IOTA

Blockchain and Cryptocurrencies 2024/2025

IOTA Move Smart Contracts

Mirko Zichichi
Applied Research Engineer, IOTA Foundation
mirko.zichichi@iota.org

Mirko Zichichi

Applied Research Engineer
IOTA Foundation

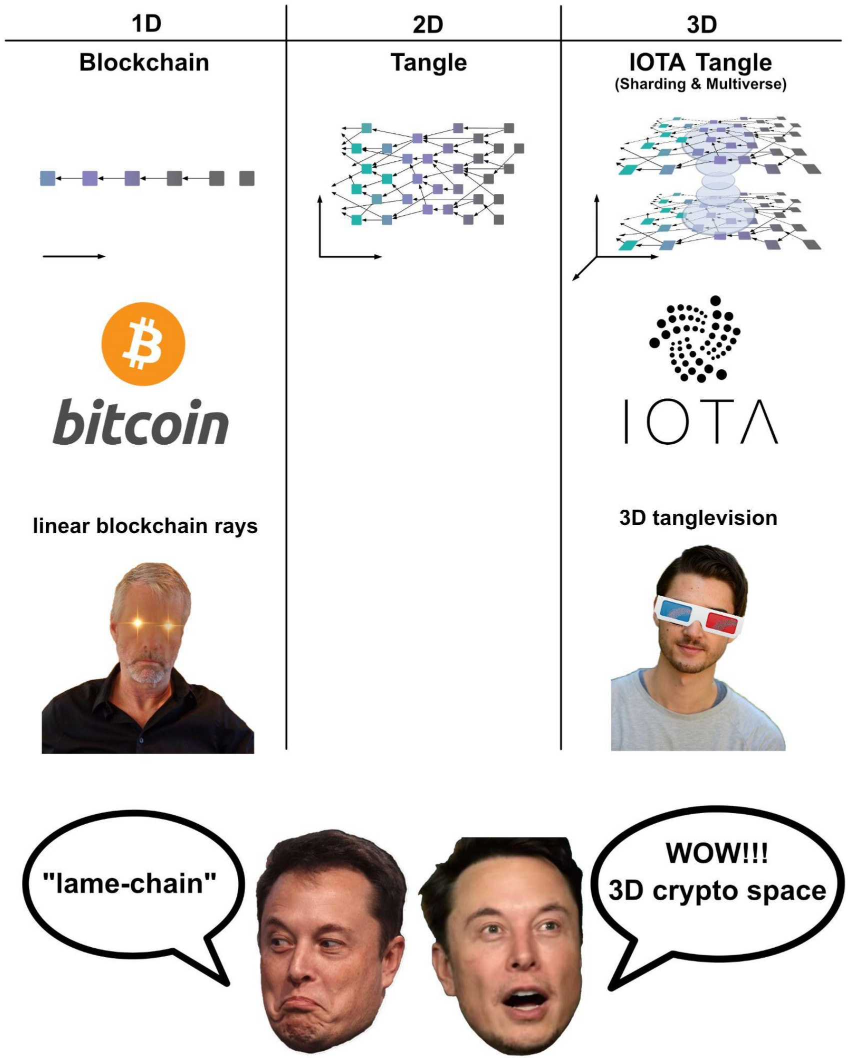
Ph.D. in **Law, Science and Technology**
Universidad Politécnica de Madrid -
University of Bologna - University of Turin

"Decentralized Systems for the Protection and Portability of Personal Data"

Supervisors:
prof. Stefano Ferretti, prof. Victor Rodriguez-Doncel

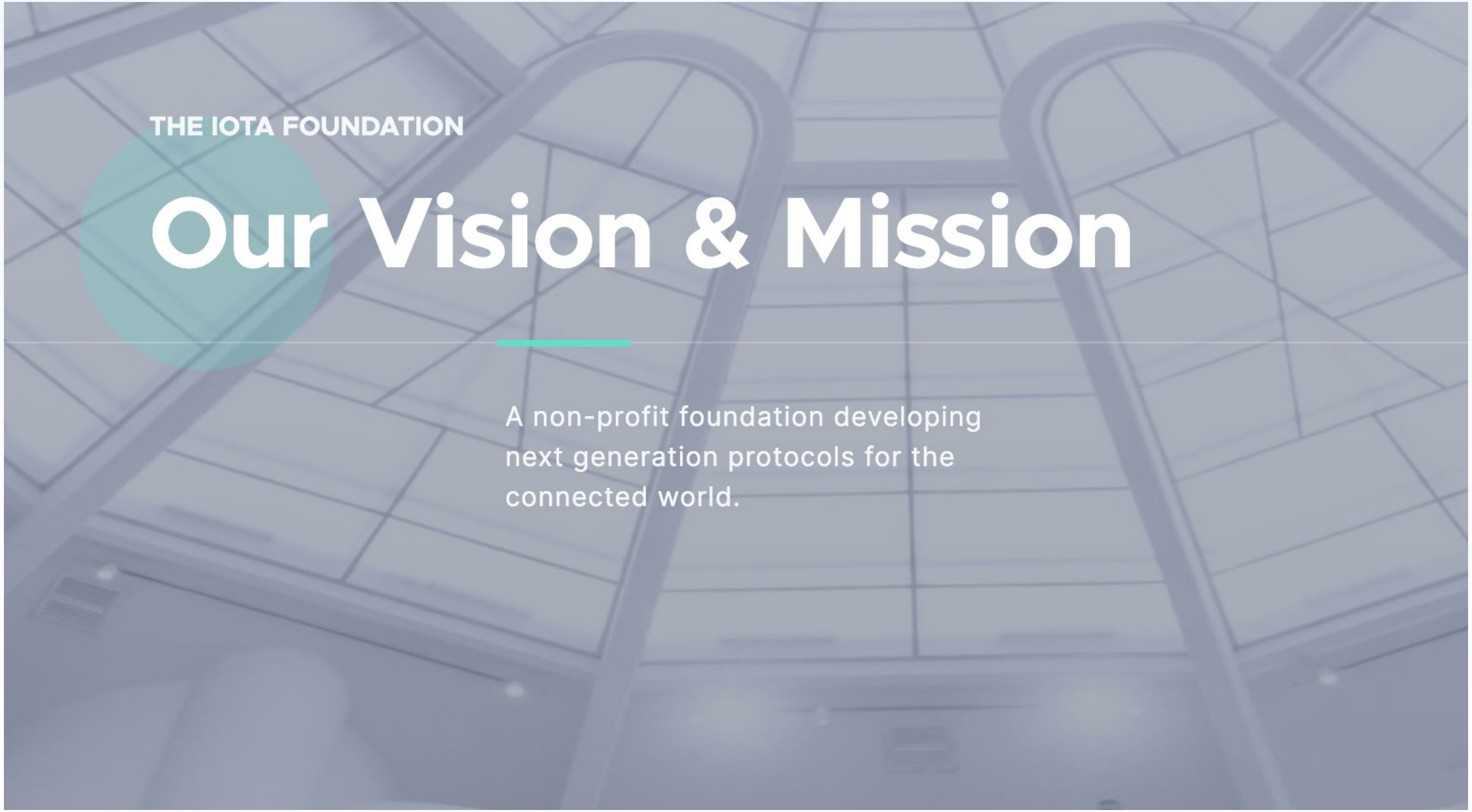
IOTA

IOTA



1D
2D
3D
Blockchain
Tangle
IOTA Tangle
(Sharding & Multiverse)
bitcoin
IOTA
linear blockchain rays
3D tanglevision
"lame-chain"
WOW!!!
3D crypto space

IOTA Foundation



IOTA Foundation

We collaborate with our community and partners to deliver sustainable, real-world impact. Together, we are shaping a new digital economy, removing unnecessary friction and unlocking human potential. Our global network of thinkers, tinkerers, leaders and doers are working together to pioneer the future.

Our Goals

- Research and implement the foundational protocol layer.
- Standardise the protocol to ensure its widespread adoption.
- Develop production-ready open-source software.
- Educate on our technologies and promote their use cases.

European blockchain regulatory sandbox for DLTs

“The sandbox establishes a pan-European framework for regulatory dialogues to increase legal certainty for innovative blockchain technology solutions [...] across industry sectors such as energy & utilities, education, healthcare, mobility, finance & insurance, and logistics & supply chains.”

Web3 Identification Solution – A Decentralised and Secure Approach to User Authentication

The Web3 Identification Solution caters to the regulatory needs of Web3 and DeFi projects and enables them to interact seamlessly with verified users while excluding unverified addresses.

IOTA

IDnow.

walt.id

Bloom

HAVN

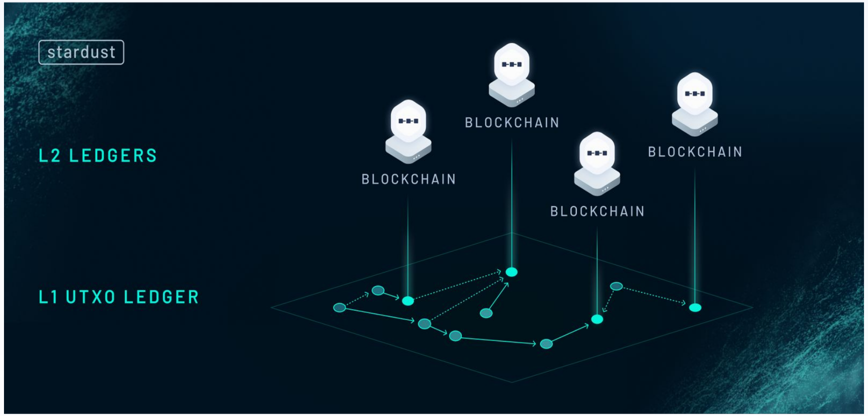
IOTA Stiftung (IDnow GmbH, walt.id GmbH, Bloom Labs Limited, Spyce5 GMBH)

https://ec.europa.eu/digital-building-blocks/sites/display/EBSISANDCOLLAB/European+Blockchain+Sandbox+announces+the+selected+projects+for+the+second+cohort#EuropeanBlockchainSandboxannouncestheselectedprojectsforthesecondcohort-IOTAStiftung

IOTA (EVM) Smart Contracts (past)

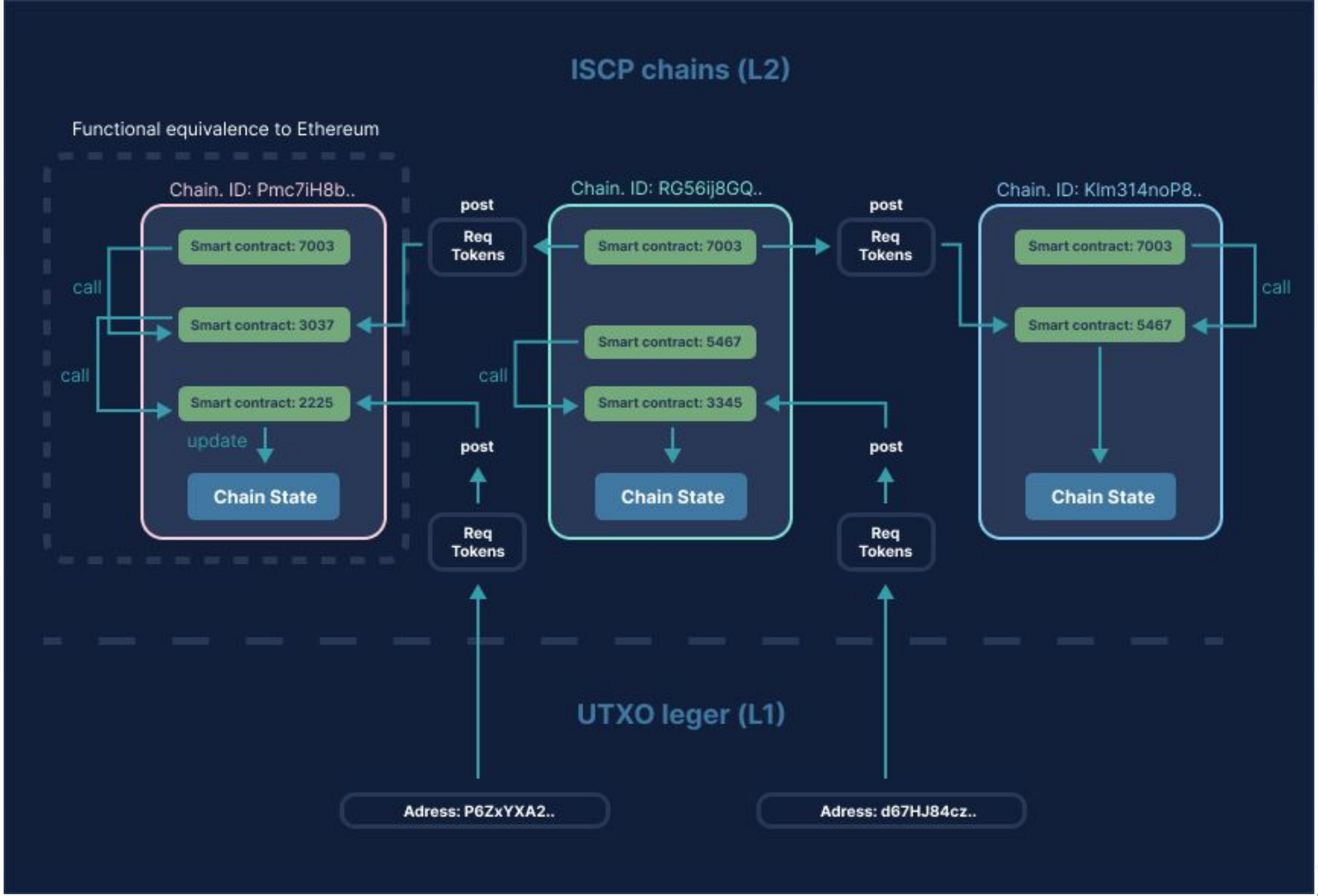
Current Solution: IOTA EVM

- It's a **Layer 2 (L2) solution** where smart contracts are handled off-tangle in their dedicated blockchain
- The blockchain is run by a permissioned committee of nodes.
- Uses Ethereum technology (EVM)
- Periodically commits the state to the **L1**



- **Layer 1** -> **Stardust VM**
- limited in its capabilities: you can't write your own apps, but you can:
  - Create fungible tokens
  - Create NFTs
  - Store data and/or commitments on-tangle.
- Enhancing L1 with a better operating system -> **increases network's utility**

https://wiki.iota.org/learn/smart-contracts/core_concepts/smart-contracts/



https://wiki.iota.org/learn/smart-contracts/core_concepts/smart-contracts/

IOTA

Move

the Language for Secure Next Gen Smart Contracts

Move

Resource

Flexibility

Security

Libra → Diem → 

https://www.theverge.com/2019/6/26/18716326/facebook-libra-cryptocurrency-blockchain-irs-starbucks

*"Hey, **Libra** will have **smart contracts**, it is important to ensure that their programming on the blockchain is secure"*

Some Facebook chief, circa 2017.

*"**The scarcest resource in the world is not time or money, but man's brain power.***

When these are used to develop software, if you can amplify brain capacity, i.e. do more per unit of time, this is one of the most impactful things you can achieve."

Blackshear, Sam, et al. "Move: A Language With Programmable Resources"
https://diem-developers-components.netlify.app/papers/diem-move-a-language-with-programmable-resources/2020-05-26.pdf (2020).

Move language design

Move’s design rationale

- Representation of **transitions** and **state** that encodes **ownership of digital assets** in an open software system

- Handle two properties of that are intrinsic of physical assets:
    - Controlled **Scarcity** ->
      duplicating existing assets should be *prohibited*,
      creating new assets should be a *privileged* operation.
    - **Access control** ->
      A participant in the system should be able to protect her *assets* with access control policies.

Blackshear, Sam, et al. "Move: A Language With Programmable Resources"
https://diem-developers-components.netlify.app/papers/diem-move-a-language-with-programmable-resources/2020-05-26.pdf (2020).

Critiques of the existing blockchain languages

- **Indirect representation of assets**

  An asset is encoded using an integer, but an integer value is not the same as an asset.

- **Scarcity is not extensible**

  The scarcity protections are hardcoded directly in the language semantics, i.e., not built-in.

- **Access control is not flexible**

  It needs to be made clear how to extend the language to allow programmers to define custom access control policies instead of only the signature scheme.

Blackshear, Sam, et al. "Move: A Language With Programmable Resources"
https://diem-developers-components.netlify.app/papers/diem-move-a-language-with-programmable-resources/2020-05-26.pdf (2020).

IOTA

Design goals: First-class Resources

- **The ability to define custom resource types** with semantics inspired by linear logic
    - a resource can never be copied or implicitly discarded,
    - only moved between program storage locations.

- Move programmers can **protect access to critical operations with modules**
    - creation, destruction, and update

- **A module declares resource types and procedures that encode the rules for its declared resources.**

Blackshear, Sam, et al. "Move: A Language With Programmable Resources"
https://diem-developers-components.netlify.app/papers/diem-move-a-language-with-programmable-resources/2020-05-26.pdf (2020).

IOTA

Design goals: Flexibility

- **Flexible code composition**
  - The relationship between modules/resources/procedures is similar to OOP's classes/objects/methods.

- The difference is that a Move module can declare multiple resource types (or zero resource types), and procedures have no notion of a self or this value.

- **No dynamic dispatch**
  - Dynamic dispatching means that the compiler does not know which method is being called and decides which method to call at runtime.
  - The static dispatching used in Move makes it easier for **verification** tools to verify the bytecode and for **increasing security**.

Blackshear, Sam, et al. "Move: A Language With Programmable Resources"
https://diem-developers-components.netlify.app/papers/diem-move-a-language-with-programmable-resources/2020-05-26.pdf (2020).

Design goals: Security

- **Move's bytecode**
  - higher-level than assembly
  - yet lower-level than a source language

- It mixes
  - the use of a high-level programming language with a **compiler that checks safety** properties
  - the use of a low-level untyped assembly that performs **safety checks at runtime**

Blackshear, Sam, et al. "Move: A Language With Programmable Resources"
https://diem-developers-components.netlify.app/papers/diem-move-a-language-with-programmable-resources/2020-05-26.pdf (2020).

Design goals: Verifiability

- Move's execution performs a **lightweight on-chain verification.**

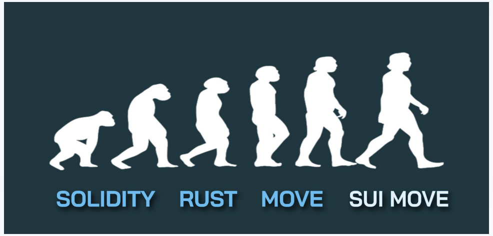
- **Limited mutability**
  - Every mutation to a Move value occurs through a reference.
  - Move's bytecode verifier uses a "borrow checking" scheme similar to Rust.

- **Modularity**
  - Move modules enforce data abstraction and localize critical operations on resources.

Blackshear, Sam, et al. "Move: A Language With Programmable Resources"
https://diem-developers-components.netlify.app/papers/diem-move-a-language-with-programmable-resources/2020-05-26.pdf (2020).

Move

Move Language



- Domain Specific Language for programming with assets
- Inherits **memory and type safety** concepts from Rust
  - **Compiler catches errors** that would normally go undetected in Solidity
- Treats **assets as first class citizens** that can travel between SC boundaries
- Programs are **formally verifiable**
- Built-in language level **permission controls**
  - Transparent what an SC can do with your assets (**read only, mutate, transfer**)

IOTA

Move "Resource"

- Move represents assets using **user-defined linear resource types**.
- Move has ordinary types like *integers and addresses that can be copied*, but **resources can only be moved**.
- Move **resource safety** -> analogous to conservation of mass in the physical world
- ***Linearity***:
    - prevents "double spending" by moving a resource twice
    - forces a procedure to move all of its resources, avoiding accidental loss.

Blackshear, Sam, et al. "Resources: A safe language abstraction for money." arXiv preprint https://arxiv.org/abs/2004.05106 (2020).

IOTA

Move “Resource”

Solidity

```
contract Bank
mapping (address => uint) credit;

function deposit() payable {
  amt =
    credit[msg.sender] + msg.value
  credit[msg.sender] = amt
}

function withdraw() {
  uint amt = credit[msg.sender];
  msg.sender.transfer(amt);
  credit[msg.sender] = 0;
}
```

Move

```
module Bank
use 0x0::Coin;
resource T { balance: Coin::T }
resource Credit { amt: u64, bank: address }

fun deposit(
  coin: Coin::T,
  bank: address
): Credit {
  let amt = Coin::value(&coin);
  let t = borrow_global<T>(copy bank);
  Coin::deposit(&mut t.balance, move coin);
  return Credit {
    amt: move amt, bank: move bank
  };
}

fun withdraw(credit: Credit): Coin::T {
  Credit { amt, bank } = move credit;
  let t = borrow_global<T>(move bank);
  return Coin::withdraw(
    &mut t.balance, move amt
  );
}
```

Blackshear, Sam, et al. "Resources: A safe language abstraction for money." arXiv preprint https://arxiv.org/abs/2004.05106 (2020).

Move Resource Safety

- At the beginning and end of a transaction script, all of the resources in the system reside in the **global state** *GS*.
- **Resource safety** is a conservation property that relates the set of resources present in state GS_{pre} before the script to the set of resources present in state GS_{post} after the script.
- In general terms, must **guarantee** that:
  - A **resource** *M::T* that is **present** in **post-state** GS_{post} was also **present** in **pre-state** GS_{pre} unless it is introduced by a *Pack* (Move bytecode for resource creation) inside *M* during script execution
  - A **resource** *M::T* that was **present** in **pre-state** GS_{pre} is also **present** in **post-state** GS_{post} unless it is eliminated by an *Unpack* (Move bytecode for resource deletion) inside *M* during script execution

Blackshear, Sam, et al. "Resources: A safe language abstraction for money." arXiv preprint https://arxiv.org/abs/2004.05106 (2020).

Move Executable Bytecode

- A Move execution platform relies on a **compiler** to transform ***source language*** programs into programs in the Move **bytecode language**.

Bytecode	Source code
MvLoc $\langle x_0 \rangle$	`move credit`
Unpack $\langle s_1 \rangle$	`Credit {amt, bank}=...`
BorrowGlobal $\langle s_0 \rangle$	`borrow_global<T>(...)`
BorrowField $\langle f_0 \rangle$	`&mut t.balance`
Call $\langle h_0 \rangle$	`Coin::withdraw(...)`
Ret	`return`

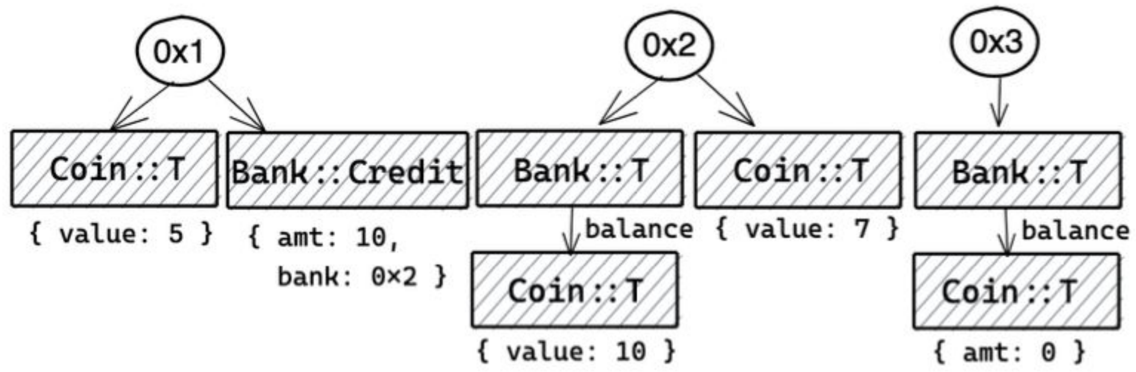
- The Move execution platform relies on a *load-time* **bytecode verifier**, that enforces *type, memory, and resource safety*.
  - If the safety guarantees were only enforced by the compiler, an adversary could subvert them by writing malicious bytecode directly and deploying it

Blackshear, Sam, et al. "Resources: A safe language abstraction for money." arXiv preprint https://arxiv.org/abs/2004.05106 (2020).

IOTA

Move Persistent Global State

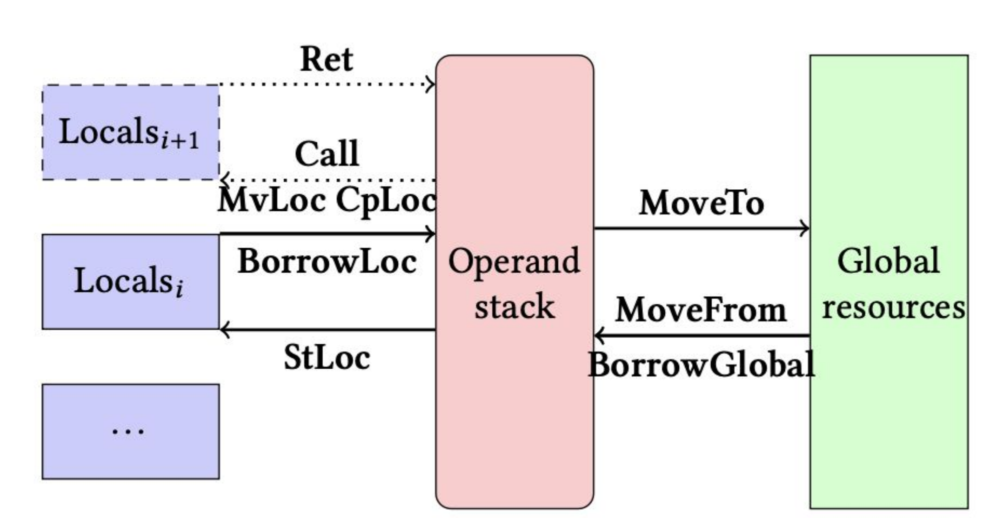
- Move execution occurs in the context of a **persistent global state** organized as a partial **map** from ***account addresses*** -> ***resource data values***



Blackshear, Sam, et al. "Resources: A safe language abstraction for money." arXiv preprint https://arxiv.org/abs/2004.05106 (2020).

Move Execution

- Begins by executing the **main procedure of the transaction script**
- A **procedure** is defined by a ***type signature*** and an ***executable body*** (Move bytecode commands).
- Procedure calls are implemented using a standard **call stack** containing frames with a ***procedure name***, a set of ***local variables***, and a ***return address***.



Blackshear, Sam, et al. "Resources: A s

Move Module, i.e., the Smart Contract

- A Move module can declare both **record types** and **procedures**.
- **Records** can store primitive data values (booleans, addresses, …) as well as other record values:
  - each record is declared as a resource or non-resource;
  - non-resource records cannot store resource records;
  - only resources can be stored in the global state.
- **Module's strong encapsulation:**
  privileged ***operations*** on the module's declared ***types*** can **only** be ***performed by procedures in the module***

Blackshear, Sam, et al. "Resources: A safe language abstraction for money." arXiv preprint https://arxiv.org/abs/2004.05106 (2020).

IOTA

Move References

- Move supports **references to** *records* and *primitive values*:
  all reads and writes of record fields occur through a reference.
- ,References are either:
    - exclusive/mutable -> ***&mut***
    - read-only -> **&**
- References are different from other Move values because they are **transient**
    - each reference must be created during the execution of a transaction script
      and **released before the end** of that transaction script.

Blackshear, Sam, et al. "Resources: A safe language abstraction for money." arXiv preprint https://arxiv.org/abs/2004.05106 (2020).

Sui Move Flavor

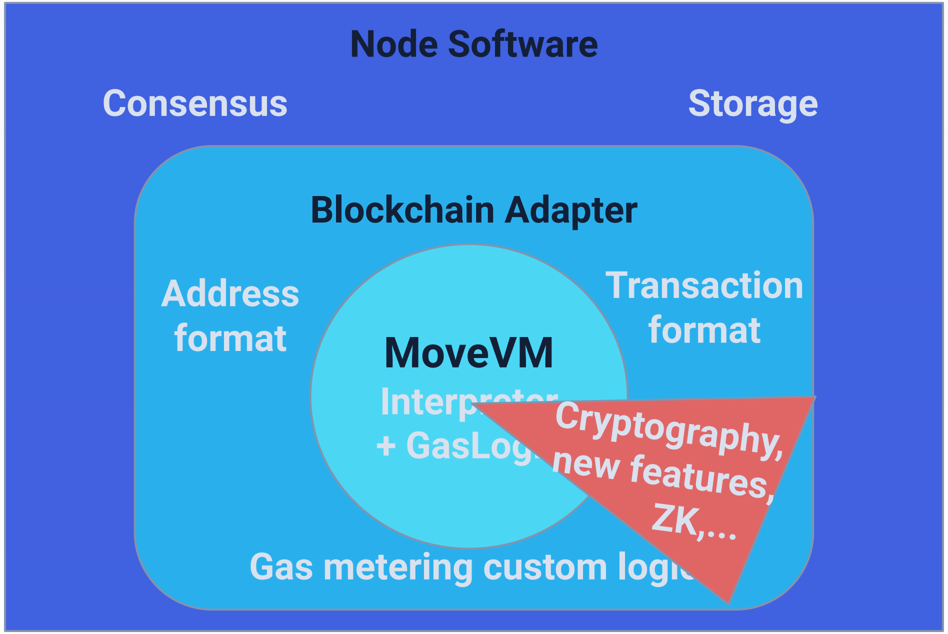
Move Virtual Machine

- **Blockchain agnostic:** we define how accounts and transactions work

- Core VM is **easily extensible** with:
  - Cryptography, signature schemes, ZKP verifiers
  - Blockchain specific features (mana generation, system transactions, account concept, etc.)

- Built-in **gas metering and safe math**: no undefined behavior is possible

Move Modularity

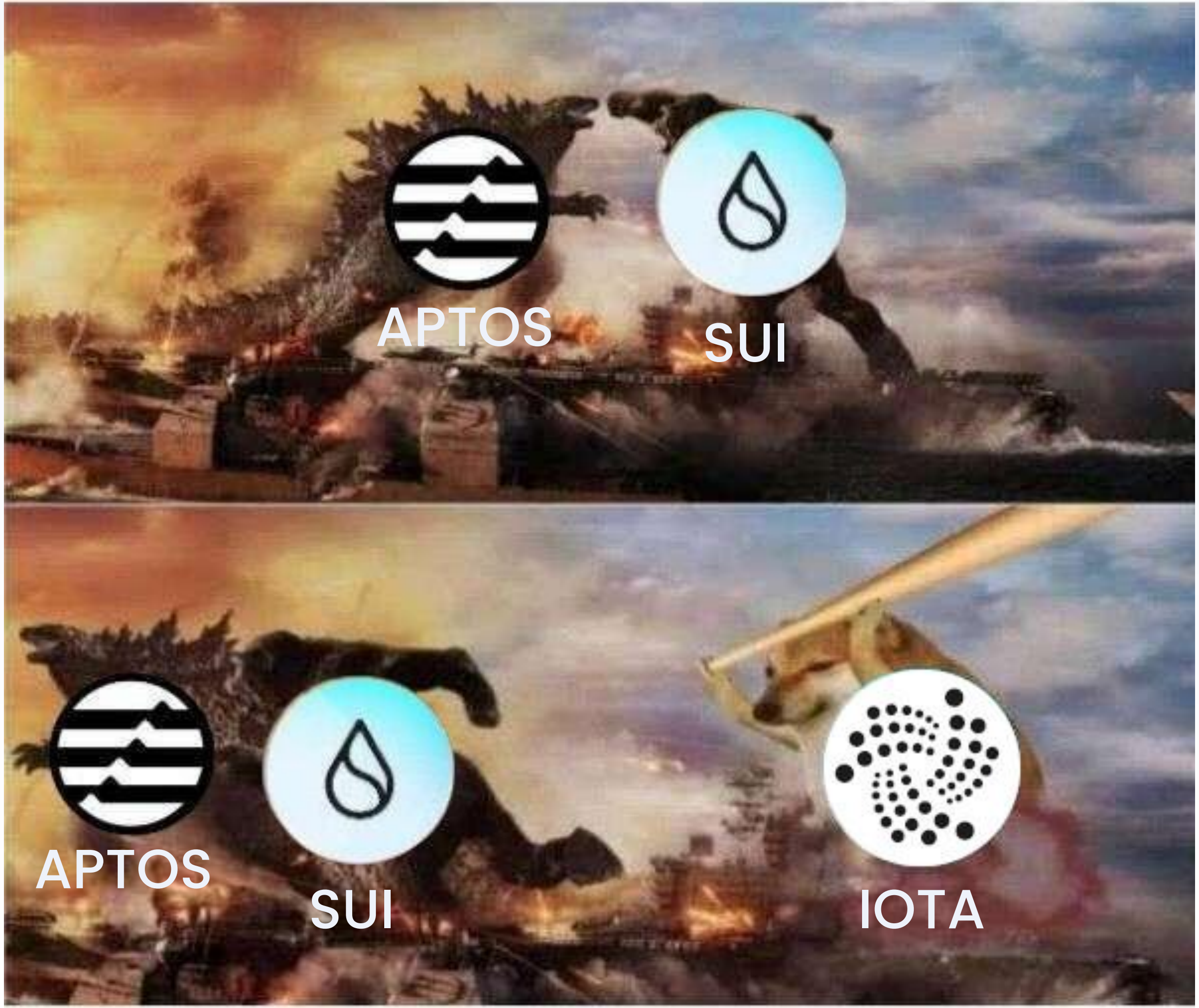


Move on Account vs Object Ledger

- **Unified Memory - Account Based Ledger**: EVM, WASM, ISC, Aptos, Core Move
  - Only sequential* execution
  - Convenient as you can access any memory location without prior request
- **Partitioned Memory - Object Based Ledger**: Sui Move, Cardano, Radix, Stardust, etc.
  - Parallel execution is possible, as **each SC names which objects it will touch**
  - Heavy usage of a particular SC doesn’t degrade others
  - Execution needs only a fraction of the memory
  - UTXO is a special case of the object ledger

Move in Aptos vs Sui

https://academy-public.coinmarketcap.com/srd-optimized-uploads/a60864117eaa4a1b83631cc3cacd53fc.png



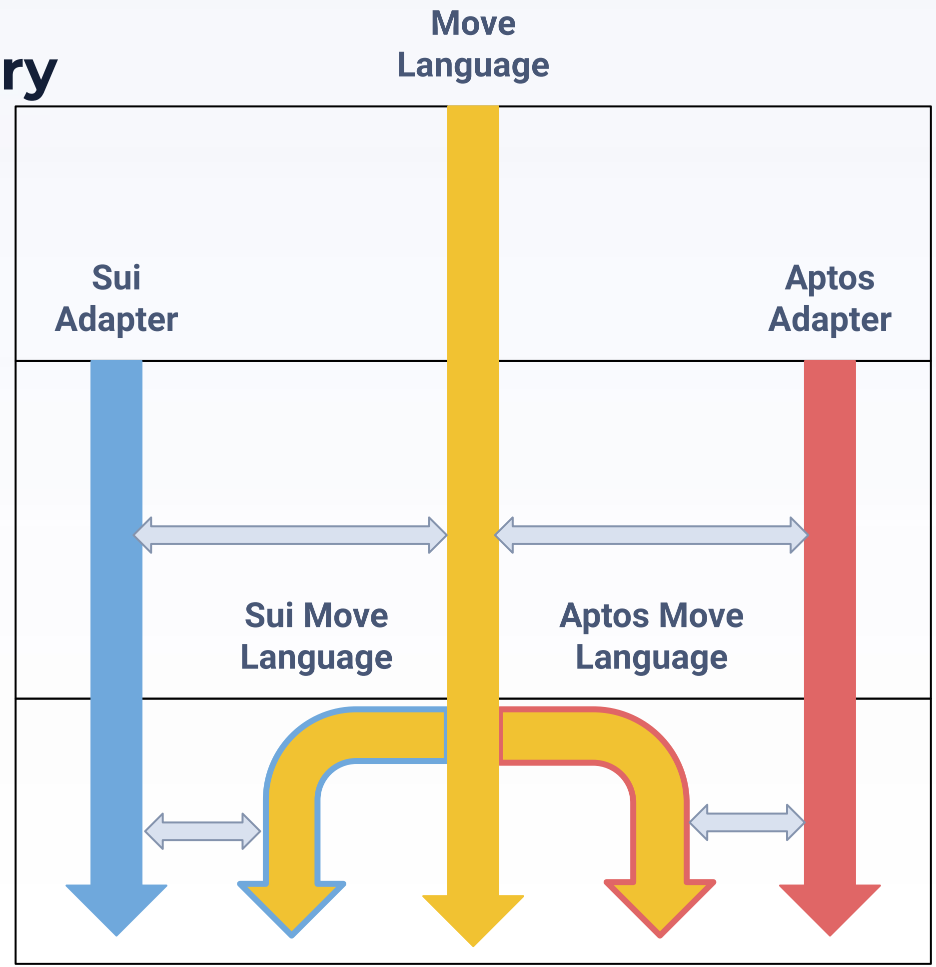
IOTA

Move History

Early Move
Libra/Diem
2018-2021

Move Adapters
Sui & Aptos
2022-2023

Move 2024
Sui & Aptos Forks



IOTA flavored Move

Key differences between (Diem/Aptos) Move and IOTA/Sui Move (1/2)

- **Object-Centric Global Storage**
  - In (Diem) Move, transactions can **freely access resources,** *move_to* and *move_from*.
  - In IOTA Move transaction inputs are *explicitly specified using unique identifiers* for **objects** (as opposed to resources) and **packages** (sets of modules).
- **Addresses Represent Object IDs**
  - IOTA repurposes the address type as a **32-byte identifier** used for both *objects (object id)* and *accounts (address)*.
- **Objects with Key Ability and Globally Unique IDs**
  - In (Diem) Move, the *key ability* indicates that a type is a **resource**, which, along with an account address, can serve as a key in global storage.
  - In IOTA Move, the *key ability* denotes an **object type** and requires the struct's first field to be ***id: UID*** (which becomes the object id).

https://docs.iota.org/developer/iota-101/move-overview/

IOTA

0. Basics - Custom Types

A **structure** in IOTA Move is a *custom type* that contains *key-value pairs*, where the key is the name of a property, and the value is what's stored.

Struct

```
struct Color {
    red: u8,
    green: u8,
    blue: u8,
}
```

https://docs.iota.org/developer/iota-101/objects/object-model

0. Basics - Abilities

- Abilities are keywords in IOTA Move that define **how types behave at the compiler level**
  - ***copy***: the value of this type can be copied
    - usually basic types: Coin is an asset type that should not be duplicated, so it should not have copy ability
  - ***drop***: the value of this type can be automatically destroyed at the end of the scope
    - for types without drop ability, not destroying them manually will cause a compilation error.
  - ***key***: a type that can appear as a key in global storage
  - ***store***: the value of this type can be stored (for example, in another struct)

- Custom types that have the abilities *key* and *store* are considered to be **assets** in IOTA Move.
  - Assets are stored in global storage and can be transferred between accounts.

https://docs.iota.org/developer/getting-started/create-a-module

IOTA

1. Object Basics

- The first field of the **struct** must be the id of the object with type **UID**

Struct

```
struct Color {
    red: u8,
    green: u8,
    blue: u8,
}
```

Object

```
struct ColorObject has key {
    id: UID,
    red: u8,
    green: u8,
    blue: u8,
}
```

https://docs.iota.org/developer/iota-101/objects/object-model

IOTA

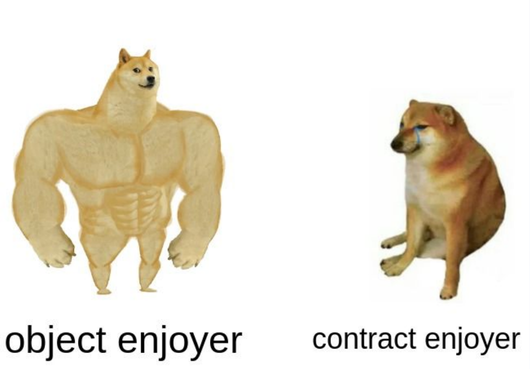
1. Object Basics - Key

- In Move the **key** ability denotes a type that can appear as a key in global storage
- Diem Move uses a **(type, address)-indexed map**
- IOTA Move uses a **map keyed** by **object IDs**.

```
use iota::object::UID;

struct ColorObject has key {
    id: UID,
```

https://docs.iota.org/developer/iota-101/objects/object-model

1. Object Basics - Create an Object



- The only way to create a new UID for a IOTA object is to call **object::new**.

```
use iota::object;
// tx_context::TxContext creates an alias to the TxContext struct in the tx_context module.
use iota::tx_context::TxContext;


fun new(red: u8, green: u8, blue: u8, ctx: &mut TxContext): ColorObject {
    ColorObject {
        id: object::new(ctx),
        red,
        green,
        blue,
    }
}
```

https://docs.iota.org/developer/iota-101/objects/object-model

1. Object Basics - Store an Object

- The constructor puts the object value in a local variable.
- The object can then be placed in persistent global storage.

```
public entry fun create(red: u8, green: u8, blue: u8, ctx: &mut TxContext) {
    let color_object = new(red, green, blue, ctx);
    transfer::transfer(color_object, tx_context::sender(ctx))
}
```

https://docs.iota.org/developer/iota-101/objects/object-model

2. Owned, Shared and Immutable Objects

- Objects in IOTA can have different types of **ownership**, with three categories:
  - **Owned mutable** object -> is owned by an address/object
  - **Shared mutable** object -> anyone can use it in a transaction
  - **Immutable** object -> an object that can't be mutated, transferred or deleted.

- In other blockchains, ***every object is shared***
  - In IOTA Move programmers have the choice to implement a particular use-case using **shared objects, owned objects, or a combination**.

- In IOTA, a transaction that touches a shared object needs to pass through the consensus mechanism. Whilst, a transaction that touches only owned objects does not need it.

https://docs.iota.org/developer/iota-101/objects/shared-owned

2. *Owned*, Shared and Immutable Objects

- **Address Owned object:** *exclusively accessible to their owner*
    - The owner is a 32-byte user address or object ID
    - Does not require consensus to be modified

```
module examples::custom_transfer {
  // Error code for trying to transfer a locked object
  const EObjectLocked: u64 = 0;

  public struct O has key {
    id: UID,
    // An `O` object can only be transferred if this field is `true`
    unlocked: bool
  }

  // Check that `O` is unlocked before transferring it
  public fun transfer_unlocked(object: O, to: address) {
     assert!(object.unlocked, EObjectLocked);
     iota::transfer::transfer(object, to)
  }
}
```

https://docs.iota.org/developer/iota-101/objects/object-ownership/address-owned

2. Owned, *Shared* and Immutable Objects

- **Shared object:** *anyone can read or write this object.*
  - mutable owned objects are single-writer
  - shared objects require to sequence reads and writes

```
/// Init function is often ideal place for initializing
/// a shared object as it is called only once.
fun init(ctx: &mut TxContext) {
    transfer::transfer(ShopOwnerCap {
        id: object::new(ctx)
    }, tx_context::sender(ctx));

    // Share the object to make it accessible to everyone!
    transfer::share_object(DonutShop {
        id: object::new(ctx),
        price: 1000,
        balance: balance::zero()
    })
}
```

https://docs.iota.org/developer/iota-101/objects/object-ownership/shared

2. Owned, Shared and *Immutable* Objects

- Immutable objects have no owner, so anyone can use them without the need for ordering
  - packages are immutable objects
  - you can freeze an initially mutable object

```
public entry fun freeze_object(object: ColorObject) {
    transfer::freeze_object(object)
}
```

https://docs.iota.org/developer/iota-101/objects/object-ownership/immutable

IOTA

3. Using Objects

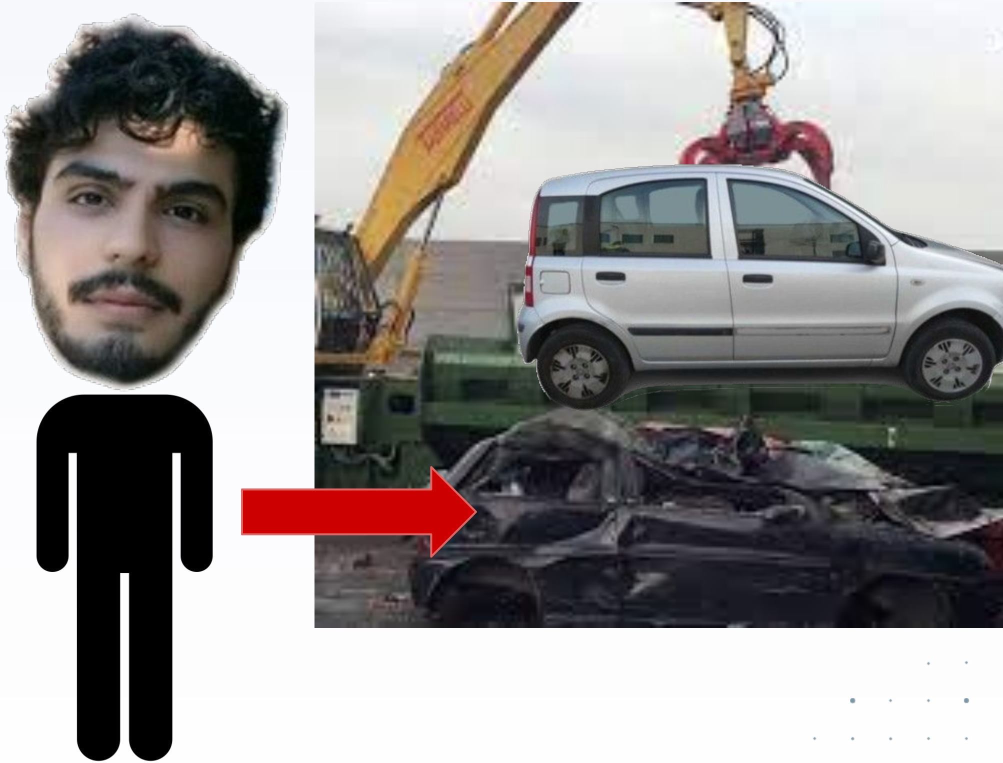
- IOTA Move **authentication mechanisms** ensure ***only you can use objects owned by you*** or ***shared*** in function calls.
- The object can be passed as a parameter to a function in two ways (core Move):
  - Pass by value
    - *ColorObject*
  - Pass by reference
    - *&ColorObject*
    - *&mut ColorObject*

https://docs.iota.org/developer/iota-101/objects/object-ownership/address-owned

Pass a value to a function ***by-value***

Pass a value to a function ***by-value***

Pass a value to a function ***by-value***



3. Using Objects - Pass by Value

- Pass objects by value into an entry function means the **object is moved out of storage**.
- Objects **cannot** be arbitrarily **dropped** and must be either consumed (e.g., transferred) or deleted

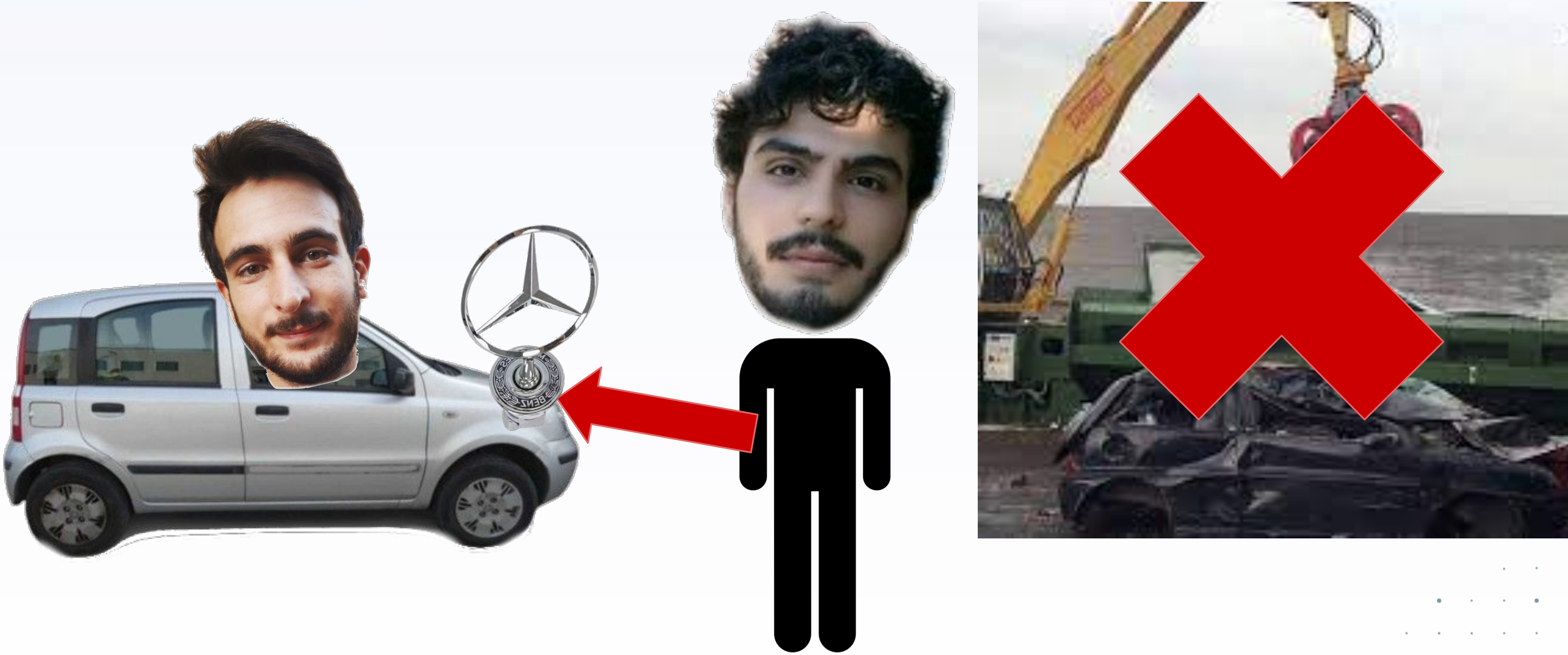
```
    public entry fun delete(object: ColorObject) {
        let ColorObject { id, red: _, green: _, blue: _ } = object;
        object::delete(id);
    }


public entry fun transfer(object: ColorObject, recipient: address) {
    transfer::transfer(object, recipient)
}
```

https://docs.iota.org/developer/iota-101/objects/object-ownership/address-owned

IOTA

“Borrow” a value with ***mutable ref (&mut)***

“Borrow” a value with ***mutable ref (&mut)***



“Borrow” a value with ***read-only ref (&)***

3. Using Objects - Pass by Reference

- **Read-only references** (&) allow you to read data from the object
- **Mutable references** (&mut) allow you to mutate the data in the object.

```
/// Copies the values of `from_object` into `into_object`.
public entry fun copy_into(from_object: &ColorObject, into_object: &mut ColorObject) {
    into_object.red = from_object.red;
    into_object.green = from_object.green;
    into_object.blue = from_object.blue;
}
```

https://docs.iota.org/developer/iota-101/objects/object-ownership/address-owned

IOTA

4. Object Wrapping

- In IOTA Move, you can organize data structs by putting a field of **struct** type in another
- To embed a struct type in an object struct (with a key ability), the struct type must have the **store ability**.

```
struct Wrapping has key {
    id: UID,
    obj: Wrapped,
}

struct Wrapped has key, store {
    value: u64,
}
```

https://docs.iota.org/developer/iota-101/objects/object-ownership/wrapped

IOTA

4. Object Wrapping

- When an object is **wrapped** into another object:
  - it **no longer exists independently** on the ledger; it becomes part of the data of the object that wraps it;
  - is no longer **findable** by its *objectID*;
  - is no longer passable as an argument in transactions procedures calls; the only access point is through the wrapping object (you need to pass this as argument).
- **Unwrapping**
  - you can then take out the wrapped object and transfer it to an address;
  - when an object is unwrapped, it becomes an independent object again;
  - **wrapped objects cannot be unwrapped unless the wrapping object is destroyed**

https://docs.iota.org/developer/iota-101/objects/object-ownership/wrapped

4. Object Wrapping

```
struct ObjectWrapper has key {
    id: UID,
    original_owner: address,
    to_swap: Object,
}
public entry fun request_swap(object: Object, service_address: address, ctx:
    let wrapper = ObjectWrapper {
        id: object::new(ctx),
        original_owner: tx_context::sender(ctx),
        to_swap: object,
    };
    transfer::transfer(wrapper, service_address);
}
 public entry fun execute_swap(wrapper1: ObjectWrapper, wrapper2: ObjectWrap
    // Unpack both wrappers, cross send them to the other owner.
    let ObjectWrapper {
        id: id1,
        original_owner: original_owner1,
        to_swap: object1,
    } = wrapper1;

    let ObjectWrapper {
        id: id2,
        original_owner: original_owner2,
        to_swap: object2,
    } = wrapper2;

    // Perform the swap.
    transfer::transfer(object1, original_owner2);
    transfer::transfer(object2, original_owner1);
}
```

https://docs.iota.org/developer/iota-101/objects/object-ownership/wrapped

5. Dynamic Fields

- IOTA Move provides **dynamic fields** with arbitrary *names*, added and removed on-the-fly (not fixed at publish), which can store heterogeneous values.

- This approach overcomes the following limitations:
  - Object's have a finite set of fields, fixed when its module is declared.
  - Objects can become very large if they wrap several other objects (high gas fees).
  - It is not possible to store a collection of objects (e.g., vector) of heterogeneous types.

https://docs.iota.org/developer/iota-101/objects/dynamic-fields/

5. Dynamic Fields - Add field

- This function takes the **Child object** *by value* and makes it a *dynamic field* of the **Parent object** with **name *b"child"***;
    - sender address owns the Parent object;
    - the Parent object owns the Child object, and can refer to it by the name *b"child"*.

```
use iota::dynamic_object_field as ofield;

public fun add_child(parent: &mut Parent, child: Child) {
    ofield::add(&mut parent.id, b"child", child);
}
```

https://docs.iota.org/developer/iota-101/objects/dynamic-fields/

5. Dynamic Fields - Access field

```
use iota::dynamic_object_field as ofield;

public fun mutate_child(child: &mut Child) {
    child.count = child.count + 1;
}

public fun mutate_child_via_parent(parent: &mut Parent) {
    mutate_child(ofield::borrow_mut(
        &mut parent.id,
        b"child",
    ));
}
```

https://docs.iota.org/developer/iota-101/objects/dynamic-fields/

5. Dynamic Fields - Remove field

```
use iota::dynamic_object_field as ofield;

public fun delete_child(parent: &mut Parent) {
    let Child { id, count: _ } = reclaim_child(parent);

    object::delete(id);
}

public fun reclaim_child(parent: &mut Parent, ctx: &mut TxContext): Child {
    ofield::remove(
        &mut parent.id,
        b"child",
    );

}
```

https://docs.iota.org/developer/iota-101/objects/dynamic-fields/

6. Transfer to Object

- *Transfer objects to an object ID* works in the **same way as an object transfer to an address** (using the same functions)

- Transfering an object to another object means establishing a form of **parent-child** authentication relationship.
  - Objects transferred to another object can be **received** by the owner of the parent object.
  - The **parent** (receiving) object **module defines the access control** for receiving a child obj.

```
// Transfers the object `b` to the address 0xADD
iota::transfer::public_transfer(b, @0xADD);

// Transfers the object `c` to the object with object ID 0x0B
iota::transfer::public_transfer(c, @0x0B);
```

https://docs.iota.org/developer/iota-101/objects/transfers/transfer-to-object

IOTA

6. Transfer to Object - Receive

- After an object *c* has been sent to another object *p*, *p* must then receive *c* to do anything with it.
- The module of the type of *p* defines access control policies and other restrictions on *c*

```
/// This function will receive a coin sent to the `Account` object and then
/// join it to the balance for each coin type.
/// Dynamic fields are used to index the balances by their coin type.
public fun accept_payment<T>(account: &mut Account, sent: Receiving<Coin<T>>) {
    // Receive the coin that was sent to the `account` object
    // Since `Coin` is not defined in this module, and since it has the `store`
    // ability we receive the coin object using the `transfer::public_receive` function.
    let coin = transfer::public_receive(&mut account.id, sent);
    let account_balance_type = AccountBalance<T>{};
    let account_uid = &mut account.id;

    // Check if a balance of that coin type already exists.
    // If it does then merge the coin we just received into it,
    // otherwise create new balance.
    if (df::exists_(account_uid, account_balance_type)) {
        let balance: &mut Coin<T> = df::borrow_mut(account_uid, account_balance_type);
        coin::join(balance, coin);
    } else {
        df::add(account_uid, account_balance_type, coin);
    }
}
```

https://docs.iota.org/developer

7. One-Time Witness (OTW)

- Special type guaranteed to have **at most one instance**: useful for limiting certain actions to only happen once (e.g., creating a coin). The only instance is passed to its module's init function when its package is published. In Move, a type is considered a OTW if:
  - Its name is the **same as its module's names**, all **uppercased**.
  - It has **ONLY** the **drop ability**
  - It has **no fields**, or a single bool field.

```
module examples::mycoin {

    /// Name matches the module name
    struct MYCOIN has drop {}

    /// The instance is received as the first argument
    fun init(witness: MYCOIN, ctx: &mut TxContext) {
        /* ... */
    }
}
```

https://docs.iota.org/developer/iota-101/move-overview/one-time-witness

IOTA

8. Generics

- Generics are **abstract stand-ins *for concrete types*** or other properties.

```
struct Box<T> {
    value: T
}
```

- **Conditions** to enforce that the type passed into the generic *must have certain abilities*.

```
// T must be copyable and droppable
struct Box<T: store + drop> has key, store {
    value: T
}
```

- Using generics in functions

```
public fun create_box<T>(value: T): Box<T> {
        Box<T> { value }
    }
```

```
// value will be of type storage::Box<bool>
    let bool_box = storage::create_box<bool>(true);
// value will be of the type storage::Box<u64>
    let u64_box = storage::create_box<u64>(1000000);
```

https://intro.sui-book.com/unit-three/lessons/2_intro_to_generics.html#intro-to-generics

9. Hot Potato Pattern

1. This pattern requires that **function B** must be called ***immediately after*** **function A**, when **function A** returns a **hot potato** and **function B** consumes it.
2. Flash loan:
   a. create a **`Receipt` struct** that
      - cannot be discarded because it does not have `drop`,
      - cannot be put in persistent storage because it does not have `key`,
      - cannot be transferred or wrapped because it does not have `store`.

   b. Have a **`loan`** function that requests a loan of `amount` from `lender` and returns the **`Receipt`**

   c. the only way to get rid of it is to call **`repay`** at some point forcing to pay back the debt.

https://docs.iota.org/developer/iota-101/move-overview/patterns/hot-potato

10. Capability Pattern

- This pattern enables the **authorization of specific actions with an object**.
  - e.g., the UpgradeCap is used to authorize the upgrading of packages.
  - e.g. the TreasuryCap grants the authority to manage a Coin treasury functions.

```
/// Type representing the capability to create new `Item`s.
public struct AdminCap has key { id: UID }

/// Custom NFT-like type representing an item.
public struct Item has key, store { id: UID, name: String }

/// Module initializer, called once during the module's deployment.
/// This function creates a single instance of `AdminCap` and assigns it to the publisher.
fun init(ctx: &mut TxContext) {
    transfer::transfer(AdminCap {
        id: object::new(ctx)
    }, tx_context::sender(ctx))
}

/// Function to create a new `Item`. It requires `AdminCap` to authorize the action.
public fun create_item(_: &AdminCap, name: String, ctx: &mut TxContext): Item {
    let item = Item {
        id: object::new(ctx),
        name,
    };
    item
```

https://docs.iota.org/developer/iota-101/move-overview/patterns/capabilities

Interacting with a IOTA Move Module

0. Write a IOTA Move Package

```
module my_first_package::my_module {

    // Imports
    use iota::object::{Self, UID};
    use iota::transfer;
    use iota::tx_context::{Self, TxContext};

    // Struct definitions
    struct Sword has key, store {
        id: UID,
        magic: u64,
        strength: u64,
    }

    struct Forge has key, store {
        id: UID,
        swords_created: u64,
    }

    // Module initializer to be executed when this module is published
    fun init(ctx: &mut TxContext) {
        let admin = Forge {
            id: object::new(ctx),
            swords_created: 0,
        };
        // Transfer the forge object to the module/package publisher
        transfer::public_transfer(admin, tx_context::sender(ctx));
    }

    // Accessors required to read the struct attributes
    public fun magic(self: &Sword): u64 {
        self.magic
    }

    public fun strength(self: &Sword): u64 {
        self.strength
    }

    public fun swords_created(self: &Forge): u64 {
        self.swords_created
    }

    // Public/entry functions

    // Private functions
}
```

https://docs.iota.org/developer/getting-started/create-a-module

1. Build and Publish a IOTA Move Package

```
$ iota move build
$ iota move test
$
$
$ iota client publish --gas-budget 5000000
```

```
#[test]
public fun test_sword() {
    // Create a dummy TxContext for testing.
    let mut ctx = tx_context::dummy();

    // Create a sword.
    let sword = Sword {
        id: object::new(&mut ctx),
        magic: 42,
        strength: 7,
    };

    // Check if accessor functions return correct values.
    assert!(magic(&sword) == 42 && strength(&sword) == 7, 1);
}
```

https://docs.iota.org/developer/getting-started/build-test
https://docs.iota.org/developer/getting-started/publish

IOTA

2. Interact with a Package

- Now that the package is on chain you can use the `$ iota client call` command to make individual calls to package functions

```
iota client call \
--package
0x83a30c4c3cbdd33068d36fc18d1f097f9196b79a475b7fe69f517063b376dd23 \
--module luckyplumber \
--function get_mad \
--type-args
0xd95b4510206e13fbe9413bc61183ac3b8375c8971adc54c81eeb9c96d61b5ff1::btfa
::BTFType \
--args 44
0x59f9ed7d8f7c7ed490a63e572c87705e23667570564251e3a20ceedf9c8f961d
--gas-budget 50000000 \
```

https://docs.iota.org/references/cli/client

2. Interact with a Package - PTB

- You can construct more advanced blocks of transactions using the `$ iota client ptb` command.

- In general, transactions on IOTA are composed of:
    - a number of **commands**
    - that execute on **inputs**
    - to define some **results**

https://docs.iota.org/references/cli/ptb

IOTA

3. Programmable Transaction Blocks

- The **inputs value** of a PTB is value is a vector of arguments, either *objects* or *pure values*
- The **commands value** of a PTB is a vector of commands using *inputs* or *results* to execute code
  - *TransferObjects* sends (one or more) objects to a specified address
  - *SplitCoins* splits off (one or more) coins from a single coin. It can be any iota::coin::Coin<_>
  - *MergeCoins* merges (one or more) coins into a single coin
  - *MakeMoveVec* creates a vector of Move values
  - ***MoveCall*** invokes either an *entry* or a *public* Move function in a published package.
  - *Publish* creates a new package and calls the init function of each module in the package.
  - *Upgrade* upgrades an existing package.
- The **result values** is a vector of values that can be produced by each command; the type of the value can be any arbitrary Move type, not limited to objects or pure values.
- A PTB can perform up to 1,024 unique operations in a single execution.

https://docs.iota.org/developer/iota-101/transactions/ptb/prog-txn-blocks

3. Programmable Transaction Blocks

```
$ iota client ptb \
--move-call 0xd95b4510206e13fbe9413bc61183ac3b8375c8971adc54c81eeb9c96d61b5ff1::pkg::func
"<0xd95b4510206e13fbe9413bc61183ac3b8375c8971adc54c81eeb9c96d61b5ff1::pkg1::TYPE1,0xd95b451
0206e13fbe9413bc61183ac3b8375c8971adc54c81eeb9c96d61b5ff1::pkg2::TYPE2>"
@0x0b72fb4d8106699c773bf58fd0a49ffe3a08bdd58f245946d160ed5463f7ba47 99 true \
--assign result_variable \
--move-call iota::tx_context::sender \
--assign sender \
--transfer-objects "[result_variable.2]" sender \
--move-call 0xd95b4510206e13fbe9413bc61183ac3b8375c8971adc54c81eeb9c96d61b5ff1::pkg::func2
"<0xd95b4510206e13fbe9413bc61183ac3b8375c8971adc54c81eeb9c96d61b5ff1::pkg1::TYPE1"
@0x0b72fb4d8106699c773bf58fd0a49ffe3a08bdd58f245946d160ed5463f7ba47 result_variable.0 \
--gas-budget 50000000
```

https://docs.iota.org/references/cli/ptb

4. *public* vs *entry* functions

- The **public** modifier allows a function to be *called from a PTB* and also *from other modules*
  - NO restrictions on parameters

- The ***entry*** modifier allows a function to be called directly from a PTB as a module "entrypoint".
  - entry functions **parameters must be inputs** to the PTB (not results of previous command)
  - only allowed to return types that have drop

- Use the *entry* modifier when:
  - You want strong guarantees that your function is not being combined with third-party module functions (e.g., swap protocol that does not want a flash loan)
  - *public* function signatures must be maintained by upgrades (entry function not).
  - It is also possible to create a *public entry* function, can be called by other modules

https://docs.iota.org/developer/iota-101/move-overview/entry-functions

IOTA

5. Binary Canonical Serialization (BCS)

- BCS is a **serialization format** developed in the context of the Diem blockchain
  - now extensively used in most of the blockchains based on Move (IOTA, Sui, Aptos, 0L).
- BCS is *not only used in the Move VM*, but also used in **transaction and event coding**.

```
var { bcs, fromHEX } = require('@mysten/bcs');
const Calzone = bcs.struct('Calzone', {
    flour: bcs.u16(),
    tomato_sauce: bcs.u16(),
    cheese: bcs.u16(),
});
const hex = "0a00030062027201120 0c800b4000000"
const calzone = Calzone.parse(fromHEX(hex));
```

https://docs.iota.org/references/ts-sdk/bcs
https://github.com/zefchain/bcs
https://www.npmjs.com/package/@mysten/bcs

What’s left?

- Collections
- Events
- Package upgrades
- Proper Testing
- Clock and Random objects
- …

- https://docs.iota.org/developer/iota-101/move-overview/
- https://docs.iota.org/references/cli/client
- https://intro.sui-book.com/unit-one/lessons/1_set_up_environment.html

IOTA

Thank you!

Mirko Zichichi

Applied Research Engineer, IOTA Foundation
mirko.zichichi@iota.org

IOTA

The code and documentation must NOT be shared outside!