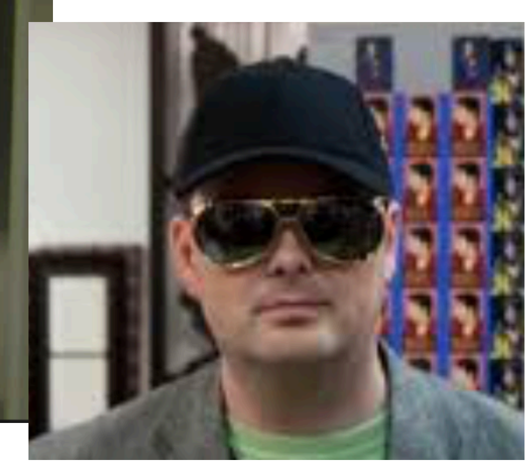
Reasoning for Open Systems

Sophia Drossopoulou, Imperial College London
work with
James Noble (VU Wellington), Mark Miller (Google),
Toby Murray (Uni Melbourne),

and also She Peng Loh and Emil Klasan (Imperial)

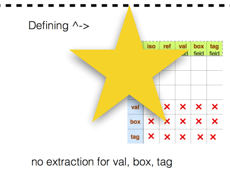


Open Systems

- Objects carry out business with other objects of unknown provenance.
  Therefore, *our* objects need to be very robust.
  To specify such robust code, classical pre- and post- condition specifications
    - not always sufficient
    - not always convenient
- New concepts for such robust specs: rather than talk about pre- and post-state we want to which *reflect* over the executions
    - invariants
    - authority (who may access)
    - permission (who may modify)
    - heap topology (domination)
    - trust (have we established that some object adheres to its spec)
    - necessary rather than sufficient conditions
    - reflect on trace of calls

	Mint &Purse	Escrow	[Grant Matcher]	DOM & Proxies	Coin & DAO
invariant	★	★	★	★	★
necessary conduitions	★	★	★	★	★
authority	★	★	★	★	Defining ^-> ★ no extraction for val, box, tag
permission	★	★	★	★	★
topology				★	
trust		★	★		
reflect on call trace					★

Today

	Mint &Purse	Escrow	[Grant Matcher]	DOM & Proxies	Coin & DAO
invariant	★	★	★	★	★
necessary conduitions	★	★	★	★	★
authority	★	★	★	★	★ 
permission	★	★	★	★	★
topology				★	
trust		★	★		
reflect on call traces					★

Today
Reasoning about Authority Attenuation

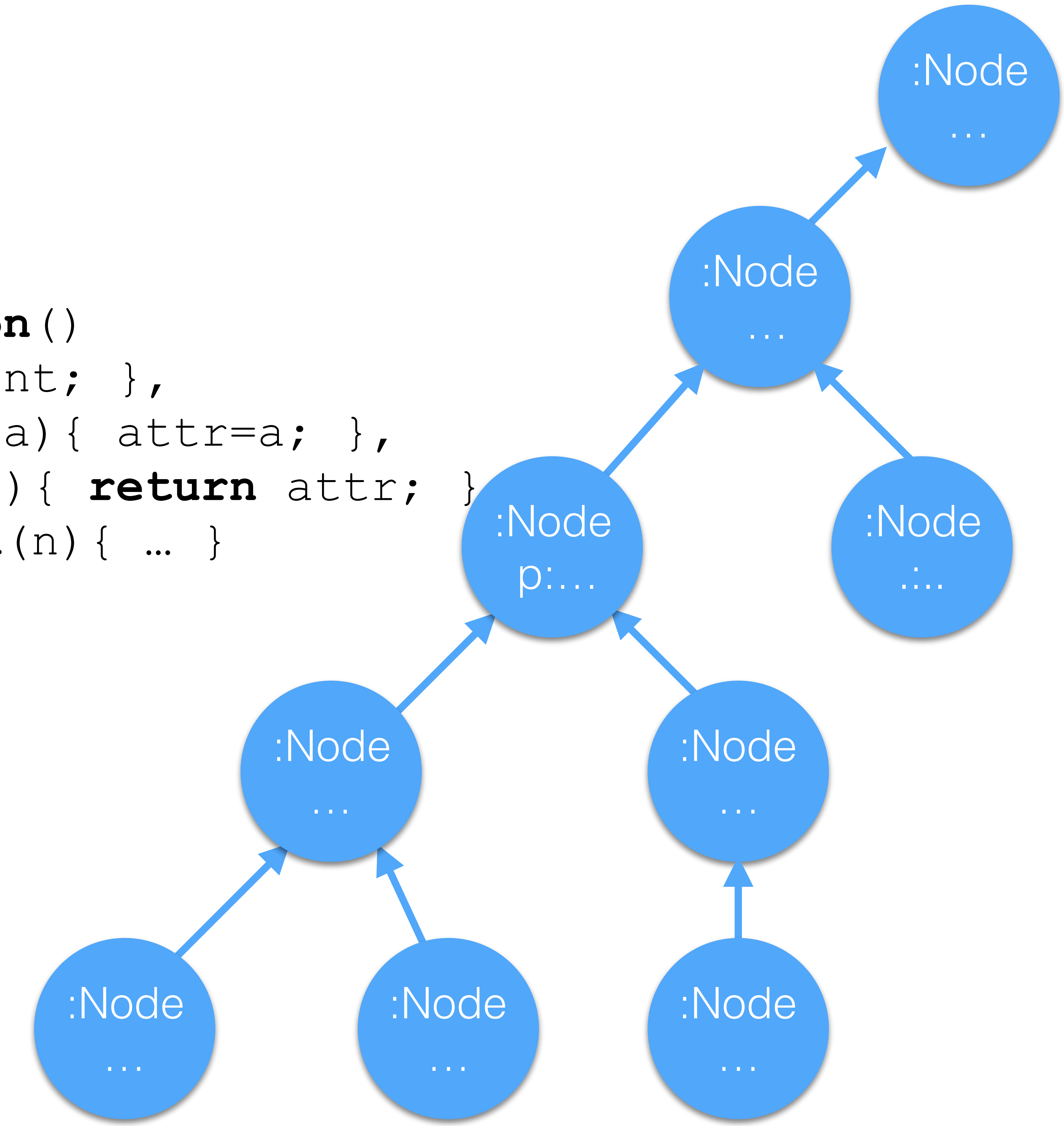
Shu Peng Loh and Sophia Drossopoulou

Proxies
this talk

- *Proxy* objects give secure access to *some* but not all capabilities of another object.
- We argue that the formal specification of attenuation requires concepts of
  - authority
  - permission and domination (graph theoretic property)
  - necessary rather than sufficient conditions
- We apply this to DOM-tree example [Devriese, Birkedahl & Piessens, Euro S&P 2016]
  - we specify proxy's access to trees
  - specification is "simple"
  - specification allows us to reason in the presence of unknown code, and of unknown provenance

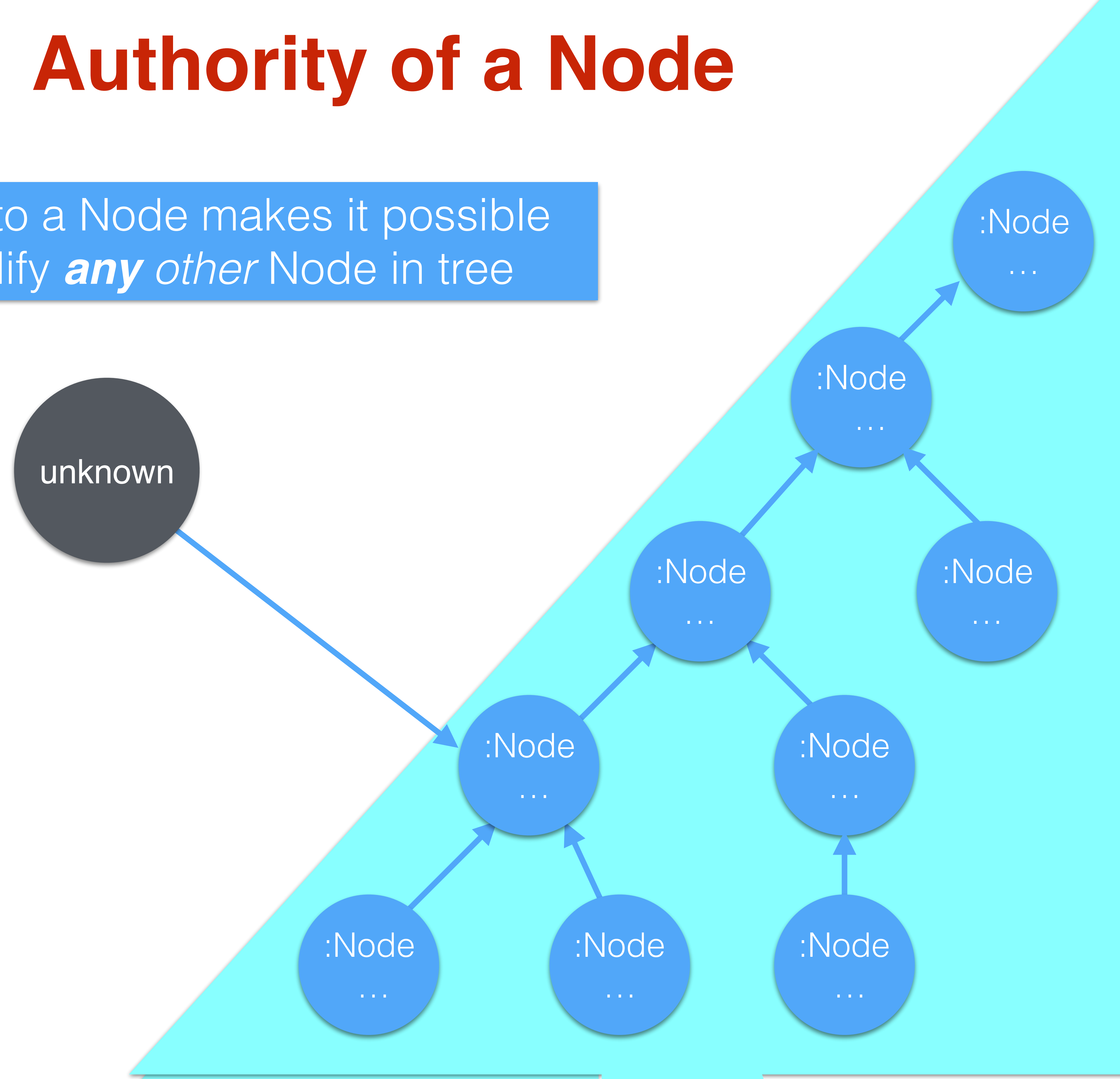
Node

```
function Node(par,a) {
  var parent = par
  var attr = a
  var children = …
  return freeze ({
       getParent: function()
            { return parent; },
       setAttr: function(a){ attr=a; },
       getAttr: function(){ return attr; }
       setChild: function(n){ … }
  })
}
```



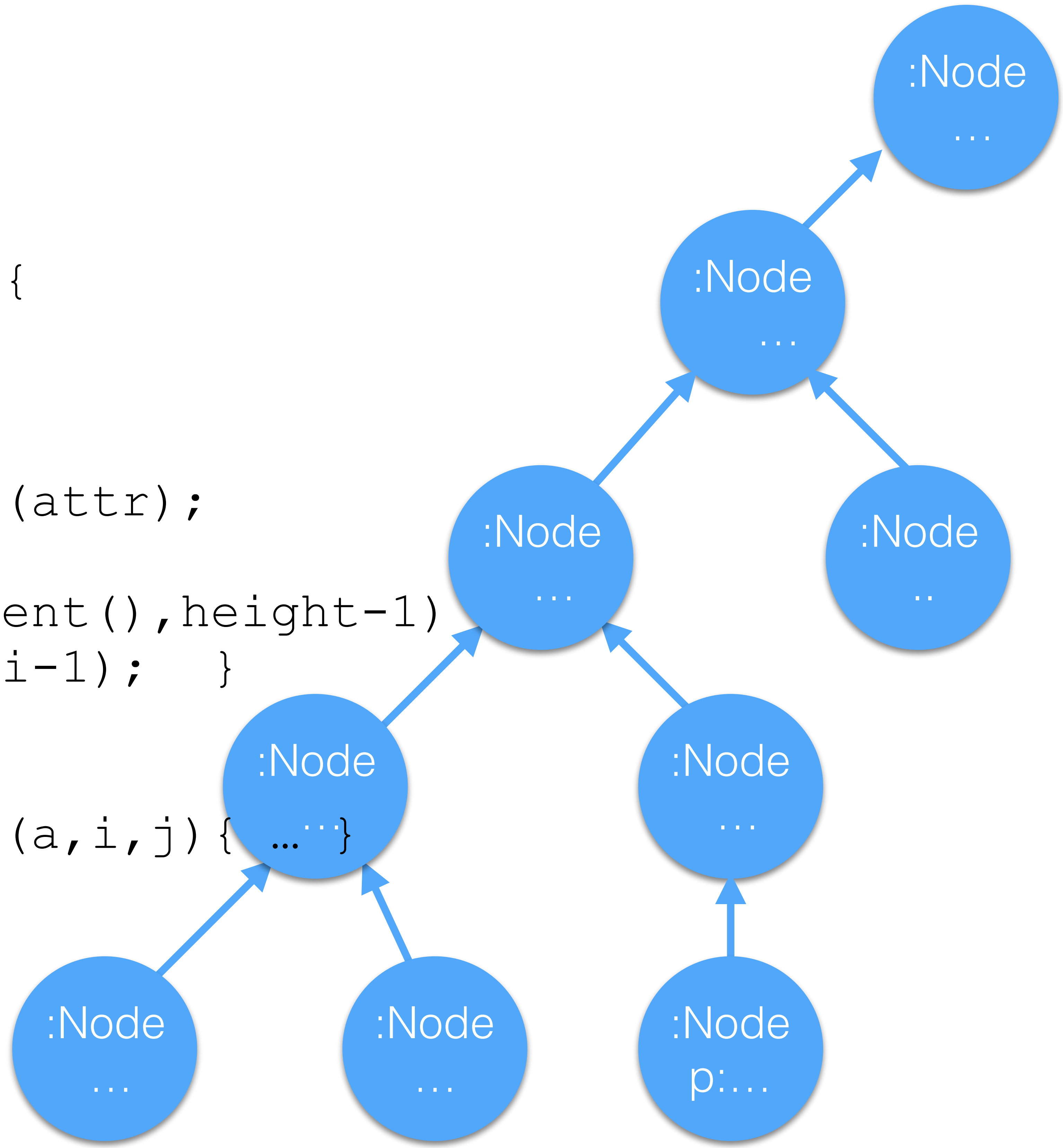
Authority of a Node

Access to a Node makes it possible to modify ***any*** *other* Node in tree



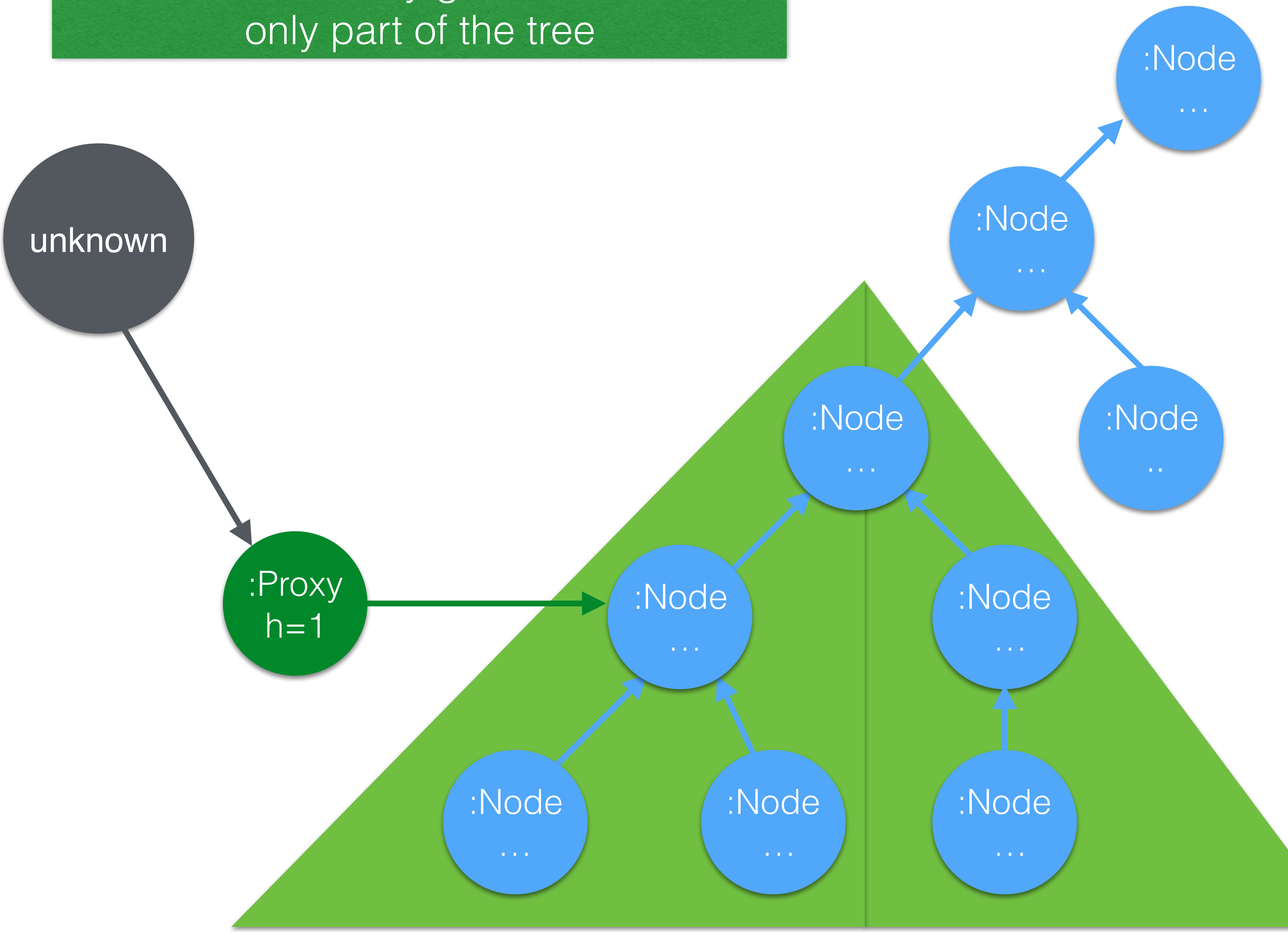
Proxy

```
function Proxy(nd,h) {
  var node = nd
  var height = h
  return
    freeze ( {
    setAttr: function(a,i){
       if (height<i){
          return; };
       else if ( i==0 ){
             node.setAttr(attr);
      } else {
          Proxy(nd.getParent(),height-1)
              .setAttr(a,i-1);   }
            }
    },
   setChildAttr: function(a,i,j){ ... }
  } )
}
```



Authority of a Proxy

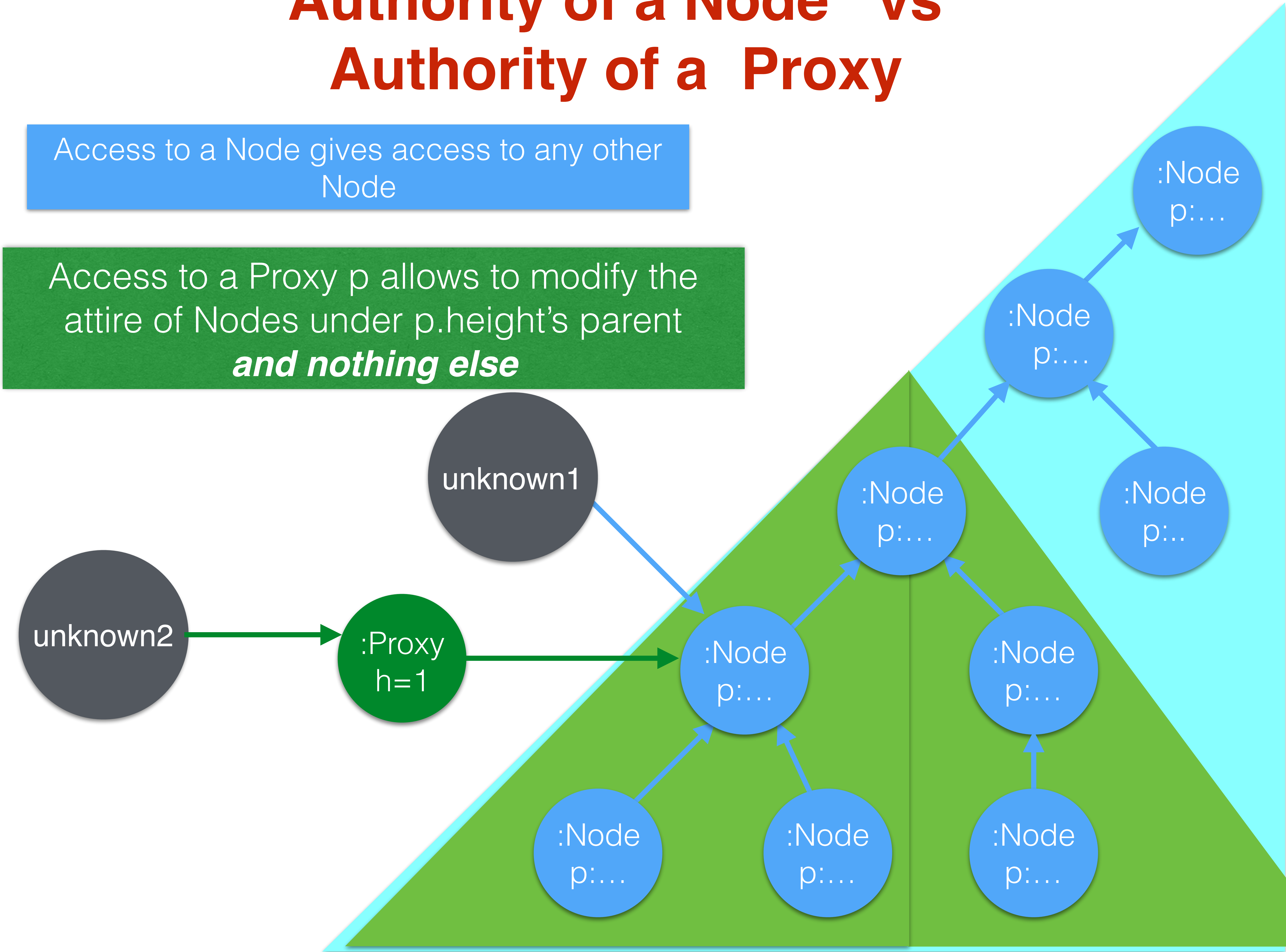
Acces to a Proxy gives access to only part of the tree



Authority of a Node vs Authority of a Proxy

Access to a Node gives access to any other Node

Access to a Proxy p allows to modify the attire of Nodes under p.height's parent
and nothing else



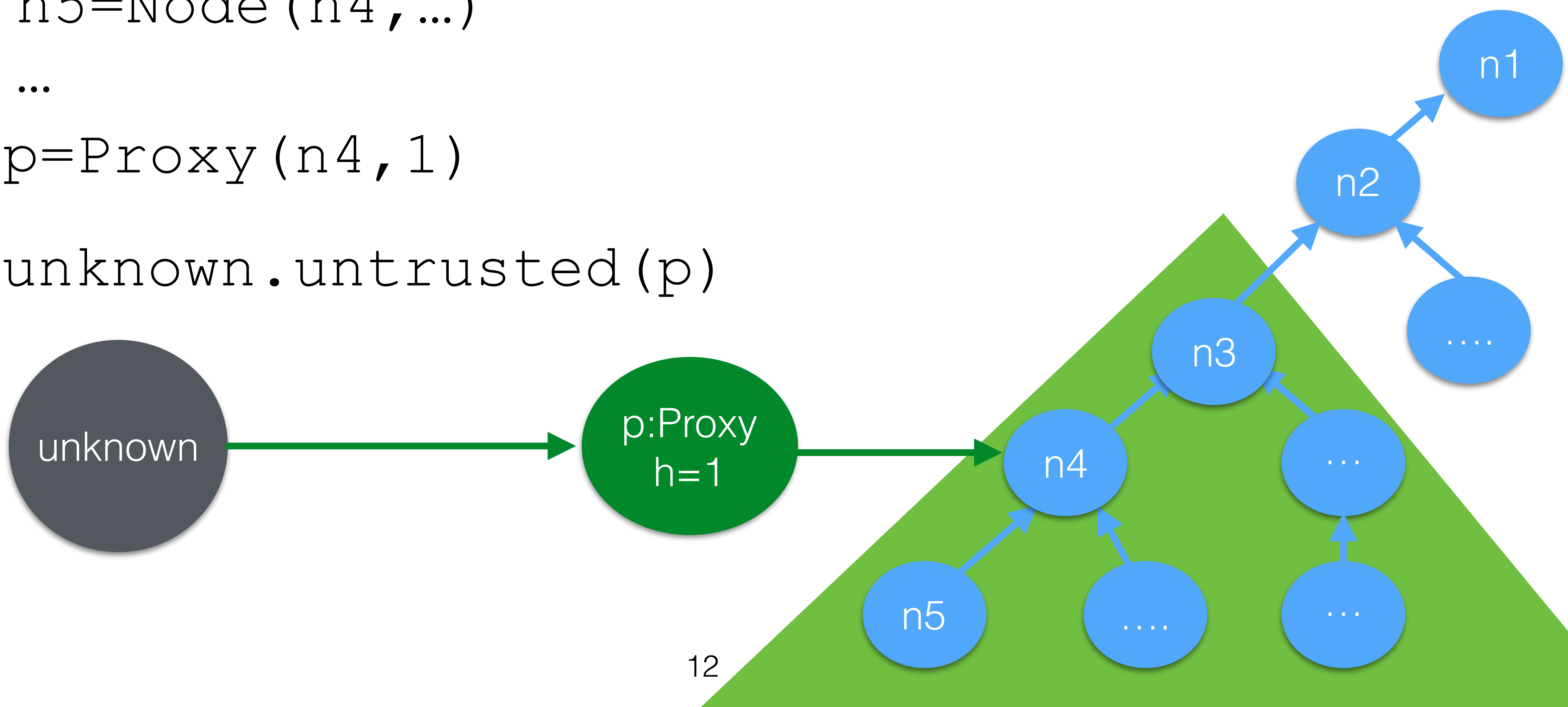
Today's aim

```
function mm(o){
    n1=Node(…)
    n2=Node(n1,…)
    n3=Node(n2,…)
    n4=Node(n3,…)
    n5=Node(n4,…)
    …
   p=Proxy(n4,1)

   unknown.untrusted(p)
```

This code leaves `n1, n2` unaffected!

How to show, even though we know nothing about `unknown` and `untrusted`?



Specifying Node/Proxy

the “conventional” part

We describe the effect of calls on methods on `Node` and on `Proxy`

Specifying Node/Proxy

the “conventional” part


```
nd:Node  { n.setAttr(x) }  nd.attr==x
```

Specifying Proxy- 1

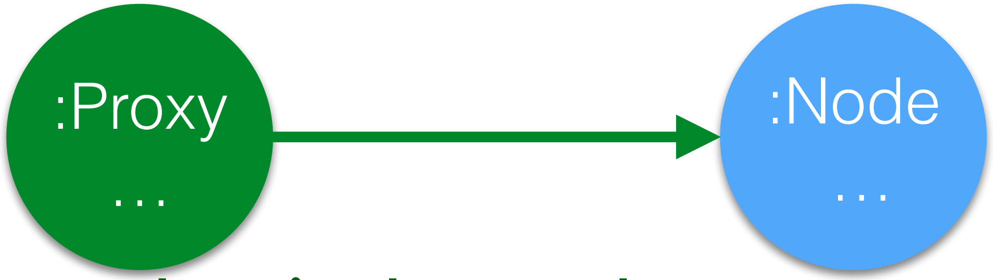
the “conventional” part


```
p:Proxy ∧  p.height==k
  { any_code }
p.height==k
```

Note: This is an *invariant.*

Specifying Proxy - 2

the “conventional” part




```
p:Proxy ∧  p.node==nd ∧ p.height>=k
   { p.setAttr(a,k) }
```

$nd.parent^k.attr == a$

Note: We are describing *sufficient* conditions.

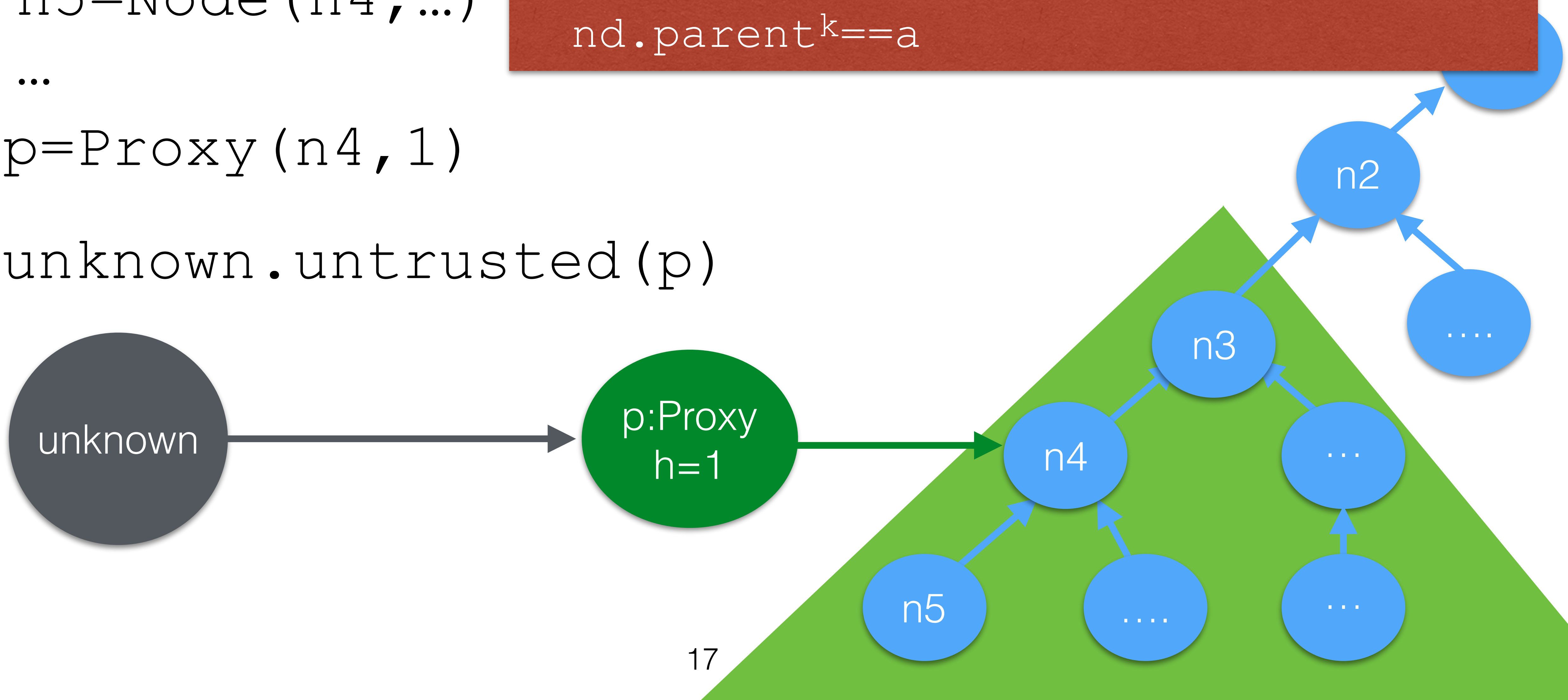
"Conventional" spec does not do

```
function mm(o){
    n1=Node(…)
    n2=Node(n1,…)
    n3=Node(n2,…)
    n4=Node(n3,…)
    n5=Node(n4,…)
    …
   p=Proxy(n4,1)

   unknown.untrusted(p)
```

```
nd:Node
{ nd.setAttr(a) }
nd.attr==x
```

```
p:Proxy ∧  p.node==nd ∧ p.height>=k
{ p.setAttr(a,k) }
nd.parent^k==a
```



Specifying Node/Proxy

the “unconventional” part

```
x,y objects of unknown provenance
  { x.m(y) }
which part of DOM unaffected?
```

We will be describing *necessary* conditions.

We need new concepts for *affecting* and *accessing*.

Specifying Proxy

the “unconventional” part - 2

Concepts for *affecting* and *accessing*.

Under what circumstances may a Proxy be accessed?

Under what circumstances may a Node be modified?

In order to specify Proxy we need some new predicates

Affecting and Accessing

new concepts

WillAffect(o,o') expresses that
at some future point in time,
object o will cause change of state in object o'

Definition

$M, \sigma \vDash$ *WillAffect*(o,o') iff
$\exists\ \sigma' \in Reach(M,\sigma).$
$[\ \sigma'(\mathtt{this}) = o\ \wedge$
$\exists\ \sigma'' \in Reach(M,\sigma').\ \exists f.\ \sigma''(o'.f) \neq \sigma'(o'.f)\ ]$

Reach(M,σ): intermediate configurations reachable from σ.

Affecting and Accessing - 2

new concepts

WillCall(o,o') expresses that

at some future point in time,

object o will (indirectly) call a method on object o'

Definition

$M, \sigma \models$ *WillCall*(o,o') iff

$\exists\ \sigma' \in Reach(M,\sigma).$

$[\ \ \sigma'(this) = o\ \wedge$

$\exists\ \sigma'' \in Reach(M,\sigma').\ \sigma''(this) = o'\ ]$

Reach(M,σ): intermediate configurations reachable from σ.

Affecting and **Accessing** - 3

new concepts

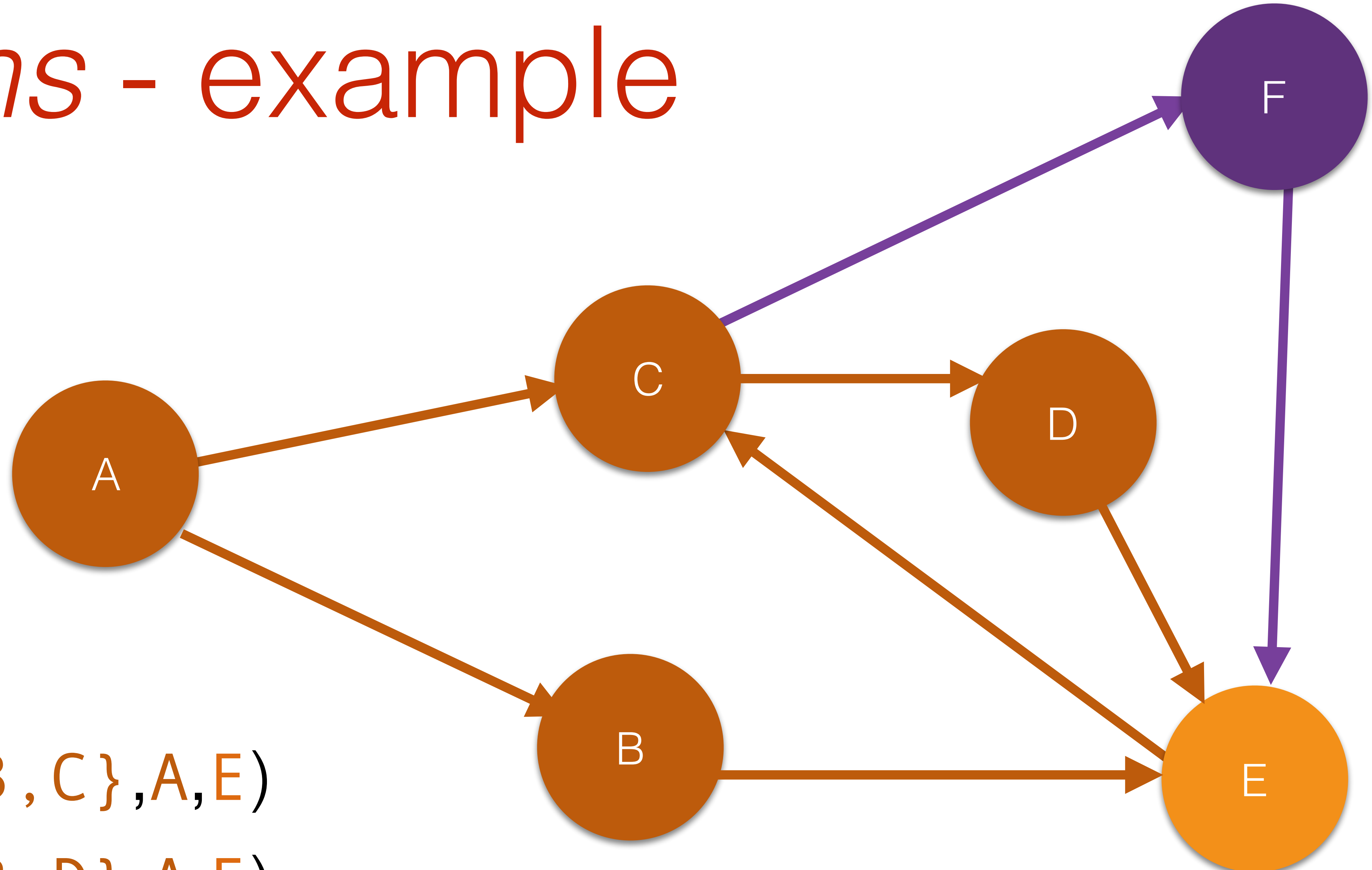
Doms(S,o,o') expresses that
any path which leads from object o to object o'
goes through some object in the set S

Definition

$M, \sigma \models$ *Doms*(S,o,o') iff

$\forall f_1, \ldots f_n. [\ \sigma(o.f_1.\ldots f_n) = o' \ \rightarrow\ \exists k.\ \sigma(o.f_1.\ldots f_k) \in S$

Doms - example



$Doms(\{B, C\}, A, E)$

$Doms(\{B, D\}, A, E)$

$\neg Doms(\{B, D\}, A, E)$

Definition

$M, \sigma \vDash \; Doms(S, o, o') \quad$ iff

$$\forall\, f_1, \ldots f_n.\ [\ \sigma(o.f_1 \ldots f_n) = o' \;\rightarrow\; \exists\, k.\ \sigma(o.f_1 \ldots f_k) \in S$$

Having introduced the new predicates, we return to the specification of some general, language, properties, and the specification of `Node`, and `Proxy`.

`Node` is encapsulated

$\forall$ `nd`:`Node`,`o`:`Object`.

$[\ \ \mathit{WillAffect}(\mathtt{o},\mathtt{nd}) \rightarrow \mathit{WillCall}(\mathtt{o},\mathtt{nd})\ \ ]$

Note: This is a *necessary* condition.

Calls through dominators

$$\forall\, o, o':\texttt{Object}.\;\; [\;\; \mathit{WillCall}(o,o') \land \mathit{Doms}(S,o,o') \rightarrow \exists o''\in S\,.\; \mathit{WillCall}(o,o'') \land \mathit{WillCall}(o'',o') \;\;]$$

Note: This is another *necessary* condition.

Specifying Proxy Calls

- $\forall$ p:Proxy. $\forall$ nd:Node.

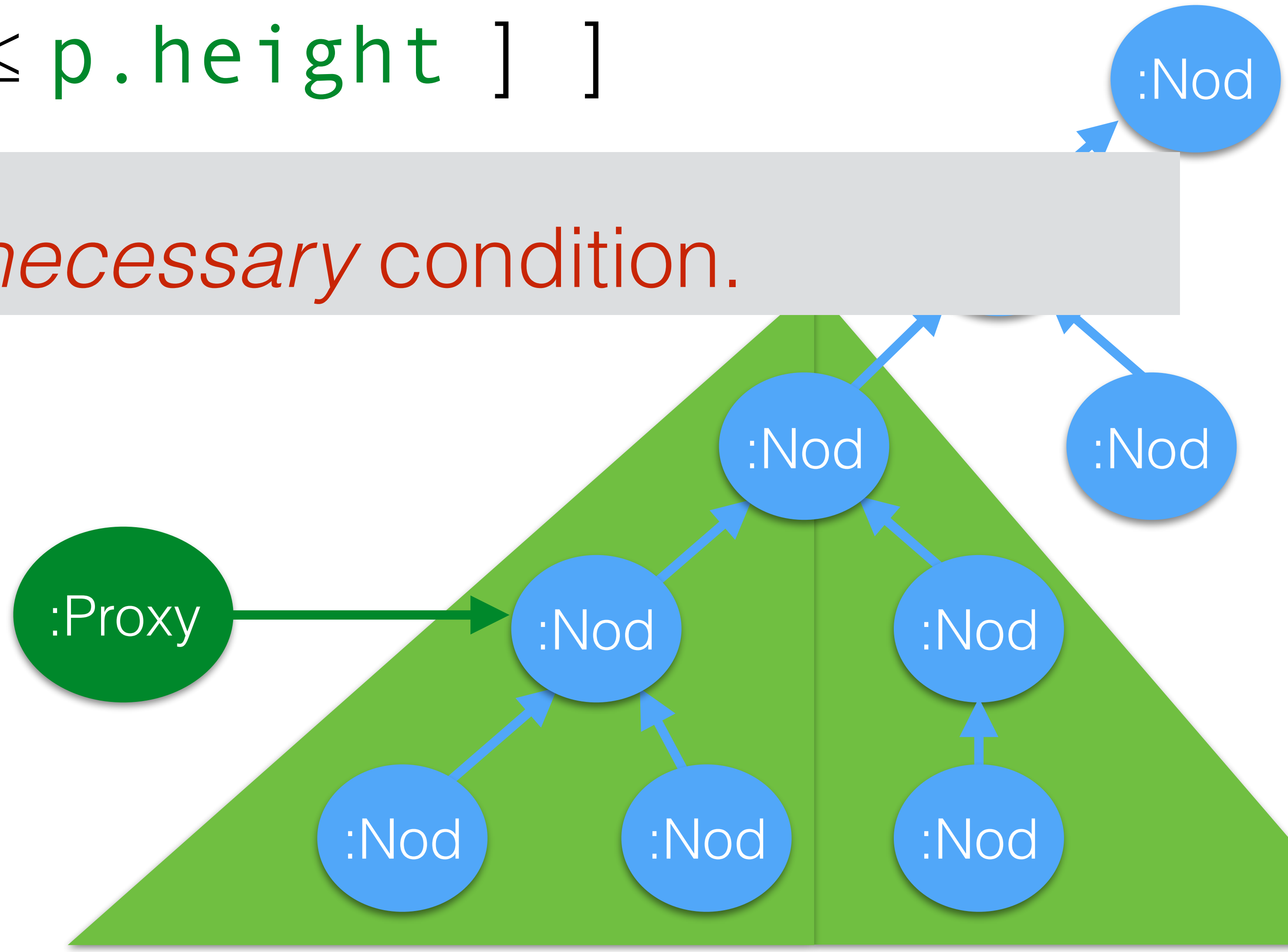
  $[\ \mathit{WillCall}(p,nd) \rightarrow$

  $\exists\ j,k.[\ nd.parent^{j} = p.node.parent^{k}$

  $\wedge\ k \leq p.height\ ]\ ]$

Note: This is another *necessary* condition.

A proxy may modify the properties of all descendants of the height-th parent of the Node it points to



Specifying Proxy no Leaks

$$\texttt{o1,o2:Object} \wedge \texttt{p:Proxy} \wedge \texttt{nd:Node} \wedge$$

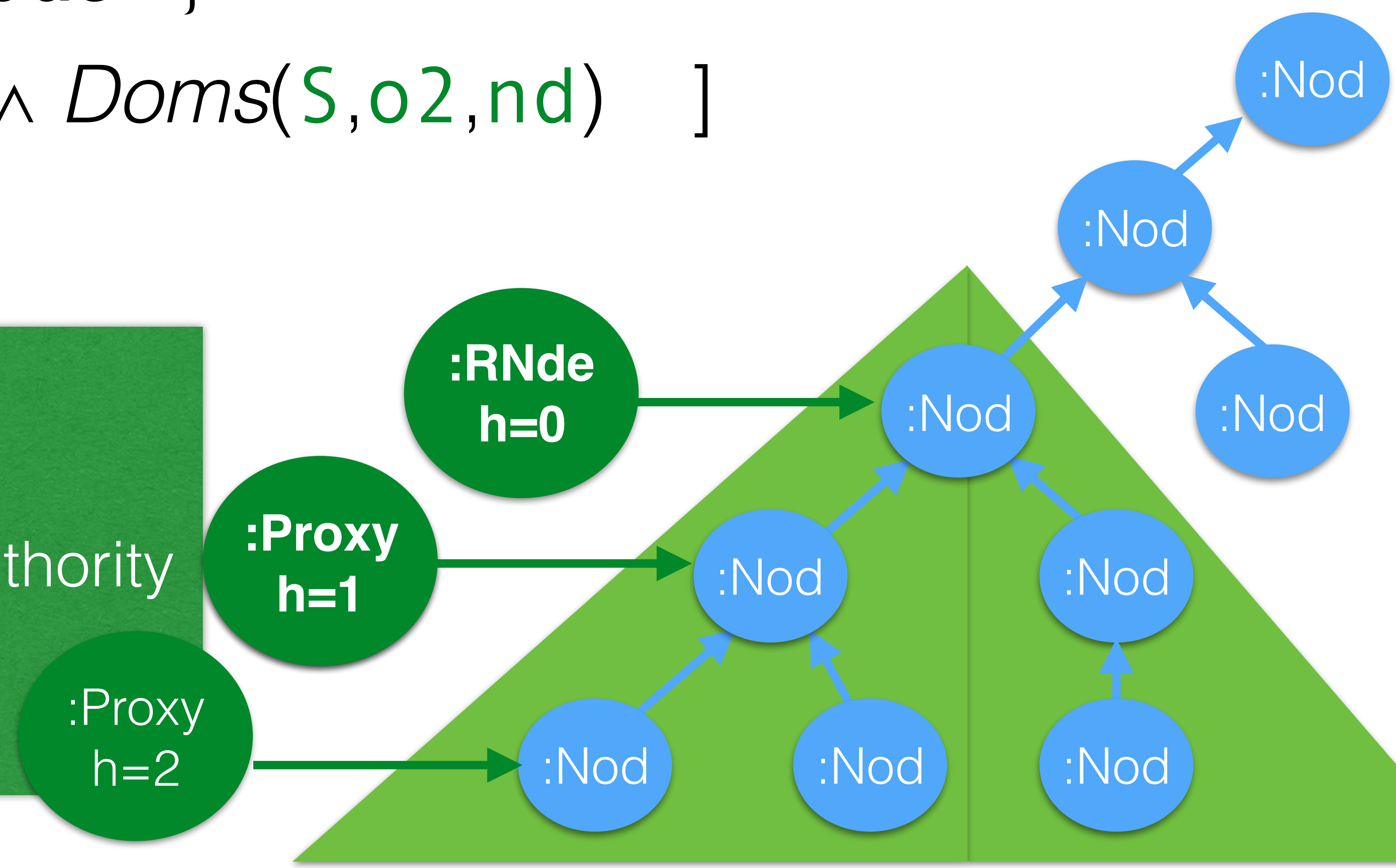
$$\texttt{S} \subseteq \texttt{Proxy} \wedge \mathit{Doms}(\texttt{S,o1,n}) \wedge \mathit{Doms}(\texttt{S,o2,n}) \wedge$$

$$\mathit{Vars}(\texttt{any\_code}) \subseteq \{ \texttt{o1, o2} \}$$

$$\{ \texttt{any\_code} \}$$

$$[\; \mathit{Doms}(\texttt{S,o1,nd}) \wedge \mathit{Doms}(\texttt{S,o2,nd}) \;]$$

Proxies do not leak Authority



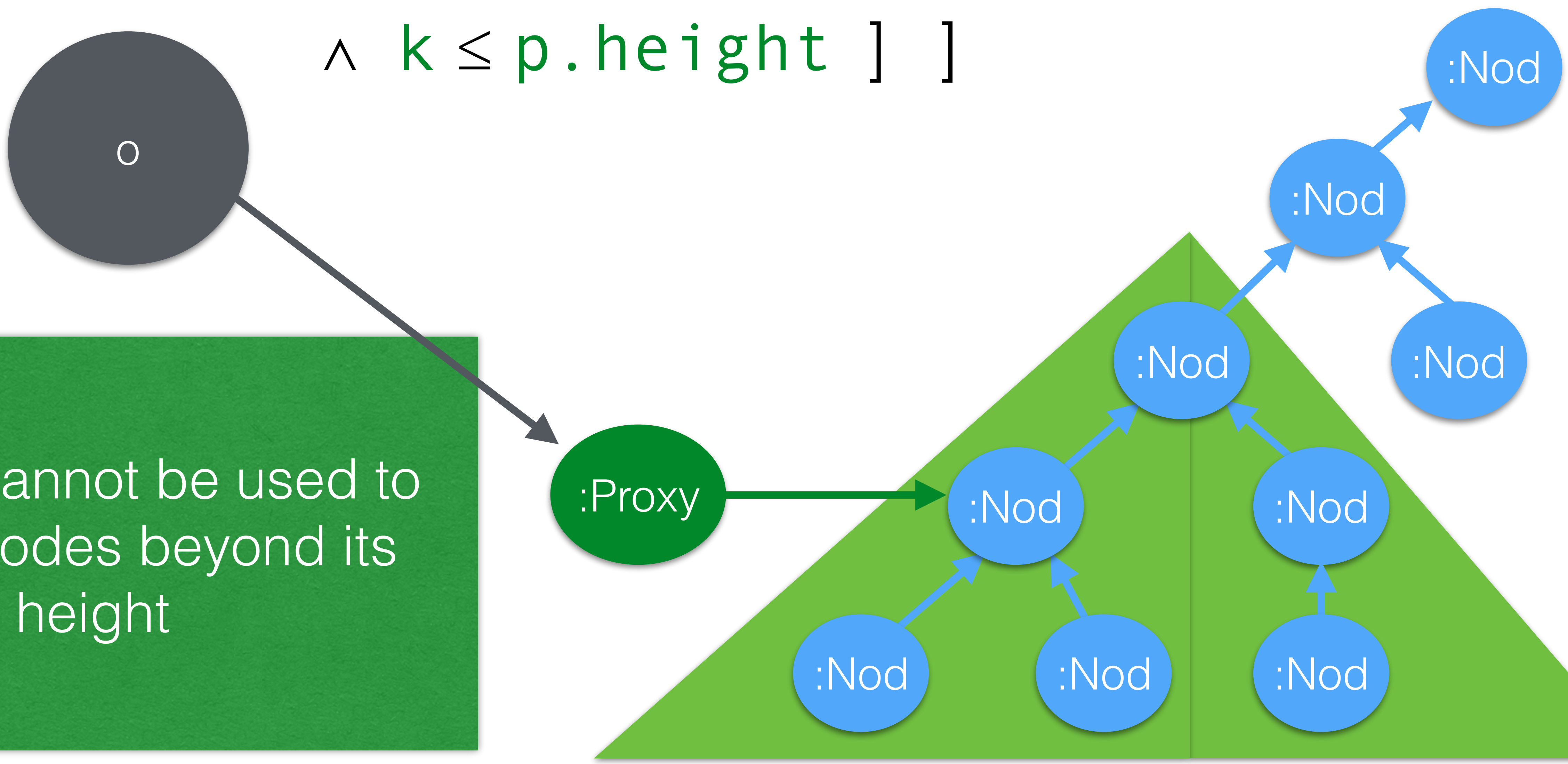
Consequence of previous

$$\forall\, \mathtt{o}{:}\mathtt{Object}.\ \forall\, \mathtt{p}{:}\mathtt{Proxy}.\ \forall\, \mathtt{nd}{:}\mathtt{Node}.$$

$$[\ \mathit{Doms}(\{\mathtt{p}\},\mathtt{o},\mathtt{nd}) \wedge \mathit{WillAffect}(\mathtt{o},\mathtt{nd})\ \rightarrow$$

$$\exists\, \mathtt{j},\mathtt{k}.[\ \mathtt{nd.parent}^{\mathtt{j}} = \mathtt{p.node.parent}^{\mathtt{k}}$$

$$\wedge\ \mathtt{k} \leq \mathtt{p.height}\ ]\ ]$$



A proxy cannot be used to modify nodes beyond its height

Putting these specs to work

,

`unknown` object of unknown provenance

`untrusted` is some arbitrary method

```
p:Proxy
  { unknown.untrusted(p) }
```

which part of DOM unaffected?

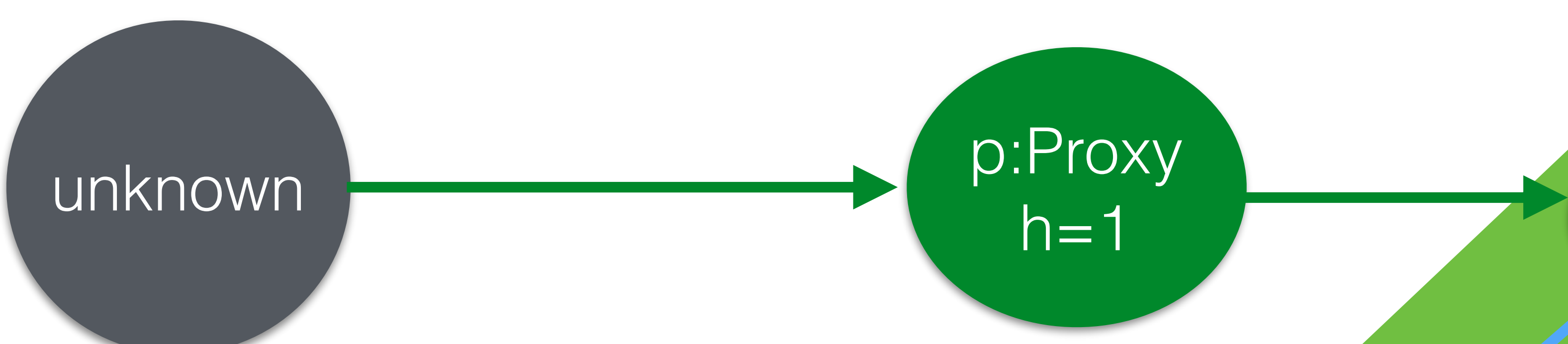
Putting these specs to work

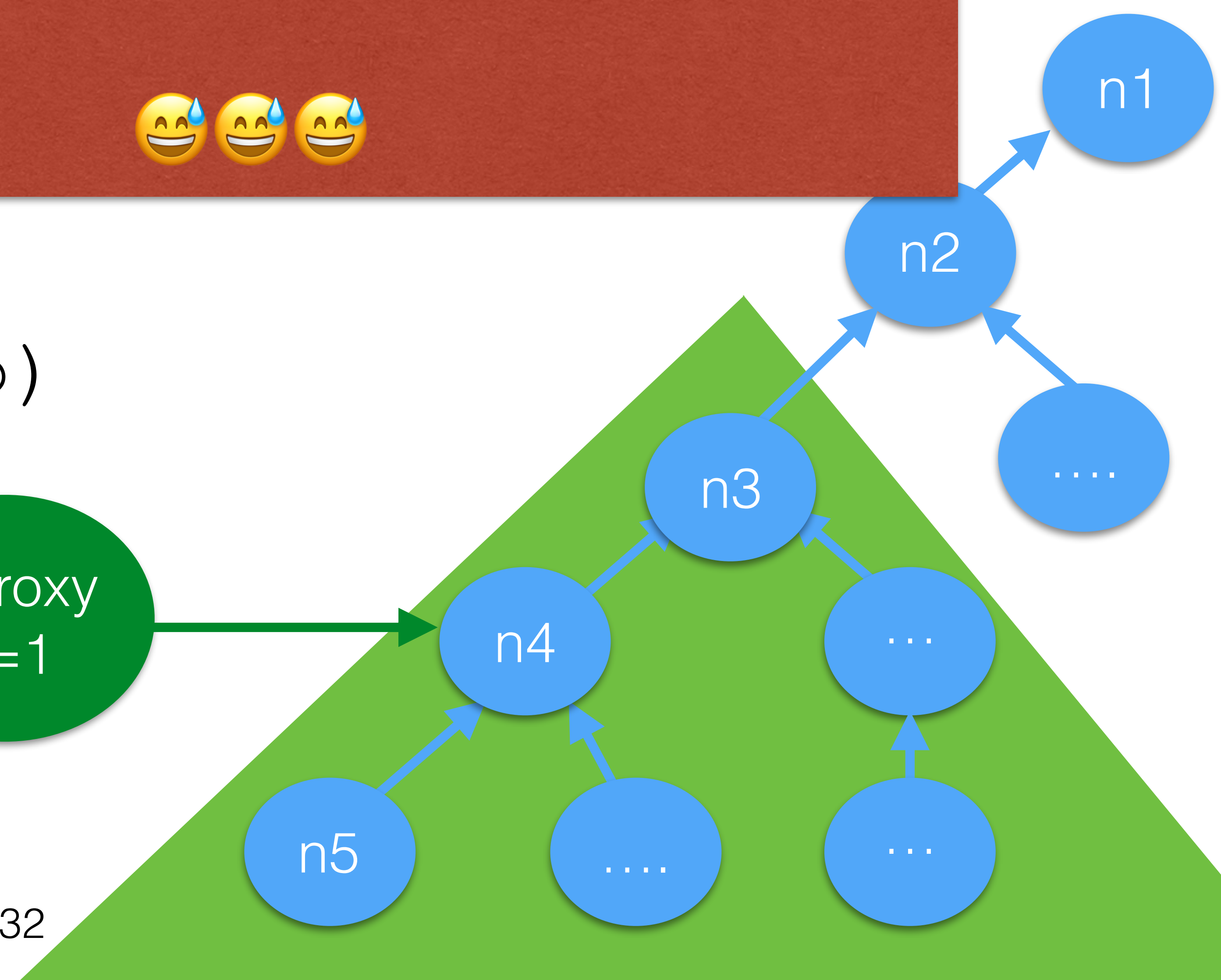
```
function mm(c
    n1=Node(…)
    n2=Node(n1
    n3=Node(n2
    n4=Node(n3
    n5=Node(n4
    …
   p=Proxy(n4,1)

   unknown.untrusted(p)
```

Using the specifications from above,
and even though we know nothing
about `unknown` and `untrusted`,
we can prove that
the above leaves n1 and n2 unaffected!

😅😅😅





Why is this a *holistic* spec?

```
function ProxyLeak(nd,h) {
  var node = nd
  var height = h
  return
    freeze ( {
      // as earlier
      setAttr: function(a,i){    …     },
      // as earlier
     setChildAttr: function(a,i,j){ … }
     // new
     leak: function( ){ return node.parent }
  } )
}
```

Why is this a *holistic* spec? - 2
ProxyLeak does *not* satisfy spec below

```
function ProxyLeak(nd,h) {
  ..
  return
    freeze ( {
      …        // new
      leak: function( ){ return node.parent }
  } )
}
```

o1,o2:Object ∧ p:ProxyLeak ∧ nd:Node ∧

S⊆Proxy ∧ *Doms*(S,o1,n) ∧ *Doms*(S,o2,n) ∧

Vars(any_code) ⊆ { o1, o2 }

{ any_code }

[*Doms*(S,o1,nd) ∧ *Doms*(S,o2,nd)]

Summary

- We defined
  - *WillAffect*, *WillCall* (reflect over execution)
  - *Doms* (reflect over state)
- For the DOM-tree example [Devriese at al Euro S&P 2016]
  - specification is “simple”
  - specification allows us to reason in the presence of code of unknown provenance
  - using necessary as well as sufficient conditions
- Similar style appeared in more examples