
    %3 = fsub double %2, %0
    ret double %3
}
```

The reason is one specific case:  $0.0 + -0.0 - 0.0$  returns  $0.0$  according to the floating point standard, but simplification to  $b$  would cause it to return  $-0.0$ . This is not such a performance blunder, but consider that  $10 * a / 10$  also does not get simplified. The  $/$  operation is expensive.

The reader might make the observation that the above simplifications must be done by the programmer before compiling the program. But doing that would not be possible if there was just a little more abstraction involved:  $g(a, b) = a + b$ ;  $f(a, b) = g(a, b) - a$ .

LLVM does have flags to allow these optimizations[15], but in general, verifying these optimizations can be made is a hard [62]. LLVM makes the commitment to not talk in terms of real numbers at all, and only talk in terms of machine representable numbers.

In a well-designed symbolic-numeric system, we can view variables as reals when compiling. One can also imagine pattern-matching based rewrites of more complex expressions with instructions at a much higher level than at the LLVM level. For example, consider the rule `@rule ~x^2 + ~y^2 => hypot(~x, ~y)` in Snippet ??.

This might look like a micro-optimization but consider the expression `f.(A, B) .+ C` where `f` is applied element-wise on arrays `A` and `B` and the result is added element-wise with `C`. If there was a simplification performed on `f`, then this expression will never load `A` into memory, resulting in huge benefits in performance.

These are the exact kinds of rewrites that the Finch [2] structured array compiler performs on loops to obtain performance benefits. Finch uses the symbolic system described in this thesis.



## Chapter 3

# Synthesis of a symbolic-numeric system

In this chapter we develop an understanding of scientific computing, how the Julia language solves its problems, and finally how a symbolic-numeric system can be designed to compound Julia’s productivity and performance. Our system will be built on multiple-dispatch, homoiconicity, and staged programming—all features of Julia. Before Julia, libraries like NumPy [43] were written as a high-level productivity language layered on top of libraries in a low-level performance language. Julia addresses productivity and performance in a single language, this is why a symbolic-numeric system built in Julia can work to serve all levels of the scientific computing stack—from the very low level scalar operations to the high level array computations. Research languages like Fortress [76] also thought about simultaneous productivity and performance, but have not attained real-world popularity and remain obscure partly because they didn’t tend to the needs of the real-world community.

## 3.1 Nature of scientific computing

Scientific computing is characterized by the following facts:

1. **The same operators operate on a large variety of objects.** For example, the multiplication operator `*` works on pairs of numbers, matrices, a matrix and a vector, a row vector and any of the previous objects. Additionally it may act on symbolic expressions representing any of these objects. Further, there can be many kinds of matrices: dense, sparse, diagonal, or other structured storages.
2. **Really about domains rather than types.** Scientific problems care about sets of values rather than on the type. Whether a value is real or complex, positive or negative, greater than 1, symmetric, Hermitian, matter more than the type representing their storage.
3. **Dynamic languages are preferred.** A scripting-style workflow is common in scientific computing. An exploratory workflow is common in science—load a script or module, load some data, try different hypotheses in a read-eval-print-loop (REPL).
4. **Performance is paramount.** the lesser the amount of time a computation takes, the bigger the size of the problem users can solve. Research advances at the cutting-edge of performance. However, this requirement is in conflict with the previous one—dynamic languages introduce uncertainty at runtime, causing slowness. The answer to this problem before Julia was to “vectorize” operations. That is make operations on matrices and vectors really fast by implementing them in a lower-level language like C, and provide surface-level interaction with it through a dynamic language. This is the two-language problem, as we will see in the next section, this distinction is not necessary.
5. **Need for domain-specific languages.** Mathematical formulations of many problems need an entirely different way of expressing the problems than imperative programming. For example, an optimization library needs a language to express con-



straints, a modeling and simulation library needs a language to express differential relationships between quantities. In these examples, variables are placeholders for numerical values during simulation rather than variables of the host language. Yet, the DSL needs to achieve its solutions fast. To solve such problems there has been a tendency to create a new compiler for every problem. Some examples of DSLs are COPASI, BioNetGen, PySB.

6. **Interdisciplinary collaboration.** Scientific computing effort mirrors the collaboration in real world research. For example, a climate modeler needs code about geometry, cartography, differential equations, PDE discretizations, visualization, and data storage. Experts from these fronts should be able to provide tools that are palatable to climateologists.

For a longer discussion of the nature of technical (synonymous to scientific) computing, see Chapter 2 of *Abstraction in Technical Computing* by Jeff Bezanson [50]. In the next section we see how the Julia language meets most of these needs. Then we consider how the Symbolic-numeric system can be built on top of it.

## 3.2 Contribution of Julia

### 3.2.1 Dynamic multiple-dispatch

Multiple-dispatch is a way of organizing code that is a generalization of single-dispatch popular in class-based object-oriented programming languages like Java, Python, etc.

For example, using single-dispatch, a bidiagonal matrix implementation could have the following code. The bidiagonal matrix can be either an upper or a lower triangular matrix where only the diagonal and one subdiagonal are nonzero.

```
class Bidiagonal <: AbstratMatrix  
    function initialize(diag, subdiag, type)
```

```

    self.diag = diag
    self.subdiag = subdiag
    self.type = type
end
function multiply(B::AbstractMatrix)
    ... # matrix-matrix product
end
function add(B::Bidiagonal)
    if self.type == B.type
        return Bidiagonal(...)
    else
        return Tridiagonal(...)
    end
end
end
...
end

```

Snippet 3.1: Sketch of a Bidiagonal matrix implementation in an imaginary single-dispatch language

Let's say, the operation  $A * B$  in this language is syntactic sugar for `A.multiply(B)`, and  $A + B$  for `A.add(B)`. Some remarks can be made about this:

- The method to perform  $A * B$  where `A` is a `Bidiagonal` and `B` is any other kind of `AbstractArray` *belongs to* the class `Bidiagonal`.
- In  $A * B$ , if `B` is a different subclass of `AbstractMatrix`, let's say, a column-major dense matrix, of type `Matrix`, then the definition here only works to multiply  $A * B$  and it's clear that  $B * A$  must be handled by the class `Matrix`. But `Matrix` maybe defined in a different library (presumably a standard library), and the library implementing the `Bidiagonal` class cannot modify `Matrix`'s definition to add this method. You can

imagine an arrangement where `Bidiagonal` can define a `leftmultiply` method, but that would be ambiguous with `Matrix.multiply(B::Bidiagonal)` if there needed to be such a method. Even if there was a way to break the tie in such a case, this seems overall like a messy situation, a hack.

- The methods on `Bidiagonal` become fixed once the `Bidiagonal` class is written down. Let's say a `Solver` class wants to implement a `solve` function which works on various matrix types. After having imported the module with `Bidiagonal`, it can implement `Solver.solve(b::Bidiagonal, x::AbstractVector)`. On the other hand, if `Bidiagonal` module came into existence after the `Solver` module, and yet wanted to provide a `solve` functionality so that existing users of `Solver.solve` could also use `Bidiagonal` matrices, it would be out of luck.

These are some of concerns Julia's dynamic multiple-dispatch can address, apart from providing performance. Now let's see what the above code would look like in Julia:

```
struct Bidiagonal{T} <: AbstractMatrix{T}
    diag::Vector{T}
    subdiag::Vector{T}
    type::Symbol
end

function *(A::Bidiagonal, B::AbstractMatrix)
    ... # matrix-matrix product
end

function *(A::AbstractMatrix, B::Bidiagonal)
    ...
end

function *(A::Bidiagonal, B::Bidiagonal)
    # this method is necessary because it is ambiguous which of
```

```

        # the above are to be chosen in case both arguments are Bidiagonal
end

function +(A::Bidiagonal, B::Bidiagonal)
    if self.type == B.type
        return Bidiagonal(...)
    else
        return Tridiagonal(...)
    end
end

end

import SolverModule: solve

function solve(A::Bidiagonal, b::AbstractVector)
    ... # add a solve method for Bidiagonal
end

```

Snippet 3.2: Sketch of a Bidiagonal matrix implementation in Julia

- The definitions of the methods have been moved out of the definition of the datastructure.
- `*` naturally has the right methods: `Bidiagonal * AbstractMatrix`, `AbstractMatrix * Bidiagonal`, and `Bidiagonal * Bidiagonal`. The first argument is no more “special”, the name multiple-dispatch is used to denote that the dispatch (selection of code) occurs on the types of all the arguments.
- We completely avoided talking about ownership: no one class owns the generic operator

`*`, it stands by itself. Any function which is written using `*` will be polymorphic on these inputs. (For example a product function which takes a vector of objects).

- New definitions of `*` can be loaded dynamically into the environment. This sometimes might require a recompilation of methods previously defined. See the paper on world age in Julia [4] for more on this.
- If `solve` is a generic function from a `SolverModule` module, the module defining `Bidiagonal` can import the `SolverModule`, and add a method to `solve`. Now another module which uses `SolverModule` alone, does not at all need to know about `Bidiagonal`. If however, it imports both `SolverModule` and `Bidiagonal`, it can call `solve` on `Bidiagonal` matrices!

Language design has implications on the social phenomena of development of an ecosystem of packages, especially in an open source ecosystem. The above multiple dispatch strategy clearly sets up a platform for a kind of layered abstraction—new types can be added to the system that depend on old types, and can also affect how existing operators behave on the new types as well as in conjunction with old types. We say “old” and “new” here to mean an earlier or a later layer of abstraction. We will explain more on type hierarchy, parametric types, and their use in performance in the next section.

In single-dispatch, the class becomes the unit of abstraction. In the extreme case like Java where all methods must be inside a class, there is a tendency to orient programming towards the nouns in the system. Sometimes you will end up inventing new nouns for the sake of doing this. In Julia, the generic function, the verb, is the unit of abstraction. This is in line with mathematics where operators act on objects of various types, and there is little need for an actor who needs to save some hidden state between operations [89].

### 3.2.2 Type inference, code selection and code specialization

Type inference in Julia works by first looking at types of inputs to a function and propagating a best guess output type for every function call and variable in the function body. For

example consider the function:

```
function f(A, B, C)
    (A + B) * C
end
```

Suppose `f` is called with 3 complex `Float64` numbers, we can naturally infer the the following types for subexpressions:

```
function f(A::Complex{Float64},
           B::Complex{Float64},
           C::Complex{Float64})::Complex{Float64}
    ((A + B)::Complex{Float64} * C)::Complex{Float64}
end
```

Julia uses dynamic types that are different from types in statically compiled languages in that they don't need to be completely erased after compilation. Types can be present at runtime. Properties like the number of diagonals are important in numerical computing, yet they are highly dynamic. Consider the `add` and `+` methods on `Bidiagonal` above. If both inputs are upper bidiagonal, or both are lower bidiagonal, the output is a bidiagonal matrix; if however, one is upper and the other is lower, the output becomes `Tridiagonal`. The type of this function is inferred as a Union type.

```
function f(A::Bidiagonal{Float64},
           B::Bidiagonal{Float64},
           C::Bidiagonal{Float64})::Matrix{Float64}
    ((A + B)::Union{Bidiagonal{Float64}, Tridiagonal{Float64}} * C)::Matrix{Float64}
end
```

The inferred type of `(A + B)` is `Union{Bidiagonal{Float64}, Tridiagonal{Float64}}`.

Which subsequent `*` method should `(A+B) * C` call? The answer is of course, it depends on the output at runtime! Since this is a simple Union with two types, Julia can compile a branch here:

```

TMP = (A + B)
if TMP isa Bidiagonal
    # call * (::Bidiagonal{Float64}, ::Matrix{Float64}) on (TMP, C)
elseif TMP isa Tridiagonal
    # call * (::Tridiagonal{Float64}, ::Matrix{Float64}) on (TMP, C)
end

```

This is known as Union-splitting. Note that Union splitting is not enough in cases where the inferred type is an *abstract type* or else there are too many types in the Union – in such cases Julia resorts to doing dispatch at runtime. This can be quite slow, so there is incentive to write code in such a way that it can be inferred and it does not create a large number of Unions.

Multiple-dispatch is used to select the right method to call as we saw in this section, but it turns out that this means of abstraction also has an important consequence for performance – it can be used to write algorithms that are fast for specific combinations of input types. The field of scientific computing is rife with the need to do this [50], and it results in a language that is both performant and highly productive. It supports a layered approach to abstraction in scientific open source community where new types and efficient methods are constantly added to existing generic functions. Any function that use those generic functions continue to benefit from this.

### 3.2.3 Macro programming

Julia supports Scheme-style Hygienic macros. Macros read an argument expression and rewrite it into a different one, which is then actually evaluated. This poses an important limitation—if a macro is looking at a function call, it has no way of looking *inside* the body of that function.

Imagine a macro `@hypot_optimize` that takes in an expression that matches the structure:  $a^2 + b^2$  and rewrites it as `hypot(a, b)`:

```
@hypot_optimize x^2 + y^2 # works!
```

```
len(x, y) = x^2 + y^2
```

```
@hypot_optimize len(x, y) # does not!
```

The first invocation works since the expression passed to the function call matches the criteria. However, the second invocation cannot work even though it is essentially doing the same operation. This is where symbolic tracing becomes more powerful than macro programming as we see in the example in Section 1.6.4.

In the Symbolics system, we use macros heavily to provide syntactic conveniences. One should remember that macros are just that: syntactic conveniences, and does not add to the expressive power of the base language [28].

### 3.2.4 Staged compilation

Staged compilation is the generation of the body of a method during compile time, based on the concrete types of the input arguments it is called with. For every combination of input argument types it only generates the body once and Julia reuses it the next time it sees the same argument types. A staged function is written with the `@generated` macro, and has the same syntax a functions, except, within the body, the formal arguments refer to the type of the corresponding inputs.

```
julia> @generated function foo(x)
```

```
    Core.println(x)
```

```
    return :(x * x)
```

```
end
```

```
julia> foo(2)
```

```
Int64
```

```
4
```



```
julia> foo(3)
```

```
6
```

In this example, the first time `foo` was called, it printed the type `Int64` along with the output of `x*x`, showing that it ran the body of the function with the type in order to take its return value as the new function body! We defer longer discussion of generated functions to the Julia documentation [19].

## 3.3 Symbolic-numeric subsystem

### 3.3.1 Expressions: A term interface

The purpose of a symbolic subsystem is expression manipulation. But what is an expression? The answer cannot be a single datastructure, because we want a system that adapts to the needs of the domain. Some expressions may require compact representation (like Polynomials that expand on construction), while others may require symbols that don't behave like numbers (such as symbolic arrays), and yet others may not even deal with numerical computing at all, but be concerned with symbolic representation of code of a richer variety with loops and conditionals. To cater to all these needs and to allow reuse of manipulation functions we came up with a “term interface” that defines what an expression is and how to construct it, without talking about the underlying datastructure. This starting point creates the balance necessary for an ecosystem of symbolic packages to interoperate successfully. To reiterate, the requirements we keep in mind as we build the expression interface are:

1. **Encapsulation of representation:** no matter what datastructure is used to store an expression, expression manipulation code should be able to 1) traverse and 2) create such expressions without actually knowing about the datastructure's internals.
2. **Symbolic polymorphism:** symbols of different types, and with different operator behaviors, must be permissible.

3. **Metadata:** different modules should be able to attach metadata to expressions without namespace conflicts.

Hence, the lowest common denominator view of an expression is the typed S-expression with metadata. This is what the expression interface allows. To implement the expression interface, a module defining a term representation (i.e. a data structure for a specific kind of terms), must extend the following generic functions.

- `symtype(expr)` – returns the symbolic type of the expression. We will refer to this simply as `symtype`. This defaults to the type of `expr` itself, making sure it applies to all julia objects, symbolic and non-symbolic. In other words, non-symbolic objects can be nodes in a symbolic tree.
- `issym(expr)` – returns true if `expr` is a symbol. (Distinct from Julia’s `Symbol`. This symbol has a `symtype` as well as a name). If true, `nameof(expr)` must be defined.
- `nameof(expr)` – the Julia Symbol that gives the name of `expr`.
- `isexpr(expr)` – returns a boolean which checks if `expr` is a symbolic expression. If yes, the below functions must apply to it.
- `head(expr)` – a head of the expression. This distinguishes the different kinds of expressions. A function call is one such type.
- `children(expr)` – returns the arguments to construct the expression.
- `maketerm(T, head, children, symtype, metadata=nothing)` – returns an expression. **T** is a super type which T descends from (or a sibling of T, if desired).

All but the last interface functions deal with querying the S-expression. The last one, `maketerm`, creates an S-expression. By using "the closest available **T**", an expression manipulation function can work on all expression currently in the ecosystem and that will ever be created in the future.

A generic symbolic substitution function can be written using only these interface functions, and avoiding completely any implementation detail of the representation.

```
function substitute(expr, dict)
  if haskey(dict, expr)
    dict[expr]
  elseif isexpr(expr)
    # recursively substitute children
    maketerm(typeof(expr),
              head(expr),
              map(c -> substitute(c, dict), children(expr)))
  else expr end
end
```

Snippet 3.3: substitute function to replace variables with values.

### Example: a closure of symbolic expression types

Consider the following representation of expressions: `Basic` represents a generic function, `Mul` represents multiplication of powers, `Add`

```
abstract type NumberExpr end

struct Basic <: NumberExpr
  f
  args::Vector
end

struct Mul <: NumberExpr # a monomial
  dict::Dict # base => power
end
```

```

struct Add <: NumberExpr
    dict::Dict # term => coefficient
end

```

Something like  $\sin(\theta)x - 2y$  maybe represented as:

```
Add(Dict(Mul(Dict(Basic(sin, θ)=>1, x=>1)=>1, y=>-2)))
```

Of course, the construction can be made to happen via methods to the `sin`, `*` and `+` generic functions from Base Julia.<sup>1</sup>

In other words, it represents the expression.

```
1*(sin(θ)^1 * x^1) + -2 * y
```

One can overload the printing of these types such that julia prints them in a way that is human friendly, with the no-ops like `1*` and `^1` removed.

Representing with 3 types has benefits over doing it with just one `Basic` type:

- `Mul` can merge monomial terms. For example, `Mul(Dict(x=>1, y=>2)) * Mul(Dict(x=>1))` is `Mul(Dict(x=>2, y=>2))`. This operation is on the order of the size of the smaller term, but tends to lead to compact expressions and lesser need for simplification.
- Similarly `Add(2, Dict(x=>1, y=>2)) + Add(3, Dict(x=>1)) = Add(Dict(5, x=>2, y=>2))`. `Add` provides the same benefit as `Mul`.

The term interface is implemented as follows:

#### Snippet 3.4: Term interface definition for `Add` and `Mul`

---

<sup>1</sup>In fact, defining new symbolic types that are numbers are common enough and nuanced enough that Symbolics uses a macro (`@number_methods`) to generate these methods automatically.

```

isexpr(x::NumberExpr) = true
head(x::NumberExpr) = x isa Basic ? x.f : x isa Add ? + : *
children(x::Basic) = x.args
children(m::Mul) = [k^v for (k, v) in m.dict] # every child is a power
children(a::Add) = [k*v for (k, v) in a.dict] # every child is coeff*term
maketerm(::Type{<:NumberExpr}, head, children) = Basic(head, children)
maketerm(::Type{<:NumberExpr}, ::typeof(+), children) = construct_add(children)
maketerm(::Type{<:NumberExpr}, ::typeof(*), children) = construct_mul(children)

```

This makes all existing and future symbolic manipulation functions such as `substitute` work on `Basic`, `Mul` and `Add`.

The term interface functions, without any specific methods, are defined in an extremely lightweight package called `TermInterface` [35]. Any package can include the module and add methods to the functions to define a new term or, just as well, to write term rewriters. We will explore term rewriting in 3.3.3

## Concrete expression types

These are a few examples of the symbolic expression types that support the term interface in the wild:

- `Sym{T}`: a symbol with symtype `T`.
- `Term{T}`: a basic term capturing a function call `f(args...)`
- `BasicSymbolic`: a compact type to represent canonical forms of numeric expressions. This is a combination of `Add` and `Mul` shown in the example above into one datastructure to save compile time.
- Code nodes such as `Func`, `Let`, `Cond`, `Block`, `Assignment`, `SetArray`, `MakeArray`, `MakeSparseArray`, `MakeTuple`, `SpawnFetch{T}`. Each one represents a specific fragment of Julia code. They are used in the code generation phase we discuss in Chapter 4.

- **ArrayOp**: a term type to represent array operations in Einstein summation notation. See Section 4.5 for more on symbolic arrays.
- **Expr**: the Julia Expr itself is given the term interface by Metatheory.jl [14], an equality saturation library. Note that this library also works out of the box with all the above types, thanks to the term interface.
- **FinchNode**: These are nodes in the Finch [2] sparse tensor co-iteration compiler. These represent an intermediate representation internal to Finch, including function calls, indexing, caching, loop nests, sieve and so on.

### Definition: Symbolic tracing

We use the term symbolic tracing to mean executing code with symbols. The result of such a run is a symbolic trace of the program.

### 3.3.2 Partial information

The term interface also allows attaching metadata to terms. The interface an expression implementation must define for metadata is:

- `metadata(x::T, d::AbstractDict{DataType, Any}) →x'` where the output `x'` is also of type `T` but stores the metadata somewhere inside it.
- `metadata(x::T) →d` where the output `d` is an abstract dictionary with data types as key, and any kind of values.

Once these methods exist, `TermInterface` will automatically provide:

- `hasmetadata(x, K)`: checks whether `x` has metadata `K`
- `setmetadata(x, K, v)`: sets a single metadata entry, with key `K` and value `v`
- `getmetadata(x, K)`: get the metadata entry with key `K`

Note the use of datatypes as keys. This provides automatic namespacing of metadata. An alternative could have been using symbols, but if two modules could attach a metadata with the key, say `:source` onto the same object, they will confuse each other. However they can both define a type called `Source`, and these types will be automatically namespaced by Julia as `Module1.Source` and `Module2.Source`.

### 3.3.3 Building on the term interface

The term interface is the exact right amount of abstraction necessary to build the symbolic-numeric system on top of. As we saw in the implementation of `substitute`, this interface provides an easy way to write storage-agnostic term rewriters. In the next chapter we look at the concrete implementation of `Symbolics.jl` package with the starting point of this term interface. Two important next pieces come together:

- **Term rewriting**, i.e., taking a term and rewriting it into a different term or value. A simple definition of a rewriter will serve to be quite useful: A rewriter is a function that takes an expression and returns either an expression, a constant value, or `nothing`. `nothing` represents that the input symbolic expression did not merit a rewrite. This sentinel result is useful in short-circuiting the rewriting process.

Term rewriting code should work only using the term interface described above in order to become useful on an unbounded set of symbolic types. We explore the various kinds of possible rewriters in the next chapter in Section 4.2.

- **Code generation**: By representing terms flush with Julia function objects and literal values, along with symbolic variables, the terms in Julia can faithfully represent code in the language. This allows a natural code generation which directly turns symbolic expressions into Julia AST, and finally into a generated function which executes that AST. We develop the code generator in Section 4.4.





# Chapter 4

## The Symbolics.jl system

This chapter builds on the background in previous chapter. It shows the construction of the Symbolics.jl system, enumerates its features, discusses its contributions (Section 4.9), and compares it with existing (Section 4.10).

### 4.1 Expression types and canonicalization

#### 4.1.1 Numeric expressions

Symbolics represents defines a number of expression types that implement the term interface defined in Section 3.3.1. The `Sym` and `Term` types are the most generic and simple – `Sym` represents an unknown, `Term` represents a function call. `Add`, `Mul`, and `Div`<sup>1</sup> that together try to represent numeric expressions in a simplified canonical form. They are described below. This canonical form is inspired by SymEngine [81] and Reduce.

- **`Sym{T}(name)`**: A symbolic named unknown (also called a symbol or a variable in this text.). `name` is a Julia Symbol. `T` is the symtype. To create a symbol that

---

<sup>1</sup>In practice we combine all these types into a single struct called `BasicSymbolic` and use `Unityper.jl` [27] to generate a specialized type. This is to avoid unnecessary compiler overhead and increase the type stability of symbolics code. For the sake of this explanation, it serves us better to talk about them as separate types.

represents a function, and is callable with arguments, one can use the special type `FnType{ArgType, OutputType}`. Note that this type is defined by Symbolics and is not the type of Julia functions. Every Julia function has a unique type that is a subtype of `Function` and does not encode in its parameters the input and output types. `FnType` lets us know what type the term formed by calling a symbolic function is going to be.

- **Term{T}(f, args)**: A basic term of symtype `T` representing the function call `f(args...)`. Note that `f` can itself be a `Sym` or a symbolic expression of `FnType` described above. Unitary functions and functions that aren't commutative or associative, such as `sin` or `hypot` are stored as `Terms`. Commutative and associative operations like `+`, `*`, and their complementary operations like `-`, `/` and `^`, when used on terms of symtype `<:Number`, stand to gain from the use of more efficient datastructures.
- **Add{T}(coeff, dict)**: Linear combinations such as  $\alpha_1 x_1 + \alpha_2 x_2 + \dots + \alpha_n x_n$  are represented by `Add(Dict(x1 =>  $\alpha_1$ , x2 =>  $\alpha_2$ , ..., xn =>  $\alpha_n$ ))`. Here, any  $x_i$  may itself be other types mentioned here, except for `Add`. When an `Add` is added to an `Add`, we merge their dictionaries and add up matching coefficients to create a single “flattened” `Add`.
- **Mul{T}(coeff, dict)**: Similarly,  $x_1^{m_1} x_2^{m_2} \dots x_{m_n}$  is represented by `Mul(Dict(x1 => m1, x2 => m2, ..., xn => mn))`.  $x_i$  may not themselves be `Mul`, multiplying a `Mul` with another `Mul` returns a “flattened” `Mul`.
- **Div{T}(p, q)**:  $p/q$  is represented by `Div(p, q)`. The result of `*` on `Div` is maintained as a `Div`. For example, `Div(p1, q1) * Div(p2, q2)` results in `Div(p1 * p2, q1 * q2)` and so on. The effect is, in `Div(p, q)`, `p` or `q` or, if they are `Mul`, any of their multiplicands is not a `Div`. So `Muls` must always be nested inside a `Div` and can never show up immediately wrapping it. This rule sets up an expression so that it helps the `simplify_fractions` procedure described in Section 4.7.

## 4.1.2 Implementation of the term interface

For all the numerical types above, the `head` function is defined to return `BasicSymbolic`. This essentially indicates that the expression is one of the above types. This allows us to create a common `maketerm` method that accepts any of these types and returns an object of one of these types. We do this because rewriting, for example, an `Add` might return in a `Mul`.

The vector `children(<expr>)` returns for every kind of expression is given below:

Type	children
<code>Term{T}(f, args)</code>	<code>[f, args...]</code>
<code>Add{T}(coeff, dict)</code>	<code>[+, coeff, sort_args([v*k for (k,v) in dict])...]</code>
<code>Mul{T}(coeff, dict)</code>	<code>[*, coeff, sort_args([k^v for (k,v) in dict])...]</code>
<code>Div{T}(p, q)</code>	<code>[\, p, q]</code>

Table 4.1: Definition of `children` on various storage types of terms

Since iteration on dictionaries is unordered, we have used a `sort_args` to denote that the monomials of `Add` and `Mul` are ordered. Symbolics uses the reverse degree lexicographic ordering [17] by default.

## 4.1.3 Polynomial representation

While `Add` and `Mul` and other types above serve quite well when performing a symbolic trace, they are not the most efficient representation for polynomial algebra. Abstract algebra modules in Julia [30] [56] have much faster representations of multi-variate polynomials. The limitation of such packages though is that they cannot represent transcendental functions in their expressions.

To get the best of both worlds, i.e., be able to represent any expression including polynomials with transcendental terms, and to be able to work with those polynomials efficiently, Symbolics has a `PolyForm` expression type.

They are designed specifically for multi-variate polynomials. They provide common algorithms such as multi-variate polynomial GCD. The restrictions that make it fast also mean some things are not possible: Firstly, DynamicPolynomials can only represent flat polynomials. For example,  $(x-3)*(x+5)$  can only be represented as  $(x^2) + 15 - 8x$ . Secondly, DynamicPolynomials does not have ways to represent generic Terms such as  $\sin(x-y)$  in the tree.

`PolyForm` type holds a polynomial and the mappings necessary to present the polynomial as a Symbolics expression (i.e. by defining `head` and arguments). The mappings constructed for the conversion are 1) a bijection from DynamicPolynomials `Variable` type to a Symbolics `Sym`, and 2) a mapping from `Syms` to non-polynomial terms that the `Syms` stand-in for. These terms may themselves contain `PolyForm` if there are polynomials inside them. The mappings are transiently global, that is, when all references to the mappings go out of scope, they are released and re-created.

```
julia> @syms x y
(x, y)
```

```
julia> PolyForm((x-3)*(y-5))
x*y + 15 - 5x - 3y
```

In the polynomial, terms which are not `+`, `*`, or `^` calls are replaced with a unique symbol obtained by hashing the term, and a mapping from this new symbol to the original expression it stands-in for is maintained as stated above.

```
julia> p = PolyForm((sin(x) + cos(x))^2)
(cos(x)^2) + 2cos(x)*sin(x) + (sin(x)^2)

julia> p.p # this is the actual DynamicPolynomial stored
cos_36584109372687415492 +
2cos_3658410937268741549 * sin_10720964503106793468 +
sin_107209645031067934682
```

By default, polynomials inside non-polynomial terms are not also converted to PolyForm. For example,

```
julia> PolyForm(sin((x-3)*(y-5)))  
sin((x - 3)*(y - 5))
```

But the `recurse=true` keyword argument will do this.

```
julia> PolyForm(sin((x-3)*(y-5)), recurse=true)  
sin(x*y + 15 - 5x - 3y)
```

Polynomials are constructed by first turning symbols and non-polynomial terms into DynamicPolynomials-style variables and then applying the `+`, `*`, `^` operations on these variables. You can control the list of the polynomial operations with the `Fs` keyword argument. It is a `Union` type of the functions to apply. For example, let's say you want to turn terms into polynomials by only applying the `+` and `^` operations, and want to preserve `*` operations as-is, you could pass in `Fs=Union{typeof(+), typeof(^)}`

```
julia> PolyForm((x+y)^2*(x-y), Fs=Union{typeof(+), typeof(^)}, recurse=true)  
((x - (y)))*((x^2) + 2x*y + (y^2))
```

Note that in this case `recurse=true` was necessary to indicate that the polynomialization should descend into the `*` operation.

#### 4.1.4 Default implicit assumptions, and how to avoid them

By default, the canonicalization process assumes several things, we present the assumptions with an example for each one:

All these can be avoided by initializing `x` and `y` to be `LiteralReal` rather than the implicit `Real` variables:

Input	Output	Implicit Assumption
$y*x$	$x*y$	* is commutative
$x/(x^2)$	$1/x$	$x \neq 0$
$x-x$	$0$	$x \neq \text{NaN}$ and $x \neq \text{Inf}$
$2(x+y)$	$2x + 2y$	+ is distributive
$x+x$	$2x$	
$x*x$	$x^2$	

Table 4.2: Implicit assumptions made in automatic simplification

```
julia> @variables x::LiteralReal y::LiteralReal;
```

```
julia> y*x, x/(x^2), x-x, 2*(x+y), x+x, x*x
```

```
(y * x, x / (x ^ 2), x - x, 2 * (x + y), x + x, x * x)
```

`LiteralReal` is a subtype of Julia's `Real` and hence can be used to trace through the same code that accepts `Real` variables. This works by defining the numerical methods on all `Symbolic{LiteralReal}` to result in a `Term`.

## 4.2 Term rewriting

### What is a rewriter?

A rewriter is any function or function-like callable object that accepts an expressions and returns either `nothing` or an expression. One can always assume that receiving `nothing` from a rewriter means that the expression did not need to be rewritten. Rewriters are also free to return the unmodified expression in this case, but `nothing` is preferred so as to allow short-circuiting in chains of rewriters.

## 4.2.1 Rule-based rewriting

A rule matches terms of a certain form and rewrites them by executing a right-hand-side expression.

The syntax of `@rule` is: `@rule <pattern> => <consequent>` where `<pattern>` is a pattern to match, and `<consequent>` is a piece of code to run once a match has occurred. The argument to a rule is an expression, and the rule matches the entire expression.

Symbolics has a pattern matching language for easily querying terms and accessing parts inside them. Patterns are essentially `Terms` that may contain a slot or segment, the pattern matching algorithm matches the internals of the term with an input term part-by-part storing any slots or segments in a match dictionary.

An example rule is:

```
r = @rule sin(2(~x)) => 2sin(~x)*cos(~x)
```

This matches expressions like `sin(2x)` or `sin(2acos(θ))` and rewrites them into `2sin(x)*cos(x)` and `2sin(acos(θ))*cos(acos(θ))` respectively.

`x` is called a slot and is a stand-in for any subexpression.

Because a rule is a rewriter, it can be called with an expression:

```
r(sin(2x))
# => 2sin(x)*cos(x)
r1(sin(3z))
# => nothing
```

Rules can have many slot variables:

```
r2 = @rule sin(~x + ~y) => sin(~x)*cos(~y) + cos(~x)*sin(~y);
r2(sin(α+β)) # => sin(α)*cos(β) + cos(α)*sin(β)
```

A segment variable in place of a slot variable matches 0 or more argument subexpressions at once. `~~xs` in the following example is a segment variable:

```

@rule(+(~~xs) => ~~xs)(x + y + z)
# => [x, y, z]

r = @rule ~x * +(~~ys) => sum(map(y-> ~x * y, ~~ys));
r(z * (w+w+α+β))

# => w*z + w*z + z*α + z*β

```

The pattern matcher is implemented with continuation-passing-style backtracking code. It is similar to the one described in [42].

## Predicates

Slots and segments may be annotated with predicate functions in Symbolics. For instance, the expression  $\sim x :: \mathbf{g}$  attaches a predicate  $\mathbf{g}$  to a segment variable, where  $\mathbf{g}$  is a function that accepts a vector of expressions and returns a boolean value. It is imperative that if the same slot or segment variable is repeated within a pattern, only one occurrence may include a predicate.

In the following example in Julia we use a predicate function to rewrite

```

@syms a b c d

r = @rule ~x + ~~y::(ys->iseven(length(ys))) => "odd terms";

r(a + b + c + d) # => nothing
r(b + c + d)     # => "odd terms"

```

### 4.2.2 Associative-Commutative Rules

An expression  $f(x, f(y, z, u), v, w)$ , a  $f$  is said to be associative if the expression is equivalent to  $f(x, y, z, u, v, w)$  and commutative if the order of arguments does not



matter. Symbolics has a special `@acrule` macro meant for rules on functions that are associative and commutative such as addition and multiplication of real and complex numbers.

A `@rule` for a pythagorean identity does not match the commutated expression:

```
pyid = @rule sin(~x)^2 + cos(~x)^2 => 1
```

```
pyid(cos(x)^2 + sin(x)^2) # => nothing
```

But `@acrule` can:

```
acpyid = @acrule sin(~x)^2 + cos(~x)^2 => 1
```

```
acpyid(sin(x)^2 + cos(x)^2) # => 1
```

```
acpyid(cos(x)^2 + sin(x)^2) # => 1
```

### 4.3 Rewriter combinators

A rewriter is any callable object that takes an expression and returns an expression or `nothing`. If `nothing` is returned that means there was no changes applicable to the input expression. The Rules we created above are rewriters.

The `SymbolicUtils.Rewriters` module contains some types that create and transform rewriters.

- `Empty()` is a rewriter that always returns `nothing`
- `Chain(itr)` chain an iterator of rewriters into a single rewriter that applies each chained rewriter in the given order. If a rewriter returns `nothing` this is treated as a no-change.
- `RestartedChain(itr)` like `Chain(itr)` but restarts from the first rewriter once on the first successful application of one of the chained rewriters.
- `IfElse(cond, rw1, rw2)` runs the `cond` function on the input, applies `rw1` if `cond` returns true, `rw2` if it returns false

- `If(cond, rw)` is the same as `IfElse(cond, rw, Empty())`
  - `Prewalk(rw; threaded=false, thread_cutoff=100)` returns a rewriter that does a pre-order (*from top to bottom and from left to right*) traversal of a given expression and applies the rewriter `rw`. `threaded=true` will use multi threading for traversal. Note that if `rw` returns `nothing` when a match is not found, then `Prewalk(rw)` will also return `nothing` unless a match is found at every level of the walk. If you are applying multiple rules, then `Chain` already has the appropriate passthrough behavior. If you only want to apply one rule, then consider using `PassThrough`.
- `thread_cutoff`
- is the minimum number of nodes in a subtree that should be walked in a threaded spawn.
- `Postwalk(rw; threaded=false, thread_cutoff=100)` similarly does post-order (*from left to right and from bottom to top*) traversal.
  - `Fixpoint(rw)` returns a rewriter that applies `rw` repeatedly until there are no changes to be made.
  - `PassThrough(rw)` returns a rewriter that if `rw(x)` returns `nothing` will instead return `x` otherwise will return `rw(x)`.

The Finch sparse tensor compiler uses Symbolics' rule-based rewriting system. Below is the table of number of places in the code Finch uses `@rule` and the rewriter combinators:

Type	Occurances
<code>@rule</code>	267
<code>Prewalk</code>	11
<code>Postwalk</code>	54
<code>Chain</code>	34
<code>Fixpoint</code>	34

This demonstrates the effectiveness of the rewriter abstraction.

## 4.4 Code generation

Symbolics has term types to represent code, they can be constructed recursively to create complex pieces of code. Calling the `toexpr` function will turn terms into Julia expressions. In the table below, we show an example of each term type and the resulting Julia expression on calling `toexpr` on it.

Term	Julia Expr ( <code>toexpr</code> )
<code>Assignment(a, b) also a ← b</code>	<code>a = b</code>
<code>Let([a ← foo(b)], body)</code>	<code>let a = foo(b); body; end</code>
<code>Func([x, y], [w, z←a], body)</code>	<code>function (x,y; w, z=a) body end</code>
<code>Func(DestructuredArgs([x,y]), [], body)</code>	<code>function (_arg1) let (x,y)=_arg1 body end end</code>
<code>SetArray(out, [a,b,c])</code>	<code>begin out[1] = a; out[2] = b; out[3] = c end</code>
<code>MakeArray([a,b,c], <b>Array</b>)</code>	<code>vcat(a,b,c)</code>
<code>MakeArray([a,b,c])</code>	<code>@SArray([a,b,c])</code>
<code>MakeSparseArray(A::<b>SparseMatrixCSC</b>)</code>	<code>SparseMatrixCSC(A.col...)</code>
<code>MakeSparseArray(v::<b>SparseVector</b>)</code>	<code>SparseVector(...)</code>
<code>SpawnFetch([f(a),g(b)], combine)</code>	<code>combine(fetch.([@spawn(f(a)),@spawn(g(b))]))</code>
<code>LiteralExpr(expr)</code>	<code>expr</code>

Table 4.3: Term and its corresponding Julia expression after calling `toexpr`

Note that in the examples above, `a`, `b`, `x`, `y`, `w`, `z`, are symbols, for example, those constructed by the `@variables` macro. `A` and `v` are a sparse matrix and a sparse vector, respectively, with symbolic expressions as non-zero values.

As an escape hatch, to allow expressing arbitrary Julia expressions, Symbolics provides `LiteralExpr(expr)` term which wraps a native Julia expression. The Julia expression may contain Symbolics terms interpolated within them, in which case Symbolics finds them and turns them into Julia expressions.

#### 4.4.1 `build_function`

Symbolics provides a convenience function `build_function(exprs, args...)` to construct numerical functions from symbolic expressions. It takes a vector of expressions `exprs`, and a sequence of argument symbols `args`, and returns two Julia expressions: both are Julia function expressions. The first one computes the symbolic expressions in `exprs` on the input arguments, and returns an array (out-of-place); the second one takes an extra first argument which is the output array, and fills the output with the computed results (in-place).

#### 4.4.2 Staged Compilation

We use the library `RuntimeGeneratedFunctions` [71] to compile functions at runtime and execute them on the fly. Given a function expression, `RuntimeGeneratedFunctions` creates a type unique to that expression. Function application on this type is a generated function (see Section 3.2.4) which splices in the original expression as the body. Hence you can think of it as a generated function parameterized by its own body.

The limitation of this approach is that, the functions may not have closures in their body. However it is possible to get around this in most cases by rewriting the closures in the body as other `RuntimeGeneratedFunctions` and interpolating those into the function expression instead.

### 4.5 Symbolic arrays

Symbolics supports symbols that represent arrays of known size of any dimension. Note that this is not an array of symbols: a  $10000 \times 10000$  matrix can be represented as a single symbol

and not an array of a 100 million different scalar symbols. Operations on array symbols yield symbolic expressions. We implemented the same operations as Julia's Base library does on regular Julia arrays. Some example invocations are as follows:

```
julia> @variables X[1:10, 1:5] y[1:5] b[1:10];
```

```
julia> X*y
```

```
(X*y)[1:10]
```

```
julia> prod(X, dims=1)
```

```
prod(X, dims=1)[1, 1:5]
```

```
julia> exp.(X .* y')
```

```
(broadcast(exp, broadcast(*, X, adjoint(y))))[1:10,1:5]
```

Here  $X$  is a matrix of size  $10 \times 5$ ,  $y$  and  $b$  are vectors of lengths 5 and 10 respectively.

Internally, Symbolics encodes array operations with a modified Einstein-summation notation [25] [13]. Our notation is general enough to represent N-dimensional array operations such as map, broadcast, reduce (along any dimension(s)), mapreduce (combines the former two in one operator), slicing. A set of example Julia expressions and the resulting underlying storage is given in the table 4.1.

```
julia> Symbolics.show_arrayop[] = true; # printing flag to show array notation
```

```
julia> X*y
```

```
@arrayop(_[i] := X[i, k]*y[k])
```

```
julia> prod(X, dims=1)
```

```
@arrayop(_[1,j] := X[i, j]) (*)
```

```
julia> exp.(X .* y')
@arrayop(_[i,j] := exp(
(@arrayop(_[i,j] := X[i, j])*
 @arrayop(_[1,i] := y[i]))[1, j]))[i, j]))
```

Some features of this notation are:

- It does not make a distinction between up and down indices, any index left out from the output is assumed to be contracted.
- It allows using a reducer function that is not  $+$  to reduce contracted dimensions, this means we can represent arbitrary reduce operations. For example, `@arrayop (j,) A[i, j] (*)` is the same as `dropdims(reduce(*, A, dims=1))`.
- It allows reducing a dimension to have the size 1, rather than completely deleting it. For example, `@arrayop (1, j,) A[i, j] (*)` is the same as `reduce(*, A, dims=1)`.
- We can offset indices on the right hand side to look at neighboring elements. For example `@arrayop (i,j) A[i+1, j] + A[i, j+1] + A[i, j-1] + A[i-1, j]` will sum the elements in the cardinal directions of `A[i, j]`. The resulting array will be of size `size(A) .- 2`, since it has to exclude the calculation at the boundaries of `A` which result in out-of-bounds access.
- We can bound the range of indices that applies to an index symbol, resulting in the ability to represent slicing of arrays. For example, `@arrayop (i, j) A[i, j] where (j in 1:10)` is equivalent to `A[i, 1:10]`.

The benefits of encoding the Einstein summation notation are:

- Shape checking and propagation can be implemented in one central place to work on a generic expression, as opposed to separately for every operation supported.

- Code generation for array operations can take advantage of this notation to generate loops as well as optimize those loops. It’s possible to call out to tensor notation compilers such as Tullio.jl [1]. These compilers can generate code that executes on GPUs as well as code perform automatic differentiation on these expressions. Symbolics.jl currently does not support writing to Tullio because Tullio expressions create closures which are not currently supported in Julia’s generated functions.
- Symbolically computing all or some elements of an array expression, called “scalarization”, can be performed by substituting the required indices in the Einstein summation notation, and simplifying the resulting expression.

Code fragment	Symbolics Einsum Notation	Reducer
<code>map(f,A[1:10,:])</code>	<code>out[i,j] := f(A[i,j]) (i in 1:10)</code>	
<code>A'</code>	<code>out[i,j] := conj(A[j,i])</code>	
<code>A*b</code>	<code>out[i] := A[i,k]*b[k]</code>	+
<code>A*B</code>	<code>out[i,j] := A[i,k]*B[k,j]</code>	+
<code>p=prod(A, dims=2)</code>	<code>out[i,1] := A[i,j]</code>	*
<code>A .+ p</code>	<code>out[i,j] := A[i,j]+p[i,1]</code>	

Table 4.4: Array operations and their corresponding encoding in our notation. The reducer column shows the reduction function used. An empty reducer entry means, no reducer function is necessary since no dimension is contracted.

### 4.5.1 ArrayMaker: nesting arrays

ArrayMaker can be used to construct an array that is made of blocks of subarrays. ArrayMaker is used to implement `hcat`, `vcats` concatenation functions. It also plays a crucial role in the making of the `MethodOfLines.jl` package for partial differential equation discretizations. Which require the construction of highly structured matrices with interesting boundary conditions.

ArrayMaker is constructed with a shape and a sequence of slices `ArrayMaker(shape, sequence)`. The shape is the shape of the array (in julia shape is called `axes` and is a tuple of index ranges representing the range of indices of the output array (usually starting from 1) along every dimension of the array); the sequence is a vector of pairs: each pair contains an index set (an index into a view of the array) and the corresponding right-hand-side array which must fill the indices in the view.

The `@makearray` macro can be used to create a new Julia variable which points to an ArrayMaker object constructed from the body of the macro call. The body is a block where every statement is a pair (`=>`) where the left hand side is the index range and the right hand side is the array to be broadcasted into that index range.

```
julia> @makearray y[1:5, 1:5] begin
    y[2:end-1, 1:end] => @arrayop (i,j) x[i-1, j] + x[i+1, j]
    y[1, 1:end]      => @arrayop (j,) (x[1, j]+x[2, j])/2
    y[end, 1:end]    => @arrayop (j,) (x[end-1, j]+x[end-2, j])/2
end
(ArrayMaker((1:5, 1:5), begin
    (2:4, 1:5) => @arrayop(_[i,j] := x[-1 + i, j] + x[1 + i, j])
    (2:4, 1:5) => @arrayop(_[i,j] := x[-1 + i, j] + x[1 + i, j])
    (1, 1:5) => @arrayop(_[j] := (x[1, j] + x[2, j]) / 2)
    (5, 1:5) => @arrayop(_[j] := (x[4, j] + x[3, j]) / 2)
end))[1:5,1:5]
```

Note that `y` is an array symbolic expression just like any other. And can be combined with other array operations. `Symbolics.scalarize` function turns a symbolic array into an array of symbolic expressions by forcing elementwise evaluation.

```
julia> Symbolics.scalarize((y + y')[2,2])
2x[1, 2] + 2x[3, 2]
julia> Symbolics.scalarize((y+y')[2,3])
x[1, 3] + x[2, 2] + x[3, 3] + x[4, 2]
```



The `ArrayMaker` functionality is used in `MethodOfLines.jl` package for PDE discretizations. It becomes the perfect tool for the job of composing arrays that are highly structured, with the composability far better than composing loops as Julia expressions.

## 4.6 Primitive registration

Users can create or extend existing Julia functions with Symbolic methods by a convenience macro called `@register_symbolic`.

For example,

```
@register_symbolic hypot(x::Number, y::Number)::Real
```

Adds the methods:

```
hypot(x::Symbolic{Number}, y::Symbolic{Number})
```

```
hypot(x::Number, y::Symbolic{Number})
```

```
hypot(x::Symbolic{Number}, y::Number)
```

Which all return `Term(hypot, [x, y], type=Real)`.

Array function registration uses the `@register_array_symbolic` macro, where one can also designate how the output array size is computed.

```
@register_array_symbolic vandermonde(x::AbstractVector) begin
    size=(length(x), length(x))
end
```

## 4.7 Simplification

We use the rule-based rewriting system described above to implement the `simplify` routine. In practice, while tracing code symbolically, a lot of simplification occurs due to the `Add`, `Mul` and `Div` types we described at the beginning of this chapter. Simplification as a separate step is sometimes necessary.

Our simplification routine uses 80 rules which are sequenced using the rewriter combinators described in Section 4.3. We manually arranged the rules to make sure that the normalizing rules are applied at the beginning. The default simplifier, and the simplifier it calls on Number expressions look like this:

```

function default_simplifier(; kw...)
  IfElse(has_trig_exp,
    Postwalk(IfElse(x->symtype(x) <: Number,
      Chain((number_simplifier(),
        trig_exp_simplifier()))),
      If(x->symtype(x) <: Bool,
        bool_simplifier()))
    ; kw...),
  Postwalk(Chain((If(x->symtype(x) <: Number,
    number_simplifier()),
    If(x->symtype(x) <: Bool,
    bool_simplifier()))))
    ; kw...))

end

function number_simplifier()
  rule_tree = [If(istree, Chain(ASSORTED_RULES)),
    If(x -> !isadd(x) && is_operation(+)(x),
      Chain(CANONICALIZE_PLUS)),
    If(is_operation(+), Chain(PLUS_DISTRIBUTE)),
    If(x -> !ismul(x) && is_operation(*) (x),
      Chain(CANONICALIZE_TIMES)),
    If(is_operation(*), MUL_DISTRIBUTE),
    If(x -> !ispow(x) && is_operation(^)(x),

```

```

Chain(CANONICALIZE_POW)),
If(is_operation(^), Chain(POW_RULES)),
] |> Chain

rule_tree
end

```

`isadd`, `ismul` branches make sure that redundant rules are not applied on `Add` and `Mul` which are already simplified.

In the future a performant e-graph based rewriter can replace this rewriter, allowing bi-directional rules and better searches for simplified expressions. Simplification is one of the most vaguely described, yet often debated topics in symbolic computing. See “Algebraic simplification: a guide to the perplexed” by Joel Moses [65] for a discussion on different schools of algebraic simplification. With e-graphs it would be possible for the user to provide custom cost functions to be minimized on the expression, essentially parameterizing the notion of simplicity.

## 4.8 Algebra and calculus functions

- **Differential calculus:** Users can represent differentials in expressions with the syntax `Differential(t)(x(t))` which means  $\frac{dx(t)}{dt}$ . It is often convenient to define a short-form constant `const D = Differential(t)` for the differentials with respect to the independent variables in their system, then one can write `D(x(t))`, `D(D(x(t)))`, etc. The APIs `Symbolics.gradient`, `Symbolics.jacobian`, `Symbolics.hessian` compute the gradient, Jacobian, and Hessian respectively.
- **Sparsity computation:** we provide functions to obtain sparsity of Jacobians and Hessians via `Symbolics.jacobian_sparsity` and `Symbolics.hessian_sparsity` respectively. Jacobian sparsity is computed in the fashion described in the example in Section 1.6.2. Hessian sparsity is computed by rewriting expressions into the degree 2 polynomial

which has the equivalent Hessian sparsity, and repeating the process until a single at most degree 2 polynomial is left behind. The Hessian sparsity of the final expression is easy to compute and represents the Hessian sparsity of the original expression. A few example Hessian sparsity matrices and their equivalent degree-2 polynomials are shown in Table 4.5.

- **Simplification of polynomial fractions:** We provide the function `simplify_fractions` which takes a fraction expression, turns the numerator and denominator into polynomials (see Section 4.1.3 for more on Polynomials) and cancels the GCD from them.
- **Expansion:** `Symbolics.expand` expands an expression by distributing multiplication of polynomial subexpressions.
- **Groebner bases:** `Symbolics.groebner_basis` takes a vector of polynomials and returns the groebner basis for them.
- **Semi-linear form:** `Symbolics.semilinear_form(exprs, vars)` expresses `exprs` in the form  $A * vars + f(vars)$  and returns `A` and `f(vars)`. This is beneficial for compilation of `exprs`. Since, the matrix `A * vars` will be faster to compile, and sometimes already parallel, than when the output of the same is written out statement-by-statement.

Symbolics community is working on the following functionality:

- Symbolic integration (Xinghui Hu)
- Matrix derivatives (Roni Edwin)
- Factorization and root finding (Yassin ElBedwihy)

The future development of Symbolics will focus on high-performance symbolic root finding, expansive algorithm selection for a Gröbner basis.

Symbolics and its ecosystem are quickly becoming a common foundation for the construction of next-generation Domain-Specific Languages in Julia, all of which will benefit from the coming advancements as the system grows.







code fragment	polynomial	sparsity
<code>deg2rad(x[1])</code>	$x_1$	
<code>log(x[1])</code>	$x_1^2$	
<code>x[1] + x[4]</code>	$x_1 + x_4$	
<code>x[1] * x[4]</code>	$x_1x_4$	
<code>q = x[1]/x[4]</code>	$x_1x_4^2$	
<code>asin(q)*x[3]</code>	$(x_1^2x_4^2)x_3$	

Table 4.5: Hessian sparsity construction for an expression in `x[1:4]`. The  $4 \times 4$  sparsity pattern for each intermediate value is shown. The provenance polynomial has the same hessian sparsity pattern.

## 4.9 Contributions

The ideas in Symbolics have existed in various systems. However, Symbolics.jl is the first symbolic-numeric system written in and for a host language that is also a numerical language. This not only allows code generation which is the focus of previous work, but also allows symbolic analysis and transformation of code written in a useful subset of the language. This advances the state of the art in the following ways:

- The transformations are accessible to the scientific user, do not require any knowledge of compiler development, and is high-level as one is dealing with expressions that support arithmetic operators supplemented by the power of a modern computer algebra system.
- By the syntactic convenience of `@register_symbolic`, which adds symbolic methods to existing generic functions, the whole ecosystem of Julia’s scientific packages can be

called from a symbolic expression. The numerical code generated from it can naturally call these numerically.

- The symbolic representation, manipulation, and code generation capabilities are extensible by an ecosystem of packages whose power increases by a network effect, this prevents the siloing in symbolic-numeric systems we describe in Chapter 2.

We note that out of the systems we surveyed, only Symbolics, JAX [10], MATLAB’s Symbolic Math Toolbox (SMT) [78] and Scmutils [34] can trace numerical code written in the host language faithfully. JAX does this by providing a drop-in replacement to NumPy, while Symbolics uses symbols of different types. JAX is not explicitly Symbolic, and we discuss more about it in Section 4.10.5. MATLAB SMT is embedded within MATLAB, and supports array symbols that match the operator behavior of MATLAB. SMT does not expose a pattern-based rewrite language. MATLAB itself is not a good language to implement numerical algorithms since it is interpreted and lacks fast loops. Symbols of custom types, and hence custom arithmetic methods on them are not supported by this system. Scmutils is written in a variation on Scheme that supports generic functions (multi-methods) based on predicate dispatch. It works by adding arithmetic methods directly on quoted Scheme expressions. Since Scheme expressions are untyped, this forces expressions to have a single meaning—i.e., as numeric calculations. Custom symbol types can be defined, but Scheme expressions containing them still remain untyped and hence only have the numeric meaning. In contrast, in Symbolics, both variables and expressions can represent a trace of any Julia type and their behavior on functions can vary.

## 4.10 Comparisons

We present a comparison of features in Table 4.6, and describe some details on the systems we surveyed in the next several paragraphs. Note at the outset that apart from Symbolics, MATLAB-SMT, Scmutils, and JAX, other systems are not able to faithfully trace and reproduce numerical code.

Package	↓	Host language	Numerically fast	Custom symbols	Generic functions	O(1) space Arrays	Compiles to
Feature	→						
Mathematica		C <sup>1</sup>	✓	✗	✗	✗	C/Fortran
SymPy		Python	✓	✓	✓	✓	Python and C
Maxima		Common Lisp	✓	✓	✓	✓	Fortran
MATLAB-SMT			✓	✗	✗	✓	C, Fortran, MATLAB
Magma		C	✓	✗	✗	✗	–
Reduce		Lisp (PSL)	✓	✗	✗	✗	Fortran
AXIOM		Lisp	✓	✓	✓	✓	Fortran
JAX		Python	✓	✓	✗	✗	Python and XLA
scmutils		Scheme	✓	✓	✓	✗	Scheme
Modelica		C	✓	✗	✗	✓	C
Maple		Lisp	✓	✗	✗	✓	C and Fortran
Symbolics		Julia	✓	✓	✓	✓	Julia

Table 4.6: Table of features supported by various symbolic systems. The orange ✓ mark represents that the system is numerically fast by calling out to code written in a different, low-level language because the host language is itself slow. Symbolics, MATLAB-SMT, JAX and Scmutils are in a position to be able to treat numerical code as Symbolic expressions.

<sup>1</sup> “extension of C which supports certain memory management and object-oriented features” [87]

**Mathematica** is written in “an extension of C which supports certain memory management and object-oriented features” [87]. Mathematica does not have support for symbolic arrays although in the assumption system you can specify that a variable is a member of a matrix vector space. However, this does not prevent a matrix variable from being commuted in multiplication, for example. Mathematica has a compiler which is can be explicitly invoked or is automatically invoked when it is valid and beneficial to do so, e.g. in higher-order functions like `Map` and `Table` [48]. Mathematica is not a good language to implement numerical algorithms since it lacks a transparent memory layout and efficient for loops. There is no explicit support for generic programming in Mathematica. This means it is not possible for the user to create custom operator behavior.

**SymPy** [63] is written in Python, and has support for symbolic matrices. It can compile symbolics to Python functions or to C and invoke it from Python using the `autowrap` function [82]. However, as we discuss in 4.10.3, because of lack of consistency with numerical libraries, the codegen can be unintuitive to use. The code generated in C also has the limitations of the C language and breaks the closure property of symbolic-numeric programming, in that the C code cannot call back a library code in Python if it needs to. SymPy has extensive support for tensors and tensor symbols, including `Matrix` and `Vector` types. It can also generate efficient tensor contraction code in C. Symbolics’ implementation is closest to that of SymPy, and hence we consider in greater detail in the coming sections.

**Maxima** [40] is written in Common Lisp, it supports arrays up to 5 dimensions and has a `gentran` function [79] that can—

generate FORTRAN, C, and RATFOR code from Maxima language code. `Gen-  
tran` can translate mathematical expressions, iteration loops, conditional branch-  
ing statements, data type information, function definitions, matrices and arrays,  
and more.

**Maple** [60] by Waterloo Maple Software is a language for symbolic, numerical and graphical applications. It has a `Fortran` command to generate Fortran code from Maple expressions. Maple supports class-based object-oriented programming and operator overloading.



[21] Method dispatch is based on all arguments, and proceeds left to right. The first object of a class with the method gets called.

**Magma** [9] is an abstract algebra language designed for computations in algebra, number theory, algebraic geometry, and algebraic combinatorics. It supports imperative and functional styles of programming and has a type system suitable for abstract algebra. It is possible for users to evaluate strings of Magma code at runtime using `eval` [20]. It also allows creation of numerical arrays performing linear algebra operations on them. Magma does not concern itself with emitting C or Fortran code.

**MATLAB Symbolic Math Toolkit** is a computer algebra system embedded in MATLAB. This software supports Matrix symbols, and has the same operators as MATLAB. Symbolic Math Toolbox provides functions for generating MATLAB functions, Simulink Function blocks, and equations based on the Simscape language. MATLAB supports class-based object-oriented programming [77]. However it is not possible to create custom symbols with custom operator behavior.

**Reduce** [44] is a comprehensive symbolic language. In Reduce, there are no generic functions (although they do use the phrase generic function to refer to function symbols), it supports matrix and tensor variables, produces Fortran code, has no facility for custom types or classes.

**AXIOM** describes itself as “a self-contained toolbox designed to meet your scientific programming needs, from symbolics, to numerics, to graphics” [51]. Generic programming abilities in AXIOM are provided by hierarchy of type classes to manipulate and propagate domains over values. It supports array symbols and tensor operations. It is a symbolically focused environment, and hence does not try to also be a language which is fast for numerical code. It can generate FORTRAN code from symbolic expressions.

**Scmutils** is a Scmutils is a package developed by Gerald J. Sussman, Jack Wisdom, Chris Hanson et. al. [34]. It is implemented in the Scheme language, and works by adding arithmetic methods to the default homoiconic expressions of Scheme. We described the limitation of this in the previous section. Scmutils does not support array symbols, and can

compile to machine code using the existing facilities to do so in Scheme.

In the following subsections, we consider the advantages of using Julia as a host language for a symbolic-numeric system vs. Python as a host language, taking the example of SymPy+numpy. Our goal is to show that 1) we solve the two-language problem in symbolic-numeric programming (see Section 1.4); 2) the same benefits of multiple dispatch that makes organizing numerical code great, also makes organizing symbolic ecosystem just as great. After this, we make some remarks on JAX and compare it to Symbolics.

### 4.10.1 Modularity and extensibility

The design of terms in Symbolics is similar in spirit to that in SymPy. SymPy is written with class-based object-oriented programming. A superclass `Expr` assumes the existence of an `.args` and `.func` properties, whereas in our case we have the `head` and `children` accessor generic functions which serve the same purpose of querying an expression. Note that the notion of a symbolic type (`sympy`), is very informal in SymPy (in the listing below, we see them appear as string tags) whereas in Symbolics, `sympy`s map to Julia types such as `Real` and `AbstractArray` – we keep close to the numerical language, while also utilizing the generality of abstract types.

Instead of class-based inheritance, we use the minimal term interface described in Section 3.3.1. SymPy must use single-dispatch, which comes with code organization problems exacerbated in scientific computing as described in Section 3.2.1. For example, if a `BandedMatrices` module wanted to create a symbolic `BandedMatrix` method, and wanted to set up a way to multiply it with existing symbolic matrices, for instance `BasicMatrix`, it would have to somehow inform the existing `BasicMatrix` class about this new type (at least for the case `A * B` where `A` is an instance of `BasicMatrix` and `B` is an instance of `BandedMatrix`). In SymPy, the `__mul__` method of `BasicMatrix` executes the following piece of dispatch code:

```
self, other, T = _unify_with_other(self, other)

if T == "possible_scalar":
```

```

    try:
        return self._eval_scalar_mul(other)
    except TypeError:
        return NotImplemented
elif T == "is_matrix":
    if self.shape[1] != other.shape[0]:
        raise ShapeError(f"Matrix size mismatch: {self.shape} * {other.shape}.")
    m = self._eval_matrix_mul(other)
    ...
    return m
else:
    return NotImplemented

```

Here `_unify_with_other` is used as an imperfect substitute for multiple dispatch. It returns a tag called `T` which tells what kind of Symbolic expression to produce as output, and a runtime branch then performs the right action. This function also converts `other` to be in a form that `_eval_matrix_mul` down the line.

We believe it is much more natural that this multiplication be addressed by the `BandedMatrices` module instead, and Julia's multiple-dispatch makes this possible, and avoids the ad-hoc dispatch as in SymPy.

To reinforce this point, consider that with Symbolics we can easily define the term interface on the default Julia expression type `Expr` which is obtained on quoting Julia code. For example, `Metatheory.jl` [14]'s E-graph rewriting works both on Julia expressions and Symbolics expressions by way of this interface. This would not be possible in a language with static dispatch, say, C++. Mathematica does is not object-oriented, and does not have an interface for creating new kinds of expressions.

An important benefit of using Julia as the implementation language is its performance. Julia is performant in comparison to Python even in applications heavy on dynamic dispatch

(dispatch that cannot be inferred at compile time) such as symbolic computing. When one real-world robotics application was ported from SymPy (being called via the Julia wrapper) to Symbolics, we found that symbolically tracing into the mass matrix computation of a rigid body dynamical system with 7 degrees of freedom took 3.721925 seconds in Symbolics, while SymPy takes 16.333147 seconds resulting in a 4.4x speed up. <sup>2</sup>

## 4.10.2 Symbolic tracing of numerical code

Symbolic tracing in Julia works based on generic function calls. For example there is only one function `*`, and `*` always refers to this function (except in degenerate cases like bootstrapping of the language and deliberate attempts to break something). In a plain julia session, `*` has over 200 methods. To have these many methods, creators of many types must have agreed on its consistent meaning: on numbers it is multiplication, on strings it is concatenation, on matrices it is matrix multiply, on a matrix and a vector it is a matrix-vector multiply, and so on.

In Python `*` is allowed to be opinionated, since there is a tendency to think of the `__mul__` method (which `*` desugars to) as a property of whichever class is defining it. In `numpy`, `*` works on matrices of matching dimensions and performs an element-wise multiplication. In `sympy` it performs a matrix multiplication.

```
from sympy import Matrix, MatrixSymbol
import numpy

>>> M = Matrix([[1, 7], [-2, 3]]);
>>> N = Matrix([[0, 1], [1, 0]]);
>>> M * N
      Matrix([
          [7,  1],
          [3, -2]])
```

---

<sup>2</sup>Reproducible code at <https://github.com/JuliaSymbolics/SymbolicUtils.jl/pull/254>

```

>>> nM = numpy.array([[1, 7], [-2, 3]]);
>>> nN = numpy.array([[0, 1], [1, 0]]);
>>> nM * nN
array([[ 0,  7],
       [-2,  0]])

```

Now consider the trace of a simple function `mult` written meant to take matrix arguments in, say `numpy`:

```

>>> def mult(a, b): return a * b.T

>>> A = MatrixSymbol('A', 2,2)
>>> mult(A,A);
A*A.T

```

Let's say we trace this function as `mult(A,A)`, with matrix symbol `A` from `SymPy`, the resulting expression will not mean the same thing as a fragment of `numpy` code which is only one possible instance of "numerical code" in Python. In the latter case, `a*b.T` is commutative, and in the former case it is not! How can one implement term manipulation on this trace and be sure that the resulting expression can be turned into numerical code that uses `numpy`? It's possible but the amount of unwanted complexity is unfortunately a lot.

### 4.10.3 Numerical code generation

We can continue with the same example and see how the differing meanings of `*` makes the semantics of code generation awkward:

```

>>> f=sympy.lambdify([A], mult(A, A), 'numpy')
>>> f(nM)

```

```

array([[50, 19],
       [19, 13]])

>>> mult(nM, nM)
array([[ 1, -14],
       [-14,  9]])

```

The regenerated code is not faithful to the original.

When generating code in Julia, native non-symbolic values used within symbolic expressions simply get interpolated into the Julia expression (without a copy). This means expressions containing non-symbolic matrices, big integers etc, also behave faithfully as in the base language, and not according to the target language such as C. SymPy has this limitation in its `autowrap` utility which emits and compiles C code for symbolic expressions.

#### 4.10.4 Function registration

When we register functions as symbolic-level primitives, any calls to that function with symbols becomes a node in the symbolic trace. Here the head of the node is simply the generic function object itself!

Note that this is different from defining symbols which behave as functions. Symbols create a new binding while registering a Julia function simply adds the necessary methods to accept one or more arguments to it as symbolic.

In SymPy, one can create function symbols either as `Function('f')`, or by creating a class that extends the `Function` class. Note that this is the same as creating a function symbol. In python there can be no notion of function registration, for example, it's not possible to make the following work:

```

A = MatrixSymbol('A', 2,2)
numpy.matmul(nM, A)

```

When a function symbol such as `Function('matmul')` is lambdified, it simply writes the identifier 'matmul' into the output code, and it's the job of the user then to make sure the identifier function `matmul` is present in the environment and points to `numpy.matmul` (for example, by doing `from numpy import matmul`).

#### 4.10.5 Notes on JAX

JAX [10] uses symbolic tracing to optimize, differentiate, schedule numpy-like array code. It is designed to be drop-in replacement of NumPy [43] in Python. The trace is a custom data structure called `Jaxpr` which is similar to a single-static-assignment intermediate representation Julia produces after type-inference (See Section 3.2.2 for `@code_typed` example). The main difference in JAX from type inference of Julia is that:

- All functions are assumed to have no side effects.
- Tracing of functions with JAX arrays will cause the traced code to propagate type as well as size information whereas in Julia, sizes are not considered known at compile time.

However, these assumptions and behavior are equivalent to the assumptions that Symbolics makes during the tracing process with symbols.

All code optimizations are written as interpreters of `Jaxprs`, and can be composed. Finally a `jit` function can turn the `Jaxpr` into fast code, specifically utilizing the XLA [72] instruction set.

The subset of NumPy/Python JAX compiles is similar to what Symbolics is interested in tracing (and hence compiling).

```
import jax.numpy as jnp
```

```
def Dense(x, w, b):  
    return jnp.matmul(x, w) + b
```

```
jax_fn = jit(Dense)
```

We envision the future evolution of symbolic-numeric software as a fusion of symbolic tracing and interpreter-based approaches. An example of this hybrid paradigm is showcased in the `ReversePropagation.jl` [74] package, which harnesses Symbolics to construct a reverse-mode interval propagator, as well as reverse-mode automatic differentiator. This package closely mirrors the structure of `Jaxpr`, demonstrating the potential of this idea.

## 4.11 Limitations of Symbolics

**Branches:** A big limitation of Symbolics is that symbolic expressions appearing in branch conditions immediately error since only booleans are allowed as branch conditions. A tracing mechanism based on abstract interpretation [16] can detect these cases and decide to rewrite the trace to represent branches symbolically.

**Need for wrapper types:** Since Julia does not support multiple inheritance, to integrate seamlessly with Julia’s type system, we define expression types as subtypes of the abstract `Real` and `AbstractArray` types, respectively. Consequently, our system is compatible only with functions and types that can accept inputs as general as these abstract types.

In Julia, there is often an incentive to specify the most precise type possible to achieve an efficient memory layout and optimized performance. This is especially true when it comes to types of fields in structs. If the type is specialized by a type parameter, then it can hold symbolic expressions, otherwise it cannot. Thankfully, critical types like `Complex{T}` are parameterized based on the types of their fields, which allows symbolic expressions to be part of them.

**Compilation:** Julia effectively balances specialization and abstraction in its code compilation process. However, code generated using the Symbolics library can become excessively lengthy, which often results in significantly prolonged compile times. Techniques like common subexpression elimination help to an extent, but some kind of compression strategy



which can automatically find larger abstractions (discovering libraries of functions to make programs shorter) could become useful in the future.



# Chapter 5

## Case studies

In this chapter we present some case studies of using Symbolics’ Symbolic-numerical abilities to solve problems, and we demonstrate the effectiveness and power of Symbolics. The first use case of ModelingToolkit shows the effectiveness of Symbolics as a lingua-franca for Modeling and Simulation software in the SciML (“Scientific machine learning”) field. With ModelingToolkit as the base, several scientific modeling frameworks have been written all utilizing the symbolic-numeric programming style. The second use case on Convex optimization exemplifies the productivity advantage of Symbolics over implementing the same problem using plain multiple-dispatch this also provides a look into one subsystem within SciML that benefits from Symbolics. Next we consider Catalyst.jl a chemical reaction network framework, where Symbolics is used to build a domain-specific compiler for chemical reaction network simulations. Finally, we consider ReversePropagation.jl, a project for reverse mode interval propagation and reverse mode differentiation showcasing the composable nature of Symbolics when used for transforming programs.

### 5.1 SciML: Scientific Machine Learning

SciML is an ecosystem of Julia projects aiming to bring together high-quality implementation of scientific modeling under a consistent umbrella abstraction that creates a sum greater than

its parts. Symbolics acts as a symbolic *lingua-franca* in SciML. Before we get to the symbolic aspects of the SciML story, let's look at the numerical part of SciML.

### 5.1.1 Numerical building blocks

The numerical building blocks of SciML are listed below. We also show an example invocation for each item. Note the generic nature of the `solve` function, it takes 2 arguments: a problem and a solver type, and returns a solution, and the problem type varies within an across each project below. The result of `solve` is a solution array-like object.

1. **LinearSolve.jl**: Unified high-performance linear solvers.

*Problem*: solve for  $b$  in  $Au = b$ .  $A$  can be a matrix, or the problem can be expressed as a numerical function  $A(u, p) = b$ , where  $p$  are parameters.

```
prob = LinearProblem(A, b)
sol = solve(prob, KrylovJL_GMRES())
```

2. **NonlinearSolve.jl**: Unified non-linear solver.

*Problem*: find value  $u$  such that  $f(u) = 0$ .  $f$  can be specified by a numerical function  $f(u, p)$  where  $p$  are parameters.

```
prob = NonlinearProblem(f, u0, p)
sol = solve(prob, TrustRegion())
```

3. **DifferentialEquations.jl** [69] [70]: Unified interface for *all* differential equations.

*Problem*: given a differential equation problem, evolve its solution from a start state  $u_0$  over some timespan. Notice the generality of this definition—the problem itself can be polymorphic.

```
prob = ODEProblem(f, u0, tspan)
sol = solve(prob, Tsit5())
```

Currently, these problem types (and one or more solvers for them) are supported: Discrete equations (function maps, discrete stochastic (Gillespie/Markov) simulations), Ordinary differential equations (ODEs), Split and Partitioned ODEs (Symplectic integrators, IMEX Methods), Stochastic ordinary differential equations (SODEs or SDEs), Stochastic differential-algebraic equations (SDAEs), Random differential equations (RODEs or RDEs), Differential algebraic equations (DAEs), Delay differential equations (DDEs), Neutral, and algebraic delay differential equations (NDDEs, RDDEs, and DDAEs), Stochastic delay differential equations (SDDEs), Experimental support for stochastic neutral, retarded, and algebraic delay differential equations (SNDDEs, SRDDEs, and SDDAEs), Mixed discrete and continuous equations (Hybrid Equations, Jump Diffusions), (Stochastic) partial differential equations ((S)PDEs) (with both finite difference and finite element methods).

4. **Optimization.jl** [22]: Unified solutions of optimization problems.

*Problem:*

$$\text{minimize } f(u, p)$$

$$\text{subject to } g(u, p) \leq 0, h(u, p) = 0$$

```
prob = OptimizationProblem(f, p)
sol = solve(prob, Optimisers.ADAM(0.05))
```

5. **Integrals.jl** [69]: Unified quadrature interface

*Problem:*

$$\int_{lb}^{ub} f(t, p) dt$$

```
f(x, p) = sin(x * p)
prob = IntegralProblem(f, (-2, 5), 1.7) # over x in [-2,5]
sol = solve(prob, QuadGKJL())
```

### 5.1.2 Symbolic-numeric modeling language

ModelingToolkit (MTK) is a symbolic equation-based modeling library built on top of Symbolics. It takes equations written with Symbolics expressions, and allows the creation of the Problem types from across the SciML ecosystem that described in the previous subsection.

```
using ModelingToolkit, OrdinaryDiffEq
@parameters t σ ρ β
@variables x(t) y(t) z(t)
D = Differential(t)
eqs = [D(x) ~ σ*(y-x), D(y) ~ x*(ρ-z)-y, D(z) ~ x*y - β*z]
@named lorenz1 = ODESystem(eqs)
u0 = [x => 1.0, y => 0.0, z => 0.0]
p = [σ => 10.0, ρ => 28.0, β => 8/3]
tspan = (0.0, 100.0)
prob = ODEProblem(lorenz1, u0, tspan, p)
sol = solve(prob, Tsit5())
```

ModelingToolkit produces the following code to be used by `ODEProblem`, using Symbolics's `build_function` function (Section 4.4.1):

```
(out, arg1, arg2, t) = begin
    out[1] = (-1 * arg1[1] + arg1[2]) * arg2[1]
    out[2] = -1 * arg1[2] + arg1[1] * (-1 * arg1[3] + arg2[2])
    out[3] = arg1[1] * arg1[2] + (-1 * arg1[3]) * arg2[3]
    nothing
end
```

which is then solved using a `Tsit5` [84] solver.

In this example we see that an `ODESystem` is converted into an `ODEProblem`. As one can imagine, this is not the only kind of equation system ModelingToolkit supports. There

are also: Stochastic differential(-algebraic) equations (`SDESystem`), Partial differential equations (`PDESystem`), Nonlinear solve equations (`NonlinearSystem`), Optimization problems (`OptimizationSystem`), Continuous-Time Markov Chains (`JumpSystem`), and Nonlinear Optimal Control (`ControlSystem`).

Turning Systems into Problems sometimes requires turning a System into another System first, to make it feasible to dispatch the problem to the right solver.

## System transformations

ModelingToolkit can provide a suite of system transformation functions that takes one system and turns it into a different, often more useful, system.

`PDESystems` are turned into `ODESystems` by various discretization techniques; Differential Algebraic Equations (DAEs) are transformed to eliminate variables, and do DAE index reduction.

A simple example of a system transformation is turning a system which connects many named subsystems, into a single flattened system:

```
@named lorenz2 = ODESystem(eqs)
@variables a; @parameters γ
connections = [0 ~ lorenz1.x + lorenz2.y + a*γ]
@named connected = ODESystem(connections,t,[a],[γ],systems=[lorenz1,lorenz2])
u0 = [lorenz1.x => 1.0,lorenz1.y => 0.0,lorenz1.z => 0.0,
      lorenz2.x => 0.0,lorenz2.y => 1.0,lorenz2.z => 0.0,
      a => 2.0]
p = [lorenz1.σ => 10.0,lorenz1.ρ => 28.0,lorenz1.β => 8/3,
     lorenz2.σ => 10.0,lorenz2.ρ => 28.0,lorenz2.β => 8/3,
     γ => 2.0]
tspan = (0.0,100.0)
prob = ODEProblem(connected,u0,tspan,p)
sol = solve(prob,Rodas4())
```

The flattening is automatically invoked in the call to `ODEProblem` in this case, but transformations can be explicit, and user-defined. Some explicit transformations provided by MTK are:

- **Structural simplification** (`structural_simplify`) removes the numerically-unnecessary modeling abstractions to solve a single differential equation.
- **DAE Index reduction** (`dae_index_lowering`) transforms a higher index DAE into an index-1 DAE via Pantelides algorithm [67] or its extensions [75].
- **Alias elimination** (`alias_elimination`) Elimination of singular equations which amount to  $0 = 0$ , while being able to recover the original variables [66].
- **Nonlinear tearing** reorders equations to reduce the size of the problem which requires a nonlinear solve [3].

These transformations can be chained together and applied to a system as necessary.

A user-defined transformation function can use the full power of Symbolics. For example, probabilistic robustness can be assessed via the trace of the Jacobian of an ODE [41]. This is a simple 7 lines of code, which appends the following equation onto the system:

```
D(trJ) ~ trJ*-tr(calculate_jacobian(sys))
```

Note that we use `<lhs> ~ <rhs>` to construct equations in Symbolics since `<lhs> = <rhs>` is reserved for assignment in Julia.

To perform this robustly, one can call in sequence `dae_index_lowering`, then `structural_simplify`, after that `liouville_transform(sys)` which could append the equation above.

Now we reproduce an example of structural simplification from [58].

### Example of structural simplification

A pendulum system can be specified, as follows, by 5 state variables,  $x$ ,  $y$ , the velocities  $v_x$  and  $v_y$ , and  $T$ , with parameters  $g$  for the acceleration due to gravity, and the fixed length of



the pendulum  $L$ :

$$x' = v_x \quad (5.1)$$

$$v'_x = T x \quad (5.2)$$

$$y' = v_y \quad (5.3)$$

$$v'_y = T y - g \quad (5.4)$$

$$0 = x^2 + y^2 - L^2 \quad (5.5)$$

We add a symbolic-tracing detour to the story by assuming a user has written down a pendulum drift function instead of symbolically specifying the equations, and is currently just using `DifferentialEquations.jl` to solve it:

```
function pendulum!(du, u, p, t)
    x, dx, y, dy, T = u
    g, L = p
    du[1] = dx; du[2] = T*x
    du[3] = dy; du[4] = T*y - g
    du[5] = x^2 + y^2 - L^2
end

pendulum_fun! = ODEFunction(pendulum!, mass_matrix=Diagonal([1,1,1,1,0]))
u0 = [1.0, 0, 0, 0, 0]; p = [9.8, 1]; tspan = (0, 10.0)
pendulum_prob = ODEProblem(pendulum_fun!, u0, tspan, p)
```

The Jacobian of the algebraic equation with respect to  $L$  is singular, making this an index-3 DAE. Since most DAE solvers require an index-1 DAE, a user may easily and inadvertently write such an “unsolvable” equation, leading to a non-convergent solving process and leaving the user to think the issue is a “bad solver”. For example, if the user attempts to solve this with the `Rodas4` method for DAEs, the solver almost immediately exits due to claims of instability.

`modelingtoolkitize` is a function that takes a drift function such as `pendulum!` and turns it into a ModelingToolkit system. An important benefit of `modelingtoolkitize` is that it also generates the Jacobian of the drift wherever necessary—and it can use the sparsity information described in 1.6.2 to do this.

Because the underlying problem is structural, we can use the `modelingtoolkitize` function to transform the user’s equations to MTK symbolic representations, and subsequently use `dae_index_lowering` to generate an index-1 form which is numerically solvable. The algebraic equation can then be replaced via a `rootfind` inside the right hand side system (done as part of the `ODAEProblem` construction) to arrive at a simple non-stiff ODE. These transformations are performed via:

```
traced_sys = modelingtoolkitize(pendulum_prob)
pendulum_sys = structural_simplify(dae_index_lowering(traced_sys))
prob = ODAEProblem(pendulum_sys, Pair[], tspan)
sol = solve(prob, Tsit5())
```

The method automatically transforms the user’s code to the form:

$$x' = v_x \tag{5.6}$$

$$v'_x = xT \tag{5.7}$$

$$y' = v_y \tag{5.8}$$

$$v'_y = yT - g \tag{5.9}$$

$$0 = 2(v_x^2 + v_y^2 + y(yT - g) + Tx^2) \tag{5.10}$$

which is mathematically equivalent but numerically more stable. The graph algorithm automatically figures out that only the fifth equation needs to be changed, and it needs to be differentiated twice for the numerical solution to be stable. This variant of the model is successfully solved with an explicit Runge-Kutta method for non-stiff ODEs [84], as shown in Figure 5-1. Such automated transformation features allows users of the numerical environ-

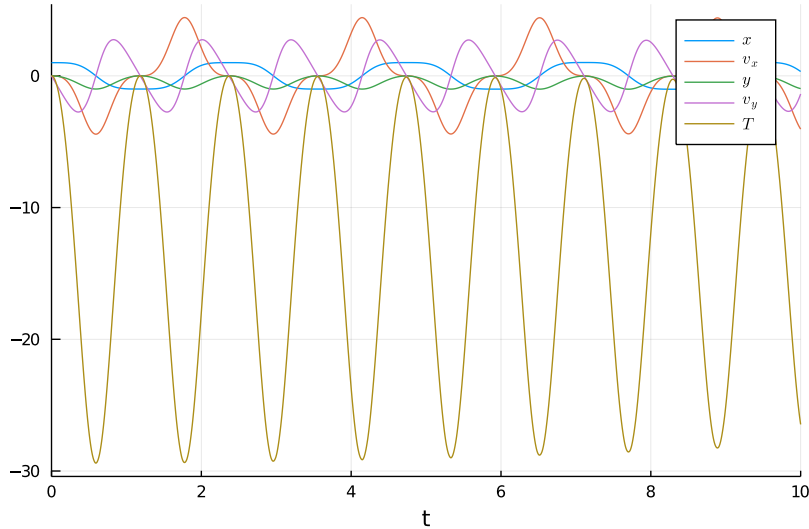


Figure 5-1: **Index-3 DAE model of a pendulum solved by an explicit Runge-Kutta method.** Pendulum model successfully solved by the Tsit5 [84] 5th order Runge-Kutta method [83] after performing index reduction via the Pantelides algorithm [67] and reducing the nonlinear system.

ment to make use of symbolic graph algorithms like Pantelides algorithm to nearly invisibly achieve better numerical stability. This abstract interpretation process thus enables the application of such discrete algorithms by the users of numerical functionality of standard programming languages.

In this paper, models reduced by MTK are shown to outperform the Dymola Modelica compiler on an HVAC model by 590x at 3% error.

### 5.1.3 Moving the methodology forward in PDEs

Solutions to PDEs are often written on a case-by-case basis. For example, it is common for users to write a fast loop to solve a single problem. Publications implement new methods on a small machine, and leave the fast implementation that runs on modern hardware to later engineering work. Good algorithms, and a high-quality fast implementation are both required in the future.

New methods in PDE solvers amount to a small improvement or mathematical trick

that applies to one part of the solver pipeline. By delineating every step of the process of going from a problem to its solution—reducing it into a computable form, turning that into a better computable form, finally choosing the right solver and parameters for that problem, generating code from symbolic information, we can allow these new methods to be actually implemented as a small change to this pipeline.

## 5.2 Convex optimization

Modern computational convex optimization has been significantly shaped by Disciplined Convex programming (DCP) [38]. From Grant, et. al [38]:

Disciplined convex programming is inspired by the practices of those who regularly study and use convex optimization in research and applications. They do not simply construct constraints and objective functions without advance regard for convexity; rather, they draw from a mental library of functions and sets whose convexity properties are already known, and combine and manipulate them in ways which convex analysis insures will produce convex results. When it proves necessary to determine the convexity properties of a new function or set from basic principles, that function or set is added to the mental library to be reused in other models.

Disciplined convex programming formalizes this strategy, and includes two key components:

1. An atom library: an extensible collection of functions and sets, or atoms, whose properties of curvature/shape (convex/concave/affine), monotonicity, and range are explicitly declared.
2. A convexity ruleset, drawn from basic principles of convex analysis, that governs how atoms, variables, parameters, and numeric values can be combined to produce convex results.

In DCP, optimization requires inferring whether a function is actually convex, its sign and monotonicity. This is essentially a symbolic analysis.

The application of this ruleset to programs lends itself to term-rewriting. Our implementation <https://github.com/Vaibhavadixit02/SymbolicAnalysis.jl> of DCP can then be leveraged in the Optimization.jl [22] framework to certify convexity. Optimization.jl is summarized briefly in Section 5.1.

### 5.2.1 Symbolics vs. multiple-dispatch for implementation

In the past, DCP has been implemented in Julia in the Convex.jl project [85] using Julia’s multiple-dispatch. Here the scheme of implementation is roughly:

- For every primitive function supported such as `exp`, encode the “trace” using a different type, e.g. `ExpAtom`, containing `children` that are the arguments to `exp`, and `size` which holds the output size of the expression.
- Define methods on generic functions `sign`, `monotonicity`, `curvature`, `evaluate`. These functions take an `ExpAtom` object and respectively return 1) the sign of `exp`, 2) the monotonicity, 3) the curvature, 4) the result of executing the node (e.g., `exp.(evaluate(node.children[1]))`). In all cases these functions may recursively call themselves on the children to make the final choice of output.

The implementation also needs to bookkeep and verify that the `Atoms` are composed in valid ways, for example:

```
mutable struct AdditionAtom <: AbstractExpr
    children::Vector{AbstractExpr}
    size::Tuple{Int,Int}

function AdditionAtom(x::AbstractExpr, y::AbstractExpr)
    # find the size of the expression = max of size of x and size of y
```

```

sz = if x.size == y.size
    x.size
elseif y.size == (1, 1)
    y = y * ones(x.size)
    x.size
elseif x.size == (1, 1)
    x = x * ones(y.size)
    y.size
else
    error(
        "[AdditionAtom] cannot add expressions of sizes $(x.size) and $(y.size)",
    )
end

# See if we're forming a sum of more than two terms and condense them
...
end

```

Note that, presumably to simplify implementation, they represent all values as matrices, including scalars which are  $1 \times 1$  matrices. In the code above, the branches for `y.size == (1, 1)` and `x.size == (1, 1)` are handling these edge cases. An unfortunate consequence of this is that `AbstractExpr` will fit into neither the `Number` type hierarchy, nor the `AbstractArray` type hierarchy of Julia.

In contrast, our implementation in `Symbolics` starts off at a much higher-level of abstraction. We already have symbolic expressions that can be scalar expressions, matrices and vectors, and the size checking and validation are taken care of by `Symbolics`.

We use the metadata system described in Section 3.3.2 to layer on top of the expressions metadata for sign, convexity, and monotonicity information. Not being tied to using multiple-dispatch to propagate rules, we can simply store the rules in a lookup table. For example, our code to propagate the sign looks like a rewriter pass on the expression. It starts at the

leaf nodes, and applies the first of a series of rules and propagates this information up the tree (Postwalk).

```
function propagate_sign(ex)
    rs = [@rule ~x::issym => hassign(~x) ? ~x : setsign(~x, AnySign)
          @rule ~x::istree => setsign(~x, (dcprule(head(~x),
            children(~x)...).sign)) where {hasdcprule(head(~x))}]
          @rule *(~~x) => setsign(~MATCH, mul_sign(~~x))
          @rule +(~~x) => setsign(~MATCH, add_sign(~~x))
    ]
    ex = Postwalk(RestartedChain(rs))(ex)
    return ex
end
```

The section of the code in Convex.jl that sets up various expression types and defines the required methods on them (by performing `cloc` on `atoms/` subdirectory) contains 2215 lines of Julia code. Our implementation contains only 625 lines of code.

This is a big productivity and maintainability advantage.

## 5.3 Catalyst

Catalyst is a library for modeling simulations of chemical reaction networks (CRNs). Catalyst supports simulating stochastic chemical kinetics (jump process), chemical Langevin equation (stochastic differential equation), and reaction rate equations (ODEs). CRNs have applications in synthetic biology, epidemics, chemical reaction research, systems biology, pharmacology.

Catalyst has a domain-specific language which produces Symbolics.jl equations that are then turned into one of the systems in SciML according to need.

```
using Catalyst
rs = @reaction_network begin
```

```

    α, S + I --> 2I
    β, I --> R
end
p = [:α => .1/100, :β => .01]
tspan = (0.0,500.0)
u0 = [:S => 99.0, :I => 1.0, :R => 0.0]

# Solve ODEs.
oproblem = ODEProblem(rs, u0, tspan, p)
osol = solve(oproblem, Tsit5())

```

It can also turn the same reaction systems into discrete jump problem:

```

# Solve Jumps.
dprob = DiscreteProblem(rs, u0, tspan, p)
jprob = JumpProblem(rs, dprob, Direct())
jsol = solve(jprob, SSAS stepper())

```

### 5.3.1 Symbolic-numeric research advantage

Catalyst has been used to simulate systems with 3744 species (Variables), and 58276 Reactions (each reaction with approximately 2 terms). This is the nature of the computations in this field, and of course requires automatic code generation starting from symbolic representation of equations.

A second advantage is more interesting, from [57]:

Chemical reaction networks with polynomial ODEs (a condition that holds for pure mass action systems) can be easily converted to symbolic steady-state systems of polynomial equations. Polynomial methods such as homotopy continuation (implemented by the HomotopyContinuation.jl Julia package) can be used to reliably compute all roots of a polynomial system [11]. This is an effective



approach for finding multiple steady states of a system. When the CRN contains Hill functions (with integer exponents), by multiplying by the denominators, one generates a polynomial system with identical roots to the original, on which homotopy continuation can still be used.

By representing reaction systems as ODEs and in the form of Homotopy continuation at the same time, Catalyst was able to push the state of the art, apart from being the fastest tool for many CRNs as shown through comprehensive benchmarks.

## 5.4 ReversePropagation.jl

ReversePropagation.jl is a Julia package for reverse propagation along a syntax tree, using source-to-source transformation via Symbolics.

Interval arithmetic can propagate a intervals through an expression. Consider, for example, propagating the intervals  $x \in [-10, 10]$  and  $y \in [-10, 10]$  through  $x^2 + y^2$ :

```
julia> using Symbolics, ReversePropagation, IntervalArithmetic
julia> @variables x y;
julia> replace(x^2 + y^2, x=>Interval(-10..10), y=>Interval(-10..10))
[0, 200]
```

Meaning  $x^2 + y^2 \in [0, 200]$ .

But now let's say we come to know that  $x^2 + y^2 \in [0, 1]$ . How can we propagate this information back, so that we can *contract* the original assumptions of  $x \in [-10, 10]$  and  $y \in [-10, 10]$ ?

Reverse propagation provides the function `forward_backward_contractor` to do this.

```
julia> contractor_func = forward_backward_contractor(x^2 + y^2, [x, y])
julia> contractor_func(Interval(-10..10) × Interval(-10..10), 0..1)
((-1, 1), [-1, 1]), [0, 200])
```

The result contains two parts: the decimated interval box stating that  $x \in [-1, 1]$  and  $y \in [-1, 1]$  such that  $x^2 + y^2 \in [0, 1]$ , and the result of the forward evaluation.

Here the `contractor_func` is a function specifically created from the  $x^2 + y^2$  expression.

The algorithm to construct `contractor_func` is called the HC4 contraction [49]. In the left-hand side Snippet in 5.1 we see the reverse-mode propagation of intervals. First this code computes `c` to be the output of the forward pass, next it intersects the result with the known constraint of the output. Then propagates this back up through  $x^2 + y^2$ .

Juxtaposed is the reverse differentiation pass which is also available in `ReversePropagation.jl`, it is the same backward propagation process that is used for both. Line by line comparison of the transformations reveal an abstract relationship between them.

Since both transformations are in Symbolics expressions, it becomes possible to compose them together. For example, we can ask the question if we constrain the gradient  $\partial_x(f(x, y)) = [0, 0], \partial_y(f(x, y)) = [0, 0]$  what are the intervals in of  $x$  and  $y$ ? So this would bound the regions where maxima, minima or saddle points of the functions lie.

<code>a ← x<sup>2</sup></code>	<code>q ← x<sup>2</sup></code>
<code>b ← y<sup>2</sup></code>	<code>r ← y<sup>2</sup></code>
<code>c ← a + b</code>	<code>s ← q + r</code>
<code>output ← c</code>	<code>s1 ← 1</code>
<code>c ← c ∩ constraint # <i>constraint on output</i></code>	<code>q1 ← s1</code>
<code>a, b ← (c-b) ∩ a, (c-a) ∩ b</code>	<code>r1 ← s1</code>
<code>y ← (sqrt(y) ∪ -sqrt(y)) ∩ y</code>	<code>ȳ ← 2r1*y</code>
<code>x ← (sqrt(x) ∪ -sqrt(x)) ∩ x</code>	<code>ẋ ← 2q1*x</code>

Snippet 5.1: On the left, the Symbolics expressions generated for reverse propagation of intervals; on the right reverse-mode automatic differentiation.

The author of this project is a mathematician with little knowledge about compilers, but is able to create this sophisticated software thanks to Symbolics' easy to understand interface. If one were to use Julia's intermediate representation (IR) object, then there is an

additional complexity of maintaining slots etc.

The reverse propagation expressions are generated by doing a function call to a single function `rev`. For example,  $a, b \leftarrow (c-b) \cdot a, (c-a) \cdot b$  comes from  $(c, a, b) = \text{rev}(+, c, a, b)$ . But by symbolically tracing this, the function call can be removed. Similarly, the reverse differentiation snippet traces through the `rrule` function of `ChainRules.jl` [52] to obtain these expressions. This is an inlining pass and the complexity of implementing this on IR is a lot more, in fact, it takes hacks to select the right method to inline. In contrast, here we can just allow functions to be called with symbols. The IR is a moving target to work with, since it can change between Julia versions, but Symbolic interface remains the same.



# Chapter 6

## Conclusion

### 6.1 Advancing beyond the expressiveness of Julia

The *Julia thesis* by Jeff Bezanson titled “Abstraction in technical computing” analyzes the field of scientific computing and provides the rationale for the design of the Julia language [50].

Multiple-dispatch—i.e., directing control flow based on types of all arguments in a function call, which we review in Section 3.2—provides the means for code-selection as well as code-specialization necessary to implement a language that can provide both the productivity of a dynamic language and the performance comparable to a statically compiled language.

Two methods are more powerful than multiple-dispatch in deciding what a given piece of code in a language should do:

- **Predicate-based dispatch:** where values are tested with boolean predicates to see if a method should admit them.
- **Pattern-based rewrites:** where a pattern matcher looks an expression to see if it can rewrite it into something more specific.

However both are *too powerful* in another sense: they make compilation undecidable. The Julia thesis remarks the following in considering of pattern-based rewriting as a means

of dispatch:

There has always been a divide between “numeric” and “symbolic” languages in the world of technical computing. To many people the distinction is fundamental, and we should happily live with both kinds of languages. But if one insists on an answer as to which approach is the right one, then the answer is: symbolic. Programs are ultimately symbolic artifacts. Computation is limited only by our ability to describe it, and symbolic tools are needed to generate, manipulate, and query these descriptions.

With the system we presented in this thesis, users can essentially use predicates on runtime values, as well as pattern-based rewriting, when they wish to utilize its power to compile a highly-specialized piece of code to do performance-intensive tasks. User-created transformations may not only be optimizations, but can be transformations that are not possible purely through multiple-dispatch, allowing programs to become the basis for generating different programs rather than simply procedures that do what’s asked of them.

# Bibliography

- [1] ABBOTT, M., ALUTHGE, D., N3N5, PURI, V., ELROD, C., SCHAUB, S., LUCIBELLO, C., BHATTACHARYA, J., CHEN, J., CARLSSON, K., AND GELBRECHT, M. mcabbott/Tullio.jl: v0.3.7, Oct. 2023.
- [2] AHRENS, W., DONENFELD, D., KJOLSTAD, F., AND AMARASINGHE, S. Looplets: A Language for Structured Coiteration. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization* (Montréal QC Canada, Feb. 2023), ACM, pp. 41–54.
- [3] BAHAREV, A., SCHICHL, H., AND NEUMAIER, A. Tearing systems of nonlinear equations I. A survey.
- [4] BELYAKOVA, J., CHUNG, B., GELINAS, J., NASH, J., TATE, R., AND VITEK, J. World age in Julia: optimizing method dispatch in the presence of eval. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 1–26.
- [5] BEZANSON, J., EDELMAN, A., KARPINSKI, S., AND SHAH, V. B. Julia: A Fresh Approach to Numerical Computing. *SIAM Review* 59, 1 (Jan. 2017), 65–98.
- [6] BLACKFORD, L. S., PETITET, A., POZO, R., REMINGTON, K., WHALEY, R. C., DEMMEL, J., DONGARRA, J., DUFF, I., HAMMARLING, S., HENRY, G., ET AL. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software* 28, 2 (2002), 135–151.
- [7] BOND, E., AUSLANDER, M., GRISOFF, S., KENNEY, R., MYSZEWSKI, M., SAMMET, J., TOBEY, R., AND ZILLES, S. FORMAC an experimental formula manipulation Compiler. In *Proceedings of the 1964 19th ACM national conference on -* (Not Known, 1964), ACM Press, pp. 112.101–112.1019.
- [8] BORGES, C. F. An Improved Algorithm for hypot(a,b), June 2019. arXiv:1904.09481 [cs, math].
- [9] BOSMA, W., CANNON, J., AND PLAYOUST, C. The Magma algebra system. I. The user language. *J. Symbolic Comput.* 24, 3-4 (1997), 235–265. Computational algebra and number theory (London, 1993).

- [10] BRADBURY, J., FROSTIG, R., HAWKINS, P., JOHNSON, M. J., LEARY, C., MACLAURIN, D., NECULA, G., PASZKE, A., VANDERPLAS, J., WANDERMAN-MILNE, S., AND ZHANG, Q. JAX: composable transformations of Python+NumPy programs, 2018.
- [11] BREIDING, P., AND TIMME, S. Homotopycontinuation.jl: A package for homotopy continuation in julia. In *Mathematical Software – ICMS 2018* (Cham, 2018), J. H. Davenport, M. Kauers, G. Labahn, and J. Urban, Eds., Springer International Publishing, pp. 458–465.
- [12] BRIGHAM, E. O., AND MORROW, R. E. The fast fourier transform. *IEEE Spectrum* 4, 12 (1967), 63–70.
- [13] CARROLL, S. M. Lecture Notes on General Relativity, Dec. 1997. arXiv:gr-qc/9712019.
- [14] CHELI, A. Metatheory.jl: Fast and elegant algebraic computation in Julia with extensible equality saturation. *Journal of Open Source Software* 6, 59 (2021), 3078.
- [15] CHRIS LATTNER, E. A. LLVM Language Reference Manual.
- [16] COUSOT, P. *Principles of abstract interpretation*. The MIT Press, Cambridge, Massachusetts, 2021.
- [17] COX, D. A., LITTLE, J. B., AND O’SHEA, D. *Ideals, varieties, and algorithms: an introduction to computational algebraic geometry and commutative algebra*, 3rd ed ed. Undergraduate texts in mathematics. Springer, New York, 2007.
- [18] DEMIN, A., AND GOWDA, S. Groebner.jl: A package for gröbner bases computations in julia, 2024.
- [19] DEVELOPERS, T. J. *Metaprogramming – The Julia Documentation*, 2014.
- [20] DEVELOPERS, T. M. *Magma Handbook – Runtime Evaluation: the eval Expression*.
- [21] DEVELOPERS, T. M. *Maple Programming Guide – Object-oriented programming*.
- [22] DIXIT, V. K., AND RACKAUCKAS, C. Optimization.jl: A unified optimization package, Mar. 2023.
- [23] DOCUMENTATION, S. Simulation and model-based design, 2020.
- [24] EATON, J. W., BATEMAN, D., HAUBERG, S., AND WEHBRING, R. *GNU Octave version 5.2.0 manual: a high-level interactive language for numerical computations*, 2020.
- [25] EINSTEIN, A. Die Grundlage der allgemeinen Relativitätstheorie. *Annalen der Physik* 354, 7 (Jan. 1916), 769–822.



- [26] ELMQVIST, H., HENNINGSSON, T., AND OTTER, M. Systems modeling and programming in a unified environment based on julia. In *International Symposium on Leveraging Applications of Formal Methods* (2016), Springer, pp. 198–217.
- [27] ELROD, C., AND MA, Y. *Unityper.jl*. Github, 2024.
- [28] FELLEISEN, M. On the expressive power of programming languages. *Science of Computer Programming* 17, 1 (1991), 35–75.
- [29] FIEKER, C., HART, W., HOFMANN, T., AND JOHANSSON, F. Nemo/hecke: Computer algebra and number theory packages for the julia programming language. In *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation* (New York, NY, USA, 2017), ISSAC '17, ACM, pp. 157–164.
- [30] FIEKER, C., HART, W., HOFMANN, T., AND JOHANSSON, F. Nemo/hecke: computer algebra and number theory packages for the julia programming language. In *Proceedings of the 2017 acm on international symposium on symbolic and algebraic computation* (2017), pp. 157–164.
- [31] FRIGO, M., AND JOHNSON, S. The design and implementation of fftw3. *Proceedings of the IEEE* 93, 2 (2005), 216–231.
- [32] FRITZSON, P., AND ENGELSON, V. Modelica—a unified object-oriented language for system modeling and simulation. In *European Conference on Object-Oriented Programming* (1998), Springer, pp. 67–90.
- [33] THE GAP GROUP. *GAP – Groups, Algorithms, and Programming, Version 4.13.0*, 2024.
- [34] GERALD J. SUSSMAN, E. A. SCMUTILS Reference Manual.
- [35] GOWDA, S., CHELI, A., AND AHRENS, W. *TermInterface.jl*. Github, 2023.
- [36] GOWDA, S., MA, Y., CHELI, A., GWÓZZDŹ, M., SHAH, V. B., EDELMAN, A., AND RACKAUCKAS, C. High-performance symbolic-numeric via multiple dispatch. *ACM Communications in Computer Algebra* 55, 3 (Sept. 2021), 92–96.
- [37] GOWDA, S., MA, Y., CHURAVY, V., EDELMAN, A., AND RACKAUCKAS, C. Sparsity programming: Automated sparsity-aware optimizations in differentiable programming. In *Program Transformations for ML Workshop at NeurIPS 2019* (2019).
- [38] GRANT, M., BOYD, S., AND YE, Y. *Disciplined Convex Programming*. Springer US, Boston, MA, 2006, pp. 155–210.
- [39] GRIEWANK, A., AND WALTHER, A. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, second ed. Society for Industrial and Applied Mathematics, USA, 2008.

- [40] GROUP, T. M. *Maxima, a Computer Algebra System*, 2009.
- [41] HALDER, A., LEE, K., AND BHATTACHARYA, R. Probabilistic robustness analysis of f-16 controller performance: An optimal transport approach. In *2013 American Control Conference* (2013), IEEE, pp. 5562–5567.
- [42] HANSON, C., AND SUSSMAN, G. J. *Software Design for Flexibility: How to Avoid Programming Yourself into a Corner*. MIT Press, 2021.
- [43] HARRIS, C. R., MILLMAN, K. J., VAN DER WALT, S. J., GOMMERS, R., VIRTANEN, P., COUNAPEAU, D., WIESER, E., TAYLOR, J., BERG, S., SMITH, N. J., KERN, R., PICUS, M., HOYER, S., VAN KERKWIJK, M. H., BRETT, M., HALDANE, A., DEL RÍO, J. F., WIEBE, M., PETERSON, P., GÉRARD-MARCHANT, P., SHEPPARD, K., REDDY, T., WECKESSER, W., ABBASI, H., GOHLKE, C., AND OLIPHANT, T. E. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362.
- [44] HEARN, A. C. REDUCE 2: A system and language for algebraic manipulation. In *Proceedings of the second ACM symposium on Symbolic and algebraic manipulation - SYMSAC '71* (Los Angeles, California, United States, 1971), ACM Press, pp. 128–133.
- [45] INC., T. M. Matlab version: 9.13.0 (r2022b), 2022.
- [46] INC., W. R. Mathematica, Version 14.0. Champaign, IL, 2024.
- [47] INC., W. R. Codegenerate, wolfram language function, 2010. Champaign, IL, 2024.
- [48] INC., W. R. Functioncompile, wolfram language function, 2010. Champaign, IL, 2024.
- [49] JAULIN, L., Ed. *Applied interval analysis*. Springer, London ; New York, 2001.
- [50] JEFFREY WERNER BEZANSON. Abstraction in technical computing. *PhD Thesis* (Jan. 2024).
- [51] JENKS, R. D., AND SUTOR, R. S. *Introduction to AXIOM*. Springer New York, New York, NY, 1992, pp. 1–8.
- [52] JULIADIFF. *ChainRules.jl: forward and reverse mode differentiation primitives for Julia*, 2022.
- [53] KEDWARD, L. J., ARADI, B., ČERTÍK, O., CURCIC, M., EHLERT, S., ENGEL, P., GOSWAMI, R., HIRSCH, M., LOZADA-BLANCO, A., MAGNIN, V., MARKUS, A., PAGONE, E., PRIBEC, I., RICHARDSON, B., SNYDER, H., URBAN, J., AND VANDENPLAS, J. The state of fortran. *Computing in Science Engineering* 24, 2 (2022), 63–72.
- [54] KJOLSTAD, F., KAMIL, S., CHOU, S., LUGATO, D., AND AMARASINGHE, S. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (Oct. 2017), 1–29.

- [55] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis and transformation. pp. 75–88.
- [56] LEGAT, B., TIMME, S., WEISSER, T., KAPELEVICH, L., SAJKO, N., MANUEL, DEMIN, A., HANSKNECHT, C., RACKAUCKAS, C., TAGBOT, J., FORETS, M., AND HOLY, T. JuliaAlgebra/DynamicPolynomials.jl: v0.5.7, May 2024.
- [57] LOMAN, T. E., MA, Y., ILIN, V., GOWDA, S., KORSBO, N., YEWALE, N., RACKAUCKAS, C., AND ISAACSON, S. A. Catalyst: Fast and flexible modeling of reaction networks. *PLOS Computational Biology* 19, 10 (10 2023), 1–19.
- [58] MA, Y., GOWDA, S., ANANTHARAMAN, R., LAUGHMAN, C., SHAH, V., AND RACKAUCKAS, C. Modelingtoolkit: A Composable Graph Transformation System For Equation-Based Modeling, 2021.
- [59] MA, Y., GOWDA, S., ANANTHARAMAN, R., LAUGHMAN, C., SHAH, V. B., AND RACKAUCKAS, C. Modelingtoolkit: A composable graph transformation system for equation-based modeling. *CoRR abs/2103.05244* (2021).
- [60] MAPLESOFT, A DIVISION OF WATERLOO MAPLE INC.. Maple.
- [61] MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM* 3, 4 (Apr. 1960), 184–195.
- [62] MENENDEZ, D., NAGARAKATTE, S., AND GUPTA, A. Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM. In *Static Analysis*, X. Rival, Ed., vol. 9837. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016, pp. 317–337. Series Title: Lecture Notes in Computer Science.
- [63] MEURER, A., SMITH, C. P., PAPROCKI, M., ČERTÍK, O., KIRPICHEV, S. B., ROCKLIN, M., KUMAR, A., IVANOV, S., MOORE, J. K., SINGH, S., RATHNAYAKE, T., VIG, S., GRANGER, B. E., MULLER, R. P., BONAZZI, F., GUPTA, H., VATS, S., JOHANSSON, F., PEDREGOSA, F., CURRY, M. J., TERREL, A. R., ROUČKA, V., SABOO, A., FERNANDO, I., KULAL, S., CIMRMAN, R., AND SCOPATZ, A. Sympy: symbolic computing in python. *PeerJ Computer Science* 3 (Jan. 2017), e103.
- [64] MODIA. *Modia: Modeling and simulation of multidomain engineering systems*, 2024.
- [65] MOSES, J. Algebraic Simplification: A Guide for the Perplexed.
- [66] OTTER, M., AND ELMQVIST, H. Transformation of differential algebraic array equations to index one form. In *Proceedings of the 12th International Modelica Conference* (2017), Linköping University Electronic Press.
- [67] PANTELIDES, C. C. The consistent initialization of differential-algebraic systems. *SIAM Journal on Scientific and Statistical Computing* 9, 2 (1988), 213–231.

- [68] R CORE TEAM. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2021.
- [69] RACKAUCKAS, C., AND NIE, Q. Differentialequations.jl—a performant and feature-rich ecosystem for solving differential equations in julia. *Journal of Open Research Software* 5, 1 (2017).
- [70] RACKAUCKAS, C., AND NIE, Q. Confederated modular differential equation apis for accelerated algorithm development and benchmarking. *Advances in Engineering Software* 132 (2019), 1–6.
- [71] RUNTIMEGENERATEDFUNCTIONS. *RuntimeGeneratedFunctions.jl: Functions generated at runtime without world-age issues or overhead*, 2024.
- [72] SABNE, A. Xla : Compiling machine learning for peak performance, 2020.
- [73] SAMMET, J. E. The beginning and development of FORMAC (FORMula MANipulation Compiler). In *The second ACM SIGPLAN conference on History of programming languages* (Cambridge Massachusetts USA, Mar. 1993), ACM, pp. 209–230.
- [74] SANDERS, D. *ReversePropagation.jl*. Github, 2022.
- [75] SHIMAKO, K., AND KOGA, M. An extension of pantelides algorithm for consistent initialization of differential-algebraic equations using minimally singularity. In *2020 59th Annual Conference of the Society of Instrument and Control Engineers of Japan (SICE)* (2020), IEEE, pp. 1676–1681.
- [76] STEELE, G. L., ALLEN, E., CHASE, D., FLOOD, C., LUCHANGCO, V., MAESSEN, J.-W., AND RYU, S. *Fortress (Sun HPCS Language)*. Springer US, Boston, MA, 2011, pp. 718–735.
- [77] THE MATHWORKS, I. *Operator Overloading – MATLAB & Simulink*.
- [78] THE MATHWORKS, I. *Symbolic Math Toolbox*. Natick, Massachusetts, United State, 2019.
- [79] THE MAXIMA DEVELOPERS. *Introduction to Gentran*. <https://maxima.sourceforge.io/docs/manual/maxima278.html>.
- [80] THE SAGE DEVELOPERS. *SageMath, the Sage Mathematics Software System*. <https://www.sagemath.org>.
- [81] THE SYMENGINE DEVELOPERS. *SymEngine*. <https://www.symengine.org>.
- [82] THE SYMPY DEVELOPERS. *SymPy - autowrap() function*. <https://docs.sympy.org/latest/modules/utilities/autowrap.html>.

- [83] TSITOURAS, C. Runge–kutta pairs of order 5(4) satisfying only the first column simplifying assumption. *Computers Mathematics with Applications* 62, 2 (2011), 770–775.
- [84] TSITOURAS, C. Runge–kutta pairs of order 5 (4) satisfying only the first column simplifying assumption. *Computers & Mathematics with Applications* 62, 2 (2011), 770–775.
- [85] UDELL, M., MOHAN, K., ZENG, D., HONG, J., DIAMOND, S., AND BOYD, S. Convex optimization in Julia. In *Proceedings of the 1st First Workshop for High Performance Technical Computing in Dynamic Languages* (2014), IEEE Press, pp. 18–28.
- [86] VELTMAN, M., AND WILLIAMS, D. N. Schoonschip '91, 1993.
- [87] WOLFRAM, S. *The Mathematica book: the definitive best-selling presentation of Mathematica by the creator of the system*, 5., ed ed. Wolfram Media [u.a.], Champaign, Ill., 2003.
- [88] WOLFRAM, S. Tini veltman (1931–2021): From assembly language to a nobel prize, 2021.
- [89] YEGGE, S. Execution in the kingdom of nouns, 2006.