Programmable Transaction Blocks

On Sui, a transaction is more than a basic record of the flow of assets. Transactions on Sui are composed of a number of commands that execute on inputs to define the result of the transaction. Termed programmable transaction blocks (PTBs), these groups of commands define all user transactions on Sui. PTBs allow a user to call multiple Move functions, manage their objects, and manage their coins in a single transaction--without publishing a new Move package. Designed with automation and transaction builders in mind, PTBs are a lightweight and flexible way of generating transactions. More intricate programming patterns, such as loops, are not supported, however, and in those cases you must publish a new Move package.

As mentioned, each PTB is comprised of individual transaction commands (sometimes referred to simply as transactions or commands). Each transaction command executes in order, and you can use the results from one transaction command in any subsequent transaction command. The effects, specifically object modifications or transfers, of all transaction commands in a block are applied atomically at the end of the transaction. If one transaction command fails, the entire block fails and no effects from the commands are applied.

A PTB can perform up to 1,024 unique operations in a single execution, whereas transactions on traditional blockchains would require 1,024 individual executions to accomplish the same result. The structure also promotes cheaper gas fees. The cost of facilitating individual transactions is always more than the cost of those same transactions blocked together in a PTB.

The remainder of this topic covers the semantics of the execution of the transaction commands. It assumes familiarity with the Sui object model and the Move language. For more information on those topics, see the following documents:

There are two parts of a PTB that are relevant to execution semantics. Other transaction information, such as the transaction sender or the gas limit, might be referenced but are out of scope. The structure of a PTB is:

Looking closer at the two main components:

Inputs and results are the two types of values you can use in transaction commands. Inputs are the values that are provided to the PTB, and results are the values that are produced by the PTB commands. The inputs are either objects or simple Move values, and the results are arbitrary Move values (including objects).

The inputs and results can be seen as populating an array of values. For inputs, there is a single array, but for results, there is an array for each individual transaction command, creating a 2D-array of result values. You can access these values by borrowing (mutably or immutably), by copying (if the type permits), or by moving (which takes the value out of the array without re-indexing).

Input arguments to a PTB are broadly categorized as either objects or pure values. The direct implementation of these arguments is often obscured by transaction builders or SDKs. This section describes information or data the Sui network needs when specifying the list of inputs, [Input] . Each Input is either an object, Input::Object(ObjectArg) , which contains the necessary metadata to specify to object being used, or a pure value, Input::Pure(PureArg) , which contains the bytes of the value.

For object inputs, the metadata needed differs depending on the type of ownership of the object . The data for the ObjectArg enum follows:

If the object is owned by an address (or it is immutable), then use ObjectArg::ImmOrOwnedObject(ObjectID, SequenceNumber, ObjectDigest) . The triple respectively specifies the object's ID, its sequence number (also known as its version), and the digest of the object's data.

If an object is shared, then use Object::SharedObject { id: ObjectID, initial_shared_version: SequenceNumber, mutable: bool } . Unlike ImmOrOwnedObject , a shared objects version and digest are determined by the network's consensus protocol. The initial_shared_version is the version of the object when it was first shared, which is used by consensus when it has not yet seen a transaction with that object. While all shared objects can be mutated, the mutable flag indicates whether the object is to be used mutably in this transaction. In the case where the mutable flag is set to false , the object is read-only, and the system can schedule other read-only transactions in parallel.

If the object is owned by another object, as in it was sent to an object's ID via the TransferObjects command or the sui::transfer::transfer function, then use ObjectArg::Receiving(ObjectID, SequenceNumber, ObjectDigest) . The object data is the same as for the ImmOrOwnedObject case.

For pure inputs, the only data provided is the BCS bytes, which are deserialized to construct Move values. Not all Move values can be constructed from BCS bytes. This means that even if the bytes match the expected layout for a given Move type, they cannot be deserialized into a value of that type unless the type is one of the types permitted for Pure values. The following types are allowed to be used with pure values:

Interestingly, the bytes are not validated until the type is specified in a command, for example in MoveCall or MakeMoveVec . This means that a given pure input could be used to instantiate Move values of several types. See the Arguments section for more details.

Each transaction command produces a (possibly empty) array of values. The type of the value can be any arbitrary Move type, so unlike inputs, the values are not limited to objects or pure values. The number of results generated and their types are specific to each transaction command. The specifics for each command can be found in the section for that command, but in summary:

Each command takes Argument s that specify the input or result being used. The usage (by-reference or by-value) is inferred based on the type of the argument and the expected argument of the command. First, examine the structure of the Argument enum.

Input(u16) is an input argument, where the u16 is the index of the input in the input vector. For example, given an input vector of [Object1, Object2, Pure1, Object3] , Object1 is accessed with Input(0) and Pure1 is accessed with Input(2) .

GasCoin is a special input argument representing the object for the SUI coin used to pay for gas. It is kept separate from the other inputs because the gas coin is always present in each transaction and has special restrictions (see below) not present for other inputs. Additionally, the gas coin being separate makes its usage explicit, which is helpful for sponsored transactions where the sponsor might not want the sender to use the gas coin for anything other than gas.

The gas coin cannot be taken by-value except with the TransferObjects command. If you need an owned version of the gas coin, you can first use SplitCoins to split off a single coin.

This limitation exists to make it easy for the remaining gas to be returned to the coin at the end of execution. In other words, if the gas coin was wrapped or deleted, then there would not be an obvious spot for the excess gas to be returned. See the Execution section for more details.

NestedResult(u16, u16) uses the value from a previous command. The first u16 is the index of the command in the command vector, and the second u16 is the index of the result in the result vector of that command. For example, given a command vector of [MoveCall1, MoveCall2, TransferObjects] where MoveCall2 has a result vector of [Value1, Value2] , Value1 would be accessed with NestedResult(1, 0) and Value2 would be accessed with NestedResult(1, 1) .

Result(u16) is a special form of NestedResult where Result(i) is roughly equivalent to NestedResult(i, 0) . Unlike NestedResult(i, 0) , Result(i) , however, this errors if the result array at index i is empty or has more than one value. The ultimate intention of Result is to allow accessing the entire result array, but that is not yet supported. So in its current state, NestedResult can be used instead of Result in all circumstances.

For the execution of PTBs, the input vector is populated by the input objects or pure value bytes. The transaction commands are then executed in order, and the results are stored in the result vector. Finally, the effects of the transaction are applied atomically. The following sections describe each aspect of execution in greater detail.

At the beginning of execution, the PTB runtime takes the already loaded input objects and loads them into the input array. The objects are already verified by the network, checking rules like existence and valid ownership. The pure value bytes are also loaded into the array but not validated until usage.

The most important thing to note at this stage is the effects on the gas coin. At the beginning of execution, the maximum gas budget (in terms of SUI) is withdrawn from the gas coin. Any unused gas is returned to the gas coin at the end of execution, even if the coin has changed owners.

Each transaction command is then executed in order. First, examine the rules around arguments, which are shared by all commands.

You can use each argument by-reference or by-value. The usage is based on the type of the argument and the type signature of the command.

The transaction fails if an argument is used in any form after being moved. There is no way to restore an argument to its position (its input or result index) after it is moved.

If an argument is copied but does not have the drop ability, then the last usage is inferred to be a move. As a result, if an argument has copy and does not have drop , the last usage must be by value. Otherwise, the transaction will fail because a value without drop has not been used.

The borrowing of arguments has other rules to ensure unique safe usage of an argument by reference. If an argument is:

Object inputs have the type of their object T as you might expect. However, for ObjectArg::Receiving inputs, the object type T is instead wrapped as sui::transfer::Receiving . This is because the object is not owned by the sender, but instead by another object. And to prove ownership with that parent object, you call the sui::transfer::receive function to remove the wrapper.

The GasCoin has special restrictions on being used by-value (moved). You can only use it by-value with the TransferObjects command.

Shared objects also have restrictions on being used by-value. These restrictions exist to ensure that at the end of the transaction the shared object is either still shared or deleted. A shared object cannot be unshared (having the owner changed) and it cannot be wrapped. A shared object:

Pure values are not type checked until their usage. When checking if a pure value has type T , it is checked whether T is a valid type for a pure value (see the previous list). If it is, the bytes are then validated. You can use a pure value with multiple types as long as the bytes are valid for each type. For example, you can use a string as an ASCII string std::ascii::String and as a UTF8 string std::string::String . However, after you mutably borrow the pure value, the type becomes fixed, and all future usages must be with that type.

The command has the form TransferObjects(ObjectArgs, AddressArg) where ObjectArgs is a vector of objects and AddressArg is the address the objects are sent to.

The command has the form SplitCoins(CoinArg, AmountArgs) where CoinArg is the coin being split and AmountArgs is a vector of amounts to split off.

The command has the form MergeCoins(CoinArg, ToMergeArgs) where the CoinArg is the target coin in which the ToMergeArgs coins are merged into. In other words, you merge multiple coins (ToMergeArgs) into a single coin (CoinArg).

The command has the form MakeMoveVec(VecTypeOption, Args) where VecTypeOption is an optional argument specifying the type of the elements in the vector being constructed and Args is a vector of arguments to be used as elements in the vector.

This command has the form MoveCall(Package, Module, Function, TypeArgs, Args) where Package::Module::Function combine to specify the Move function being called, TypeArgs is a vector of type arguments to that function, and Args is a vector of arguments for the Move function.

The command has the form Publish(ModuleBytes, TransitiveDependencies) where ModuleBytes are the bytes of the module being published and TransitiveDependencies is a vector of package Object ID dependencies to link against.

When the transaction is signed, the network verifies that the ModuleBytes are not empty. The module bytes ModuleBytes: [[u8]] contain the bytes of the modules being published with each [u8] element is a module.

The transitive dependencies TransitiveDependencies: [ObjectID] are the Object IDs of the packages that the new package depends on. While each module indicates the packages used as dependencies, the transitive object IDs must be provided to select the version of those packages. In other words, these object IDs are used to select the version of the packages marked as dependencies in the modules.

After the modules in the package are verified, the init function of each module is called in same order as the module byte vector ModuleBytes .

The command produces a single result of type sui::package::UpgradeCap , which is the upgrade capability for the newly published package.

The command has the form Upgrade(ModuleBytes, TransitiveDependencies, Package, UpgradeTicket) , where the Package indicates the object ID of the package being upgraded. The ModuleBytes and TransitiveDependencies work similarly as the Publish command.

For details on the ModuleBytes and TransitiveDependencies , see the Publish command . Note though, that no init functions are called for the upgraded modules.

The Package: ObjectID is the Object ID of the package being upgraded. The package must exist and be the latest version.

The UpgradeTicket: sui::package::UpgradeTicket is the upgrade ticket for the package being upgraded and is generated from the sui::package::UpgradeCap . The ticket is taken by value (moved).

The command produces a single result type sui::package::UpgradeReceipt which provides proof for that upgrade. For more details on upgrades, see Upgrading Packages .

At the end of execution, the remaining values are checked and effects for the transaction are calculated.

For inputs, the following checks are performed:

For results, the following checks are performed:

Any remaining SUI deducted from the gas coin at the beginning of execution is returned to the coin, even if the owner has changed. In other words, the maximum possible gas is deducted at the beginning of execution, and then the unused gas is returned at the end of execution (all in SUI). Because you can take the gas coin only by-value with TransferObjects , it will not have been wrapped or deleted.

The total effects (which contain the created, mutated, and deleted objects) are then passed out of the execution layer and are applied by the Sui network.

Let's walk through an example of a PTB's execution. While this example is not exhaustive in demonstrating all the rules, it does show the general flow of execution.

Suppose you want to buy two items from a marketplace costing 100 MIST . You keep one for yourself, and then send the object and the remaining coin to a friend at address 0x808 . You can do that all in one PTB:

The inputs include the friend's address, the marketplace object, and the value for the coin split. For the commands, split off the coin, call the market place function, send the gas coin and one object, grab your address (via sui::tx_context::sender), and then send the remaining object to yourself. For simplicity, the documentation refers to the package names by name, but note that in practice they are referenced by the package's Object ID.

To walk through this, first look at the memory locations, for the gas object, inputs, and results

Here you have two objects loaded so far, the gas coin with a value of 1_000_000u64 and the marketplace object of type some_package::some_marketplace::Marketplace (these names and representations are shortened for simplicity going forward). The pure arguments are not loaded, and are present as BCS bytes.

Note that while gas is deducted at each command, that aspect of execution is not demonstrated in detail.

Before execution, remove the gas budget from the gas coin. Assume a gas budget of 500_000 so the gas coin now has a value of 500_000u64 .

Now you can execute the commands.

The first command SplitCoins(GasCoin, [Input(2)]) accesses the gas coin by mutable reference and loads the pure argument at Input(2) as a u64 value of 100u64 . Because u64 has the copy ability, you do not move the Pure input at Input(2) . Instead, the bytes are copied out.

For the result, a new coin object is made.

This gives us updated memory locations of

Now the command, MoveCall("some_package", "some_marketplace", "buy_two", [], [Input(1), NestedResult(0, 0)]) . Call the function some_package::some_marketplace::buy_two with the arguments Input(1) and NestedResult(0, 0) . To determine how they are used, you need to look at the function's signature. For this example, assume the signature is

where Item is the type of the two objects being sold.

Since the marketplace parameter has type &mut Marketplace , use Input(1) by mutable reference. Assume some modifications are being made into the value of the Marketplace object. However, the coin parameter has type Coin , so use NestedResult(0, 0) by value. The TxContext input is automatically provided by the runtime.

This gives updated memory locations, where _ indicates the object has been moved.

Assume that buy_two deletes its Coin object argument and transfers the Balance into the Marketplace object.

TransferObjects([GasCoin, NestedResult(1, 0)], Input(0)) transfers the gas coin and first item to the address at Input(0) . All inputs are by value, and the objects do not have copy so they are moved. While no results are given, the ownership of the objects is changed. This cannot be seen in the memory locations, but rather in the transaction effects.

You now have updated memory locations of

Make another Move call, this one to sui::tx_context::sender with the signature

While you could have just passed in the sender's address as a Pure input, this example demonstrates calling some of the additional utility of PTBs; while this function is not an entry function, you can call the public function, too, because you can provide all of the arguments. In this case, the only argument, the TxContext , is provided by the runtime. The result of the function is the sender's address. Note that this value is not treated like the Pure inputs--the type is fixed to address and it cannot be deserialized into a different type, even if it has a compatible BCS representation.

You now have updated memory locations of

Finally, transfer the remaining item to yourself. This is similar to the previous TransferObjects command. You are using the last Item by-value and the sender's address by-value. The item is moved because Item does not have copy , and the address is copied because address does have copy .

The final memory locations are

At the end of execution, the runtime checks the remaining values, which are the three inputs and the sender's address. The following summarizes the checks performed before effects are given:

After these checks are performed, generate the effects.

Transaction type

There are two parts of a PTB that are relevant to execution semantics. Other transaction information, such as the transaction sender or the gas limit, might be referenced but are out of scope. The structure of a PTB is:

Looking closer at the two main components:

Inputs and results are the two types of values you can use in transaction commands. Inputs are the values that are provided to the PTB, and results are the values that are produced by the PTB commands. The inputs are either objects or simple Move values, and the results are arbitrary Move values (including objects).

The inputs and results can be seen as populating an array of values. For inputs, there is a single array, but for results, there is an array for each individual transaction command, creating a 2D-array of result values. You can access these values by borrowing (mutably or immutably), by copying (if the type permits), or by moving (which takes the value out of the array without re-indexing).

Input arguments to a PTB are broadly categorized as either objects or pure values. The direct implementation of these arguments is often obscured by transaction builders or SDKs. This section describes information or data the Sui network needs when specifying the list of inputs, [Input] . Each Input is either an object, Input::Object(ObjectArg) , which contains the necessary metadata to specify to object being used, or a pure value, Input::Pure(PureArg) , which contains the bytes of the value.

For object inputs, the metadata needed differs depending on the type of ownership of the object . The data for the ObjectArg enum follows:

If the object is owned by an address (or it is immutable), then use ObjectArg::ImmOrOwnedObject(ObjectID, SequenceNumber, ObjectDigest) . The triple respectively specifies the object's ID, its sequence number (also known as its version), and the digest of the object's data.

If an object is shared, then use Object::SharedObject { id: ObjectID, initial_shared_version: SequenceNumber, mutable: bool } . Unlike ImmOrOwnedObject , a shared objects version and digest are determined by the network's consensus protocol. The initial_shared_version is the version of the object when it was first shared, which is used by consensus when it has not yet seen a transaction with that object. While all shared objects can be mutated, the mutable flag indicates whether the object is to be used mutably in this transaction. In the case where the mutable flag is set to false , the object is read-only, and the system can schedule other read-only transactions in parallel.

If the object is owned by another object, as in it was sent to an object's ID via the TransferObjects command or the sui::transfer::transfer function, then use ObjectArg::Receiving(ObjectID, SequenceNumber, ObjectDigest) . The object data is the same as for the ImmOrOwnedObject case.

For pure inputs, the only data provided is the BCS bytes, which are deserialized to construct Move values. Not all Move values can be constructed from BCS bytes. This means that even if the bytes match the expected layout for a given Move type, they cannot be deserialized into a value of that type unless the type is one of the types permitted for Pure values. The following types are allowed to be used with pure values:

Interestingly, the bytes are not validated until the type is specified in a command, for example in MoveCall or MakeMoveVec . This means that a given pure input could be used to instantiate Move values of several types. See the Arguments section for more details.

Each transaction command produces a (possibly empty) array of values. The type of the value can be any arbitrary Move type, so unlike inputs, the values are not limited to objects or pure values. The number of results generated and their types are specific to each transaction command. The specifics for each command can be found in the section for that command, but in summary:

Each command takes Argument s that specify the input or result being used. The usage (by-reference or by-value) is inferred based on the type of the argument and the expected argument of the command. First, examine the structure of the Argument enum.

Input(u16) is an input argument, where the u16 is the index of the input in the input vector. For example, given an input vector of [Object1, Object2, Pure1, Object3] , Object1 is accessed with Input(0) and Pure1 is accessed with Input(2) .

GasCoin is a special input argument representing the object for the SUI coin used to pay for gas. It is kept separate from the other inputs because the gas coin is always present in each transaction and has special restrictions (see below) not present for other inputs. Additionally, the gas coin being separate makes its usage explicit, which is helpful for sponsored transactions where the sponsor might not want the sender to use the gas coin for anything other than gas.

The gas coin cannot be taken by-value except with the TransferObjects command. If you need an owned version of the gas coin, you can first use SplitCoins to split off a single coin.

This limitation exists to make it easy for the remaining gas to be returned to the coin at the end of execution. In other words, if the gas coin was wrapped or deleted, then there would not be an obvious spot for the excess gas to be returned. See the Execution section for more details.

NestedResult(u16, u16) uses the value from a previous command. The first u16 is the index of the command in the command vector, and the second u16 is the index of the result in the result vector of that command. For example, given a command vector of [MoveCall1, MoveCall2, TransferObjects] where MoveCall2 has a result vector of [Value1, Value2] , Value1 would be accessed with NestedResult(1, 0) and Value2 would be accessed with NestedResult(1, 1) .

Result(u16) is a special form of NestedResult where Result(i) is roughly equivalent to NestedResult(i, 0) . Unlike NestedResult(i, 0) , Result(i) , however, this errors if the result array at index i is empty or has more than one value. The ultimate intention of Result is to allow accessing the entire result array, but that is not yet supported. So in its current state, NestedResult can be used instead of Result in all circumstances.

For the execution of PTBs, the input vector is populated by the input objects or pure value bytes. The transaction commands are then executed in order, and the results are stored in the result vector. Finally, the effects of the transaction are applied atomically. The following sections describe each aspect of execution in greater detail.

At the beginning of execution, the PTB runtime takes the already loaded input objects and loads them into the input array. The objects are already verified by the network, checking rules like existence and valid ownership. The pure value bytes are also loaded into the array but not validated until usage.

The most important thing to note at this stage is the effects on the gas coin. At the beginning of execution, the maximum gas budget (in terms of SUI) is withdrawn from the gas coin. Any unused gas is returned to the gas coin at the end of execution, even if the coin has changed owners.

Each transaction command is then executed in order. First, examine the rules around arguments, which are shared by all commands.

You can use each argument by-reference or by-value. The usage is based on the type of the argument and the type signature of the command.

The transaction fails if an argument is used in any form after being moved. There is no way to restore an argument to its position (its input or result index) after it is moved.

If an argument is copied but does not have the drop ability, then the last usage is inferred to be a move. As a result, if an argument has copy and does not have drop , the last usage must be by value. Otherwise, the transaction will fail because a value without drop has not been used.

The borrowing of arguments has other rules to ensure unique safe usage of an argument by reference. If an argument is:

Object inputs have the type of their object T as you might expect. However, for ObjectArg::Receiving inputs, the object type T is instead wrapped as sui::transfer::Receiving . This is because the object is not owned by the sender, but instead by another object. And to prove ownership with that parent object, you call the sui::transfer::receive function to remove the wrapper.

The GasCoin has special restrictions on being used by-value (moved). You can only use it by-value with the TransferObjects command.

Shared objects also have restrictions on being used by-value. These restrictions exist to ensure that at the end of the transaction the shared object is either still shared or deleted. A shared object cannot be unshared (having the owner changed) and it cannot be wrapped. A shared object:

Pure values are not type checked until their usage. When checking if a pure value has type T , it is checked whether T is a valid type for a pure value (see the previous list). If it is, the bytes are then validated. You can use a pure value with multiple types as long as the bytes are valid for each type. For example, you can use a string as an ASCII string std::ascii::String and as a UTF8 string std::string::String . However, after you mutably borrow the pure value, the type becomes fixed, and all future usages must be with that type.

The command has the form TransferObjects(ObjectArgs, AddressArg) where ObjectArgs is a vector of objects and AddressArg is the address the objects are sent to.

The command has the form SplitCoins(CoinArg, AmountArgs) where CoinArg is the coin being split and AmountArgs is a vector of amounts to split off.

The command has the form MergeCoins(CoinArg, ToMergeArgs) where the CoinArg is the target coin in which the ToMergeArgs coins are merged into. In other words, you merge multiple coins (ToMergeArgs) into a single coin (CoinArg).

The command has the form MakeMoveVec(VecTypeOption, Args) where VecTypeOption is an optional argument specifying the type of the elements in the vector being constructed and Args is a vector of arguments to be used as elements in the vector.

This command has the form MoveCall(Package, Module, Function, TypeArgs, Args) where Package::Module::Function combine to specify the Move function being called, TypeArgs is a vector of type arguments to that function, and Args is a vector of arguments for the Move function.

The command has the form Publish(ModuleBytes, TransitiveDependencies) where ModuleBytes are the bytes of the module being published and TransitiveDependencies is a vector of package Object ID dependencies to link against.

When the transaction is signed, the network verifies that the ModuleBytes are not empty. The module bytes ModuleBytes: [[u8]] contain the bytes of the modules being published with each [u8] element is a module.

The transitive dependencies TransitiveDependencies: [ObjectID] are the Object IDs of the packages that the new package depends on. While each module indicates the packages used as dependencies, the transitive object IDs must be provided to select the version of those packages. In other words, these object IDs are used to select the version of the packages marked as dependencies in the modules.

After the modules in the package are verified, the init function of each module is called in same order as the module byte vector ModuleBytes .

The command produces a single result of type sui::package::UpgradeCap , which is the upgrade capability for the newly published package.

The command has the form Upgrade(ModuleBytes, TransitiveDependencies, Package, UpgradeTicket) , where the Package indicates the object ID of the package being upgraded. The ModuleBytes and TransitiveDependencies work similarly as the Publish command.

For details on the ModuleBytes and TransitiveDependencies , see the Publish command . Note though, that no init functions are called for the upgraded modules.

The Package: ObjectID is the Object ID of the package being upgraded. The package must exist and be the latest version.

The UpgradeTicket: sui::package::UpgradeTicket is the upgrade ticket for the package being upgraded and is generated from the sui::package::UpgradeCap . The ticket is taken by value (moved).

The command produces a single result type sui::package::UpgradeReceipt which provides proof for that upgrade. For more details on upgrades, see Upgrading Packages .

At the end of execution, the remaining values are checked and effects for the transaction are calculated.

For inputs, the following checks are performed:

For results, the following checks are performed:

Any remaining SUI deducted from the gas coin at the beginning of execution is returned to the coin, even if the owner has changed.

In other words, the maximum possible gas is deducted at the beginning of execution, and then the unused gas is returned at the end of execution (all in SUI). Because you can take the gas coin only by-value with TransferObjects , it will not have been wrapped or deleted.

The total effects (which contain the created, mutated, and deleted objects) are then passed out of the execution layer and are applied by the Sui network.

Let's walk through an example of a PTB's execution. While this example is not exhaustive in demonstrating all the rules, it does show the general flow of execution.

Suppose you want to buy two items from a marketplace costing 100 MIST . You keep one for yourself, and then send the object and the remaining coin to a friend at address 0x808 . You can do that all in one PTB:

The inputs include the friend's address, the marketplace object, and the value for the coin split. For the commands, split off the coin, call the market place function, send the gas coin and one object, grab your address (via sui::tx_context::sender), and then send the remaining object to yourself. For simplicity, the documentation refers to the package names by name, but note that in practice they are referenced by the package's Object ID.

To walk through this, first look at the memory locations, for the gas object, inputs, and results

Here you have two objects loaded so far, the gas coin with a value of 1_000_000u64 and the marketplace object of type some_package::some_marketplace::Marketplace (these names and representations are shortened for simplicity going forward). The pure arguments are not loaded, and are present as BCS bytes.

Note that while gas is deducted at each command, that aspect of execution is not demonstrated in detail.

Before execution, remove the gas budget from the gas coin. Assume a gas budget of 500_000 so the gas coin now has a value of 500_000u64 .

Now you can execute the commands.

The first command SplitCoins(GasCoin, [Input(2)]) accesses the gas coin by mutable reference and loads the pure argument at Input(2) as a u64 value of 100u64 . Because u64 has the copy ability, you do not move the Pure input at Input(2) . Instead, the bytes are copied out.

For the result, a new coin object is made.

This gives us updated memory locations of

Now the command, MoveCall("some_package", "some_marketplace", "buy_two", [], [Input(1), NestedResult(0, 0)]) . Call the function some_package::some_marketplace::buy_two with the arguments Input(1) and NestedResult(0, 0) . To determine how they are used, you need to look at the function's signature. For this example, assume the signature is

where Item is the type of the two objects being sold.

Since the marketplace parameter has type &mut Marketplace , use Input(1) by mutable reference. Assume some modifications are being made into the value of the Marketplace object. However, the coin parameter has type Coin , so use NestedResult(0, 0) by value. The TxContext input is automatically provided by the runtime.

This gives updated memory locations, where _ indicates the object has been moved.

Assume that buy_two deletes its Coin object argument and transfers the Balance into the Marketplace object.

TransferObjects([GasCoin, NestedResult(1, 0)], Input(0)) transfers the gas coin and first item to the address at Input(0) . All inputs are by value, and the objects do not have copy so they are moved. While no results are given, the ownership of the objects is changed. This cannot be seen in the memory locations, but rather in the transaction effects.

You now have updated memory locations of

Make another Move call, this one to sui::tx_context::sender with the signature

While you could have just passed in the sender's address as a Pure input, this example demonstrates calling some of the additional utility of PTBs; while this function is not an entry function, you can call the public function, too, because you can provide all of the arguments. In this case, the only argument, the TxContext , is provided by the runtime. The result of the function is the sender's address. Note that this value is not treated like the Pure inputs--the type is fixed to address and it cannot be deserialized into a different type, even if it has a compatible BCS representation.

You now have updated memory locations of

Finally, transfer the remaining item to yourself. This is similar to the previous TransferObjects command. You are using the last Item by-value and the sender's address by-value. The item is moved because Item does not have copy , and the address is copied because address does have copy .

The final memory locations are

At the end of execution, the runtime checks the remaining values, which are the three inputs and the sender's address. The following summarizes the checks performed before effects are given:

After these checks are performed, generate the effects.

Inputs and results

Inputs and results are the two types of values you can use in transaction commands. Inputs are the values that are provided to the PTB, and results are the values that are produced by the PTB commands. The inputs are either objects or simple Move values, and the results are arbitrary Move values (including objects).

The inputs and results can be seen as populating an array of values. For inputs, there is a single array, but for results, there is an array for each individual transaction command, creating a 2D-array of result values. You can access these values by borrowing (mutably or immutably), by copying (if the type permits), or by moving (which takes the value out of the array without re-indexing).

Input arguments to a PTB are broadly categorized as either objects or pure values. The direct implementation of these arguments is often obscured by transaction builders or SDKs. This section describes information or data the Sui network needs when specifying the list of inputs, [Input] . Each Input is either an object, Input::Object(ObjectArg) , which contains the necessary metadata to specify to object being used, or a pure value, Input::Pure(PureArg) , which contains the bytes of the value.

For object inputs, the metadata needed differs depending on the type of ownership of the object . The data for the ObjectArg enum follows:

If the object is owned by an address (or it is immutable), then use ObjectArg::ImmOrOwnedObject(ObjectID, SequenceNumber, ObjectDigest) . The triple respectively specifies the object's ID, its sequence number (also known as its version), and the digest of the object's data.

If an object is shared, then use Object::SharedObject { id: ObjectID, initial_shared_version: SequenceNumber, mutable: bool } . Unlike ImmOrOwnedObject , a shared objects version and digest are determined by the network's consensus protocol. The initial_shared_version is the version of the object when it was first shared, which is used by consensus when it has not yet seen a transaction with that object. While all shared objects can be mutated, the mutable flag indicates whether the object is to be used mutably in this transaction. In the case where the mutable flag is set to false , the object is read-only, and the system can schedule other read-only transactions in parallel.

If the object is owned by another object, as in it was sent to an object's ID via the TransferObjects command or the sui::transfer::transfer function, then use ObjectArg::Receiving(ObjectID, SequenceNumber, ObjectDigest) . The object data is the same as for the ImmOrOwnedObject case.

For pure inputs, the only data provided is the BCS bytes, which are deserialized to construct Move values. Not all Move values can be constructed from BCS bytes. This means that even if the bytes match the expected layout for a given Move type, they cannot be deserialized into a value of that type unless the type is one of the types permitted for Pure values. The following types are allowed to be used with pure values:

Interestingly, the bytes are not validated until the type is specified in a command, for example in MoveCall or MakeMoveVec . This means that a given pure input could be used to instantiate Move values of several types. See the Arguments section for more details.

Each transaction command produces a (possibly empty) array of values. The type of the value can be any arbitrary Move type, so unlike inputs, the values are not limited to objects or pure values. The number of results generated and their types are specific to each transaction command. The specifics for each command can be found in the section for that command, but in summary:

Each command takes Argument s that specify the input or result being used. The usage (by-reference or by-value) is inferred based on the type of the argument and the expected argument of the command. First, examine the structure of the Argument enum.

Input(u16) is an input argument, where the u16 is the index of the input in the input vector. For example, given an input vector of

[Object1, Object2, Pure1, Object3] , Object1 is accessed with Input(0) and Pure1 is accessed with Input(2) .

GasCoin is a special input argument representing the object for the SUI coin used to pay for gas. It is kept separate from the other inputs because the gas coin is always present in each transaction and has special restrictions (see below) not present for other inputs. Additionally, the gas coin being separate makes its usage explicit, which is helpful for sponsored transactions where the sponsor might not want the sender to use the gas coin for anything other than gas.

The gas coin cannot be taken by-value except with the TransferObjects command. If you need an owned version of the gas coin, you can first use SplitCoins to split off a single coin.

This limitation exists to make it easy for the remaining gas to be returned to the coin at the end of execution. In other words, if the gas coin was wrapped or deleted, then there would not be an obvious spot for the excess gas to be returned. See the Execution section for more details.

NestedResult(u16, u16) uses the value from a previous command. The first u16 is the index of the command in the command vector, and the second u16 is the index of the result in the result vector of that command. For example, given a command vector of [MoveCall1, MoveCall2, TransferObjects] where MoveCall2 has a result vector of [Value1, Value2] , Value1 would be accessed with NestedResult(1, 0) and Value2 would be accessed with NestedResult(1, 1) .

Result(u16) is a special form of NestedResult where Result(i) is roughly equivalent to NestedResult(i, 0) . Unlike NestedResult(i, 0) , Result(i) , however, this errors if the result array at index i is empty or has more than one value. The ultimate intention of Result is to allow accessing the entire result array, but that is not yet supported. So in its current state, NestedResult can be used instead of Result in all circumstances.

For the execution of PTBs, the input vector is populated by the input objects or pure value bytes. The transaction commands are then executed in order, and the results are stored in the result vector. Finally, the effects of the transaction are applied atomically. The following sections describe each aspect of execution in greater detail.

At the beginning of execution, the PTB runtime takes the already loaded input objects and loads them into the input array. The objects are already verified by the network, checking rules like existence and valid ownership. The pure value bytes are also loaded into the array but not validated until usage.

The most important thing to note at this stage is the effects on the gas coin. At the beginning of execution, the maximum gas budget (in terms of SUI) is withdrawn from the gas coin. Any unused gas is returned to the gas coin at the end of execution, even if the coin has changed owners.

Each transaction command is then executed in order. First, examine the rules around arguments, which are shared by all commands.

You can use each argument by-reference or by-value. The usage is based on the type of the argument and the type signature of the command.

The transaction fails if an argument is used in any form after being moved. There is no way to restore an argument to its position (its input or result index) after it is moved.

If an argument is copied but does not have the drop ability, then the last usage is inferred to be a move. As a result, if an argument has copy and does not have drop , the last usage must be by value. Otherwise, the transaction will fail because a value without drop has not been used.

The borrowing of arguments has other rules to ensure unique safe usage of an argument by reference. If an argument is:

Object inputs have the type of their object T as you might expect. However, for ObjectArg::Receiving inputs, the object type T is instead wrapped as sui::transfer::Receiving . This is because the object is not owned by the sender, but instead by another object. And to prove ownership with that parent object, you call the sui::transfer::receive function to remove the wrapper.

The GasCoin has special restrictions on being used by-value (moved). You can only use it by-value with the TransferObjects command.

Shared objects also have restrictions on being used by-value. These restrictions exist to ensure that at the end of the transaction the shared object is either still shared or deleted. A shared object cannot be unshared (having the owner changed) and it cannot be wrapped. A shared object:

Pure values are not type checked until their usage. When checking if a pure value has type T , it is checked whether T is a valid type for a pure value (see the previous list). If it is, the bytes are then validated. You can use a pure value with multiple types as long as the bytes are valid for each type. For example, you can use a string as an ASCII string std::ascii::String and as a UTF8 string std::string::String . However, after you mutably borrow the pure value, the type becomes fixed, and all future usages must be with that type.

The command has the form TransferObjects(ObjectArgs, AddressArg) where ObjectArgs is a vector of objects and AddressArg is the address the objects are sent to.

The command has the form SplitCoins(CoinArg, AmountArgs) where CoinArg is the coin being split and AmountArgs is a vector of amounts to split off.

The command has the form MergeCoins(CoinArg, ToMergeArgs) where the CoinArg is the target coin in which the ToMergeArgs coins are merged into. In other words, you merge multiple coins (ToMergeArgs) into a single coin (CoinArg).

The command has the form MakeMoveVec(VecTypeOption, Args) where VecTypeOption is an optional argument specifying the type of the elements in the vector being constructed and Args is a vector of arguments to be used as elements in the vector.

This command has the form MoveCall(Package, Module, Function, TypeArgs, Args) where Package::Module::Function combine to specify the Move function being called, TypeArgs is a vector of type arguments to that function, and Args is a vector of arguments for the Move function.

The command has the form Publish(ModuleBytes, TransitiveDependencies) where ModuleBytes are the bytes of the module being published and TransitiveDependencies is a vector of package Object ID dependencies to link against.

When the transaction is signed, the network verifies that the ModuleBytes are not empty. The module bytes ModuleBytes: [[u8]] contain the bytes of the modules being published with each [u8] element is a module.

The transitive dependencies TransitiveDependencies: [ObjectID] are the Object IDs of the packages that the new package depends on. While each module indicates the packages used as dependencies, the transitive object IDs must be provided to select the version of those packages. In other words, these object IDs are used to select the version of the packages marked as dependencies in the modules.

After the modules in the package are verified, the init function of each module is called in same order as the module byte vector ModuleBytes .

The command produces a single result of type sui::package::UpgradeCap , which is the upgrade capability for the newly published package.

The command has the form Upgrade(ModuleBytes, TransitiveDependencies, Package, UpgradeTicket) , where the Package indicates the object ID of the package being upgraded. The ModuleBytes and TransitiveDependencies work similarly as the Publish command.

For details on the ModuleBytes and TransitiveDependencies , see the Publish command . Note though, that no init functions are called for the upgraded modules.

The Package: ObjectID is the Object ID of the package being upgraded. The package must exist and be the latest version.

The UpgradeTicket: sui::package::UpgradeTicket is the upgrade ticket for the package being upgraded and is generated from the sui::package::UpgradeCap . The ticket is taken by value (moved).

The command produces a single result type sui::package::UpgradeReceipt which provides proof for that upgrade. For more details on upgrades, see Upgrading Packages .

At the end of execution, the remaining values are checked and effects for the transaction are calculated.

For inputs, the following checks are performed:

For results, the following checks are performed:

Any remaining SUI deducted from the gas coin at the beginning of execution is returned to the coin, even if the owner has changed. In other words, the maximum possible gas is deducted at the beginning of execution, and then the unused gas is returned at the end of execution (all in SUI). Because you can take the gas coin only by-value with TransferObjects , it will not have been wrapped or deleted.

The total effects (which contain the created, mutated, and deleted objects) are then passed out of the execution layer and are applied by the Sui network.

Let's walk through an example of a PTB's execution. While this example is not exhaustive in demonstrating all the rules, it does show the general flow of execution.

Suppose you want to buy two items from a marketplace costing 100 MIST . You keep one for yourself, and then send the object and the remaining coin to a friend at address 0x808 . You can do that all in one PTB:

The inputs include the friend's address, the marketplace object, and the value for the coin split. For the commands, split off the coin, call the market place function, send the gas coin and one object, grab your address (via sui::tx_context::sender), and then send the remaining object to yourself. For simplicity, the documentation refers to the package names by name, but note that in practice they are referenced by the package's Object ID.

To walk through this, first look at the memory locations, for the gas object, inputs, and results

Here you have two objects loaded so far, the gas coin with a value of 1_000_000u64 and the marketplace object of type some_package::some_marketplace::Marketplace (these names and representations are shortened for simplicity going forward). The pure arguments are not loaded, and are present as BCS bytes.

Note that while gas is deducted at each command, that aspect of execution is not demonstrated in detail.

Before execution, remove the gas budget from the gas coin. Assume a gas budget of 500_000 so the gas coin now has a value of 500_000u64 .

Now you can execute the commands.

The first command SplitCoins(GasCoin, [Input(2)]) accesses the gas coin by mutable reference and loads the pure argument at Input(2) as a u64 value of 100u64 . Because u64 has the copy ability, you do not move the Pure input at Input(2) . Instead, the bytes are copied out.

For the result, a new coin object is made.

This gives us updated memory locations of

Now the command, MoveCall("some_package", "some_marketplace", "buy_two", [], [Input(1), NestedResult(0, 0)]) . Call the function some_package::some_marketplace::buy_two with the arguments Input(1) and NestedResult(0, 0) . To determine how they are used, you need to look at the function's signature. For this example, assume the signature is

where Item is the type of the two objects being sold.

Since the marketplace parameter has type &mut Marketplace , use Input(1) by mutable reference. Assume some modifications are being made into the value of the Marketplace object. However, the coin parameter has type Coin , so use NestedResult(0, 0) by value. The TxContext input is automatically provided by the runtime.

This gives updated memory locations, where _ indicates the object has been moved.

Assume that buy_two deletes its Coin object argument and transfers the Balance into the Marketplace object.

TransferObjects([GasCoin, NestedResult(1, 0)], Input(0)) transfers the gas coin and first item to the address at Input(0) . All inputs are by value, and the objects do not have copy so they are moved. While no results are given, the ownership of the objects is changed. This cannot be seen in the memory locations, but rather in the transaction effects.

You now have updated memory locations of

Make another Move call, this one to sui::tx_context::sender with the signature

While you could have just passed in the sender's address as a Pure input, this example demonstrates calling some of the additional utility of PTBs; while this function is not an entry function, you can call the public function, too, because you can provide all of the arguments. In this case, the only argument, the TxContext , is provided by the runtime. The result of the function is the sender's address. Note that this value is not treated like the Pure inputs--the type is fixed to address and it cannot be deserialized into a different type, even if it has a compatible BCS representation.

You now have updated memory locations of

Finally, transfer the remaining item to yourself. This is similar to the previous TransferObjects command. You are using the last Item by-value and the sender's address by-value. The item is moved because Item does not have copy , and the address is copied because address does have copy .

The final memory locations are

At the end of execution, the runtime checks the remaining values, which are the three inputs and the sender's address. The following summarizes the checks performed before effects are given:

After these checks are performed, generate the effects.

Execution

For the execution of PTBs, the input vector is populated by the input objects or pure value bytes. The transaction commands are then executed in order, and the results are stored in the result vector. Finally, the effects of the transaction are applied atomically. The following sections describe each aspect of execution in greater detail.

At the beginning of execution, the PTB runtime takes the already loaded input objects and loads them into the input array. The objects are already verified by the network, checking rules like existence and valid ownership. The pure value bytes are also loaded into the array but not validated until usage.

The most important thing to note at this stage is the effects on the gas coin. At the beginning of execution, the maximum gas budget (in terms of SUI) is withdrawn from the gas coin. Any unused gas is returned to the gas coin at the end of execution, even if the coin has changed owners.

Each transaction command is then executed in order. First, examine the rules around arguments, which are shared by all commands.

You can use each argument by-reference or by-value. The usage is based on the type of the argument and the type signature of the command.

The transaction fails if an argument is used in any form after being moved. There is no way to restore an argument to its position (its input or result index) after it is moved.

If an argument is copied but does not have the drop ability, then the last usage is inferred to be a move. As a result, if an argument has copy and does not have drop , the last usage must be by value. Otherwise, the transaction will fail because a value without drop has not been used.

The borrowing of arguments has other rules to ensure unique safe usage of an argument by reference. If an argument is:

Object inputs have the type of their object T as you might expect. However, for ObjectArg::Receiving inputs, the object type T is instead wrapped as sui::transfer::Receiving . This is because the object is not owned by the sender, but instead by another object. And to prove ownership with that parent object, you call the sui::transfer::receive function to remove the wrapper.

The GasCoin has special restrictions on being used by-value (moved). You can only use it by-value with the TransferObjects command.

Shared objects also have restrictions on being used by-value. These restrictions exist to ensure that at the end of the transaction the shared object is either still shared or deleted. A shared object cannot be unshared (having the owner changed) and it cannot be wrapped. A shared object:

Pure values are not type checked until their usage. When checking if a pure value has type T , it is checked whether T is a valid type for a pure value (see the previous list). If it is, the bytes are then validated. You can use a pure value with multiple types as long as the bytes are valid for each type. For example, you can use a string as an ASCII string std::ascii::String and as a UTF8 string std::string::String . However, after you mutably borrow the pure value, the type becomes fixed, and all future usages must be with that type.

The command has the form TransferObjects(ObjectArgs, AddressArg) where ObjectArgs is a vector of objects and AddressArg is the address the objects are sent to.

The command has the form SplitCoins(CoinArg, AmountArgs) where CoinArg is the coin being split and AmountArgs is a vector of amounts to split off.

The command has the form MergeCoins(CoinArg, ToMergeArgs) where the CoinArg is the target coin in which the ToMergeArgs coins are merged into. In other words, you merge multiple coins (ToMergeArgs) into a single coin (CoinArg).

The command has the form MakeMoveVec(VecTypeOption, Args) where VecTypeOption is an optional argument specifying the type of the elements in the vector being constructed and Args is a vector of arguments to be used as elements in the vector.

This command has the form MoveCall(Package, Module, Function, TypeArgs, Args) where Package::Module::Function combine to specify the Move function being called, TypeArgs is a vector of type arguments to that function, and Args is a vector of arguments for the Move function.

The command has the form Publish(ModuleBytes, TransitiveDependencies) where ModuleBytes are the bytes of the module being published and TransitiveDependencies is a vector of package Object ID dependencies to link against.

When the transaction is signed, the network verifies that the ModuleBytes are not empty. The module bytes ModuleBytes: [[u8]] contain the bytes of the modules being published with each [u8] element is a module.

The transitive dependencies TransitiveDependencies: [ObjectID] are the Object IDs of the packages that the new package depends on. While each module indicates the packages used as dependencies, the transitive object IDs must be provided to select the version of those packages. In other words, these object IDs are used to select the version of the packages marked as dependencies in the modules.

After the modules in the package are verified, the init function of each module is called in same order as the module byte vector ModuleBytes .

The command produces a single result of type sui::package::UpgradeCap , which is the upgrade capability for the newly published package.

The command has the form Upgrade(ModuleBytes, TransitiveDependencies, Package, UpgradeTicket) , where the Package indicates the object ID of the package being upgraded. The ModuleBytes and TransitiveDependencies work similarly as the Publish command.

For details on the ModuleBytes and TransitiveDependencies , see the Publish command . Note though, that no init functions are called for the upgraded modules.

The Package: ObjectID is the Object ID of the package being upgraded. The package must exist and be the latest version.

The UpgradeTicket: sui::package::UpgradeTicket is the upgrade ticket for the package being upgraded and is generated from the sui::package::UpgradeCap . The ticket is taken by value (moved).

The command produces a single result type sui::package::UpgradeReceipt which provides proof for that upgrade. For more details on upgrades, see Upgrading Packages .

At the end of execution, the remaining values are checked and effects for the transaction are calculated.

For inputs, the following checks are performed:

For results, the following checks are performed:

Any remaining SUI deducted from the gas coin at the beginning of execution is returned to the coin, even if the owner has changed. In other words, the maximum possible gas is deducted at the beginning of execution, and then the unused gas is returned at the end of execution (all in SUI). Because you can take the gas coin only by-value with TransferObjects , it will not have been wrapped or deleted.

The total effects (which contain the created, mutated, and deleted objects) are then passed out of the execution layer and are applied by the Sui network.

Let's walk through an example of a PTB's execution. While this example is not exhaustive in demonstrating all the rules, it does show the general flow of execution.

Suppose you want to buy two items from a marketplace costing 100 MIST . You keep one for yourself, and then send the object and the remaining coin to a friend at address 0x808 . You can do that all in one PTB:

The inputs include the friend's address, the marketplace object, and the value for the coin split. For the commands, split off the coin, call the market place function, send the gas coin and one object, grab your address (via sui::tx_context::sender), and then send the remaining object to yourself. For simplicity, the documentation refers to the package names by name, but note that in practice they are referenced by the package's Object ID.

To walk through this, first look at the memory locations, for the gas object, inputs, and results

Here you have two objects loaded so far, the gas coin with a value of 1_000_000u64 and the marketplace object of type some_package::some_marketplace::Marketplace (these names and representations are shortened for simplicity going forward). The pure arguments are not loaded, and are present as BCS bytes.

Note that while gas is deducted at each command, that aspect of execution is not demonstrated in detail.

Before execution, remove the gas budget from the gas coin. Assume a gas budget of 500_000 so the gas coin now has a value of 500_000u64 .

Now you can execute the commands.

The first command SplitCoins(GasCoin, [Input(2)]) accesses the gas coin by mutable reference and loads the pure argument at Input(2) as a u64 value of 100u64 . Because u64 has the copy ability, you do not move the Pure input at Input(2) . Instead, the bytes are copied out.

For the result, a new coin object is made.

This gives us updated memory locations of

Now the command, MoveCall("some_package", "some_marketplace", "buy_two", [], [Input(1), NestedResult(0, 0)]) . Call the function some_package::some_marketplace::buy_two with the arguments Input(1) and NestedResult(0, 0) . To determine how they are used, you need to look at the function's signature. For this example, assume the signature is

where Item is the type of the two objects being sold.

Since the marketplace parameter has type &mut Marketplace , use Input(1) by mutable reference. Assume some modifications are being made into the value of the Marketplace object. However, the coin parameter has type Coin , so use NestedResult(0, 0) by value. The TxContext input is automatically provided by the runtime.

This gives updated memory locations, where _ indicates the object has been moved.

Assume that buy_two deletes its Coin object argument and transfers the Balance into the Marketplace object.

TransferObjects([GasCoin, NestedResult(1, 0)], Input(0)) transfers the gas coin and first item to the address at Input(0) . All inputs are by value, and the objects do not have copy so they are moved. While no results are given, the ownership of the objects is changed. This cannot be seen in the memory locations, but rather in the transaction effects.

You now have updated memory locations of

Make another Move call, this one to sui::tx_context::sender with the signature

While you could have just passed in the sender's address as a Pure input, this example demonstrates calling some of the additional utility of PTBs; while this function is not an entry function, you can call the public function, too, because you can provide all of the arguments. In this case, the only argument, the TxContext , is provided by the runtime. The result of the function is the sender's address. Note that this value is not treated like the Pure inputs--the type is fixed to address and it cannot be deserialized into a different type, even if it has a compatible BCS representation.

You now have updated memory locations of

Finally, transfer the remaining item to yourself. This is similar to the previous TransferObjects command. You are using the last Item by-value and the sender's address by-value. The item is moved because Item does not have copy , and the address is copied because address does have copy .

The final memory locations are

At the end of execution, the runtime checks the remaining values, which are the three inputs and the sender's address. The following summarizes the checks performed before effects are given:

After these checks are performed, generate the effects.

Example

Let's walk through an example of a PTB's execution. While this example is not exhaustive in demonstrating all the rules, it does show the general flow of execution.

Suppose you want to buy two items from a marketplace costing 100 MIST . You keep one for yourself, and then send the object and the remaining coin to a friend at address 0x808 . You can do that all in one PTB:

The inputs include the friend's address, the marketplace object, and the value for the coin split. For the commands, split off the coin, call the market place function, send the gas coin and one object, grab your address (via sui::tx_context::sender), and then send the remaining object to yourself. For simplicity, the documentation refers to the package names by name, but note that in practice they are referenced by the package's Object ID.

To walk through this, first look at the memory locations, for the gas object, inputs, and results

Here you have two objects loaded so far, the gas coin with a value of 1_000_000u64 and the marketplace object of type some_package::some_marketplace::Marketplace (these names and representations are shortened for simplicity going forward). The pure arguments are not loaded, and are present as BCS bytes.

Note that while gas is deducted at each command, that aspect of execution is not demonstrated in detail.

Before execution, remove the gas budget from the gas coin. Assume a gas budget of 500_000 so the gas coin now has a value of 500_000u64 .

Now you can execute the commands.

The first command SplitCoins(GasCoin, [Input(2)]) accesses the gas coin by mutable reference and loads the pure argument at Input(2) as a u64 value of 100u64 . Because u64 has the copy ability, you do not move the Pure input at Input(2) . Instead, the bytes are copied out.

For the result, a new coin object is made.

This gives us updated memory locations of

Now the command, MoveCall("some_package", "some_marketplace", "buy_two", [], [Input(1), NestedResult(0, 0)]) . Call the function some_package::some_marketplace::buy_two with the arguments Input(1) and NestedResult(0, 0) . To determine how they are used, you need to look at the function's signature. For this example, assume the signature is

where Item is the type of the two objects being sold.

Since the marketplace parameter has type &mut Marketplace , use Input(1) by mutable reference. Assume some modifications are being made into the value of the Marketplace object. However, the coin parameter has type Coin , so use NestedResult(0, 0) by value. The TxContext input is automatically provided by the runtime.

This gives updated memory locations, where _ indicates the object has been moved.

Assume that buy_two deletes its Coin object argument and transfers the Balance into the Marketplace object.

TransferObjects([GasCoin, NestedResult(1, 0)], Input(0)) transfers the gas coin and first item to the address at Input(0) . All inputs are by value, and the objects do not have copy so they are moved. While no results are given, the ownership of the objects is changed. This cannot be seen in the memory locations, but rather in the transaction effects.

You now have updated memory locations of

Make another Move call, this one to sui::tx_context::sender with the signature

While you could have just passed in the sender's address as a Pure input, this example demonstrates calling some of the additional utility of PTBs; while this function is not an entry function, you can call the public function, too, because you can provide all of the arguments. In this case, the only argument, the TxContext , is provided by the runtime. The result of the function is the sender's address. Note that this value is not treated like the Pure inputs--the type is fixed to address and it cannot be deserialized into a different type, even if it has a compatible BCS representation.

You now have updated memory locations of

Finally, transfer the remaining item to yourself. This is similar to the previous TransferObjects command. You are using the last Item by-value and the sender's address by-value. The item is moved because Item does not have copy , and the address is copied because address does have copy .

The final memory locations are

At the end of execution, the runtime checks the remaining values, which are the three inputs and the sender's address. The following summarizes the checks performed before effects are given:

After these checks are performed, generate the effects.