Rust-ifyng Data Collections for Compiler Optimization

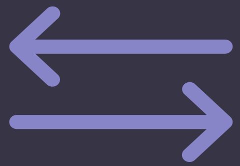
CS Undergraduate Research Showcase 2024
October 17, 2024 (🎉)

Benjamin Ye

Slide content adapted from original MemOIR paper + slides by Tommy McMichen and Simone Campanoni

Compilers are really cool!

- They do a lot of magical things we take for granted!



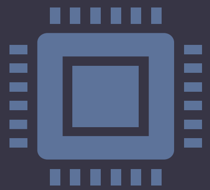
Translation

Abstractions

- Variables
- Control flow

Lowering

- Languages (e.g. C++, Swift)
- Architectures (e.g. x86, ARM)



Optimization

Compute

- Speed
- Energy

Space

- Memory
- Storage

Compilers are really cool!

- They do a lot of magical things we take for granted!

Translation

Abstractions

- Variables
- Control flow

Lowering

- Languages (e.g. C++, Swift)
- Architectures (e.g. x86, ARM)

Optimization

Compute

- Speed
- Energy

Space

- Memory
- Storage

Optimizing memory is hard

```
std:unordered_map<int, int> &table;

table[0] = 10;

table[1] = 20;

print(table[0]);
```

No production C/C++ compiler can propagate 10 to the `print` statement.

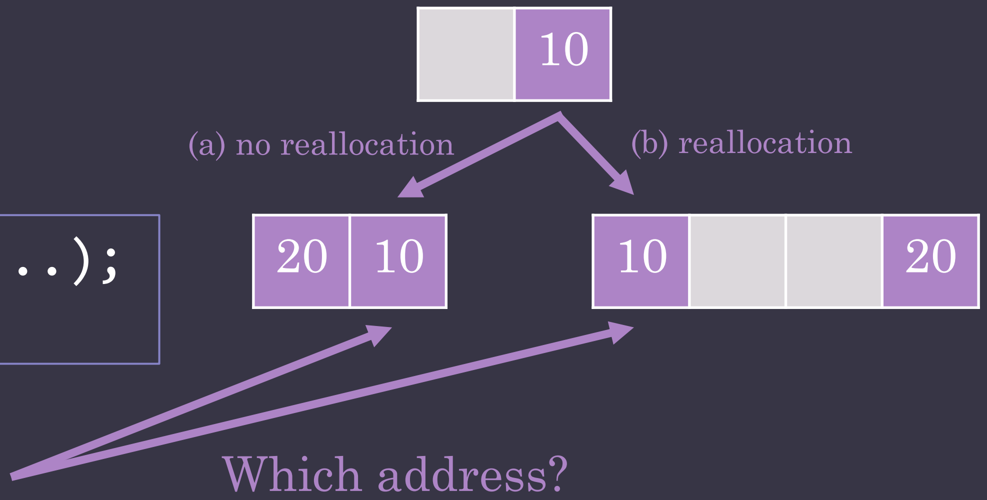
Optimizing memory is hard

```
std:unordered_map<int, int> &table;

table[0] = 10;

table[1] = 20;
realloc?(table, ...);
rehash?(table);

print(table[0]);
```



Optimizing memory is hard

```
std:unordered_map<int, int> &table;

table[0] = 10;

table[1] = 20;
realloc?(table, ...);
rehash?(table);

print(table[0]);
```

The Undecidability of Aliasing

G. RAMALINGAM
IBM T. J. Watson Research Center

Alias analysis is a prerequisite for performing most of the common program analyses such as reaching-definitions analysis or live-variables analysis. Landi [1992] recently established that it is impossible to compute statically precise alias information—either may-alias or must-alias—in languages with if statements, loops, dynamic storage, and recursive data structures: more precisely, he showed that the may-alias relation is not recursive, while the must-alias relation is not even recursively enumerable. This article presents simpler proofs of the same results.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*; F.4.1 [**Mathematical Logic**]: Computability Theory; F.4.3 [**Formal Languages**]: Decision Problems

General Terms: Languages, Theory

Additional Key Words and Phrases: Alias analysis, pointer analysis

1. INTRODUCTION

Compilers and various other programming tools make good use of static program analysis. To solve most program analysis problems, such as the problem of determining live variables, one requires alias information, or information about whether two L-valued expressions may/must have the same value at some program point. Informally, two names or L-valued expressions are said to alias each other at a particular point during program execution if both refer to the same location. In the may-alias problem, one is interested in identifying aliases that can occur during some execution of the program, while in the must-alias problem, one is interested in identifying aliases that occur in all executions of the program. Obviously, such information is relevant to most dataflow analysis problems.

Program analysis is commonly performed under the conservative assumption that all paths in the program are executable, since the problem of deciding if an arbitrary path in a program is executable is undecidable. This simplifying assumption makes it possible to solve a number of program analysis problems. Unfortunately, even this assumption is not sufficient to make the may-alias or must-alias problem decidable.

Author's address: IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598.

ACM Transactions on Programming Languages and Systems, Vol. 16, No. 5, September 1994, Pages 1467–1471.

Problem

- Data collections are high-level abstractions of memory.
- These abstractions are prematurely lowered before optimizations occur.

Problem

- Data collections are high-level abstractions of memory.
- These abstractions are prematurely lowered before optimizations occur.

How can we maintain information about these abstractions at the optimization level?

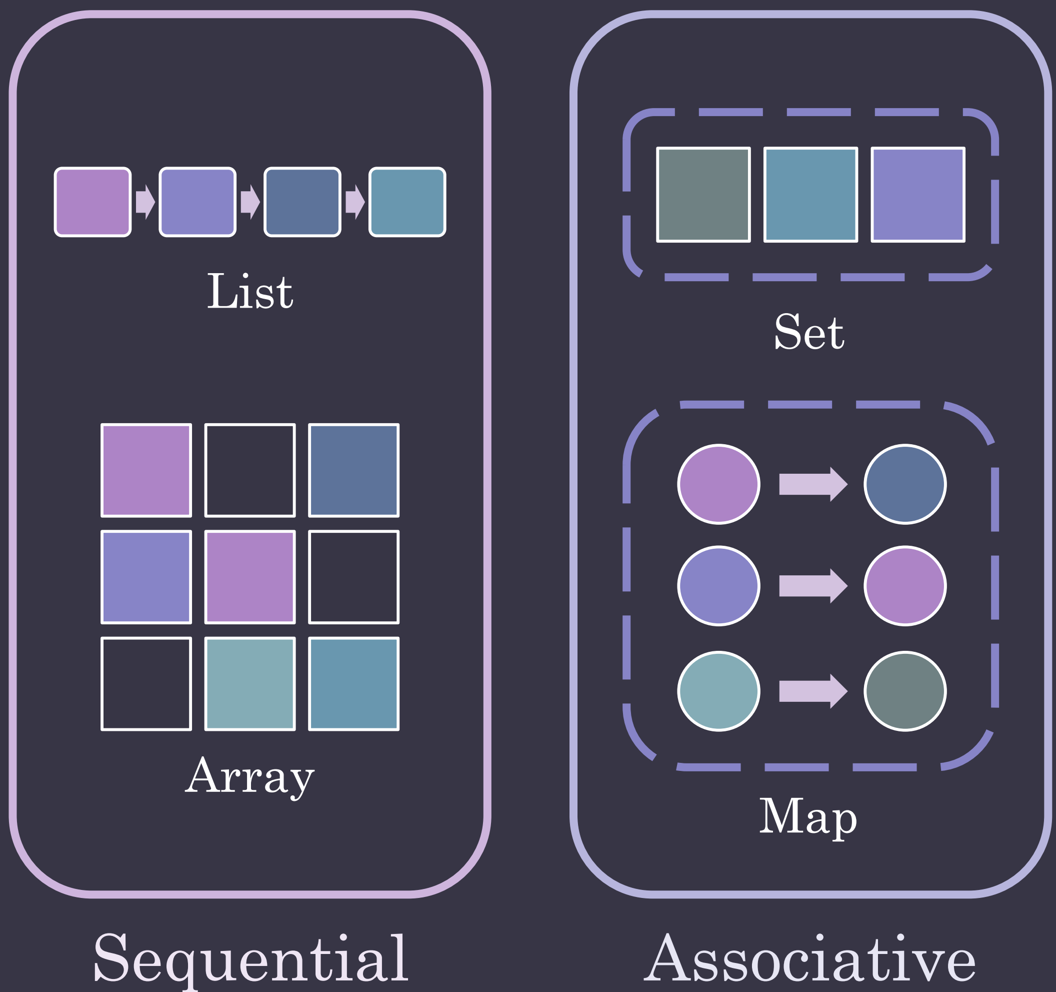
MemOIR

- Memory Object Intermediate Representation
- Intermediate compilation target designed for memory optimization
- Separates memory use from memory structure
  - Preserves higher-level information
  - Allows us to explore new optimizations!

Representing Data Collections in an SSA Form

Tommy McMichen, Nathan Greiner, Peter Zhong, Federico Sossai, Atmn Patel, Simone Campanoni
Northwestern University
Evanston, IL, USA

Data collections



Pragmatics: guarantees

- Strong memory guarantees required

```
var i = 9;
```

Pragmatics: guarantees

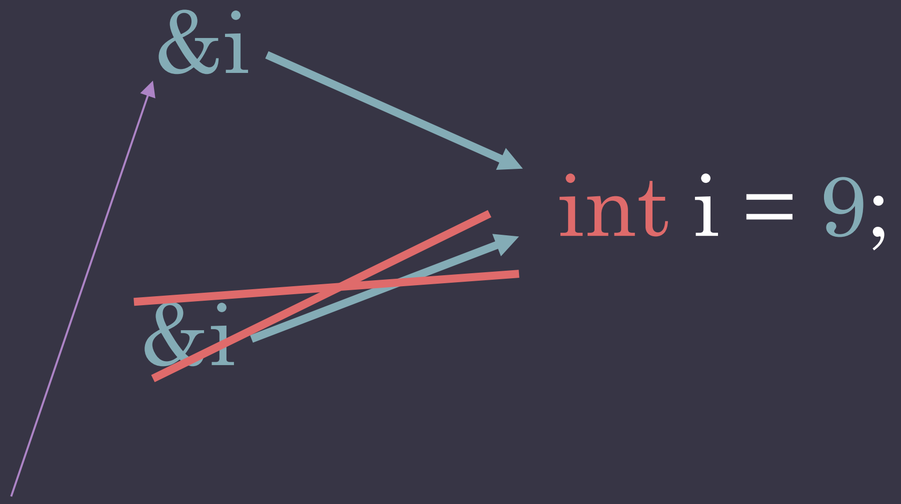
- Strong memory guarantees required
  - Strong static types

Fixed, rigorously enforced type

```
int i = 9;
```

Pragmatics: guarantees

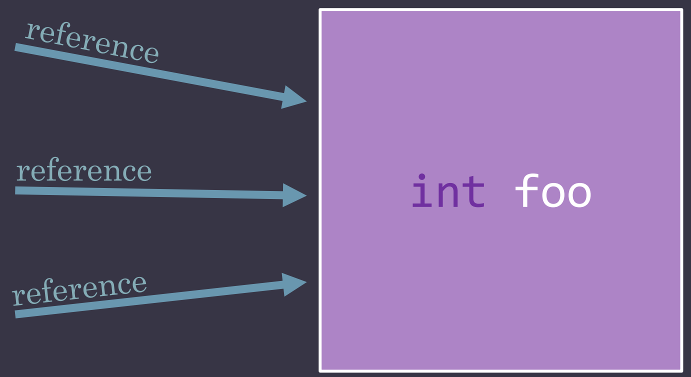
- Strong memory guarantees required
  - Strong static types
  - Single reference



Only one reference that modifies some data at any point

Pragmatics: C/C++

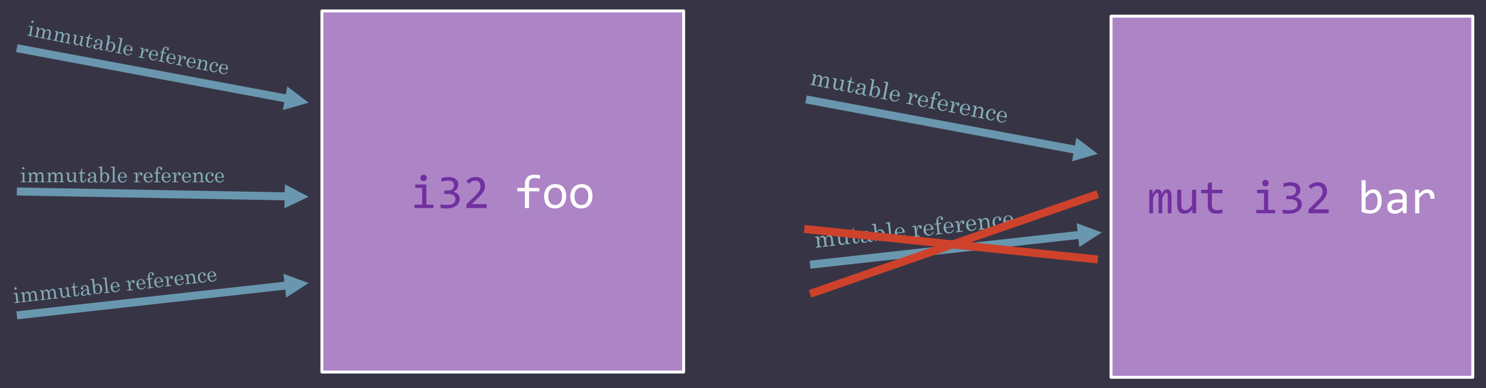
- A little clunky
  - Strong static types? Yes
  - Single reference? Manually
- Result: annoying for developers to use!



Perfectly ok!
(bad)

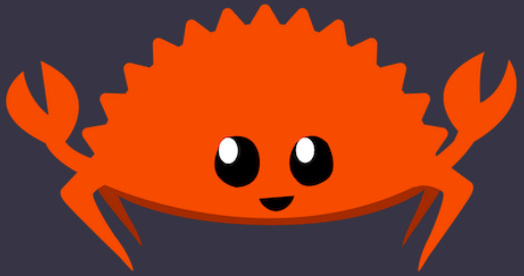
Pragmatics: Rust

- Ownership system
  - Immutable vs mutable variables
    - Immutable: any number of references
    - Mutable: only one active reference
  - Elides aliasing in variables
    - MemOIR allows us to extend that to collections!



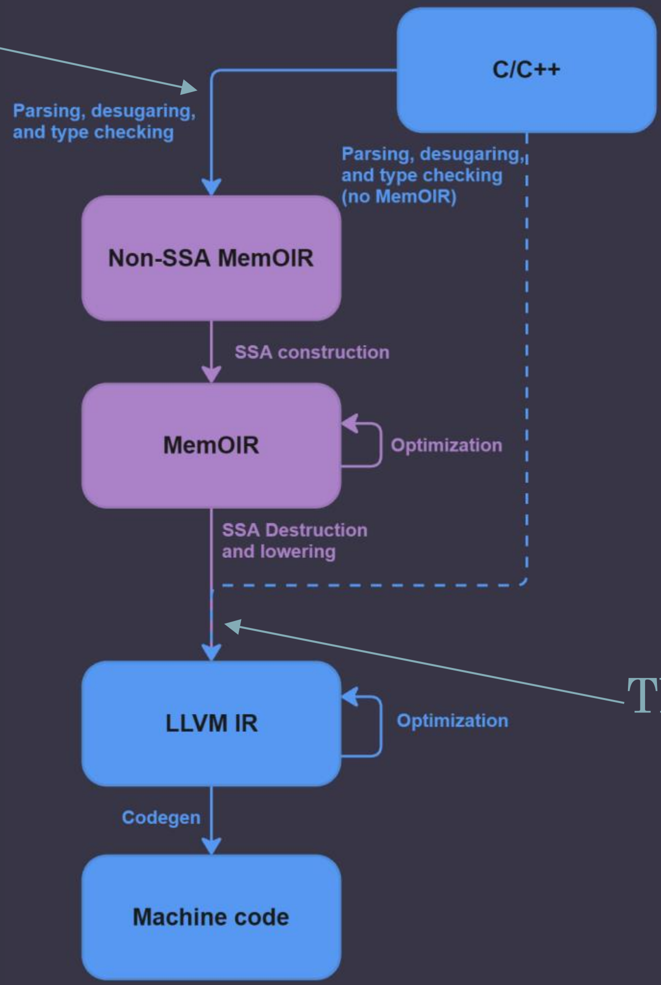
Pragmatics: Rust

- Checks both boxes
  - Strong static types? Yes
  - Single reference? Yes
- Borrow checker enforces ownership
  - Alias checking moves from programmer to compiler



Fitting rust in

This is just Clang!

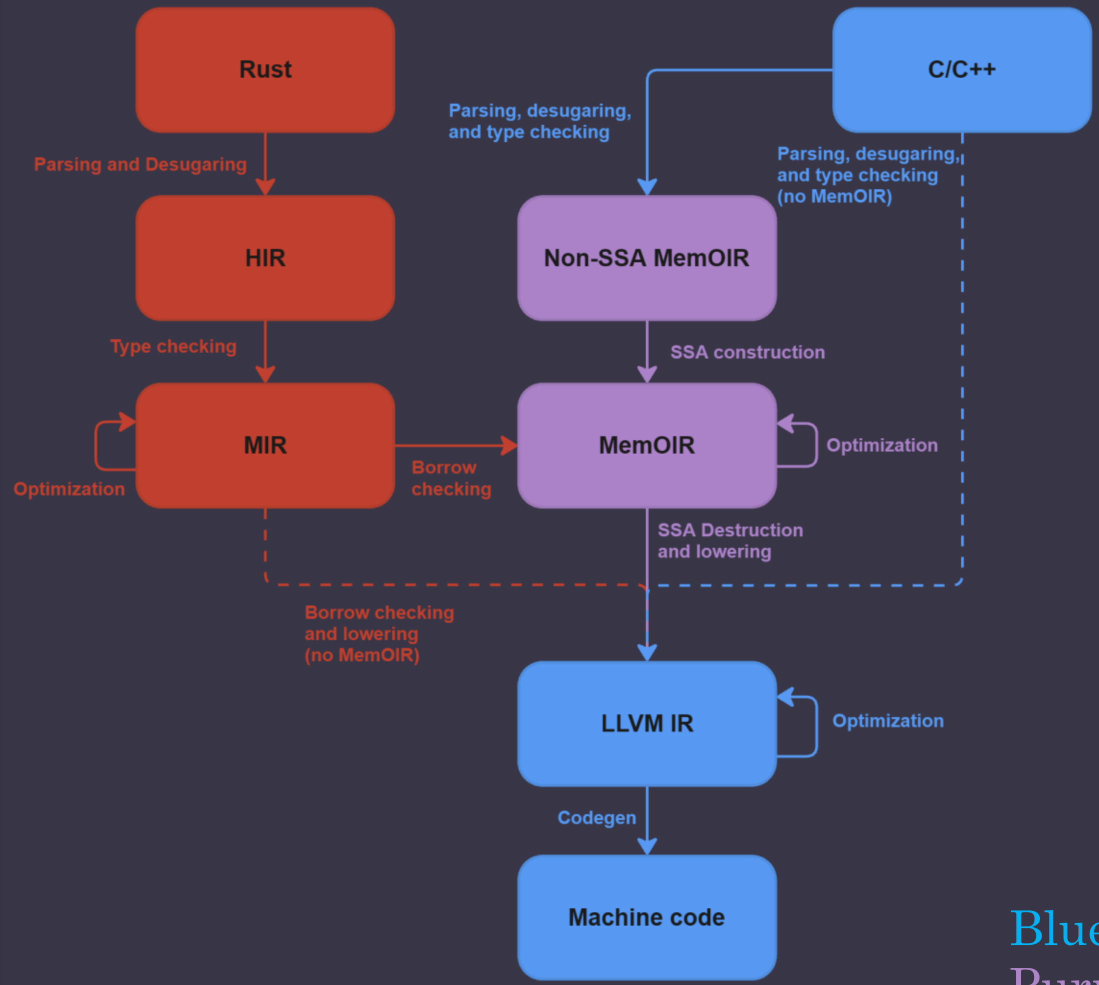


This lowers to LLVM!

(simplified)

Blue = LLVM/Clang
Purple = MemOIR

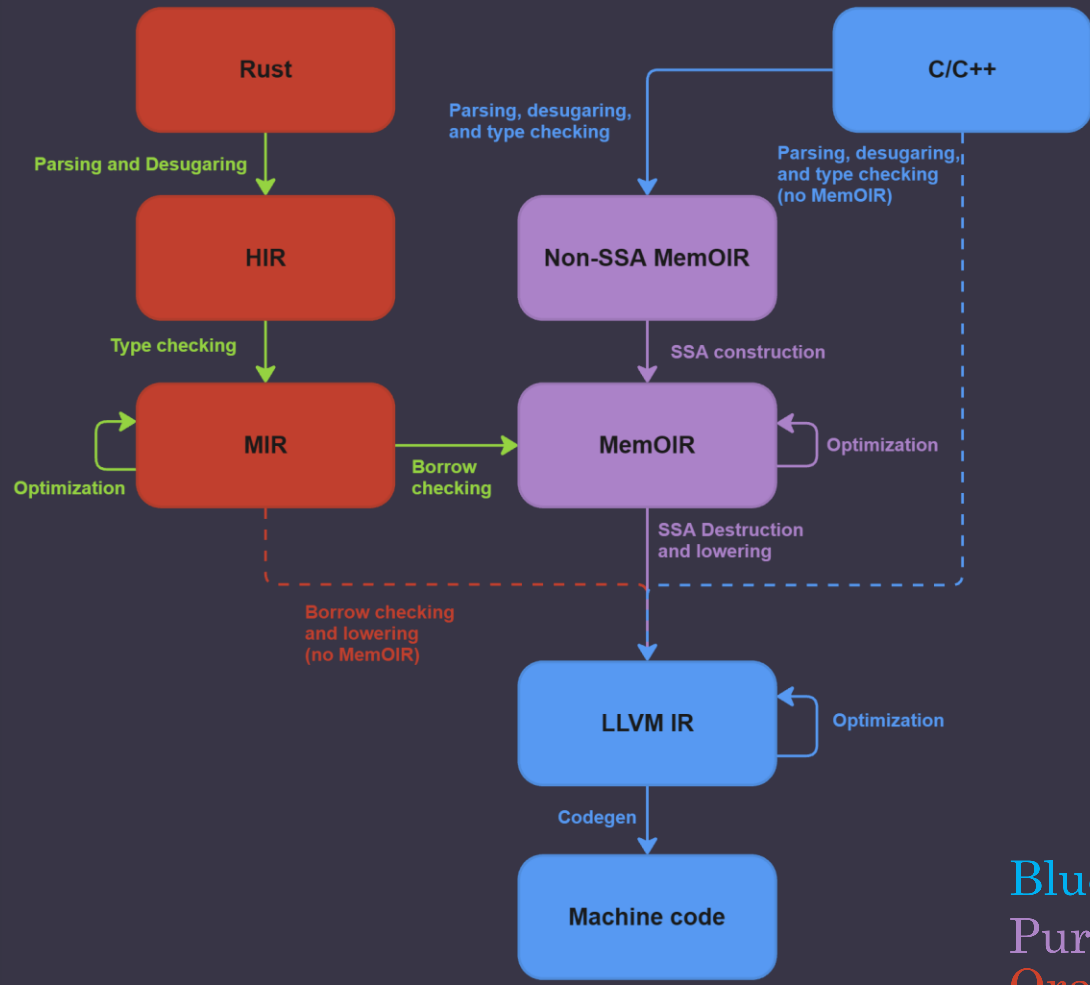
Fitting rust in



(simplified)

Blue = LLVM/Clang
Purple = MemOIR
Orange = Rust

Fitting rust in



(simplified)

Blue = LLVM/Clang
Purple = MemOIR
Orange = Rust
Green = Ben's work

Implementation mechanics

- `rust-memoir`
  - Library/language extension
  - As close to a 1:1 mapping of `Vec` and `HashMap` as possible
  - Generates MemOIR symbols
  - Occasional extra type specification required

```
vec![2, 3, 7, 9]
some_vec.iter()
HashMap::new()
some_map.insert(5)
```

→

```
seq_u32![2, 3, 7, 9]
some_seq.iter()
Assoc::<i32, u64>::new()
some_assoc.insert(5)
```

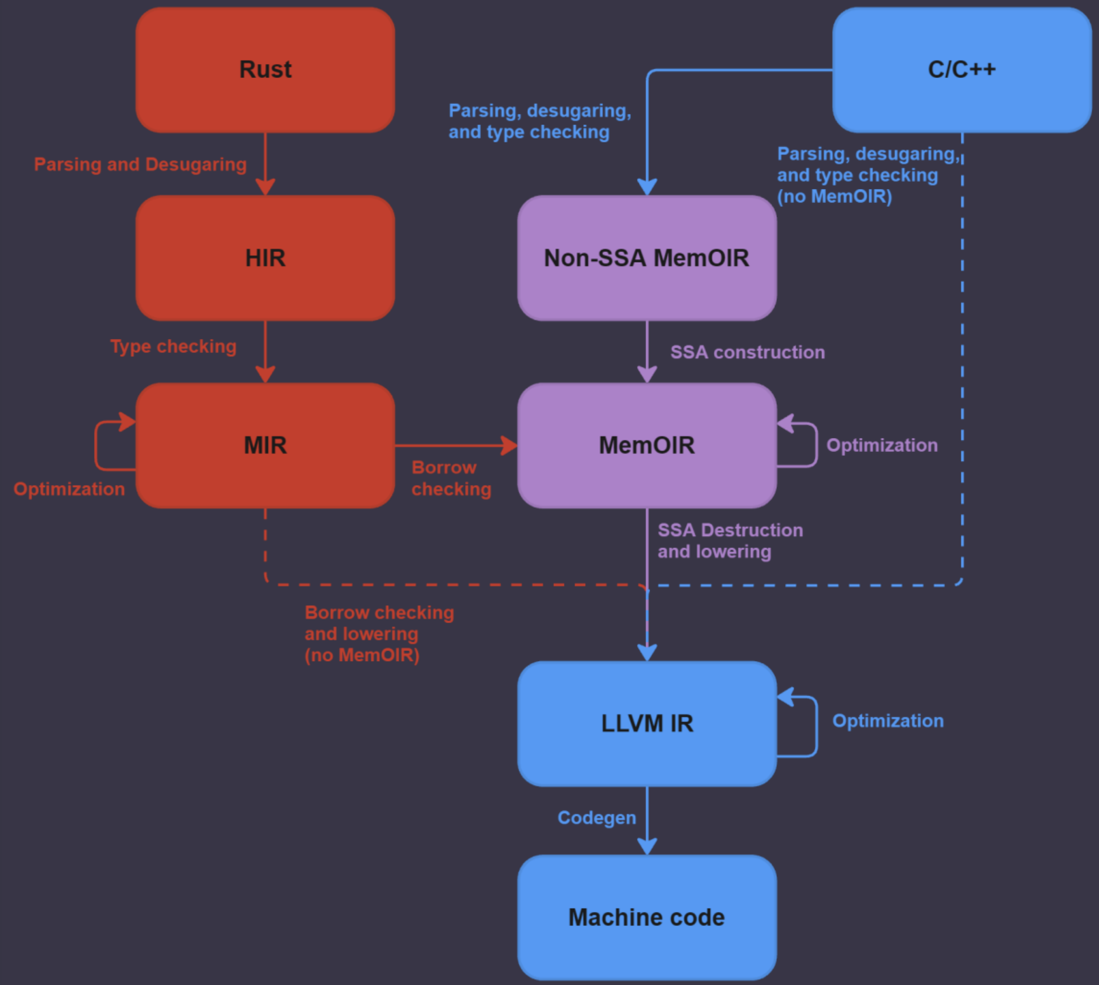
Implementation mechanics

- References
  - Overloading operators is awkward

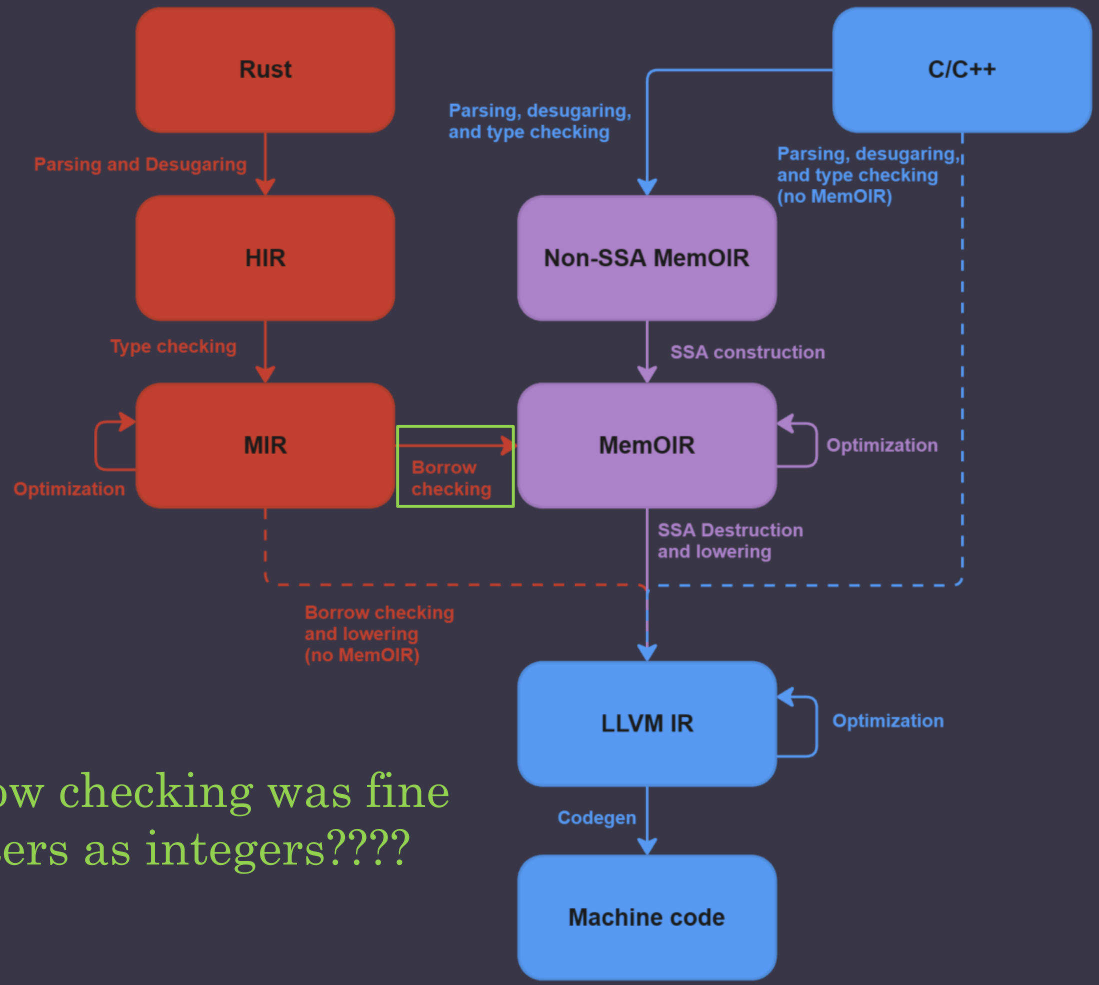
```
out = some_vec[0];
some_vec[0] = in;
```

```
let mut some_seq_0 = SeqRef::<i32>::new(&some_seq, 0);
out = some_seq_0.get();
some_seq_0.set(in);
```

Implementation mechanics

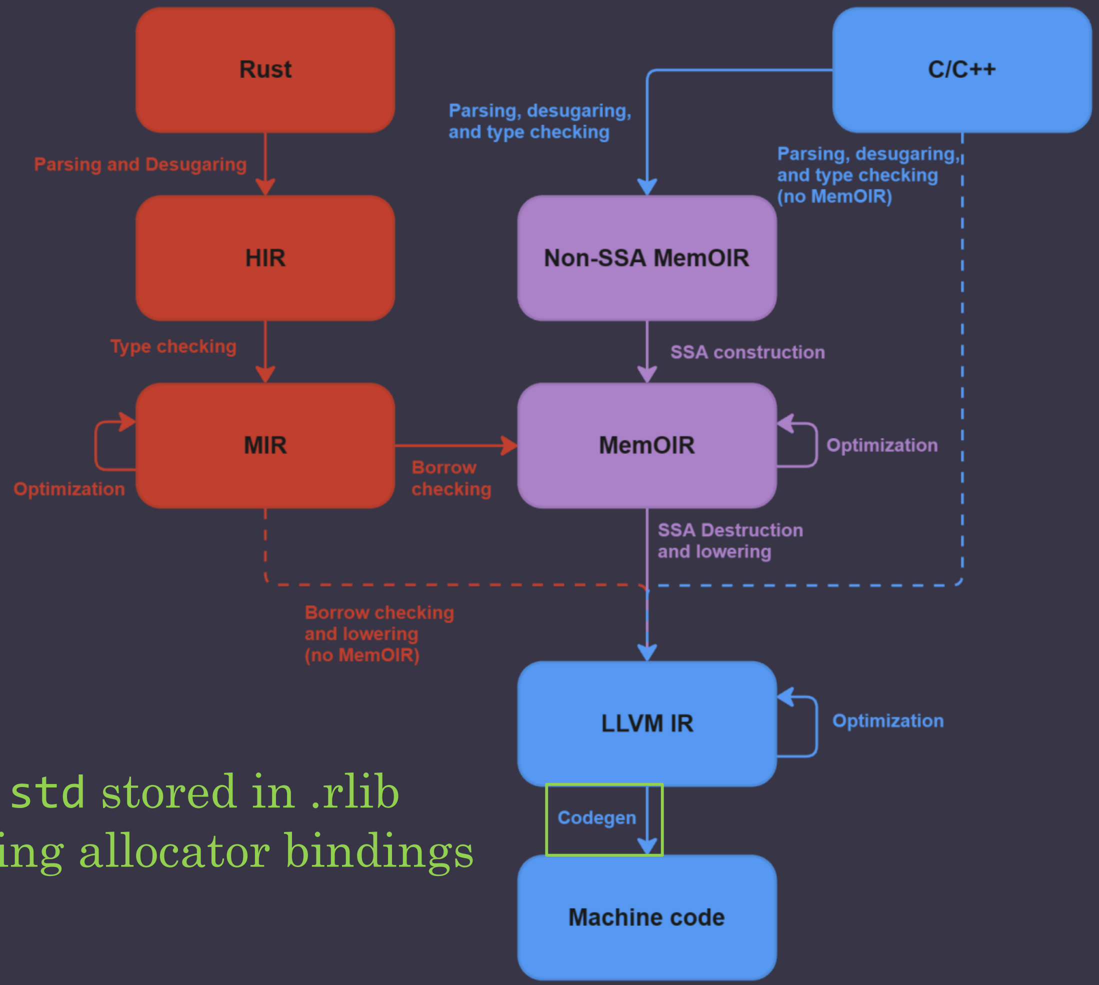


Implementation mechanics



- Borrow checking was fine
- Pointers as integers????

Implementation mechanics



- Rust `std` stored in .rlib
- Missing allocator bindings

Future work

- Get formal test runner/benchmarks working
- Extend frontend for structs/objects
- Misc. rust ergonomics

Thank you!

Representing Data Collections in an SSA Form

Tommy McMichen, Nathan Greiner, Peter Zhong, Federico Sossai, Atmn Patel, Simone Campanoni
Northwestern University
Evanston, IL, USA

***Abstract*—Compiler research and development has treated computation as the primary driver of performance improvements in C/C++ programs, leaving memory optimizations as a secondary consideration. Developers are currently handed the arduous task of describing both the *semantics* and *layout* of their data in memory, either manually or via libraries, *prematurely lowering* high-level data collections to a low-level view of memory for the compiler. Thus, the compiler can only glean conservative information about the memory in a program, e.g., alias analysis, and is further hampered by heavy memory optimizations. This paper proposes the Memory Object Intermediate Representation (MEMOIR), a language-agnostic SSA form for sequential and associative data collections, objects, and the fields contained therein. At the core of MEMOIR is a decoupling of the memory used to *store data* from that used to *logically organize data*. Through its SSA form, MEMOIR compilers can perform element-level analysis on data collections, enabling static analysis on the state of a collection or object at any given program point. To illustrate the power of this analysis, we perform dead element elimination, resulting in a 26.6% speedup on `mcf` from SPECINT 2017. With the degree of freedom to mutate memory layout, our MEMOIR compiler performs *field elision* and *dead field elimination*, reducing peak memory usage of `mcf` by 20.8%.**

***Keywords*—compilers, intermediate representation, optimization**

I. Introduction

Imperative programming languages require developers to describe their programs via direct updates to the program state. Some of these languages, namely C, give developers direct access to memory, making the ceiling for manual memory optimizations nearly unlimited. Using this degree of freedom, developers have been able to build operating systems, optimizing compilers, and interpreters.

However this manual control comes with the caveat that *all memory optimizations* must be created manually. This spawned mostly out of necessity, as compilers of the time were almost solely translation units, taking C as a portable assembly language and translating it to the target machine code. As such, developers were required to *prematurely optimize* [1] memory, before the compiler could perform meaningful optimizations.

For projects where performance is a primary goal, manual memory optimizations are prevalent throughout the source code. Anytime a developer wants to change a data structure, they must consider the implications of that change on existing memory optimizations. A daunting task, as memory optimizations are performed by careful consideration of both the data structure definition and its multitude of allocations. However this leaves compilers with lacking degrees of freedom, as these decisions are fixed *before compilation*.

As a result, production compiler optimizations either focus on scalar values or are limited in their applicability when memory is involved. Modern compilers seek to perform more aggressive transformations, such as automatic vectorization and parallelization [2–14], to fully utilize modern, multi-core processors. Such transformations require precise information about data and control dependencies in the program [15–17]. For programs operating on scalars, these dependencies can be easily analyzed with SSA forms [18,19]. However these techniques are severely limited when dealing with applications operating on complex data structures holding increasingly large amounts of data that must be stored in memory.

At present, only fixed-length arrays and objects have SSA forms [20,21]. Compilers, therefore, must rely on pointer analyses for data flow information about memory objects. This information can be improved by field-sensitive [22] and type-based [23] analyses, however common manual memory optimizations create spurious dependencies and ambiguity that the compiler cannot resolve. An example of this is allocation reuse, wherein a memory location is used to represent multiple objects over the execution of the program. This optimization is common for vectors, which may use the same memory location for different elements throughout its lifetime. This aggregates the disjoint lifetimes of individual elements into a single, long-lived lifetime. Through such premature optimizations, the compiler cannot distinguish between dependencies injected by the developer and those logically necessary.

The problems facing modern compilers are the culmination of ambiguous memory behavior and lacking degrees of freedom for dependency breaking transformations. To remedy this, the compiler requires *unambiguous memory operations* via strong guarantees about the type, allocation, and usage of memory within the program. Memory behavior must be presented in a form that can be meaningfully *analyzed* and *transformed*.

This paper proposes the Memory Object Intermediate Representation (MEMOIR). MEMOIR provides the compiler with an SSA representation for sequential and associative data collections. Additionally it defines a representation for objects and their fields. By decoupling the representation of **memory used to *store data* from the memory used to *logically organize data***, MEMOIR grants powerful guarantees for transformation and enables sparse data flow analysis for elements of collections and fields of objects via def-use chains. MEMOIR also grants the degrees of freedom necessary to change the memory layout of individual objects as well as the broader memory structure of a program. By providing an IR that is amenable to both analysis and transformation, MEMOIR compilers can emit performant code without placing the burden of memory optimization on developers.

979-8-3503-9509-9/24 © 2024 IEEE

https://mcmichen.cc/files/MEMOIR_CGO_2024.pdf

https://golf0ned.github.io/