How Apache Beam Makes Your Data Processing Go!

Robert Burke
@lostluck

Hello GopherCon, and welcome to How Apache Beam Makes Your Data Processing Go!

https://www.gophercon.com/agenda/session/944207

About Me

Robert Burke in his natural habitat

Senior SWE @ Google

Cloud Dataflow

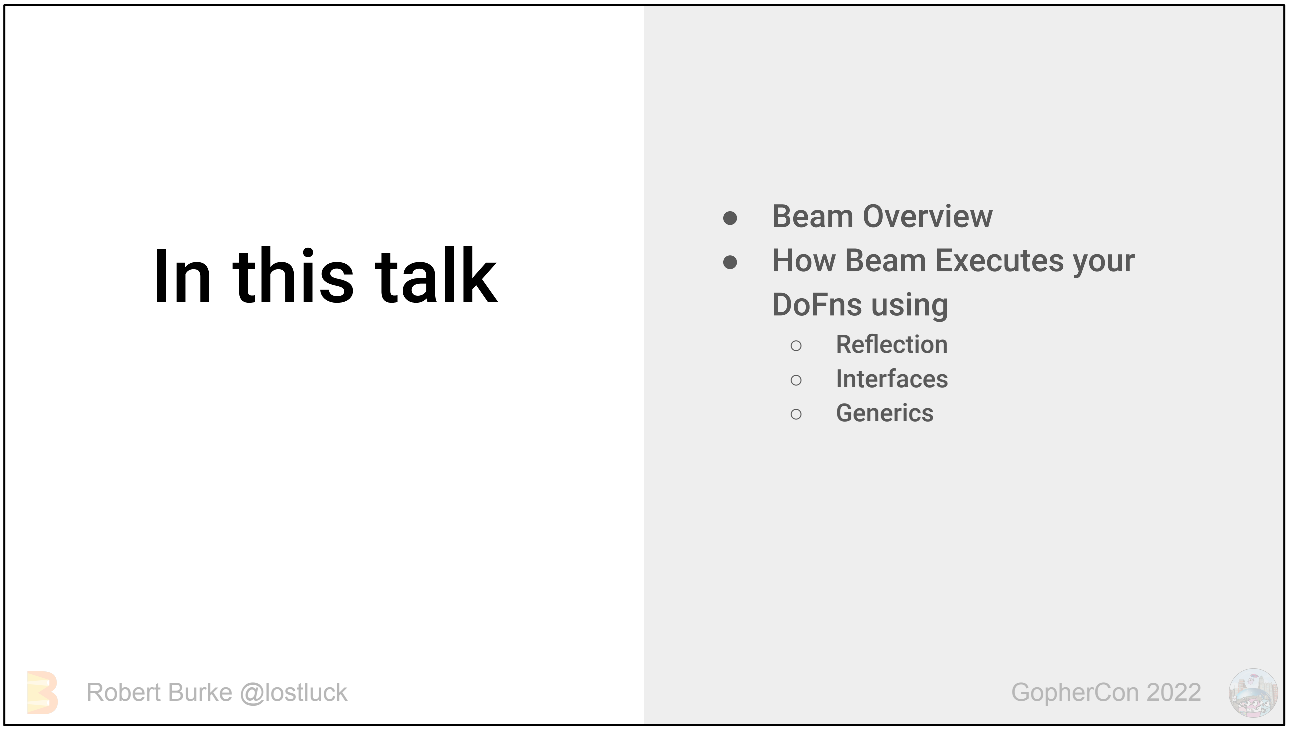
Apache Beam Go SDK

Playing with Go since v1.0

Moustache + Fun Hair + Glasses

I'm Robert Burke,
 a Senior Software Engineer at Google
on Cloud Dataflow.
I've been working on the Apache Beam Go SDK for about 4-5 years, so hopefully I know what I'm talking about by now.
I've been playing with Go since around v1, and it's been lovely to see how it's grown.

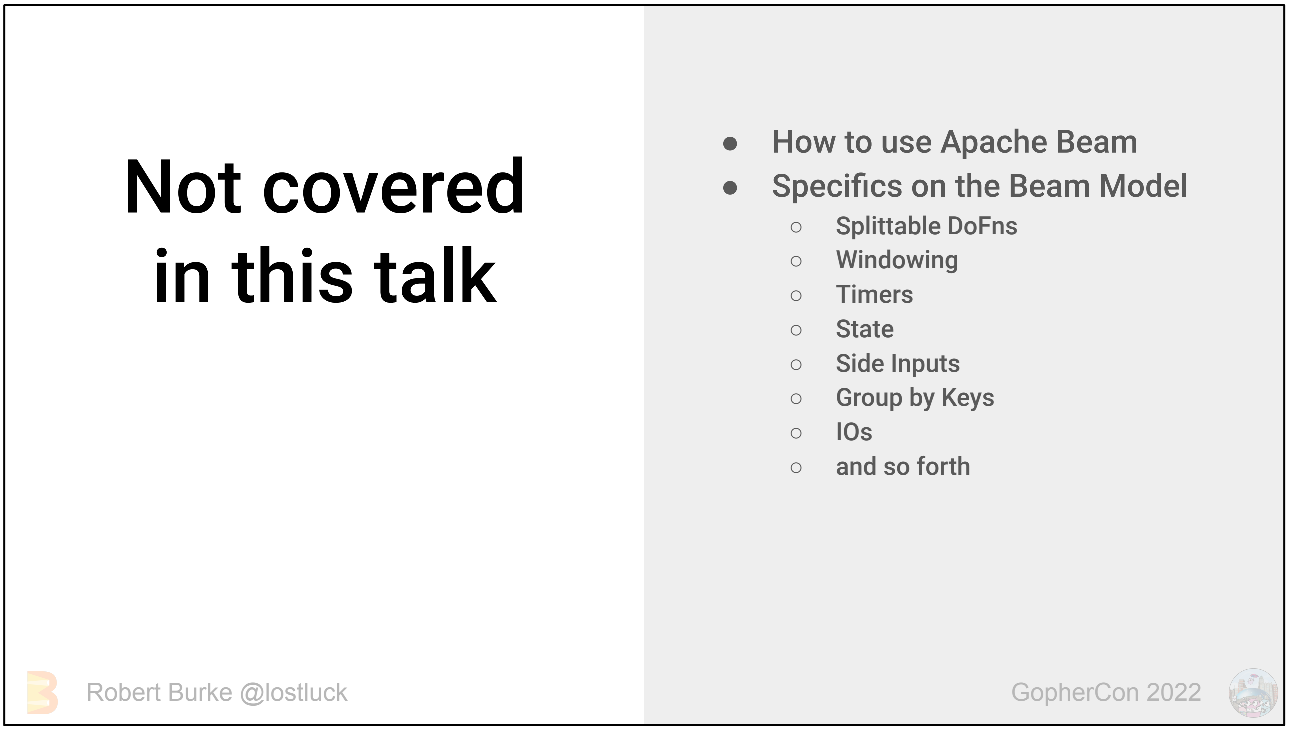
In case you haven't seen me around the conference yet, I'm the one with the combination of a moustache and matching glasses. Come say hi!



In this talk

- Beam Overview
- How Beam Executes your DoFns using
  - Reflection
  - Interfaces
  - Generics

This talk is going to cover a quick overview of what Apache Beam is,From there we get into the Go of it, and how the SDK uses Reflection, Interfaces, and Generics to execute your code, which we call DoFns.
Finally I'll get into how Go affected the SDK's design.



Not covered in this talk

- How to use Apache Beam
- Specifics on the Beam Model
  - Splittable DoFns
  - Windowing
  - Timers
  - State
  - Side Inputs
  - Group by Keys
  - IOs
  - and so forth

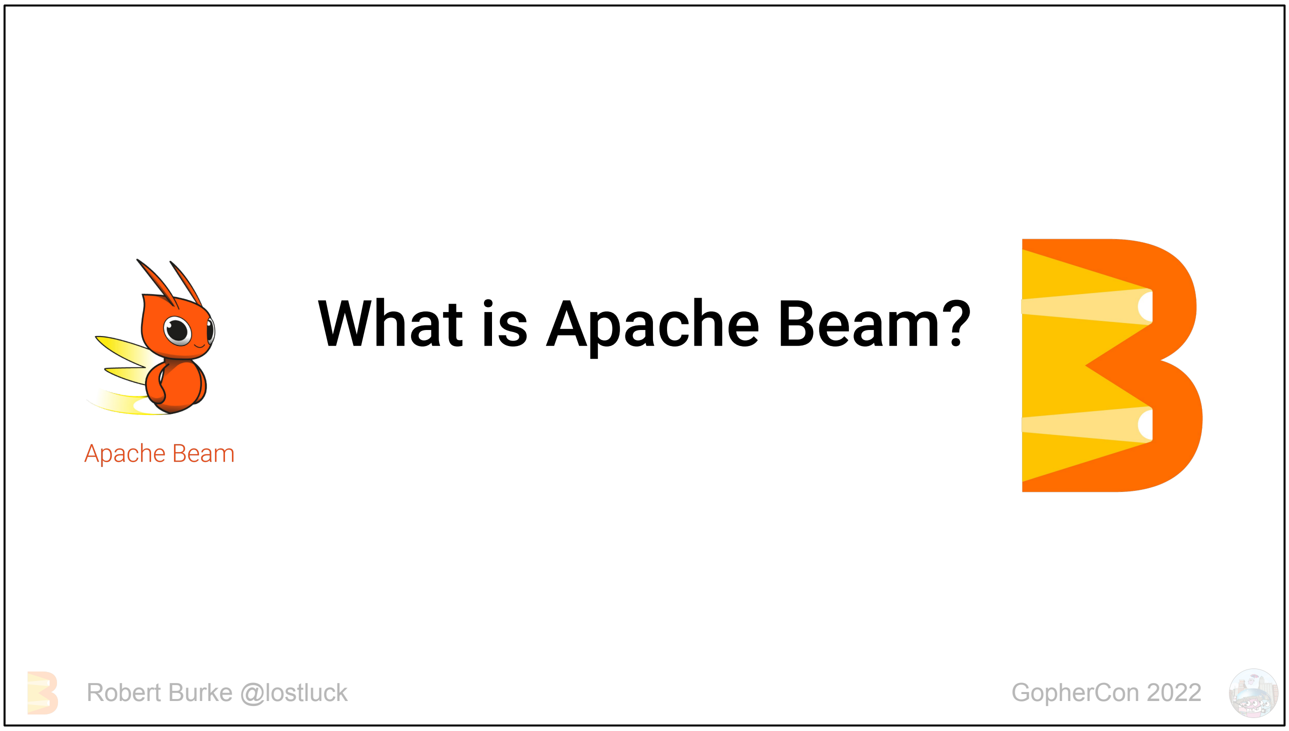
Robert Burke @lostluck

GopherCon 2022

Importantly, for those of you watching the video later, this talk doesn't cover how to use Apache Beam, or anything specific in the Beam Model.

For that the Beam Programming Guide and quickstarts are the best.

I'll be linking this slide deck in the GopherCon Discord and it has links to a variety of resources at the end, or links to code in github in the speaker notes.



Apache Beam

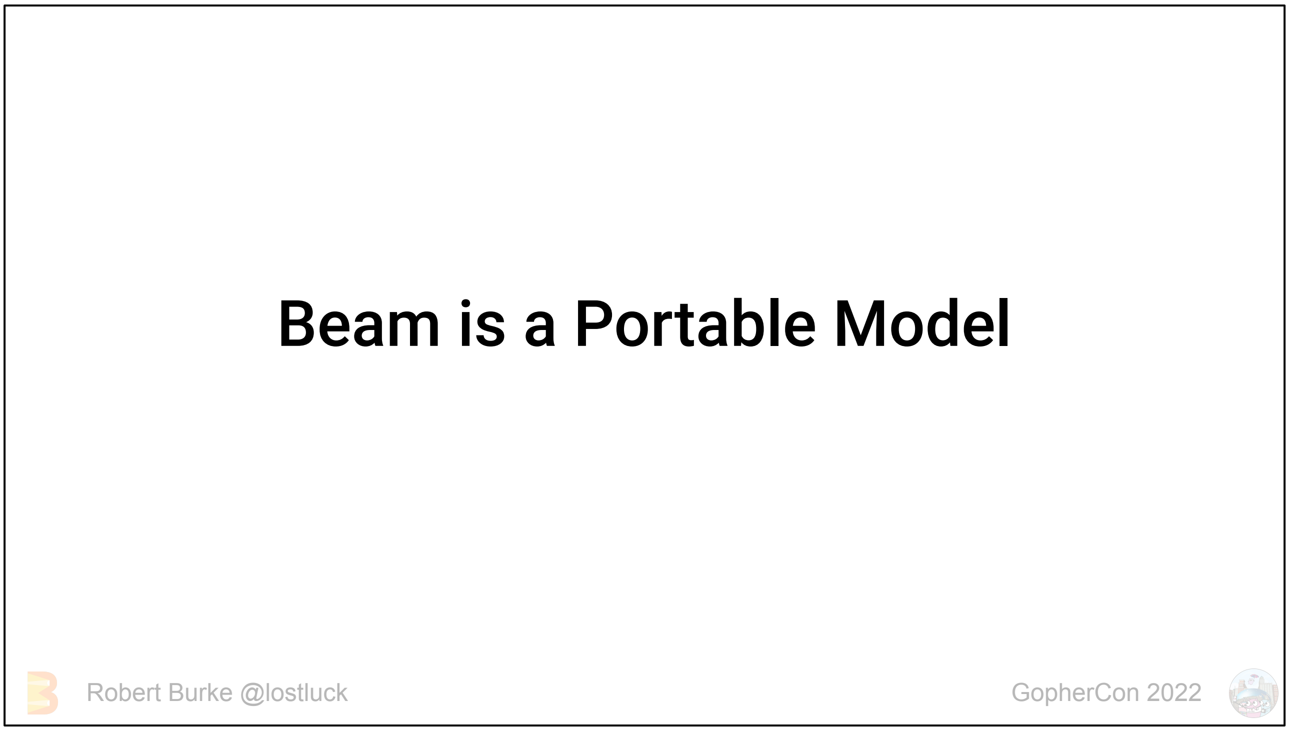
What is Apache Beam?

Robert Burke @lostluck

GopherCon 2022

- Quick Audience Poll, Everyone Raise hands, Keep up
    - You have heard of Beam before this talk
    - You have used Beam at all
    - You have used the Go SDK
- Point at remainder, these are the ones to ask questions to for later.

So for everyone else. What is Apache Beam?


Beam is a Portable Model

Robert Burke @lostluck

GopherCon 2022

Beam is a Portable Model

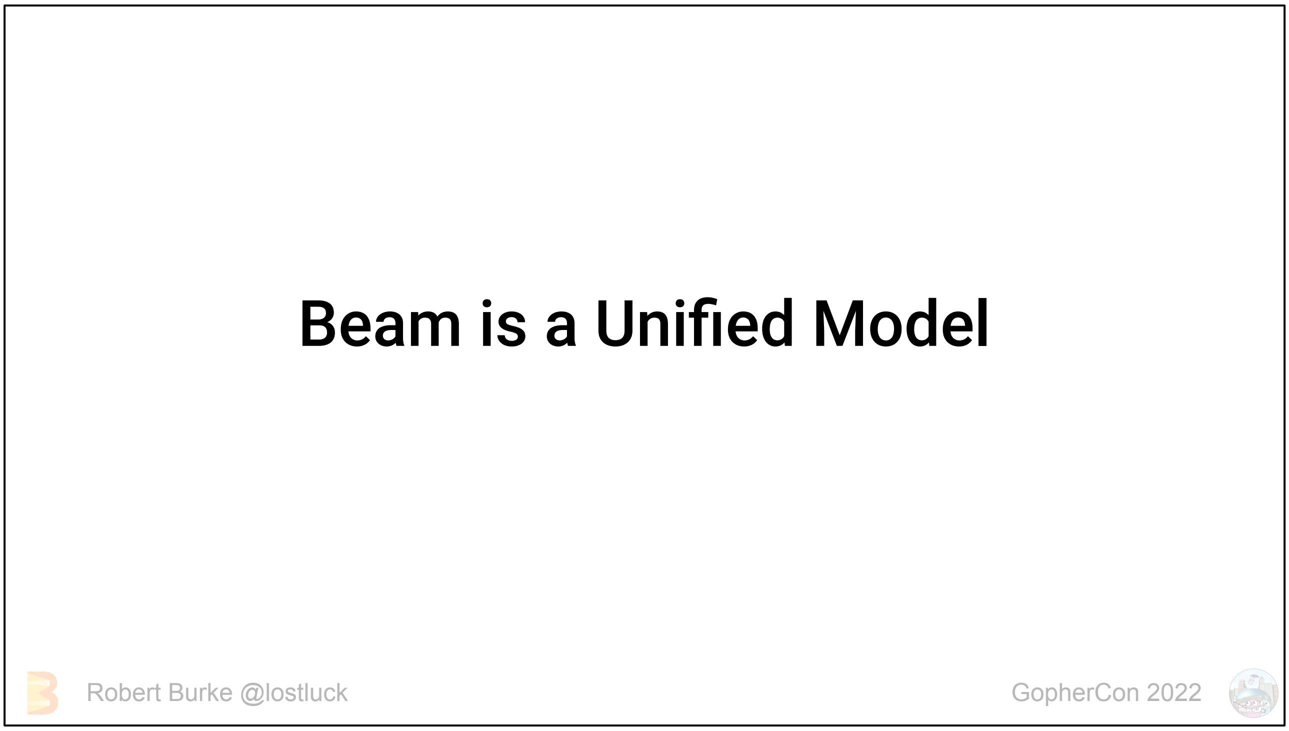
It’s not a specific language SDK or Runner project.

It’s a way to do your data processing in a language that’s familiar to you, on top of infra you are already using.

It’s a way to divide the complexities of Data Processing between Runners and User SDKs.

And not just be able to re-use pipelines on different runners as your infrastructure changes.

But also be able to re-use transforms from other SDKs in the SDK language you’re comfortable with.


Beam is a Unified Model

Robert Burke @lostluck

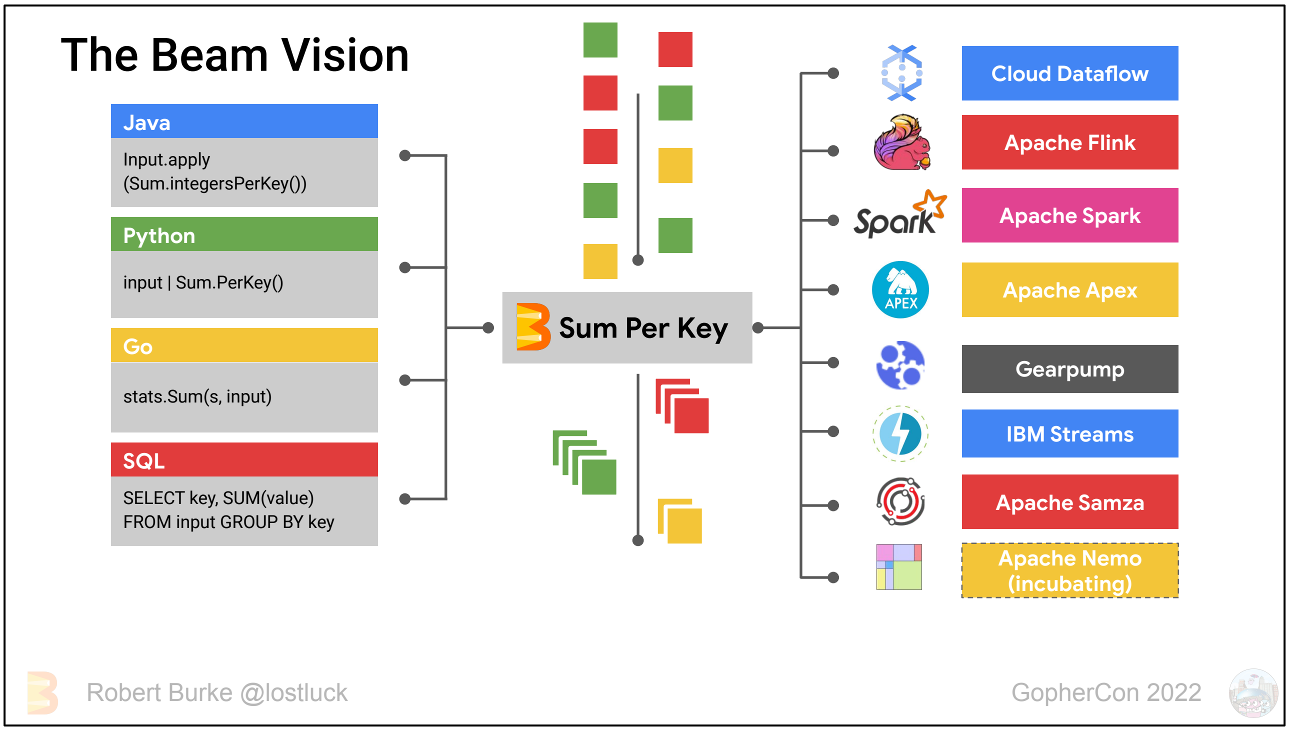
GopherCon 2022

Beam is a Unified Model

Beam resolves a tension between Batch Processing and Stream Processing.

Especially when stream processing is trading off between correctness and low latency.

It’s a way of thinking about how to process your data, so you can reuse the same transforms in both modes.



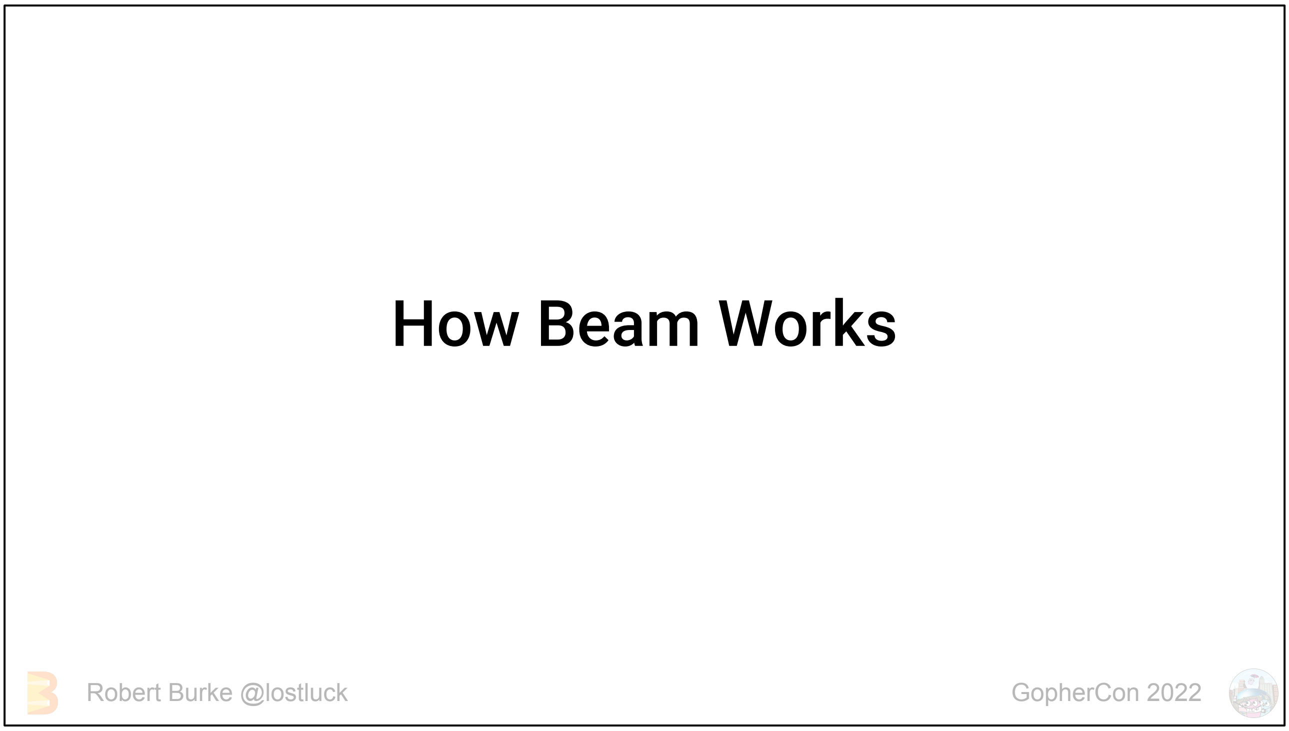
The Beam Vision

Beam wants to support End users who are just interested in writing data processing pipelines.

They want to use the language that they want to use and choose the runtime that works for them, whether it's on premises, on a hand tuned cloud cluster, or on a fully managed service.

In addition, we want to develop stable APIs and documentation to allow others in the open source community to create Beam SDKs in other languages and to provide runners for alternate distributed processing environments.

[Slide liberally borrowed from Kerry Donny-Clark]



How Beam Works

Robert Burke @lostluck

GopherCon 2022

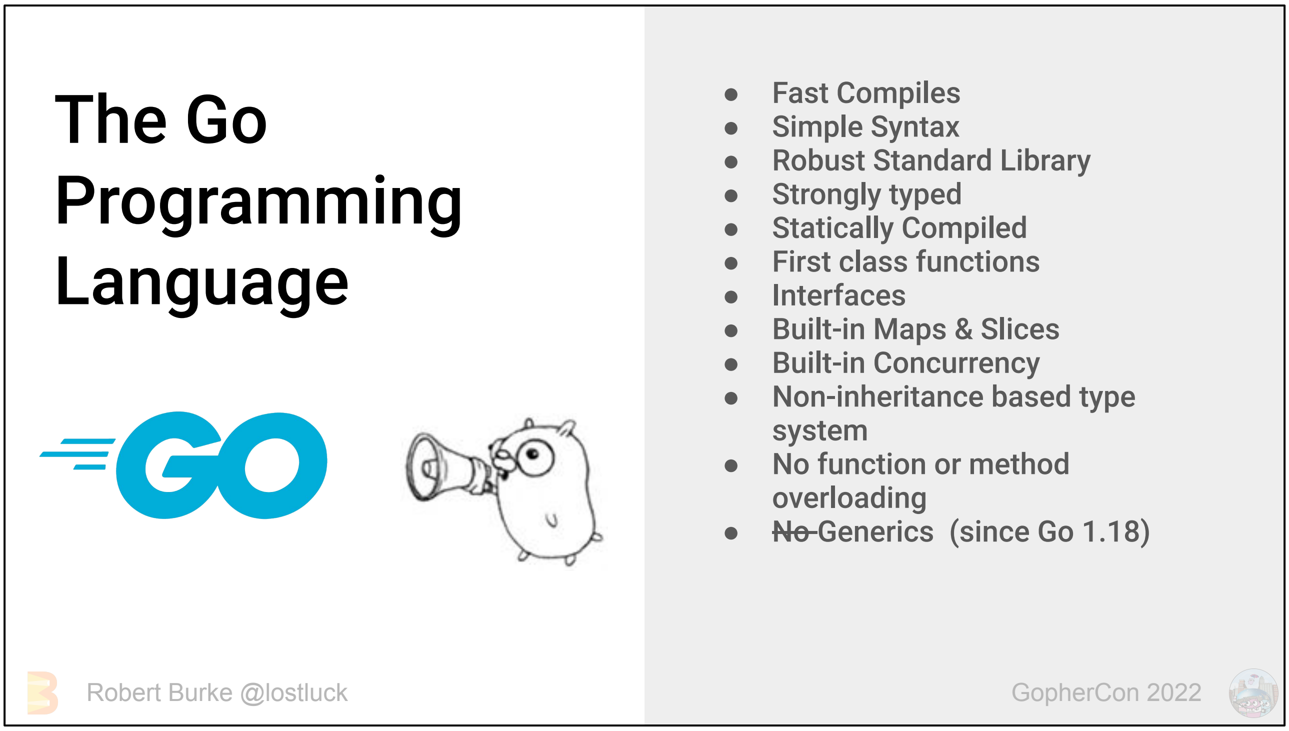
So that’s the 10-thousand feet view of Beam.

Lets get down to how Beam Works.

There are lots of problems for a Beam SDK to solve, but I’d bring it down to these two core problems.

1 How to have an API that feels good to experienced users of the language, and
2. How to execute that code.

So how do we begin to do that with Go?



The Go Programming Language

- Fast Compiles
- Simple Syntax
- Robust Standard Library
- Strongly typed
- Statically Compiled
- First class functions
- Interfaces
- Built-in Maps & Slices
- Built-in Concurrency
- Non-inheritance based type system
- No function or method overloading
- ~~No~~ Generics (since Go 1.18)

Robert Burke @lostluck

GopherCon 2022

The apocryphal story is that Go was originally conceived of during a long C++ compilation.

The Go Authors took that as a sign that programming could be faster, and they focused on making a new language that's easier to read, easier to write, compiles quickly and executes quickly.

This ethos drove the design of the language, leading to one that's Strongly typed, statically compiled, avoids inheritance and overloading.

On top of that it bring concurrency into the language itself, instead of leaving it to libraries.
It also has a standard library that is useful for the internet age.
As one of the keynotes mentioned, it's well suited for Distributed Computing.

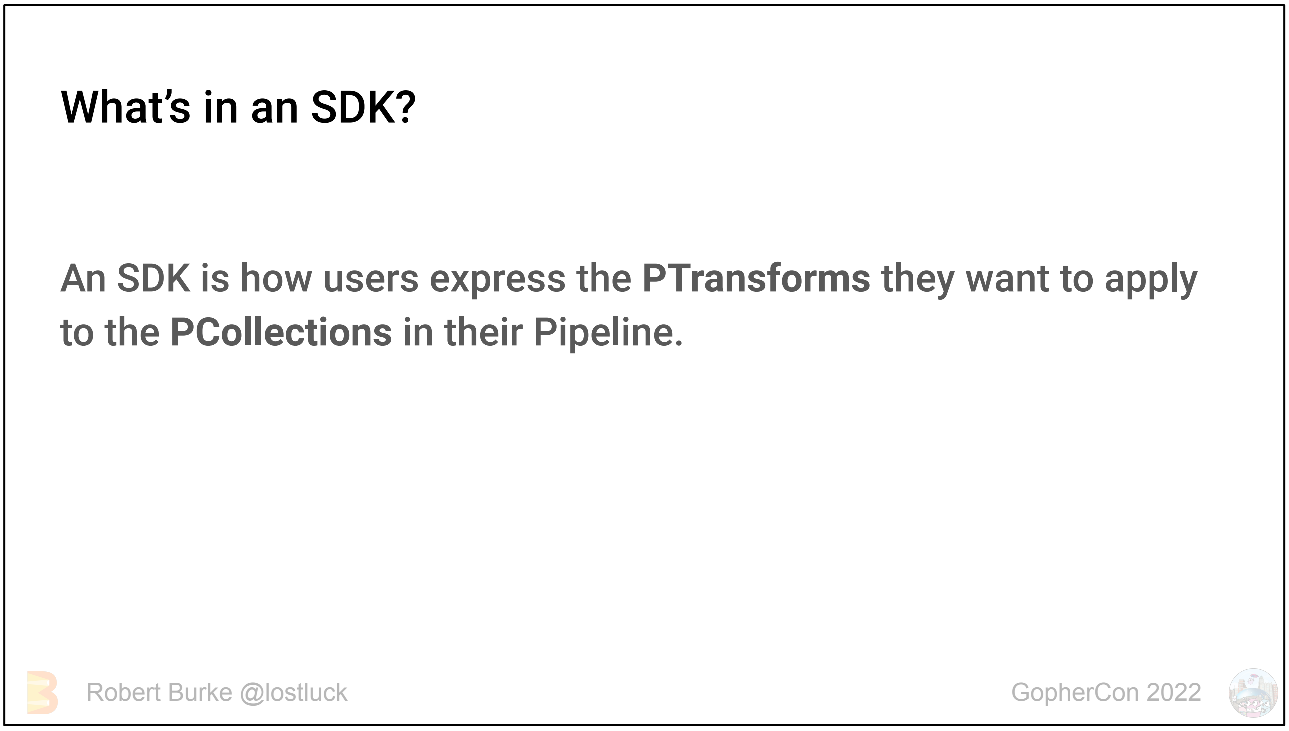
None of these ideas were new, but they had not come together in this way before.

But others seem to have agreed, and it's become the Language Upon Which The Cloud Is Built, powering things Docker and Kubernetes.

Until very recently, Go did not have Generics.

Like with any programming language, Go's features and affordances affect how people design APIs and the resulting SDKs.

That directly affects the user experience, from initial authoring, to how to deploy it to production.



What’s in an SDK?

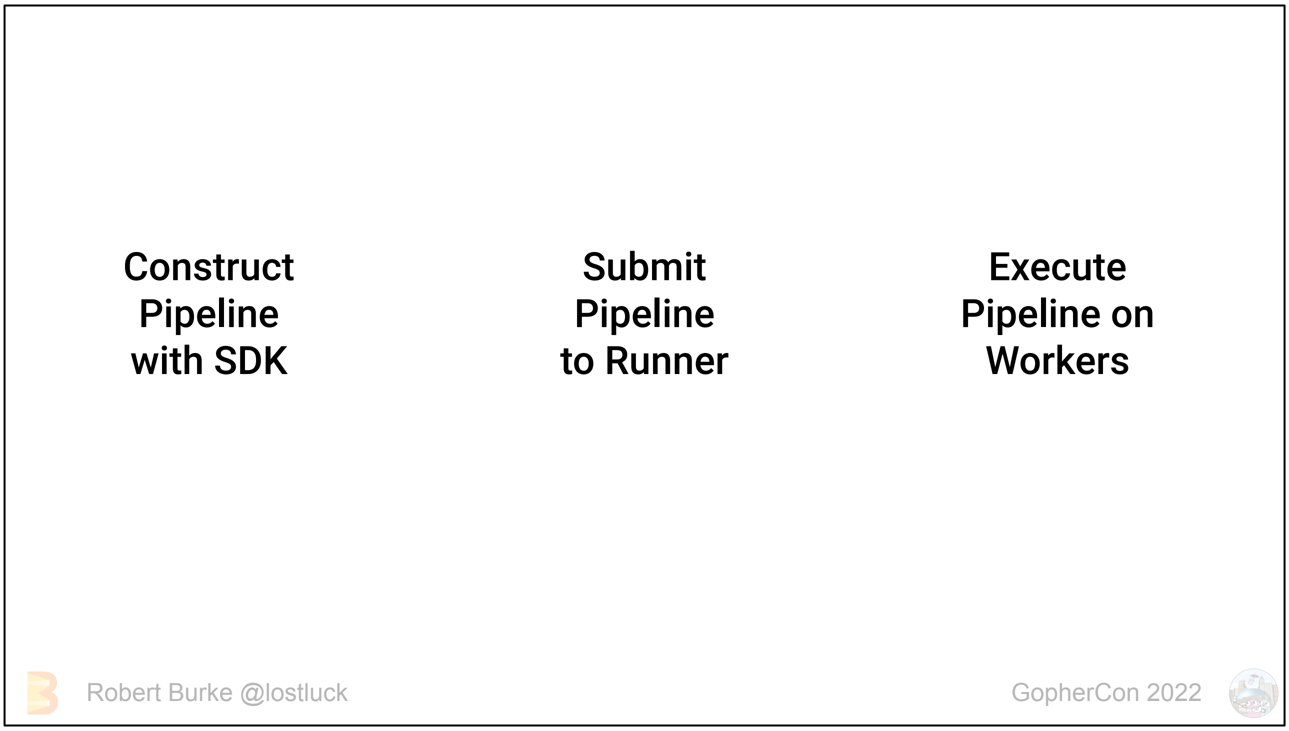
An SDK is how users express the **PTransforms** they want to apply to the **PCollections** in their Pipeline.

For Beam, the SDK is an implementation of the Beam abstract model.

A simple interpretation of this is, is how users express the PTransforms they want to apply on the PCollections in their Pipeline.

The SDK is there to make sure everything makes sense, and that it can execute your DoFns as we call them, at Pipeline Execution time, while catching potential runtime errors earlier at Pipeline Construction time.

Let’s see how that looks for Go.



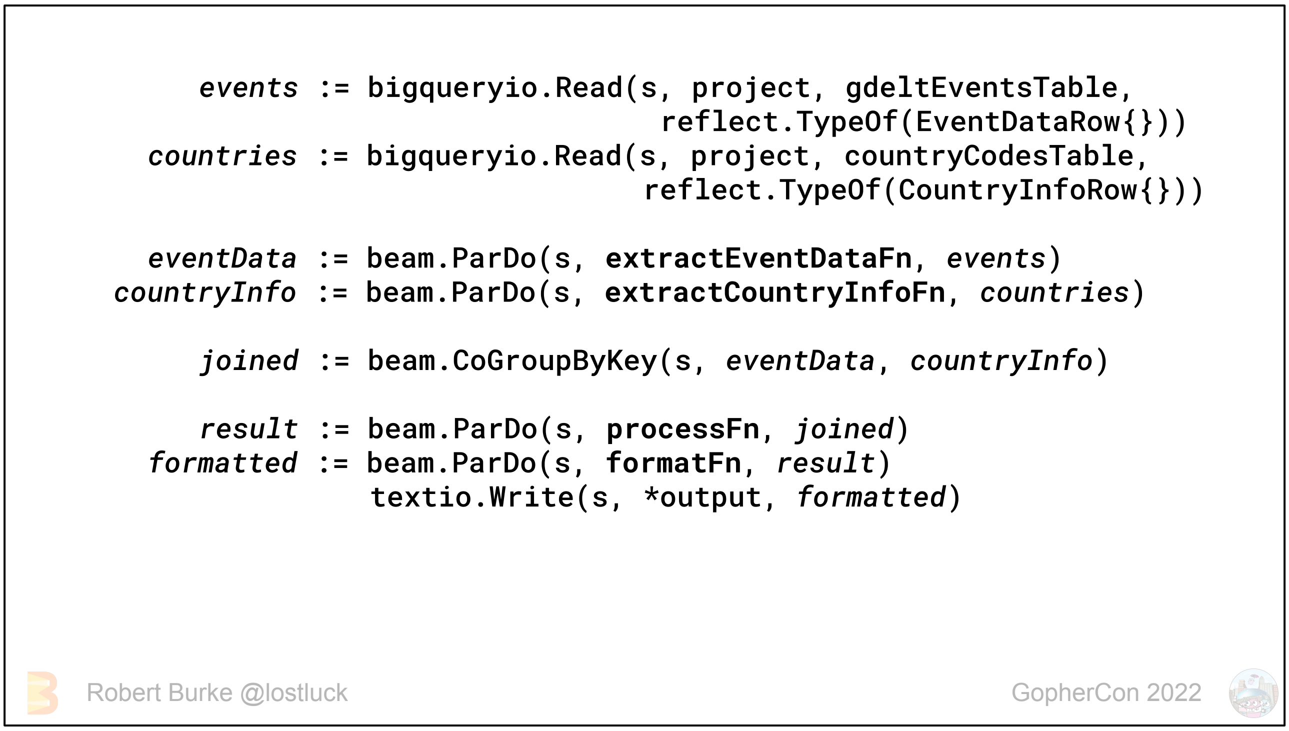
Construct Pipeline with SDK

Submit Pipeline to Runner

Execute Pipeline on Workers

3 steps

- Construct a Pipeline DAG with DoFns (your code) and other Transforms
  - Deferred execution, until submitted to runner.
- Submit that pipeline to a Runner serializing the essentials of the pipeline, and long with any necessary configuration options.
- The runner then decides how to execute it, and where, whether that’s locally, or on remote VMs.



```
    events := bigqueryio.Read(s, project, gdeltEventsTable,
                                  reflect.TypeOf(EventDataRow{}))
 countries := bigqueryio.Read(s, project, countryCodesTable,
                                 reflect.TypeOf(CountryInfoRow{}))

 eventData := beam.ParDo(s, extractEventDataFn, events)
countryInfo := beam.ParDo(s, extractCountryInfoFn, countries)

    joined := beam.CoGroupByKey(s, eventData, countryInfo)

    result := beam.ParDo(s, processFn, joined)
 formatted := beam.ParDo(s, formatFn, result)
              textio.Write(s, *output, formatted)
```

Robert Burke @lostluck

GopherCon 2022

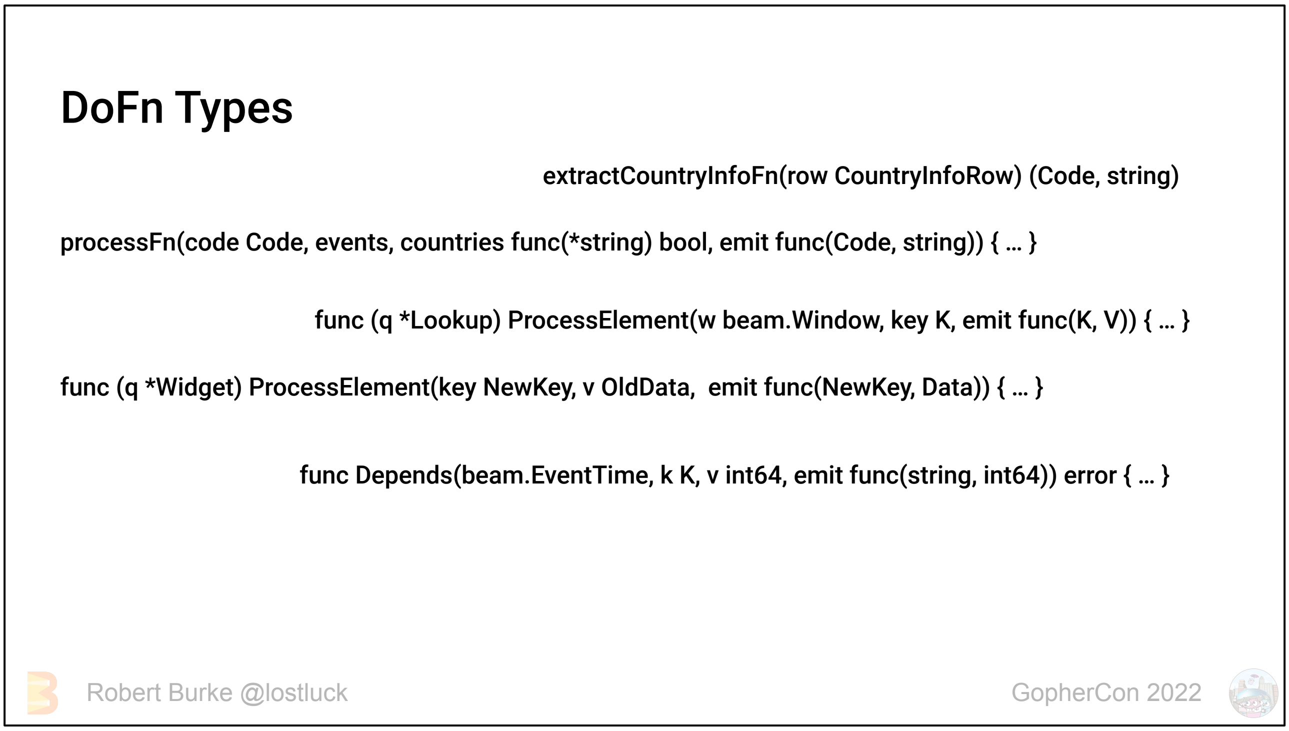
This is what pipeline construction looks like, which a bit of additional formatting.

As promised this isn't to teach you Beam technique though.

Lightly, we have transforms (the calls), which we pass in PCollections (the variables).

The 2nd parameter to the "beam.ParDo" calls are in bold. These are a pipeline's "DoFns". These are your code that you want to execute on your data.

Pipeline Borrowed from Join cookbook:
https://github.com/apache/beam/blob/master/sdks/go/examples/cookbook/join/join.go



DoFn Types

```
extractCountryInfoFn(row CountryInfoRow) (Code, string)

processFn(code Code, events, countries func(*string) bool, emit func(Code, string)) { … }

func (q *Lookup) ProcessElement(w beam.Window, key K, emit func(K, V)) { … }

func (q *Widget) ProcessElement(key NewKey, v OldData,  emit func(NewKey, Data)) { … }

func Depends(beam.EventTime, k K, v int64, emit func(string, int64)) error { … }
```

The problem we run into is DoFn types.

This is a DoFn.

So is this.

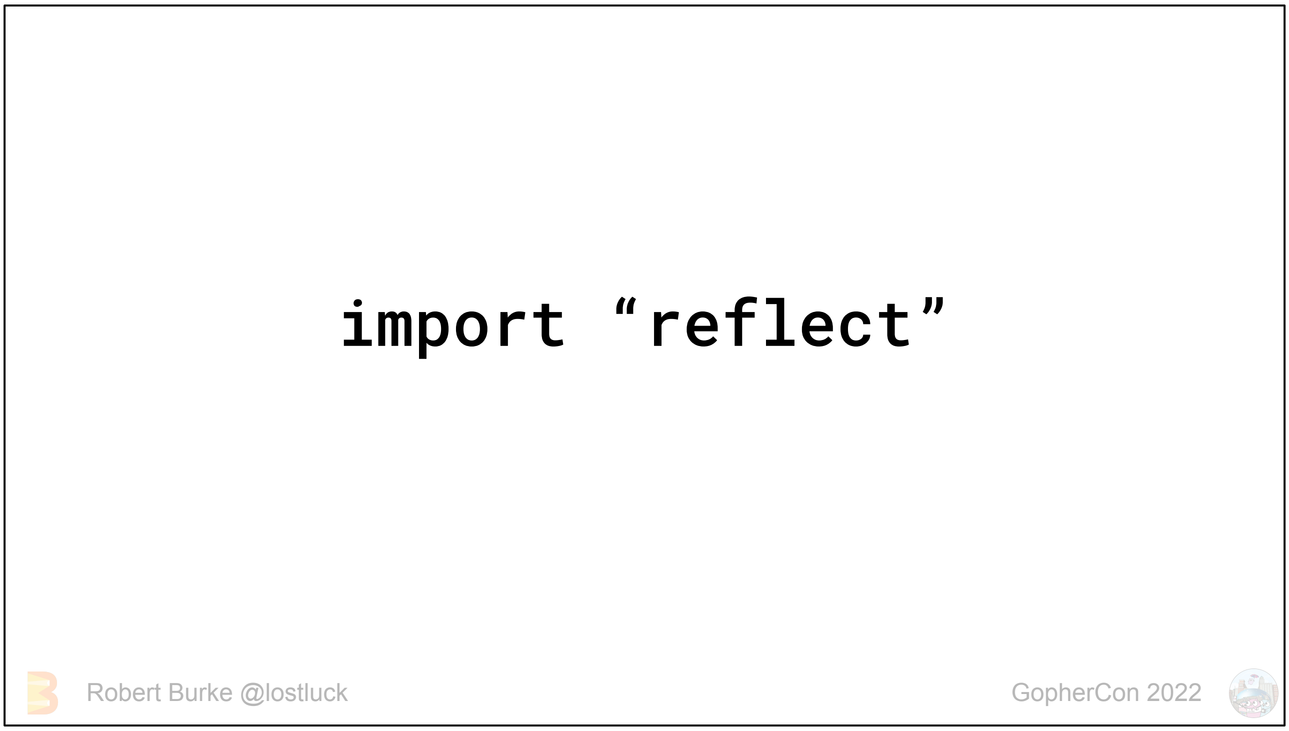
DoFns aren't only functions, they're structs with ProcessElement methods too.

But the observant of you will see that none of these types match up, so interfaces, or vanilla function types are right out.

But we need to be able to call and execute all of them.

We could have gone with an approach that just used the empty interface. "Any" I guess now.
Afterall, it's easy to do everything with interfaces.

But that approach loses information, and could require too much repetition elsewhere. So we kept the user types. This brings DoFns closer to normal Go code. Afterall, static types are a valuable Go feature.



import “reflect”

https://pkg.go.dev/reflect

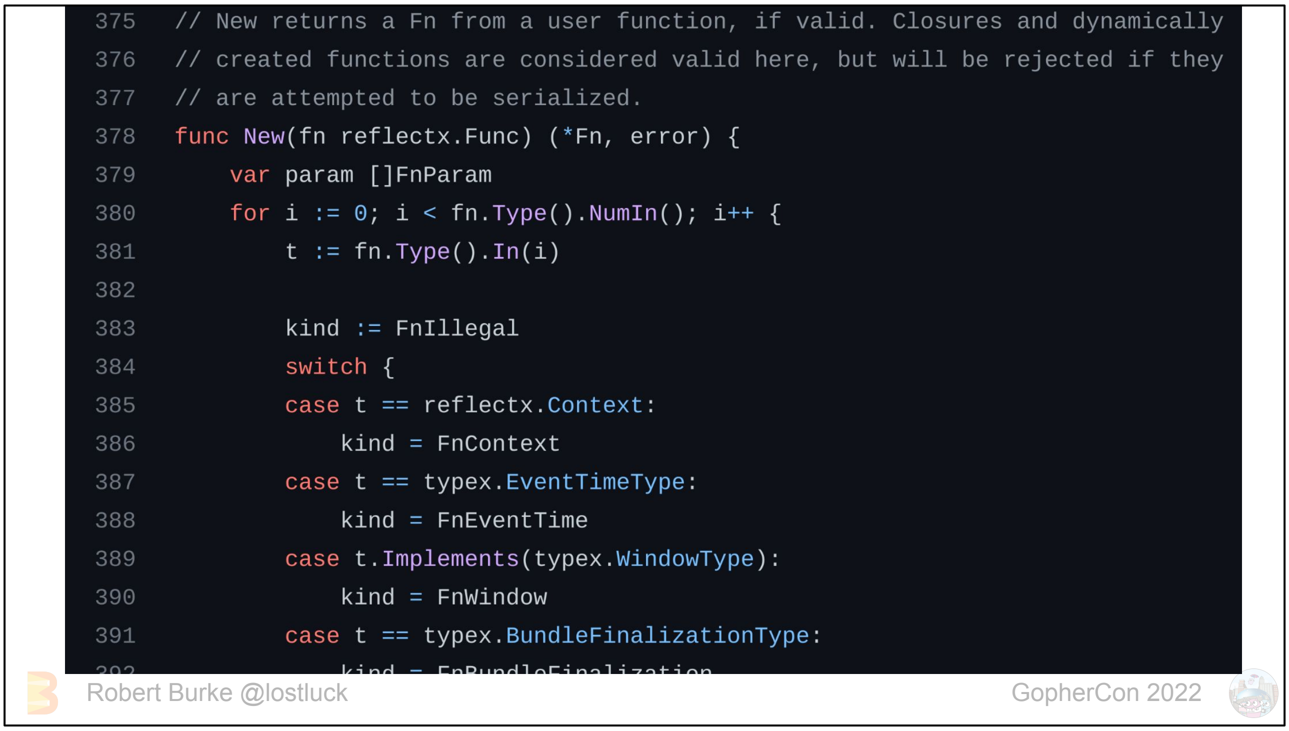
Invariably, in this sort of thing requires use of the reflect package.

`reflect.Type`

https://pkg.go.dev/reflect#Type

Reflect has a function, TypeOf which gets the type of any value you pass to it.

From there we can do a bit of work to make sure the pipeline is type safe, at construction time.



```go
// New returns a Fn from a user function, if valid. Closures and dynamically
// created functions are considered valid here, but will be rejected if they
// are attempted to be serialized.
func New(fn reflectx.Func) (*Fn, error) {
    var param []FnParam
    for i := 0; i < fn.Type().NumIn(); i++ {
        t := fn.Type().In(i)

        kind := FnIllegal
        switch {
        case t == reflectx.Context:
            kind = FnContext
        case t == typex.EventTimeType:
            kind = FnEventTime
        case t.Implements(typex.WindowType):
            kind = FnWindow
        case t == typex.BundleFinalizationType:
            kind = FnBundleFinalization
```

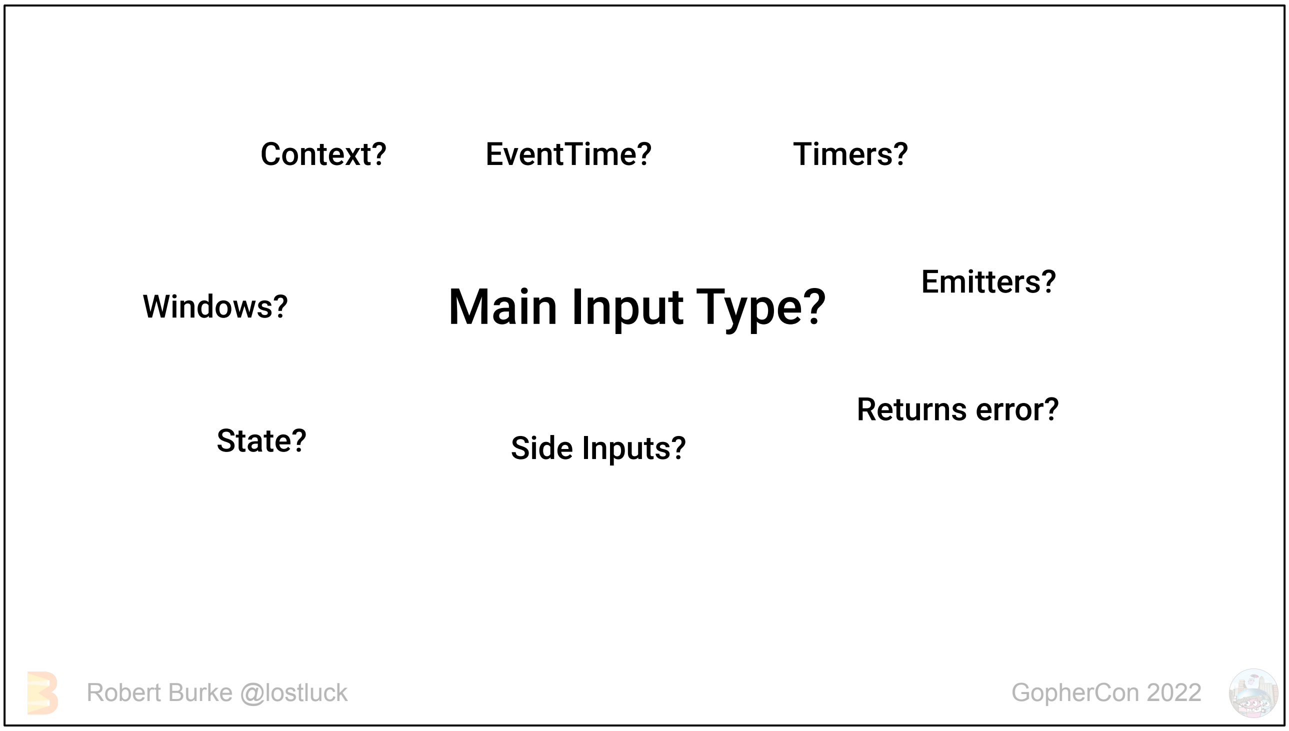
Robert Burke @lostluck GopherCon 2022

Whenever you write reflection based code, you get used to writing switches.

So we take the function or method type, and look through it’s parameters.

This tells us a lot about how the DoFn processes elements.

https://github.com/apache/beam/blob/master/sdks/go/pkg/beam/core/funcx/fn.go#L375



Context? EventTime? Timers?

Windows?

Main Input Type?

Emitters?

State? Side Inputs? Returns error?

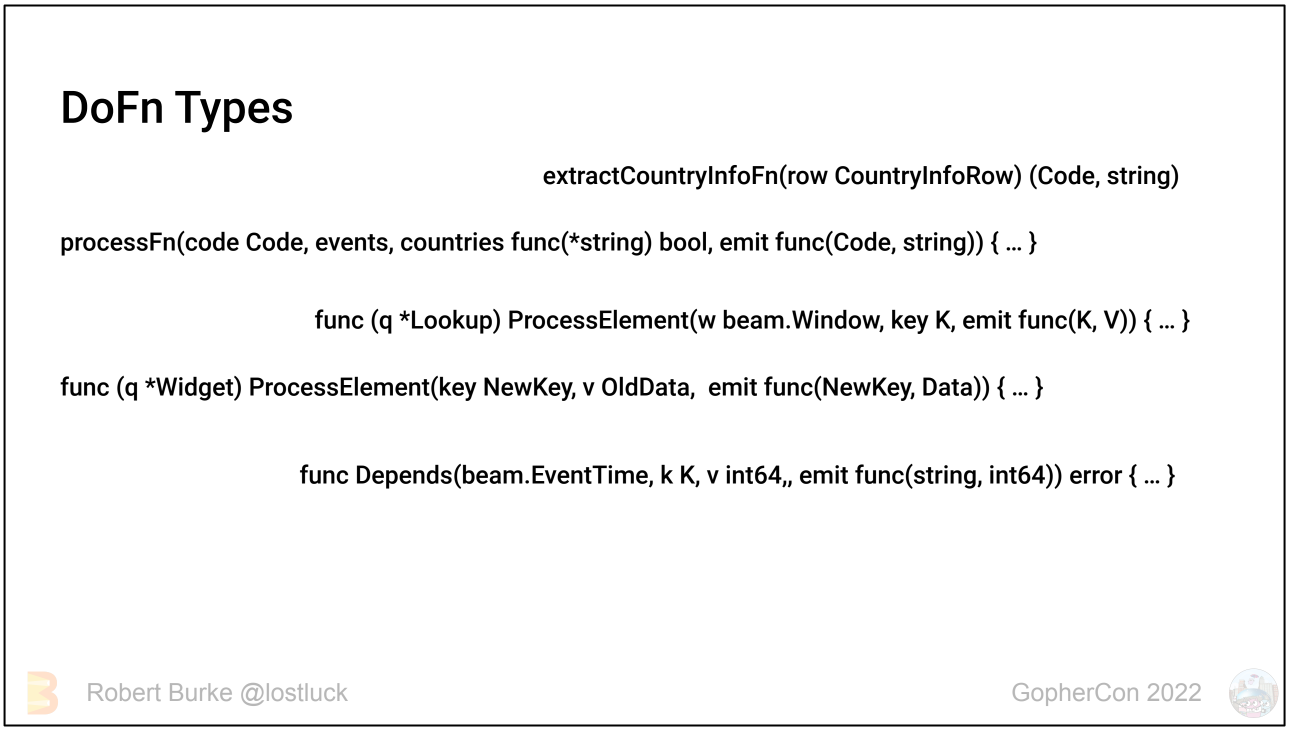
All DoFns have a Main Input type, and we need that to make sure the pipeline works.

But then there’s all the rest.

Most of these are things the framework will generate for you, but in the end, we want you to use your own types for your data. You know it best afterall.

This moves a class of error from Pipeline Execution time to Pipeline Construction time and avoids many costly errors.

Imagine this happens at the end of a mutli day pipeline run. That ‘s a costly error we want to avoid.

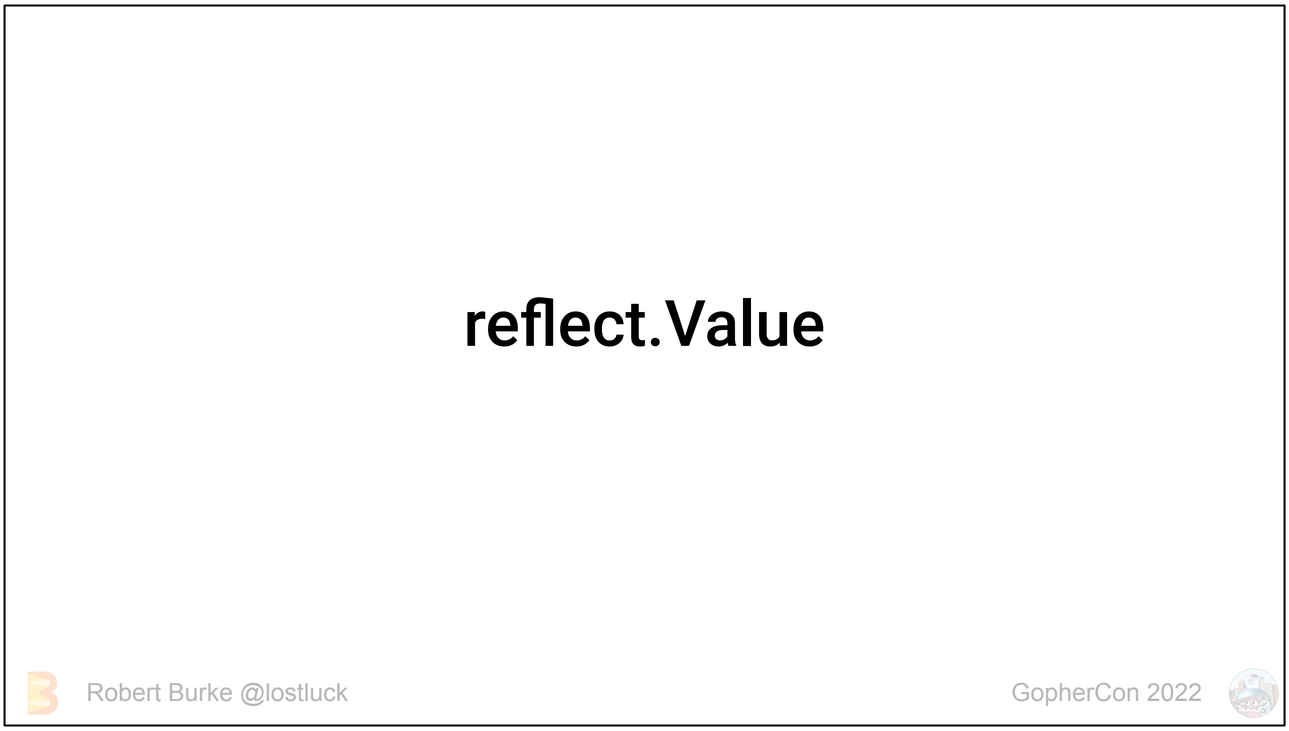


DoFn Types

```
                                   extractCountryInfoFn(row CountryInfoRow) (Code, string)

processFn(code Code, events, countries func(*string) bool, emit func(Code, string)) { … }

                    func (q *Lookup) ProcessElement(w beam.Window, key K, emit func(K, V)) { … }

func (q *Widget) ProcessElement(key NewKey, v OldData,  emit func(NewKey, Data)) { … }

                   func Depends(beam.EventTime, k K, v int64,, emit func(string, int64)) error { … }
```

So how can we call all of these then?

Reflection rides again.



reflect.Value

Robert Burke @lostluck GopherCon 2022

Reflection doesn’t just let your code introspect types.
It lets you look at and write meta code about the values too.

The reflect package has a function, ValueOf that produces a reflect.Value.

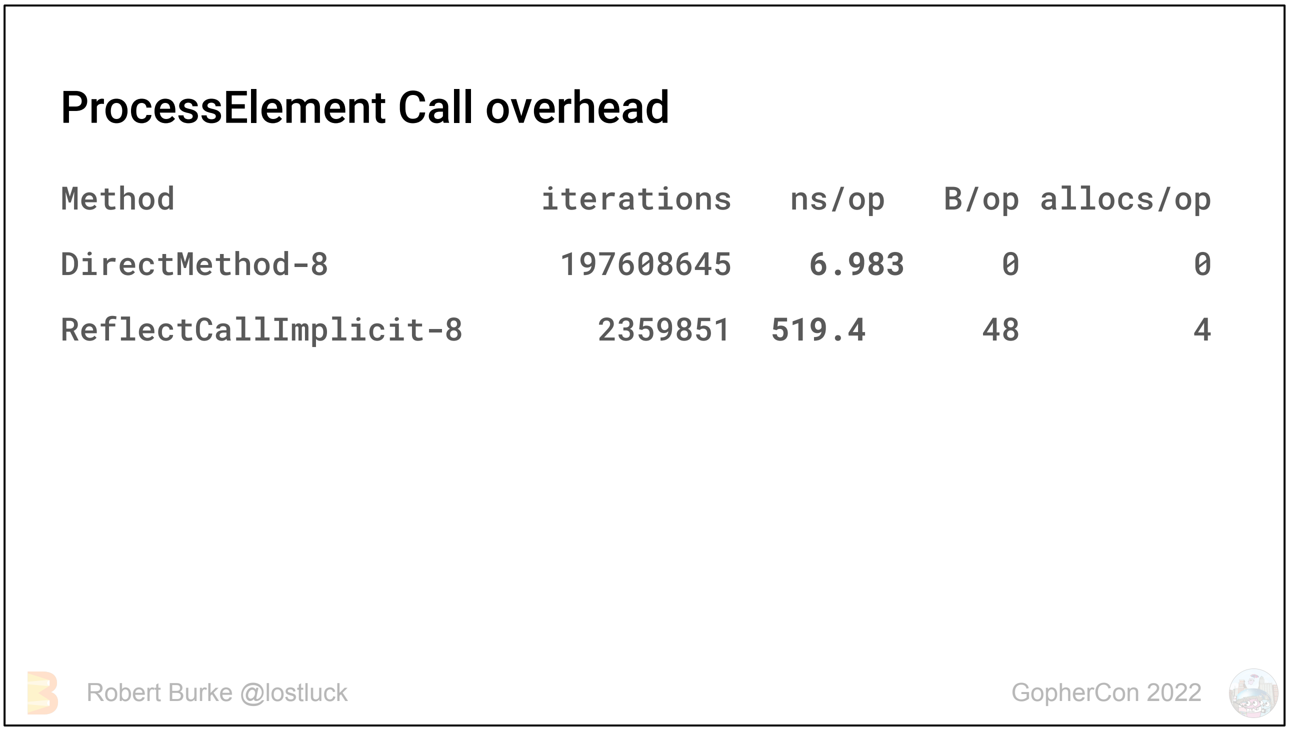
https://pkg.go.dev/reflect#Value

func (v Value) Call(in []Value) []Value

And one of the many methods of reflect.Value, is Call.

If the value isn't a func or a method, it panics, but ultimately Call ends up taking it's parameters and calling it's function value.

OK, so we have a way to execute these functions. It's in the standard library.
How does it perform though?



ProcessElement Call overhead

```
Method                    iterations   ns/op    B/op allocs/op
DirectMethod-8             197608645    6.983     0         0
ReflectCallImplicit-8        2359851  519.4      48         4
```

Robert Burke @lostluck GopherCon 2022

So to get a comparison, we wrote some micro benchmarks for how we're calling a structural DoFn's ProcessElement method. To get a clear overhead measurement, we need a baseline, so here's calling the method directly.

On my chromebook here:
~7ns per call. Not bad. This is the minimum Go's compiler will allow us for now. This is great.

Lets see how reflection goes.

519 ns per call. And a bunch of allocations.

That's 75 times slower.

Now this is a very naive implementation, but we felt we can do better.
But first, we need to make it worse.

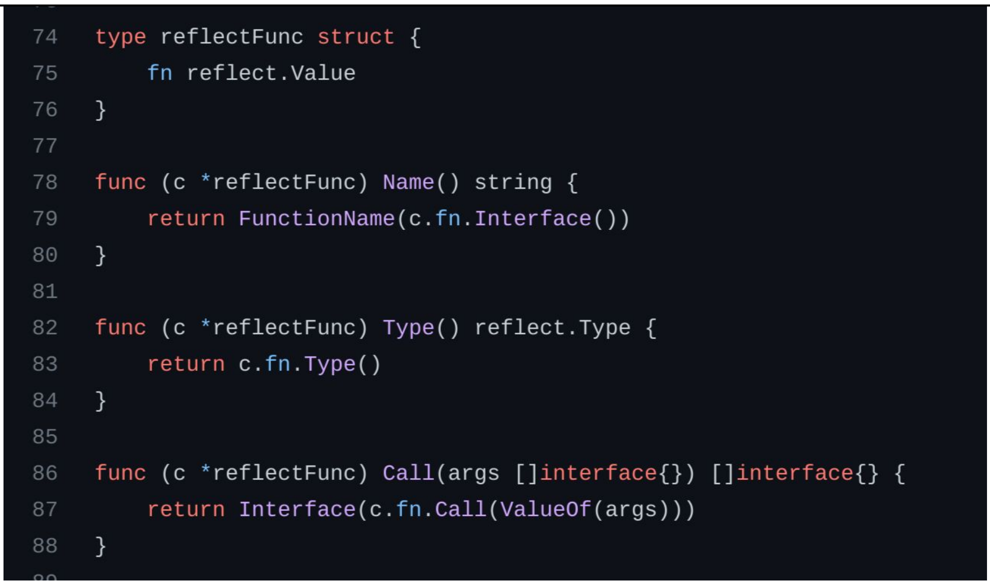
We need add indirection.

Run on my chromebook

goos: linux
goarch: amd64
pkg: github.com/apache/beam/sdks/v2/go/pkg/beam/core/runtime/exec

cpu: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz

https://github.com/apache/beam/blob/master/sdks/go/pkg/beam/core/runtime/exec/fn_test.go#L704



```go
type reflectFunc struct {
    fn reflect.Value
}

func (c *reflectFunc) Name() string {
    return FunctionName(c.fn.Interface())
}

func (c *reflectFunc) Type() reflect.Type {
    return c.fn.Type()
}

func (c *reflectFunc) Call(args []interface{}) []interface{} {
    return Interface(c.fn.Call(ValueOf(args)))
}
```

Robert Burke @lostluck GopherCon 2022

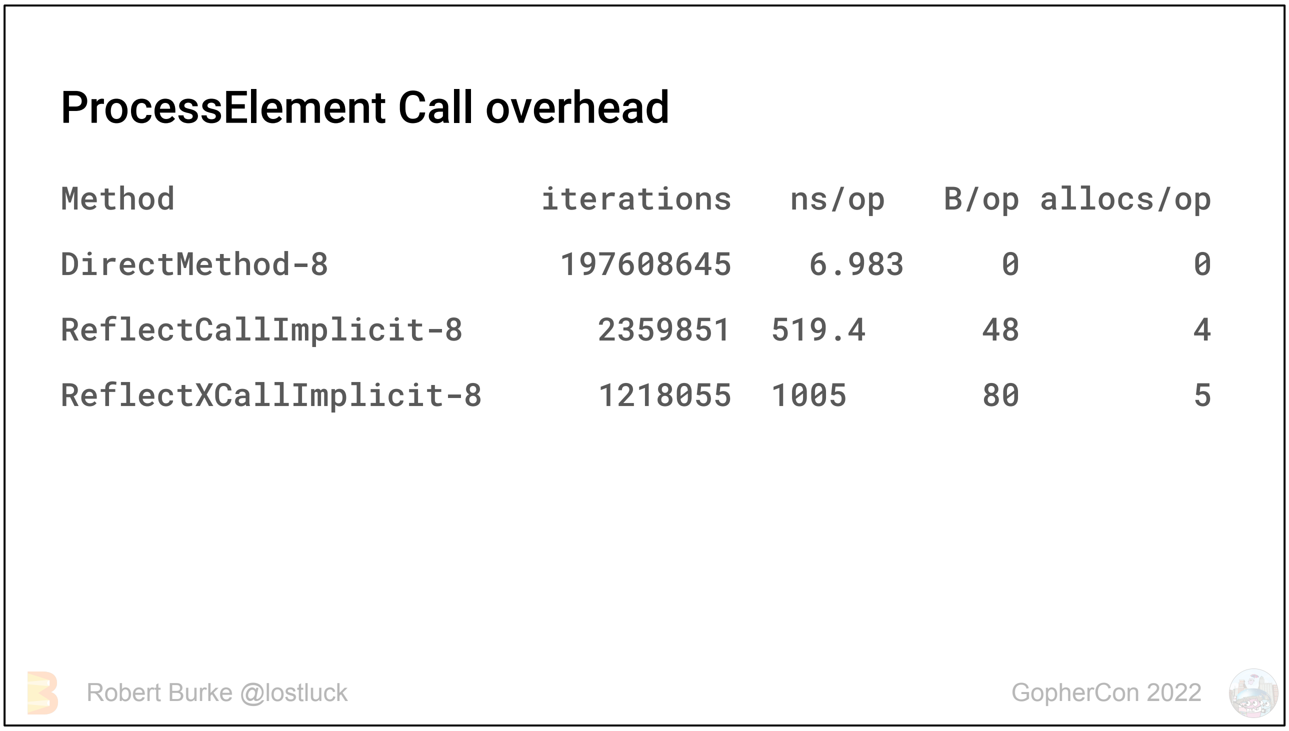
In this case, we indirect through interfaces{}, and specifically, slices of interfaces.

You'll note that our Call method is very similar to the reflect.Value's call method. But instead slices of reflect.Values

This reflectFunc type adapts a reflect.Value to use slices of interfaces for Call instead.

The "ValueOf" function converts a slice of interface{} to a slice of reflect.Values
And the Interfaces function does the reverse , converting a slice of reflect.Values to a slice of interface{}.

So how much worse is this?



ProcessElement Call overhead

```
Method                   iterations   ns/op    B/op allocs/op
DirectMethod-8            197608645    6.983      0         0
ReflectCallImplicit-8       2359851  519.4       48         4
ReflectXCallImplicit-8      1218055  1005        80         5
```

Just under twice as bad as vanilla Call, but 143x times worse than our baseline.

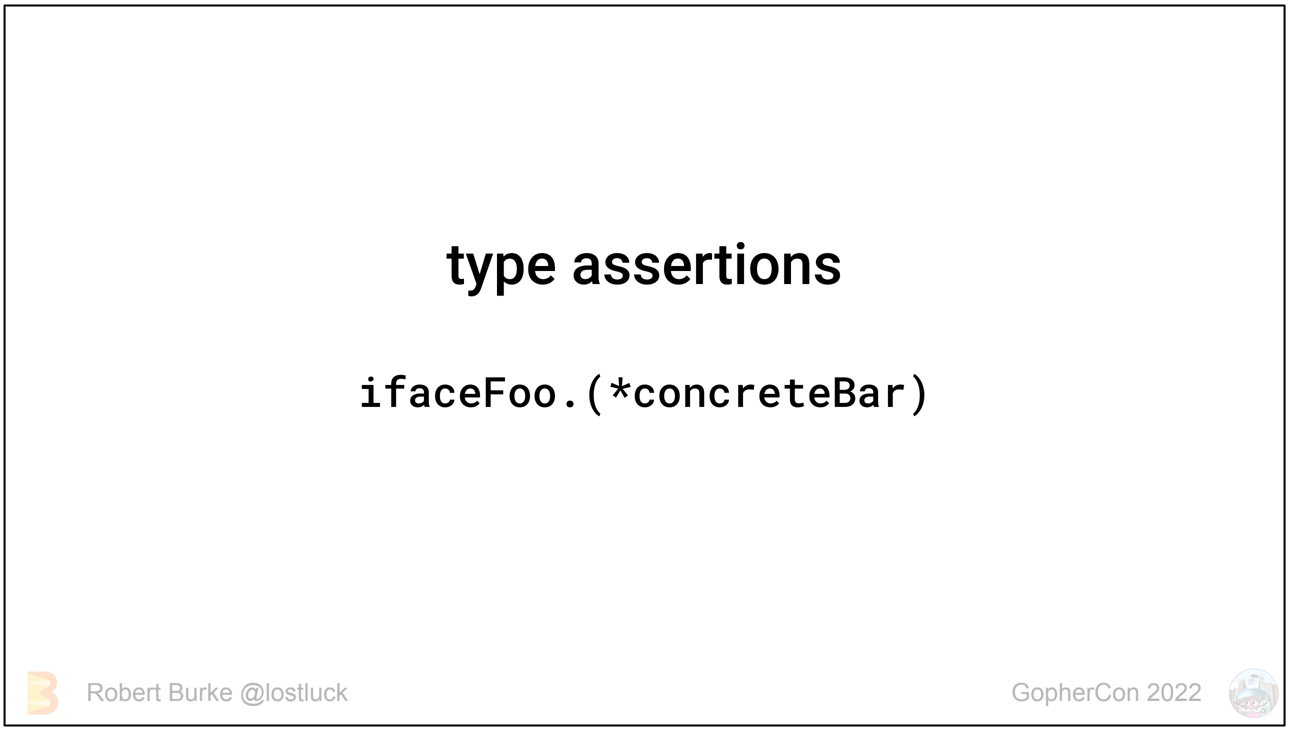
So why did we do this? Why go through interfaces?

Because interfaces lets us use type assertions.

Run on my chromebook

goos: linux
goarch: amd64
pkg: github.com/apache/beam/sdks/v2/go/pkg/beam/core/runtime/exec
cpu: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz

https://github.com/apache/beam/blob/master/sdks/go/pkg/beam/core/runtime/exec/fn_test.go#L704



type assertions

```
ifaceFoo.(*concreteBar)
```

Robert Burke @lostluck

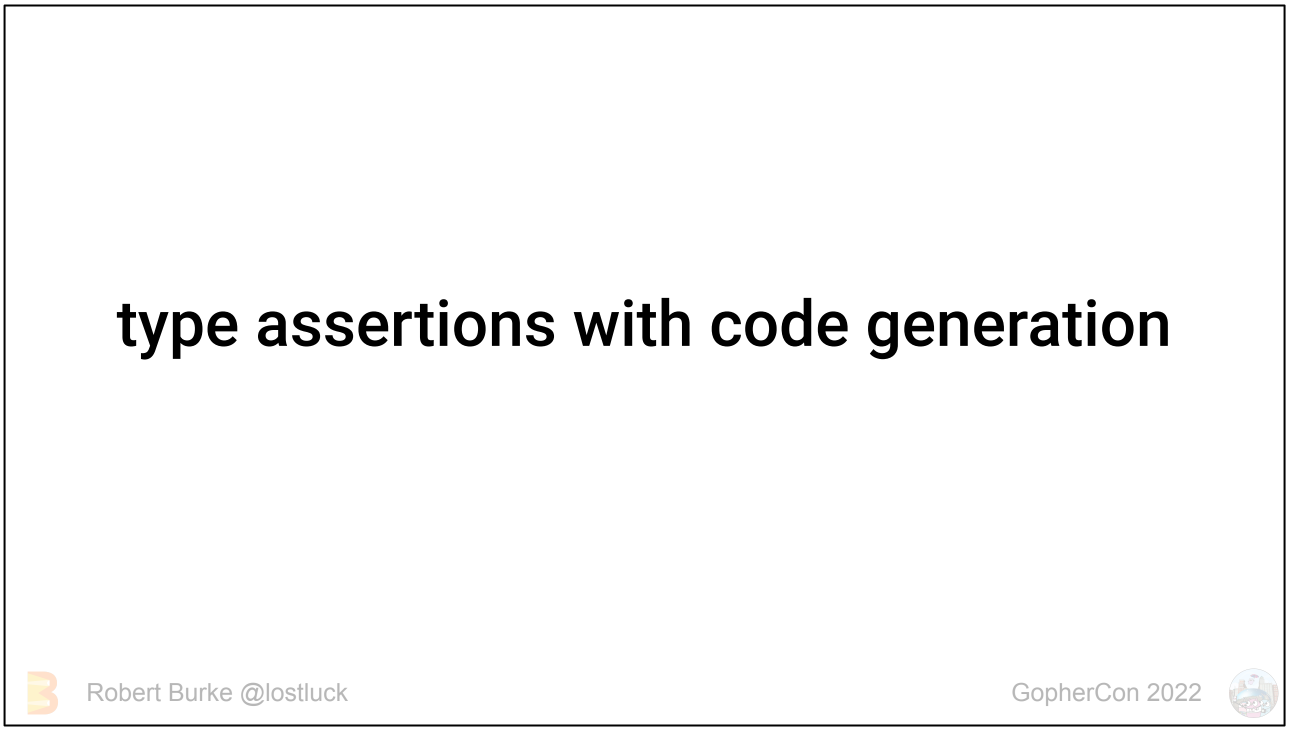
GopherCon 2022

Type assertions are very fast and well optimized by the compiler.

But they require writing the specific types at coding time so the compiler can assist.

You could write these by hand, but we’re programmers, we automate it instead.

So in the earlier days of the Beam Go SDK, we used code generation to automate making these.

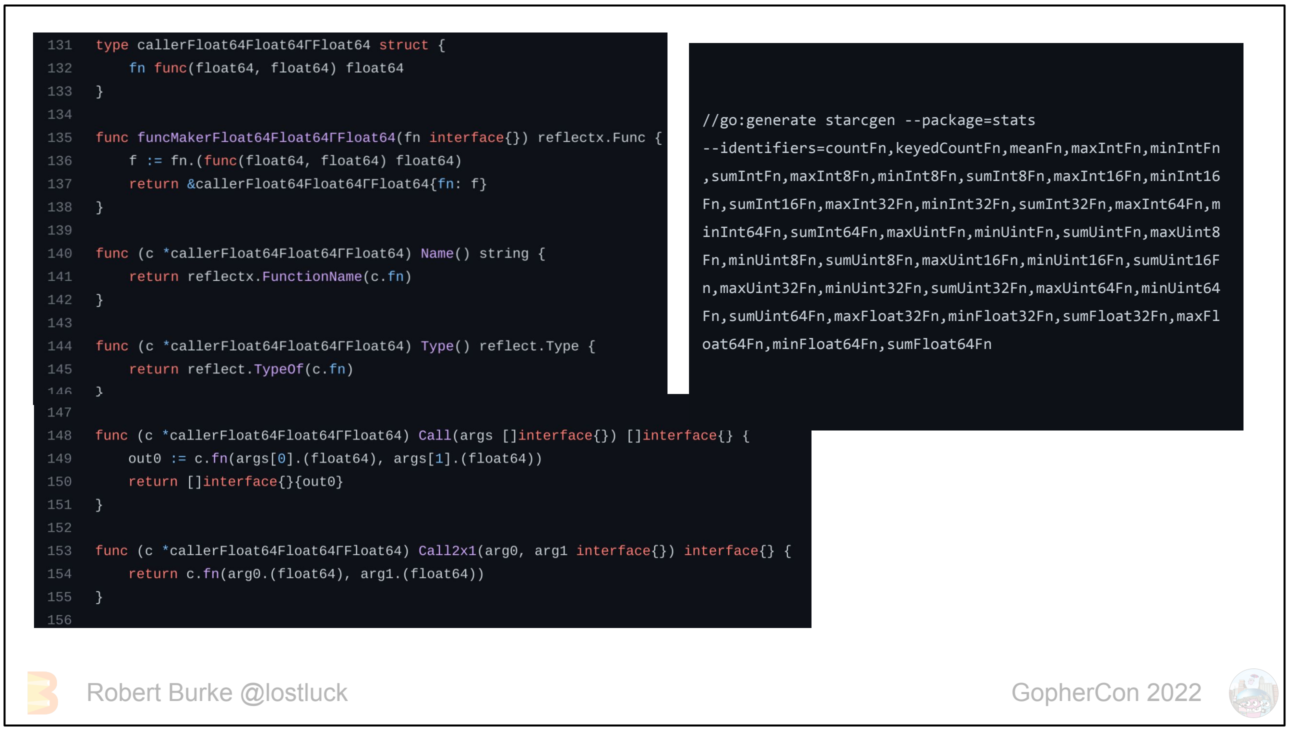


type assertions with code generation

Robert Burke @lostluck

GopherCon 2022

With a bit of static analysis, I wrote a tool that could look at your DoFns and pick out the types, and then generate something like our reflective caller, but doesn't.



```go
type callerFloat64Float64ΓFloat64 struct {
    fn func(float64, float64) float64
}

func funcMakerFloat64Float64ΓFloat64(fn interface{}) reflectx.Func {
    f := fn.(func(float64, float64) float64)
    return &callerFloat64Float64ΓFloat64{fn: f}
}

func (c *callerFloat64Float64ΓFloat64) Name() string {
    return reflectx.FunctionName(c.fn)
}

func (c *callerFloat64Float64ΓFloat64) Type() reflect.Type {
    return reflect.TypeOf(c.fn)
}

func (c *callerFloat64Float64ΓFloat64) Call(args []interface{}) []interface{} {
    out0 := c.fn(args[0].(float64), args[1].(float64))
    return []interface{}{out0}
}

func (c *callerFloat64Float64ΓFloat64) Call2x1(arg0, arg1 interface{}) interface{} {
    return c.fn(arg0.(float64), arg1.(float64))
}
```

```
//go:generate starcgen --package=stats
--identifiers=countFn,keyedCountFn,meanFn,maxIntFn,minIntFn
,sumIntFn,maxInt8Fn,minInt8Fn,sumInt8Fn,maxInt16Fn,minInt16
Fn,sumInt16Fn,maxInt32Fn,minInt32Fn,sumInt32Fn,maxInt64Fn,m
inInt64Fn,sumInt64Fn,maxUintFn,minUintFn,sumUintFn,maxUint8
Fn,minUint8Fn,sumUint8Fn,maxUint16Fn,minUint16Fn,sumUint16F
n,maxUint32Fn,minUint32Fn,sumUint32Fn,maxUint64Fn,minUint64
Fn,sumUint64Fn,maxFloat32Fn,minFloat32Fn,sumFloat32Fn,maxFl
oat64Fn,minFloat64Fn,sumFloat64Fn
```

Robert Burke @lostluck GopherCon 2022

The code looked like this.

And if users wanted to, they could call the generator on their own DoFns with go-generate.

This example is from the stats package, so it's got a lot of very similar but not identical functions.
This is a caller for functions that take in two float64s, and return a single float64.
Also included is a factory function, that takes in an interface, and type asserts to the expected function type, returning a Func interface.

It has a single field, the function it's wrapping. Has the same Name, Type and Call methods so it implements the Func interface. It also has a Call2x1 method, so it can avoid slice allocation because we know the Arity of the function.

Either call method simply type asserts that the values passed in are to be float64s, and then call the function normally.

I'm going to move on from the Code Generator though. It's buggy, and hard to use, and users need to remember to use it.

Because in go 1.18, I'm not sure if you know this. Something big happened.

Generics came to Go.

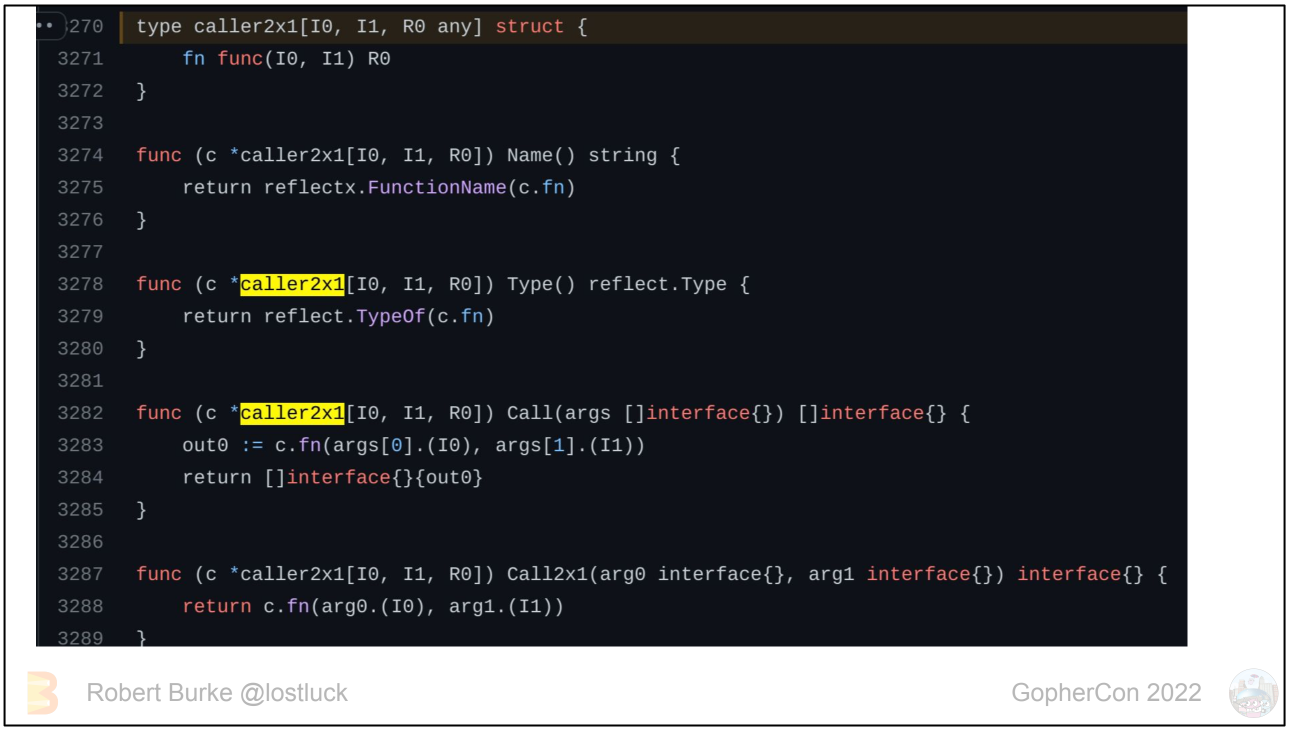
https://github.com/apache/beam/blob/master/sdks/go/pkg/beam/transforms/stats/stats.shims.go#L131

type assertions with generics

And Generics are a big deal for something like this.

The code we had there is very boilerplate heavy.
It doesn't actually do anything with the types other than use them for type assertions.
The values are merely passed around.

And these days, the generic version is the easiest way to show the substitutions at work.



```go
type caller2x1[I0, I1, R0 any] struct {
    fn func(I0, I1) R0
}

func (c *caller2x1[I0, I1, R0]) Name() string {
    return reflectx.FunctionName(c.fn)
}

func (c *caller2x1[I0, I1, R0]) Type() reflect.Type {
    return reflect.TypeOf(c.fn)
}

func (c *caller2x1[I0, I1, R0]) Call(args []interface{}) []interface{} {
    out0 := c.fn(args[0].(I0), args[1].(I1))
    return []interface{}{out0}
}

func (c *caller2x1[I0, I1, R0]) Call2x1(arg0 interface{}, arg1 interface{}) interface{} {
    return c.fn(arg0.(I0), arg1.(I1))
}
```

Robert Burke @lostluck GopherCon 2022

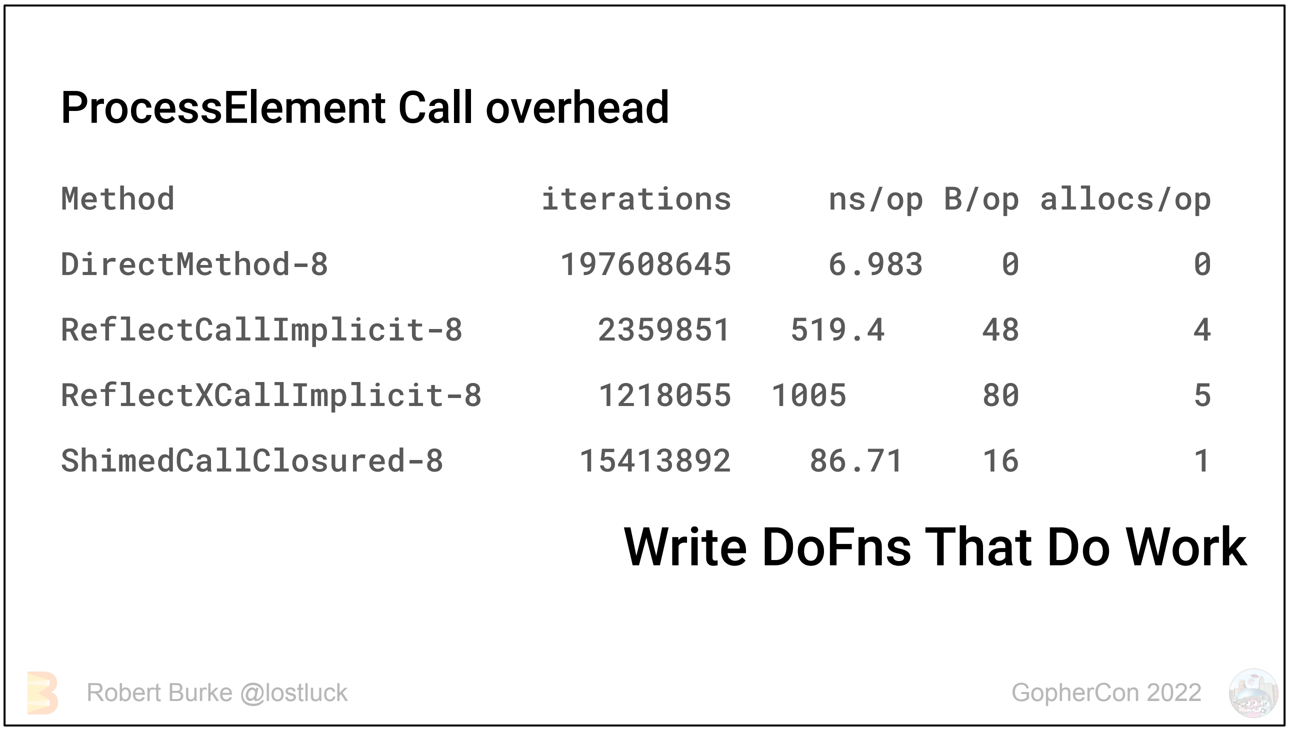
This is the same scaffolding as before.

This is the generic caller for functions that take in 2 parameters, and return a single parameter.

But if you look at the call method, we type assert to our generic parameters, and call the function as before. Easy peasy.

So how does this perform?

https://github.com/apache/beam/blob/master/sdks/go/pkg/beam/register/register.go#L3270



ProcessElement Call overhead

```
Method                   iterations     ns/op B/op allocs/op
DirectMethod-8            197608645     6.983    0          0
ReflectCallImplicit-8       2359851   519.4     48          4
ReflectXCallImplicit-8      1218055  1005       80          5
ShimedCallClosured-8       15413892    86.71    16          1
```

Write DoFns That Do Work

Robert Burke @lostluck GopherCon 2022

87ns. This is a dramatic improvement.

While it’s still over 12x slower than native Go code, this is “overhead” for each call to the function. The lesson here is to amortize it. Write DoFns that do work, and you won’t notice those 80 extra nano seconds.

OK. But how do we make use of this generic approach then? How does the system learn of these things?

That much is pretty easy.

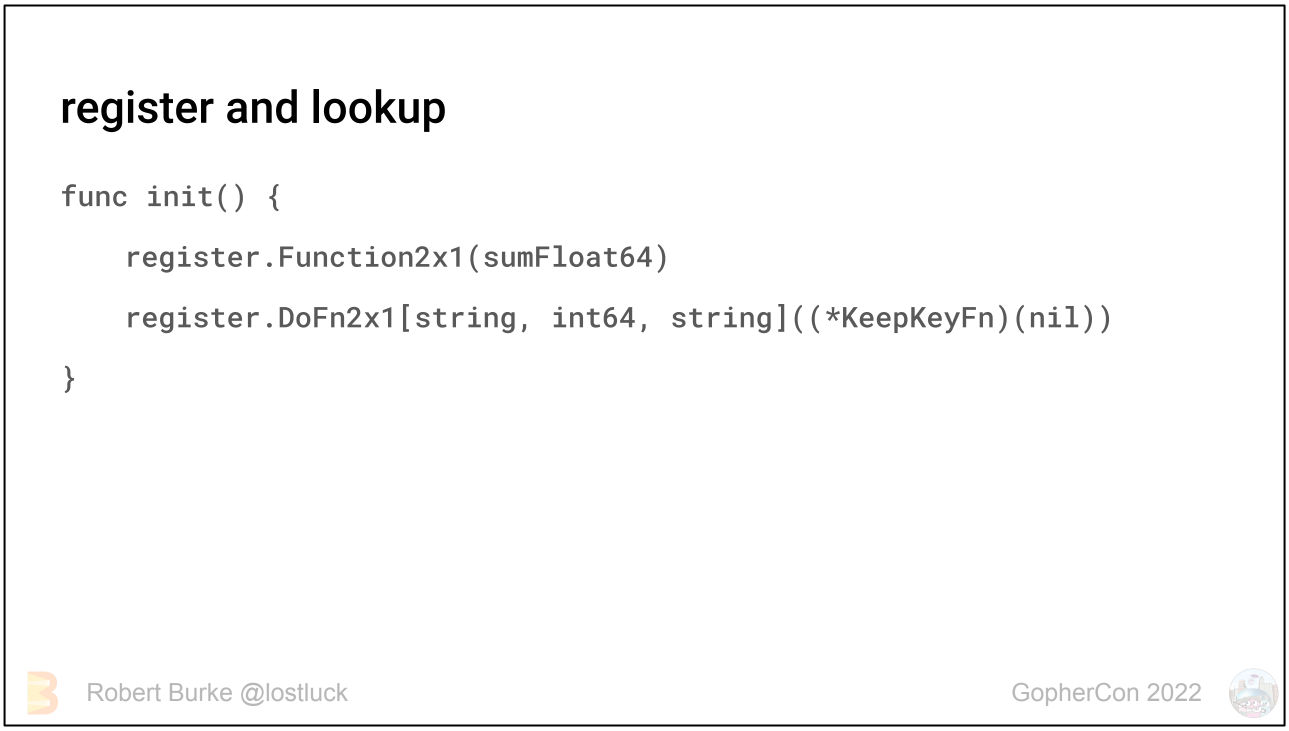
Run on my chromebook

goos: linux
goarch: amd64
pkg: github.com/apache/beam/sdks/v2/go/pkg/beam/core/runtime/exec
cpu: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz

https://github.com/apache/beam/blob/master/sdks/go/pkg/beam/core/runtime/exec/fn_test.go#L704

register and lookup

We register them and look them up!



register and lookup

```
func init() {
    register.Function2x1(sumFloat64)
    register.DoFn2x1[string, int64, string]((*KeepKeyFn)(nil))
}
```

We register them and look them up!

We have a package “register” which has all the generic registration functions for Beam.

Functional DoFns are simple enough that the compiler can infer the types appropriately.
StructuralDoFns with the additional Methods to check need to have manually specified types. That’s the cost of additional configuration I guess.

It must be done before the beam.Init call, I recommend in an init block, but then it’s available to the system and the workers.

To serialize, we have the fully qualified names for the types and functions through the reflect package, which allows us to key off of them for looking up our generated factories. Those are registered by those helpers too.

You be asking, does this make a difference? I’ve mentioned it reduces overhead, I’ve showed benchmark numbers, but despite the way this slide deck may appear, I’m a visual person.

https://pkg.go.dev/github.com/apache/beam/sdks/v2/go/pkg/beam/register

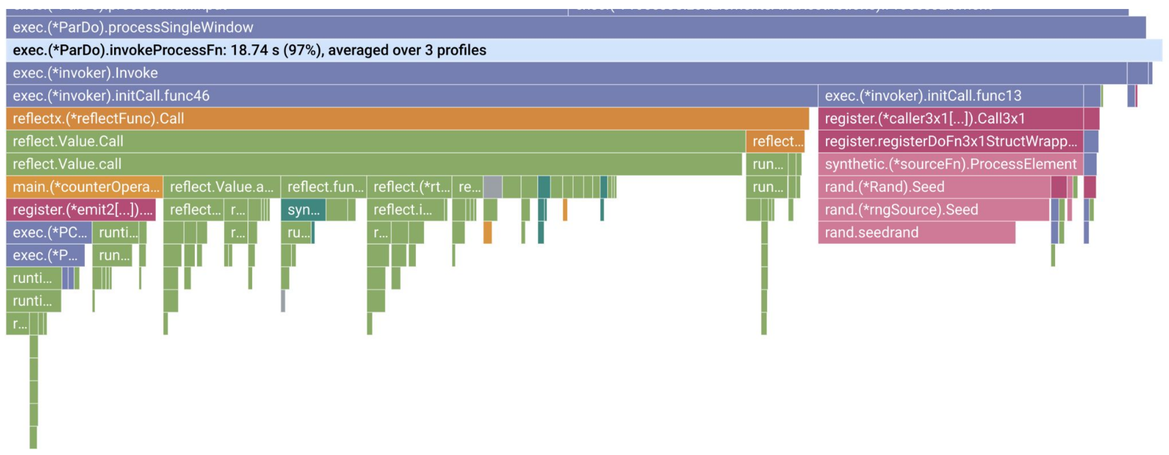
Robert Burke @lostluck

GopherCon 2022

So I ran a load test pipeline on Google Cloud Dataflow, with and without the caller adapters.

And since I used the upcoming 2.42.0 version of the Beam Go SDK, I was able to collect some profiles with Cloud Profiler too.

```
func init() {
    beam.RegisterType(reflect.TypeOf(*counterOperationFn)(nil))
}
```



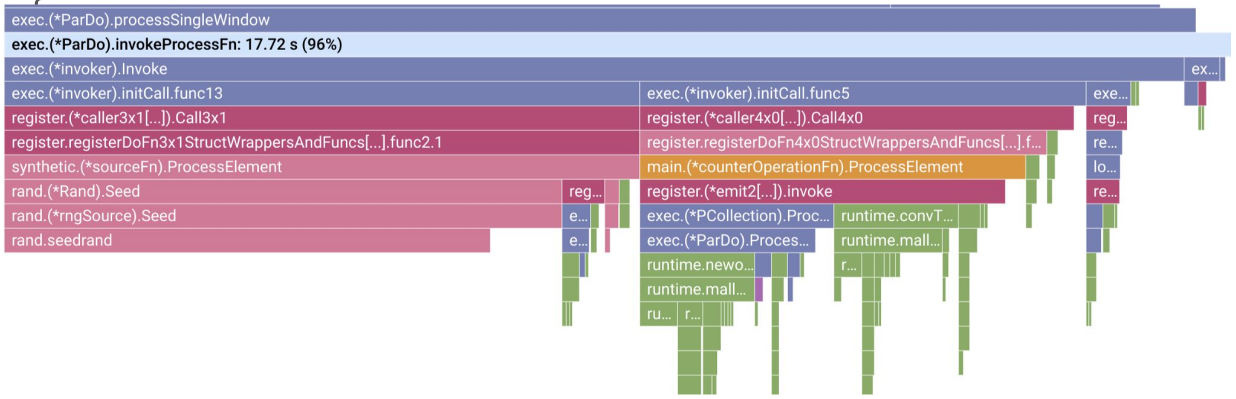
The load test pipeline I used, sets a random seed and passes random byte slides elements through a line of DoFns, in this case "counterOperationFn".

To use the old reflect way, you need only use the soon to be deprecated beam.RegisterType method or beam.RegisterFunction methods.

Cloud Profiler colour codes packages pretty nicely, and in this case that little sliver of gold on the bottom left side of the graph. All that green is reflection.

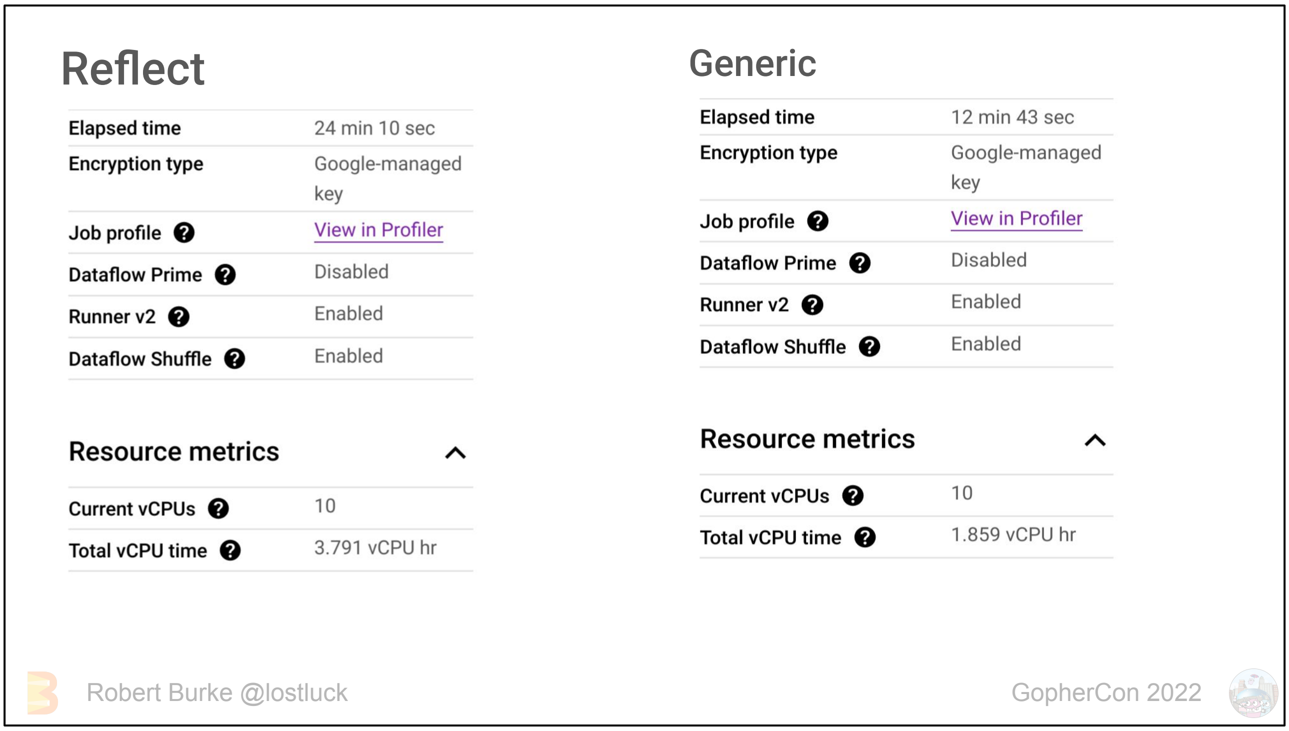
Now compare when we register things properly:

```
func init() {
    register.DoFn4x0[context.Context, []byte, []byte,
            func([]byte, []byte)]((*counterOperationFn)(nil))
}
```



Now look a how much space the random generation takes up, and all that green? Now the conversion from byte slices to interfaces. Spectacular.

And since I ran them on Dataflow. We get some resource usage too.



Reflect

Elapsed time	24 min 10 sec
Encryption type	Google-managed key
Job profile	View in Profiler
Dataflow Prime	Disabled
Runner v2	Enabled
Dataflow Shuffle	Enabled

Resource metrics

Current vCPUs	10
Total vCPU time	3.791 vCPU hr

Generic

Elapsed time	12 min 43 sec
Encryption type	Google-managed key
Job profile	View in Profiler
Dataflow Prime	Disabled
Runner v2	Enabled
Dataflow Shuffle	Enabled

Resource metrics

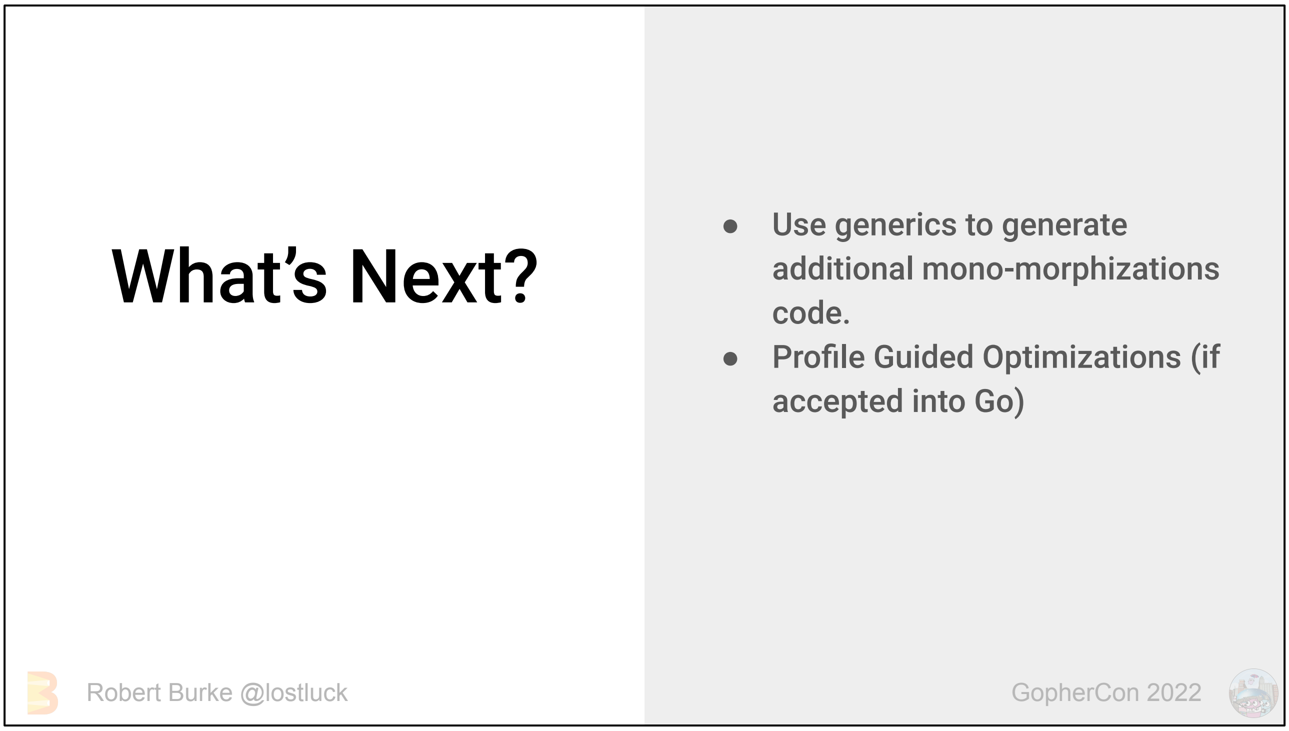
Current vCPUs	10
Total vCPU time	1.859 vCPU hr

The load test pins them to 5 workers each

At the tops you can see the time: It takes about half the time.

And it costs about half as much CPU. That math checks out.

If you’d like a Demo of Dataflow, come find me tomorrow at the Google booth.



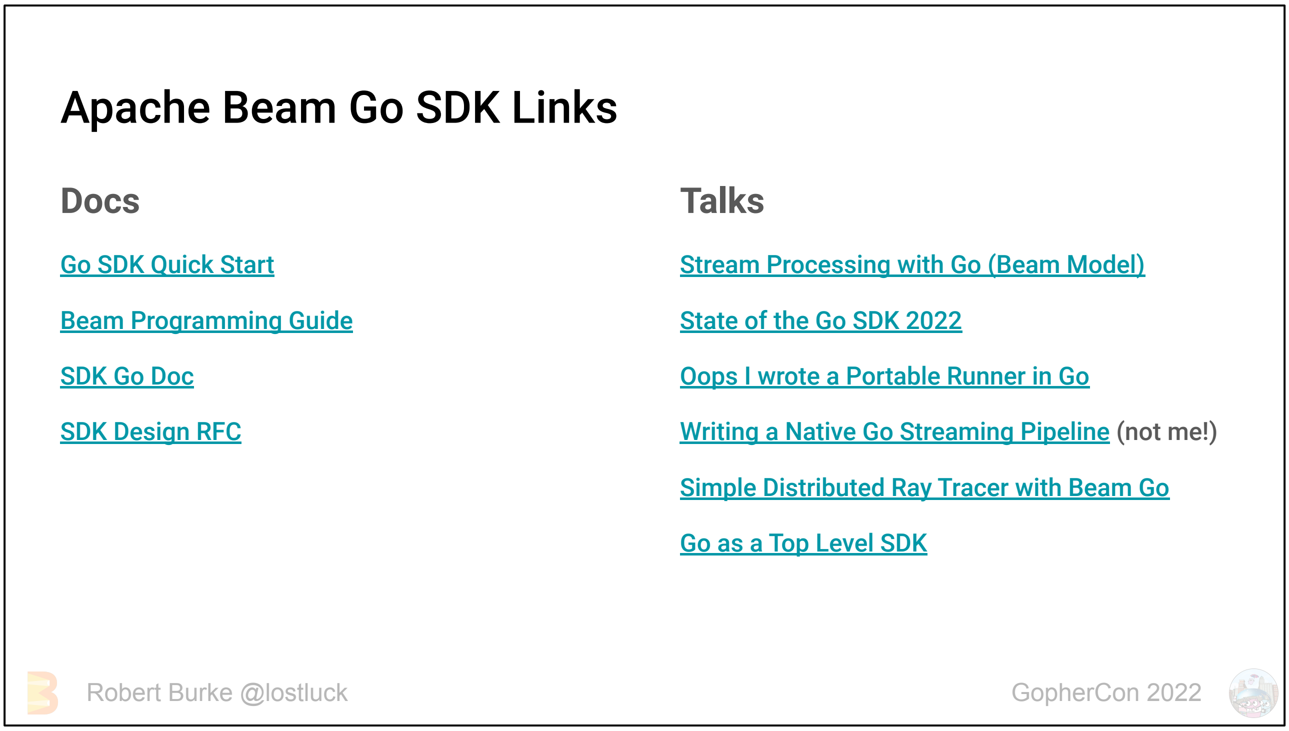
What's Next?

- Use generics to generate additional mono-morphizations code.
- Profile Guided Optimizations (if accepted into Go)

So what's next? Can we go faster?

I think so. Having the generic registrations gives us so much information about DoFns, that we can specialize our execution further. This combined with future compiler improvements like Profile Guided Optimization, and we should see further reduced overhead for sure.

PGO https://github.com/golang/go/issues/55022



Apache Beam Go SDK Links

Docs

Go SDK Quick Start

Beam Programming Guide

SDK Go Doc

SDK Design RFC

Talks

Stream Processing with Go (Beam Model)

State of the Go SDK 2022

Oops I wrote a Portable Runner in Go

Writing a Native Go Streaming Pipeline (not me!)

Simple Distributed Ray Tracer with Beam Go

Go as a Top Level SDK

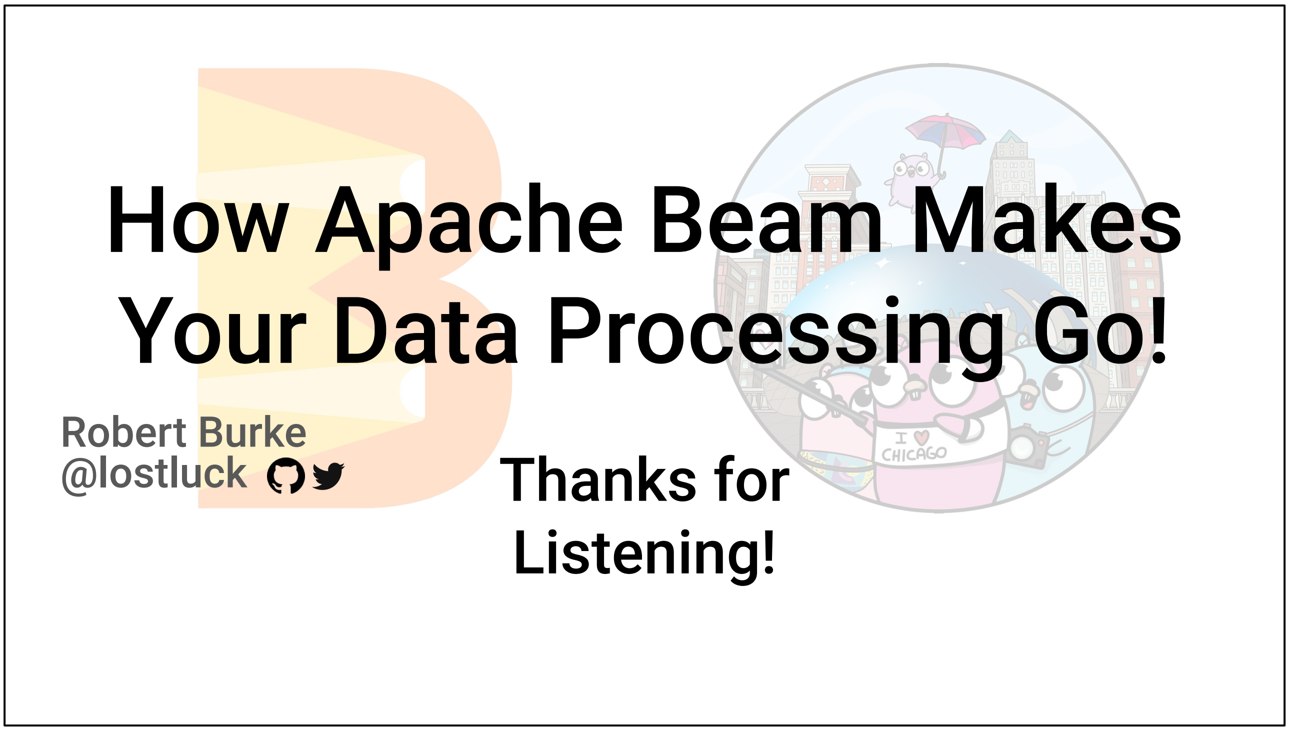
Robert Burke @lostluck

GopherCon 2022

If you'd like to learn more about Beam and the Go SDK,

I'll link this slide in my channel in the Discord in a moment, and tweet it from my twitter too. It's got links to many previous talks of mine on the SDK, and to shared beam Ressources.

I hope you find them useful!



How Apache Beam Makes Your Data Processing Go!

Robert Burke
@lostluck

Thanks for Listening!

I have been Robert Burke, and this has been How Apache Beam Makes Your Data Processing Go!
Thanks for listening GopherCon!
Bye!

https://www.gophercon.com/agenda/session/944207