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

Anthony Hunt

April 29, 2025

Contents

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 [**spivey1989birthday**] 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 [**boostBimap1**, **boostBimap2**] 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 [**bimapStackOverflow**] describes a solution with two maps of type $A \rightarrow B*$ and $B \rightarrow A*$. 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 $A \rightarrow B*$ 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.

- Calculating the intersection of sets with a sorted representation may be done in $O(m+n)$ time through a mergesort-like step, or even in $O(n \log \frac{m}{n})$ using exponential search [**sortedIntersectionStackOverflow**].
- 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.

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.

3 Rewriting 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.

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

$$\{\, f(x) \mid x \in S \,\} \cup \{\, y \mid y \in T \,\} \to \{\, f(x) \mid x \in S \vee f(x) \in T \,\} \tag{5}$$
$$\{\, f(x) \mid x \in S \,\} \cap \{\, y \mid y \in T \,\} \to \{\, f(x) \mid x \in S \wedge f(x) \in T \,\} \tag{6}$$
$$\{\, f(x) \mid x \in S \,\} \setminus \{\, y \mid y \in T \,\} \to \{\, f(x) \mid x \in S \wedge f(x) \notin T \,\} \tag{7}$$

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 2.

$$\{ f(x) \mid x \in S \} \cup \{ g(y) \mid y \in T \} \rightarrow \quad \begin{array}{l} \{ z \mid (x \in S \wedge f(x) = z) \\ \quad \vee (y \in T \wedge g(y) = z) \} \end{array} \tag{8}$$

$$\{ f(x) \mid x \in S \} \cap \{ g(y) \mid y \in T \} \rightarrow \quad \begin{array}{l} T_2 := \{ g(y) \mid y \in T \}; \\ \{ f(x) \mid x \in S \} \cap T_2 \end{array} \tag{9}$$

$$\{ f(x) \mid x \in S \} \setminus \{ g(y) \mid y \in T \} \rightarrow \quad \begin{array}{l} T_2 := \{ g(y) \mid y \in T \}; \\ \{ f(x) \mid x \in S \} \setminus T_2 \end{array} \tag{10}$$

Additionally, a few specialized cases arise when the function f is injective and identical between set constructions.

$$\{ f(x) \mid x \in S \} \cup \{ f(y) \mid y \in T \} \rightarrow \{ f(x) \mid x \in S \vee x \in T \} \tag{11}$$

$$\{ f(x) \mid x \in S \} \cap \{ f(y) \mid y \in T \} \rightarrow \{ f(x) \mid x \in S \wedge x \in T \} \tag{12}$$

$$\{ f(x) \mid x \in S \} \setminus \{ f(y) \mid y \in T \} \rightarrow \{ f(x) \mid x \in S \wedge x \notin T \} \tag{13}$$

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 rule:

$$\{ x \mid x \in \{ y \mid y \in S \} \} \rightarrow \{ x \mid x \in S \} \tag{14}$$

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

$$\{\, f(x) \mid (x \in S \wedge p(x)) \vee (x \in T \wedge q(x)) \,\} \rightarrow \text{Algorithm 6} \tag{15}$$

$$\{\, z \mid (x \in S \wedge f(x) = z \wedge p(x)) \vee (y \in T \wedge g(y) = z \wedge q(y)) \,\} \rightarrow \text{Algorithm 7} \tag{16}$$

$$\{\, f(x) \mid x \in S \wedge p(x) \,\} \rightarrow \text{Algorithm 8} \tag{17}$$

And a specialized version of equation 15, preferring the use of intersections to reduce sequential "or"-clause loops:

$$\{\, f(x) \mid (x \in S \wedge p(x)) \vee (x \in S \wedge q(x)) \,\} \rightarrow \{\, f(x) \mid x \in S \wedge (p(x) \vee q(x)) \,\} \tag{18}$$

Algorithm 6 RHS of Rewrite Rule 15

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

4 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.

Algorithm 7 RHS of Rewrite Rule 16

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 forreturn A

Algorithm 8 RHS of Rewrite Rule 17

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

- 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.

4.1 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 will be supported:

- For booleans, operators like $\vee$, $\wedge$, $\implies$, $\iff$, and $\impliedby$ will be part of the language core, in addition to quantification statements like $\exists$ and $\forall$. Enabling these operators will enhance the expressivity of set constructions and maintain parity between the specification and implementation source code as much as possible.

- Integers and floats will contain standard mathematical operations, from addition to exponentiation. Most other operations can be implemented efficiently by way of functions, so we are not as concerned. Range notation of the form $x \in 1..n$ may prove useful. 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.

4.2 Compiler Pipeline

4.2.1 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 [**pythonGrammar**] and discrete mathematics based on a book by Gries and Schneider [**griesAndSchneider**]. 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. These may be implemented as named tuples or relations, depending on implementation testing.
- 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.

4.2.2 AST and AST transformations

The next step of our compiler is translating the CFG into a malleable AST. The Python library Lark [**lark**] 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. In consideration for the design, we present two possibilities:

1. Create a sub-language to define rewrite rules for the entire modelling language, using variables to represent abstraction over internal structures. This language would look similar to the language we are attempting to build, but domain-specific to rewriting. For example, we could create a list of generic rewrite rules in the following manner:

```
a: int
b: int
c: int | c != 0

# BEGIN REWRITE RULES
```

```
a + b - b = a
a + 0 = a
a * c / c = a
a * 0 = 0
```

This notation is simple for integer optimization, but could be extended to include iterable actions and structures:

```
S: Set
T: Set
R: Relation[Any, A] # type of A is generic
a: A
E: Set | len(E) == 1

# BEGIN REWRITE RULES
S - T + T = S # set difference and union
S := S + E = S.add(E.pop())
S := {x | x in S - T} = S := {x | x in S if x not in T}

for (fst, snd) in R:
  if snd == a:
    return fst
=
# Requires efficient inverse structure
return (R ** -1)[a]

# Requires assumption about the data structure's internal API
R ** -1[a] = R.lookup_reverse(a)
```

The advantages of this approach are clear: new rewrite rules can be specified in programming language format, allowing developers to extend optimizations in a manner similar to writing regular library functions. Then, once these rules are parsed into AST form and passed to a rewrite system, they can be broadly applied across an entire program. Assuming this interface can provide all the dials and switches necessary to provide optimization steps, expert users can quickly advance the language for entire codebases. However, providing in depth rewrite rules for function inlining, determining properties of set implementations (ie. intersection of internally sorted sets) may be difficult to implement or otherwise extremely verbose. Both the target language AST and rewrite rules would need some context attachment to ensure correct optimizations are made in both the actions and representation of a program.

2. Embedding rewrite steps directly in the compiler through specific, targeted, AST translations. For example, writing the first addition rule of method one directly in Python may look something like `Sub(Add(id1(), id2()), id2()) = id1()`. This effectively skips the "middle-man" grammar but requires direct AST setup within the compiler source text. Results

of function applications may carry necessary corresponding information that could allow for deeper optimizations. For example, expressing the optimal intersection of two internally sorted sets may look something like:

```
class SortedSet(Set):
  properties=["sorted"]
  values: list

# Rough idea for a modified version of the last step in mergesort
def merge_sorted_lists(lst1: list, lst2: list) -> list:
  longest_list, other_list = lst1, lst2 if len(lst1) > len(lst2) else lst2,
  ret_list = []
  i, j = 0, 0
  while i < len(longest_list):
    if j >= len(other_list):
      break

    if longest_list[i] == longest_list[j]
      ret_list.append(longest_list[i])
      i += 1
      j += 1
    elif longest_list[i] > other_list[j]:
      j += 1
    else: # longest_list[i] < other_list[j]
      i += 1
  return ret_list

def rewrite_sorted_intersection(lhs: Intersection[SortedSet, SortedSet]) ->
  # Caller would need to check for the "sorted" property
  return SortedSet(merge_sorted_lists(lhs.left, lhs.right))
```

Both methods take inspiration from realized rewrite systems: the mini-language approach is used in GHC through `Haskell Core` [**haskellCore**], and the latter through Egglog, an equality saturation library [**egglogPython**, **eggPaper**]. Since we expect the number of example-based optimizations to increase with continued data gathering from relevant domains, keeping a handwritten list of such optimizations may be necessary. By selecting a rewrite rule input style similar to method one, the handwritten records may serve doubly as a fundamental part of our optimizer.

The particular system of rewriting remains undecided due to upfront time investments and struggles in working with niche tools. An initial test with Egglog's equality saturation techniques revealed inflexible APIs that failed to accommodate custom types and non-standard operators. Exploring an older but more mature version of equality saturation through the Rust library Egg [**eggRust**], we again faced challenges related to type restrictions and purely AST-based rewrite rules. In the coming weeks, a dedicated rewriting system

may be chosen or written from scratch based on equality saturation or rewriting with strategies.

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).

5 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.