Safety Features in the Move Language

Meng Xu *(University of Waterloo)*[1]

January 23, 2023

[1] All views personal and do not reflect the stance of my affliations.

Outline

Basics

- Move is a programming language designed for safe and correct modeling of state-transition systems.
  - Its primary goal is to support smart contracts but the language has more potential than that.
  - The Move language is competing with mainly Ethereum Virtual Machine based languages, including Solidity, Vyper, Yul, WASM, etc.

Basics

- Move is a programming language designed for safe and correct modeling of state-transition systems.
  - Its primary goal is to support smart contracts but the language has more potential than that.
  - The Move language is competing with mainly Ethereum Virtual Machine based languages, including Solidity, Vyper, Yul, WASM, etc.

- Move is live and evolving in real world.
  - Aptos: Layer-1 blockchain, in mainnet
  - Sui, Layer-1 blockchain, in devnet
  - Starcoin, Layer-1 blockchain, in mainnet
  - 0L, Layer-1 blockchain, in mainnet
  - Pontem, DApp parachain, in testnet
  - Celo, Layer-1 blockchain, not launched
  - Diem, Layer-1 blockchain, sold to Silvergate
  - ChainX, Layer-2 blockchain, in mainnet

Some fun facts

- Move is developed and maintained by a small team: 18 in total
  - Language design: 3
  - Bytecode verifier: 2
  - Virtual machine: 2
  - Compiler: 2
  - Formal verification: 5
  - Tools and ecosystem: 4

Some fun facts

- Move is developed and maintained by a small team: 18 in total
  - Language design: 3
  - Bytecode verifier: 2
  - Virtual machine: 2
  - Compiler: 2
  - Formal verification: 5
  - Tools and ecosystem: 4

- Out of which 15 are from the Diem project. After Meta discontinued Diem,
  - Aptos (valued at $4 billion): 6
  - Sui (valued at $2 billion): 6
  - Academia and research labs: 3

Motivation

The **goal** of this talk is to advertise the language to other areas of research in security and privacy field to see if the language has anything to offer in other problem settings, such as:

- Design and reason about correctness of security policies
- Implement / prototype privacy-preserving protocols
- Prevent certain type of programming bugs / vulnerabilities
- .....

A tour about safety features in Move

A tour about safety features in Move

- Move type system
- Move verification system

A tour about safety features in Move

- Move type system
  - Type safety
  - Resource safety
  - Reference safety
- Move verification system

A tour about safety features in Move

- Move type system
  - Type safety
  - Resource safety
  - Reference safety
- Move verification system
  - Struct invariants
  - Unit specification (per single function)
    * Pre-conditoin
    * Abort conditions
    * Post-conditions
  - State-machine specification
    * Global invariants (single-state invariant)
    * Global update invariants (two-state invariant)

The Move programming language

- Move is based on the concepts of
  - borrow semantics (like Rust) and
  - linear types (like the `unique_ptr` in C++)
- Move supports high-order functions (in progress).
  - a.k.a., dynamic dispatching
  - Not all callsites have to be determined statically
- A Move program interacts with external states through a small and fixed set of APIs
- Comes with a fully integrated **specification language** supporting pre/post conditions and global state invariants
  - Uses full first-order predicate calculus
  - including universal and existential quantification

Outline

It's all about capturing intention

It's all about capturing intention

At a philosophical level, both the Move type system and the formal verification system (*i.e.*, the Move Prover) aim to solicite and formalize intentions from Move developers.

It's all about capturing intention

At a philosophical level, both the Move type system and the formal verification system (*i.e.*, the Move Prover) aim to solicite and formalize intentions from Move developers.

Tip of advice: Do not fight against the type system nor the Prover.

Core components of the Move type system

- Type safety
- Resource safety
- Reference safety

Type safety

Type safety

Summary: Move is a strongly/statically typed language with strictly no type conversions.

Type safety

Summary: Move is a strongly/statically typed language with strictly no type conversions.

In more details:

- Every variable and expression in Move has one and only one type.
- The type is known at compile time.
- The type can never be changed whatsoever.

Violations of type safety: an example

```
1  module 0x1::Account {
2      struct Coin has key {
3          balance: u64
4      }
5
6      public fun wrong1(acc: signer) {
7          let val = 1000;
8          move_to<Coin>(&acc, val);
9      }
10
11     public fun wrong2(acc: signer) {
12         let val = 1000;
13         let coin: Coin = val;
14         move_to<Coin>(&acc, coin);
15     }
16 }
```

Resource safety

Resource safety

Summary: Move enforces fine-grained control on permissible "behaviors" of all objects belong to a specific type.

Resource safety

Summary: Move enforces fine-grained control on permissible "behaviors" of all objects belong to a specific type.

In more details:

- Scarcity of a resource via linear typing
  - `copy` and `drop` ability
- Longevity of a resource via interaction with the global storage
  - `key` and `store` ability
- Encapsulation of a resource via module-based access control
  - Logic about a resource is encapsulated within a module

Violations of resource scarcity: an example

```
1  module 0x1::Account {
2      // coin does not have `copy` nor `drop`
3      struct Coin has key { balance: u64 }
4
5      fun spend_coin(coin: Coin) {
6          let Coin { balance: _ } = coin;
7      }
8
9      fun dup_coin(coin: Coin) {
10         let new_coin = copy coin;
11         spend_coin(coin);
12         spend_coin(new_coin);
13     }
14
15     fun dup_coin_via_ref(coin: &Coin) {
16         let new_coin = *coin;
17         spend_coin(new_coin);
18     }
19
20     fun silent_drop_coin(_coin: Coin) {}
21 }
```

Resource-safe pattern example: hot potatoes

```
1  module 0x1::Capability { struct HotPotato { value: u64 } }
2
3  module 0x1::Account {
4      use 0x1::Capability::HotPotato;
5      struct Wrap has key { potato: HotPotato }
6
7      fun cannot_drop(_potato: HotPotato) {}
8      fun cannot_copy(potato: HotPotato): (HotPotato, HotPotato) {
9          let new_potato = copy potato;
10         (potato, new_potato)
11     }
12     fun cannot_store(potato: HotPotato, account: signer) {
13         let wrap = Wrap { potato };
14         move_to(&account, wrap);
15     }
16     fun cannot_access(potato: HotPotato): HotPotato {
17         let _ = potato.value;
18         potato
19     }
20     fun cannot_destroy(potato: HotPotato) {
21         let HotPotato { value: _ } = potato;
22     }
23     fun must_return(potato: HotPotato): HotPotato { potato }
24 }
```

Reference safety

Reference safety

Summary: Move ensures that there are no dangling references to both function locals and global storage — via ownership rules.

Reference safety

Summary: Move ensures that there are no dangling references to both function locals and global storage — via ownership rules.

In more details:

- Any function local variable is uniquely owned at any time.
- Any `Global<T>` is uniquely owned at any time.

By *uniquely owned*, it means that a slot has:

- at most one writer and no readers, OR
- N (≥ 0) readers and no writers.

Outline

Why do we need Move Prover given these typing rules?

Why do we need Move Prover given these typing rules?

There will be more complicated semantic / intention that cannot be captured by the typing system.

Examples of such advanced semantics

- Struct invariant
- Unit specification (per single function)
  - Pre-conditoin
  - Abort conditions
  - Post-conditions
- State-machine specification
  - Global invariants (single-state invariant)
  - Global update invariants (two-state invariant)

Struct invariant

Struct invariants allows you to specify complicated relations among the fields of a struct type which have to hold at runtime.

Struct invariant: example

```
module 0x1::Account {
    struct NonZeroU64 {
        value: u64
    }
    spec NonZeroU64 {
        invariant value > 0;
    }
}
```

Struct invariant: example

```
module 0x1::Account {
    struct NonZeroU64 {
        value: u64
    }
    spec NonZeroU64 {
        invariant value > 0;
    }
}
```

```
1  module 0x1::Account {
2      ...
3
4      fun create_zero(): NonZeroU64 {
5          NonZeroU64 { value: 0 }
6      }
7
8      fun create_x_checked(x: u64): NonZeroU64
9          assert!(x != 0, 1);
10         NonZeroU64 { value: x }
11     }
12 }
```

Struct invariant: example

```
1  module 0x1::Account {
2      struct SumIsConst {
3          a: u64,
4          b: u64,
5      }
6      spec SumIsConst {
7          invariant a >= b;
8          invariant a + b == 100;
9      }
10
11     fun create_valid(x: u64): SumIsConst {
12         assert!(x >= 50, 1);
13         SumIsConst { a: x, b: 100 - x }
14     }
15 }
```

Function specification

For people not familiar with formal verification, function specification can be loosely considered as exhaustive unit testing.

Function specification: running example

```
1 module 0x1::Account {
2     struct Account has key { balance: u64 }
3
4     fun withdraw(account: address, amount: u64) acquires Account {
5         let ptr: &mut Account = borrow_global_mut<Account>(account);
6         ptr.balance = ptr.balance - amount;
7     }
8 }
```

The unit testing joke

A software engineer walks up to an ATM and ...

The unit testing joke

A software engineer walks up to an ATM and ...

- `withdraw(@0x1, 10);`
- `withdraw(@0x1, 1000);`
- `withdraw(@0x1, 99999999999999999999999999);`
- `withdraw(@0x0, 0);`
- `withdraw(@0x0, 20);`
- `withdraw(@0xdeadbeef, -123);`
- `withdraw(@0xbaadf00d, 666 * 2);`

The unit testing joke

A software engineer walks up to an ATM and ...

- `withdraw(@0x1, 10);`
- `withdraw(@0x1, 1000);`
- `withdraw(@0x1, 99999999999999999999999999);`
- `withdraw(@0x0, 0);`
- `withdraw(@0x0, 20);`
- `withdraw(@0xdeadbeef, -123);`
- `withdraw(@0xbaadf00d, 666 * 2);`

Q: what is the engineering testing?

The unit testing joke

A software engineer walks up to an ATM and ...

- `withdraw(@0x1, 10);`
- `withdraw(@0x1, 1000);`
- `withdraw(@0x1, 99999999999999999999999999);`
- `withdraw(@0x0, 0);`
- `withdraw(@0x0, 20);`
- `withdraw(@0xdeadbeef, -123);`
- `withdraw(@0xbaadf00d, 666 * 2);`

Q: what is the engineering testing?

- On correct inputs, the system should finish with desired balance.
- On erroneous inputs, the system should abort the transaction.

Abort conditions in the function specification

```
1  module 0x1::Account {
2      struct Account has key { balance: u64 }
3
4      fun withdraw(account: address, amount: u64) acquires Account {
5          let ptr: &mut Account = borrow_global_mut<Account>(account);
6          ptr.balance = ptr.balance - amount;
7      }
8      spec withdraw {
9          aborts_if !exists<Account>(account);
10         aborts_if global<Account>(account).balance < amount;
11     }
12 }
```

Trick: abort condition discovery

Trick: abort condition discovery

```
1  module 0x1::Account {
2      struct Account has key { balance: u64 }
3
4      fun withdraw(account: address, amount: u64) acquires Account {
5          let ptr: &mut Account = borrow_global_mut<Account>(account);
6          ptr.balance = ptr.balance - amount;
7      }
8      spec withdraw {
9          // let the prover discover the abort conditions for you
10         aborts_if false;
11     }
12 }
```

Post conditions in the function specification

Post conditions in the function specification

```
1  module 0x1::Account {
2      struct Account has key { balance: u64 }
3
4      fun withdraw(account: address, amount: u64) acquires Account {
5          let ptr: &mut Account = borrow_global_mut<Account>(account);
6          ptr.balance = ptr.balance - amount;
7      }
8      spec withdraw {
9          aborts_if !exists<Account>(account);
10         aborts_if global<Account>(account).balance < amount;
11
12         // on successful function return
13         ensures global<Account>(account).balance ==
14             old(global<Account>(account).balance) - amount;
15     }
16 }
```

Global invariant

Global invariant

Global invariants allow you to treat the entire global storage as a state machine and express both:

- Legal (and illegal) states *AND*
- Legal (and illegal) state transitions.

Global invariant

Global invariants allow you to treat the entire global storage as a state machine and express both:

- Legal (and illegal) states *AND*
- Legal (and illegal) state transitions.

NOTE: global invariants are higher-level constructs that allow you directly specify the state machine you have in mind *without knowing the implementation details* (*i.e.*, functions).

Global invariant: example

```
module 0x1::Account {
    struct Account has key { balance: u64 }

    spec module {
        fun bal(a: address): u64 { global<Account>(a).balance }

        // All accounts must have a minimal balance of 100
        invariant forall a: address where exists<Account>(a):
            bal(a) >= 100;

        // Any withdraw cannot exceed the 10% value limit
        invariant update forall a: address where exists<Account>(a):
            old(bal(a)) - bal(a) <= old(bal(a)) / 10;
    }
}
```

Global invariant: example

```
module 0x1::Account {
    struct Account has key { balance: u64 }
    struct Cheque { value: u64 }

    fun withdraw(account: address, amount: u64) acquires Account {
        let ptr: &mut Account = borrow_global_mut<Account>(account);
        ptr.balance = ptr.balance - amount;
    }
    fun make_cheque(account: address, amount: u64): Cheque acquires Account {
        let ptr = borrow_global_mut<Account>(account);
        ptr.balance = ptr.balance - amount;
        Cheque { value: amount }
    }

    spec module {
        fun bal(a: address): u64 { global<Account>(a).balance }

        invariant forall a: address where exists<Account>(a):
            bal(a) >= 100;

        invariant update forall a: address where exists<Account>(a):
            old(bal(a)) - bal(a) <= old(bal(a)) / 10;
    }
}
```

$\langle$ **End** $\rangle$