A graph model for data and workflow provenance

Umut Acar, Peter Buneman, **James Cheney**,
Natalia Kwasnikowska,
Jan van den Bussche, & Stijn Vansummeren

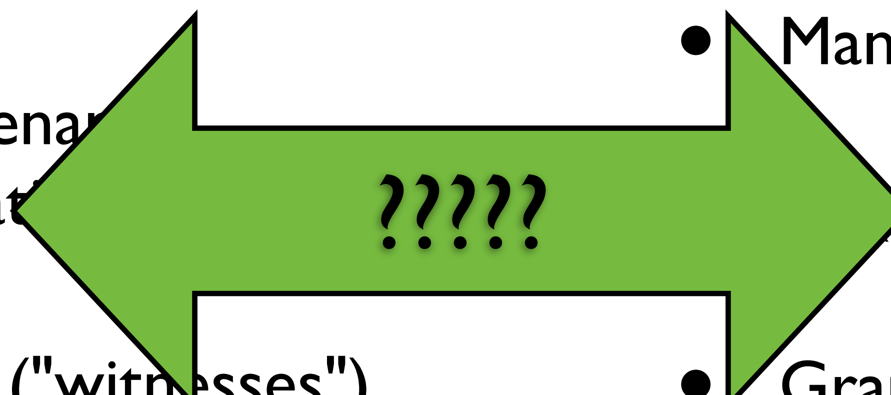
TaPP 2010

Provenance in ...

- Databases
  - Mainly for (nested) relational model
  - Where-provenance ("source location")
  - Lineage, why ("witnesses")
  - How/semiring model
  - Relatively formal
- Workflows
  - Many different systems
  - Many different models
    - (converging on OPM?)
  - Graphs/DAGs
  - Relatively informal

Provenance in ...

- Databases
  - Mainly for (nested) relational model
  - Where-provena
    ("source locat
  - Lineage, why ("witnesses")
  - How/semiring model
  - Relatively formal



- Workflows
  - Many different systems
  - Many different models
    - converging on OPM?)
  - Graphs/DAGs
  - Relatively informal

This talk

- Relate database & workflow "styles"
- Develop a common graph formalism
- Need a common, expressive language that
  - supports many database queries
  - describes some (simple) workflows

Previous work

- Dataflow calculus (DFL), based on nested relational calculus (NRC)
  - Provenance "run" model by Kwasnikowska & Van den Bussche (DILS 07, IPAW 08)
- "Provenance trace" model for NRC
  - by (Acar, Ahmed & C. '08)
- Open Provenance Model (bipartite graphs)
  - (Moreau et al. 2008-9), used in many WF systems

NRC/DFL background

- A very simple, functional language:
  - basic functions +, *,... & constants `0,1,2,3...`
  - variables `x,y,z`
  - pair/record types `(A:e,...,B:e)`, π_A`(e)`
  - **collection (set) types**
    - `{e,...}`   `e ∪ e`   `{e | x in e'}`   `∪e`

An example

An example

- Suppose R = {(1,2,3), (4,5,6), (9,8,7)}

An example

- Suppose R = {(1,2,3), (4,5,6), (9,8,7)}

```
sum { x * y | (x,y,z) in R, x < y}
```

An example

- Suppose R = {(1,2,3), (4,5,6), (9,8,7)}

```
  sum { x * y | (x,y,z) in R, x < y}

= sum { x * y | (x,y,z) in {(1,2,3), (4,5,6)}}
```

An example

- Suppose R = {(1,2,3), (4,5,6), (9,8,7)}

```
  sum { x * y | (x,y,z) in R, x < y}

= sum { x * y | (x,y,z) in {(1,2,3), (4,5,6)}}

= sum {1 * 2, 4 * 5}
```

An example

- Suppose R = {(1,2,3), (4,5,6), (9,8,7)}

```
  sum { x * y | (x,y,z) in R, x < y}

= sum { x * y | (x,y,z) in {(1,2,3), (4,5,6)}}

= sum {1 * 2, 4 * 5}

= sum {2,20}
```

An example

- Suppose R = {(1,2,3), (4,5,6), (9,8,7)}

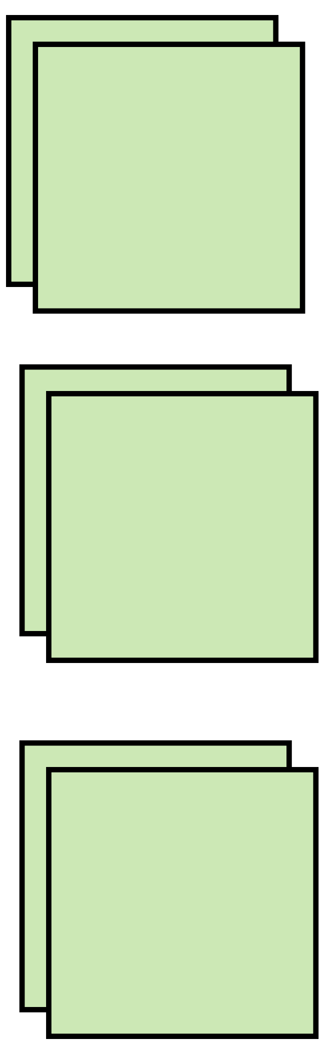
```
  sum { x * y | (x,y,z) in R, x < y}

= sum { x * y | (x,y,z) in {(1,2,3), (4,5,6)}}

= sum {1 * 2, 4 * 5}

= sum {2,20}

= 22
```

Another example

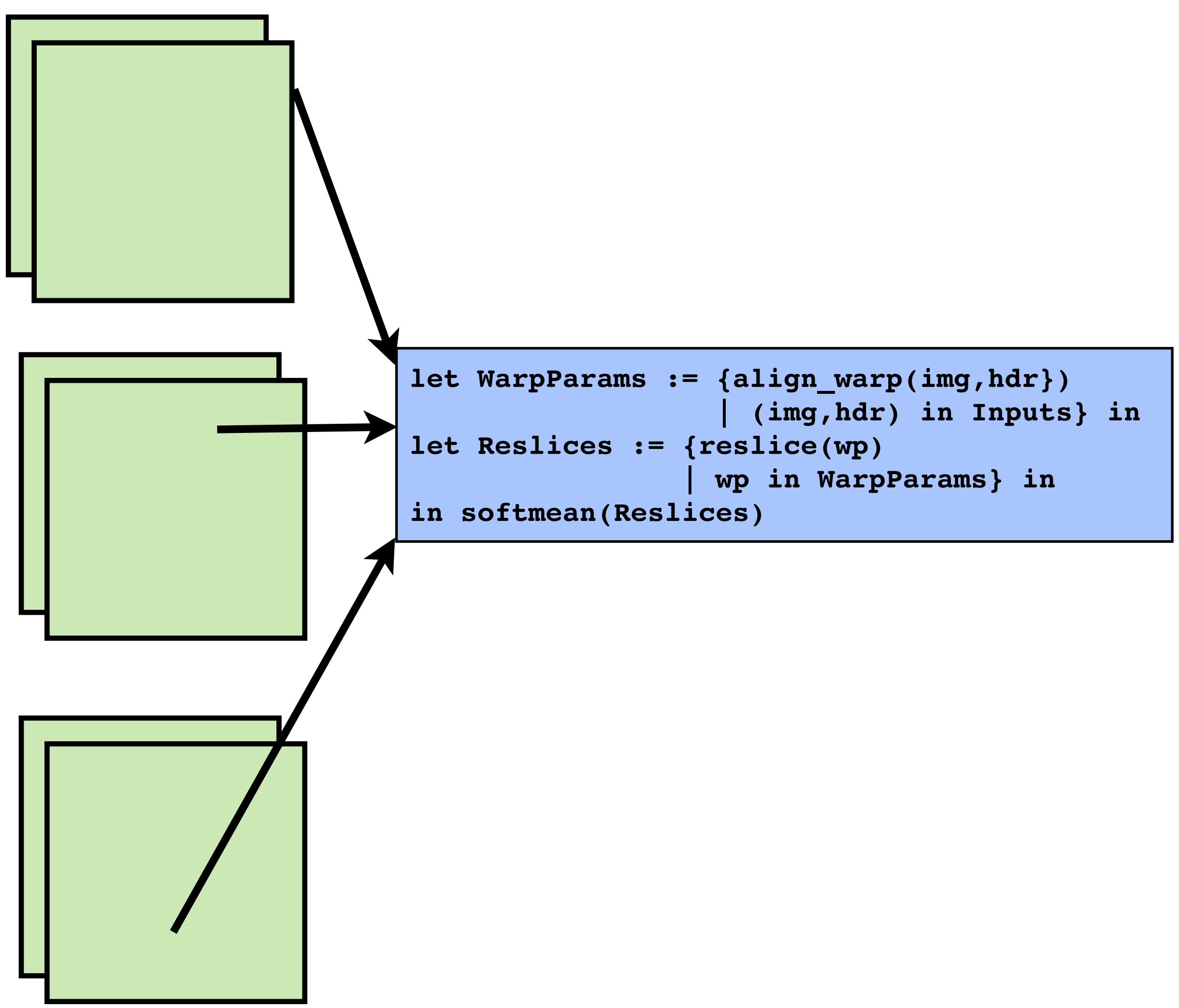
- In DFL, built-in functions / constants can be whole programs & files,
  - as in Provenance Challenge 1 workflow:

```
let WarpParams := {align_warp(img,hdr})
                   | (img,hdr) in Inputs} in
let Reslices := {reslice(wp)
                 | wp in WarpParams} in
softmean(Reslices)
```

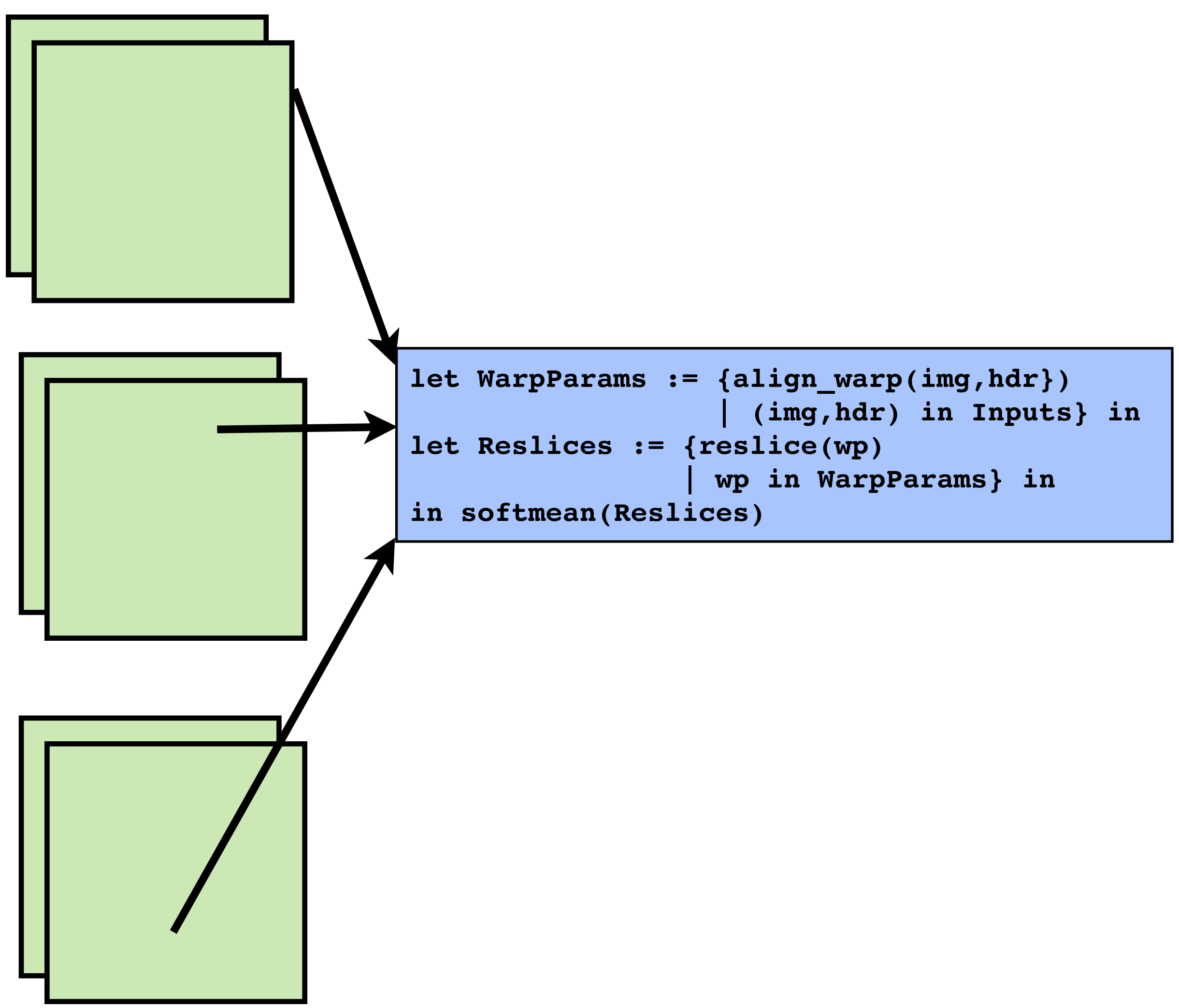
Goal: Define "provenance graphs" for DFL

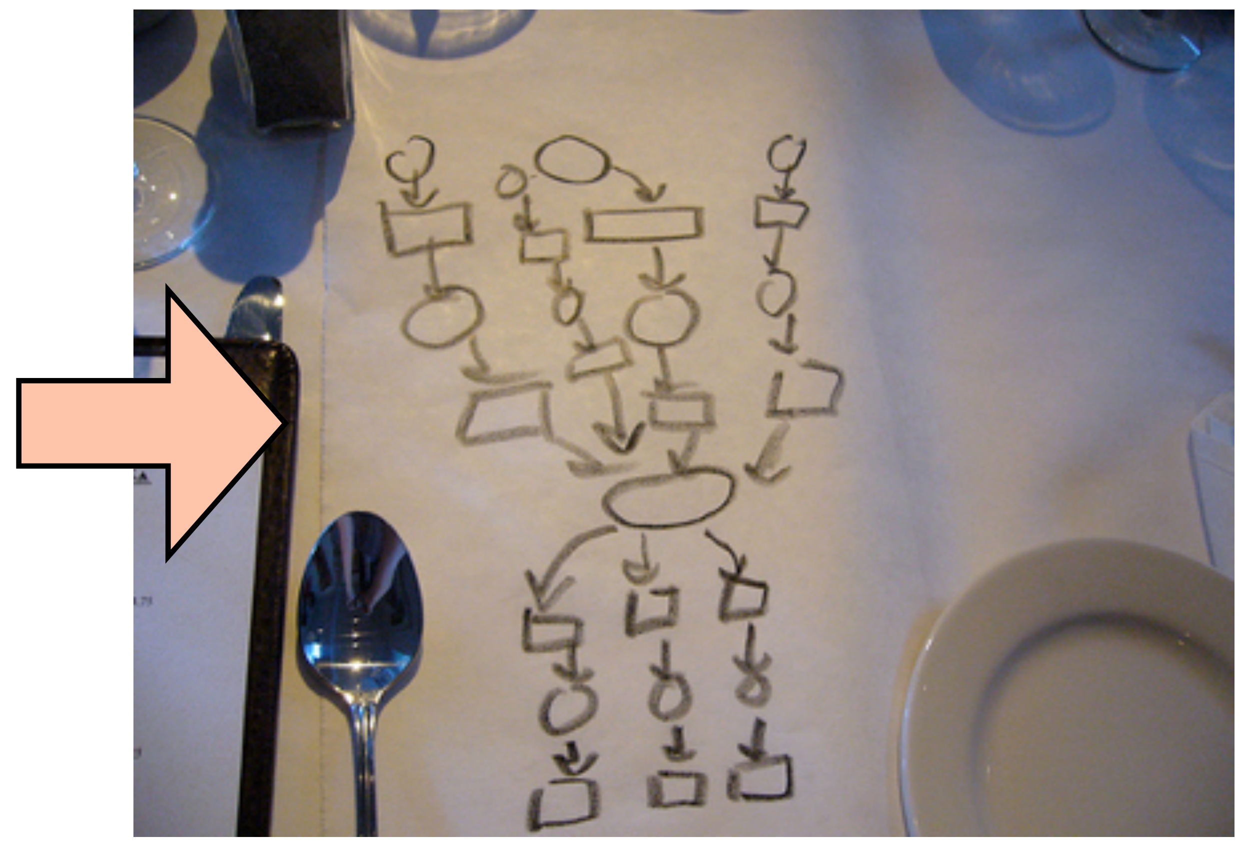


Goal: Define "provenance graphs" for DFL



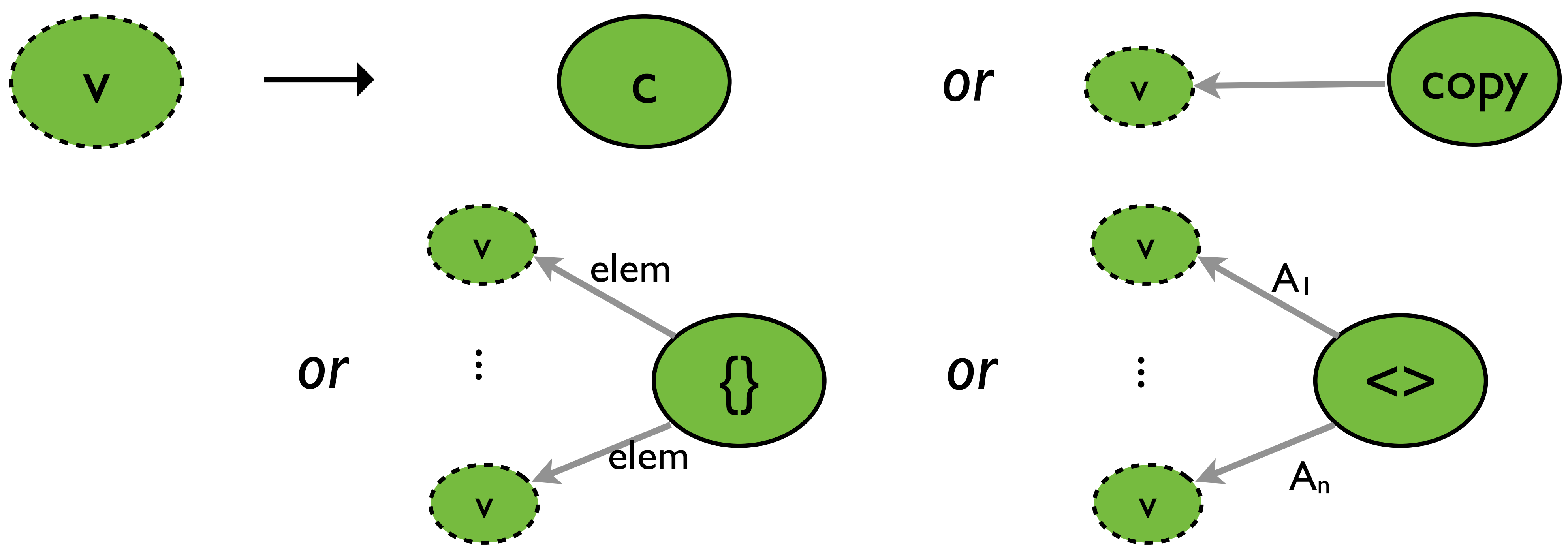
Goal: Define "provenance graphs" for DFL



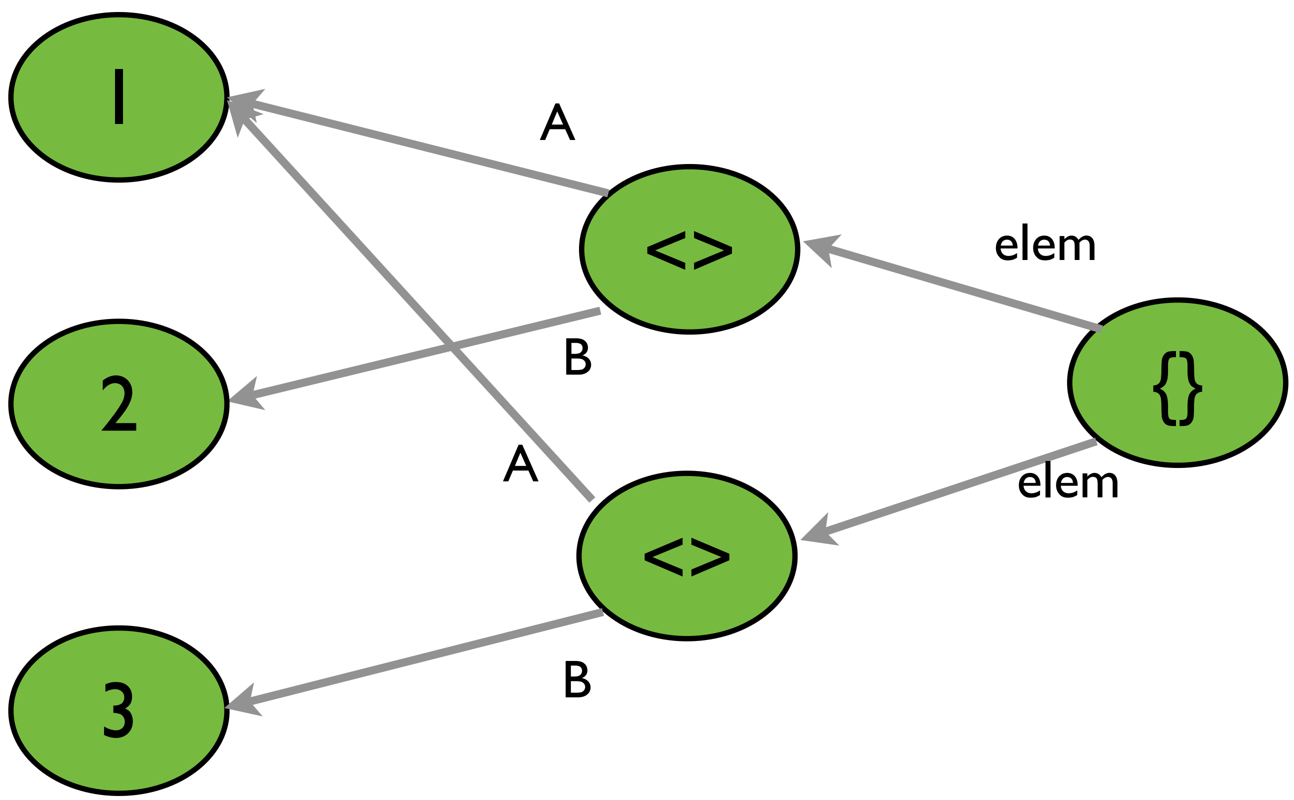


http://www.flickr.com/photos/schneertz/679692806/

First step: values

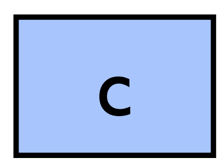


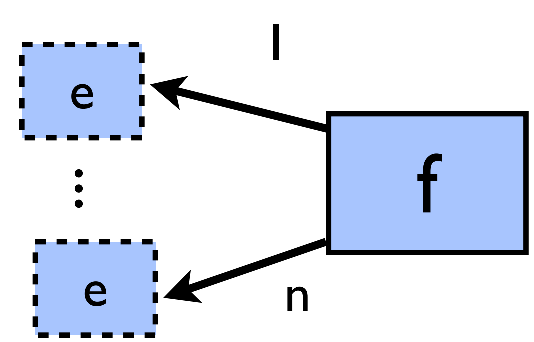
Example value



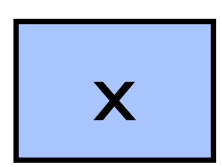
Next step: evaluation nodes ("process")

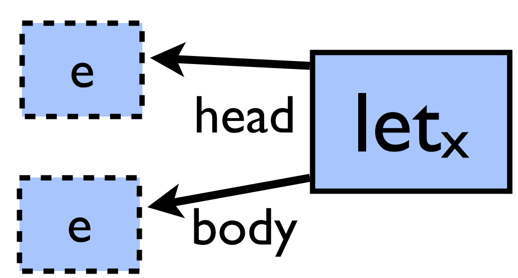
Constants, primitive functions





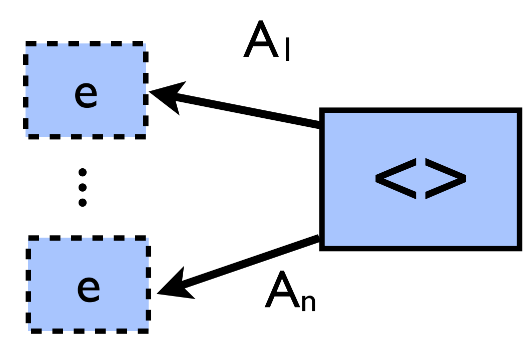
Variables & temporary bindings



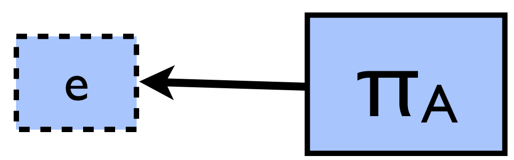


Pairing

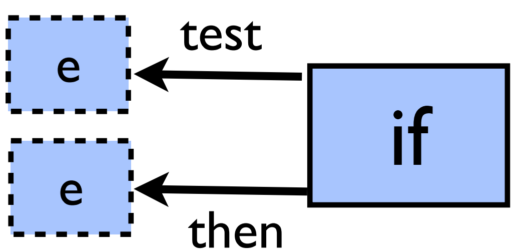
Record building

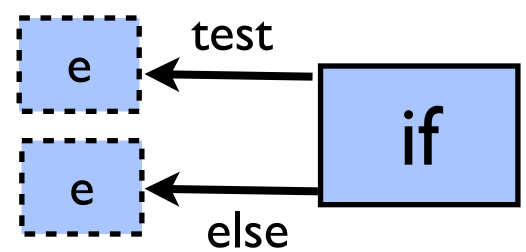


Field lookup



Conditionals

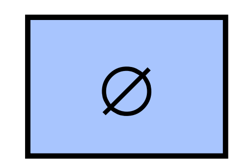




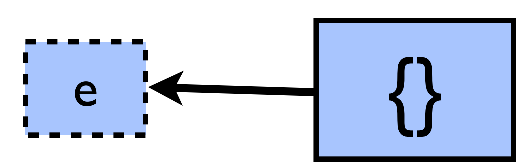
Note: Only taken branch is recorded

Sets: basic operations

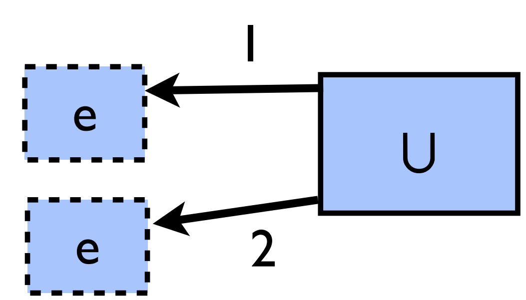
Empty set



Singleton

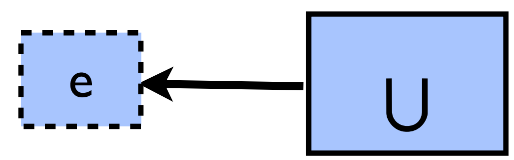


Union

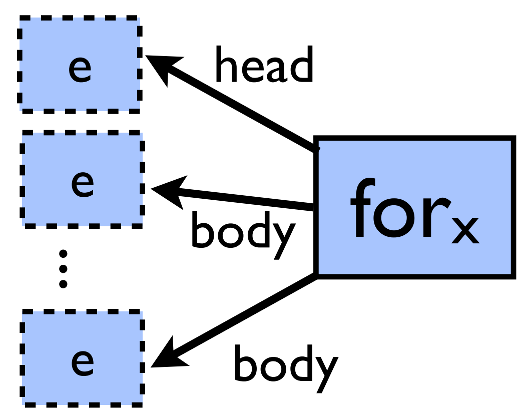


Sets: complex operations

Flattening

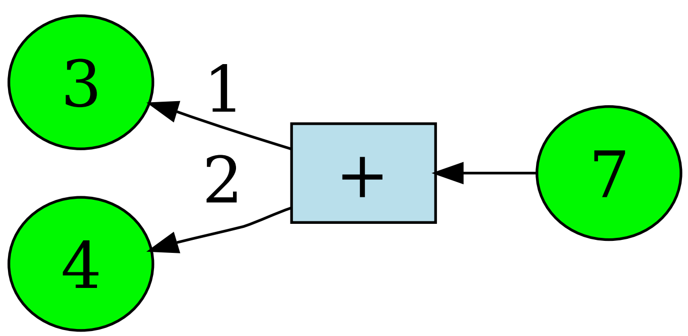


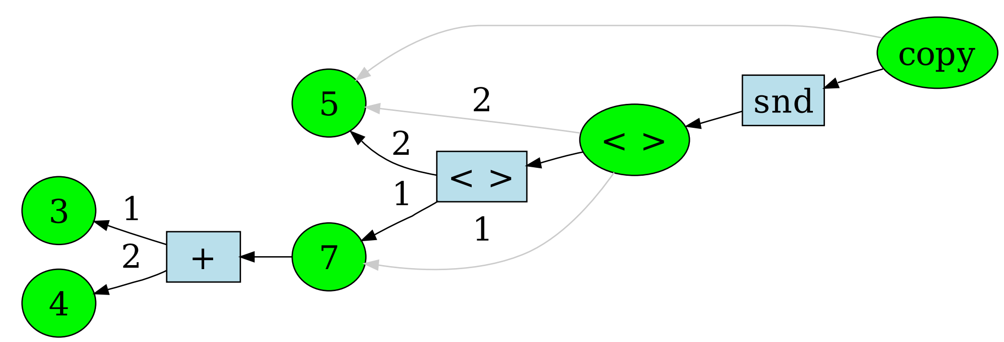
Iteration

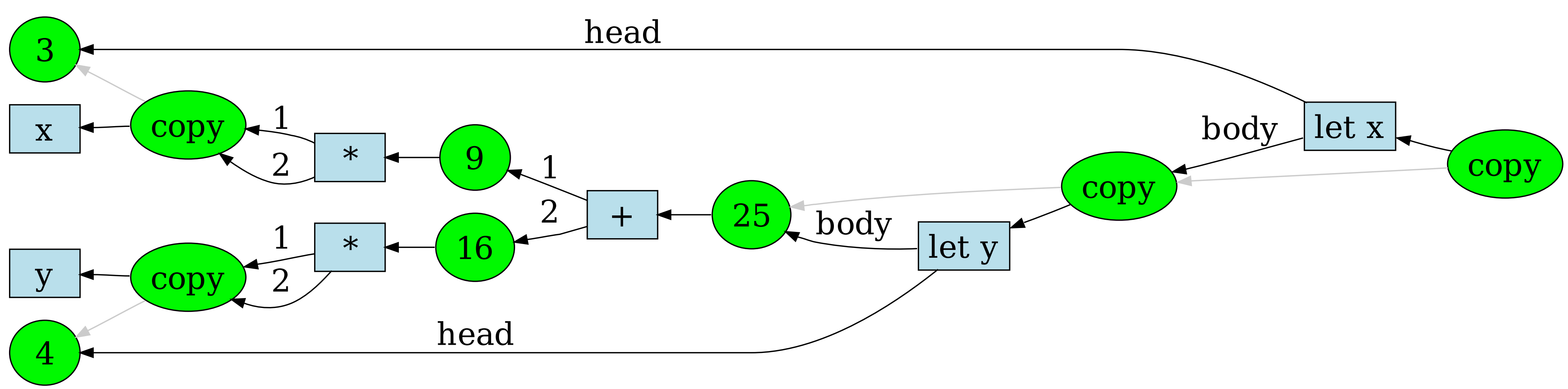


Provenance graphs

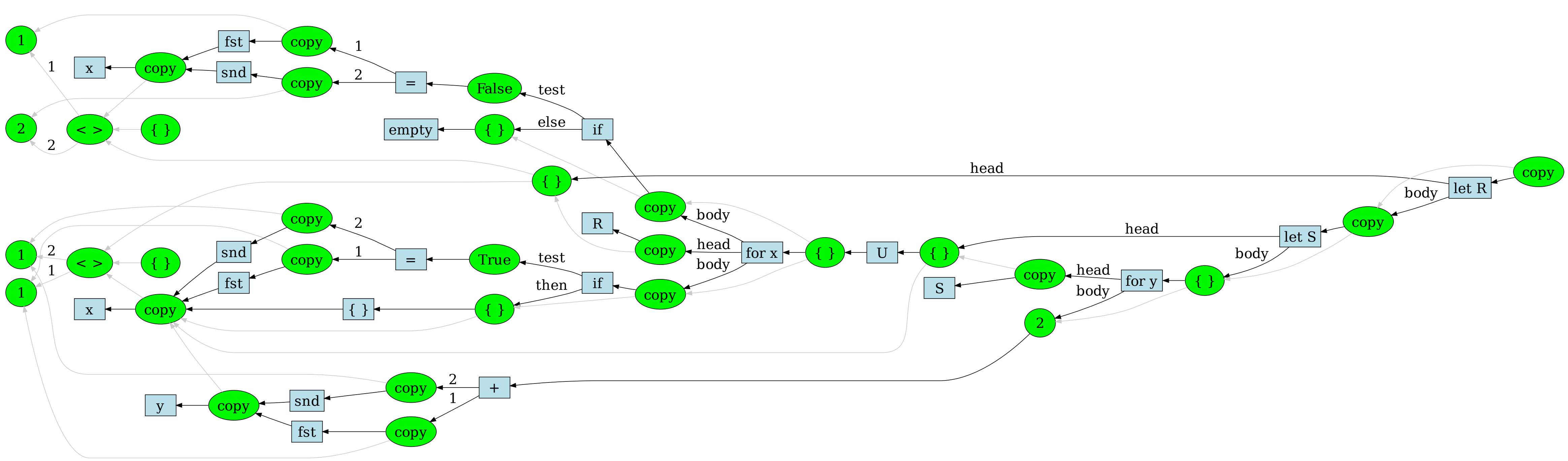
- are graphs with "both value and evaluation structure"



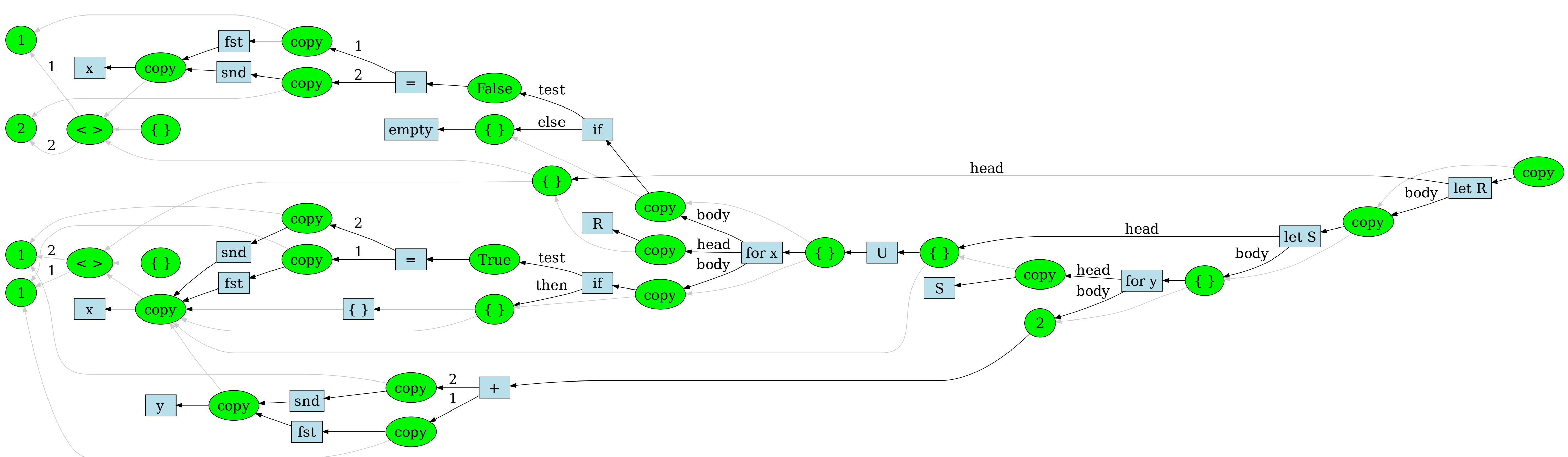




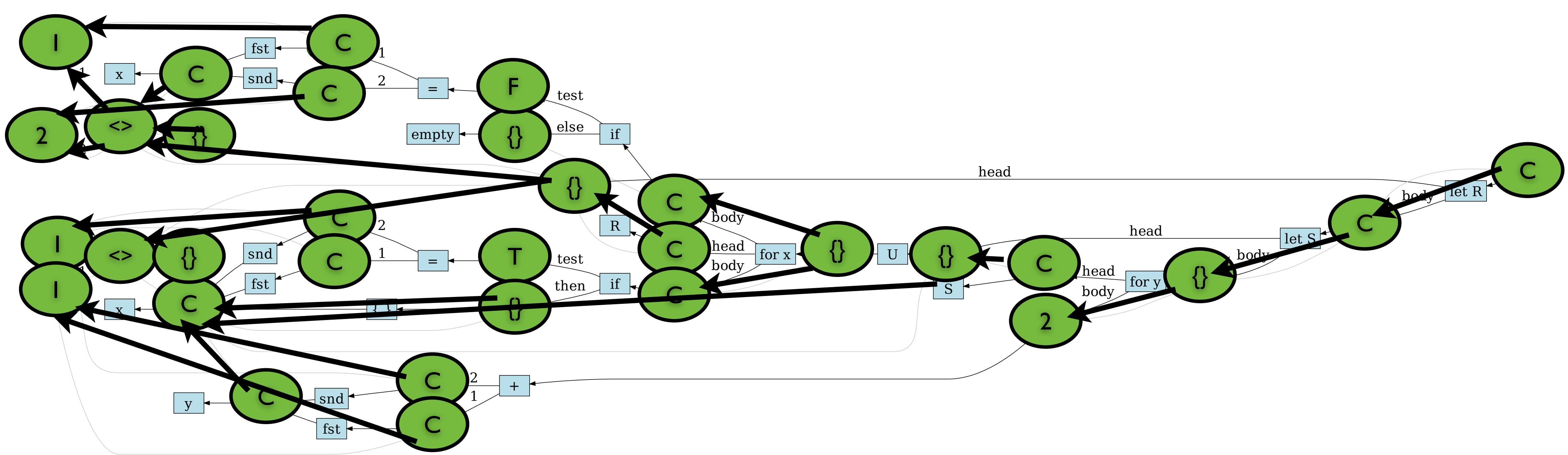
A bigger example



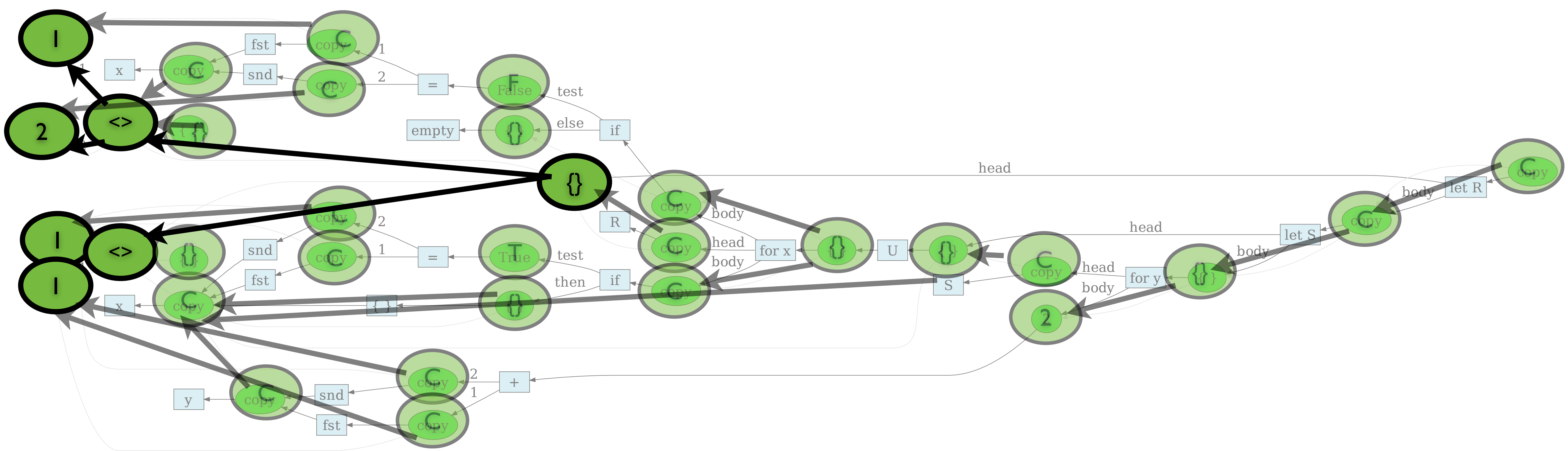
Value structure



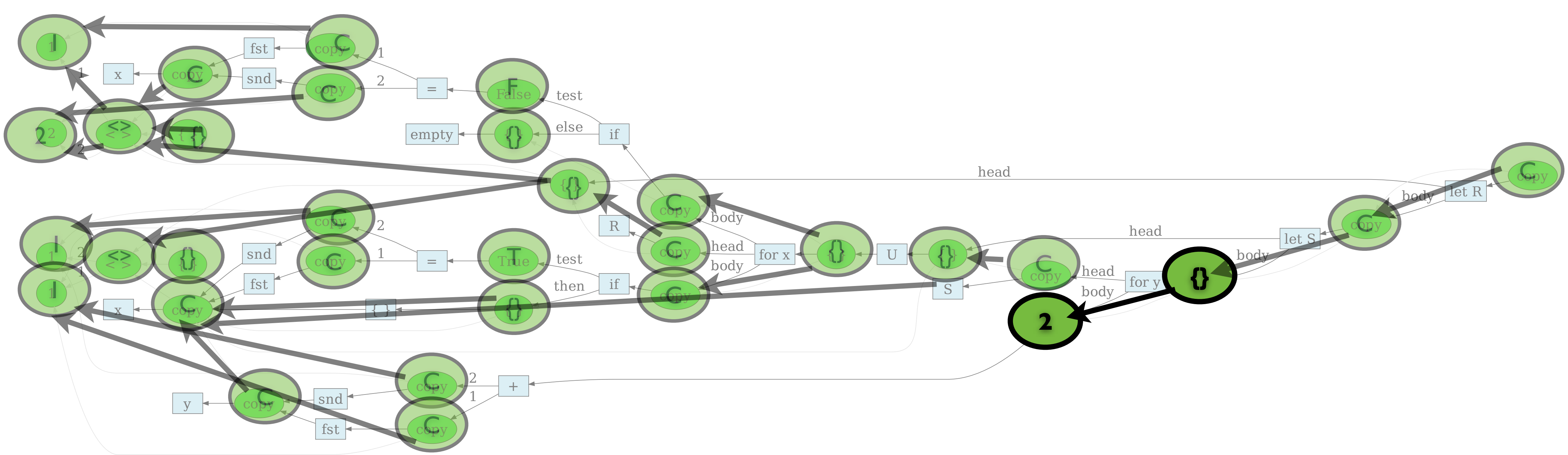
Value structure



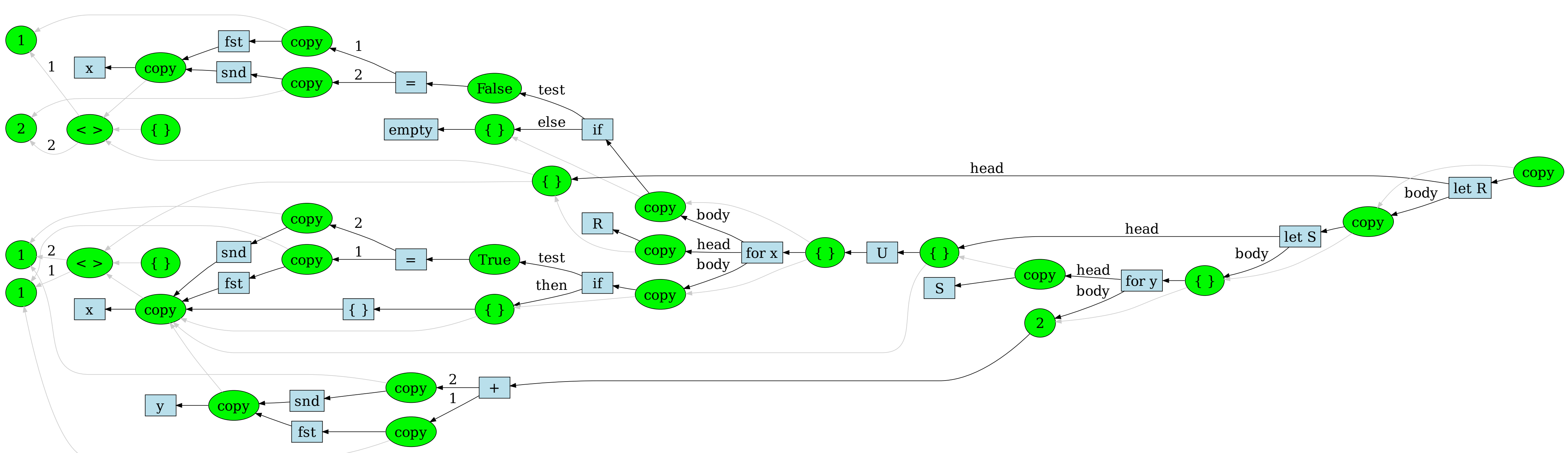
Input values



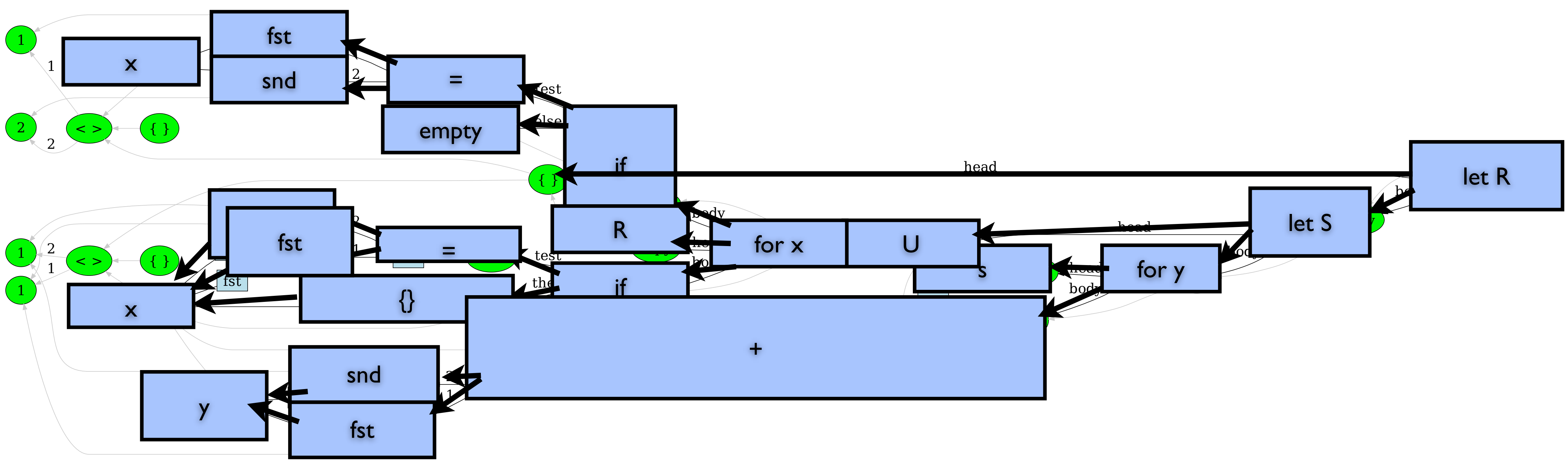
Return value



Expression structure

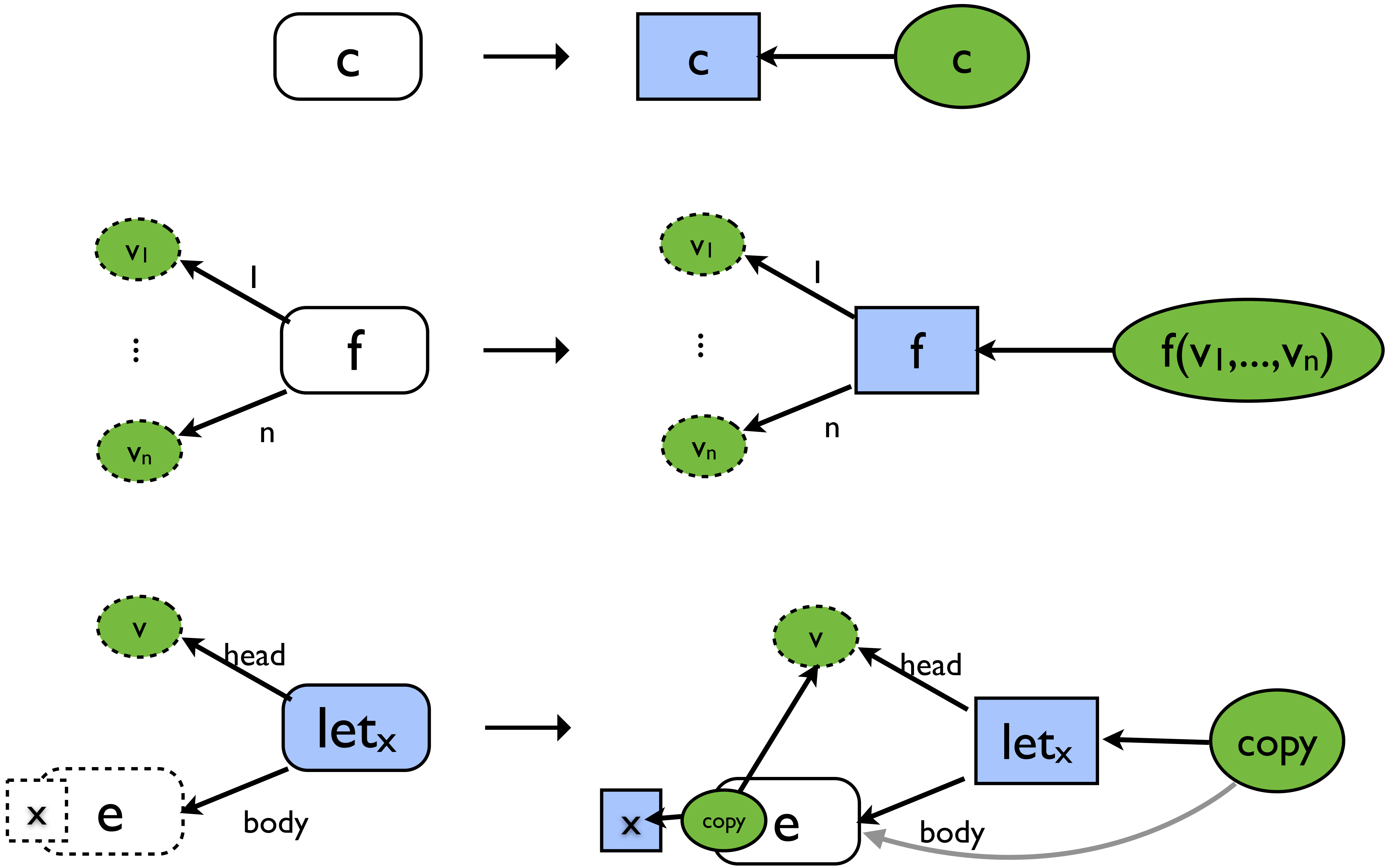


Expression structure

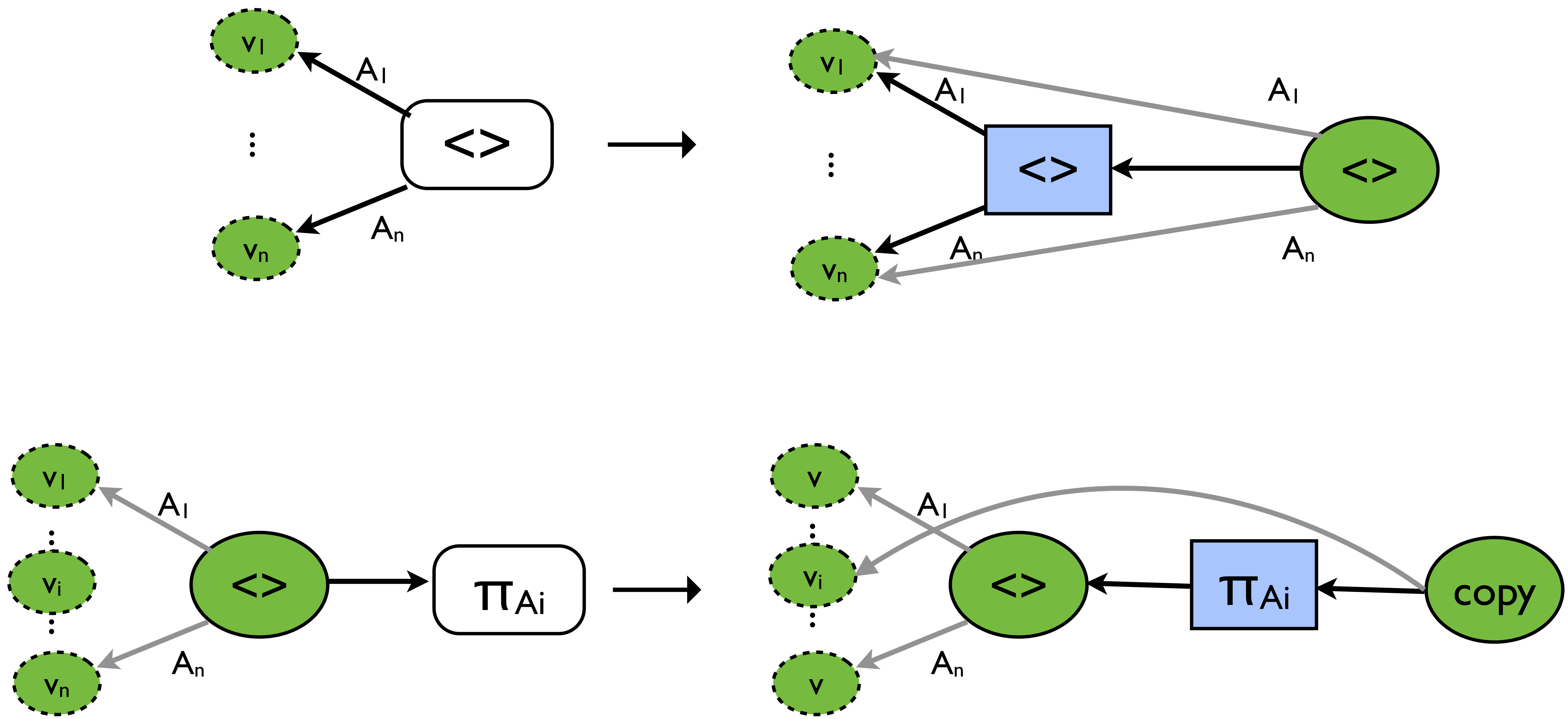


Building provenance graphs

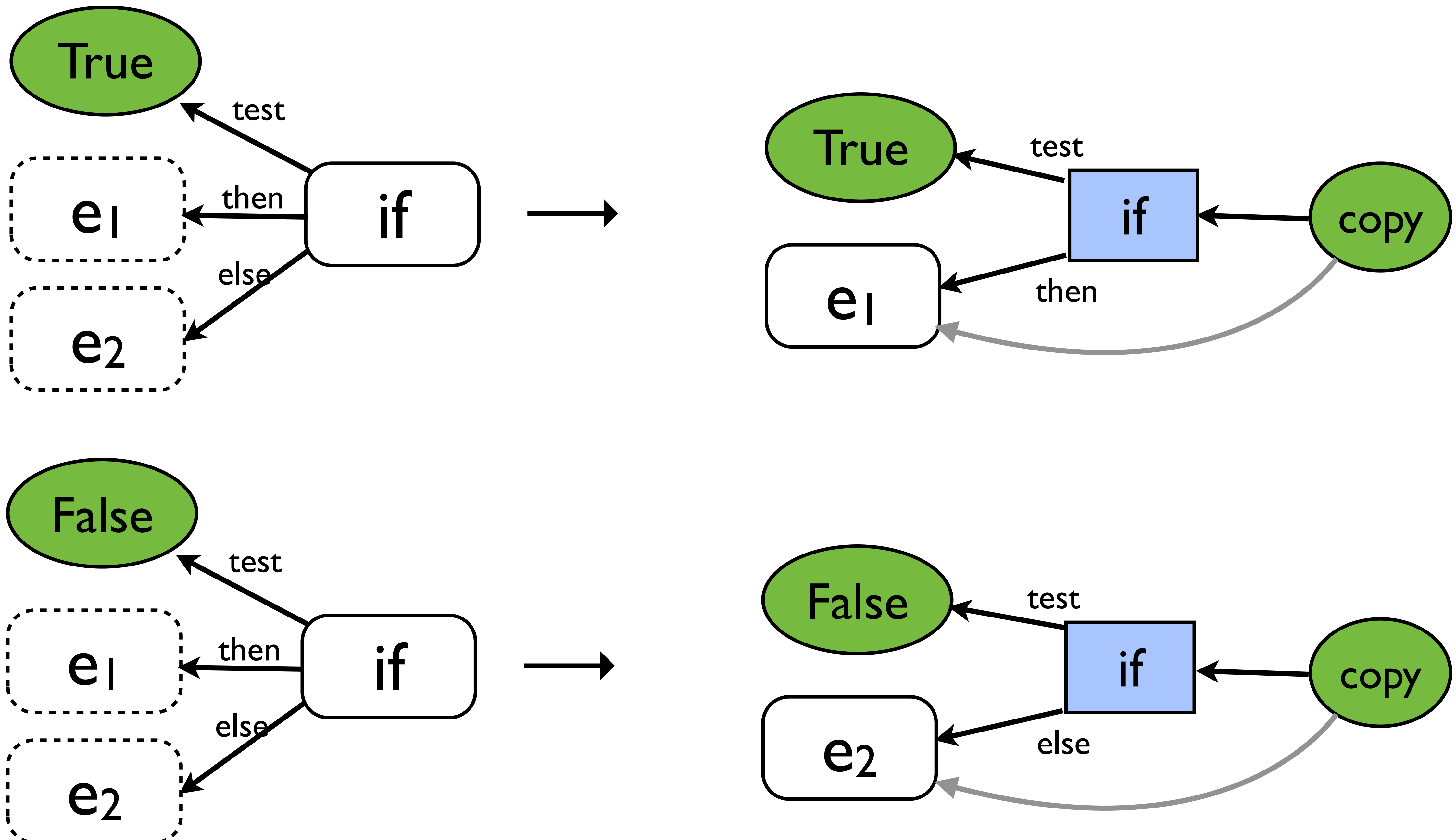
- is complicated
- Here we'll use high-level "graph rewrite rule" formalism
- Mostly because it is nicer to look at than formal version

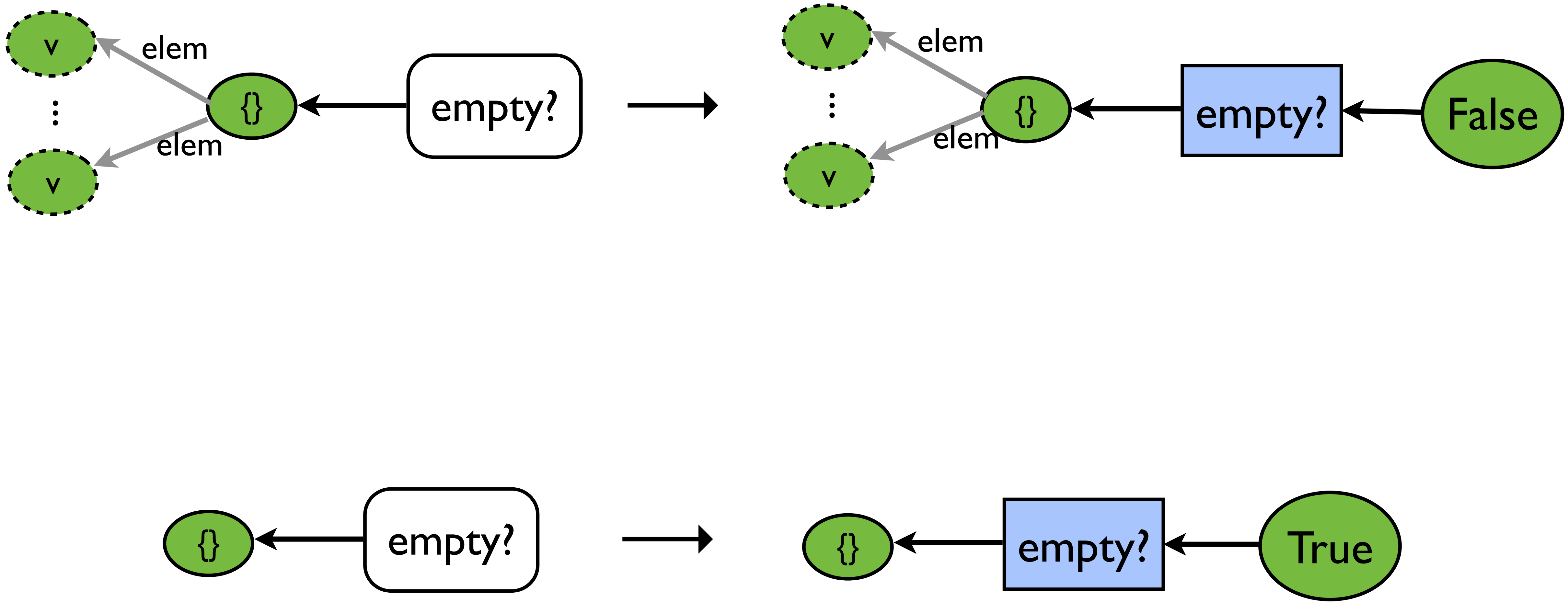


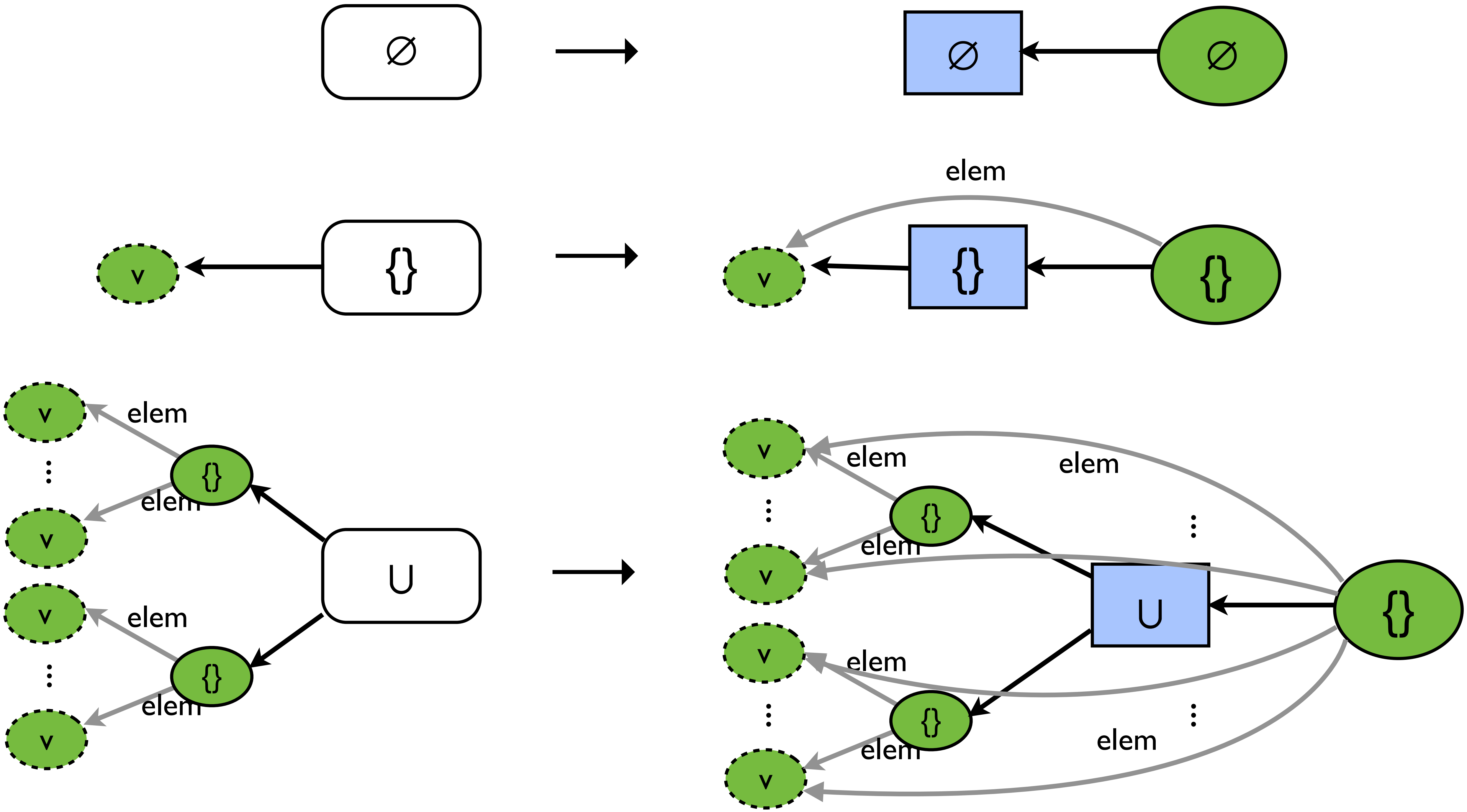
c
c
c
v1
1
f
vn
n
v1
1
f
f(v1,...,vn)
vn
n
v
head
letx
x
e
body
v
head
letx
copy
x
copy
e
body



v_1
A_1
<>
A_n
v_n
v_1
A_1
<>
A_1
<>
A_n
v_n
A_n
v_1
A_1
v_i
<>
π_{A_i}
v_n
A_n
v
A_1
v_i
<>
π_{A_i}
copy
v
A_n

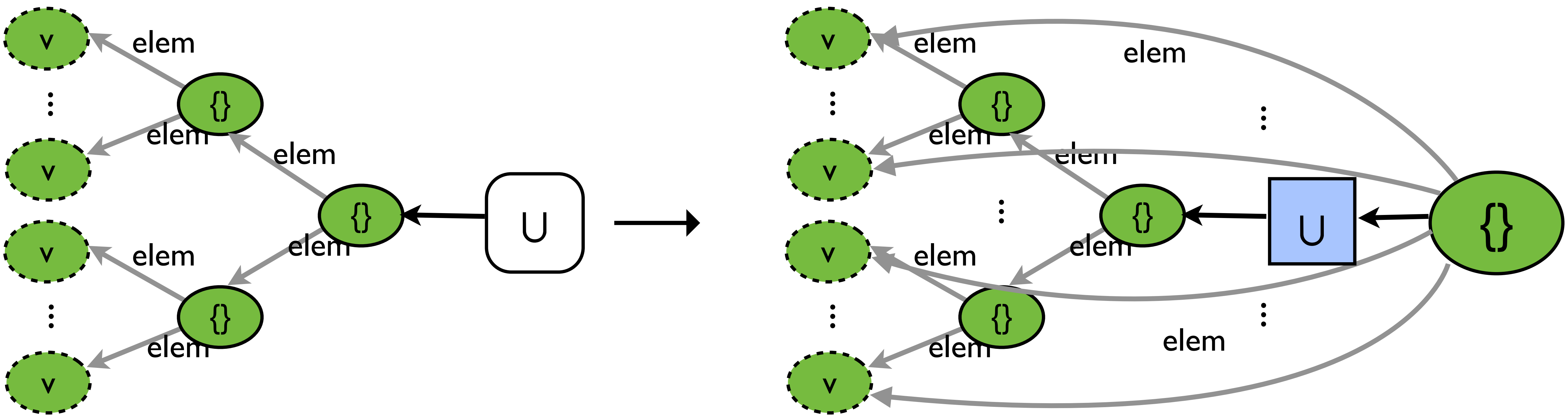

True
test
then
else
if
True
test
if
copy
then
False
test
then
else
if
False
test
if
copy
else


v
elem
{}
elem
v
empty?
v
elem
{}
elem
v
empty?
False
{}
empty?
{}
empty?
True

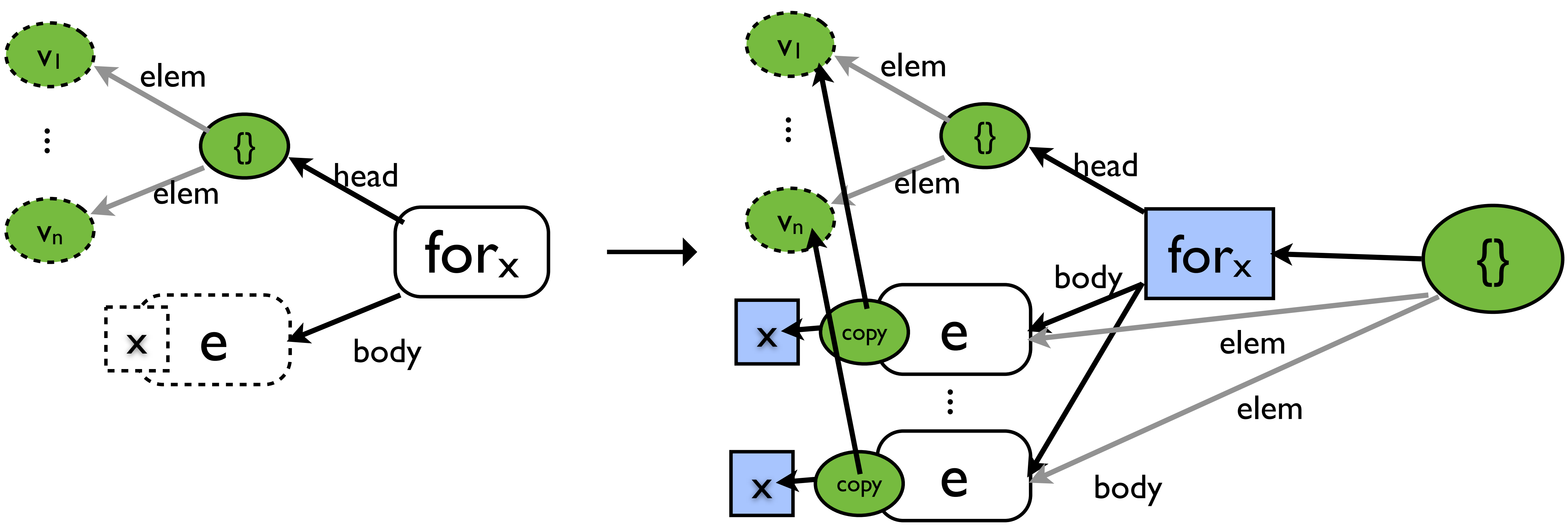


∅
∅
∅
elem
v
{}
v
{}
{}
elem
elem
elem
elem
elem
∪
elem
elem
elem
elem
elem
elem
elem
∪

OK, take a deep breath!

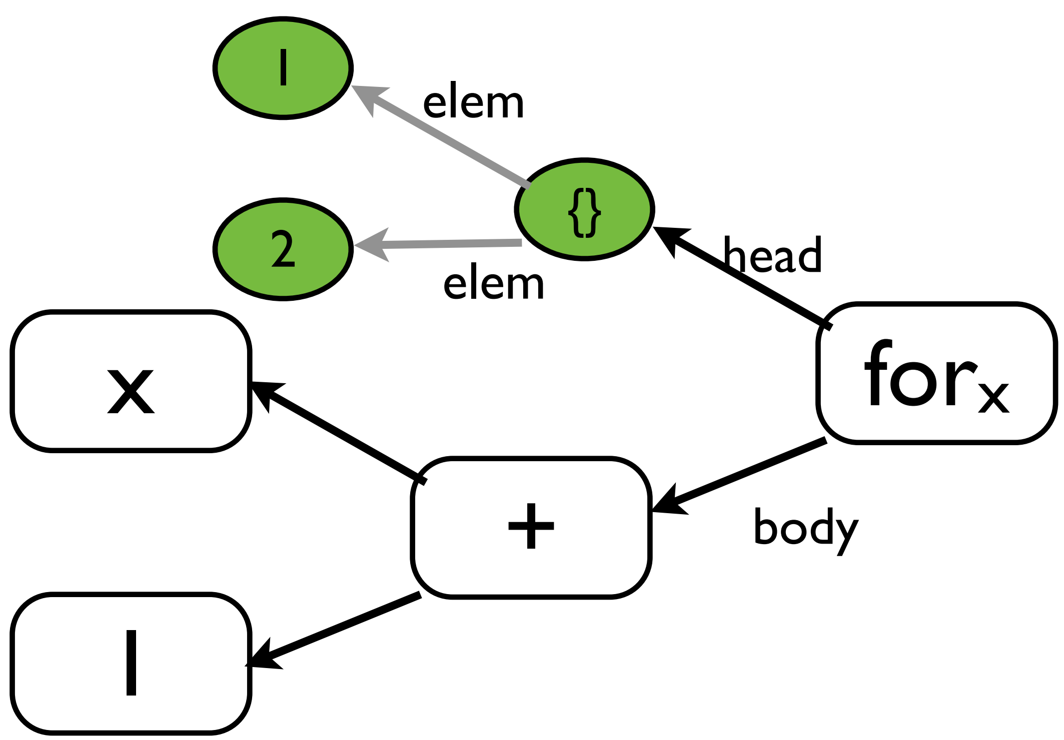


v
elem
{}
U
elem
{}
U

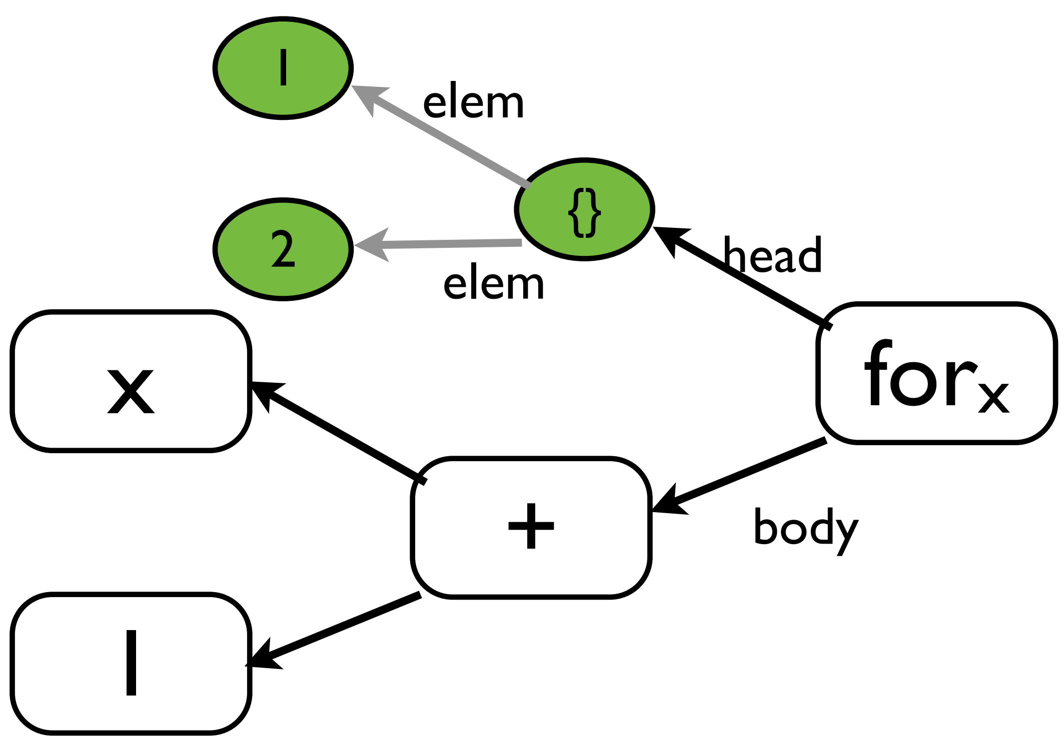


v1
vn
elem
{}
head
forx
x
e
body
copy
elem
body

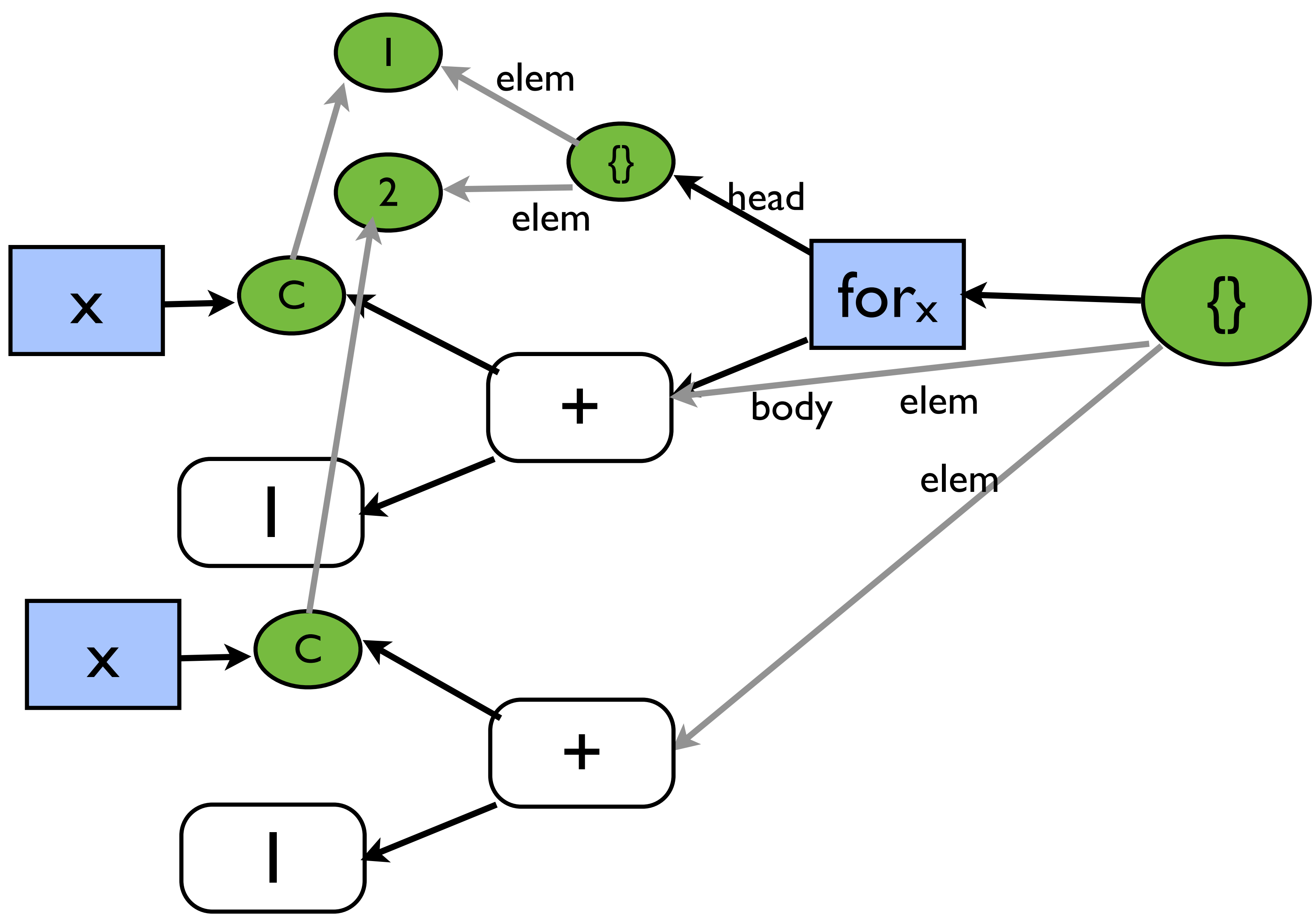
An example



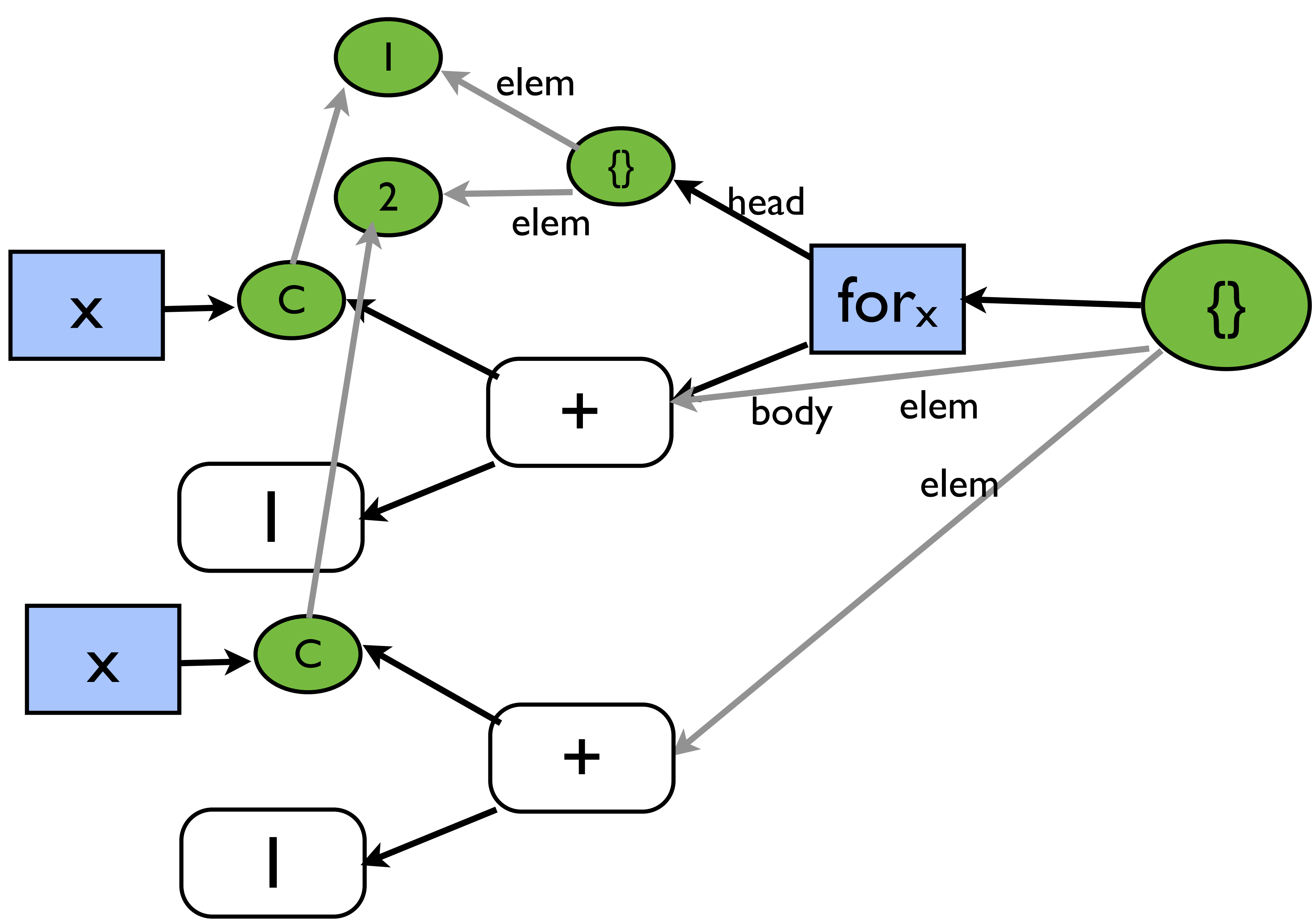
An example



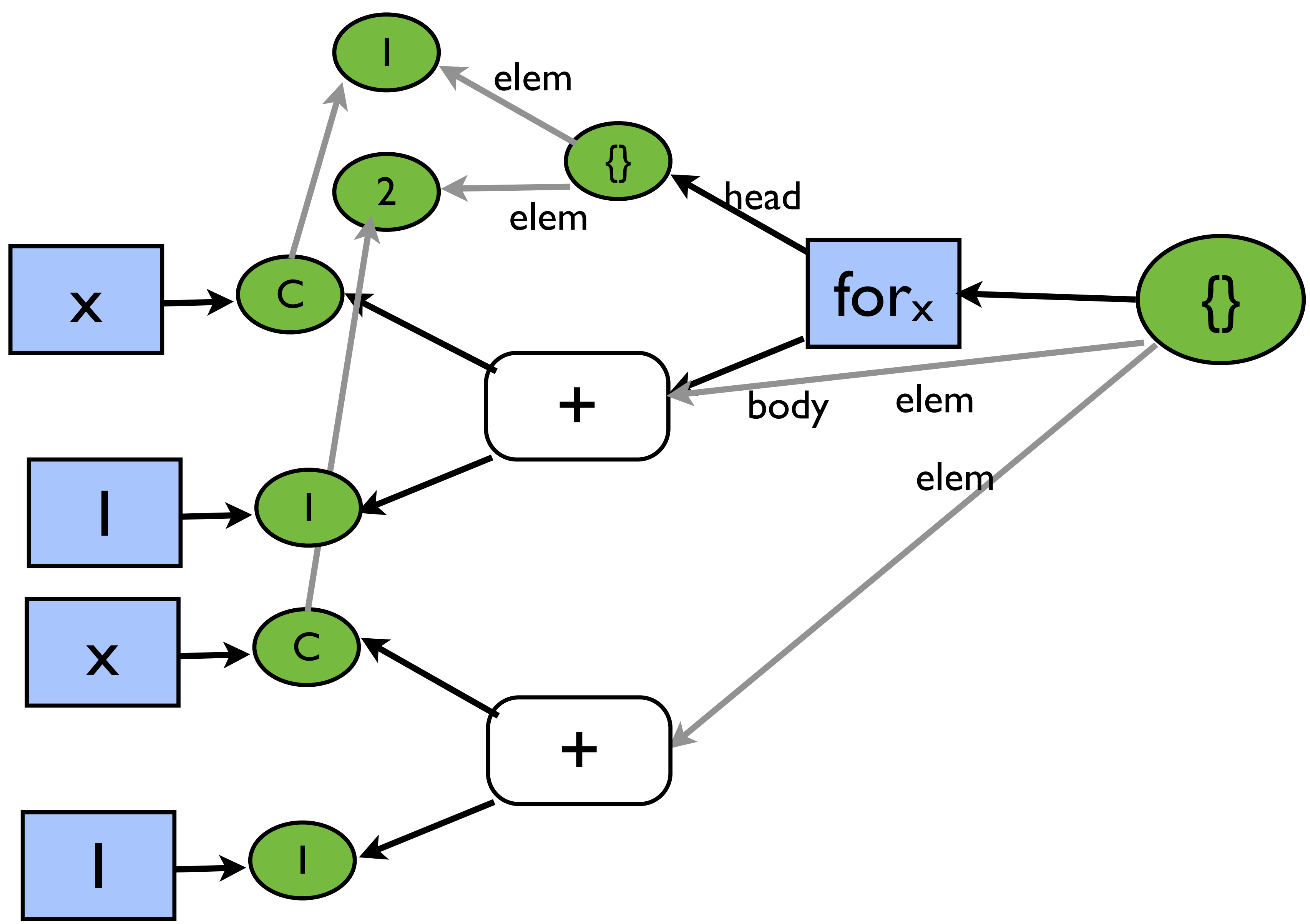
An example



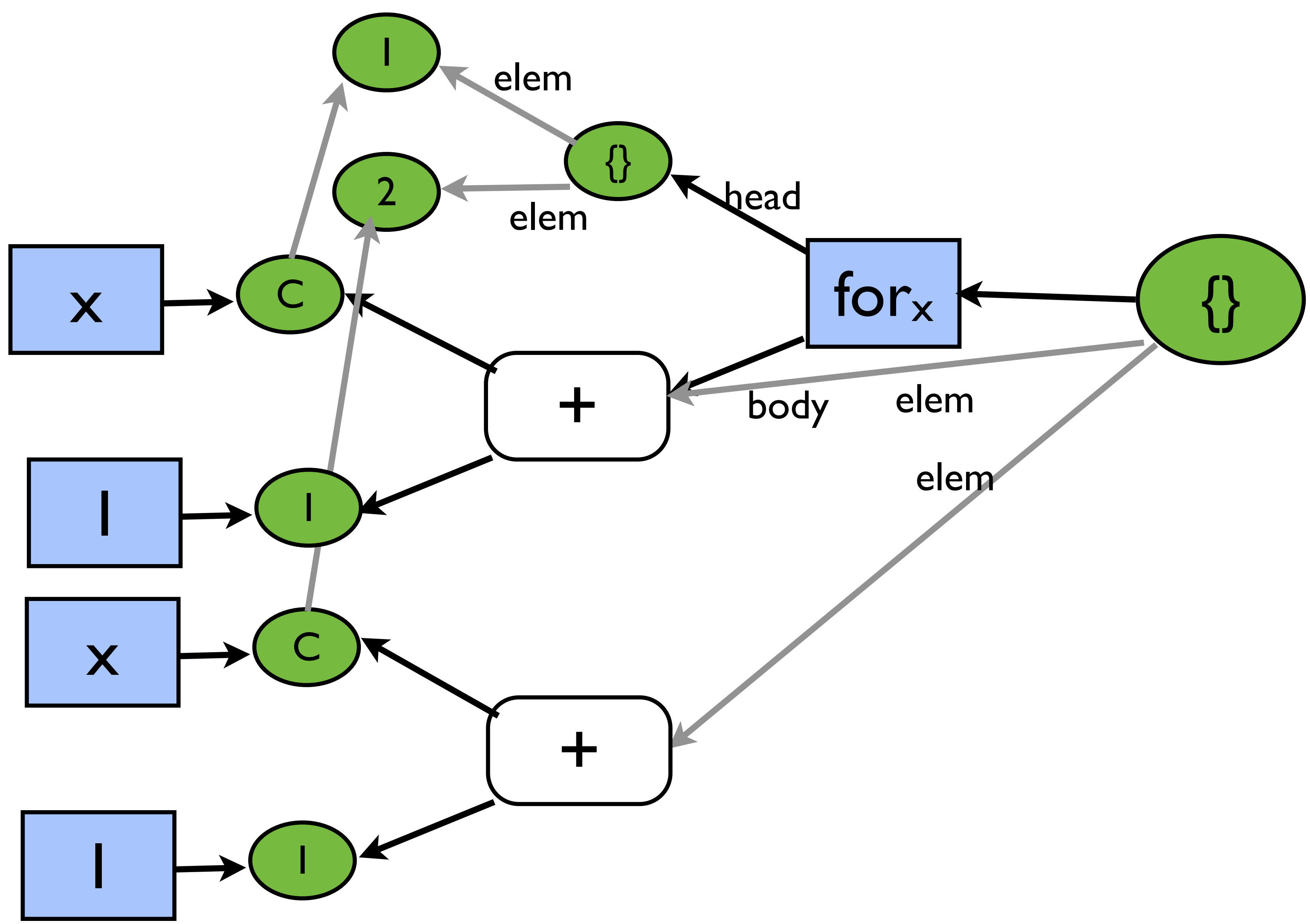
An example



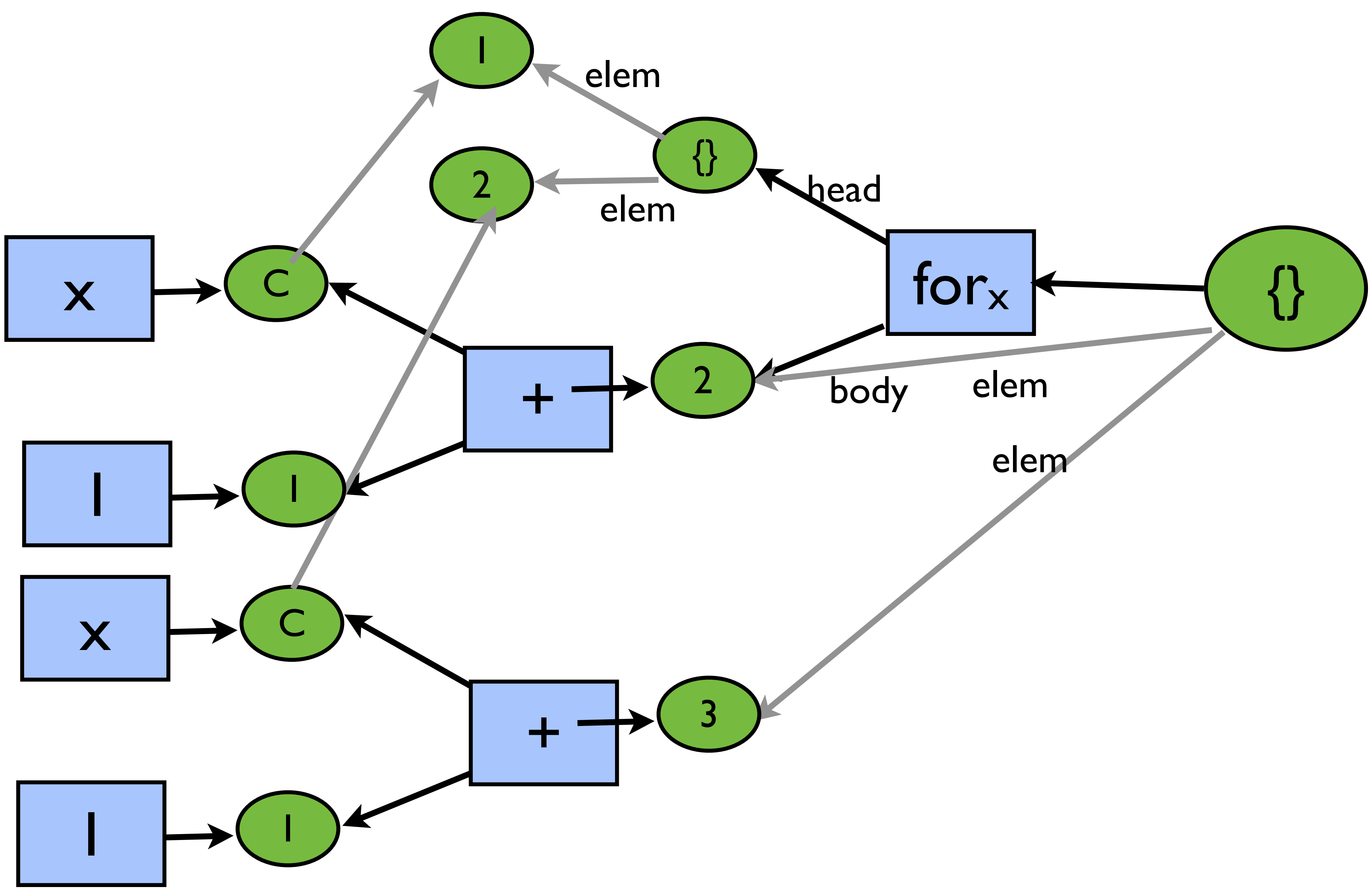
An example



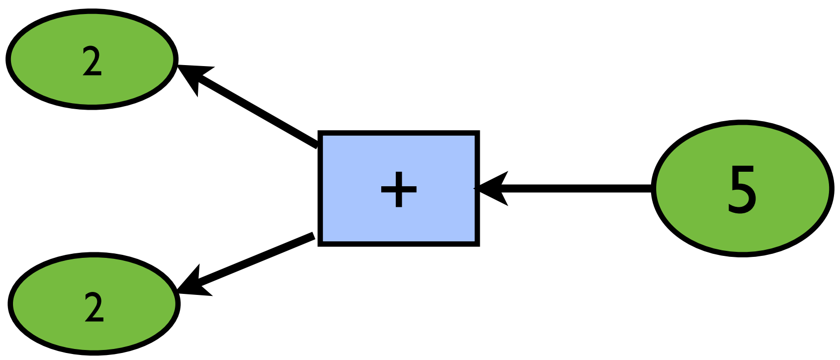
An example



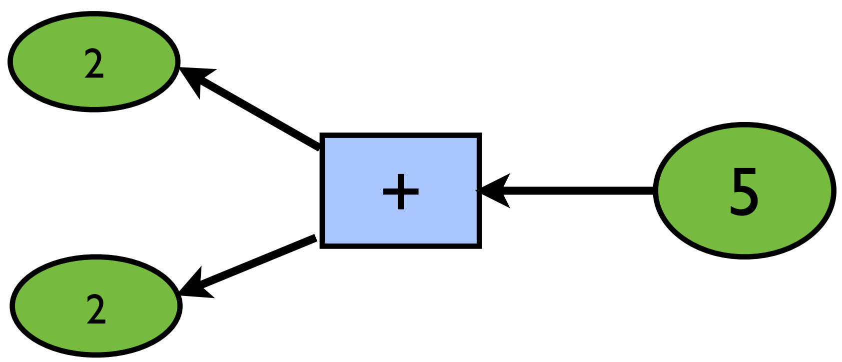
An example

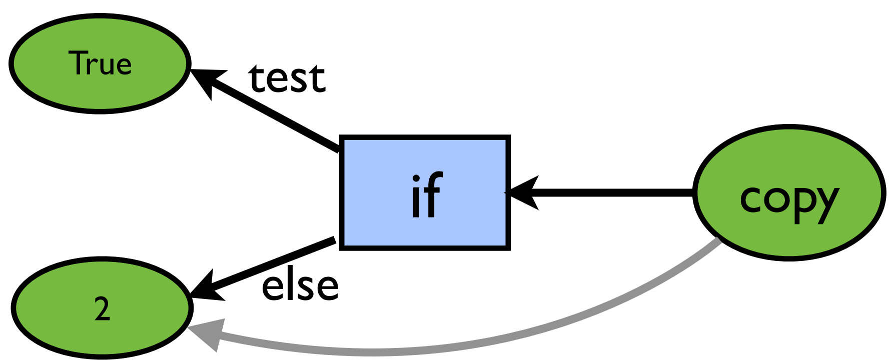


Graphs can "lie" (inconsistency)

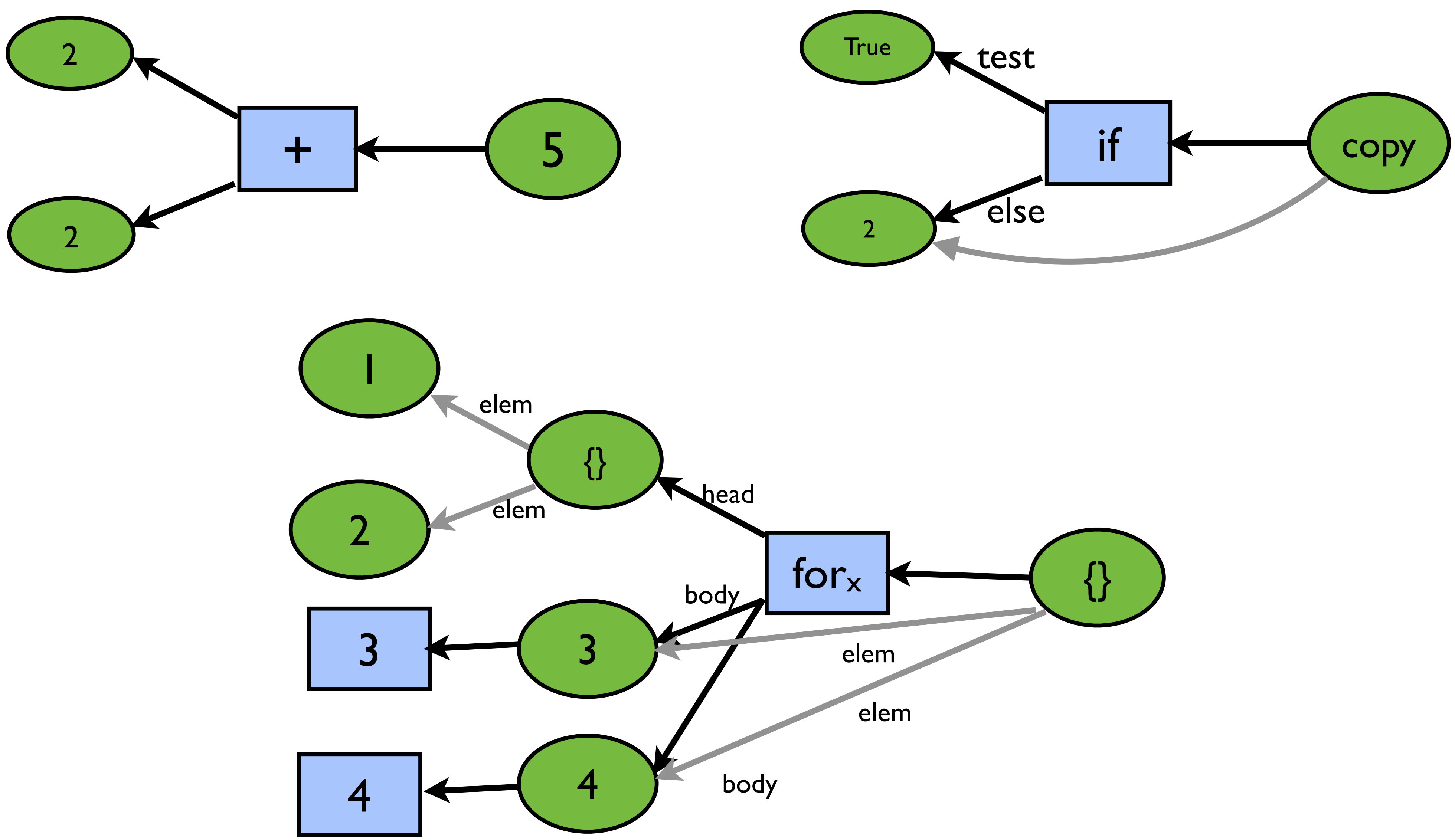


Graphs can "lie" (inconsistency)

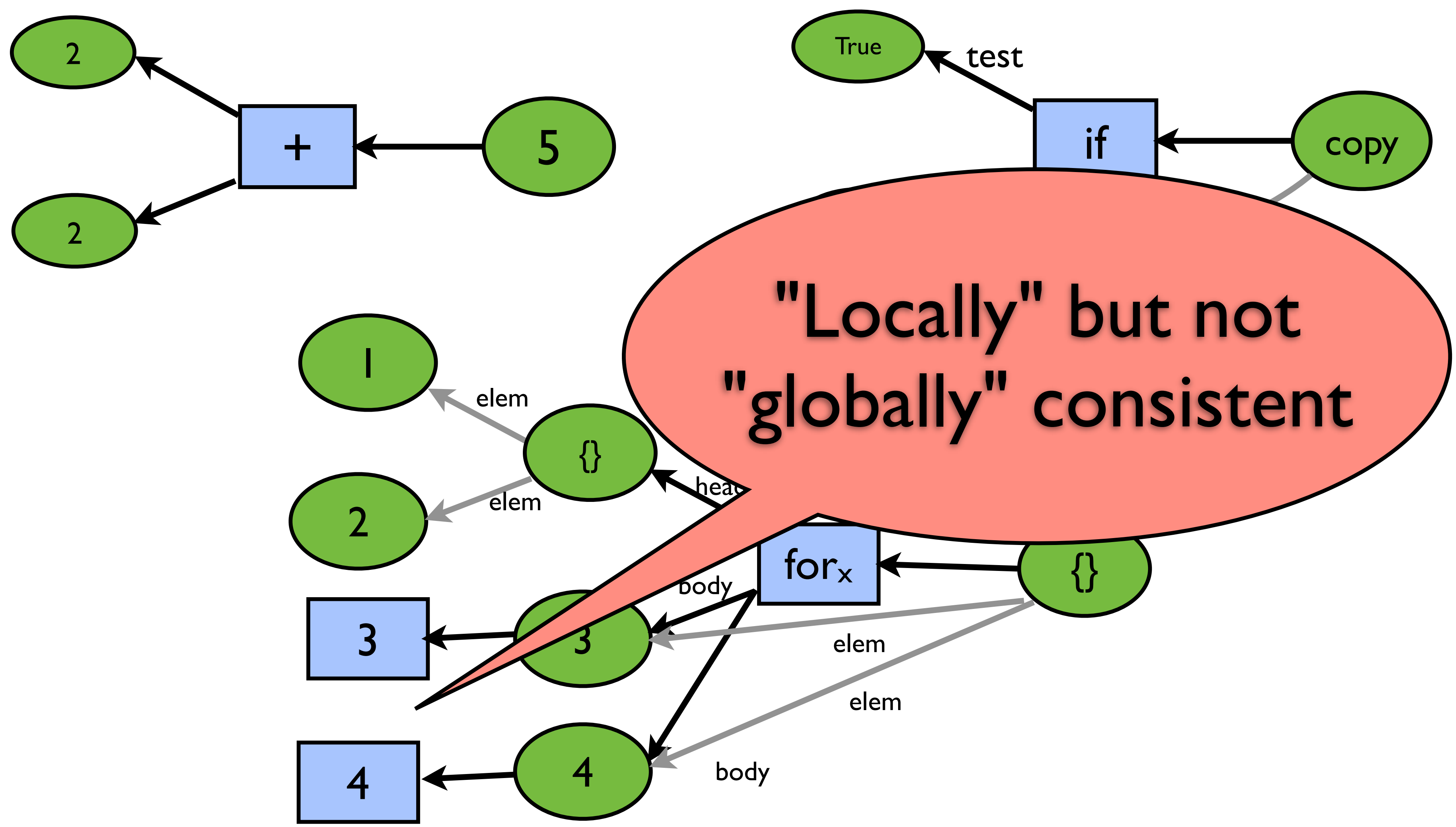




Graphs can "lie" (inconsistency)



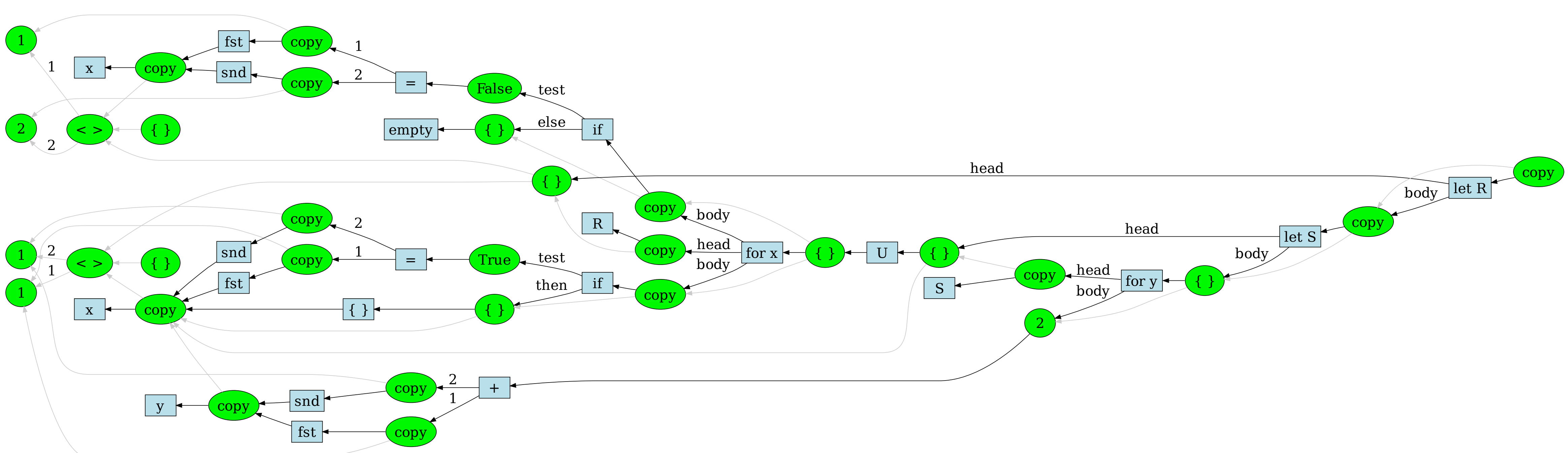
Graphs can "lie" (inconsistency)



Graph queries

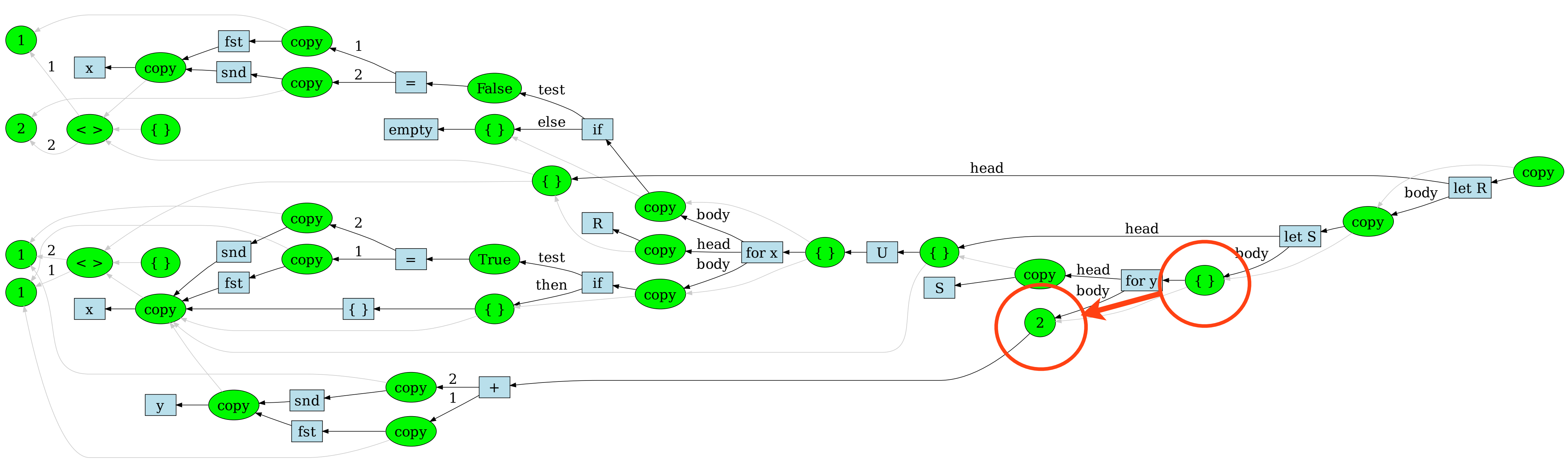
- Many possible approaches
- In paper: some Datalog
  - Maybe overkill, seems fragile
- In code: some "annotation propagation" traversals
  - Seems to handle where, "explanations", "summaries"

Explaining



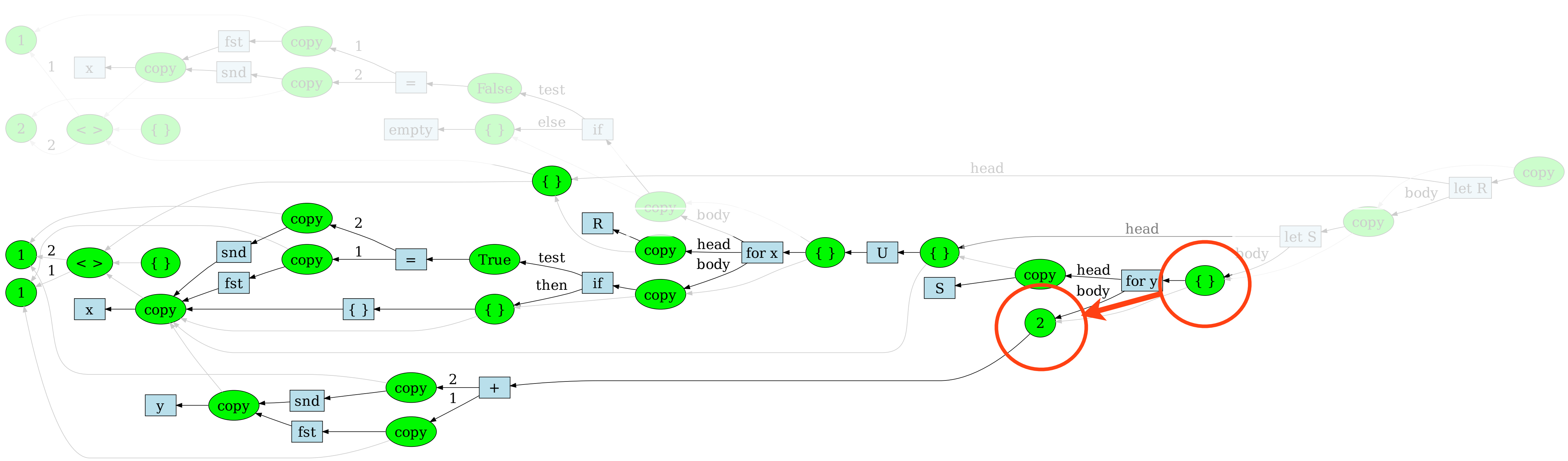
1
x
copy
fst
snd
copy
copy
=
False
test
else
if
empty
{ }
2
< >
{ }
head
{ }
let R
copy
body
copy
let S
head
body
R
copy
for x
{ }
U
{ }
copy
for y
{ }
S
2
True
then
+
y
snd
fst

Explaining



1
fst
copy
1
x
copy
snd
copy
2
=
False
test
2
< >
{ }
empty
{ }
else
if
{ }
head
copy
body
let R
copy
copy
2
R
copy
body
head
for x
{ }
U
{ }
head
let S
body
1
2
< >
{ }
snd
copy
1
=
True
test
1
fst
if
copy
copy
head
for y
{ }
S
body
x
copy
{ }
{ }
then
2
copy
2
+
1
y
copy
snd
fst
copy

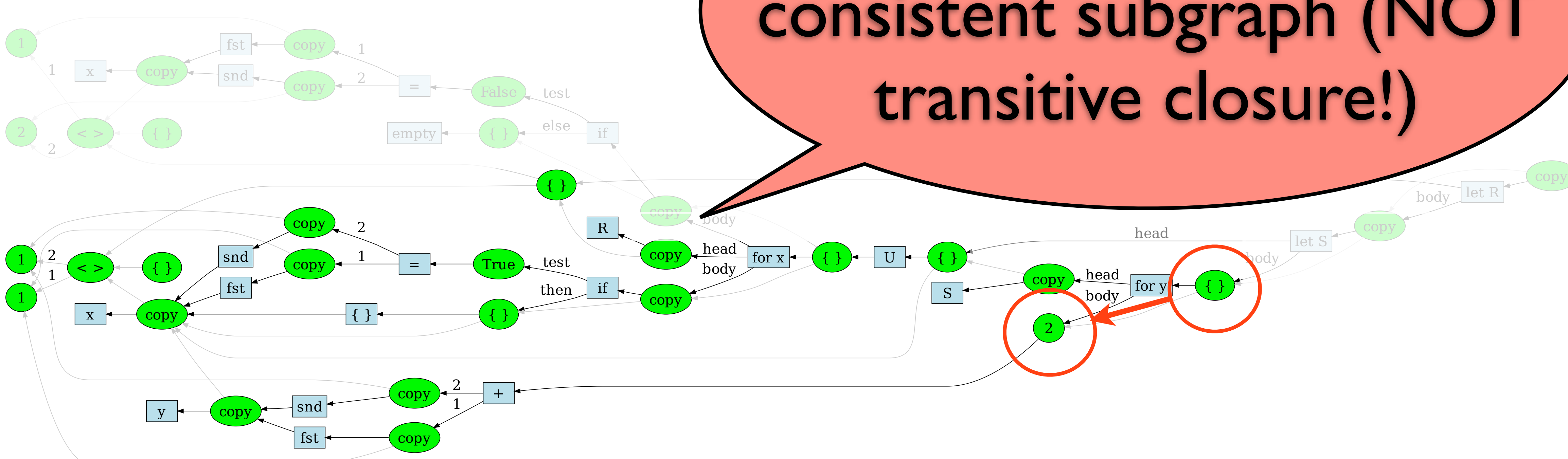
Explaining



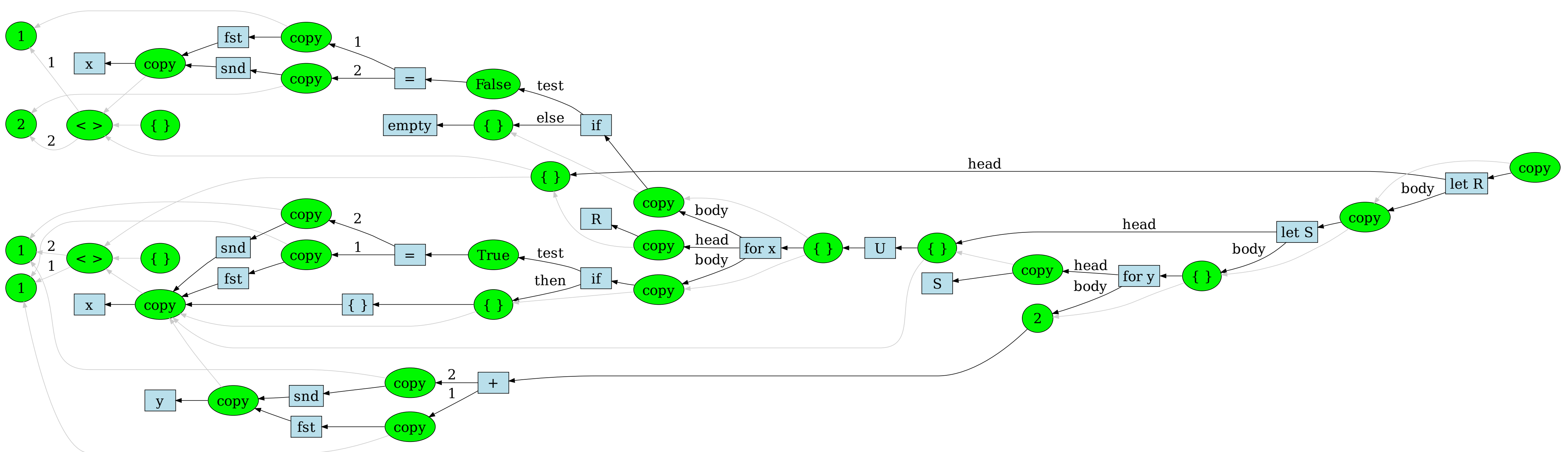
copy
2
1
snd
fst
copy
=
True
test
if
then
copy
{ }
R
copy
head
for x
body
{ }
U
{ }
S
copy
head
for y
body
{ }
2
+
copy
snd
y
copy
fst
copy
1
1
< >
{ }
x
copy

Explaining

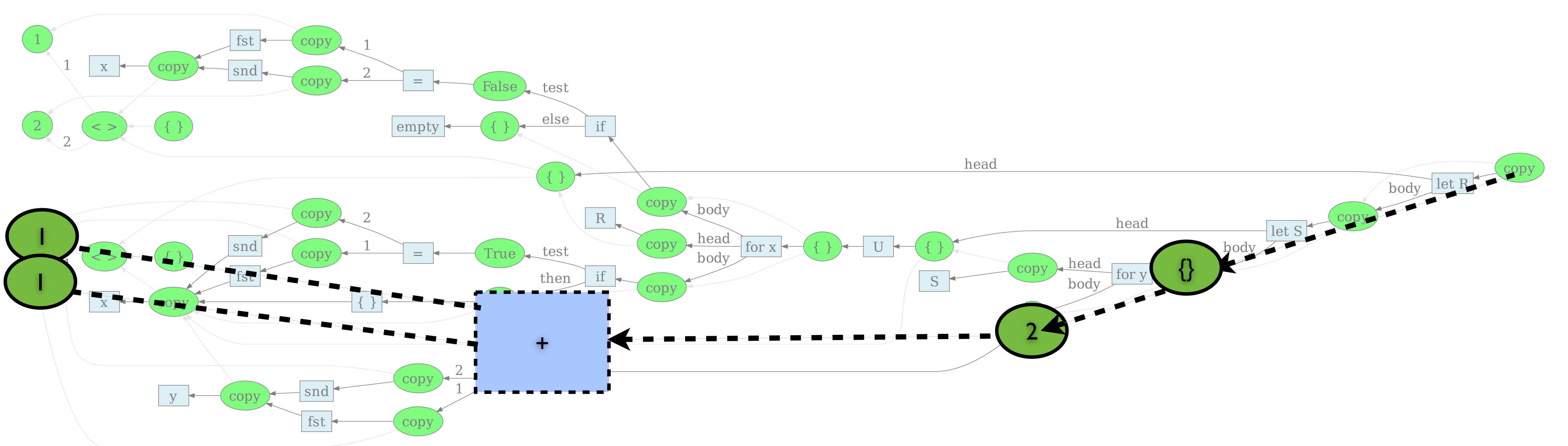
Note: Smallest consistent subgraph (NOT transitive closure!)



Summarizing

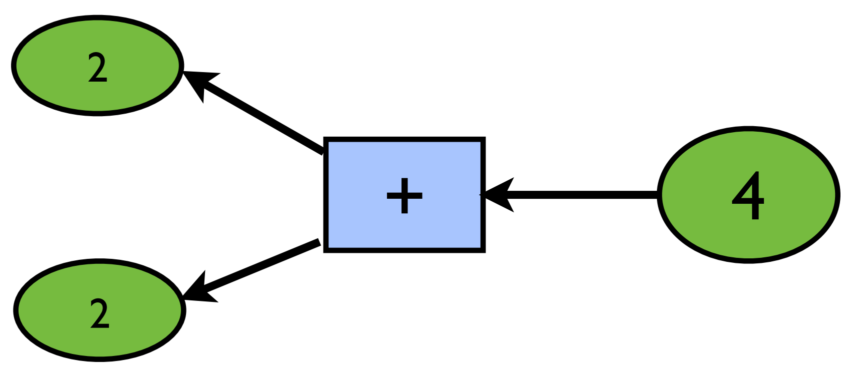


Summarizing



Graphs are partially "replayable"

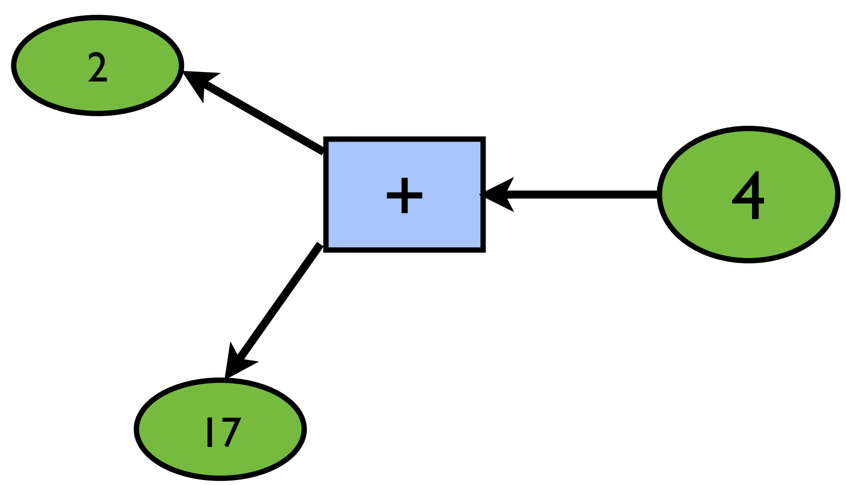
- If we change a value node, can try to "readjust" to recover consistency



- Formalized in (Acar, Ahmed, Cheney 08)

Graphs are partially "replayable"

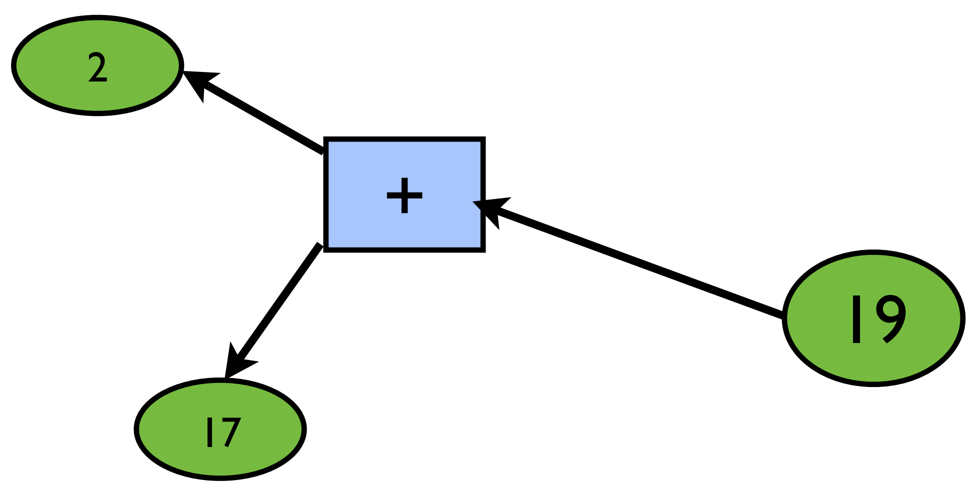
- If we change a value node, can try to "readjust" to recover consistency



- Formalized in (Acar, Ahmed, Cheney 08)

Graphs are partially "replayable"

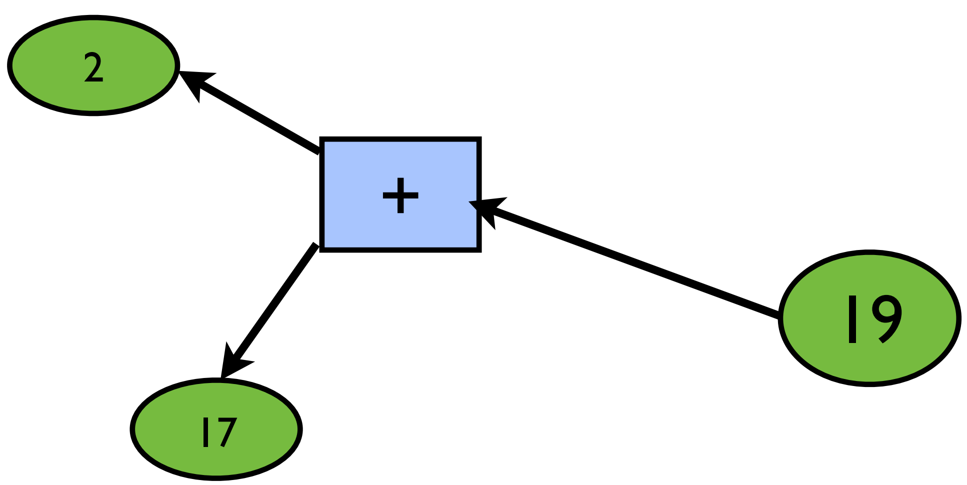
- If we change a value node, can try to "readjust" to recover consistency

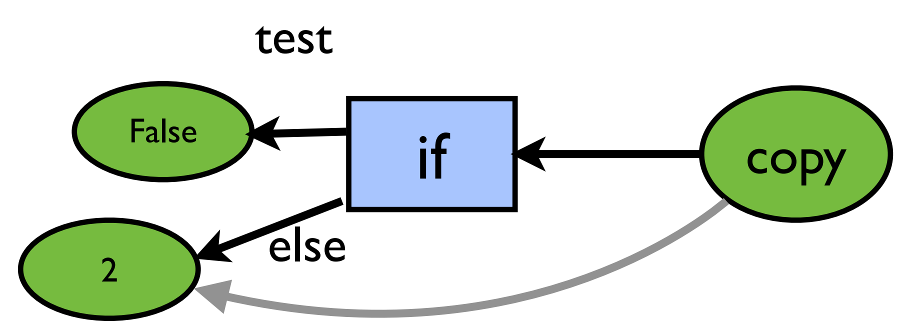


- Formalized in (Acar, Ahmed, Cheney 08)

Graphs are partially "replayable"

- If we change a value node, can try to "readjust" to recover consistency

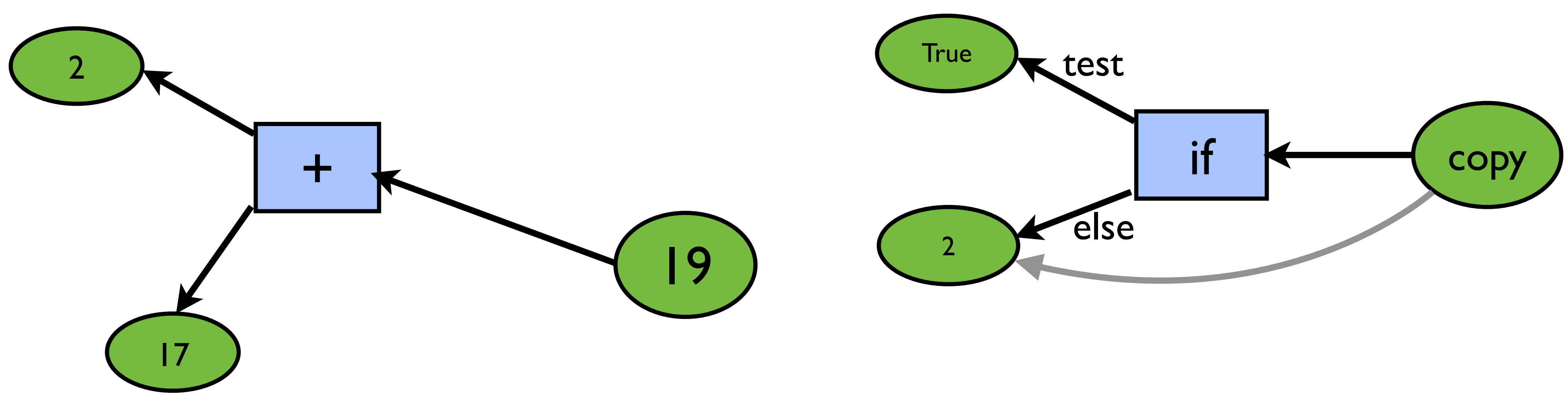




- Formalized in (Acar, Ahmed, Cheney 08)

Graphs are partially "replayable"

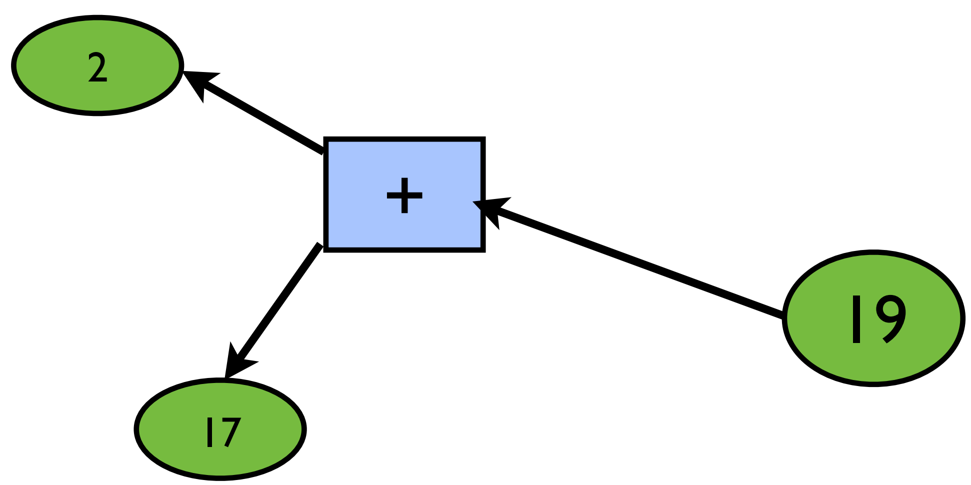
- If we change a value node, can try to "readjust" to recover consistency

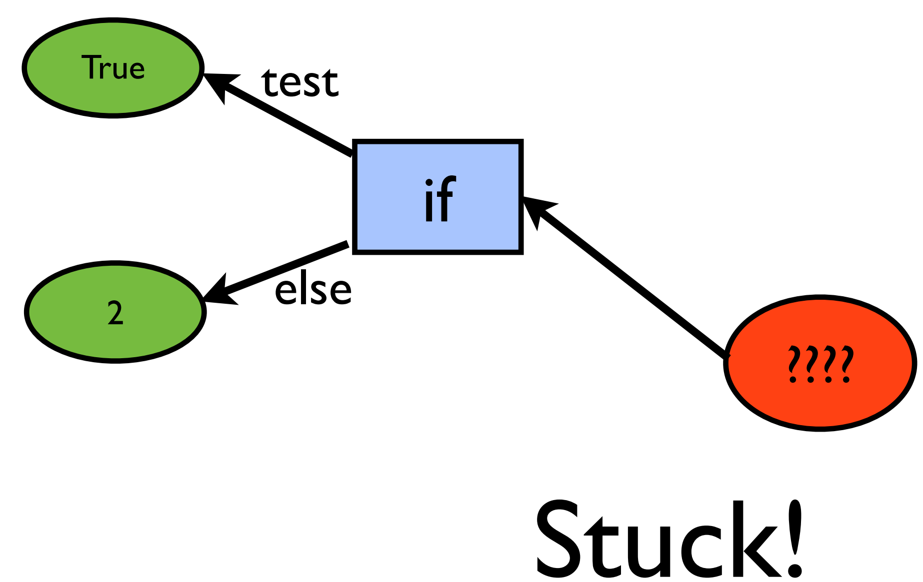


- Formalized in (Acar, Ahmed, Cheney 08)

Graphs are partially "replayable"

- If we change a value node, can try to "readjust" to recover consistency





- Formalized in (Acar, Ahmed, Cheney 08)

Implementation in Haskell

- Summarized in paper, full code on request
  - roughly 250 LOC for basic evaluator
  - another 300 for graphviz translation, basic queries, examples
- Point?
  - No claim of efficiency/scalability but easy to understand, experiment
  - Elucidates some tricky details that pictures hide
  - Similar "lightweight modeling" might be valuable for understanding/relating other WF/DB models

Related work

- This work synthesizes/rearranges ideas from several previous works & "folklore"
  - traces (Acar, Ahmed, Cheney 2008)
  - runs (Kwasnikowska, van den Bussche, DILS 2007, IPAW 2008)
  - OPM graphs (Moreau et al. IPAW 2008 etc.)
    - and many workflow systems
- More can be done to relate DB & workflow models

Future work

- This is work in progress
- Next steps:
  - Extending to understand/model other workflow features
  - Better grasp of "real" queries and features needed
  - Implementa(tion|ability)?
  - Optimization?

Conclusions

- DB & WF provenance have much in common
- We develop common graph model
  - with both intuitive & precise presentations
- Still much to do to relate and integrate DB & WF models
  - let alone integrate models at scale in real systems