

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

Anthony Hunt

April 27, 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
2.2	Other Combinations of Sets, Sequences, and Relations	5
3	A Very High Level Language	6
3.1	Supported Operators and ADTs	7
3.2	Compiler Pipeline	8
3.2.1	Grammar	8
3.2.2	AST and AST transformations	9
4	Conclusion	11

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 most 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 of 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 $lookup : Name \rightarrow Date$. Likewise, we denote $lookup^{-1} : Date \rightarrow Name$, which returns a list 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 python, a straightforward method of modelling the data is to maintain two dictionaries:

```

1 from typing import TypeAlias, Callable
2
3 Name: TypeAlias = str
4 # Represent days of the year from 0 to 365, for simplicity
5 Date: TypeAlias = int
6
7 birthday_book: dict[Name, Date] = {
8     "Alice": 25,
9     "Bob": 30,
10    "Charlie": 35,
11    "Kevin": 35,
12 }
13
14 # Overriding the dict class would keep this reverse mapping hidden
15 birthday_book_inverse: dict[Date, list[Name]] = {
16     25: "Alice",
17     30: "Bob",
18     35: ["Charlie", "Kevin"],
19 }
20
21 def birthday_book_lookup(name: Name) -> Date | None:
22     return birthday_book.get(name)
23
24 def birthday_book_reverse_lookup(date: Date) -> list[Name]:
25     return birthday_book_inverse.get(date, [])

```

Now, `birthday_book_lookup` and `birthday_book_reverse_lookup` correspond to $lookup$ and $lookup^{-1}$ respectively. This method performs reasonably well on simple data types, but results in maintaining two dictionaries with duplicated data. 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.

Although Python uses hashes for dictionary lookup under the hood [10], references to an object must be preserved to resolve hash collisions, in which case Python compares key values with `==`. Duplication is particularly bad for entries that contain large quantities of data, like nested records or tuples.

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:

```

1 # Assume the `hash(x)` function corresponds to these input/output
  pairs:
2 # hash("Alice") = 0
3 # hash("Bob") = 1
4 # hash("Charlie") = 2
5 # hash("Kevin") = 3
6
7 # Linear probing. This is a suboptimal collision resolver
8 # but only used as part of the example
9 def collision_resolver(hash: int) -> int:
10     return hash + 1
11
12 birthday_book: dict[int, Date] = {
13     0: 25,
14     1: 30,
15     2: 35,
16     3: 35,
17 }
18
19 birthday_book_inverse: dict[int, list[Name]] = {
20     25: "Alice",
21     30: "Bob",
22     35: ["Charlie", "Kevin"],
23 }
24
25 def birthday_book_lookup(name: Name) -> Date | None:
26     hashed_name = hash(name)
27
28     # If two names happen to have the same hash, need to resolve the
29     # collision manually
30     # This case is extremely rare though, so we expect only 1
31     # execution iteration for this loop
32     while True:
33         possible_date = birthday_book.get(hashed_name)
34         # No entry found for this name
35         if possible_date is None:
36             return None
37
38         names_corresponding_to_date = birthday_book_reverse_lookup(
39             possible_date)
40         if name in names_corresponding_to_date:
41             return possible_date
42
43         hashed_name = collision_resolver(hashed_name)
44
45 def birthday_book_reverse_lookup(date: Date) -> list[Name]:
46     # Let python take care of any hash collisions here
47     return birthday_book_inverse.get(date, [])

```

By keeping one complete dictionary, the other can rely on cross-dictionary lookups to resolve hashing collisions. Unfortunately, we still needed 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. Note that in testing many-to-one relations that “collided” often (ie. had the same key but different values), the running time of our naive hash implementation took far longer than plain bidirectional mapping, likely due to a poor choice in

collision resolution function.

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, or slightly less efficient to allow one-to-many. Alternatively, a post on StackOverflow [6] 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.

2.2 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 $\{\text{elem} \mid \text{set generators} : \text{conditions}\}$ (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:

$$\begin{aligned} S \cup T &= \{x \mid x \in S \vee x \in T\} \\ S \cap T &= \{x \mid x \in S \wedge x \in T\} \\ S \setminus T &= \{x \mid x \in S \wedge \neg x \in T\} \end{aligned}$$

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 $\{\text{elem for elem in iterable if <condition>}\}$ (eg. $\{x \text{ for } x \text{ in } S \text{ if } x > 0\}$). 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 $\{f(x) \mid x \in S \setminus T\}$. Translated to more Python-like syntax, this becomes $\{f(x) \text{ for } x \text{ in } S - T\}$. 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 `temp = S - T`, then perform $\{f(x) \text{ for } x \text{ in temp}\}$. 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 [4].
- Mapping over a difference of sets $S \setminus T$ where $T \ll S$ 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 $T \ll S$ may be serviced with sequential `add` operations
- A similar index tracking idea could be used for sets that “consume” another set. For example, consider the below statements:

```

1 consumed = set()
2 producer = {1, 2, 3, 5, 7, 12, 15} # Any set with elements
3 while len(consumed) != len(producer):
4     elem_to_consume = (producer - consumed).pop()
5     ... # producer set used elsewhere
6     consumed |= {elem_to_consume}
7     if bool(random.getrandbits(1)):
8         producer |= {random.random()}

```

The `producer` set may be needed elsewhere in the code, so we cannot destroy its contents to build `consumed`. In this case, internally representing `producer` as an array of ordered elements and `consumed` as an index into the first n elements of `producer` will be suitable. New elements are appended to the list representation of `producer` (assuming no duplicates), so the addition and consumption of elements remain completely isolated while reducing duplicate data and repeated set constructions.

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 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.

3.1 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 \Leftarrow 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.

3.2 Compiler Pipeline

3.2.1 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 [5]. 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 at our GitHub.

3.2.2 AST and AST transformations

The next step of our compiler is translating the CFG into a malleable AST. The Python library Lark [7] 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:

```
1 a: int
2 b: int
3 c: int | c != 0
4
5 # BEGIN REWRITE RULES
6 a + b - b = a
7 a + 0 = a
8 a * c / c = a
9 a * 0 = 0
```

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

```
1 S: Set
2 T: Set
3 R: Relation[Any, A] # type of A is generic
4 a: A
5 E: Set | len(E) == 1
6
7 # BEGIN REWRITE RULES
8 S - T + T = S # set difference and union
9 S := S + E = S.add(E.pop())
10 S := {x | x in S - T} = S := {x | x in S if x not in T}
11
12 for (fst, snd) in R:
13     if snd == a:
14         return fst
15 =
16 # Requires efficient inverse structure
17 return (R ** -1)[a]
18
19 # Requires assumption about the data structure's internal API
20 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:

```

1 class SortedSet(Set):
2     properties=["sorted"]
3     values: list
4
5     # Rough idea for a modified version of the last step in
6     # mergesort
7     def merge_sorted_lists(lst1: list, lst2: list) -> list:
8         longest_list, other_list = lst1, lst2 if len(lst1) > len(
9             lst2) else lst2, lst1
10        ret_list = []
11        i, j = 0, 0
12        while i < len(longest_list):
13            if j >= len(other_list):
14                break
15
16            if longest_list[i] == longest_list[j]:
17                ret_list.append(longest_list[i])
18                i += 1
19                j += 1
20            elif longest_list[i] > other_list[j]:
21                j += 1
22            else: # longest_list[i] < other_list[j]
23                i += 1
24        return ret_list
25
26 def rewrite_sorted_intersection(lhs: Intersection[SortedSet,
27     SortedSet]) -> SortedSet:
28     # Caller would need to check for the "sorted" property
29     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` [8], and the latter through Egglog, an equality saturation library [3, 13]. 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 [2], 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. bi-directional maps for the birthday book problem).

4 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] *Crate Egg*. 2025. URL: <https://docs.rs/egg/latest/egg/> (visited on 04/27/2025).
- [3] *Egglog Python*. 2025. URL: <https://egglog-python.readthedocs.io/latest/> (visited on 04/27/2025).
- [4] David Eisenstat. *What is the fastest algorithm for intersection of two sorted lists?* 2016. URL: <https://stackoverflow.com/questions/40527584/what-is-the-fastest-algorithm-for-intersection-of-two-sorted-lists> (visited on 04/27/2025).
- [5] 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.
- [6] *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).
- [7] *Lark-Parser/lark*. 2025. URL: <https://github.com/lark-parser/lark> (visited on 04/27/2025).
- [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.
- [13] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. “egg: Fast and extensible equality saturation”. In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). DOI: 10.1145/3434304.