Beam SDKs Don’t Have to Look the Same

A Quick Look at an alternative Go SDK design

Robert Burke
lostluck@apache.org

BEAM SUMMIT

September 4-5, 2024

Sunnyvale, CA. USA

We know what a Beam SDK looks like, right?

BEAM SUMMIT


@OnTimer(END_OF_BUFFERING_ID)
PCollection<Integer> wordLengths = words.apply(
MapElements.into(TypeDescriptors.integers())
.via((String word) -> word.length()));
@StateId(NUM_ELEMENTS_IN_
long[], Long> storedBat
class FooFn extends DoFn<S
@ProcessElement
@Key K key,
[Final Output PCollection] = [Initial Input PCollection].apply([First Transform])
.apply([Second Transform])
.apply([Third Transform])
BEAM SDK
Java

BEAM SDK

```
@on_timer(WATERMARK_TIMER)
```

```
word_lengths = words | beam.FlatMap(lambda word: [len(word)])
```

```
def start_bundle(self):
```

```
window=beam.DoFn.WindowParam,
key=beam.DoFn.KeyParam,
```

```
class FooFn(beam.DoFn):
    def process(
```

```
(self, element, restriction):
```

```
create_watermark_estimator(self, state):
```

```
[Final Output PCollection] = ([Initial Input PCollection] | [First Transform]
                              | [Second Transform]
                              | [Third Transform])
```

```
func (fn *FooFn) OnTimer(sp state.Provider,
```

```
func wordLengths(word string) int { return len(word) }
func init()                       { register.Function1x1(wordLengths) }

func applyWordLenAnon(s beam.Scope, words beam.PCollection) beam.PCollection {
     return beam.ParDo(s, wordLengths, words)
}
```

BEAM SDK

```
func (fn *splittableDoFn) SplitRestriction(filename string, rest offsetrange.Restriction) (splits []offsetrange.Restriction)
```

```
     [Second PCollection] := beam.ParDo(scope, [First Transform], [Initial Input PCollection])
     [Third PCollection] := beam.ParDo(scope, [Second Transform], [Second PCollection])
     [Final Output PCollection] := beam.ParDo(scope, [Third Transform], [Third PCollection])
```

One of these things are not like the others

BEAM SUMMIT

The Differences with Go

vs Java and Python

- No Overloading
  - Prevents PCollection.apply like syntax
- No Inheritance
  - DoFns and their methods are inferred reflectively*
- No built in Serialization or Pickling
  - Prevents closures and reliable anonymous functions
- No Annotations
  - Prevents targeting methods or fields for specific uses.
- No Generics
  - SDK must typecheck itself.
- No user defined Iterators

BEAM SUMMIT

The Differences with Go

vs Java and Python

- No Overloading
  - Prevents PCollection.apply like syntax
- No Inheritance
  - DoFns and their methods are inferred reflectively*
- No built in Serialization or Pickling
  - Prevents closures and reliable anonymous functions
- No Annotations
- ~~No Generics~~
  - SDK must typecheck itself?
- ~~No user defined Iterators~~

BEAM SUMMIT

Why these aspects matter

- Affects how discoverable features are to users
  - IDE Support, Documentation
- Affects how we can add features to the SDK
  - How much code is required to correctly implement a feature
- Affects performance at construction time
  - Is the framework aware of this features use?
  - Can the SDK put it into the portable pipeline graph?
  - Can the compiler prevent mistakes entirely?
- Affects performance at execution time
  - Can the SDK make better choices for efficient execution?

BEAM SUMMIT

The Challenge:
Does a Beam SDK *have* to look like that?

BEAM SUMMIT

An alternative Go SDK

Designed for Today's Go

- Make use of Generics
- Not constrained to look like existing SDKs
- Not constrained by compatibility.
- Can it be easier for users?
- Can it be easier for maintainers?

BEAM SUMMIT

What can we do differently, with Today's Go?

- How small can we make the User API surface?
- Can we have compile time type safety at pipeline construction?
- Can we reduce the burden of graph construction?
- Can we avoid Registering DoFns?
- Can we enable anonymous funcs or closures?
- <many more questions>

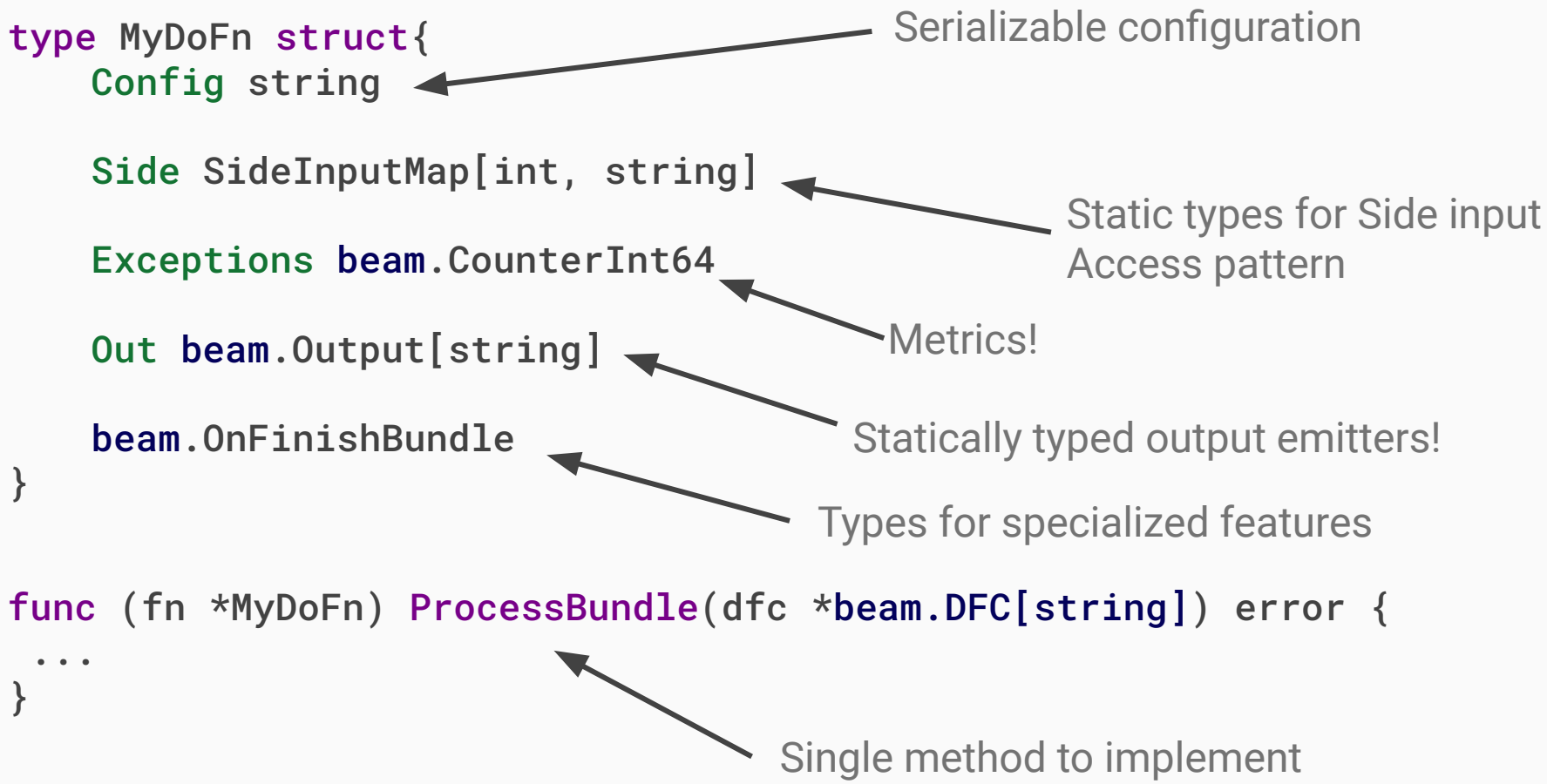
BEAM SUMMIT

Reading Go - quick notes

- Types are specified after names,
  - “var name string”
- Function return types are after the parameters
  - func calculateWeight(thing MyThing) float64 { … }
- Generic Type Parameters are declared in square brackets
  - type MyCollection[T any] struct{ … }

BEAM SUMMIT

```
type MyDoFn struct{
    Config string

    Side SideInputMap[int, string]

    Exceptions beam.CounterInt64

    Out beam.Output[string]

    beam.OnFinishBundle
}

func (fn *MyDoFn) ProcessBundle(dfc *beam.DFC[string]) error {
  ...
}
```



```
type MyDoFn struct{ ... }

func (fn *MyDoFn) ProcessBundle(dfc *beam.DFC[string]) error {
    // StartBundle happens on the ProcessBundle call

    // Configure the ProcessElement function on the DFC.
    dfc.Process(func(ec beam.ElmC, elm string) error {
        fn.Processed.Inc(dfc, 1)
        fn.Output.Emit(ec, newElm)
        return nil
    })

    fn.OnBundleFinish.Do(dfc, func() error {
        // Do some FinishBundle in the callback.
        return nil
    })
    // Per Bundle state cleaned up by Garbage collector
    return nil
}
```

Field Based Design

- Allows users to express intent around
  - DoFn configuration
  - Access Metadata
  - Side Metrics
  - Counters
  - State
  - Timers
  - Emitters
- Framework can observe that intent ahead of time

```
type MyDoFn struct{
    Config string

    Side SideInputMap[int, string]

    Exceptions beam.CounterInt64
    Size       beam.DistributionInt64

    LastVal beam.StateValue[string]
    Callback   beam.TimerEventTime

    Out beam.Output[string]
    Partition []beam.Output[string]

    beam.OnFinishBundle
    beam.ObserveWindow
}
```

BEAM SUMMIT

Combiners

- Dedicated “shape” based wrapper functions to build.
  - beam.FullCombine
  - beam.SimpleMerge
  - beam.AddMerge
  - beam.MergeExtract
- To be used by both
  - beam.CombinePerKey(
  - beam.CombineGlobal

```
type MeanFn[E constraints.Integer | constraints.Float] struct{}

type meanAccum[E constraints.Integer | constraints.Float] struct {
        Count int32
        Sum   E
}

func (MeanFn[E]) AddInput(a meanAccum[E], i E) meanAccum[E] {
        a.Count += 1
        a.Sum += i
        return a
}

func (MeanFn[E]) MergeAccumulators(a meanAccum[E], b meanAccum[E])
meanAccum[E] {
        return meanAccum[E]{Count: a.Count + b.Count, Sum: a.Sum + a.Sum}
}

func (MeanFn[E]) ExtractOutput(a meanAccum[E]) float64 {
        return float64(a.Sum) / float64(a.Count)
}

...
beam CombinePerKey(s,  keyedSrc.Output,
        beam FullCombine(MeanFn[int]{}))
...
```

BEAM SUMMIT

Compile Time type safety

- Need to have element types in the processing method type.
- Need to have a way of indicating output types.
- Want to have pipeline safety at compile time

BEAM SUMMIT

Pipeline Construction

- Happens in a function provided to beam.Run
  - A “no pipeline approach”
  - Allows inline, or separated construction
- Also serves as the “init” switch
- Still to come: Operational Options
  - Construction time options
  - Worker side options

```
pr, err := beam.Run(ctx, func(s *beam.Scope) error {
        imp := beam.Impulse(s)
        src := beam.ParDo(s, imp, &SourceFn{
                Count: 10,
        })
        other := beam.ParDo(s, imp, &OtherSourceFn{
                Config: “configuration!”,
        })
        inc := beam.ParDo(s, src.Output, &MyIncDoFn{
                Side:  beam.AsSideIter(other.Output)
        })
        lw := beam.Map(s, inc.Output, func(v int) int { return v - 1 },
beam.Name("Decrement"))
        beam.ParDo(s, lw, &DiscardFn[int]{}, beam.Name("sink"))
        return nil
}, beam.Name("testjob"))
```

BEAM SUMMIT

How do workers work?

- Pipeline Construction is re-done to register DoFns on the worker.
  - Non-deterministic construction could cause problems. Mitigatable by computing non-deterministic values at construction time, and transmitting them via pipeline options.
- Bundle Processing DoFns are strung together in a type safe way via reflection, and the DFC parameter.
  - Details are not included in this talk.
- Very little between DoFns: Outputs nearly directly passed to consumers

BEAM SUMMIT

What about Splittable DoFns?

Still in progress, current attempt:

```
type BoundedSDF[FAC RestrictionFactory[E, R, P], E any, T Tracker[R, P], R
Restriction[P], P, WES any] struct{}


fn.BoundedSDF.Process(dfc,
        func(rest OffsetRange) *ORTracker {...},

        func(ec ElmC, elm int, or OffsetRange, tc TryClaim[int64]) error { ... })
```

BEAM SUMMIT

Future Work

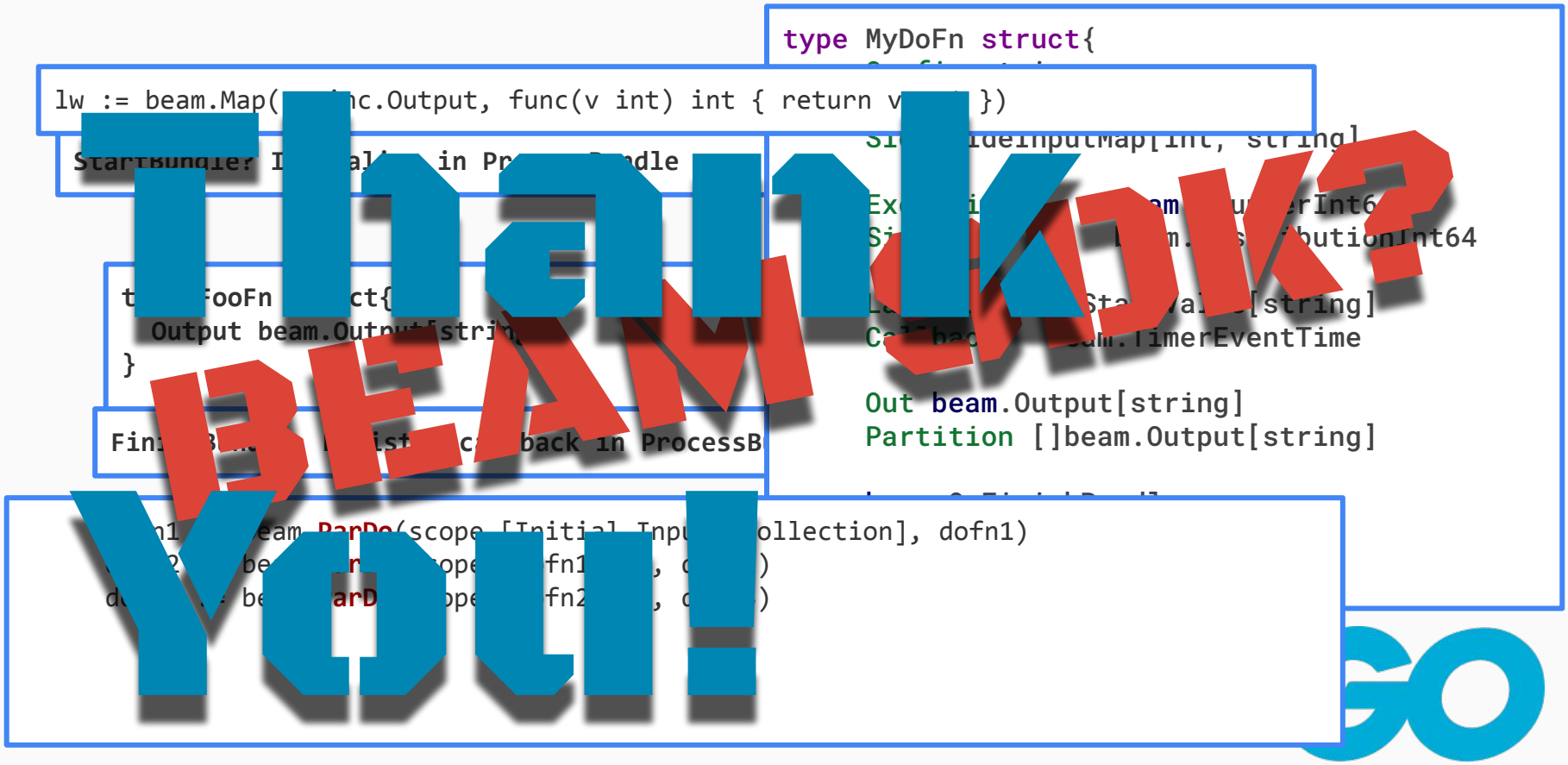
- Submit jobs to Dataflow
  - Can already execute in Docker!
- State and Timer support
- Windowing Strategy and Triggers

BEAM SUMMIT

What can we do differently, with Today's Go?

- How small can we make the User API surface?
  - YES!
- Can we have compile time type safety at pipeline construction?
  - YES!
- Can we reduce the burden of graph construction?
  - A little.
- Can we avoid Registering DoFns?
  - Yes!
- Can we enable anonymous funcs or closures?
  - Yes!

BEAM SUMMIT


lw := beam.Map(
StartBundle?
t FooFn ct{
Output beam.Output[strin
}
Fini
ist
back in ProcessB
type MyDoFn struct{
Si ideInputMap[int, string]
Ex i
Si
La
Ca
Out beam.Output[string]
Partition []beam.Output[string]
eam.ParDo(scope [Initial Inpu ollection], dofn1)
be r ope fn1
be arD ope fn2
Thank
BEAM
You!
GO