The Beginnings of an Optimizing Compiler for a Very High Level Language

Anthony Hunt

April 30, 2025

Contents

1 Introduction **2**

2 Exploring Examples **2**
2.1 The Birthday Book Problem 2
2.1.1 Handling Relations: Many-to-One and Many-to-Many . . 3
2.1.2 Extending the Problem with Two Birthday Books 4
2.2 Other Combinations of Sets, Sequences, and Relations 6

3 A TRS for a Very High Level Language **8**
3.1 Rewriting Rules for Set Notation 8
3.2 Completeness of Rewriting Set Notation 12

4 A Backdrop for Concrete Sets **15**
4.1 Sets in Other Languages 15
4.2 Running Time of Optimized Sets 16

5 A Very High Level Language **17**
5.1 Supported Operators and ADTs 17
5.2 Compiler Pipeline 18
5.2.1 Grammar 18
5.2.2 AST and AST transformations 19

6 Future Work **20**

7 Conclusion **20**

1 Introduction

In the modern era of programming, optimizing compilers must make aggressive choices in pursuit of the most efficient, yet correct, representation of source code. For low level languages like C and Rust, automatic optimizations on-par with human abilities are a mature part of the software ecosystem and a reasonable expectation for performance-oriented programmers. The effectiveness of a compiler directly influences the contributors and community of its source language, providing sweeping benefits across entire codebases at minimal cost.

However, in domains that do not require peak efficiency, high level languages focused on readability and ease-of-use have become foundational. As software development continues to grow more accessible to the general public, the responsibilities of compilers to shoulder the burden of optimization are even more pronounced. Languages like Python and JavaScript have numerous libraries and runtimes that are built on efficient languages but expose only easy-to-use, native APIs.

In the last two papers of this series, we explored a collection of abstract data types in practice and recent research into term rewriting systems for optimizing compilers. Now, we turn our attention to a prototypical implementation of a very high level language, usable by expressive discrete math notation and built with the efficiency of a low-level language. Although we do not present a full compiler as of yet, we discuss a few critical findings that will serve well in a long-term implementation.

The rest of this paper explores theoretical optimizations on select examples, a few implementation tests, then outlines a roadmap for a full compiler implementation.

2 Exploring Examples

Optimizations-by-example are effective in probing a domain for natural language designs, common expressions, and current pain points. As the novelty of our language stems from its emphasis of abstract data types for modelling using sets, sequences, and relations, the following examples aim to cover a sufficient amount of real-world usage for which we focus our design and optimization efforts.

2.1 The Birthday Book Problem

An older paper by Spivey [12] outlines a simple but fitting modelling problem we now explore at length: A birthday book tracks the relationship between the names of people and their birthdays. All names are assumed to be unique, yet two or more people may share the same birthday. Using this book, we attempt to answer two main questions:

- When is the birthday of a particular person?

- Who has a birthday on a given date?

These questions correspond to the lookup and reverse lookup tasks on a set of relations. In other words, to find the birthday of a person, we require a function $lookup : Name \to Date$. Likewise, we denote $lookup^{-1} : Date \to \{Name\}$, which returns a set of names that are born on a specific date. Then, our high level language should provide an optimized version of these functions that returns from $lookup$ or $lookup^{-1}$ as fast as possible.

2.1.1 Handling Relations: Many-to-One and Many-to-Many

The relational data structure of this question involves a many-to-one correspondence, where many names can map to the same date. In many programming languages, a straightforward method of modelling the data is to maintain two dictionaries, one for regular lookups and another for reverse lookups. As a more concrete example, let $birthday\_book$ denote an example relation of birthday book items, where keys are names and dates are represented by integers. A mapping of this book would be stored as:

$$birthday\_book = \{\text{Alice} \mapsto 25, \text{Bob} \mapsto 30, \text{Charlie} \mapsto 35, \text{Kevin} \mapsto 35\}$$

Then, the reverse mapping could be stored like so:

$$birthday\_book^{-1} = \{25 \mapsto \text{Alice}, 30 \mapsto \text{Bob}, 35 \mapsto [\text{Charlie, Kevin}]\}$$

Lookups and inverse lookups would search through these dictionaries as a hashmap accordingly.

When implemented in Python, this method performs well on small data types, but has the undesirable consequence of maintaining two sets of duplicated data for all operations. This duplication is especially inefficient for entries that contain large quantities of data, like nested records or tuples. For example, if Kevin were to be removed from the birthday book, both the regular birthday book dictionary and the list of names in the inverse dictionary would need to be modified.

To eliminate some of the duplicated data, we could instead transform the keys of one of the dictionaries to use pre-hashed values directly. For example, consider a hash function $h(x)$ that happens to return the following, provided our input of names:

$$\begin{aligned} h(\text{Alice}) &= 0 \\ h(\text{Bob}) &= 1 \\ h(\text{Charlie}) &= 2 \\ h(\text{Kevin}) &= 3 \end{aligned}$$

Further, let $collision\_resolver$ be some reasonable but deterministic hash collision resolution function. Then, the birthday book dictionary can contain the

following entries, with hashes in place of names:

$$birthday\_book = \{0 \mapsto 25, 1 \mapsto 30, 2 \mapsto 35, 3 \mapsto 35\}$$

To resolve any collisions from our pre-hash function h, we can define a modified lookup function as in Algorithm 1.

Algorithm 1 *lookup* using a pre-hashed dictionary

Require: n : Name
  $hash\_name \leftarrow h(n)$
  while True **do**
    $possible\_date \leftarrow birthday\_book[hash\_name]$
    if $possible\_date = None$ **then return** $None$
    end if
    $names\_for\_date \leftarrow lookup^{-1}(possible\_date)$
    if $n \in names\_for\_date$ **then return** $possible\_date$
    end if
    $hash\_name \leftarrow collision\_resolver(hash\_name)$
  end while

By keeping one complete dictionary, the other can rely on cross-dictionary lookups to resolve hashing collisions. Unfortunately, we still need to duplicate at least one set of either names or dates to resolve hashing collisions, but for larger data types, this method could use 25% less space with a little runtime cost.

Similar data structures with less overhead and clever elimination of duplicated data via pointers could prove useful in providing O(1) reverse lookups. For example, C++'s Boost.Bimap library [1, 11] stores pairs of values and provides efficient views for both forward and reverse lookups, with the caveat that the relation must be one-to-one. Alternatively, a post on StackOverflow [4] describes a solution with two maps of values to pointers (with types $A \to B*$ and $B \to A*$). In this case, data is stored in two sets of pairs that couple a concrete value with its opposing pointer. Then, the *lookup* function can dereference the pointer received from the $A \to B*$ to find the concrete value of B stored in the second set.

2.1.2 Extending the Problem with Two Birthday Books

Extending the problem, we demonstrate the ergonomics of set operators over real world models. With consideration for the bidirectional mapping setup from the previous questions, we seek to efficiently implement the following scenarios:

1. Two groups of friends become acquainted, each with an originally independent birthday book. Eventually, they wish to merge their records together, removing duplicate entries for mutual friends.

2. A subset of the original friend group eventually have a falling out and wish to separate from the main group. This problem then requires one birthday book to be separated into two.

3. When planning for a birthday party, a group of friends group wants to acknowledge not only the person's birthday, but any and all national holidays associated with that day.

A straightforward model for the first scene would assume that two groups A and B have an existing birthday book. Then the new birthday book will consist of the union of these two relations. Keeping in mind the invariant that names within friend groups must remain unique, the new set C can be constructed as in Algorithm 2.

Algorithm 2 Birthday Book Union

Require: $A, B : \text{Name} \times \text{Date}$
  $C \leftarrow A$
  for $b \in B$ **do**
    if $b.name \notin A$ **then**
      $C.add(b)$
    end if
  end for
  return C

The second setting could be resolved through a set difference of the small friend group from the large friend group. Let L be the remaining friends from the group separation. Then, Algorithm 3 contains the corresponding programmatic model.

Algorithm 3 Birthday Book Separation

Require: $A, B : \text{Name} \times \text{Date}$
  $L \leftarrow \{\}$
  for $a \in A$ **do**
    if $a.name \notin B$ **then**
      $L.add(a)$
    end if
  end for
  return L

Finally, the third situation could be expressed through the composition of two relations. Let $H : Holiday \times Date$ be a relation of holidays to their respective dates. Then, those people who have a birthday from set A coinciding with a holiday may be found by $C = A \,;\, H^{-1}$. Corresponding code is shown in Algorithm 4.

Algorithm 4 Birthday Book Composition

Require: $A : \text{Name} \times \text{Date}, H : \text{Holiday} \times \text{Date}$
  $C \leftarrow \{\}$
  for $a \in A$ **do**
    $h \leftarrow H.lookup^{-1}(a.date)$
    if $h \neq None$ **then**
      $C.add(\langle a.name, h.holiday \rangle)$
    end if
  end for
  return C

2.2 Other Combinations of Sets, Sequences, and Relations

In general, the feasibility of an efficient compiler for a very high level language relies on the idea that rewrite rules combined with efficient implementations of data types will reduce the runtime of complex set manipulation expressions. Dissecting several examples from a catalogue of Event B models revealed that the most common modelling expressions make use of conditions on sets, unions, intersections, and differences. Because the implementation of sets is the least constrained out of the "collection" types, we focus optimization efforts on manipulating those implementations for more efficient results.

Set construction notation of the form $\{\, elem \mid generators : conditions \,\}$ (eg. $\{\, x \mid x \in S : x > 0 \,\}$) appears often when dealing with models founded on discrete math. Further, operations on sets, like union, intersection, and difference, can be rewritten in set construction form:

$$S \cup T = \{\, x \mid x \in S \vee x \in T \,\}$$
$$S \cap T = \{\, x \mid x \in S \wedge x \in T \,\}$$
$$S \setminus T = \{\, x \mid x \in S \wedge \neg x \in T \,\}$$

Theoretically, using the lower-level boolean forms of these expressions should enable cleverer forms of execution that prevent unnecessary set constructions, either by way of lazy evaluation or condition manipulation.

In Python, set construction for the most part translates to a set comprehension of the form `{elem for elem in iterable if <condition>}` (eg. `{x for x in S if x > 0}`). Additionally, operators for union, intersection, and difference are `|`, `&`, and `-`.

We now list a few simple but common examples of set-based expression optimizations:

- A function f applied over the difference of two sets may be written in a modelling language as $\{f(x) | x \in S \setminus T\}$. Translated to more Python-like syntax, this becomes `{f(x) for x in S - T}`. The default implementation of such an expression, with no knowledge over the side effects of S, T, and f, would first construct a temporary set `text = S - T`, then perform `{f(x) for x in temp}`. However, a more efficient implementation

would rewrite the initial expression to `{f(x) for x in S if x not in T}`, effectively trading the overhead of temporary set construction for $|S|$ $O(1)$ lookup operations.

- Mapping over a difference of sets $S \setminus T$ where $T << S$ could also be performed by successive `[S.pop(x) for x in T]` operations, mutating S directly and avoiding another set construction. An implementation of `pop` could also move selected elements to one end of the list and keep track of an index of "deleted" elements for future operations. Likewise, $S \cup T$ with $T << S$ may be serviced with sequential `add` operations

- A similar index tracking idea could be used for sets that "consume" another set. For example, consider Algorithm 5, where P and C represent a producer and consumer set respectively. The P set may be needed elsewhere in the code, so we cannot destroy its contents to build C. In this case, internally representing P as an array of ordered elements and C as an index into the first n elements of P will be suitable. New elements are appended to the list representation of P (assuming no duplicates), so the addition and consumption of elements remain completely isolated while reducing duplicate data and repeated set constructions.

- Early stopping in quantification based expressions. For example, the statement $\forall x \mid x \in S \cap T \wedge p(x)$ would normally require construction of an intermediate set for $S \cap T$ and then a loop to check the property of all resulting elements. However, we do not necessarily need to construct any sets in this question. Instead, we can combine all terms into a single for-loop over only one of the sets, as seen in Algorithm 6. Similar set manipulations can be used for existential quantification.

Algorithm 5 Set Producer-Consumer

Require: $P : Set, C : Set, C = \emptyset$
  while $|P| \neq |C|$ **do**
    $e \leftarrow (P - C).pop()$ ... ▷ Function body, P, C read but not modified
    $C \leftarrow C \cup e$
    $P \leftarrow P \cup n$ ... ▷ P may produce new elements
  end while

Different flavours of optimizations for complex expressions and other "collection" data types will be added as needed, but the deep static analysis and code awareness is evidently integral to the optimization process. In the next section, we discuss the concepts of such a language, an optimization process, and an interface for continuous, extendable optimizations.

Algorithm 6 Early Stopping for Universal Quantification over Intersection

Require: S, T, p
  $M \leftarrow min(S, T)$ ▷ min compares the number of elements in a set
  for $x \in M$ **do**
    if then$\neg(x \in T \wedge p(x))$ **return** $false$
    end if
  end for
  return $true$

3 A TRS for a Very High Level Language

3.1 Rewriting Rules for Set Notation

In Section 2, we presented a selection of examples wherein expressive set notation for modelling could be rewritten for efficient code generation. Now, we extend this collection to optimize generic combinations of set operators as a complete collection of set-oriented rewrite rules. As we derive this system, assume each iterable data type holds efficient representations of size calculation and set membership.

To simplify all terms in the rewrite system, we introduce a rule to exhaustively reduce sets into set construction notation. Note that for all rules, capital letters represent sets and lowercase for elements and functions:

$$S \to \{\, x \mid x \in S \,\} \tag{1}$$

Then, our basic operators $\cup, \cap, \setminus$ can be reduced to combined construction notation with boolean conditions:

$$\{\, x \mid x \in S \,\} \cup \{\, y \mid y \in T \,\} \to \{\, x \mid x \in S \vee x \in T \,\} \tag{2}$$
$$\{\, x \mid x \in S \,\} \cap \{\, y \mid y \in T \,\} \to \{\, x \mid x \in S \wedge x \in T \,\} \tag{3}$$
$$\{\, x \mid x \in S \,\} \setminus \{\, y \mid y \in T \,\} \to \{\, x \mid x \in S \wedge x \notin T \,\} \tag{4}$$

A proof of correctness for these types of rules is directly given by the definition of each operator and variable substitution. We present the proof of union below, recognizing similar forms for intersection and set difference:

Proof.

$$\begin{aligned}
& S \cup T \\
=& \{\, x \mid x \in S \,\} \cup \{\, y \mid y \in T \,\} \\
=& \{\, z \mid (x \in S \wedge x = z) \vee (y \in T \wedge y = z) \,\} \\
=& \{\, z \mid z \in S \vee (y \in T \wedge y = z) \,\} \\
=& \{\, z \mid z \in S \vee z \in T \,\} \\
=& \{\, x \mid x \in S \vee x \in T \,\}
\end{aligned}$$

□

Although the proof is relatively simple, such proofs are of particular importance to ensuring correctness of variables used within one another. A generalized form of these rules, involving additional predicates $p, q : Any \to Boolean$, may be required. We observe that predicates joined to a generator $x \in S$ with $\vee$ are malformed, since any x satisfying the predicate may be valid.

$$\{\, x \mid x \in S \wedge p(x) \,\} \cup \{\, y \mid y \in T \wedge q(y) \,\} \to \{\, x \mid (x \in S \wedge p(x)) \vee (x \in T \wedge q(x)) \,\} \tag{5}$$

$$\{\, x \mid x \in S \wedge p(x) \,\} \cap \{\, y \mid y \in T \wedge q(y) \,\} \to \{\, x \mid x \in S \wedge x \in T \wedge p(x) \wedge q(x) \,\} \tag{6}$$

$$\{\, x \mid x \in S \wedge p(x) \,\} \setminus \{\, y \mid y \in T \wedge q(y) \,\} \to \{\, x \mid x \in S \wedge x \notin T \wedge p(x) \wedge q(x) \,\} \tag{7}$$

Often times, we find that functions are applied to the generated elements, similar to $\{\, f(x) \mid x \in S \,\}$. The above operator-based rules can therefore be expressed as:

$$\{\, f(x) \mid x \in S \,\} \cup \{\, y \mid y \in T \,\} \to \{\, f(x) \mid x \in S \vee f(x) \in T \,\} \tag{8}$$

$$\{\, f(x) \mid x \in S \,\} \cap \{\, y \mid y \in T \,\} \to \{\, f(x) \mid x \in S \wedge f(x) \in T \,\} \tag{9}$$

$$\{\, f(x) \mid x \in S \,\} \setminus \{\, y \mid y \in T \,\} \to \{\, f(x) \mid x \in S \wedge f(x) \notin T \,\} \tag{10}$$

A further generic form might be considered in which functions can be mapped over both operator arguments, however, the lack of relationship between a function mapped over the first set versus a possibly different function mapped over the second set restricts the amount of feasible algebraic manipulation. In this case, we first evaluate one operation fully to then apply one of the rules in the previous paragraph. Note that unions of this form gain no tangible benefit from an extra set construction, so we opt to join entries together as seen in the partial derivation of rewrite rule two. An optional predicate within each set construction may be carried through the rewrite rule as seen in equations five to seven, but for conciseness, we omit these rules for now.

$$\{\, f(x) \mid x \in S \,\} \cup \{\, g(y) \mid y \in T \,\} \to \{\, z \mid (x \in S \wedge f(x) = z) \vee (y \in T \wedge g(y) = z) \,\} \tag{11}$$

$$\{\, f(x) \mid x \in S \,\} \cap \{\, g(y) \mid y \in T \,\} \to T_2 := \{\, g(y) \mid y \in T \,\}; \{\, f(x) \mid x \in S \,\} \cap \{\, y \mid y \in T_2 \,\} \tag{12}$$

$$\{\, f(x) \mid x \in S \,\} \setminus \{\, g(y) \mid y \in T \,\} \to T_2 := \{\, g(y) \mid y \in T \,\}; \{\, f(x) \mid x \in S \,\} \setminus \{\, y \mid y \in T_2 \,\} \tag{13}$$

Additionally, a few specialized cases arise when the function f is injective and identical between set constructions. Further predicate clauses (as seen in equations five to seven) may be brought over as above and renamed as functions

of x.

$$\{ f(x) \mid x \in S \} \cup \{ f(y) \mid y \in T \} \to \{ f(x) \mid x \in S \vee x \in T \} \tag{14}$$
$$\{ f(x) \mid x \in S \} \cap \{ f(y) \mid y \in T \} \to \{ f(x) \mid x \in S \wedge x \in T \} \tag{15}$$
$$\{ f(x) \mid x \in S \} \setminus \{ f(y) \mid y \in T \} \to \{ f(x) \mid x \in S \wedge x \notin T \} \tag{16}$$

In targeting expressions that make use of set operators nested within set constructors, like $\{ x \mid x \in f(S, T) \}$, we create the following rules:

$$\{ x \mid x \in \{ y \mid y \in S \} \} \to \{ x \mid x \in S \} \tag{17}$$
$$\{ x \mid x \in \{ y \mid y \in S \wedge p(x) \} \wedge q(x) \} \to \{ x \mid x \in S \wedge p(x) \wedge q(x) \} \tag{18}$$
$$\{ x \mid x \in \{ y \mid y \in S \wedge p(x) \} \} \to \{ x \mid x \in S \wedge p(x) \} \tag{19}$$
$$\{ x \mid x \in \{ y \mid y \in S \} \wedge q(x) \} \to \{ x \mid x \in S \wedge q(x) \} \tag{20}$$

Finally, we can now target the boolean statements within set constructions. However, we first give some intuition towards the overarching principles behind non-trivial terms. Generally, breaking down each operation into set builder notation enables us to perform the following actions:

- One membership operator within each set of "and"-clauses will act as an element generator. Then, other clauses will act as conditions for an internal if-statement. For example, if the rewrite system observes an intersection of the form $\{ x \mid x \in S \wedge x \in T \}$, the set construction operation must iterate over at least one of S and T. Then, the other will act as a condition to check every iteration.

- A collection of "or"-clauses will generally be split into multiple sequential loops. Therefore, each clause must contain at least one element generator. Any "and"-clauses nested within an "or"-clause group will then act as a condition as specified above.

- Generators must be statements of the form $x \in S$, where x is used in the "element" portion of the set construction. Statements like $x \notin T$ or checking a property $p(x)$ must act like conditions since they do not produce any iterable elements.

- Any boolean expression for conditions may be rewritten as a combination of $\neg, \vee$, and $\wedge$ expressions. Therefore, by converting all set notation down into boolean notation and then generating code based on set constructor booleans, we can accommodate any form of predicate function.

The following rules formalize this intuition, where generators S (and T when

needed) are the smallest sets within a local "and"-clause:

$$\{\, f(x) \mid (x \in S \wedge p(x)) \vee (x \in T \wedge q(x)) \,\} \rightarrow \text{Algorithm 7} \tag{21}$$

$$\{\, z \mid (x \in S \wedge f(x) = z \wedge p(x)) \vee (y \in T \wedge g(y) = z \wedge q(y)) \,\} \rightarrow \text{Algorithm 8} \tag{22}$$

$$\{\, f(x) \mid x \in S \wedge p(x) \,\} \rightarrow \text{Algorithm 9} \tag{23}$$

And a specialized version of equation 21, preferring the use of intersections to reduce duplicate "or"-clause loops:

$$\{\, f(x) \mid (x \in S \wedge p(x)) \vee (x \in S \wedge q(x)) \,\} \rightarrow \{\, f(x) \mid x \in S \wedge (p(x) \vee q(x)) \,\} \tag{24}$$

Algorithm 7 RHS of Rewrite Rule 21

Require: x, f, S, T, p, q
  $A \leftarrow \emptyset$
  for $x \in S$ **do**
    if $p(x)$ **then**
      $A.add(f(x))$
    end if
  end for
  for $x \in T$ **do**
    if $q(x)$ **then**
      $A.add(f(x))$
    end if
  end for
  return A

Algorithm 8 RHS of Rewrite Rule 22

Require: x, y, f, g, S, T, p, q
  $A \leftarrow \emptyset$
  for $x \in S$ **do**
    if $p(x)$ **then**
      $A.add(f(x))$
    end if
  end for
  for $y \in T$ **do**
    if $q(y)$ **then**
      $A.add(f(y))$
    end if
  end for
  return A

Algorithm 9 RHS of Rewrite Rule 23

Require: x, f, S, p
  $A \leftarrow \emptyset$
  for $x \in S$ **do**
    if $p(x)$ **then**
      $A.add(f(x))$
    end if
  end for
  return A

Further rules are possible, provided relationships between sets are programmed alongside the programmed model. For example, subset properties enable the following:

$$\{ f(x) \mid x \in S \vee x \in T \} \text{ where } S \subseteq T \rightarrow \{ f(x) \mid x \in T \} \tag{25}$$

$$\{ f(x) \mid x \in S \wedge x \in T \} \text{ where } S \subseteq T \rightarrow \{ f(x) \mid x \in S \} \tag{26}$$

$$\{ f(x) \mid x \in S \wedge x \notin T \} \text{ where } S \subseteq T \rightarrow \emptyset \tag{27}$$

And with predicates for $\cup$ and $\cap$:

$$\{ f(x) \mid (x \in S \wedge p(x)) \vee (x \in T \wedge q(x)) \} \text{ where } S \subseteq T \rightarrow \{ f(x) \mid x \in T \wedge (p(x) \vee q(x)) \} \tag{28}$$

$$\{ f(x) \mid (x \in S \wedge p(x)) \wedge (x \in T \wedge q(x)) \} \text{ where } S \subseteq T \rightarrow \{ f(x) \mid x \in S \wedge p(x) \wedge q(x) \} \tag{29}$$

Assuming associativity and commutativity of boolean operations, the collection of equations presented here should be sufficient to cover all possible set constructions involving routine set operators $\cup, \cap, \setminus$ and boolean condition operators $\vee, \wedge, \neg$. To ensure our TRS is correct and complete, we now turn to an analysis of termination and critical pair resolution.

3.2 Completeness of Rewriting Set Notation

With the approximately 30 or so equations presented in Section 3, we now provide a proof of TRS termination through lexicographic path orders. Referring to the algorithm noted in Section 2.3.2 of our previous Term Rewriting Survey, we list all rules with associated passing conditions. For the purpose of lexicographic ordering, set notation can be rewritten as a function application on the generated elements and conditions (ie. $\{ x \mid x \in S \} = b(x, x \in S)$ where b represents set construction). Further, the costs of operations are organized as listed in Table 1

A full formal proof is rather verbose, but outlines for condition applications follow below:

- Equation 1: LHS function cost is 49, and RHS function cost is 10. Then, by Condition 2.2, closed set notation costs more than both children of the RHS.

Function structure	Cost
Set operators $\cup, \cap, \setminus$	50
Closed-set notation like S and T, symbolized by capital letters	49
Sequential statement operator ;	25
Set construction in the form of $b(e, c)$	10
Set element operator $\in$. For the sake of termination, closed-set notations within this operator have a cost of 0	5
$\wedge$ operator	3
$\vee$ operator	2
Element functions	1
Element variables	0
Fully algorithmic output	0

Table 1: Lexicographic function cost

- Equations 2-4, 5-11, and 14-16: Set operators have the highest cost. By Condition 2.2, LHS $>_{lpo}$ RHS
- Equations 12, 13: Set operations cost more than sequential statements. Then, the first child of the sequential statement costs less than a set operation, so repeated applications of Condition 2.2 are satisfied. For the second child of the sequential statement (which contains a set operator), Condition 2.3 is satisfiable since a function application is reduced to a lone variable.
- Equation 17: Directly satisfies Condition 2.1 since RHS is equal in cost to the nested child of the LHS.
- Equations 18-20: Intuitively, these equations eliminate nested sets and logically should become lexicographically simpler. Nested applications of Conditions 2.2 and 2.3 are sufficient.
- Equations 21-29: Require nested applications of Conditions 2.2 and 2.3, but are all straightforward simplifications of subterms.

Although most of this simplification process logically conforms to a termination proof, it is not without its flaws. The clause made in the set operator entry of Table 1 is the root of the fallacy: Unbounded application of the first equation results in a non-terminating TRS, since set construction notation for S can be infinitely nested upon itself. To alleviate this, we make use of rewriting strategies as discussed below.

With a semi-termination guarantee for our rewrite system, we can now focus on resolving overlapping rules to prove confluence. Critical pairs within this system are created primarily by specialized versions of general rules. For example, if we assume a boolean function $t(x) = x \wedge True$ can be substituted

within the TRS, rules like equations 2-4 become obsolete in favour of more general equations 5-7. Although the result of both paths will evaluate the same, we opt to separate these rules, preventing $t(x)$ from becoming a rewrite rule to prevent further complications with termination. Similarly, equations 8-10 can be considered specialized forms of equations 11-13, where $g(y) = y$, and 25-29 as subset-only rules for 21 and 23. Finally, the last critical pairs may be considered for 14-16 as specializations of 11-13.

While these specializations generate functionally identical code to their generic counterparts, additional properties may vastly increase efficiency and are thus desired. Making use of strategies, we can design a function that will first attempt to apply specialized equations whenever generic equations are considered. More specifically, we can break down our strategy into the following:

1. We first attempt to repeatedly apply equation 1 once for each closed set in the term's structure, using a bottom up approach. This strategy will effectively convert all closed set variables into set construction notation. Then, equations 17-20 can exhaustively perform early-stage cleanup and set de-nesting. For example, consider the derivation for an expression $S \cup T$: Rule 1 will apply once to both children, creating $\{ x \mid x \in S \} \cup \{ x \mid x \in T \}$, and then once on the entire $\cup$ operator, creating $\{ x \mid x \in (\{ x \mid x \in S \} \cup \{ x \mid x \in T \}) \}$. Then, rule 17 will eliminate the superfluous set construction, returning the desired value of $\{ x \mid x \in S \} \cup \{ x \mid x \in T \}$.

2. Stage 2 of this process breaks down all set operations into corresponding boolean-constructor notation, using equations 2-16. Although equations 12 and 13 add set operations back into the term, exhaustive application of equations 9 and 10 will eventually terminate.

3. Once all instances of set notation are eliminated, stage 3 focuses on logical condition manipulation. Equations 25-29 are vital for subset relationships, equation 24 for combining set generators, and 17-20 to flatten set constructions. Near the end of this stage, element generators will be chosen based on the size of sets within a group of "and"-clauses. For example, $\{ x \mid x \in S \wedge x \in T \}$ may choose either S or T as the eventual loop generator, depending on available set size information. Since this process is run at compile time, information on S and T may not be available, so one generator may be chosen at random.

4. Finally, equations 21 to 23 will enable efficient code generation for all possible set constructions. In implementation, the RHS algorithms will be replaced with LLVM or WebAssembly notation.

Now that we have a complete rewrite system for sets, we give attention to more concrete implementations of set-based loops.

4 A Backdrop for Concrete Sets

4.1 Sets in Other Languages

Before exploring our own concrete implementation of sets, we first examine the current state of work within this subdomain.

List comprehensions have been a standardized construct in several paradigms of programming, ranging from relational databases to functional languages. Generally, there are three methods of set-like comprehension:

1. SQL-like queries on relational databases of the form `SELECT entry FROM table WHERE property(entry)`.
2. Functional list comprehensions similar to set construction, like the Haskell statement `[x | x <- xs, p x]`.
3. Python-like comprehensions that are more akin to syntactic sugar for nested loops: `{x for x in xs if pp(x)}`.

Microsoft's implementation of SQL Server contains an entire engine dedicated to optimizing queries to and from a relational database. Optimization steps relevant to our domain make use of constant folding, early subexpression evaluation, and clause manipulation. However, since databases often deal with massive quantities of stable data chunks, concepts like indexing, intermediate evaluation, parallel query processing, batch evaluation, and distributed architectures are far more valuable to reducing overall runtimes. Compared to modelling domains, which often involve rapid changes of relatively small pieces of data, the overhead from all these optimization steps are unlikely to bear fruit.

In Haskell, list comprehensions are integral to the concise manipulation of individual elements. Essentially, comprehensions of this functional flavour act as syntactic sugar for `map` and `filter` functions, using plain discrete mathematics expressions. Haskell provides additional syntax for multi-generator comprehensions (eg. `[(a,b) | a <- as, b <- bs]`) and zipped multi-generator comprehensions (eg. `[(a,b) | a <- as | b <- bs]`). The former acts as a nested for loop, creating $|as| \times |bs|$ elements in total, while the latter processes elements from both sources in a single loop, generating only $min(|as|, |bs|)$ elements.

Similarly, Python places for-loop notation directly within the list, where multiple "for" keywords within a comprehension behave as directly nested loops. For example, a comprehension matching the same output as Haskell's multi-generator comprehension would be denoted by `[(a,b) for a in as for b in bs]`.

The language of comprehensions in our notation will draw directly from discrete mathematics but act as a hybrid between Haskell and Python. A key difference, however, is the optimization steps we are able to make through the compilation process. In Python, side-effectful functions and a very strict for-loop like syntax prevent rewrite rules from being effective mediums for optimization. Such functions also mean that execution order is constrained to the location in which the statement was written. On the other hand, Haskell performs lazy

```
intersect :: [a] -> [a] -> [a]
intersect [] _ = []
intersect _ [] = []
intersect xs ys =
    [x | x <- xs, any ((==) x) ys]
```

(a) A simplified version of Haskell's intersection function, adapted from [2]

Require: x, f, S, p
  $A \leftarrow \emptyset$
  for $x \in S$ **do**
    if $p(x)$ **then**
      $A.add(f(x))$
    end if
  end for
  return A

(b) The output of intersection rewrite rule 23.

Figure 1: Comparison of Haskell's definition of intersection versus our rewrite rule optimization

evaluation of elements, which may avoid complete construction of intermediate sets but requires an unpredictable amount of overhead in both memory and running time.

4.2 Running Time of Optimized Sets

By applying optimizing rewrite rules at compile time, we retain the benefit of Haskell-like set comprehension semantics while cutting out the uncertainty of lazy evaluation. The optimizations themselves are similar in essence to Haskell's standard library, as shown in Figure 1 but we attempt to perform them earlier, more aggressively, and with modelling-oriented semantics. To this point, while Haskell's use of `any` requires folding the entire list of `ys` and thus runs in $O(n)$, an efficient check for set membership through a hash table would require only $O(1)$ time.

In the set of rewrite rules that generate code from Section 3.1, we optimize under the assumptions that set membership is efficient and loops will always iterate over the smallest viable set. Then, all loops defined by the rewrite set generally take time proportional to $O(n)$, provided non-membership conditions only require constant time. More specifically, union operations for sets S and T with one generating element should only require at most $O(|S| + |T|)$ time, since loops are executed sequentially. Then, intersections only use at most $O(min(|S|, |T|))$ time, generating over the shorter of the two sets. Finally, set differences always loop over the first operator, requiring $O(|S|)$ time.

Memory usage for all operations is likely more than an entirely lazy approach, but far better compared to eager procedural languages. Most of the time, rewrite rules can eliminate intermediate set creation from temporary unions, intersections, and set differences. However, in more complex environments for intersection and set difference, as in equations 12 and 13 respectively, it becomes more prudent to make use of an intermediate set for the chance of future optimizations. For example, an unoptimized implementation of $\{ f(x) \mid x \in S \} \cap \{ g(y) \mid y \in T \}$ would construct both arguments and

then calculate the intersection. However, by constructing only one of the arguments and incorporating that construction in the evaluation of the second, we effectively eliminate an entire loop over the larger set.

5 A Very High Level Language

Although the details of implementation, representation, and optimization are still in their infancy, we will eventually require a vehicle to rapidly test optimizations as they appear. This project therefore involves the creation of a compiler pipeline that supports swift iteration, extension, and execution. To reiterate, the goals of our very high level language are as follows:

- Provide a natural, expressive, easy-to-use medium of communicating modelling specifications.
- Compile programmed models into efficient, correct, and verifiable code.

The first of these goals is primarily motivated by language design and largely impacted by real-world user testing. For the target audience of domain experts who may not necessarily be the most adept at low level programming, this objective is essential to creating a viable solution. Then, our work on the second goal justifies the existence of the language over a dialect of another easy-to-use language. The overarching belief is that data types carry powerful innate semantics that could be harnessed for aggressive optimization, providing efficient executable code.

5.1 Supported Operators and ADTs

Within the language, we care about optimization for two types of data: primitive values like integers, floating point numbers, strings, etc, and collections, like sets, sequences, and relations, as mentioned in previous sections.

On primitives, all standard operations are supported:

- For booleans, operators like $\vee$, $\wedge$, $\implies$, $\iff$, and $\impliedby$ are part of the language core, with future additions for quantification statements like $\exists$ and $\forall$. Enabling these operators enhances the expressivity of set constructions and maintains parity between the specification and implementation source code as much as possible.
- Integers and floats are coupled with standard mathematical operations, from addition to exponentiation. Most numerical operations can be implemented efficiently by way of functions, so we are not as concerned with numbers as a core part of the language. Depending on the implementation of our optimizer, the rewrite system may be extendable with custom rules particularly for complex mathematical expressions. However, we disregard most of these optimizations for now, since low level numerical optimization is made available through optimizers like LLVM.

- Strings as specialized sequences of characters with syntactic sugar.

And on collections, we cover all common set notations, providing a library of functions where operators have no suitable ASCII translation:

- Sets are central to model-oriented programming. Set operators like $\cup$, $\cap$, $\setminus$, etc. will all be high priority optimization targets, making use of boolean-level representations and loose implementation restrictions to ensure high performance code generation. Most of these optimizations will make use of rewriting, though the level at which rewriting occurs may be subject to change based on initial tests. For example, providing an interface in the rewrite system at the language level could equip library developers with tools to optimize functions without the use of a lower level language. However, encapsulating rewrite rules as part of the integrated compiler pipeline may allow higher quality optimizations. For example, decisions to represent sets as a list, or even an ordered list, might only be possible with lower level AST manipulations and precise static analysis.

- Sequences are essentially refined multi-sets with less flexibility. Because the elements in a sequence must maintain a particular order, any benefits from choosing between different implementation methods may be limited. Operations like concatenation, appending, prepending, calculating the number of unique elements, counting the number of element repetitions, can all be optimized through static analysis.

- Relations have similar operations to sets, but by nature are more complex. Relation composition, inverse, functions on the domain and range of a relation, and even treating key-value pairs as sets may be necessary. Since elements are unordered but may be repeated, the implementation of relations may greatly influence the efficiency of execution.

All of the types mentioned here have been meticulously studied as fundamental parts of many common languages and algorithms. However, extensive static analysis-based rewriting has not been seen on many modern languages, largely due to innate complexity of custom types. By encouraging the use of a select few, highly optimized abstract data types instead of broad optimizations across concrete types, we can provide a simple interface with efficient execution.

5.2 Compiler Pipeline

5.2.1 Grammar

As the first step in processing the language, a CFG is used to convert plain text into a structured AST. For this draft, we have selected a grammar based on Python's familiar syntax [9] and discrete mathematics based on a book by Gries and Schneider [3]. Aside from standardized language expression operators, control flow statements, and function definitions, the main features of this grammar are as follows:

- Additional operators (and overloaded operators) dedicated to sets, sequences, and relations. With static typing, these will enable expressive statements closely resembling modelling specifications.
- Set construction-like comprehension statements for sets, sequences, and relations.
- Record types (structs) in place of classes, implemented as relations.
- Enums as static sets to collect names into a specific type. For example, when modelling a stop light intersection, the three stages of a light may be most clearly modelled as an enum with 3 states: red, yellow, and green.

The complete current grammar is available on GitHub.

5.2.2 AST and AST transformations

The next step of our compiler is translating the CFG into a malleable AST. The Python library Lark [5] provides several choices of parsers, lexers, tree traversal methods, and top level interfaces to ease the process of parsing. From the grammar, Lark generates a generic AST `Transformer` that is traversable through methods corresponding to each grammar rule.

With the frontend of the compiler complete, we now move on to the optimization stage. This stage is perhaps the most complex to design and implement, since these decisions directly impact the balance of optimization capabilities and extensibility. For the subset of the language related to sets and set notation, we have implemented a simplified version of the rewrite strategy described at the end of Section 3.2. The current implementation collapses all sets connected by operators $\cup, \cap$, and $\setminus$ into a single set connected by booleans. Then, generators are extracted from a list of candidates (ie. from the right side of the vertical bar in standard set notation, p in the structure $\{ x \mid p \}$). Unions, or "or"-clauses within set notation, generally require two sequential set loops, while top-level "and"-clauses for intersections and set differences share a single construction loop. Specifically, the prototype compiler is compliant with rewrite rules 1-4, 17, 21, and 23.

After rewriting terms to eliminate all superfluous operators, the compiler must decide on the best representation of data, given the context of relevant operations, object lifetimes, and expected memory usage. Finding generic optimizations across all possible use cases of data types may very well be undecidable, requiring complete knowledge of the code's runtime behaviours before the code has actually executed, but optimizing common operations may reach an acceptable level of improvement. We may further make use of modelling domain knowledge to heuristically optimize data representations (ex. bidirectional maps for the birthday book problem).

More concretely, LLVM defines several different types of sets that may be optimal depending on actions and expected size [6]. The built-in `SmallSet` saves large quantities of memory, for instance, but comes with linear access times.

`Dense-` and `Sparse-`Sets on the other hand, provide access through hash tables and use more memory to prevent several `malloc` calls every insertion. Ordered elements within `Vector`s can enable extremely fast insertions and reasonably fast lookups. For our compiler, efficient lookups are preferred over efficient insertions, since we aim to trade intermediary set constructions for extra lookups for each element.

The remaining compilation step then lowers the optimized AST into LLVM IR. The Python library `llvmlite` [7] provides APIs to generate IR statements directly from within Python, enabling straightforward code generation through tree traversals. However, since LLM uses static single assignment form and C++-level constructs for high efficiency, a post-optimization concrete AST may be necessary to replace chunks of the AST with a closer-to-executable representation.

6 Future Work

Before we conclude this paper, it is important to illustrate two key points of research that should be resolved for a very high level language but are beyond the scope of this project.

In addition to set construction notation, we observed that a noticeable number of modelling examples make use of quantification logic within set generators. Currently, rewrite rules to optimize quantifier logic are not part of the Section 3 system. Instead, the language currently treats such expressions naively, looping over a given domain to find existential and universal quantification.

Second, generators for multiple variables within a comprehension must be optimized. A naive implementation may start with a double nested loop over both variable generators, however, any optimizations to reduce the number of superfluous loops will result in large leaps in runtime efficiency. Any opportunity to remove $O(n^2)$ and beyond code should be followed aggressively, but such a topic will need to be explored in detail, beyond the work of this current project.

7 Conclusion

The beginning stages of building a new programming language are arguably one of the most exciting and fundamental parts of the compiler development process. Exploring theoretical optimizations isolated from specific architectures and system limitations can lead to drastic improvements in overall performance with zero additional hardware costs. For our very high level language, we attempt to combine focused optimization techniques on a discrete mathematics language familiar to all computer scientists and software developers.

References

[1] Matias Capeletto. *Boost C++ Libraries: Boost.Bimap*. 2012. URL: https://www.boost.org/doc/libs/1_85_0/libs/bimap/doc/html/index.html (visited on 04/27/2025).

[2] *Core data structures and operations - Data.List*. 2025. URL: https://hackage.haskell.org/package/base-4.21.0.0/docs/Data-List.html#v:intersect (visited on 04/30/2025).

[3] 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.

[4] *Is there a more efficient implementation for a bidirectional map?* 2019. URL: https://stackoverflow.com/questions/21760343/is-there-a-more-efficient-implementation-for-a-bidirectional-map (visited on 04/27/2025).

[5] *Lark-Parser/lark*. 2025. URL: https://github.com/lark-parser/lark (visited on 04/27/2025).

[6] *LLVM Programmer's Manual*. 2025. URL: https://llvm.org/docs/ProgrammersManual.html#ds-set (visited on 04/30/2025).

[7] *LLVMLite: A lightweight LLVM python binding for writing JIT compilers*. 2025. URL: https://github.com/python/cpython/blob/main/Objects/dictobject.c (visited on 04/30/2025).

[8] 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.

[9] *Python 3.13 Reference: Full Grammar specification*. 2025. URL: https://docs.python.org/3/reference/grammar.html (visited on 04/27/2025).

[10] *Python/cpython: dictobject.c*. 2025. URL: https://github.com/python/cpython/blob/main/Objects/dictobject.c (visited on 04/27/2025).

[11] Boris Schaling. *The Boost C++ Libraries: Chapter 13. Boost.Bimap*. 2025. URL: https://theboostcpplibraries.com/boost.bimap (visited on 04/27/2025).

[12] 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.