Università Ca' Foscari Venezia

Resource-aware Programming for the Algorand Platform

L. Benetollo, M. Bugliesi, S. Crafa, S. Rossi, A. Spanò

Meeting PRIN NiRvAna 2025

Urbino, May 29-31

Contents

- **Linear Types** for Resource-aware Programming
- The **Move** language
- Porting Move to the **Algorand** blockchain

Resource-aware Programming

- Smart contracts generally manipulate data representing valuable objects (assets, currency etc.)
- It should not be possible to duplicate or lose an asset

```
module Marketplace

   struct Order {
      customerId : address
      productId  : uint
   }

   fun buyProduct(productId, money) {
      let order = Order { … }
      processOrder(order)
      deposit(money)
   }
```

VALUABLE

```
   fun processOrder(order) {
      save(order)
   }
```

OK

Resource-aware Programming

- Smart contracts generally manipulate data representing valuable objects (assets, currency etc.)
- It should not be possible to duplicate or lose an asset

```
module Marketplace

   struct Order {
      customerId : address
      productId  : uint
   }

   fun buyProduct(productId, money) {
      let order = Order { … }
      processOrder(order)
      deposit(money)
   }
```

VALUABLE

```
fun processOrder(order) {
   save(order)
}
```

OK

1

```
fun processOrder(order) {
   if isAvailable(order.productId)
      save(order)

}
```

DROP ERROR

Resource-aware Programming

- Smart contracts generally manipulate data representing valuable objects (assets, currency etc.)
- It should not be possible to duplicate or lose an asset

```
module Marketplace

   struct Order {
      customerId : address
      productId  : uint
   }

   fun buyProduct(productId, money) {
      let order = Order { … }
      processOrder(order)
      deposit(money)
   }
```

VALUABLE

```
fun processOrder(order) {
   save(order)
}
```

OK

1

```
fun processOrder(order) {
   if isAvailable(order.productId)
      save(order)

}
```

DROP ERROR

2

```
fun processOrder(order) {
   if isAvailable(order.productId)
      save(order)
   save(order)
}
```

COPY ERROR

Resource-aware Programming

- Smart contracts generally manipulate data representing valuable objects (assets, currency etc.)
- It should not be possible to duplicate or lose an asset

```
module Marketplace

   struct Order {
      customerId : address
      productId  : uint
   }

   fun buyProduct(productId, money) {
      let order = Order { … }
      processOrder(order)
      deposit(money)
   }
```

VALUABLE

Forgetting to deposit the money is also an error.

```
fun processOrder(order) {
   save(order)
}
```

OK

A compile-time check would be great!

1

```
fun processOrder(order) {
   if isAvailable(order.productId)
      save(order)

}
```

DROP ERROR

2

```
fun processOrder(order) {
   if isAvailable(order.productId)
      save(order)
   save(order)
}
```

COPY ERROR

Linear Types in Move

- The Move language supports linear types
- User-defined datatypes can be *tagged* with **capabilities**

```
struct Copiable has copy {
   /* fields */
}
```

Can be **copied**, not dropped

```
struct Droppable has drop {
   /* fields */
}
```

Can be **dropped**, not copied

```
struct Normal has copy, drop {
   /* fields */
}
```

Can be **copied** *and* **dropped**

```
struct Linear {
   /* fields */
}
```

Cannot be copied or dropped

Assets are Linear datatypes

- Non-copiable datatypes can only be **moved** through scopes
- Prevents **double-spending** at compile-time

```
struct Coin {
    amount : u64
}

public fun mint(n : u64) { Coin { amount: n } }
public fun spend(c : Coin) { /* spend money somehow */ }
```

integers are copiable but the enclosing type is not

- Other modules cannot access fields (Information Hiding)

```
let c = mint(1000);
spend(c);
spend(c);
```

argument ‘c’ is **moved**

ERROR : variable ‘c’ does not exist anymore because it has been moved

The drop capability

- A **drop** can happen in two sites:
  - Assignment
  - End of scope
- Disabling drop **avoids asset loss**

```
let x = 8;
x = 9;
```

Value 8 is dropped when left-value is overwritten

```
{
   let x = 8;
}
```

Value 8 is dropped at the end of the scope

Introducing AlgoMove

And now let’s introduce…

AlgoMove

A Move embedding for Algorand

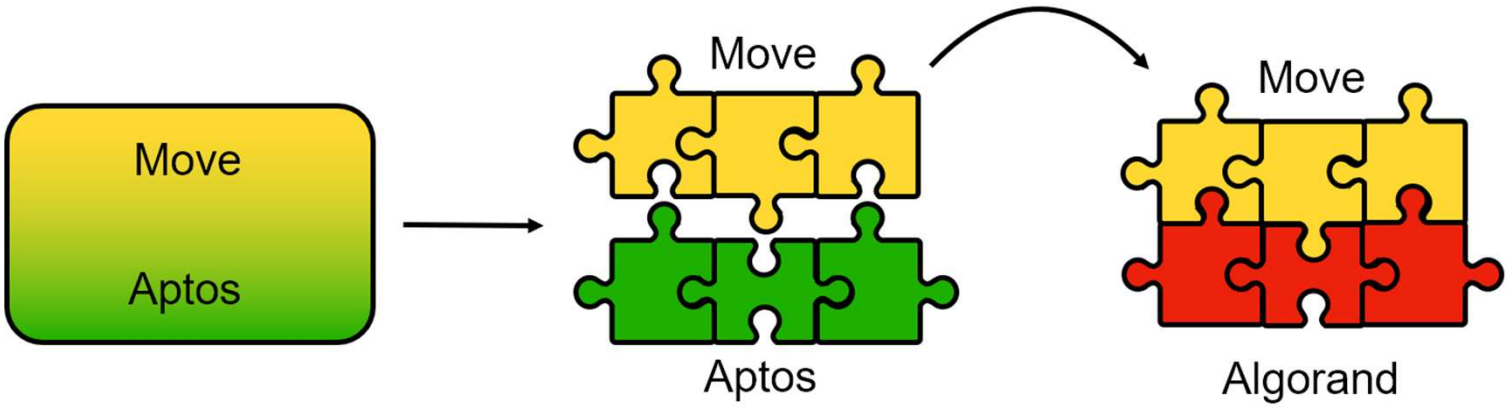
Why Algorand?

PRO

- Great security features
- Safe transaction system
- Efficient

CONS

- No high-level languages
- No static properties
- All checks at runtime



TEAL/PyTeal

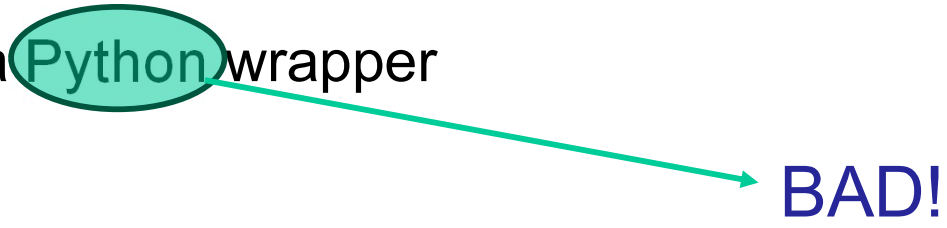
Programming Algorand is no easy task

- TEAL is a low-level assembly-like language
- PyTeal is a Python wrapper

TEAL/PyTeal

Programming Algorand is no easy task

- TEAL is a low-level assembly-like language
- PyTeal is a Python wrapper



No typing or static checks!

PyTeal Example

```
class S:
    val = LocalStateValue(
        stack_type=pt.TealType.uint64,
    )

app = Application("S", state=S())

@app.external
def set(new_val: pt.abi.Uint64) -> pt.Expr:
    return app.state.val.set(new_val.get())

@app.external
def get(*, output: pt.abi.Uint64) -> pt.Expr:
    return output.set(app.state.val.get())

@app.external
def update(new_val: pt.abi.Uint64) -> pt.Expr:
    return app.state.val.set(new_val.get())

@app.opt_in
def opt_in() -> pt.Expr:
    return app.initialize_local_state()
```

Local state definiton

Local state set

Local state get

Local state update

A nightmare!

Move Example

Move is an *elegant* language instead:

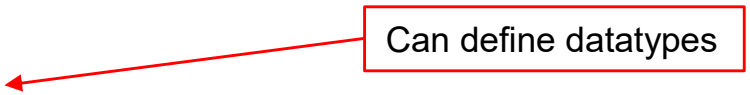
```
struct Coin {
   amount : u64
}


public fun deposit(account: &signer, value: u64) {
    let coins = Coin { amount: value };
    move_to(account, coins)
}


public fun withdraw(account: &signer, value: u64): Coin {
    let Coin { amount: total } = move_from<Coin>(account);
    deposit(account, total - value);
    Coin { amount: value }
}
```

Move Example

Move is an *elegant* language instead:




```
struct Coin {
    amount : u64
}
```

```
public fun deposit(account: &signer, value: u64) {
    let coins = Coin { amount: value };
    move_to(account, coins)
}
```

```
public fun withdraw(account: &signer, value: u64): Coin {
    let Coin { amount: total } = move_from<Coin>(account);
    deposit(account, total - value);
    Coin { amount: value }
}
```

Move Example

Move is an *elegant* language instead:

Can define datatypes

Functions have understandable signatures

```
struct Coin {
    amount : u64
}


public fun deposit(account: &signer, value: u64) {
    let coins = Coin { amount: value };
    move_to(account, coins)
}


public fun withdraw(account: &signer, value: u64): Coin {
    let Coin { amount: total } = move_from<Coin>(account);
    deposit(account, total - value);
    Coin { amount: value }
}
```

Move Example

Move is an *elegant* language instead:

```
struct Coin {
    amount : u64
}


public fun deposit(account: &signer, value: u64) {
    let coins = Coin { amount: value };
    move_to(account, coins)
}


public fun withdraw(account: &signer, value: u64): Coin {
    let Coin { amount: total } = move_from<Coin>(account);
    deposit(account, total - value);
    Coin { amount: value }
}
```

Can define datatypes

Functions have understandable signatures

Functional-style constructors

Move Example

Move is an *elegant* language instead:

```
struct Coin {
    amount : u64
}
```

Can define datatypes

```
public fun deposit(account: &signer, value: u64) {
    let coins = Coin { amount: value };
    move_to(account, coins)
}
```

Functions have understandable signatures

Functional-style constructors

Store stuff on the blockchain

```
public fun withdraw(account: &signer, value: u64): Coin {
    let Coin { amount: total } = move_from<Coin>(account);
    deposit(account, total - value);
    Coin { amount: value }
}
```

Move Example

Move is an *elegant* language instead:

```
struct Coin {
    amount : u64
}
```

Can define datatypes

```
public fun deposit(account: &signer, value: u64) {
    let coins = Coin { amount: value };
    move_to(account, coins)
}
```

Functions have understandable signatures

Functional-style constructors

Store stuff on the blockchain

```
public fun withdraw(account: &signer, value: u64): Coin {
    let Coin { amount: total } = move_from<Coin>(account);
    deposit(account, total - value);
    Coin { amount: value }
}
```

Retrieve stuff from the blockchain

Move Example

Move is an *elegant* language instead:

```
struct Coin {
    amount : u64
}


public fun deposit(account: &signer, value: u64) {
    let coins = Coin { amount: value };
    move_to(account, coins)
}


public fun withdraw(account: &signer, value: u64): Coin {
    let Coin { amount: total } = move_from<Coin>(account);
    deposit(account, total - value);
    Coin { amount: value }
}
```

Can define datatypes

Functions have understandable signatures

Functional-style constructors

Store stuff on the blockchain

Retrieve stuff from the blockchain

Supports pattern matching

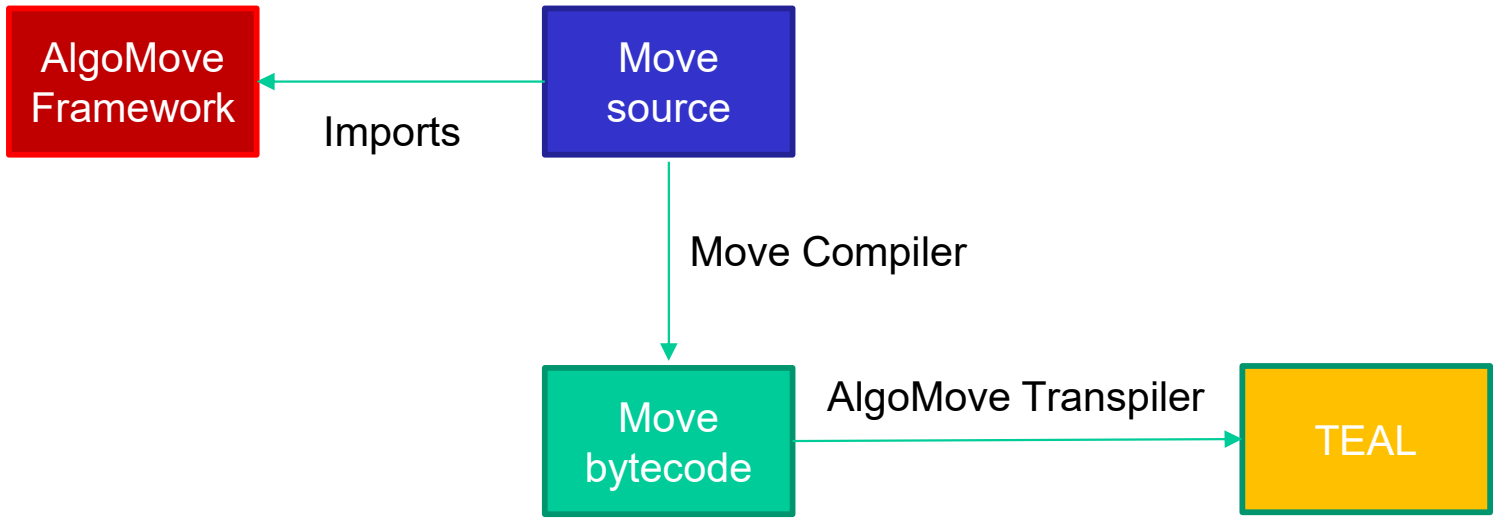
The AlgoMove Architecture

So what is AlgoMove exactly?

A Framework

+

A Transpiler

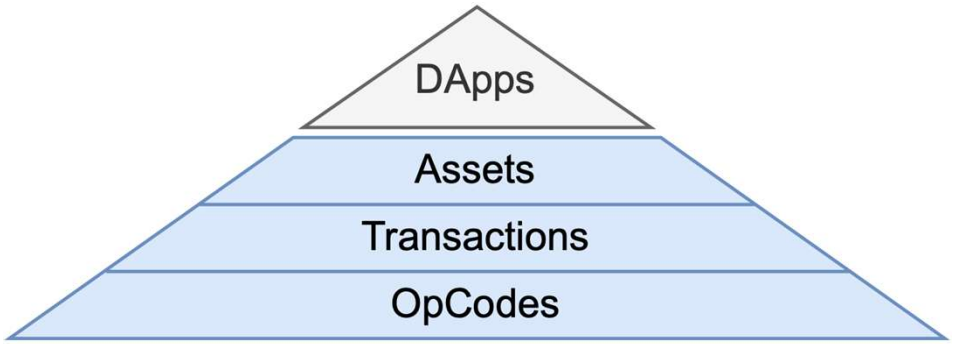


The AlgoMove Framework

The ***Assets*** Layer: to create and manipulate assets and valuables

The ***Transactions*** Layer: to perform transactions and inner transaction

The ***OpCodes*** Layer: low-level access to TEAL instructions



The AlgoMove Transpiler

Move Source

```
struct S has key {
  x: u64,
  y: u64
}

fun sel(acc: address, n: u64) acquires S {
  let s = borrow_global<S>(acc);
  let ry: &u64 = &s.y;
  let rn: &mut u64 = &mut n;
  *rn = *ry + *rn
}
```

Move Bytecode

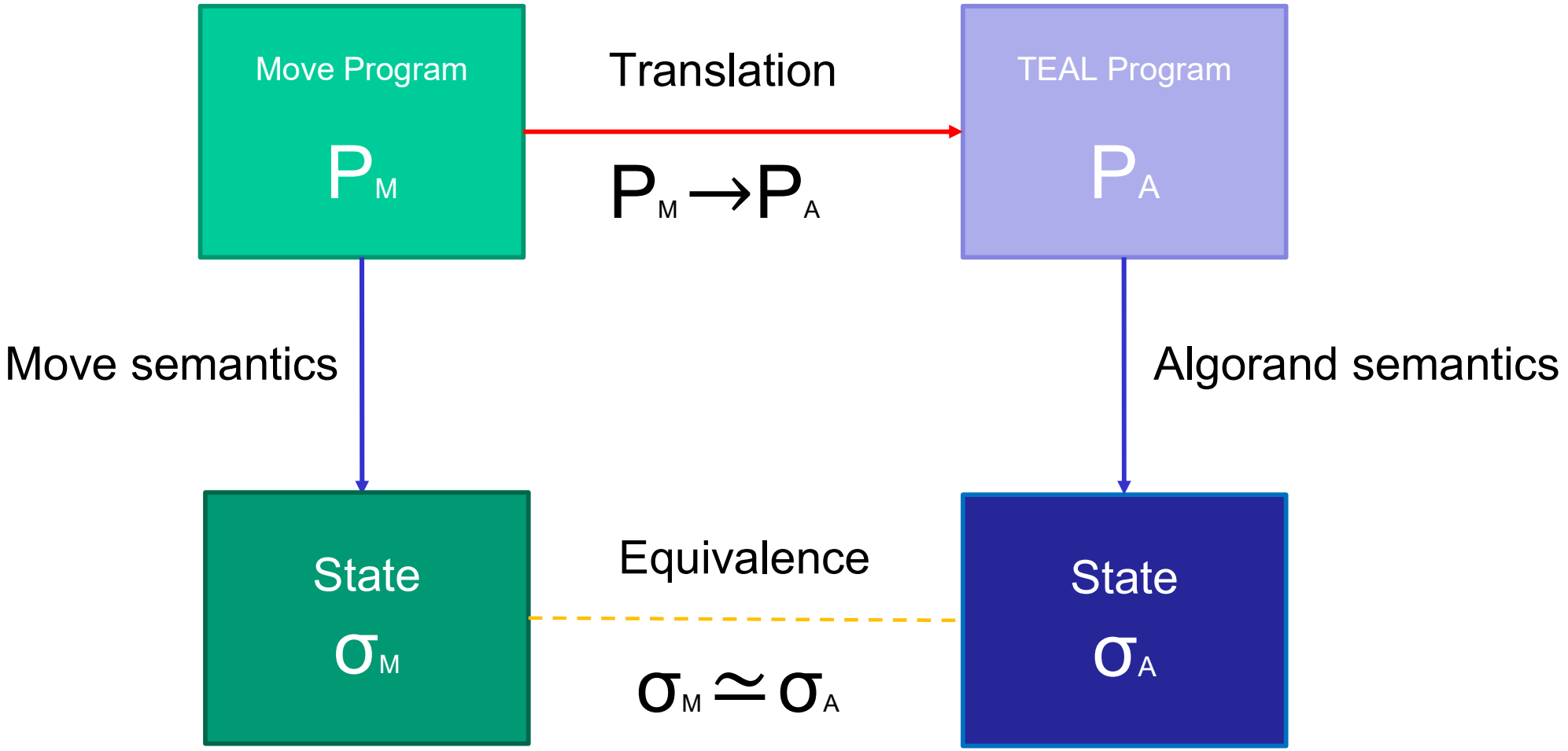
```
sel(Arg0: address,
    Arg1: u64): u64

0: MoveLoc[0]
1: ImmBorrowGlobal[0](S)
2: ImmBorrowField[0](S.y)
3: StLoc[3]
4: MutBorrowLoc[1]
5: StLoc[2]
6: MoveLoc[3]
7: ReadRef
8: CopyLoc[2]
9: ReadRef
10: Add
11: MoveLoc[2]
12: WriteRef
13: Ret
```

TEAL

```
sel:
   proto 2 1
   load 0
   load 1
   load 2
   load 3
   frame_dig -1
   store 0
   frame_dig -2
   store 1
0: load 0
1: pushbytes 0x00 0x00
   swap
   concat
   // 00 00 Arg0 [34B]
2: pushbytes 0x01 0x00
   replace2 0
   pushbytes 0x0008 0x0008
   concat
   // 01 00 Arg0 0008 0008 [38B]
3: store 2
4: pushbytes 0x03 0x01
   // 03 01 [2B]
5: store 3
6: load 2
7: callsub read_ref
8: load 3
9: callsub read_ref
10: +
11: load 2
12: callsub write_ref
13: cover 4
    store 3
    store 2
    store 1
    store 0
    retsub
```

Soundness of Translation



Thank you.