# Local reasoning for robust observational equivalence

Koko Muroya
(RIMS, Kyoto University)

joint work with
Dan R. Ghica & Todd Waugh Ambridge
(University of Birmingham)

LFCS Seminar (online), 04 December 2020

# Overview

1. Motivation: robustness of observational equivalence

2. Hypernet semantics

3. Locality & step-wise reasoning

4. Example: cbv linear β-law

# Overview

1. Motivation: robustness of observational equivalence

2. Hypernet semantics

3. Locality & step-wise reasoning

4. Example: cbv linear β-law

# Observational equivalence on program fragments

*“Do two program fragments behave the same?”*

*“Is it safe to replace a program fragment with another?”*

```
let x = 100 in
let y = 50 in
y + y
```

# Observational equivalence on program fragments

*“Do two program fragments behave the same?”*

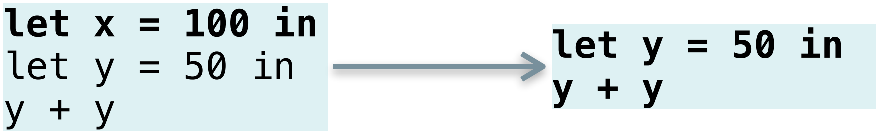
*“Is it safe to replace a program fragment with another?”*

```
let x = 100 in
let y = 50 in
y + y
```

# Observational equivalence on program fragments

*“Do two program fragments behave the same?”*

*“Is it safe to replace a program fragment with another?”*



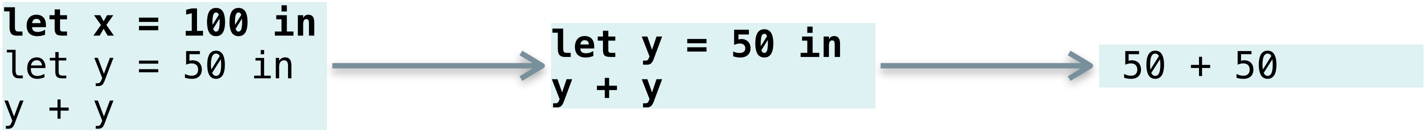
```
let x = 100 in
let y = 50 in
y + y
```

```
let y = 50 in
y + y
```

# Observational equivalence on program fragments

*“Do two program fragments behave the same?”*

*“Is it safe to replace a program fragment with another?”*



```
let x = 100 in
let y = 50 in
y + y
```

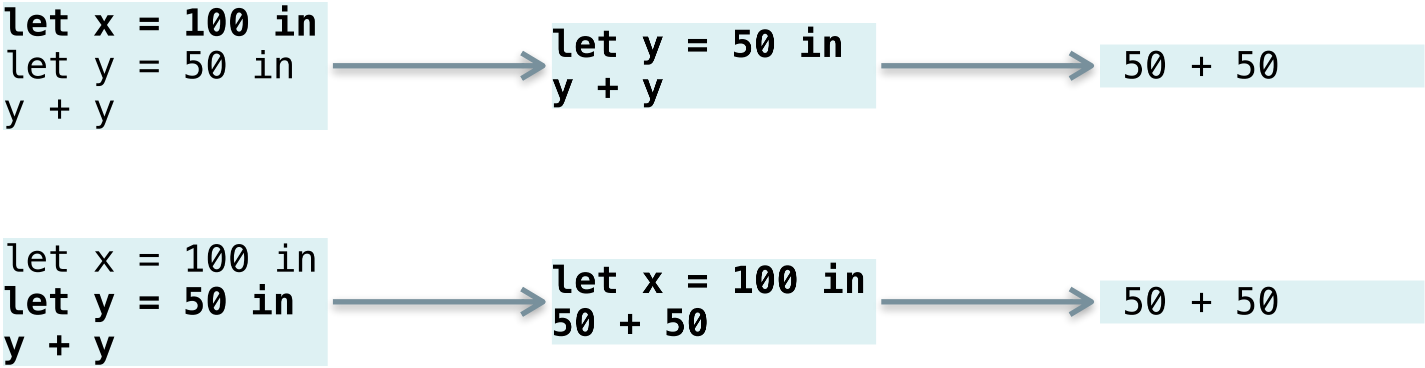
```
let y = 50 in
y + y
```

```
50 + 50
```

# Observational equivalence on program fragments

*“Do two program fragments behave the same?”*

*“Is it safe to replace a program fragment with another?”*



```
let x = 100 in
let y = 50 in
y + y
```

```
let y = 50 in
y + y
```

```
50 + 50
```

```
let x = 100 in
let y = 50 in
y + y
```

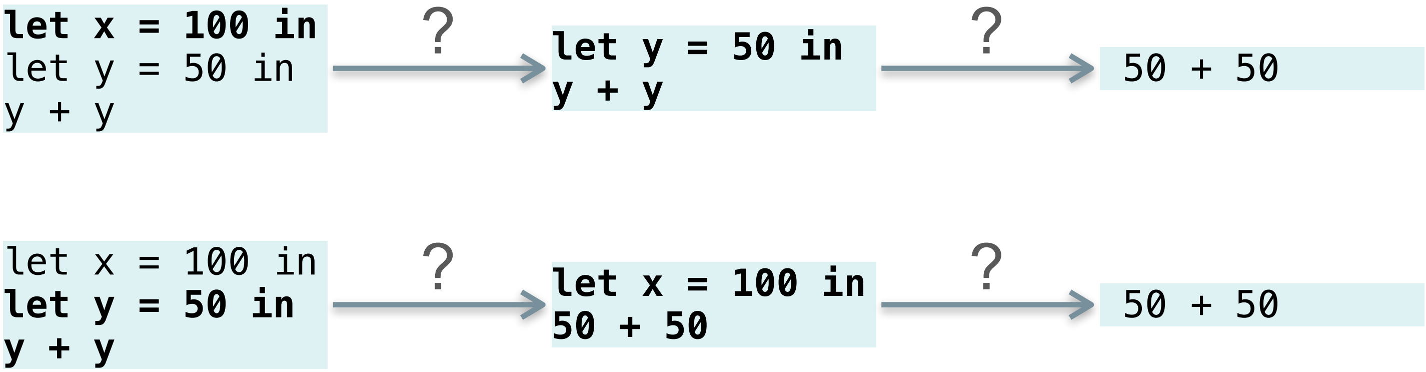
```
let x = 100 in
50 + 50
```

```
50 + 50
```

# Observational equivalence on program fragments

*“Do two program fragments behave the same?”*

*“Is it safe to replace a program fragment with another?”*



```
let x = 100 in
let y = 50 in
y + y
```

```
let y = 50 in
y + y
```

```
50 + 50
```

```
let x = 100 in
let y = 50 in
y + y
```

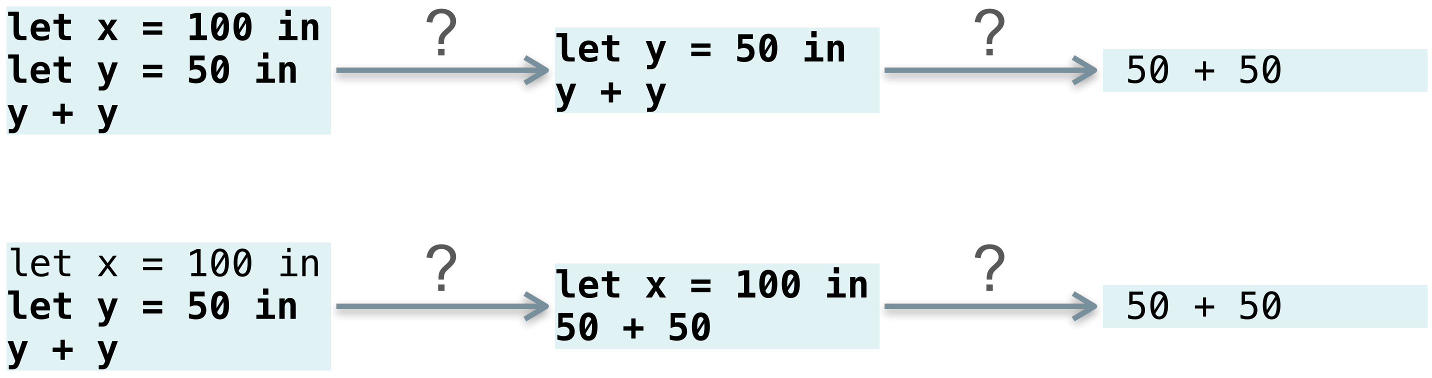
```
let x = 100 in
50 + 50
```

```
50 + 50
```

# Observational equivalence on program fragments

*“Do two program fragments behave the same?”*

*“Is it safe to replace a program fragment with another?”*



```
let x = 100 in
let y = 50 in
y + y
```

```
let y = 50 in
y + y
```

```
50 + 50
```

```
let x = 100 in
let y = 50 in
y + y
```

```
let x = 100 in
50 + 50
```

```
50 + 50
```

If YES (*“Two program fragments are observationally equal.”*):

- justification of compiler optimisation
- program verification

# Observational equivalence on program fragments

*“Do two program fragments behave the same?”*

# Observational equivalence on program fragments

*"~~Do two~~ program fragments behave the same?"*

*"**What** program fragments behave the same?"*

the beta-law

$$(\lambda x\,.\,M)\,N \;\simeq\; M[x := N]$$

a parametricity law

$$\texttt{let}\ a = \texttt{ref}\ 1\ \texttt{in}\ \lambda x\,.\,(a := 2;\, !a) \;\simeq\; \lambda x.2$$

# Robustness of observational equivalence

*“~~Do~~ two program fragments behave the same?”*

*“**When do** program fragments behave the same?”*

the beta-law

$$(\lambda x\,.\,M)\,N \;\simeq\; M[x := N]$$

Does the beta-law always hold?

# Robustness of observational equivalence

*“~~Do~~ two program fragments behave the same?”*

*“**When do** program fragments behave the same?”*

the beta-law

$$(\lambda x\,.\,M)\,N \;\simeq\; M[x := N]$$

Does the beta-law always hold?

**No**, it’s violated if program contexts use OCaml’s Gc module:

$$(\lambda x.0)\,100 \;\not\simeq\; 0$$

for memory management

# Robustness of observational equivalence

*"~~Do~~ two program fragments behave the same?"*

*"**When do** program fragments behave the same?"*

the beta-law

$$(\lambda x\,.\,M)\,N \ \simeq\ M[x := N]$$

Does the beta-law always hold?

**No**, it's violated if program contexts use OCaml's Gc module:

$$(\lambda x.0)\,100 \ \not\simeq\ 0$$

for memory management

How **robust** is the beta-law then?

# Robustness of observational equivalence

~~*“Do two program fragments behave the same?”*~~

*“What fragments, **in which contexts,** behave the same?”*

# Robustness of observational equivalence

~~*"Do two program fragments behave the same?"*~~

*"What fragments, **in which contexts,** behave the same?"*

… in the presence of (arbitrary) language features:

pure vs. effectful (e.g. `50 + 50` vs. `ref 1`)

encoded vs. native (e.g. `State` vs. `ref`)

extrinsics (e.g. `Gc.stat`)

foreign language calls

# Robustness of observational equivalence

~~*“Do two program fragments behave the same?”*~~

*“What fragments, **in which contexts,** behave the same?”*

… in the presence of (arbitrary) language features

Our (big) goal:

analysing robustness/fragility of observational equivalence,

using a general framework

# Robustness of observational equivalence

~~*“Do two program fragments behave the same?”*~~

*“What fragments, **in which contexts,** behave the same?”*

… in the presence of (arbitrary) language features

Our result:

analysing robustness/fragility of observational equivalence,

using <u>a *graphical* framework</u>

- hypernet semantics: a *graphical* abstract machine
- *local & step-wise* reasoning to prove observational equivalence, with the concept of *robustness*

# Overview

1. Motivation: robustness of observational equivalence

2. Hypernet semantics

3. Locality & step-wise reasoning

4. Example: cbv linear β-law

# Hypernet semantics

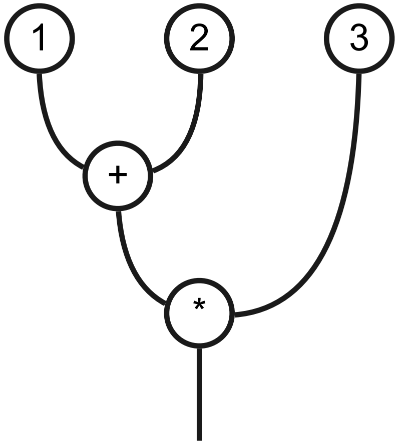
- program execution by a *graphical* abstract machine
    - programs as

      certain hierarchical hypergraphs (*“hypernets”*)
    - execution as

      step-by-step strategical update of hypernets

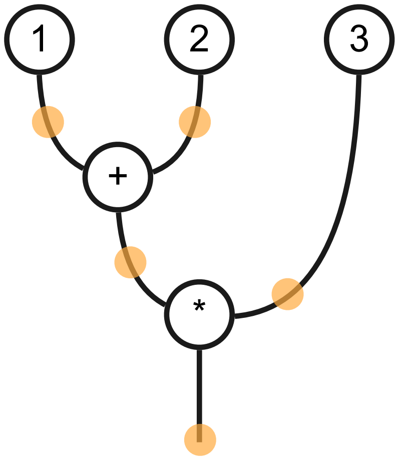
# Programs, graphically as *hypernets*

Idea: abstracting away variable names, and more…

| program | hypernet (hierarchical hypergraph) |
|---|---|
| `(1 + 2) * 3` |  |

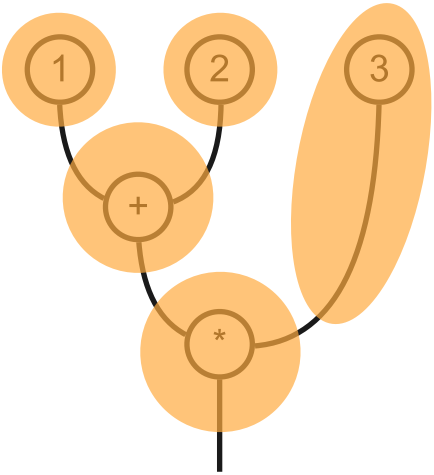
# Programs, graphically as *hypernets*

Idea: abstracting away variable names, and more…

| program | hypernet (hierarchical hypergraph) |
|---|---|
| `(1 + 2) * 3` | nodes<br> |

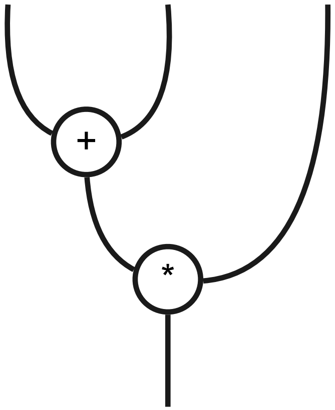
# Programs, graphically as *hypernets*

Idea: abstracting away variable names, and more…

| program | hypernet (hierarchical hypergraph) |
|---|---|
| `(1 + 2) * 3` | hyperedges<br> |

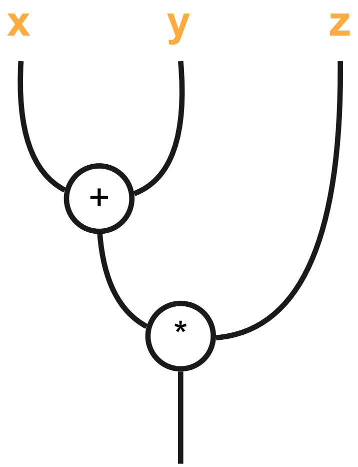
# Programs, graphically as *hypernets*

Idea: abstracting away variable names, and more…

| program | hypernet (hierarchical hypergraph) |
|---|---|
| `(x + y) * z`<br>`(i + j) * k` |  |

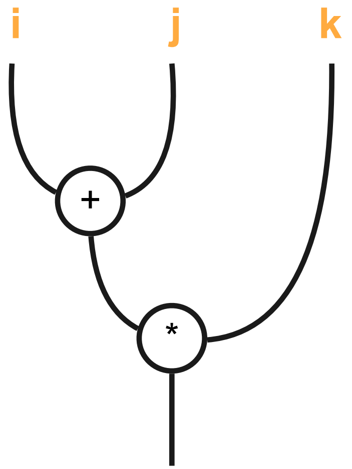
# Programs, graphically as *hypernets*

Idea: abstracting away variable names, and more…

| program | hypernet (hierarchical hypergraph) |
|---|---|
| `(x + y) * z`<br>`(i + j) * k` |  |

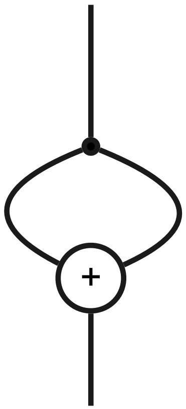
# Programs, graphically as *hypernets*

Idea: abstracting away variable names, and more…

| program | hypernet (hierarchical hypergraph) |
|---|---|
| `(x + y) * z`<br>`(i + j) * k` |  |

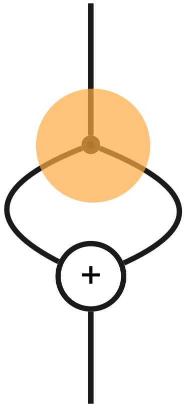
# Programs, graphically as *hypernets*

Idea: abstracting away variable names, and more…

| program | hypernet (hierarchical hypergraph) |
|---|---|
| `x + x` |  |

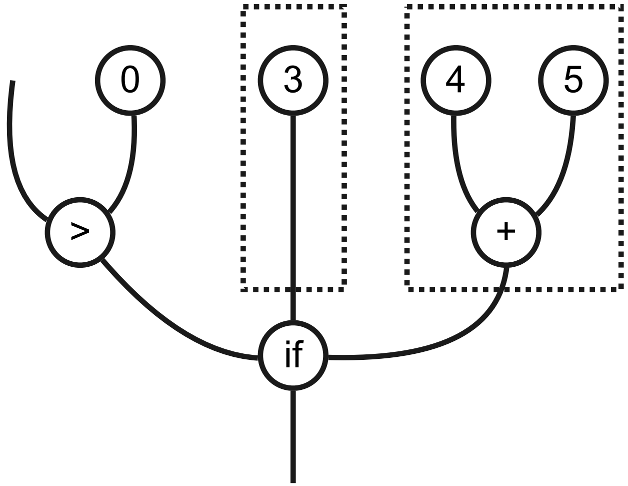
# Programs, graphically as *hypernets*

Idea: abstracting away variable names, and more…

| program | hypernet (hierarchical hypergraph) |
|---|---|
| `x + x` | sharing<br><br> |

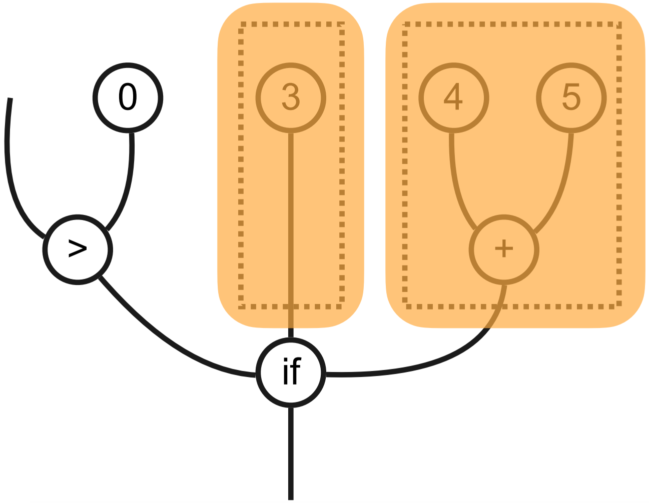
# Programs, graphically as *hypernets*

Idea: abstracting away variable names, and more…

| program | hypernet (hierarchical hypergraph) |
|---|---|
| `if x > 0`<br>`then 3`<br>`else 4 + 5` |  |

# Programs, graphically as *hypernets*

Idea: abstracting away variable names, and more…

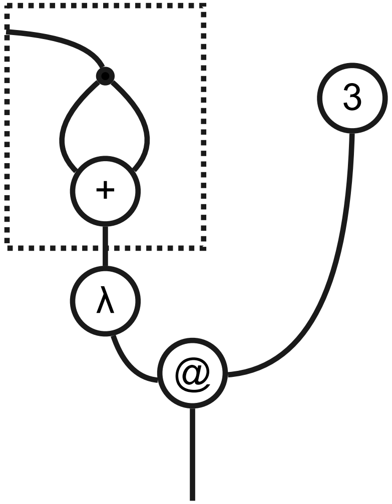
| program | hypernet (hierarchical hypergraph) |
|---|---|
| `if x > 0`<br>`then 3`<br>`else 4 + 5` |  |

hierarchical hyperedge
(hyperedge labelled with hypergraph)

…representing *deferred* computation

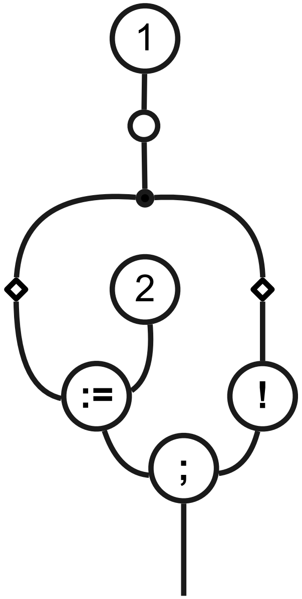
# Programs, graphically as *hypernets*

Idea: abstracting away variable names, and more…

| program | hypernet (hierarchical hypergraph) |
|---|---|
| `(λx. x + x) 3` |  |

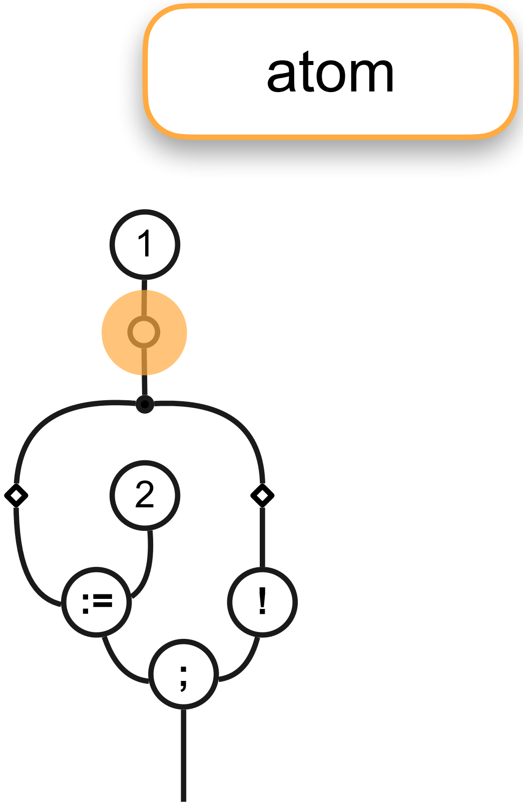
# Programs, graphically as *hypernets*

Idea: abstracting away variable names, and more…

| program | hypernet (hierarchical hypergraph) |
|---|---|
| `new a = 1 in`<br>`a := 2; !a` |  |

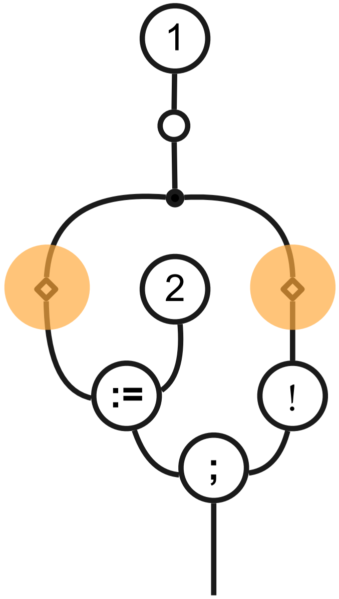
# Programs, graphically as *hypernets*

Idea: abstracting away variable names, and more…

| program | hypernet (hierarchical hypergraph) |
|---|---|
| `new a = 1 in`<br>`a := 2; !a` |  |

# Programs, graphically as *hypernets*

Idea: abstracting away variable names, and more…

| program | hypernet (hierarchical hypergraph) |
|---|---|
| `new a = 1 in`<br>`a := 2; !a` |  atom<br>occurences |

# Programs, graphically as *hypernets*

Idea: abstracting away variable names, and more…

# Programs, graphically as *hypernets*

Idea: abstracting away variable names, and ~~more…~~

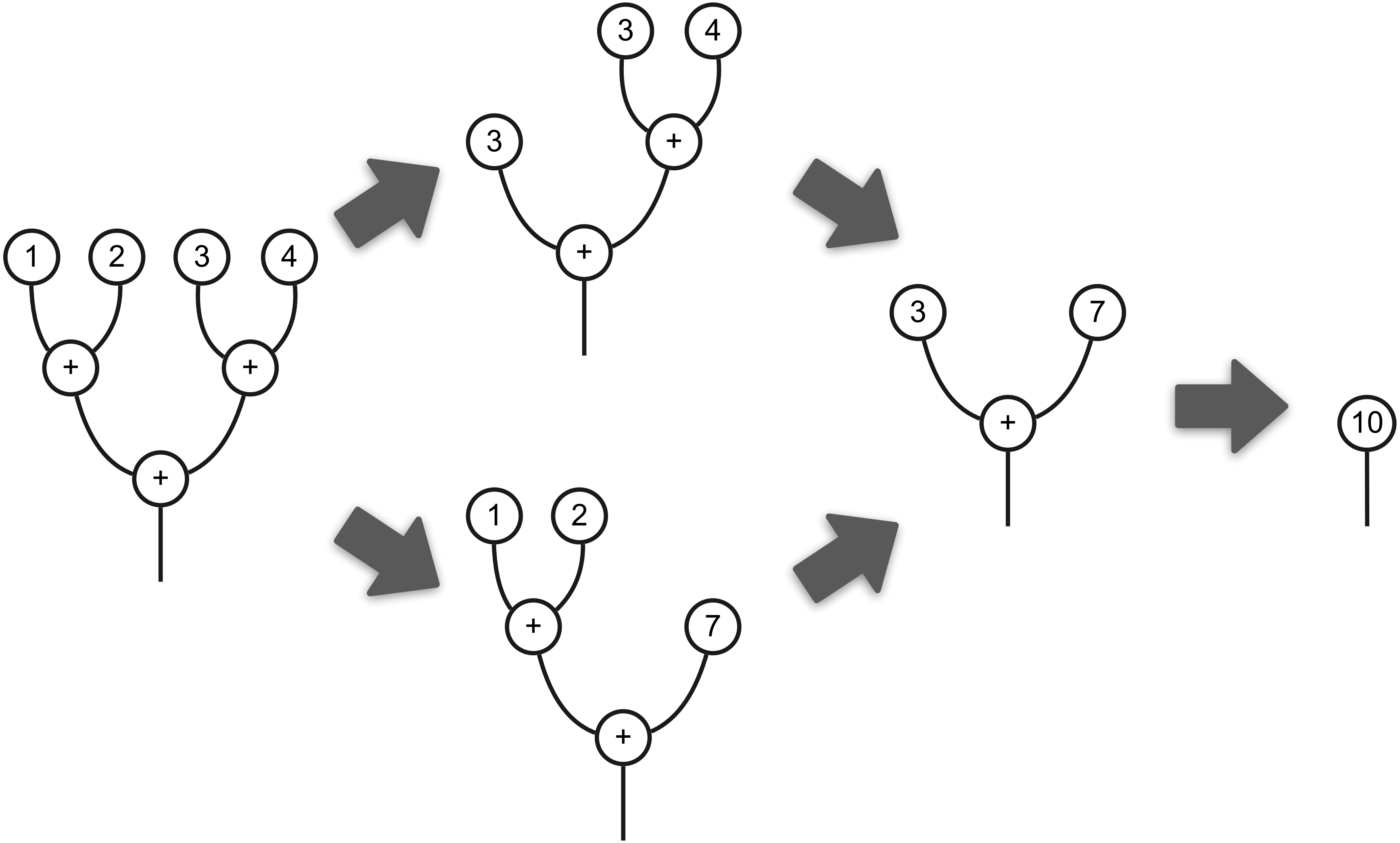
- making blocks of deferred computation explicit
- accommodating atoms (reference names/locations)

# Program execution, graphically

Idea: updating hypernets step-by-step

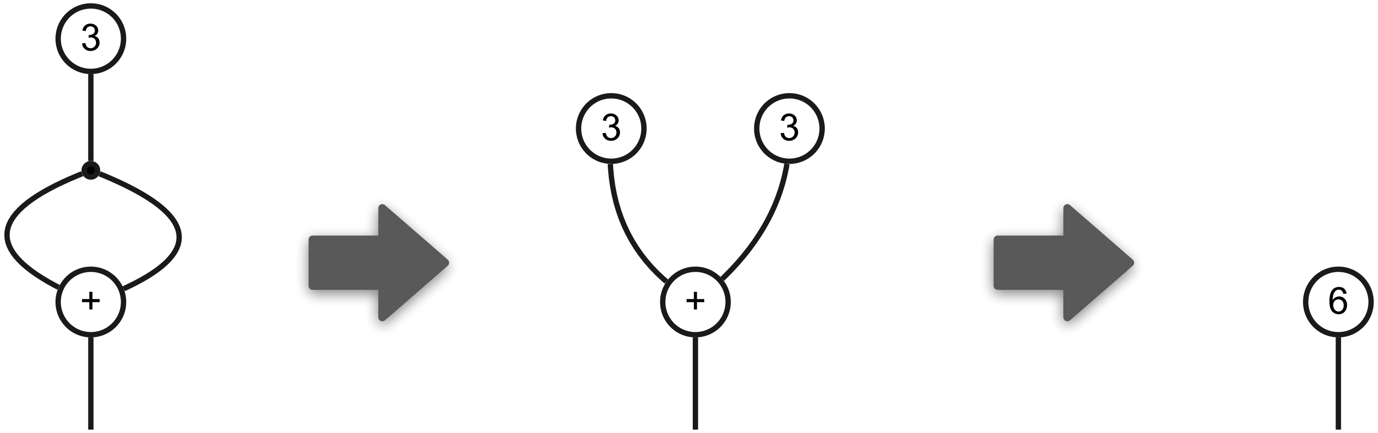
# Program execution, graphically

Idea: updating hypernets step-by-step



# Program execution, graphically

Idea: updating hypernets step-by-step

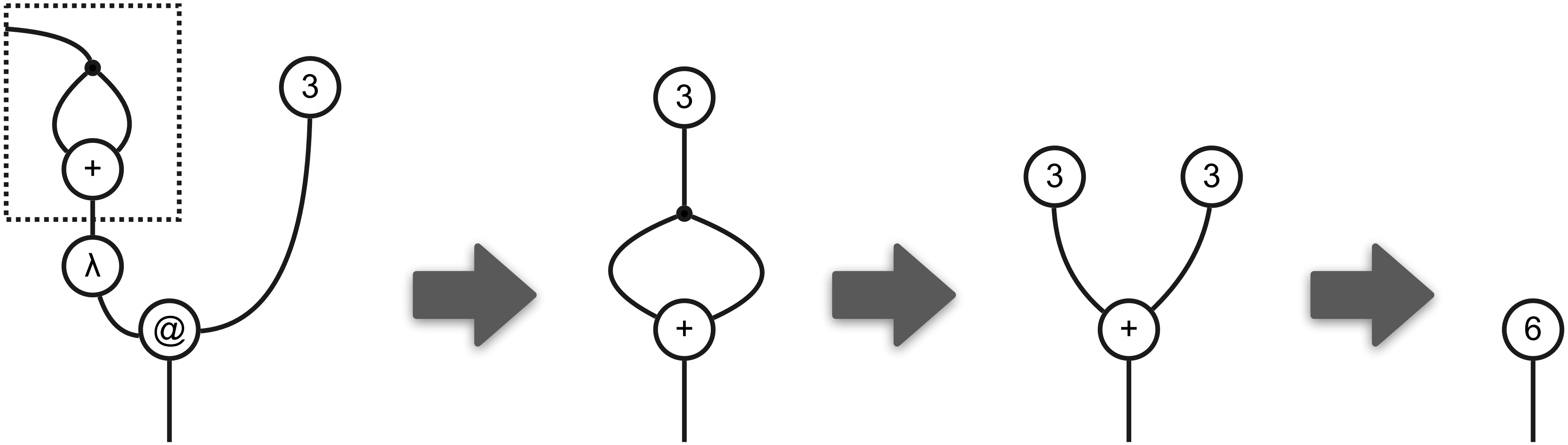


```
let x = 3 in
x + x
```

```
3 + 3
```

# Program execution, graphically

Idea: updating hypernets step-by-step



```
(λx. x + x) 3
```

```
let x = 3 in
x + x
```

```
3 + 3
```

# Program execution, graphically

Idea: updating hypernets step-by-step

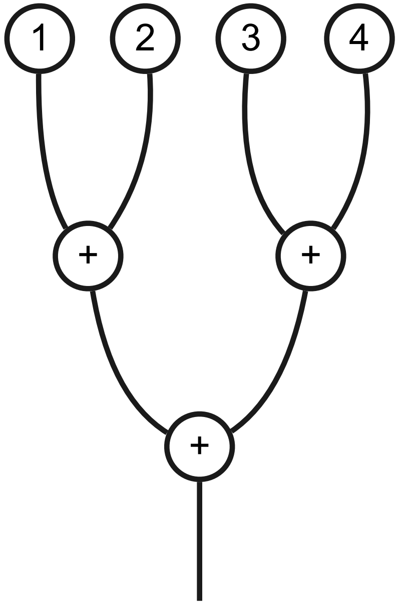
… and strategically, using *focus* with three modes:

- ? depth-first redex search
- ✓ backtracking
- ↯ triggering update of hypernet

# Program execution, graphically

Idea: updating hypernets step-by-step

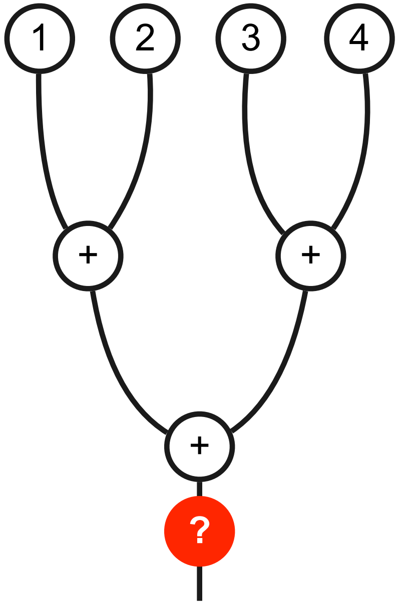
… and strategically, using *focus*



# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

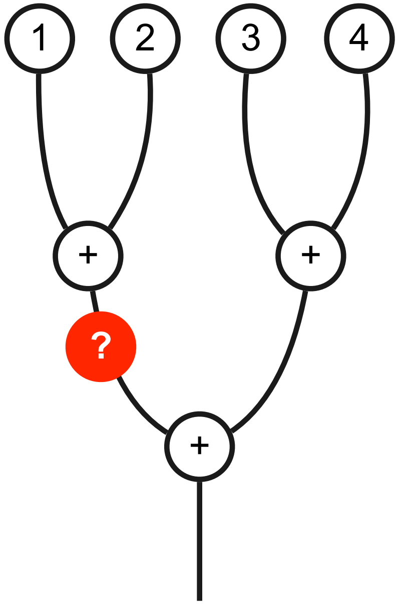


depth-first redex search

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

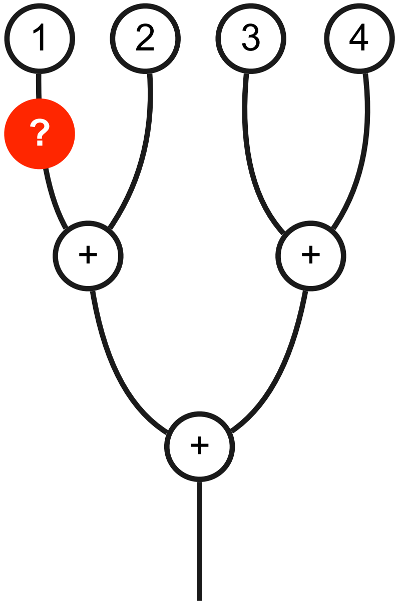


depth-first redex search

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

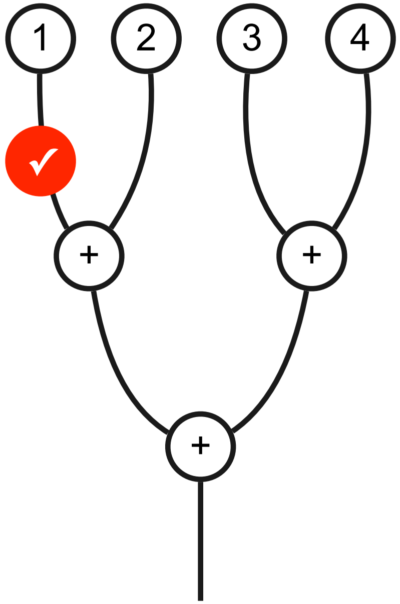


depth-first redex search

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

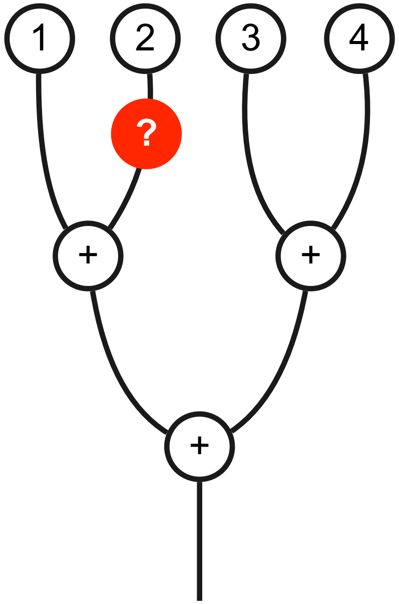


backtracking

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

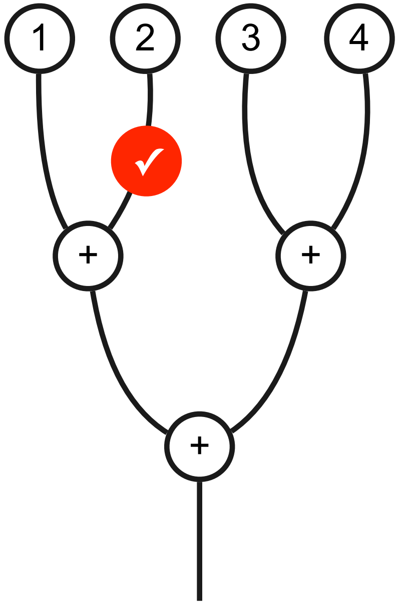


depth-first redex search

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

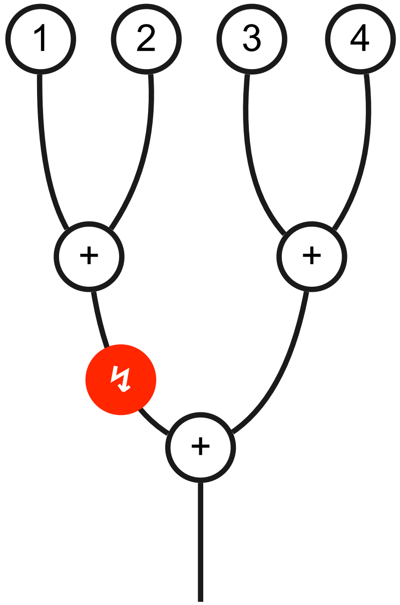


backtracking

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

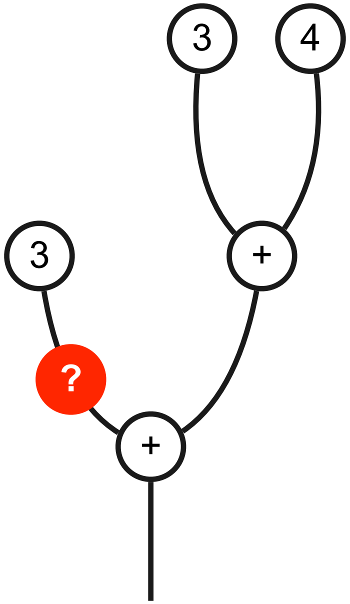


triggering update of hypernet

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

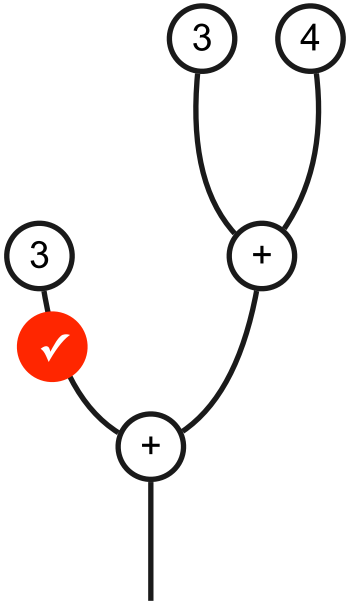


depth-first redex search

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

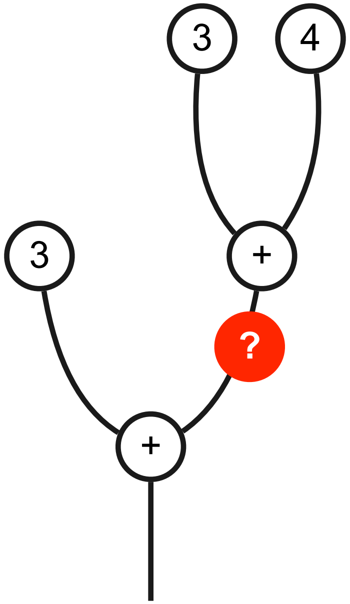


backtracking

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

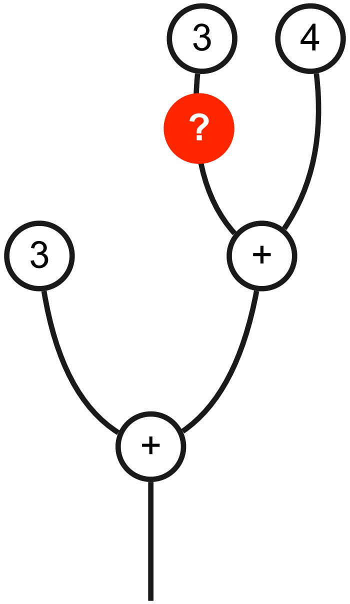


depth-first redex search

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

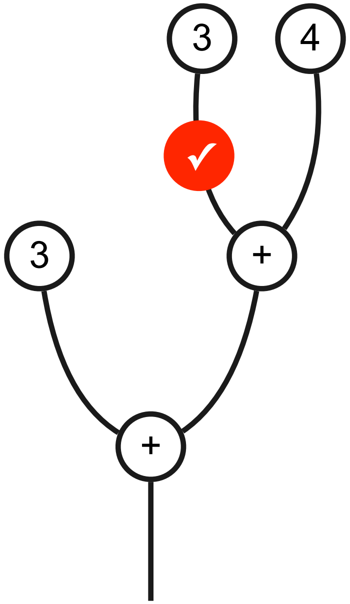


depth-first redex search

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

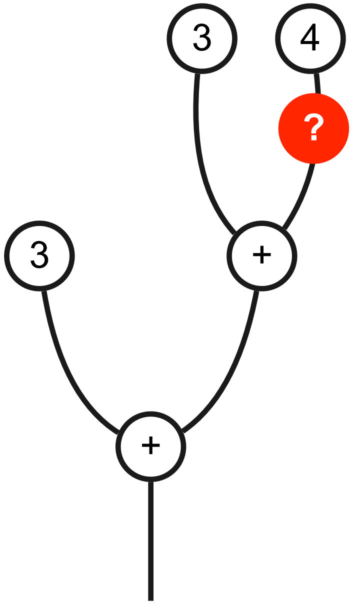


backtracking

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

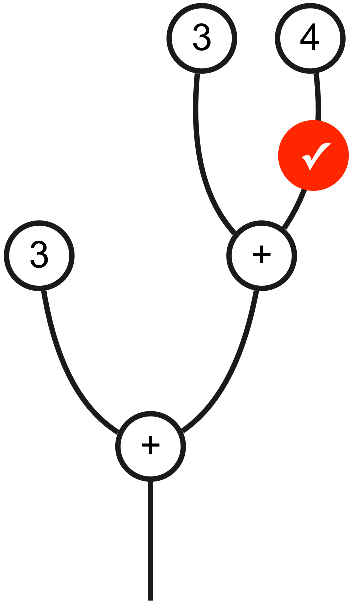


depth-first redex search

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

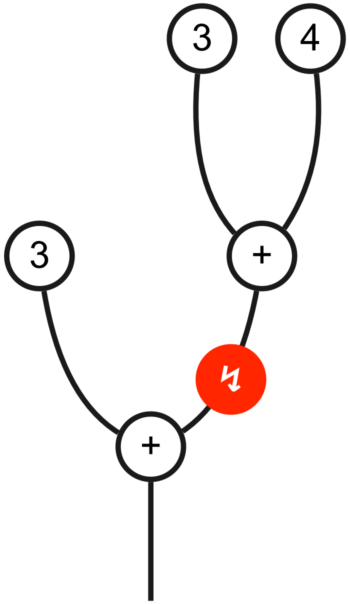


backtracking

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*



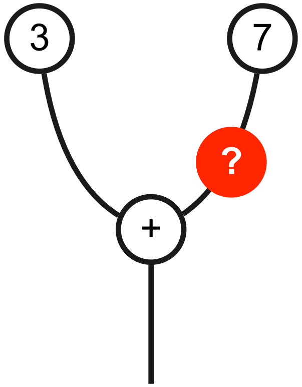
triggering update of hypernet

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

depth-first redex search

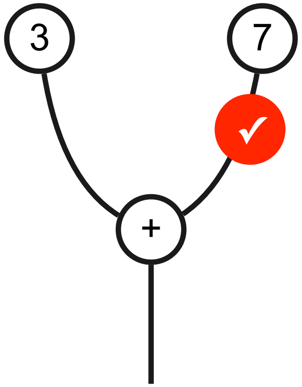


# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

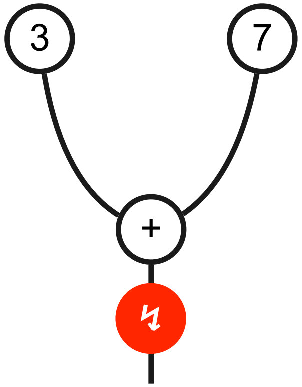
backtracking



# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*



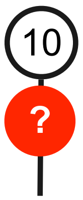
triggering update of hypernet

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

depth-first redex search

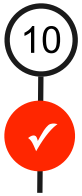


# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

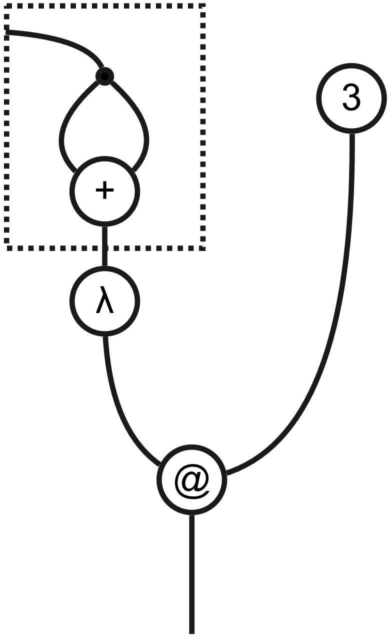
backtracking



# Program execution, graphically

Idea: updating hypernets step-by-step

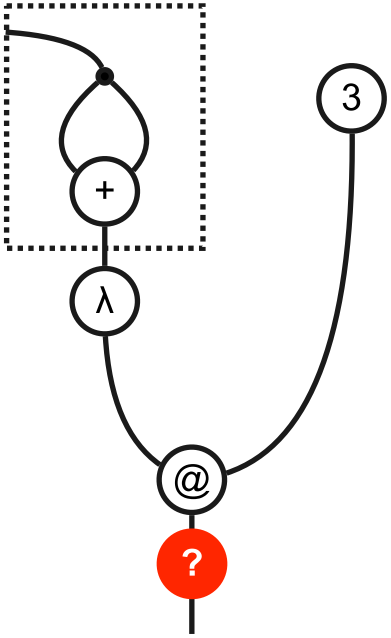
… and strategically, using *focus*



# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

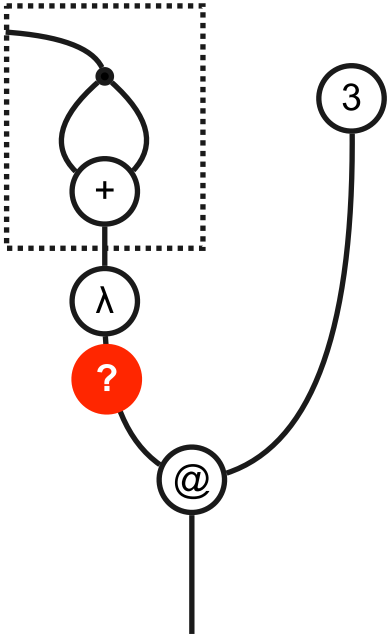


depth-first redex search

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

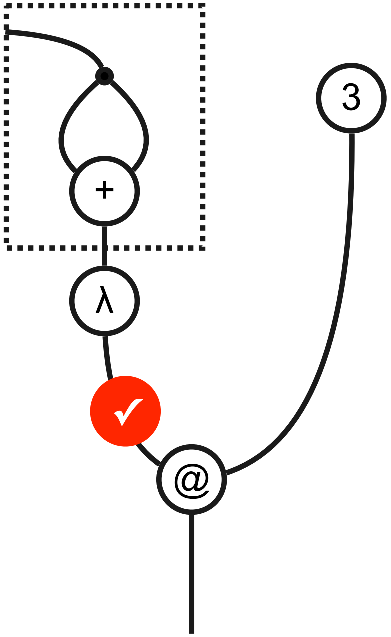


depth-first redex search

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

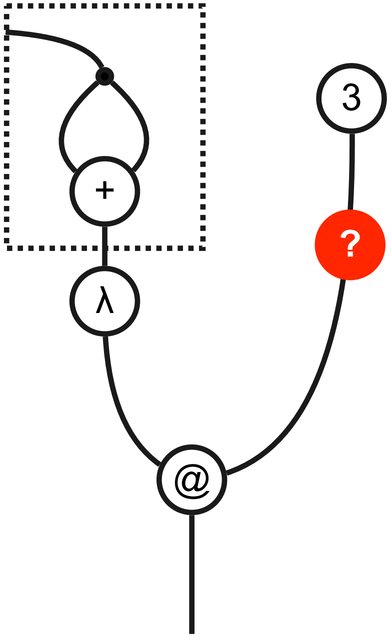


backtracking

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

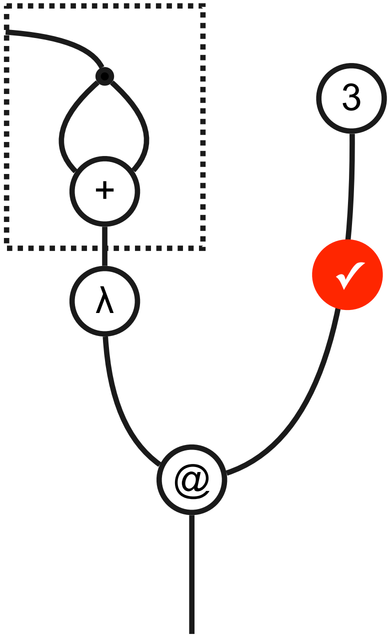


depth-first redex search

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

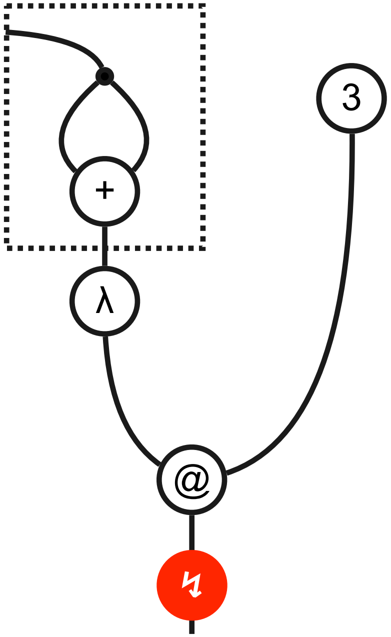


backtracking

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

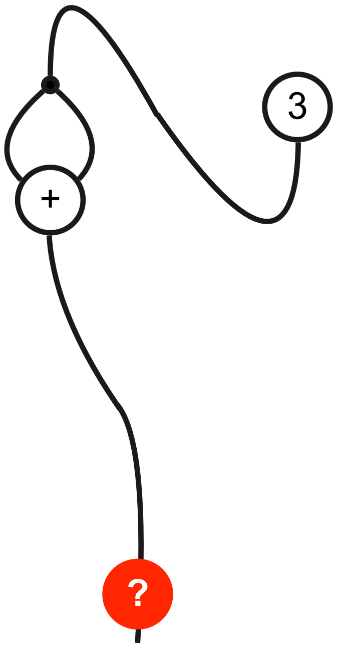


triggering update of hypernet

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

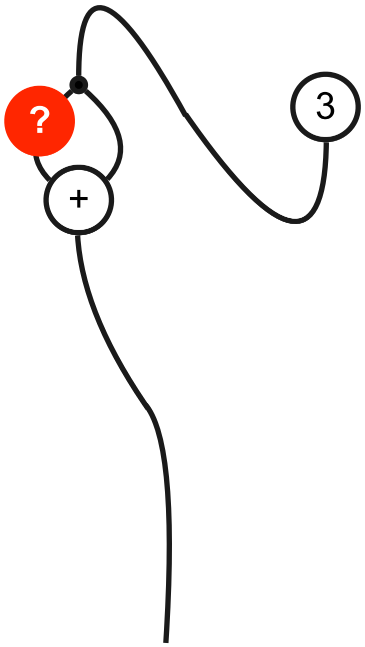


depth-first redex search

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

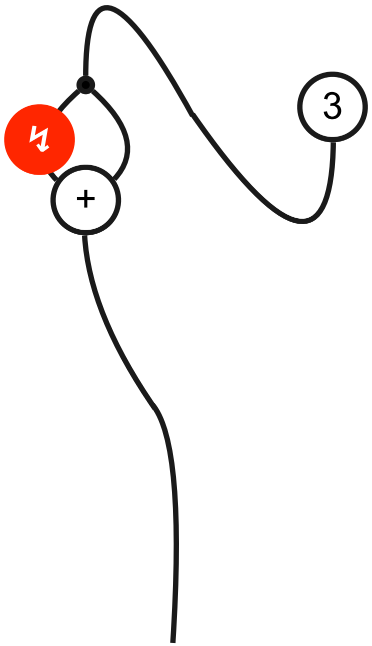


depth-first redex search

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

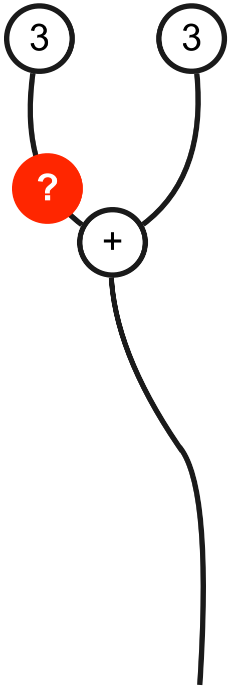


triggering update of hypernet

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

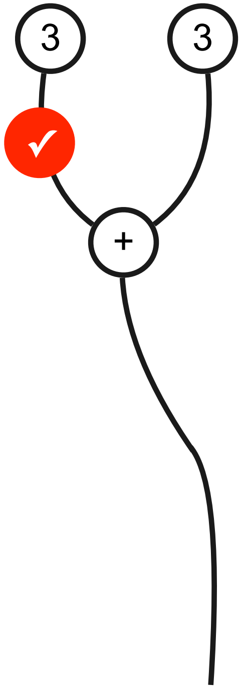


depth-first redex search

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

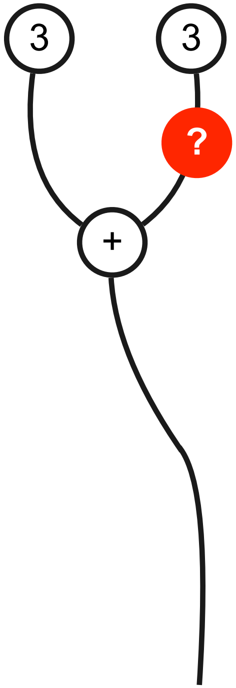


backtracking

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

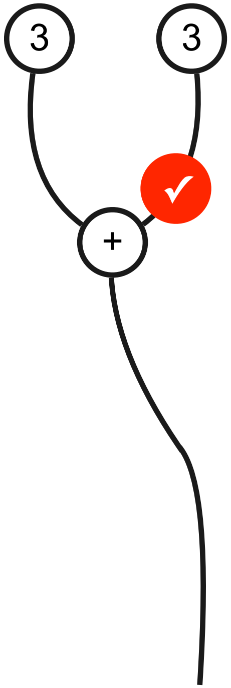


depth-first redex search

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

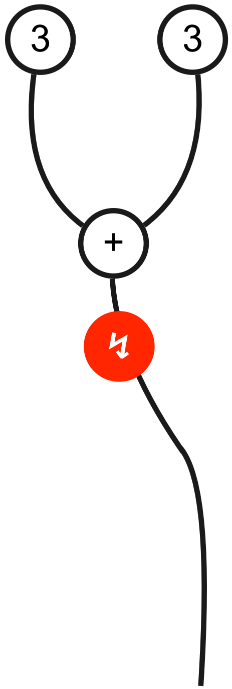


backtracking

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*



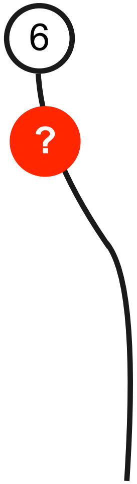
triggering update of hypernet

# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

depth-first redex search

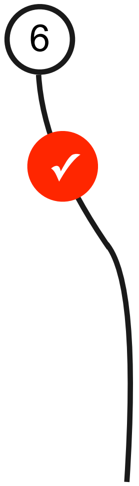


# Program execution, graphically

Idea: updating hypernets step-by-step

… and strategically, using *focus*

backtracking



# Hypernet semantics

- program execution by a *graphical* abstract machine
  - programs as

    certain hierarchical hypergraphs (*“hypernets”*)
  - execution as

    step-by-step strategical update of hypernets

# Hypernet semantics

- program execution by a *graphical* abstract machine
  - programs as

    certain hierarchical hypergraphs (*“hypernets”*)
  - execution as

    step-by-step strategical update of hypernets
  - state = hypernet with focus ? ✓ ↯
  - transition = move of focus, or update of hypernet

# Overview

1. Motivation: robustness of observational equivalence

2. Hypernet semantics

3. Locality & step-wise reasoning

4. Example: cbv linear β-law

# Proof of observational equivalence, using *locality*

*“Do two program fragments behave the same?”*

# Proof of observational equivalence, using *locality*

~~*“Do two program fragments behave the same?”*~~

*“Do two sub-graphs behave the same in hypernet semantics?”*

# Proof of observational equivalence, using *locality*

~~*“Do two program fragments behave the same?”*~~

*“Do two sub-graphs behave the same in hypernet semantics?”*

★ Sub-graphs can represent parts of a program that are not necessarily well-formed,

e.g. parts relevant to a certain reference:

```
… new a = 1 in … (λx. a := 2; !a) … (λx. a := 2; !a) …
```

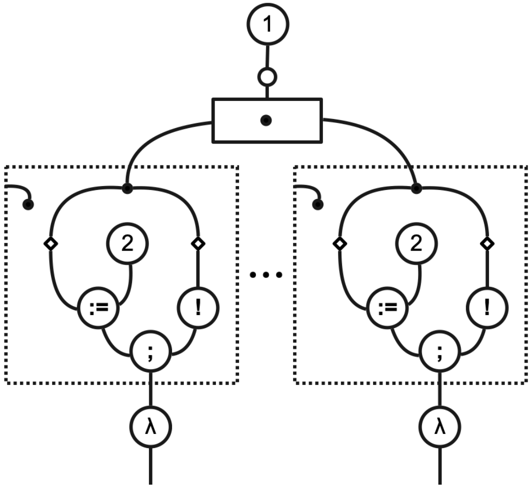
# Proof of observational equivalence, using *locality*

~~*“Do two program fragments behave the same?”*~~

*“Do two sub-graphs behave the same in hypernet semantics?”*

★ Sub-graphs can represent parts of a program that are not necessarily well-formed,
e.g. parts relevant to a certain reference:

```
… new a = 1 in … (λx. a := 2; !a) … (λx. a := 2; !a) …
```



# Proof of observational equivalence, using *locality*

~~*“Do two program fragments behave the same?”*~~

*“Do two sub-graphs behave the same in hypernet semantics?”*

★ Sub-graphs can represent parts of a program that are not necessarily well-formed,

e.g. parts relevant to a certain reference:

```
… new a = 1 in … (λx. a := 2; !a) … (λx. a := 2; !a) …
```

Idea of *locality*:

analysing behaviour of program fragments,

by tracing sub-graphs during execution

# Proof of observational equivalence, using *locality*

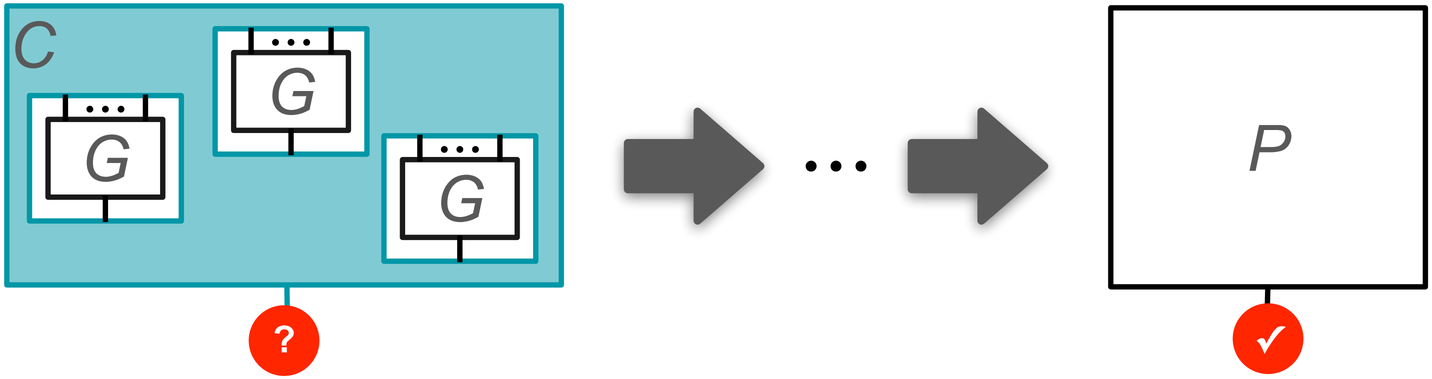
*Claim: “Behaviour of a sub-graph G can be matched by behaviour of a sub-graph H.”*

# Proof of observational equivalence, using *locality*

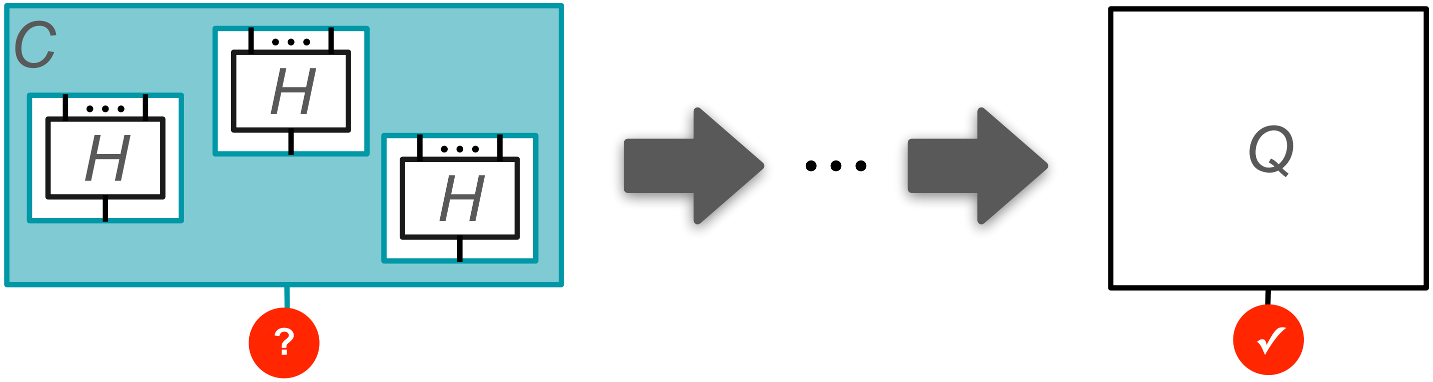
*Claim: "Behaviour of a sub-graph G can be matched by behaviour of a sub-graph H."*

For any context *C*,

if



then



Muroya (RIMS, Kyoto U.)

# Proof of observational equivalence, using *locality*

*Claim: "Behaviour of a sub-graph G can be matched by behaviour of a sub-graph H."*

Proof idea (simplified):

1. take **contextual closure** *R* of *(G,H)*

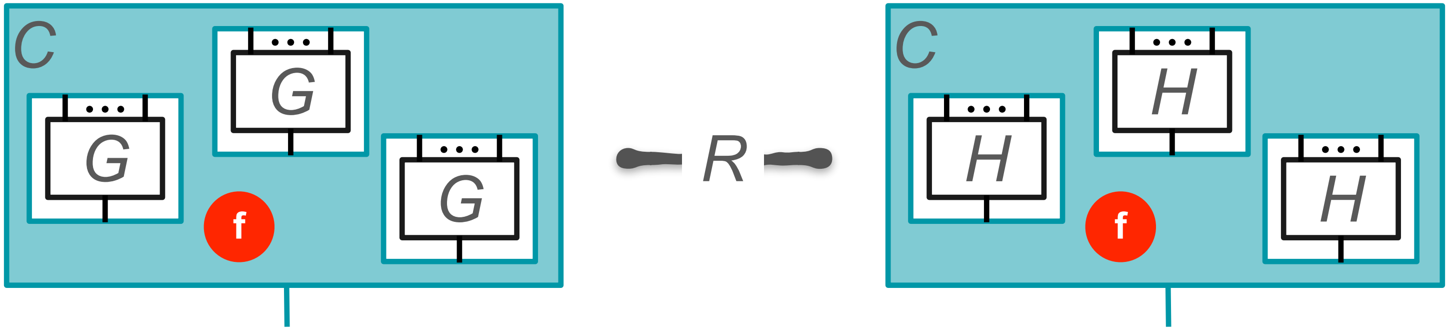
2. prove that the contextual closure *R* is a ***-simulation**

# Proof of observational equivalence, using *locality*

*Claim: "Behaviour of a sub-graph G can be matched by behaviour of a sub-graph H."*

Proof idea (simplified):

1. take **contextual closure** *R* of *(G,H)*



for any context *C* with focus

2. prove that the contextual closure *R* is a ***-simulation**

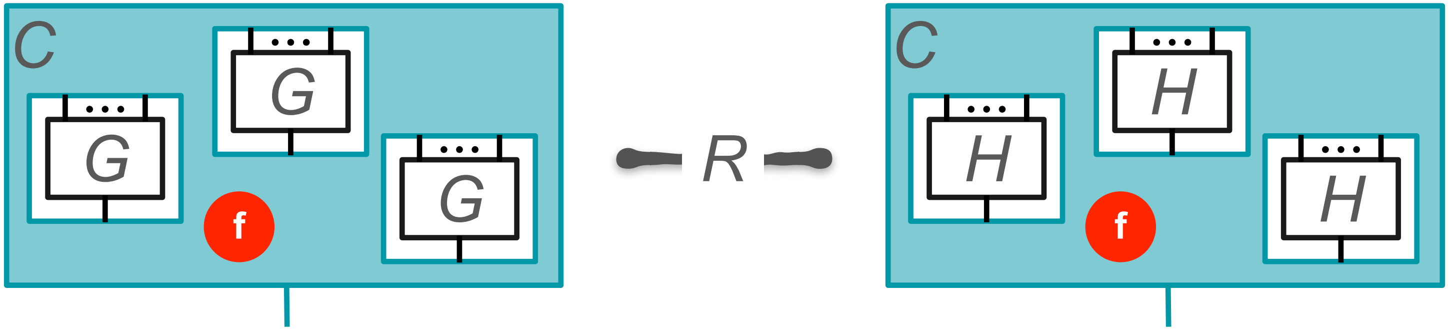
# Proof of observational equivalence, using *locality*

*Claim: "Behaviour of a sub-graph G can be matched by behaviour of a sub-graph H."*

R is closed under contexts, by definition

Proof idea (simplified):

1. take **contextual closure** *R* of *(G,H)*



for any context *C* with focus

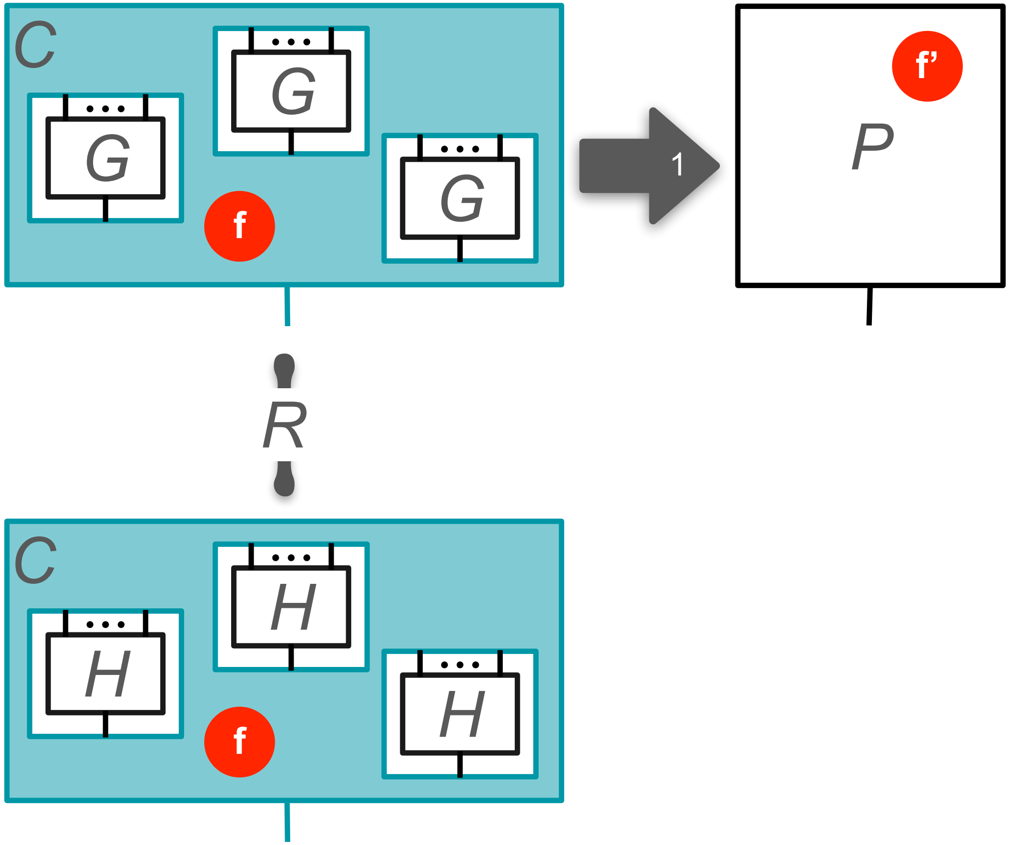
2. prove that the contextual closure *R* is a ***-simulation**

Muroya (RIMS, Kyoto U.)

# Proof of observational equivalence, using *locality*

Proof idea (simplified):

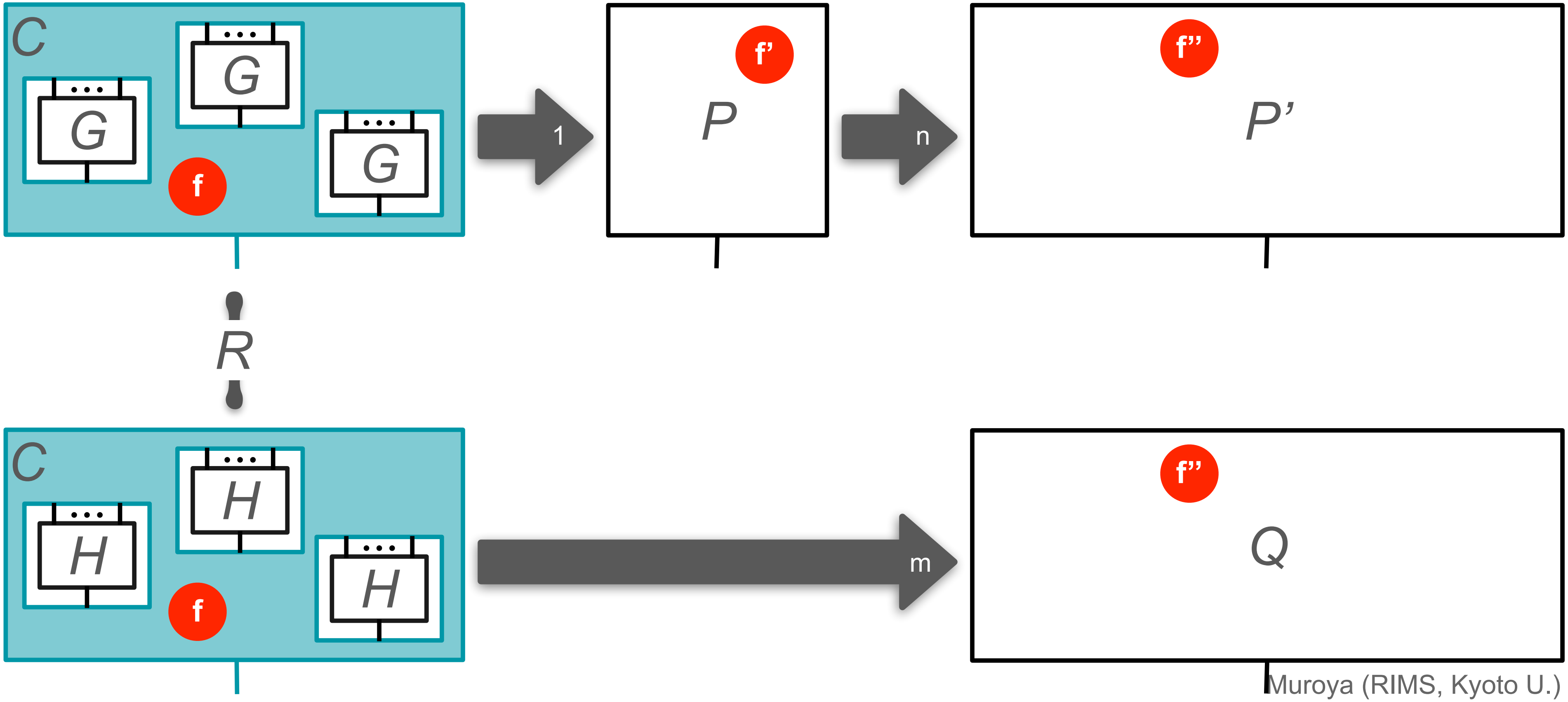
2. prove that the contextual closure *R* is a ***-simulation**



# Proof of observational equivalence, using *locality*

Proof idea (simplified):

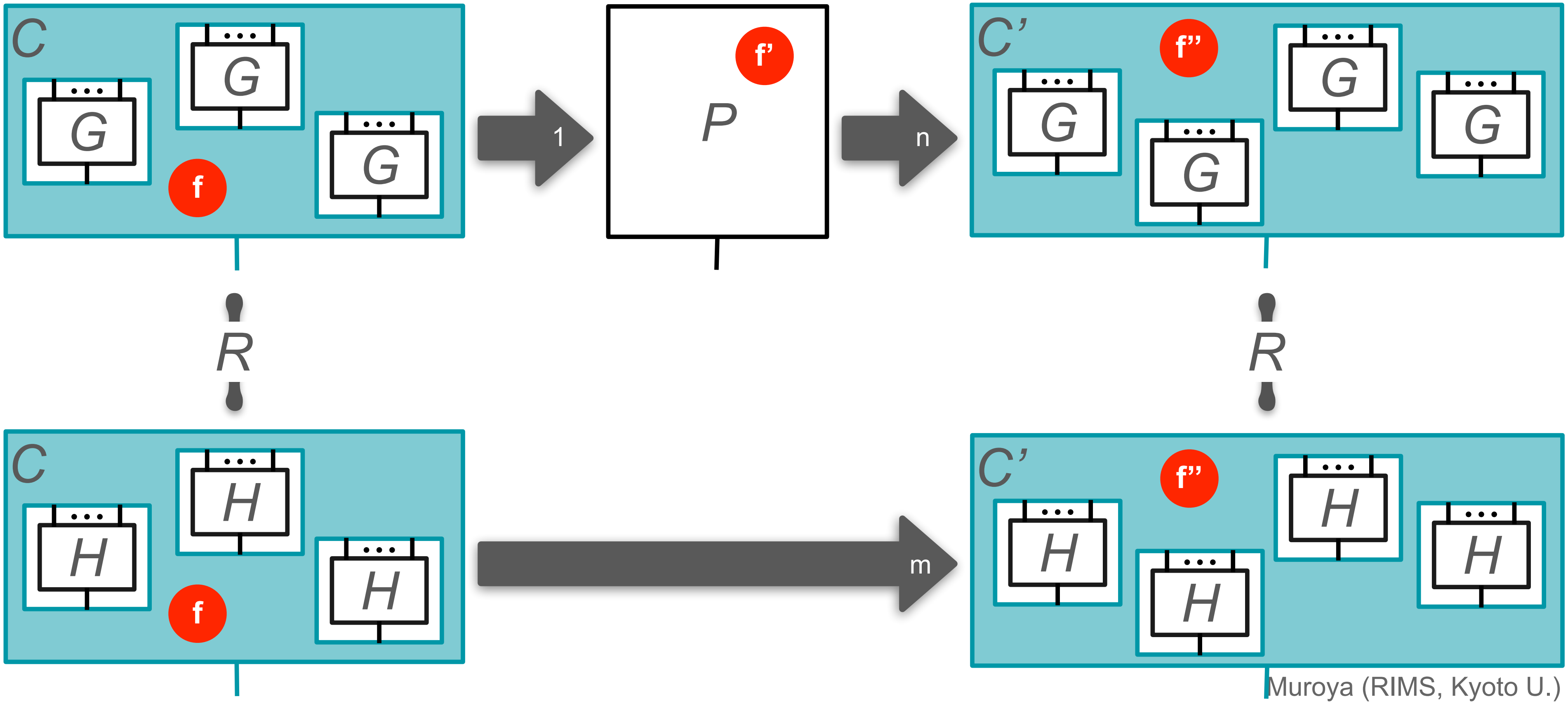
2. prove that the contextual closure *R* is a ***-simulation**



# Proof of observational equivalence, using *locality*

Proof idea (simplified):

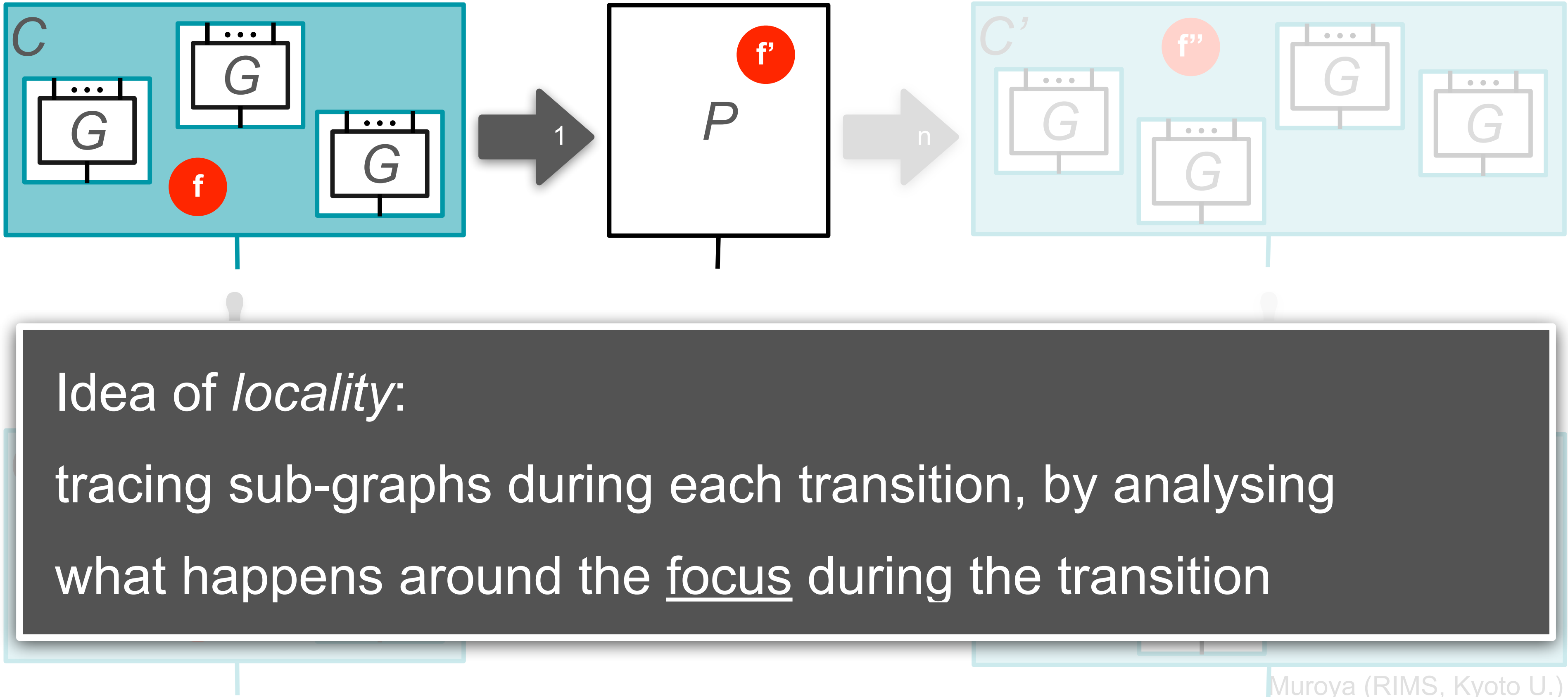
2. prove that the contextual closure *R* is a ***-simulation**



# Proof of observational equivalence, using *locality*

Proof idea (simplified):

2. prove that the contextual closure *R* is a ***-simulation**



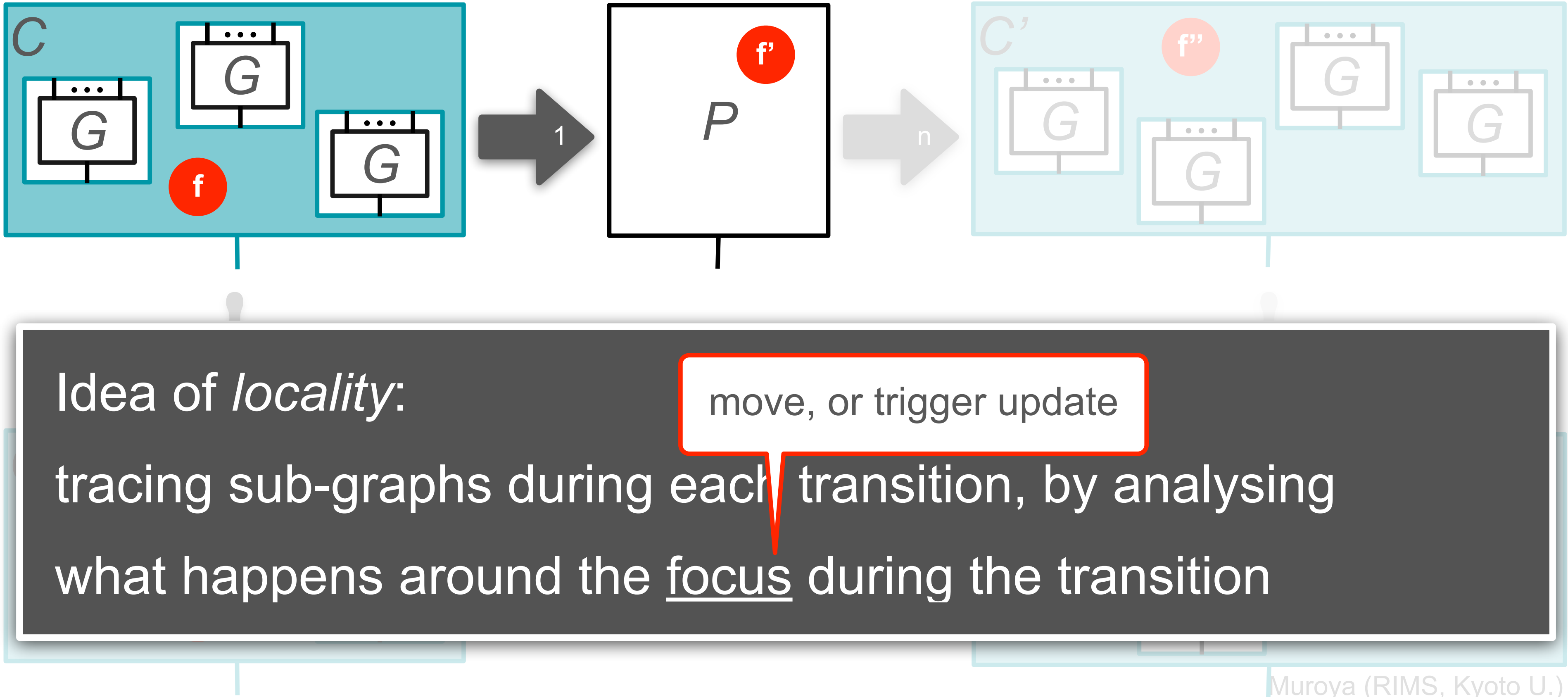
Idea of *locality*:

tracing sub-graphs during each transition, by analysing what happens around the focus during the transition

# Proof of observational equivalence, using *locality*

Proof idea (simplified):

2. prove that the contextual closure *R* is a ***-simulation**



Idea of *locality*:

move, or trigger update

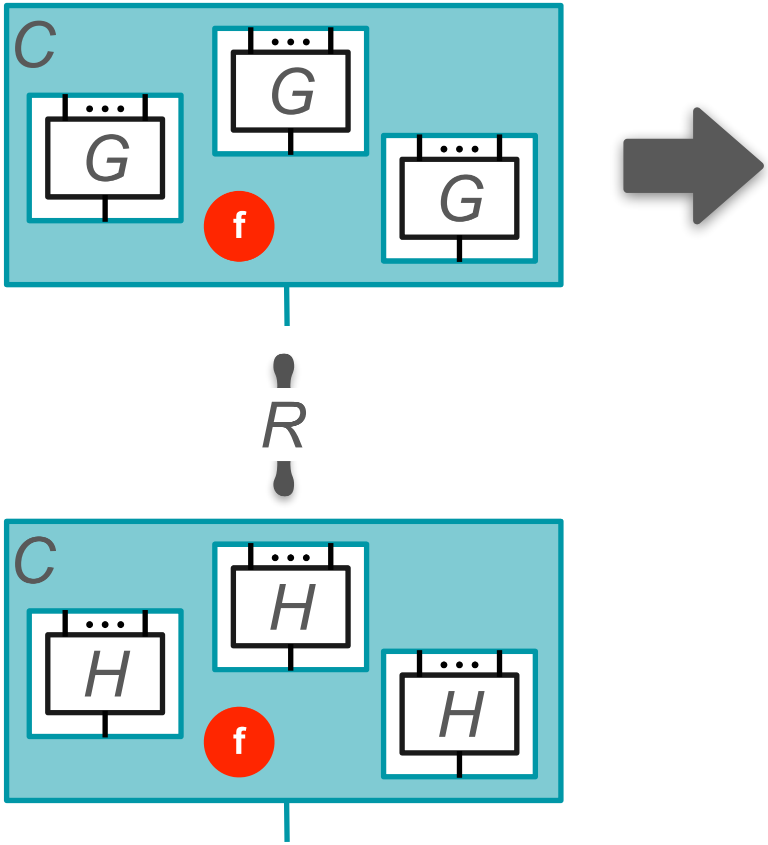
tracing sub-graphs during each transition, by analysing what happens around the focus during the transition

# Proof of observational equivalence, using *locality*

Proof idea (simplified):

2. prove that the contextual closure *R* is a ***-simulation**

Case (1) move of focus ? or ✓ inside context



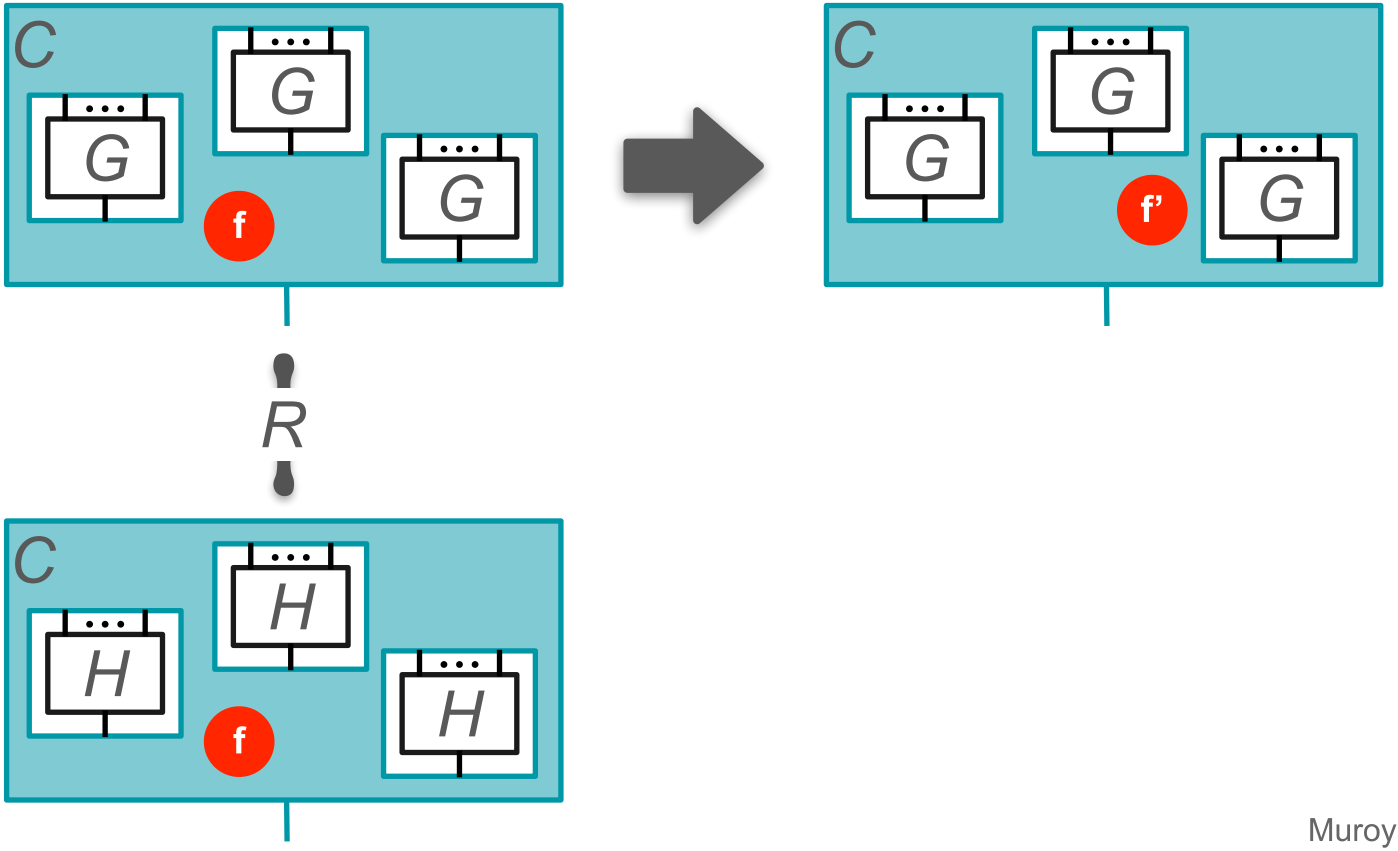
Muroya (RIMS, Kyoto U.)

# Proof of observational equivalence, using *locality*

Proof idea (simplified):

2. prove that the contextual closure *R* is a ***-simulation**

Case (1) move of focus ? or ✓ inside context

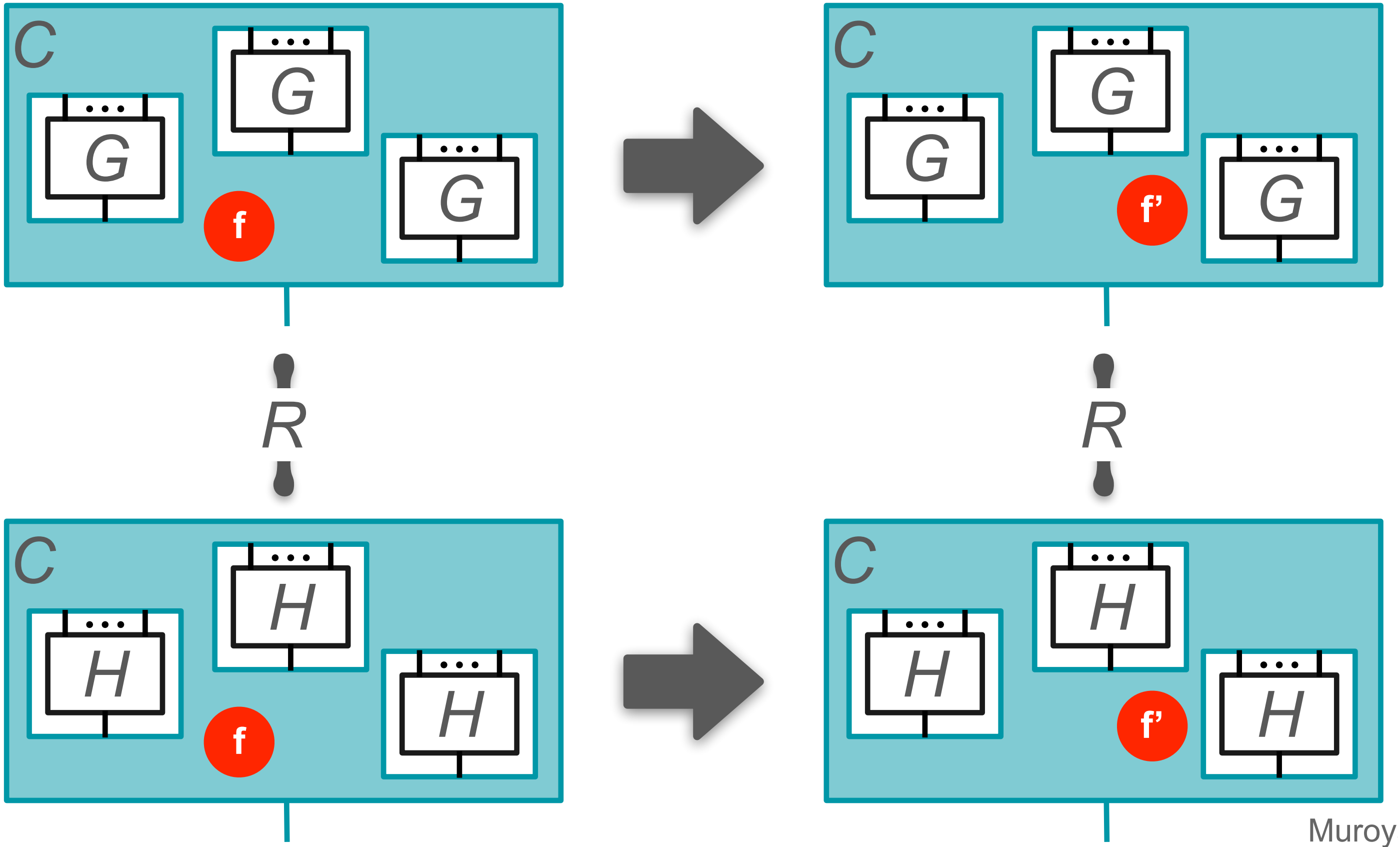


# Proof of observational equivalence, using *locality*

Proof idea (simplified):

2. prove that the contextual closure *R* is a ***-simulation**

Case (1) move of focus ? or ✓ inside context

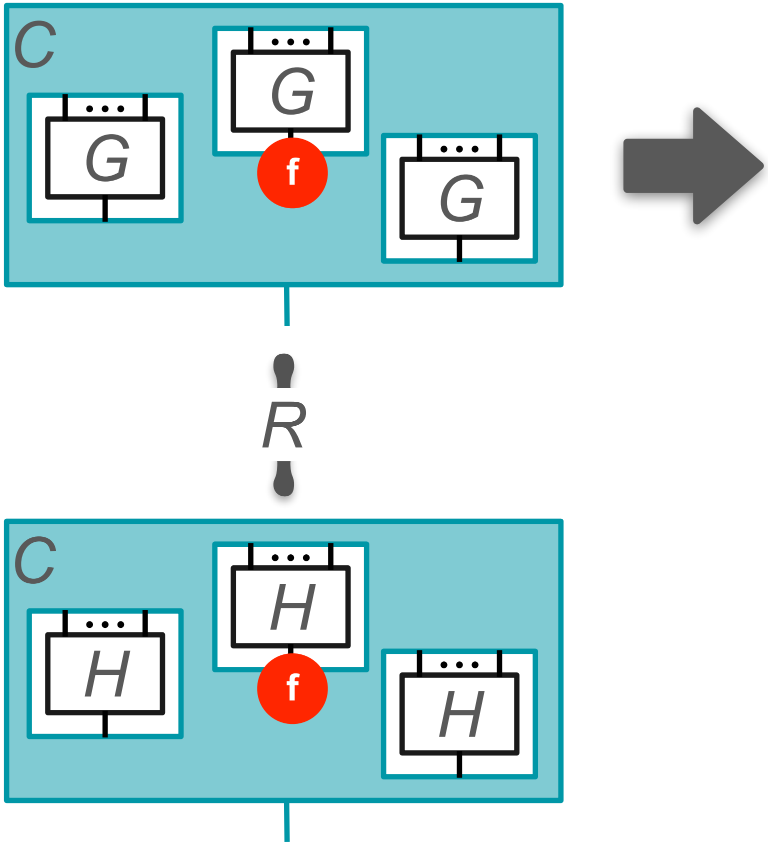


# Proof of observational equivalence, using *locality*

Proof idea (simplified):

2. prove that the contextual closure *R* is a ***-simulation**

Case (2) move of focus ? or ✓ , entering *G*

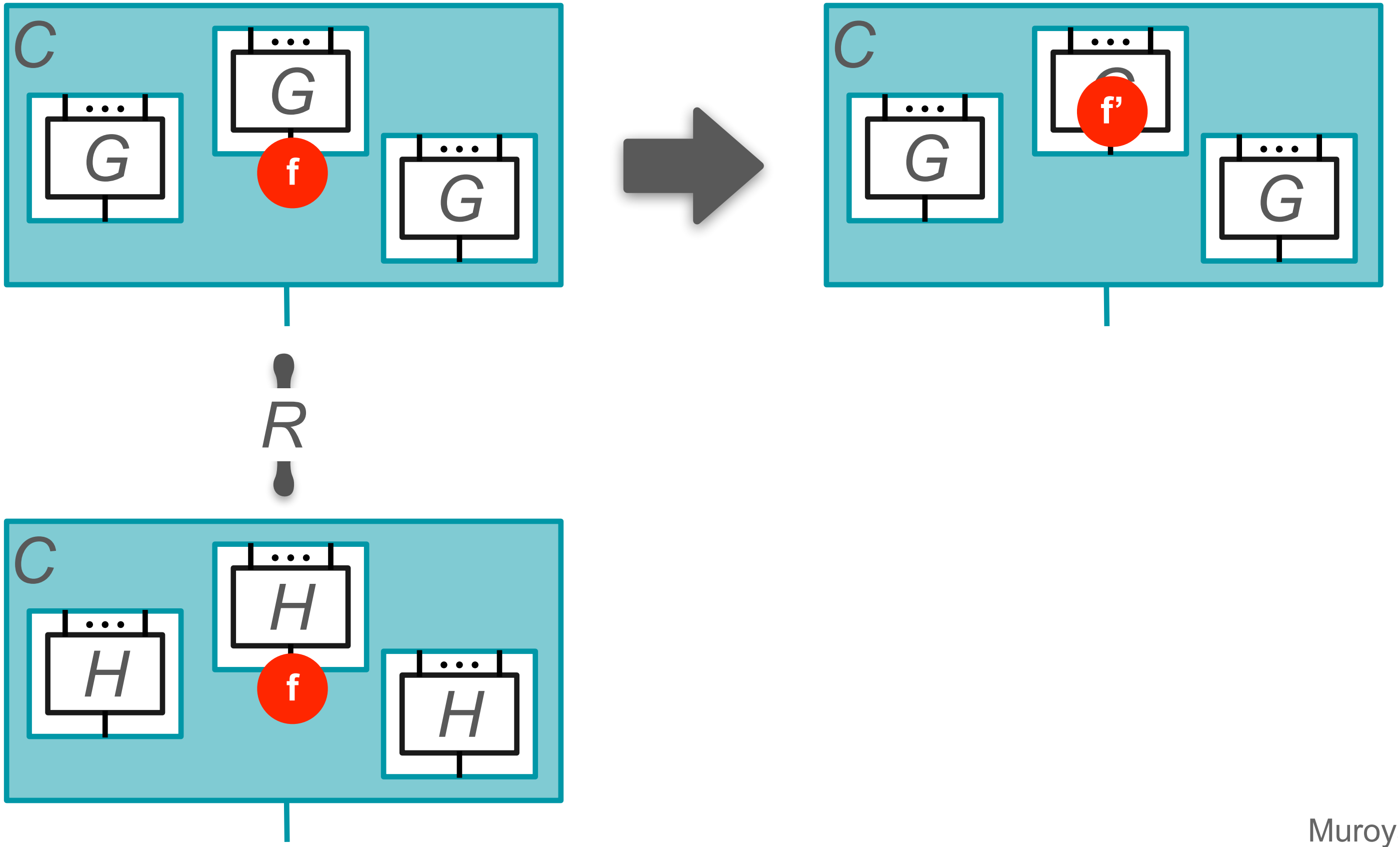


# Proof of observational equivalence, using *locality*

Proof idea (simplified):

2. prove that the contextual closure *R* is a ***-simulation**

Case (2) move of focus ? or ✓ , entering *G*

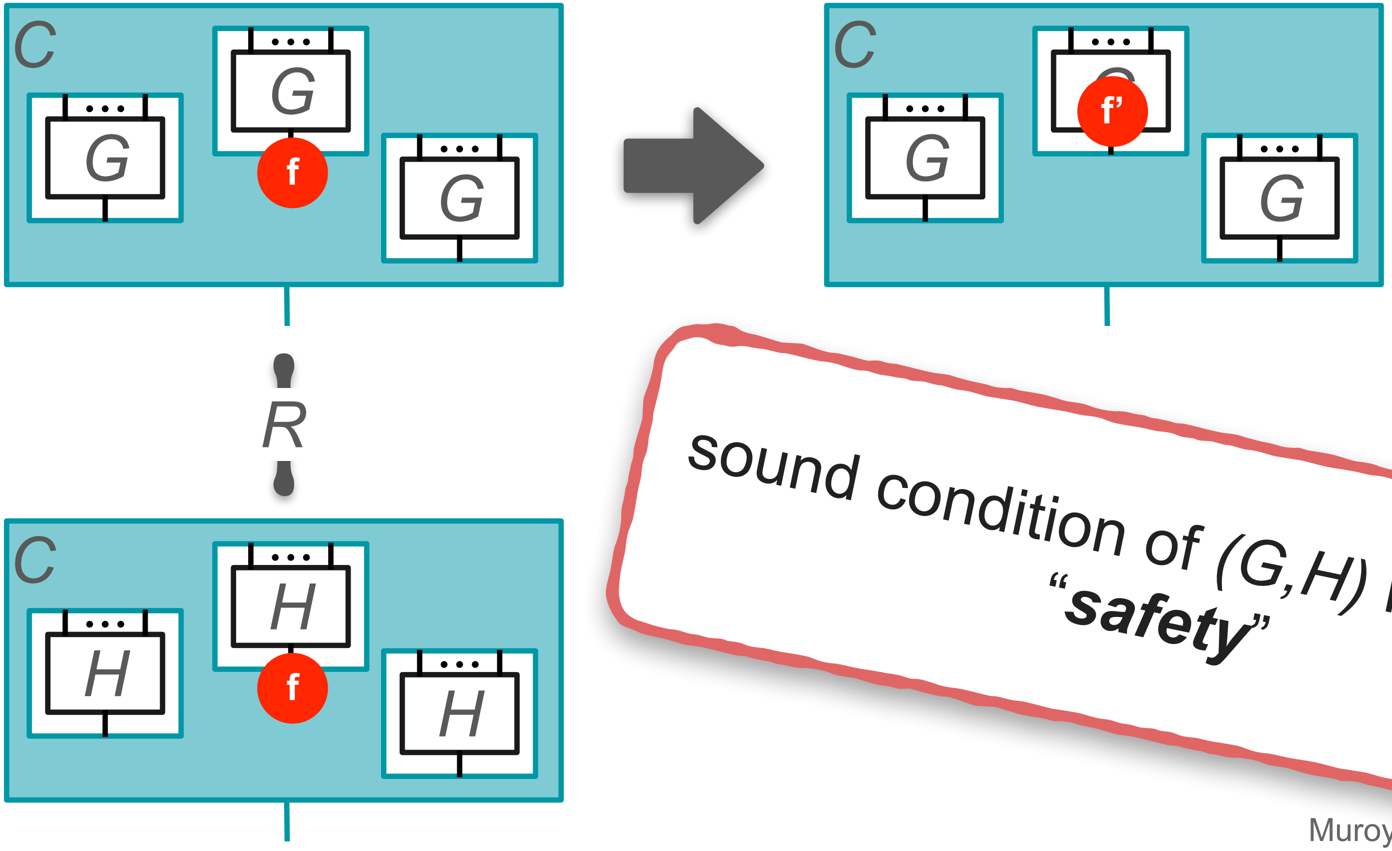


# Proof of observational equivalence, using *locality*

Proof idea (simplified):

2. prove that the contextual closure *R* is a ***-simulation**

Case (2) move of focus ? or ✓ , entering *G*



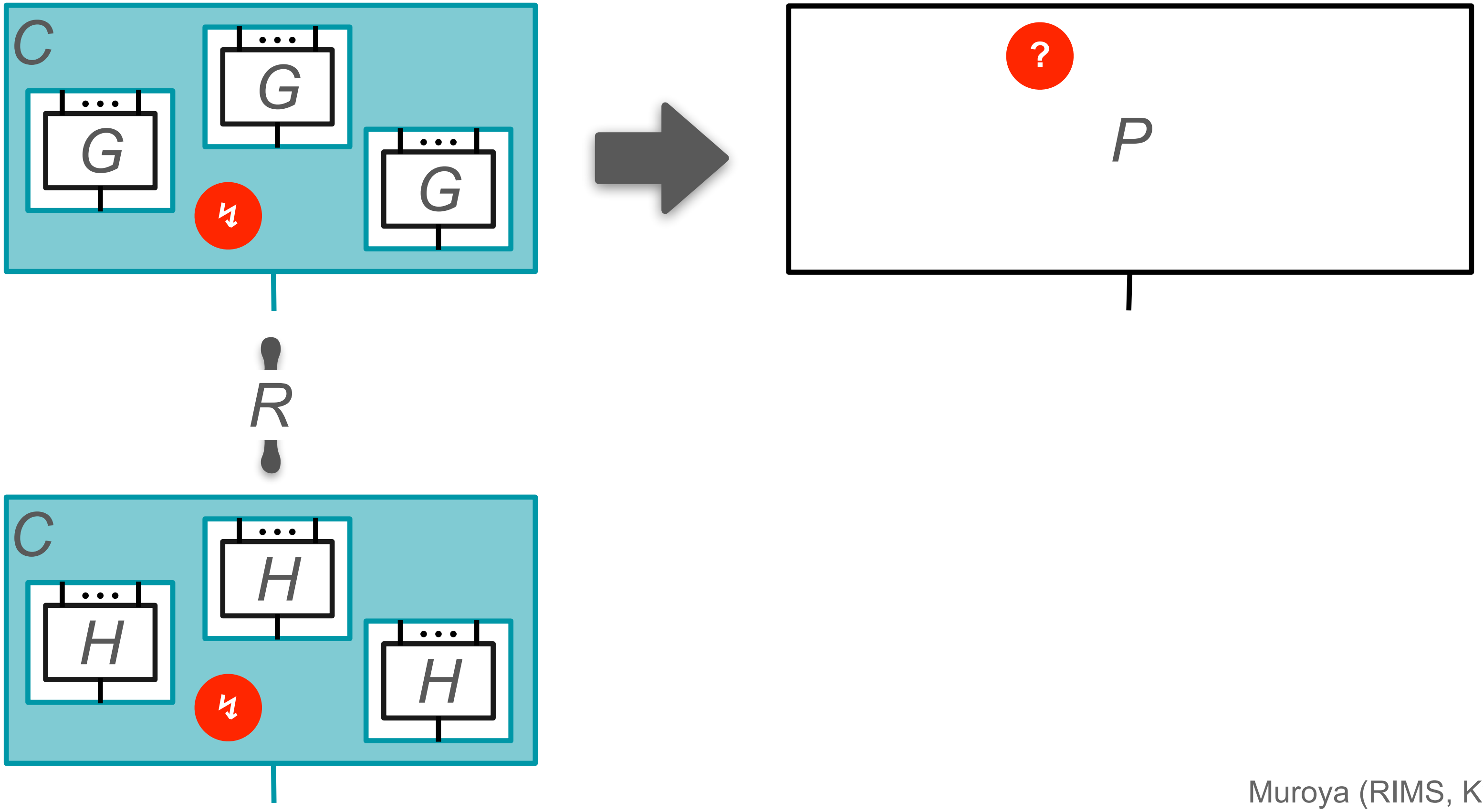
sound condition of (*G*,*H*) identified: “***safety***”

# Proof of observational equivalence, using *locality*

Proof idea (simplified):

2. prove that the contextual closure *R* is a ***-simulation**

Case (3) update of hypernet

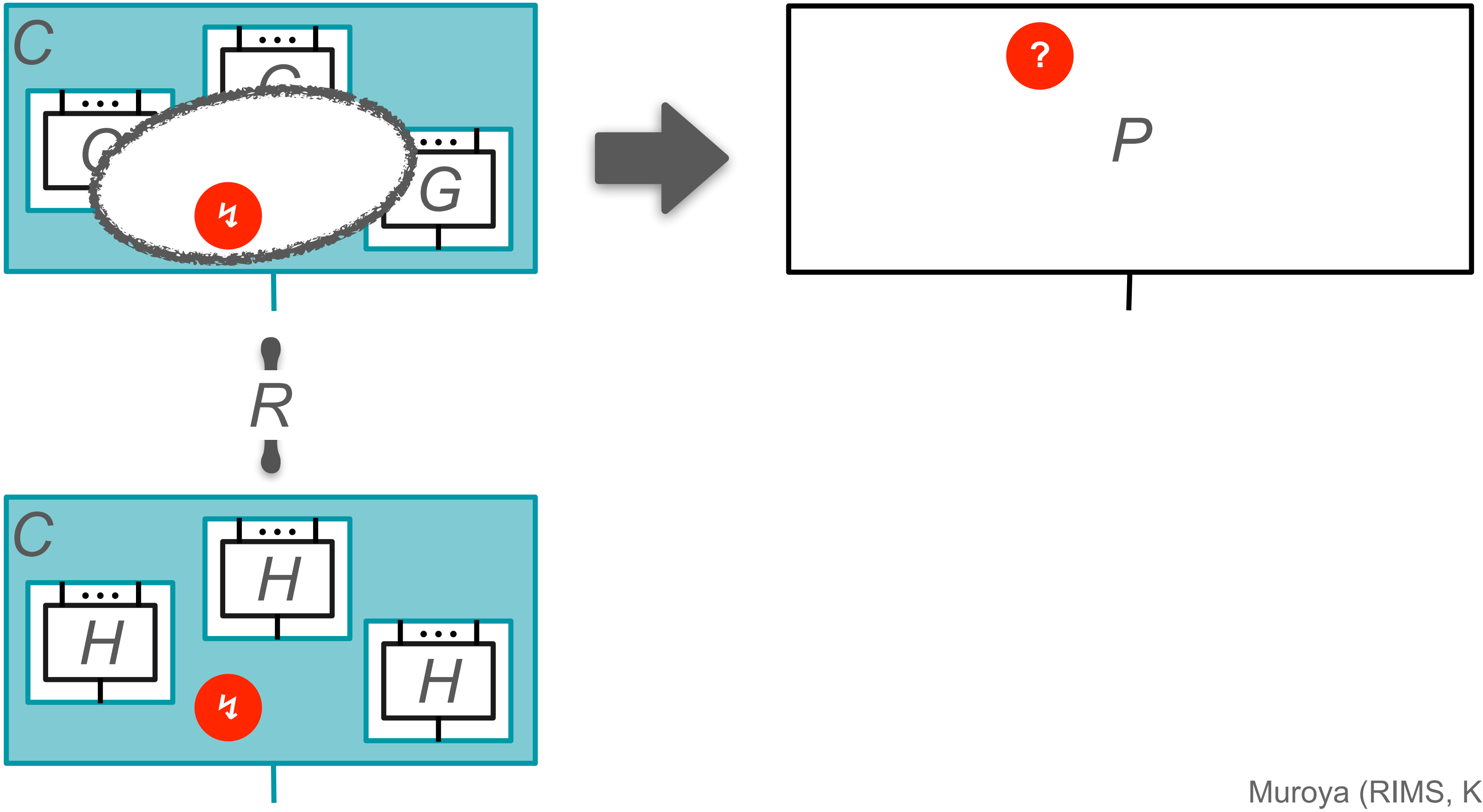


# Proof of observational equivalence, using *locality*

Proof idea (simplified):

2. prove that the contextual closure *R* is a ***-simulation**

Case (3) update of hypernet

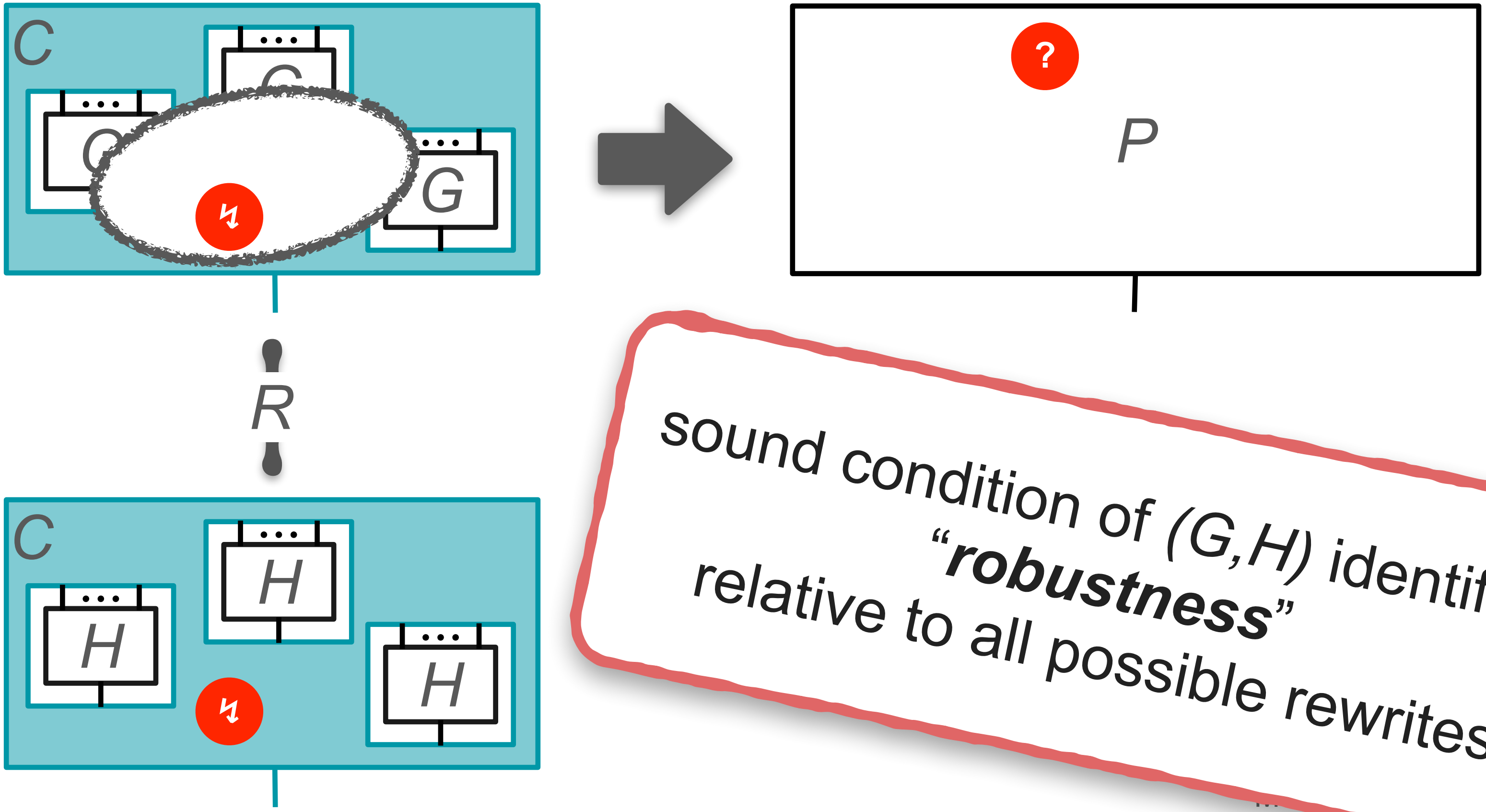


# Proof of observational equivalence, using *locality*

Proof idea (simplified):

2. prove that the contextual closure *R* is a ***-simulation**

Case (3) update of hypernet



sound condition of (G,H) identified:
“**robustness**”
relative to all possible rewrites

# Proof of observational equivalence, using *locality*

*Claim: "Behaviour of a sub-graph G can be matched by behaviour of a sub-graph H."*

Proof idea (simplified):

1. take **contextual closure** *R* of *(G,H)*

2. prove that the contextual closure *R* is a ***-simulation**

   by case analysis

# Proof of observational equivalence, using *locality*

*Claim: “Behaviour of a sub-graph G can be matched by behaviour of a sub-graph H.”*

Proof idea (simplified):

1. take **contextual closure** *R* of *(G,H)*
2. prove that the contextual closure *R* is a ***-simulation** by case analysis

Partial Characterisation Theorem

***Robust*** *and* ***safe*** *templates induce observational equivalences.*

(for deterministic & “reasonable” languages)

# Overview

1. Motivation: robustness of observational equivalence

2. Hypernet semantics

3. Locality & step-wise reasoning

4. Example: cbv linear β-law

# Example: cbv linear β-law

Proof methodology:

1. prepare a template *{(G,H)}*

2. prove that the template *{(G,H)}* is **robust** and **safe**

3. apply the Partial Characterisation Theorem

Partial Characterisation Theorem

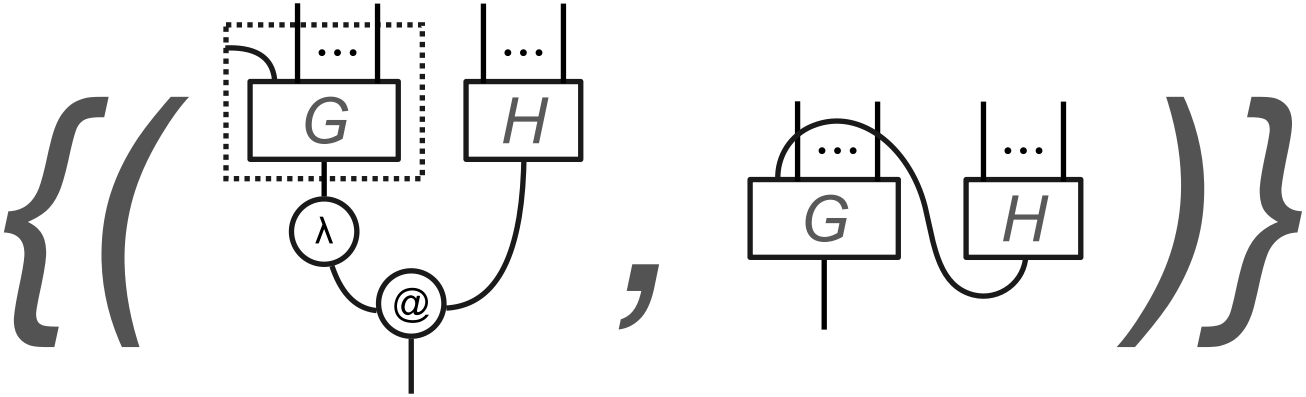
***Robust*** *and* ***safe*** *templates induce observational equivalences.*

(for deterministic & “reasonable” languages)

# Example: cbv linear β-law

Proof methodology:

1. prepare *the cbv linear β-template*:



where *H* represents a *value*

2. prove that the cbv linear β-template is **robust** and **safe**

Partial Characterisation Theorem

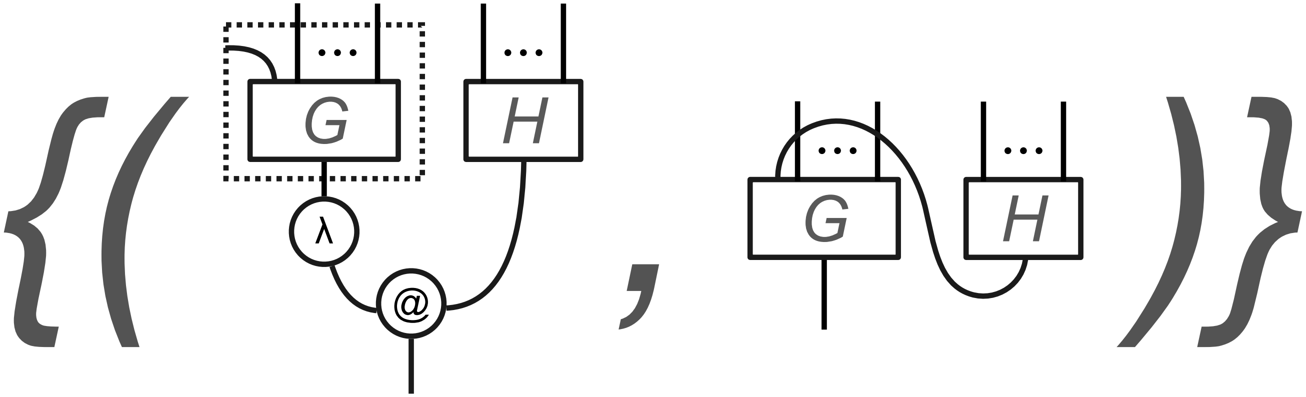
***Robust** and **safe** templates induce observational equivalences.*

(for deterministic & “reasonable” languages)

# Example: cbv linear β-law

Proof methodology:

1. prepare *the cbv linear β-template*:



where *H* represents a *value*

2. prove that the cbv linear β-template is **robust** and **safe**

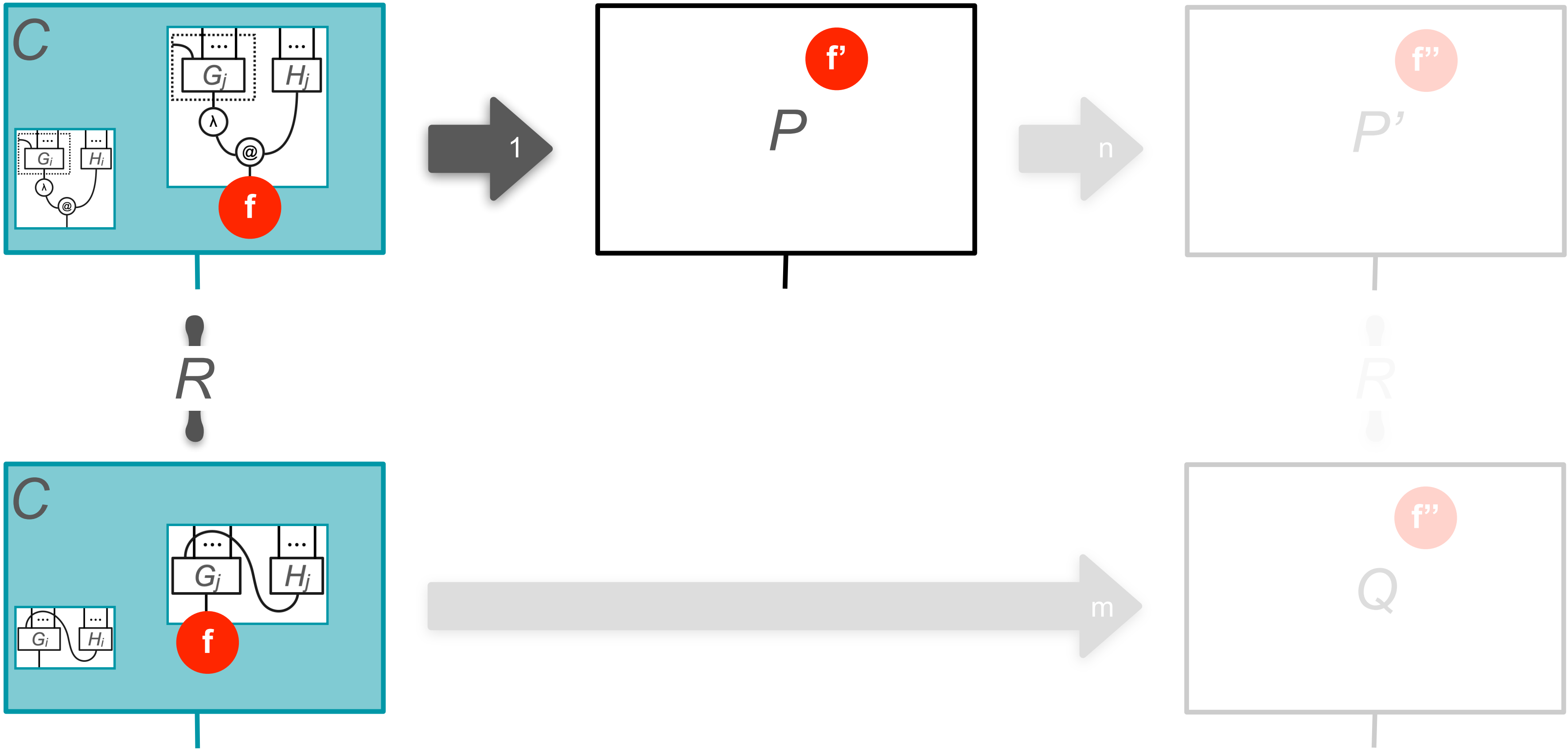
Partial Characterisation Theorem

***Robust** and **safe** templates induce observational equivalences.*

(for deterministic & "reasonable" languages)

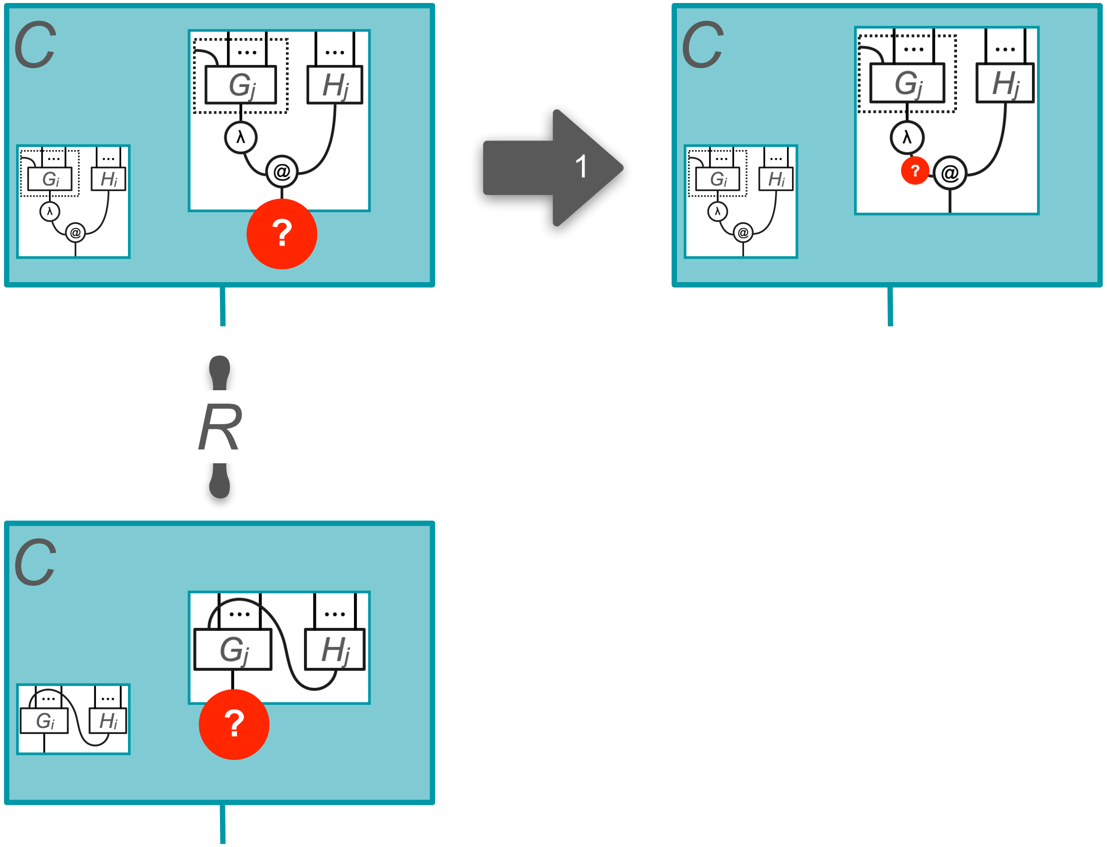
# *Safety* of cbv linear β-template

Aim: when focus ? or ✓ enters $G_j$,



# *Safety* of cbv linear β-template

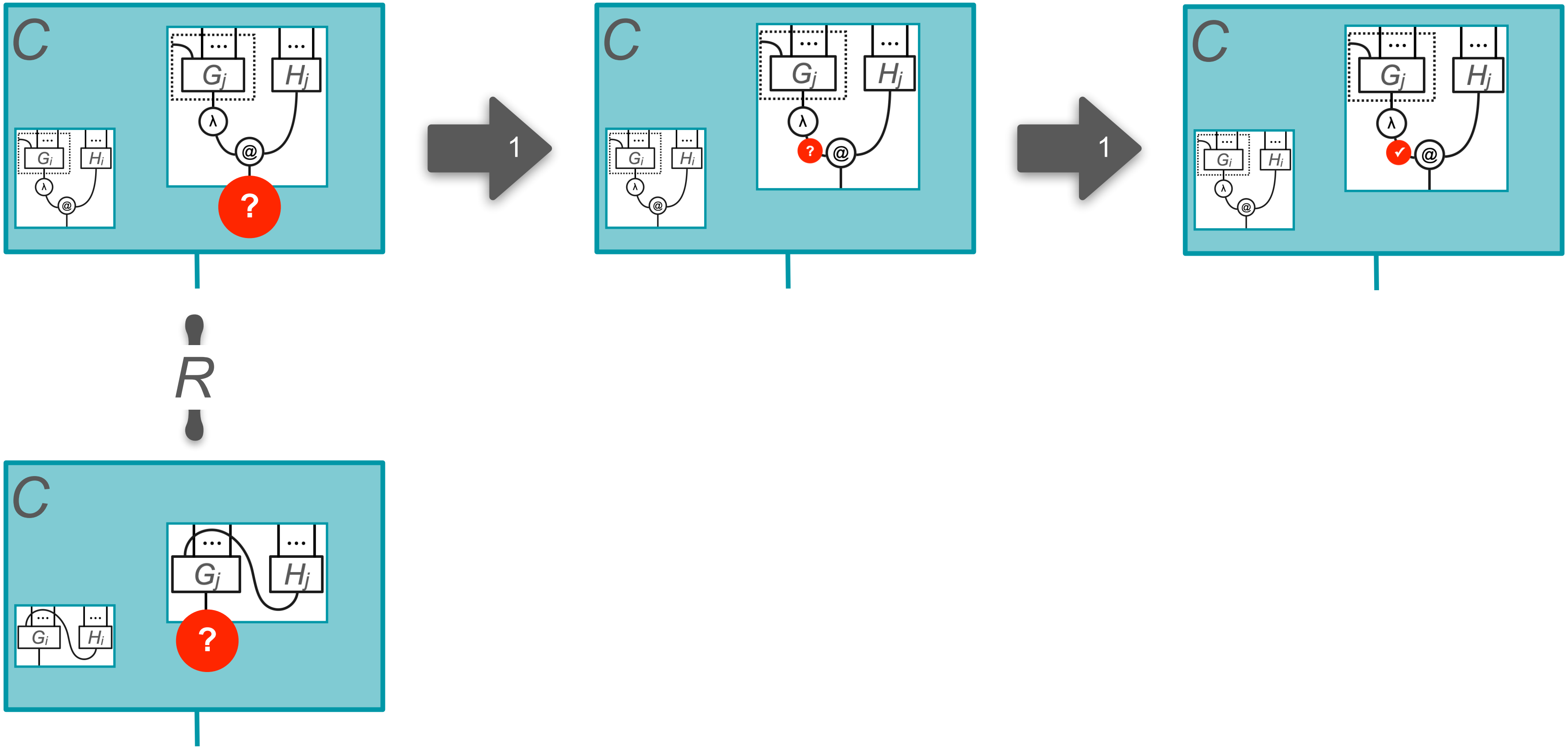
Key scenario: when focus ? enters $G_j$,



Muroya (RIMS, Kyoto U.)

# *Safety* of cbv linear β-template

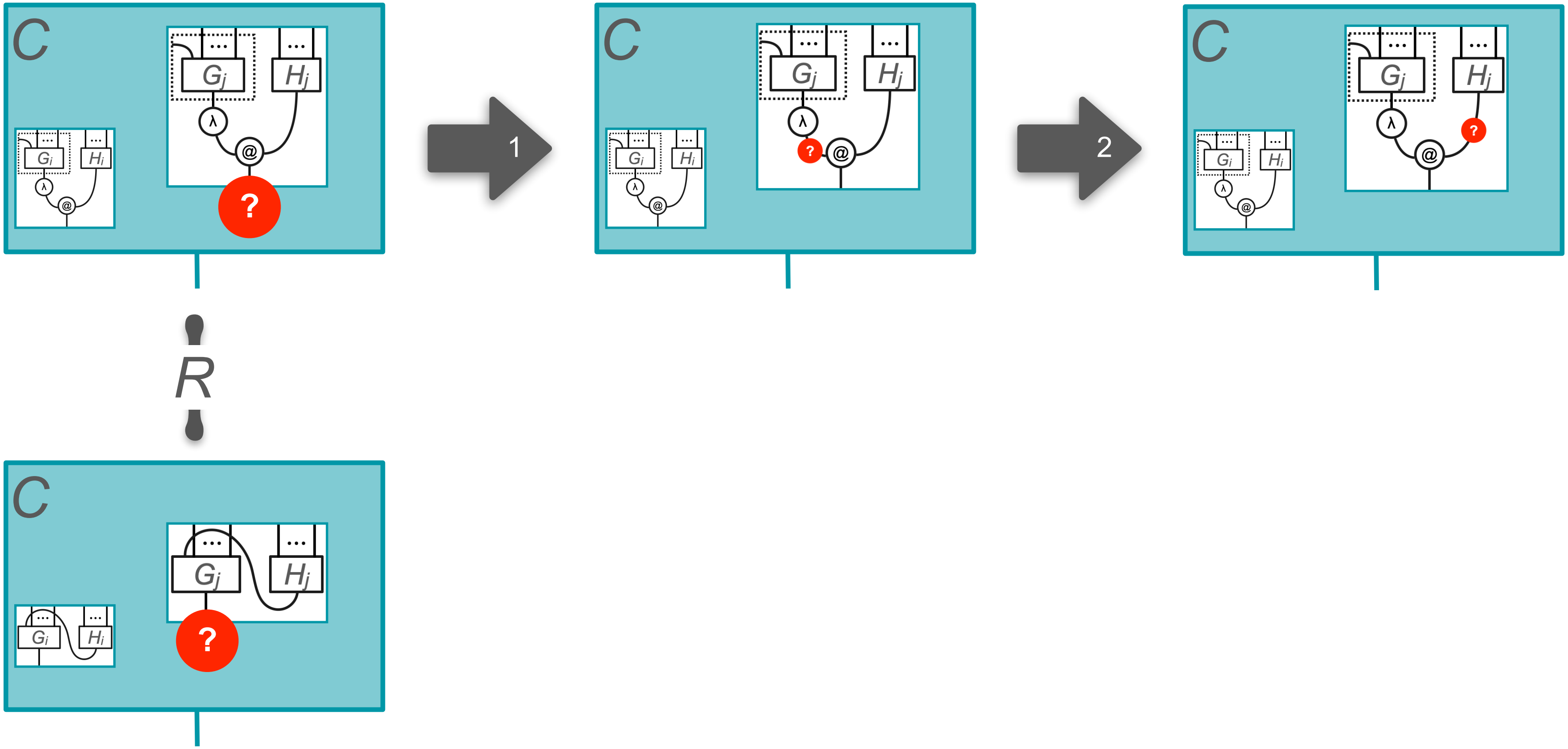
Key scenario: when focus ? enters $G_j$,



Muroya (RIMS, Kyoto U.)

# *Safety* of cbv linear β-template

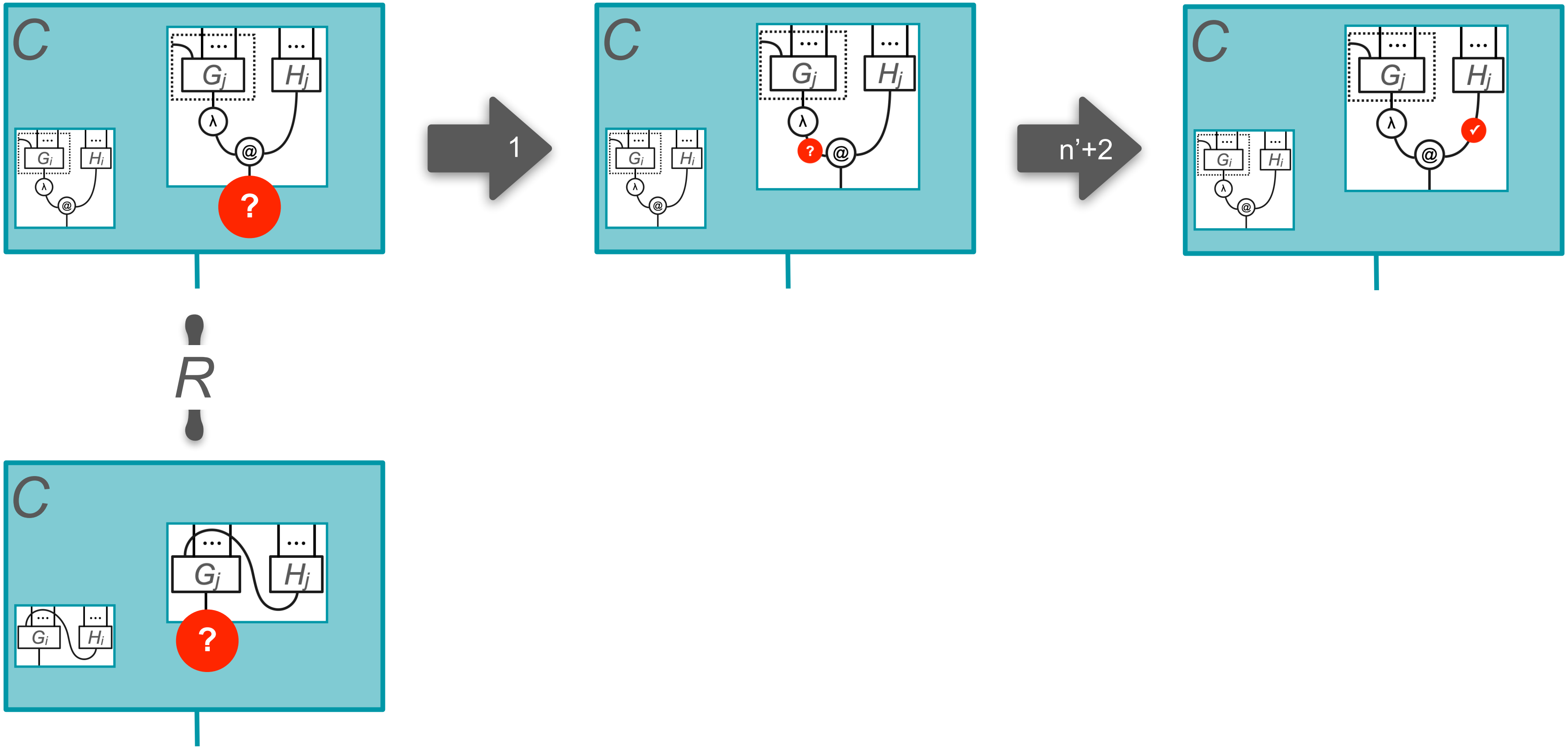
Key scenario: when focus ? enters $G_j$,



# *Safety* of cbv linear β-template

Key scenario: when focus ? enters $G_j$,

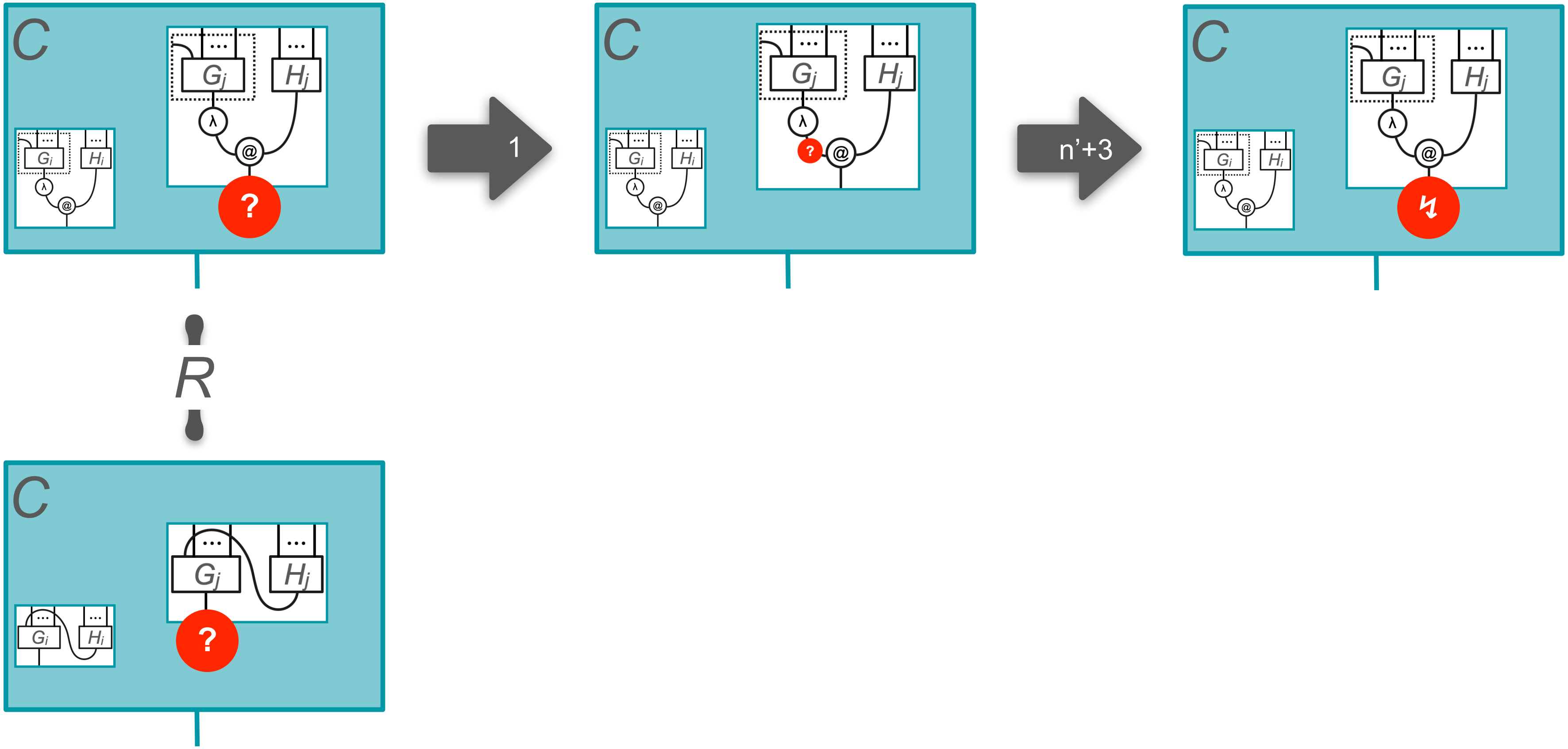
because $H_j$ represents a value,



# *Safety* of cbv linear β-template

Key scenario: when focus ? enters $G_j$,

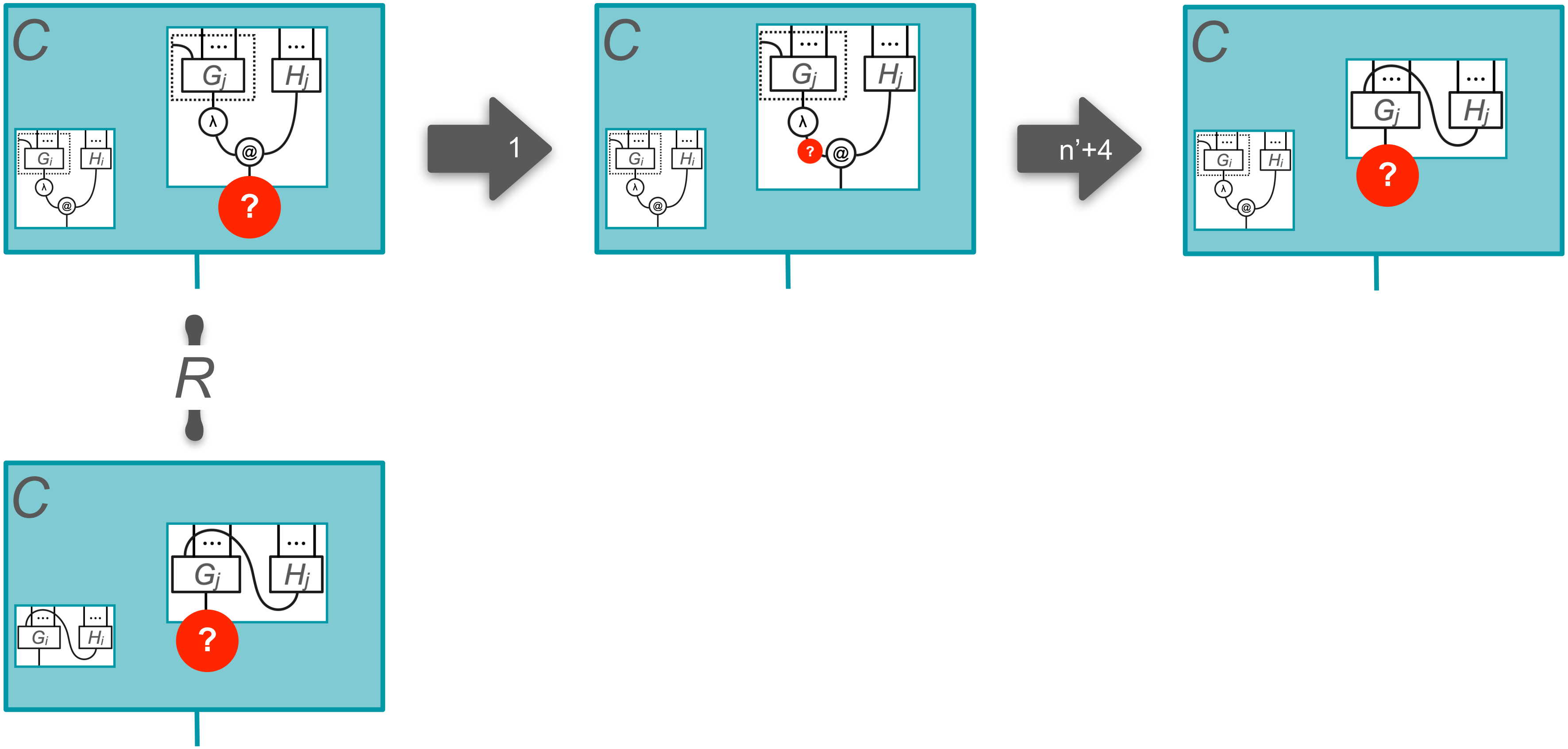
because $H_j$ represents a value,



# *Safety* of cbv linear β-template

Key scenario: when focus ? enters $G_j$,

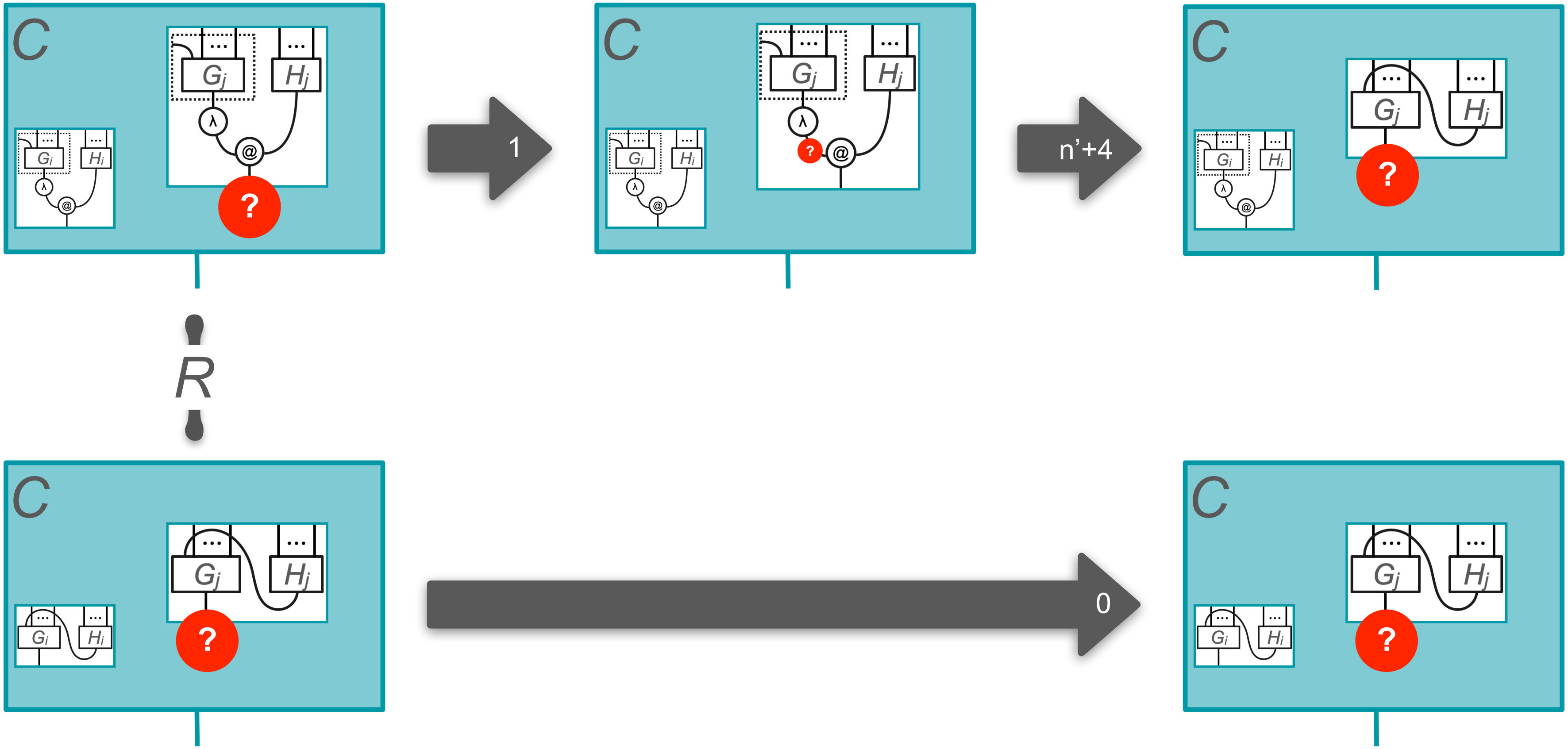
because $H_j$ represents a value,



# *Safety* of cbv linear β-template

Key scenario: when focus ? enters $G_j$,

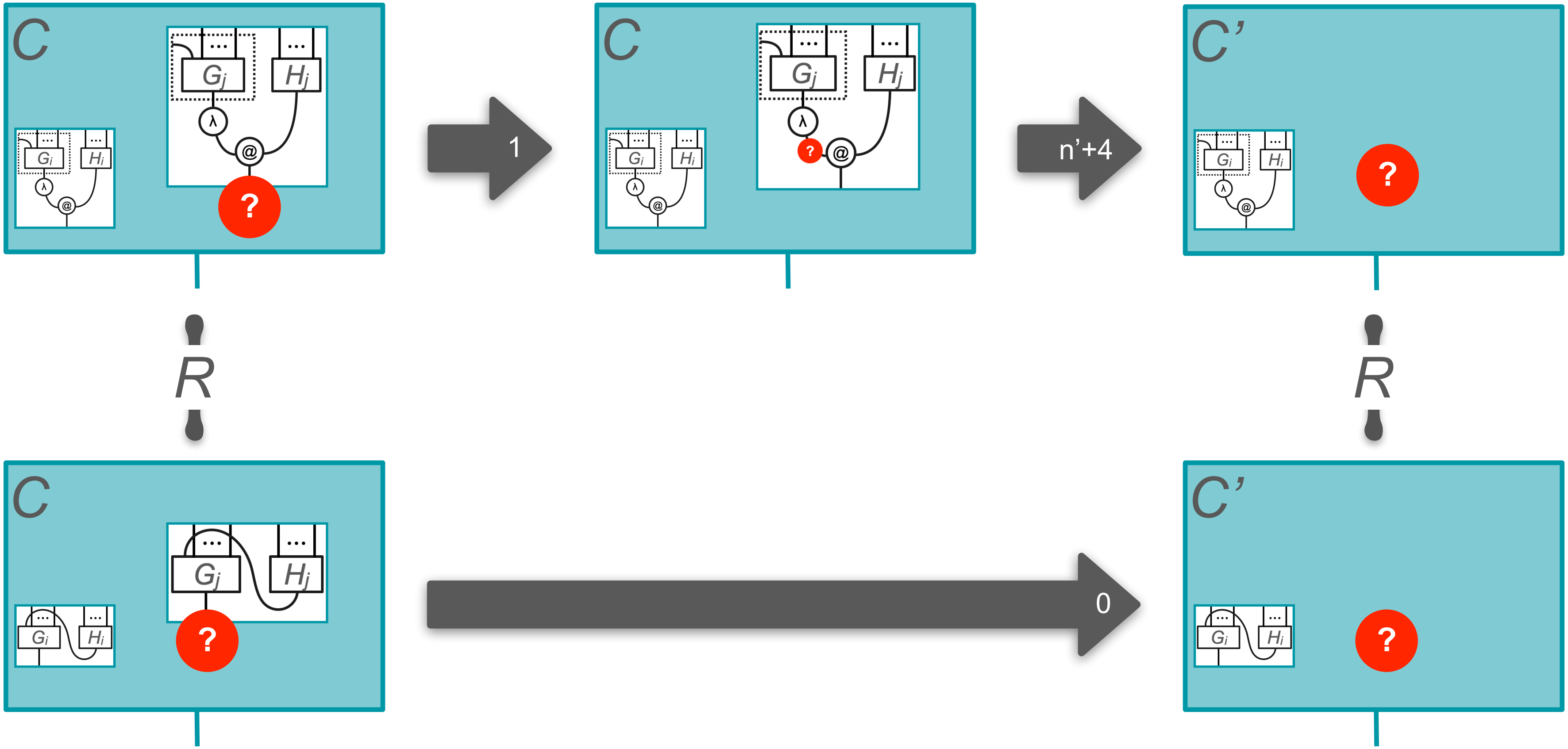
because $H_j$ represents a value,



# *Safety* of cbv linear β-template

Key scenario: when focus ? enters $G_j$,

because $H_j$ represents a value,

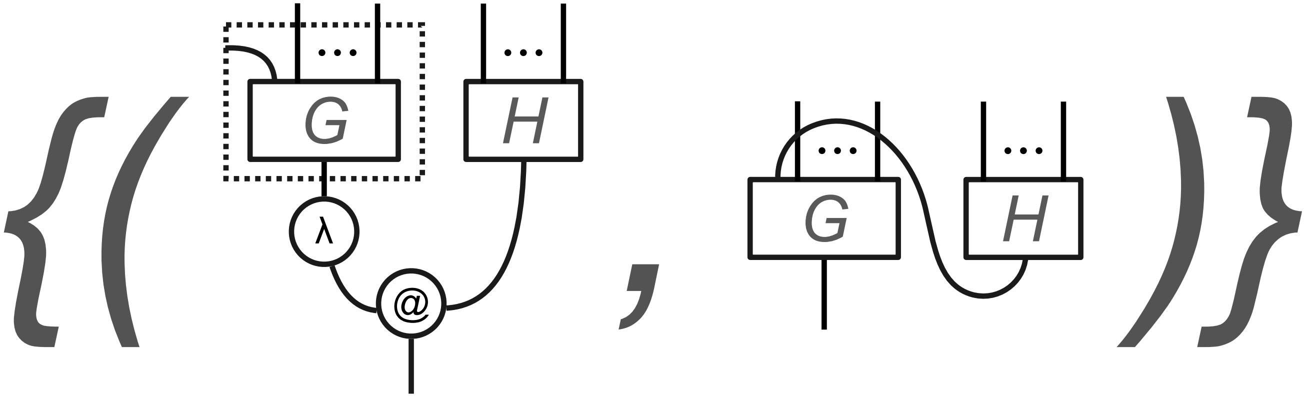


Muroya (RIMS, Kyoto U.)

# Example: cbv linear β-law

Proof methodology:

1. prepare *the cbv linear β-template*:



where *H* represents a *value*

2. prove that the cbv linear β-template is **robust** and **safe**

Partial Characterisation Theorem

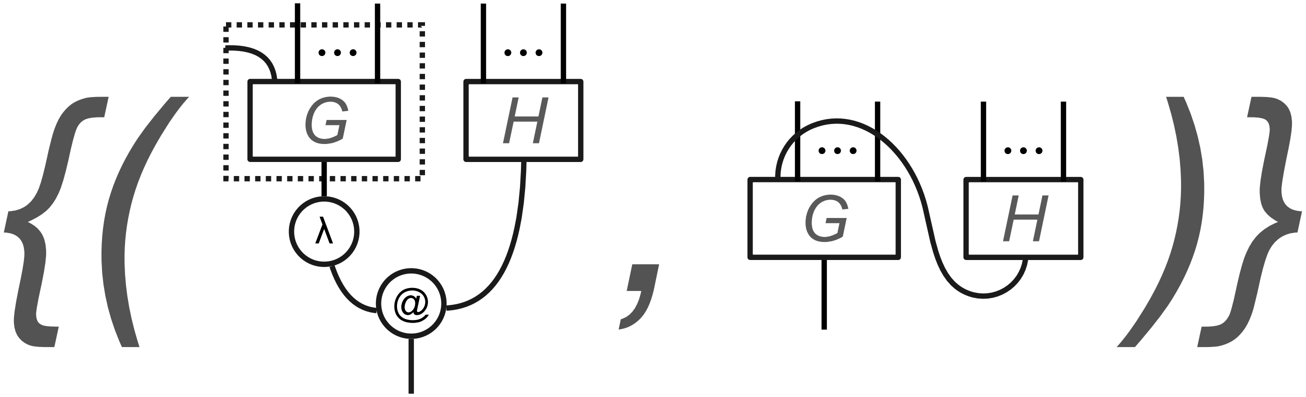
***Robust** and **safe** templates induce observational equivalences.*

(for deterministic & "reasonable" languages)

# Example: cbv linear β-law

Proof methodology:

1. prepare *the cbv linear β-template*:



where *H* represents a *value*

2. prove that the cbv linear β-template is **robust** and **safe**

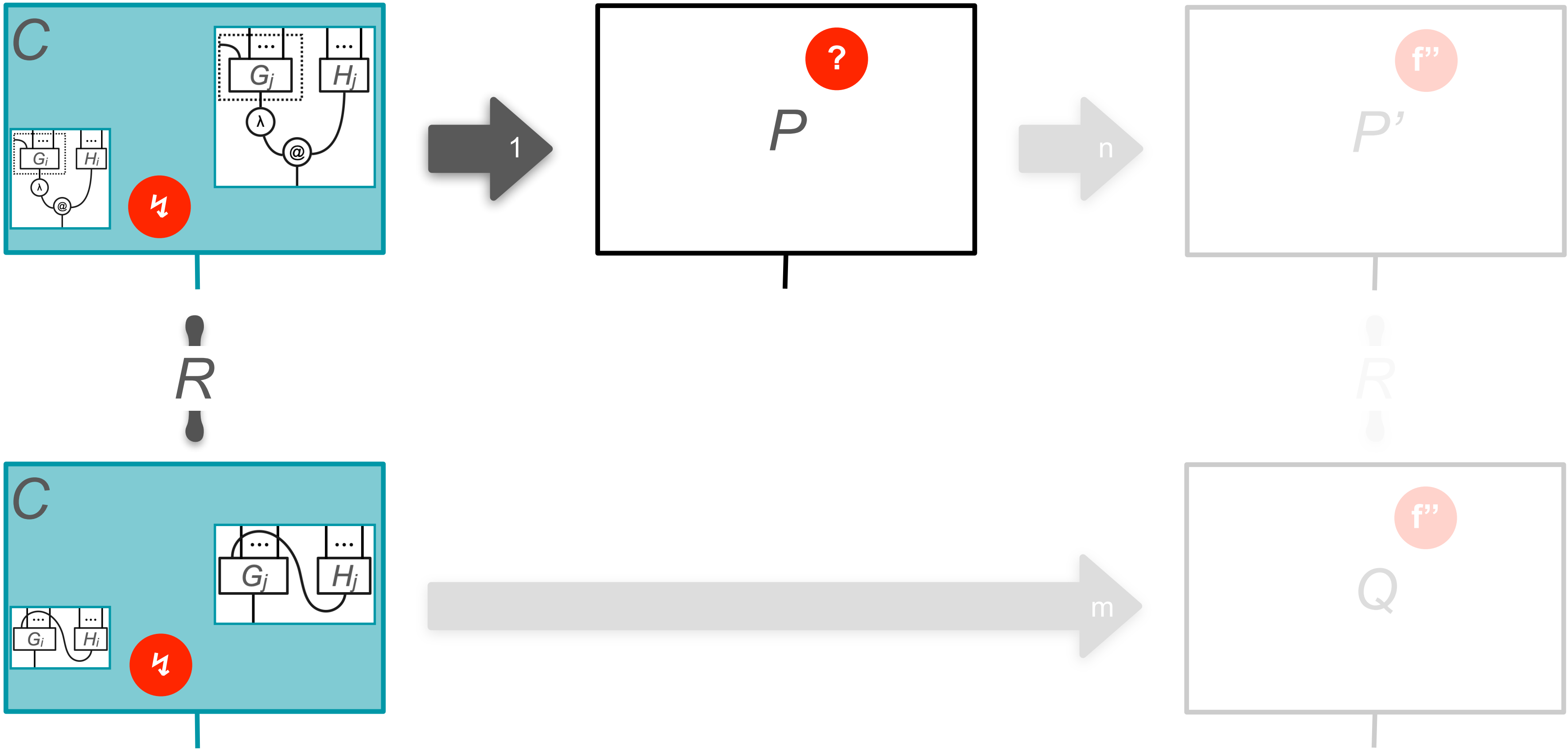
Partial Characterisation Theorem

***Robust** and **safe** templates induce observational equivalences.*

(for deterministic & “reasonable” languages)

# *Robustness* of cbv linear β-template

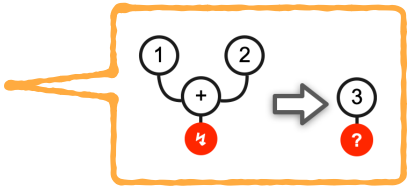
Aim: for any possible rewrite triggered by focus ↯ ,

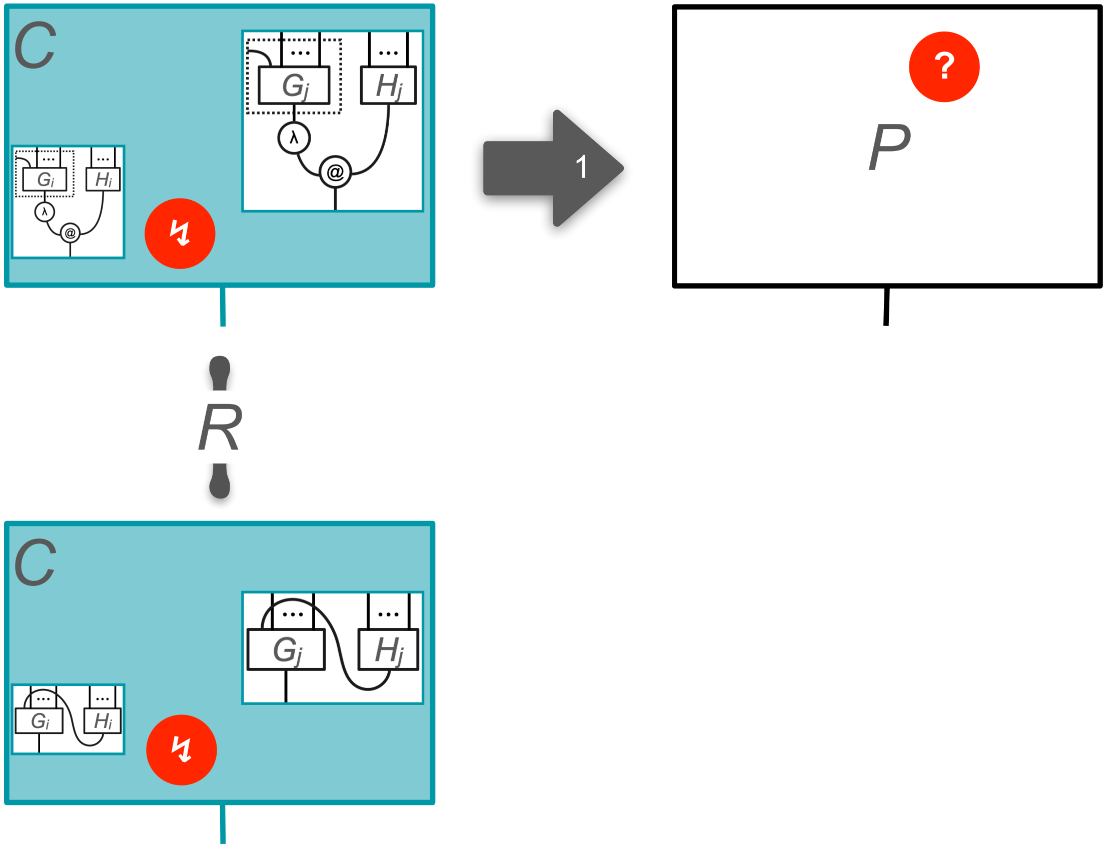


Muroya (RIMS, Kyoto U.)

# *Robustness* of cbv linear β-template

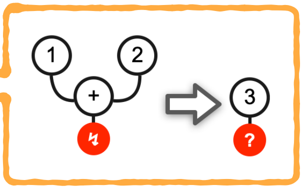
Example (1) arithmetic rewrite

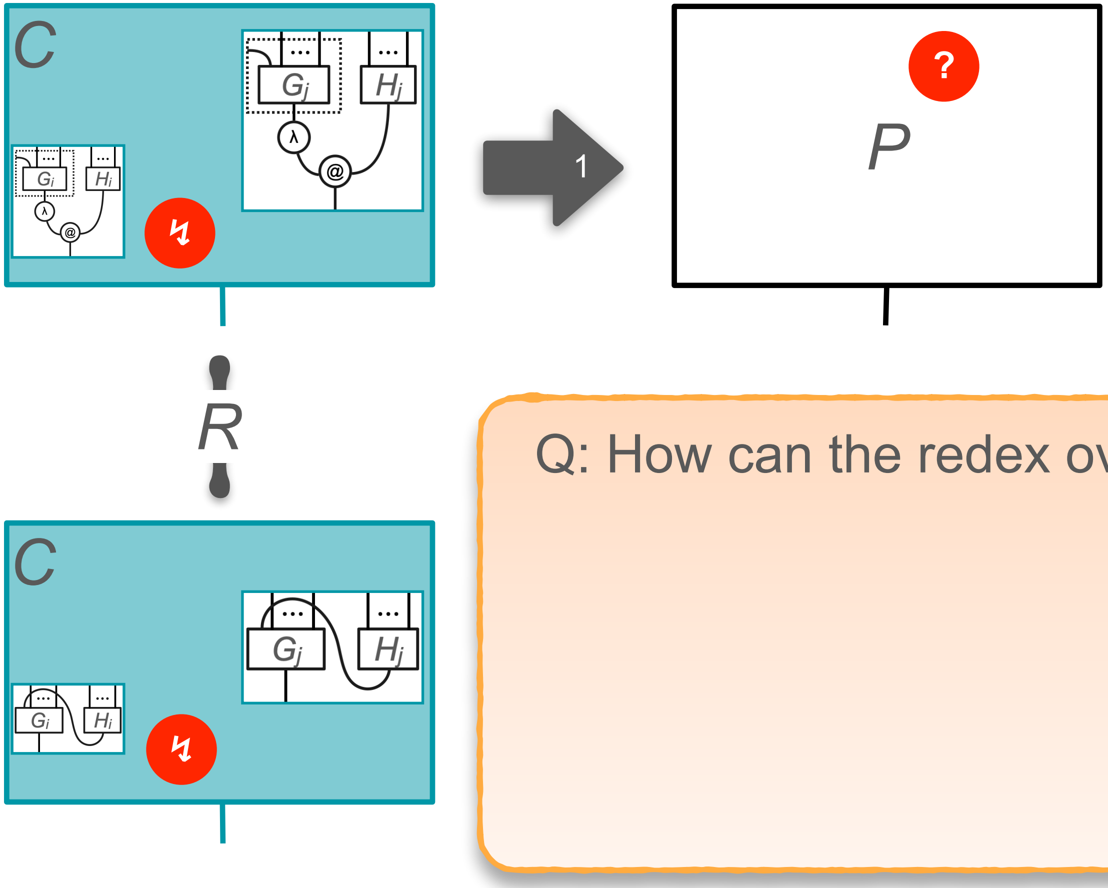




# *Robustness* of cbv linear β-template

Example (1) arithmetic rewrite

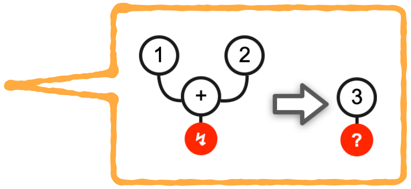


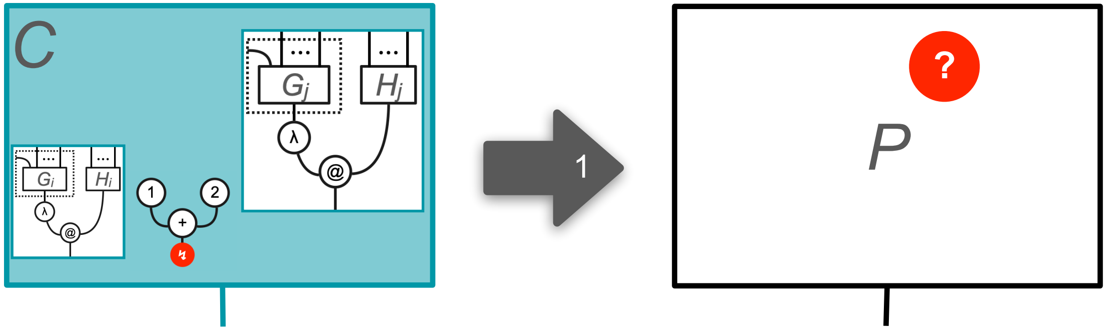


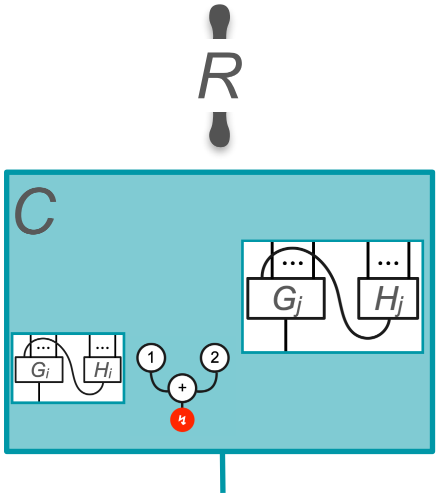
Q: How can the redex overlap with the template?

# *Robustness* of cbv linear β-template

Example (1) arithmetic rewrite







Q: How can the redex overlap with the template?

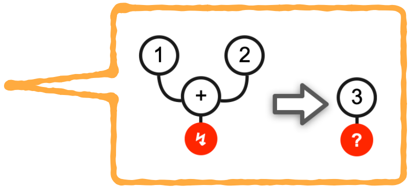
A: No overlap is possible!

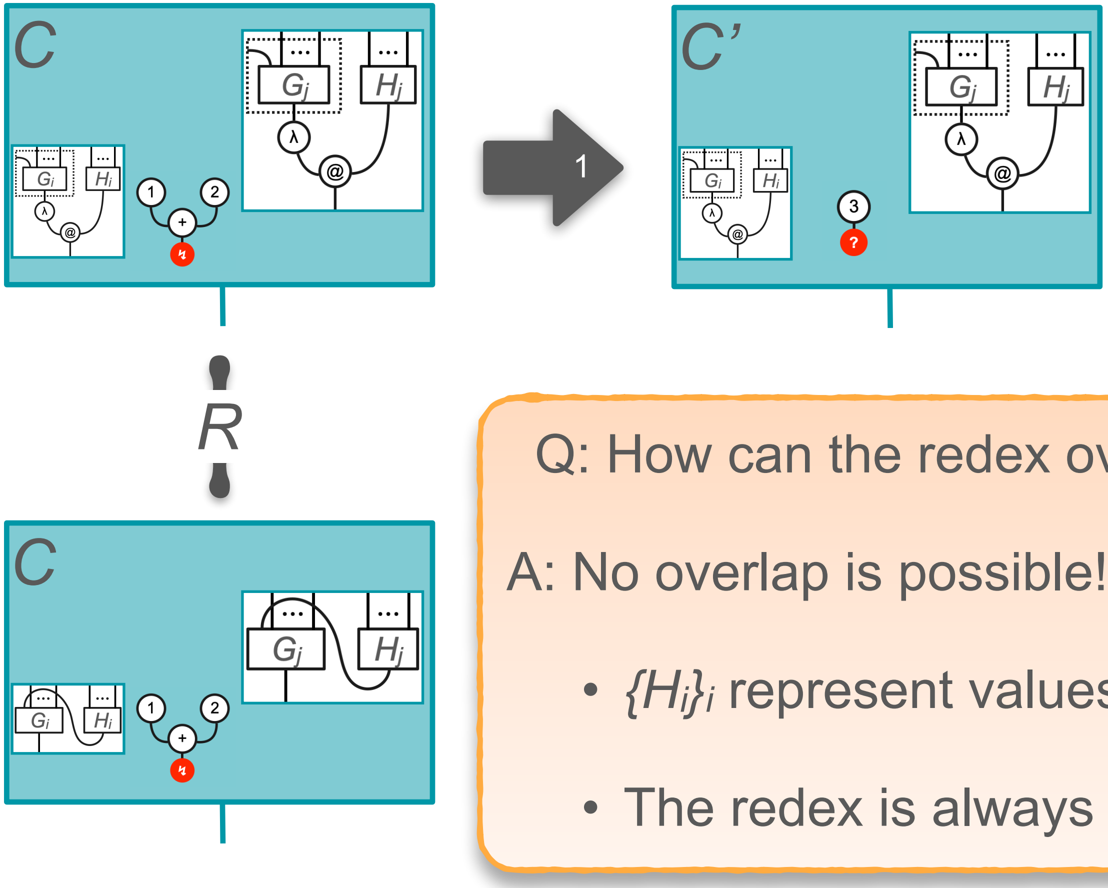
- *{H$_j$}$_i$* represent values.
- The redex is always outside a box .

Muroya (RIMS, Kyoto U.)

# *Robustness* of cbv linear β-template

Example (1) arithmetic rewrite





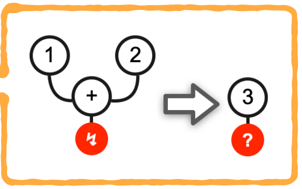
Q: How can the redex overlap with the template?

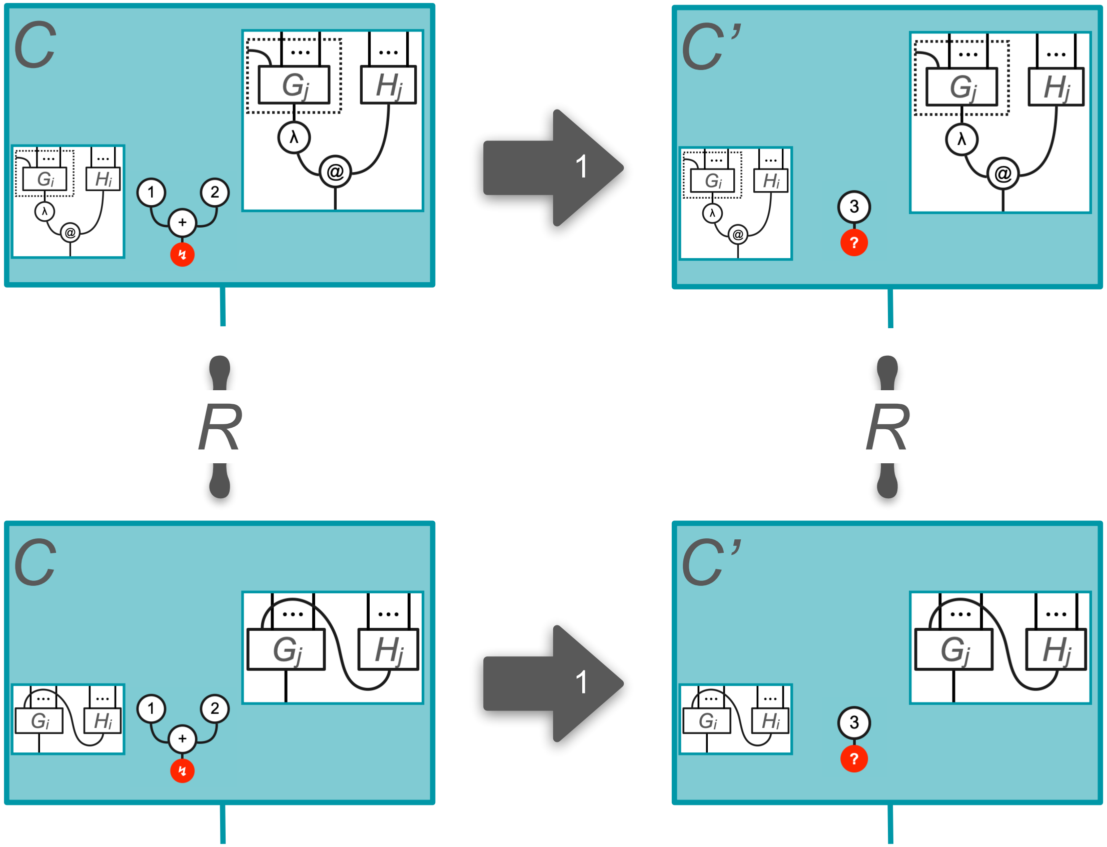
A: No overlap is possible!

- *{Hj}j* represent values.
- The redex is always outside a box.

# *Robustness* of cbv linear β-template

Example (1) arithmetic rewrite

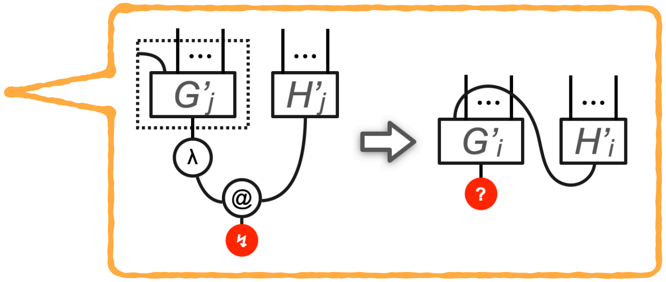


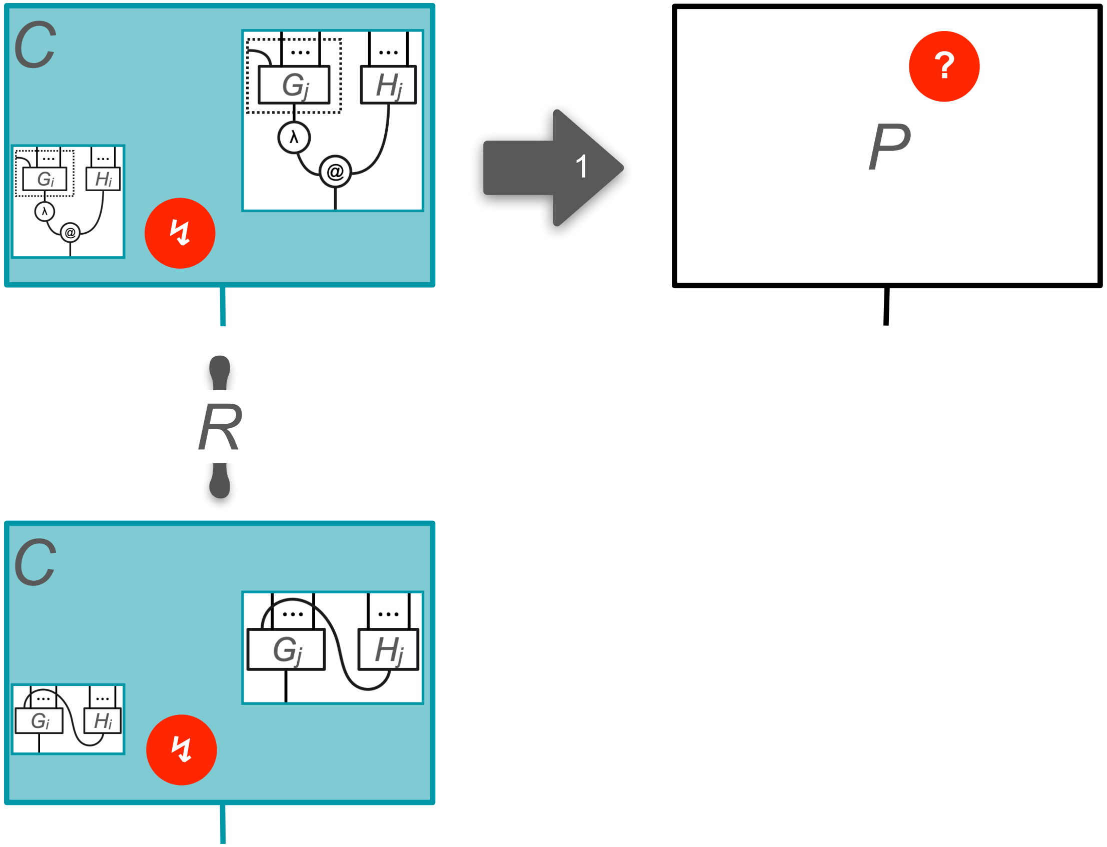


robustness relative to arithmetic rewrite

# *Robustness* of cbv linear β-template

Example (2) cbv linear β-reduction

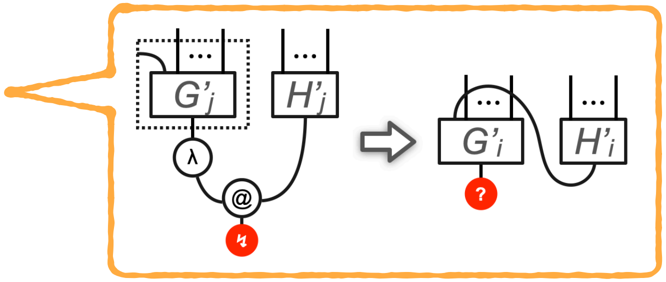


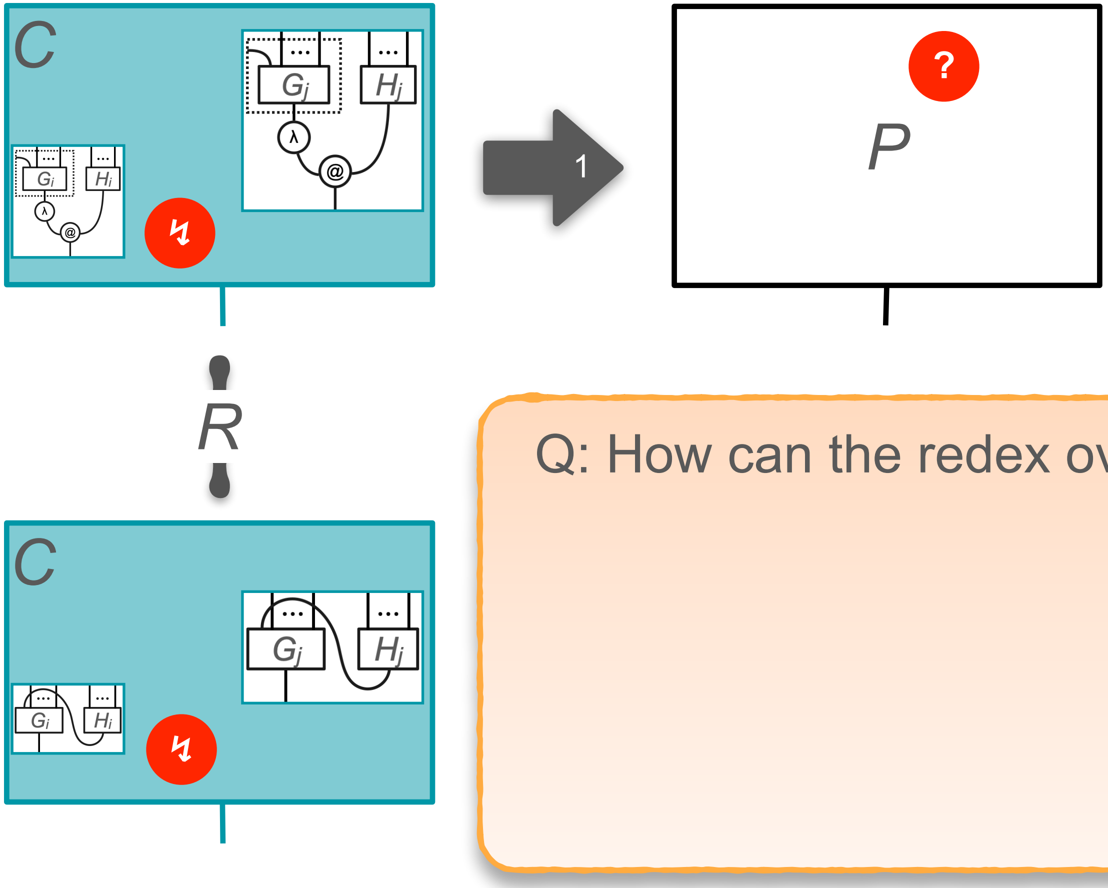


Muroya (RIMS, Kyoto U.)

# *Robustness* of cbv linear β-template

Example (2) cbv linear β-reduction

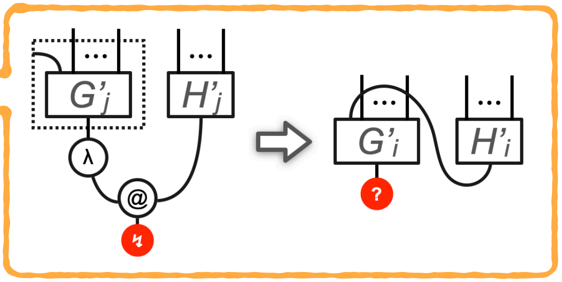


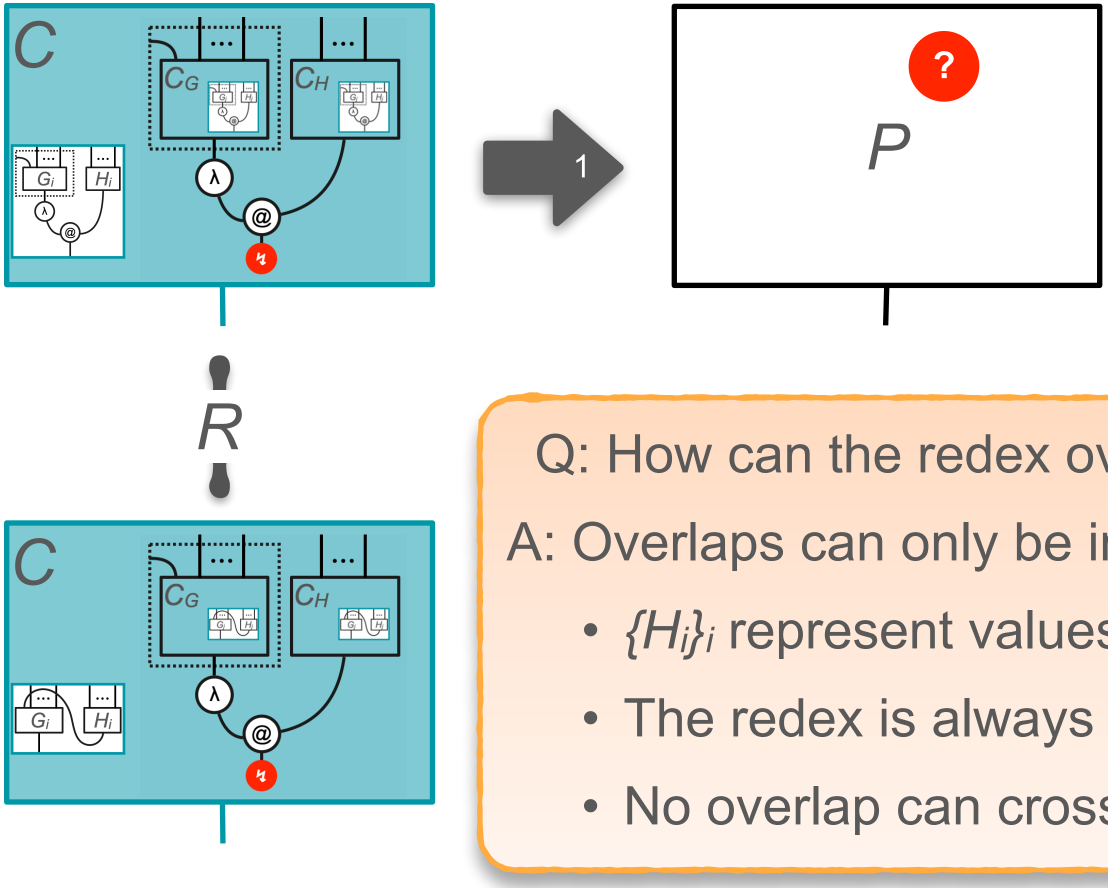


Q: How can the redex overlap with the template?

# *Robustness* of cbv linear β-template

Example (2) cbv linear β-reduction





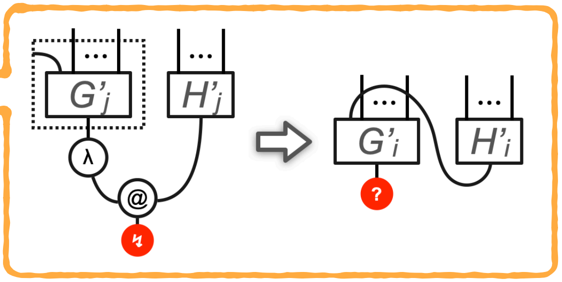
Q: How can the redex overlap with the template?

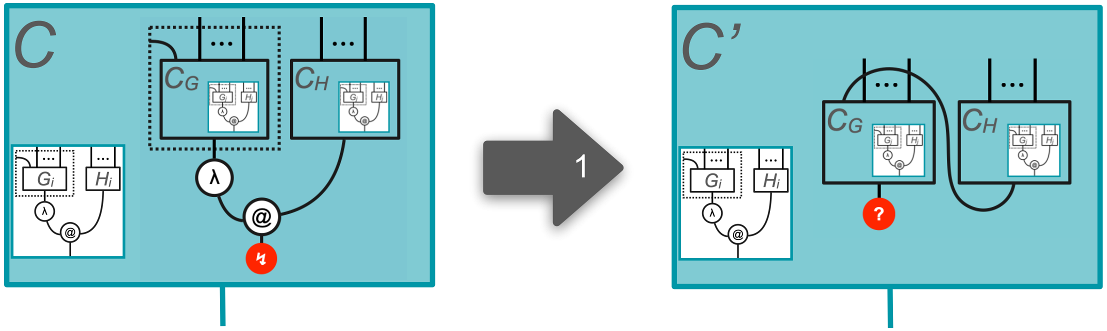
A: Overlaps can only be inside boxes of the redex.

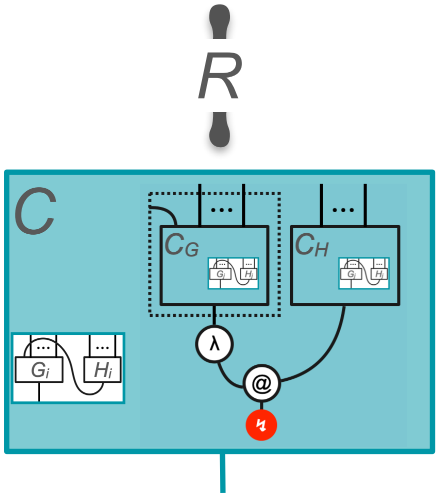
- $\{H_i\}_i$ represent values.
- The redex is always outside a box .
- No overlap can cross the boundary of a box .

# *Robustness* of cbv linear β-template

Example (2) cbv linear β-reduction







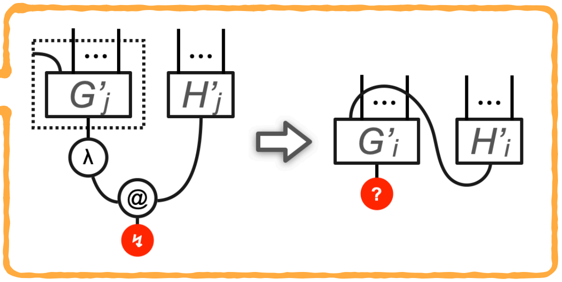
Q: How can the redex overlap with the template?

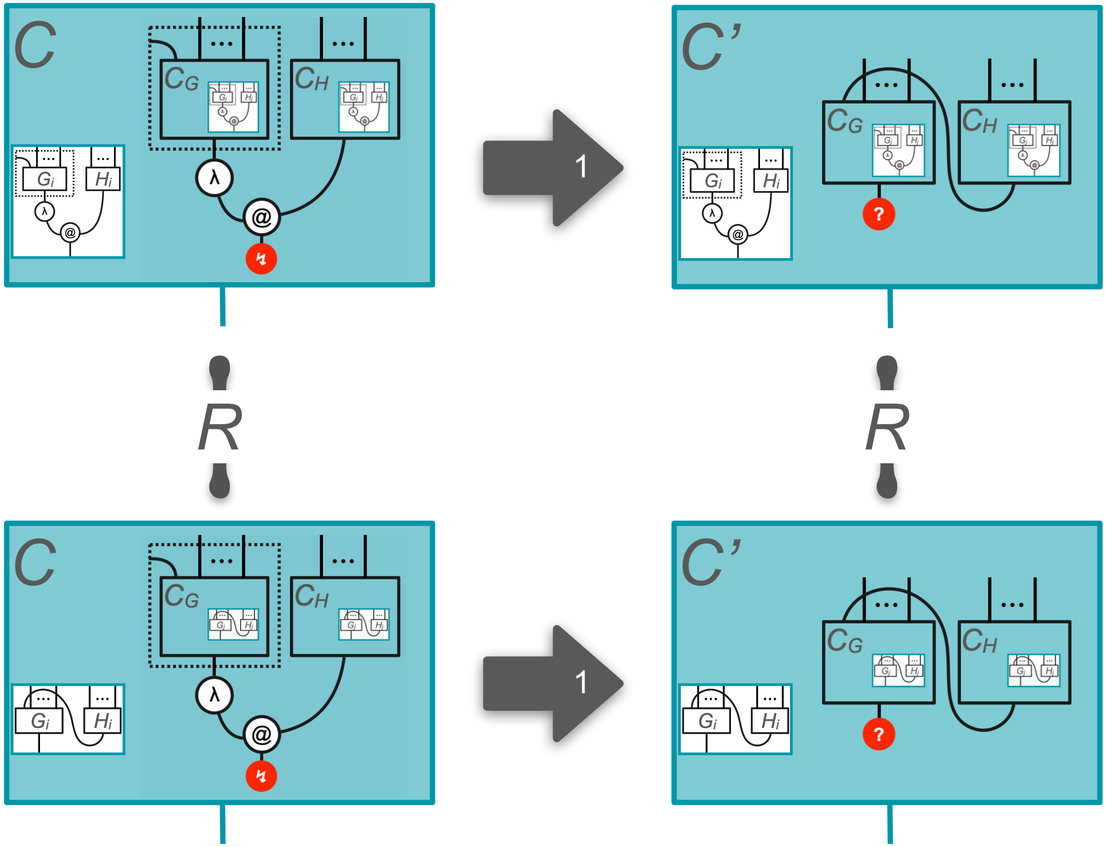
A: Overlaps can only be inside boxes of the redex.

- *$\{H_i\}_i$* represent values.
- The redex is always outside a box .
- No overlap can cross the boundary of a box .

# *Robustness* of cbv linear β-template

Example (2) cbv linear β-reduction



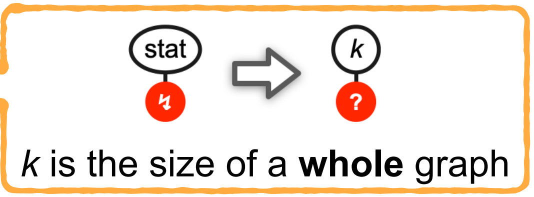


robustness relative to

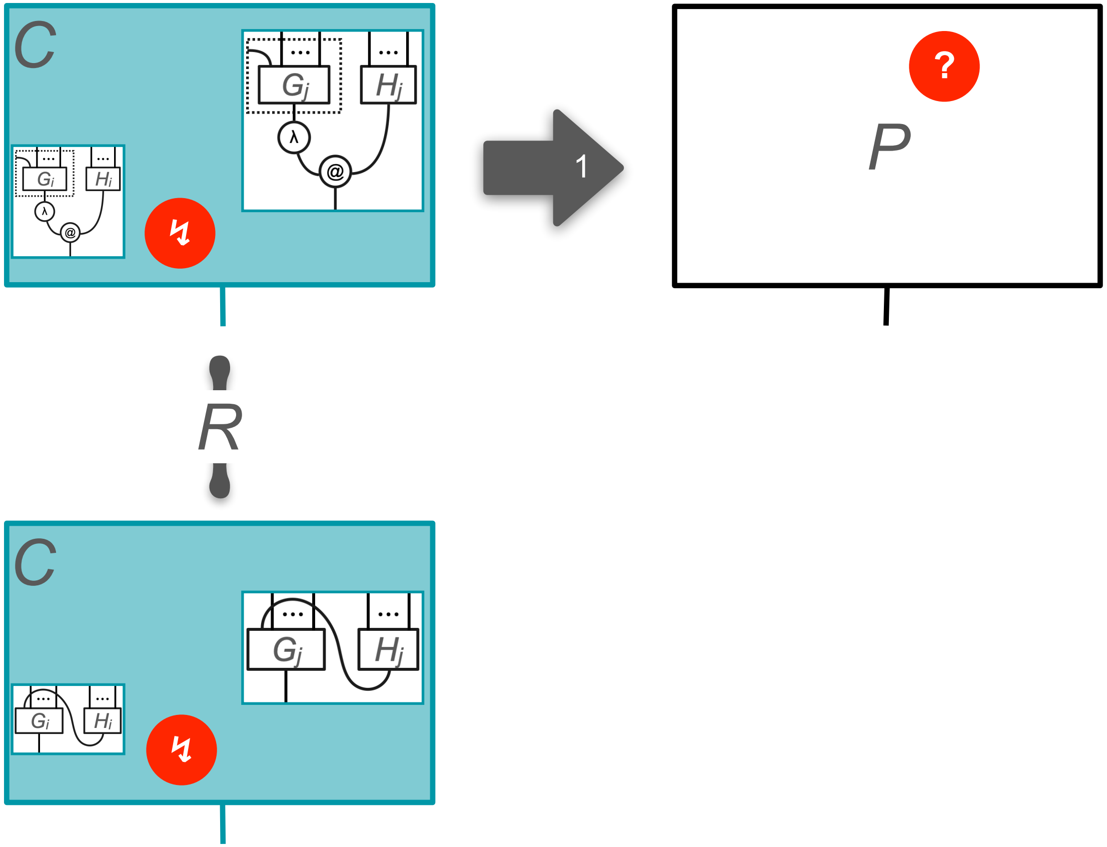
arithmetic rewrite

# *Robustness* of cbv linear β-template

Example (3) measurement of space usage

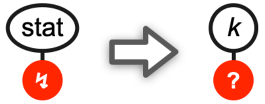


*k* is the size of a **whole** graph

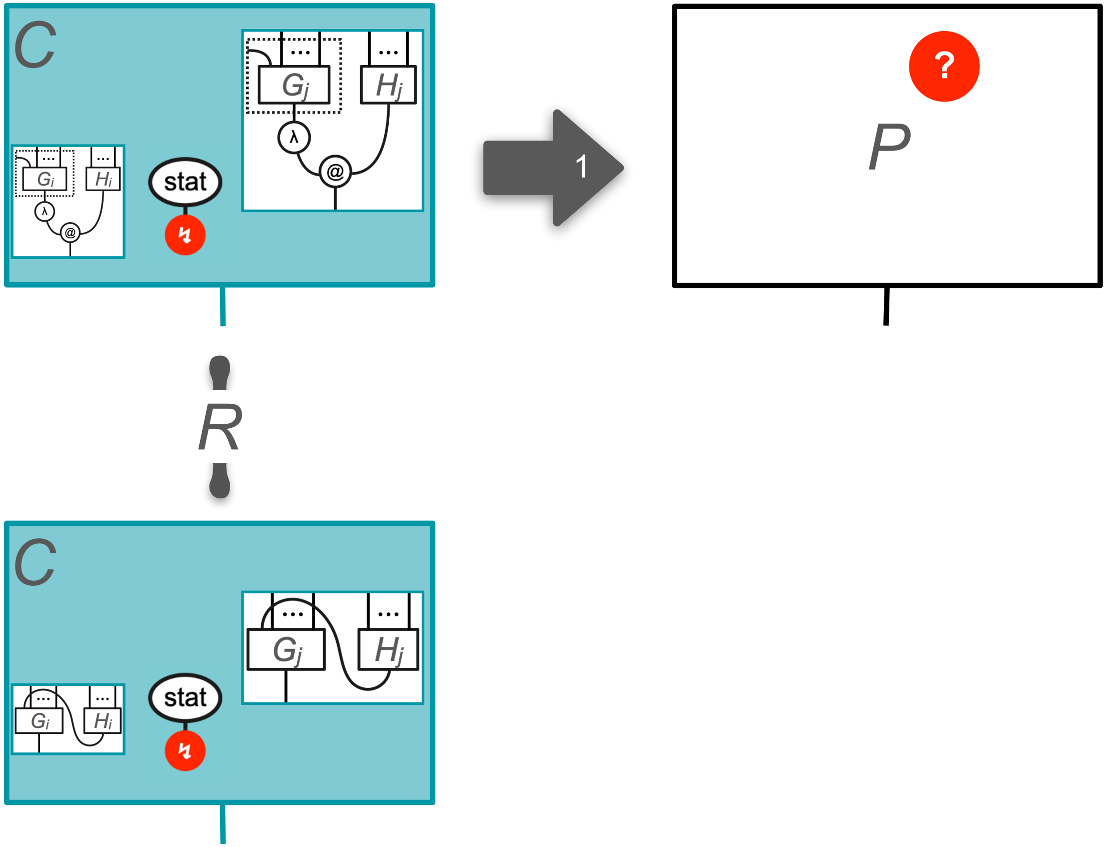


# *Robustness* of cbv linear β-template

Example (3) measurement of space usage

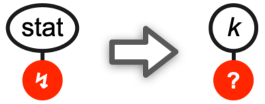


*k* is the size of a **whole** graph

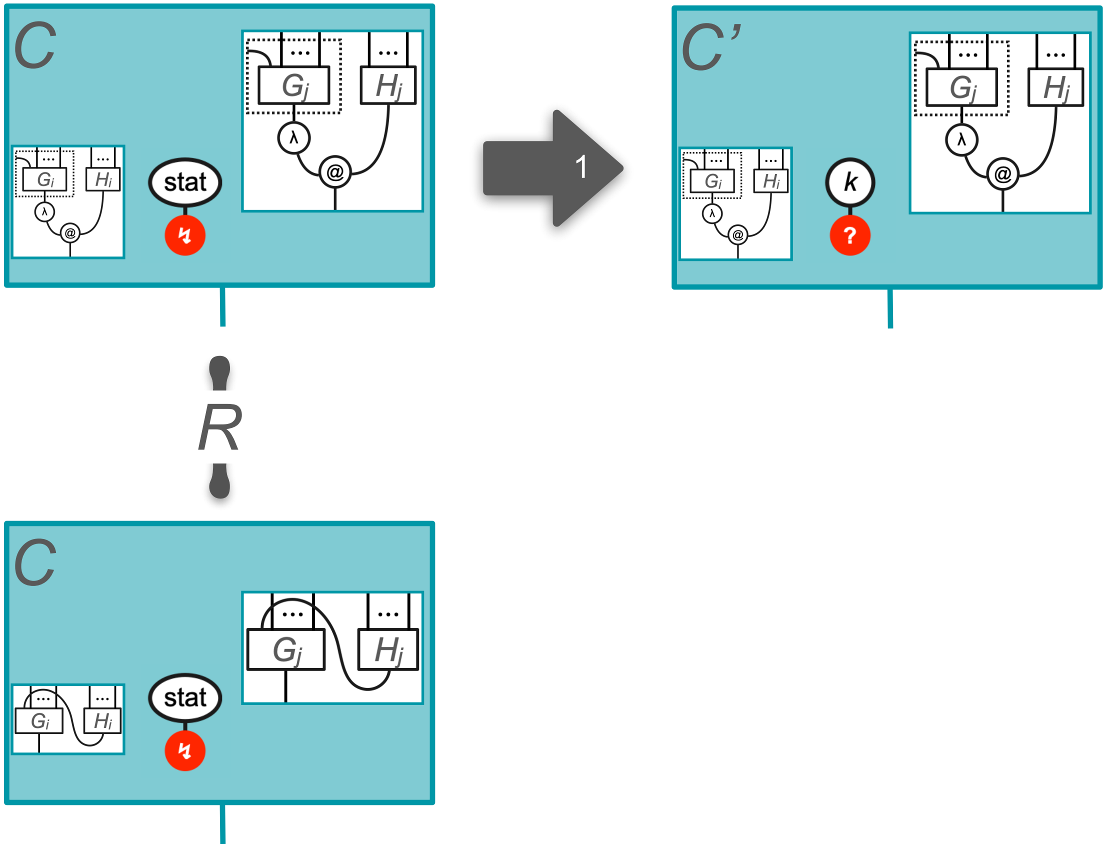


# *Robustness* of cbv linear β-template

## Example (3) measurement of space usage

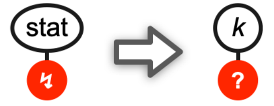


*k* is the size of a **whole** graph

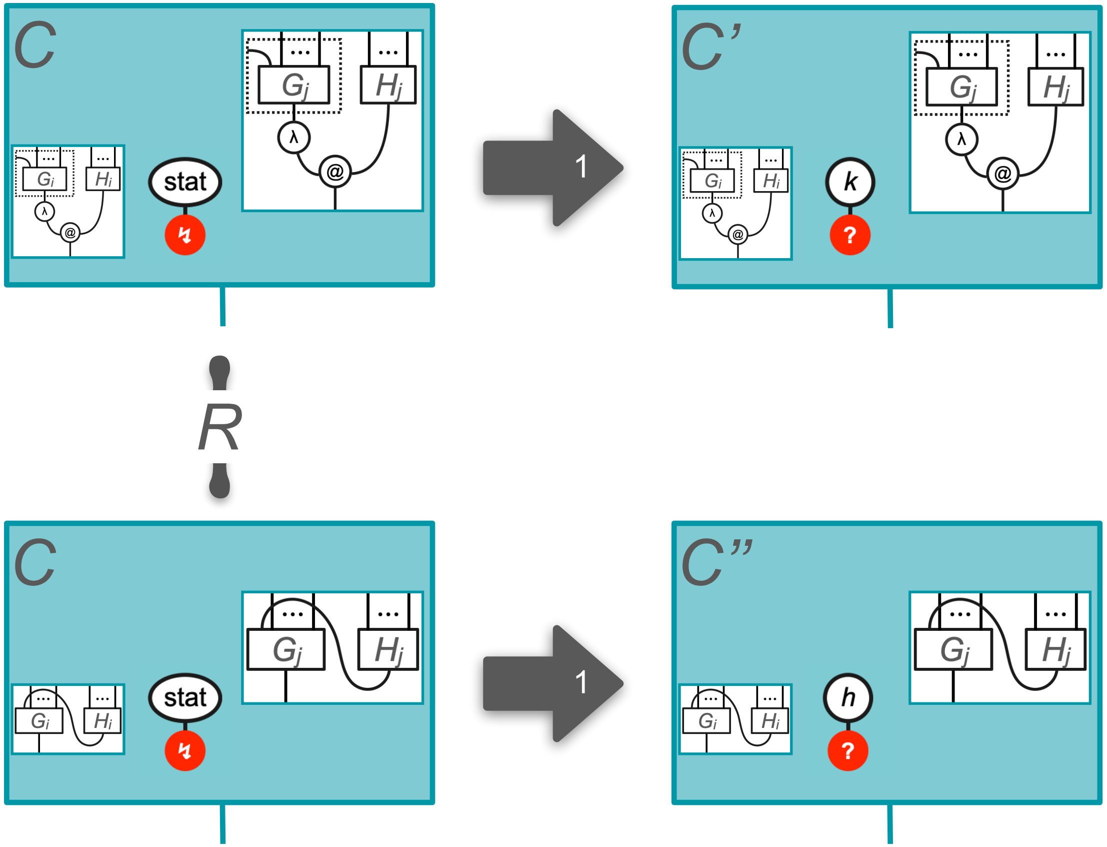


# *Robustness* of cbv linear β-template

Example (3) measurement of space usage

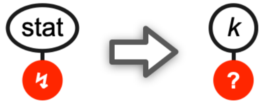


*k* is the size of a **whole** graph

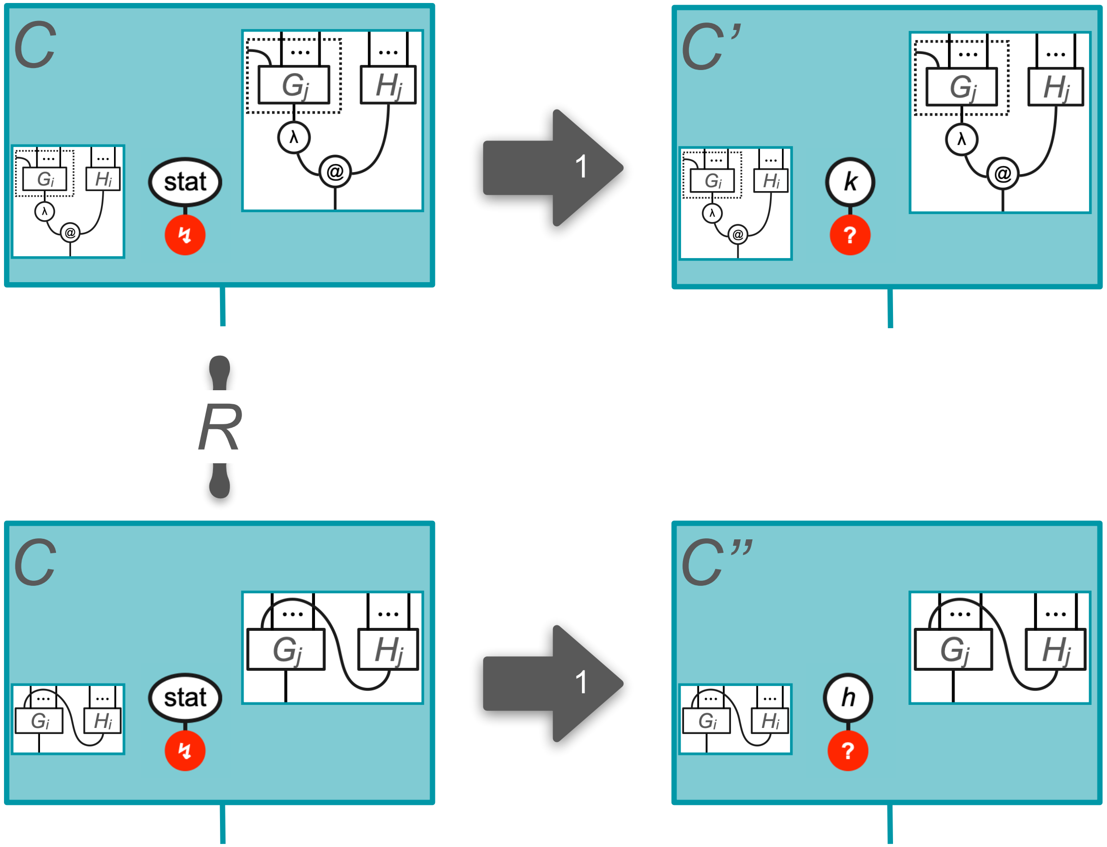


# *Robustness* of cbv linear β-template

## Example (3) measurement of space usage



*k* is the size of a **whole** graph



~~robustness relative to~~

~~`stat`~~

due to λ and @ ,

$k \geq h$

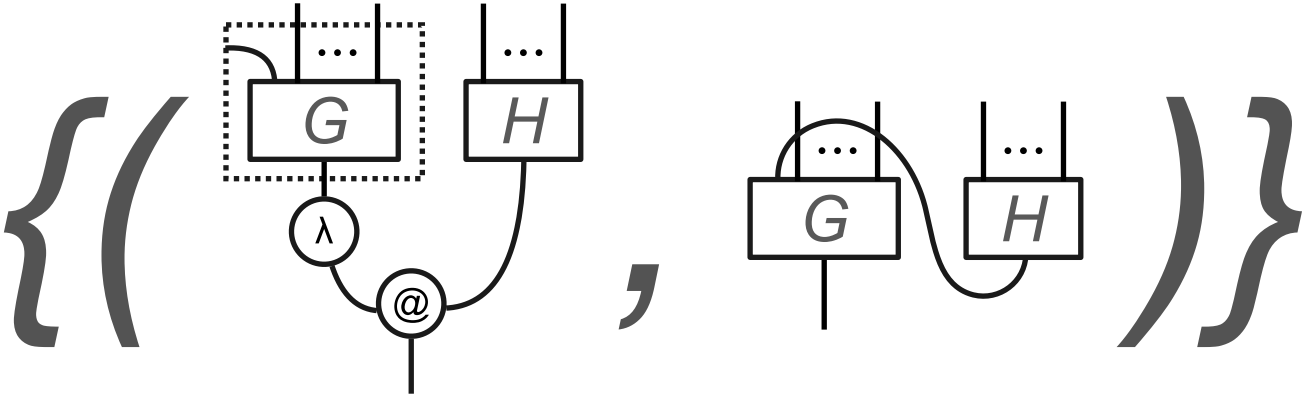
and hence possibly

$C' \neq C''$

# Example: cbv linear β-law

Proof methodology:

1. prepare *the cbv linear β-template*:



where *H* represents a *value*

2. prove that the cbv linear β-template is **robust** and **safe**

… relative to arithmetic and cbv linear β-reduction

# Example: cbv linear β-law

Proof methodology:

2. prove that the cbv linear β-template is **robust** and **safe**

   … relative to arithmetic and cbv linear β-reduction

3. apply the Partial Characterisation Theorem

Partial Characterisation Theorem

***Robust** and **safe** templates induce observational equivalences.*

(for deterministic & “reasonable” languages)

# Example: cbv linear β-law

Proof methodology:

2. prove that the cbv linear β-template is **robust** and **safe**

   … relative to arithmetic and cbv linear β-reduction

3. apply the Partial Characterisation Theorem

<u>Proposition (cbv linear β-law)</u>
*The cbv linear β-template induces observational equivalence, if arithmetic and cbv linear β-reduction are the only computation allowed.*

# Partiality

Partial Characterisation Theorem

***Robust*** *and* ***safe*** *templates induce observational equivalences.*

(for deterministic & “reasonable” languages)

- The cbv linear β-template is not robust relative to `stat` (measurement of space usage).
- What can we say about the cbv linear β-law, in the presence of `stat`?

# Partiality

Partial Characterisation Theorem

***Robust** and **safe** templates induce observational equivalences.*

(for deterministic & “reasonable” languages)

- The cbv linear β-template is not robust relative to `stat` (measurement of space usage).
- What can we say about the cbv linear β-law, in the presence of `stat`?
  - The counterexample of robustness would provide a counterexample of the law, in the presence of conditional statements (e.g. `if`).
  - The template can be extended so it is robust relative to `stat`, if a language allows no computation to distinguish numbers.

# Partiality

Partial Characterisation Theorem

***Robust*** *and* ***safe*** *templates induce observational equivalences.*

(for deterministic & “reasonable” languages)

If a template is safe but fails to be robust, either:

(1) The intended observational equivalence fails too.

- Counterexamples of robustness would suggest how the observational equivalence could be violated.

(2) The intended observational equivalence actually holds.

- There may be a bigger, robust, template.
- Counterexamples of robustness would suggest how the template could be extended.

# Overview

1. Motivation: robustness of observational equivalence

2. Hypernet semantics

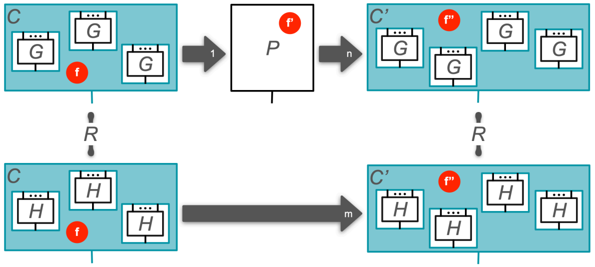
3. Locality & step-wise reasoning

4. Example: cbv linear β-law

# Conclusion

- a (general) framework for analysing and proving robustness of observational equivalence
    - hypernet semantics: a *graphical* abstract machine
    - *local* & *step-wise* reasoning to prove observational equivalence, with the concept of *robustness*
- current key limitation: determinism

# Future directions



- dealing with nondeterminism
    - overcoming unsoundness of *-simulation
- Sand's improvement theory
    - incorporating cost reduction in observational equivalence
    - introducing quantitative restrictions on *-simulation
- (semi-)automating robustness & safety check
    - exploiting techniques of critical pair analysis