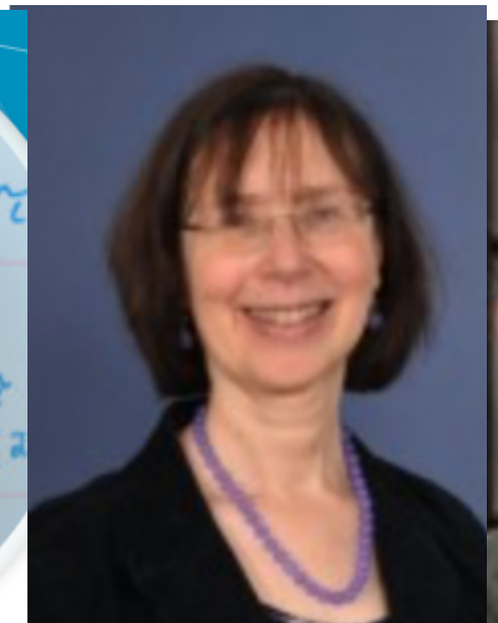
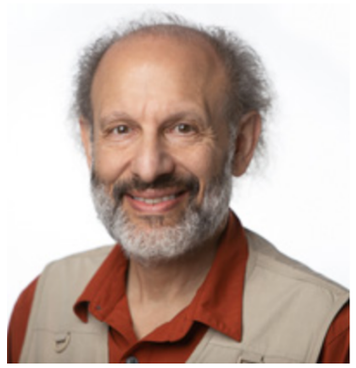
Holistic Specifications for Robust Code

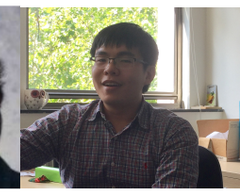
Julian Mackay (VU Wellington, New Zealand)
Susan Eisenbach (Imperial College),
James Noble (Creative Research & Programming, New Zealand)
Sophia Drossopoulou (Meta and Imperial College)

and
Toby Murray (Univ Wellington), Mark Miller (Agoric)

and
Matthew Rather and Shupeng Loh and Emil Klasan (Imperial)



Holistic Specifications for Robust Code

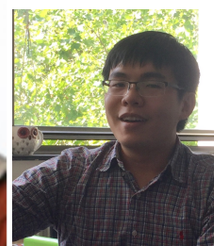
with support from

- New Zealand Marsden grants
- Ethereum Foundation
- Facebook
- Agoric
- copious “free time”









Holistic Specification of Code

need new encapsulation features

Ownership types?

… WHAT should these features guarantee?

Encapsulation features are *mechanisms for* robustness, but what are the guarantees?

Today

- Functional Specifications do not adequately address Robustness
- What do we propose?
- What does it mean
- How do we prove?
- Is it sound?

Today

- **Functional Specifications do not adequately address Robustness**

- What do we propose?
- What does it mean
- How do we prove?
- Is it sound?

Functional Specs

- *closed* world
- *sufficient* conditions for some action/effect
- talk of *individual* function

Robustness considerations

- *open* world
- *necessary* conditions for some action/effect
- *emergent* behaviour

```
module Mod1
  class Account
    field balance:int
    field pwd: Password
    method transfer(dest:Acc
      if this.pwd==pwd'
        this.balance-=100
        dest.balance+=100
  class Password
```

```
module Mod2
  class Account
    field balance:int
    field pwd: Password
    method transfer(..)
      ... as earlier ...
    method set(pwd': Object)
      this.pwd=pwd'

  class Password
```

```
module Mod3
  class Account
    field balance:int
    field pwd: Password
    method transfer(..)
      ... as earlier ...
    method set(pwd',pwd'': Object)
      if (this.pwd==pwd')
        this.pwd=pwd''
  class Password
```

SpecA: { a,a':Account ∧ a.pwd = pwd' }
a.transfer(a',pwd')
{ $a.balance_{post} = a.balance_{pre} - 100 \wedge a'.balance_{post} = a'.balance_{pre} + 100$ }

{ a,a':Account ∧ a.pwd ≠ pwd' }
a.transfer(a',pwd')
{ $a.balance_{post} = a.balance_{pre} \wedge a'.balance_{post} = a'.balance_{pre}$ } }

traditional

traditional

SpecB: ∀a:Account.[$a.password_{pre} = a.password_{post}$]

Mod1 ⊨ SpecA
Mod2 ⊨ SpecA
Mod3 ⊨ SpecA

Mod1 ⊨ SpecB
Mod2 ⊭ SpecB
Mod3 ⊭ SpecB

traditional specs do *not* express robustness!

Today

- Functional Specifications do not adequately address Robustness
- **What do we propose?**
  - What does it mean
  - How do we prove?
  - Is it sound?

```
module Mod1
  class Account
    field balance:int
    field pwd: Password
    method transfer(dest:Acc
      if this.pwd==pwd'
        this.balance-=100
        dest.balance+=100
  class Password
```

```
module Mod2
  class Account
    field balance:int
    field pwd: Password
    method transfer(..)
      ... as earlier ...
    method set(pwd': Object)
      this.pwd=pwd'

  class Password
```

```
module Mod3
  class Account
    field balance:int
    field pwd: Password
    method transfer(..)
      ... as earlier ...
    method set(pwd',pwd'': Object)
      if (this.pwd==pwd')
        this.pwd=pwd''
  class Password
```

SpecA: { a,a':Account ∧ a.pwd = pwd' }
a.transfer(a',pwd')
{ $a.balance_{post} = a.balance_{pre} - 100 \wedge a'.balance_{post} = a'.balance_{pre} + 100$ }

{ a,a':Account ∧ a.pwd ≠ pwd' }
a.transfer(a',pwd')
{ $a.balance_{post} = a.balance_{pre} \wedge a'.balance_{post} = a'.balance_{pre}$ } }

SpecB: ∀a:Account.[$a.password_{pre} = a.password_{po}$

Holistic

Mod1 ⊨ SpecA
Mod2 ⊨ SpecA
Mod3 ⊨ SpecA

Mod1 ⊨ SpecB
Mod2 ⊭ SpecB
Mod3 ⊭ SpecB

Want a specification SpecC, which distinguishes ***robust*** modules (Mod1 and Mod3) from **non-robust** (Mod2)

Mod1 ⊨ SpecC
Mod2 ⊭ SpecC
Mod3 ⊨ SpecC

```
module Mod1
  class Account
    field balance:int
    field pwd: Password
    method transfer(dest:Acc
      if this.pwd==pwd'
        this.balance-=100
        dest.balance+=100
  class Password
```

```
module Mod2
  class Account
    field balance:int
    field pwd: Password
    method transfer(..)
      ... as earlier ...
    method set(pwd': Object)
      this.pwd=pwd'

  class Password
```

```
module Mod3
  class Account
    field balance:int
    field pwd: Password
    method transfer(..)
      ... as earlier ...
    method set(pwd',pwd'': Object)
      if (this.pwd==pwd')
        this.pwd=pwd''
  class Password
```

SpecA: { a:Account ∧ a.pwd = pwd' }
a.transfer(a',pwd')
{ $a.balance_{post} = a.balance_{pre} - 100$ ∧ $a'.balance_{post} = a'.balance_{pre} + 100$ }

{ a:Account { a:Account ∧ a':Account ∧ a.pwd =/= pwd' }
a.transfer(a',pwd')
{ $a.balance_{post} = a.balance_{pre}$ ∧ $a'.balance_{post} = a'.balance_{pre}$ } }

SpecB: ∀a:Account.[$a.pwd_{pre} = a.pwd_{post}$]

Mod1 ⊨ SpecA
Mod2 ⊨ SpecA
Mod3 ⊨ SpecA

Mod1 ⊨ SpecB
Mod2 ⊭ SpecB
Mod3 ⊭ SpecB

Unless you know the password, you cannot take money, no matter how many function calls you make

Mod1 ⊨ SpecC
Mod2 ⊭ SpecC
Mod3 ⊨ SpecC

```
module Mod1
  class Account
    field balance:int
    field pwd: Password
    method transfer(dest:Acc
      if this.pwd==pwd'
        this.balance-=100
        dest.balance+=100
  class Password
```

```
module Mod2
  class Account
    field balance:int
    field pwd: Password
    method transfer(..)
      ... as earlier ...
    method set(pwd': Object)
      this.pwd=pwd'

  class Password
```

```
module Mod3
  class Account
    field balance:int
    field pwd: Password
    method transfer(..)
      ... as earlier ...
    method set(pwd',pwd'': Object)
      if (this.pwd==pwd')
        this.pwd=pwd''
  class Password
```

SpecA: { a:Account ∧ a.pwd = pwd' }
a.transfer(a',pwd')
{ $a.balance_{post} = a.balance_{pre} - 100$ ∧ $a'.balance_{post} = a'.balance_{pre} + 100$ }

{ a:Account { a:Account ∧ a':Account ∧ a.pwd = pwd' }
a.transfer(a',pwd')
{ $a.balance_{post} = a.balance_{pre}$ ∧ $a'.balance_{post} = a'.balance_{pre}$ } }

SpecB: ∀a:Account.[$a.pwd_{pre} = a.pwd_{post}$]

Mod1 ⊨ SpecA
Mod2 ⊨ SpecA
Mod3 ⊨ SpecA

Mod1 ⊨ SpecB
Mod2 ⊭ SpecB
Mod3 ⊭ SpecB

SpecC: **from** (a:Account ∧ a.balance = bal)
to (a.balance < bal)
onlyIf (∃o. <o **external**> ∧ <o **access** a.pwd >)

Mod1 ⊨ SpecC
Mod2 ⊭ SpecC
Mod3 ⊨ SpecC

Notes about SpecC

```
SpecC:  from ( a:Account ∧ a.balance = bal )
        to   ( a.balance < bal )
        onlyIf ( ∃o. <o external> ∧ <o access a.pwd > )
```

- `SpecC` does not talk about any particular code (functional specifications do)
- `SpecC` does not talk about any particular API (eg is agnostic about the names of functions used to take money from the account)
- `SpecC` describes necessary condition for some effect to take place
- `balance` is a ghost field, and may be implemented in different ways by different modules
- Several different modules (with different APIs) may satisfy `SpecC`

How useful is SpecC?

```
SpecC:  from ( a:Account ∧ a.balance = bal )
        to   ( a.balance < bal )
        onlyIf ( ∃o. <o external> ∧ <o access a.pwd > )
```

With SpecC, I can confidently pass my account to any, potentially untrusted context:
An expected payment may or may not be made, but provided my password is not known in that context, my money will not be stolen.

```
1  module Mod1
2        ...
3      method cautious(untrusted:Object)
4          a = new Account
5          p = new Password
6          a.set(null,p)
7          ...
8          untrusted.make_payment(a)
9          ...
```

If Account satisfies SpecC,
and line 7 does not leak the password to untrusted,
then
the balance of a does not decrease by the call untrusted (line 8).

Note that the guarantee from above holds, *even* when in the context of cautious there exists external access to the password.

Holistic Specifications — examples

- ERC20
- [DAO]
- DOM attenuation
- [DAO in VerX]
- [Purse and Crowdfunder]
- Escrow

Today

- Functional Specifications do not adequately address Robustness
- What do we propose?
- **What does it mean?**
- How do we prove?
- Is it sound?

Assertions and Holistic Specifications

e ::= this | x | e.fld | …

A ::= e>e | e=e | …

| A → A | A ∧ A | ∃x. A | …

| **<** x **access** y **>** permission

| < x **calls** y.m(,z1,..zn) > control

| < x **external** > viewpoint

S ::= A Invariant

| **from** A **to** A **onlyIf** A Multi-step nec. precondition

| **from** A **next** A **onlyIf** A Single-step nec. Precondition

| **from** A **next** A **onlyThrough** A Intermediate condition

Preliminaries

We define in a“conventional” way:

module	M : Ident —-> ClassDef ∪ PredicateDef ∪ FunctionDef
configuration	σ : Heap × Stack × Code
execution	M, σ ⇝ σ’

Semantics of Assertions

"Unconventional part"

A ::= … | < x **access** y> | < **external** x > | < x c**alls** y.m(z_1,..z_n) >

- M,σ ⊨ < x **access** y> iff
  - x has a field pointing to y, or
  - x is the receiver and y and argument, in one frame in σ
- M,σ ⊨ < **external** x> iff
  - The class of x is not from **M**
- M,σ ⊨ < x **calls** y.m(z_1,..z_n) > iff
  - x is the top receiver in σ, and
  - the continuation in σ starts with the call y.m(z_1,..z_n)

Semantics of Specifications

M ⊨ **from** A_{orig} **to** A_{fin} **onlyIf** A_{nec} iff

For all M_{ext}, σ, σ':
If
- σ ∈ Arising(M_{ext};M)
- M,σ ⊨ A_{orig}
- M_{ext} ; M, σ ⇝* σ'
- M,σ' ⊨ A_{fin}

then
- M,σ ⊨ A_{nec}

If σ … ⇝* … σ' then σ … ⇝* … σ'
⊨ $A_{orig.}$ ⊨ A_{fin} ⊨ $A_{orig} \wedge \mathbf{A_{nec}}$ ⊨ A_{fin}

Semantics of Specifications

M ⊨ **from** A_{orig} **to** A_{fin} **onlyIf** A_{nec} iff

For all M_{ext}, σ, σ':
If

- σ ∈ Arising(M_{ext};M)
- M,σ ⊨ A_{orig}
- M_{ext}; M, σ ↝* σ'
- M,σ' ⊨ A_{fin}

then

- M,σ ⊨ A_{nec}

External module; *open* world

Semantics of Specifications

M ⊨ **from** A_{orig} **to** A_{fin} **onlyIf** A_{nec} iff

For all M_{ext}, σ, σ':
If

- σ ∈ Arising(M_{ext};M)
- M,σ ⊨ A_{orig}
- M_{ext} ; M, σ ↝* σ'
- M,σ' ⊨ A_{fin}

then

- M,σ ⊨ A_{nec}

External states execution:
Definitions from the union of Only states with external receivers are observable

Semantics of Specifications

M ⊨ **from** A_{orig} **to** A_{fin} **onlyIf** A_{nec} iff

For all M_{ext}, σ, σ':
If

- σ ∈ Arising(M_{ext};M)
- M,σ ⊨ A_{orig}
- M_{ext} ; M, σ ↝* σ'
- M,σ' ⊨ A_{fin}

then

- M,σ ⊨ A_{nec}

Only consider states that arise from execution of M and M_{ext}, starting from initial state

Semantics of Specifications - 2

$M \models$ **from** A_{orig} **next** A_{fin} **onlyIf** A_{nec} iff

For all M_{ext}, σ, σ':
If

- $\sigma \in$ Arising(M_{ext};M)
- $M,\sigma \models A_{orig}$
- M_{ext} ; M, $\sigma \rightsquigarrow \sigma'$
- $M,\sigma' \models A_{fin}$

then

- $M,\sigma \models A_{nec}$

If $\sigma \rightsquigarrow \sigma'$ then $\sigma \rightsquigarrow \sigma'$

$\models A_{orig.}$ $\models A_{fin}$ $\models A_{orig} \wedge \mathbf{A_{nec}}$ $\models A_{fin}$

Semantics of Specifications - 3

$M \models$ **from** A_{orig} **to** A_{fin} **onlyThrough** A_{intm} iff

For all M_{ext}, σ_1, σ_2 … , σ_n

If

- $\sigma_1 \in$ Arising(M_{ext};M)
- $M,\sigma_1 \models A_{orig}$
- M_{ext} ; M, $\sigma_1 \rightsquigarrow \sigma_2 \rightsquigarrow$. … σ_n
- $M, \sigma_n \models A_{fin}$

then, there exists k

- $M, \sigma_k \models A_{nec}$

If σ_1 … $\rightsquigarrow^*$ … σ_n $\models A_{orig.}$ $\models A_{fin}$ then σ_1 … $\rightsquigarrow^* \sigma_k$ … $\rightsquigarrow^*$ … σ_n $\models A_{orig}$ $\models \mathbf{A_{nec}}$ $\models A_{fin}$

Summary of our Proposal

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

| **<** x **access** y **>** — permission

| < x **calls** y.m(,z1,..zn) > — control

| < **external** x > — viewpoint

S ::= A — Invariant

| **from** A **to** A **onlyIf** A — Multi-step nec. precondition

| **from** A **next** A **onlyIf** A — Single-step nec. Precondition

| **from** A **next** A **onlyThrough** A — Intermediate condition

M, σ ⊨ A

Arising(M ; M')

M ⊨ S

Today

- Functional Specifications do not adequately address Robustness
- What do we propose?
- What does it mean?
- **How do we prove?**
- Is it sound?

Proving Adherence to Holistic Specifications

1st Idea: **from** A*orig* **to** A*fin* **onlyIf** A*nec*

is equivalent with

∀ `stmts`. { A*orig* ∧ ¬ A*nec* } `stmts` { ¬ A*fin* }

2nd Idea: `M` ⊢ **from** A*orig* ∧ x:C ∧ <_ **calls** x.m(y1,..yn)> **next** A*fin* **onlyIf** A*nec*

can be proven using the functional spec of `C::m(_.._)`

3rd Idea: `M` ⊢ **from** A*orig* **next** A*fin* **onlyIf** A*nec*

can be proven using 2nd idea, and provided that
A*fin* is encapsulated by M, and that for all m ∈ M

`M` ⊢**from** A*orig* ∧ x:C ∧ <_ **calls** x.m(y1,..yn)> **next** A*fin* **onlyIf** A*nec*

Def. An assertion A is *encapsulated* in module M, if it can only be invalidated by calling methods from M

4th Idea: We can combine results from previous steps to obtain module's *emergent* behaviour.

2nd Idea: Per-method necessary conditions

$$\frac{M \vdash \{x : C \wedge P_1 \wedge \neg P\}\ \texttt{res} = x.m(\overline{z})\ \{\neg P_2\}}{M \vdash \texttt{from}\ P_1 \wedge x : C \wedge \langle \_\ \texttt{calls}\ x.m(\overline{z})\rangle\ \texttt{next}\ P_2\ \texttt{onlyIf}\ P}$$

For example,

Mod2 |- **from** a:Account ∧ a.balance=b ∧ <_ **calls** a.transfer(a',pwd') >
next a.balance<b
onlyIf a.pwd=pwd'

And also,

Mod2 |- **from** a:Account ∧ a=/=a' ∧ a.balance=b ∧ <_ **calls** a.transfer(a',pwd') >
next a.balance=b
onlyIf a.pwd=/=pwd'

Assertion Encapsulation

Definition 4.1 (Assertion Encapsulation). An assertion A' is *encapsulated* by module M and assertion A, written as $M \models A \Rightarrow Enc(A')$, if and only if for all external modules M', and all states σ, σ' such that $Arising(M, M', \sigma)$:

$$\left.\begin{array}{l} - M'; M,\ \sigma \rightsquigarrow \sigma' \\ - M, \sigma' \triangleleft \sigma \models \neg A' \\ - M, \sigma \models A \wedge A' \end{array}\right\} \Rightarrow \quad \exists x,\ m,\ \overline{z}.(\ M, \sigma \models \langle \_ \texttt{ calls } x.m(\overline{z}) \rangle \wedge \langle x \texttt{ internal} \rangle\ )$$

For example,

Mod3 ⊨ a:Account => **Enc**(a.balance=b)

Note that

M ⊨ A => **Enc**(A')

does *not* imply that

M ⊨ A => **Enc**(¬ A')

3rd Idea: One-step necessary conditions

for all $C \in dom(M)$ *and* $m \in M(C).\texttt{mths}$,

$$M \vdash \texttt{from}\ A_1 \wedge x : C \wedge \langle \_\ \texttt{calls}\ x.m(\overline{z}) \rangle\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A_3$$

$$M \vdash A_1 \longrightarrow \neg A_2$$

$$M \vdash A_1 \Rightarrow Enc(A_2)$$

$$M \vdash \texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A_3$$

For example,

Mod2 |- a:Account => Enc(¬ (∃o. <o **external**> ∧ <o **access** a.pwd >))

Mod3 |- a:Account => Enc(¬ (∃o. <o **external**> ∧ <o **access** a.pwd >))

Mod3 |- **from** a:Account
next ∃o. <o **external**> ∧ <o **access** a.pwd >
onlyIf ∃o. <o **external**> ∧ <o **access** a.pwd >

But

Mod2 |-/- **from** a:Account
next ∃o. <o **external**> ∧ <o **access** a.pwd >
onlyIf ∃o. <o **external**> ∧ <o **access** a.pwd >

4th Idea: Emergent Behaviour

$$\frac{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThough}\ A_3 \qquad M \vdash \texttt{from}\ A_3\ \texttt{to}\ A_2\ \texttt{onlyThough}\ A}{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThough}\ A}$$

$$\frac{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThough}\ A_3 \qquad M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_3\ \texttt{onlyIf}\ A}{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A}$$

For example,

Mod3 |- **from** a:Account ∧ a.balance=b ∧ <_ **calls** a.transfer(a',pwd') >
to a.balance<b
onlyIf ∃o. <o **external**> ∧ <o **access** a.pwd >

And Mod2 |- **from** a:Account ∧ a.balance=b ∧ <_ **calls** a.transfer(a',pwd') >
next a.balance<b
onlyIf ∃o. <o **external**> ∧ <o **access** a.pwd >

But

Mod2 |- **from** a:Account ∧ a.balance=b ∧ <_ **calls** a.transfer(a',pwd') >
to a.balance<b
onlyIf ∃o. <o **external**> ∧ <o **access** a.pwd >

All the rules …

$$\frac{M \vdash \texttt{from}\ A\ \texttt{next}\ \neg A\ \texttt{onlyIf}\ A'}{M \vdash \texttt{from}\ A\ \texttt{to}\ \neg A\ \texttt{onlyThough}\ A'}\ (\textsc{Changes})$$

$$\frac{M \vdash A_1 \longrightarrow A_1' \qquad M \vdash A_2 \longrightarrow A_2' \qquad M \vdash A_3' \longrightarrow A_3 \qquad M \vdash \texttt{from}\ A_1'\ \texttt{to}\ A_2'\ \texttt{onlyThough}\ A_3'}{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThough}\ A_3}\ (\longrightarrow)$$

$$\frac{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThough}\ A \qquad M \vdash \texttt{from}\ A_1'\ \texttt{to}\ A_2\ \texttt{onlyThough}\ A'}{M \vdash \texttt{from}\ A_1 \vee A_1'\ \texttt{to}\ A_2\ \texttt{onlyThough}\ A \vee A'}\ (\vee \mathrm{I}_1)$$

$$\frac{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThough}\ A \qquad M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2'\ \texttt{onlyThough}\ A'}{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2 \vee A_2'\ \texttt{onlyThough}\ A \vee A'}\ (\vee \mathrm{I}_2)$$

$$\frac{M \vdash A_1 \longrightarrow A_1' \qquad M \vdash A_2 \longrightarrow A_2' \qquad M \vdash A_3' \longrightarrow A_3 \qquad M \vdash \texttt{from}\ A_1'\ \texttt{to}\ A_2'\ \texttt{onlyIf}\ A_3'}{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A_3}\ (\textsc{If}\text{-}\longrightarrow)$$

$$\frac{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A \qquad M \vdash \texttt{from}\ A_1'\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A'}{\ldots}\ (\textsc{If}\text{-}\vee \mathrm{I}_1)$$

$$\frac{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A'\ \texttt{onlyThough}\ \texttt{false} \qquad M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThough}\ A \vee A}{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThough}\ A}$$

$$M \vdash \texttt{from}\ A'\ \texttt{to}\ A_2\ \texttt{only}\ldots$$

$$\frac{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThough}\ A_3 \qquad M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_3\ \texttt{onlyThough}\ A}{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThough}\ A}$$

$$\frac{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A}{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThough}\ A}$$

$$\frac{\forall y,\ M \vdash \texttt{from}\ \ldots}{M \vdash \texttt{from}\ \ldots}$$

$$\frac{\forall y,\ M \vdash \texttt{from}\ \ldots}{M \vdash \texttt{fro}\ \ldots}$$

$$\frac{M \vdash \{x : C \wedge \ldots}{M \vdash \texttt{from}\ P_1 \wedge x : C \wedge \ldots}$$

$$\frac{M \vdash \{x : C \wedge \ldots}{M \vdash \texttt{from}\ \texttt{inside}(y) \wedge x : C \wedge \ldots}$$

$$\textit{for all } C \in \textit{dom}(M) \textit{ and } m \in M(C).\texttt{mths}, \quad M \vdash \texttt{from}\ A_1 \wedge x : C \wedge \langle \_\ \texttt{calls}\ x.m(\overline{z}) \rangle\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A_3$$

$$\frac{M \vdash A_1 \longrightarrow \neg A_2 \qquad M \vdash A_1 \Rightarrow \mathit{Enc}(A_2)}{M \vdash \texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A_3}$$

$$\frac{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A}{M \vdash \texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A}\ (\textsc{If1}\text{-}\textsc{If})$$

$$\frac{M \vdash A_1 \longrightarrow A_1' \qquad M \vdash A_2 \longrightarrow A_2' \qquad M \vdash A_3' \longrightarrow A_3 \qquad M \vdash \texttt{from}\ A_1'\ \texttt{next}\ A_2'\ \texttt{onlyIf}\ A_3'}{M \vdash \texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A_3}\ (\textsc{If1}\text{-}\longrightarrow)$$

$$\frac{M \vdash \texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A \qquad M \vdash \texttt{from}\ A_1'\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A'}{M \vdash \texttt{from}\ A_1 \vee A_1'\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A \vee A'}\ (\textsc{If1}\text{-}\vee \mathrm{I}_1)$$

$$\frac{M \vdash \texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A \qquad M \vdash \texttt{from}\ A_1\ \texttt{next}\ A_2'\ \texttt{onlyIf}\ A'}{M \vdash \texttt{from}\ A_1\ \texttt{next}\ A_2 \vee A_2'\ \texttt{onlyIf}\ A \vee A'}\ (\textsc{If1}\text{-}\vee \mathrm{I}_2)$$

$$\frac{M \vdash \texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A \vee A' \qquad M \vdash \texttt{from}\ A'\ \texttt{to}\ A_2\ \texttt{onlyThough}\ \texttt{false}}{M \vdash \texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A}\ (\textsc{If1}\text{-}\vee \mathrm{E})$$

$$\frac{M \vdash \texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A \qquad M \vdash \texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A'}{M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyIf}\ A \wedge A'}\ (\textsc{If1}\text{-}\wedge \mathrm{I})$$

$$\frac{\forall y,\ M \vdash \texttt{from}\ ([y/x]A_1)\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A}{M \vdash \texttt{from}\ \exists x.[A_1]\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A}\ (\textsc{If1}\text{-}\exists_1)$$

$$\frac{\forall y,\ M \vdash \texttt{from}\ A_1\ \texttt{next}\ ([y/x]A_2)\ \texttt{onlyIf}\ A}{M \vdash \texttt{from}\ A_1\ \texttt{next}\ \exists x.[A_2]\ \texttt{onlyIf}\ A}\ (\textsc{If1}\text{-}\exists_2)$$

$$\ldots\ \texttt{onlyIf}\ A \vee A' \qquad \texttt{onlyThough}\ \texttt{false}}{\texttt{onlyIf}\ A}\ (\textsc{If}\text{-}\vee \mathrm{E})$$

$$\ldots\ A_3 \qquad \texttt{onlyIf}\ A}{\texttt{onlyIf}\ A}\ (\textsc{If}\text{-}\textsc{Trans})$$

$$\texttt{onlyIf}\ A_1\ (\textsc{If}\text{-}\textsc{Start})$$

$$\ldots_2)\ \texttt{onlyIf}\ A \qquad \texttt{onlyIf}\ A}{\texttt{onlyIf}\ A}\ (\textsc{If}\text{-}\exists_2)$$

And we have proven

Mod3 |- **from** a:Account ∧ a.balance=b
to a.balance<b
onlyIf ∃o. <o **external**> ∧ <o **access** a.pwd >

Mod_{best} |- **from** a:Account ∧ a.balance=b
to a.balance<b
onlyIf ∃o. <o **external**> ∧ <o **access** a.pwd >

where Mod_{best} has a different API, and consists of **Bank**, with a Ledger holding the Accounts' balances, and balance is a recursively defined ghost field

```
module Mod_best
  class Account
    field password : Object
    method authenticate(pwd : Object) : bool
      (PRE:  a : Account ∧ b : Bank
       POST: b.balance(a)_old == b.balance(a)_new)
      (PRE:  a : Account
       POST: res != a.password)
      (PRE:  a : Account
       POST: a.password_old == a.password_new)
      {return pwd == this.password}
    method changePassword(pwd : Object, newPwd : Obje
      (PRE:  a : Account
       POST: res != a.password)
      (PRE:  a : Account ∧ b : Bank
       POST: b.balance(a)_old == b.balance(a)_new)
      (PRE:  a : Account ∧ pwd != this.password
```

```
    class Bank
      field book : Ledger
      ghost intrnrl balance(acc) : int = book.balance(acc)
      method transfer(pwd : Object, amt : int, from : Account, to : Account) : void
        (PRE:  a : Account ∧ b : Bank ∧ ¬ (a == acc1 ∧ a == acc2)
         POST: b.balance(a)_old a= b.balance(a)_new)
        (PRE:  a : Account
         POST: res != a.password)
        (PRE:  a : Account
         POST: a.password_old == a.password_new)
        {if (from.authenticate(pwd))
           book.transfer(amt, from, to)}

confined class
  field acc1 : Account
  field bal1 : int
  field acc2 : Account
  field bal2 : int
  ghost intrnl balance(acc) : int =
    if acc == acc1
      bal1
    else if acc == acc2
      bal2
    else -1
  method transfer(amt : int, from : Account, to : Account) : void
    (PRE:  a : Account ∧ b : Bank ∧ (a != acc1 ∧ a != acc2)
     POST: b.balance(a)_old == b.balance(a)_new)
```

Today

- Functional Specifications do not adequately address Robustness
- What do we propose?
- What does it mean
- How do we prove?
- **Is it sound?**

4.3 Soundness of the Necessity Logic

THEOREM 4.4 (SOUNDNESS). *Assuming a sound Assert proof system, $M \vdash A$, and a sound encapsulation inference system, $M \vdash A \implies Enc(A')$, and and that on top of these systems we built the Necessity logic according to the rules in Figures 2, and 3, and 5, and 4, then, for all modules M, and all Necessity specifications S:*

$$M \vdash S \quad \textit{implies} \quad M \vDash S$$

PROOF. by induction on the derivation of $M \vdash S$.

□

Coq formalism deviates slightly from the system as presented here, mostly in the formalization of the *Assert* language. The Coq version of *Assert* restricts variable usage to expressions, and allows only addresses to be used as part of non-expression syntax. For example, in the Coq formalism we can write assertions like $x.f$ == `this` and $x == \alpha_y$ and $\langle \alpha_x$ `access` $\alpha_y \rangle$, but we cannot write assertions like $\langle x$ `access` $y \rangle$, where x and y are variables, and α_x and α_y are addresses. The reason for this restriction in the Coq formalism is to avoid spending significant effort encoding variable renaming and substitution, a well-known difficulty for languages such as Coq. This restriction does not affect the expressiveness of our Coq formalism: we are able to express assertions such as $\langle x$ `access` $y \rangle$, by using addresses and introducing equality expressions to connect variables to address, *i.e.* $\langle \alpha_x$ `access` $\alpha_y \rangle \wedge \alpha_x == x \wedge \alpha_y == y$.

Summary

- Functional Specifications do not adequately address Robustness
- We propose Holistic specifications, which describe the necessary condition for some effect to take place
- We gave a formal semantics
- We developed a proof system starting with the insight that

  from A*orig* **to** A*fin* **onlyIf** A*nec*

  is equivalent with

  $\forall$ `stmts`. { A*orig* $\wedge$ $\neg$ A*nec* } `stmts` { $\neg$ A*fin* }

- We have proven soundness of the proof system, and used the proof system to prove adherence for two modules

Thank you

Holistic Specifications — examples

- ERC20
- [DAO]
- DOM attenuation
- [DAO in VerX]
- [Purse and Crowdfunder]
- Escrow

Example2: ERC20

a popular standard for initial coin offerings;

clients can buy and transfer tokens, and can designate other clients to transfer on their behalf.

In particular, a client may call

- `transfer`: transfer some of her tokens to another clients,
- `approve`: 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

Holistic spec - reduce balance

```
≜ from e : ERC20 ∧ e.balance(p) = m + m' ∧ m > 0
   next e.balance(p) = m'
   onlyIf ∃ p' p''.[⟨p' calls e.transfer(p, m)⟩ ∨
          e.allowed(p, p'') ≥ m ∧ ⟨p'' calls e.transferFrom(p', m)⟩]
```

An ERC20 client's (p) balance decreases *only* if that client, or somebody allowed by that client (p'), made a payment.

Holistic spec - allowed

```
from e : ERC20 ∧ p : Object ∧ p' : Object ∧ m : Nat
 next e.allowed(p, p') = m
 onlyIf ⟨p calls e.approve(p′, m)⟩ ∨
        (e.allowed(p, p') = m ∧
         ¬ ((⟨p′ calls e.transferFrom(p, _)⟩ ∨
                ⟨p calls e.allowed(p, _)⟩)) ∨
        ∃ p''. [e.allowed(p, p') = m + m' ∧ ⟨p′ calls e.transferFrom(p″, m′
```

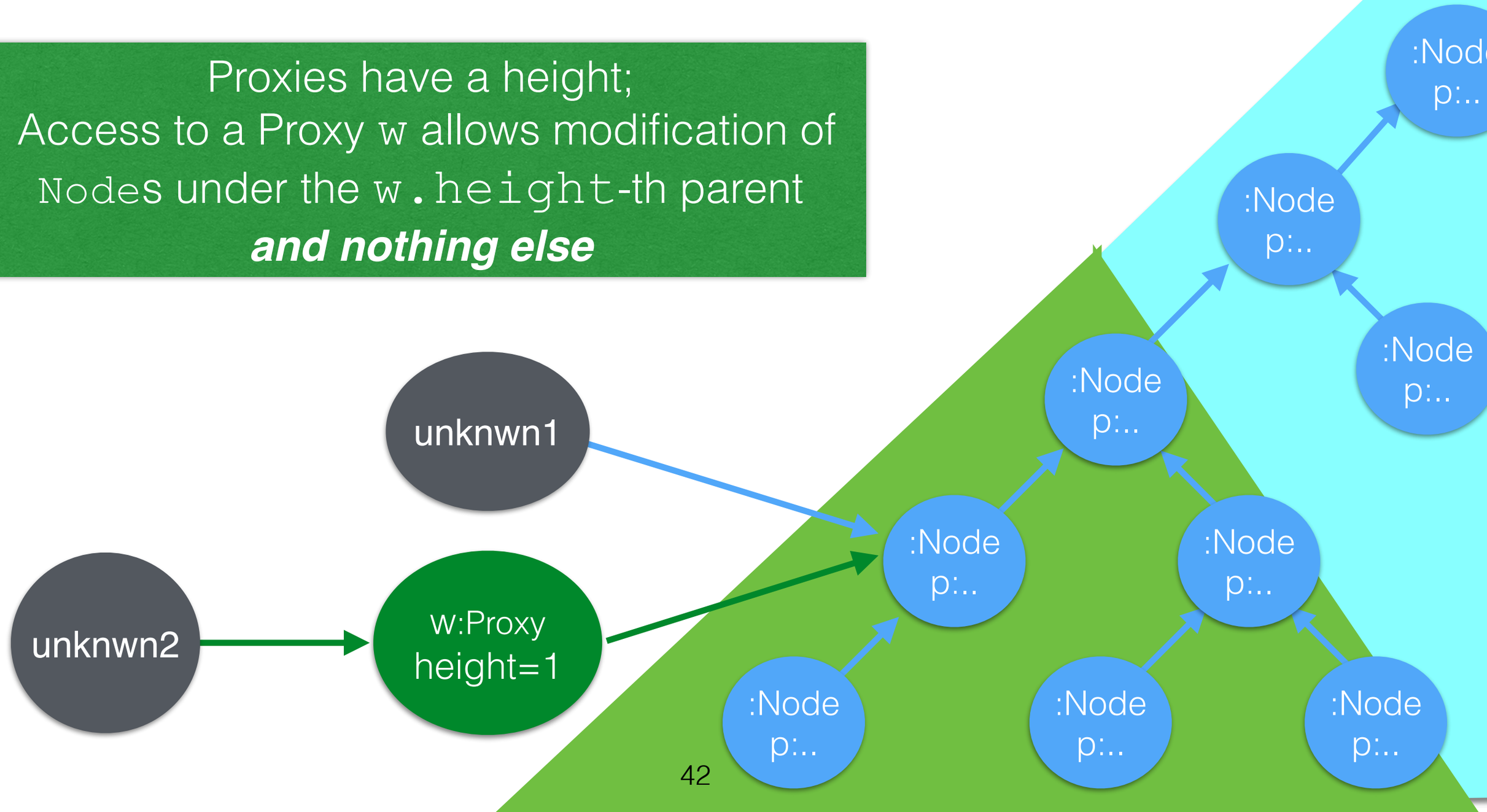
`p′` is allowed in the next step to spend `m` from `p`'s account,
only if in the current step

`p′` is allowed to spend `m` and does not spend any for `p`'s money
or
`p` authorises `p′` to spend `m` on their behalf
or
`p` was allowed to spend `m`+ `m`' and spends `m′`

Example 3: DOM attenuation

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

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



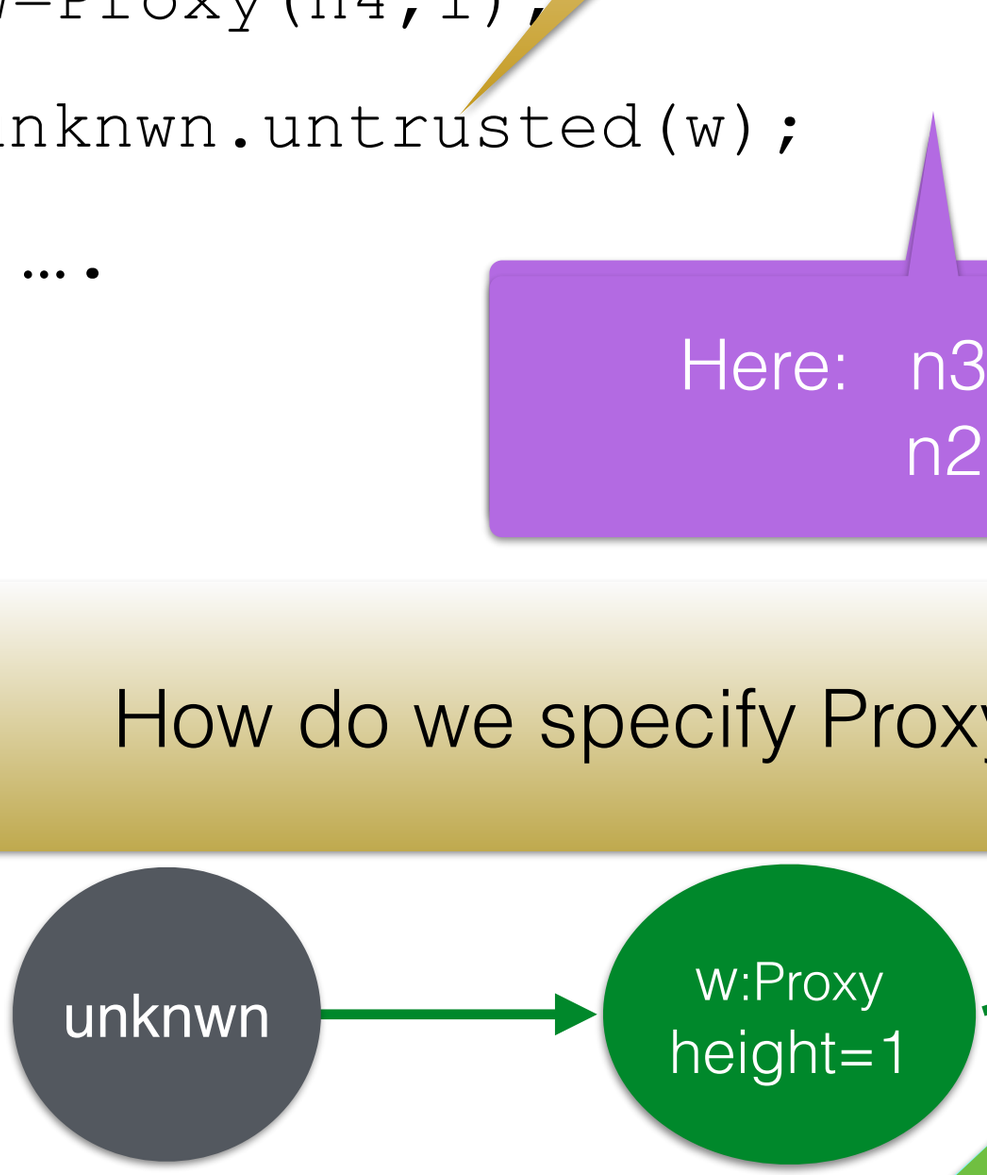
DOM attenuation

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

open world

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

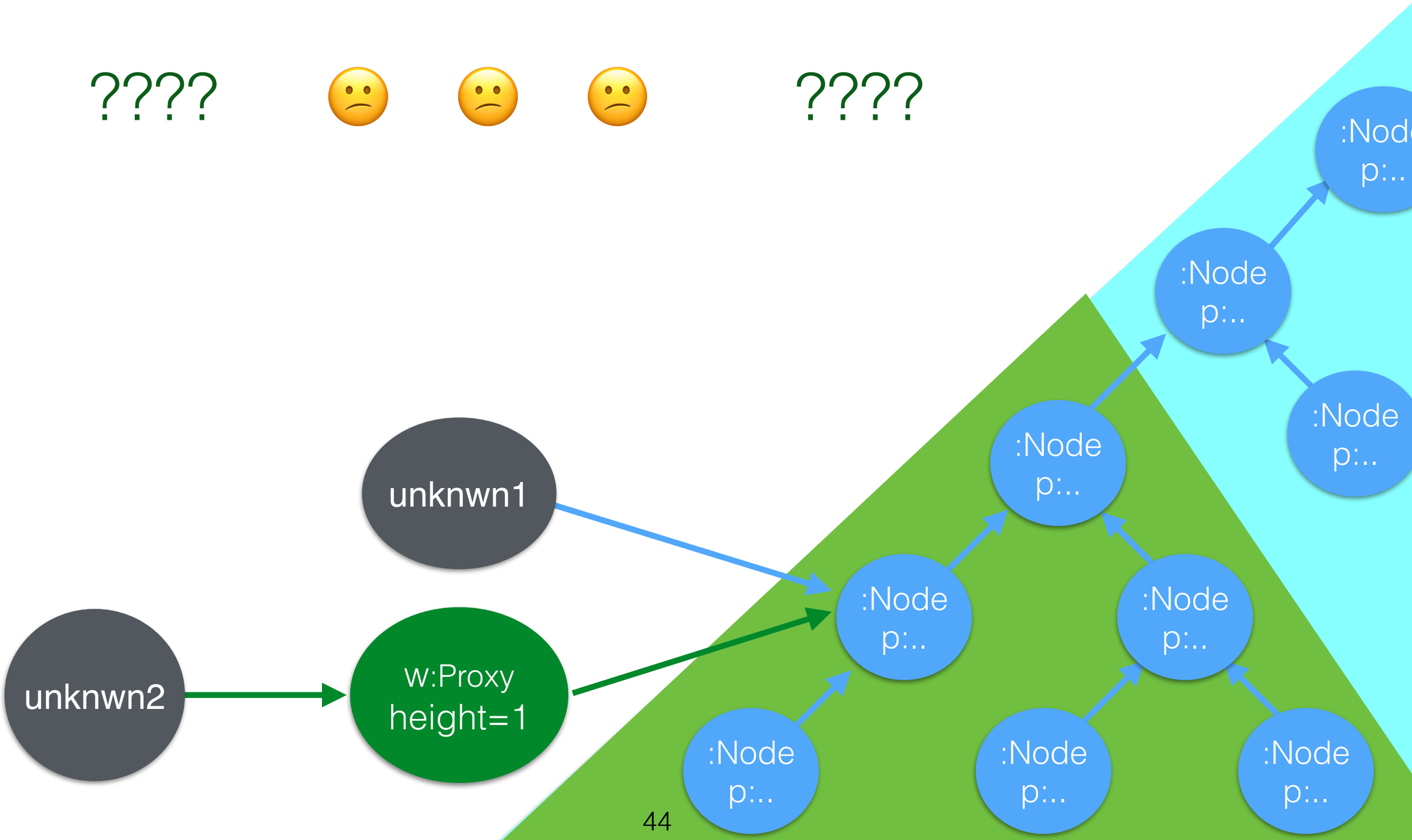
How do we specify Proxy?



classical

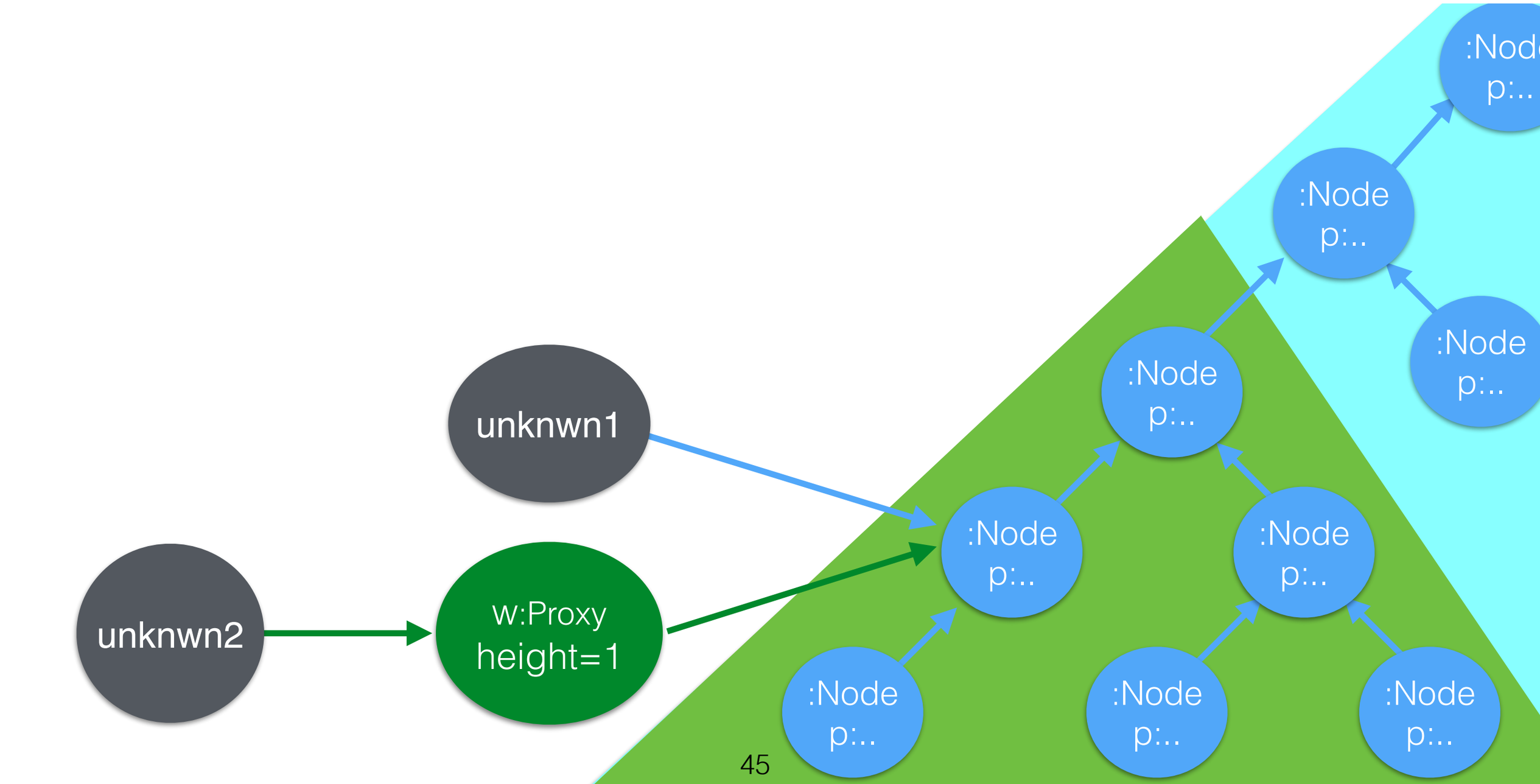
Access to Proxy `w` allows modification of `Node`s under the `w.height`-th parent and nothing else

???? 😕 😕 😕 ????



Holistic

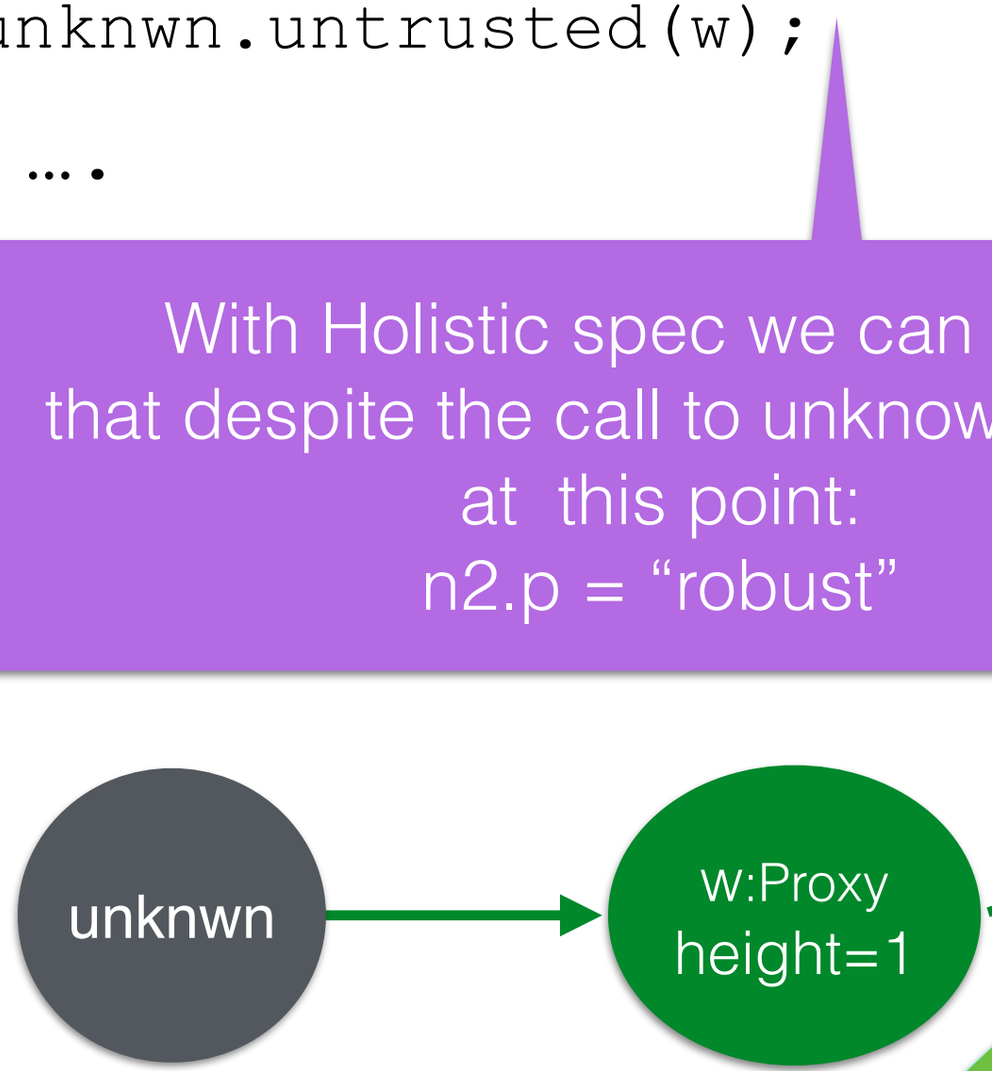
```
from nd : Node ∧ nd.property = p  to nd.property != p
onlyIf ∃ o.[ ⟨o external⟩ ∧
             ( ∃ nd':Node.[ ⟨o access nd'⟩ ]  ∨
               ∃ pr:Proxy,k:ℕ.[⟨o access pr⟩ ∧ nd.parent^k=pr.node.parent^pr.height ]
```



using Holistic spec

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

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



Today

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

- **Trust and Obeys**

Swapsies on the Internet

First Steps towards Reasoning about Risk and Trust in an Open World

Sophia Drossopoulou[1], James Noble[2], Mark S. Miller[3]

[1]Imperial College London, [2]Victoria University Wellington, [3]Google Inc.

Abstract

Contemporary open systems use components developed by many different parties, linked together dynamically in unforeseen constellations. Code needs to live up to strict *security specifications:* it has to ensure the correct functioning of its objects when they collaborate with external objects which

careful to take only stickers you're definitively willing to risk losing when you go to meet the gorilla in year ten.

Internet Swapsies Playground swapsies is just one example of an interaction between *mutually untrusting* parties in an *open system* — other examples include so-called dark Internet markets (like Silk Road and Evolution) and even larger

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]

Exchange money and goods
buyer and seller do not trust one another

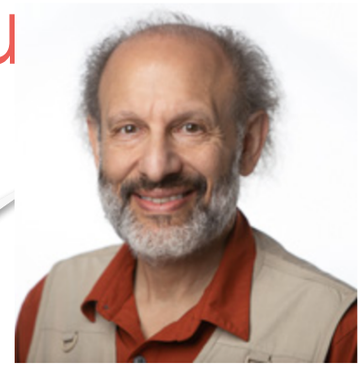
Distributed Electronic Rights in JavaScript

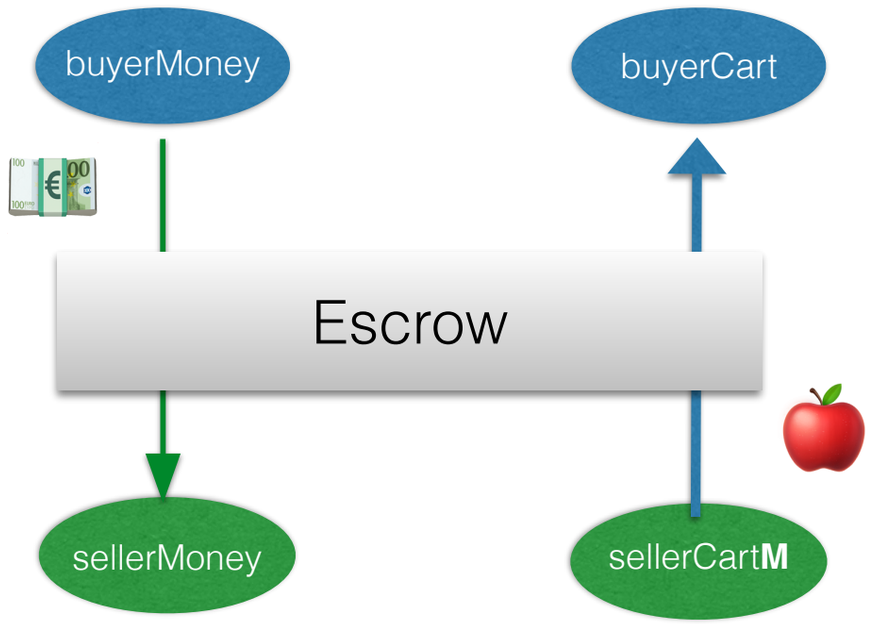
Mark S. Miller[1], Tom Van Cutsem[2], and Bill Tulloh

[1] Google, Inc.
[2] Vrije Universiteit Brussel

Abstract. Contracts enable mutually suspicious parties to cooperate safely through the exchange of rights. Smart contracts are programs whose behavior enforces the terms of the contract. This paper shows how such contracts can be specified elegantly and executed safely, given an appropriate distributed, secure, persistent, and ubiquitous computational fabric. JavaScript provides the ubiquity but must be significantly extended to deal with the other aspects. The first part of this paper is a progress report on our efforts to turn JavaScript into this fabric

Exchanging money for goods without mutual tru





That *is* a surprise!

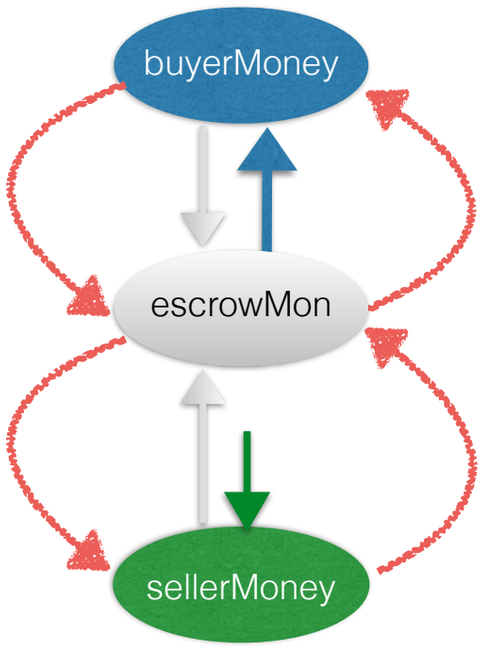
```
cla                                    y,
                                       ney:Purse<Money>,
                                       rt,
                                       erCart: Purse<Goods>)
        : bool
}
```

¬res → "no transfer" ∧
(¬ buyerMoney **obeys** ValidPurse ∧ sellerMoney **obeys** ValidPurse ∨
buyerMoney **obeys** ValidPurse ∧ ¬ sellerMoney **obeys** ValidPurse ..)

res → "transfer happened" ∧ buyerMoney **obeys** ValidPurse
∧ sellerMoney **obeys** ValidPurse
∧ …

∨ ¬(buyerMoney **obeys** ValidPurse) ∧ ¬ sellerMoney **obeys** ValidPurse

Exchanging money for goods without mutual trust



escrowMoney = sellerMoney.sprout()

// sellerMoney **obeys** ValidPurse → escrowMoney **obeys** ValidPurse

res= escrowMoney. deposit (buyerMoney,**0**)

if **!res** then **return** res

// sellerMoney **obeys** ValidPurse → buyerMoney **obeys** ValidPurse

res= buyerMoney. deposit(escrowMoney,**0**)

if **!res** then **return** res

// buyerMoney **obeys** ValidPurse → escrowMoney **obeys** ValidPurse

res= escrowMoney. deposit (buyerMoney,**0**)

if **!res** then **return** res

// buyerMoney **obeys** ValidPurse → seller **obeys** ValidPurse

// sellerMoney **obeys** ValidPurse ⟷ buyerMoney **obeys** ValidPurse)

Summary

- We argue that robustness characterisation requires a mix of classical and Holistic specifications.
- We developed a novel specification language for Holistic
- We applied to examples from contracts and from object capabilities literature.
- We developed a logic for reasoning about Holistic
- We used the logic to prove several examples

Further Work

- Relax some of the underlying language restrictions
- Module hierarchies
- Reason about encapsulation
- Better ways to go from classical method spec to per-module Holistic spec
- Tool?
- Revisit trust and Obey
- …

What was challenging/ fun/interesting

- Holistic Specs are second class!
- Two-module execution
- Coq model vs renaming
- ghostfields (cycles), while keeping assertions classical
- Holistic Logic, and avoid talking about specific intermediate configurations

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 Proxies)

ERC20 classical spec - transfer - 2

What if c1's balance not large enough?

e:ERC20 ∧ this = c ∧ e.balance(c1) < m

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

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

ERC20 classical spec
- what about the other functions?

ERC20 classical spec

Is that robust?

a “super-cleint,” authorised on all?

a function that takes 0.5% from each account?

sufficient
for chan e of balance

I am worried about who/what can reduce my balance

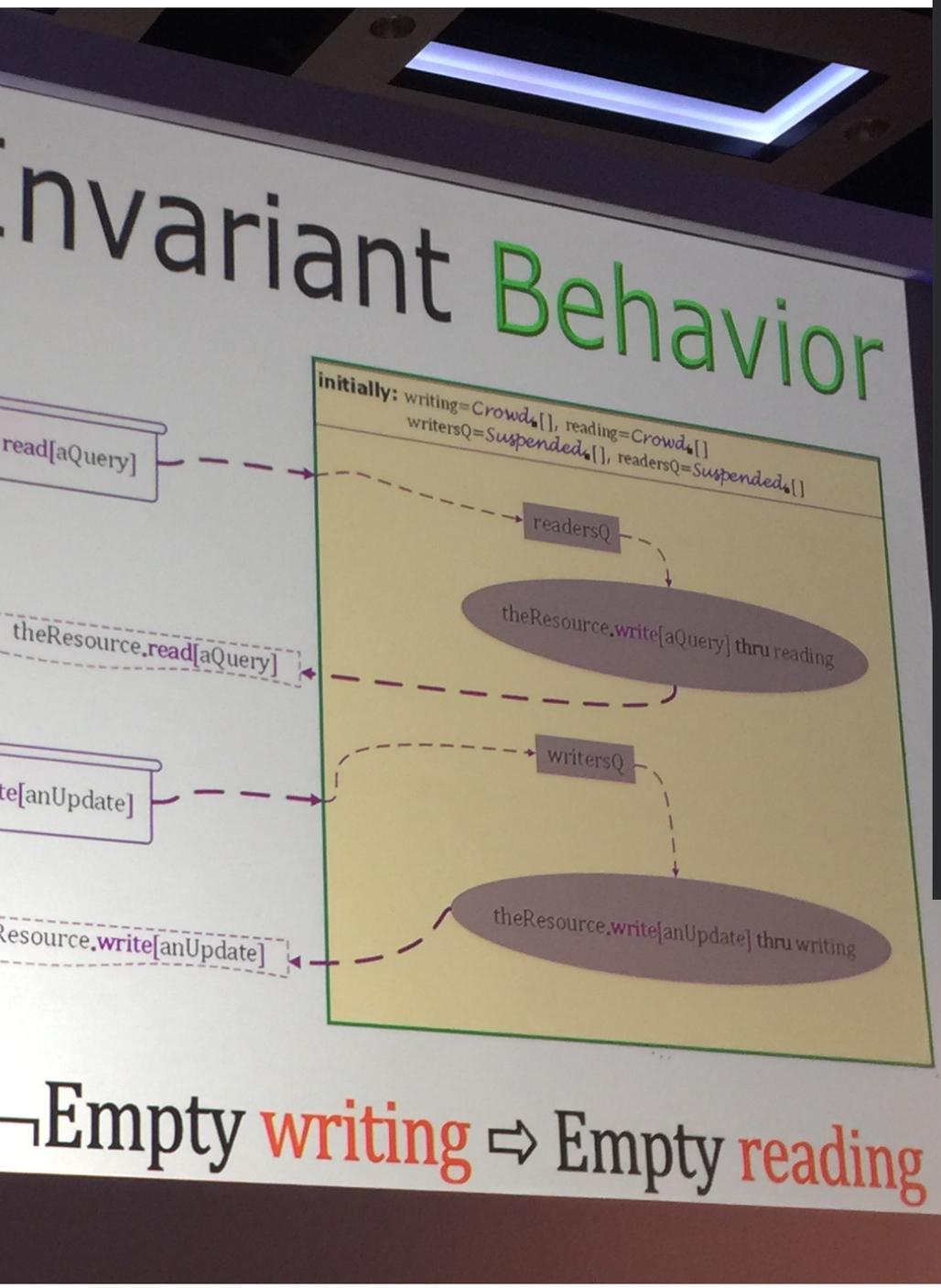
can authority increase?

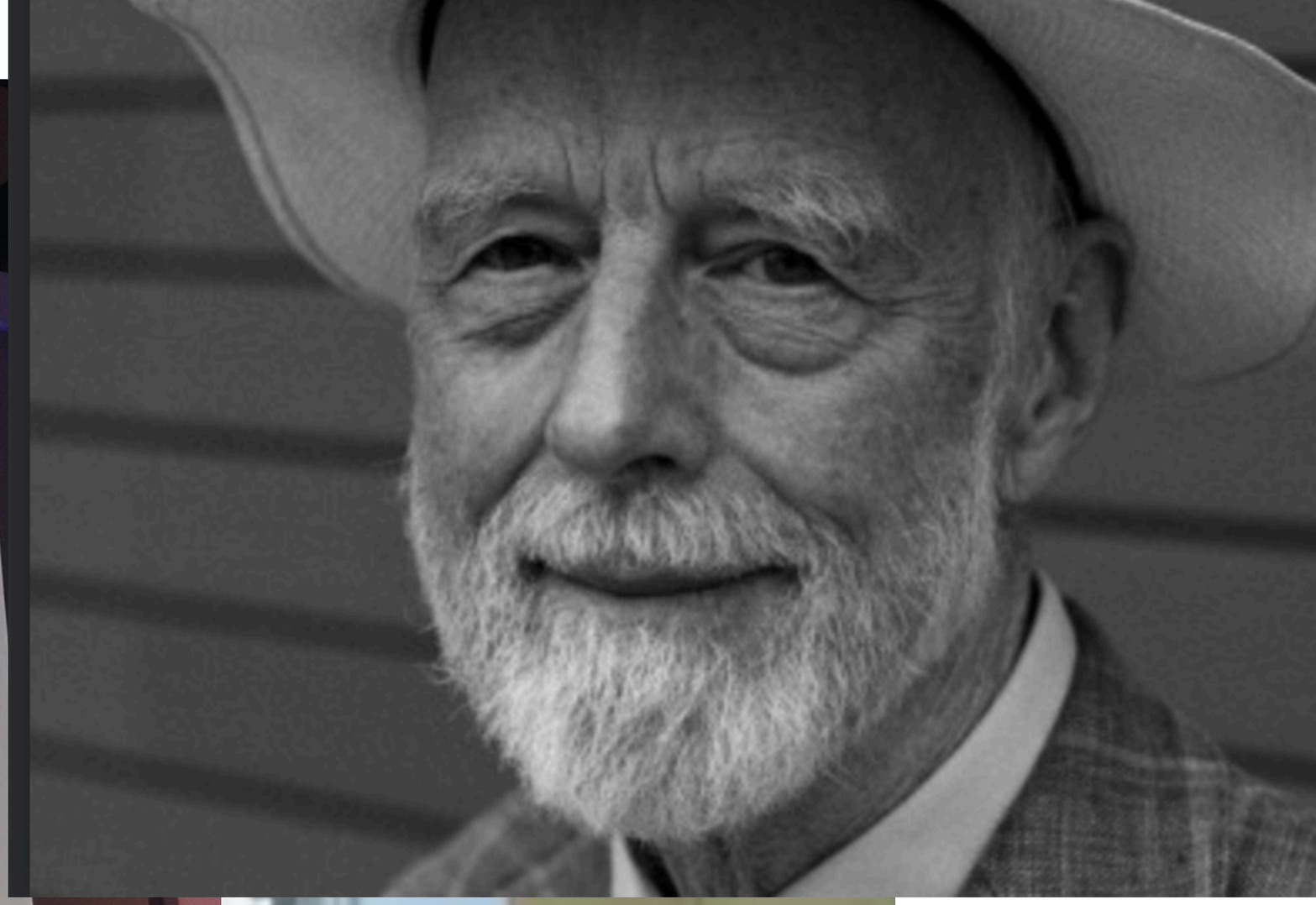
```
e.balance(c1) ≧ m
    { e.transferFrom                m) }
e.balance(c1) = e.bala                pre -m ∧
e.balance(c2) = e.balanc              re +m ∧ e.Authc
```

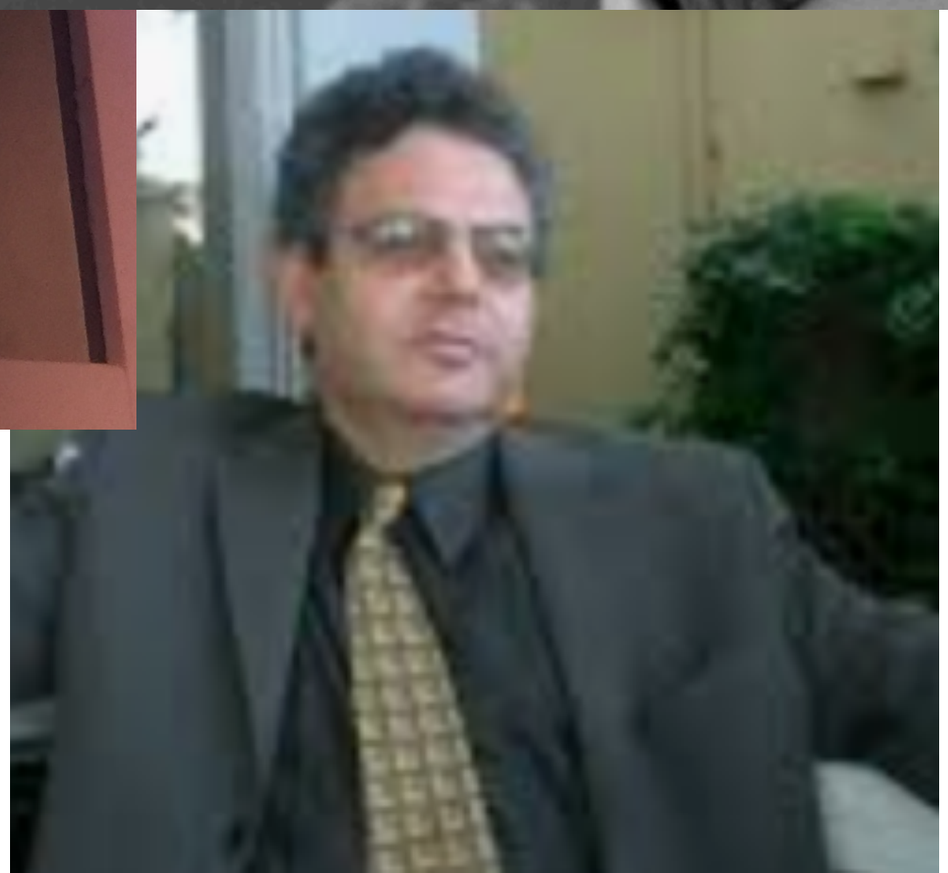
```
RC20 ∧ thi = c1 ∧ e.balance(c1) < m
        { .transf
∀ c. e.balance(c)
```

```
zed(c
anc (c) pre ∧
c2,m) →e.Authori-ed(c1,c2,m)]
```

```
… { e.allowanceOf(c2) } ….
… { e.balanceOf(c) } ….
```


nvariant Behavior
read[aQuery]
initially: writing=Crowd[], reading=Crowd[]
writersQ=Suspended[], readersQ=Suspended[]
readersQ
theResource.read[aQuery]
theResource.write[aQuery] thru reading
te[anUpdate]
writersQ
Resource.write[anUpdate]
theResource.write[anUpdate] thru writing
Empty writing ⇨ Empty reading





Holistic specs - invariants ++

- state
- **time**
- **space**
- **control**
- **permission**
- **authority**
- **(when/where do they hold)**

Holistic spec - authority

effect

`e.Authorized(c1,c2,m)` ≙

Prev(`c1.`**Calls**`(e,approve,c2,m)`)

∨

Prev(`e.Authorized(c1,c2,m+m')` ∧ `c2.`**Calls**`(e,transferFrom,c1,_,m')`)

∨

Prev(`e.Authorized(c1,c2,m)` ∧ ¬ `c2.`**Calls**`(e,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 … ….

Holistic

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

Classical

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

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

Authorized(cl,cl') ≜ ∃m: Nat. **Prev**(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: ∀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 = ∑ cl ∈ 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 ….

Holistic

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

Classical

- per function; *sufficient* conditions for some action/effect
- *explicit* about individual function, and *implicit* 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 **Prev**(d.Balance(cl),m)
∧ **Prev**(cl.**Calls**(d.payIn(m'))
0 if **Prev**(cl.**Calls**(d.repayIn())

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**:….

[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 ∧ **Will** (**Changes**(a.balance)) **in** S)

→

necessary condition

∃o∈S. [External(o) ∧ 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 is external to the module, and can access a directly now.

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**: **??????**

[Miller et al, Financial Crypto 2000]