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

Anthony Hunt

April 27, 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 ecosystem and a reasonable expectation for most efficiency-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. For languages that lower the skill floor of software development, becoming 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 present a few critical findings that will serve well in a long-term implementation.

The rest of this paper explores theoretical optimizations on select examples, along with 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 language designs, 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 the birthday book problem as follows: 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 today?

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 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:

```python
from typing import TypeAlias, Callable

Name: TypeAlias = str
# Represent days of the year from 0 to 365, for simplicity
Date: TypeAlias = int

birthday_book: dict[Name, Date] = {
  "Alice": 25,
  "Bob": 30,
  "Charlie": 35,
  "Kevin": 35,
}

# Overriding the dict class would keep this reverse mapping hidden
birthday_book_inverse: dict[Date, list[Name]] = {
  25: "Alice",
  30: "Bob",
  35: ["Charlie", "Kevin"],
}

def birthday_book_lookup(name: Name) -> Date | None:
  return birthday_book.get(name)

def birthday_book_reverse_lookup(date: Date) -> list[Name]:
  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 [**pythonDictImplementation**], 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:

```python
# Assume the `hash(x)` function corresponds to these input/output pairs:
# hash("Alice") = 0
# hash("Bob") = 1
# hash("Charlie") = 2
# hash("Kevin") = 3

# Linear probing. This is a suboptimal collision resolver
# but only used as part of the example
def collision_resolver(hash: int) -> int:
  return hash + 1

birthday_book: dict[int, Date] = {
  0: 25,
  1: 30,
  2: 35,
  3: 35,
}

birthday_book_inverse: dict[int, list[Name]] = {
  25: "Alice",
  30: "Bob",
  35: ["Charlie", "Kevin"],
}

def birthday_book_lookup(name: Name) -> Date | None:
  hashed_name = hash(name)

  # If two names happen to have the same hash, need to resolve the collision manually
  # This case is extremely rare though, so we expect only 1 execution iteration for this loop
  while True:
    possible_date = birthday_book.get(hashed_name)
    # No entry found for this name
    if possible_date is None:
      return None

    names_corresponding_to_date = birthday_book_reverse_lookup(possible_date)
    if name in names_corresponding_to_date:
      return possible_date

    hashed_name = collision_resolver(hashed_name)

def birthday_book_reverse_lookup(date: Date) -> list[Name]:
  # Let python take care of any hash collisions here
  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 very 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, or slightly less efficient to allow one-to-many. Alternatively, a post on StackOverflow [**bimapStackOverflow**] describes a solution with two maps of type $A \to B*$ 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.

2.2 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 set implementations for more efficient results.

Set construction notation of the form {elem | set generators : conditions} (eg. $\{x|x \in S : x > 0\}$) 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:

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

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 the below statements:

```
consumed = set()
producer = {1, 2, 3, 5, 7, 12, 15} # Any set with elements
while len(consumed) != len(producer):
  elem_to_consume = (producer - consumed).pop()
  ... # producer set used elsewhere
  consumed |= {elem_to_consume}
  if bool(random.getrandbits(1)):
    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, representing `producer` as an array of 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.

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 rapid 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 above.

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 $\cup$, $\cap$, $\setminus$, $^{-1}$, 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 niche static analysis.

- Sequences are essentially refined multi-sets with less flexibility. Because the representation of lists dictates all elements maintain a particular order, any benefits from choosing between different implementation methods may be one sided, like arrays versus linked lists. 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 any concrete type, we can provide a simple interface with efficient execution.

3.2 Grammar

As the first step in processing the source for any 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 at our GitHub.

3.3 Compiler Pipeline

3.3.1 Parsing with Lark

3.3.2 Equality Saturation with Egg(log)

3.3.3 Rewrite Rules for Common Expression Structures

4 Conclusion