Design of an Efficient Map-Based Programming Language

NIELS KRUK, University of Twente, The Netherlands

Supervisor: Peter Lammich

There are a lot of different kinds of programming languages and paradigms. None of them seem to focus solely on maps. Here, map refers to the mathematical definition of a map: a function that associates the elements from one set with another. The goal of this study is to explore the idea of a map-based programming language and understand what its advantages and disadvantages could be. For this study, we designed a prototype programming language called MPL (Map Programming Language), where the only composite types are maps. Then we compared the language to other languages by the time and space complexity of common data structures and ease of use. The comparison of common data structures indicates that even though there is some overhead for some structures, the amortised time and space complexity are equivalent to optimal implementations in other languages. These results indicate that map-based languages are a viable option when enough time is spent optimising them until the overhead compared to other languages is reduced.

1 INTRODUCTION

The development of programming languages over the past few decades has been driven by the need to address existing limitations or explore alternative approaches to programming. This has resulted in numerous paradigms and strategies aimed at improving programming.

In mathematics, a map is a function that associates each element of a set with an element of another set. In computer science, the concept is used in a quite similar manner: a map can be used along with an index element to access the element the map associates the index with. When also taking the memory of a computer into account, a map is a region of memory and a function that indexes into that region of memory.

Despite the large number of programming languages, there do not seem to be any that have been specifically designed to fully optimise the use of maps. To address this gap, this paper proposes the design of a map-based programming language prototype. We evaluate this prototype on its map-based paradigm by limiting the use of composite data types to only maps. This study explores the capabilities of a map-based language and identifies how this paradigm might improve programming and for which types of problems other paradigms are better. To explore the idea of a map-based programming language, we answered the following research question:

RQ: What should a map-based programming language look like, and what are the advantages and disadvantages of using a map-based programming language?

We answered the research question by answering the following sub-questions:

RQ1: Which types of problems benefit from being solved with a map-based programming language?

RQ2: For which types of problems are highly optimised maps less efficient than conventionally used data structures?

RQ3: How does the time and space efficiency of common operations compare to that of other mainstream imperative languages?

RQ4: What syntax, semantics, and features are needed for a map-based programming language to be usable and readable?

RQ5: What are the advantages and disadvantages of using a map-based programming language instead of a library that adds map optimizations to a language?

To answer these sub-questions, we wrote a compiler for a map-based language prototype called MPL[7] using the LLVM infrastructure. Using this compiler, we were able to test the language with various programming problems and do empirical tests to aid in answering the questions.

2 RELATED WORK

Related literature was gathered using Google Scholar, and IEEE using search terms such as “map”, “map based”, “programming language”, “perfect hashing”, “dynamic perfect hashing” and “runtime analysis”.

In mathematics and computer science, a lot of research has been done on how to use maps efficiently. In 1977, there was already an article published about perfect hash functions and how they could be computed for small static key sets [10]. Over the years, a lot of research has been done. In 2007, a study found a method to find perfect hash functions for larger static key sets. Now it is feasible to find a perfect hash function for a static key set that contains more than a billion keys [3]. There has also been research on the case where the key set is not static. In 1984, a study found a method to do dynamic perfect hashing [5].

Even though a lot of research has been done into maps, we have not been able to find a study about the advantages and disadvantages of a map-based programming language.

Research has been done on the use of maps to solve specific problems and how maps compare to other methods, some examples are [2, 6, 8]. These examples are proof of the usefulness of maps. As research continues to be done on maps, the usefulness of map-based programming languages will continue to increase as well.

3 LANGUAGE DESCRIPTION

MPL is a map-based programming language prototype that is designed to have an easily understandable syntax. In this language, the amount or type of white space is not significant. The top-level scope of a program is reserved for definitions. A program needs one main function, which will be the entry point.

3.1 Types

MPL is a strongly typed language. The whole program is type checked during compilation and will give errors if types are used incorrectly. The primary types in the language are: int, bool, char,

TScIT 39, July 7, 2023, Enschede, The Netherlands

float and void (unit type). Besides these primary types there are 3 types of maps.

Map: The default map. It is similar to hashmaps in other languages. The amortised time complexity for inserting and retrieving is O(1), and the space complexity of the map is O(n). The specific implementation is not optimised yet in MPL. In a real map-based language, a highly optimised map would be important, but this falls outside the scope of this prototype and is not of importance to the conclusions of this research.

Perfect Map: A map with constant keys known at compile time and a fast, minimal, perfect hash function. When keys are known at compile time, a collisionless minimal size map can be constructed, reducing the allocated space and lookup time.

Multi variable type map: A map with variable value types. It is useful for bundling associated values together when they are of different types.

The notation of the normal map type is the key set with a right arrow to the value set, all enclosed in square brackets. The normal map type is often used with the void (unit) type. When the value set is void, the map will function like a set. If the key set is void, the map either has void as a key and holds a value, or the map is empty and does not hold a value. This is functionally similar to a nullable pointer.

The notation of the perfect map is similar to the normal map, the difference is the perfect keyword in front of the map type. What in other languages would be a string is in MPL a collisionless minimal size map that maps indices to characters.

Lastly, the notation of the multi variable type map is an identifier enclosed in square brackets; before it can be used, it needs to be defined somewhere in the top level of the program.

```
[int -> int]
[int -> void]
[void -> int]
perfect [int -> char]
[LinkedListNode]
```

Example top level definition of the LinkedListNode map type:

```
[LinkedListNode] = [
    NEXT -> [LinkedListNode]
    VALUE -> int
]
```

3.2 Variables

A variable is created by writing a type, identifier, equal sign, and expression. All primary types are allocated on the stack. All Maps are allocated on the heap; this means that map-type variables on the stack only hold pointers. After a variable is defined, it can be reassigned by writing the identifier of the variable, an equal sign, and an expression for the new value.

```
int i = 1
bool b = true
char c = 'c'
float f = 1234.5436
perfect [int -> char] myString = "hello"
i = 2
```

3.3 Expressions

Expressions follow the default mathematical order of precedence. All variables in expressions are passed by value. The map-type variables only hold references to the heap where the map data is stored. This means that the map data is not copied, only the memory address of the map data.

3.4 Control flow

MPL has if-else statements, while loops, and for each loops. If statements and while loops have standard syntax.

```
if ifBoolExpression {
    ...
} else elseIfBoolExpression {
    ...
} else {
    ...
}

while whileBoolExpression {
    ...
}
```

The syntax for the for loop is the for keyword, a variable name for the keys, an arrow, a variable name for the values, the in keyword, a map, and a code block.

```
for keyVariable -> valueVariable in myMap {
    ...
}
```

3.5 Memory management

MPL uses manual memory management. Only maps are heap-allocated and have to be manually managed. New memory can be allocated by using the new keyword, the type, and a variable name to bind the newly allocated map to.

```
new [int -> int] myMap
new [LinkedListNode] myLinkedList
```

To free memory, the free keyword is used in combination with a map variable.

```
free myMap
free myLinkedList
```

This will only free that specific map. If a map contains other maps as keys or values and you want to free those as well, then you need to free those maps manually:

```
    new [int -> [int -> int]] myMapMap
    ...
    for _ -> innerMap in myMapMap {
        free innerMap
    }
    free myMapMap
```

3.6 Functions

Functions need to be defined at the top level of the program. They are defined and called like this:

```
fn int fib(int a){
    if a < 2 {
        return 1
    }
    return fib(a-2) + fib(a-1)
}
```

When functions have more arguments, comma-separated lists are used in the function definition and when calling the function.

3.7 Map methods

Map variables can be used to call the following methods:

int size() : Returns the number of key-value pairs in the map.
void insert(Key, Value): Updates or inserts a key-value pair into the map.
void clear(): Clears all entries from the map and resizes to the minimum size.
Value get(Key): Return the value associated with the key if the key exists in the map, otherwise this function has undefined behaviour.
bool remove(Key): tries to removes a key-value pair from the map and returns whether it could remove the key-value pair. (This function does not work with perfect maps)
[void -> Value] getMaybe(Key): If the map contains this key, get the value, store that value in another map that has key type void, and return this map. If the map does not contain this key, return an empty map with key type void.

3.8 Perfect maps

Currently, there are two methods of constructing perfect maps. The first one is a trivial perfect map made from a string literal, as seen before. The second one describes all the keys and values, and the compiler will brute-force a perfect hash function with those keys. The syntax for finding a perfect map from strings to integers is the following:

```
find perfect [perfect [int -> char] -> int] words = [
    "i" -> 0
    "am" -> 1
    "hashing" -> 2
    "strings" -> 3
    ...
]
```

Now, throughout the whole program, the global map variable `'words'` can be used like any other map.

3.9 Comments

Comments are written between slashes and asterisks like this:

```
/* this is a
   multi line comment */
```

4 METHODOLOGY

This section details the steps taken to answer each of the research questions. First, a compiler that compiles the simple map-based language prototype MPL to LLVM IR was programmed.

4.1 Answering RQ1, RQ2 and RQ3

In order to answer the first three research questions, we solved as many programming problems as time allowed. We cover problems with different intended solution strategies; this way, we will be able to identify the strengths and weaknesses of the language. We then compared the solution with another language that uses LLVM, specifically Rust. After this, a time complexity analysis of both solutions was done, checking for execution speed and memory usage. We combined the result of the quantitative analysis with the personal experience of solving the problems in both languages to answer RQ1 and RQ2.

To also answer RQ3, we benchmarked the operations used in the solutions against the equivalent operations in the other language. We answered RQ3 by comparing the relative difference between the results from the benchmarks.

4.2 Answering answer RQ4

In order to answer RQ4, experience programming in the language was needed. This made it possible to see which features were missing or unnecessary, and how the syntax and semantics could be changed to make the language more readable and usable.

While answering the other sub-research questions, a lot of time was spent programming in the map-based programming language. During this period, we had the opportunity to refine the feature set, syntax, and semantics. This allowed us to answer RQ4.

4.3 Answering RQ5

To answer RQ5, we tried out pthash[9] and rust-phf[4], two popular libraries that apply the same optimizations that have been added in the map-based language prototype: the compile-time perfect minimal hash map. After trying these libraries, we will explain the differences we perceived.

5 RESULTS

In this section, we give the collected data relevant to the specific research questions and some of our thoughts and interpretations of the results.

5.1 Results for RQ1

Most programming languages have some sort of map type and can implement a solution to a problem in the same way a map-based

language can. This means that map-based programming languages do not bring revolutionary methods to solving problems. The small advantages that map-based languages could have are good defaults and a high level of optimisation for maps, making the map-based language faster or more memory efficient than other languages when the usage of maps is an efficient strategy for the problem. An example of a better default is choosing a fast hashing function as opposed to a cryptographically secure hashing function. Rust, which uses a cryptographically secure hashing function by default, has a lot of overhead when using a hashmap in cases where it does not need to be cryptographically secure. An example of an optimisation is the perfect map type in MPL, which is faster and more memory efficient compared to languages that do not have such maps.

5.2 Results for RQ2

In MPL, all problems that are usually solved using arrays in other languages have some overhead. The same holds for data structures that are internally represented by an array, such as stacks, heaps, or dynamically sized arrays. This overhead is caused by the need to hash the keys, check if the keys are equal, and check if the key-value pair is still valid. This overhead could be removed by allowing the creation of trivial, perfect minimal hash maps at runtime; these have the key set $\{0, 1, \cdots, n-1\}$ and the identity function as a hash function. This removes the need to check if the keys are equal and if the key-value pair is still valid. After compiler optimisations, the identity hashing function will be optimised out. Then indexing into a trivial, perfect, minimal hash map will be equivalent to indexing into an array. This can then be used to implement other data structures that are internally represented by an array without incurring overhead. This means that there are no programming problems where highly optimised maps are less efficient than conventionally used data structures because these data structures can be implemented using maps without overhead.

5.3 Results for RQ3

In this research, we benchmarked the following operations on a 64-bit Windows 10 computer in a WSL Ubuntu 20.04.6 LTS environment with an AMD Ryzen 5 2600 Six-Core Processor and 8 GB of RAM. Both MPL and Rust were compiled with the highest optimisation level.

- Map and set insertion (Table 1 & 2)
- Map lookup (Table 3)
- Linked list and vector push and pop (Table 6, 7 & 8)
- Sorted binary tree insertion and removal (Table 4 & 5)

Because the amortised time complexity of insertion and lookup using a hash map is O(1) and the space complexity of using a hash map is O(n), all other data structures can be created by substituting the underlying arrays that languages usually use with a hash map in a map-based language. This means that if all operations are implemented optimally, they will have the same time complexity as optimal implementations of these data structures in other languages. The following benchmarks are still useful for measuring the overhead caused by the use of maps instead of arrays.

5.3.1 Map and set insertion. Table 1 verifies that insertion has an amortised O(1) time complexity. The main causes of variance in the speed for a different number of keys are the percentage of the capacity filled at the end of the benchmark and cache misses. If the time is measured just before and just after a rehashing, the latter will include the time for allocating more memory and for copying over all the data. This will make a big difference in the average time per insertion. The chance of cache misses will grow as the capacity of the maps grows. This will slowly increase the average time per insertion.

It is interesting to note that our language prototype is a lot faster than Rust, even though our maps are not really optimised yet. This is caused by the fact that Rust's default hash function is SipHash 1-3 [1], which is a cryptographically secure hash function that protects against HashDoS attacks, which can be useful but is most of the time not necessary.

Table 2 can be used to estimate how long it has been since the last rehash. The table shows that MPL's set and map just finished a rehash after 10^7 keys and are about to do another rehash after 10^8 keys. This information can aid in understanding the results from Table 1.

5.3.2 Map lookup. Table 3 verifies that MPL's map has an amortised O(1) time complexity. Here it can again be seen that Rust's default, SipHash 1-3, is not optimal for speed benchmarks.

5.3.3 Linked list and vector push and pop. The MPL map benchmark in Table 6 and Table 7 use a map that inserts the value that needs to be pushed and uses as key the current size of the map. When popping, it returns the value at key = size − 1 and removes that key-value pair from the map.

For the linked lists, pushing seems to be two and a half times as slow for MPL's FILO and about five times as slow for MPL's FIFO. For popping, both MPL's FILO and FIFO are slightly more than four times as slow. We speculate that this is caused by the lack of a pointer type in MPL. This forces the programer to use maps for referencing memory addresses, which is substantial overhead when there has been no effort spent on optimising this usage of maps. MPL's map benchmark uses a data structure functionally similar to Rust's VecDeque benchmark; however, as also seen in Table 1, inserting into MPL's maps has substantial overhead compared to Rust's Vecs.

In Table 8 we can see that MPL's linked lists need more than four times as much space. This could also explain why it is significantly slower compared to Rust's linked list. This table also verifies that all the data structures have O(1) space complexity, as expected.

5.3.4 Sorted binary tree insertion and removal. Rust does not have a sorted binary tree data structure by default. So we programmed the same implementation in both Rust and MPL. This gives a good comparison of the relative speed between the two languages when they execute the same algorithm. On average, inserting and removing from a binary tree should be $O(\log n)$. In Table 4, both implementations seem to follow this time complexity. MPL's tree is slower for the same reason that its linked lists are slower. Maps are not that well optimised yet to serve as pointers, and Table 5 shows that more memory needs to be allocated for MPL's binary trees.

Table 1. Fill Benchmark (ns/key)

Keys	MPL set	MPL map	MPL set float	Rust Vec	Rust Set	Rust Map
10^2	17.87	19.01	33.17	7.48	47.81	48.15
10^3	9.16	10.04	37.23	3.20	58.18	62.52
10^4	9.81	13.76	37.59	2.77	56.59	58.42
10^5	13.26	21.86	36.44	2.60	52.37	52.14
10^6	18.34	36.61	68.88	4.19	92.91	111.52
10^7	25.59	44.17	104.06	4.66	150.93	157.67
10^8	19.66	32.06	98.88	4.81	195.02	214.68

Table 2. Space Benchmark (byte/key)

Keys	MPL set	MPL map	MPL set float	Rust Vec	Rust Set	Rust Map
10^2	12.37	23.09	12.37	10.48	9.44	18.40
10^3	14.93	28.17	14.93	8.22	14.38	28.72
10^4	17.09	32.28	17.09	13.11	11.47	22.94
10^5	19.48	36.79	19.48	10.49	9.18	18.35
10^6	22.19	41.91	22.19	8.39	14.68	29.36
10^7	25.27	47.74	25.27	13.42	11.74	23.49
10^8	12.79	24.17	12.79	10.74	9.40	18.79

Table 3. Lookup Benchmark 50% hit (ns/key)

Keys	MPL map	Rust map
10^2	44.18	21.14
10^3	52.33	15.59
10^4	37.36	15.99
10^5	35.91	21.25
10^6	38.46	64.70
10^7	39.04	111.75
10^8	39.21	157.04

Table 4. Sorted Binarytree Benchmark (ns/key)

Keys	MPL insert	MPL remove	Rust insert	Rust remove
10^2	320.69	469.52	37.51	265.80
10^3	449.26	774.51	68.51	392.29
10^4	695.79	1352.46	135.15	574.11
10^5	1469.73	1984.38	418.12	862.65
10^6	3087.29	2320.85	1205.55	1129.48

Table 5. Sorted Binarytree Space Benchmark (byte/key)

Keys	MPL	Rust
10^2	43.49	24.08
10^3	43.05	24.01
10^4	43.00	24.00

5.4 Results for RQ4

Almost all The features that are currently implemented aid in the usability or readability of the language. The only currently implemented feature about which we are not sure whether it is a good addition to the language is the 'getMaybe' method. The initial purposes of 'getMaybe' were to protect against a key not existing in a map and to avoid having to index into a map twice, which happens in the common programming pattern of first checking for the key's existence and then looking up the value. Another big advantage of 'getMaybe' is that the method 'get' can be reserved for the case when the key is assumed to be in the map. This makes the key equality check unnecessary when called using a perfect hash map. There are two problems with the current approach. The first is that because 'getMaybe' creates a new map, which makes memory management inconvenient. It is easy to forget to free the result, and when reassigning a variable with 'getMaybe' you will most of the time first need to free the current value to avoid memory leaks. Here is an example:

```
fn int problemsWithGetMaybe([int->int] A){
    new [void->int] maybevalue
    if condition {
        /* free a just created map */
        free maybevalue
        maybevalue = A.getMaybe(1234)
    }
    if maybevalue.size() == 0 {
        maybevalue.insert(0)
    }
   /* Instead of just returning first copy the value,
       then free the map and only then return */
    int returnvalue = maybevalue.get()
    free maybevalue
    return returnvalue
}
```

Table 6. Push Benchmark (ns/key)

Size	MPL linked list FILO	MPL linked list FIFO	MPL map	Rust LinkedList	Rust VecDeque
10^2	37.59	71.24	13.10	15.67	10.72
10^3	40.77	66.25	12.42	15.76	5.01
10^4	38.51	80.02	15.64	15.85	4.13
10^5	37.06	71.58	13.38	15.37	3.76
10^6	39.07	81.64	17.82	15.88	4.57
10^7	41.97	81.88	26.74	15.96	5.95

Table 7. Pop Benchmark (ns/key)

Size	MPL linked list FILO	MPL linked list FIFO	MPL map	Rust LinkedList	Rust VecDeque
10^2	62.93	48.99	4.88	11.45	3.91
10^3	59.77	47.39	3.95	11.43	3.48
10^4	60.19	53.98	4.97	13.07	3.44
10^5	63.13	57.81	3.90	10.87	3.52
10^6	66.26	69.14	3.92	16.95	3.62
10^7	64.51	62.51	4.84	14.72	3.70

Table 8. Push-Pop Data Structures Space Benchmark (bytes/key)

Size	MPL linked list FILO	MPL linked list FIFO	MPL map	Rust LinkedList	Rust VecDeque
10^2	34.49	34.34	23.09	8.24	10.56
10^3	34.05	34.03	28.17	8.02	8.22
10^4	34.00	34.00	32.28	8.00	13.11
10^5	34.00	34.00	36.79	8.00	10.49
10^6	34.00	34.00	41.91	8.00	8.39
10^7	34.00	34.00	47.74	8.00	13.42

One way to improve this is by allowing if and else statements to evaluate to a value in expressions like a ternary operator. You could have the value equal to the result of the last line, like this:

```
[void->int] maybevalue = if condition {
        A.getMaybe(1234)
    } else {
        new [void->int]
    }
```

Another possibility is to modify the arguments of 'getMaybe' to take the map object it will insert the value into if it exists; this will then prevent new map allocations. This would look something like this:

```
new [void->int] maybevalue
if condition {
    A.getMaybe(1234, maybevalue)
}
```

The last possibility we came up with that specifically addresses this issue is that you could add another type to the language that can hold the returned value of 'getMaybe' and allocate it on the stack. If this option is chosen, it is important to pay attention to clarifying which maps are stack allocated and which maps are heap allocated and need to be manually freed.

The other problem with 'getMaybe' is that even though it solves the problem of indexing multiple times in the specific scenario where you want to check for existence and then return the value, it does not help with the other scenarios where you would index twice, such as retrieving and deleting or retrieving and updating a key-value pair. One way to solve this is to add a method for all combinations of actions you could want to do to a key-value pair like: 'getRemove', 'getMaybeInsertWith', 'updateWithLambdaFunctionGet', 'insertIfEmptyOtherwiseRemove', etc. Although technically feasible, this is probably not the best way to handle this issue. A better way is to bind a variable to a code block that represents the key-value pair and add some methods to update, remove, or retrieve the data. This could look something like this:

```
with pair from map[key]  {
    if pair.isValid(){
        pair.insert(pair.get() + 1)
    }else{
        pair.insert(0)
    }
}
```

With both this solution and the previous one when using a function like 'updateWithLambdaFunctionGet' it is important to think of a way to handle the case where the map gets updated in other locations than the pair. This could trigger a rehashing and invalidate the pair pointer. The safest way is to disallow updating the map within this code block. Another strategy could be to keep track of rehashes and update the pair pointer after a rehash.

5.4.1 Potential improvements. We were considering extending the syntax for finding a perfect hashmap with the following:

```
    ...
    "programming" -> 14
    "language" -> 15
] with (strKey) {strKey.get(0), strKey.size()}
```

This syntax would indicate that the combination of the first character and the size can uniquely identify all keys. This means that it is possible to use only these values to calculate the hash, which will make hashing faster. It is also sometimes possible to let the compiler calculate how to optimally and uniquely identify the keys. This is impossible when a complex expression is needed, such as `{strKey.get((strKey.size()-1)/2)}` to retrieve the middle value from the string. Another disadvantage of having the compiler calculate this is that it makes assumptions about what valid keys are. In the case of `{strKey.get(0), strKey.size()}` the assumption is that there exists a zeroth element. This is true for all strings except the empty string. As a programmer, you can then keep this in mind while using the map. If the compiler finds a set of values it can extract to uniquely identify the keys, you do not have control over the requirements for valid keys. This is not a problem when you only look up keys that are in the map, but a lookup with keys that are not inside the map could cause undefined behaviour.

5.5 Results for RQ5

After having tried out both pthash and rust-phf we can say that there are no big disadvantages to using these libraries instead of a map-based language. When you use a library, it needs to be manually installed as a dependency instead of already having the features accessible. This is not a big disadvantage in the case of pthash and rust-phf because the installation process is quite easy for both of these libraries. Another potential disadvantage could be that the maps are constants. The only things that have to stay constant are the keys because the hash function depends on them. If you wanted to count specific words in a text, you could use the hash function and the array of keys from the static program memory and allocate a separate array to store the values in dynamic memory. This could look something like this:

```
find perfect [perfect [int -> char] -> _ ] words = [
    "i"
    "am"
    "hashing"
    "strings"
]
...
    new [words -> int] count
    for key -> _ in words {
        count.insert(key, 0)
    }
    while nextWordInText {
        count.insert(count.get(nextWord) + 1)
    }
...
```

In both pthash and rust-phf a separate array will have to be allocated to keep track of the number of occurrences of the words. This has some slight spatial and temporal overhead, which will most likely not be a problem.

6 CONCLUSION

After exploring the idea of a map-based programming language, we came to the conclusion that it is a viable paradigm. Using a map-based programming language is usually not a big advantage or disadvantage compared to other languages. There are specific problems where it can be advantageous to use a map based programming language. An example is when a hash map is useful and the keys can be known at compile time, then a map-based programming language can find an optimal hashing function for that set of keys. In MPL, there is a lot of overhead compared to other languages, which could be disadvantageous if performance is important. This is, however, a specific property of MPL. In general, map-based programming should be able to optimise away most of the overhead. If someone is willing to use libraries when using other languages, the advantage a map-based language could have goes away, as the problems it tries to solve can also be solved using a library. Then it is up to preference as to which language or paradigm should be chosen.

For further work, research could be done on specific optimisations that could improve the efficiency of a map-based language. Another option is to propose novel syntax and semantics that could improve readability or ease of use. Any research into new hashing algorithms and their advantages and disadvantages could contribute to improving map-based languages. And lastly, research into what maps can be useful for can increase the overall usefulness of map-based languages.

REFERENCES

[1] Jean-Philippe Aumasson and Daniel J Bernstein. 2012. SipHash: a fast short-input PRF. In *Progress in Cryptology-INDOCRYPT 2012: 13th International Conference on Cryptology in India, Kolkata, India, December 9-12, 2012. Proceedings 13*. Springer, 489–508.

[2] Ankita Bihani and Anupriya Gagneja. 2017. Graph Processing Library in Rust. (2017). http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm

[3] Fabiano C. Botelho and Nivio Ziviani. 2007. External perfect hashing for very large key sets. *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management* (2007). https://doi.org/10.1145/1321440.1321532

[4] Steven Fackler, Yuki Okushi, and Austin Bonander. 2023. rust-phf. https://github.com/rust-phf/rust-phf.

[5] Michael L. Fredman, János Komlós, and Endre Szemerédi. 1984. Storing a sparse table with 0 (1) worst case access time. *J. ACM* 31, 3 (1984), 538–544. https://doi.org/10.1145/828.1884

[6] Chanchal Khemani, Jay Doshi, Juhi Duseja, Krapi Shah, Sandeep Udmale, and Vijay Sambhe. 2019. *Solving Rubik's Cube Using Graph Theory: ICCI-2017*. 301–317. https://doi.org/10.1007/978-981-13-1132-1_24

[7] Niels Kruk. 2023. MPL. https://github.com/Pinchoboo/language.

[8] Ryan Marcus. 2023. Learned Query Superoptimization. arXiv:arXiv:2303.15308
[9] Giulio Ermanno Pibiri and Roberto Trani. 2021. PTHash: Revisiting FCH minimal perfect hashing. *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval* (2021). https://doi.org/10.1145/3404835.3462849
[10] Renzo Sprugnoli. 1977. Perfect hashing functions. *Commun. ACM* 20, 11 (1977), 841–850. https://doi.org/10.1145/359863.359887