The Move Book

Generics are a way to define a type or function that can work with any type. This is useful when you want to write a function which can be used with different types, or when you want to define a type that can hold any other type. Generics are the foundation of many advanced features in Move including collections, abstract implementations, and more.

In this chapter we already mentioned the vector type, which is a generic type that can hold any other type. Another example of a generic type in the standard library is the Option type, which is used to represent a value that may or may not be present.

To define a generic type or function, a type signature needs to have a list of generic parameters enclosed in angle brackets (< and >). The generic parameters are separated by commas.

In the example above, Container is a generic type with a single type parameter T , the value field of the container stores the T . The new function is a generic function with a single type parameter T , and it returns a Container with the given value. Generic types must be initialized with a concrete type, and generic functions must be called with a concrete type, although in some cases the Move compiler can infer the correct type.

In the test function test_generic , we demonstrate three equivalent ways to create a new Container with a u8 value. Because numeric constants have ambiguous types, we must specify the type of the number literal somewhere (in the type of the container, the parameter to new , or the number literal itself); once we specify one of these the compiler can infer the others.

You can define a type or function with multiple type parameters. The type parameters are separated by commas.

In the example above, Pair is a generic type with two type parameters T and U , and the new_pair function is a generic function with two type parameters T and U . The function returns a Pair with the given values. The order of the type parameters is important, and should match the order of the type parameters in the type signature.

If we added another instance where we swapped type parameters in the new_pair function, and tried to compare two types, we'd see that the type signatures are different, and cannot be compared.

Since the types for pair1 and pair2 are different, the comparison pair1 == pair2 will not compile.

In the examples above we focused on instantiating generic types and calling generic functions to create instances of these types. However, the real power of generics lies in their ability to define shared behavior for the base, generic type, and then use it independently of the concrete types. This is especially useful when working with collections, abstract implementations, and other advanced features in Move.

In the example above, User is a generic type with a single type parameter T , with shared fields name , age , and the generic metadata field, which can store any type. No matter what metadata is, all instances of User will contain the same fields and methods.

In some cases, you may want to define a generic type with a type parameter that is not used in the fields or methods of the type. This is called a phantom type parameter . Phantom type parameters are useful when you want to define a type that can hold any other type, but you want to enforce some constraints on the type parameter.

The Coin type here does not contain any fields or methods that use the type parameter T . It is used to differentiate between different types of coins, and to enforce some constraints on the type parameter T .

In the example above, we demonstrate how to create two different instances of Coin with different phantom type parameters USD and EUR . The type parameter T is not used in the fields or methods of the Coin type, but it is used to differentiate between different types of coins. This helps ensure that the USD and EUR coins are not mistakenly mixed up.

Type parameters can be constrained to have certain abilities. This is useful when you need the inner type to allow certain behaviors, such as copy or drop . The syntax for constraining a type parameter is T: + .

The Move Compiler will enforce that the type parameter T has the specified abilities. If the type parameter does not have the specified abilities, the code will not compile.

In the Standard Library

In this chapter we already mentioned the vector type, which is a generic type that can hold any other type. Another example of a generic type in the standard library is the Option type, which is used to represent a value that may or may not be present.

To define a generic type or function, a type signature needs to have a list of generic parameters enclosed in angle brackets (< and >

). The generic parameters are separated by commas.

``bash /// Container for any typeT`. public struct Container has drop { value: T, }

/// Function that creates a new `Container` with a generic value `T`. public fun new(value: T): Container { Container { value } } ```

In the example above, Container is a generic type with a single type parameter T , the value field of the container stores the T . The new function is a generic function with a single type parameter T , and it returns a Container with the given value. Generic types must be initialized with a concrete type, and generic functions must be called with a concrete type, although in some cases the Move compiler can infer the correct type.

```bash

# [test]

fun test_container() { // these three lines are equivalent let container: Container = new(10); // type inference let container = new(10); // create a new `Container` with a `u8` value let container = new(10u8);

```
assert!(container.value == 10, 0x0);

// Value can be ignored only if it has the `drop` ability.
let Container { value: _ } = container;
```

} ```

In the test function test_generic , we demonstrate three equivalent ways to create a new Container with a u8 value. Because numeric constants have ambiguous types, we must specify the type of the number literal somewhere (in the type of the container, the parameter to new , or the number literal itself); once we specify one of these the compiler can infer the others.

You can define a type or function with multiple type parameters. The type parameters are separated by commas.

``bash /// A pair of values of any typeTandU`. public struct Pair { first: T, second: U, }

/// Function that creates a new `Pair` with two generic values `T` and `U`. public fun new_pair(first: T, second: U): Pair { Pair { first, second } } ```

In the example above, Pair is a generic type with two type parameters T and U , and the new_pair function is a generic function with two type parameters T and U . The function returns a Pair with the given values. The order of the type parameters is important, and should match the order of the type parameters in the type signature.

```bash

# [test]

fun test_generic() { // these three lines are equivalent let pair_1: Pair = new_pair(10, true); // type inference let pair_2 = new_pair(10, true); // create a new `Pair` with a `u8` and `bool` values let pair_3 = new_pair(10u8, true);

```
assert!(pair_1.first == 10, 0x0);
assert!(pair_1.second, 0x0);

// Unpacking is identical.
let Pair { first: _, second: _ } = pair_1;
let Pair { first: _, second: _ } = pair_2;
let Pair { first: _, second: _ } = pair_3;
```

} ```

If we added another instance where we swapped type parameters in the new_pair function, and tried to compare two types, we'd see that the type signatures are different, and cannot be compared.

```bash

# [test]
```

fun test_swap_type_params() { let pair1: Pair = new_pair(10u8, true); let pair2: Pair = new_pair(true, 10u8);

```
// this line will not compile
// assert!(pair1 == pair2, 0x0);

let Pair { first: pf1, second: ps1 } = pair1; // first1: u8, second1: bool
let Pair { first: pf2, second: ps2 } = pair2; // first2: bool, second2: u8

assert!(pf1 == ps2); // 10 == 10
assert!(ps1 == pf2); // true == true
```

} ```

Since the types for pair1 and pair2 are different, the comparison pair1 == pair2 will not compile.

In the examples above we focused on instantiating generic types and calling generic functions to create instances of these types. However, the real power of generics lies in their ability to define shared behavior for the base, generic type, and then use it independently of the concrete types. This is especially useful when working with collections, abstract implementations, and other advanced features in Move.

```
bash /// A user record with name, age, and some generic metadata public struct User<T> { name:
String, age: u8, /// Varies depending on application. metadata: T, }
```

In the example above, User is a generic type with a single type parameter T , with shared fields name , age , and the generic metadata field, which can store any type. No matter what metadata is, all instances of User will contain the same fields and methods.

```bash /// Updates the name of the user. public fun update_name(user: &mut User, name: String) { user.name = name; }

/// Updates the age of the user. public fun update_age(user: &mut User, age: u8) { user.age = age; } ```

In some cases, you may want to define a generic type with a type parameter that is not used in the fields or methods of the type. This is called a phantom type parameter . Phantom type parameters are useful when you want to define a type that can hold any other type, but you want to enforce some constraints on the type parameter.

```
bash /// A generic type with a phantom type parameter. public struct Coin<phantom T> { value: u64 }
```

The Coin type here does not contain any fields or methods that use the type parameter T . It is used to differentiate between different types of coins, and to enforce some constraints on the type parameter T .

```bash public struct USD {} public struct EUR {}

# [test]

fun test_phantom_type() { let coin1: Coin = Coin { value: 10 }; let coin2: Coin = Coin { value: 20 };

```
// Unpacking is identical because the phantom type parameter is not used.
let Coin { value: _ } = coin1;
let Coin { value: _ } = coin2;
```

} ```

In the example above, we demonstrate how to create two different instances of Coin with different phantom type parameters USD and EUR . The type parameter T is not used in the fields or methods of the Coin type, but it is used to differentiate between different types of coins. This helps ensure that the USD and EUR coins are not mistakenly mixed up.

Type parameters can be constrained to have certain abilities. This is useful when you need the inner type to allow certain behaviors, such as copy or drop . The syntax for constraining a type parameter is T: + .

``bash /// A generic type with a type parameter that has thedrop` ability. public struct Droppable { value: T, }

/// A generic struct with a type parameter that has the `copy` and `drop` abilities. public struct CopyableDroppable { value: T, // T must have the `copy` and `drop` abilities } ```

The Move Compiler will enforce that the type parameter T has the specified abilities. If the type parameter does not have the specified abilities, the code will not compile.

```bash /// Type without any abilities. public struct NoAbilities {}
```

[test]

fun test_constraints() { // Fails - `NoAbilities` does not have the `drop` ability // let droppable = Droppable { value: 10 };

```
// Fails - `NoAbilities` does not have the `copy` and `drop` abilities
// let copyable_droppable = CopyableDroppable<NoAbilities> { value: 10 };
```

} ```

Generic Syntax

To define a generic type or function, a type signature needs to have a list of generic parameters enclosed in angle brackets (< and >). The generic parameters are separated by commas.

``bash /// Container for any type`T`. public struct Container has drop { value: T, }

/// Function that creates a new `Container` with a generic value `T`. public fun new(value: T): Container { Container { value } } ```

In the example above, Container is a generic type with a single type parameter T , the value field of the container stores the T . The new function is a generic function with a single type parameter T , and it returns a Container with the given value. Generic types must be initialized with a concrete type, and generic functions must be called with a concrete type, although in some cases the Move compiler can infer the correct type.

```bash

# [test]

fun test_container() { // these three lines are equivalent let container: Container = new(10); // type inference let container = new(10); // create a new `Container` with a `u8` value let container = new(10u8);

```
assert!(container.value == 10, 0x0);

// Value can be ignored only if it has the `drop` ability.
let Container { value: _ } = container;
```

} ```

In the test function test_generic , we demonstrate three equivalent ways to create a new Container with a u8 value. Because numeric constants have ambiguous types, we must specify the type of the number literal somewhere (in the type of the container, the parameter to new , or the number literal itself); once we specify one of these the compiler can infer the others.

You can define a type or function with multiple type parameters. The type parameters are separated by commas.

``bash /// A pair of values of any type`T`and`U`. public struct Pair { first: T, second: U, }

/// Function that creates a new `Pair` with two generic values `T` and `U`. public fun new_pair(first: T, second: U): Pair { Pair { first, second } } ```

In the example above, Pair is a generic type with two type parameters T and U , and the new_pair function is a generic function with two type parameters T and U . The function returns a Pair with the given values. The order of the type parameters is important, and should match the order of the type parameters in the type signature.

```bash

# [test]

fun test_generic() { // these three lines are equivalent let pair_1: Pair = new_pair(10, true); // type inference let pair_2 = new_pair(10, true); // create a new `Pair` with a `u8` and `bool` values let pair_3 = new_pair(10u8, true);

```
assert!(pair_1.first == 10, 0x0);
assert!(pair_1.second, 0x0);

// Unpacking is identical.
```

```
let Pair { first: _, second: _ } = pair_1;
let Pair { first: _, second: _ } = pair_2;
let Pair { first: _, second: _ } = pair_3;

} ```

If we added another instance where we swapped type parameters in the new_pair function, and tried to compare two types, we'd see that the type signatures are different, and cannot be compared.

```bash

# [test]

fun test_swap_type_params() { let pair1: Pair = new_pair(10u8, true); let pair2: Pair = new_pair(true, 10u8);

// this line will not compile
// assert!(pair1 == pair2, 0x0);

let Pair { first: pf1, second: ps1 } = pair1; // first1: u8, second1: bool
let Pair { first: pf2, second: ps2 } = pair2; // first2: bool, second2: u8

assert!(pf1 == ps2); // 10 == 10
assert!(ps1 == pf2); // true == true

} ```

Since the types for pair1 and pair2 are different, the comparison pair1 == pair2 will not compile.

In the examples above we focused on instantiating generic types and calling generic functions to create instances of these types. However, the real power of generics lies in their ability to define shared behavior for the base, generic type, and then use it independently of the concrete types. This is especially useful when working with collections, abstract implementations, and other advanced features in Move.

bash /// A user record with name, age, and some generic metadata public struct User<T> { name: String, age: u8, /// Varies depending on application. metadata: T, }

In the example above, User is a generic type with a single type parameter T , with shared fields name , age , and the generic metadata field, which can store any type. No matter what metadata is, all instances of User will contain the same fields and methods.

```bash /// Updates the name of the user. public fun update_name(user: &mut User, name: String) { user.name = name; }

/// Updates the age of the user. public fun update_age(user: &mut User, age: u8) { user.age = age; } ```

In some cases, you may want to define a generic type with a type parameter that is not used in the fields or methods of the type. This is called a phantom type parameter . Phantom type parameters are useful when you want to define a type that can hold any other type, but you want to enforce some constraints on the type parameter.

bash /// A generic type with a phantom type parameter. public struct Coin<phantom T> { value: u64 }

The Coin type here does not contain any fields or methods that use the type parameter T . It is used to differentiate between different types of coins, and to enforce some constraints on the type parameter T .

```bash public struct USD {} public struct EUR {}

# [test]

fun test_phantom_type() { let coin1: Coin = Coin { value: 10 }; let coin2: Coin = Coin { value: 20 };

// Unpacking is identical because the phantom type parameter is not used.
let Coin { value: _ } = coin1;
let Coin { value: _ } = coin2;

} ```

In the example above, we demonstrate how to create two different instances of Coin with different phantom type parameters USD and EUR . The type parameter T is not used in the fields or methods of the Coin type, but it is used to differentiate between different

types of coins. This helps ensure that the USD and EUR coins are not mistakenly mixed up.

Type parameters can be constrained to have certain abilities. This is useful when you need the inner type to allow certain behaviors, such as copy or drop . The syntax for constraining a type parameter is T: + .

``bash /// A generic type with a type parameter that has thedrop` ability. public struct Droppable { value: T, }

/// A generic struct with a type parameter that has the `copy` and `drop` abilities. public struct CopyableDroppable { value: T, // T must have the `copy` and `drop` abilities } ```

The Move Compiler will enforce that the type parameter T has the specified abilities. If the type parameter does not have the specified abilities, the code will not compile.

```bash /// Type without any abilities. public struct NoAbilities {}

# [test]

fun test_constraints() { // Fails - `NoAbilities` does not have the `drop` ability // let droppable = Droppable { value: 10 };

```
// Fails - `NoAbilities` does not have the `copy` and `drop` abilities
// let copyable_droppable = CopyableDroppable<NoAbilities> { value: 10 };
```

} ```

Multiple Type Parameters

You can define a type or function with multiple type parameters. The type parameters are separated by commas.

``bash /// A pair of values of any typeTandU`. public struct Pair { first: T, second: U, }

/// Function that creates a new `Pair` with two generic values `T` and `U`. public fun new_pair(first: T, second: U): Pair { Pair { first, second } } ```

In the example above, Pair is a generic type with two type parameters T and U , and the new_pair function is a generic function with two type parameters T and U . The function returns a Pair with the given values. The order of the type parameters is important, and should match the order of the type parameters in the type signature.

```bash

# [test]

fun test_generic() { // these three lines are equivalent let pair_1: Pair = new_pair(10, true); // type inference let pair_2 = new_pair(10, true); // create a new `Pair` with a `u8` and `bool` values let pair_3 = new_pair(10u8, true);

```
assert!(pair_1.first == 10, 0x0);
assert!(pair_1.second, 0x0);

// Unpacking is identical.
let Pair { first: _, second: _ } = pair_1;
let Pair { first: _, second: _ } = pair_2;
let Pair { first: _, second: _ } = pair_3;
```

} ```

If we added another instance where we swapped type parameters in the new_pair function, and tried to compare two types, we'd see that the type signatures are different, and cannot be compared.

```bash

# [test]

fun test_swap_type_params() { let pair1: Pair = new_pair(10u8, true); let pair2: Pair = new_pair(true, 10u8);
```

```
// this line will not compile
// assert!(pair1 == pair2, 0x0);

let Pair { first: pf1, second: ps1 } = pair1; // first1: u8, second1: bool
let Pair { first: pf2, second: ps2 } = pair2; // first2: bool, second2: u8

assert!(pf1 == ps2); // 10 == 10
assert!(ps1 == pf2); // true == true

} ```
```

Since the types for pair1 and pair2 are different, the comparison pair1 == pair2 will not compile.

In the examples above we focused on instantiating generic types and calling generic functions to create instances of these types. However, the real power of generics lies in their ability to define shared behavior for the base, generic type, and then use it independently of the concrete types. This is especially useful when working with collections, abstract implementations, and other advanced features in Move.

```
bash /// A user record with name, age, and some generic metadata public struct User<T> { name:
String, age: u8, /// Varies depending on application. metadata: T, }
```

In the example above, User is a generic type with a single type parameter T , with shared fields name , age , and the generic metadata field, which can store any type. No matter what metadata is, all instances of User will contain the same fields and methods.

```bash /// Updates the name of the user. public fun update_name(user: &mut User, name: String) { user.name = name; }

/// Updates the age of the user. public fun update_age(user: &mut User, age: u8) { user.age = age; } ```

In some cases, you may want to define a generic type with a type parameter that is not used in the fields or methods of the type. This is called a phantom type parameter . Phantom type parameters are useful when you want to define a type that can hold any other type, but you want to enforce some constraints on the type parameter.

```
bash /// A generic type with a phantom type parameter. public struct Coin<phantom T> { value: u64 }
```

The Coin type here does not contain any fields or methods that use the type parameter T . It is used to differentiate between different types of coins, and to enforce some constraints on the type parameter T .

```bash public struct USD {} public struct EUR {}

# [test]

fun test_phantom_type() { let coin1: Coin = Coin { value: 10 }; let coin2: Coin = Coin { value: 20 };

// Unpacking is identical because the phantom type parameter is not used.
let Coin { value: _ } = coin1;
let Coin { value: _ } = coin2;

} ```
```

In the example above, we demonstrate how to create two different instances of Coin with different phantom type parameters USD and EUR . The type parameter T is not used in the fields or methods of the Coin type, but it is used to differentiate between different types of coins. This helps ensure that the USD and EUR coins are not mistakenly mixed up.

Type parameters can be constrained to have certain abilities. This is useful when you need the inner type to allow certain behaviors, such as copy or drop . The syntax for constraining a type parameter is T: + .

```bash /// A generic type with a type parameter that has thedrop` ability. public struct Droppable { value: T, }

/// A generic struct with a type parameter that has the `copy` and `drop` abilities. public struct CopyableDroppable { value: T, // T must have the `copy` and `drop` abilities } ```
```

The Move Compiler will enforce that the type parameter T has the specified abilities. If the type parameter does not have the specified abilities, the code will not compile.

```bash /// Type without any abilities. public struct NoAbilities {}
```
```

[test]

fun test_constraints() { // Fails - `NoAbilities` does not have the `drop` ability // let droppable = Droppable { value: 10 };

```
// Fails - `NoAbilities` does not have the `copy` and `drop` abilities
// let copyable_droppable = CopyableDroppable<NoAbilities> { value: 10 };
```

} ```

Why Generics?

In the examples above we focused on instantiating generic types and calling generic functions to create instances of these types. However, the real power of generics lies in their ability to define shared behavior for the base, generic type, and then use it independently of the concrete types. This is especially useful when working with collections, abstract implementations, and other advanced features in Move.

```
bash /// A user record with name, age, and some generic metadata public struct User<T> { name:
String, age: u8, /// Varies depending on application. metadata: T, }
```

In the example above, User is a generic type with a single type parameter T , with shared fields name , age , and the generic metadata field, which can store any type. No matter what metadata is, all instances of User will contain the same fields and methods.

```bash /// Updates the name of the user. public fun update_name(user: &mut User, name: String) { user.name = name; }

/// Updates the age of the user. public fun update_age(user: &mut User, age: u8) { user.age = age; } ```

In some cases, you may want to define a generic type with a type parameter that is not used in the fields or methods of the type. This is called a phantom type parameter . Phantom type parameters are useful when you want to define a type that can hold any other type, but you want to enforce some constraints on the type parameter.

```
bash /// A generic type with a phantom type parameter. public struct Coin<phantom T> { value: u64 }
```

The Coin type here does not contain any fields or methods that use the type parameter T . It is used to differentiate between different types of coins, and to enforce some constraints on the type parameter T .

```bash public struct USD {} public struct EUR {}

# [test]

fun test_phantom_type() { let coin1: Coin = Coin { value: 10 }; let coin2: Coin = Coin { value: 20 };

```
// Unpacking is identical because the phantom type parameter is not used.
let Coin { value: _ } = coin1;
let Coin { value: _ } = coin2;
```

} ```

In the example above, we demonstrate how to create two different instances of Coin with different phantom type parameters USD and EUR . The type parameter T is not used in the fields or methods of the Coin type, but it is used to differentiate between different types of coins. This helps ensure that the USD and EUR coins are not mistakenly mixed up.

Type parameters can be constrained to have certain abilities. This is useful when you need the inner type to allow certain behaviors, such as copy or drop . The syntax for constraining a type parameter is T: + .

```bash /// A generic type with a type parameter that has the`drop` ability. public struct Droppable { value: T, }

/// A generic struct with a type parameter that has the `copy` and `drop` abilities. public struct CopyableDroppable { value: T, // T must have the `copy` and `drop` abilities } ```

The Move Compiler will enforce that the type parameter T has the specified abilities. If the type parameter does not have the specified abilities, the code will not compile.

```bash /// Type without any abilities. public struct NoAbilities {}

# [test]

fun test_constraints() { // Fails - `NoAbilities` does not have the `drop` ability // let droppable = Droppable { value: 10 };

```
// Fails - `NoAbilities` does not have the `copy` and `drop` abilities
// let copyable_droppable = CopyableDroppable<NoAbilities> { value: 10 };
```

} ```

Phantom Type Parameters

In some cases, you may want to define a generic type with a type parameter that is not used in the fields or methods of the type. This is called a phantom type parameter . Phantom type parameters are useful when you want to define a type that can hold any other type, but you want to enforce some constraints on the type parameter.

`bash /// A generic type with a phantom type parameter. public struct Coin<phantom T> { value: u64 }`

The Coin type here does not contain any fields or methods that use the type parameter T . It is used to differentiate between different types of coins, and to enforce some constraints on the type parameter T .

```bash public struct USD {} public struct EUR {}

# [test]

fun test_phantom_type() { let coin1: Coin = Coin { value: 10 }; let coin2: Coin = Coin { value: 20 };

```
// Unpacking is identical because the phantom type parameter is not used.
let Coin { value: _ } = coin1;
let Coin { value: _ } = coin2;
```

} ```

In the example above, we demonstrate how to create two different instances of Coin with different phantom type parameters USD and EUR . The type parameter T is not used in the fields or methods of the Coin type, but it is used to differentiate between different types of coins. This helps ensure that the USD and EUR coins are not mistakenly mixed up.

Type parameters can be constrained to have certain abilities. This is useful when you need the inner type to allow certain behaviors, such as copy or drop . The syntax for constraining a type parameter is T: + .

``bash /// A generic type with a type parameter that has the`drop` ability. public struct Droppable { value: T, }

/// A generic struct with a type parameter that has the `copy` and `drop` abilities. public struct CopyableDroppable { value: T, // T must have the `copy` and `drop` abilities } ```

The Move Compiler will enforce that the type parameter T has the specified abilities. If the type parameter does not have the specified abilities, the code will not compile.

```bash /// Type without any abilities. public struct NoAbilities {}

# [test]

fun test_constraints() { // Fails - `NoAbilities` does not have the `drop` ability // let droppable = Droppable { value: 10 };

```
// Fails - `NoAbilities` does not have the `copy` and `drop` abilities
// let copyable_droppable = CopyableDroppable<NoAbilities> { value: 10 };
```

} ```

Constraints on Type Parameters

Type parameters can be constrained to have certain abilities. This is useful when you need the inner type to allow certain behaviors, such as copy or drop . The syntax for constraining a type parameter is T: + .

```bash /// A generic type with a type parameter that has the`drop` ability. public struct Droppable { value: T, }

/// A generic struct with a type parameter that has the `copy` and `drop` abilities. public struct CopyableDroppable { value: T, // T must have the `copy` and `drop` abilities } ```

The Move Compiler will enforce that the type parameter T has the specified abilities. If the type parameter does not have the specified abilities, the code will not compile.

```bash /// Type without any abilities. public struct NoAbilities {}

# [test]

fun test_constraints() { // Fails - `NoAbilities` does not have the `drop` ability // let droppable = Droppable { value: 10 };

```
// Fails - `NoAbilities` does not have the `copy` and `drop` abilities
// let copyable_droppable = CopyableDroppable<NoAbilities> { value: 10 };
```

} ```

Further Reading