Problem overview

The beacon chain pyspecs are defined in an incremental way: the specs are split in phases. A phase declares its own definitions (constants, classes, methods, etc) and inherit definitions from prior phases.

This raises two questions: how exactly the phase own declarations should be combined to obtain a resulting spec. And how resulting (i.e. combined) phased specs interact during runtime (e.g. during upgrade to a newer phase).

In the write up, we mostly focus on the first question. Specs interaction during runtime is reflected in specs code (e.g. upgrade_to_xxx

methods), so the second question is partially covered in the context of merging spec definitions.

Basically, there are two approaches to combine phase definitions: either copy necessary definitions or import (reference) them. In practice, both approaches can be applied to different parts.

The main consequence of copying is duplication, which often leads to problems. However, copying introduces a new entity in a different context, so it is easier to attach contextual information to it. As a consequence, static analysis can be more precise.

There is no clear winner and, moreover, which approach is better (and for which part of pyspecs) may depend on a purpose. We assume three main goals:

- definition of Eth2 beacon chain specs (baseline)
- formal analysis of the specs
- specs implementations

One prominent consequence of duplication is generation of new entities (Python classes or methods, in our case), which directly impacts nominal type system. Due to dynamic nature of Python, it’s rarely an issue, though specs or libraries they depend on can call methods like isinstance

, issubclass

, type

, etc which behavior can be affected by such duplication.

Therefore, it’s perhaps more important from a formal method’s point of view, which is based on a static type system (it would be much more difficult to reason formally about a dynamic language and one has to introduce some type system anyway).

Another important consequence is that duplication of entities can require duplication of associated development: libraries, annotations, lemmas, proofs, etc. Since software quality is ubiquitous in the context of blockchains there are several directions where formal methods are applied. Therefore, such duplication can seriously affect such developments (dealing with them is rather straightforward but likely to be labor consuming).

Current state

Merge (combine) procedure

Currently, the merge (combine) procedure is straightforward: a definition from a prior phase is simply copied to the effective phase spec, unless it’s re-defined in a superseding phase (e.g. see setup.py).

For example, a class like Slot

(Epoch

, Root

, Gwei

, BLSPubkey

, BLSSignature

, etc) defined in phase0

phase is mapped to phase0.Slot

, altair.Slot

, merge.Slot

, etc in the generated code of phase0

, altair

, merge

, etc phases. Same happens for method definitions, i.e. there is phase0.compute_committee

, altair.compute_committee

, merge.compute_committee

, etc.

However, imports (references) to prior phase definitions are present too. For example, merge.BeaconState

extends phase0.BeaconState

(i.e. the former references the latter when declaring it as a super-class). Another example is upgrade_to_altair(pre: phase0.BeaconState)

defined in altair

, which imports phase0.BeaconState

in addition to altair.BeaconState

.

Nominal vs Structural subtyping

Statically typed subset of Python (used to express pyspecs) and associated static analysis tools (likeMyPy) support both Nominal and a form of Structural subtyping (e.g. see PEP484, PEP544 or here).

However, supported structural subtyping is based on Protocol

s, i.e. a name and a definition is still introduced. It is thus less convenient to express in source code. For the reason, we'll treat pyspecs as based on Nominal subtyping.

Problems and consequences

We consider problems in the context of nominal subtyping. It's largely not an issue with structural subtyping, which is though more tricky to employ in Python.

Copying entities

Full phase interaction in runtime (e.g. during upgrade) in presence of copying means that there can be two or more classes corresponding to the same original concept. For example, phase0.Slot

and altair.Slot

should be the same, in general. However, it can be tricky to achieve with nominal subtyping, since they have different names, which makes them different types. Perhaps, contrarily to pyspecs developers' intentions (see an example of such problem).

One consequence is that one need to copy other classes and methods that depend such affected classes. It's already done for phase definitions (within the current merge approach), however, it should be done for affected library methods too.

In the context of Formal Methods, formal developments like proofs, lemmas, etc may need to be duplicated too. Though with expressive enough Formal Methods (e.g. based on Higher-Order Logic, Type Theory) the problem can be alleviated with appropriate equivalence proofs.

Another consequence is that one may need to introduce conversion methods, e.g. to convert entities like Slot

, Epoch

, etc between phases, e.g. during phase upgrade. It's not difficult to automate though.

Sub-classing entities

While importing an entity allows to avoid problems caused by duplication, it can conflict with typing rules in some cases. At the moment, some phases in pyspecs (merge

, sharding

) define some classes as extending a corresponding class from a prior phase (e.g. merge.BeaconBlockBody

extends phase0.BeaconBlockBody

) with adding new fields or altering existing ones.

Altering a field

Altering a field adheres to certain limitations, if one wants to preserver type checking soundness:

- a field cannot be deleted in a subclass
- if a field type is modified, the new type should be a sub-type of the field's type in the super-class
- if a field's type allows mutation (e.g. it's a mutable collection), additional restrictions apply (e.g. MutableList[T]

is invariant, which means MutableList[A]

is not a subclass of MutableList[B]

, unless A

is the same as B

).

Since data-structures in pyspecs are often mutable, the last one is a serious restriction when one wants to alter a field.

Adding a field

Adding a new field though lacks such constraints. However, the new field might not be seen without appropriate checking/casting, if only a reference to the super-class is available.

For example, merge.BeaconBlockBody

extends phase0.BeaconBlockBody

. phase0.BeaconBlock

contains a field body

which points to an instance of phase0.BeaconBlockBody

, so merge.BeaconBlockBody

can be assigned to it. However, when reading the body

field one can only be sure that it has type phase0.BeaonBlockBody

. Thus new fields, added in merge

are not readily available, one should perform appropriate checks and casts.

To avoid the problem, one may consider inlining super-class instead of sub-classing it. However, the new version won't be a subclass of the entity in the prior phase (e.g. merge.BeaconBlockBody

won't be a subclass of phase0.BeaconBlockBody

, one can see an example of such approach in altair

). Thus one has to copy necessary classes and methods which reference this new entity version (that should be done transitively).

Option/variant types

In the presence of sub-typing, one may need to discriminate possible sub-types, given a reference to a super-type. E.g. call isinstance

or type

, etc.

Currently, pyspecs employs an alternative approach for ssz.Union

types (it is similar, but not identical to python's typing.Union

). The ssz.Union

has a selector

field, which determines which variant is held in its value

field. Compared to the standard Union

, the selector

field contains the necessary type info in an implicit form.

Such approach has an advantage that it's not directly affected by introducing entity copies. However, there will be problems with type-checking, if using off-the-shelve tools like MyPy, which are unaware of the convention.

See also #2333 as one more example.

Faulty ghost methods

One problem with the current merge approach arises when a field is dropped in an entity re-defined in a later phase (it's assumed here that the new definition does not

subclass the prior one). An example is altair

phase, where some fields are replaced in BeaconState

(e.g. previous_epoch_participation

instead of previous_epoch_attestations

). The methods from phase0

which reference the fields are still copied (e.g. get_matching_source_attestations

), which leads to type-checking problems. However, in practice such methods are never called in the altair

phase (they are "zombie" methods) and it's safe to drop them.

The problem can be resolved with "tree shake" approach (seebelow).

Invocation of reflection methods

Currently, there is one method in pyspecs which invokes type()

method (replace_empty_or_append

in custody_game

phase). However, it is only supplied with instances of CustodyChunkChallengeRecord

class, which is not redefined. It's rather a generic (template) method.

In addition, libraries can call isinstance()

, issublass()

or type()

too. For example, debug/encode.py

or debug/decode.py

use the methods. However, they apply them to types that common to all phases (like ssz.List

, ssz.Vector

, Container

, etc).

Therefore, in theory pyspecs behavior can be affected by how phase definitions are merged (whether entity copies are introduced or not), though it looks like unlikely in practice.

Problem analysis

The analysis of the problems above suggests that it may be beneficial to combine phase definitions differently depending on a goal.

Extra copies introduce duplication, which can complicate and/or stress static analysis or formal verification pipeline. They are less of a problem (if any) if just reading pyspecs or executing them.

However, static type checking/analysis - and heavier weight formal methods in general - are important tools for pyspecs development.

One reason is that full blown Formal Verification is resource consuming and only performed after a phase specification becomes stable. Thus pyspecs development should be supported with lighter-weight Formal Methods. Static type checking is attractive here, since it doesn't need extensive (and/or inconvenient, which can hurt readability) code annotations.

Second, static type analysis is able to establish important properties, which can be used to reduce efforts during Formal Verification and/or to generate tests more efficiently. E.g. detecting read-only fields, absence/presence of aliasing, parallel loops, etc.

Third, Formal Verification is based on static typing. Dynamic Python features can become tricky to model and reason about in a Formal Verification development. But if type discipline is not enforced during early stages (where full-blown Formal Verification is not applied), then it's highly likely that some dynamic features will leak in. One typically will need to re-write pyspecs, perhaps, several times, during Formal Verification development stage. As there are other pyspecs users (i.e. client implementer teams), which expect the specs to be in stable state, such specs modification can be a problem. Typically, one would prefer pyspecs to be in a “proper” shape before Formal Verification stage begins.

Formal Verification pipeline automation

There are several formal developments related to Eth2 beacon chain specs:using Coq, using K-framework, using Dafny.

Since, pyspecs are actively developed, it would be interesting to sync somehow pyspecs and such Formal Verification developments. Doing it manually is labor-intensive and error-prone. Semi-automated translation can be a way to go, especially assuming that there are semantic restrictions which fit well pyspecs and greatly simplify formal work (no floating point, no concurrency, limited OO features, mostly parallel loops, limited aliasing (including implicit copies when working with SSZ, many immutable data structures), object graph regularities).

Static type analysis is important to automate such transformations. Basically, one need to infer missing type information and potential implicit data dependencies (e.g. aliasing). Typing problems and “dangerous” aliasing are likely to indicate a bug. And when static analysis can establish some property then translation to other languages can be greatly simplified. For example, one can prove that when a location is destructively updated there is only one reference to it. Thus it's much easier to transform code to a non-destructive form.

How to alleviate the problems

Nominal vs Structural subtyping

Definition duplication leads to problems in the case of nominal type system. It's not a problem with structural subtyping - the structure is preserved during copying, so a duplicate entity will look the same, only its name being changed. The latter is the problem in the context of nominal typing. It's thus suggestive of using structural subtyping instead of nominal.

However, in practice, there are problems.

First, Python is based on nominal subtyping, though a form of structural subtyping is supported too. Since methods like isinstance

or type

can be called inside libraries, it’s still a problem, even of pyspecs themselves would be based on structural subtyping only.

Second, structural subtyping can miss certain problems, some of them being rather nasty. One example is Slot

and Epoch

types, which are backed by uint64

(see e.g. #1962 as an example of a real bug). However, they are different entities and shouldn’t be mixed up. It’s however not that difficult to mix them up (which happens in practice). With structural subtyping, however, they are the same entities, so it won’t help to detect such problems. Whereas nominal subtyping is able to distinguish Slot

and Epoch

, since their names are different.

Actually, there are many such cases in pyspecs, when a new name is introduced, backed by a simple type, like uintXX

or ByteXX

.

Third, in the context of Python the standard way to introduce structural subtyping is to use Protocol

's, which can be inconvenient (it requires introducing a name and a definition, which can hurt readability). Note, however, that custom static type analysis, which is based on structural subtyping is still possible, even if Protocol

's are not used.

Hybrid and customized subtyping

It may be useful to develop a customized approach to subtyping, combining benefits of both nominal and structural approaches.

Structural subtyping with ghost fields

One way is to use structural subtyping as the base approach, but, for example, introduce a so called “ghost” fields, which will make entities with similar structure look different. E.g. to distinguish Slot

from Epoch

(and other similarly structured entities), one may add kind: SlotType

field to Slot

and kind: EpochType

field to Epoch

. The entities are incompatible structures now. One however should ensure that such fields are read-only and can hold only one value (so that such field can be safely excluded from state).

Such “ghost” fields can be hide from end users, e.g. added during pre-processing of pyspecs (in a static analysis pipeline).

Inferring structural annotations

One more variation can be employed to alleviate overhead and restrictions associated with typing.Protocol

. For example, there is a method, which receives BeaconState

, however, it uses only one field of it (for example, consider get_current_epoch(state: BeaconState) -> Epoch

).

With nominal subtyping it’s rather restrictive, i.e. BeaconState

s from other phases have the slot

field, while not necessarily are sub-classes of phase0.BeaconState

. However, one may infer with static analysis that the method needs only slot field of Slot

type. Thus a structural type annotation can be inferred for such methods.

The approach even more attractive if distinguish read-only and modifying accesses - i.e. one can infer not only which fields are used, but also how they are used (e.g. one can infer that get_current_epoch

needs get_slot()

method, but doesn't need set_slot(Slot)

method).

"Smart" merge of definitions

We have investigated an approach to merge phase definitions, which tries to avoid unnecessary copying. The most prominent example is entities like Slot

, Epoch

, BLSKey

, etc which are not altered in later phases. One thus can import them, instead of copying, which gets rid of a significant portion of entity duplication and related problems.

The approach works as follows:

- Determine which classes are re-define in the current phase - let's call them affected classes

set

- Calculate which classes from prior phases that depend on the affected classes
- they should be included in the affected classes

too, as a type of at least one of their field has changed

- Repeat the previous step, until affected classes

stops changing (i.e. calculate "transitive closure")

- Similarly, calculate the initial affected methods

set - include there methods which are re-defined in the current phase and methods which depend on classes from the affected classes

set

- Similarly, calculate transitive closure of the affected methods

set, by adding methods which depend on methods in the affected mehods

set until it saturates

The approach currently ignores sub-classing (let's assume that sub-classes' fields are inlined first), though it can be modified to deal with it.

Tree shake

There is an additional "tree shake" step in the above "smart" merge procedure, which leaves only methods, which are reachable from a root set of methods. It's needed to drop methods from prior phases, which can be incompatible with latest changes (e.g. accesses fields which are dropped in the current phase - see an example in the above Faulty ghost methods section).

Avoid sub-classing

Beacon chain specs developers deliberately avoid following certain Object-Oriented patterns like implementation inheritance (in particular, phase combining is not based on object-oriented inheritance). However, sub-classing is present. As we can see above, it can lead to type-checking problems, both when one wants to add a new field or alter existing one.

Since implementation is not used (and is not actually welcome, at least, if one wants to be consistent with the current style the pyspecs are written with) it becomes more tricky to resolve such problems.

A more holistic approach would be to avoid sub-classing altogether and inline super-classes. It may require copying affected definitions (classes and methods), but the typing problems imposed by sub-classing will be avoided.

Summary

It’s currently not clear whether one should change the current phase merge procedure as a way to obtain the “baseline” beacon chain pyspecs. With the dynamic nature of Python it doesn’t look like a real problem.

There are problems in static analysis context, but they can be resolved by combining phase definitions differently within static analysis pipeline. We are planning to investigate the above structural annotation inference idea together with the “smart” merge approach and other ideas.

One risk here is to that such customized phase merge will contradict the “baseline” merge approach.