Università degli Studi di Padova

Università Ca' Foscari Venezia

Linear Typing for Asset-aware Programming

M. Bugliesi, S. Crafa, G. Dal Sasso, S. Rossi, A. Spanò

Meeting PRIN NiRvAna 2024

Udine, June 6-8

Valuables

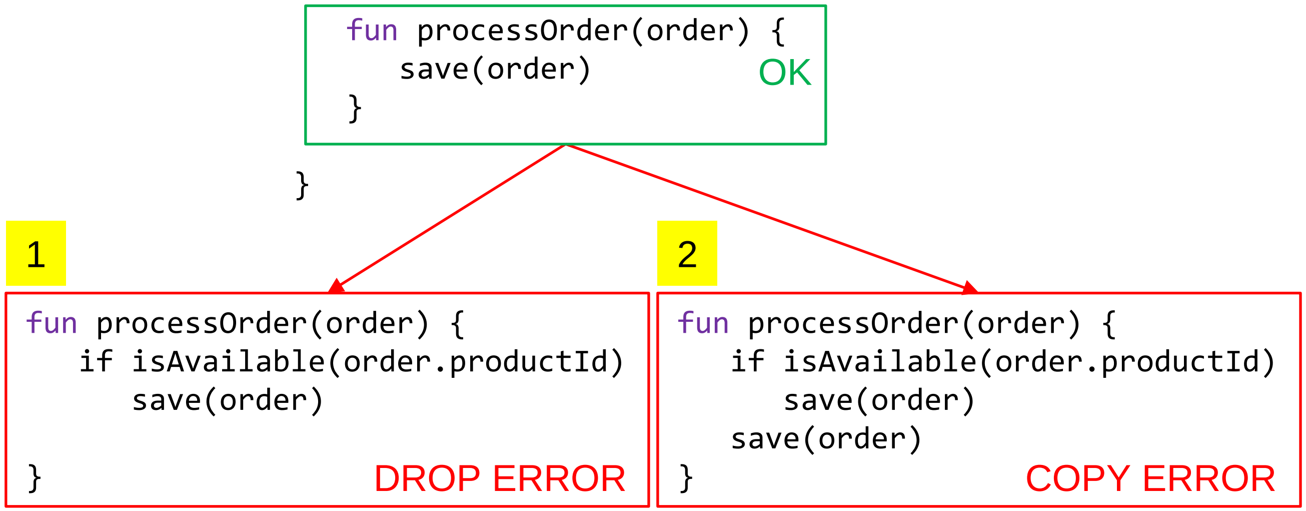
- Smart contracts generally manipulate data representing valuable objects (assets)
- It should not be possible to duplicate or lose an asset

```
module Marketplace {

   struct Order {
      customerId : Address,
      productId  : String,
      date       : Timestamp,
      priority   : Integer,
      …
   }

   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)
}
```

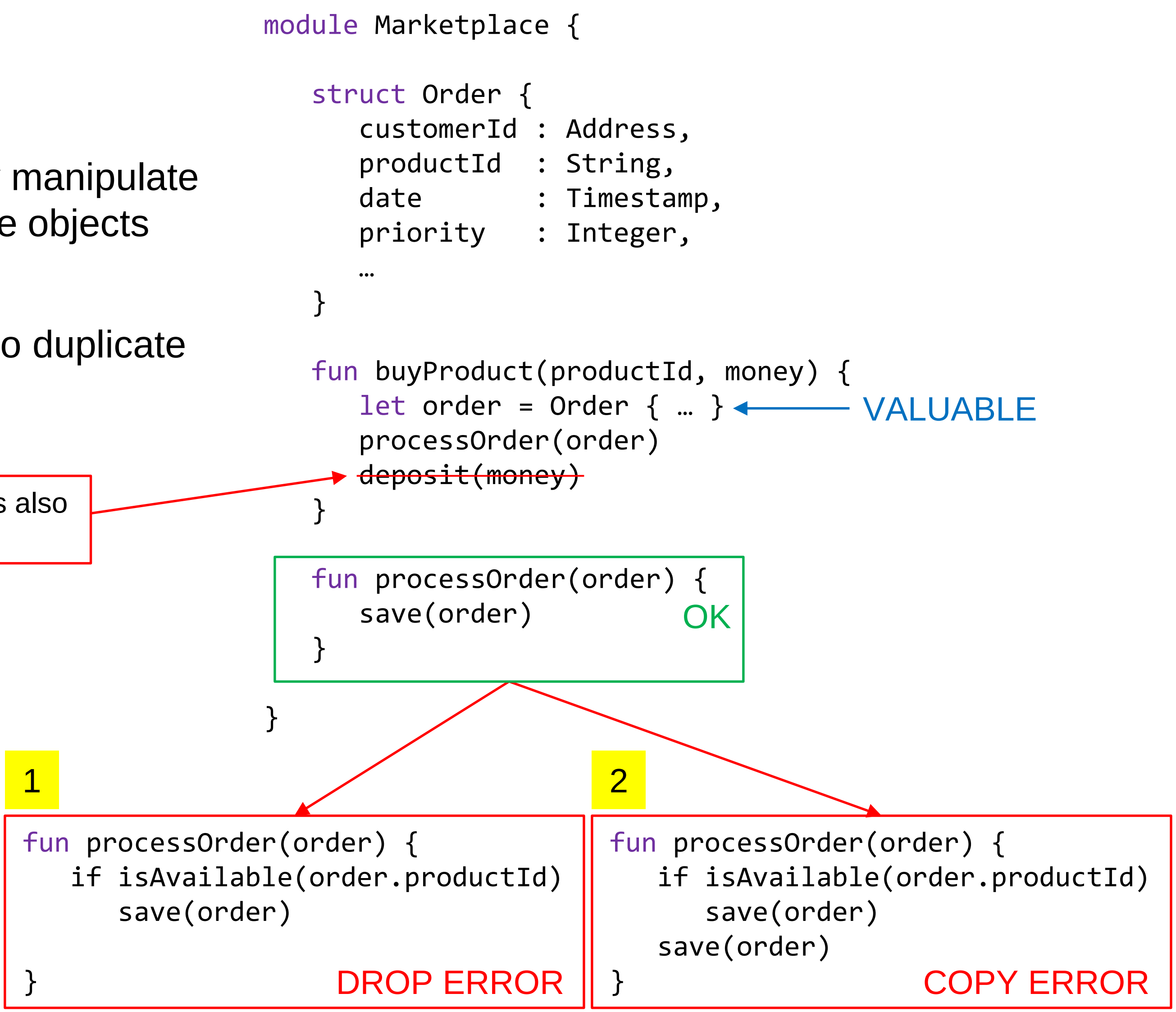
COPY ERROR

Valuables

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

Forgetting to deposit the money is also an error.

A compile-time check would be useful in asset-aware programming!



```
module Marketplace {

   struct Order {
      customerId : Address,
      productId  : String,
      date       : Timestamp,
      priority   : Integer,
      …
   }

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

   fun processOrder(order) {
      save(order)
   }

}
```

VALUABLE

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

Linear Types in Move

- 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

Moving Linear Datatypes

- Non-copiable data 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 struct 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 Ability

- 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 replaced

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

Value 8 is dropped at the end of the scope

Formalization of Move

$$\begin{array}{llll}
\text{Value } v & ::= & n & \text{integer} \\
 & | & \underline{\texttt{struct}\ \{k\}\ \mathrm{M.S}\,[\,\overline{v}\,]} & \text{struct value} \\
 & & & k \in K
\end{array}$$

$$\begin{array}{llll}
\text{Type T} & ::= & \mathrm{Int} & \text{integer} \\
 & | & \mathrm{M.S} & \text{struct name}
\end{array}$$

$$\begin{array}{lll}
\mathrm{FD} & ::= & \texttt{fun}\ \mathrm{F}\,(\,\overline{x} : \overline{\mathrm{T}}\,) : \mathrm{T}_r\ \{t_b\} \qquad\qquad \text{function definition} \\
\mathrm{SD} & ::= & \texttt{str}\ \mathrm{S}\,\{\,\top\,,\ \overline{\mathrm{T}}\,\} \ \ | \ \ \texttt{str}\ \mathrm{S}\,\{\,\bot\,,\ \overline{\mathrm{T}}\,\} \quad \text{struct definition} \\
\mathrm{MD} & ::= & \mathrm{M}\,\{\,\overline{\mathrm{SD}}\,,\ \overline{\mathrm{FD}}\,\} \qquad\qquad\qquad \text{module definition} \\
\mathrm{P} & ::= & \overline{\mathrm{MD}} \qquad\qquad\qquad\qquad\qquad \text{program}
\end{array}$$

$$\begin{array}{llll}
\text{Term } t & ::= & v & \text{value} \\
 & | & x & \text{variable} \\
 & | & x\,.\,j & \text{select } j\text{-th field of } x \\
 & | & \texttt{let}\ x = t_1\ \texttt{in}\ t_2 & \text{let binding} \\
 & | & \texttt{call}\ \mathrm{M.F}\,[\,\overline{t}\,] & \text{function call} \\
 & | & \texttt{pack}\ \mathrm{M.S}\,[\,\overline{t}\,] & \text{constructor} \\
 & | & \texttt{unpack}\ \{\overline{x}\} = t_1\ \texttt{in}\ t_2 & \text{deconstructor} \\
 & | & \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 & \\
 & | & \texttt{pub}\ t & \text{publish a resource} \\
 & | & \underline{\texttt{exec}\ \mathrm{M}\ t} & \text{function body} \\
 & | & \underline{v\,.\,j} & \text{select } j\text{-th field of } v
\end{array}$$

Highlights

Type system and operational semantics

- *Resource Preservation*: assets cannot be duplicated or accidentally lost at runtime
- Proves double-spending is prevented at compile time
- Equivalent to theorem by Blackshear et al. for bytecode lifted to source code
- Helps proving properties hold when compiling Move into other bytecodes
- Mechanized in **Agda**

A *pure* subset of Move

- No side effects (assignment), no references
- Monadic representation of CPS

CPS and State Monads in a nutshell

A simplified example:

CPS

```
struct Coin {
   amount : u64
}

let c1 = mint(100);
let c2 = spend(c1, 10);
let c3 = spend(c2, 30);
```

State Monad

```
type state = Coin


do mint(100);
   spend(10);
   spend(30);
```

State Monad automatizes the Continuation-Passing Style

Basic properties

Lemma 5 (Substitution). *Given* $\mathrm{M}_v \ni \Delta_1 \vdash v : \mathrm{T}_v \rhd \Delta_2$, *the following two properties hold:*

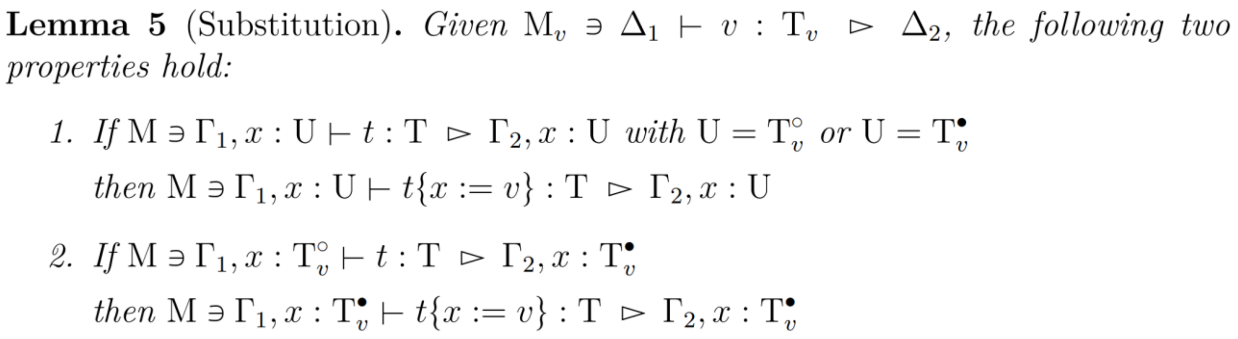
1. *If* $\mathrm{M} \ni \Gamma_1, x : \mathrm{U} \vdash t : \mathrm{T} \rhd \Gamma_2, x : \mathrm{U}$ *with* $\mathrm{U} = \mathrm{T}_v^{\circ}$ *or* $\mathrm{U} = \mathrm{T}_v^{\bullet}$
   then $\mathrm{M} \ni \Gamma_1, x : \mathrm{U} \vdash t\{x := v\} : \mathrm{T} \rhd \Gamma_2, x : \mathrm{U}$
2. *If* $\mathrm{M} \ni \Gamma_1, x : \mathrm{T}_v^{\circ} \vdash t : \mathrm{T} \rhd \Gamma_2, x : \mathrm{T}_v^{\bullet}$
   then $\mathrm{M} \ni \Gamma_1, x : \mathrm{T}_v^{\bullet} \vdash t\{x := v\} : \mathrm{T} \rhd \Gamma_2, x : \mathrm{T}_v^{\bullet}$

Lemma 6 (Type preservation). *If* $\mathrm{M} \ni \Gamma_1 \vdash t : \mathrm{T} \rhd \Gamma_2$ *and* $\mathrm{M} \ni t \rightarrow t'$ *then:*

$$\mathrm{M} \ni \Gamma_1 \vdash t' : \mathrm{T} \rhd \Gamma_2$$

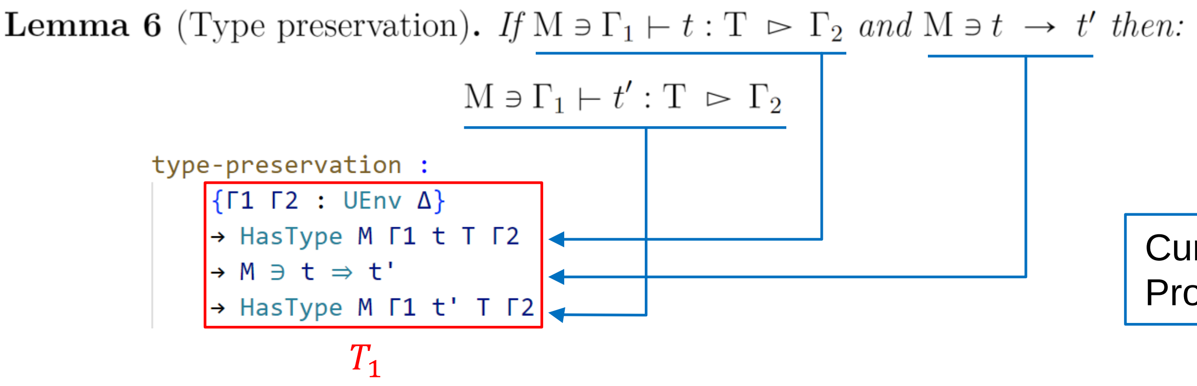
Theorem 1 (Type Safety). *If* $\mathrm{M} \ni \varnothing \vdash t : \mathrm{T} \rhd \varnothing$ *and* $\mathrm{M} \ni t \rightarrow^{\star} t'$ *then, either* t' *is a value or there exists a term* t'' *such that* $\mathrm{M} \ni t' \rightarrow t''$.

Basic properties



Lemma 5 (Substitution). *Given* $\mathrm{M}_v \ni \Delta_1 \vdash v : \mathrm{T}_v \rhd \Delta_2$*, the following two properties hold:*

1. *If* $\mathrm{M} \ni \Gamma_1, x : \mathrm{U} \vdash t : \mathrm{T} \rhd \Gamma_2, x : \mathrm{U}$ *with* $\mathrm{U} = \mathrm{T}_v^{\circ}$ *or* $\mathrm{U} = \mathrm{T}_v^{\bullet}$ *then* $\mathrm{M} \ni \Gamma_1, x : \mathrm{U} \vdash t\{x := v\} : \mathrm{T} \rhd \Gamma_2, x : \mathrm{U}$
2. *If* $\mathrm{M} \ni \Gamma_1, x : \mathrm{T}_v^{\circ} \vdash t : \mathrm{T} \rhd \Gamma_2, x : \mathrm{T}_v^{\bullet}$ *then* $\mathrm{M} \ni \Gamma_1, x : \mathrm{T}_v^{\bullet} \vdash t\{x := v\} : \mathrm{T} \rhd \Gamma_2, x : \mathrm{T}_v^{\bullet}$



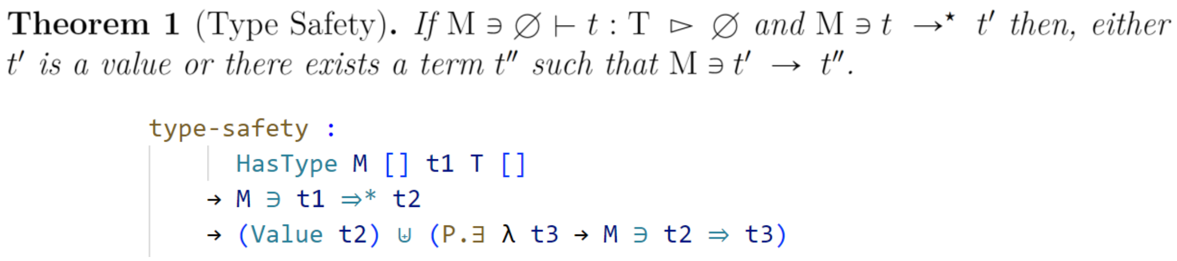
Lemma 6 (Type preservation). *If* $\underline{\mathrm{M} \ni \Gamma_1 \vdash t : \mathrm{T} \rhd \Gamma_2}$ *and* $\underline{\mathrm{M} \ni t \rightarrow t'}$ *then:*

$$\underline{\mathrm{M} \ni \Gamma_1 \vdash t' : \mathrm{T} \rhd \Gamma_2}$$

```
type-preservation :
    {Γ1 Γ2 : UEnv Δ}
  → HasType M Γ1 t T Γ2
  → M ∋ t ⇒ t'
  → HasType M Γ1 t' T Γ2
```

T_1

Curry Howard correspondence
Proof of Lemma 6 ⇔ Value : T_1



Theorem 1 (Type Safety). *If* $\mathrm{M} \ni \varnothing \vdash t : \mathrm{T} \rhd \varnothing$ *and* $\mathrm{M} \ni t \rightarrow^{\star} t'$ *then, either* t' *is a value or there exists a term* t'' *such that* $\mathrm{M} \ni t' \rightarrow t''$.

```
type-safety :
    HasType M [] t1 T []
  → M ∋ t1 ⇒* t2
  → (Value t2) ⊎ (P.∃ λ t3 → M ∋ t2 ⇒ t3)
```

FM: the Pack and Unpack

Value v	$::=$	n	integer
	$\mid$	$\underline{\texttt{struct}\,\{k\}\,\mathrm{M.S}\,[\,\overline{v}\,]}$	struct value $k \in K$

Type T	$::=$	Int	integer
	$\mid$	M.S	struct name

FD	$::=$	$\texttt{fun}\ \mathrm{F}\,(\,\overline{x} : \overline{\mathrm{T}}\,) : \mathrm{T}_r\,\{t_b\}$	function definition
SD	$::=$	$\texttt{str}\ \mathrm{S}\,\{\,\top\,,\ \overline{\mathrm{T}}\,\} \quad \mid \quad \texttt{str}\ \mathrm{S}\,\{\,\bot\,,\ \overline{\mathrm{T}}\,\}$	struct definition
MD	$::=$	$\mathrm{M}\,\{\,\overline{\mathrm{SD}}\,,\ \overline{\mathrm{FD}}\,\}$	module definition
P	$::=$	$\overline{\mathrm{MD}}$	program

Term t	$::=$	v	value
	$\mid$	x	variable
	$\mid$	$x\,.\,j$	select j-th field of x
	$\mid$	$\texttt{let}\ x = t_1\ \texttt{in}\ t_2$	let binding
	$\mid$	$\texttt{call}\ \mathrm{M.F}\,[\,\overline{t}\,]$	function call
	$\mid$	$\texttt{pack}\ \mathrm{M.S}\,[\,\overline{t}\,]$	constructor
	$\mid$	$\texttt{unpack}\ \{\overline{x}\} = t_1\ \texttt{in}\ t_2$	deconstructor
	$\mid$	$\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3$	
	$\mid$	$\texttt{pub}\ t$	publish a resource
	$\mid$	$\underline{\texttt{exec}\ \mathrm{M}\ t}$	function body
	$\mid$	$\underline{v\,.\,j}$	select j-th field of v

Resource Preservation

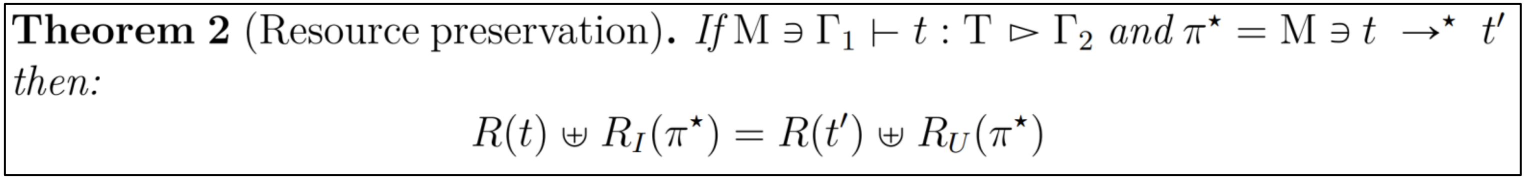
We proved that in FM resource values (linear struct values) can't be duplicated and can't be lost during the execution of a program.

- The programmer can't create a new resource without **explicitly** doing so with a pack.
- The programmer can’t delete a resource without **explicitly** doing so with an unpack.

Resource Preservation

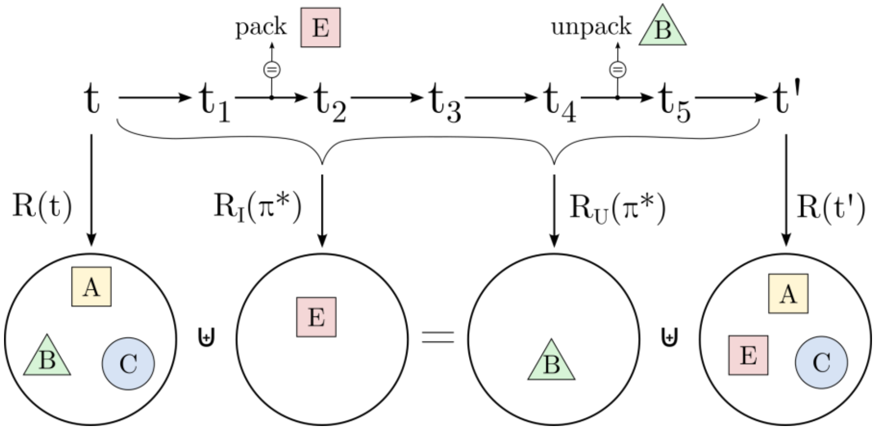
We proved that in FM resource values (linear struct values) can't be duplicated and can't be lost during the execution of a program.

- The programmer can't create a new resource without **explicitly** doing so with a pack.
- The programmer can’t delete a resource without **explicitly** doing so with an unpack.



Theorem 2 (Resource preservation). *If* $\mathrm{M} \ni \Gamma_1 \vdash t : \mathrm{T} \rhd \Gamma_2$ *and* $\pi^\star = \mathrm{M} \ni t \ \rightarrow^\star \ t'$ *then:*

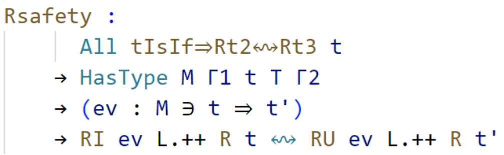
$$R(t) \uplus R_I(\pi^\star) = R(t') \uplus R_U(\pi^\star)$$



Resource Preservation

During evaluation, we mark newly created struct values with a fresh identifier k in order to distinguish them.

$$\frac{k \in K \text{ is fresh}}{\mathrm{M} \ni \texttt{pack}\,\mathrm{M.S}\,[\,\overline{v}\,] \;\longrightarrow\; \texttt{struct}\,\{k\}\,\mathrm{M.S}\,[\,\overline{v}\,]}\ \text{E-PACKED}$$



```
Rsafety :
    All tIsIf⇒Rt2↭Rt3 t
  → HasType M Γ1 t T Γ2
  → (ev : M ∋ t ⇒ t')
  → RI ev L.++ R t ↭ RU ev L.++ R t'
```

Theorem 2 (Resource preservation). *If* $\mathrm{M} \ni \Gamma_1 \vdash t : \mathrm{T} \rhd \Gamma_2$ *and* $\pi^\star = \mathrm{M} \ni t \;\rightarrow^\star\; t'$ *then:*

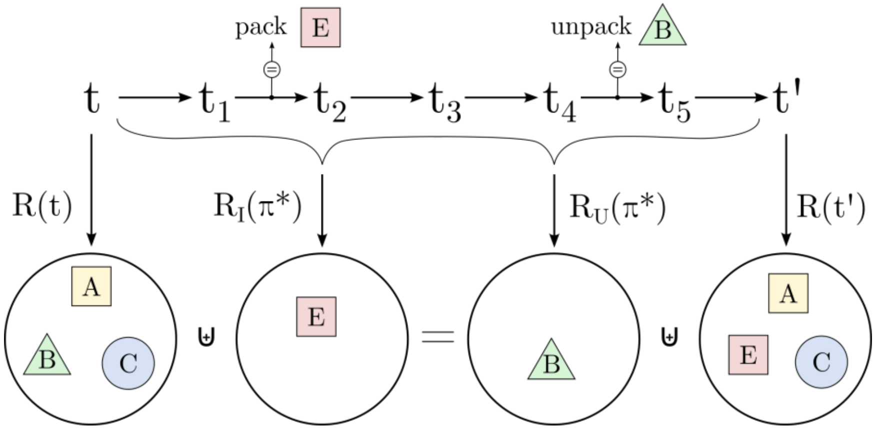
$$R(t) \uplus R_I(\pi^\star) = R(t') \uplus R_U(\pi^\star)$$

Resources introduced by π^*.

$R_I(\pi^*)$: The identifiers k of the linear structs explicitly created during π^*.

Resources used by π^*.

$R_U(\pi^*)$: The identifiers k of the linear structs explicitly unpacked during π^*.



Thank you.