TL;DR

Ethereum PoS Consensus pyspecs are expressed using destructive (i.e. in-place) memory updates. Reasoning about such updates can be complicated due to aliasing. However, in-place updates are (much) faster and often more readable. These two aspects are important, since the pyspecs are executable and employ nested data structures quite intensively.

The core problem is that a destructive update of a location can affects other parts of code via aliased references. Often unintended, such update can break invariants and assumptions throughout the entire codebase. Therefore, it’s important to limit aliasing - ideally, one should make updates via exclusive references only. Such “exclusive mutable reference” form can be translated to a pure form without much code blowup (as there is no aliases to propagate updates to).

The exclusive mutable reference

requirement may seem too severe, however, a practical implementation can allow for richer forms of aliasing, transforming them to the exclusive mutable reference

form under the hood (or rejecting the program, if such reduction is not possible).

Moreover, destructive updates via non-exclusive references can be dangerous and thus inappropriate, in the sense that they introduce opportunities for bugs (e.g. a missing copy operation). Ensuring that any mutable reference is exclusive prevents certain class of bug (e.g. one should make a copy of an immutable object to be able to modify it).

Such approach is aligned with the real pyspecs code: aliasing is relatively rare here, due to extensive use of copy operations and domain specifics. We therefore believe that imposing such constraint has a minor impact on how pyspecs are written, while allows to catch certain class of bugs and benefit from automated translation to pure form (without significant code blow up).

Actually, the approach is largely inspired by Rust’sownership memory model, which is built around the same exclusive mutable reference principle. The success of the Rust proves that such restriction is indeed practical: a reasonable memory access discipline allows to obtain memory safety guarantees.

In the post, we first formulate a simple memory model, dubbed “Mutable Forest” Memory Model, which describes the exclusive mutable reference

principle in a more rigorous way. We discuss motivation for such model. We then discuss how the model can be applied in practice, allowing for richer aliasing without loosing of benefits of the stricter approach.

Since our target language is Python and our main goal is transformation to pure form, we follow a somewhat different route than Rust’s designers, however, the differences are not very significant.

Mutable Forest

Memory Model

Base memory model assumptions

We assume that memory consists of a set of objects, each object having a set of fields. Each field can either contain a value type (like an atomic type or a structure, like tuple or record) or a reference to an object (can be nil

too). There can be two reference kinds: a reference to mutable

objects and a reference to immutable

ones.

An immutable object

is an object, which fields and fields of its descendants (reachable via references from it) are not allowed to be modified.

A mutable object

is an object such that either some its fields can be modified or it has a descendant, which fields can be modified.

NB

One can distinguish directly

or indirectly

mutable object, but this distinction is not important here.

Perhaps contrary to object-oriented languages, we assume that immutability

is enforced on the reference level rather than on the object level

So, there can be:

- an immutable

reference, which cannot be used to modify fields of objects reachable from the reference

- a mutable

reference, which allows to modify fields of an object it immediately points to (given that the field is not marked read-only)

Exact implementation details are not specified, but a most straightforward approach is to have a reference-to-immutable-object

type and if there is a field or a variable of this type, static type checker prevents one from modifying fields reachable from the reference.

One reason to employ such approach is that a mutable object can reach immutable state without explicit conversions, i.e. when there exists no mutable references to it. Under Safe Aliasing

property specified further, when one obtains an immutable reference, then there exists no mutable references to the object, so one can be sure the object is a truly immutable one.

Note that a mutable object is allowed to have a field pointing to an immutable object. Such field can be modified - contrary to fields of the objects reachable from it, which stay immutable.

Aliasing

Aliasing is a situation where a memory location can be accessed via different symbolic names. In our case, a memory location corresponds to a field (of an object).

NB

We have discussed object

references before. We can introduce a related concept of a field reference as a pair of object reference and a field accessor. The latter can be a field name or an index (in a list or an array).

If there are two reference to the same location (i.e. a field of an object), one of them being mutable

, there can arise an implicit data dependency. In other words, if one updates the location (via the mutable

reference), then before-update information obtained via aliased references is no longer valid in the after-update memory state.

The situation is potentially dangerous as a bug can be introduced - i.e. an update can unintendedly break an invariant or an assumption in other parts of code, which can access the location via aliases. In order to avoid certain class of bugs and the necessity to re-establish facts and invariants in the after-update memory state, we want to avoid such situations altogether.

NB

Such situations are sometimes called data races in a multi-threading context (see also here). However, similar problems arise in the single-threaded context too.

We therefore formulate Safe Aliasing

property (adapted from Rust's The Rules of References):

for any location and at any point of time:

- either all references to the location are immutable
- or, there is a single (exclusive) mutable reference to it

NB

Immutable aliasing is not typically a problem, at least from correctness point of view. It can become an obstacle for optimizations though e.g. see here.

Mutable Forest

Memory Model definition

There is a simple observation, laying in the foundation of our model. Let’s take all objects connected by mutable references. If the Safe Aliasing

property holds then any mutable object is a part of a tree, since a tree node can have at most one parent (corresponds to a field or a variable holding a reference to the node in our case).

Vice versa, if any mutable object is a part of a tree the Safe Aliasing

property is effectively enforced.

Since a set of trees is called forest, the set of mutable objects under Safety Aliasing

property can be seen as a mutable forest.

For simplicity, let’s assume there is a single

reference, so that only objects reachable from the

exist (e.g. in a current view/state).

We can now define the Mutable Forest

memory model as a base model (described above), where all mutable objects constitute a part of a tree (originating from the

reference). Or, alternatively, as a base model, where Safe Aliasing

property holds.

Model properties

Ownership transfer

A single exclusive reference can be seen as an owner of the object it points to. An ownership can be transferred from one reference to another using an atomic swap operation (e.g. a parallel assignment), which becomes the primary tool for maintaining the Safe Aliasing

property during memory graph modifications.

Stack vs Heap

An important property of the model is that it doesn’t explicitly distinguish stack frames and heap objects. This makes the model inconvenient in practice, since one cannot keep a reference to a mutable object both on stack and on heap (in a field of another object) - it will obviously violate the Safe Aliasing

property.

In practice, it may be necessary to allow on-stack aliasing when passing mutable references to method calls. We will discuss how to deal with such aliases below.

Use cases

Let’s consider a couple of examples: transformation to pure form and enforcing immutability.

Transform to pure form

As our target language is Python, we consider two destructive update forms here:

a.f = 1 # update of a field c[i] = 1 # update of a list or a dictionary's element

One can define non-destructive counterparts, which make and return a copy of the self

object, where one field or element is affected, while rest left intact, e.g.

a = a.updated(f = 1) c = c.updated_at(i, 1)

Nested data structures

One can update nested data structures

a.f.g = 1 a.c[i] = 2

which will look somewhat ugly in the non-destructive case.

a = a.updated(f = a.f.updated(g = 1)) a = a.updated(c = a.c.updated_at(i, 2))

Updates of deeply nested structures will look even more horrible, so one should perhaps consider using Lens/Optics.

Effect of Aliasing

In presence of aliasing, i.e. two expressions referencing the same location, situation can become more complicated.

For example,

assume a.f.g != 1

b = a.f b.g = 1 assert a.f.g == 1 # shouldn't fail

However, one cannot just simply translate the code to

assume a.f.g != 1

b = a.f b = b.updated(g = 1) assert a.f.g == 1 # fail

b.updated(...)

returns a new version of the object referenced by b

, whereas, the object referenced by a

has been left intact.

The problem is that a.f

and b

point to the same object, so an update of b.g

is seen when reading a.f.g

. One just should propagate changes to aliases in the non-destructive version of the example.

assume a.f.g != 1

b = a.f b = b.updated(g = 1)

propage changes to aliases

a = a.updated(f = b) assert a.f.g == 1 # ok now

So, one should perform an alias analysis when transforming an impure code to pure form. Such analysis is not decidable in general though. Restrictions on aliasing should be thus introduced, if a more or less straight-forward translation is desired.

The Mutable Forest

memory model requires that any mutable reference is exclusive. Thus, if an object field is modified then it is performed using a mutable reference and therefore no other reference (i.e. alias) exist. Thus no information need to be propagated.

Ensuring immutability

Immutable objects are natural in software engineering, e.g. they can be used to represent facts or express design principles. Inadvertent modification of an object, which should actually be immutable can become a serious problem, which can be difficult to debug. It's thus tempting to enforce immutability in some way.

One quite common, but error-prone approach is to define a read-only interface to an entity and extend it with a mutating interface (e.g. that can be a IBeaconState

and IMutableBeaconState

interfaces in the case of beacon chain implementation). However, such approach can lead to problems in practice, since a code which has a reference to the read-only interface can't assume it's an immutable object, it just not allowed to modify the object. But there can exist another reference to the object, which allows modification. Or the read-only interface can be cast to the mutating one.

A safer approach would be to define two related classes: immutable and mutable ones (e.g. BeaconState

and MutableBeaconState

), and convert one into another, when necessary. Such approach is slower and less convenient, but can prevent such bugs more reliably.

There can be a third approach, which uses static analysis to determine when an object is safe to modify (e.g. using typestate analysis). For example, initially an object is mutable, however, once it reaches certain state (e.g. it's assigned to an immutable

reference), then it becomes immutable and is not allowed to be modified. For example,

a = A() a.f = 1 # okay, in a mutable state b.some_field_marked_immutable = a

reached immutable state

a.f = 1 - error now

c = a.f # read access is still ok though

In the Mutable Forest

memory model, any mutable

reference is exclusive. So, when one assigns the mutable

reference to an immutable one then the exclusive mutable reference should either be destroyed or cast as an immutable

reference.

Practical Considerations

On-stack aliasing

As noted above, the model itself doesn't distinguish stack frames and heap objects. Therefore, introducing an on-stack alias for an object referenced from heap and vice versa will violate the Safe Aliasing

property.

Constraining programs from introducing on-heap aliases is actually fine (at least, in our situation), since tracking heap aliases can be much more problematic.

Tracking on-stack aliases is easier though (given that there are no pointers to stack variables). Moreover, on-stack aliases are often introduced when passing references as parameters to another method: the traditional approach keeps references both on the caller's stack frame and on the stack frame of the callee. A similar problem can arise when returning a mutable reference.

We therefore propose an extension of the Mutable Forest

memory model, which allows on-stack aliases. The extension is again largely inspired by the Rust'sownership memory model. Though there are some differences in our approach to make it more suitable for the subset of Python, employed to express the pyspecs.

Extended Mutable Forest model

In the extended Mutable Forest model, we assume that the Safe Aliasing

property still holds for the heap objects, however on-stack aliases can be introduced. That means, heap object graph satisfies the property, if all on-stack aliases are dropped.

In order to simplify alias analysis, we additionally assume that at the beginning of a method call, the Safe Aliasing

property holds too. It means that when passing a reference to a mutable object as an actual parameter of a method call, its ownership is transferred.

Similarly, when returning a reference to a mutable object as a result of a method call, its ownership is transferred from the callee's stack frame to the caller's one. However, since the callee's stack frame is destroyed after the call is over, it's not a problem.

Borrowing vs Claiming

In many cases, it can be natural to transfer ownership of a formal parameter back to the caller. Following the Rust's terminology, we call such ownership transfer as borrowing

. However, in our model a method which receives a mutable reference as a parameter is not necessarily obliged to return ownership back. We call such ownership transfer as claiming

(i.e. claiming ownership of the parameter).

Casting as immutable ref and back

It can be the case that a reference to an mutable object is passed to a method which expects an immutable reference. In such case, an alternative approach is possible: the mutable reference is cast as an immutable one. So, the caller loses ownership of the object, but can still read its fields - Safety Aliasing

property allows multiple immutable references.

Additionally, it can be the case that the immutable parameter doesn't escape the method call (i.e. neither it is returned nor assigned to a field of an escaping object). In that case, after the method invocation is over, the caller becomes the exclusive owner of the object again. Therefore one can cast the immutable reference as a mutable one back.

On-stack aliases during a method call

The same three approaches can be applied inside a single method call too.

Let's assume that an on-stack alias is introduced, i.e. either a = b

or a = b.f

(a = b[i]

) statement has been executed. Let's call a

as aliased

variable and b

as base

variable - basically, a

is an alias to a node of the tree that starts from b

.

At some point of time the aliased

variable will be destroyed:

- either a method invocation is over
- or the aliased

variable is implicitly destroyed because it's never used after some point in program (i.e. the variable becomes 'dead')

- or it’s explicitly destroyed (e.g. by assigning nil

or by invoking del a

in the Python’s case)

Let’s call the period between alias creation and destruction as live range

(of the aliased

variable).

Then again, there can be three cases:

- if the aliased

variable has not escaped, then it can borrow ownership when created and return it back to the base

variable after the live range is over. The base

variable must not be used during the live range, but can be used afterward.

- the aliased

variable has escaped, i.e. its ownership has been transferred to another field or variable. Then ownership cannot be returned back to the base

variable, which corresponds to claiming

case above. The base

variable lost ownership and must not be used.

- the aliased

variable has escaped via assigning to an immutable field. Then both the base

and the aliased

variables are cast as immutable references and can be used to read the object fields. The object becomes effectively immutable, as there are no mutable references exist now.

Assigning to a field of an object

Let’s now consider the a.f = b

statement. The statement creates an on-heap alias, which can be much more difficult to track. Thus our model disallows introducing such on-heap aliases for objects referenced from stack variables. In order to fulfill the requirement, the on-stack b

reference should be invalidated after the assignment. For example,

def meth(a: A): b = B() b.g = 1 a.f = b # c = b.g - not okay, b has been implicitly invalidated

Examples

Here are some examples illustrating how the Safe Aliasing

property looks like in a Python-like programs.

borrowing mutable parameter

A callee can mutate a borrowed object and pass it as a borrowing

parameter to other calls. It is not allowed to escape.

A caller regains full ownership after the call is over.

def meth(borrow mut a: A) -> A: a.f = 1 # can mutate meth_b(a) # can pass, if the first parameter of 'meth_b' is borrowing b = B() # b.g = a - can't do that # return a - can't return

a = A() meth(a) a.f = 2 # ownership is returned back

claiming mutable parameter

A callee obtains full ownership of the claiming parameter, so such parameter can escape via return or assigning to a field of an escaping object.

After the call is over, the caller cannot use a reference passed as claiming

parameters.

def meth(claim mut a: A) -> A: a.f = 1 meth_b(a) b = B() # b.g = a - is not okay when 'a' will be returned return a

a = A() a2 = meth(a)

a.f = 1 - is not okay, ownership is lost

b = a.f - is not okay either

a2.f = 2 # okay, ownerhsip is back via return

or

def meth(claim mut a: A) -> B: a.f = 1 meth_b(a) b = B() b.g = a # meth_b(a) - can't use 'a' anymore return b # but it wouldn't be okay to return 'a'

a = A() b = meth(a)

a.f = 1 - is not okay again

b.g.f = 1 # okay, ownerhsip is back via return

immutable parameter

An immutable

parameter of a method forces an actual parameter become an immutable one.

A caller can pass a reference to a mutable object, since it’s an exclusive owner of such object. It loses full ownership, but can still read its fields.

def meth(immutable a: A) -> B: b = B() b.f = a # okay if assigning to an immutable ref return b

a = A() a.f = 1 b = meth(a)

b = a.f # okay, it's accessible

a.f = 1 - not okay, it's an immutable ref

Conclusion

One of the goals to introduce the Mutable Forest

memory model is to build a foundation for transforming programs with destructive updates into pure form. In general, such conversion can transofrm code so that it would be very difficult to read and/or analyze formally. On the opposite side, if every memory update is performed via an exclusive reference, transformation to pure form results in a code which is much closer to original.

There are still problems in practice, when dealing with the pyspecs code. We will discuss the problems and how they can be resolved in a follow-up post.