High-Level Optimization of Abstract Data Types

Anthony Hunt

September 20, 2025

1 Thesis In-Progress Summary

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 [**TODO**] was used in the 1990's to ensure safe design of the Paris Metro signaling system [**TODO**]. Further, Hoare logic, adopted by Dafny [**TODO**], 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 [**TODO**] 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 [**TODO**], conveniently harnessing decades of low level optimization work [**TODO**].

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.

Recent coursework has continued this trend toward programming language projects. My capstone project, for example, explored visual methods of programming language interaction, using diagrams to gain a deeper understanding of information flow. Simultaneously, a one-on-one directed readings course with Dr. Emil Sekerinski allowed me to explore recent literature on languages, modelling methods [**eventBBook**], and rewriting [**baader1998term**]. I spent several weeks researching formal mathematics in ten diverse languages, making note of strengths, weaknesses, and possible efficiency improvements. Then, I completed a survey on historical and recent rewriting techniques, from strategies [**elco1998building**] to equational saturation [**tate2009equality**]. In the final weeks of my undergrad, I built a prototypical optimizing compiler for set-based operations.

Continuing work on the optimizing compiler in the summer months initiates the first phase of my Master's project: generating efficient implementations of abstract data types [**hunt2025generating**]. Oftentimes, formal specifications for real-world modelling only make use of boolean logic, number theory, and collection types foundational to discrete mathematics (as seen in the birthday book problem [**spivey1989birthday**]). Sets, sequences, and relations may very well be powerful enough to accomplish a large subset of programming tasks that otherwise demand complex, concrete data types. Further, collection types come with a rich set of universally agreed-upon semantics [**griesAndSchneider**] that allow extensive compilation time optimizations. Through my research, I seek to answer the following: To what extent can abstract data types be used to model real-world phenomena? Then, using semantics as a foundation for optimization, are abstract data type operations more efficient than similar tasks in conventional languages?

McMaster's depth and breadth in formal methods, programming language theory, and compilation align closely with my goal of building a more usable interface for reasoning and communicating. By continuing my education, I plan to learn about type theory, compiler optimization, and influential approaches to modelling computation. Further, I am excited about advancing the core areas of modern languages, especially data structure optimization, formal computation techniques, and industry-standard code generation. Researching under this program and experimenting with compilation techniques will grant me the skills needed to back my ideals for future language development.

3 Contributions and Statements

4 Equity, Diversity, and Inclusion Considerations

5 Bibliography