# Holistic Specifications
## characterization of robustness


Sophia Drossopoulou

and

James Noble (VU Wellington),
Mark Miller (Agorics),
Toby Murray (Uni Melbourne)
Shupeng Loh  and Emil Klasan (Imperial)


# Functional

- *services offered* by objects/ data structure to *clients*,

- what *will* happen, under *correct use*

- *sufficient* conditions

# Functional vs

- *services offered* by objects/ data structure to *clients*,

- what *will* happen, under *correct use*

- *sufficient* conditions

# Functional vs Robust

- *services offered* by objects/ data structure to *clients*,

- what *will* happen, under *correct use*

- *sufficient* conditions

# Functional vs Robust

- *services offered* by objects/ data structure to *clients*,
- what *will* happen, under *correct use*
- *sufficient* conditions

- *preserved properties* of the objects/data structure
- what *will not* happen, under *arbitrary* use
- *necessary* conditions

# Today

- Functional ≠ Robustness
- **Robustness in terms of the Bank/Account Example**
- Holistic Specification:
  "Classical assertions"
    + Time
    + Space
    + Access
    + Authority
  "in an open world"
- Examples

# Bank/Account

- Banks and Accounts
- Accounts hold money
- Money can be transferred between Accounts
- A banks' currency  = sum of balances of accounts held by bank

[Miller et al, Financial Crypto 2000]

# Bank/Account - 2

- **Pol_1**: With two accounts of same bank one can transfer money between them.
- **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- **Pol_3**: The bank can only inflate its own currency
- **Pol_4**: No one can affect the balance of an account they do not have.
- **Pol_5**: Balances are always non-negative.
- **Pol_6**: A reported successful deposit can be trusted as much as one trusts the account one is depositing to.

[Miller et al, Financial Crypto 2000]

# Bank/Account - 2

functional

- **Pol_1**: With two accounts of same bank one can transfer money between them.
- **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- **Pol_3**: The bank can only inflate its own currency
- **Pol_4**: No one can affect the balance of an account they do not have.
- **Pol_5**: Balances are always non-negative.
- **Pol_6**: A reported successful deposit can be trusted as much as one trusts the account one is depositing to.

[Miller et al, Financial Crypto 2000]

# Bank/Account - 2

functional

- **Pol_1**: With two accounts of same bank one can transfer money between them.
- **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- **Pol_3**: The bank can only inflate its own currency
- **Pol_4**: No one can affect the balance of an account they do not have.
- **Pol_5**: Balances are always non-negative.
- **Pol_6**: A reported successful deposit can be trusted as much as one trusts the account one is depositing to.

[Miller et al, Financial Crypto 2000]

# Bank/Account - 2

functional | robustness

- **Pol_1**: With two accounts of same bank one can transfer money between them.
- **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- **Pol_3**: The bank can only inflate its own currency
- **Pol_4**: No one can affect the balance of an account they do not have.
- **Pol_5**: Balances are always non-negative.
- **Pol_6**: A reported successful deposit can be trusted as much as one trusts the account one is depositing to.

[Miller et al, Financial Crypto 2000]

# Bank/Account - 2

functional | robustness

- **Pol_1**: With two accounts of same bank one can transfer money between them.
- **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- **Pol_3**: The bank can only inflate its own currency
- **Pol_4**: No one can affect the balance of an account they do not have.
- **Pol_5**: Balances are always non-negative.
- **Pol_6**: A reported successful deposit can be trusted as much as one trusts the account one is depositing to.

[Miller et al, Financial Crypto 2000]

# Bank/Account - 2

functional | robustness

- **Pol_1**: With two accounts of same bank one can transfer money between them.
- **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- **Pol_3**: The bank can only inflate its own currency
- **Pol_4**: No one can affect the balance of an account they do not have.
- **Pol_5**: Balances are always non-negative.
- **Pol_6**: A reported successful deposit can be trusted as much as one trusts the account one is depositing to.

[Miller et al, Financial Crypto 2000]

# Bank/Account - 2

functional | robustness

- **Pol_1**: With two accounts of same bank one can transfer money between them.
- **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- **Pol_3**: The bank can only inflate its own currency
- **Pol_4**: No one can affect the balance of an account they do not have.
- **Pol_5**: Balances are always non-negative.
- **Pol_6**: A reported successful deposit can be trusted as much as one trusts the account one is depositing to.

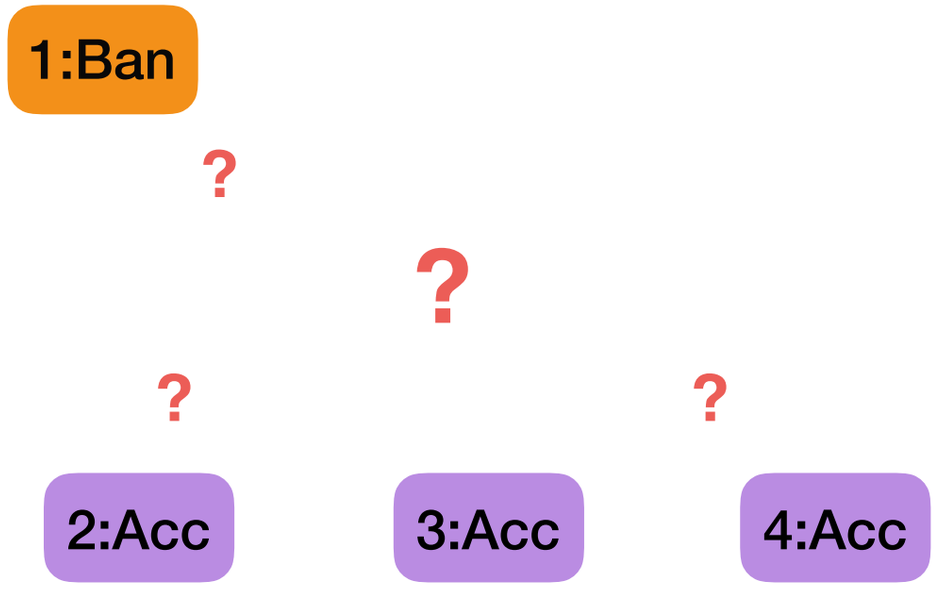
[Miller et al, Financial Crypto 2000]

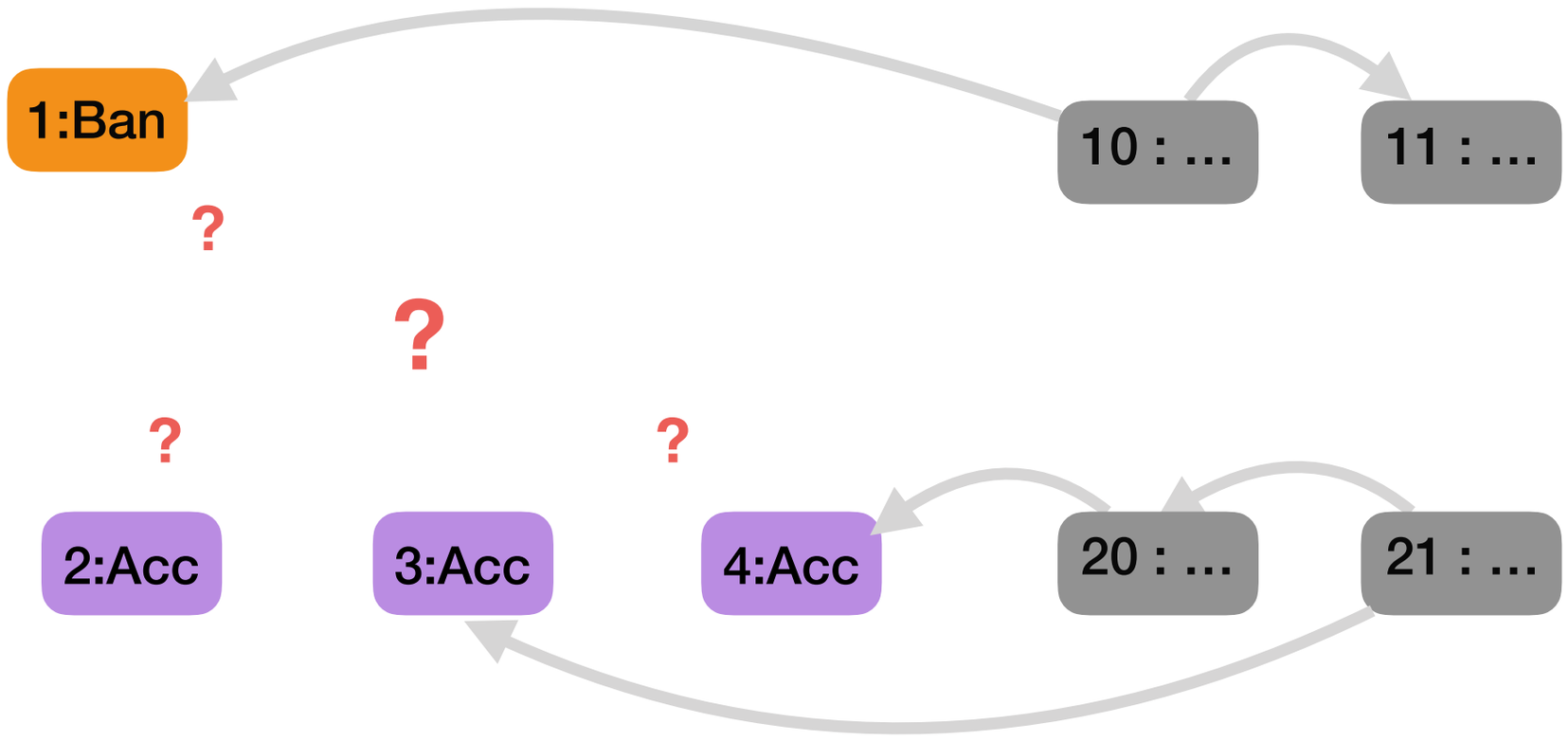
# Bank/Account - 2

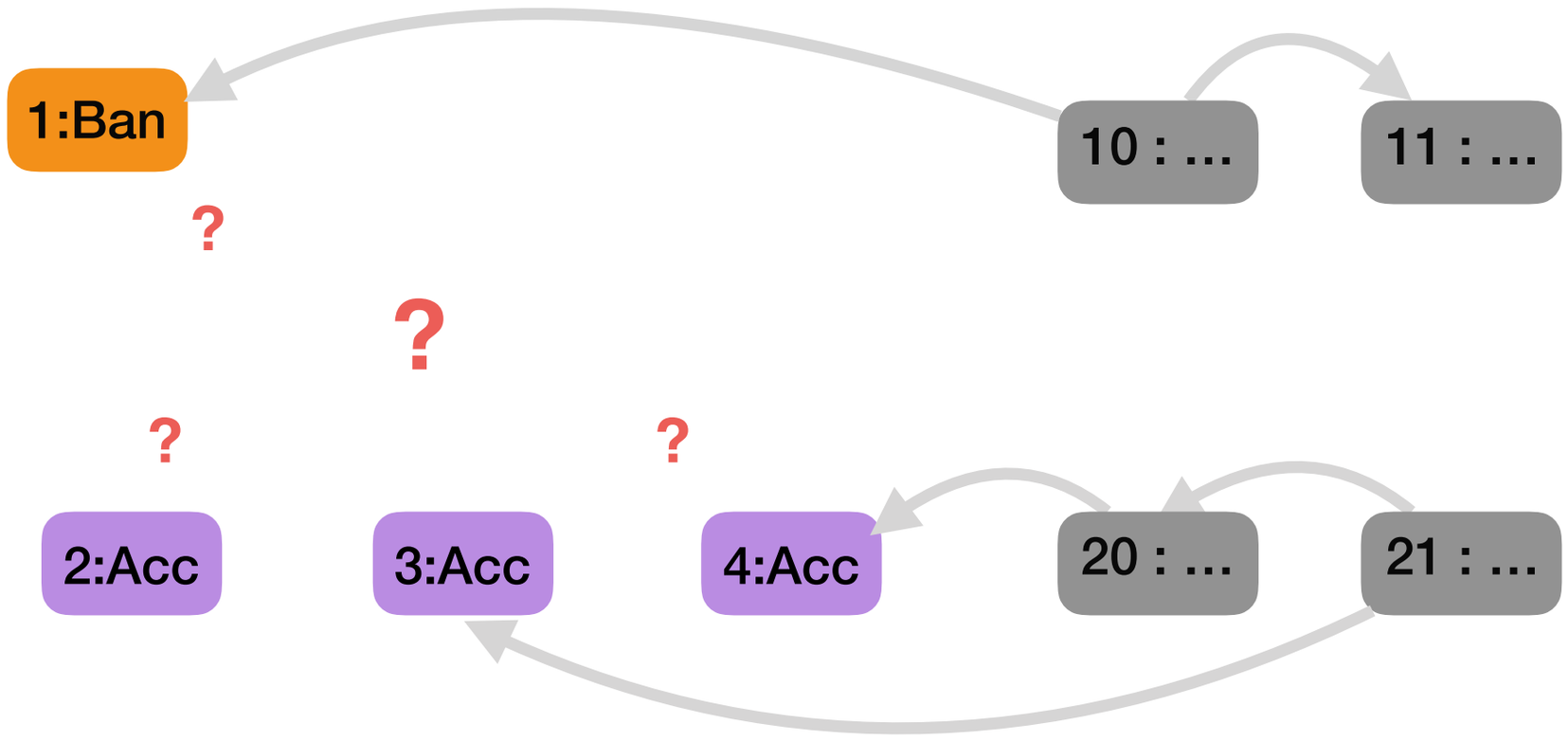
functional | robustness

- **Pol_1**: With two accounts of same bank one can transfer money between them.
- **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- **Pol_3**: The bank can only inflate its own currency
- **Pol_4**: No one can affect the balance of an account they do not have.
- **Pol_5**: Balances are always non-negative.
- **Pol_6**: A reported successful deposit can be trusted as much as one trusts the account one is depositing to.

[Miller et al, Financial Crypto 2000]

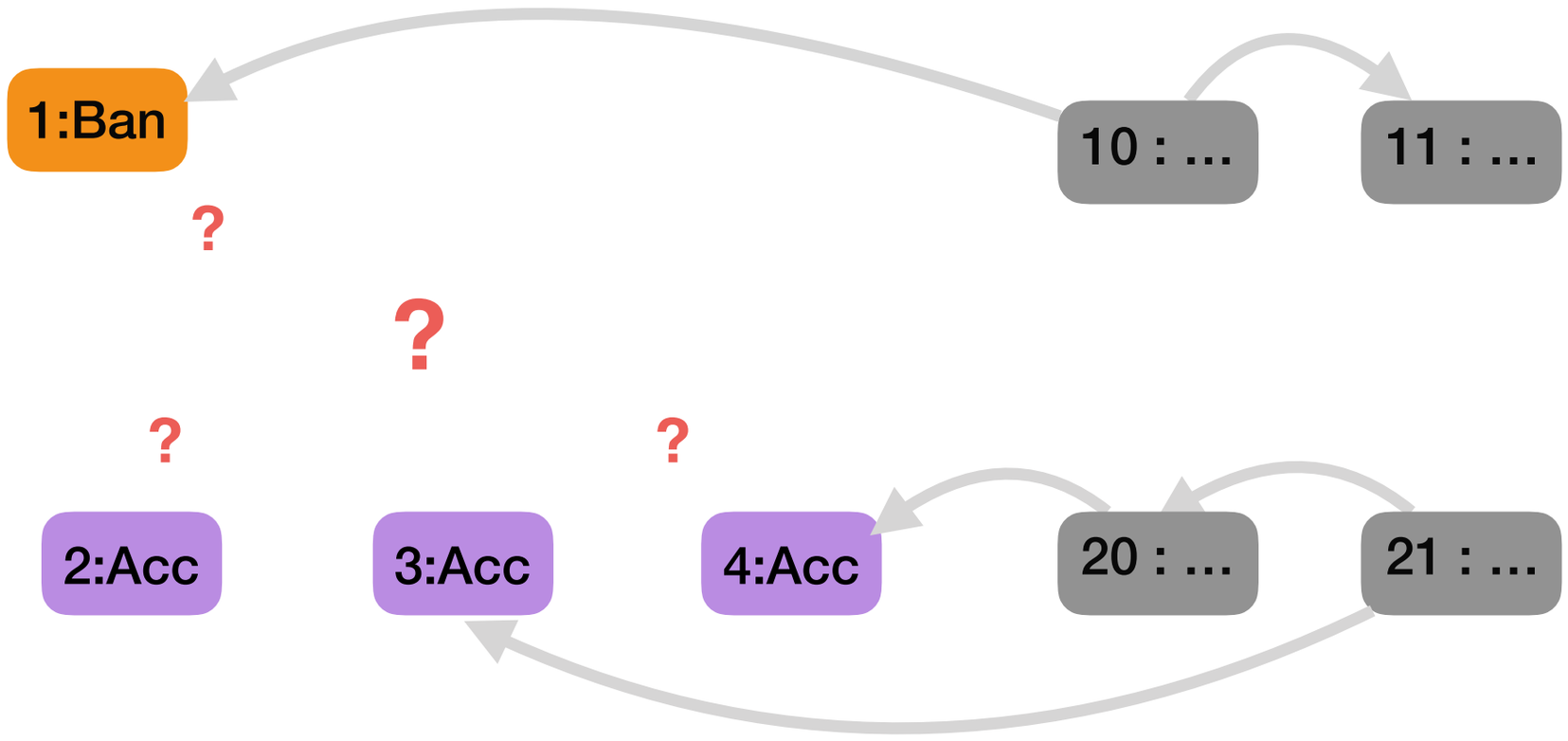

1:Ban
?
?
?
?
2:Acc
3:Acc
4:Acc


1:Ban
10 : …
11 : …
?
?
?
?
2:Acc
3:Acc
4:Acc
20 : …
21 : …




Should the following be possible?

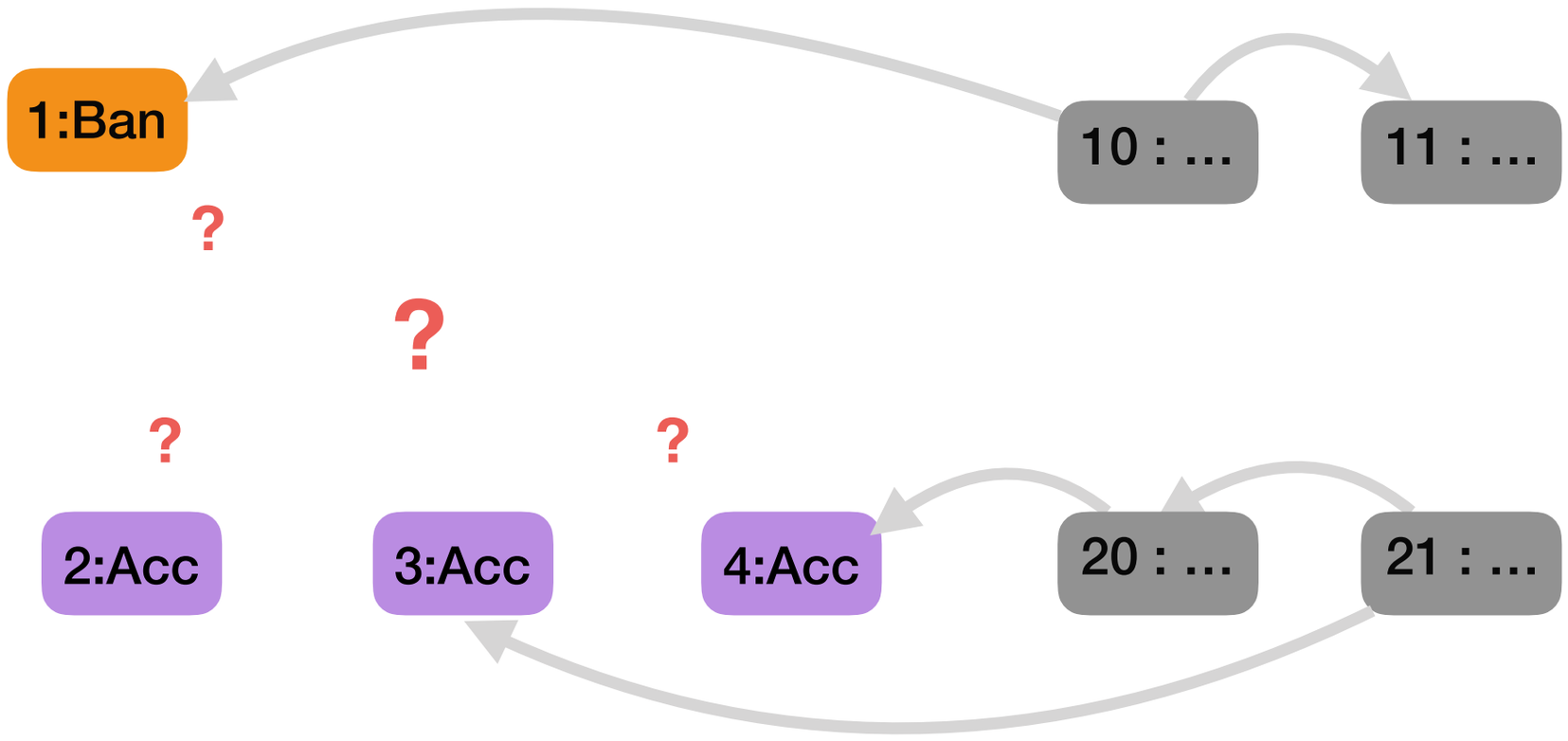
- 21 takes money from 4.
- 21 takes money from 2.
- 10 affects the currency.
- 10 takes money from 4.
- 21 finds out 2’s balance.




Should the following be possible?

- 21 takes money from 4.
- 21 takes money from 2.
- 10 affects the currency.
- 10 takes money from 4.
- 21 finds out 2's balance.

- **PoI_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- **PoI_4**: No one can affect the balance of an account they do not have.




Should the following be possible?

- 21 takes money from 4. ✅
- 21 takes money from 2. ❌
- 10 affects the currency. ✅
- 10 takes money from 4. ❌
- 21 finds out 2's balance. ?

- **PoI_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- **PoI_4**: No one can affect the balance of an account they do not have.

# MBA1: Code

```
class Bank {

}
```

```
class Account {
  fld myBank        // a Bank
  fld balance       // a number
  Account(aBank,amt){ myBank=aBank; balance=amt }
  fun deposit(destination,amt)
    { if myBank==destination.myBank then
        { this.balance-=amt;
          destination.balance+=amt } }
}
```

# MBA1: Code

```
class Bank {

}
```

```
class Account {
  fld myBank        // a Bank
  fld balance       // a number
  Account(aBank,amt){ myBank=aBank; balance=amt }
  fun deposit(destination,amt)
    { if myBank==destination.myBank then
        { this.balance-=amt;
          destination.balance+=amt } }
}
```

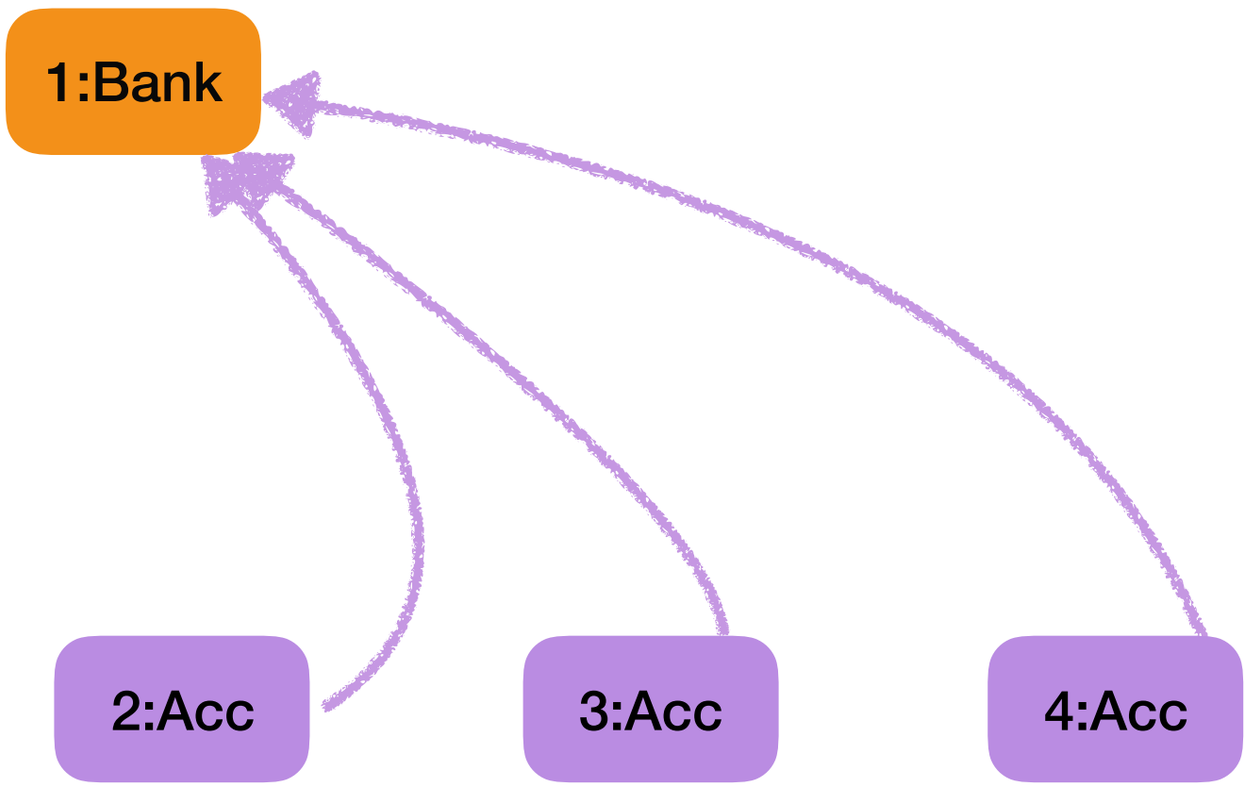
Note: bank.currency is a model field

# MBA1: Objects

# MBA1: Objects


```
class Bank{   }
```

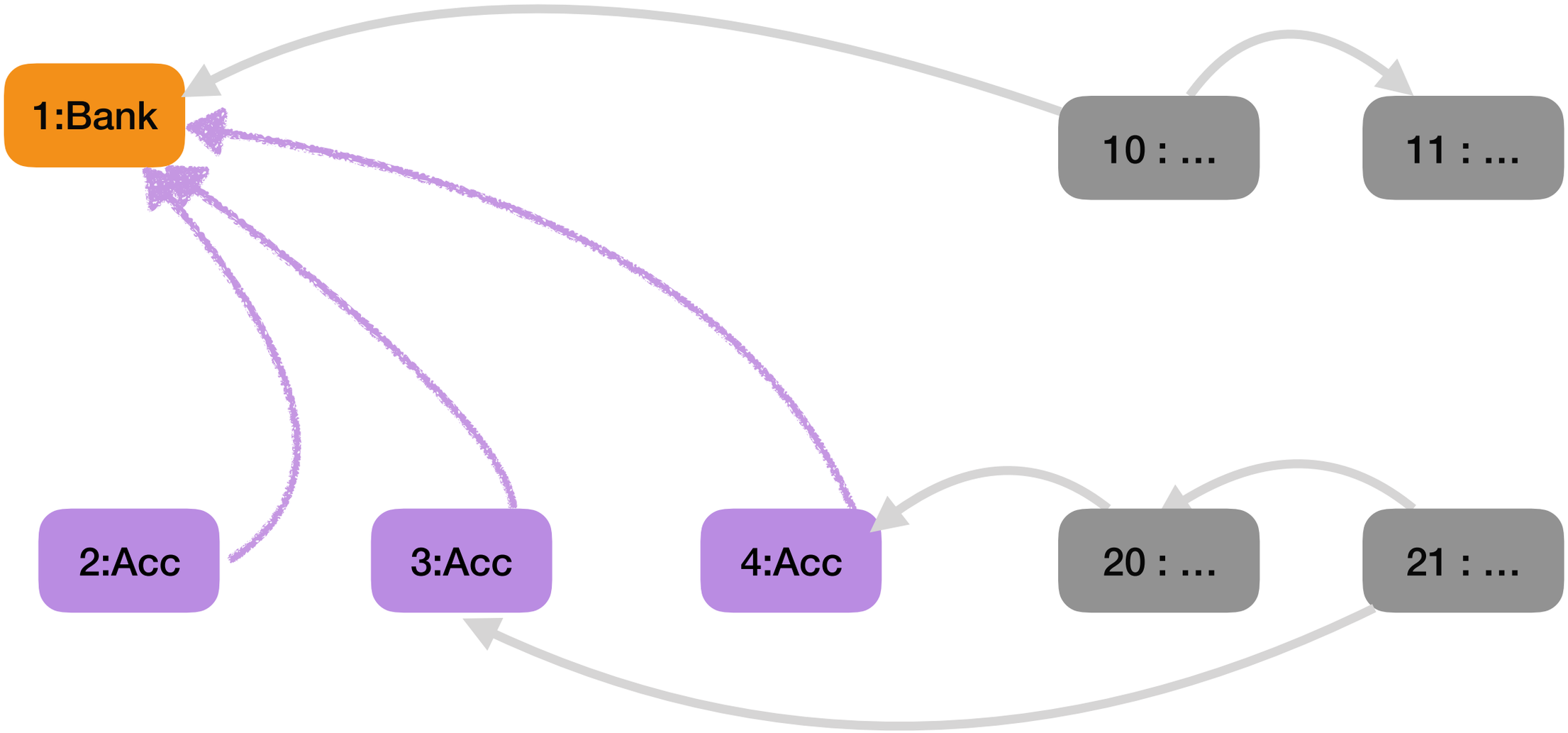
# MBA1: Objects




```
class Bank{   }

class Account {
   fld myBank
   fld balance
   ….
  }
```

# MBA1: Objects



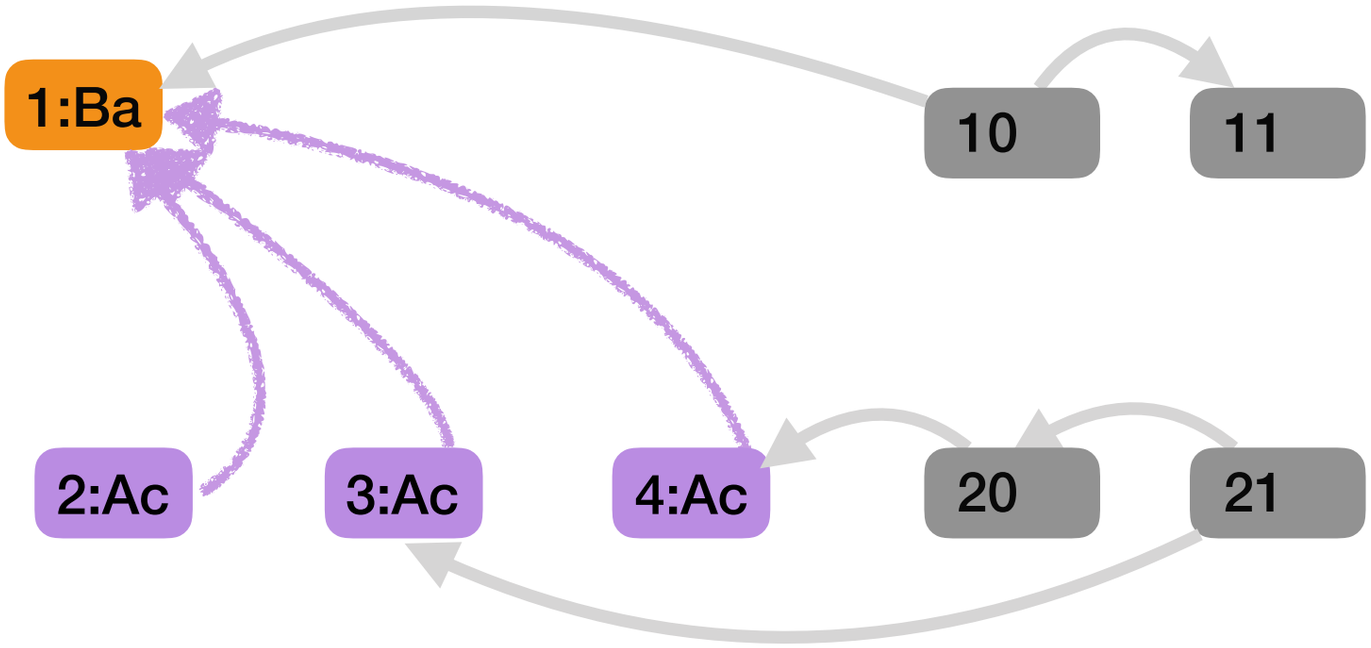

```
class Bank{   }

class Account {
   fld myBank
   fld balance
   ….
  }
```

```
class Unknown1 {
   fld …
   ….
  }
```

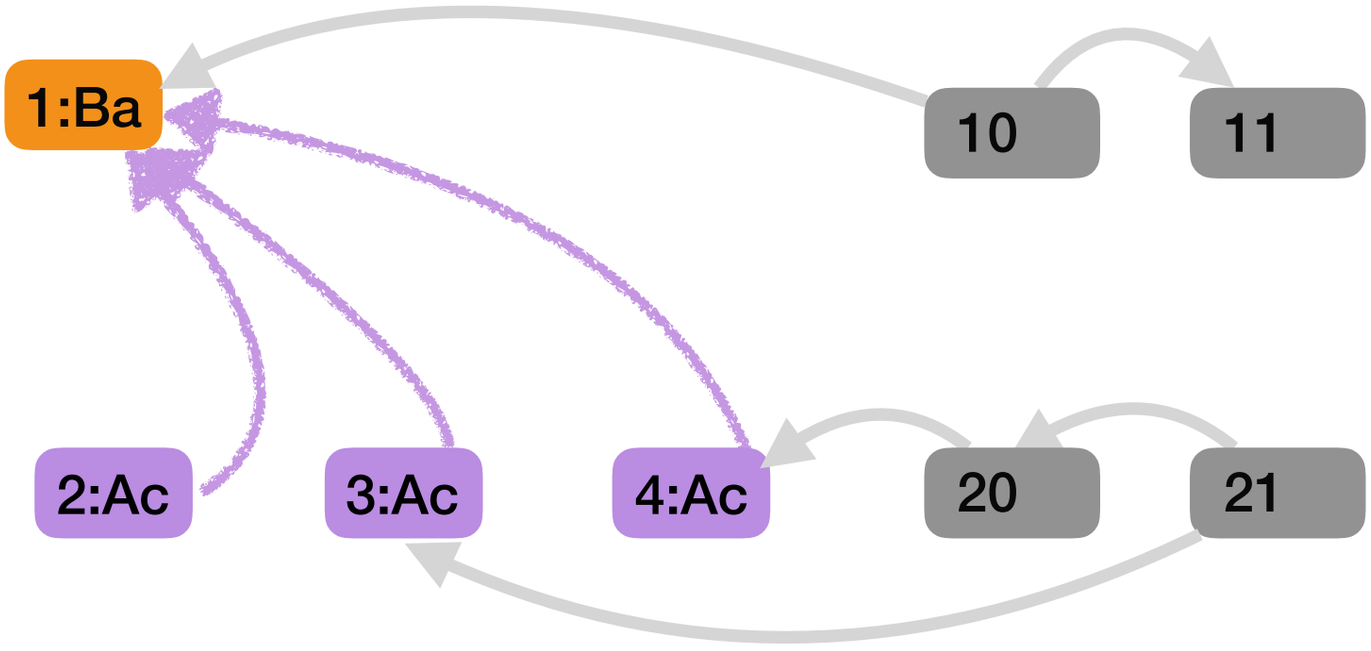
```
class Unknown2 {
   fld …
   ….
  }
```

# MBA1: Adherence to Policies




- **Pol_1**: With two accounts of same bank one can transfer money between them.
- **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- **Pol_4**: No one can affect the balance of an account they do not have.

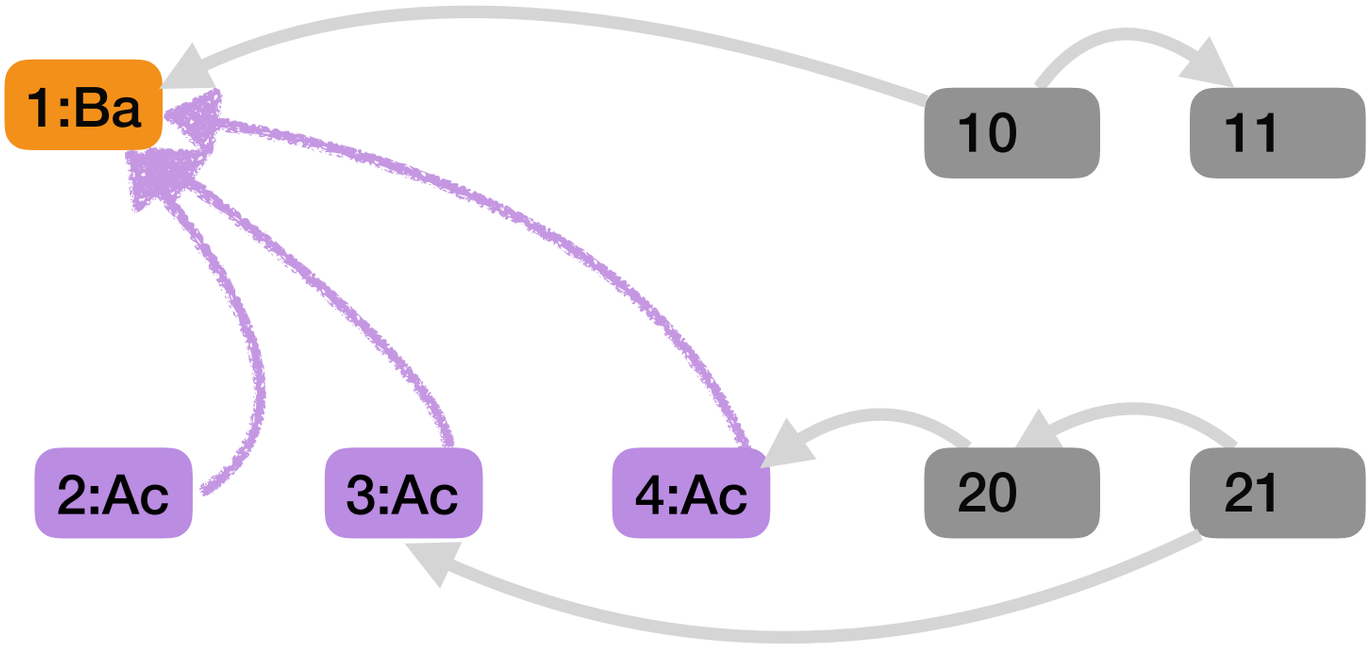
# MBA1: Adherence to Policies




✅

- **Pol_1**: With two accounts of same bank one can transfer money between them.
- **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- **Pol_4**: No one can affect the balance of an account they do not have.

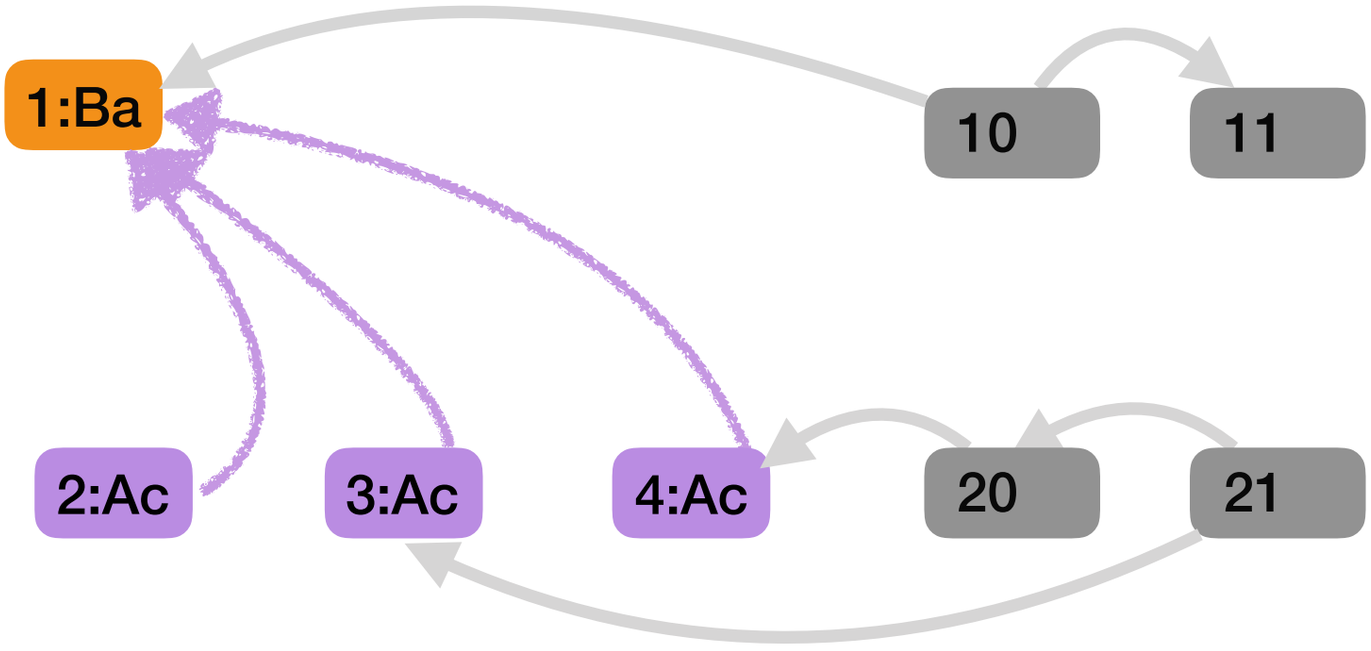
# MBA1: Adherence to Policies




✅
- **Pol_1**: With two accounts of same bank one can transfer money between them.

❌
- **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- **Pol_4**: No one can affect the balance of an account they do not have.

# MBA1: Adherence to Policies

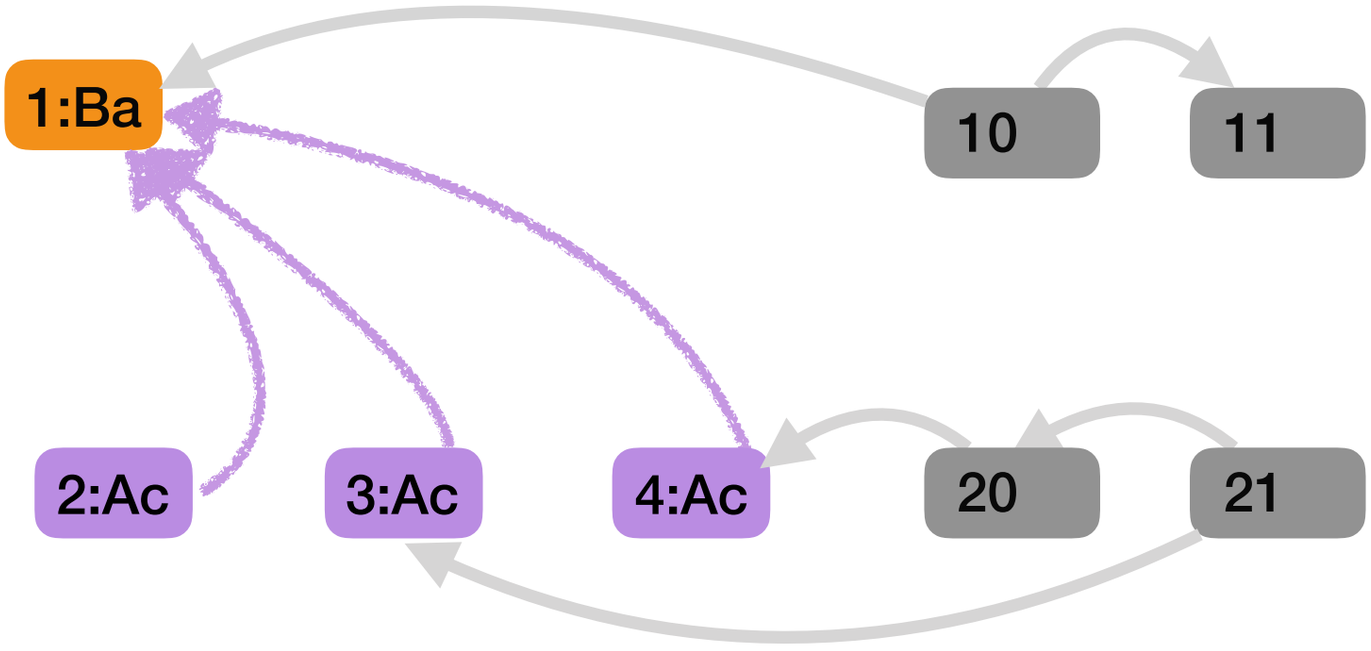



✅ - **Pol_1**: With two accounts of same bank one can transfer money between them.

❌ - **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency

✅ - **Pol_4**: No one can affect the balance of an account they do not have.

# MBA1: Adherence to Policies




$MBA1 \models Pol\_1$

$MBA1 \not\models Pol\_2$

$MBA1 \models Pol\_4$

- ✅ **Pol_1**: With two accounts of same bank one can transfer money between them.
- ❌ **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- ✅ **Pol_4**: No one can affect the balance of an account they do not have.

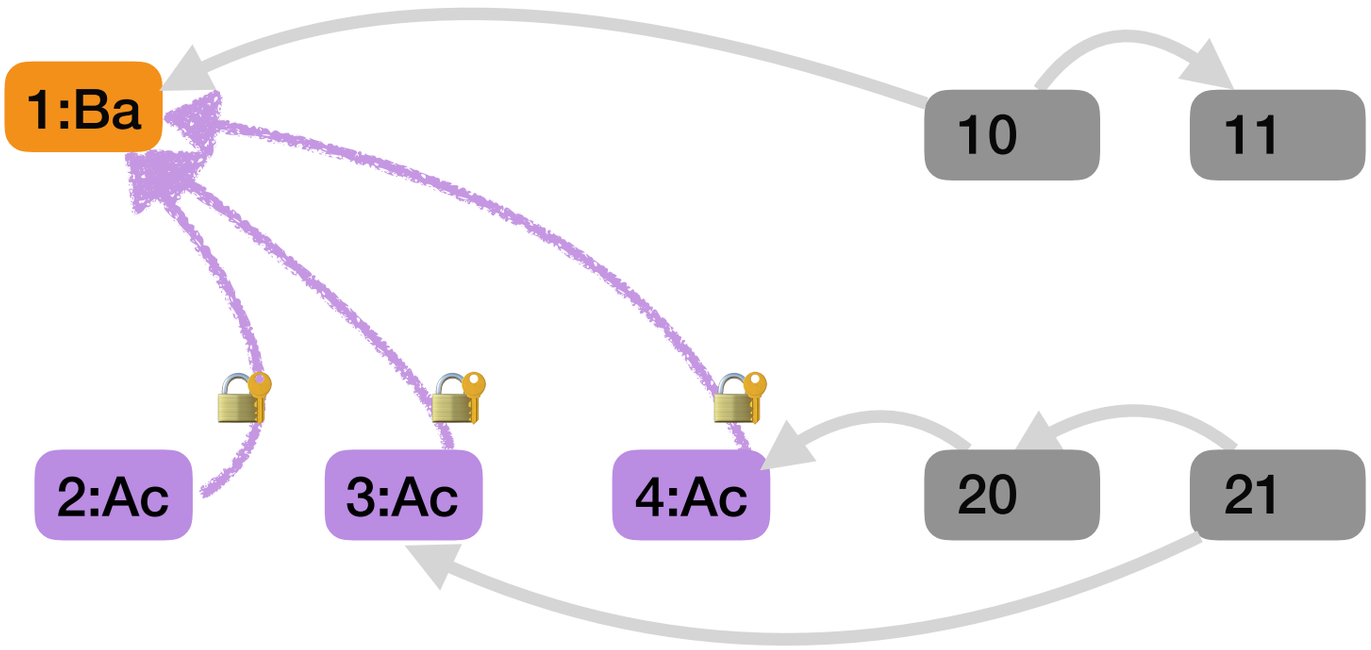
# MBA1s - safe

```
class Bank {

}
```

```
class Account {
  class private fld myBank        // a Bank
  class private fld balance       // a number
  Account(aBank,amt){ myBank=aBank; balance=amt }
  fun deposit(destination,amt)
    { if myBank==destination.myBank then
        { this.balance-=amt;
          destination.balance+=amt } }
}
```

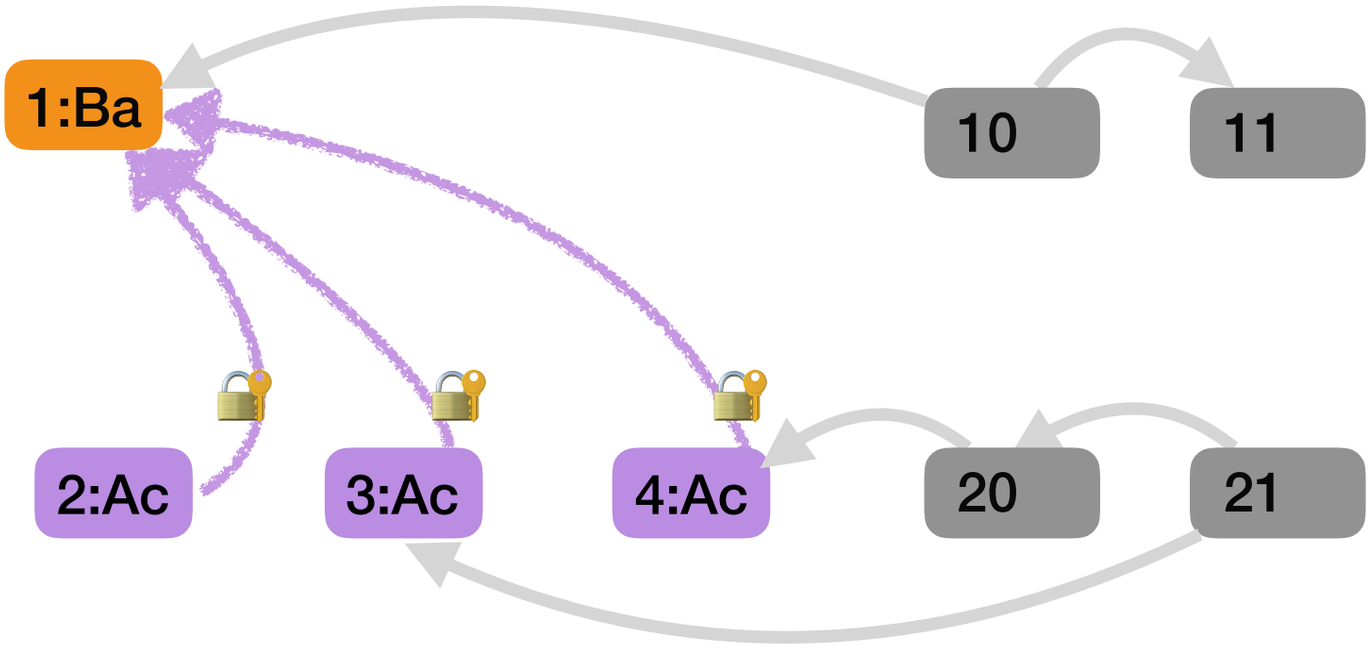
# MBA1s: Adherence to Policies




- **Pol_1**: With two accounts of same bank one can transfer money between them.
- **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- **Pol_4**: No one can affect the balance of an account they do not have.

🔐 indicates a private field

# MBA1s: Adherence to Policies




$MBA1s \models Pol\_1$

$MBA1s \models Pol\_2$

$MBA1s \models Pol\_4$

- ✅ **Pol_1**: With two accounts of same bank one can transfer money between them.
- ✅ **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- ✅ **Pol_4**: No one can affect the balance of an account they do not have.

🔐 indicates a private field

# MBA2: Code

```
class Bank {
  fld book // a Node
  Bank( ){ book=null }
  fun makeAccount(amt){ … }
  fun deposit(src,dest,amt){
    srce=book.get(src);
    destn=book.get(dest);
    if srce.balance>amt then
       { srce.balance-=amt;
         destn.balance+=amt }  }
}
```

```
class Node {
  fld balance       // a number
  fld next          // a Node
  fld theAccount    // an Account
  fun get(acc){
     if theAccount==acc
     then{ this; }
     else{ … next.get(acc) … }
  }
```

```
class Account {
  fld myBank        // a Bank
  Account(aBank){ myBank=aBank }
  fun deposit(destination,amt)
    { myBank.deposit(this,destination,amt) }
}
```

# MBA2: Code

```
class Bank {
  fld book // a Node
  Bank( ){ book=null }
  fun makeAccount(amt){ … }
  fun deposit(src,dest,amt){
    srce=book.get(src);
    destn=book.get(dest);
    if srce.balance>amt then
       { srce.balance-=amt;
         destn.balance+=amt }  }
}
```

```
class Node {
  fld balance       // a number
  fld next          // a Node
  fld theAccount    // an Account
  fun get(acc){
```

bank.currency is model field

account.balance is model field

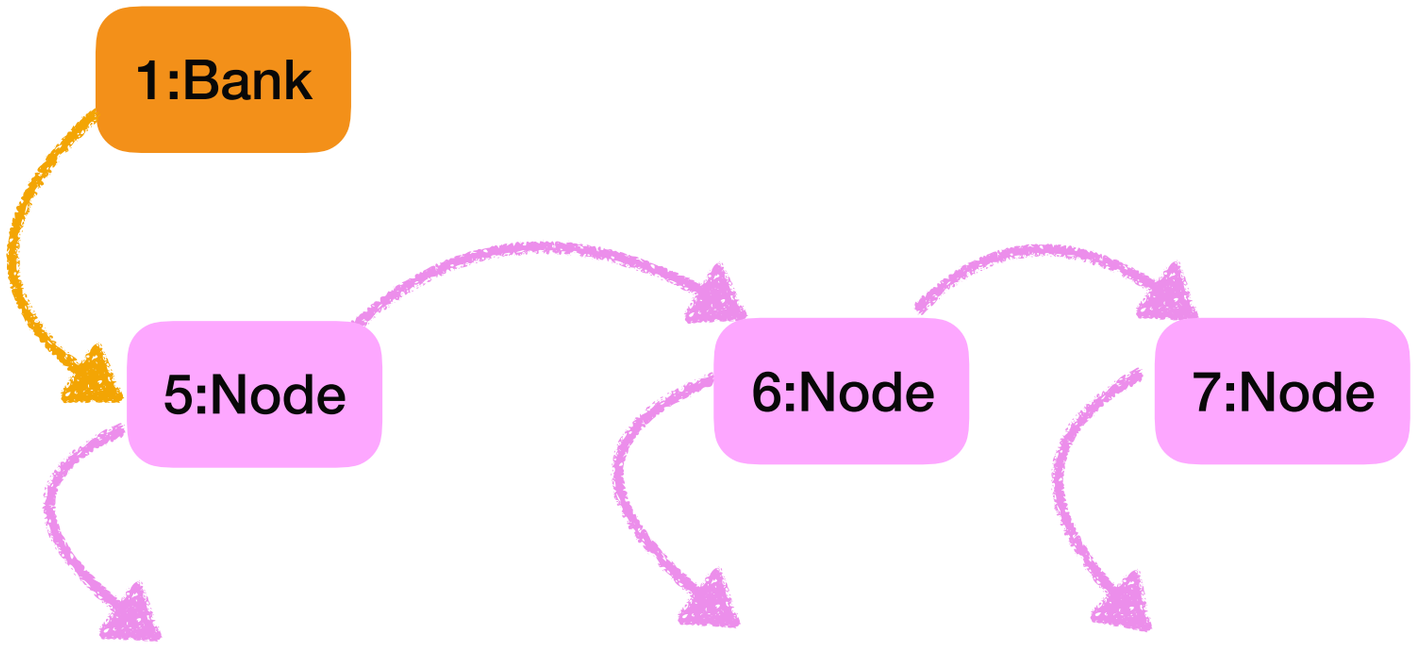
```
class Account {
  fld myBank        // a Bank
  Account(aBank){ myBank=aBank }
  fun deposit(destination,amt)
    { myBank.deposit(this,destination,amt) }
}
```

# MBA2: Objects

# MBA2: Objects


```
class Bank {
  fld book
  …
}
```

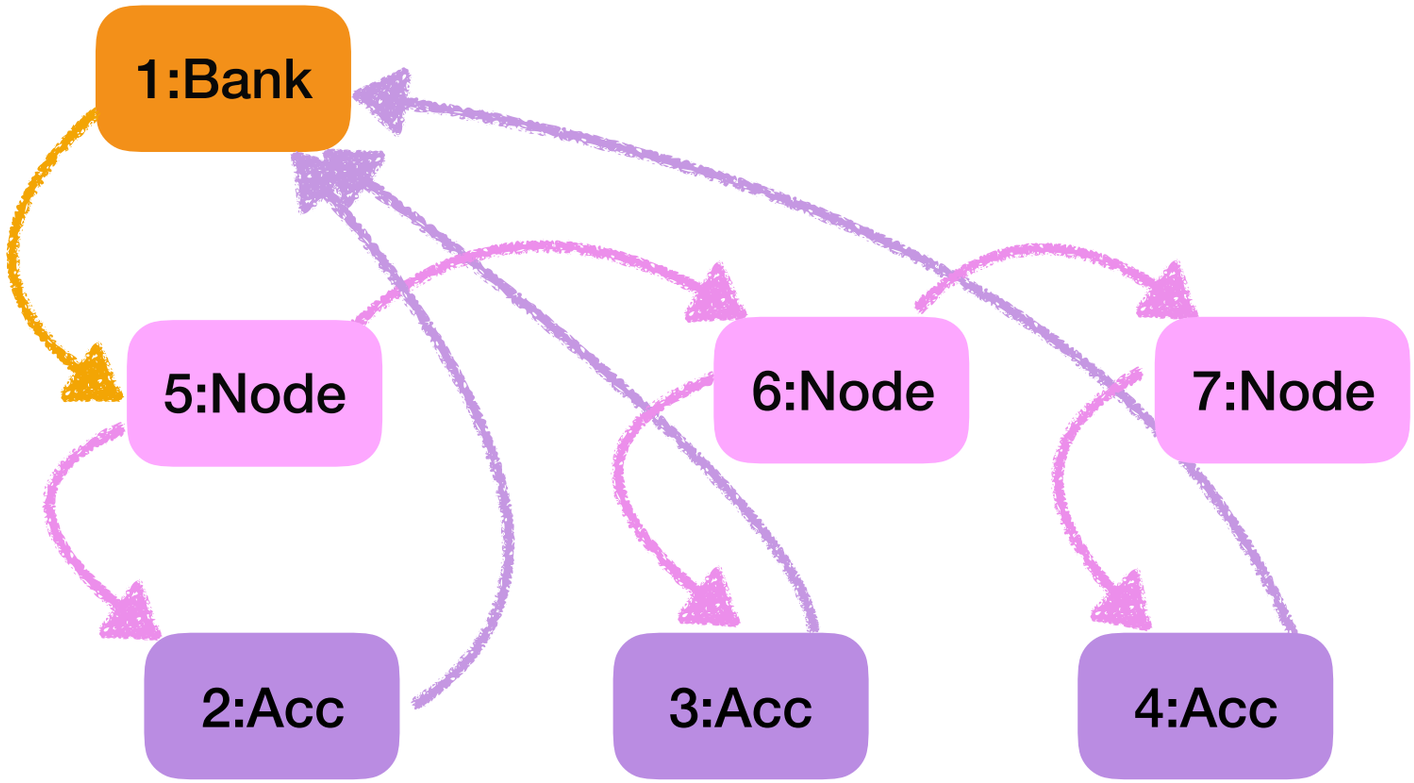
# MBA2: Objects




```
class Bank {
  fld book
  …
}
```

```
class Node {
  fld balance
  fld next
  fld theAccount
  …
}
```

# MBA2: Objects

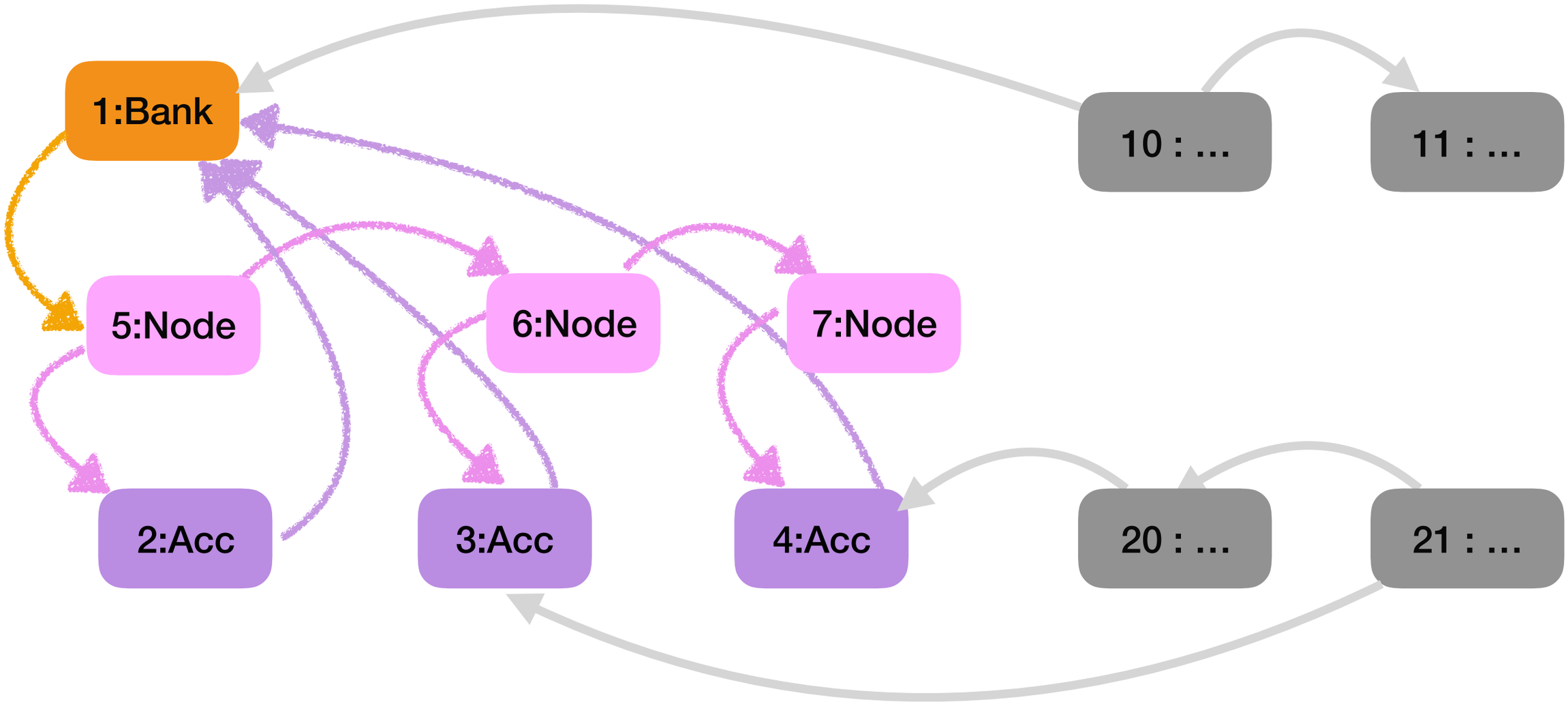



```
class Bank {
  fld book
  …
}
```

```
class Node {
  fld balance
  fld next
  fld theAccount
  …
}
```

```
class Account {
  fld myBank
  ….
}
```

# MBA2: Objects




```
class Bank {
  fld book
  …
}
```

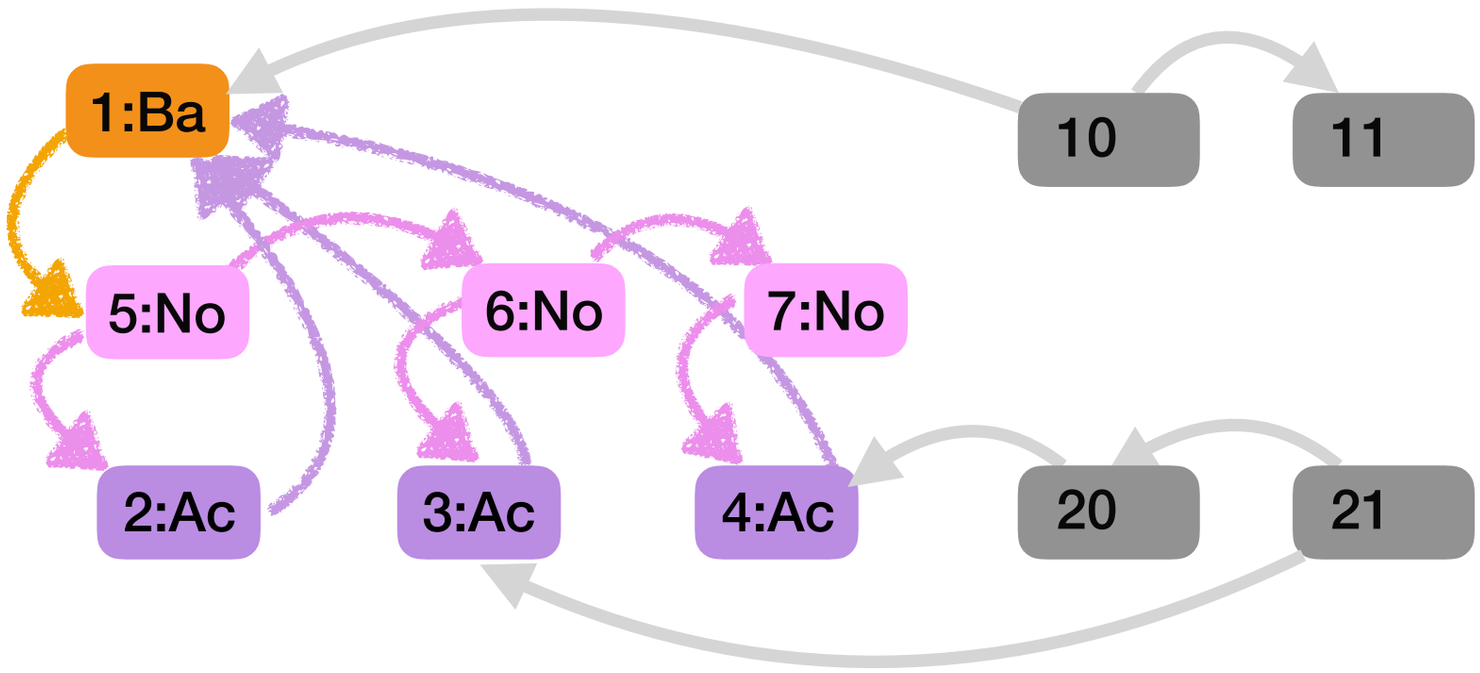
```
class Node {
  fld balance
  fld next
  fld theAccount
  …
}
```

```
class Account {
  fld myBank
  ….
}
```

```
class Unknown1 {
  fld …
  ….
}
```

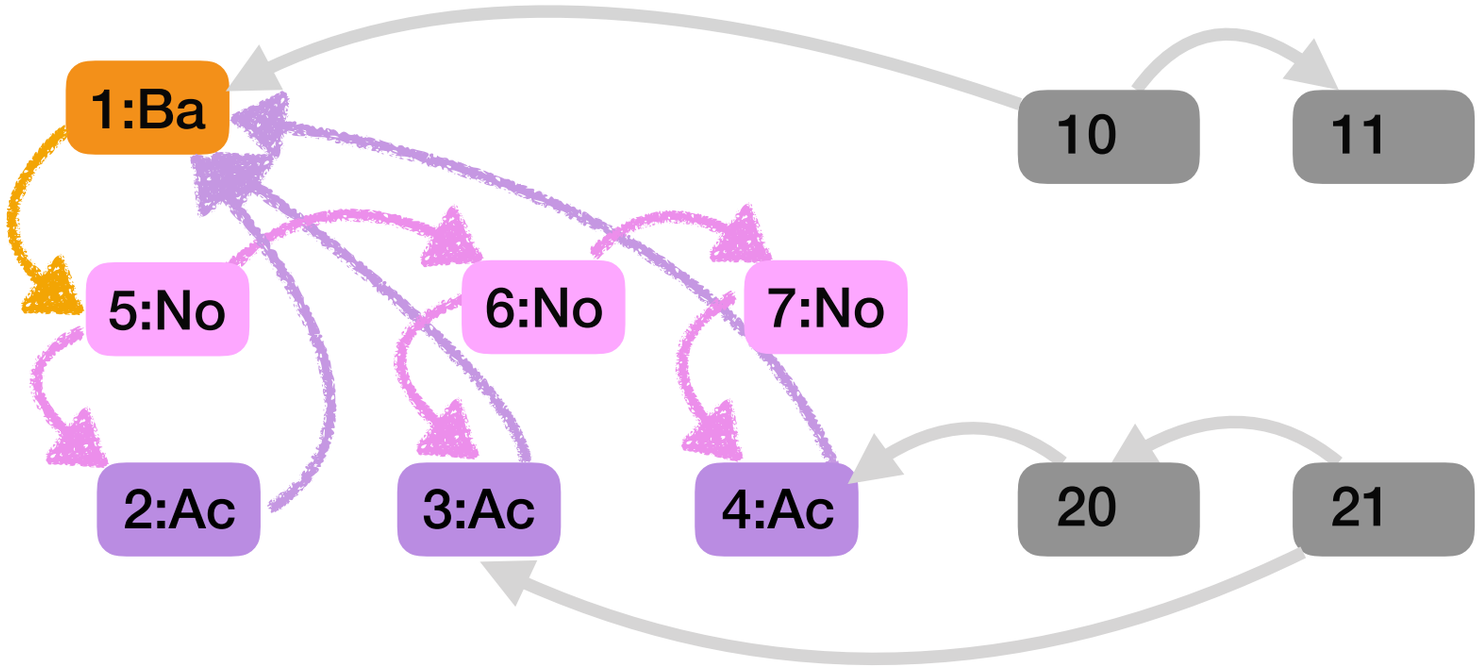
```
class Unknown1 {
  fld …
  ….
}
```

# MBA2: Adherence to Policies




- **Pol_1**: With two accounts of same bank one can transfer money between them.
- **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- **Pol_4**: No one can affect the balance of an account they do not have.

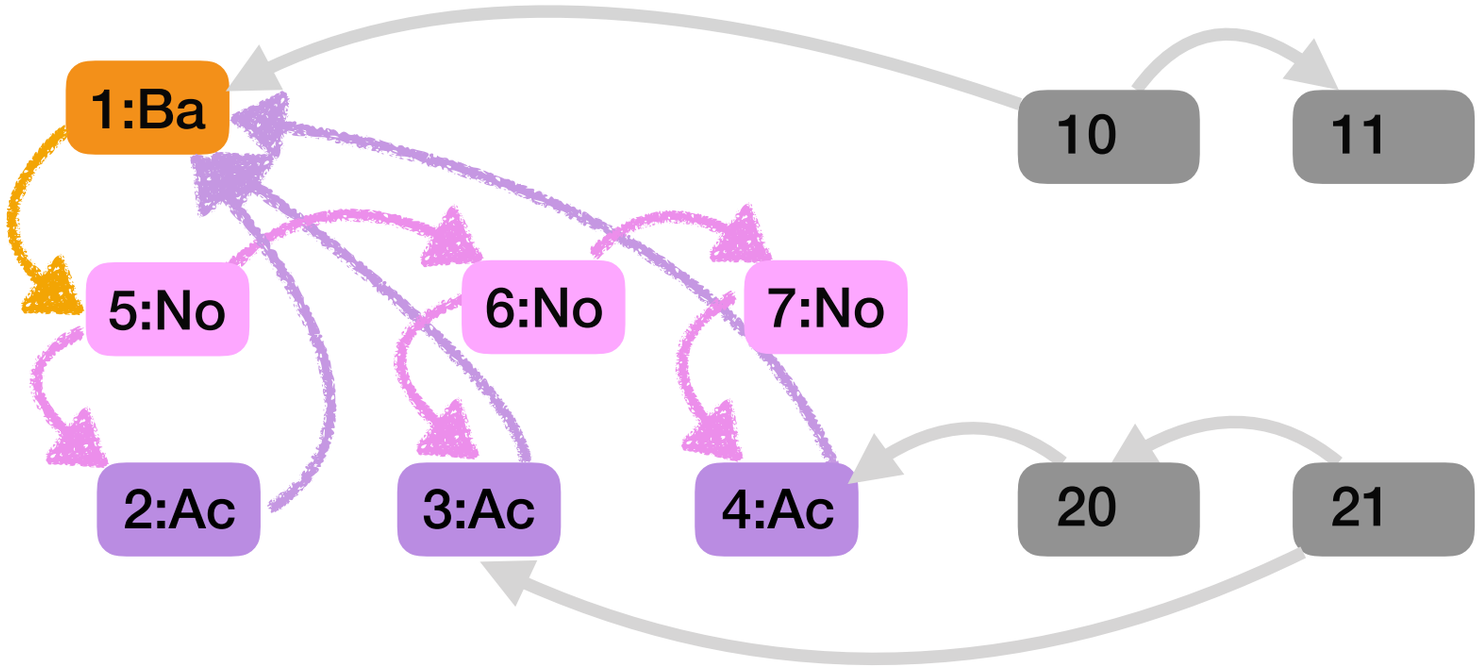
# MBA2: Adherence to Policies




✅

- **Pol_1**: With two accounts of same bank one can transfer money between them.
- **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- **Pol_4**: No one can affect the balance of an account they do not have.

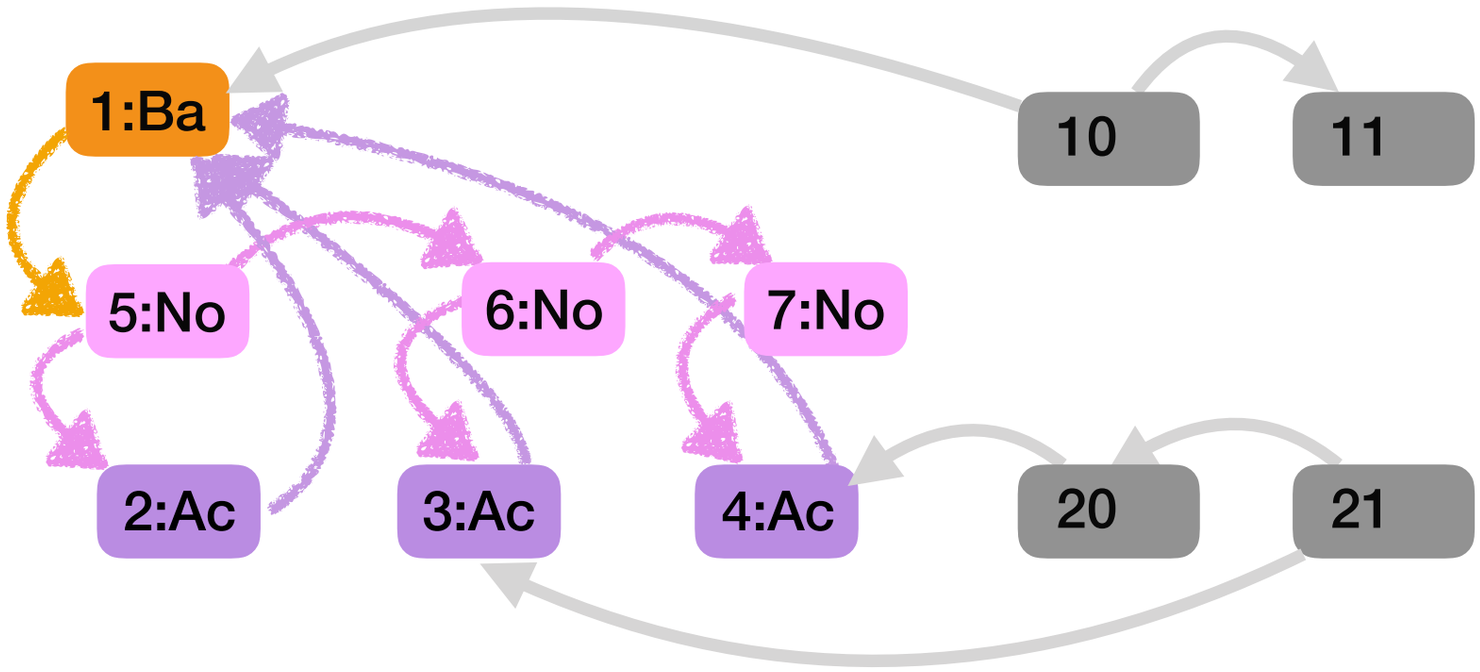
# MBA2: Adherence to Policies




✅
- **Pol_1**: With two accounts of same bank one can transfer money between them.

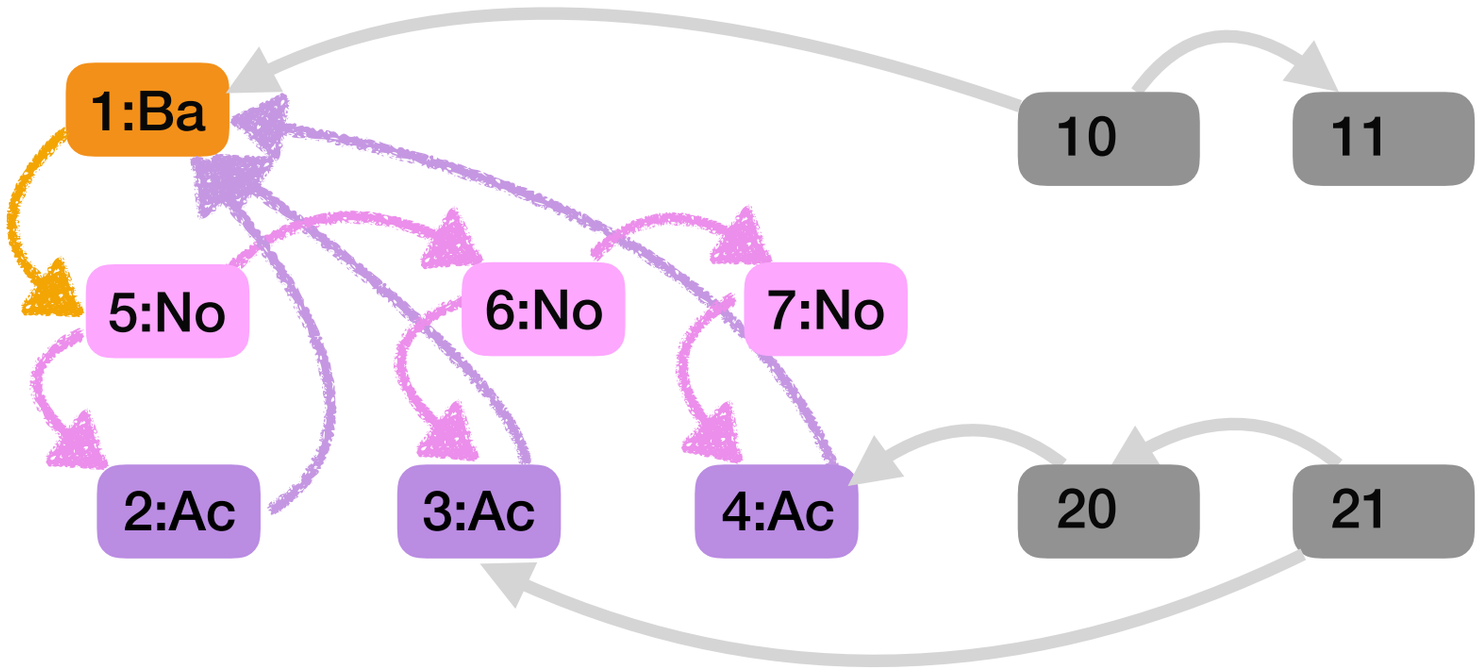
❌
- **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- **Pol_4**: No one can affect the balance of an account they do not have.

# MBA2: Adherence to Policies




- ✅ **Pol_1**: With two accounts of same bank one can transfer money between them.
- ❌ **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- ❌ **Pol_4**: No one can affect the balance of an account they do not have.

# MBA2: Adherence to Policies




MBA2 ⊨ Pol_1

MBA2 ⊭ Pol_2

MBA2 ⊭ Pol_4

- ✅ **Pol_1**: With two accounts of same bank one can transfer money between them.
- ❌ **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- ❌ **Pol_4**: No one can affect the balance of an account they do not have.

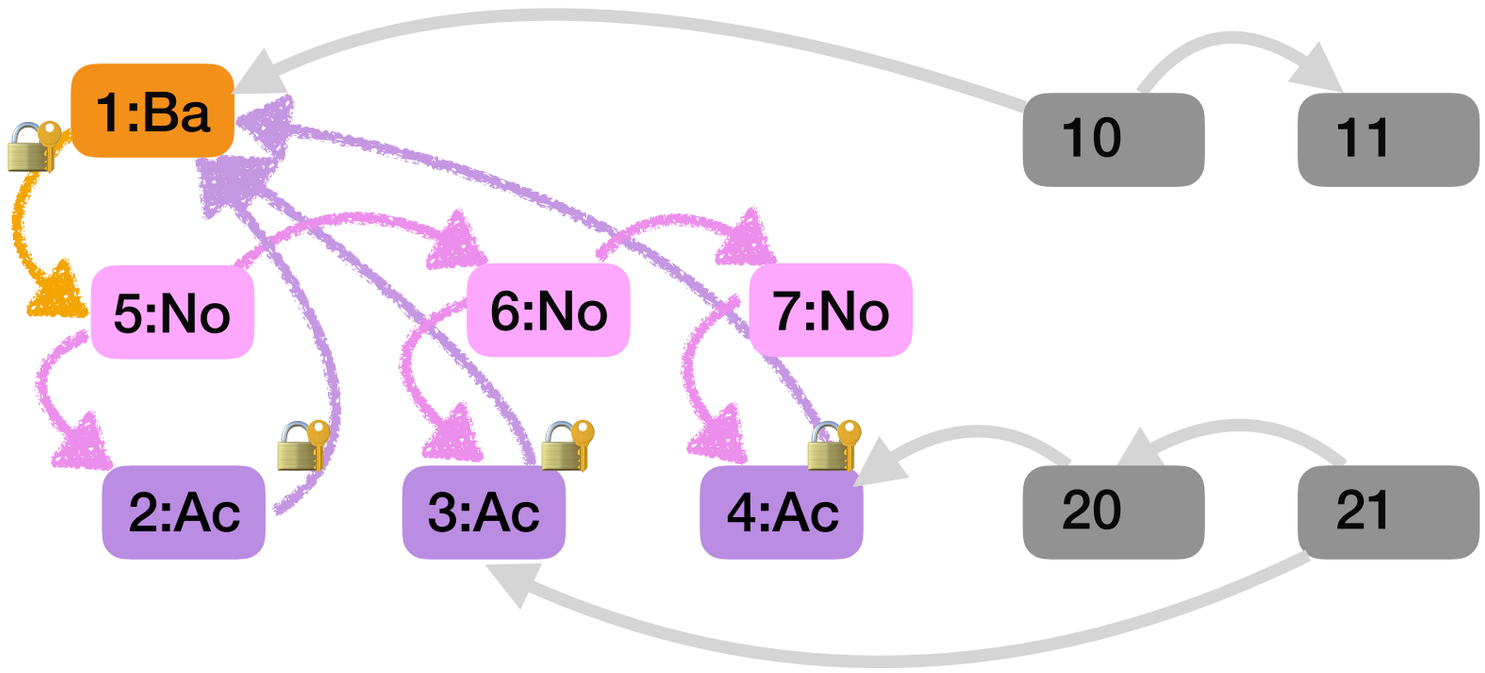
# MBA2s: safe Code

```
class Bank {
  instance private fld book // a Node
  Bank( ){ book=null }
  fun makeAccount(amt){ … }
  fun deposit(src,dest,amt){
    srce=book.get(source);
    destn=book.get(dest);
    if srce.balance>amt then
       { srce.balance-=amt;
         destn.balance+=amt }  }
}
```

```
class Node {
  fld balance       // a number
  fld next          // a Node
  fld theAccount    // an Account
  fun get(acc){
     if theAccount==acc
     then{ this; }
     else{ … next.get(acc) … }
  }
```

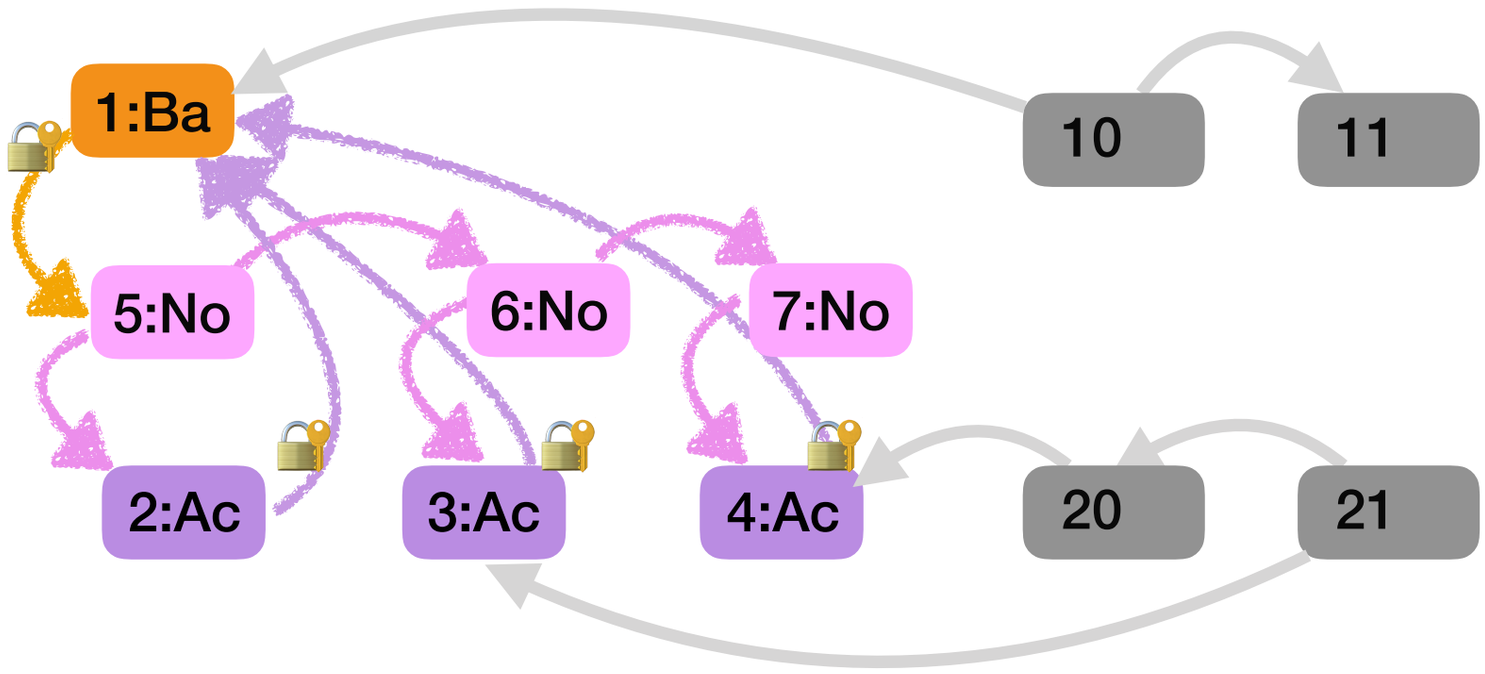
```
class Account {
  instance private fld myBank        // a Bank
  Account(aBank){ myBank=aBank }
  fun deposit(destination,amt)
    { myBank.deposit(this,destination,amt) }
}
```

# MBA2s: Adherence to Policies




- **Pol_1**: With two accounts of same bank one can transfer money between them.
- **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- **Pol_4**: No one can affect the balance of an account they do not have.

# MBA2s: Adherence to Policies




MBA2 ⊨ Pol_1

MBA2 ⊨ Pol_2

MBA2 ⊨ Pol_4

- ✅ **Pol_1**: With two accounts of same bank one can transfer money between them.
- ✅ **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency
- ✅ **Pol_4**: No one can affect the balance of an account they do not have.

# Research Questions:

- Formalize policies  such as  Pol_1,... Pol_5
- Meaning of Mx ⊨ Pol_v

# Today

- Functional ≠ Robustness
- Robustness in terms of the Bank/Account Example
- **Holistic Specification:**
  "Classical assertions" + Time
  + Space
  + Permission
  + Authority
  + "in an open world"
- Examples

# Assertions

# Assertions

e ::= this | x | e.fld | func(e1,…en) | …

# Assertions

e ::= this | x | e.fld | func(e1,…en) | …

A ::= e>e | e=e | P(e1,..en) | …

# Assertions

e ::= this | x | e.fld | func(e1,…en) | …

A ::= e>e | e=e | P(e1,..en) | …

| A → A | A ∧ A | ∃x. A | …

# Assertions

e ::= this | x | e.fld | func(e1,…en) | …

A ::= e>e | e=e | P(e1,..en) | …

| A → A | A ∧ A | ∃x. A | …

| Access(e,e’)

# Assertions

e ::= this | x | e.fld | func(e1,…en) | …

A ::= e>e | e=e | P(e1,..en) | …

| A → A | A ∧ A | ∃x. A | …

| Access(e,e’)

| Changes(e)

# Assertions

e ::= this | x | e.fld | func(e1,…en) | …

A ::= e>e | e=e | P(e1,..en) | …

| A → A | A ∧ A | ∃x. A | …

| Access(e,e’)

| Changes(e)

| •A | ○A

# Assertions

e ::= this | x | e.fld | func(e1,…en) | …

A ::= e>e | e=e | P(e1,..en) | …

| A → A | A ∧ A | ∃x. A | …

| Access(e,e’)

| Changes(e)

| •A | ○A

| A @ S

# Assertions

e ::= this | x | e.fld | func(e1,…en) | …

A ::= e>e | e=e | P(e1,..en) | …

| A → A | A ∧ A | ∃x. A | …

| Access(e,e’)

| Changes(e)

| •A | ○A

| A @ S

| x.Call(y,m,z1,..zn)

# Assertions

e ::= this | x | e.fld | func(e1,…en) | …

A ::= e>e | e=e | P(e1,..en) | …

| A → A | A ∧ A | ∃x. A | …

| Access(e,e’) permission

| Changes(e) authority

| •A | ○A time

| A @ S space

| x.Call(y,m,z1,..zn) call

# Formalizing Pol_1

- **Pol_1**: With two accounts of same bank one can transfer money between them.

# Formalizing Pol_1

- **Pol_1**: With two accounts of same bank one can transfer money between them.

$$\begin{aligned} \text{Pol\_1} \equiv\ & a1{:}\text{Account} \wedge a2{:}\text{Account} \wedge a1 \neq a2 \wedge \\ & a1.\text{myBank} = a2.\text{myBank} \wedge \\ & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \wedge \\ & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \wedge \\ & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \wedge \\ & \rightarrow \end{aligned}$$

# Formalizing Pol_1

- **Pol_1**: With two accounts of same bank one can transfer money between them.

$$\begin{aligned} \text{Pol\_1} \equiv\ & a1{:}\text{Account} \wedge a2{:}\text{Account} \wedge a1 \neq a2 \wedge \\ & a1.\text{myBank} = a2.\text{myBank} \wedge \\ & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \wedge \\ & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \wedge \\ & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \wedge \end{aligned}$$

$$\rightarrow$$

- ( a1.balance = … - amt ∧
  a2.balance = … + amt )

# Formalizing Pol_1

- **Pol_1**: With two accounts of same bank one can transfer money between them.

$$
\begin{aligned}
\text{Pol\_1} \equiv\ & a1{:}\text{Account} \wedge a2{:}\text{Account} \wedge a1 \neq a2 \wedge \\
& a1.\text{myBank} = a2.\text{myBank} \wedge \\
& a1.\text{balance} = b1 > \text{amt} \wedge \\
& a2.\text{balance} = b2 \wedge \\
& \_.\text{Call}(a1,\text{transfer},a2,\text{amt}) \wedge
\end{aligned}
$$

$$\rightarrow$$

# Formalizing Pol_1

- **Pol_1**: With two accounts of same bank one can transfer money between them.

$$
\begin{aligned}
\text{Pol\_1} \equiv\ & a1{:}\text{Account} \wedge a2{:}\text{Account} \wedge a1 \neq a2 \wedge \\
& a1.\text{myBank} = a2.\text{myBank} \wedge \\
& a1.\text{balance} = b1 > \text{amt} \wedge \\
& a2.\text{balance} = b2 \wedge \\
& \_.\text{Call}(a1,\text{transfer},a2,\text{amt}) \wedge
\end{aligned}
$$

$$\rightarrow$$

- $( a1.\text{balance} = b1 - \text{amt} \wedge$
  $a2.\text{balance} = b2 + \text{amt} )$

# Formalizing Pol_2

- **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency

*This says*: If some execution which starts now and which involves at most the objects from S modifies b.currency at some future time, then at least one of the objects in S can access b directly now, and this object is not internal to b.

# Formalizing Pol_2

- **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency

$$\text{Pol\_2} \triangleq b{:}\text{Bank} \wedge \bullet(\text{Changes}(b.\text{currency})) \; @ \; S \rightarrow$$

*This says*: If some execution which starts now and which involves at most the objects from S modifies b.currency at some future time, then at least one of the objects in S can access b directly now, and this object is not internal to b.

# Formalizing Pol_2

- **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency

$$\text{Pol\_2} \triangleq \text{b:Bank} \wedge \bullet(\text{Changes(b.currency)}) @ S \rightarrow \exists o \in S.[\, \text{Access}(o,b) \wedge o \notin \text{Internal}(b) \,]$$

*This says*: If some execution which starts now and which involves at most the objects from S modifies b.currency at some future time, then at least one of the objects in S can access b directly now, and this object is not internal to b.

# Formalizing Pol_2 -2

- **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency

$$\begin{aligned} \text{Pol\_2} \triangleq\ & b{:}\text{Bank} \wedge \\ & \bullet(\ \text{Changes}(b.\text{currency})\ )\ @\ S \\ & \rightarrow \\ & \exists o \in S.[\ \text{Access}(o,b) \wedge o \notin \text{Internal}(b)\ ] \end{aligned}$$

equivalent to

$$\begin{aligned} & b{:}\text{Bank} \wedge \\ & \forall o \in S.[\ \neg\ \text{Access}(o,b) \vee o \in \text{Internal}(b)\ ] \\ & \rightarrow \\ & \neg\ (\bullet(\ \text{Changes}(b.\text{currency})\ )\ @\ S\ ) \end{aligned}$$

# Formalizing Pol_2 -3

- **Pol_2**: Only someone with the Bank of a given currency can violate conservation of that currency

Pol_2 reformulated

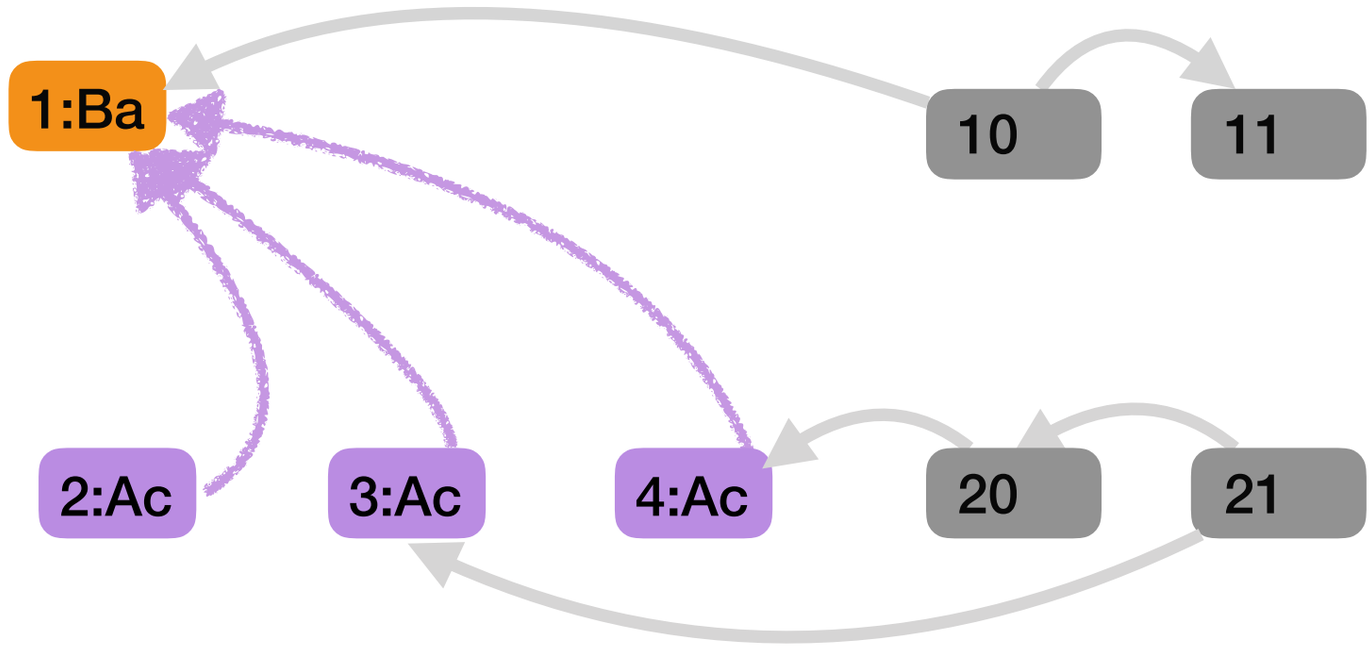
$$b{:}Bank \;\wedge\; \forall o \in S.[\; Access(o,b) \;\rightarrow\; o \in Internal(b)\;] \;\rightarrow\; \neg\,(\bullet\,(\,Changes(b.currency)\,)\;@\;S\,)$$

*This says*: A set S whose elements have direct access to b only if they are internal to b is insufficient to modify b.currency.

Internal

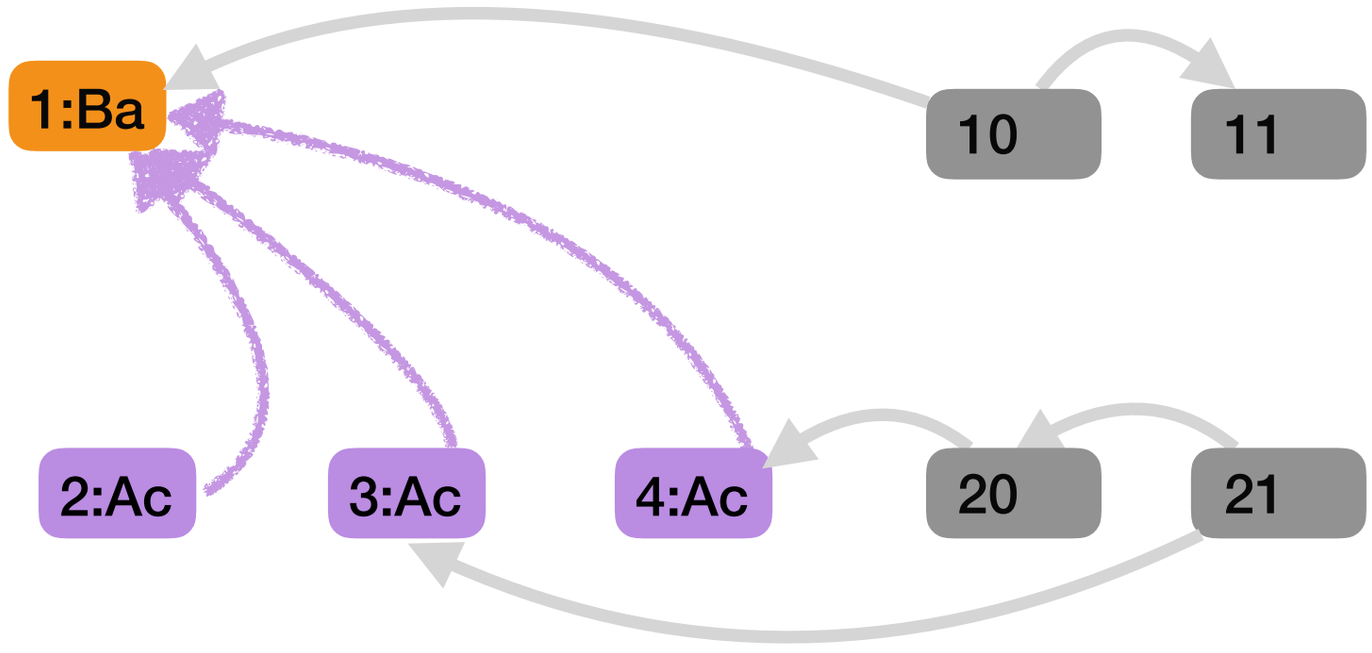
# Internal

MBA1:

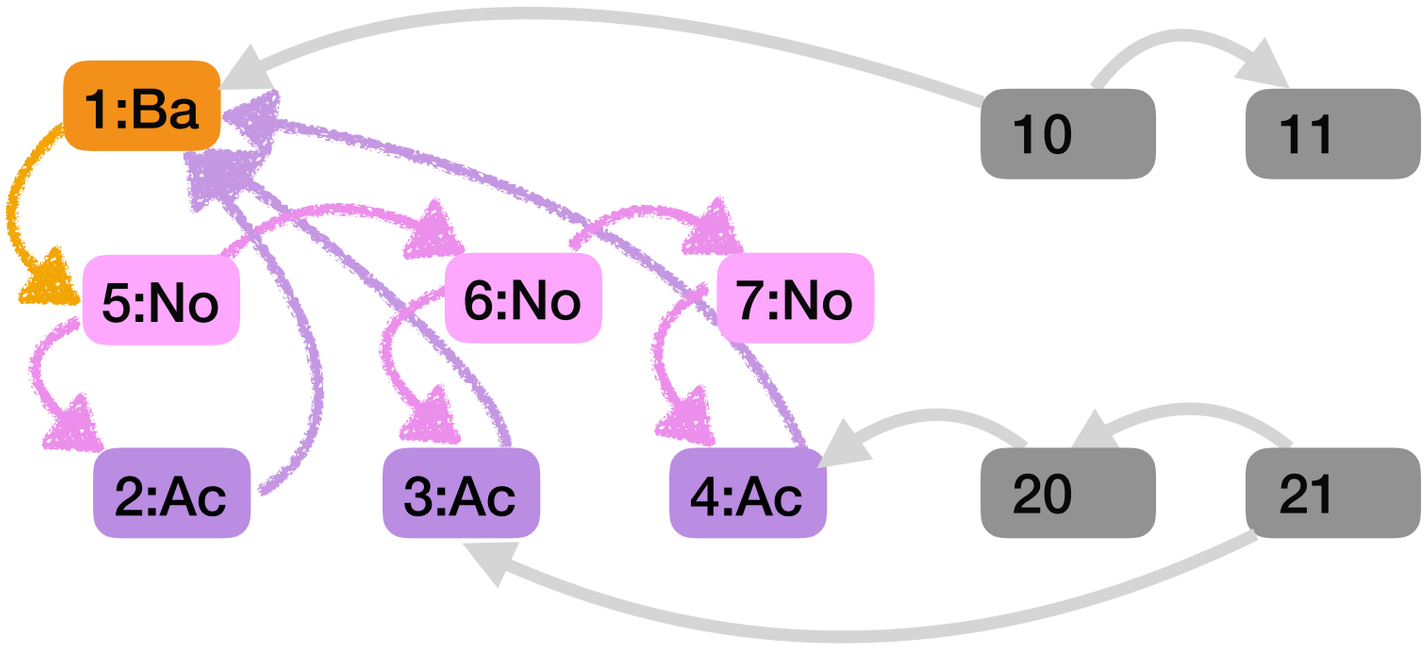


# Internal

MBA1:

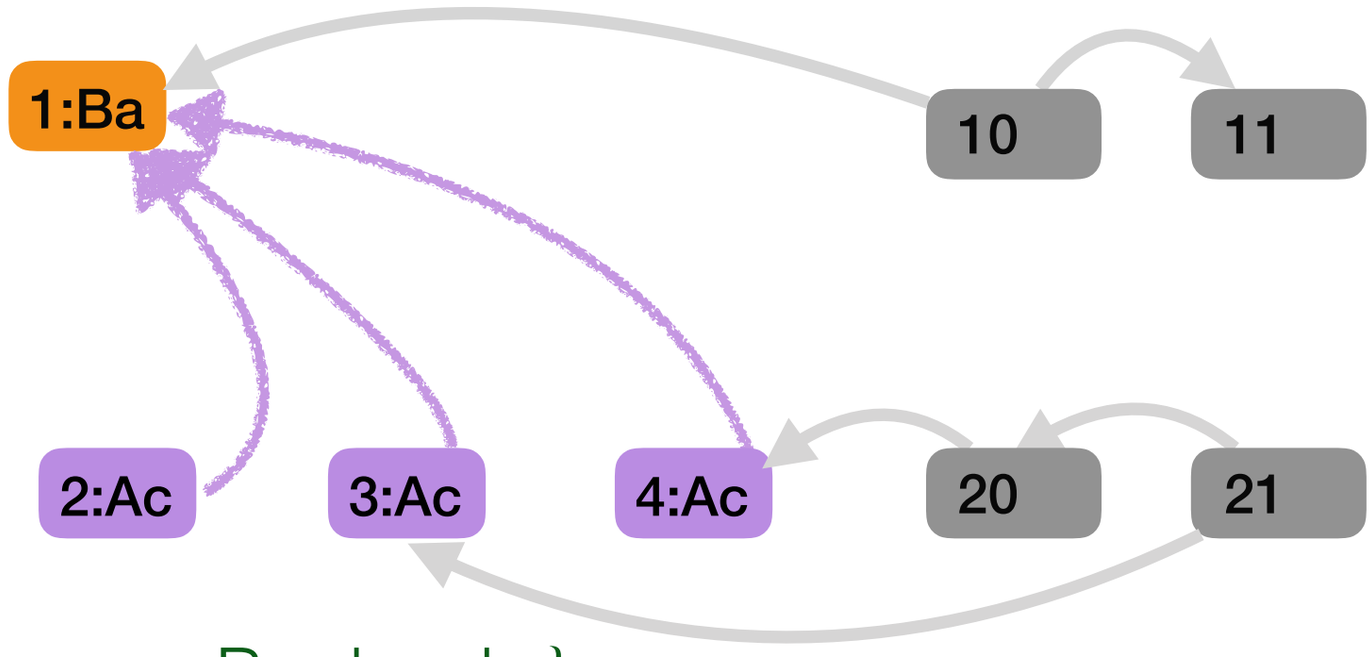



MBA2:



# Internal

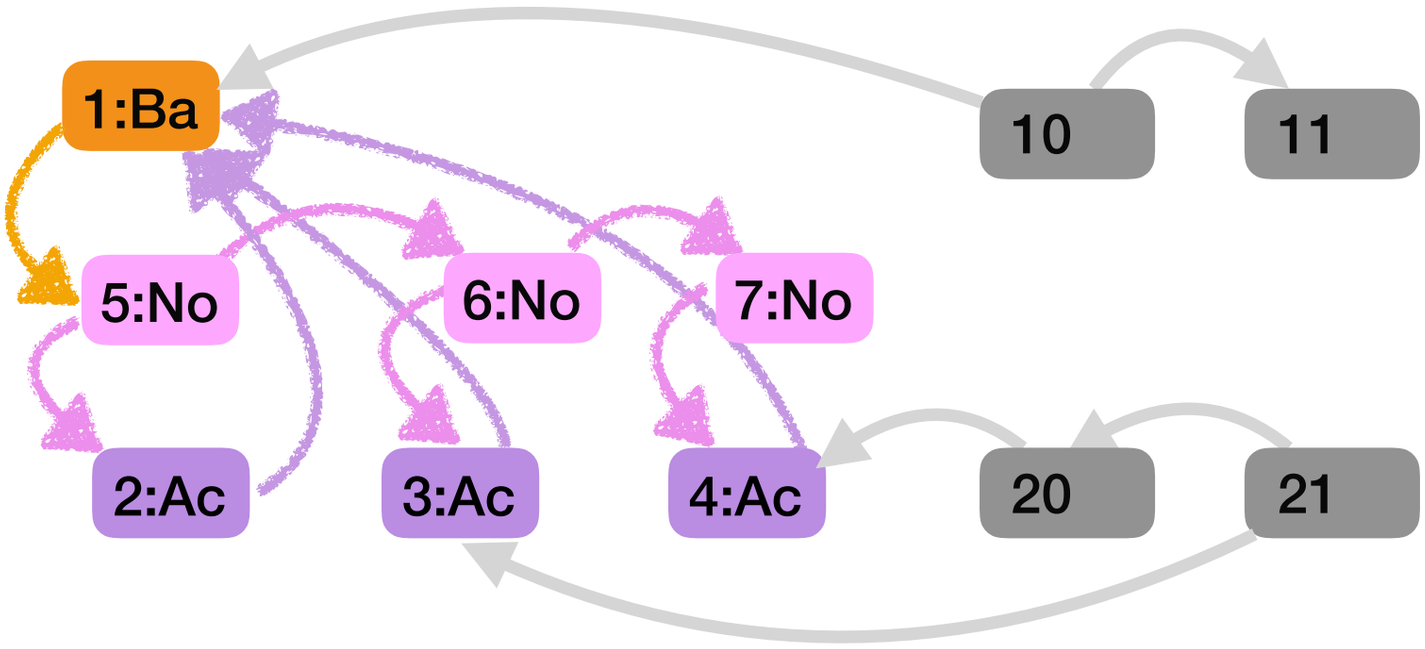
MBA1:




**Internal**(b) ≡

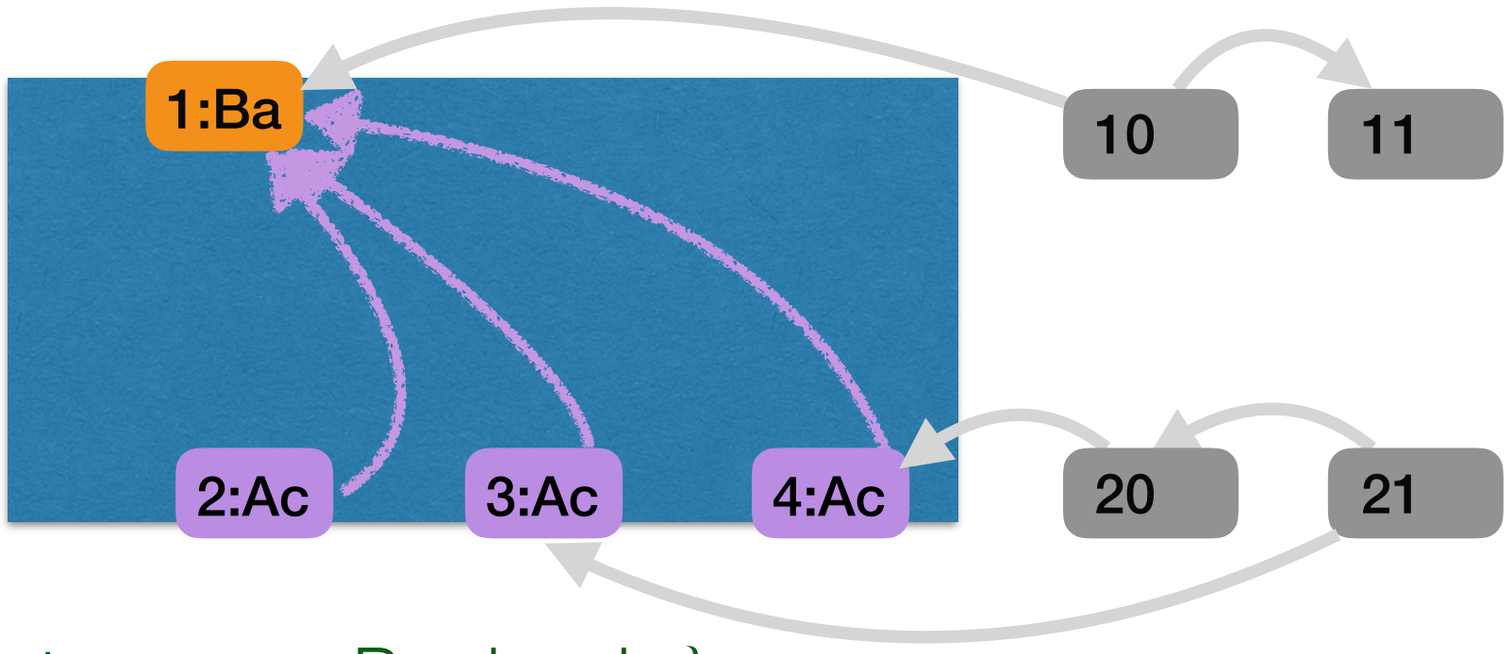
{ b } ∪ { a | a:Account ∧ a.myBank = b }

MBA2:



# Internal

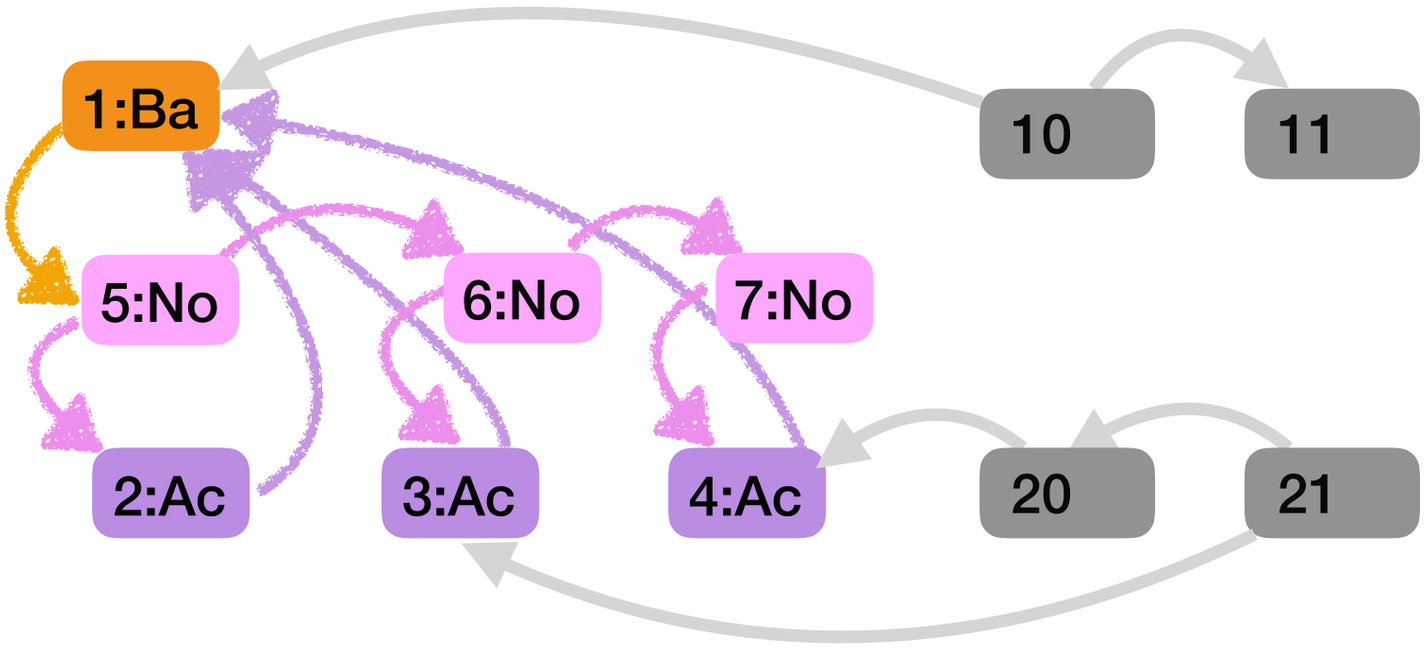
MBA1:




**Internal**(b) ≡

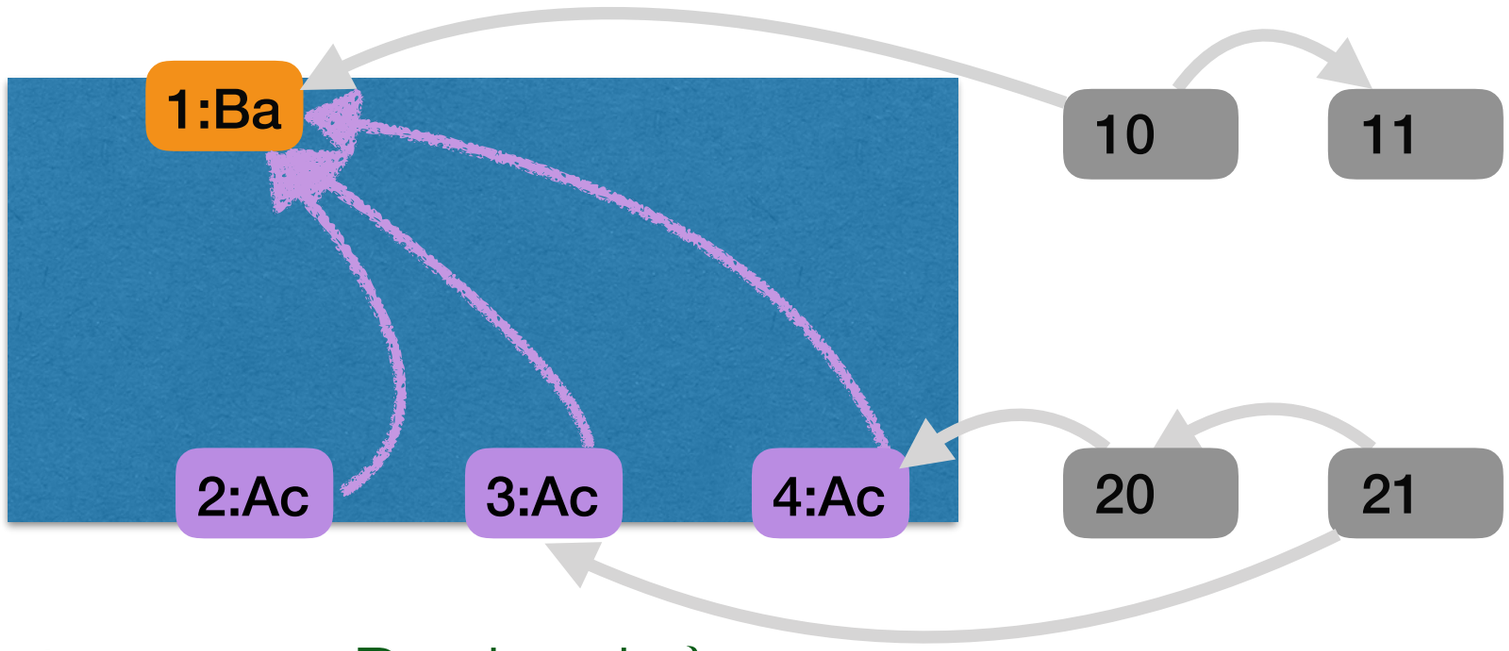
{ b } ∪ { a | a:Account ∧ a.myBank = b }

MBA2:



# Internal

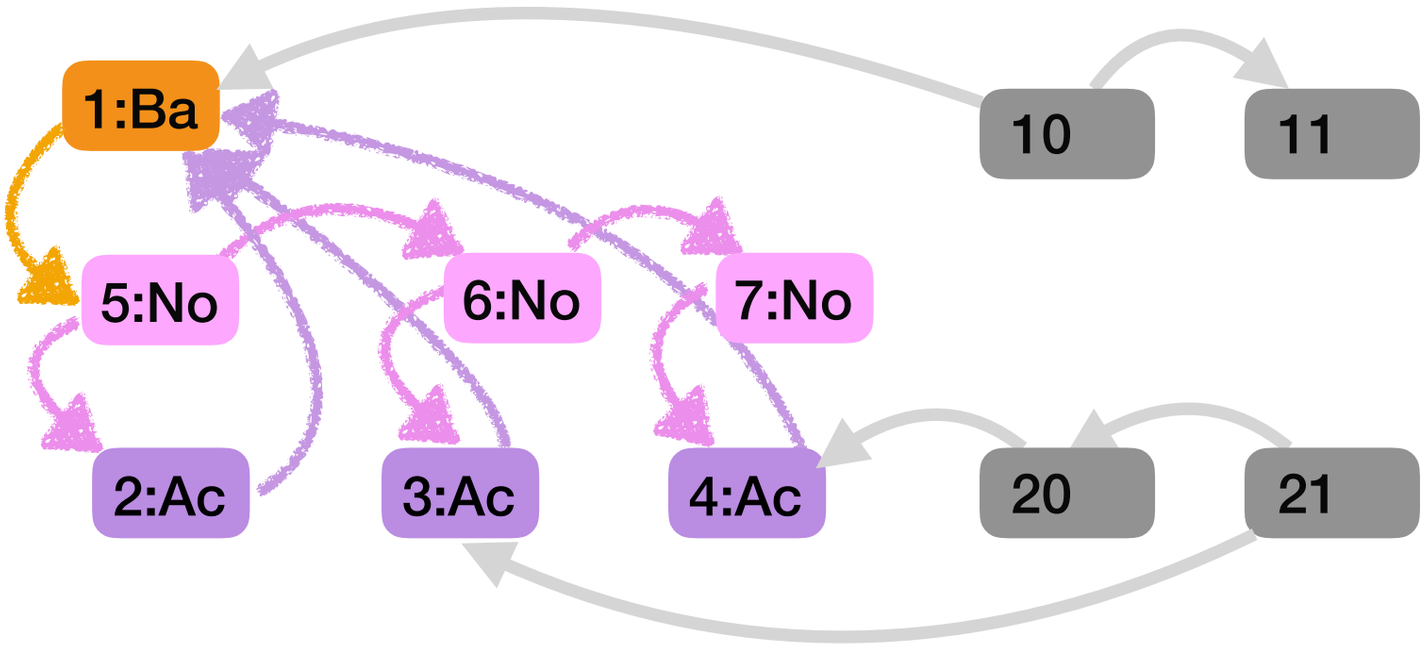
MBA1:




**Internal**(b) ≡

{ b } ∪ { a | a:Account ∧ a.myBank = b }

MBA2:



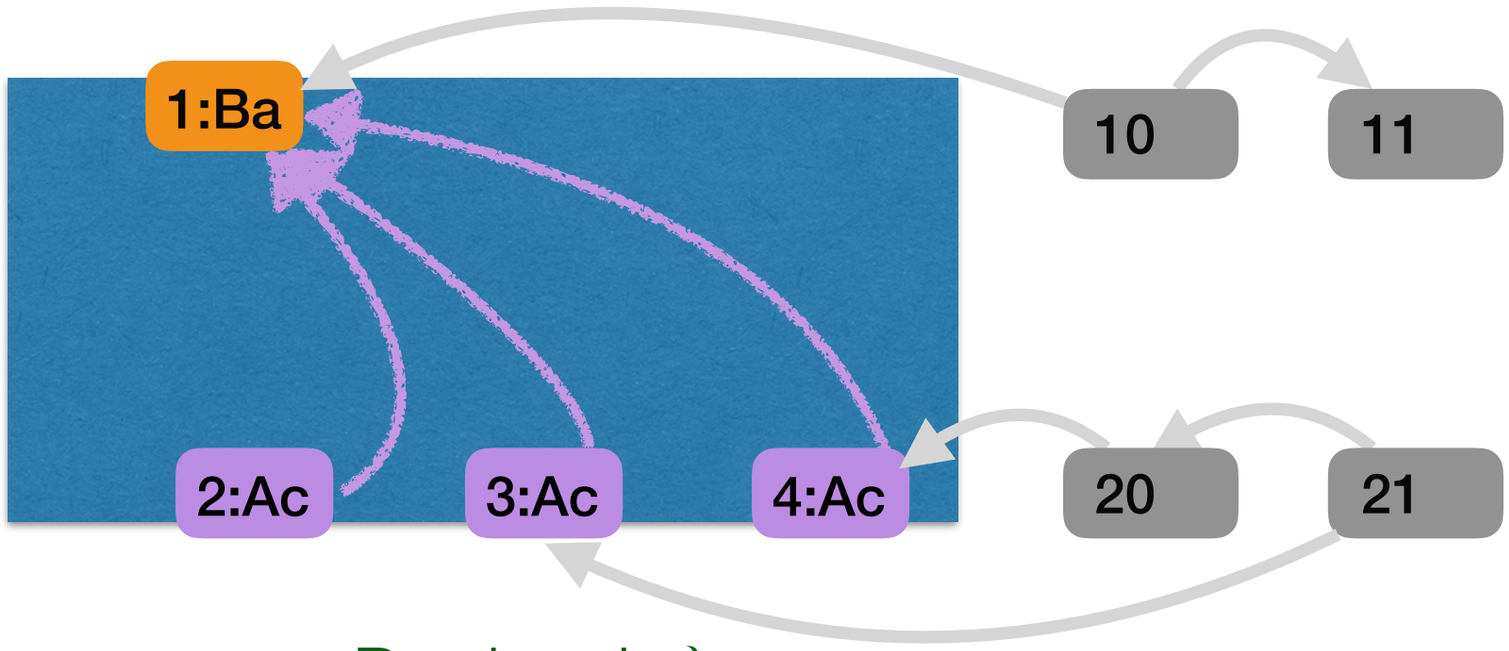

**Internal**(b) ≡

{ b } ∪ { n | n:Node ∧ ∃k.b.book$^k$.next=n } ∪

{ n | a:Account ∧ ∃k.b.book$^k$.myAccount=a }

# Internal

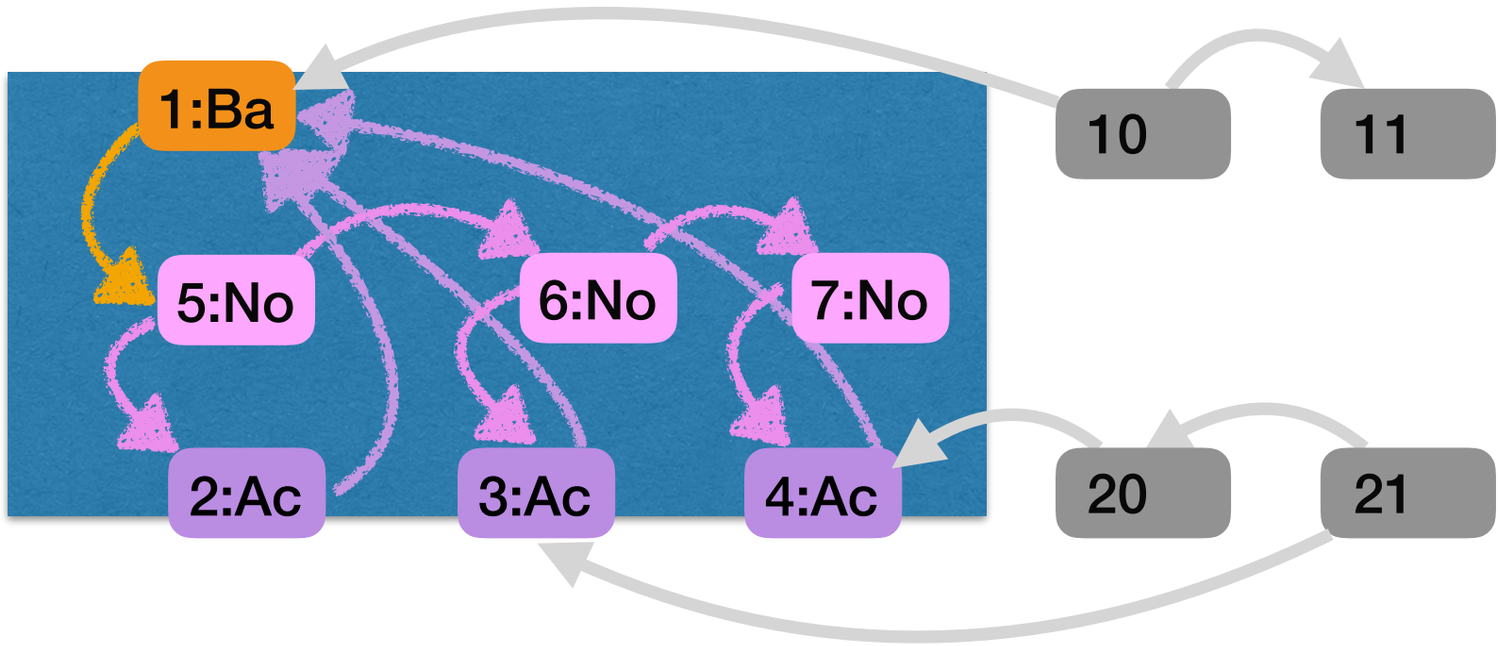
MBA1:




**Internal**(b) ≡

{ b } ∪ { a | a:Account ∧ a.myBank = b }

MBA2:



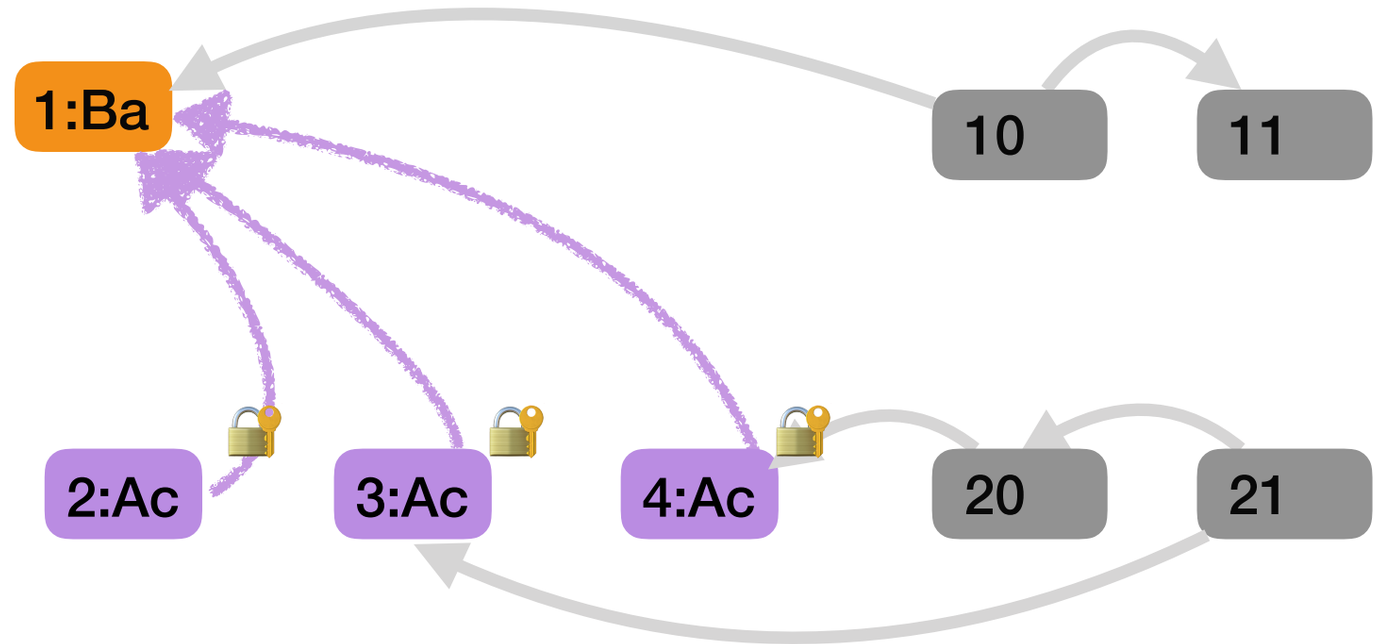

**Internal**(b) ≡

{ b } ∪ { n | n:Node ∧ ∃k.b.book$^k$.next=n } ∪

{ n | a:Account ∧ ∃k.b.book$^k$.myAccount=a }

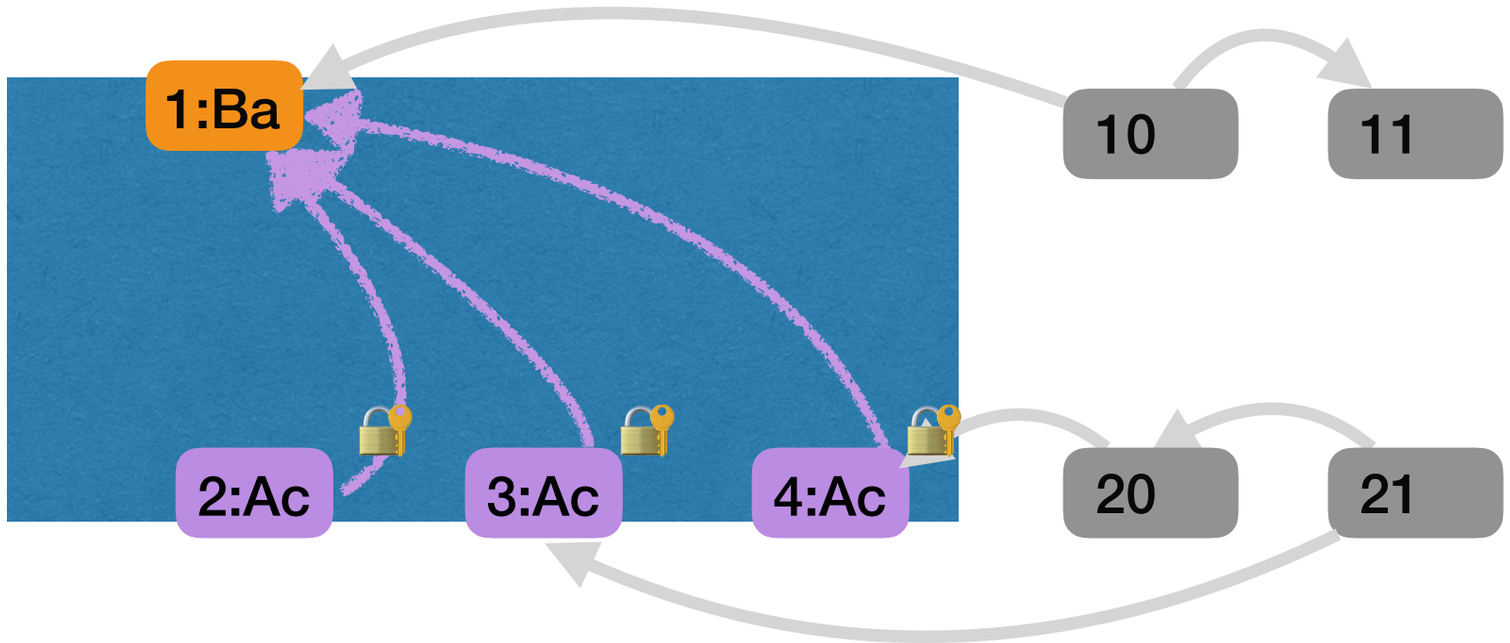
# Guarantees of PoI_2 in MBA1s

PoI_2 ≡ b:Bank ∧
∀o∈S.[ ¬ Access(o,b) ∨ o∈Internal(b) ]
→
¬ (• ( Changes(b.currency) ) @ S )



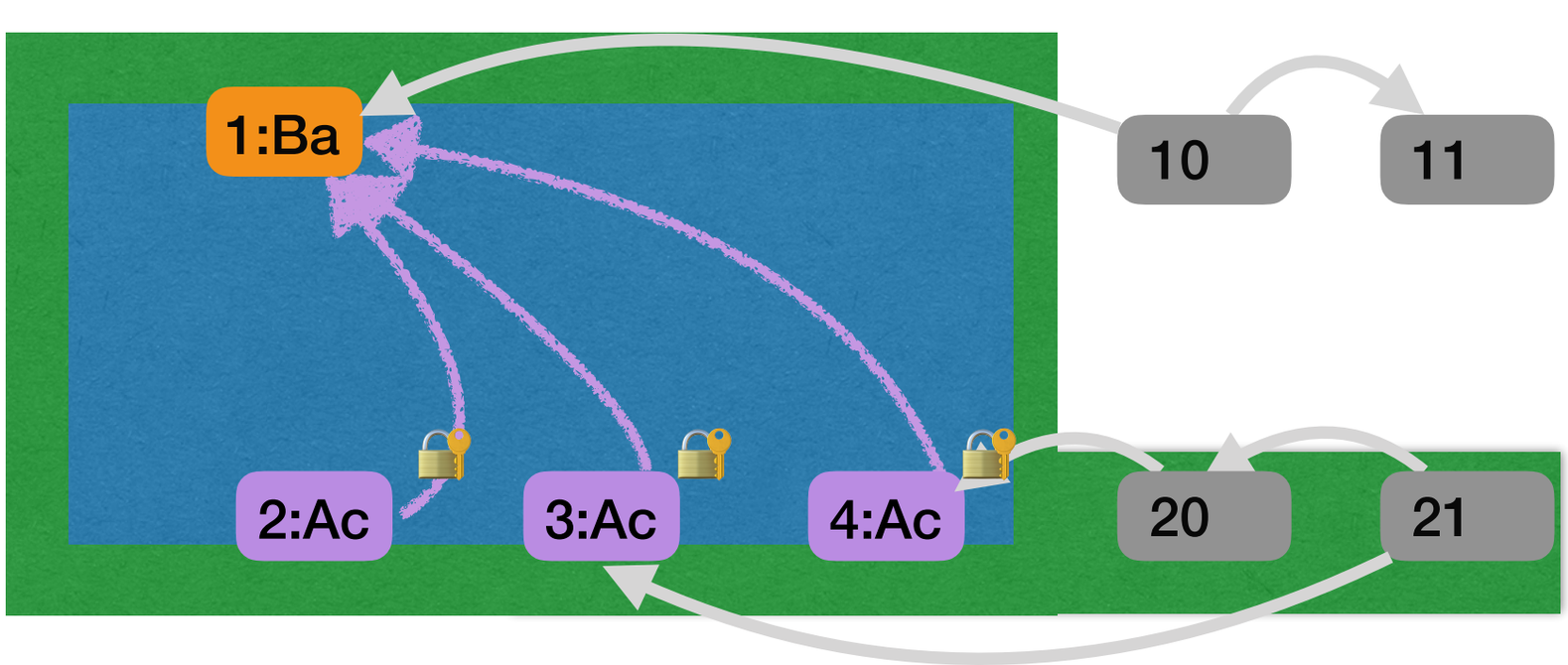
# Guarantees of PoI_2 in MBA1s

$$\text{PoI\_2} \equiv b{:}\text{Bank} \wedge \forall o \in S.[\, \neg\, \text{Access}(o,b) \vee o \in \text{Internal}(b)\,] \rightarrow \neg\,(\bullet\,(\,\text{Changes}(b.\text{currency})\,)\,@\,S\,)$$



# Guarantees of PoI_2 in MBA1s

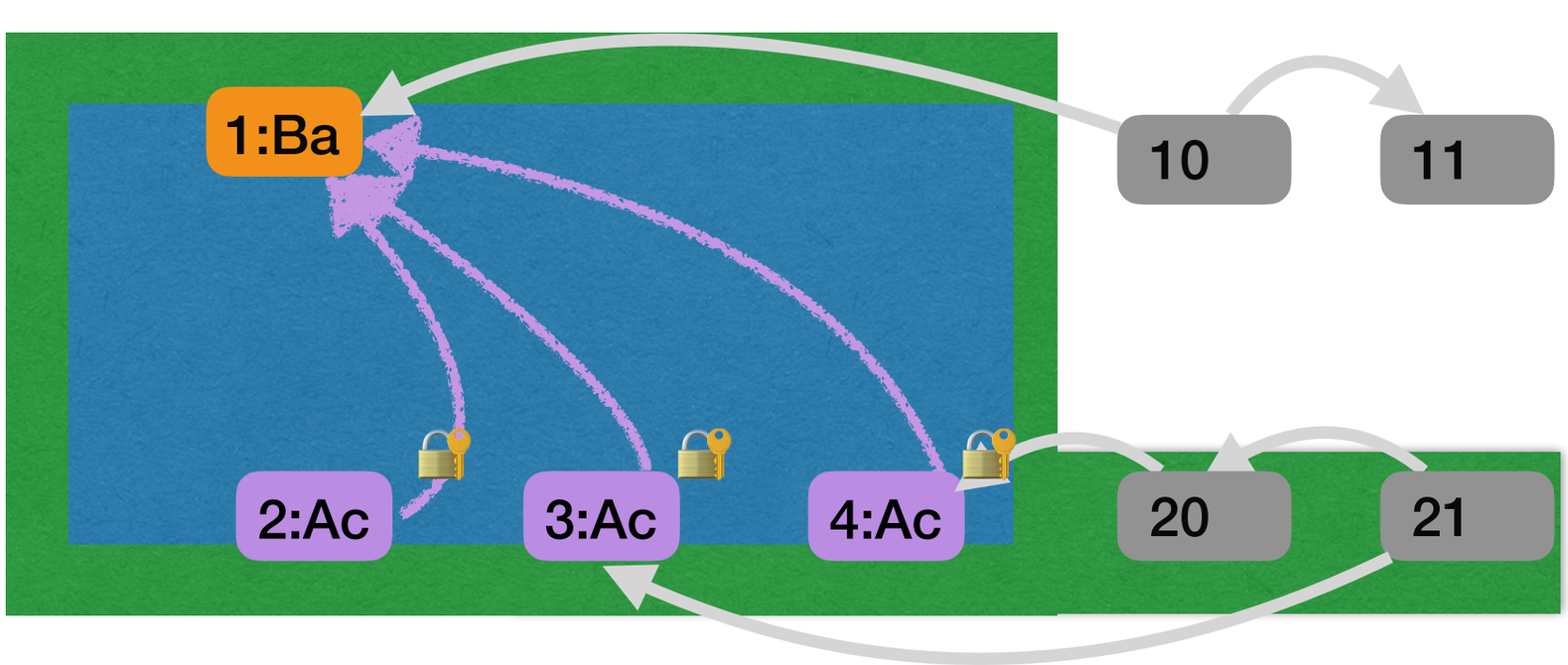
PoI_2 ≡ b:Bank ∧
∀o∈S.[ ¬ Access(o,b) ∨ o∈Internal(b) ]
→
¬ (• ( Changes(b.currency) ) @ S )




1 is not on stack, and execution involves at most 1,2,3,4,20,21

# Guarantees of Pol_2 in MBA1s

$$\begin{aligned} \text{Pol\_2} \equiv\ & b{:}\text{Bank} \wedge \\ & \forall o \in S.[\ \neg\ \text{Access}(o,b)\ \vee\ o \in \text{Internal}(b)\ ] \\ & \qquad \rightarrow \\ & \neg\ (\bullet\ (\ \text{Changes}(b.\text{currency})\ )\ @\ S\ ) \end{aligned}$$




1 is not on stack, and execution involves at most 1,2,3,4,20,21

——> no change in 1.currency

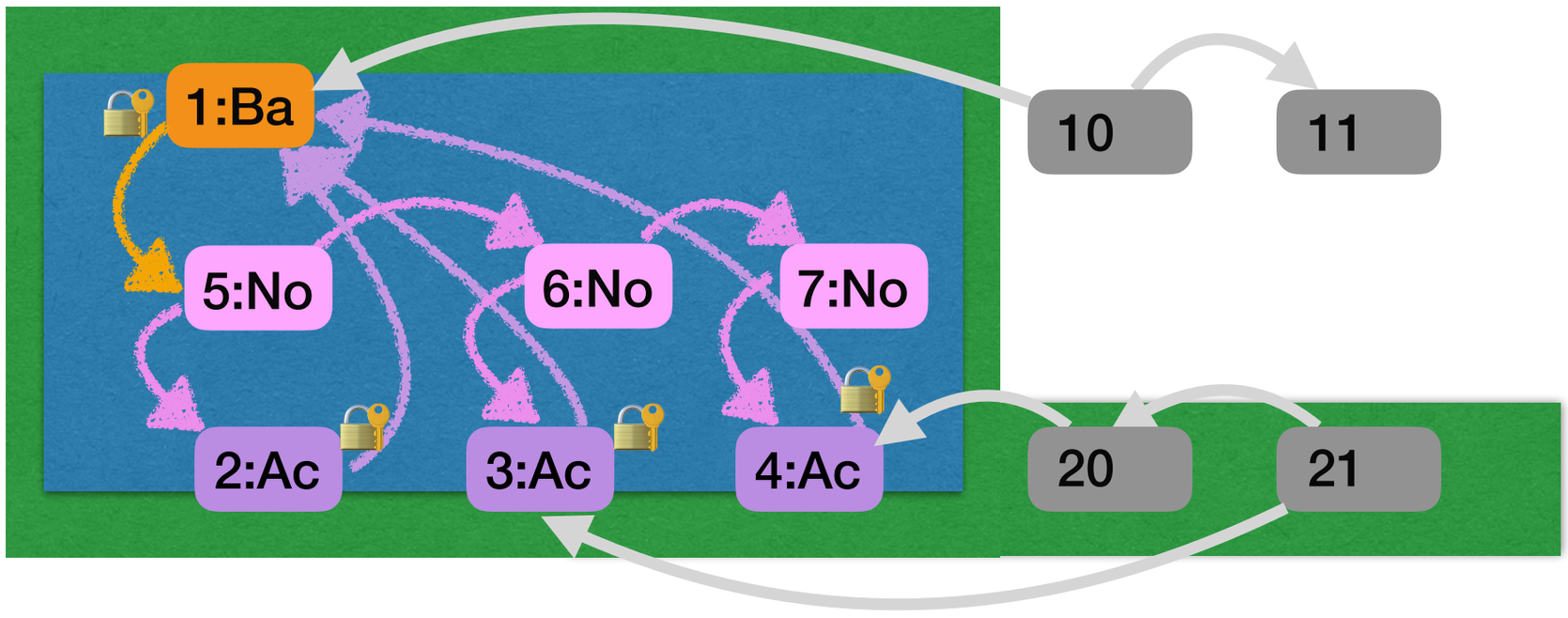
# Guarantees of PoI_2 in MBA2s

PoI_2 ≡ b:Bank ∧

∀o∈S.[ ¬ Access(o,b) ∨ o∈Internal(b) ]

→

¬ (• ( Changes(b.currency) ) @ S )

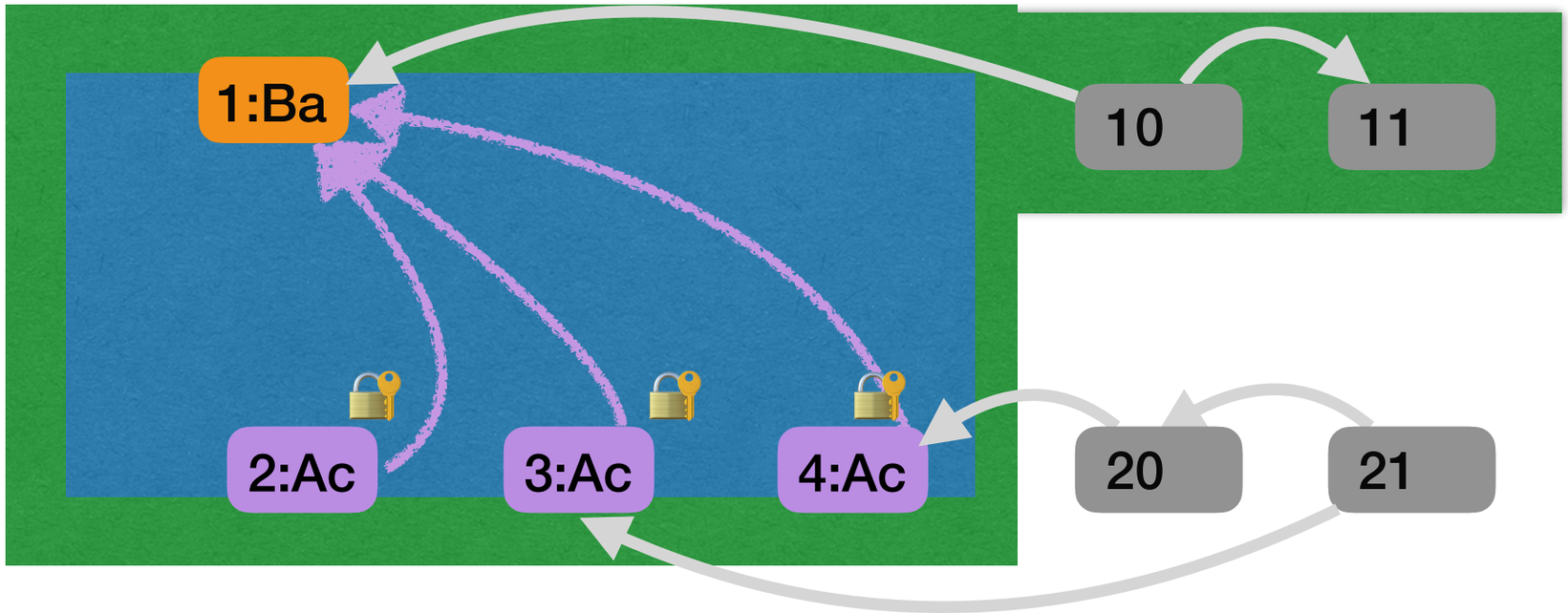



1 is not on stack, and execution involves at most 1,2,3,4,**5**,**6**,**7**,20,21;

——> no change in 1.currency

# *Absence* of Guarantee of Pol_2 in MBA1s

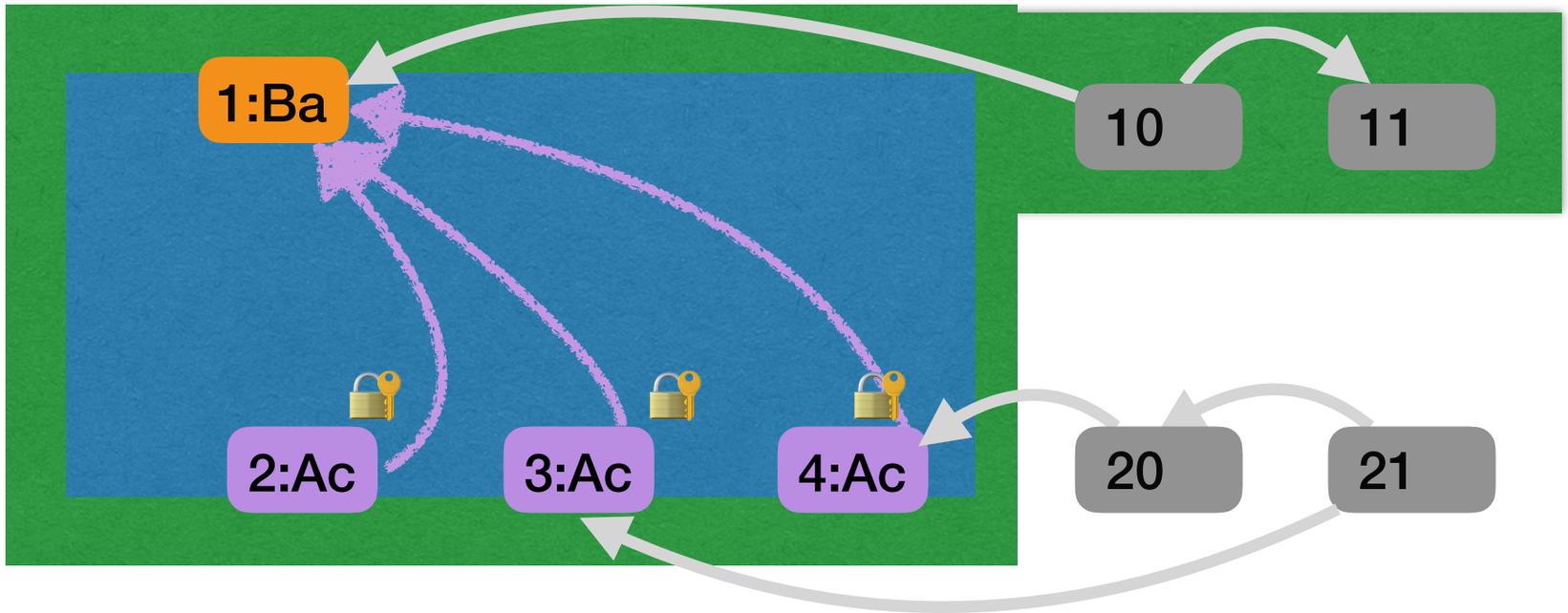
$$\text{Pol\_2} \equiv b{:}\text{Bank} \wedge \forall o \in S.[\, \neg\, \text{Access}(o,b) \vee o \in \text{Internal}(b)\,] \rightarrow \neg\,(\bullet\,(\,\text{Changes}(b.\text{currency})\,)\,@\,S\,)$$




1 is not on stack, and execution involves at most 1,2,3,4,10,11;

# *Absence* of Guarantee of Pol_2 in MBA1s

$$\text{Pol\_2} \equiv b{:}\text{Bank} \wedge \forall o \in S.[\, \neg\, \text{Access}(o,b) \vee o \in \text{Internal}(b)\,] \rightarrow \neg\,(\bullet\,(\,\text{Changes}(b.\text{currency})\,)\,@\,S\,)$$

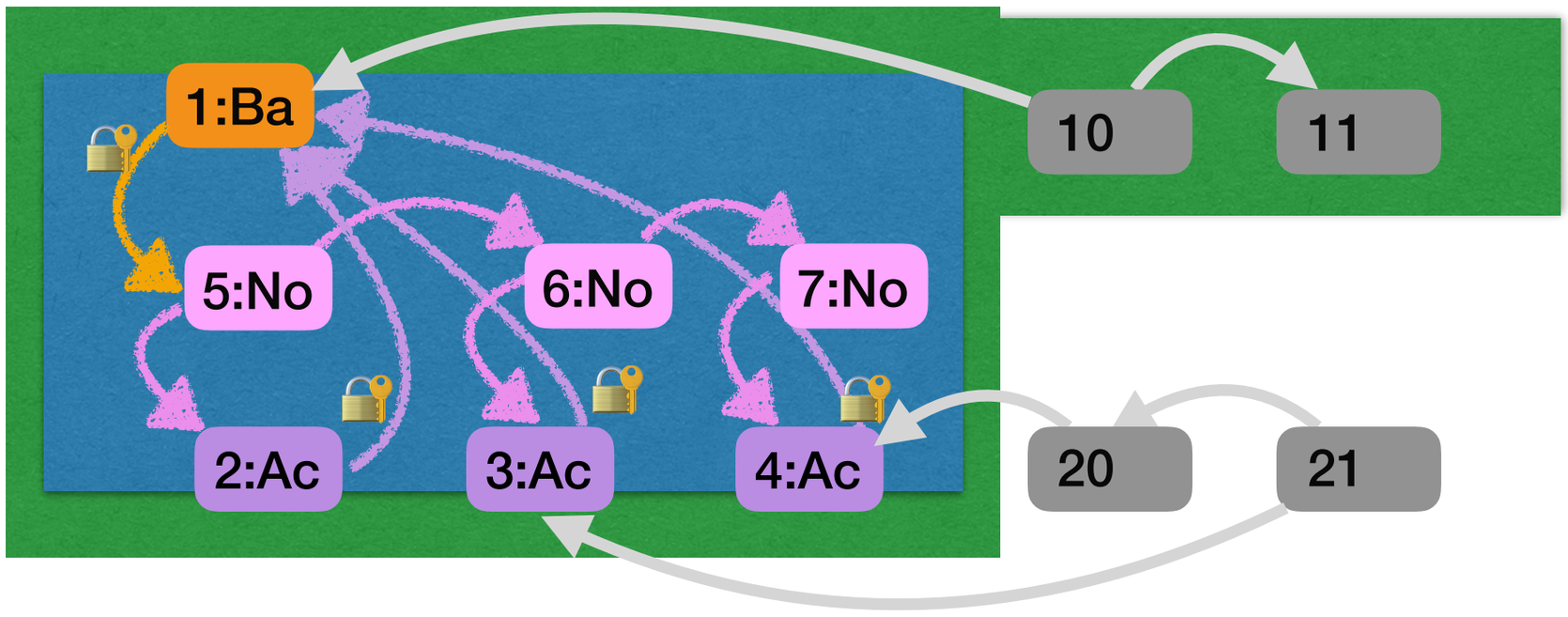



1 is not on stack, and execution involves at most 1,2,3,4,10,11;

change in 1.currency *possible*

# *Absence of* Guarantee of Pol_2 in MBA2

Pol_2 ≡ b:Bank ∧
∀o∈S.[ ¬ Access(o,b) ∨ o∈Internal(b) ]
→
¬ (• ( Changes(b.currency) ) @ S )




1 is not on stack, and execution involves at most 1,2,3,4,10,11;

change in 1.currency *possible*

# Formalizing Pol_4

- **Pol_4**: No one can affect the balance of an account they do not have.

# Formalizing Pol_4

- **Pol_4**: No one can affect the balance of an account they do not have.

Pol_4 ≜ a:Account ∧
• ( Changes(a.balance) ) @ S
→

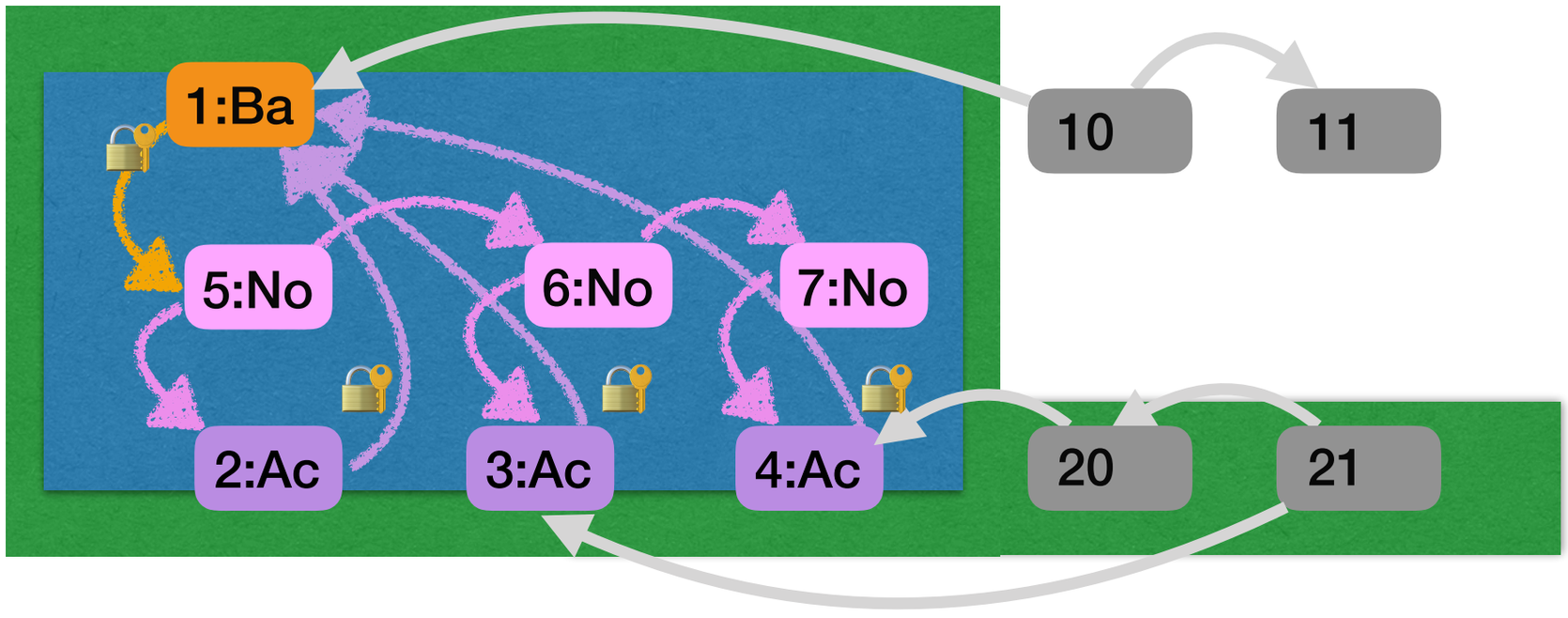
# Formalizing Pol_4

- **Pol_4**: No one can affect the balance of an account they do not have.

$$\text{Pol\_4} \triangleq a{:}\text{Account} \wedge \bullet(\text{Changes}(a.\text{balance})) \text{ @ } S \rightarrow \exists o \in S.[\text{Access}(o,a) \wedge o \notin \text{Internal}(a)]$$

# Guarantees of Pol_4 in MBA2s

Pol_2 ≡ a:Account ∧ a∉FrameVals ∧
• ( Changes(a.balance) ) @ S
→
∃o∈S.[ Access(o,a) ∧ o∉Internal(a) ]




2 not on stack, and execution involves at most 1,2,3,4,**5**,**6**,**7**,20,21;
——> no change in 2.balance

# Giving a meaning to our Assertions

# Giving a meaning to our Assertions

We define  in a“conventional” way (omit from slides):

| | |
|---|---|
| module | M : Ident —-> ClassDef ∪ PredicateDef  ∪ FunctionDef |
| configuration | σ  :  Heap × Continuations ×  Expression |
| execution | M, σ ⇝ σ’ |

# Giving a meaning to our Assertions

We define  in a“conventional” way (omit from slides):

| | |
|---|---|
| module | M : Ident —-> ClassDef ∪ PredicateDef ∪ FunctionDef |
| configuration | σ : Heap × Continuations × Expression |
| execution | M, σ ⇝ σ’ |

Define module concatenation * so that

M*M’ undefined, iff dom(M)∩dom(M’) ≠∅

otherwise

M*M’(id) = M(id) if M’(id) undefined, else M’(id)

# Giving a meaning to our Assertions

We define  in a“conventional” way (omit from slides):

| | |
|---|---|
| module | M : Ident —-> ClassDef ∪ PredicateDef ∪ FunctionDef |
| configuration | σ : Heap × Continuations × Expression |
| execution | M, σ ⇝ σ’ |

Define module concatenation * so that

M*M’ undefined, iff dom(M)∩dom(M’) ≠∅

otherwise

M*M’(id) = M(id) if M’(id) undefined, else M’(id)

We will define M, σ ⊨ A

Initial(σ) and Arising(M)

M ⊨ A

# Giving a meaning to our Assertions

We define  in a“conventional” way (omit from slides):

| | |
|---|---|
| module | M : Ident —-> ClassDef ∪ PredicateDef ∪ FunctionDef |
| configuration | σ : Heap × Continuations × Expression |
| execution | M, σ ⇝ σ’ |

Define module concatenation * so that

M*M’ undefined, iff dom(M)∩dom(M’) ≠∅

otherwise

M*M’(id) = M(id) if M’(id) undefined, else M’(id)

**Lemma** M*M’ = M’*M

We will define M, σ ⊨ A

Initial(σ) and Arising(M)

M ⊨ A

# Expressions and Assertions - reminder

e ::= this | x | e.fld | func(e1,…en) | …

A ::= e>e | e=e | P(e1,..en) | …

| A → A | A ∧ A | ∃x. A | …

| Access(e,e') permission

| Changes(e) authority

| •A | ○A time

| A @ S space

| Call(x,m,x1,..xn) call

# Giving a meaning to Expressions

e ::= this | x | e.fld | func(e1,…en) | …

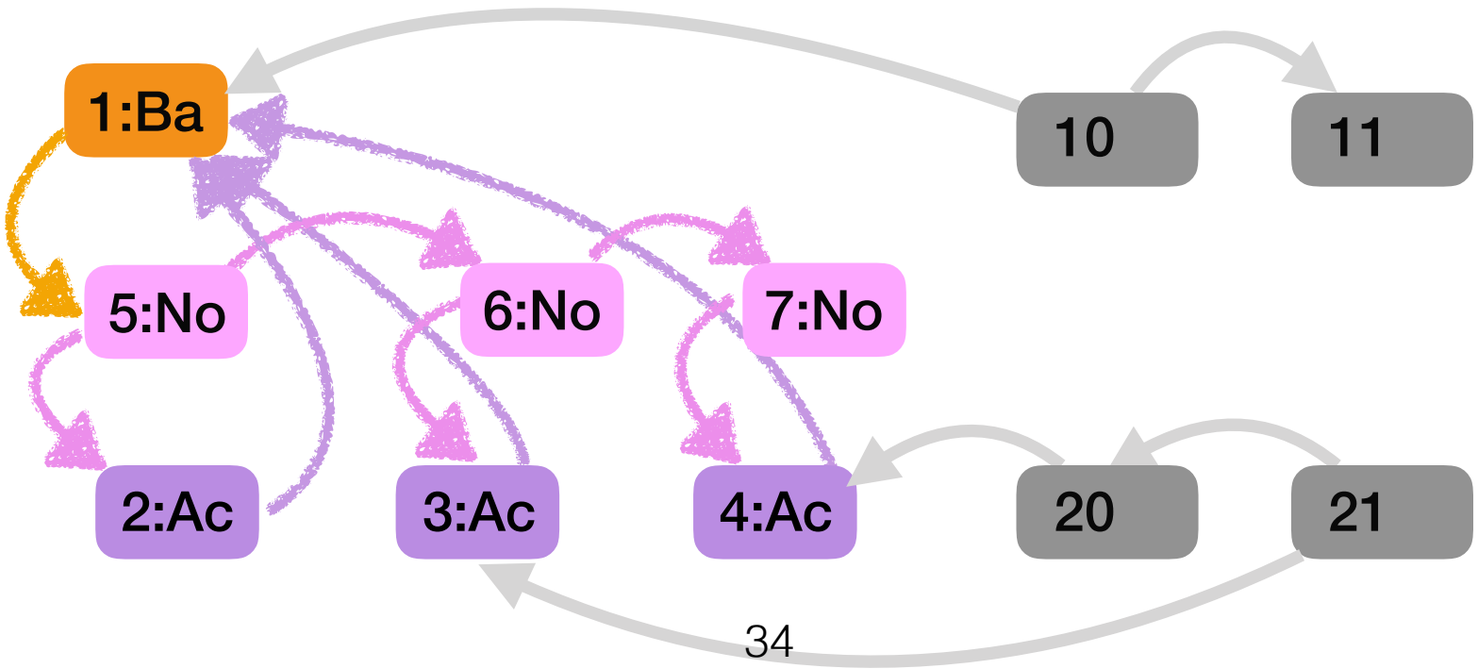
Define $\lfloor e \rfloor_{M,\sigma}$ as expected

# Giving a meaning to Expressions

e ::= this | x | e.fld | func(e1,…en) | …

Define $\lfloor e \rfloor_{M,\sigma}$ as expected

Eg, with x mapping to 3, we have $\lfloor x.myBank.book.next \rfloor_{M,\sigma} = 6$



# Giving a meaning to Assertions

“Conventional part”

A ::= e>e | A → A | P(e1,..en) | ∃x.A | …

# Giving a meaning to Assertions

"Conventional part"

A ::= e>e | A → A | P(e1,..en) | ∃x.A | …

We define M, σ ⊨ A

# Giving a meaning to Assertions

"Conventional part"

$A ::= e{>}e \mid A \rightarrow A \mid P(e1,..en) \mid \exists x.A \mid \ldots$

We define $M, \sigma \models A$

$M,\sigma \models e{>}e'$ iff $\lfloor e \rfloor_{M,\sigma} > \lfloor e' \rfloor_{M,\sigma}$

$M,\sigma \models A{\rightarrow}A'$ iff $M,\sigma \models A$ implies $M,\sigma \models A'$

$M,\sigma \models P(e_1,\ldots e_n)$ iff $M,\sigma \models M(P)[x_1 \mapsto \lfloor e_1 \rfloor_{M,\sigma}, \ldots x_n \mapsto \lfloor e_n \rfloor_{M,\sigma}]$

$M,\sigma \models \exists x.A$ iff $M,\sigma[z \mapsto \iota] \models A[x \mapsto z]$
for some $\iota \in dom(\sigma.heap)$, and $z$ free in $A$

# Giving meaning to Assertions

"Unconventional part"

A ::= Access(x,y) | Changes(e) | •A | ○A | A @ S | Call(x,m,$x_1$,..$x_n$)

# Giving meaning to Assertions

"Unconventional part"

A ::= Access(x,y) | Changes(e) | •A | ○A | A @ S | Call(x,m,$x_1$,..$x_n$)

M,σ ⊨ Access(e,e') iff $\lfloor e.fld \rfloor_{M,\sigma} = \lfloor e' \rfloor_{M,\sigma}$ for some field fld ∨

$\lfloor this \rfloor_{M,\sigma} = \lfloor e \rfloor_{M,\sigma} \wedge \lfloor y \rfloor_{M,\sigma} = \lfloor e' \rfloor_{M,\sigma}$

∧ y is a parameter of the current function

M,σ ⊨ Changes(e) iff $M, \sigma \rightsquigarrow \sigma' \wedge \lfloor e \rfloor_{M,\sigma} \neq \lfloor e \rfloor_{M,\sigma'}$

M,σ ⊨ •A iff $\exists \sigma', \sigma'', \phi . [\ \sigma = \sigma'.\phi \wedge M,\phi \rightsquigarrow^* \sigma'' \wedge M, \sigma'' \models A\ ]$

M,σ ⊨ ○A iff $\forall \sigma_0 . [\ Initial(\sigma_0) \wedge M,\sigma_0 \rightsquigarrow^* \sigma \rightarrow$

$\exists \sigma_1 . (\ M,\sigma_0 \rightsquigarrow^* \sigma_1 \wedge M,\sigma_1 \rightsquigarrow^+ \sigma \wedge M,\sigma_1 \models A\ )\ ]$

M,σ ⊨ A@S iff $M,\sigma@_{ls} \models A$ where $ls = \lfloor S \rfloor_{M,\sigma}$ and $\sigma@_{ls}$ ...

M,σ ⊨ x.Call(y,m,$z_1$,..$z_n$) iff $\lfloor this \rfloor_{M,\sigma} = \lfloor x \rfloor_{M,\sigma} \wedge \ldots$

# Giving meaning to Assertions
# - the full truth -

$M,\sigma \models$ Access(e,e') iff … as before …

$M,\sigma \models$ Changes(e) iff $M, \sigma \rightsquigarrow \sigma' \wedge \lfloor e \rfloor_{M,\sigma} \neq \lfloor e[z \mapsto y] \rfloor_{M,\sigma'[y \mapsto \sigma(z)]}$

where $\{z\}$=Free(e) $\wedge$ y fresh in e, $\sigma,\sigma'$

$M,\sigma \models \bullet A$ iff $\exists \sigma',\sigma'',\phi.[\ \sigma = \sigma'.\phi \wedge M,\phi \rightsquigarrow^* \sigma' \wedge M, \sigma'[y \mapsto \sigma(z)] \models A[z \mapsto y]\ ]$

where $\{z\}$=Free(A) $\wedge$ y fresh in A, $\sigma,\sigma'$

$M,\sigma \models \circ A$ iff $\forall \sigma_0.[\ \text{Initial}(\sigma_0) \rightarrow \exists \sigma_1.(\ M,\sigma_0 \rightsquigarrow^* \sigma_1 \wedge M,\sigma_1 \rightsquigarrow^+ \sigma \wedge M,\sigma_1[y \mapsto \sigma(z)] \models A[z \mapsto y]\ ]$

where $\{z\}$=Free(A) $\wedge$ y fresh in A, $\sigma_1,\sigma$

$M,\sigma \models$ A@S iff … as before …

$M,\sigma \models$ x.Call(y,m,$z_1$,..$z_n$) iff … as before …

# Giving meaning to Assertions
## - the full truth -

$M,\sigma \models$ Access(e,e') iff … as before …

$M,\sigma \models$ Changes(e) iff $M, \sigma \rightsquigarrow \sigma' \wedge \lfloor e \rfloor_{M,\sigma} \neq \lfloor e[z \mapsto y] \rfloor_{M,\sigma'[y \mapsto \sigma(z)]}$

where $\{z\}$=Free(e) ∧ y fresh in e, σ,σ'

$M,\sigma \models \bullet A$ iff $\exists \sigma', \sigma'', \phi.[\ \sigma = \sigma'.\phi \wedge M,\phi \rightsquigarrow^* \sigma' \wedge M, \sigma'[y \mapsto \sigma(z)] \models A[z \mapsto y]\ ]$

where $\{z\}$=Free(A) ∧ y fresh in A, σ,σ'

$M,\sigma \models \circ A$ iff $\forall \sigma_0.[\ \text{Initial}(\sigma_0) \rightarrow \exists \sigma_1.(\ M,\sigma_0 \rightsquigarrow^* \sigma_1 \wedge M,\sigma_1 \rightsquigarrow^+ \sigma \wedge M,\sigma_1[y \mapsto \sigma(z)] \models A[z \mapsto y]\ ]$

where $\{z\}$=Free(A) ∧ y fresh in A, $\sigma_1$,σ

$M,\sigma \models A@S$ iff … as before …

$M,\sigma \models x.\text{Call}(y,m,z_1,..z_n)$ iff … as before …

# Giving meaning to Assertions
## - the full truth -

$M,\sigma \models$ Access(e,e') iff … as before …

$M,\sigma \models$ Changes(e) iff $M, \sigma \rightsquigarrow \sigma' \wedge \lfloor e \rfloor_{M,\sigma} \neq \lfloor e[z \mapsto y] \rfloor_{M,\sigma'[y \mapsto \sigma(z)]}$

where $\{z\}$=Free(e) ∧ y fresh in e, σ,σ'

$M,\sigma \models \bullet A$ iff $\exists \sigma', \sigma'', \phi.[\ \sigma = \sigma'.\phi \wedge M,\phi \rightsquigarrow^* \sigma' \wedge M, \sigma'[y \mapsto \sigma(z)] \models A[z \mapsto y]\ ]$

where $\{z\}$=Free(A) ∧ y fresh in A, σ,σ'

$M,\sigma \models \circ A$ iff $\forall \sigma_0.[\ \text{Initial}(\sigma_0) \rightarrow \exists \sigma_1.(\ M,\sigma_0 \rightsquigarrow^* \sigma_1 \wedge M,\sigma_1 \rightsquigarrow^+ \sigma \wedge M,\sigma_1[y \mapsto \sigma(z)] \models A[z \mapsto y]\ ]$

where $\{z\}$=Free(A) ∧ y fresh in A, $\sigma_1$,σ

$M,\sigma \models A@S$ iff … as before …

$M,\sigma \models x.\text{Call}(y,m,z_1,..z_n)$ iff … as before …

# Giving meaning to Assertions
## - the full truth -

$M,\sigma \models$ Access(e,e') iff … as before …

$M,\sigma \models$ Changes(e) iff $M, \sigma \rightsquigarrow \sigma' \wedge \lfloor e \rfloor_{M,\sigma} \neq \lfloor e[z \mapsto y] \rfloor_{M,\sigma'[y \mapsto \sigma(z)]}$

where $\{z\}$=Free(e) ∧ y fresh in e, σ,σ'

$M,\sigma \models \bullet A$ iff $\exists \sigma', \sigma'', \phi.[\ \sigma = \sigma'.\phi \wedge M,\phi \rightsquigarrow^* \sigma' \wedge M, \sigma'[y \mapsto \sigma(z)] \models A[z \mapsto y]\ ]$

where $\{z\}$=Free(A) ∧ y fresh in A, σ,σ'

$M,\sigma \models \circ A$ iff $\forall \sigma_0.[\ \text{Initial}(\sigma_0) \rightarrow \exists \sigma_1.(\ M,\sigma_0 \rightsquigarrow^* \sigma_1 \wedge M,\sigma_1 \rightsquigarrow^+ \sigma \wedge M,\sigma_1[y \mapsto \sigma(z)] \models A[z \mapsto y]\ ]$

where $\{z\}$=Free(A) ∧ y fresh in A, $\sigma_1$,σ

$M,\sigma \models A@S$ iff … as before …

$M,\sigma \models x.\text{Call}(y,m,z_1,..z_n)$ iff … as before …

# Giving meaning to Assertions
## - the full truth -

$M,\sigma \models$ Access(e,e') iff … as before …

$M,\sigma \models$ Changes(e) iff $M, \sigma \rightsquigarrow \sigma' \wedge \lfloor e \rfloor_{M,\sigma} \neq \lfloor e[z \mapsto y] \rfloor_{M,\sigma'[y \mapsto \sigma(z)]}$

where $\{z\}$=Free(e) ∧ y fresh in e, σ,σ'

$M,\sigma \models \bullet A$ iff $\exists \sigma', \sigma'', \phi.[\ \sigma = \sigma'.\phi \wedge M,\phi \rightsquigarrow^* \sigma' \wedge M, \sigma'[y \mapsto \sigma(z)] \models A[z \mapsto y]\ ]$

where $\{z\}$=Free(A) ∧ y fresh in A, σ,σ'

$M,\sigma \models \circ A$ iff $\forall \sigma_0.[\ \text{Initial}(\sigma_0) \rightarrow \exists \sigma_1.(\ M,\sigma_0 \rightsquigarrow^* \sigma_1 \wedge M,\sigma_1 \rightsquigarrow^+ \sigma \wedge M,\sigma_1[y \mapsto \sigma(z)] \models A[z \mapsto y]\ ]$

where $\{z\}$=Free(A) ∧ y fresh in A, $\sigma_1$,σ

$M,\sigma \models A@S$ iff … as before …

$M,\sigma \models x.\text{Call}(y,m,z_1,..z_n)$ iff … as before …

# Giving meaning to Assertions

$M,\sigma \models \text{Access}(x,y)$ iff Initial > $\lfloor y \rfloor_{M,\sigma}$ $\vee$

$\lfloor x.f \rfloor_{M,\sigma} > \lfloor y \rfloor_{M,\sigma} \vee$

$\lfloor \text{this} \rfloor_{M,\sigma} = \lfloor x \rfloor_{M,\sigma} \wedge \lfloor y \rfloor_{M,\sigma} = \lfloor z \rfloor_{M,\sigma} \wedge \ldots$

$M,\sigma \models \text{Changes}(e)$ iff $M, \sigma \rightsquigarrow \sigma' \wedge \lfloor e \rfloor_{M,\sigma} \neq \lfloor e \rfloor_{M,\sigma'}$

$M,\sigma \models \bullet A$ iff $\exists \sigma'.[\, M,\sigma \rightsquigarrow \sigma' \wedge M, \sigma' \models A \,]$

$M,\sigma \models \circ A$ iff $\exists \sigma_0,\sigma_1.[\, \text{Initial}(\sigma_0) \wedge M,\sigma_0 \rightsquigarrow^* \sigma_1 \wedge M,\sigma_1 \rightsquigarrow^* \sigma \wedge$

$M,\sigma_1 \models A \,]$

$M,\sigma \models A@S$ iff $M,\sigma@ls \models A$ where $ls = \lfloor S \rfloor_{M,\sigma}$ and $\sigma@_{ls}$ ...

$M,\sigma \models \text{Call}(x,m,x_1,..x_n)$ iff $\lfloor \text{this} \rfloor_{M,\sigma} = \lfloor x \rfloor_{M,\sigma} \wedge \ldots$

# Giving outstanding definitions rtions

$M,\sigma \models Access(x,y)$ iff Initial > $\lfloor y \rfloor_{M,\sigma}$ ∨

$\lfloor x.f \rfloor_{M,\sigma} > \lfloor y \rfloor_{M,\sigma}$ ∨

$\lfloor this \rfloor_{M,\sigma} = \lfloor x \rfloor_{M,\sigma} \wedge \lfloor y \rfloor_{M,\sigma} = \lfloor z \rfloor_{M,\sigma} \wedge \ldots$

$M,\sigma \models Changes(e)$ iff $M, \sigma \rightsquigarrow \sigma' \wedge \lfloor e \rfloor_{M,\sigma} \neq \lfloor e \rfloor_{M,\sigma'}$

$M,\sigma \models \bullet A$ iff $\exists \sigma'.[\, M,\sigma \rightsquigarrow \sigma' \wedge M, \sigma' \models A \,]$

$M,\sigma \models \circ A$ iff $\exists \sigma_0,\sigma_1.[\, Initial(\sigma_0) \wedge M,\sigma_0 \rightsquigarrow^* \sigma_1 \wedge M,\sigma_1 \rightsquigarrow^* \sigma \wedge M,\sigma_1 \models A \,]$

$M,\sigma \models A@S$ iff $M,\sigma@ls \models A$ where $ls = \lfloor S \rfloor_{M,\sigma}$ and $\sigma@_{ls}$ ...

$M,\sigma \models Call(x,m,x_1,..x_n)$ iff $\lfloor this \rfloor_{M,\sigma} = \lfloor x \rfloor_{M,\sigma} \wedge \ldots$

# Giving outstanding definitions rtions

$M,\sigma \models \text{Access}(x,y)$ iff Initial > $\lfloor y \rfloor_{M,\sigma}$ ∨

$\lfloor x.f \rfloor_{M,\sigma} > \lfloor y \rfloor_{M,\sigma}$ ∨

$\lfloor \text{this} \rfloor_{M,\sigma} = \lfloor x \rfloor_{M,\sigma} \wedge \lfloor y \rfloor_{M,\sigma} = \lfloor z \rfloor_{M,\sigma} \wedge \ldots$

$M,\sigma \models \text{Changes}(e)$ iff $M, \sigma \rightsquigarrow \sigma' \wedge \lfloor e \rfloor_{M,\sigma} \neq \lfloor e \rfloor_{M,\sigma'}$

$M,\sigma \models \bullet A$ iff $\exists \sigma'.[\, M,\sigma \rightsquigarrow \sigma' \wedge M, \sigma' \models A \,]$

$M,\sigma \models \circ A$ iff $\exists \sigma_0,\sigma_1.[\, \text{Initial}(\sigma_0) \wedge M,\sigma_0 \rightsquigarrow^* \sigma_1 \wedge M,\sigma_1 \rightsquigarrow^* \sigma \wedge M,\sigma_1 \models A \,]$

$M,\sigma \models A@S$ iff $M,\sigma@\text{ls} \models A$ where $\text{ls} = \lfloor S \rfloor_{M,\sigma}$ and $\sigma@_{\text{ls}}$ ...

$M,\sigma \models \text{Call}(x,m,x_1,..x_n)$ iff $\lfloor \text{this} \rfloor_{M,\sigma} = \lfloor x \rfloor_{M,\sigma} \wedge \ldots$

# Giving outstanding definitions rtions

$M,\sigma \models \text{Access}(x,y)$ iff Initial > $\lfloor y \rfloor_{M,\sigma}$ ∨

$\lfloor x.f \rfloor_{M,\sigma} > \lfloor y \rfloor_{M,\sigma}$ ∨

$\lfloor \text{this} \rfloor_{M,\sigma} = \lfloor x \rfloor_{M,\sigma} \wedge \lfloor y \rfloor_{M,\sigma} = \lfloor z \rfloor_{M,\sigma} \wedge \ldots$

$M,\sigma \models \text{Changes}(e)$ iff $M, \sigma \rightsquigarrow \sigma' \wedge \lfloor e \rfloor_{M,\sigma} \neq \lfloor e \rfloor_{M,\sigma'}$

$M,\sigma \models \bullet A$ iff $\exists \sigma'.[\, M,\sigma \rightsquigarrow \sigma' \wedge M, \sigma' \models A \,]$

$M,\sigma \models \circ A$ iff $\exists \sigma_0,\sigma_1.[\, \text{Initial}(\sigma_0) \wedge M,\sigma_0 \rightsquigarrow^* \sigma_1 \wedge M,\sigma_1 \rightsquigarrow^* \sigma \wedge M,\sigma_1 \models A \,]$

$M,\sigma \models A@S$ iff $M,\sigma@\text{ls} \models A$ where $\text{ls} = \lfloor S \rfloor_{M,\sigma}$ and $\sigma@_{\text{ls}}$ ...

$M,\sigma \models \text{Call}(x,m,x_1,..x_n)$ iff $\lfloor \text{this} \rfloor_{M,\sigma} = \lfloor x \rfloor_{M,\sigma} \wedge \ldots$

# outstanding definitions: Initial

A runtime configuration is initial iff
1) The heap contains only one object, of class Object
2) The continuation consists of just one frame,
   where this points to that object.

Note: The expression can be arbitrary

Initial(σ) iff σ.heap=(1↦(Object,…) ) ∧ σ.continuations=(this↦1).[ ]

# outstanding definitions: @

# outstanding definitions: @

σ@ls = ( σ.heap@ls, σ.continuations, σ.expression)

# outstanding definitions: @

σ@ls = ( σ.heap@ls, σ.continuations, σ.expression)

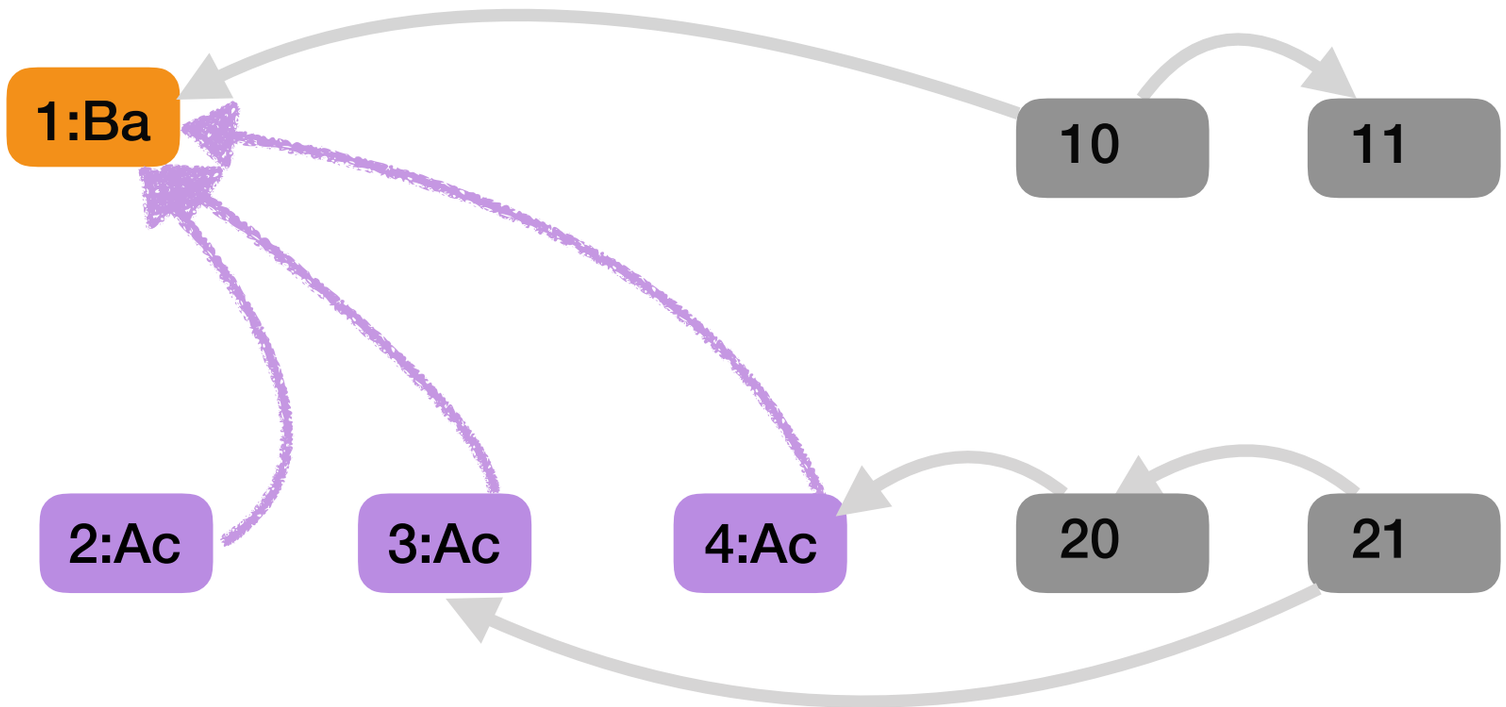
where dom(hp.@IS)=IS  and ↦∀α∈ S. hp@IS(α)=hp(α)

# outstanding definitions: @

σ@ls = ( σ.heap@ls, σ.continuations, σ.expression)

where dom(hp.@IS)=IS  and ↦∀α∈ S. hp@IS(α)=hp(α)

eg, given hp:

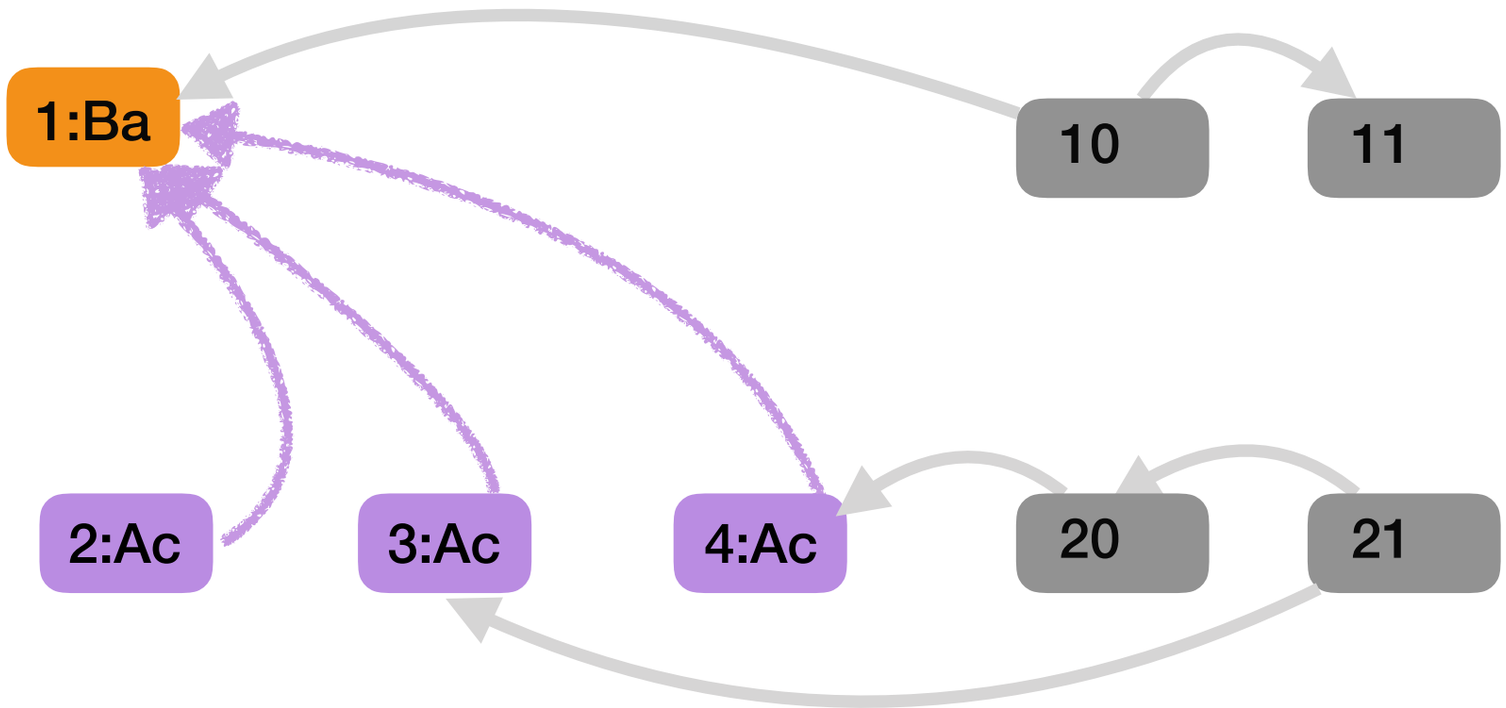


# outstanding definitions: @

σ@ls = ( σ.heap@ls, σ.continuations, σ.expression)

where dom(hp.@IS)=IS  and ↦∀α∈ S. hp@IS(α)=hp(α)

eg, given hp:



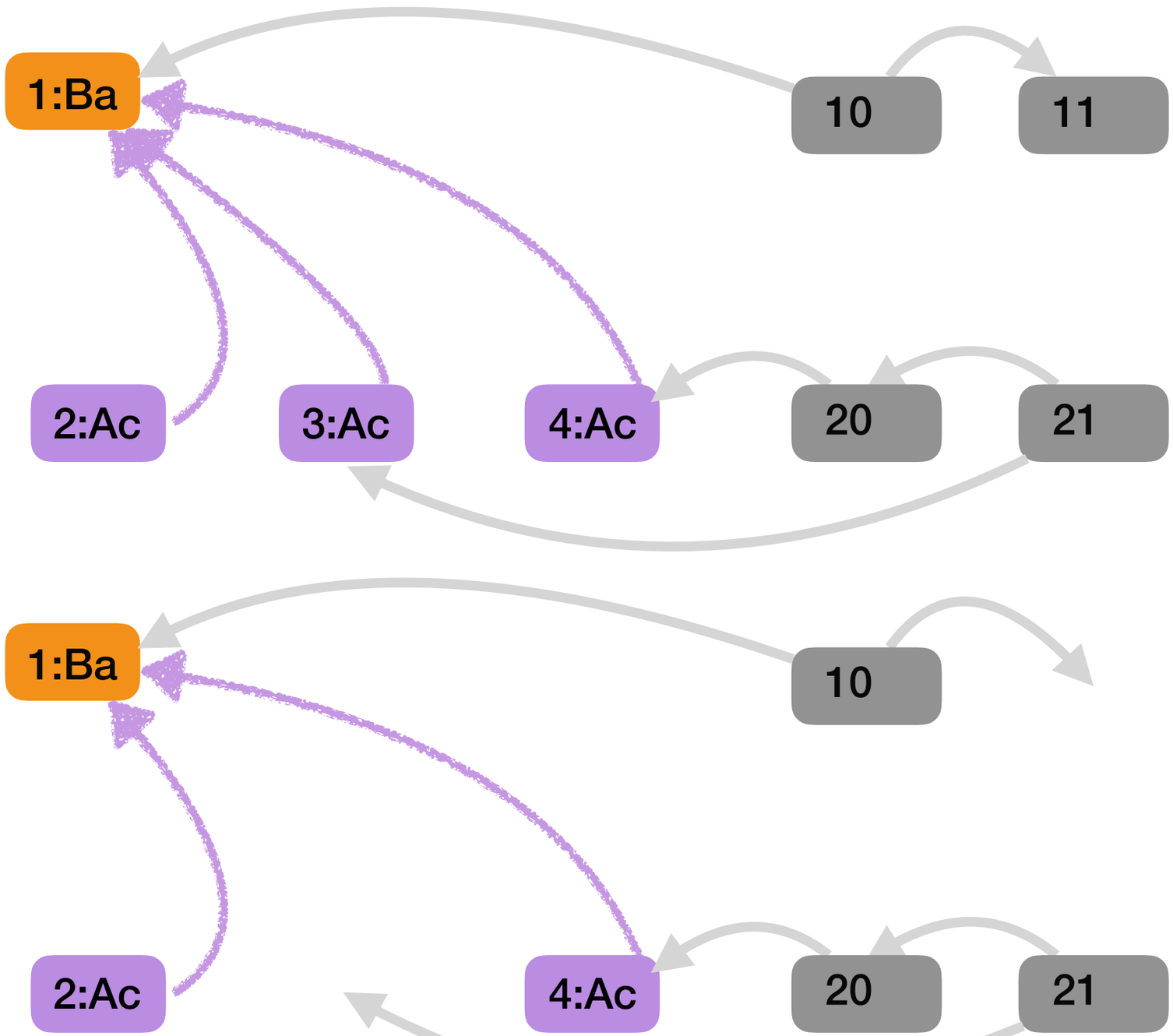

hp@{1,2,4,10,20,21}:

# outstanding definitions: @

σ@ls = ( σ.heap@ls, σ.continuations, σ.expression)

where dom(hp.@IS)=IS  and ↦∀α∈ S. hp@IS(α)=hp(α)

eg, given hp:




hp@{1,2,4,10,20,21}:

# Giving meaning to Assertions

$M \models A$ iff $\forall \sigma \in \text{Arising}(M*M').\ M*M', \sigma \models A$

A module M satisfies an assertion A if all runtime configurations σ which arrive from execution of code from M*M' (for any module M'), satisfy A.

# outstanding definitions: Arising

Arising(M) = { σ | ∃M',$\sigma_0$. [ Initial($\sigma_0$) ∧ M*M' is defined ∧ M'*M,$\sigma_0 \rightsquigarrow^* \sigma$ ] }

# outstanding definitions: Arising

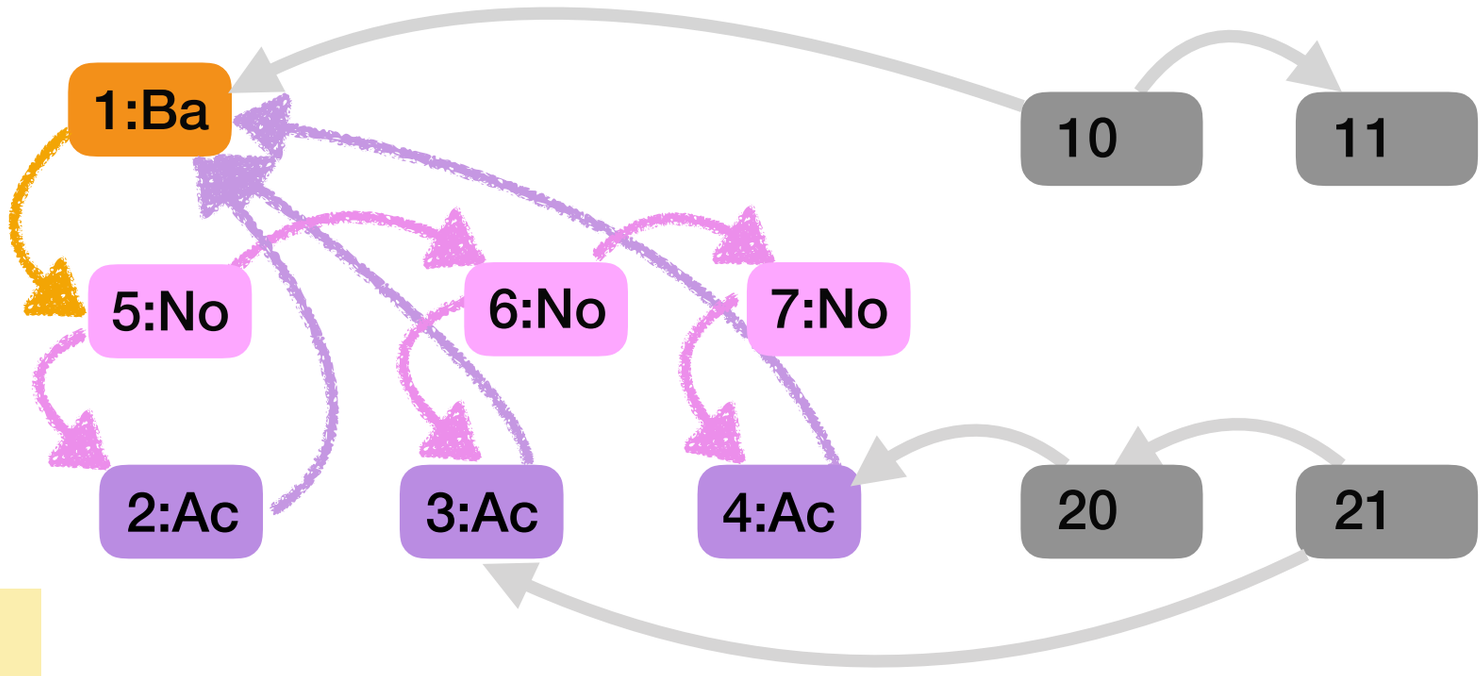
Arising(M) = { σ | ∃M',$\sigma_0$. [ Initial($\sigma_0$) ∧ M*M' is defined ∧ M'*M,$\sigma_0 \rightsquigarrow^* \sigma$ ] }

Open World

# outstanding definitions: Arising

Arising(M) = { σ | ∃M',$\sigma_0$. [ Initial($\sigma_0$) ∧ M*M' is defined ∧ M'*M,$\sigma_0 \rightarrow^* \sigma$ ] }

E.g., Arising(MBA2s).heap **and** Arising(MBA2).heap contain:

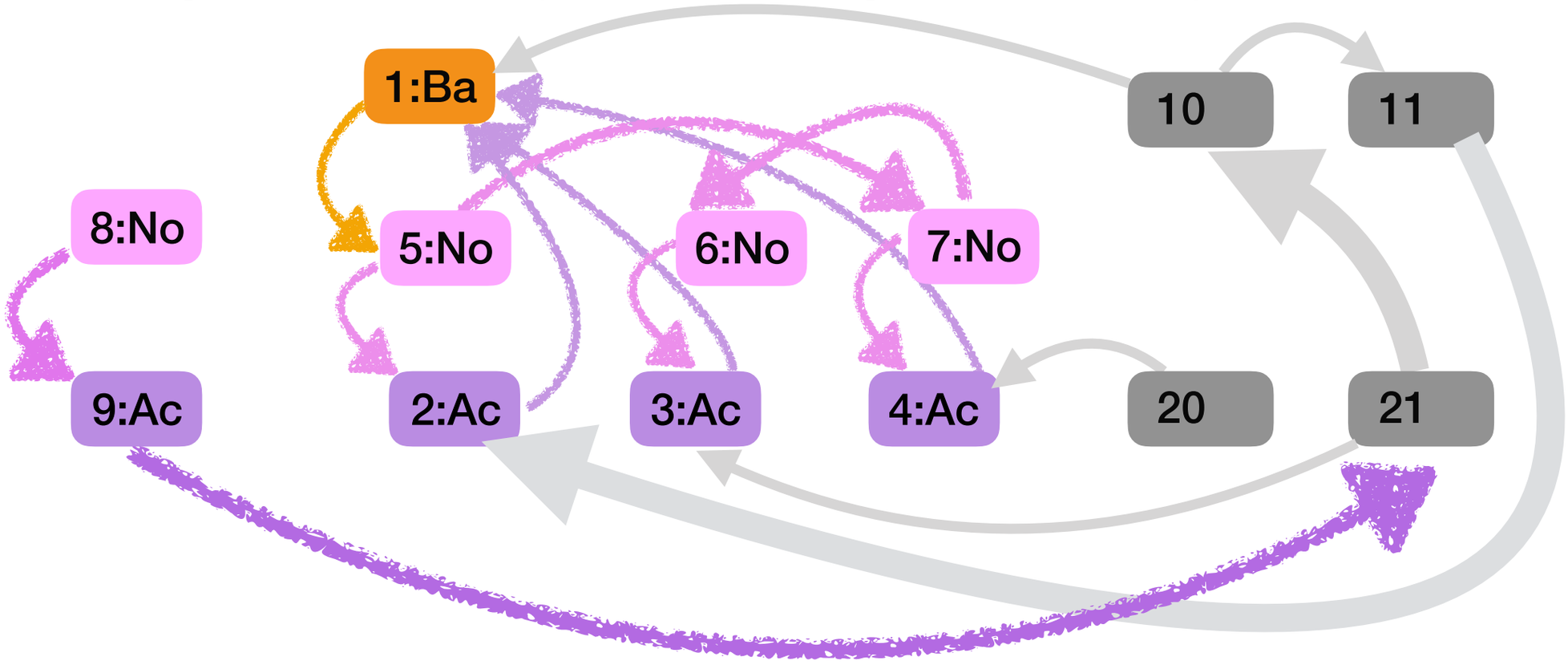



Open World

# outstanding definitions - Arising

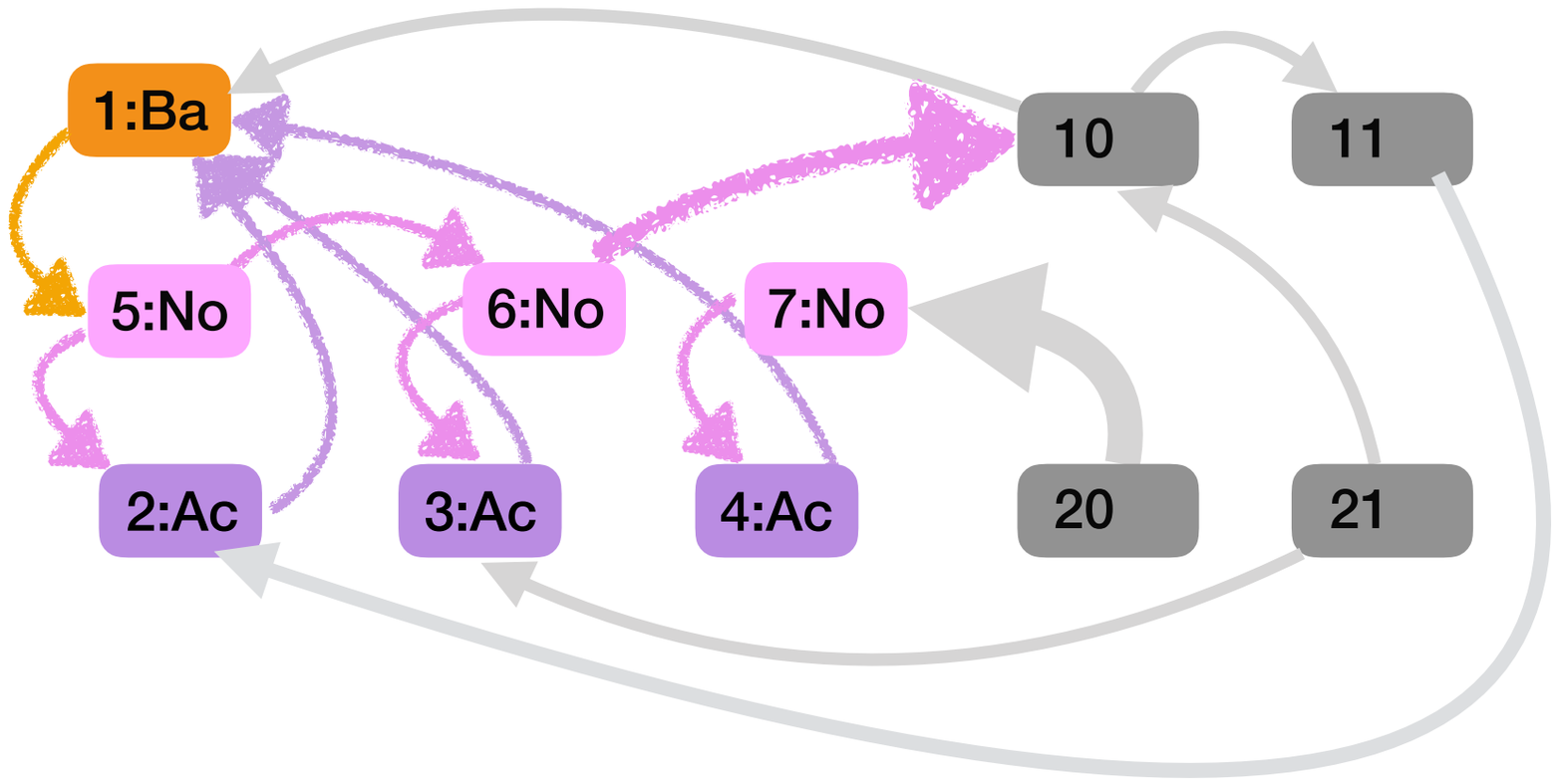
Arising(M) = { σ | ∃M',$\sigma_0$. [ Initial($\sigma_0$) ∧ M*M' is defined ∧ M'*M,$\sigma_0 \rightsquigarrow^* \sigma$ ] }

Also, Arising(MBA2s).heap **and** Arising(MBA2).heap contain:



Arising(M) = { σ | ∃M',$\sigma_0$. [ Initial($\sigma_0$) ∧ M*M' is defined ∧ M'*M,$\sigma_0 \rightsquigarrow^* \sigma$ ] }

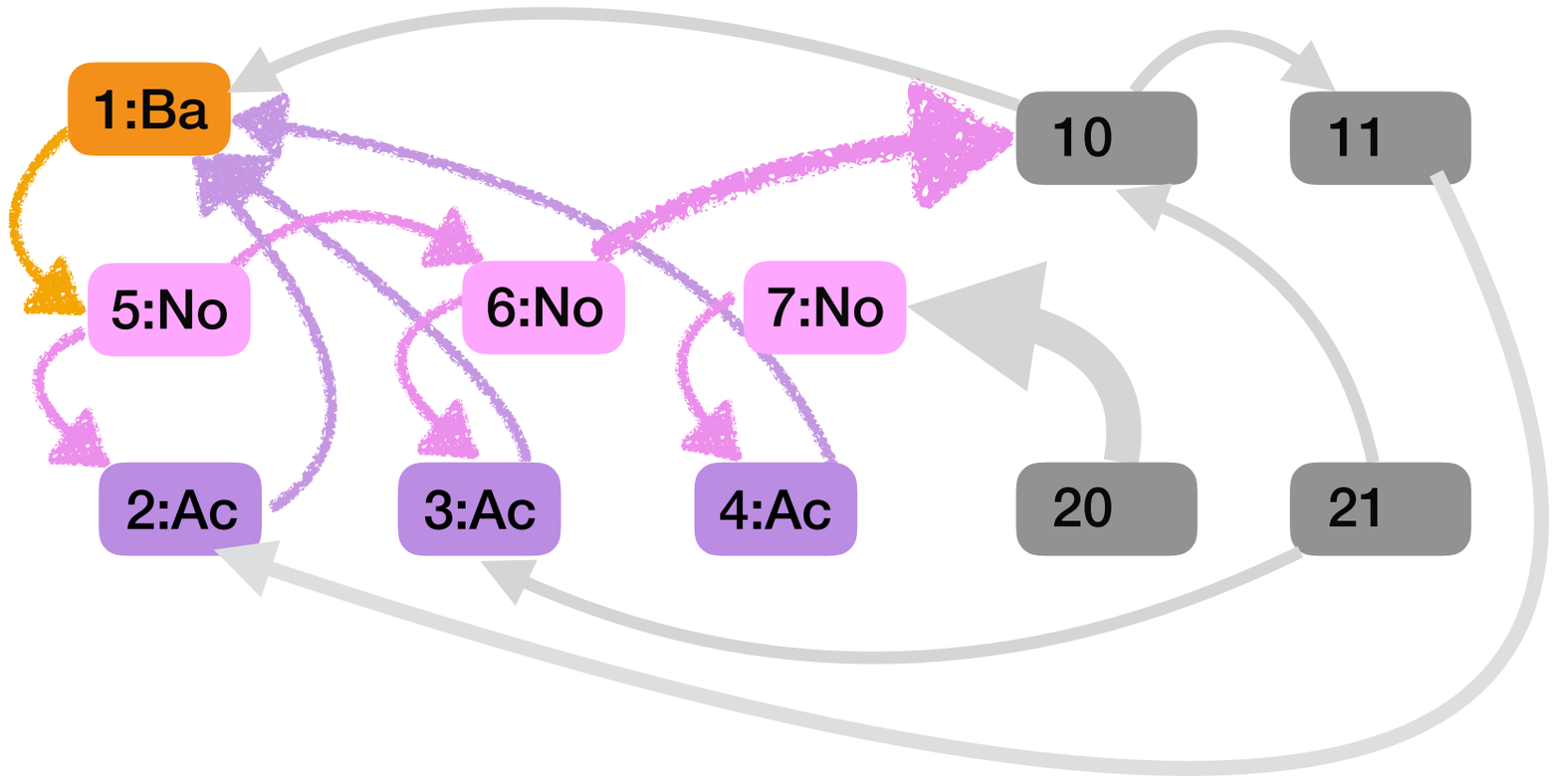
But the following *is* in Arising(MBA2).heap but is *not* in Arising(MBA2s).heap



# outstanding definitions - Arising

Arising(M) = { σ | ∃M',$\sigma_0$. [ Initial($\sigma_0$) ∧ M*M' is defined ∧ M'*M,$\sigma_0 \rightsquigarrow^* \sigma$ ] }

But the following *is* in Arising(MBA2).heap but is *not* in Arising(MBA2s).heap



# Giving meaning to Assertions

$M \models A$ iff $\forall \sigma \in \text{Arising}(M*M').\quad M*M', \sigma \models A$

# Giving meaning to Assertions

$M \models A$ iff $\forall \sigma \in \text{Arising}(M*M').\ M*M', \sigma \models A$

"Lemma"

# Giving meaning to Assertions

$M \models A$ iff $\forall \sigma \in \text{Arising}(M*M').\ M*M', \sigma \models A$

"Lemma"

- $\text{MBA1s} \models \text{Pol\_1}$
- $\text{MBA1s} \models \text{Pol\_2}$
- $\text{MBA1s} \models \text{Pol\_4}$
- $\text{MBA2s} \models \text{Pol\_1}$
- $\text{MBA2s} \models \text{Pol\_2}$
- $\text{MBA2s} \models \text{Pol\_4}$

Proof sketches are "holistic".

Proof sketches use more framing notions,
and require frames are self-framing.

# Entailments

Definitions

- $M \models A \subseteq A'$ iff $\forall \sigma \in \text{Arising}(M).\ [\ M, \sigma \models A \ \rightarrow\ M, \sigma \models A'\ ]$
- $M \models A \sqsubseteq A'$ iff $M \models A \ \rightarrow\ M \models A'$

# Entailments

Definitions

- $M \models A \subseteq A'$ iff $\forall \sigma \in \text{Arising}(M).\ [\ M, \sigma \models A \rightarrow M, \sigma \models A'\ ]$
- $M \models A \sqsubseteq A'$ iff $M \models A \rightarrow M \models A'$

Facts

- $M \models A \subseteq A'$ implies $M \models A \sqsubseteq A'$
- $M \models A \sqsubseteq A'$ does not imply $M \models A \subseteq A'$
- $M \models (\bullet A \rightarrow A') \sqsubseteq (A' \rightarrow \circ A)$

# Entailments

Definitions

- $M \models A \subseteq A'$ iff $\forall \sigma \in \text{Arising}(M).\ [\ M, \sigma \models A \rightarrow M, \sigma \models A'\ ]$
- $M \models A \sqsubseteq A'$ iff $M \models A \rightarrow M \models A'$

Facts

- $M \models A \subseteq A'$ implies $M \models A \sqsubseteq A'$
- $M \models A \sqsubseteq A'$ does not imply $M \models A \subseteq A'$
- $M \models (\bullet A \rightarrow A') \sqsubseteq (A' \rightarrow \circ A)$

- $M \models A @ S$ and $M \models S \subseteq S'$ imply $M \models A @ S'$?
- $M, \sigma \models (\bullet A)@S$ imply $M \models \bullet(A@S)$?

# Entailments

Definitions

- $M \models A \subseteq A'$ iff $\forall \sigma \in \text{Arising}(M).\ [\ M, \sigma \models A \rightarrow M, \sigma \models A'\ ]$
- $M \models A \sqsubseteq A'$ iff $M \models A \rightarrow M \models A'$

Facts

- $M \models A \subseteq A'$ implies $M \models A \sqsubseteq A'$
- $M \models A \sqsubseteq A'$ does not imply $M \models A \subseteq A'$
- $M \models (\bullet A \rightarrow A') \sqsubseteq (A' \rightarrow \circ A)$

# Entailments

Definitions

- $M \models A \subseteq A'$ iff $\forall \sigma \in Arising(M).\ [\ M, \sigma \models A \rightarrow M, \sigma \models A'\ ]$
- $M \models A \sqsubseteq A'$ iff $M \models A \rightarrow M \models A'$

Facts

- $M \models A \subseteq A'$ implies $M \models A \sqsubseteq A'$
- $M \models A \sqsubseteq A'$ does not imply $M \models A \subseteq A'$
- $M \models (\bullet A \rightarrow A') \sqsubseteq (A' \rightarrow \circ A)$

- $M \models A @ S$ and $M \models S \subseteq S'$ does *not* imply $M \models A @ S'$
- We call $A$ *monotonic*, if $M,\sigma \models A @ S$ and $M,\sigma \models S \subseteq S'$ imply $M,\sigma \models A @ S'$

- If $A$ monotonic, then
  $M,\sigma \models (\bullet A)@S$ and $M,\sigma \models S'=Allocated$ imply $M,\sigma \models \bullet (A @ (S \cup (Allocated \setminus S')))$

# Example2: DAO - simplified

DAO, a “hub that disperses funds”; (https://www.ethereum.org/dao).
In a simplified form it allows clients to contribute and retrieve their funds (by calling `payIn(…)` and `repay()`).

# Example2: DAO - simplified

DAO, a “hub that disperses funds”; (https://www.ethereum.org/dao).
In a simplified form it allows clients to contribute and retrieve their funds (by calling `payIn(…)` and `repay()`).

Pol_DAO_withdraw

≙

**∀** cl:External. **∀**d:DAO. **∀**.n’:Nat.

[ cl.Calls(`d.repay()` ) ∧ ○ ( cl.Calls(`d.payIn(n)` )

∧ ¬( ○ cl.Calls(`d.repay()` )

→

d.ether≧`n` ∧ • (d.Calls(cl.send(`n` )) ]

# Example2: DAO - simplified

DAO, a “hub that disperses funds”; (https://www.ethereum.org/dao).
In a simplified form it allows clients to contribute and retrieve their funds (by calling `payIn(…)` and `repay()`).

Pol_DAO_withdraw

≙

**∀** cl:External. **∀**d:DAO. **∀**.n’:Nat.
[ cl.Calls(`d.repay()` ) ∧ ○ ( cl.Calls(`d.payIn(n)` )
∧ ¬( ○ cl.Calls(`d.repay()` )

→

d.ether≧`n` ∧ • (`d`.Calls(cl.send(`n` )) ]

*This says*: If a client cl asks to be repaid (cl.Calls(`d.repay()`) and in the past they had contributed (○ ( cl.Calls(`d.payIn(n)`)) and not withdrawn their contribution (¬( ○ cl.Calls(`d.repay()` )),
then the DAO will have enough funds (d.ether≧`n`) and will send the money to client (• `d`.Calls(cl.send(`n` ))).

# Example2: DAO continued

**Vulnerability**: Through repeated calls of a buggy version of repay(), a client could deplete all funds of the DAO and thus the DAO could not repay its other clients.

# Example2: DAO continued

**Vulnerability**: Through repeated calls of a buggy version of repay(), a client could deplete all funds of the DAO and thus the DAO could not repay its other clients.

$$
\begin{aligned}
&\text{Pol\_DAO\_withdraw} \\
&\quad \triangleq \\
&\quad \forall\, \text{cl:External}.\ \forall \text{d:DAO}.\ \forall .\text{n:Nat}. \\
&\quad [\ \text{cl.Calls}(\texttt{d.repay()})\ \ \wedge \circ(\ \text{cl.Calls}(\texttt{d.payIn(n)}) \\
&\qquad\qquad \wedge \neg(\ \circ\ \text{cl.Calls}(\texttt{d.repay()})\ ) \\
&\quad \rightarrow \\
&\quad \text{d.ether} \geqq \texttt{n} \wedge\ \bullet\ (\texttt{d.Calls}(\text{cl.send}(\texttt{n}))\ ]
\end{aligned}
$$

# Example2: DAO continued

**Vulnerability**: Through repeated calls of a buggy version of repay(), a client could deplete all funds of the DAO and thus the DAO could not repay its other clients.

$$
\begin{aligned}
&\text{Pol\_DAO\_withdraw}\\
&\quad\triangleq\\
&\quad\forall\ \text{cl:External}.\ \forall\text{d:DAO}.\ \forall.\text{n:Nat}.\\
&\quad[\ \text{cl.Calls}(\texttt{d.repay()})\ \ \wedge \circ(\ \text{cl.Calls}(\texttt{d.payIn(n)})\\
&\qquad\qquad\wedge \neg(\ \circ\ \text{cl.Calls}(\texttt{d.repay()})\\
&\quad\rightarrow\\
&\quad\text{d.ether}\geqq\texttt{n}\ \wedge\ \bullet(\texttt{d.Calls(cl.send(n))}\ ]
\end{aligned}
$$

This specification avoids the vulnerability:
A contract which satisfies Pol_DAO_withdraw will always be able to repay all its customers.

# Example2: DAO continued

**Vulnerability**: Through repeated calls of a buggy version of repay(), a client could deplete all funds of the DAO and thus the DAO could not repay its other clients.

$$
\begin{aligned}
&\text{Pol\_DAO\_withdraw} \\
&\quad \triangleq \\
&\quad \forall\, \text{cl:External}.\ \forall \text{d:DAO}.\ \forall .\text{n:Nat}. \\
&\quad [\ \text{cl.Calls}(\texttt{d.repay()})\ \ \wedge \bigcirc(\ \text{cl.Calls}(\texttt{d.payIn(n)}) \\
&\qquad\qquad \wedge \neg(\ \bigcirc\ \text{cl.Calls}(\texttt{d.repay()})) \\
&\quad \rightarrow \\
&\quad \text{d.ether} \geqq \texttt{n}\ \wedge\ \bullet\ (\texttt{d.Calls(cl.send(n))}\ ]
\end{aligned}
$$

This specification avoids the vulnerability:
A contract which satisfies Pol_DAO_withdraw will always be able to repay all its customers.

# Example2: a possible classical spec

Assume that the DAO keeps a directory of contributions, and require:
R1: that the directory is compatible with the amount of ether
kept in the DAO, and
R2: that withdraw reduces the ether but that amount.

# Example2: a possible classical spec

Assume that the DAO keeps a directory of contributions, and require:
R1: that the directory is compatible with the amount of ether
kept in the DAO, and
R2: that withdraw reduces the ether but that amount.

R1: ∀d:DAO. d.ether = ∑ `cl` such that `d.directory(cl)` defined `d.directory(cl)`

# Example2: a possible classical spec

Assume that the DAO keeps a directory of contributions, and require:
R1: that the directory is compatible with the amount of ether
kept in the DAO, and
R2: that withdraw reduces the ether but that amount.

R1: ∀d:DAO. d.ether = ∑ `cl` such that `d.directory(cl)` defined `d.directory(cl)`

R2: cl:External ∧ d:DAO ∧ n:Nat ∧ d.directory(cl)=n
{ `d.repay()` ∧ `caller`=cl }
d.directory(cl)=0 ∧ d.Calls(cl.send(`n`))

# Example2: a possible classical spec

Assume that the DAO keeps a directory of contributions, and require:
R1: that the directory is compatible with the amount of ether
kept in the DAO, and
R2: that withdraw reduces the ether but that amount.

R1: ∀d:DAO. d.ether = ∑ `cl` such that `d.directory(cl)` defined `d.directory(cl)`

R2: cl:External ∧ d:DAO ∧ n:Nat ∧ d.directory(cl)=n
{ `d.repay()` ∧ `caller`=cl }
d.directory(cl)=0 ∧ d.Calls(cl.send(`n`))

*R2 says*: If client cl has m tokens (d.directory(cl)=n) and asks to be repaid (cl calls `d.repay()`) then all his tokens will be sent (d.Calls(cl.send(`n`))) and no tokens will be left (d.directory(cl)=0).
Together with R2, this spec avoids the vulnerability, *provided* the attack goes through the function repay.

# Example2: classical spec vs holistic spec

Assume that the DAO keeps a directory of contributions, and require:
R1: that the directory is compatible with the amount of ether
kept in the DAO, and
R2: that withdraw reduces the ether but that amount.

R1: ∀d:DAO. d.ether = $\sum_{\texttt{cl such that d.directory(cl) defined}}$ `d.directory(cl)`

R2:

$$\texttt{cl:External} \wedge \texttt{d:DAO} \wedge \texttt{n:Nat} \wedge \texttt{d.directory(cl)=n}$$
$$\{\texttt{d.repay()} \wedge \texttt{caller=cl}\}$$
$$\texttt{d.directory(cl)=0} \wedge \texttt{d.Calls(cl.send(n))}$$

This classical specification is insufficient to avoid the vulnerability *in general*, as it does not prevent *other* functions from affecting the contents of d.directory.

To avoid the vulnerability in general, we would need to either manually inspect the specification of all the functions in the DAO, or add another holistic spec, promising, eg that only calls by cl can affect the contents of d.directory(cl)

# Example3: ERC20 - simplified

a popular standard for initial coin offerings. (https://theethereum.wiki/w/index.php/ERC20_Token_Standard); allows clients to buy and transfer tokens, and to designate other clients to transfer on their behalf.

# Example3: ERC20 - simplified

a popular standard for initial coin offerings. (https://theethereum.wiki/w/index.php/ERC20_Token_Standard); allows clients to buy and transfer tokens, and to designate other clients to transfer on their behalf.

$$
\begin{aligned}
&\mathrm{PoI\_ERC20\_withdraw} \\
&\quad \triangleq \\
&\forall\, e{:}\mathrm{ERC20}.\ \forall\, cl{:}\ \mathrm{Client}. \\
&[\ \ e.balance(cl) < e.balance(cl)_{pre}\ \ ) \\
&\quad \rightarrow \\
&[\ \circ\ (\ \exists cl'{:}\ \mathrm{Client}.\ \exists m{:}\ \mathrm{Nat}. \\
&\qquad [\ \ cl.\mathrm{Calls}(\texttt{e.transfer}(cl',m)\ )\ \ ) \\
&\qquad\quad \vee \\
&\qquad \exists cl''{:}\ \mathrm{Client}. \\
&\qquad\ \ \mathrm{Authorized}(c,cl'')\ \wedge\ c''.\mathrm{Calls}(\texttt{e.transferFrom}(c,cl',m)\ )\ ]
\end{aligned}
$$

# Example3: ERC20 - simplified

a popular standard for initial coin offerings. (https://theethereum.wiki/w/index.php/ERC20_Token_Standard); allows clients to buy and transfer tokens, and to designate other clients to transfer on their behalf.

$$
\begin{aligned}
&\mathrm{PoI\_ERC20\_withdraw} \\
&\quad \triangleq \\
&\forall\, e{:}\mathrm{ERC20}.\ \forall\, cl{:}\ \mathrm{Client}. \\
&[\ \ e.balance(cl) < e.balance(cl)_{pre}\ \ ) \\
&\quad \rightarrow \\
&[\ \circ\ (\ \exists cl'{:}\ \mathrm{Client}.\ \exists m{:}\ \mathrm{Nat}. \\
&\qquad [\ \ cl.\mathrm{Calls}(\texttt{e.transfer}(cl',m)\ )\ \ ) \\
&\qquad\quad \vee \\
&\qquad \exists cl''{:}\ \mathrm{Client}. \\
&\qquad\ \ \mathrm{Authorized}(c,cl'')\ \wedge\ c''.\mathrm{Calls}(\texttt{e.transferFrom}(c,cl',m)\ )\ ]
\end{aligned}
$$

*This says*: A client's balance decreases only if that client, or somebody authorised by that client, made a payment.

# Example3: ERC20 - Authorized

# Example3: ERC20 - Authorized

Authorized(c,c') ≜ ∃m: Nat. ○ ( c.Calls(`e.approve`(c',m) ) )

# Example3: ERC20 - Authorized

Authorized(c,c') ≜ ∃m: Nat. ○ ( c.Calls(`e.approve`(c',m) ) )

*This says*: A client cl' is authorised by another client cl, iff at some time in the past the latter informed the tokenholder that it authorised the former.

# Example3: ERC20 - classical spec

# Example3: ERC20 - classical spec

e:ERC20 ∧ e.balance(cl) >m ∧ e.balance(cl') = m' ∧ cl≠cl'

{ `e.transfer(`cl',m) ∧ Caller=cl}

$e.balance(cl) = e.balance(cl)_{pre} - m \wedge e.balance(cl')_{pre} = m' + m$

e:ERC20 ∧ e.balance(cl) >m ∧ e.balance(cl') = m' ∧ cl≠cl'

∧ Authorized `(e,`cl`,`cl'')

{ `e.transferFrom(`cl',m) ∧ Caller=cl''}

$e.balance(cl) = e.balance(cl)_{pre} - m \wedge e.balance(cl')_{pre} = m' + m$

e:ERC20 ∧ e.balance(cl) >m ∧ e.balance(cl') = m'

{ `e.allow(`cl') ∧ Caller=cl }

Authorized `(e,`cl`,`cl'')

# Example3: ERC20 - classical vs holistic

e:ERC20 ∧ e.balance(cl) >m ∧ e.balance(cl') = m' ∧ cl≠cl'

{ e.transfer(cl',m) ∧ Caller=cl}

e.balance(cl) = $e.balance(cl)_{pre}$ -m ∧ $e.balance(cl')_{pre}$ = m'+m

e:ERC20 ∧ e.balance(cl) >m ∧ e.balance(cl') = m' ∧ cl≠cl'

∧ Authorized(e,cl,cl'')

{ e.transferFrom(cl',m) ∧ Caller=cl''}

e.balance(cl) = $e.balance(cl)_{pre}$ -m ∧ $e.balance(cl')_{pre}$ = m'+m

e:ERC20 ∧ e.balance(cl) >m ∧ e.balance(cl') = m'

{ e.allow(cl') ∧ Caller=cl }

Authorized(e,cl,cl'')

# Example3: ERC20 - classical vs holistic

e:ERC20 ∧ e.balance(cl) >m ∧ e.balance(cl') = m' ∧ cl≠cl'

{ e.transfer(cl',m) ∧ Caller=cl}

e.balance(cl) = e.balance(cl)$_{pre}$ -m ∧ e.balance(cl')$_{pre}$ = m'+m

e:ERC20 ∧ e.balance(cl) >m ∧ e.balance(cl') = m' ∧ cl≠cl'
∧ Authorized(e,cl,cl'')

{ e.transferFrom(cl',m) ∧ Caller=cl''}

e.balance(cl) = e.balance(cl)$_{pre}$ -m ∧ e.balance(cl')$_{pre}$ = m'+m

e:ERC20 ∧ e.balance(cl) >m ∧ e.balance(cl') = m'

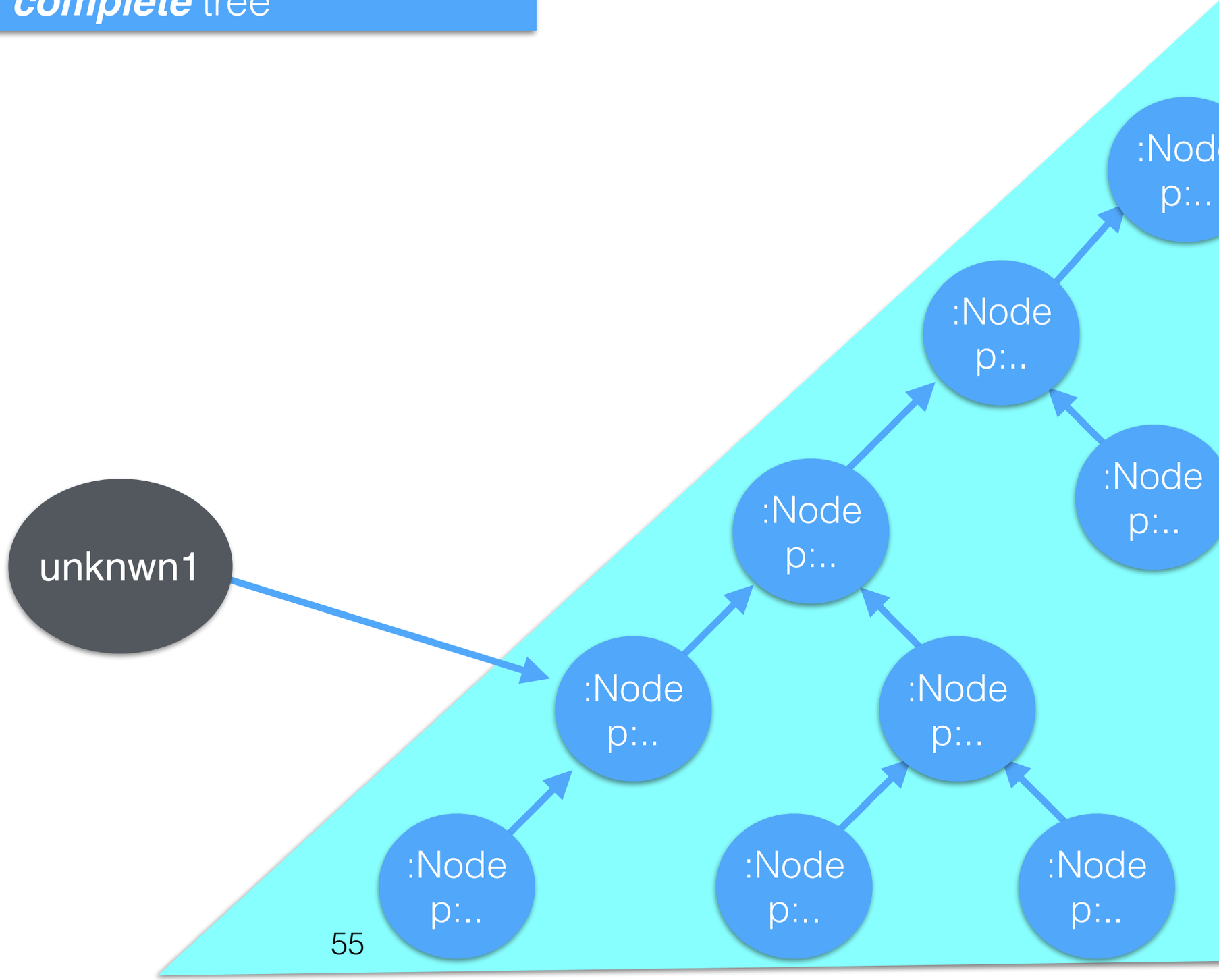
{ e.allow(cl') ∧ Caller=cl }

Authorized(e,cl,cl'')

The above does not determine whether there are other means to transfer tokens, or to authorise clients. For this we would need to inspect the classic specs of all the functions, or add holistic aspects

# Example4: DOM attenuation

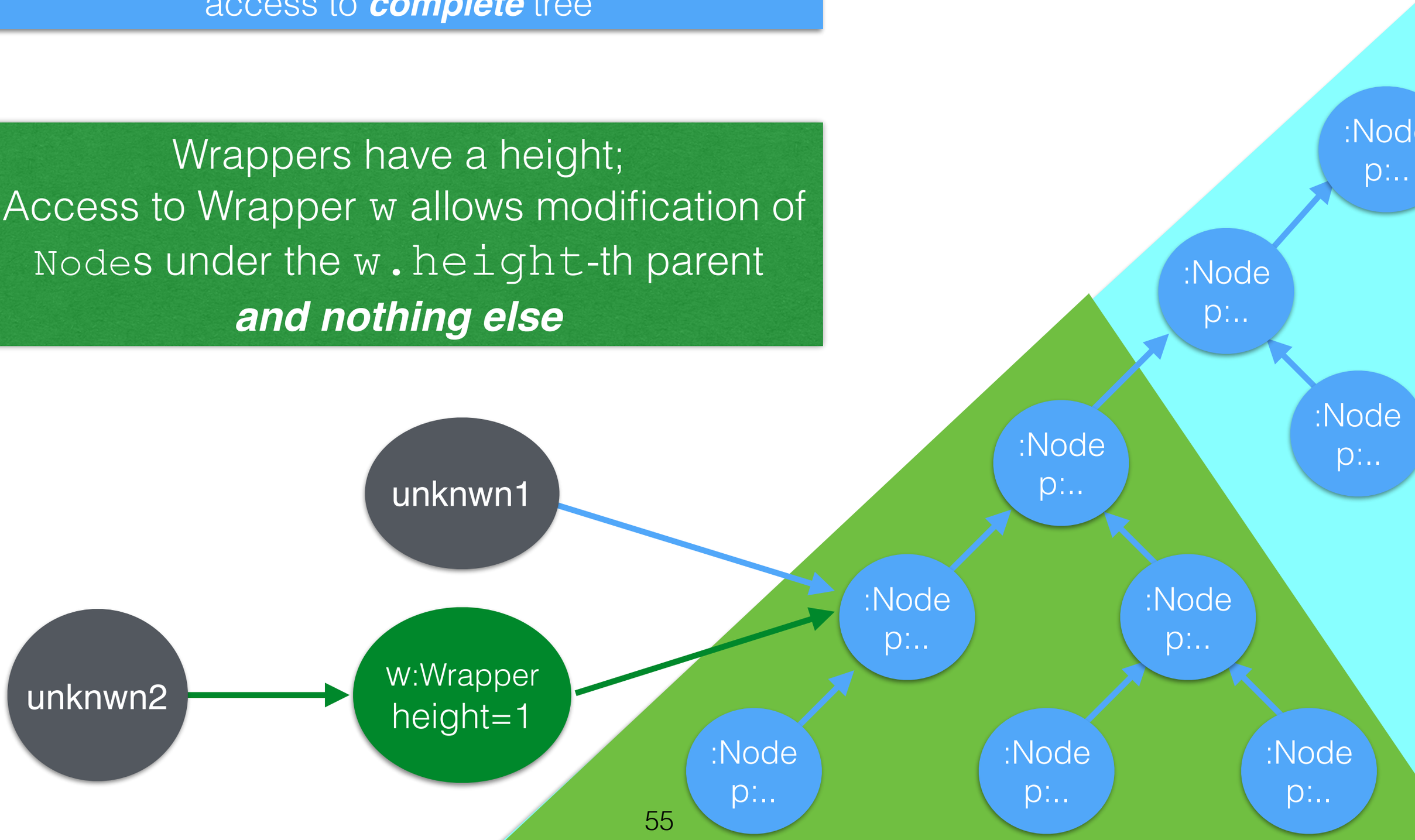
Access to any `Node` gives
access to ***complete*** tree



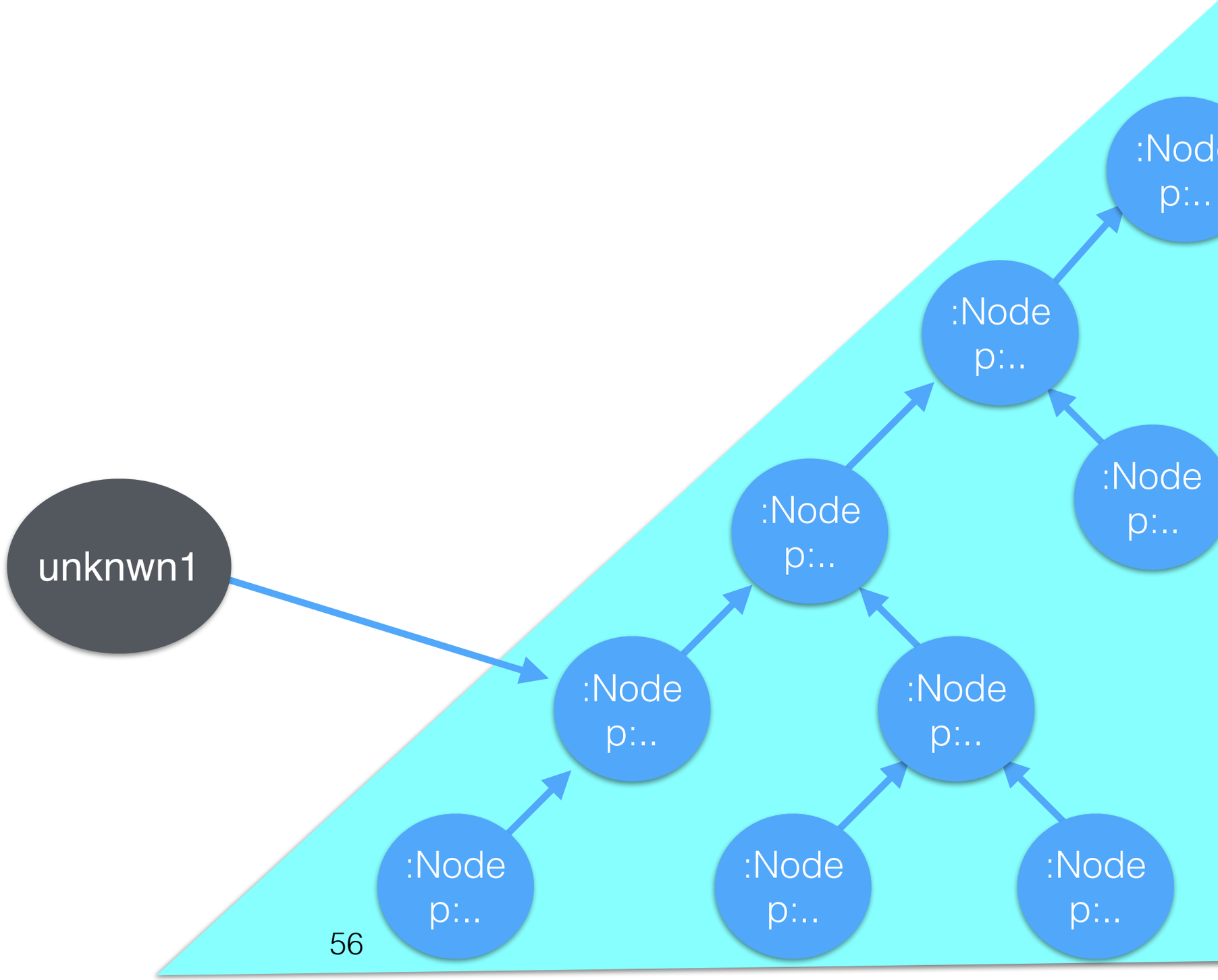
# Example4: DOM attenuation

Access to any `Node` gives
access to ***complete*** tree

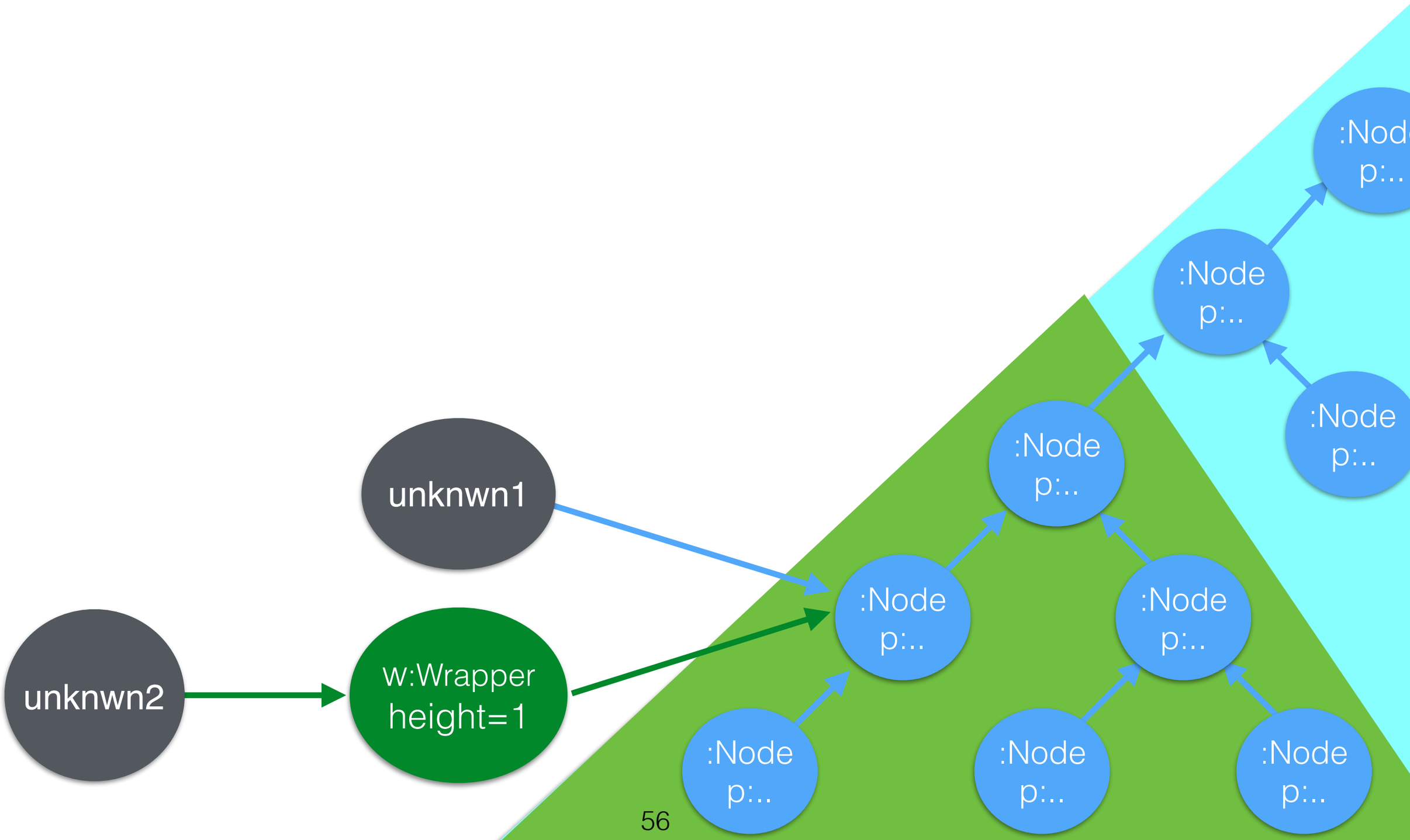
Wrappers have a height;
Access to Wrapper `w` allows modification of `Node`s under the `w.height`-th parent
***and nothing else***



# Example4: DOM attenuation - 2



# Example4: DOM attenuation - 2



# Example4: DOM attenuation - 2

Pol_W ≙

**∀**S:Set. **∀**nd:Node.[

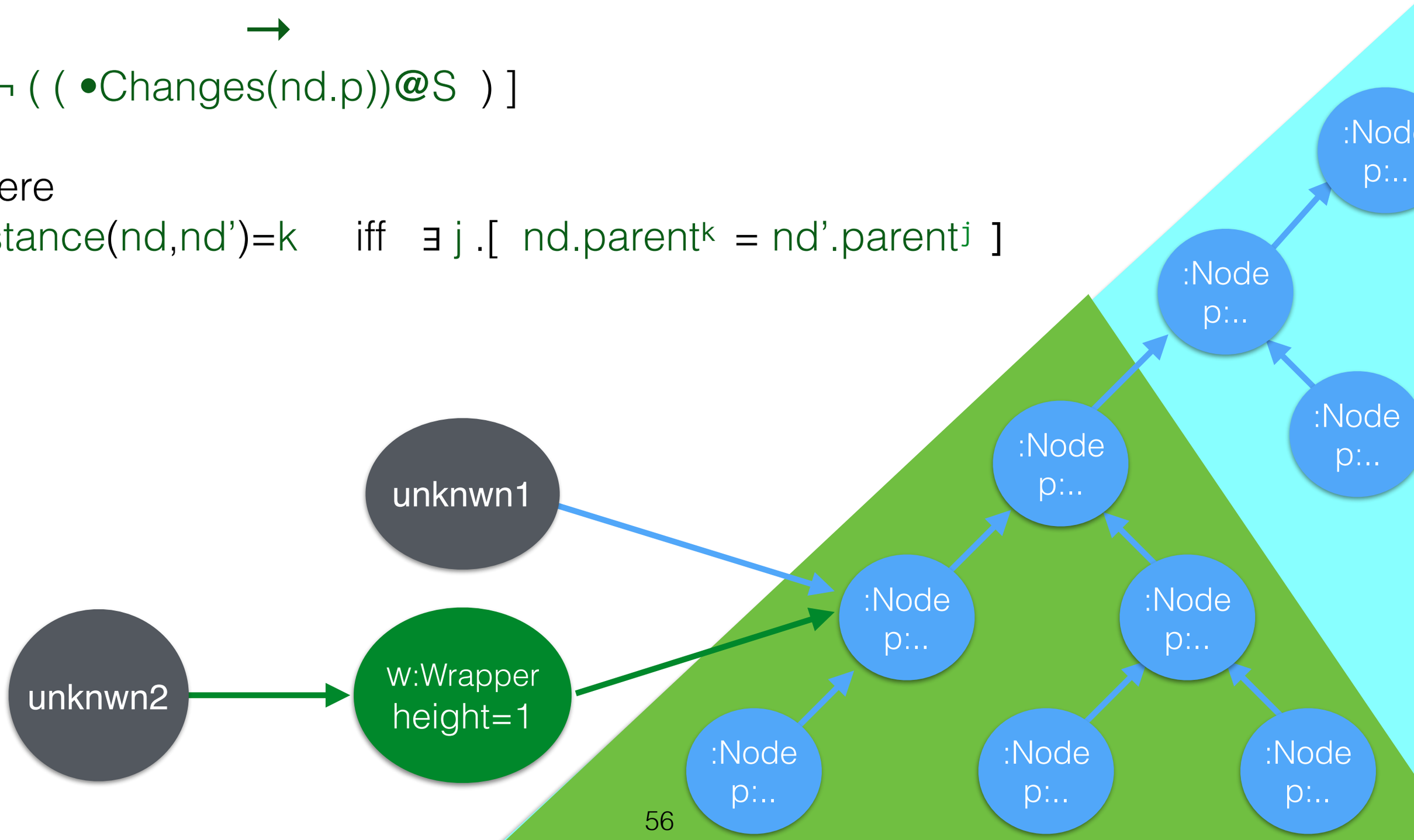
[ [ Access(s,nd) → s:Node ∨ s:Wrapper ∧ Distance(s.node,nd)>s.height ]

→

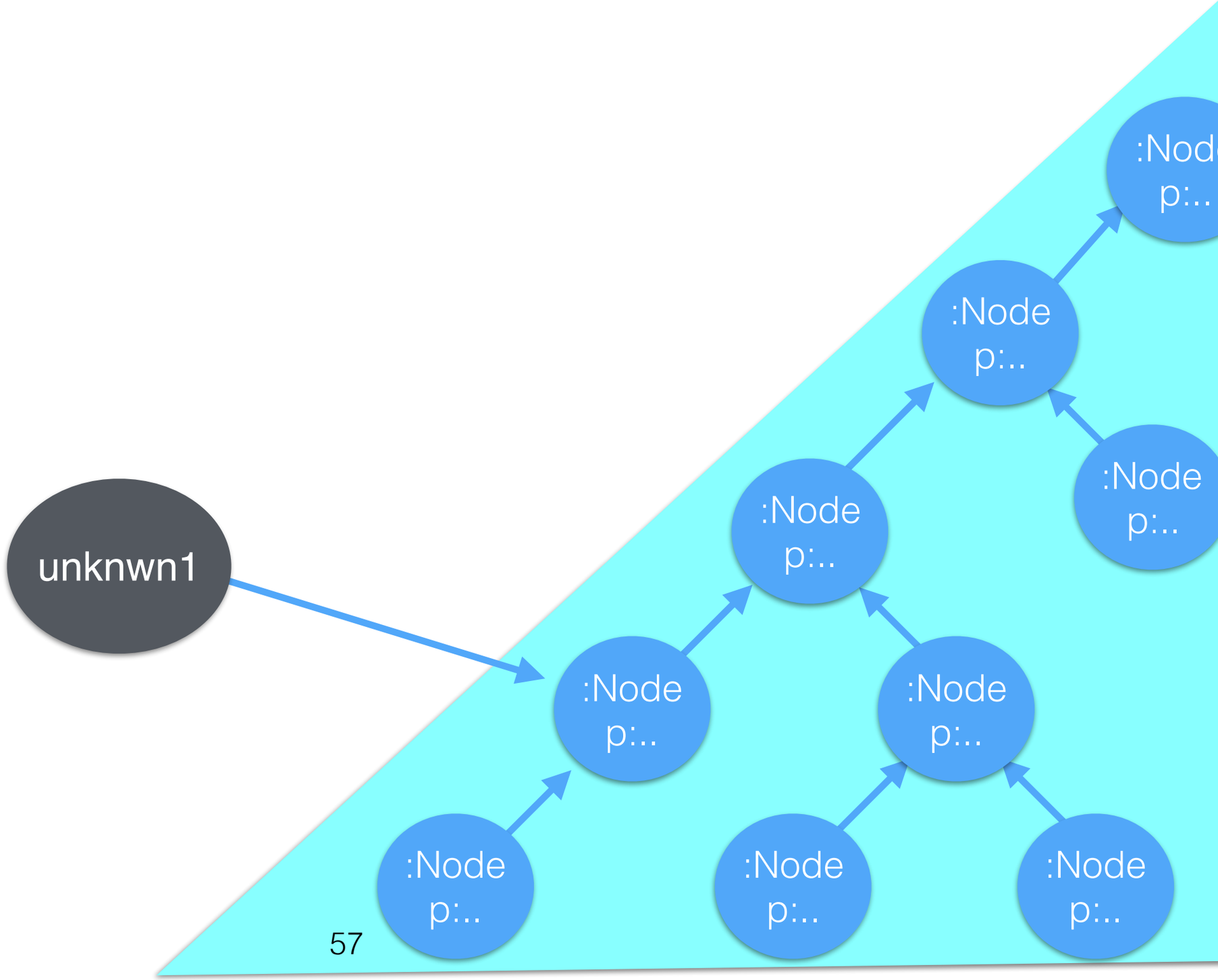
¬ ( ( •Changes(nd.p))**@**S ) ]

where

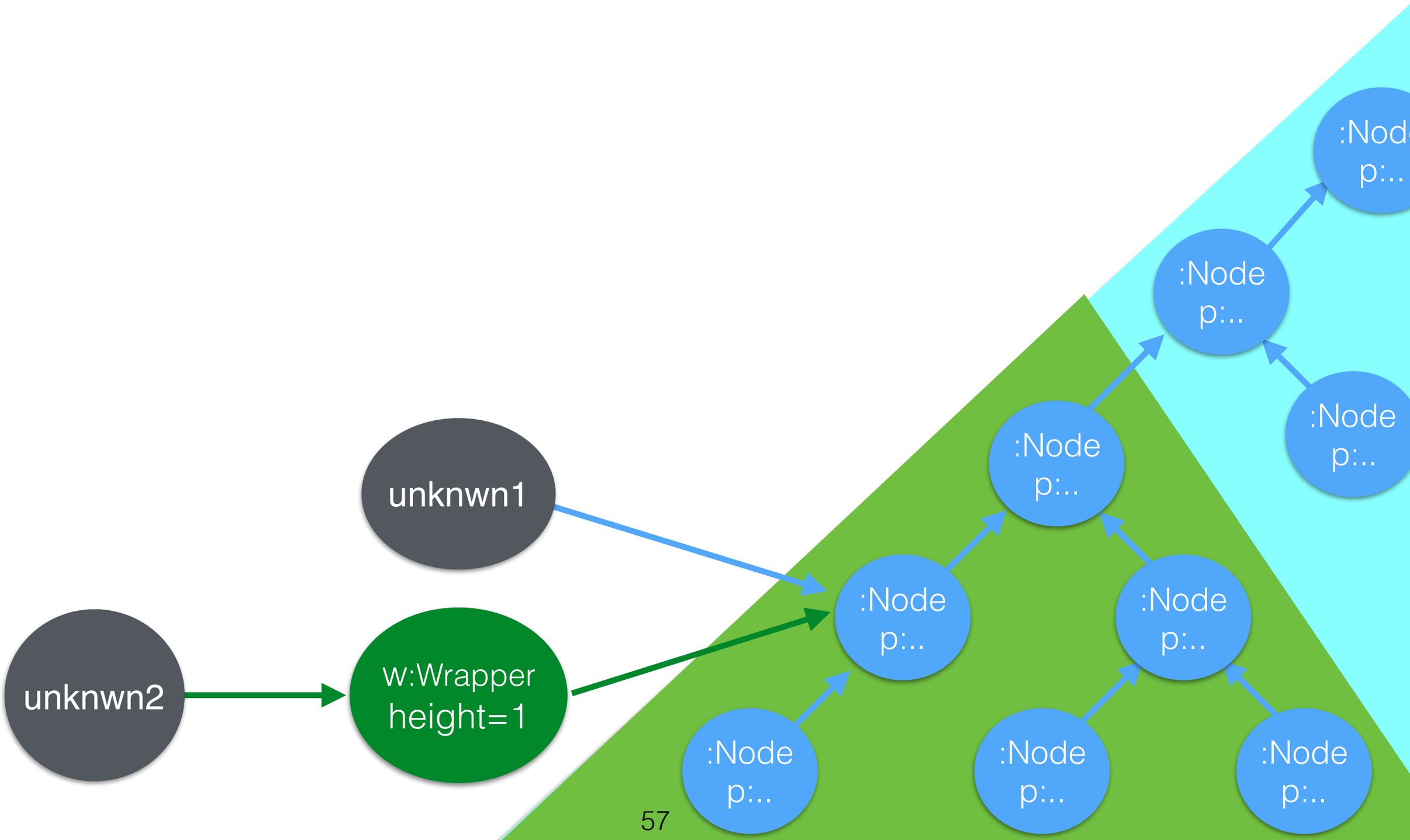
Distance(nd,nd')=k iff ∃ j .[ nd.parent$^{k}$ = nd'.parent$^{j}$ ]



# Example4: DOM attenuation - 3



# Example4: DOM attenuation - 3



# Example4: DOM attenuation - 3

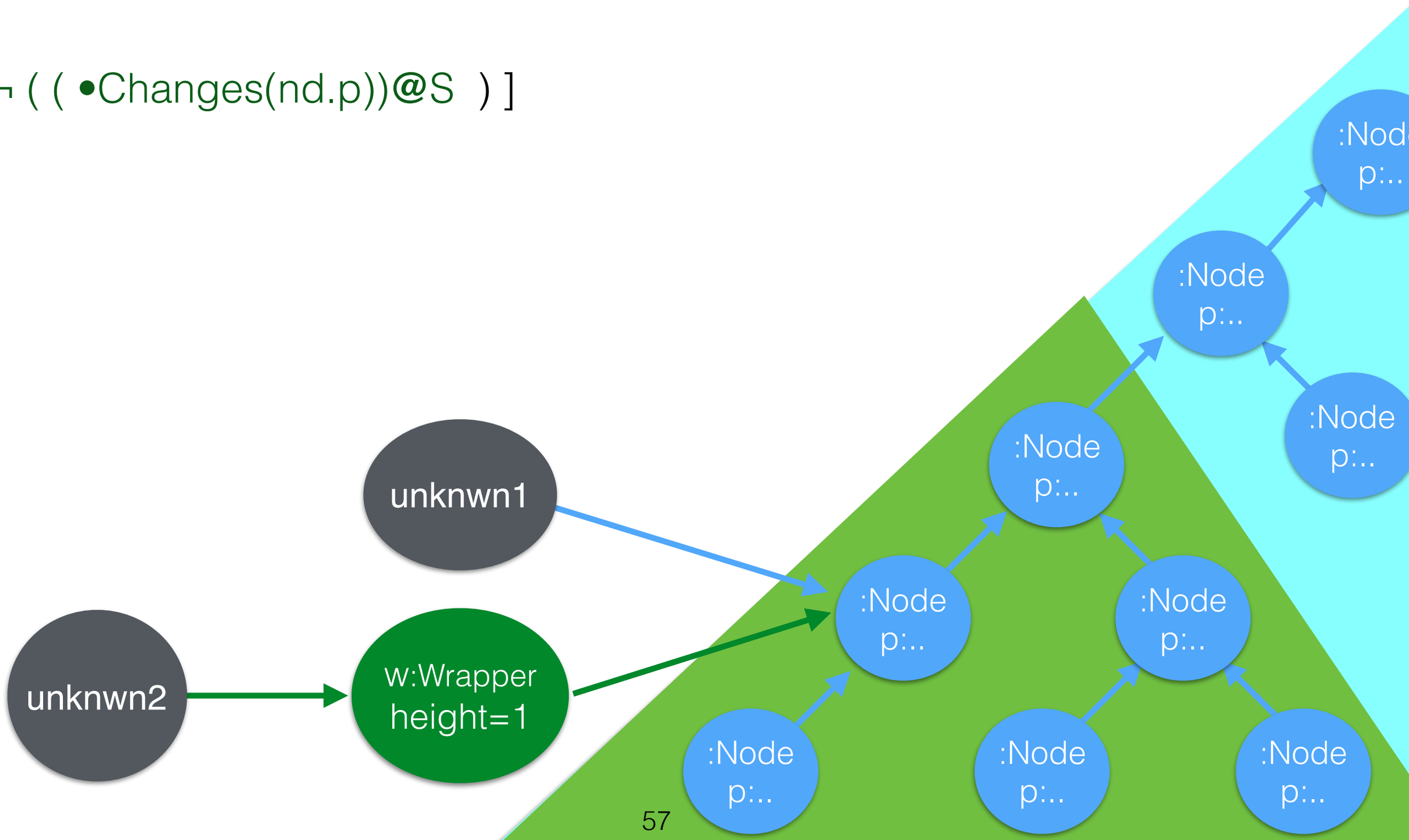
Pol_W ≙

**∀**S:Set. **∀**nd:Node.[

[ [ Access(s,nd) → s:Node ∨ s:Wrapper ∧ Distance(s.node,nd)>s.height ]

→

¬ ( ( •Changes(nd.p))**@**S ) ]



# Example4: DOM attenuation - 3

Pol_W ≙

**∀**S:Set. **∀**nd:Node.[

[ [ Access(s,nd) → s:Node ∨ s:Wrapper ∧ Distance(s.node,nd)>s.height ]

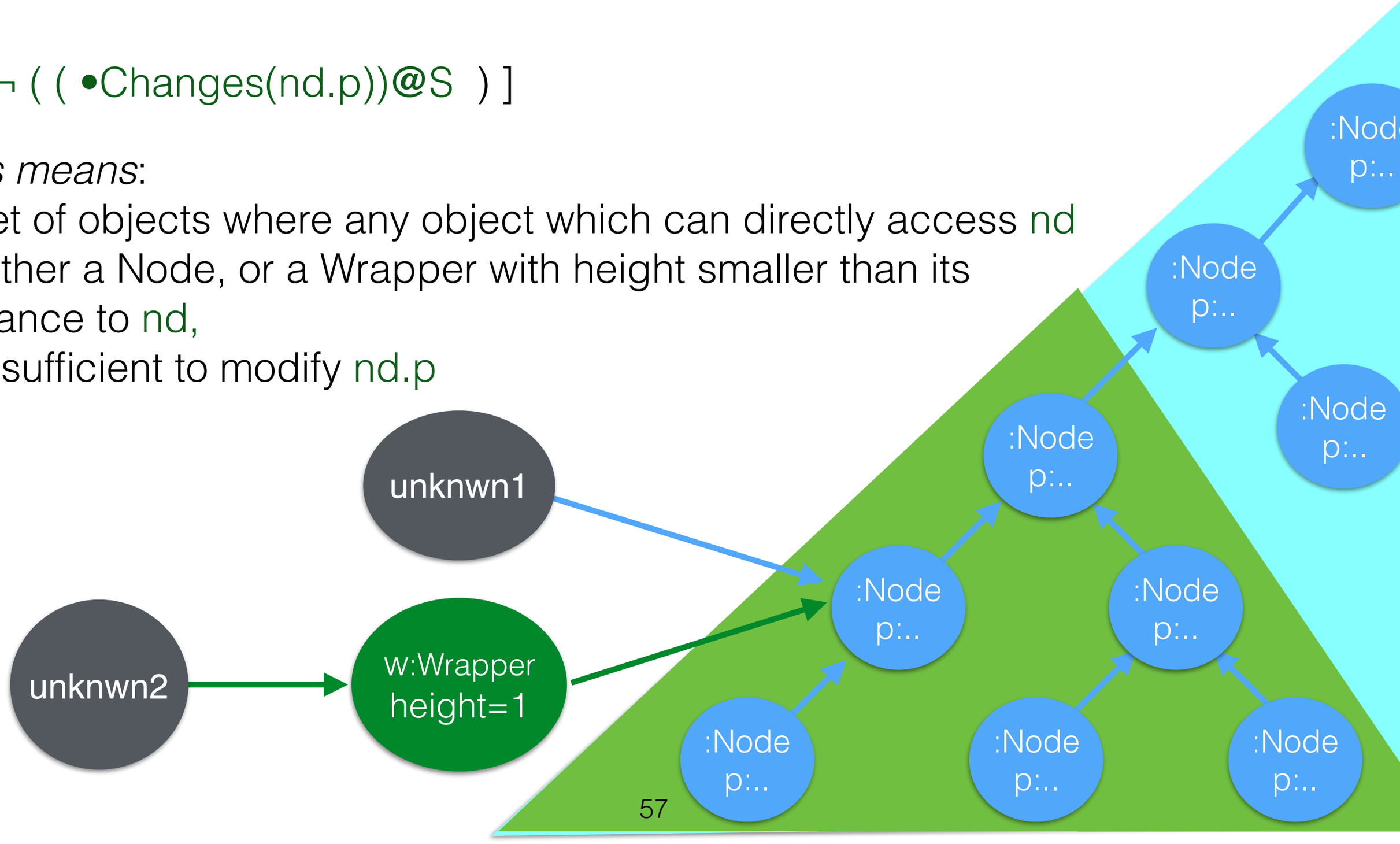
→

¬ ( ( •Changes(nd.p))**@**S ) ]

*This means*:

A set of objects where any object which can directly access nd is either a Node, or a Wrapper with height smaller than its distance to nd,

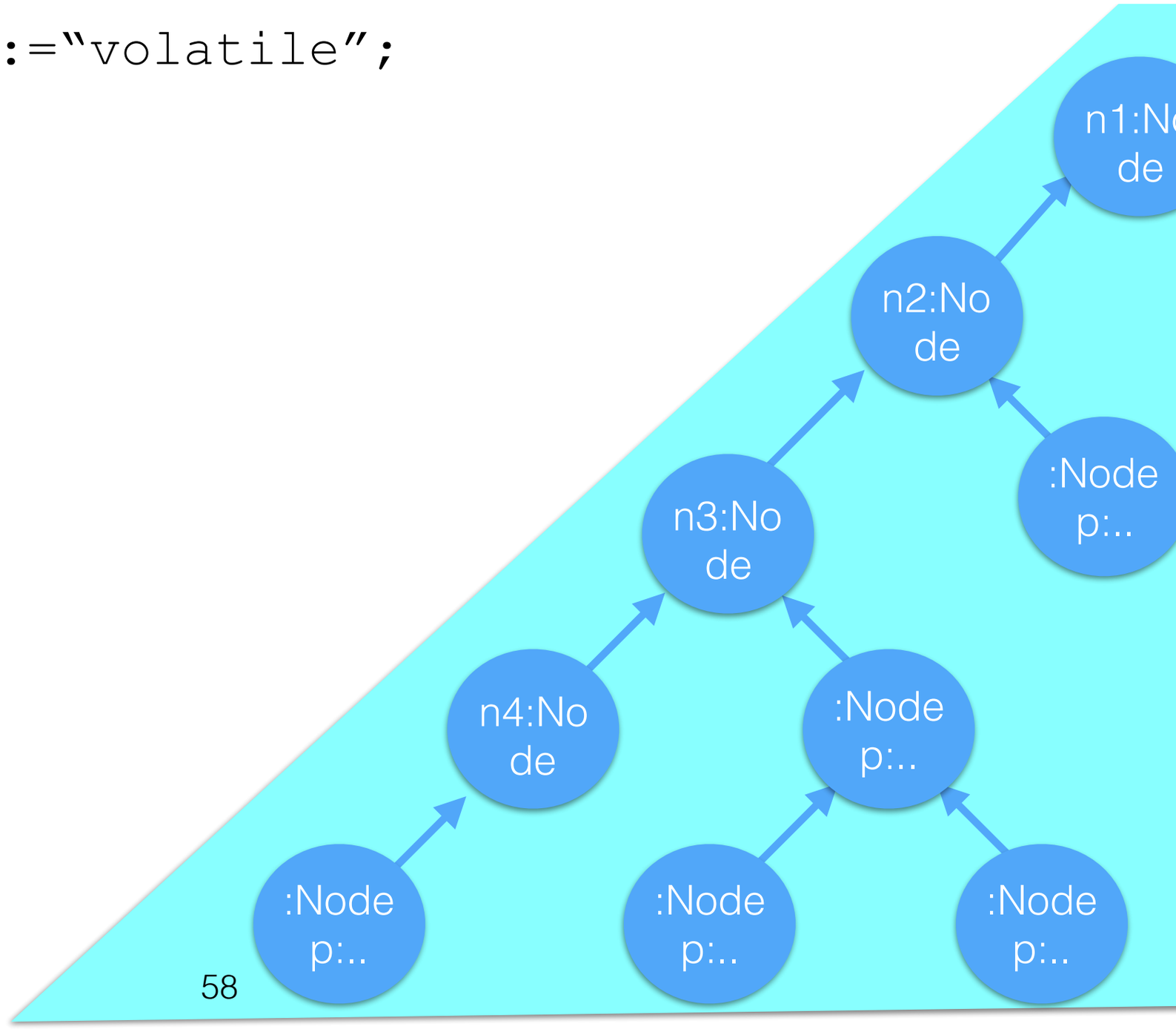
is insufficient to modify nd.p



# Example4: DOM attenuation - use

```
function mm(unknwn){
  n1:=Node(…); n2:=Node(n1,…);n3:=Node(n2,…);n4:=Node(n3,…);
  n2.p:="robust"; n3.p:="volatile";
  w=Wrapper(n4,1);
  unknwn.untrusted(w);

  ….
… •
```

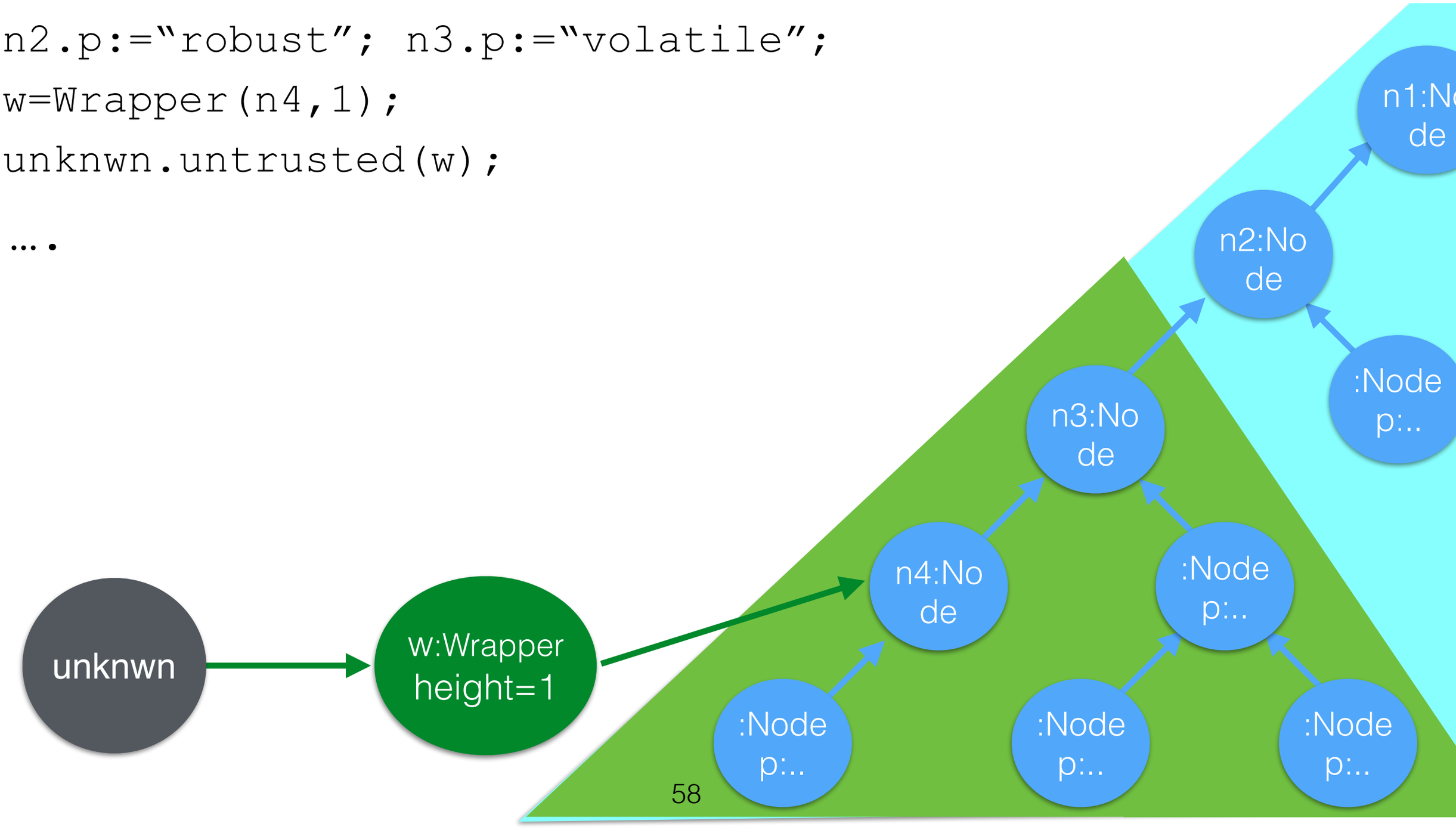


# Example4: DOM attenuation - use

```
function mm(unknwn){
  n1:=Node(…); n2:=Node(n1,…);n3:=Node(n2,…);n4:=Node(n3,…);
  n2.p:="robust"; n3.p:="volatile";
  w=Wrapper(n4,1);
  unknwn.untrusted(w);

  ….
```

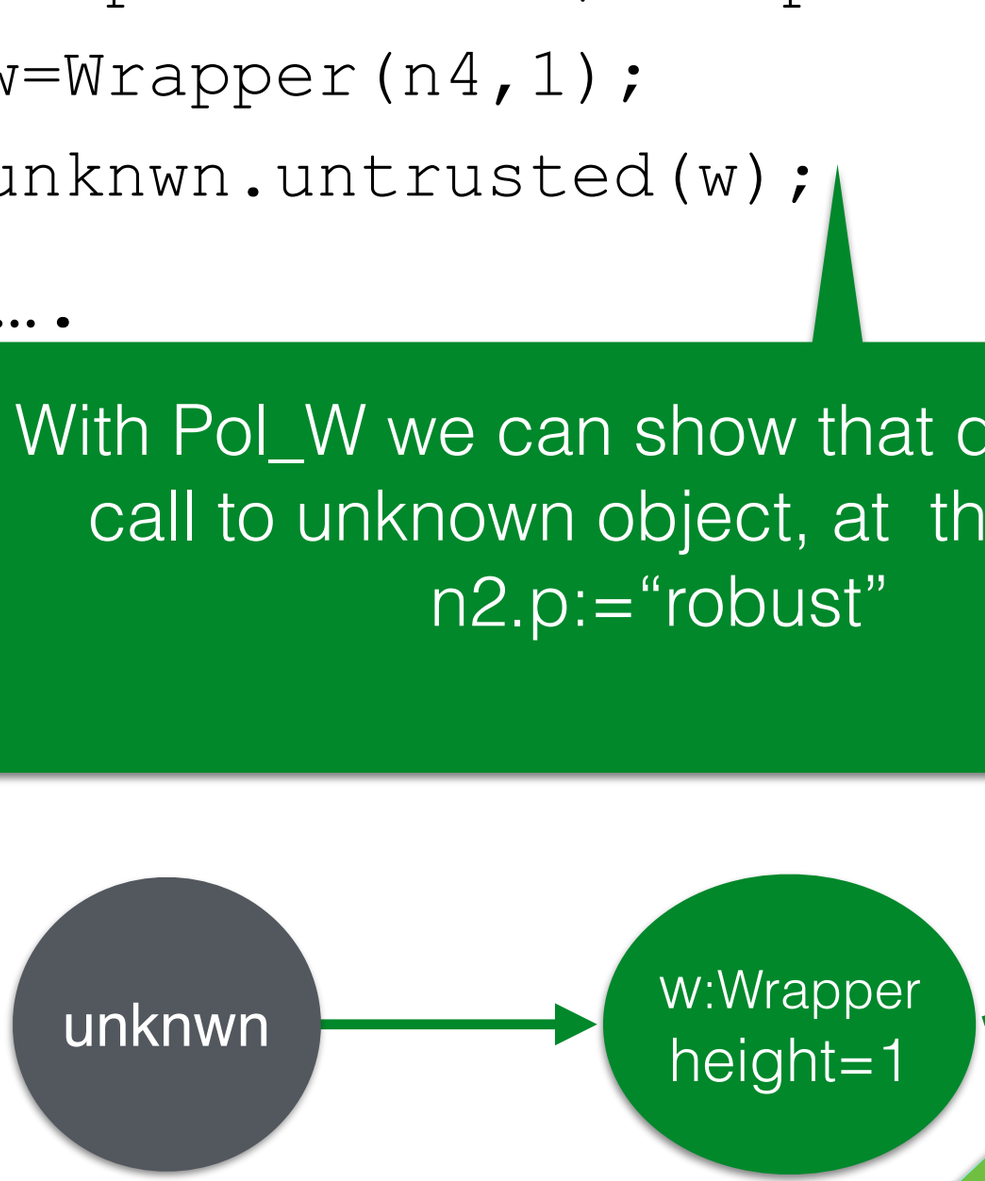
… •



# Example4: DOM attenuation - use

```
function mm(unknwn){
  n1:=Node(…); n2:=Node(n1,…);n3:=Node(n2,…);n4:=Node(n3,…);
  n2.p:="robust"; n3.p:="volatile";
  w=Wrapper(n4,1);
  unknwn.untrusted(w);
  … .
```

With PoI_W we can show that despite the call to unknown object, at this point: n2.p:="robust"



# Summary of our Proposal

A ::= e>e | e=e | P(e1,..en) | …

| A → A | A ∧ A | ∃x. A | …

| Access(x,y) — permission

| Changes(e) — authority

| •A | ○A — time

| A @ S — space

| x.Calls(y,m,z1,..zn) — call

Initial(σ)

Arising(M)

M, σ ⊨ A

M ⊨ A

# Functional vs Robust

- *services offered* by objects/ data structure to *clients*,
- what *will* happen, under *correct use*
- *sufficient* conditions

- *preserved properties* of the objects/data structure
- what *will not* happen, under *arbitrary* use
- *necessary* conditions

# Functional vs Robust

- *services offered* by objects/ data structure to *clients*,
- what *will* happen, under *correct use*
- *sufficient* conditions

$M \models A \{ code \} A'$

- *preserved properties* of the objects/data structure
- what *will not* happen, under *arbitrary* use
- *necessary* conditions

# Functional vs Robust

- *services offered* by objects/ data structure to *clients*,
- what *will* happen, under *correct use*
- *sufficient* conditions

$$M \models A \{ code \} A'$$

- *preserved properties* of the objects/data structure
- what *will not* happen, under *arbitrary* use
- *necessary* conditions

$$M \models A$$

# Functional vs Robust

- *services offered* by objects/ data structure to *clients*,
- what *will* happen, under *correct use*
- *sufficient* conditions

$$M \models A \{ code \} A'$$

- *preserved properties* of the objects/data structure
- what *will not* happen, under *arbitrary* use
- *necessary* conditions

$$M \models A$$

# Functional vs Robust

- *services offered* by objects/ data structure to *clients*,
- what *will* happen, under *correct use*
- *sufficient* conditions

$$M \models A \{ code \} A'$$

- *preserved properties* of the objects/data structure
- what *will not* happen, under *arbitrary* use
- *necessary* conditions

$$M \models A$$

$$M \models \bullet A \rightarrow A'$$

# Functional vs Robust

- *services offered* by objects/ data structure to *clients*,
- what *will* happen, under *correct use*
- *sufficient* conditions

$$M \models A \{ code \} A'$$

- *preserved properties* of the objects/data structure
- what *will not* happen, under *arbitrary* use
- *necessary* conditions

$$M \models A$$

$$M \models \bullet A \rightarrow A'$$

$$M \models A' \rightarrow \neg (\bullet A)$$

# Classical Specification

- fine-grained
- per function

# vs

# Holistic Specification

- ADT as a hole
- emergent behaviour

*Which one is more accurate*?
Classical.

*Which one is more expressive*?
For a “closed” ADT (no functions can be added, all functions have classical specs, and ghost state  has known representation), the holistic specs can be proven.

*When do we need holistic specs*?
* When the holistic aspect more important
(eg cannot lose money unless I authorised).
* When we do not have “closed ADTs.
* When we want to reason in an open world (eg DOM attenuation)

# Classical Specification

- fine-grained
- per function

# vs

# Holistic Specification

- ADT as a hole
- emergent behaviour

*Which one is more accurate*?
Classical.

*Which one is more expressive*?
For a “closed” ADT (no functions can be added, all functions have classical specs, and ghost state  has known representation), the holistic specs can be proven.

*When do we need holistic specs*?
* When the holistic aspect more important
(eg cannot lose money unless I authorised).
* When we do not have “closed ADTs.
* When we want to reason in an open world (eg DOM attenuation)