One question that has come up in our resource machine specification efforts is whether resource logics should be able to see all resources which are present (consumed or created) in a partial transaction, or whether they should only be able to explicitly check for inclusion of specific resources or resources with particular properties. In particular, the former would allow a resource logic to perform a kind of “forall” check, of the form:

logic (...) newResources = forall resource <- newResources if resource.kind == B return false ...

This kind of check is not possible if a resource logic only sees a subset of the resources consumed, as it could no longer exclude the possibility of a resource with kind B

(for example) existing outside of the subset which it sees.

A resource logic can already enforce arbitrary positive

checks (statements about the consumption or creation of such-and-such a resource with such-and-such a property) by requiring the existence of a temporary resource which must be balanced, and whose logic encodes the check in question. A forall check would add the ability to enforce negative

checks (statements about the non-creation or non-consumption of such-and-such a resource with such-and-such a property). In a certain sense, this is more powerful. However, it comes with a number of potential downsides:

Validity under compositionality

Resource logics no longer obey validity under compositionality. If resource logics only check a subset, once a resource logic returns true, a proof can be created, and the proof will remain valid no matter what other partial transactions the original partial transaction is composed with. In a naive implementation, if resource logics can iterate over all consumed and created resources, proofs can only be made at the end, which means that we would give up both the added privacy that would otherwise come from partial solving and the possibility of hybrid public-private transactions (which would typically entail adding consumed/created resources in the post-ordering execution stage). A more sophisticated implementation would split up the resource logic into the regular logic, with only positive checks and for which a proof can be made as soon as the positive checks are satisfied, and a set of negative checks for which proofs must individually be made about subsequent resources added to the partial transaction (in the created or consumed sets). This would recover the privacy benefits (sans, of course, the information in the negative checks) and a form of validity under compositionality, but it will add some implementation complexity.

Relatedly, negative checks create a counterintuitive disequivalence between combined balanced partial transactions and separate balanced partial transactions. Consider two partial transactions A and B, both balanced. If negative checks are allowed, A may be a valid balanced transaction, and B may be a valid balanced transaction, but A + B may be an invalid

transaction due to some negative condition in a resource logic in A which is violated by some resource in B or vice-versa.

Resource wrapping

Plausibly, the intention of negative checks may be to prevent interaction with some undesirable resource, perhaps a user or asset one wishes to sanction for whatever reason. Regardless of the motivations in doing so, this approach is not necessarily as effective as one might imagine, because any resource can always be wrapped in another resource which directly controls it but would not trigger the negative check. Say that the negative check checks for non-existence of a resource with kind B - I can simply create a new resource kind B’, where anyone can create one B’ from one B, or convert a B’ back into a B. Of course, additional negative checks could be added, but this game of identifier cat-and-mouse does not seem to provide the original property we thought we might be able to obtain.

My current conclusion is that this kind of negative check is worth continuing to investigate, but not yet worth implementing. Discussion welcome!