Simulating References

Everything on the Sui blockchain is an object. When you develop Move packages for the Sui network, you're typically manipulating or using on-chain objects in some way through functionality available in the Sui API. For most API functions, you provide an object by reference.

References are a key construct when programming in Move and on Sui. Most of the functionality available in the Sui API takes objects by reference.

There are two ways to use an object:

Programmable transaction blocks (PTBs) do not currently allow the use of object references returned from one of its transaction commands. You can use input objects to the PTB, objects created by the PTB (like MakeMoveVec), or returned from a transaction command by value as references in subsequent transaction commands. If a transaction command returns a reference, however, you can't use that reference in any call, significantly limiting certain common patterns in Move.

The Sui framework includes a borrow module that offers a solution to the reference problem. The module provides access to an object by value but builds a model that makes it impossible to destroy, transfer, or wrap the object retrieved. The borrow module exposes a Referent object that wraps another object (the object you want to reference). The module uses the hot potato pattern (via a Borrow instance) to allow retrieval of the wrapped object by value. Within the same PTB, the module then forces the object to be returned to the Referent . The Borrow instance guarantees that the object returned is the same that was retrieved.

As an example, consider the following module stub that exposes an object (Asset) and a function (use_asset) to use that object.

The function use_asset takes an immutable reference to the asset (&Asset), which is a common pattern in an API definition.

Now consider another module that uses this asset.

This module creates an object (AssetManager) that references the object (Asset) created in the previous module (a_module).

You could then write a Move function that retrieves an object by reference and passes it to the use_asset function.

The two functions in do_something are not valid within a PTB, however, because PTBs do not support a reference returned by a function and passed to another function.

To make this operation valid within a PTB, you would need to include functionality from the borrow module. Consequently, you could change the another_module code to the following:

Now the PTB can retrieve the asset, use it in a call to use_asset , and return the asset.

The Borrow object is the key to the guarantees the borrow module offers. The definition of Borrow is struct Borrow { ref: address, obj: ID } which makes it such that you cannot drop or save its instance anywhere, so it must be consumed in the same transaction that retrieves it (hot potato). Moreover, fields in the Borrow struct make sure that the object returned is for the same Referent and the object that was originally held by the Referent instance. In other words, there is no way to either keep the object retrieved or to swap it with another object in a different Referent .

Using a Referent is a very explicit and intrusive change. That has to be taken into consideration when designing a solution.

Support for references in a PTB is planned, which is a much more natural and proper pattern for APIs.

You must consider the implications of using the borrow module and whether you have a mechanism to later move to a more natural, reference pattern.

Finally, the Referent model forces the usage of a mutable reference and returns an object by value. Both have significant implications when designing an API. You must be careful in what logic your modules provide and how objects are exposed.

Extending the previous example, a PTB that calls use_asset is written as follows:

The borrow module

The Sui framework includes a borrow module that offers a solution to the reference problem. The module provides access to an object by value but builds a model that makes it impossible to destroy, transfer, or wrap the object retrieved. The borrow module exposes a Referent object that wraps another object (the object you want to reference). The module uses the hot potato pattern (via

a Borrow instance) to allow retrieval of the wrapped object by value. Within the same PTB, the module then forces the object to be returned to the Referent . The Borrow instance guarantees that the object returned is the same that was retrieved.

As an example, consider the following module stub that exposes an object (Asset) and a function (use_asset) to use that object.

The function use_asset takes an immutable reference to the asset (&Asset), which is a common pattern in an API definition.

Now consider another module that uses this asset.

This module creates an object (AssetManager) that references the object (Asset) created in the previous module (a_module).

You could then write a Move function that retrieves an object by reference and passes it to the use_asset function.

The two functions in do_something are not valid within a PTB, however, because PTBs do not support a reference returned by a function and passed to another function.

To make this operation valid within a PTB, you would need to include functionality from the borrow module. Consequently, you could change the another_module code to the following:

Now the PTB can retrieve the asset, use it in a call to use_asset , and return the asset.

The Borrow object is the key to the guarantees the borrow module offers. The definition of Borrow is struct Borrow { ref: address, obj: ID } which makes it such that you cannot drop or save its instance anywhere, so it must be consumed in the same transaction that retrieves it (hot potato). Moreover, fields in the Borrow struct make sure that the object returned is for the same Referent and the object that was originally held by the Referent instance. In other words, there is no way to either keep the object retrieved or to swap it with another object in a different Referent .

Using a Referent is a very explicit and intrusive change. That has to be taken into consideration when designing a solution.

Support for references in a PTB is planned, which is a much more natural and proper pattern for APIs.

You must consider the implications of using the borrow module and whether you have a mechanism to later move to a more natural, reference pattern.

Finally, the Referent model forces the usage of a mutable reference and returns an object by value. Both have significant implications when designing an API. You must be careful in what logic your modules provide and how objects are exposed.

Extending the previous example, a PTB that calls use_asset is written as follows:

Considerations

The Borrow object is the key to the guarantees the borrow module offers. The definition of Borrow is struct Borrow { ref: address, obj: ID } which makes it such that you cannot drop or save its instance anywhere, so it must be consumed in the same transaction that retrieves it (hot potato). Moreover, fields in the Borrow struct make sure that the object returned is for the same Referent and the object that was originally held by the Referent instance. In other words, there is no way to either keep the object retrieved or to swap it with another object in a different Referent .

Using a Referent is a very explicit and intrusive change. That has to be taken into consideration when designing a solution.

Support for references in a PTB is planned, which is a much more natural and proper pattern for APIs.

You must consider the implications of using the borrow module and whether you have a mechanism to later move to a more natural, reference pattern.

Finally, the Referent model forces the usage of a mutable reference and returns an object by value. Both have significant implications when designing an API. You must be careful in what logic your modules provide and how objects are exposed.

Extending the previous example, a PTB that calls use_asset is written as follows:

Example

Extending the previous example, a PTB that calls use_asset is written as follows: