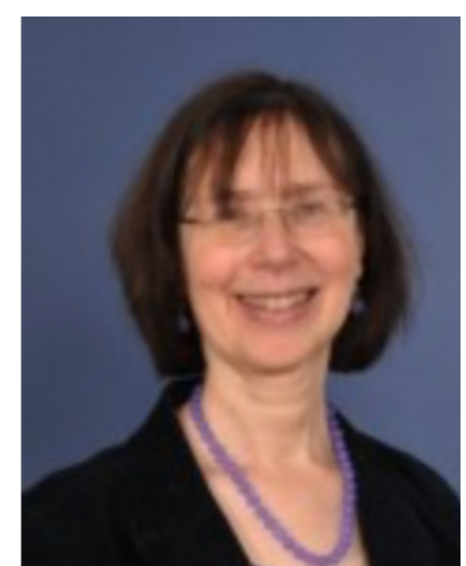
Holistic Specifications for Robust Code

Sophia Drossopoulou
Imperial College London

based on prior work with James Noble (VU Wellington),
Toby Murray (Uni Melbourne) , Mark Miller (Agorics),
and Susan Eisenbach, Shupeng Loh and Emil Klasan (Imperial)



Today

- Traditional Specifications do not adequately address Robustness
- Holistic Specifications — Summary and by Example
- Holistic Specification Semantics

Today

- **Traditional Specifications do not adequately address Robustness**
- Holistic Specifications — Summary and Examples
- Holistic Specification Semantics

Traditional Specification Languages do not adequately address robustness considerations

Traditional Specs

- designed for *closed* world
- pre- and post condition for each function; thus give *sufficient* conditions for some action/effect
- *explicit* about each individual function, and *implicit* about emergent behaviour

Robustness considerations

- concerned with *open* world
- *necessary* conditions for some action/effect
- *explicit* about emergent behaviour

Today

- Traditional Specifications do not adequately address Robustness
- **Holistic Specifications — Summary Examples**
- Holistic Specification Semantics

Holistic Assertions — summary

e ::= this | x | e.fld | …

A ::= e>e | e=e | …

| A → A | A ∧ A | ∃x. A | …

| **Access**(e,e') permission

| **Changes**(e) authority

| **Will**(A) | **Was**(A) time

| A **in** S space

| x.**Calls**(y,m,z1,..zn) control

| x **obeys** A trust

Holistic Assertions — examples

- ERC20
- DAO
- DOM attenuation
- Bank & Account
- Escrow

Example1: ERC20

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.

In particular, a client may call

- `transfer`: transfer some of her tokens to another clients,
- `allow`: authorise another client to transfer some of her tokens on her behalf.
- `transferFrom`: cause another client's tokens to be transferred

Moreover, ERC20 keeps for each client

- `balance` the number of tokens she owns

classical specs - Hoare triples

Hoare triples

A { `x` **calls** `y.f(args)` } **A'**

If **A** hods, and `x` calls `y.f(args)`, then, **A'** holds after the call

precondition

postcondition

ERC20 classical spec - transfer

For any ERC20 contract e, and different clients c1, c2.

c1's balance is larger than m.

{ c1 calls `e.transfer(c2,m)` }

c1's balance decreases by m, and c2's balance increases by m.

$$e{:}ERC20 \;\wedge\; this = c1 \neq c2 \;\wedge\; e.balance(c1) > m$$

{ `e.transfer(c2,m)` }

$$e.balance(c1) = e.balance(c1)_{pre} - m \;\wedge\; e.balance(c2) = e.balance(c2)_{pre} + m$$

precondition

postcondition

ERC20 classical spec - transfer - 2

What if c1's balance not large enough?

$e{:}ERC20 \quad \wedge \quad this = c \quad \wedge \quad e{,}balance(c1) < m$

```
{    e.transfer(c2,m)    }
```

$\forall\ c.\ e{,}balance(c) = e{,}balance(c)_{pre}$

ERC20 classic spec - authorised transfer

For any ERC20 contract e, and different clients c1, c2, c3.

c1 is authorised to spend at least m on c2's behalf and

c2's balance is at least m

{ c1 calls `e.transferFrom(c2,c3,m)` }

c2's balance decreases by m, and c3's balance increases by m.

```
e:ERC20  ∧   this = c1≠c2≠c3≠c1  ∧
e.Authorized(c1,c2,m')  ∧  m'≧ m
e.balance(c1) ≧m
     {  e.transferFrom(c2,c3,m)  }
e.balance(c1) = e.balance(c1)pre -m  ∧
e.balance(c2) = e.balance(c2)pre +m  ∧
e.Authorized(c1,c2,m'-m)
```

ERC20 classic spec - authorised transfer - 2

What if c1 is not authorised, or c1''s authorisation is insufficient, or c2 ihas insufficient tokens?

e:ERC20 $\wedge$ this = c1≠c2≠c3≠c1 $\wedge$

(¬ e.Authorized(c1,c2,m)

   ∨ e.Authorized(c1,c2,m') ∧ m'<m

   ∨ e.balance(c1) < m)

      { e.transferFrom(cl',m) }

∀ c. e.balance(c) = e.balance(c)$_{pre}$ ∧

∀ c,m. [e.Authorized(c1,c2,m)⟷e.Authorized(c1,c2,m)]

ERC20 classic spec - authorising

```
e:ERC20  ∧ this  = c1
     { e.approve(c2,m)  }
e.Authorized(c1,c2,m)
```

ERC20 classical spec

Is that robust?

a “super-cleint,” authorised on all?

a function that takes 0.5% from each account?

who/what can affect my balance?

sufficient ... for chan... of balance

RC20 ∧ thi... = c1 ∧ e,bala...(c1) < m
{ .transf...
∀ c. e,balance(c) ...

e.

∧ e.balance(c1) ≥m
{ e.transferFrom ... m) }
e.balance(c1) = e.bala... pre -m ∧
e.balance(c2) = e.balanc... pre +m ∧ e.Auth...

e:ERC20 ∧ this = c1≠c2... 1 ∧
(¬ e.Authorized(c1,c2... ∨ e.Authorized(c1,c2,m’)
∨ e.balance(c1) < m)
{ e.transferFrom(c2, m) }
∀ c. e.balance(c) = e.balan...(c) pre ∧
∀ c,m. [e.Authorized(c1,c2,m) → e.Authorized(c1,c2,m)]

... { e.totalSupply() } ....

... { e.allowanceOf(c2) } ....

... { e.balanceOf(c) } ....

holistic spec - reduce balance

∀ e:ERC20. ∀ c1: Client. ∀ m: Nat.

[e.balance(c1) = **Was**(e.balance(c1))-m

effect

⟶

(∃c2,c3: Client.

Was (c1.**Calls**(e,transfer,c2,m)))

∨

Was (e.Authorized(c1,c2,m) ∧

c2.**Calls**(e,transferFrom,c1,c3,m)))]

necessary condition

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

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

e.Authorized(c1,c2,m) ≐ **Was***(c1.**Calls**(e.approve(c2,m)))

holistic spec - authority

effect

$$e.\mathrm{Authorized}(c1,c2,m) \doteq$$

$$\mathbf{Was}(\ c1.\mathbf{Calls}(e,\mathtt{approve},c2,m)\ )$$
$$\vee$$
$$\mathbf{Was}(\ e.\mathrm{Authorized}(c1,c2,m+m') \wedge c2.\mathbf{Calls}(e,\mathtt{transferFrom},c1,\_,m')\ )$$
$$\vee$$
$$\mathbf{Was}(\ e.\mathrm{Authorized}(c1,c2,m) \wedge \neg\, c2.\mathbf{Calls}(e,\mathtt{transferFrom},c1,\_,\_)\ )$$

necessary conditions

c2 is authorised by c1 for m iff

in previous step c1 informed e that it authorised c2 for m
or
in previous step c2 was authorised for m+m' and spent m' for c1
or
in previous step c2 was authorised for m and did not spend c1

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'')

… another 7 specs ….

Classical

- per function; :*sufficient* conditions for some action/effect
- *explicit* about individual function, and *implicit* about emergent behaviour

Holistic

- *necessary* conditions for some action/effect
- *explicit* about emergent behaviour

∀ e:ERC20. ∀ cl: Client.
[e.balance(cl) = **Was**(e.balance(cl)) - m)
→
(∃cl',cl'': Client.
Was (cl.**Calls**(`e.transfer`(cl',m)))
∨
Was (Authorized(e,cl, cl'') ∧ cl''.**Calls**(`e.transferFrom`(cl,cl',m)))]

Authorized(cl, cl') ≐ ∃m: Nat. **Was***(cl.**Calls**(`e.approve`(cl',m)))

Example2: DAO simplified

DAO, a "hub that disperses funds"; (https://www.ethereum.org/dao).
… clients may contribute and retrieve funds :

- `payIn(m)` pays into DAO m on behalf of client
- `repay()` withdraws all moneys from DAO

Vulnerability: Through a buggy version of `repay()`, a client could re-enter the call and deplete all funds of the DAO.

classical spec

Assuming DAO keeps a directory of contributions, and require:

R1: directory is compatible with the amount of ether kept in the DAO, and

R2: that withdraw reduces the ether by that amount.

R1: $\forall$d:DAO. d.ether = $\sum_{\texttt{cl} \in \text{dom}(\texttt{d.directory})}$ `d.directory(cl)`

R2: d:DAO ∧ n:Nat ∧ d.directory(cl)=n>0 ∧ this=cl

{ `d.repay()` }

d.directory(cl)=0 ∧ d.**Calls**(cl,send,n)

d:DAO ∧ n:Nat ∧ d.directory(cl)=0 ∧ this=cl

{ `d.repay()` }

"nothing changes"

This spec avoids the vulnerability, *provided* the attack goes through the function repay.

To avoid the vulnerability in general, we need to inspect the specification of *all* the functions in the DAO. DAO - interface has *nineteen* functions.

holistic

∀ cl:External. ∀d:DAO. ∀.n:Nat.
[cl.**Calls**(`d.repay()`) ∧ d.Balance(cl) = n
→
d.ether≥`n` ∧ **Will**(d.**Calls**(cl.send(`n`))]

This specification avoids the vulnerability, regardless of which function introduces it:
The DAO will always be able to repay all its customers.

d.Balance(cl) ≡	0	if cl.**Calls**(d,initialize())
	m+m'	if **Was**(d.Balance(cl),m) ∧ cl.**Calls**(d.payIn(m'))
	0	if **Was**(cl.**Calls**(d.repayIn())
	Was(d.Balance(cl))	otherwise

classical vs holistic

∀d:DAO.   d.ether = $\sum_{cl \in dom(d.directory)}$ d.directory(cl)

d:DAO ∧ n:Nat ∧ d.directory(cl)=n>0 ∧ this=cl
{ d.repay() }
d.directory(cl)=0 ∧ d.Calls(cl.send(n))

d:DAO ∧ n:Nat ∧ d.directory(cl)=0 ∧ this=cl
{ d.repay() }
"nothing changes"

specs for another 19 functions

Classical

- per function; *sufficient* conditions for some action/effect
- *explicit* about individual function, and *implicit* about emergent behaviour

Holistic

- *necessary* conditions for some action/effect
- *explicit* about emergent behaviour

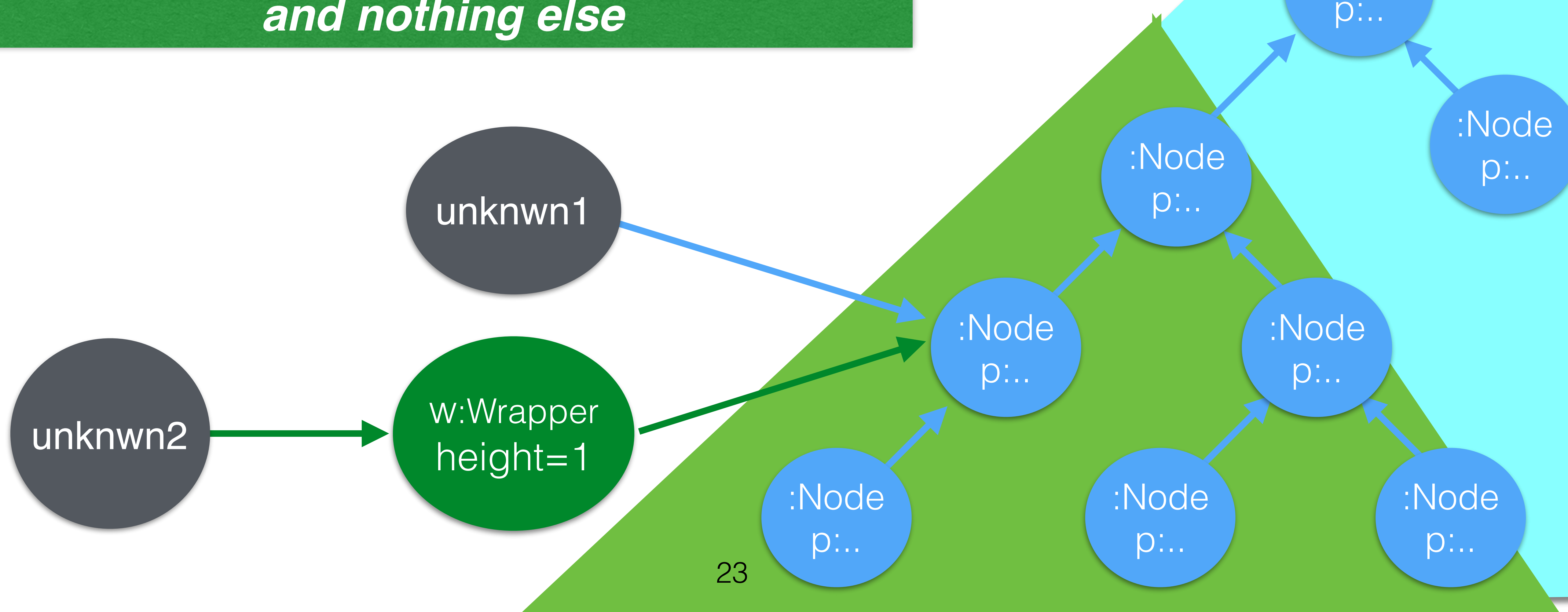
∀ cl:External. ∀d:DAO. ∀.n:Nat.
[cl.**Calls**(d.repay()) ∧ d.Balance(cl) = n
→
d.ether≥n ∧ **Will**(d.**Calls**(cl.send(n))]

d.Balance(cl) ≡	m	if cl.**Calls**(d,initialize,m)
	m+m'	if **Was**(d.Balance(cl),m) ∧ **Was**(cl.**Calls**(d.payIn(m'))
	0	if **Was**(cl.**Calls**(d.repayIn())

Example 3: 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



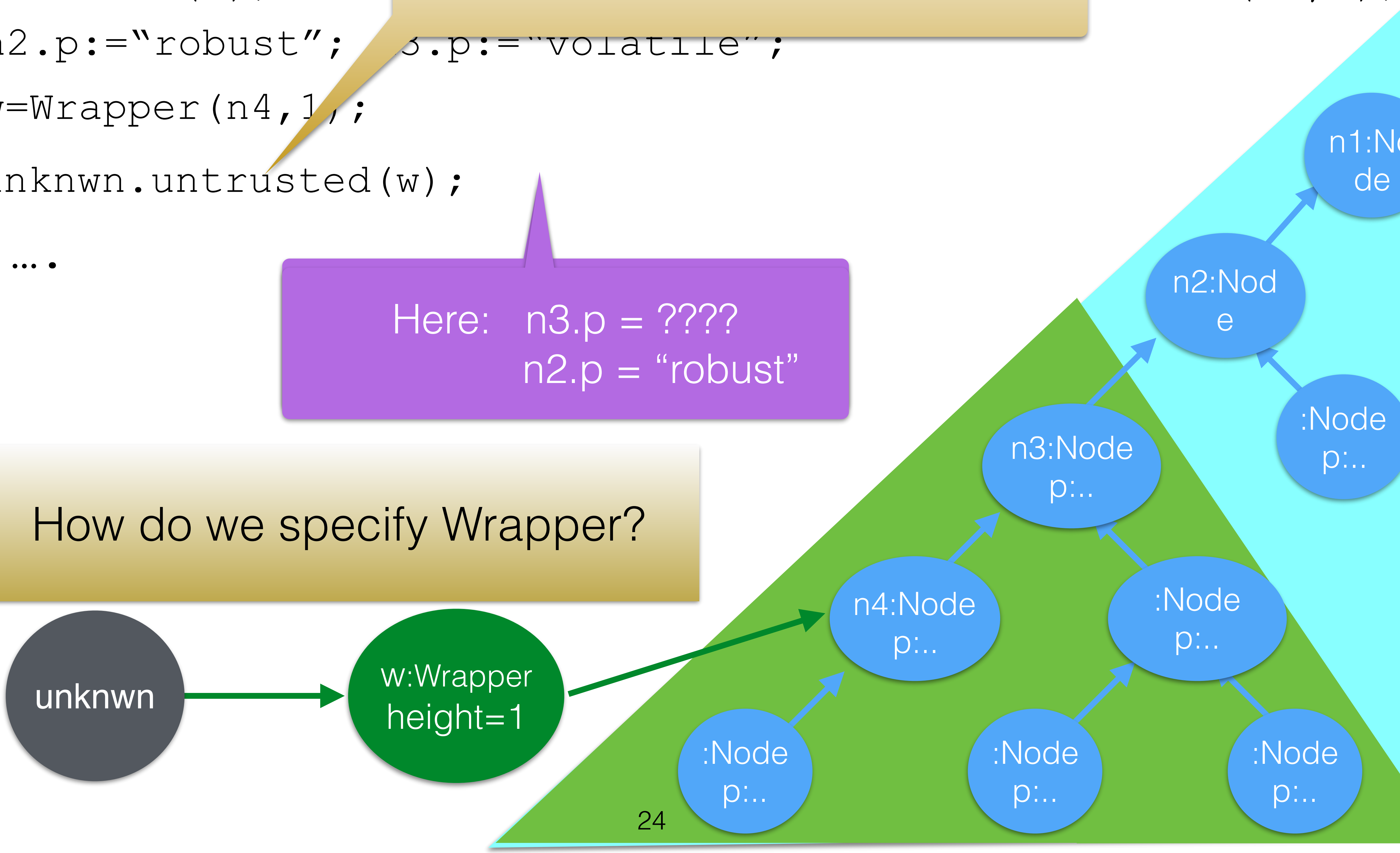
DOM attenuation

```
function mm(unknwn)
   n1:=Node(…); n2:=                              :=Node(n3,…);
   n2.p:="robust";   3.p:="volatile";
   w=Wrapper(n4,1);
   unknwn.untrusted(w);
   ….
```

open world

Here: n3.p = ????
n2.p = "robust"

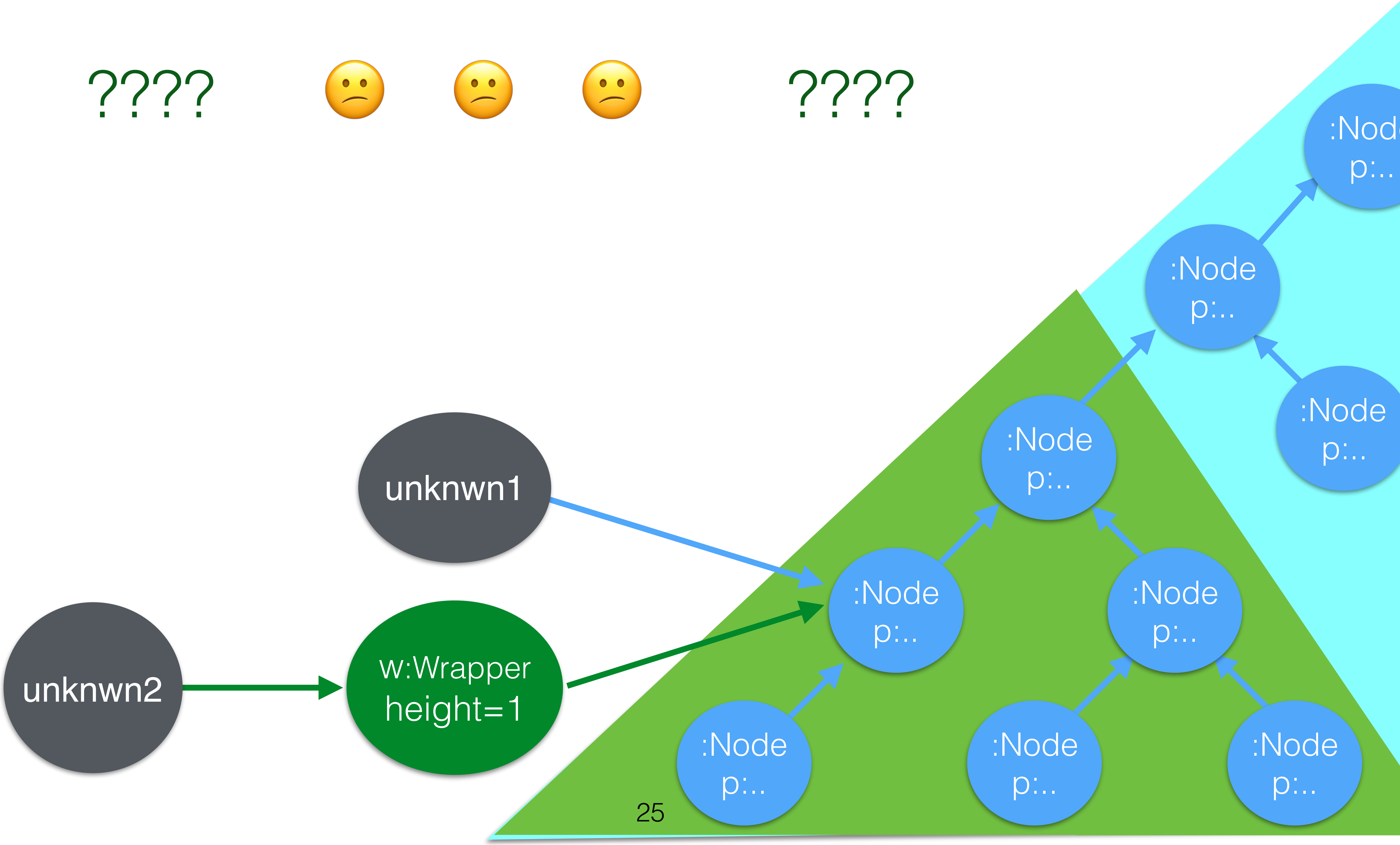
How do we specify Wrapper?



classical

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

???? 😕 😕 😕 ????



holistic

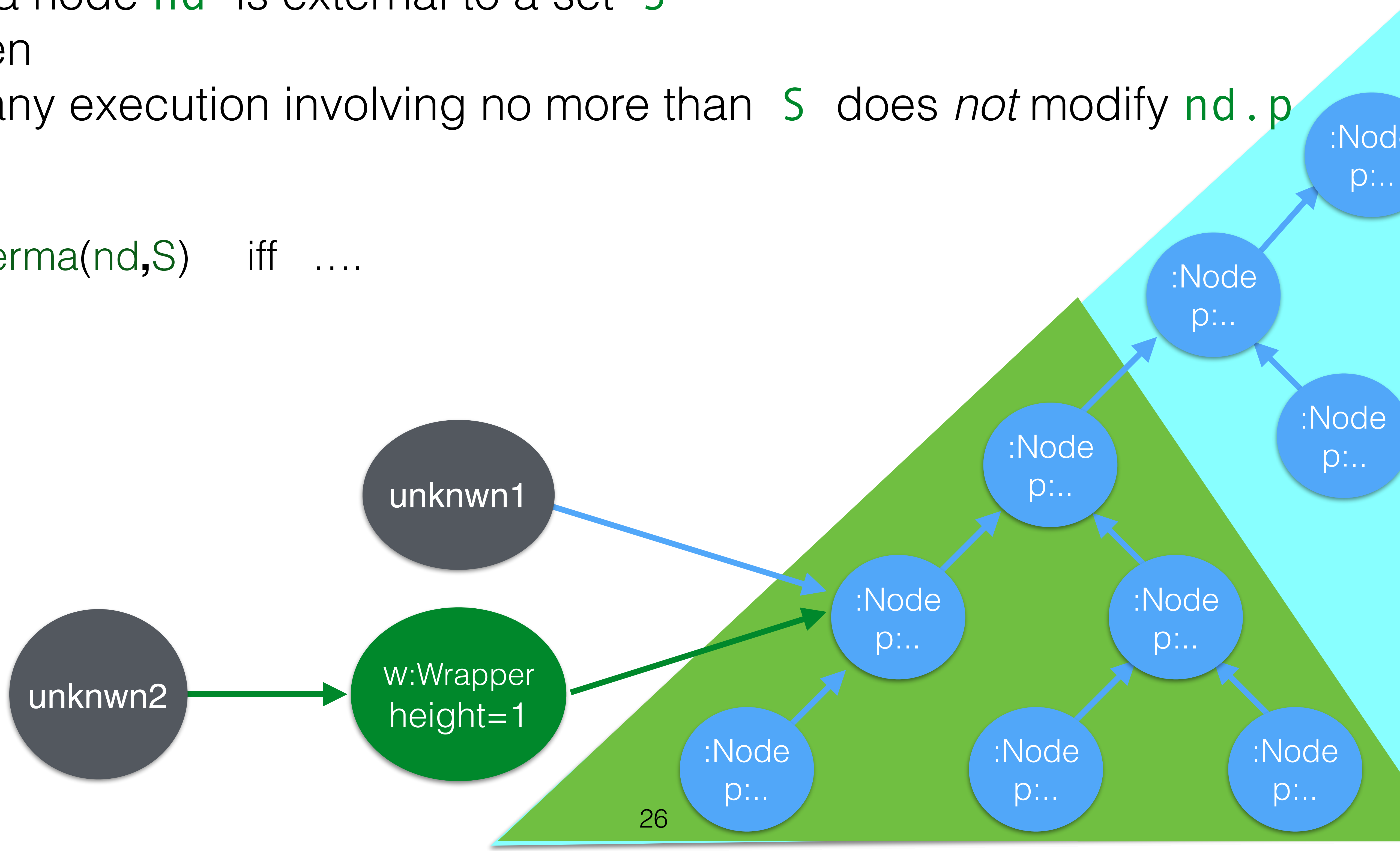
If

a node nd is external to a set S

then

any execution involving no more than S does *not* modify nd.p

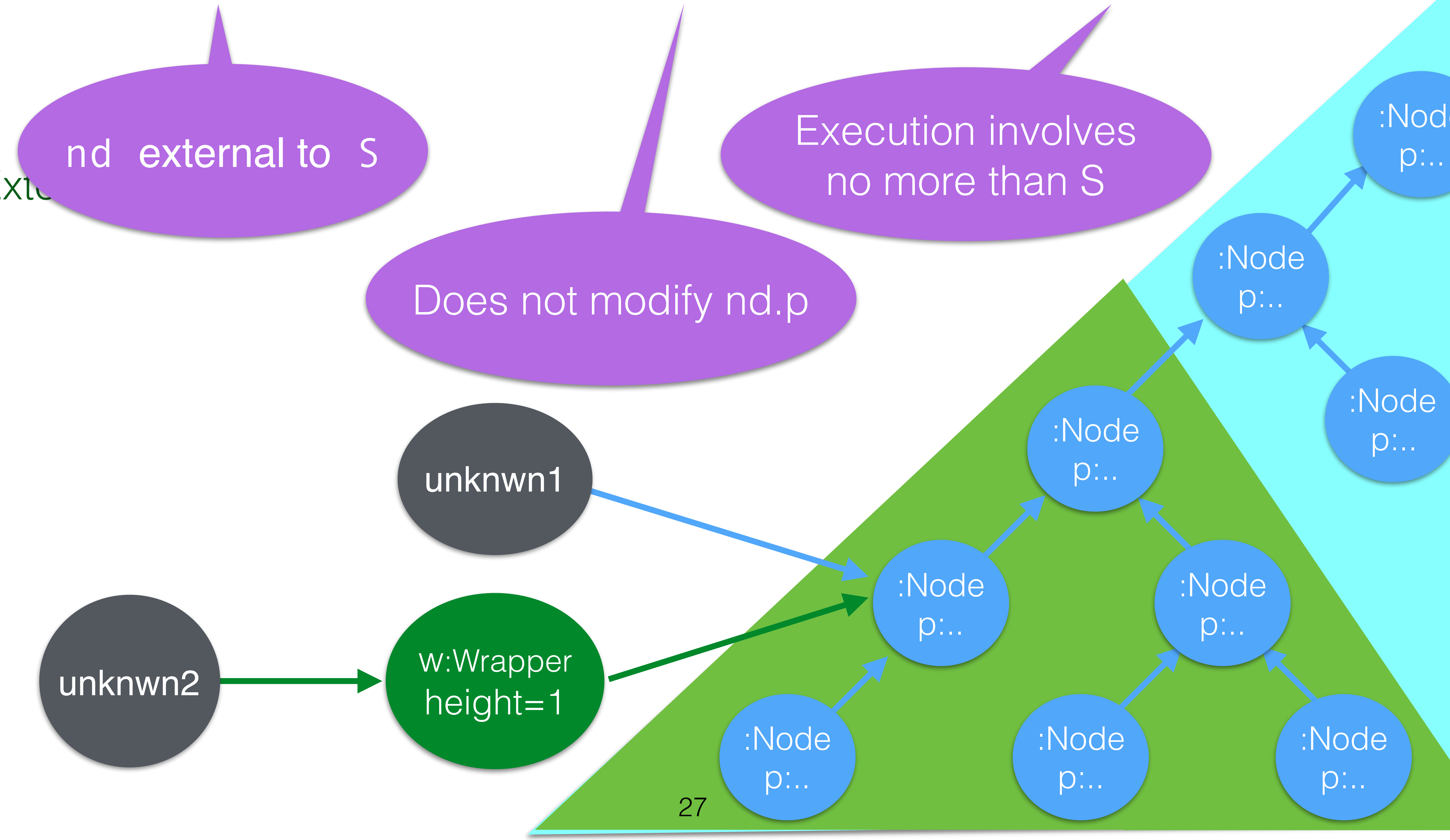
Externa(nd,S) iff ....



holistic

∀S:Set. ∀nd:Node.

[Extenal(nd,S) → ¬ (**Will**(**Changes**(nd.p)) **in** S)]



holistic

∀S:Set. ∀nd:Node.

[Extenal(nd,S) → ¬ (**Will**(**Changes**(nd.p)) **in** S)]

Extenal(nd,S) iff ∀o∈S. ∀path

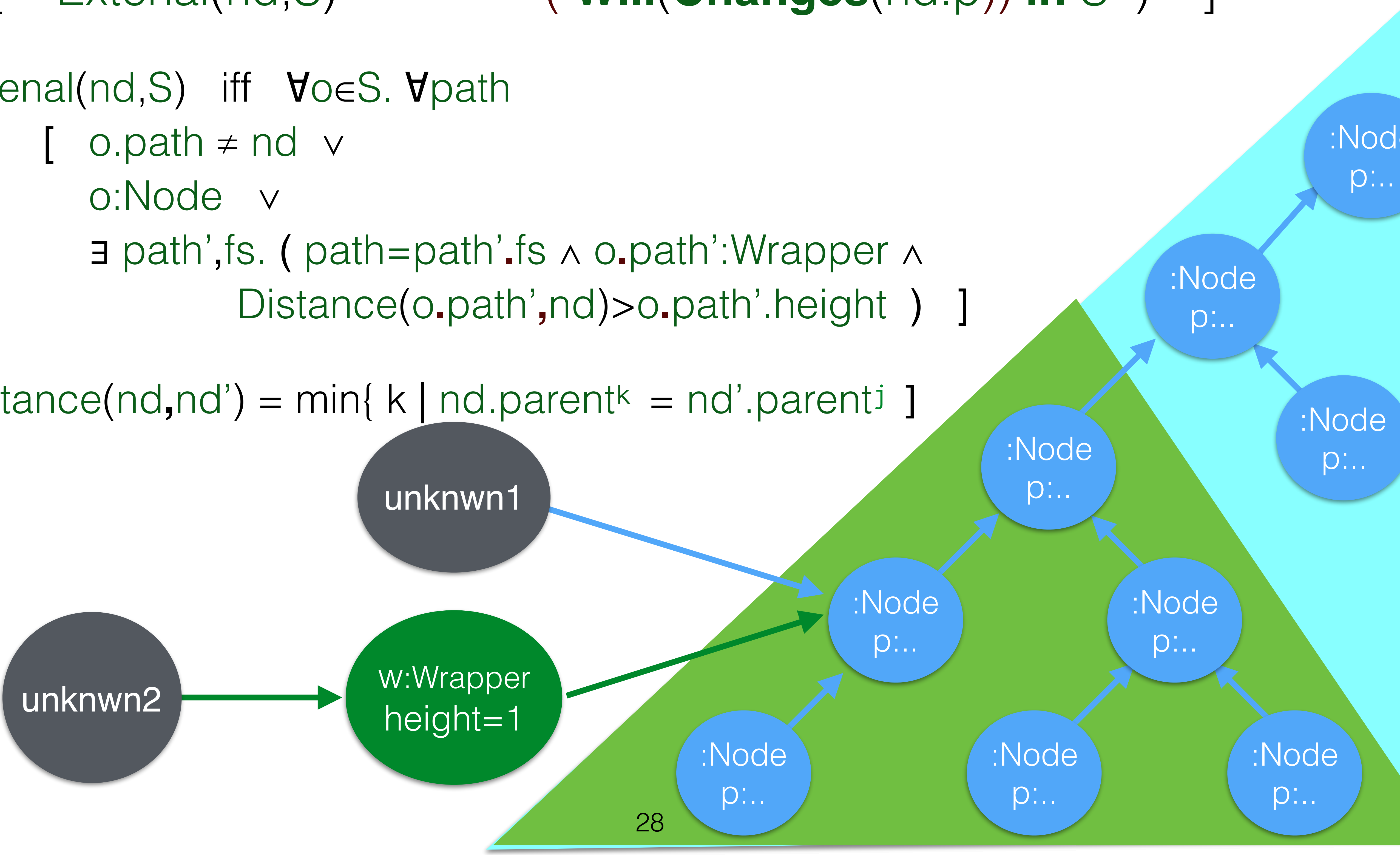
[o.path ≠ nd ∨

o:Node ∨

∃ path',fs. (path=path'.fs ∧ o.path':Wrapper ∧

Distance(o.path',nd)>o.path'.height)]

Distance(nd,nd') = min{ k | $nd.parent^k$ = $nd'.parent^j$]

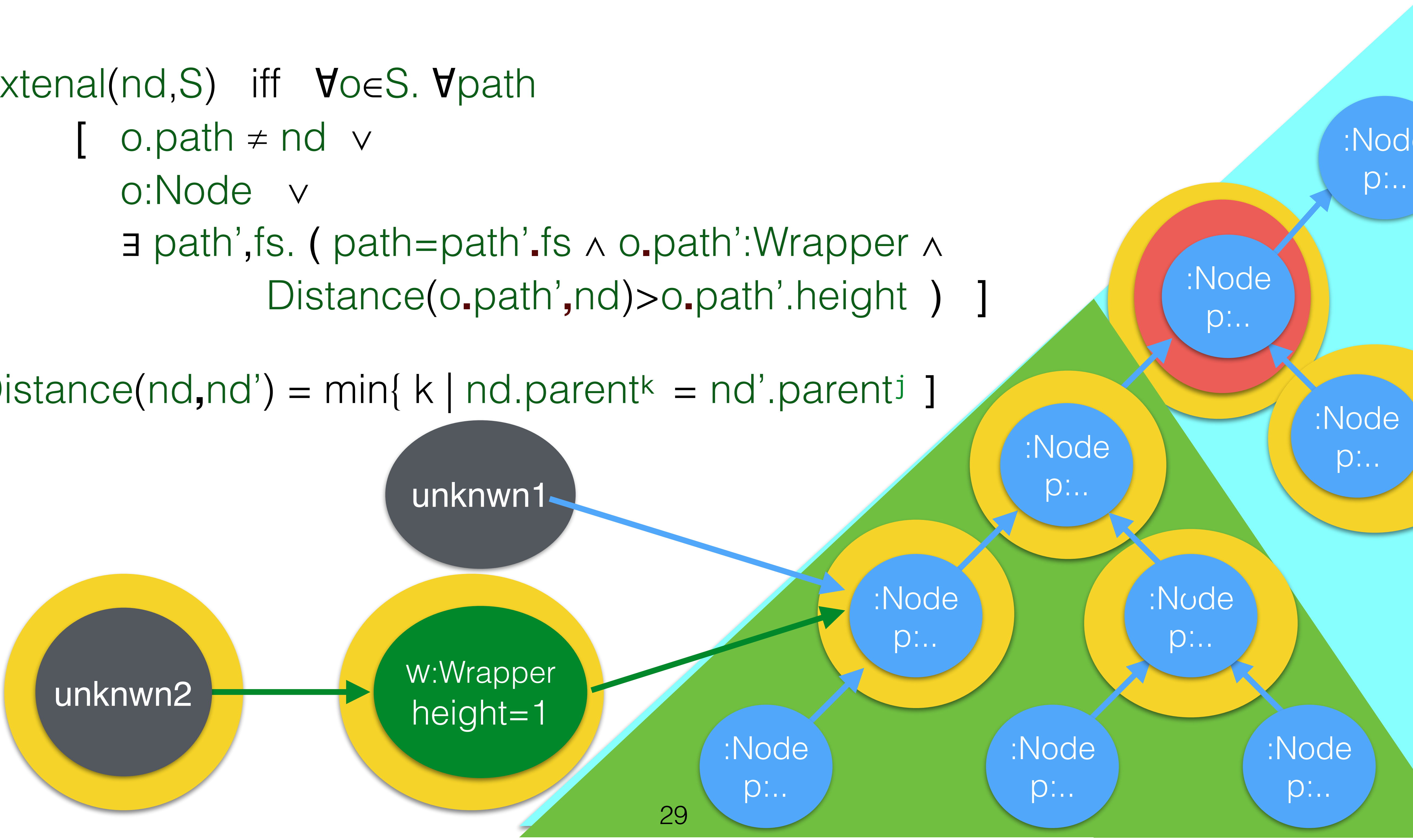


external

Extenal(**RedNode**, **YellowSet**)

Extenal(nd,S) iff $\forall o \in S.\ \forall path$
[$o.path \neq nd$ $\vee$
$o{:}Node$ $\vee$
$\exists$ path',fs. (path=path'.fs $\wedge$ o.path':Wrapper $\wedge$
Distance(o.path',nd)>o.path'.height)]

Distance(nd,nd') = min{ k | $nd.parent^{k}$ = $nd'.parent^{j}$]

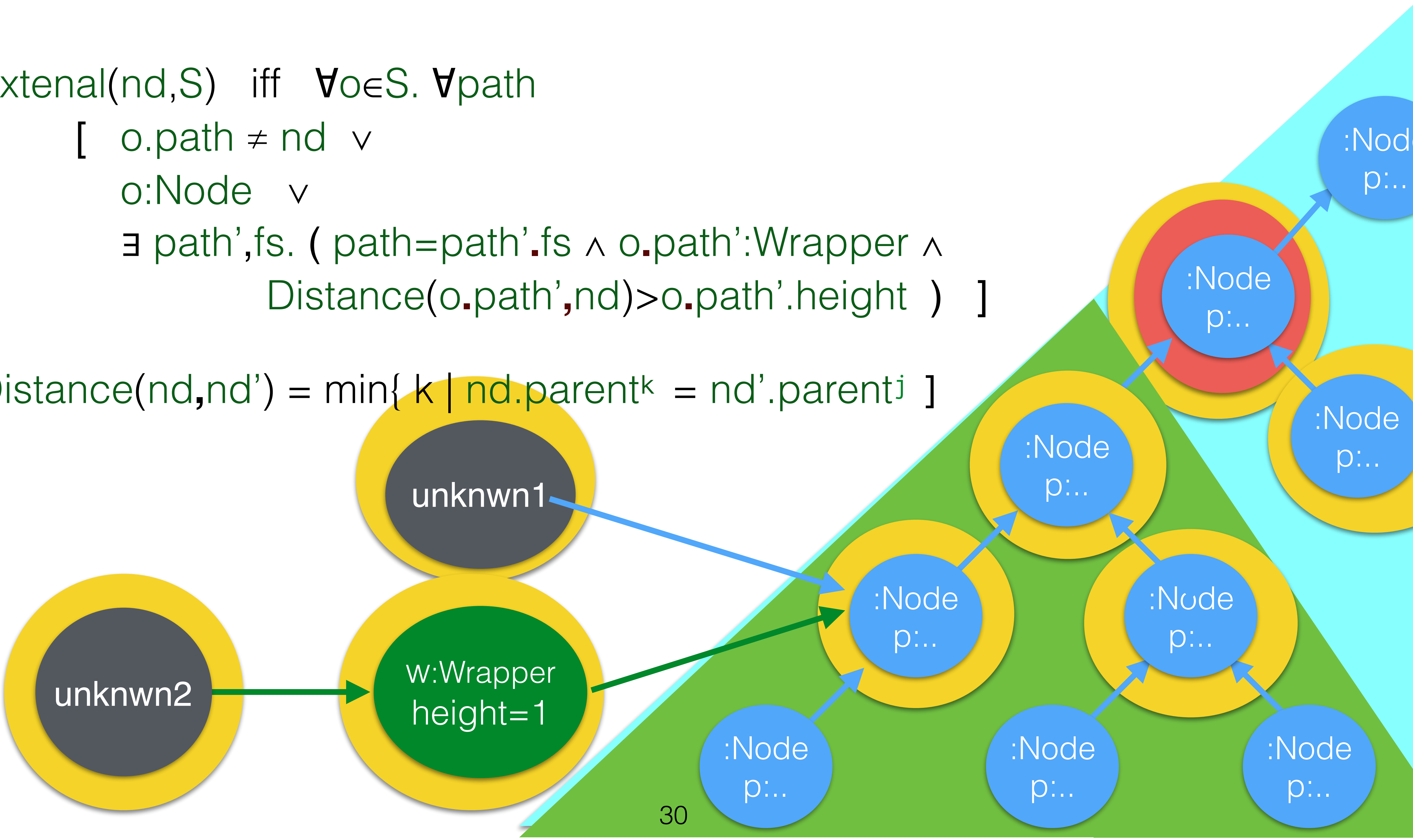


external

¬ Extenal(**RedNode**, **YellowSet**)

Extenal(nd,S) iff ∀o∈S. ∀path
[o.path ≠ nd ∨
o:Node ∨
∃ path’,fs. (path=path’.fs ∧ o.path’:Wrapper ∧
Distance(o.path’,nd)>o.path’.height)]

Distance(nd,nd’) = min{ k | nd.parent^k = nd’.parent^j]



external

$\neg$ Extenal(**RedNode**, **YellowSet**)

Extenal(nd,S) iff $\forall o \in S.\ \forall path$

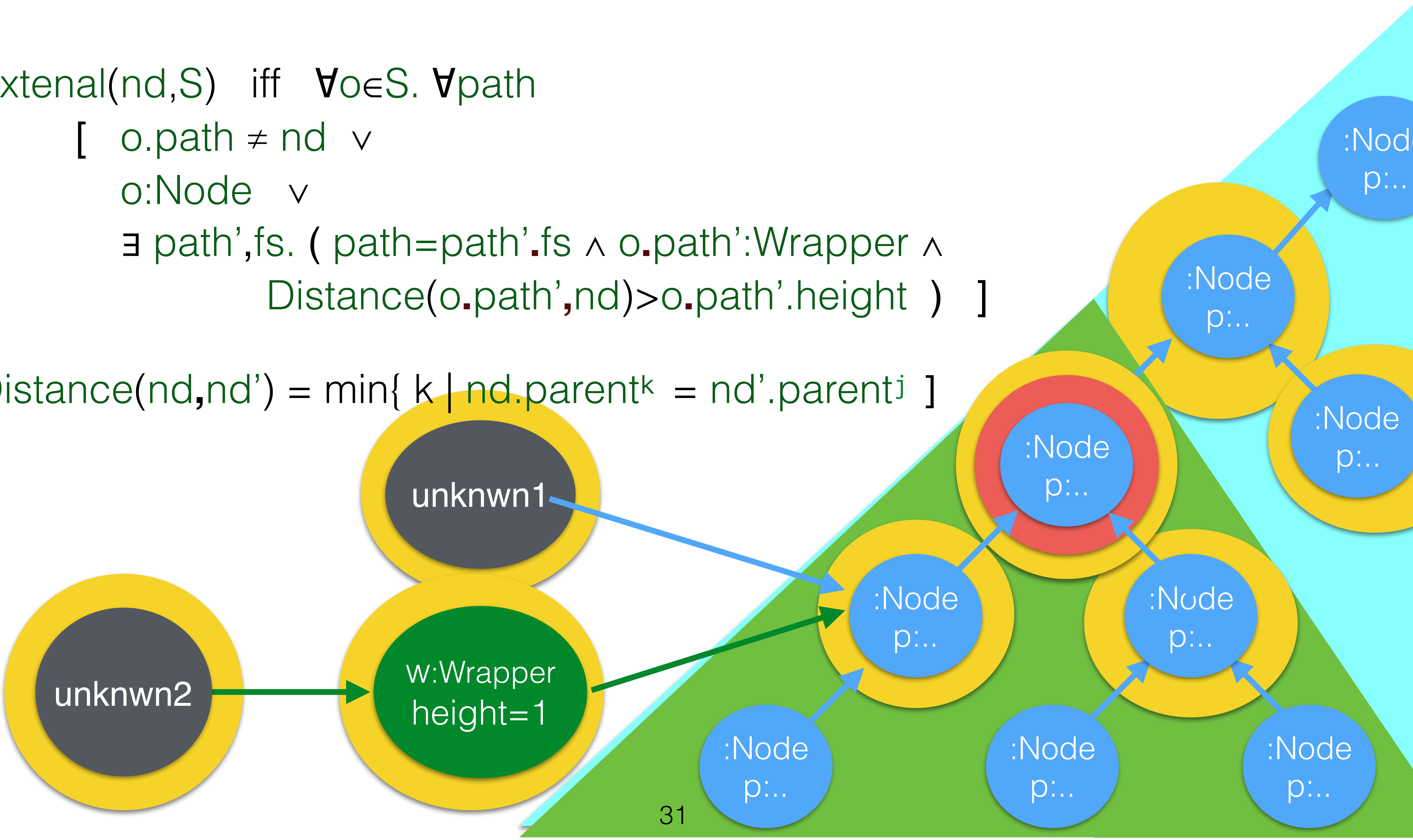
$[\ \ o.path \neq nd\ \ \lor$

$o{:}Node\ \ \lor$

$\exists\ path', fs.\ (\ path = path'.fs\ \land\ o.path'{:}Wrapper\ \land$

$Distance(o.path', nd) > o.path'.height\ )\ \ ]$

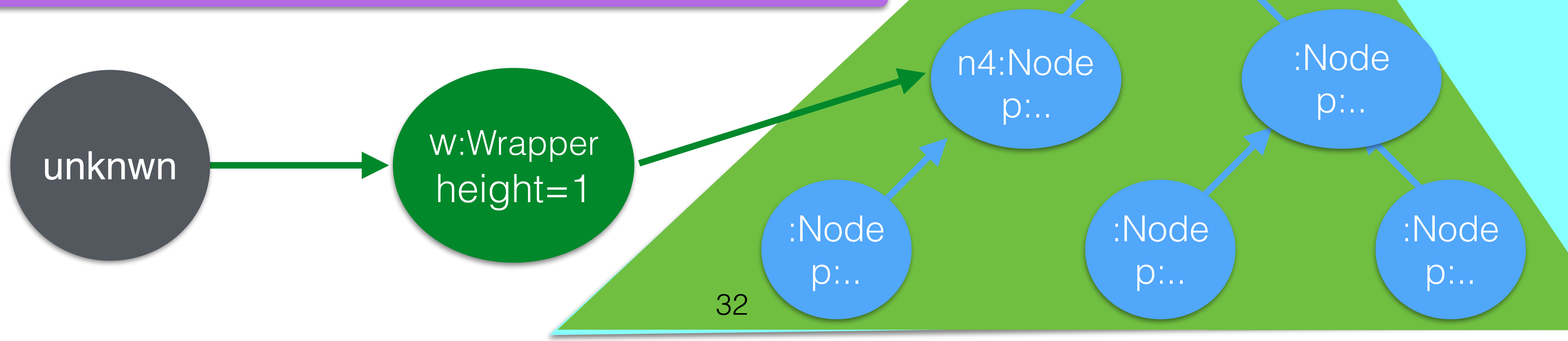
$Distance(nd, nd') = \min\{\ k \mid nd.parent^k\ =\ nd'.parent^j\ \}$



using holistic spec

```
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 holistic spec we can show
that despite the call to unknown object,
at this point:
n2.p = “robust”



Bank and 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

classical | 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]

Pol_4 — holistic

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

$$a{:}Account \;\; \wedge \;\; \textbf{Will}\,(\, \textbf{Changes}(a.balance)\,)\; \textbf{in}\; S\;)$$

$$\rightarrow$$

necessary condition

$$\exists o \in S.\; Access(o,a)$$

This says: If some execution starts now and involves at most the objects from S, and modifies a.balance at some future time, then at least one of the objects in S can access a directly now.

Pol_4 — classical

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

???? 😕 😕 😕 ????

Today

- Traditional Specifications do not adequately address Robustness
- Holistic Specifications — Summary and by Example
- **Holistic Specification Semantics**

Giving meaning to holistic Assertions

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

module	M : Ident —-> ClassDef ∪ PredicateDef ∪ FunctionDef
configuration	σ : Heap × Stack × Code
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
- (M1*M2)*M3 = M1*(M2*M3)
- M, σ ⇝ σ' ∧ M*M' defined ⟶ M*M', σ ⇝ σ'

We will define

- M, σ ⊨ A
- Initial(σ) and Arising(M)
- M ⊨ A

Giving meaning to holistic Assertions

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

module M : Ident —-> ClassDef ∪ PredicateDef ∪ FunctionDef
configuration σ : Heap × Stack × Code
execution $M, \sigma \rightarrow \sigma'$

Define module concatenation * so that
$M*M'$ undefined, iff $dom(M) \cap dom(M') \neq \emptyset$
otherwise
$(M*M')(id) = M(id)$ if $M'(id)$ undefined, else $M'(id)$

We will define $M, \sigma \models A$
Initial(σ) and Arising(M)
$M \models A$

Holistic Assertions — summary

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

| **Will**(A) | **Was**(A) time

| A **in** S space

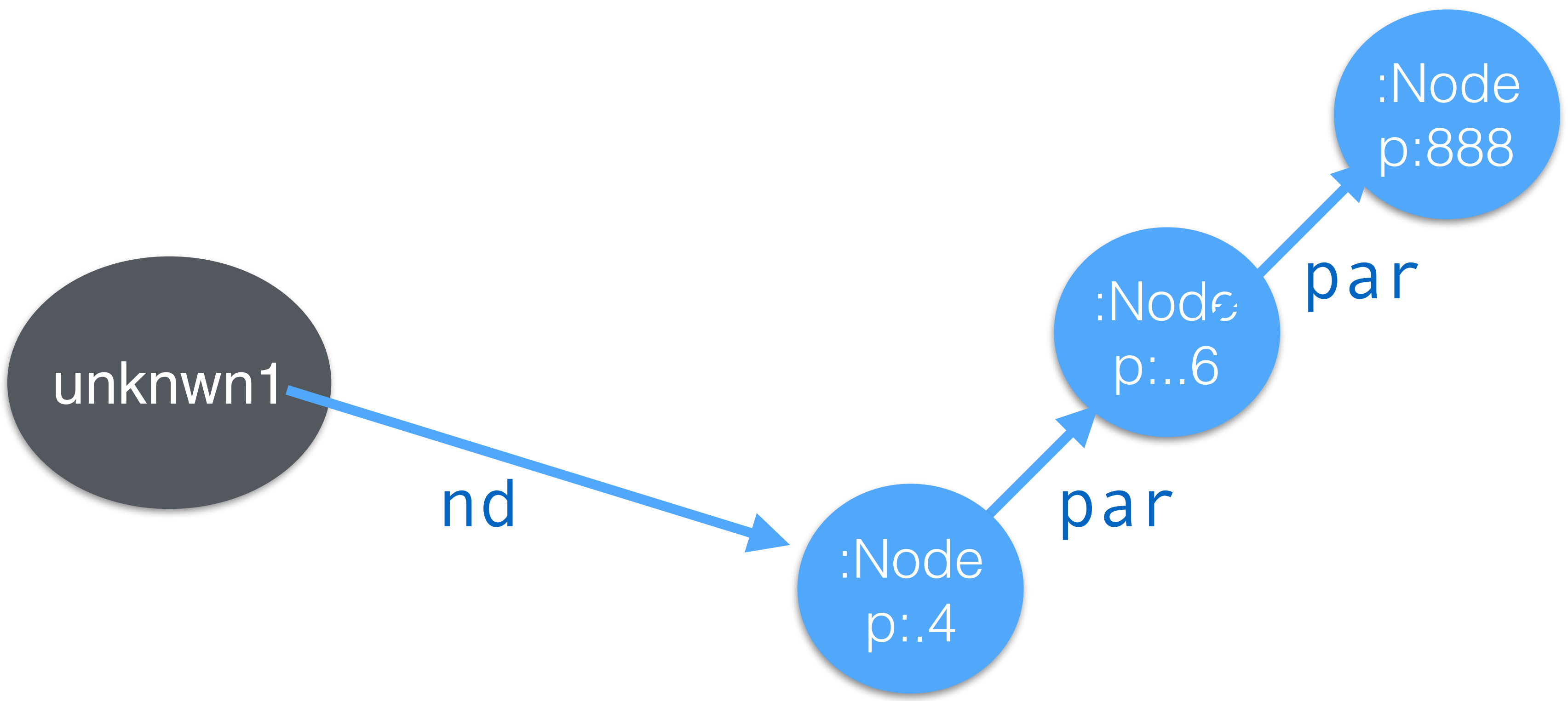
| x.**Call**(y,m,z1,..zn) control

| x **obeys** A trust

Semantics of Expressions

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

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



Eg, $\lfloor \text{unknwn1.nd.par.par.p} \rfloor_{M,\sigma} = 888$

Semantics of holistic Assertions

"Conventional part"

$A \ ::= \ e>e \ \mid \ A \rightarrow A \ \mid \ \exists x.A \ \mid \ \dots$

We define $M, \sigma \vDash A$

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

$M,\sigma \vDash A \rightarrow A'$ iff $M,\sigma \vDash A$ implies $M,\sigma \vDash A'$

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

Semantics of holistic Assertions

"Unconventional part"

$A ::= \textbf{Access}(x,x') \mid \textbf{Changes}(e) \mid \textbf{Will}(A) \mid A\ \textbf{in}\ S \mid x.\textbf{Calls}(y,m,z_1,..z_n)$

$M,\sigma \models \textbf{Access}(x,x')$ iff $\lfloor x \rfloor_{M,\sigma} = \lfloor x' \rfloor_{M,\sigma} \quad \vee$

$\lfloor x.fld \rfloor_{M,\sigma} = \lfloor x' \rfloor_{M,\sigma}$ for some field $fld \quad \vee$

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

$\wedge$ y is formal parameter of current function

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

$M,\sigma \models \textbf{Will}(A)$ iff $\exists \sigma'.[\ M,\sigma \rightsquigarrow^* \sigma' \wedge M, \sigma' \models A\ ]$

$M,\sigma \models A\ \textbf{in}\ S$ iff $M,\sigma@_{Os} \models A$ where $Os = \lfloor S \rfloor_{M,\sigma}$

$M,\sigma \models x.\textbf{Calls}(y,m,z_1,..z_n)$ iff $\lfloor this \rfloor_{M,\sigma} = \lfloor x \rfloor_{M,\sigma} \wedge \sigma.code{=}y'.m(z_1'..z_n') \wedge ..$

Semantics of holistic 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, σ, σ'

$M,\sigma \models$ **Will**(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, σ, σ'

$M,\sigma \models$ A **In** S iff $M,\sigma@_{Os} \models A$ where $Os = \lfloor S \rfloor_{M,\sigma}$

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

Givin… outstanding definitions …rtions

$M,\sigma \models$ **Access**(x,y) iff ....

$M,\sigma \models$ **Changes**(e) iff ....

$M,\sigma \models$ **Will**(A) iff ...

$M,\sigma \models A$ **in** S iff M,σ**@**$OS \models A$ where $OS = \lfloor S \rfloor_{M,\sigma}$

$M,\sigma \models x.$**Calls**$(y.m(x_1,..x_n))$ iff ...

outstanding definitions: Initial, and Arising

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

The code can be arbitrary

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

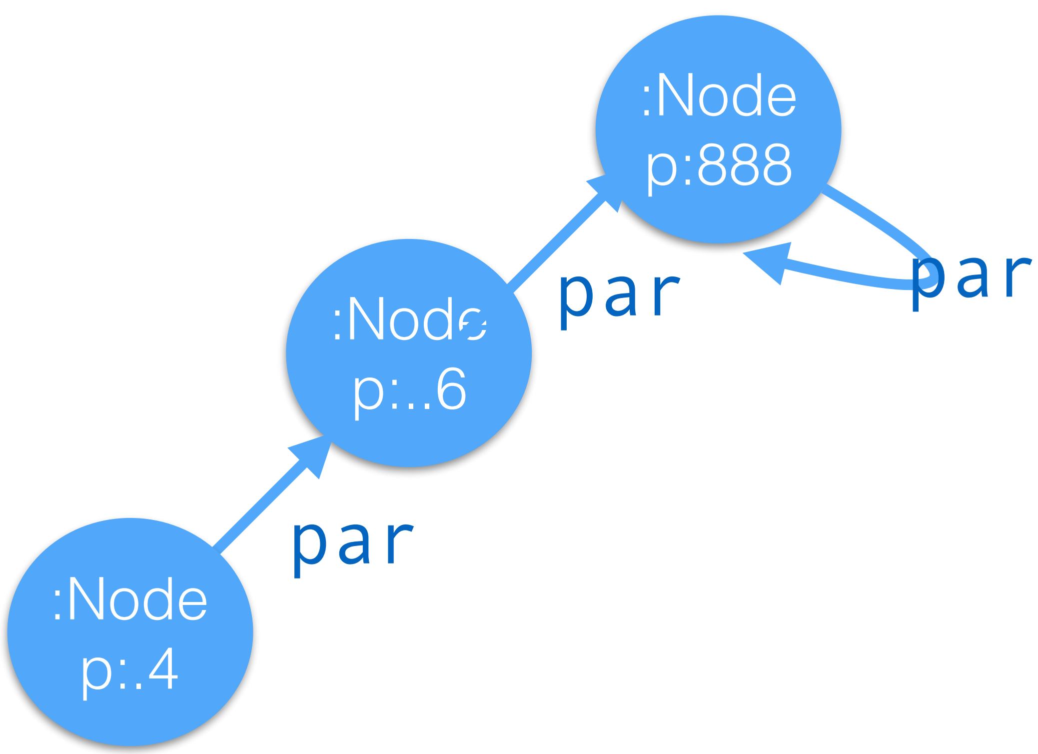
A runtime configuration σ *arises* from a module M if there is some initial configuration σ_0 whose execution M in reaches σ in a finite number of steps.

$$\text{Arising}(M) = \{ \sigma \mid \exists \sigma_0.\ \text{Initial}(\sigma_0) \land M, \sigma_0 \leadsto^* \sigma \}$$

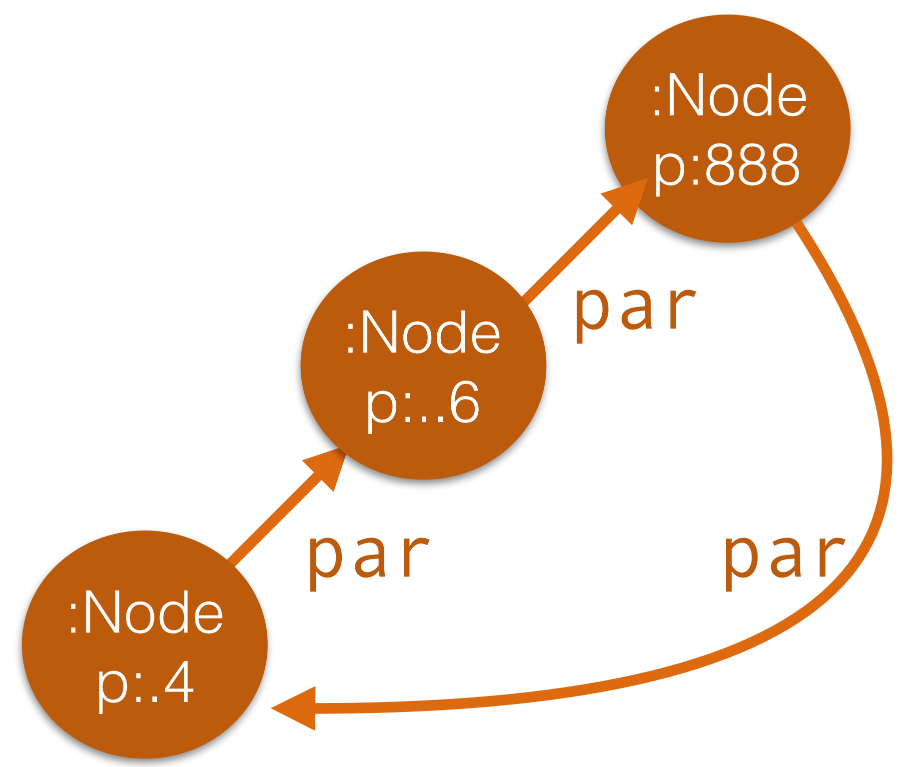
Arising expresses "defesnsiveness"

Assume a Tree-module, M_{tree}.

`blue configuration`
arises from M_{tree}*M'
for some module M'



`brown configuration`
does not arise from M_{tree}*M'
for *any* module M'



Giving meaning to Assertions

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

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

open world

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 $M \models A \subseteq A'$
- $M \models (\ \mathbf{Will}(A) \rightarrow A'\ ) \sqsubseteq (\ A' \rightarrow \mathbf{Was}(A)\ )$

- $M \models A @ S$ and $M \models S \subseteq S'$ imply $M \models A @ S'$
- $M \models A @ S$ and $M \models S \subseteq S'$ does not imply $M \models A @ S'$?
- $M,\sigma \models (\bullet A)@S$ imply $M,\sigma \models \bullet(A@S')$?
- We call $(\bullet A)@S$ *monotonic*, if $M,\sigma \models (\bullet A)@S$ 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 $M,\sigma \models \bullet(A @ (S \cup (Allocated \setminus S')))$

Summary of our Proposal

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

| **Access**(x,y) permission

| **Changes**(e) authority

| **Will**(A) | **Was**(A) time

| A **in** S space

| x.**Calls**(y,m,z1,..zn) call

Initial(σ)

M, σ ⊨ A

Arising(M)

M ⊨ A

Classical Specification vs Holistic Specification

- fine-grained
- per function

- ADT as a whole
- emergent behaviour

Which is "stronger"?
"Closed" ADT with classical spec implies holistic spec.
(closed: no functions can be added, all functions have classical specs, ghost state has known representation)

Why do we need holistic specs?
* "closed ADT" is sometimes too strong a requirement.
* Holistic aspect is cross-cutting (eg no payment without authorization)
* Allows reasoning in open world (eg DOM wrappers)