Hash Consing in JuliaSymbolics for Efficient Symbolic-Numeric Code Generation

Bowen Zhu[1], Aayush Sabharwal[2], Songchen Tan[1],
Yingbo Ma[3], Alan Edelman[1], and Christopher Rackauckas[1]
[1]Massachusetts Institute of Technology
[2]JuliaHub
[3]Independent Researcher

bowenzhu@mit.edu, aayush.sabharwal@juliahub.com, songchen@mit.edu

mayingbo5@gmail.com, edelman@mit.edu, crackauc@mit.edu

Abstract

We present a software contribution for JuliaSymbolics that adds transparent hash consing to the SymbolicUtils.jl term representation used by Symbolics.jl and ModelingToolkit.jl. Immutable `BasicSymbolic` compound terms are interned through weak-reference tables, turning repeated expression trees into shared directed acyclic graphs without changing the user API. On a 1,122-variable B cell receptor model, the implementation reduces Jacobian-construction memory by about 2× and improves symbolic Jacobian time by about 3.2×. On XSteam-based thermodynamic benchmarks, generated-code compilation improves by nearly 10× and numerical evaluation by 20×–100×, while low-repetition construction cases may pay lookup overhead.

1 Software contribution

The software contribution is a hash-consed term layer for JuliaSymbolics, the ecosystem around Symbolics.jl, SymbolicUtils.jl, and ModelingToolkit.jl [1, 2]. These packages support symbolic-numeric modeling, transformations of differential-algebraic systems, and compiled Julia kernels. The change is an infrastructure improvement: the same modeling code constructs a more compact internal representation.

Before this work, structurally equal subexpressions could be allocated repeatedly as distinct Julia objects. That tree-oriented representation is flexible, but it is costly in large symbolic-numeric workloads: memory pressure grows, equality tests traverse structure, and code generation must rediscover sharing lost during term construction. Our implementation makes live, structurally equal immutable expressions share one object, exposing a DAG while preserving Symbolics.jl syntax. Unlike conventional common-subexpression elimination, which discovers repetition after construction, hash consing prevents many duplicate objects from being allocated and makes sharing available to later passes.

2 Implementation

Hash consing, originating in work on arithmetic expressions and later Lisp and functional-language runtimes [3, 4, 5], queries a table whenever a compound immutable object is created. If a structurally equal object exists, the constructor returns it, saving memory and reducing equality of shared terms to pointer comparison.

In JuliaSymbolics the relevant construction path lives in SymbolicUtils.jl and the TermInterface.jl protocol [6, 7]. The interned units are immutable `BasicSymbolic` compound terms built through this path. Structurally, a term is its operator or head, ordered arguments, symtype/domain information, and relevant metadata after existing constructor-level simplifications. Leaf symbols, constants, and mutable or ad hoc objects are not the main target unless they pass through the same immutable constructor path; immutability makes it safe to return one representative object to multiple users.

The constructor hashes each compound term, looks it up in a weak-value table, and returns an existing canonical term or inserts a representative of its structural class. Weak values share live expressions without keeping obsolete objects alive forever [8, 9]. The implementation uses a global weak-value interning table for canonical representatives. In multithreaded contexts, task-local hash-consing caches and memoization tables support lock-free operation; identical work may still be duplicated across threads, so cross-thread caching of memoized results remains a future optimization.

Hash consing detects structural equality of the internal representation, not arbitrary mathematical equivalence. Thus `x^0*y^2`, `y^2`, and `y^2*z^0` share only if existing normalization reduces them to the same structural term, such as `y^2`; the hash table does not prove equivalence. Hash values are cached on terms, and collisions fall back to structural checks, so the table is an accelerator rather than a soundness assumption.

Hash consing alone is not enough: tree-oriented algorithms can still revisit a shared subterm many times. We therefore adapted code generation to track visited shared terms, emit temporary bindings, and preserve evaluation order. For instance, in `(x+y)^2 + sin(x+y)`, both `x+y` occurrences can share the same representative, letting code generation emit `t1 = x + y` and reuse `t1` instead of recomputing the subterm.

3 Demonstration and evaluation

The presentation will construct repeated Symbolics.jl expressions, show the temporaries introduced for shared DAG nodes, and report benchmark results for two scientific-modeling workloads: a B cell receptor signaling model with 1,122 ODEs and 24,388 reactions [10], and symbolic thermodynamic models derived from XSteam.jl and the IAPWS-IF97 water-steam formulation [11, 12]. The live example establishes the invariant that repeated terms become shared nodes; the benchmarks show effects on memory, generated code size, compilation latency, and evaluation time.

The reported measurements were run on Ubuntu Linux (kernel 5.15.0-58-generic) on x86_64 with an AMD EPYC 7502 32-core processor, Julia 1.10.9, SymbolicUtils.jl 3.25.1, and Symbolics.jl 6.36.0. We measure wall-clock time and allocated memory for symbolic Jacobian construction, code generation, compilation, and numerical evaluation through ordinary Symbolics.jl and ModelingToolkit.jl construction paths. Benchmarks used SymbolicUtils.jl 3.25.1; the documentation citation refers to public documentation for the same TermInterface/SymbolicUtils concepts accessed during

manuscript preparation.

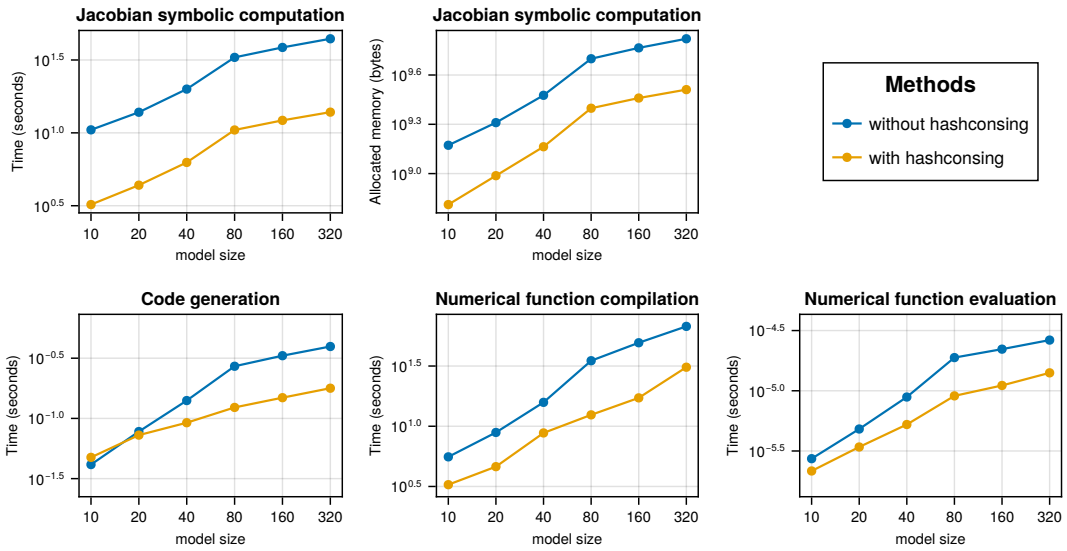

Figure 1: BCR benchmark results. Representative BCR and XSteam takeaways are summarized in Table 1.

Table 1: Representative effects of hash consing and DAG-aware code generation.

Workload or regime	Representative effect
BCR Jacobian construction	About 2× less memory and about 3.2× faster symbolic Jacobian construction
XSteam-generated numerical code	Nearly 10× generated-code compilation improvement and 20×–100× numerical evaluation speedup
Low-repetition construction	Lookup overhead can dominate before downstream DAG-aware passes exploit sharing

The mixed behavior is informative. Reaction-network Jacobians contain repeated local motifs, so construction-time interning helps Jacobian construction and memory immediately. XSteam-derived formulas may expose fewer duplicate unknown-variable terms during initial construction, so early Jacobian gains can be smaller or overhead can appear. Yet generated numerical programs contain repeated expensive thermodynamic fragments, so DAG-aware code generation reduces compiled code size, compilation latency, and evaluation work. Users should evaluate the whole symbolic-to-numeric pipeline, not only construction time.

There are also clear limitations. Hash consing is most attractive for immutable terms created through centralized constructors; mutable or ad hoc construction requires extra invalidation or normalization machinery. Weak references prevent unbounded table growth, but lookup cost can dominate when few repeated terms are produced; adaptive policies based on expression size, operator distribution, or observed hit rate are natural future work.

4 Related software

Other computer algebra systems expose related representation questions; for example, Symbolica reached a 1.0 release as a high-performance Rust and Python library [13]. Hash consing is also complementary to e-graph rewriting: e-graphs represent equivalence classes of expressions, whereas hash consing canonicalizes structurally identical terms. Combining hash-consed sharing with equality-saturation techniques relevant to JuliaSymbolics is a natural next development [14].

5 Conclusion

The presented software contribution is a transparent hash-consed representation for JuliaSymbolics terms, together with DAG-aware code generation and benchmark evidence on large scientific models. The main lesson is that maximal sharing is most valuable when downstream symbolic-numeric stages exploit it explicitly. The result is a concrete implementation in an active symbolic computation ecosystem that improves performance on representative workloads and exposes practical design tradeoffs for future symbolic software.

References

[1] S. Gowda, Y. Ma, A. Cheli, M. Gwóźdź, V. B. Shah, A. Edelman, and C. Rackauckas. High-performance symbolic-numerics via multiple dispatch. *ACM Commun. Comput. Algebra*, 55(3):92–96, 2022.

[2] Y. Ma, S. Gowda, R. Anantharaman, C. Laughman, V. B. Shah, and C. Rackauckas. ModelingToolkit: A composable graph transformation system for equation-based modeling. *CoRR*, abs/2103.05244, 2021.

[3] A. P. Ershov. On programming of arithmetic operations. *Commun. ACM*, 1(8):3–6, 1958.

[4] E. Goto. Monocopy and associative algorithms in an extended Lisp. Technical Report TR 74-03, University of Tokyo, 1974.

[5] J. Goubault. Implementing functional languages with fast equality, sets and maps: an exercise in hash consing. *Journées Francophones des Langages Applicatifs*, pages 222–238, 1994.

[6] JuliaSymbolics contributors. SymbolicUtils.jl documentation, version 4.24, 2026. `https://docs.sciml.ai/SymbolicUtils/v4.24/`.

[7] JuliaSymbolics contributors. TermInterface.jl documentation, 2026. `https://juliasymbolics.github.io/TermInterface.jl/dev/`.

[8] A. W. Appel and M. J. R. Gonçalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, 1993.

[9] T. DePrato. WeakValueDicts.jl. GitHub repository, 2020. `https://github.com/twavv/WeakValueDicts.jl`.

[10] D. Barua, W. S. Hlavacek, and T. Lipniacki. A computational model for early events in B cell antigen receptor signaling: Analysis of the roles of Lyn and Fyn. *The Journal of Immunology*, 189(2):646–658, 2012.

[11] hzgzh. XSteam.jl. GitHub repository, 2024. `https://github.com/hzgzh/XSteam.jl`.

[12] IAPWS. Revised release on the IAPWS Industrial Formulation 1997 for the thermodynamic properties of water and steam, 2007. `https://www.iapws.org/relguide/IF97-Rev.html`.

[13] B. Ruijl. Symbolica: A modern computer algebra system, 2025. `https://symbolica.io/posts/stable_release/`.

[14] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), 2021.