LMS/BCS-FACS Evening Seminar — London and Sydney.

Probabilistic Datatypes: Automating verification for abstract probabilistic reasoning

Annabelle McIver
Chris Chen
Carroll Morgan

Probabilistic Programs – a brief history

Randomised algorithms for resource, security, performance.

← 1980-ish →

Efficient reasoning principles, semantics, algebras, logics.

- Probabilistic power domains (Jones, Plotkin, Saheb-Djahromi)
- The probabilistic Monad (Giry)
- Source-level reasoning for sequential programs (no non-determinism) (Kozen)

← 1990-ish →

- Source-level reasoning for sequential programs, pGCL (yes non-determinism!)

Probabilistic Programs – a brief history

Randomised algorithms for resource, security, performance.

1980-ish

Efficient reasoning principles, semantics, algebras, logics.

- Probabilistic power domains (Jones, Plotkin, Saheb-Djahromi)
- The probabilistic Monad (Giry)
- Source-level reasoning for sequential programs (no non-determinism) (Kozen)

1990-ish

- Source-level reasoning for sequential programs, pGCL (yes non-determinism!)

Probabilistic Programs — a brief history

Randomised algorithms for resource, security, performance.

1980-ish

Efficient reasoning principles, semantics, algebras, logics.

- Probabilistic power domains (Jones, Plotkin, Saheb-Djahromi)
- The probabilistic Monad (Giry)
- Source-level reasoning for sequential programs (no non-determinism) (Kozen)

1990-ish

- Source-level reasoning for sequential programs, pGCL (yes non-determinism!)

Probabilistic Programs – a brief history

Randomised algorithms for resource, security, performance.

1980-ish

Efficient reasoning principles, semantics, algebras, logics.

- ❖ Probabilistic power domains (Jones, Plotkin, Saheb-Djahromi)
- ❖ The probabilistic Monad (Giry)
- ❖ Source-level reasoning for sequential programs (no non-determinism) (Kozen)

1990-ish

- ❖ Source-level reasoning for sequential programs, pGCL (yes non-determinism!)

Efficient Reasoning for Probabilistic Programs

- Source-level reasoning for sequential programs, pGCL (yes non-determinism!)

  - Non-determinism — enables transition abstraction.
    $P \sqsubseteq Q$ means that program Q satisfies all the properties of P

  - Source-level reasoning.
    $\{\,\phi\,\}\ \ P\ \ \{\,\psi\,\}$
    is a *Probabilistic* Hoare-style triple; ϕ and ψ are real-valued functions.

  - Tools for verifying all the triples…automatically if possible!

A challenge program

```
{ 1/N } # Precondition
var c, v = 0, 1
while(v<N or c≥N){
    {Inv × [v<N or c≥N]}
    □ (v ≤ N) → v= 2v
                c= 2c  1/2⊕  c= 2c+1
    □ (v > N) → v,c= v-N,c-N
    { Inv }
}
{ [c = i] } # Post condition for any 0 ≤ i < N
```

Set c uniformly between 0 and N-1

- Why is this fast?
- Why is this a challenge?
- What is Inv ?
- What should we do?

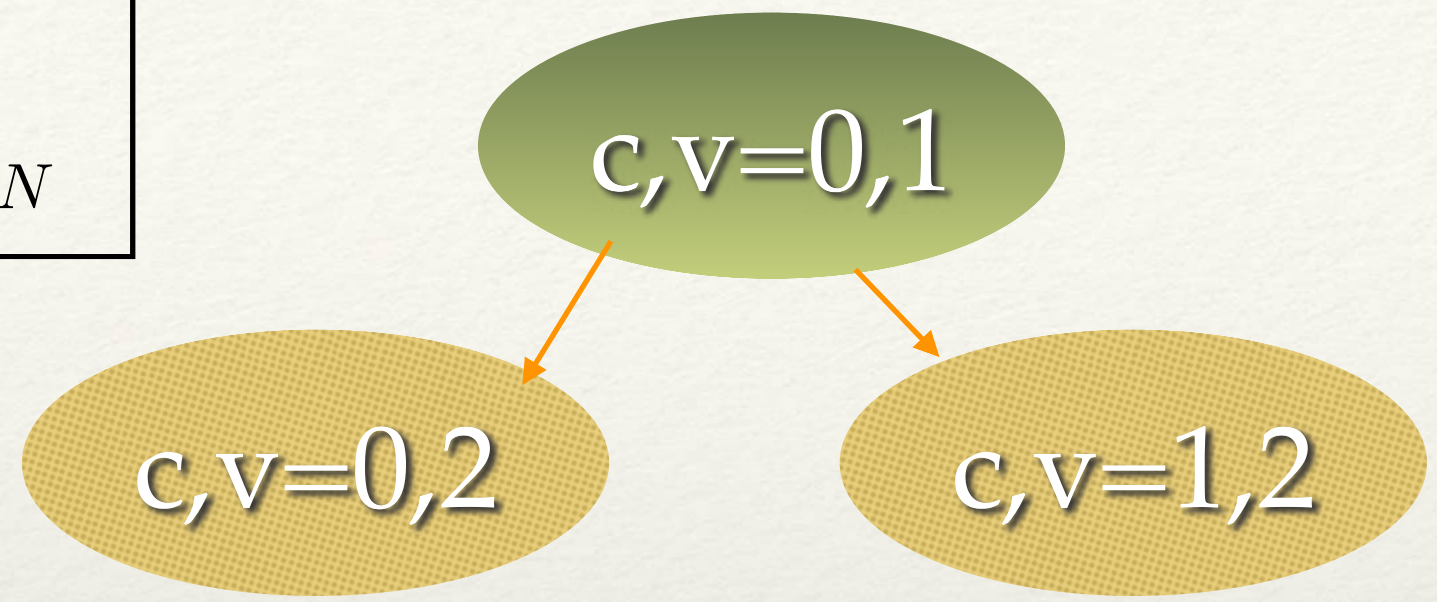
A challenge program, N=5, i=3

```
{ 1/N } # Precondition
var c, v = 0, 1
while(v<N or c≥N){
    {Inv × [v<N or c≥N]}
    □ (v ≤ N) → v= 2v
                c= 2c  1/2⊕  c= 2c+1
    □ (v > N) → v,c= v-N,c-N
    { Inv }
}
{ [c = i] } # Post condition for any 0 ≤ i < N
```

c,v=0,1

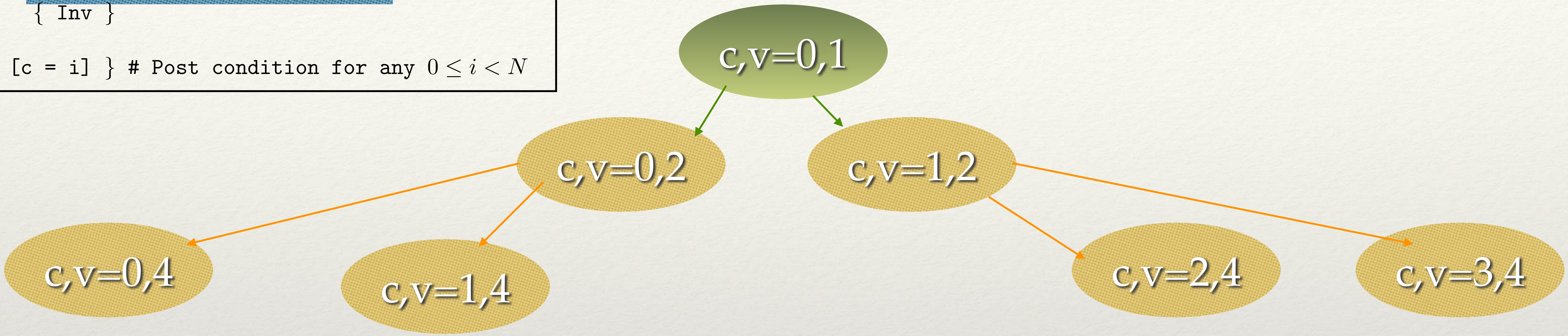
A challenge program, N=5, i=3

```
{ 1/N } # Precondition
var c, v = 0, 1
while(v<N or c≥N){
    {Inv × [v<N or c≥N]}
    □ (v ≤ N) → v= 2v
                c= 2c  1/2⊕  c= 2c+1
    □ (v > N) → v,c= v-N,c-N
    { Inv }
}
{ [c = i] } # Post condition for any 0 ≤ i < N
```



A challenge program, N=5, i=3

```
{ 1/N } # Precondition
var c, v = 0, 1
while(v<N or c≥N){
    {Inv × [v<N or c≥N]}
    □ (v ≤ N) → v= 2v
              c= 2c  1/2⊕  c= 2c+1
    □ (v > N) → v,c= v-N,c-N
    { Inv }
}
{ [c = i] } # Post condition for any 0 ≤ i < N
```



A challenge program, N=5, i=3

```
{ 1/N } # Precondition
var c, v = 0, 1
while(v<N or c≥N){
     {Inv × [v<N or c≥N]}
     □ (v ≤ N) → v= 2v
               c= 2c  1/2⊕  c= 2c+1
     □ (v > N) → v,c= v-N,c-N
     { Inv }
}
{ [c = i] } # Post condition for any 0 ≤ i < N
```

A challenge program, N=5, i=3

```
{ 1/N } # Precondition
var c, v = 0, 1
while(v<N or c≥N){
    {Inv × [v<N or c≥N]}
    □ (v ≤ N) → v= 2v
             c= 2c 1/2⊕ c= 2c+1
    □ (v > N) → v,c= v-N,c-N
    { Inv }
}
{ [c = i] } # Post condition for any 0 ≤ i < N
```

What should we do?

Abstract datatypes

Abstract

The motivation behind the work in very-high-level languages is to ease the programming task by providing the programmer with a language containing primitives or abstractions suitable to his problem area. The programmer is then able to spend his effort in the right place; he concentrates on solving his problem, and the resulting program will be more reliable as a result. Clearly, this is a worthwhile goal.

Unfortunately, it is very difficult for a designer to select in advance all the abstractions which the users of his language might need. If a language is to be used at all, it is likely to be used to solve problems which its designer did not envision, and for which the abstractions embedded in the language are not sufficient.

- Barbara Liskov and Stephen Zilles,

  Proceedings of the ACM SIGPLAN symposium on Very high level languages, 1974

An example: a quick prototype

```
class Set(N):  # New abstract version, capacity limited.
  # DTI: |ss|<=N                             # Data-type invariant.

  local:
    # PRE: N>=0                            # Data-type precondition.
    ss= {}

  def makeEmpty:
    ss= {}

  def add(s):
    PRE: |ss|!=N            # Must hold when add() is called.
    ss= ss∪{s}

  def isIn(s):
    return s∈ss
```

Taken from: Formal Methods, Informally, Carroll Morgan (to appear, CUP).

Programming with specifications

```
class Set(N):  # New abstract version, capacity limited.
  # DTI: |ss|<=N                       # Data-type invariant.
  local:
    # PRE: N>=0                        # Data-type precondition.
    ss= {}
  def makeEmpty:
    ss= {}
  def add(s):
    PRE: |ss|!=N            # Must hold when add() is called.
    ss= ss∪{s}
  def isIn(s):
    return s∈ss
```

Implementor side

```
SetmySet = Set(10);
mySet.add(3);
mySet.add(4);
Bool z = mySet.isIn(5);
```

User side

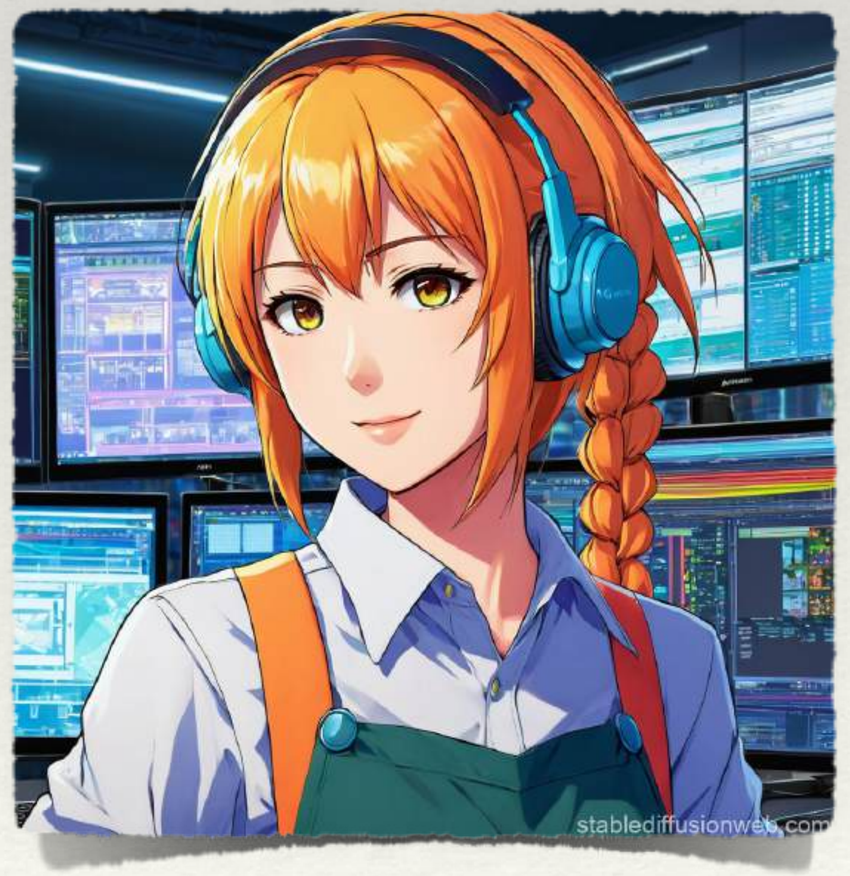
What does this mean for verification?

Definition 1. *An* encapsulated datatype *is a triple* (I, OP, F), *where* I *and* F *are two distinguished operations called respectively the initialisation and finalisation, and* OP *is an indexed set of publicly accessible operations.* [12]

Definition 2. *A datatype* (I, OP, F) *is* refined by (I', OP', F') [12] *if, for every program* $\mathcal{P}$ *expressible using the constructs mentioned above, including calls on corresponding operators in* OP *and* OP' *, we have*

$$I; \mathcal{P}(OP); F \quad \sqsubseteq \quad I'; \mathcal{P}(OP'); F' \ ,$$

where ";" indicates sequential composition.



Today's talk

- What happens when some of the behaviour can be probabilistic?
- Do the traditional proof methods (eg simulation) still work?
- Can we still use the abstract specification to prove properties of programs that use probabilistic datatypes?
- If they don't, what must be changed?
  - A semantics and refinement that distinguishes hidden and visible state;
  - Does it apply to our challenge program?
  - We need to talk about information leaks…

What happens when the datatype is probabilistic?

Abstract (specification) datatype for a coin flip

Initialisation

Operation

Probabilistic choice

Finalisation

`var coin # Local variable`

I_A: `skip`

$\texttt{Flip}_A$: `# Flip on demand.`

```
        coin:= H
```

$_{1/2}\oplus$ `coin:= T;`

```
    return coin
```

F_A: `skip`

(a) The abstract datatype

Programming with the coin datatype

```
var coin # Local variable

I_A:    skip
Flip_A: # Flip on demand.
            coin:= H
        1/2⊕ coin:= T;
        return coin
F_A:    skip
```

(a) The abstract datatype

I

```
v:= H ⊓ T;
g:= Flip()
```

F

Demonic nondeterministic choice

What is the probability that g=v finally?

Reasoning with the coin datatype

I

```
v:= H⊓T;
g:= Flip()
```

F

This is what would be expected, but can we prove it formally? What happens when we implement the datatype?

What is the probability that g=v finally?

We can reason about the program by using the abstract datatype and Hoare Triples…

```
var coin # Local variable

I_A:    skip
Flip_A: # Flip on demand.
                coin:= H
        1/2⊕    coin:= T;
            return coin
F_A:    skip
```

(a) The abstract datatype

{ Pre Condition}

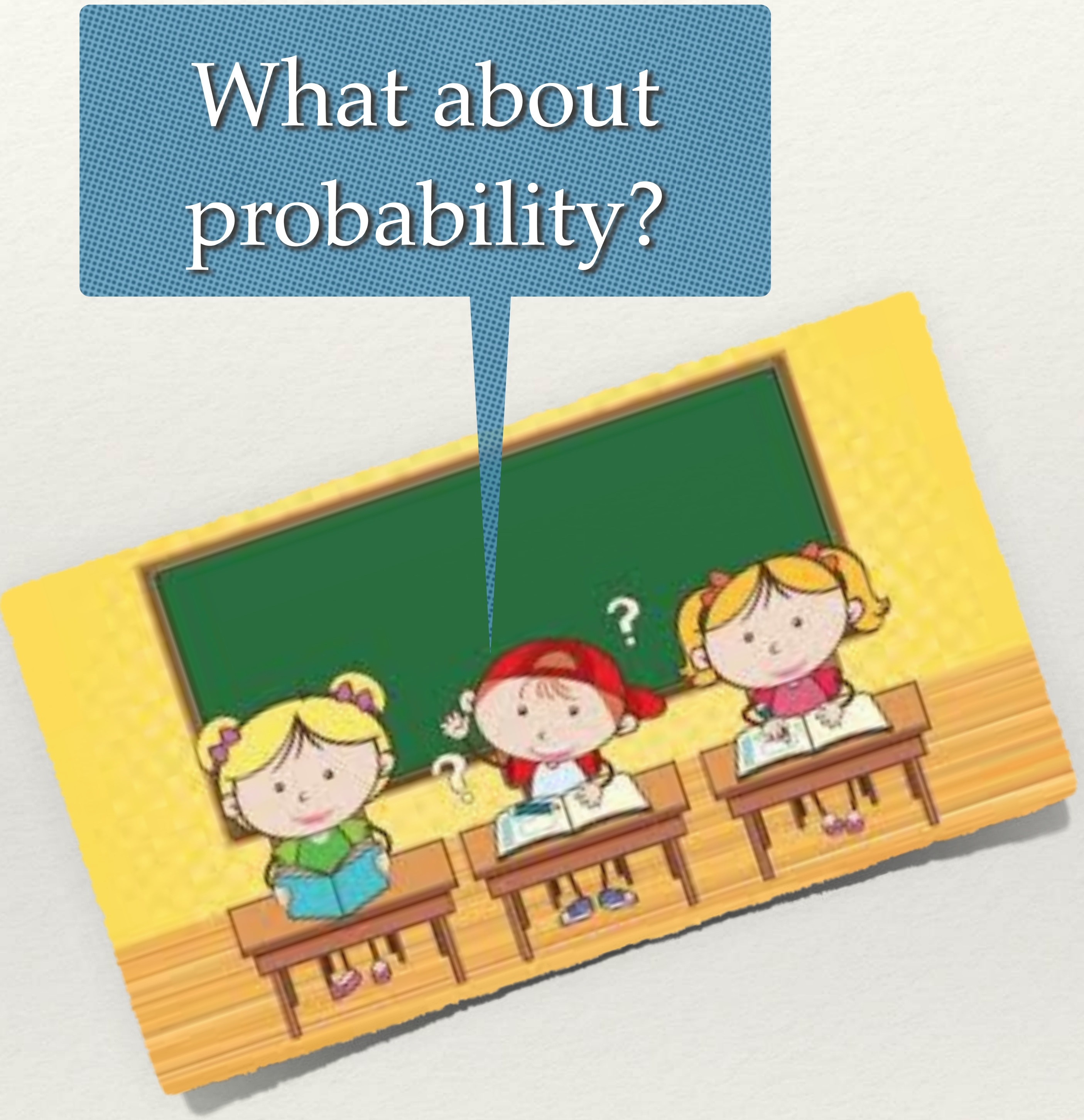
$$I$$

$$\texttt{v:= H} \sqcap \texttt{T;}$$

$$\texttt{g:= Flip()}$$

$$F$$

{ Post Condition}



We can reason about the program by using the abstract datatype and pGCL…

`var coin # Local variable`

I_A: `skip`

$\texttt{Flip}_A$: `# Flip on demand.`

```
        coin:= H
```
$_{1/2}\oplus$ `coin:= T;`

```
    return coin
```

F_A: `skip`

(a) The abstract datatype

{ p }

I

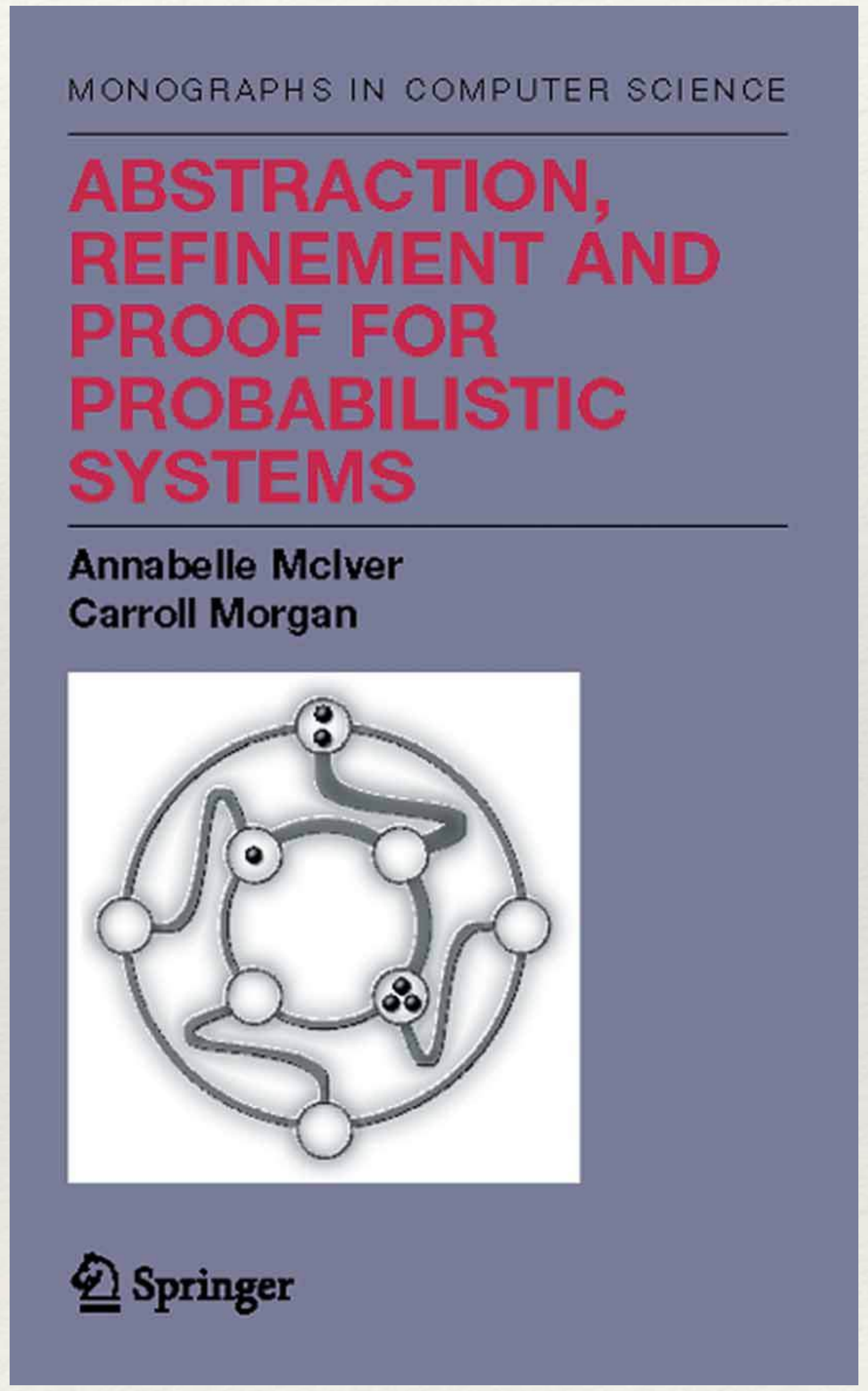
`v:= H` $\sqcap$ `T;`

`g:= Flip()`

F

{[g=v]}

wp.Prog.[g=v] = p



We can reason about the program by using the abstract datatype

I

`v:= H ⊓ T;`

`g:= Flip()`

{[g=v]}

F

{[g=v]}

wp.F.[g=v]

```
var coin # Local variable

I_A:   skip
Flip_A: # Flip on demand.
             coin:= H
        1/2⊕ coin:= T;
        return coin
F_A:   skip
```

(a) The abstract datatype

We can reason about the program by using the abstract datatype

wp."g:= Flip()".[g=v]

I

```
v:= H ⊓ T;
```

{[H=v]/2 + [T=v]/2}

```
g:= Flip()
```

{[g=v]}

F

{[g=v]}

```
var coin # Local variable

I_A:   skip
Flip_A: # Flip on demand.
            coin:= H
       1/2⊕ coin:= T;
       return coin
F_A:   skip
```

(a) The abstract datatype

We can reason about the program by using the abstract datatype

I

{ [H=H]/2 + [T=H]/2 MIN
[H=T]/2 + [T=T]/2 }

`v:= H` $\sqcap$ `T;`

{[H=v]/2 + [T=v]/2}

wp."v:= H$\sqcap$T".[g=v]

```
g:= Flip()
```

{[g=v]}

F

{[g=v]}

We can reason about the program by using the abstract datatype

I

{ 1/2 }

v:= H$_F$ $\sqcap$ T;

{[H=v]/2 + [T=v]/2}

g:= Flip()

{[g=v]}

F

{[g=v]}

We can reason about the program by using the abstract datatype

{ 1/2 }

I

{ 1/2 }

wp.I.[1/2]

`v:= H` $\sqcap$ `T;`

{[H=v]/2 + [T=v]/2}

`g:= Flip()`

{[g=v]}

F

{[g=v]}

var coin # Local variable

I_A: skip

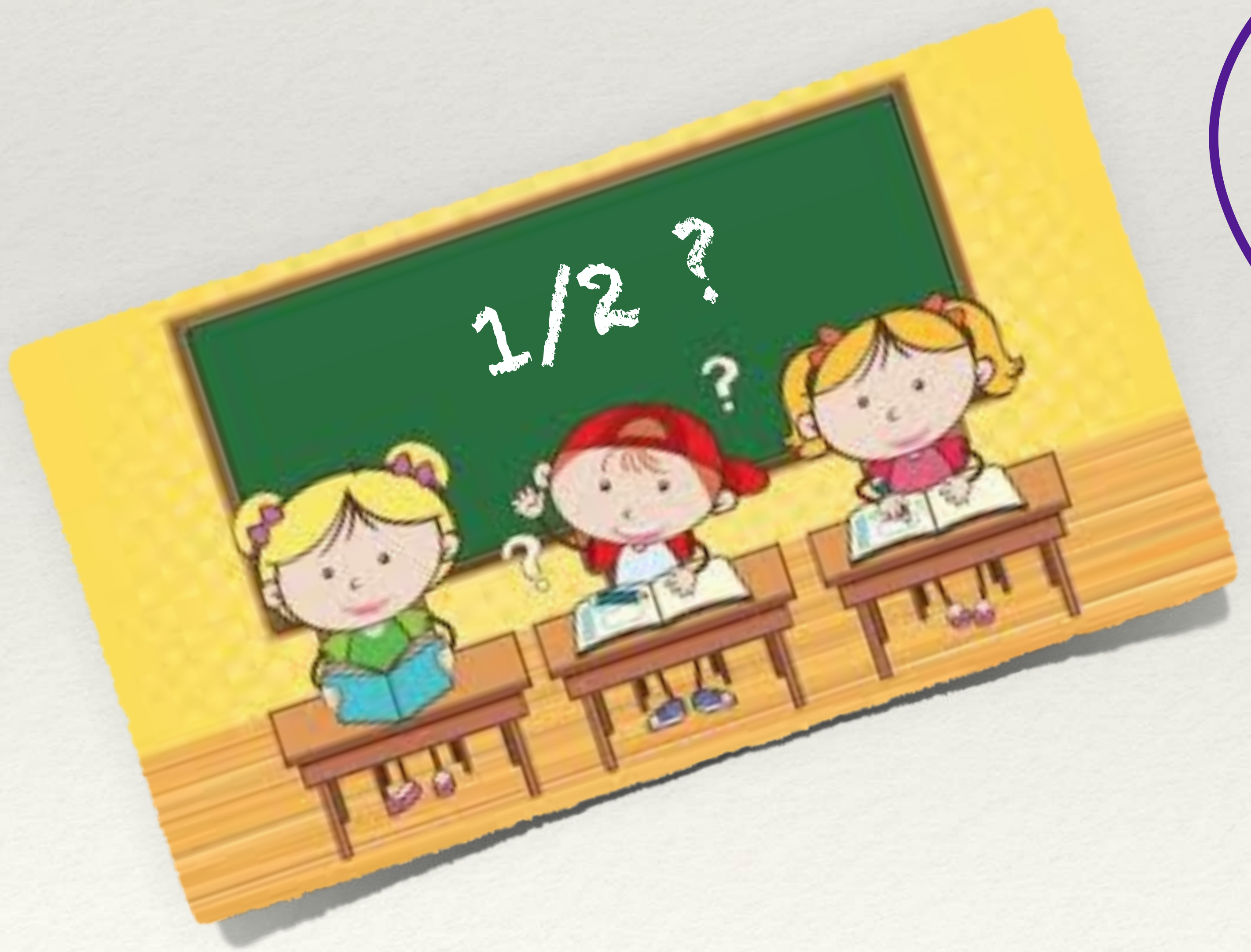
Flip_A: # Flip on demand.
coin:= H
$_{1/2}\oplus$ coin:= T;
return coin

F_A: skip

(a) The abstract datatype

Why is this valid?

wp.Prog.[g=v] = 1/2



{ 1/2 }

I $_F$

```
v:= H ⊓ T;
g:= Flip()
```

F

{[g=v]}

```
var coin # Local variable

I_A:    skip
Flip_A: # Flip on demand.
               coin:= H
          1/2⊕ coin:= T;
          return coin
F_A:    skip
```

(a) The abstract datatype

Justification using the “copy rule”…

The copy rule: inlining the code should mean the same thing

I

```
v:= H⊓T;
g:= Flip()
```

F

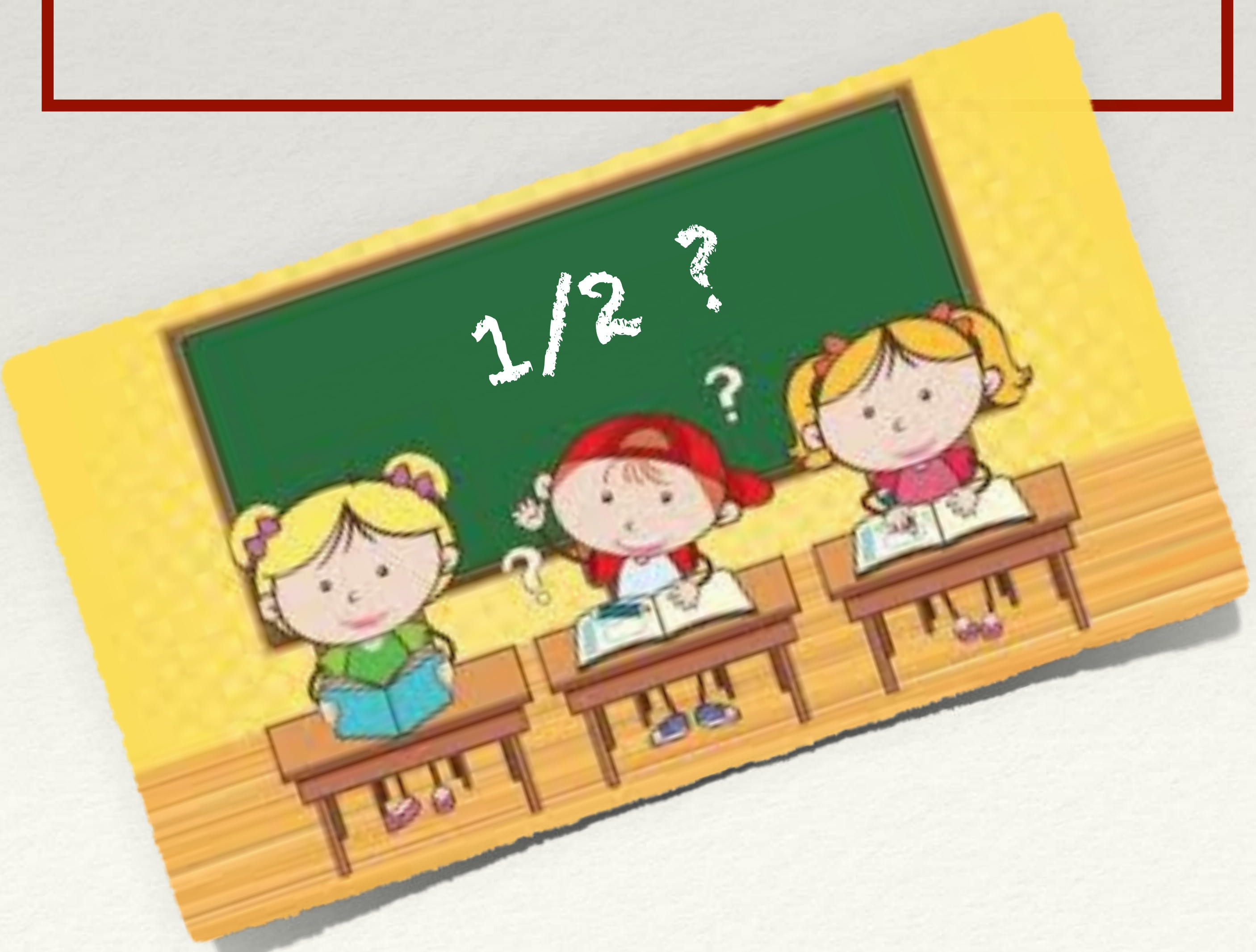
{1/2}

```
skip
v:= H⊓T;
      coin:= H
1/2⊕  coin:= T;
g := coin
skip
```

{[g=v]}

```
var coin # Local variable

I_A:    skip
Flip_A: # Flip on demand.
            coin:= H
       1/2⊕ coin:= T;
        return coin
F_A:    skip
```

(a) The abstract datatype



How does all of this work with refinement of datatypes?

A refinement example

`var coin # Local variable`

I_A: `skip`
$\texttt{Flip}_A$: `# Flip on demand.`
`coin:= H`
${}_{1/2}\oplus$ `coin:= T;`
`return coin`
F_A: `skip`

(a) The abstract datatype

`var coin,c # Local variables`

I_C: `c:=` $\texttt{H}\ {}_{1/2}\!\oplus \texttt{T}$ `# Pre-flip.`
$\texttt{Flip}_C$: `coin:= c;`
`c:=` $\texttt{H}\ {}_{1/2}\!\oplus \texttt{T}$`; # Pre-flip.`
`return coin`
F_C: `skip`

(b) The concrete datatype

Fig. 1: Abstract and concrete probabilistic datatypes

Why should the concrete refine the abstract?

A refinement example

```
var coin # Local variable
```

I_A: skip

$\texttt{Flip}_A$: # Flip on demand.
    coin:= H
  $_{1/2}\oplus$ coin:= T;
  return coin

F_A: skip

(a) The abstract datatype

```
var coin,c # Local variables
```

I_C: c:= H $_{1/2}\oplus$ T # Pre-flip.

$\texttt{Flip}_C$: coin:= c;
  c:= H $_{1/2}\oplus$ T; # Pre-flip.
  return coin

F_C: skip

(b) The concrete datatype

Fig. 1: Abstract and concrete probabilistic datatypes

The abstract datatype uses a “hidden” variable c to store a pre-flipped value — why shouldn’t that matter?

Remember the definition of refinement?

Definition 2. *A datatype* (I, OP, F) *is* refined by (I', OP', F') *[12] if, for every program* $\mathcal{P}$ *expressible using the constructs mentioned above, including calls on corresponding operators in* OP *and* OP' *, we have*

$$I; \mathcal{P}(OP); F \quad \sqsubseteq \quad I'; \mathcal{P}(OP'); F' \ ,$$

where ";" indicates sequential composition.

Demonic nondeterministic choice cannot be resolved on the basis of internal state that it cannot "access" or "observe"

I

```
v:= H⊓T;
g:= Flip()
```

F

In particular whether or not there has been a "pre-flip" of a local variable is not available, even at "run-time"

Remember the definition of refinement?

`var coin # Local variable`

I_A: `skip`

$\mathtt{Flip}_A$: `# Flip on demand.`
`coin:= H` $_{1/2}\oplus$ `coin:= T;`
`return coin`

F_A: `skip`

(a) The abstract datatype

`var coin,c # Local variables`

I_C: `c:= H` $_{1/2}\oplus$ `T # Pre-flip.`

$\mathtt{Flip}_C$: `coin:= c;`
`c:= H` $_{1/2}\oplus$ `T; # Pre-f`
`return coin`

F_C: `skip`

(b) The concrete datatype

Fig. 1: Abstract and concrete probabilistic datatypes

This is fine, because in sequential programs, hidden state can't leak before a function call....

These datatypes should therefore be equivalent in terms of their behaviour, and either one should result in g and v being equal with probability 1/2

I
`v:= H` $\sqcap$ `T;`
`g:= Flip()`
F

Let’s look again at the copy rule, this time for the concrete datatype

I

v:= H ⊓ T;

g:= Flip()

F

0 ?

c:= H $_{1/2}\oplus$ T

v:= H ⊓ T;

coin:= c;

c:= H $_{1/2}\oplus$ T;

g := coin

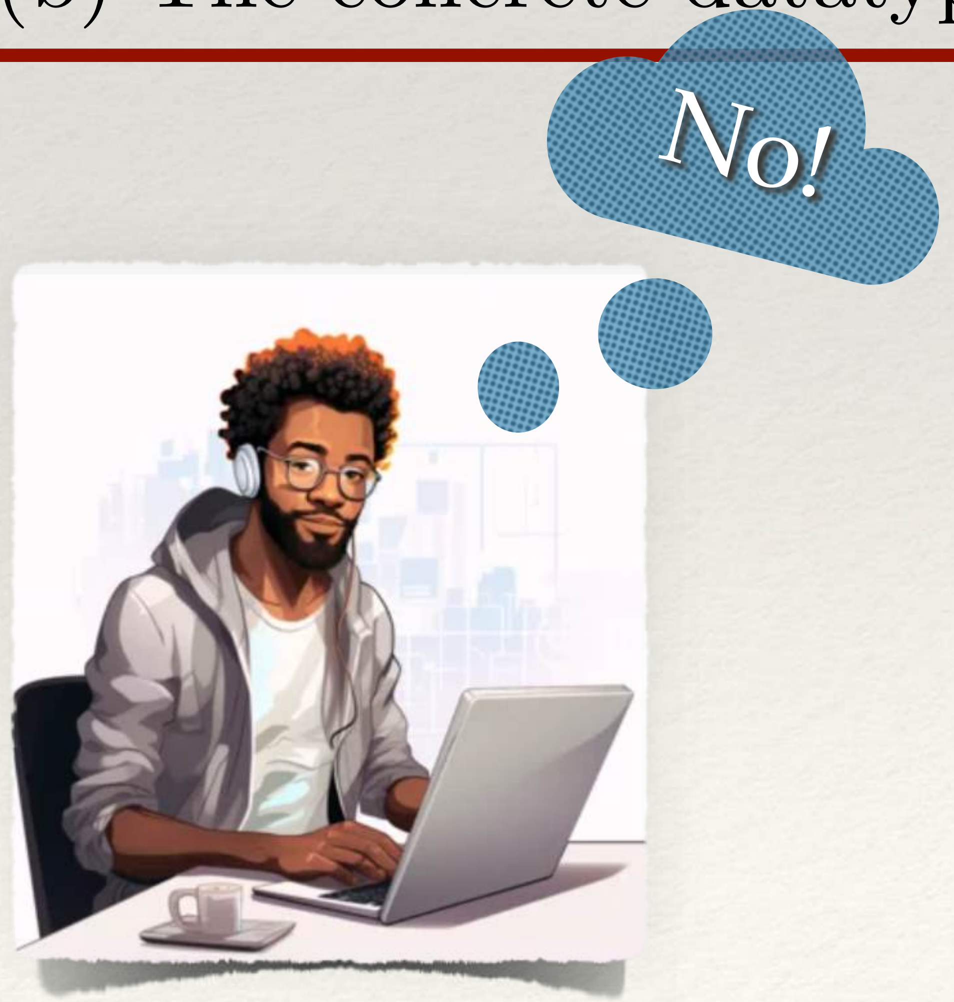
skip

{[g=v]}

In this in-lining, the hidden state is revealed! Is pGCL/MDP the wrong semantics for datatypes?

```
var coin,c # Local variables

I_C :     c:= H 1/2⊕ T # Pre-flip.
Flip_C :  coin:= c;
          c:= H 1/2⊕ T; # Pre-flip.
          return coin

F_C :     skip
```

(b) The concrete datatype



What is the right semantics for probabilistic datatypes?

Let's take another look at refinement

Definition 2. *A datatype* (I, OP, F) *is* refined by (I', OP', F') [12] *if, for every program* $\mathcal{P}$ *expressible using the constructs mentioned above, including calls on corresponding operators in* OP *and* OP'*, we have*

$$I; \mathcal{P}(OP); F \quad \sqsubseteq \quad I'; \mathcal{P}(OP'); F' \ ,$$

where ";" indicates sequential composition.

The observed behaviour of datatypes makes an (unarticulated assumption) that the calling program has "no access" to the run-time internals of the datatype. This doesn't matter when there is no probability, but, as we've seen does when probability is involved.

Simulation is a well-known technique for this situation

Definition 3. *We say that an operation rep is a* simulation *from* (I, OP, F) *to* (I', OP', F') *if –using* $j{\in}J$ *to index corresponding operations in* OP *and* OP'– *the following inequations hold [12]:*

$$I;\mathit{rep} \sqsubseteq I' \tag{1}$$

$$OP_j;\ \mathit{rep} \sqsubseteq \mathit{rep};\ OP'_j \qquad \forall j{\in}J \tag{2}$$

$$F \sqsubseteq \mathit{rep}; F' \tag{3}$$

I

```
v:= H⊓T;
g:= Flip()
```

F

Simulation *should be consistent* with a copy rule, using a semantics that reflects the operational assumptions.

We can "simulate" concrete behaviours with abstract ones

Definition 3. *We say that an operation rep is a* simulation *from* (I, OP, F) *to* (I', OP', F') *if –using* $j \in J$ *to index corresponding operations in* OP *and* OP'*– the following inequations hold [12]:*

$$I;\, rep \sqsubseteq I' \tag{1}$$

$$OP_j;\ rep \sqsubseteq rep;\ OP'_j \qquad \forall j \in J \tag{2}$$

$$F \sqsubseteq rep;\, F' \tag{3}$$

```
var coin # Local variable

I_A:   skip
Flip_A: # Flip on demand.
            coin:= H
       1/2⊕ coin:= T;
         return coin
F_A:   skip
```

(a) The abstract datatype

rep

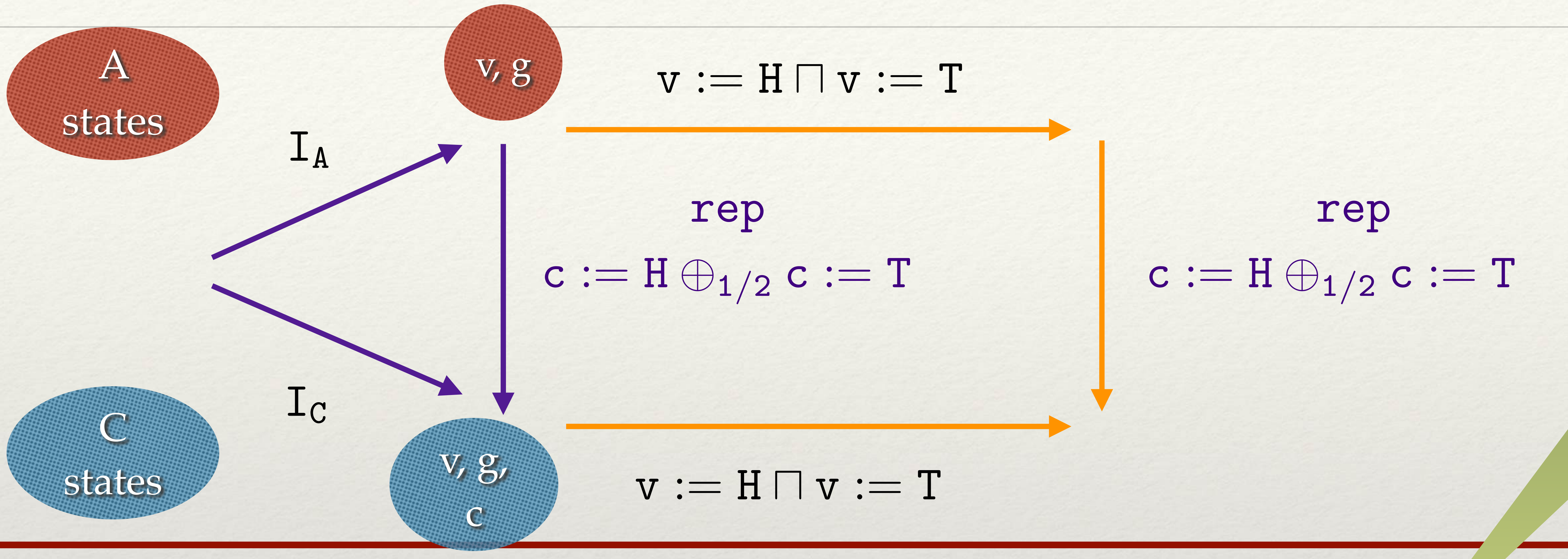
```
var coin,c # Local variables

I_C:    c:= H 1/2⊕ T # Pre-flip.
Flip_C: coin:= c;
        c:= H 1/2⊕ T; # Pre-flip.
        return coin

F_C:    skip
```

(b) The concrete datatype

rep : $\mathsf{c} := \mathsf{H} \oplus_{1/2} \mathsf{c} := \mathsf{T}$

Consistent with the MDP copy rule…?

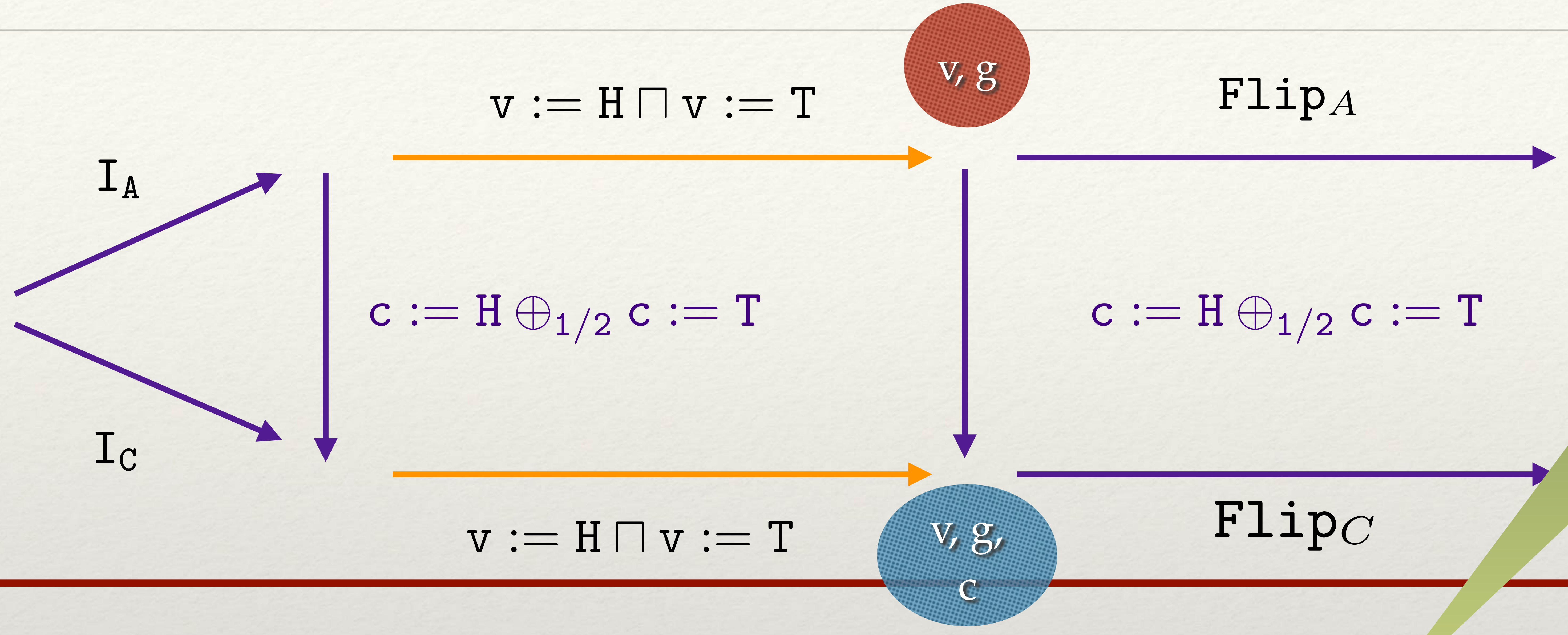
How *should* simulation work?



Glue a series of little proofs together via the rep…

I_A	I_C
v:= H ⊓ T;	v:= H ⊓ T;
g:= $\mathtt{Flip}_A$()	g:= $\mathtt{Flip}_C$()
F_A	F_C

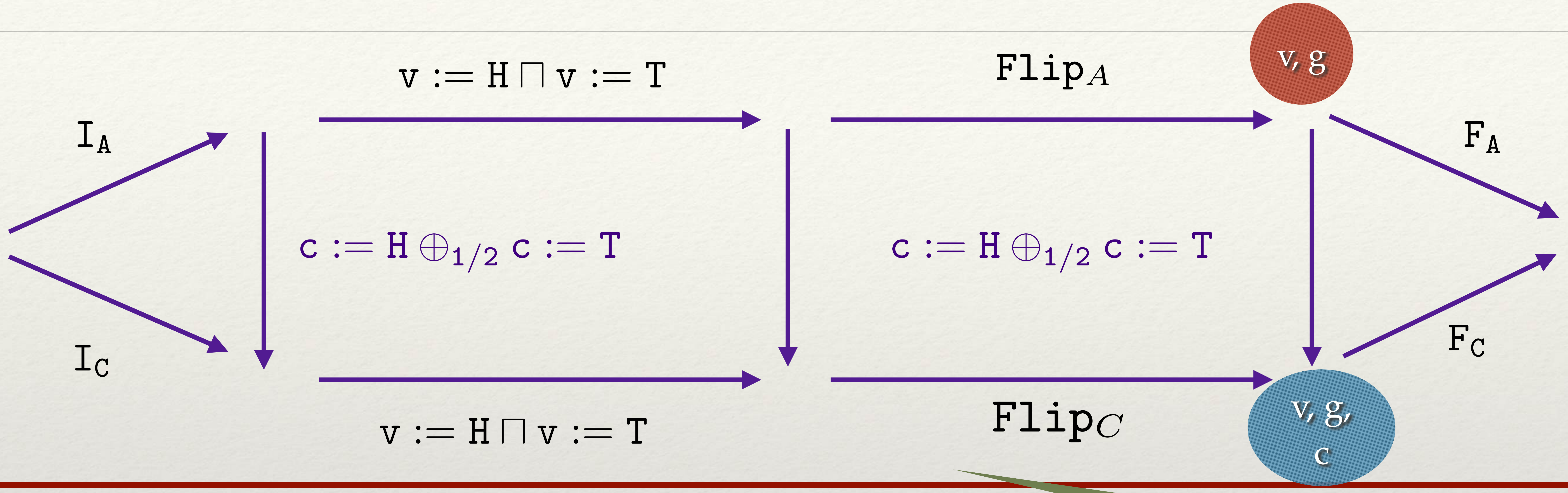
How *should* simulation work?



Glue a series of little proofs together via the rep…

I_A	I_C
v:= H ⊓ T;	v:= H ⊓ T;
g:= Flip$_A$()	g:= Flip$_C$()
F_A	F_C

How *should* simulation work?



I_A
$\mathtt{v := H \sqcap T;}$
$\mathtt{g := Flip_A()}$
F_A

$\sqsubseteq$

I_C
$\mathtt{v := H \sqcap T;}$
$\mathtt{g := Flip_C()}$
F_C

Glue a series of little proofs together via the rep…

How *should* simulation work?

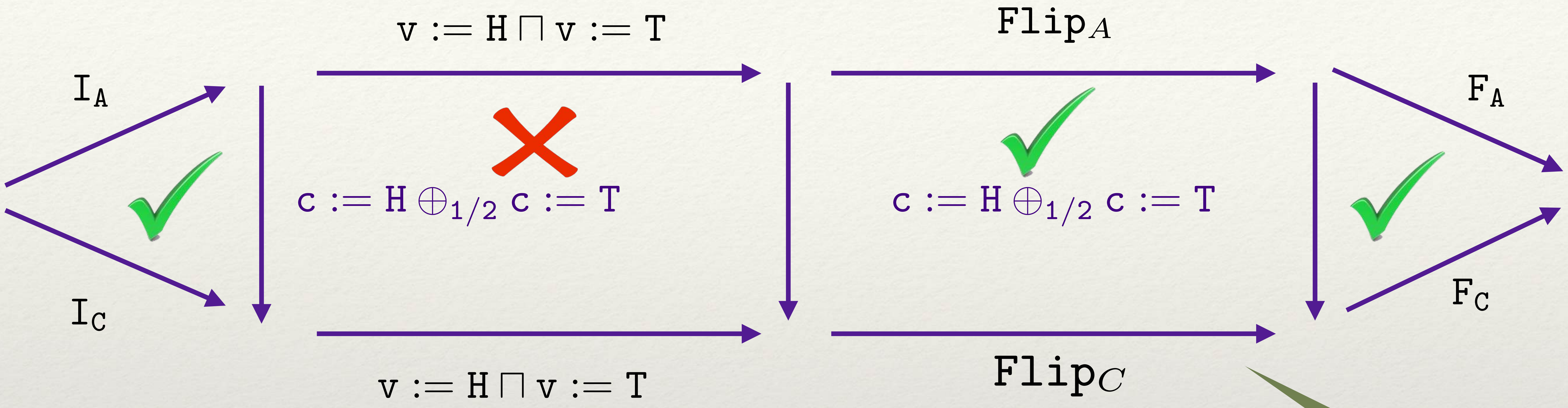
$$\begin{tikzcd}[column sep=huge, row sep=huge] {} \arrow[r, "\mathtt{v} := \mathtt{H} \sqcap \mathtt{v} := \mathtt{T}"] \arrow[d, "\mathtt{c} := \mathtt{H} \oplus_{1/2} \mathtt{c} := \mathtt{T}"] & {} \arrow[d, "\mathtt{c} := \mathtt{H} \oplus_{1/2} \mathtt{c} := \mathtt{T}"] \\ {} \arrow[r, "\mathtt{v} := \mathtt{H} \sqcap \mathtt{v} := \mathtt{T}"'] & {} \end{tikzcd}$$

If this little proof is valid, it means that probability distributes with non-determinisim

$$\begin{array}{c} \mathtt{v} := \mathtt{H} \sqcap \mathtt{v} := \mathtt{T} \\ \mathtt{c} := \mathtt{H} \oplus_{1/2} \mathtt{c} := \mathtt{T} \end{array} \quad \sqsubseteq \quad \begin{array}{c} \mathtt{c} := \mathtt{H} \oplus_{1/2} \mathtt{c} := \mathtt{T} \\ \mathtt{v} := \mathtt{H} \sqcap \mathtt{v} := \mathtt{T} \end{array}$$

But this in NOT valid using an MDP semantics for probability, which does not distinguish hidden statements, i.e. c

How *should* simulation work?



I_A
`v:= H` $\sqcap$ `T;`
`g:= Flip`$_A$`()`
F_A

$\overset{?}{\sqsubseteq}$

I_C
`v:= H` $\sqcap$ `T;`
`g:= Flip`$_C$`()`
F_C

This is not a valid proof of refinement with hidden state and probability…

What do we learn from this?

Definition 3. *We say that an operation rep is a* simulation *from* (I, OP, F) *to* (I', OP', F') *if –using* $j \in J$ *to index corresponding operations in* OP *and* OP'*– the following inequations hold [12]:*

$$I; rep \sqsubseteq I' \tag{1}$$

$$OP_j;\ rep \sqsubseteq rep;\ OP'_j \qquad \forall j \in J \tag{2}$$

$$F \sqsubseteq rep; F' \tag{3}$$

Demonic nondeterministic choice cannot be resolved on the basis of internal state that it cannot "access" or "observe"

I

`v:= H` $\sqcap$ `T;`

`g:= Flip()`

F

MDP's do not support a copy rule that distinguishes hidden (c state) from observed state.

What is the right semantics for probabilistic datatypes so that internal state is "invisible" to external nondeterminism?

Partially Observable Hidden Markov Models!

MDP versus POMDP's

MDP

$\texttt{coin} := \texttt{H} \;{}_{1/2}{\oplus}\; \texttt{coin} := \texttt{T}$

$$\boxed{\mathcal{V} \to \mathbb{P}\mathbb{D}\mathcal{V}}$$

POMDP

$\texttt{c} := \texttt{H} \;{}_{1/2}{\oplus}\; \texttt{T}$

$$\boxed{\mathcal{V} \times \mathbb{D}\mathcal{H} \to \mathbb{P}\mathbb{D}(\mathcal{V} \times \mathbb{D}\mathcal{H})}$$

The abstract datatype has a visible coin to flip; the concrete datatype has a hidden c to flip

MDP versus POMDP's

MDP

$$\texttt{coin} := \texttt{H} \;\; {}_{1/2}\!\oplus \;\; \texttt{coin} := \texttt{T}$$

$$\texttt{v} := 0 \;\; \sqcap \;\; \texttt{v} := 1$$

$$\boxed{\mathcal{V} \;\; \to \mathbb{P}\mathbb{D}\mathcal{V}}$$

POMDP

$$\texttt{c} := \texttt{H} \;\; {}_{1/2}\!\oplus \;\; \texttt{T}$$

$$\texttt{v} := 0 \;\; \sqcap \;\; \texttt{v} := 1$$

$$\boxed{\mathcal{V} \times \mathbb{D}\mathcal{H} \;\; \to \mathbb{P}\mathbb{D}(\mathcal{V} \times \mathbb{D}\mathcal{H})}$$

External nondeterminism cannot "observe" hidden coin flips inside the module

MDP versus POMDP's

MDP

$\mathtt{v} := 0 \;\sqcap\; \mathtt{v} := 1$

$\mathtt{coin} := \mathtt{H} \;{}_{1/2}\!\oplus\; \mathtt{coin} := \mathtt{T}$

$$\boxed{\mathcal{V} \quad \to \mathbb{P}\mathbb{D}\mathcal{V}}$$

POMDP

$\mathtt{v} := 0 \;\sqcap\; \mathtt{v} := 1$

$\mathtt{c} := \mathtt{H} \;{}_{1/2}\!\oplus\; \mathtt{T}$

$$\boxed{\mathcal{V} \times \mathbb{D}\mathcal{H} \quad \to \mathbb{P}\mathbb{D}(\mathcal{V} \times \mathbb{D}\mathcal{H})}$$

External nondeterminism cannot "observe" hidden coin flips inside the module

MDP versus POMDP's

MDP

$$\boxed{\mathcal{V} \quad \to \mathbb{PD}\mathcal{V}}$$

$$\mathtt{coin} := \mathtt{H} \ {}_{1/2}\!\oplus \ \mathtt{coin} := \mathtt{T}$$

$$\mathtt{v} := 0 \ \sqcap \ \mathtt{v} := 1$$

$$\left\{ \begin{array}{l} \mathtt{u}[\mathtt{coin} = \mathtt{H} \wedge \mathtt{v} = 0, \mathtt{coin} = \mathtt{T} \wedge \mathtt{v} = 1] \\ \mathtt{u}[\mathtt{coin} = \mathtt{T} \wedge \mathtt{v} = 0, \mathtt{coin} = \mathtt{H} \wedge \mathtt{v} = 1] \\ \mathtt{u}[\mathtt{coin} = \mathtt{T} \wedge \mathtt{v} = 0, \mathtt{coin} = \mathtt{H} \wedge \mathtt{v} = 0] \end{array} \right\}$$

$$\mathtt{u}[\mathtt{coin} = \mathtt{T} \wedge \mathtt{v} = 1, \mathtt{coin} = \mathtt{H} \wedge \mathtt{v} = 1]$$

POMDP

$$\boxed{\mathcal{V} \times \mathbb{D}\mathcal{H} \quad \to \mathbb{PD}(\mathcal{V} \times \mathbb{D}\mathcal{H})}$$

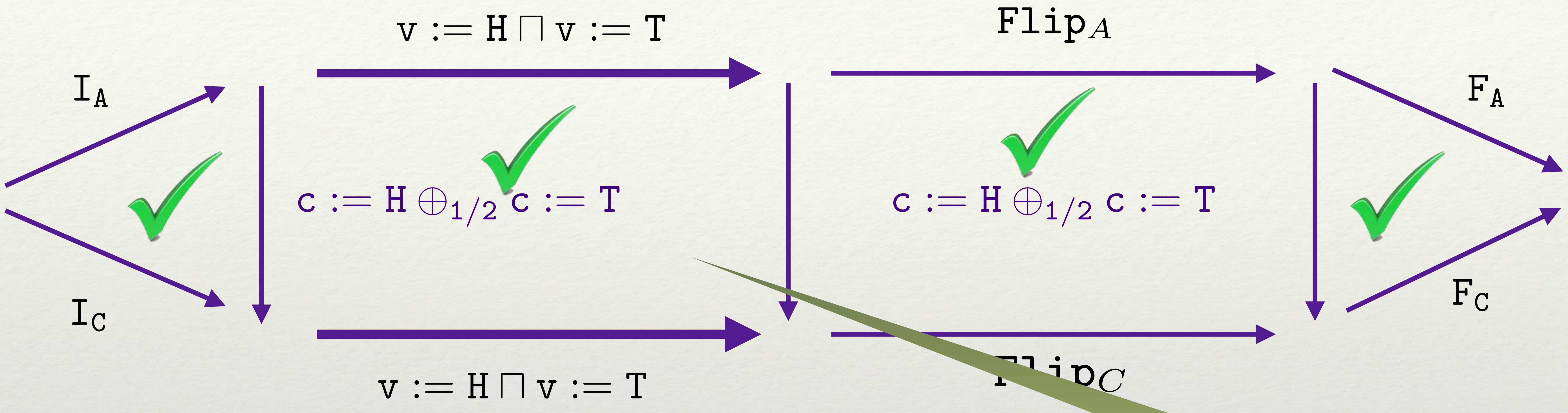
$$\mathtt{c} := \mathtt{H} \ {}_{1/2}\!\oplus \ \mathtt{T}$$

$$\mathtt{v} := 0 \ \sqcap \ \mathtt{v} := 1$$

$$\left\{ \begin{array}{l} \mathtt{v} = 0 \wedge \mathtt{u}[\mathtt{c} = \mathtt{T}, \mathtt{c} = \mathtt{H}] \\ \mathtt{v} = 1 \wedge \mathtt{u}[\mathtt{c} = \mathtt{T}, \mathtt{c} = \mathtt{H}] \end{array} \right\}$$

External nondeterminism cannot "observe" hidden coin flips inside the module

Simulation now works in POMDP's, consistent with copy rule!



I_A
v:= H⊓T;
g:= Flip$_A$()
F_A

$\sqsubseteq$

I_C
v:= H⊓T;
g:= Flip$_C$()
F_C

This little proof is now valid!

What does this mean for these developers?

I have to be explicit about the local variables and ensure that their state is not revealed early.

```
var coin # Local variable

I_A:    skip
Flip_A: # Flip on demand.
              coin:= H
        1/2⊕ coin:= T;
         return coin
F_A:    skip
```

(a) The abstract datatype

```
var coin,c # Local variable

I_C:      c:= H 1/2⊕ T # Pre-flip.
Flip_C:   coin:= c;
          c:= H 1/2⊕ T; # Pre-flip.
          return coin

F_C:      skip
```

(b) The concrete datatype

Fig. 1: Abstract and concrete probabilistic datatypes

Why should the concrete refine the abstract?

A small cadenza…

- Probabilistic invariants are sometimes simulation relations, and
-
- Refinement depends on run-time information leaks concerning about the hidden state

Definition 2. *A datatype* (I, OP, F) *is* refined by (I', OP', F') *[12] if, for every program* $\mathcal{P}$ *expressible using the constructs mentioned above, including calls on corresponding operators in* OP *and* OP' *, we have*

$$I; \mathcal{P}(OP); F \quad \sqsubseteq \quad I'; \mathcal{P}(OP'); F' \ ,$$

where ";" indicates sequential composition.

How does this work for our challenge problem?

```
{ 1/N } # Precondition
var v, t= 1, 0

while(¬t){
    □ (v≤N) → v= 2v
      if (v>N) then t= 1 N/v⊕ skip

    □ (v>N) → v= v-N
}
c= unif[0, N)
{ [c = i] } # Post condition for any 0 ≤ i < N
```

….simulates….

```
{ 1/N } # Precondition
var c, v = 0, 1
while(v<N or c≥N){
    {Inv × [v<N or c≥N]}
    □ (v ≤ N) → v= 2v
            c= 2c 1/2⊕ c= 2c+1
    □ (v > N) → v,c= v-N,c-N
    { Inv }
}
{ [c = i] } # Post
```

Caesar

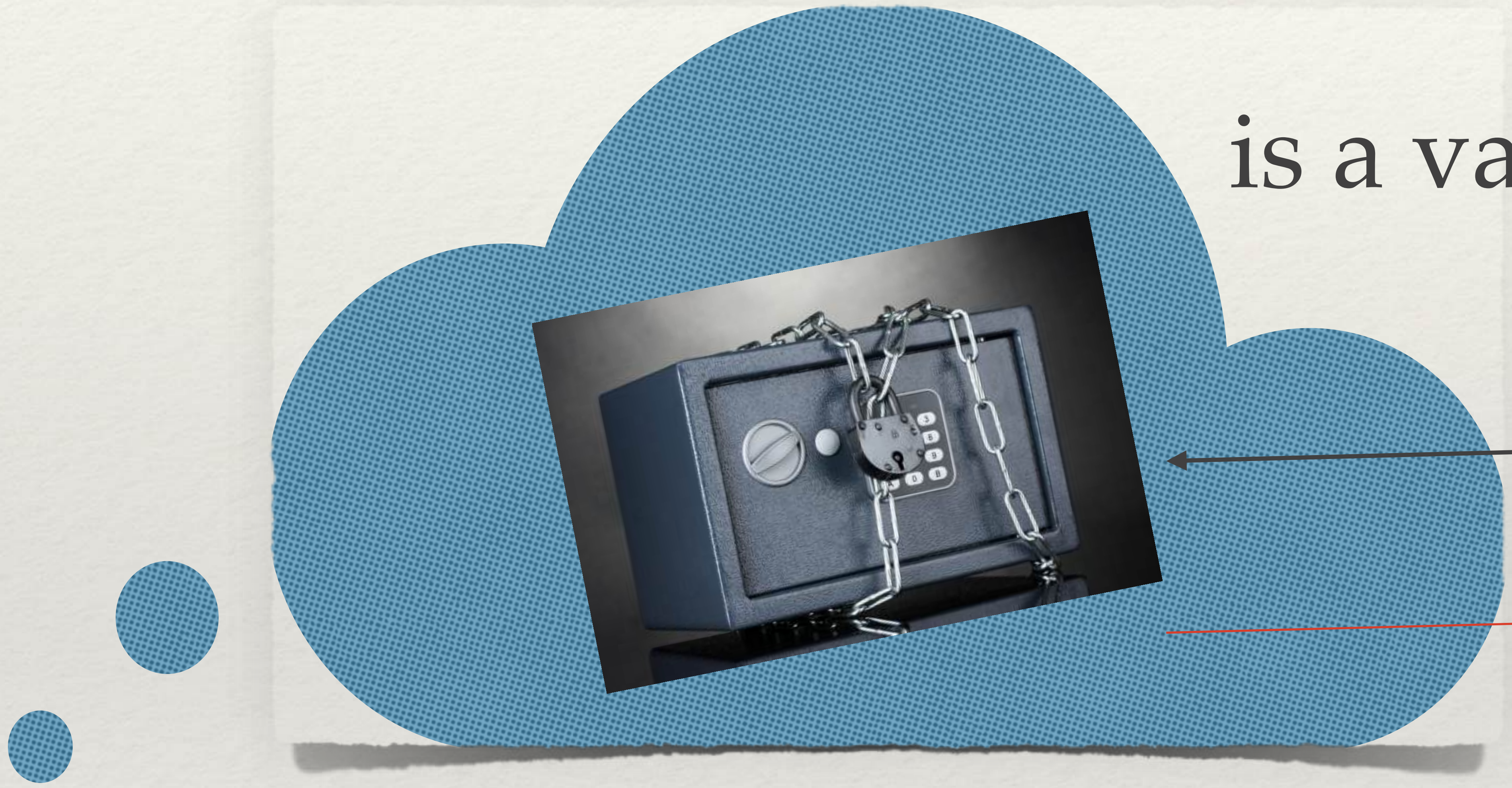
A Deductive Verifier for Probabilistic Programs

Developed at RWTH Aachen

www.caesarverifier.org/

- LHS can be fully verified automatically to set c uniformly;
- Simulation relations can be verified automatically; they encapsulate compactly probabilistic invariant properties, similar to conditional reasoning, unlike the {inv} assertion style.
- Simulation means that RHS satisfies the properties of LHS.

We can model things like secure implementation of cloud storage…



is a value in the database?

Answer should only return “yes/no” without leaking any other information.

Conclusions for today

- What happens when some of the behaviour can be probabilistic in sequential programs?
- Do the traditional proof methods for sequential programs (eg simulation) still work?

$$\begin{tikzcd}
((g,v),-) \arrow[r, "\mathtt{v:=\ 0 \sqcap 1}"] \arrow[d, "\mathtt{c:=\ 0}\ {}_{1/2}\oplus\ 1"'] & \{((g,0),-),((g,1),-)\} \arrow[d, "\mathtt{c:=0}\ {}_{1/2}\oplus\ 1"] \\
((g,v),\mathtt{u}\{0,1\}) \arrow[r, "\mathtt{v:=\ 0 \sqcap 1}"] & \{((g,0),\mathtt{u}\{0,1\}),((g,1),\mathtt{u}\{0,1\})\}
\end{tikzcd}$$

- Can we still use the abstract specification to prove properties of programs that use datatypes?
- If they don't, what must be changed?

$$\boxed{\mathcal{V} \times \mathbb{D}\mathcal{H} \quad \rightarrow \mathbb{P}\mathbb{D}(\mathcal{V} \times \mathbb{D}\mathcal{H})}$$

  - A semantics and refinement that distinguishes hidden and visible state;
  - We can now talk about information leaks.

Developed at RWTH Aachen

www.caesarverifier.org/