

Flexible Efficient Memory Management and Compile-Time Specialization via Context

Sam Estep and Joshua Sunshine, Carnegie Mellon University
@ MWPLS 2025



Problem

How do we write efficient programs?

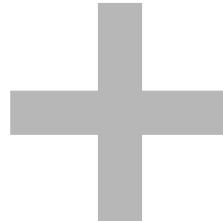
Two main ingredients of a fast program

Ingredient #1:

Tell the compiler what your program is.

In other words, give it statically typed function bodies with direct calls so it can inline things when it wants and then work its magic.

Compilers are good at this.



Ingredient #2:

Manage memory efficiently.

Memory is far away from the CPU.

The compiler largely can't change how your data structures are stored.

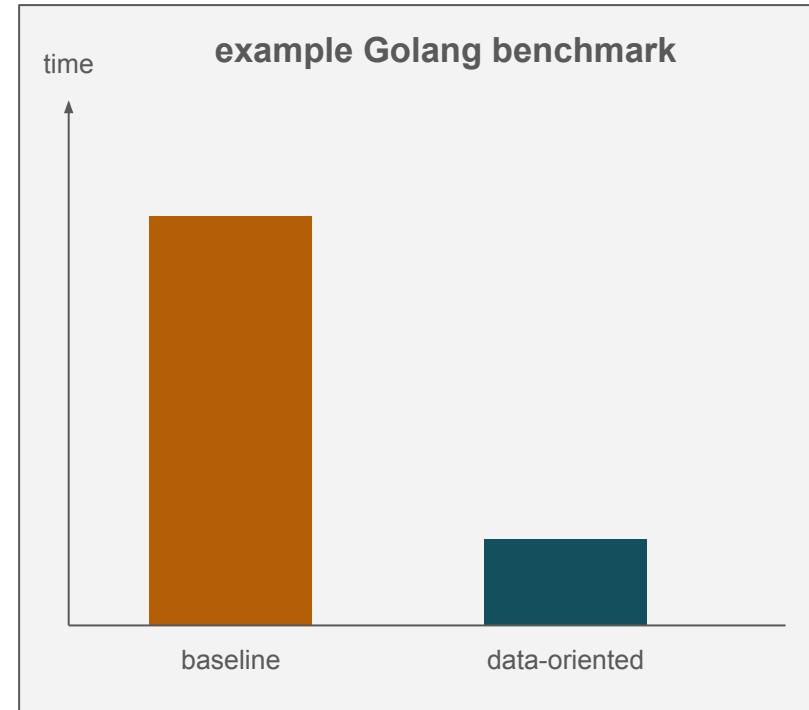
Compilers are not good at this.

Smoke check: does memory really matter? Is GC enough?

Garbage collection is not enough.

The performance gap between a naive program and an optimized program has gotten larger as hardware has gotten deeper cache hierarchies.

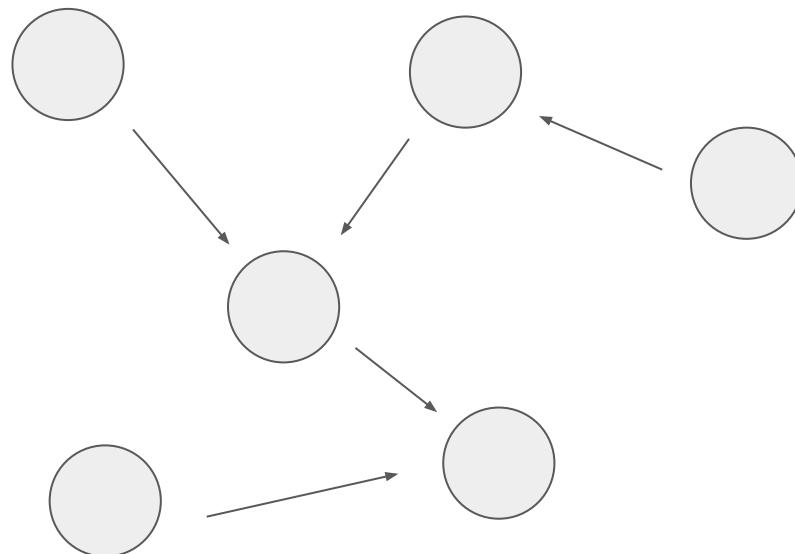
There are domains in which this additional performance matters. This work is about those domains.



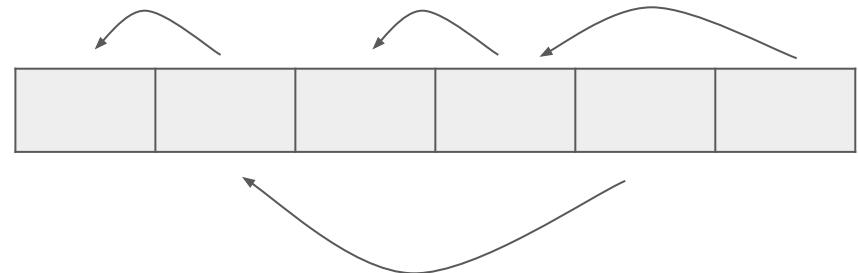
<https://jollygood.prose.sh/data-oriented-design-benchmarks>

So how do we write memory-efficient programs?

Typical code:



Performant code:



So how do we write memory-efficient programs?

Typical code:

```
enum BinOp { Add, Sub, Mul, Div }
enum Expr {
    Binary(
        BinOp,
        Box<Expr>,
        Box<Expr>,
    ),
    Literal(i64),
}
```

Performant code:

```
struct ExprPool(Vec<Expr>);
struct ExprRef(u32);
enum BinOp { Add, Sub, Mul, Div }
enum Expr {
    Binary(
        BinOp,
        ExprRef,
        ExprRef,
    ),
    Literal(i64),
}
```



<https://www.cs.cornell.edu/~asampson/blog/flattening.html>

What's the problem, then?

This needs to be
passed around everywhere.

And we're bypassing
memory features built into the language.

Overall, restructuring one's code like this
is horrendously viral and invasive.

```
struct ExprPool(Vec<Expr>);  
struct ExprRef(u32);  
enum BinOp { Add, Sub, Mul, Div }  
enum Expr {  
    Binary(  
        BinOp,  
        ExprRef,  
        ExprRef,  
    ),  
    Literal(i64),  
}
```

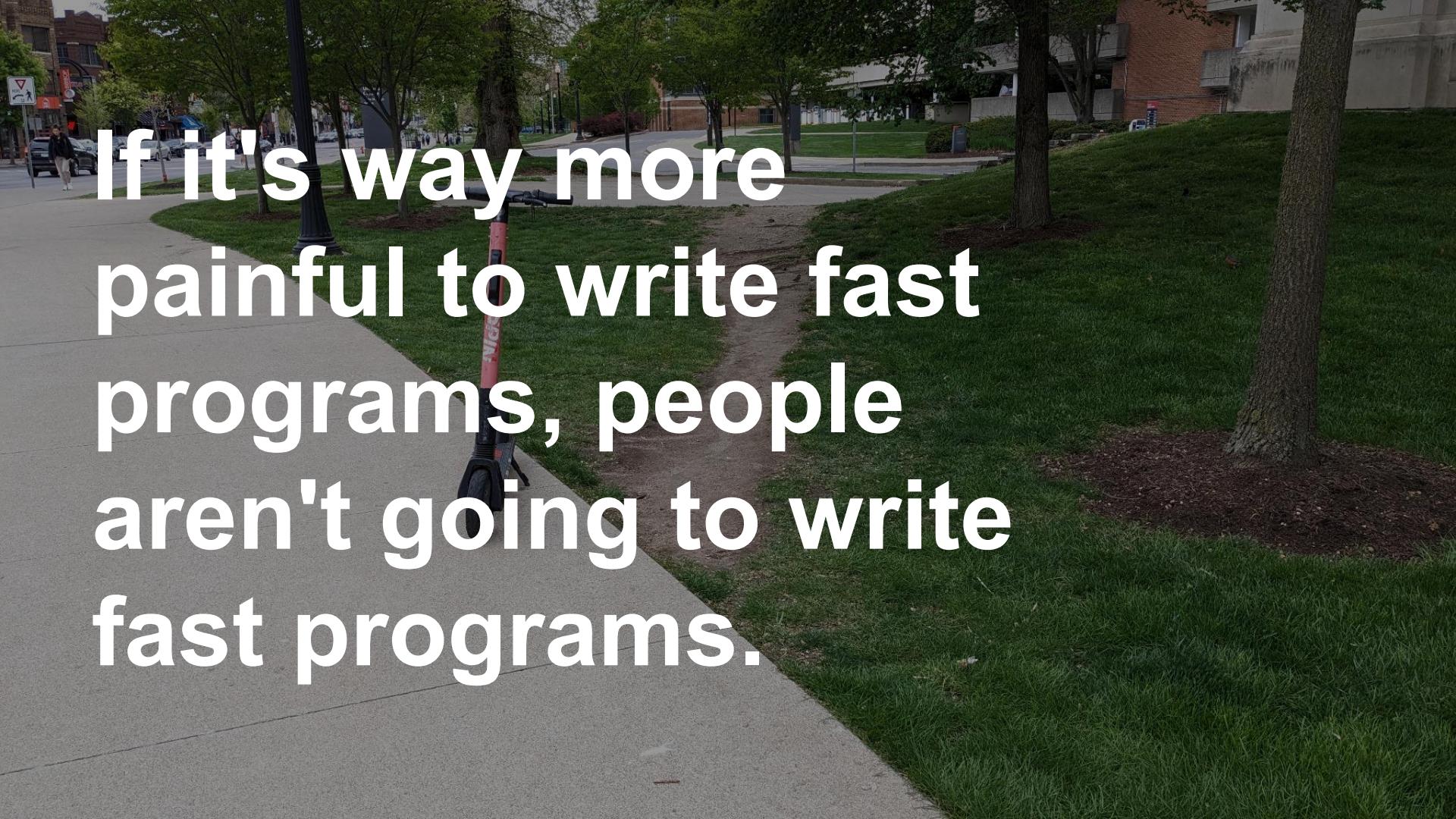
Writing fast code isn't just a bit worse, it's a lot worse

The code I'd be able to write if I weren't doing
data-oriented programming

```
let fields
    = obj.local().structdef().fields();
```

The code I'm forced to write instead

```
let fields = &self.x.ir.fields[
    self.x.ir.structdefs[
        self.x.ir.types[
            self.x.ir.locals[
                obj
            ].index()
        ].structdef()
    ].fields
];
```

A photograph of a city street scene. On the left is a paved sidewalk where a red and black electric scooter is parked. To the right of the sidewalk is a grassy area with several young trees. In the background, there are buildings, a road with a few cars, and a person walking away from the camera.

If it's way more
painful to write fast
programs, people
aren't going to write
fast programs.

Idea

Let everything be contextual.

This work builds on algebraic effects and coeffects

Effects "represent what your program does to the world"

```
effect print : String -> unit  
print "Hello!"
```

Coeffects "track what your program requires from the world"

```
let greeting =  
  if ?formal then "Good evening"  
  else "Hi"  
in greeting ++ ", " ++ name
```



<https://tomasp.net/coeffects/>

Context is composable

Defining contextual items

```
type DefinedType = (Foo, Bar);

type TypeThatCanBeBound;

fn defined_function() {
    do_stuff();
}

fn function_that_can_be_bound();

val defined_value = 42;

val value_that_can_be_bound: String;
```

Consuming contextual items

```
context composite_context =
    TypeThatCanBeBound,
    function_that_can_be_bound,
    value_that_can_be_bound,
;

fn foo[composite_context]() {
    function_that_can_be_bound();
    println(value_that_can_be_bound);
}
```

How does this help us?

Without context, it's all methods on one big context type

```
struct FooRef(u32);

struct Foo {
    x: Bar,
}

struct Context {
    foos: Vec<Foo>,
}

impl Context {
    fn bar(&self, foo: FooRef) -> Bar {
        self.foos[foo.0].x
    }

    fn example(&self, foo: FooRef) {
        self.bar(foo).something();
    }
}
```

With context built into the language...

```
type FooRef = u32;

struct Foo {
    x: Bar,
}

val foos: List[Foo];

fn FooRef.bar[foos](): Bar {
    foos[this].x
}

fn example[foos](foo: FooRef) {
    foo.bar().something();
}
```

Bring it back to ingredient #2 (manage memory efficiently)

Now we can actually write this code!

```
let fields  
= obj.local().structdef().fields();
```

Ingredient #2:

Manage memory efficiently.

Memory is far away from the CPU.

The compiler largely can't change how your data structures are stored.

Compilers are not good at this.

And contextual types help us keep ingredient #1

Ingredient #1:

Tell the compiler what your program is.

In other words, give it statically typed function bodies with direct calls so it can inline things when it wants and then work its magic.

Compilers are good at this.

```
type Int64;

context Std =
    Int64,
    # ...
;

fn do_stuff[Std]() {
    let foo: Int64 = # ...
    # ...
}

fn safe() {
    with_checked_int64[do_stuff]();
}

fn fast() {
    with_unchecked_int64[do_stuff]();
}
```

Thank you!

Interested in these ideas? Let's get in touch! estep@cmu.edu

