Beacon chain specification developers chose Python as the language to express executable parts of the Eth2 beacon chain spec (referenced to as pyspec

hereinafter). One of the most important benefit here is that a Python code is easy to read and write for general audience. Even if a programmer is not familiar with Python (but is familiar with some other popular programming language), it's not difficult to read the spec - and to learn necessary features. The pyspec developers facilitate this further by limiting themselves to a subset of Python features: a statically typed subset, avoiding of tricky Object-Oriented features, etc.

However, in the formal method context, there is an obvious downside: Python support from formal method tools is almost absent. Two notable exceptions are mypy and Nagini, both are built upon statically typed Python subset (PEP-484).

Lack of tool support additionally complicates formal work, which is already quite (or even extremely) complicated. At the same time, blockchain application require very high level of software quality, as bugs can be costly and difficult to fix after deployment. Which makes employment of formal methods inevitable.

There can be several ways to overcome the problem. One is to (regularly) translate the pyspec and its updates to a language that is more suitable for formal work (e.g. beacon chain in Dafny, beacon chain in K Framework). Even more ambitious would be to develop the executable parts of the spec in a language with better formal grounds. Both approaches have downsides: synchronizing developments in two languages is costly and error-prone, while a better language from a formal point of view will be more difficult to comprehend for a general audience.

# Tailored semantics & tools

We explore another approach: enrich the pyspec with (formal) semantics of the Python subset employed. This facilitates implementing a mechanical translation of the pyspec to other languages, including those with the state-of-the-art support from formal method tools. At the same type, the pyspec can be developed in (a subset of) Python.

Actually, it's typical for a formal method tool to be built around this principle. E.g. Why3 plugins translate to WhyML, Dafny translates to Boogie, etc.

The downside of the approach is that such development can be quite costly. We simplify overall efforts by following two principles:

- restricted set of features
- reductions to existing formal methods

## Exploiting problem domain specifics

One can simplify the job by exploiting problem domain specifics. In other words, we aim at a language with restricted features, which nevertheless fits well its intended application(s). The limitations allow for simpler formal work, including semantics, translation to other languages, formal and informal reasoning.

It would be more cost-efficient, if such semantics and tools could be applied to a wider set of problems. Therefore, while our primary focus is on the beacon chain problems, we discuss whether ideas can be applied other blockchain protocols, distributed protocols in general, network layer protocols, etc.

## Employing existing formal methods

Existing formal methods and tools impose their own sets of restrictions: formal reasoning can be overly cumbersome otherwise. So, restricting set of features can enlarge the selection of tools applicable and/or reduce efforts required to overcome such limitations.

On the other hand, too severe restriction means that it will be difficult to express desired programs and protocols in such overly restricted language. This will hinder readability and writability, so the goal won't be reached (it might be easier to use any mature existing formal method then).

# Expressiveness and Formal Reasoning

We can re-formulate the main problem as: find the right balance between expressiveness

(i.e. larger set of features) and amenability to formal reasoning

.

Actually, we are facing the problem whenever we are trying to apply formal methods to a more or less serious problem. Fortunately, there have been already done a huge amount of work, investigating various approaches to the problem and

their tradeoffs.

In the context of the beacon chain specification formal work, we propose the following set of restrictions (in addition to restrictions imposed by statically typed Python subset):

- no floating-point math
- no recursion
- “safe aliasing” rule
- no global or nonlocal writes
- no concurrency
- limited Object-Oriented features
- unambiguous type inference

NB

There are additional restrictions on Python features, which are not used in the pyspec sources, while may be cumbersome to support. We do not cover them, as they are not significant at this point of time.

The restrictions are chosen to minimize their impact on the existing pyspec sources, while retaining the ability to translate the pyspec sources to a wide selection of formal tools (without introducing complex encodings).

We justify our choices in a separate write up. In this write up we’d like to discuss the principles and how the appropriate semantic is going to be constructed.

Actually, current pyspec sources do not employ floating-point math, concurrency, global/nonlocal writes and complex Object-Oriented features. So we’ll concentrate on the “safe aliasing” rule (which is heavily inspired by the Rust’s Ownership Type System) and on inference of variable/declaration and resolving corresponding typing ambiguities.

However, the restrictions become more relevant, when considering a wider set of applications, e.g. network level protocols, other blockchains or distributed protocols, decentralized oracles, perhaps smart-contracts and precompiled contracts.

## Notes on Type System

While type system is definitely a very important part of semantics, we do not cover it here, since it’s already discussed in PEP-484. We do concentrate on the subset of features, used by pyspec sources, so describing and discussing potential restrictions and/or problems is reserved for (hopefully, not so distant) future write-ups.

# Restrictions in more details

## No floating-point math

This is an obvious restriction, as floating-point arithmetics is not employed in the beacon chain protocol and is not expected to be as it can be emulated by fixed point arithmetics and rational arithmetics, in many cases. So we just ignore it.

## No recursion

We assume that there is no recursion (including mutual one). While it can be supported in principle, current pyspec doesn’t use recursion. As in the case of floating-point math, we prefer to avoid unnecessary complications.

## “Safe aliasing” rule

Inspired by Rust’s Ownership Type System, we define “dangerous” aliasing as:

- Two or more pointers access the same data at the same time.
- At least one of the pointers is being used to write to the data.

This is a partial quotation from Rust’s References and Borrowing, see more details and a justification here.

The “safe aliasing” rule correspondingly forbids the situation of “dangerous” aliasing. Basically, for any memory location and at any point of time, two following conditions cannot be true simultaneously:

- there is one or more immutable references to the location
- there is a single mutable reference to it

NB

If there are zero references, the location should be reclaimed.

Note, that the rule conflicts with the Python reference counting in some sense. For example, consider the following program:

def method_a(a: A): # modifies 'a' ...

def method_b(): a = A() method_a(a) # may use 'a' further ...

The method_b

method creates an instance of A

and (its stack frame) holds a reference to it. At the same time, after invocation of method_a

, its stack frame holds a reference to it too. So, there are two references, while one is mutable (as method_a

modifies a

).

Rust resolves the problem by introducing borrowing

, i.e. method_a

borrows a

from method_b

and returns it back to method_b

after returning. So, while stack frames of both methods appear to hold references to the same object, they do it during disjoint periods of time (at least, it looks so to a programmer).

Similar ‘ownership transfer’ policy ensuring that there is at most one mutable reference (at each point of time) can be implemented on the semantic level.

## Destructive updates and pure form

One of the main purposes of introducing the “safe aliasing” rule is to support conversion to a pure form, without cluttering the output code.

For example, an array update a[i] = v

can be translated to its non-destructive counter-part a_1 = a.update_at(i, v)

. As there is only one mutable reference to a

then it’s easy to use the new a_1

version instead of a

afterwards. At the same time, no one references the a

version after update, so it can be safely dropped.

An update of a structure can be handled similarly, i.e. a destructive update a.field = v

can be replaced with something like a_1 = a.copy(field = v)

. Again, further uses of a

can be replaced with uses of a_1

, while the a

version can be safely dropped.

NB

If a function modifies its parameter, it may need to return the new version of the parameter, i.e. if a caller of the function uses it.

A pure form of a function is much more attractive for applying formal methods, since a formal method needs a memory model to resolve implicit dependencies between code units accessing shared memory due to aliasing. If such a model is coarse-grained then there can be a lot of potential dependencies (due to fewer restrictions on aliasing). If such a model is fine-grained then there are fewer opportunities for aliasing, however such model may require additional annotations. Or bugs can be introduced, if there is no static checking of whether the model assumptions are satisfied or not.

As our “safe aliasing” rule forbids “dangerous” aliasing, there can be either one or more several immutable references, i.e. the code is already in a “pure” form. Or there is an exclusive mutable reference, and the code is easy to transform to a pure form, by replacing destructive updates with non-destructive ones, renaming variables and so on.

NB

There is additional discussion on the Why pure functional form matters write up.

## No concurrency

While the current pyspec sources do not employ concurrency directly, it’s a more tricky restriction as it may be required in a wider context. We assume however that concurrency can be introduced in a safe and non-intrusive way.

As our “safe aliasing” principle is heavily influenced by Rust, we should rather say that concurrency becomes an orthogonal aspect. I.e. concurrency and shared memory is possible, but other control flows cannot adversely affect semantics, as “dangerous” aliasing is prohibited. Therefore, we may reason about behaviour of a function as if there is no concurrency at all.

If concurrency is needed, one may employ e.g. Actor model or other framework, which will benefit from such memory isolation.

## Global and nonlocal writes

In Python, it’s possible to write to a variable in an enclosing scope, e.g. global

, nonlocal

or assignment expression. These are obvious side effects, which complicate reasoning about programs.

From aliasing point of view, this is one more example of the “dangerous” aliasing.

In certain cases, such construct may be necessary or quite useful to implement caching/memoization. We assume that such constructs are inappropriate on the specification level, but are okay in optimized versions. For example, one can replace a method with its memoizing variant.

## Limited Object-Oriented features

Certain Object-Oriented patterns are often considered dangerous and/or counter-productive, e.g. calling methods inside a constructor or implementation inheritance. The problems can be exaggerated in the formal method context, so, it’s better to avoid them.

However, we’d still like to keep “safe” Object-Oriented features, as they are popular, convenient and familiar to general audience (and Python is intended to support the Object-Oriented paradigm).

We believe that such problems arise due to the same “dangerous” aliasing, which we avoid by the “safe aliasing” rule. So, our approach could be that a particular Object-Oriented feature is okay, unless it contradicts the “safe aliasing” rule.

However, to simplify semantic work we’d like to apply stricter restrictions, at least, initially. Currently, pyspec sources do not extensively use Object-Oriented features.

## Unambiguous type inference

Python doesn’t require that a variable is always declared before its use, in the sense that a variable is associated with its type explicitly (although PEP-484 provides a way to do that). In some cases, a variable type can be inferred, for example, if there is a single value definition (and we can infer the type of the expression defining the variable).

However, it’s possible that there are two or more control flow paths, defining two or more distinct values for the same name. It can also be the case that one path defines a value and another doesn’t. So, an ambiguity in type inference is possible.

For example,

if : a = else: a =

Another example is for

statement:

for i in : ...

In Python, i

is available after for

, if

resulted in a non-empty collection (i.e. a for

body has been executed at least once). However, if

returned an empty collection, then i

is undefined.

We use the following approach to resolve the ambiguities:

- a control flow graph for each function is constructed
- the CFG is converted to Static Single Assignment (SSA) form
- if SSA cannot be constructed (a live variable has no definition for some control flow path), then it's a mis-specified function
- in the SSA, each variable is defined only once, so one can infer its type, if there is a procedure to infer type of any expression
- when two control flow paths join, a Phi-node can be inserted, so there should be a typing rule that handles joins. There can be two cases:
- a join induced by an if

statement: the 'join' typing rule is acyclic here

- a join induced by a loop statement (for

or while

): there can be recursive dependencies between expressions, so the 'join' typing rule should support calculation of a fixpoint

- a join induced by an if

statement: the 'join' typing rule is acyclic here

- a join induced by a loop statement (for

or while

): there can be recursive dependencies between expressions, so the 'join' typing rule should support calculation of a fixpoint

We do not cover typing rule details here, they are to be discussed in a separate write up.

# Semantics implementation details

It's important to clarify our treatment of semantics here. Operational semantics seems to be a natural choice, as Python is an imperative (and interpreted) language.

However, our main goal is translation to other languages, preferably in a human-readable form. Variable declaration inference is a critical step here. Additionally, we deliberately introduced the "safe aliasing" rule to be able to translate destructive updates to a pure form. Therefore, Denitational Semantics may be more appropriate here.

In practice, we started with implementing not so formal "semantics" in an executable form, as a translation from Python AST to other languages (Kotlin currently). The process is:

- convert sources to AST
- convert AST to CFG, and CFG to SSA form
- infer variable declarations based on SSA
- type inference and type checking
- translate to a target language (using variable declaration and type inference results)

The first four steps are common to transpilation to any target language, and we are using them to define our semantics in practice. Basically, we can translate the pyspec sources to a form, which can be recognized as “semantic”.

The process can hardly be called “formal”. However, at the moment, translations to other languages is a more important and practical goal. There will be inevitable bugs introduced by the translation process itself. We plan to formalize it eventually, using Coq, perhaps except for the parsing stage.