CGS-D Scholarship Application

Anthony Hunt

September 20, 2025

1 Thesis In-Progress Summary

1.1 Making an Efficient Very High-Level Programming Language: Optimization of Abstract Data Types

Summary Sets, sequences, and relations are essential structures in formally defining system specifications. Executable modelling languages and high-level programming languages often expose these data types as syntactic sugar over more concrete notation.

Preliminary work on my current Master's thesis focused on prototyping a term rewriting system to generate efficient code from such abstract type operations. Thus far, we have conducted a literature review of abstract data types and term rewriting systems as components of optimizing compilers in the form of brief surveys. Then, to direct our exploration of the problem, we gathered around a dozen examples of formal system definitions and created a set of optimization rules to satisfy each example. Unifying these rules yielded generic optimization patterns especially when converting set operations like union or intersection to boolean-level quantification equivalents. We then implemented and presented a prototype compiler for set theory expressions at a program synthesis workshop (SYNT 2025), receiving excellent feedback incorporated in the complete project proposal.

2 Outline of Proposed Research

Since the realization of the modern computer, programming languages have fundamentally shaped technological advancements and served as an interface for building unfathomably complex machines. However, among the thousands of available languages, relatively few manage to combine intuitive, high-level notation with reliably efficient execution. Our work seeks a resolution between the formal semantics and expressivity of specification languages (like UML, Event-B, and Dafny) and the performance of modern programming language implementations (as in Haskell and Rust) to answer the following:

1. Can programming languages become more efficient and more expressive through the influence of specification languages?

2. Which specification language expressions can be optimized for both runtime and memory consumption?

3. How can we make a high-level programming language competitive with low-level language runtime performance?

System modelling and behaviour specifications commonly draw from set theory to define collections and relationships. Sets, sequences, and relations are pervasive in safety-critical software specifications, since their abstract nature, composed of theory-defined behaviours, lends well to proofs and guarantees of correctness. For example, Abrial's B method [1] was used in the 1990's to ensure safe design of the Paris Metro signaling system [14]. Further, Hoare logic, adopted by Dafny [31], is used to prove properties of imperative programs. Discrete mathematics and logic systems are essential to specification expressions that consider reliability a primary goal.

Conversely, many popular programming languages stray from theory and generally force programmers to focus on the executable implementation. Arrays, heap-allocated memory, and classes are all implementation-aware constructs that, while closer to hardware, shifts the responsibility of overall system management onto the programmer. These types of programs, even from high-level languages, can be extremely difficult to prove following standard formal methods.

While some high-level languages offer close alternatives to abstract collection types, as in Python, where lists parallel sequences and dictionaries parallel relations, they cannot support the full extent of theory-defined operators. Dictionaries alone are a many-to-one subset of abstract relations, since the underlying hashmap implementation prevents registration of duplicate keys with differing values. The Event-B specification language, on the other hand, provides convenient operators like relation composition or inverse, albeit with inefficient execution.

Programming language sets mirror theoretical operations closely and provide efficient membership lookup through hashing, but successive operations are far more inefficient and memory-consuming than necessary. For example, in Python, finding the intersection of sets in an expression like $S \cap T \cap U$ would first generate an intermediate set for $S \cap T$ and then join the temporary set with U. While inefficient, this behaviour necessarily loops through $S \cap T$ twice because of implementation aware behaviour and expected side effects. A language closer to theory would instead calculate both intersections in one loop through S, checking against membership in both T and U, without allocating more memory than absolutely required.

The new programming language we propose combines Event-B notation, Python-like code structure, and SetL's semi-automatic concrete data structure selection to enable concise expressions with mathematic semantics. Primitive operators like ($\wedge$) and ($\vee$) are expected to diverge from standard short circuiting to theory-correct commutative behaviour. Truly non-deterministic actions, like choosing a random element from a set, are natural to specifications but rarely found in conventional programming languages. We further plan to add type refinements according to set theory, enabling further static analysis and unlocking new optimizations.

Because abstract data types serve as the mathematical and programmatic foundation for our language, our term rewriting compiler transforms equational theorems into directed, provably semantic-preserving, high-level optimization passes. Rewrite rules based on abstract type and operation semantics can eliminate all excess intermediate memory allocation in complex expressions, improving upon default imperative language behaviour. Type extensions involving, for example, the injective, surjective, or totality properties of a relation can further reduce superfluous relationship lookups and implicitly direct the actualized structure of variables.

We anticipate a term rewriting system that can refine abstract structures into varying concrete implementations, depending on the situation. For example, generic sets are commonly represented by a hashset, but sets of dense or sparse integer collections may find better performance with a bitset or roaring bitsets respectively. Relations, which default to a single-direction hashmap, may find better performance in bi-directional hashmap for efficient inverse lookup and representation of many-to-many relations. Recent work in tree-based path-map structures [24] may offer even more efficient operation execution by precluding the need for an iteration over an entire relation. Once high-level optimization passes are complete, we lower the now-concrete data structures into LLVM assembly, conveniently harnessing decades of low level optimization work [28].

A successful language should be far more efficient than interpreted high-level languages like Python and JavaScript, with competitive runtime compared to idiomatically-written low-level C or Rust code. Memory usage should be optimal, undercutting any eagerly-evaluated programming language. Programming language benchmarks will be created based on three categories: algorithmically-intense computations, complex simulations, and data-heavy information systems.

Providing an efficient high-level language brings software engineering closer to the specification design and simplifies verification of program correctness. Hoare logic program guarantees may now be written directly as part of the program and used in the optimization process, rather than relegating useful semantic information to only exist in documentation form. Additionally, direct usability benefits from borrowing purer mathematical notation may increase adoption of formal methods and accessibility of safety-critical programming to novice programmers, increasing the overall safety and efficiency of software systems.

3 Contributions and Statements

3.1 Contributions to research and development

I. c.

Anthony Hunt* and Emil Sekerinski. (2025) Generating Implementations of Data Type Operations. Workshop on Synthesis (co-located with the International Conference on Computer Aided Verification). Pages: 3 (Master's work, presentation).

Anthony Hunt*, Jacques Carette, and Spencer Smith. (2021) Information Encoding and Traceability in Software. Annual Undergraduate Student Research Poster Showcase at McMaster University (Undergraduate work, poster).

I. d.

Anthony Hunt and Hamid Hosseiny. (2024) MemAutoGUI: An Enhanced Automation Approach for Accelerated Validation Focused on Usability. In: *Design and Test Technology Conference*. Pages: 10. Intel (Work done with team lead during an internship).

3.2 Most significant contributions to research and development

3.3 Applicant's statement

4 Equity, Diversity, and Inclusion Considerations

Although the creation of a mathematics-focused programming language may not present the same equity, diversity, and inclusion concerns as a human-centered experiment, we nonetheless consider accessibility impact in our work. Similar to standard programming languages, the interface language is through text based files. However, we aim to support both unicode and ASCII characters for better symbolic representation of mathematics. This, in turn, may assist text-to-speech programs for the visually impaired by using more concise and relevant symbols. As a simple example, consider the ASCII notation for the expression of a relation `S <-> T`. Replacing the "less than, dash, greater than" character sequence with ($\leftrightarrow$) is more straightforward and understandable in a programming context.

References

[1] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering.* Cambridge University Press, 2010.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998. DOI: 10.1017/CBO9781139172752.

[3] Richard S. Bird. "An Introduction to the Theory of Lists". In: *Logic of Programming and Calculi of Discrete Design.* Ed. by Manfred Broy. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 5–42. DOI: 10.1007/978-3-642-87374-4_1. (Visited on 05/11/2025).

[4] Oliver Bračevac, Guannan Wei, Songlin Jia, Supun Abeysinghe, Yuxuan Jiang, Yuyan Bao, and Tiark Rompf. "Graph IRs for Impure Higher-Order Languages: Making Aggressive Optimizations Affordable with Precise Effect Dependencies". In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (Oct. 2023). DOI: 10.1145/3622813.

[5] Mark G. J. van den Brand, Paul Klint, and Jurgen J. Vinju. "Term rewriting with traversal functions". In: *ACM Trans. Softw. Eng. Methodol.* 12.2 (Apr. 2003), pp. 152–190. DOI: 10.1145/941566.941568.

[6] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. "Stratego/XT 0.17. A language and toolset for program transformation". In: *Science of Computer Programming* 72.1 (2008). Special Issue on Second issue of experimental software and toolkits (EST), pp. 52–70. DOI: 10.1016/j.scico.2007.11.003.

[7] Martin Bravenboer and Eelco Visser. "Rewriting Strategies for Instruction Selection". In: *Rewriting Techniques and Applications.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 237–251.

[8] Bruno Buchberger. "Mathematica as a rewrite language". In: *Proceedings of the 2nd Fuji International Workshop on Functional and Logic Programming.* World Scientific. 1996, pp. 1–13. URL: https://www.researchgate.net/publication/243768023_Mathematica_as_a_Rewrite_Language.

[9] *C++ reference.* 2024. URL: https://en.cppreference.com/w/ (visited on 02/07/2025).

[10] John Cooke. *Constructing Correct Software: The Basics.* Springer, 1998. DOI: 10.1007/b138515.

[11] Alcino Cunha, Nuno Macedo, Julien Brunel, and David Chemouil. "Practical Alloy". Feb. 2025. URL: https://practicalalloy.github.io/index.html.

[12] *Dafny Reference Manual.* 2025. URL: https://dafny.org/latest/DafnyRef/DafnyRef (visited on 02/13/2025).

[13] Luca Di Grazia and Michael Pradel. "The evolution of type annotations in python: an empirical study". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* ESEC/FSE 2022. Singapore, Singapore: Association for Computing Machinery, 2022, pp. 209–220. DOI: 10.1145/3540250.3549114.

[14] S. Gerhart, D. Craigen, and T. Ralston. "Case study: Paris Metro Signaling System". In: *IEEE Software* 11.1 (1994), pp. 32–28. DOI: 10.1109/MS.1994.1279941.

[15] D. Gries and F.B. Schneider. *A Logical Approach to Discrete Math*. Monographs in Computer Science. Springer New York, 1993. DOI: 10.1007/978-1-4757-3837-7.

[16] Bastian Hagedorn, Johannes Lenfers, Thomas Kundefinedhler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. "Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies". In: *Proc. ACM Program. Lang.* 4.ICFP (Aug. 2020). DOI: 10.1145/3408974.

[17] Mary Hall, Cosmin Oancea, Anne C. Elster, Ari Rasch, Sameeran Joshi, Amir Mohammad Tavakkoli, and Richard Schulze. *Scheduling Languages: A Past, Present, and Future Taxonomy*. 2024. arXiv: 2410.19927 [cs.PL]. URL: https://arxiv.org/abs/2410.19927.

[18] A. Hsu. *Is APL Dead?* 2020. URL: https://www.sacrideo.us/is-apl-dead/ (visited on 02/07/2025).

[22] Kenneth E. Iverson. "Notation as a tool of thought". In: *Commun. ACM* 23.8 (Aug. 1980), pp. 444–465. DOI: 10.1145/358896.358899.

[23] Daniel Jackson. "Alloy: a language and tool for exploring software designs". In: *Commun. ACM* 62.9 (Aug. 2019), pp. 66–76. DOI: 10.1145/3338843.

[24] Simon Peyton Jones and Sebastian Graf. *Triemaps that match*. 2024. arXiv: 2302.08775 [cs.PL]. URL: https://arxiv.org/abs/2302.08775.

[25] S. Klabnik and C. Nichols. *The Rust Programming Language, 2nd Edition*. No Starch Press, 2023. URL: https://doc.rust-lang.org/book/.

[26] Cynthia Kop and Naoki Nishida. "Term rewriting with logical constraints". In: *International Symposium on Frontiers of Combining Systems*. Springer. 2013, pp. 343–358. DOI: 10.1007/978-3-642-40885-4_24.

[27] David Lacey and Oege de Moor. "Imperative Program Transformation by Rewriting". In: *Proceedings of the 10th International Conference on Compiler Construction*. CC '01. Berlin, Heidelberg: Springer-Verlag, 2001, pp. 52–68. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=2836d42b58370b363edbbb9956387361870ec3ac.

[28] Chris Lattner. "LLVM: An Infrastructure for Multi-Stage Optimization". MA thesis. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002. URL: https://www.llvm.org/pubs/2002-12-LattnerMSThesis-book.pdf.

[29] *LearningAPL*. 2024. URL: https://xpqz.github.io/learnapl/intro.html (visited on 02/07/2025).

[30] K Rustan M Leino. "Specification and verification of object-oriented software". In: *Engineering Methods and Tools for Software Safety and Security*. IOS Press, 2009, pp. 231–266.

[31] Rustan Leino. *Program Proofs*. MIT Press, 2023.

[32] M. Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press Series. No Starch Press, 2011.

[33] Simon Marlow and Simon Peyton-Jones. *The Architecture of Open Source Applications (Volume 2) The Glasgow Haskell Compiler*. 2012. URL: https://aosabook.org/en/v2/ghc.html.

[34] J.J. Martin. *Data Types and Data Structures*. Prentice-Hall International series in personal computing. Prentice-Hall International, 1986.

[35] Christophe Métayer and Laurent Voisin. "The Event-B Mathematical Language". 2007.

[36] Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. "Rewrite rule inference using equality saturation". In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021). DOI: 10.1145/3485496.

[37] *OMG Unified Modeling Language*. Version 2.5.1. Object Management Group. 2017. URL: https://www.omg.org/spec/UML/2.5.1/PDF (visited on 02/07/2025).

[38] Zoe Paraskevopoulou. "Verified Optimizations for Functional Languages". PhD thesis. Princeton University, 2020. URL: https://dataspace.princeton.edu/handle/88435/dsp01pr76f648c.

[39] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. "Playing by the rules: rewriting as a practical optimisation technique in GHC". In: *2001 Haskell Workshop*. ACM SIGPLAN. Sept. 2001.

[40] *Query processing architecture guide*. 2025. URL: https://learn.microsoft.com/en-us/sql/relational-databases/query-processing-architecture-guide?view=sql-server-ver16 (visited on 04/30/2025).

[41] Ken Robinson. "A Concise Summary of the Event-B mathematical toolkit". Jan. 2014. URL: https://wiki.event-b.org/images/EventB-Summary.pdf.

[42] Douglas R. Smith and Stephen J. Westfold. "Transformations for Generating Type Refinements". In: *Formal Methods. FM 2019 International Workshops*. Ed. by Emil Sekerinski et al. Cham: Springer International Publishing, 2020, pp. 371–387. DOI: 10.1007/978-3-030-54997-8_24.

[43] Jeff Smits, Toine Hartman, and Jesper Cockx. "Optimising First-Class Pattern Matching". In: *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2022. Auckland, New Zealand: Association for Computing Machinery, 2022, pp. 74–83. DOI: 10.1145/3567512.3567519.

[44] J. M. Spivey. "An introduction to Z and formal specifications". In: *Softw. Eng. J.* 4.1 (Jan. 1989), pp. 40–50. DOI: 10.1049/sej.1989.0006.

[45] *Strategic Rewriting*. 2023. URL: https://spoofax.dev/background/stratego/strategic-rewriting/strategic-rewriting/ (visited on 03/14/2025).

[46] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. "Equality saturation: a new approach to optimization". In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '09. Savannah, GA, USA: Association for Computing Machinery, 2009, pp. 264–276. DOI: 10.1145/1480881.1480915.

[47] *The Python Language Reference (3.13.2)*. Python Software Foundation. 2025. URL: https://docs.python.org/3/reference/ (visited on 02/07/2025).

[48] P.G. Thomas, H. Robinson, and J. Emms. *Abstract Data Types: Their Specification, Representation, and Use*. Oxford applied mathematics and computing science series. Clarendon Press, 1988.

[49] Yoshihito Toyama. "Commutativity of term rewriting systems". In: *Programming of future generation computers II* (1988), pp. 393–407. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=7046f12ef6e51e9c02d75d9b986532e45eb31513.

[50] A. Turon. *Rust's language ergonomics initiative*. 2017. URL: https://blog.rust-lang.org/2017/03/02/lang-ergonomics.html (visited on 02/07/2025).

[51] *Typescript Documentation*. 2024. URL: https://www.typescriptlang.org/docs/ (visited on 02/07/2025).

[52] Eelco Visser. "A Survey of Rewriting Strategies in Program Transformation Systems". In: *Electronic Notes in Theoretical Computer Science* 57 (2001). WRS 2001, 1st International Workshop on Reduction Strategies in Rewriting and Programming, pp. 109–143. DOI: 10.1016/S1571-0661(04)00270-1.

[53] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. "Building program optimizers with rewriting strategies". In: *SIGPLAN Not.* 34.1 (Sept. 1998), pp. 13–26. DOI: 10.1145/291251.289425.

[54] *Welcome to Agda's documentation!* Version 2.7.0.1. 2025. URL: https://agda.readthedocs.io/en/v2.7.0.1/# (visited on 02/07/2025).

[55] Jonne van Wijngaarden and Eelco Visser. "Program Transformation Mechanics: A classification of Mechanisms for Program Transformation with a Survey of Existing Transformation Systems". In: (Mar. 2004). URL: https://www.researchgate.net/publication/2913394_Program_Transformation_Mechanics_A_classification_of_Mechanisms_for_Program_Transformation_with_a_Survey_of_Existing_Transformation_Systems.

[56] Victor L. Winter. "Program Transformation: What, How, and Why". In: *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons, Inc., 2008. DOI: 10.1002/9780470050118.ECSE330. URL: https://doi.org/10.1002/9780470050118.ecse330.