A LANGUAGE FOR GENERATING FLAT DATA STRUCTURES

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Michael Xing

May 2026

ABSTRACT

Data structures represented with pointers are typically less performant than the same data flattened into an array. Optimizations such as turning an array of structs into a struct of arrays have been known to improve performance by taking advantage of spatial locality, and writing code that operates directly on serialized data eliminates the overhead of serialization and deserialization. However, pointer representations are typically more ergonomic and intuitive to code against compared to flattened and packed data.

I introduce a new domain specific language that allows users to define data types with natural pointer representations. The compiler then creates flattened versions of the data in Rust, with convenience methods that allow the user to write code with minimal changes. The language enables the user to customize the specific layout of the flattened version of each data structure while preserving the Rust interface, making it easy to experiment with layouts to find the best performance. When tested on code that operates on real genomic data, significant speed improvements were found by swapping out the pointer data structures for the flattened version, with minimal code changes needed.

BIOGRAPHICAL SKETCH

Michael Xing was born and raised in Charlotte, North Carolina and received his Bachelor of Arts in Computer Science, Mathematics, and Physics with a minor in Game Design from Cornell University in 2021. After graduation, he spent three years in Seattle as a Software Engineer at Microsoft before returning to Cornell University in the fall of 2024. In his time at Cornell, he served on the course staff of CS 2110 and CS 2112 for eleven semesters, where he planned and created many lab materials and discussion activities and guest lectured multiple times.

Outside of his academic activities, Michael enjoys traveling, riding roller coasters, playing the piano, and video games. After graduation, he will return to Seattle to work at Microsoft as he continues figuring out what he wants to do with his life.

This document is dedicated to everyone who has helped me along my journey, especially Priya, for convincing me that graduate school was worth trying.

ACKNOWLEDGEMENTS

To look back and take stock of the monumental team effort required to get me here is to realize that I can't begin to wrap my head around the sheer number of people without whom I would never have made it a fraction as far. Still, I have a list, undoubtedly incomplete, of gratitude I owe to people whose massive help so freely given I can never hope to repay to any meaningful degree.

Special thanks to Professor Adrian Sampson for all the assistance, patience, and insight that helped me make it through this project. I'd also like to thank Professor Andrew Myers for his continued guidance throughout my degree, Professor Andrew Campana for advising my minor, Professors Walker White and Michael Clarkson for their advice and assistance during my application, and Professors Matthew Eichhorn, Curran Muhlberger, David Gries, Dexter Kozen, and Andrew Myers again for being joys to work for.

Additional thanks to my family, including my sister Jane (I remembered you this time), for being supportive through all these years. Education is cumulative and it's difficult to imagine myself still standing here without the backing they've unconditionally provided through it all.

To all my fellow M.S. students with high moral standards and also David Lin, it's been a pleasure being in a cohort with y'all. Shoutout to Annabel Baniak and David Lin as the three of us held down the fort in the office for all those long hours.

I also want to express appreciation towards my team at Microsoft who was willing to put up with my absence for two years. Thank you to Eric Williams, Tina Biradar, and Prasanna Padmanabhan for keeping me around, and to all of Team Nighthawk for your patience as I left your Teams messages unread for weeks at a time.

My crazy idea of coming back to school would never have happened if not for all my friends who shared their experiences about the program with me in long conversations through the night, ultimately helping me decide to give this a try. Thank you to Nathaniel Bannister and especially Priya Srikumar. You are still missed.

Finally, to all the friends I've made, peers I've worked with, TAs who have given me help and feedback, professors I've learned from, students I've taught, and everyone else I've interacted with here at Cornell, thank you for making this school all that it is. At times, it feels I've spent too many years at this school, but looking back, six years in, at all that has happened, I wouldn't have traded it for the world.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The majority of high-level languages naturally express data structures such as trees and graphs using pointers between objects in memory. While semantically similar to the data's logical representation and easy to program against, the performance of data structures tend to improve when flattened into linear arrays. Doing so presents two primary benefits. First, arrays are better able to exploit spatial locality in caches. Second, linear packed data representations can be written directly to disk without needing to be separately serialized, which also means they can be directly `mmap`'d back into memory for zero-cost deserialization.

However, to write code operating on flattened data structures requires manually tracking object indices or manual pointer manipulation. The unwieldy nature of programming in this way has largely relegated bespoke handwritten flat data structures to the domain of niche optimizations.

Many solutions have been proposed to make working with flat data structures easier. Specialized compilers exist that compile code written in a traditional pointer-based style into flattened versions, but only support custom languages with limited features and adoption. Others have written runtime libraries to provide simple interfaces for operating on packed data in existing programming languages, which incur runtime cost and are often still limited in their supported data structures, as many only support trees.

I propose a new domain specific language that takes a hybrid approach. The language allows users to specify the shape of general pointer based data struc-

tures. My language supports arbitrary data types with pointers between them, able to represent complex cyclic graphs and other general data shapes. Once defined, my compiler converts the natural data type descriptions into flattened data types in Rust and generates convenience classes and methods to operate on the flattened types.

This hybrid approach allows for the simplicity and performance of compile-time flattening without the limitations of writing logic in a custom language. Once the data structure is generated, the result is a plaintext human-legible Rust file, which can be inspected and even edited if desired. All logic is then written in Rust, using the full capabilities of a popular language and optimizations available to an industrial-strength compiler.

Using my language, simple Rust programs can be converted into flattened versions and achieve an over 26% performance increase with minimal code changes in Rust, requiring only a few field accesses be replaced with equivalent getter method calls.

In addition, my language exposes some customization options, allowing the user to specify how fields should be flattened into arrays via the use of annotations. None of the customizations affect the Rust interface used to operate on the resulting data structure. This exposes powerful optimization opportunities, as programmers can toggle between various representations, like array-of-structs versus struct-of-arrays, merely by tweaking annotations and making no further code changes.

CHAPTER 2

RELATED WORK

The idea of flattening data structures into a deterministic, packed format that can be directly written to and read from disk is not new. This chapter describes some of the prior work and ways in which my language differs.

The most relevant work is by Vollmer et al [10], who introduced the Gibbon compiler that transforms programs traversing pointer-based trees into programs that operate on packed trees. Gibbon creates packed trees by flattening a pointer-based tree data structure into a linear, packed array, using a tree traversal to pack subtrees in-place. This allows the packed tree to be written directly to disk and to be `mmap`'d back into memory, providing zero-cost serialization and deserialization.

Much like a traditional binary heap, the Gibbon packing algorithm can avoid storing indices entirely by using a deterministic tree traversal to flatten the tree. This allows for more efficient traversals from the user, at the cost of expensive random access. To solve this, Gibbon provides the option of inserting indirections, which tag the size of subtrees, in the flattened format to allow for faster indexing.

The biggest limitation to Gibbon is that it only operates on trees. Since every pointer's target is flattened inline into its containing node's representation, cycles and arbitrary pointers cannot be represented. My language sacrifices some of the memory and cache improvements of Gibbon to provide a significantly more generalized output capable of flattening any arbitrary pointer-based structure, including with cycles.

Gibbon also provides a small programming language for the user to write tree traversal algorithms, which are then compiled into efficient tree traversals on the packed representation. While this allows for cleaner syntax for the client, the limited scope of the syntax and the experimental research nature of the compiler make it impractical to be used in production code.

Jamet and Vollmer [5] built upon the work seen in Gibbon by creating a type-safe library that reads and creates packed data. Unlike Gibbon, the packed data library provides an interface for the user through Haskell, which presents a more portable implementation that could potentially be ported to other languages. This library shares the key similarity with my work of providing an interface to the user in an established language, instead of using a custom language and compiler.

However, the packed data library still only operates on tree structures, like Gibbon, as it inlines nodes further down the tree within the parent node. The packed data library uses the type system to ensure that the linear scan of the packed data is retrieving the correct data. In my language, type safety is automatically guaranteed as every value of a type is segregated into separate pools, with correct type information.

Vollmer et al. [9] also builds on Gibbon by creating an intermediate language named LoCal. LoCal allows the programmer to specify the data layout of values in their serialized forms and makes it easy to switch between pointers and serialized data. This was implemented as an intermediate language in the Gibbon compiler.

LoCal provides many of the same benefits as my language in terms of allowing the programmer to customize the data layout of the serialized, packed form of its data, and the involved code is "region polymorphic" regarding where the serialized form is stored. However, the type system of LoCal uses scope to impose restrictions on the order data can be read in order to facilitate fast reads, ultimately being designed for tree structures and imposing restrictions on client code that still depends on the data layout.

Indeed, the majority of the existing literature around this issue focuses around tree serialization, dating back to work by Gil and Itai [4]. Their study examines the theoretical foundations and proposes a dynamic programming algorithm to pack trees in a way to reduce page faults in a virtual memory system. By comparison, there is very little literature around the more general problem of packing and flattening graphs in the same manner.

To that end, Feser et al. [3] proposed a language called Castor which specifies the dataset and relational queries in a database. Of note is the fact that Castor allows the user to specify the layout of the dataset in memory, providing fine-grained control over the specific memory layout of arbitrary data. However, its layout is primarily database-focused, specifying indexes and row layouts, and is less applicable to general-purpose programming.

Rompf et al. [7] took a different approach, implementing compiler passes that work as compile-time optimizations to perform advanced data structure optimizations, including converting between an array of structs and a struct of arrays. These compiler optimizations are capable of performing the same types of optimizations enabled by my language, although being built as compiler passes makes them more limited in how much control is exposed to the

programmer. More crucially, the compiler pass can only handle an array of simple structs and is not capable of optimizing graphs requiring further levels of expansion.

Finally, Elsman [1] proposes a type-safe library for data serialization that gives the programmer some control over how it serializes references. Crucially, this library is able to handle serializing circular data structures. However, the control provided is very limited, only allowing the user to specify whether duplicate references should only be serialized once, and in general does not attempt to use the serialized data as the in-memory representation of the source data structure.

Instead, my language outputs the flattened graphs described by Sampson [8], used in the FlatGFA representation of the Pollen project [2]. The key contribution is to automatically generate the Rust output based on a high level description of the data types.

CHAPTER 3

LANGUAGE DESCRIPTION

At its core, my language operates by converting user-defined data types into Rust structs. All variable-length data, such as strings and arrays, and objects referenced by a pointer are instead lifted out of individual structs into flattened arrays (known as **pools**), with the structs storing only an index into their respective pool.

3.1 Language Syntax

A data structure description contains one or more **type declarations**, exactly one of which must be marked as the **top level** type with a `@flatten` annotation. Each type declaration consists of a list of field names and their types, along with optional annotations. The syntax of the description is backwards-compatible with TypeScript, allowing any description to be used as valid TypeScript type declarations (and thus syntax-highlighted by editors that support TypeScript syntax).

Each field can be of any user-specified type except the top level type. In addition, fields can be `string`, `number` (64-bit floating point), `u64` (64-bit unsigned int), or `boolean`. Each field can be made optional with `?`, and each type can be made into an array with `[]`. Multi-dimensional arrays are possible.

Each type declaration can be for either a **reference** type or a **value** type. A reference type is declared using TypeScript's `interface` keyword and represents the type of fields which would typically be referenced with pointers. The syntax requires the top level type be a reference type. A value type is declared

using TypeScript's `type` keyword and represents the type of fields which would typically be stored inline in its containing struct. Value types will not be lifted into pools unless they are used inside variable length arrays or are explicitly lifted by a user-provided annotation.

A simple example of a data structure definition can be seen below:

```
declare type u64 = unknown;

//@flatten
interface MyDataStruct {
        values: ReferenceValue[];
}

interface ReferenceValue {
    x: string;
    y: boolean[][];
    z: ValueType;
}

type ValueType = {
    a: u64;
    b?: number;
};
```

Figure 3.1: A simple data structure definition.

3.2 Customizations

The user can customize aspects of the resulting flattened Rust structs by applying annotations to individual fields in the data structure description.

3.2.1 Manual Lifting

A pool will be automatically generated at the top level for every reference type, as well as every value type that appears inside an array. Each type shares one single pool by default, but the data layout can be customized with the `@lift` annotation.

The `@lift` annotation accepts a string argument to be used as the name of a newly generated pool. Applying a `@lift` annotation to a field forces the value of that field to be lifted out of the struct into the generated pool of the same name. Even if the field contains a value type, a `@lift` annotation will override the default behavior and lift the value. Multiple fields can be lifted into the same named pool.

3.2.2 Linking

A field containing a lifted value is instead stored as an index into its associated pool. However, if multiple fields will all be consistently stored in their respective pools at the same index (for example, if a set of pools are only used to store fields in a particular struct), it is a waste of space to store the same index multiple times.

The `@link` annotation accepts a string argument that matches the name of a pool created with the `@lift` annotation. Applying it to a field asserts that the index of each value in the given field's pool match one-to-one with the index of an associated value in the named pool passed to the annotation. As a result, the index of the linked field will not be stored.

For example, given the following input data description:

```
interface MyType {
        //@lift poolA
        a: u64;
        //@lift poolB
        //@link poolA
        b: u64;
}
```

The output Rust will only contain an index for the field `a`, since the index of `b` inside of `poolB` can be recovered by using the index of `a`.

```
pub struct MyType {
    pub a: Id<u64>,
}
```

Linking is not available for strings and arrays, as each individual value of those types represent a range of indices instead of a specific index. Linking is also not available on optional fields, as an empty value would introduce an offset into the indexing.

3.2.3 Self Linking

In the limit, where all fields of a struct are all the same index in their own respective pools, there is no need to store the struct at all, as a single index suffices to recover all of its contents. If there were an array of these structs, this would have the effect of turning the array of structs into a struct of arrays (with the arrays being the lifted pools at the top level).

The language supports omitting a field's index entirely and linking it to the index of the containing struct itself by using the `@link self` annotation. Applying this annotation automatically generates a new pool for the given field for every pool containing the parent type, ensuring a one-to-one correspondence is possible. Note this is vulnerable to exponential blowup if multiple layers of self linking are present.

Like traditional linking, self linking is not supported on strings, arrays, and optional fields. In addition, the containing struct of the field is required to be a reference type (as it must appear in a pool to be assigned an index). Note it is okay if the containing struct is in an array, as each individual entry will still have its own index in its containing pool.

3.3 Generated Rust Output

The generated Rust output contains one `struct` for each declared type in the data structure definition. The top level type will have the prefix `Flat` prepended to its name, while the rest keep their names. Each field that is lifted will be replaced with an index into its respective pool (or a span, for strings and arrays). The top level struct will contain all the generated pools.

The top level struct contains getter methods to fetch from each of its pools. Every other struct contains getter methods to retrieve all of its fields. These getters require a reference to the top level struct be passed in, as each struct otherwise has no access to the pools in which its fields are stored. These getters exist for every field, even ones that are not lifted. This allows the Rust interface to stay the same even if the data layout changes.

3.3.1 Convenience Features

The generated output also contains convenience types and wrappers which make operating on the flattened data types more natural.

Each data type apart from the top level struct is given an unwrapped version, with the same name except for the `Unwrapped` postfix appended to the end. This unwrapped struct type contains the fields of the original data type as defined by the user, with all pointers, strings, and arrays expanded back into the struct. All lifted and linked fields are also present. This can be useful if transforming the flattened data back into its natural representation.

```
interface MyType {
    x: string;
    y: boolean[];
    z: OtherType;
}
```

```
pub struct MyTypeUnwrapped<'a> {
    pub x: &'a str,
    pub y: Vec<Xbool>,
    pub z: OtherTypeUnwrapped,
}
```

Figure 3.2: A type definition (left) and its unwrapped form (right).

In addition, each data type apart from the top level struct is given a wrapper struct, with the same name except for the `Wrapper` postfix appended to the end. For types without fields that are self linked, this wrapper contains the flattened struct itself and a reference to the top level struct. This allows self-contained getter methods to be generated on the wrapper type, as it contains the reference to the top level struct needed to access the pools containing its fields.

For types with fields that are self linked, the wrapper also contains the index of the struct itself in whichever pool contains it, in addition to an enum tagging which pool it came from. This allows the getters of the self linked fields to

determine at runtime which of the automatically generated pools for self linked fields it should pull from.

```
interface MyType {
    //@link self
    x: u64;
    y: boolean[];
    z: OtherType;
}
```

```
pub struct MyTypeWrapper<'a> {
    pub inner: MyType,
    pub id: Id<MyType>,
    pub top: &'a FlatData<'a>,
    pub pool: MyTypePools,
}
```

Figure 3.3: A type definition (left) and its wrapper (right).

The enum value is currently used to determine the correct pool for each field within the getter methods at runtime via a `match` statement. The current stable implementation of const generics in Rust does not yet support using enums, though there is a proposal. In the future, rewriting this feature to statically bind the getter methods to the correct pools at compile time could pose a performance improvement.

3.3.2 Custom Generated Types

The compiled output's use of the `zerocopy` crate to get free serialization and deserialization from disk does mean that Rust's built-in boolean and optional types cannot be used, as the `zerocopy` features used require every bit pattern be a valid value. The default boolean and optional both do not provide this guarantee, so instead, custom `Xbool` and `Xoptional` types are generated.

`Xbool` is a boolean type internally backed by a `u8`, where 0 represents false and every non-zero value represents true.

`Xoptional` is a generic optional type that wraps a type and repurposes the all 1 bit pattern to represent empty, with every other bit pattern representing the corresponding value of the underlying type. This does mean that it is not safe to make a field optional if its value is expected to be represented as the all 1 bit pattern. This pattern was chosen because the all 1 bit pattern anecdotally appears far less often than the all 0 bit pattern, and repurposing a single bit pattern avoids needing to use extra bits.

3.3.3 Multidimensional Arrays

Multidimensional arrays are given a special pool in the top level, called `nested_arr_data`, which is generated if any multidimensional arrays are detected inside the data structure definition. All elements inside a multidimensional array are stored in their respective pools as usual (unless manually lifted). Then, each inner array is stored as a span inside `nested_arr_data`. A two-dimensional array is represented as a span of indices into `nested_arr_data`. Higher n dimensional arrays can be recursively created by adding the $n-1$ dimensional array spans into `nested_arr_data` again and then referencing the span of those resulting indices.

3.4 Compiler Implementation Details

3.4.1 Intermediate Representation

The compiler uses a two-stage approach to generate the flattened data structures. The input data structure description is first converted into an intermediate representation, and then the output is built from the IR.

The IR is represented as a graph and consists of a list of pools, structs, and fields, in addition to some bookkeeping about the top level struct itself (namely, the name of the struct and any fields contained directly at the top).

Each pool node stores the type of data contained within itself. A type is either a raw type (integer, floating point number, boolean) or an outgoing edge to a struct node. There are also other pool node types apart from standard type pools, as strings, arrays, and multi-dimensional arrays all have bespoke pool node types of their own.

Each struct node contains edges to every field stored within it, in addition to data about itself.

Each field node contains an edge to the struct node that contains the field. In addition, fields with data lifted into pools (automatically or manually) have an edge to the pool containing their data. For fields that are self-linked, the field node stores edges to every pool containing the type of the field's containing struct as well as every auto-generated self link pool for the field, organized as a 1-to-1 mapping from containing struct pool to generated field pool. This map-

ping is used to generate runtime code that determines which generated field pool to use for each location the containing struct may reside in.

3.4.2 Storage Multigraph

Each self linked field requires a new pool for every pool its containing type appears in, or else the indexing will not line up. To facilitate this, the compiler generates a multigraph of relations between types that contain self linked fields.

For every field in a given struct of type `A` that is self linked and has type `B`, an edge from `A` to `B` is created. Multiple fields create multiple edges in the storage multigraph. An example is seen in figure 3.4.

```
interface TypeA {
    //@link self
    x: TypeB;
    //@link self
    y: TypeB;
}
```

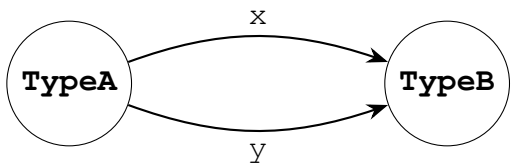

Figure 3.4: A struct (left) and its nodes and edges in the storage multigraph (right).

This means that each self linked field creates or modifies a node of its corresponding type in the storage multigraph. After every struct has been processed, every type that either contains self linked fields or is the type of a self linked field appears in the storage multigraph as a node. A node may be both a source and a sink for edges if it both appears as the type of self-linked fields and also contains self-linked fields of its own, as shown in figure 3.5.

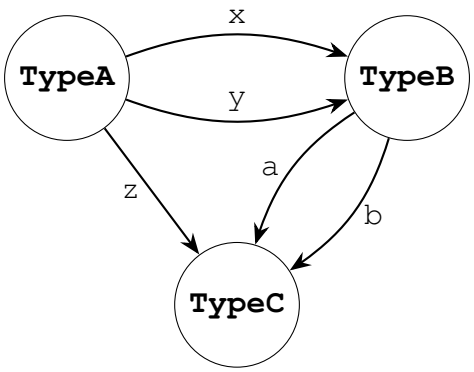

Figure 3.5: A storage multigraph where `TypeB` is both the type of self-linked fields `x` and `y` but also contains self-linked fields `a` and `b` of its own.

The storage multigraph must be a directed acyclic multigraph. It is a compile-time error to include a self-link cycle, as there is no way to generate all the needed pools.

The semantics of the next step is to unfold the storage graph into the storage forest. Note that the following description is the semantic definition of what information is needed to finish the pool generation. The compiler itself need not realize the actual forest in memory; the algorithm used will be described after the semantics.

Every single pool whose type contains self-linked fields will generate a node of its type as the root of a tree in the forest. Within each tree, its corresponding node of the same type in the storage multigraph will dictate the edges leaving the node. Each edge points to a new node identified with the type of the target of the edge in the storage multigraph. This process completes recursively until the tree's leaves are nodes in the storage multigraph with no outgoing edges.

Note that this process leaves the subtree underneath any node of a given type identical to all other subtrees of the same type, as its contents are dic-

tated by the same node in the storage multigraph. Every non-root node in the unfolded storage forest represents a pool that must be generated to store self-linked fields, and each child is the pool used to fetch the specific field's values for a struct of the parent node's type in the parent node's pool.

Using the example of the storage multigraph from figure 3.5, if the top level struct had one pool of type `TypeA` and two pools of type `TypeB` (perhaps one auto-generated and one manually created by the user with a `@lift` annotation), then the resulting unfolded forest would be shown in figure 3.6.

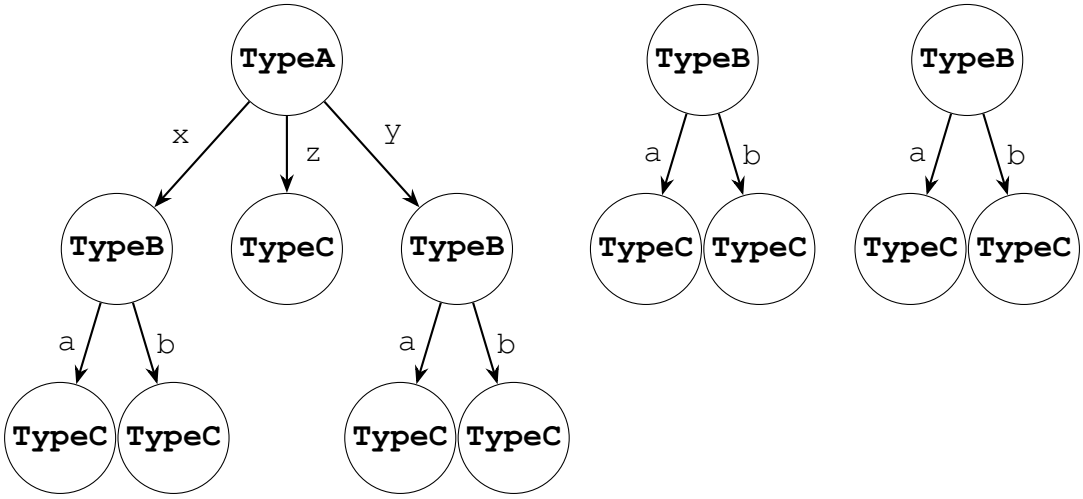

Figure 3.6: The unfolded storage forest based on the storage multigraph in figure 3.5 with one pool of `TypeA` and two pools of `TypeB`.

The compiler is able to efficiently find pools to generate without any additional memory by iterating through all existing pools and checking if each type contains self-linked fields. For all types that do, the compiler starts at the type's node in the storage multigraph and performs a depth-first search. However, the compiler does not track which nodes have already been visited, allowing the same node to be visited multiple times, along every possible path from the root. Every node it visits dictates a new pool to generate. This algorithm creates a

one-to-one correspondence with the unfolded forest described above, as every entry point corresponds to a tree root and every visit to a node corresponds to a node in the unfolded forest.

The compiler stores which node was the source of the edge leading into each node it visits. This creates a one-to-one mapping from pools of containing structs (as each node in the unfolded storage forest represents a pool of its given type) to the generated pools for self-linked field values. This mapping is used to populate the mapping of containing structs in each field node of the intermediate representation.

Finally, for all types that appear in the storage multigraph, the compiler generates an enum of every pool (automatic, manually lifted, and generated for self linking) of that type to be stored in wrappers at runtime so that field values can be fetched from the correct pool.

CHAPTER 4

EVALUATION

To evaluate the usage of my language, a series of benchmarks were written in Rust that operate on genomic data in the Bed Extensible Data (BED) format. The benchmarks were first written with a standard pointer-based implementation, parsing the text-based BED format into an in-memory array of structs. Then, using my language, the benchmarks were retrofitted to use their new flattened versions.

4.1 Experimental Setup

The benchmarks were compiled with Rust 1.90.0 in release mode and run on a Windows 11 laptop with an AMD Ryzen 7 6800HS mobile processor and 16GB of GDDR5 RAM. All benchmarks were run while connected to wall power. The Hyperfine utility was used to perform timing measurements, running each benchmark at least 10 times (more if the utility determined additional runs were necessary).

All sample BED files were acquired from the Bedtools tutorial documentation [6] for Bedtools, an existing program for operating on BED files.

Three benchmarks were written. First, the intersection benchmark operates identically to the `intersect` command in Bedtools, finding all overlapping ranges between two files. In the benchmark, to simulate high CPU load, a quadratic-time algorithm using two nested loops was used, in which every range from the `cpg.bed` file was compared against every range from the

`exons.bed` file. This command mirrors the FlatBED intersection tool in the FlatGFA project [2].

Second, the listing benchmark merely reads the file and prints every entry to the console. This benchmark showcases a simple linear scan across the data and was run against the `hesc.chromHmm.bed` file.

Third, the fetch benchmark reads the same file and prints a single entry to the console, intended to isolate the deserialization time as much as possible by performing almost nothing after the deserialization beyond a bare minimum use to ensure the compiler does not optimize away otherwise dead code.

4.2 Quantitative Results

The timing data reported by Hyperfine is shown in Table 4.1.

Benchmark	Original (ms)	Flattened (ms)	Improvement
Intersect	$31,847 \pm 393$	$23,362 \pm 1,072$	26.643%
Listing	913.4 ± 22.8	871.9 ± 17.2	4.543%
Fetch	67.9 ± 4.3	8.0 ± 0.8	88.218%

Table 4.1: Measured performance of original pointer-based implementations in Rust against flattened versions.

The data shows a significant 26.643% improvement in the intersection benchmark, a model of a real-world use case. This suggests that in read-heavy workloads, the benefits of a flattened layout in memory can already provide significant performance gains with no changes to the algorithm.

In addition, the fetch benchmark, which does almost nothing except deserialize the data, shows the immense benefit of the free deserialization provided

by being able to `mmap` flattened data directly back into memory, with an 88.218% speed improvement.

The listing benchmark shows the smallest change, likely due to the nature of its linear time scan exhibiting similar asymptotic behavior as the linear time parse from the original text format, while simultaneously not requiring enough data processing as a quadratic algorithm for the speed improvements of the data read to become noticeable. However, even this benchmark still shows a modest 4.543% improvement with minimal changes, suggesting that even in less advantageous scenarios, flattening data structures may still provide benefits.

In addition to benchmarking, the file sizes of the original text files with their flattened versions were compared and the results are shown in Table 4.2.

Test File	Original (bytes)	Flattened (bytes)	Improvement
`cpg.bed`	921,494	826,695	10.288%
`exons.bed`	30,191,881	13,275,643	56.029%
`gwas.bed`	618,299	527,315	14.715%
`hesc.chromHmm.bed`	24,250,961	17,603,492	27.411%

Table 4.2: File size of original serialized text format files against flattened binaries.

As expected, the flattened and packed binary representation also presents significant space savings over a plaintext representation, with certain file sizes being reduced by over 50%. The remarkable fact is that this is not a compression process. No decompression is needed before the flattened file is usable; in fact, less work is required to deserialize it than the original text version.

4.3 Qualitative Discussion

The process of converting the original pointer implementation to use the flattened versions was very trivial. While converting the parser for the text version to create the flattened data structure took some changes, once the flattened version was written to disk, the parsing process could be entirely replaced with a single `mmap` call, and then the remaining changes consisted of replacing every field access like `entry.name` with a getter call like `entry.get_name()`.

In other words, the ergonomics of writing code against the flattened representation is very similar to the ease of writing against the intuitive pointer version. Especially when reading values, the syntax is barely different. As part of the intersection benchmark, new structs needed to be created to represent partial overlaps, and the automatically generated `Unwrapped` data types in Rust made this quite convenient, since the `Unwrapped` version could be instantiated wherever the code originally needed a standard pointer struct.

Just as important is the fact that the specific layout of the data can be changed without any further modifications. For example, by default, the string name of each entry is lifted into a pool, but the start and end fields, which are 64 bit integers, are kept in a struct. By using the `@lift` annotation, I can force those to also be lifted into separate pools, and then I could even use the `@link` annotation to only store one index. All of these changes can be done by merely adding annotations to the data structure description.

This allows users to easily experiment with different data layouts on disk with minimal effort, making the task of profiling different implementations trivial. By reducing the burden of trying different implementation strategies and

increasing the ergonomics of writing code on various representations, the hope is that users are more likely to find and use the most optimal representation for each specific use case.

CHAPTER 5

FUTURE WORK

The compiler is in a functional state already, but multiple engineering features would make it easier to use.

First, while my compiler will automatically generate the data types and getter methods that are layout agnostic, allowing users to switch layouts without needing to change their code, my compiler does not yet generate consistent setters for each layout type. This means that the code to add data into flattened data types, before they are written to disk, still needs to change when the layout is adjusted. To make the interface fully robust, smarter setters will also need to be generated.

In addition, the Rust code needed to write and read files to disk currently needs to be manually written. Future improvements to my compiler could automatically generate the file manipulation code, as it is predominately boilerplate as well.

While my language supports optional fields, it does not support union types. Many useful data structures that would benefit from flattening, such as compiler intermediate representations, can be naturally expressed with the help of union types, so adding support would further improve the ergonomics of the language.

In addition to feature work, additional customization options can be explored. For example, at the moment the `@link` annotation indicates a one-to-one correspondence between the index of the linked pool with the target pool. However, in theory, custom indexing could be used. For example, two fields

could both be lifted to the same pool and linked to a common link target, but the first field's entries live in index $2i$ for each entry i of the link target, and the second field's entries live in index $2i + 1$. As long as the custom indexing function is bijective, the link would still be valid. This could allow data structures like a traditional binary heap to be cleanly expressed in my language.

These customizations would be able to support even more arbitrary user-specified layouts, so as long as the resulting Rust interface does not change, users have more room to freely experiment with layouts to find the one with the best performance.

On a broader scope, an even better user interface would pose opportunities for significantly more development. For one, many of the current applications of my language involve parsing an existing text format (such as BED) and converting it into its flattened version. There could feasibly be a way to specify how to parse a text format into a given data structure description, allowing the compiler to generate the parser that converts text into the flattened variant automatically.

Even more ambitious would be a way to specify logic directly on a pointer-based version. In conversations with potential users, some did express hesitation at using a new language to write logic when existing programming languages are available, and indeed my language's hybrid approach explicitly aims to allow interoperability with production languages. That said, in the course of building benchmarks, some operations, such as intersect, naturally lent themselves to creating new unwrapped structs of the original type as part of the processing, thereby materializing the unwrapped struct in memory. A way to specify the logic necessary in a natural, pointer-based way could be either com-

piled or passed to a library to be executed directly with flattened data, avoiding the overhead of unwrapping the data structure. This could dramatically expand how much the user could get done operating on pointer representations while still maintaining the human-editable Rust output after compilation.

BIBLIOGRAPHY

[1] Martin Elsman. Type-specialized serialization with sharing. In *Trends in Functional Programming*, pages 47–62, 2005.

[2] Adrian Sampson et al. Pollen: Accelerated pangenome graph queries. `https://github.com/cucapra/pollen`, 2026. [Accessed 23-04-2026].

[3] John Feser, Sam Madden, Nan Tang, and Armando Solar-Lezama. Deductive optimization of relational data storage. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[4] Joseph Gil and Alon Itai. How to pack trees. *Journal of Algorithms*, 32(2):108–132, 1999.

[5] Arthur Jamet and Michael Vollmer. Type-Safe and Portable Support for Packed Data. In Jonathan Aldrich and Alexandra Silva, editors, *39th European Conference on Object-Oriented Programming (ECOOP 2025)*, volume 333 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:19, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[6] Aaron Quinlan. Bedtools Tutorial — quinlanlab.org. `http://quinlanlab.org/tutorials/bedtools.html`. [Accessed 22-04-2026].

[7] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. *SIGPLAN Not.*, 48(1):497–510, January 2013.

[8] Adrian Sampson. One Weird Trick for Efficient Pangenomic Variation Graphs (and File Formats for Free) — cs.cornell.edu. `https://www.cs.cornell.edu/~asampson/blog/flatgfa.html`. [Accessed 23-04-2026].

[9] Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton. Local: a language for programs operating on serialized data. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 48–62, New York, NY, USA, 2019. Association for Computing Machinery.

[10] Michael Vollmer, Sarah Spall, Buddhika Chamith, Laith Sakka, Chaitanya Koparkar, Milind Kulkarni, Sam Tobin-Hochstadt, and Ryan R. Newton. Compiling Tree Transforms to Operate on Packed Representations. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:29, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.