# User Defined Coders for the Beam Go SDK

*Robert Burke (rebo@google.com, @lostluck)*
*JIRA: https://issues.apache.org/jira/browse/BEAM-3306*
*Discussion Thread*
*Created: 2018.12.28*

Note(2019.03.24): The end implementation is somewhat different than described in this document. When in doubt, trust the code.

## Goal

- Expand the variety of user defined types that can be a member of a PCollection in the Go SDK.

## Problems

- Go SDK restricts the types users can use as elements to what the type analyser considers a Concrete type.
- Go SDK doesn't offer a way for users to "hook" into beam's coding and decoding system WRT to the types of elements they use.

## Conditions

A good solution implementation must satisfy the following:

- **(A)** Must be able to support interface types in some capacity.
- **(B)** Must be able to support Pointers, Arrays, and Maps in some capacity.
- **(C)** Must avoid beam needing to depend on every possible encoding scheme or interface paring.
    - Users should be able to configure these at construction time.
- **(D)** Must avoid needing to specify a "PCoder" or similar for each DoFn.
- **(E)** Must avoid the requirement for existing workarounds.
- **(F)** May permit users control over allocation semantics (eg. Regarding pointer or reference type re-use to save allocations)
- **(G)** Must permit user DoFns and CombineFns access to the coders at runtime.
    - Eg. createFn is a good example of this.

**Non Goals:**

- The solution is not to dictate encodings schemes, or the decodability of encoded elements outside of a given pipeline binary.
- Ensuring compatibility between different programs, or subsequent versions of the same program remains in the hands of the user.
- The solution is not required to change current StructuralDoFn serialization.

# Proposal

*“If you can marshal, and unmarshal it, you can use it as a PCollection element.”*

Permit users ways to specify coders, but with a hierarchical approach so users can override defaults as necessary, and generalize and re-use where applicable.

## High Level

The hierarchy of coder inference made during pipeline construction, with more specific options handled first:

- **(1)** If the type is a “known to beam” type, use the Beam coding for it
    - This covers primitive numeric types (except complex128), []byte, and string.
    - No change from the current implementation.
- **(2)** If a type has a coder registered in the Coder Registry use that coder.
- **(3)** If a type adheres to a registered interface, use that.
    - **(a)** “I know how to marshal/unmarshal things that satisfy this interface{}”
        - Eg. proto.Message
    - **(b)** Elements that implement paired coding methods from other packages
        - Eg. encoding.BinaryMarshaller, encoding.BinaryUnmarshaller
- **(4)** Use encoding/json
    - No change from the current implementation.

This hierarchy is similar to how encoding/gob handles it’s own permitted encoding methods.

A vetting runner of some kind should be able to hook in and determine which coders given types will be using for analysis. Eg. To avoid JSON encoding or similar for performance reasons.

Currently **(1)** and **(4)** are what the Go SDK currently implements for construction time in the when inferring coders.

In addition to adding a registry for coders, a change to the type analyzer will be necessary to permit additional types (such as Pointers, Arrays, Maps, and Interfaces).

Note: Originally, there was a facility to change the default coder away from JSON, but this was determined to conflict with schemas, and is now explicitly omitted.

## Details

The core of this proposal is to ensure users have flexibility, in that the beam package doesn’t prescribe to closely how elements are codable, just that they are codable and concrete. It also takes advantage of Go interfaces, which satisfies being able to encode types that are interfaces.

Continuing to use the beam coders for primitive types **(1)** avoids compatibility issues with other Beam languages, and having a different coder tree for Beam specific usage of the primitives (eg length prefixing, window encoding, etc) than for user values.

**(2)** covers the case that custom encoder and decoder exists for the type. Values whose type matches the passed in type will use the given coder. This is done after **(1)** to permit overriding more general coders below.

A registration function will exist, and add the coder to the registry.

```
func RegisterElementCoder(t reflect.Type, enc, dec interface{}) {...}
```

**(3)** both cases **(a)** and **(b)** can be satisfied by the following.

```
func RegisterInterfaceCoder(t reflect.Type, enc, dec interface{}) {...}
```

Eg. **(a)** Handling interface types that know how to encode themselves such as protocol buffers, that require details known only to that type's package.

Pragmatically, these are unified under a single method, that checks if a values is an interface type, and handles those specially.

```
func RegisterCoder(t reflect.Type, enc, dec interface{}) {...}
```

A single function simplifies the user surface and the documentation, and avoids the need to panic in the interface version if unspecified, and permits passing around real interface types as PCollection types.

```
var protoMessageType = reflect.TypeOf((*proto.Message)(nil)).Elem()

func init() {
       beam.RegisterCoder(
          protoMessageType,
         func(in T) ([]byte, error) {
               return proto.Marshal(in.(proto.Message))
         },
         func(t reflect.Type, in []byte) (T, error) {
               val := reflect.New(t.Elem()).Interface().(proto.Message)
               if err := proto.Unmarshal(in, val); err != nil {
                       return nil, err
               }
               return val, nil
       })
}
```

Eg. **(b)** Handing pairs of interface types that a given message must implement to code themselves. Eg. encoding.BinaryMarshaller and encoding.BinaryUnmarshaller

```
type BinaryCoder interface {
        encoding.BinaryMarshaller
        encoding.BinaryUnmarshaller
}

var binaryCoderType = reflect.TypeOf((*BinaryCoder)(nil)).Elem()

func init() {
       beam.RegisterCoder(
          binaryCoderType,
         func(in T) ([]byte, error) {
               return in.(encoding.BinaryMarshaller).MarshalBinary()
         },
         func(t reflect.Type, in []byte) (T, error) {
               val := reflect.New(t.Elem()).Interface().(encoding.UnmarshalBinary)
               if err := val.BinaryUnmarshal(in); err != nil {
                       return nil, err
               }
               return val, nil
       })
}
```

This isn't intended for most users. The options for **(2)** and **(3)** likely cover more targeted uses. However, if one uses gob encoding everywhere, and has their own overrides, they should be permitted to do so.

The signatures for RegisterInterfaceCoder and RegisterElementCoder are identical but it's expected to be more explicit, and clarity. Coders registered with RegisterElementCoder will always take priority for types that match, and is a simple map lookup. While RegisterInterfaceCoder needs to determine if a type implements a given interface. A later variation could unify the two as a "RegisterCoder" method, that takes extra care when handling values of Interface kinds.

## Encoder and Decoder Signatures

The encoders are permitted to be functions with the following signatures:

```
func(T) []byte
func(T) ([]byte, error)
func(reflect.Type, T) []byte
func(reflect.Type, T) ([]byte, error)
```

where T is any real Go type,
or a function that returns the following interface:

```
type Encoder interface {
```

```
 Encode(reflect.Type, beam.T) ([]byte, error)
}
type makeEncoder func() Encoder
```

Similarly, decoders may implement the following function signatures:

```
func([]byte) T
func([]byte}) (T, error)
func(reflect.Type, []byte) T
func(reflect.Type, []byte) (T, error)
```

where T is any real Go type,
or a maker function that returns the following interface:

```
type Decoder interface {
  Decode(reflect.Type, []byte) (beam.T, error)
}
type makeDecoder func() Decoder
```

The different functional variants are in keeping with the style of the Go SDK, where arguments that aren't needed should be omit-able.

The interfaces Encoder and Decoder allow for a structural version. A novel use of this allows a Decoder to better control allocation, and maintain a pointer to a re-usable allocation of what is being decoded. The Go SDK uses a unique instance of each decoder per bundle. This would be usable directly by the decoding data manager, rather than being wrapped as the functions would be.

In particular, Beam permits arbitrarily reflectively invoking coders and using the reflectx shims to get the real type.

Eg. A valid encoder, decoder pair for user type `type UserType struct{}` is `func(UserType) []byte` and `func([]byte) UserType`. The trick is at present, there's an extra heap allocation when wrapping a []byte into an interface{}. A user may provide concrete beam.T functions instead, and perform the type assertions themselves, to avoid this micro allocation to the heap.

A benefit of having a set of pre-declared signatures and interfaces allows us to avoid using the reflectx package to invoke the coders. In particular, we avoid an []byte -> interface{} conversion that would be allocated to the heap, per element.

## Does this satisfy solution requirements?

Yes.

- **(A)** Interfaces are explicitly accounted for with option (3) and implicitly with option (2)
    - "General" interfaces may not be supportable without a known concrete implementation. Users will likely always need to use a concrete type to

encode values unless the interface includes a marshal/unmarshal style method pairing, or are “generally known” like generated protocol buffer types.

- **(B)** Pointers, Maps, and Arrays are handled similarly, at least as wrapped values, more directly with changes to the type analysis.
- **(C)** Beam doesn’t gain any additional dependencies. Those live in the user code that registers and wraps coders for beam.
- **(D)** Coders are inferred implicitly as is present. Doesn’t exclude the possibility of overriding this decision in the future, and permit coder overrides.
    - Can be worked around by adding a DoFn that manually encodes values to []byte, which beam will largely keep as is.
- **(E)** Existing work around become unnecessary, but are not excluded if desired. Existing work around should still be able to function.
    - Since beam uses protocol buffers as the FnAPI transport, the Go SDK may continue to natively register a protobuf coder to avoid breaking existing uses of that work around.
- **(F)** Users gain control over allocation semantics. For example, Structural Decoders for pointer elements could allocate an instance of the element as a field, and then return a pointer to that as the element. This is useful for very large structs that are expensive to even allocate.
- **(G)** Since the coders must be registered prior to beam.Init, anything created afterwards should retain access to MakeElement{Encoder,Decoder} and continue to use the accordingly.

## Changes to type analysis

The Go SDK does extensive type analysis of DoFn signatures to ensure We can keep the type analyzer pure of runtime configuration by changing it to mark the hard exclusions. Presently, there’s a hack to permit proto.Message types.

We can simplify it to be a blacklist rather than a whitelist, with the inclusions being the known unmanageable or unserializable types (such as interface{}, or unsafe.Pointer)

Structs with “blacklisted” types as fields are permitted, so long as those fields are unexported.

### Pointers

Once pointers come into the picture, the issue of Aliasing comes up. Aliasing is when two or more pointers to the same concrete value exist, changing something with one pointer is reflected when reading via the other.  In particular, users cannot make the assumption that aliasing will be maintained for a given element instance between two DoFns, as there may have been a encoding/decoding step between them. In general, there’s no way to maintain the aliasing between two elements across a materializing boundary, so values may be encoded multiple times, even though only a single value exists.

## Maps

Idiomatic Go encourages making the zero value useful. For example a nil slice has a defacto length (0) and can be appended to as with allocated slices.

Maps specifically need to be created before use with the make() builtin, otherwise using it will panic. Nil maps aren't useful except to get a length of 0.

This means that maps as a type themselves, or as a field in a struct must always have special handling. When decoding into a registered map type, there's an opportunity to create them.

As such, for this initial pass, the consideration would be to not support maps directly. However, it's possible to use reflection to make instances of maps using reflect.MakeMap, and then insert deserialized keys and values into it with SetMapIndex, so it's open for the future. TODO(lostluck): File a JIRA?

A work around in the meantime would be to wrap the map in a struct as an unexported field. This permits the map to be passed around as a PCollection element, but encourages writing and registering a coder for them to be handled explicitly.

## Channels

Channels are similar to maps, in that the zero value isn't terribly useful, but they're a communication primitive. Beam would have no way to reconnect a channel to it's correct endpoints.

At present, there's no way to extract the buffer size of a channel reflectively, and no way to make a channel without a known buffer size.

It's unclear if there will be a good case for the Go SDK to support coding channels natively.

## Overriding Default Beam Coders

This proposal is opting to not support direct overriding of the default beam coders. This simplifies the Go SDK implementation, but also because it's not strictly necessary in Go.

In Go it's very easy to create a new type: `type myInt int`, which can then be used to register a coder against it for custom alternative serialization of known beam types.

## Interfaces

Serializing and Deserializing specific instances isn't generally possible, but the Go SDK already accounts for this in type serialization with the `beam.RegisterType` function. This contributes the type into a registry, so that the reflect.Type instance is preserved along with full method information. This works around that the reflect package doesn't provide a serialization mechanism to synthetically create interfaces.

Example: adding a coder for the color.Color interface, which abstracts a colour representation. In particular, a user would need to declare which concrete implementation of a color to encode the value as to serialize the type.

This is similar to how the "encoding/gob" package handles encoding interface values.

Users would need to handle them specifically, which is similar to how Protocol Buffers handle serialization.

### Pointers to primitive types

It's unclear if this should be natively supported, so for the time being, no explicit action for this will be taken. Users will be able to encode values as they choose.

# Alternative Proposals

## Schemas

Schemas are an abstraction to provide Beam runners knowledge of data structure, and permit simplified specification of common computations around that structure. In particular, runners would be able to perform manipulations as needed if the runner were schema aware.

In particular, Schemas are something that represent the known structure of an element (or Row) of data. This includes primitive fields, other schemas, repeated lists, maps, and other abstractions.

Beam then provides functions to transform PCollections using a combination of Select, Join, and Grouping operators (which can also include aggregations).

Some example (using the Java API):

```
PCollection users = events.apply(Select.fields("user"));  // Select out only the
user field.

PCollection joinedEvents =
queries.apply(Join.innerJoin(clicks).byFields("user"));  // Join two
PCollections by user.

// For each country, calculate the total purchase cost as well as the top 10
purchases.
// A new schema is created containing fields total_cost and top_purchases, and
rows are created with the aggregation results.
PCollection purchaseStatistics = events.apply(
    Group.byFieldNames("country")
```

```
                .aggregateField("purchaseCost", Sum.ofLongs(), "total_cost"))
                 .aggregateField("purchaseCost", Top.largestLongs(10),
"top_purchases"))
```

Speculating based on the existing Go SDK, those examples would look like the following.

```
events := <some PCollection with event data>

// Select out only the user field. Assuming it's a string field, returns a
PCollection<string>
users := beam.Select(s, events, "user")

// Very simple approach (not recommended). Could use API iteration.
// Join two PCollections by user.
// Returns a PCollection<Schema<events + clicks>>
joinedEvents := beam.JoinInner(s, events, clicks, "user")

// For each country, calculate the total purchase cost as well as the top 10
purchases.
// A new schema is created containing fields total_cost and top_purchases, and
rows are created with the aggregation results.
purchaseStatistics := beam.GroupByField(s, events,
beam.AggregateField("purchaseCost", schemas.Sum(), "total_cost"),
beam.AggregateField("purchaseCost", schemas.Top(10), "top_purchases"),
```

For PCollections of Schema types (or not schema types), a DoFn could be based on the usertype directly, and validate that the user type matches the schema of the incoming PCollection.

Implementation wise, the FullType used during pipeline construction and verification could additionally store Schema types, and at construction time checks that either the identical type or a schema compatible type are being used.

Where a schema -> real type conversion exists, there will be a performance hit as the incoming PCollection values would need to be converted to the real types.
By default we can use reflection to perform this conversion, but it should be possible to generate custom Schema<A> -> TypeB. However, this may not necessarily be possible to statically check check without additionally knowing the construction time graph.

In principle it seems that schema conversions to a type are the intersection of a type's fields with the schema, where the Schema is permitted to have additional fields not included in a type, but the type must be fully satisfied to be correct.

For example, SchemaA[f1:int,f2;int;f3:string] could be reasonably converted to a struct{f1 int, f3 string}, but not to a struct{Foo string, f1 int, f3 string} since the SchemaA can't provide the Foo field. Obvious exclusions occur when there is a mismatch between a fieldname and type.

From this we could infer that we could probably generate for every type something that can do the Schema -> Type conversion, and Type -> Schema extraction, and connect the pieces together for something that's faster. Both can be done using reflection by default.

This informs that there's the additional registration opportunity of beam providing faster methods for extracting and reifying types from schemas, eg. `beam.RegisterTypeSchema(reflect.Type, func(interface{}) beam.Schema, func(beam.Schema) (interface{}, error))` and `beam.RegisterInterfaceSchema(reflect.Type, func(interface{}) beam.Schema, func(beam.Schema) (interface{}, error))` which could be hierarchically invoked as with the main User Defined Coder proposal above, where the context of which values to use specific instances are handled automatically eg.Values "leaving" the system

The above is assuming an approach where users are never directly using the schema types, but real manifestations of their schemas, which are simpler to manipulate. Ideally, users writing DoFns should never need to know about schemas, and simply get on with using PCollections.

## Implementation

At present no implementation exists, and none is required. A beam.Schema type is referenced above but no details are given about it at this time. It's at minimum a "notional" type in the vein of KVs or CoGBK types that PCollections may have during the type checking of a pipeline at construction time, rather than a concrete type. The API could require such a FullType be provided explicitly to know which whether to use the schema coding or not.

## Ambiguity with Coders

Java in particular has an issue where a schema is used when no coder is available, however, the "SerializableCoder" is usable for all Java Objects, so this fails. This is not currently an issue with Go as currently there's no meaningful default. The SDK uses JSON out of convenience, and debugability, but nothing relies on this behavior.

The above proposal removed the option to set a new "default" coder to use instead of JSON in response to discussion on the mailing list thread.

Aside from the issue with Schema's in Java, the main places where a conflict arises is in inferring schema use during a CoGBK, or Combine operation. This could be worked around by having different user level constructs that explicitly support schemas, which would always use the schema variants for passing values out of the FnAPI. Similarly, for nodes where explicitly passing values out to some external schema value, we could provide a `beam.ConvertToSchema(beam.PCollection) beam.PCollection` for users to force a PCollection to it's Schema equivalent form when desired, which would inform what Coder to use at execution time. In this, Go not having generic types is a benefit, rather than a detriment!

## Others?

The author is open to suggestions.

# Background

- Beam requires all element in PCollections to be codable, which the Go SDK calls Concrete. This permits runners to rearrange the DoFn graph as needed to distribute work across machines.
    - Commonly: Elements are encoded before GroupByKey operations and decoded afterwards.
- Currently by default, if an element is of a “known” type (numeric types, []byte, string) the Go SDK encodes it according to the beam coders for the same.
- At time of writing, the Go SDK is still Experimental, which means we can make breaking changes if we want to. If there are breaking changes that need to be made, we should find out now.
- There’s a hack in the type analysis to permit proto.Message values in signatures.

## Other Beam SDKs

See https://beam.apache.org/documentation/programming-guide/#data-encoding-and-type-safety for how Java and Python handle coders.

## Existing Workarounds

Users can currently work around the the beam coders by handling coding in DoFns. Eg. Encoding is taking in an element of the right type.

- Not an option when using a CombineFn, as there’s no other “hook” to override what gets sent over the wire.

Since the Go SDK and the Beam Portability FnAPI currently use Protocol Buffers, a hack exists that permits proto.Message types to be considered “concrete”. That is, at present, types are considered concrete during type analysis if the system thinks they’re protocol buffers.

# Appendix

## Glossary

**Coder:** An abstraction around how to encode and decode elements between bytes and real object types.

**Custom Coder:** See User Defined Coder.
**Concrete Type:** A term the Go SDK uses to mean a codable element type. If an element type is codable, it's concrete.
**Composite Type:** A term the Go SDK uses to refer to Beam specific concepts, in particular, KV pais, CoGBK results, and Windows.
**Default Coder:** General reflection based coder that can encode and decode almost any particular value passed in. Examples include the bob coder (encoding/gob) and the json coder (encoding/json).
**Element:** Any instance of a key or value type that can be part of a PCollection. Prefered over "value" to include keys, and avoid pointer, reference, value semantics inherent in Go. A synonym to Row.
**Row:** A Schema and Tabular abstraction of data.
**User Defined Coder:** Coders that are not "known" to Beam, or all runners. Also known as a custom coder.
**Value:** Overloaded term, that can be context dependant.

- As opposed to Pointer, WRT semantics for Go functions and methods.
    - In Go, value types and their associated pointers are distinct.
- As opposed to Key, WRT PCollection types. A value has an associated key, and both are element types.

## Related documents

- https://golang.org/pkg/encoding/gob
    - Allows custom encoding through gob.
- Other Beam languages
    - https://beam.apache.org/documentation/programming-guide/#data-encoding-and-type-safety (Python and Java Coder Registries)
- AutoValue Coding and Row Support
- Beam Schemas (TODO: Link to documentation when available.)