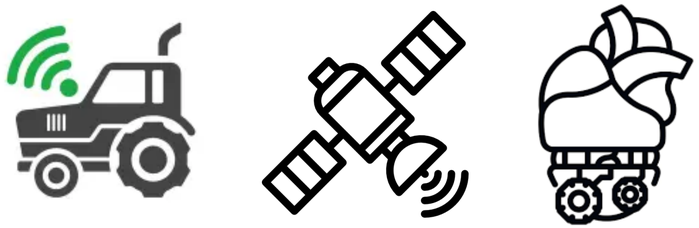
Logical Foundations of Intermittent Computing

Myra Dotzel

PhD Thesis Defense

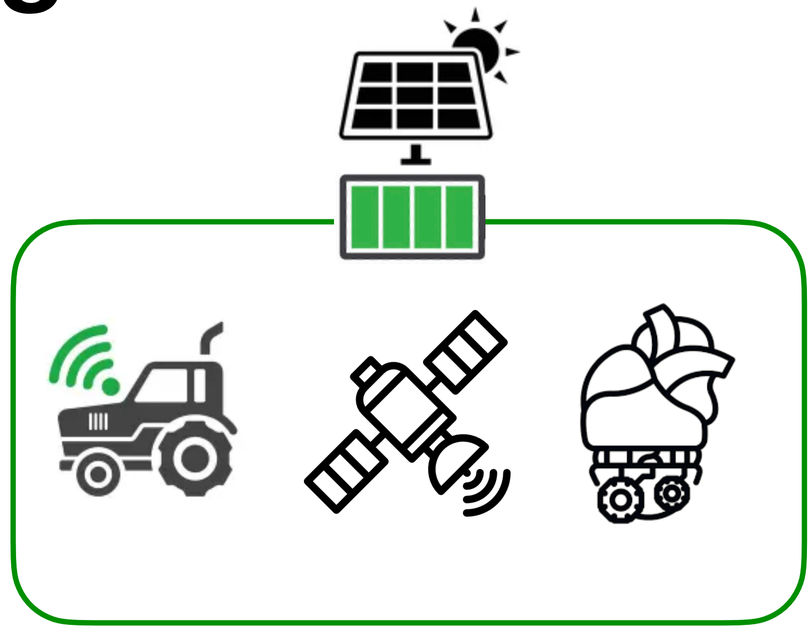
What is intermittent computing?

- tiny devices that compute in inaccessible environments



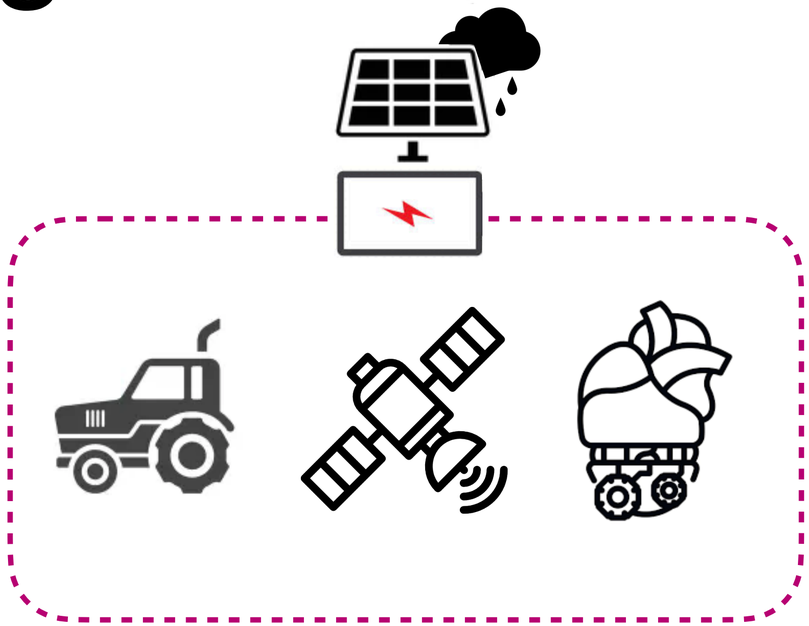
What is intermittent computing?

- tiny devices that compute in inaccessible environments
- cannot rely on continuous energy sources



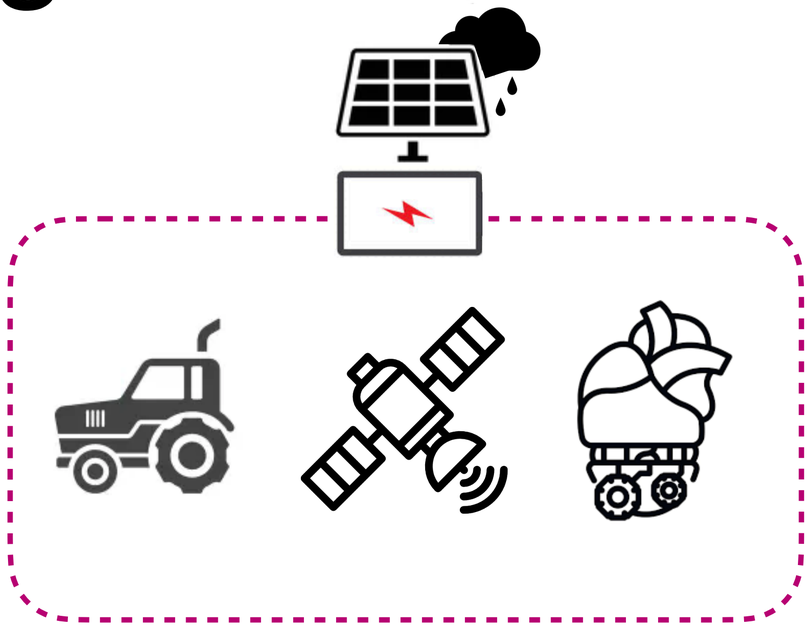
What is intermittent computing?

- tiny devices that compute in inaccessible environments
- cannot rely on continuous energy sources
- devices charge, compute, and crash



What is intermittent computing?

- tiny devices that compute in inaccessible environments
- cannot rely on continuous energy sources
- devices charge, compute, and crash
- NV survives power failures, V does not
- runtime system support needed to save state



NV

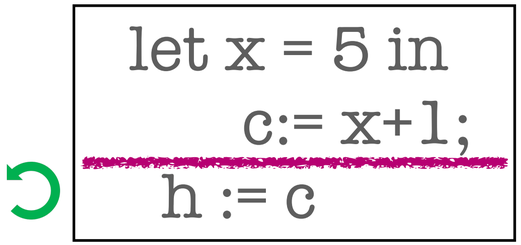
"nonvolatile memory"

V

"volatile memory"

How state is saved depends on execution mode

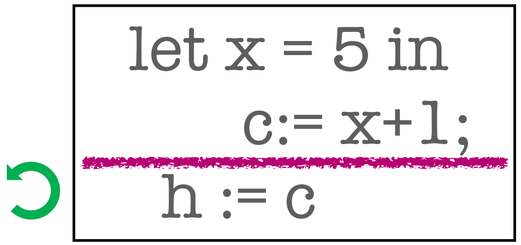
How state is saved depends on execution mode



JIT region

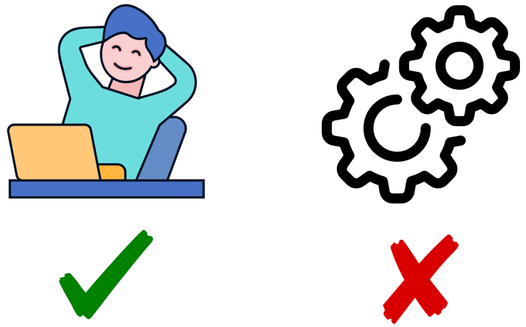
- **resume** execution
- save volatile state right **before power failure**

How state is saved depends on execution mode



JIT region

- **resume** execution
- save volatile state right **before power failure**



How state is saved depends on execution mode

```
let x = 5 in
      c:= x+1;
   h := c
```

JIT region

- **resume** execution
- save volatile state right **before power failure**

```
start_atom
    x := y;
    y := z;
    w := x + y
end_atom
```

Atomic region

- **re-execute** from the beginning
- make copies of nonvolatile state **before (re-)execution**
- different **logging styles**

How state is saved depends on execution mode

```
let x = 5 in
    c:= x+1;
h := c
```

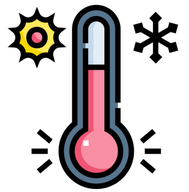
JIT region

- **resume** execution
- save volatile state right **before power failure**

```
start_atom
    x := y;
    y := z;
    w := x + y
end_atom
```

Atomic region

- **re-execute** from the beginning
- make copies of nonvolatile state **before (re-)execution**
- different **logging styles**



How state is saved depends on execution mode

```
let x = 5 in
      c:= x+1;
   h := c
```

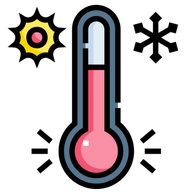
JIT region

- **resume** execution
- save volatile state right **before power failure**

```
start_atom
    x := y;
    y := z;
    w := x + y
end_atom
```

Atomic region

- **re-execute** from the beginning
- make copies of nonvolatile state **before (re-)execution**
- different **logging styles**



How state is saved depends on execution mode

```
let x = 5 in
    c:= x+1;
  h := c
```

- **resume** execution
- save volatile state right **before power failure**

Question:

Can programs execute correctly despite these power failures?

```
sta
   y := z;
   w := x + y
end_atom
```

- make copies of nonvolatile state **before (re-)execution**
- different **logging styles**

Atomic region

How state is saved depends on execution mode

```
let x = 5 in
      c:= x+1;
   h := c
```

- **resume** execution
- save volatile state right **before power failure**

Answer:

It depends on *what* we save, or ***checkpoint***

```
sta
    y := z;
    w := x + y
end_atom
```

Atomic region

- make copies of nonvolatile state **before (re-)execution**
- different **logging styles**

Which variables need to be checkpointed?

```
start_atom(?)
    x := y;
    y := z;
    w := x + y
end_atom
```

Checkpoint nothing

```
1 start_atom(_)
2     x := y;
3     y := z;
4     w := x + y
5  end_atom
```

NV	x	y	z	w
1	0	1	2	0
2	**1**	1	2	0
3	**1**	**2**	2	0

Checkpoint nothing

```
1 start_atom(_)
2     x := y;
3     y := z;
4     w := x + y
5  end_atom
```

NV	x	y	z	w
1	0	1	2	0
2	**1**	1	2	0
3	**1**	**2**	2	0
Restore	1	2	2	0

Checkpoint nothing

```
1 start_atom(_)
2     x := y;
3     y := z;
4     w := x + y
5  end_atom
```

NV	x	y	z	w
1	0	1	2	0
2	**1**	1	2	0
3	**1**	**2**	2	0
Restore	1	2	2	0
2	**2**	2	2	0

reads value of y
from failed exec.!

Checkpoint nothing

```
1 start_atom(_)
2     x := y;
3     y := z;
4     w := x + y
5  end_atom
```

NV	x	y	z	w
1	0	1	2	0
2	**1**	1	2	0
3	**1**	**2**	2	0
Restore	1	2	2	0
2	**2**	2	2	0
3	**2**	**2**	2	0

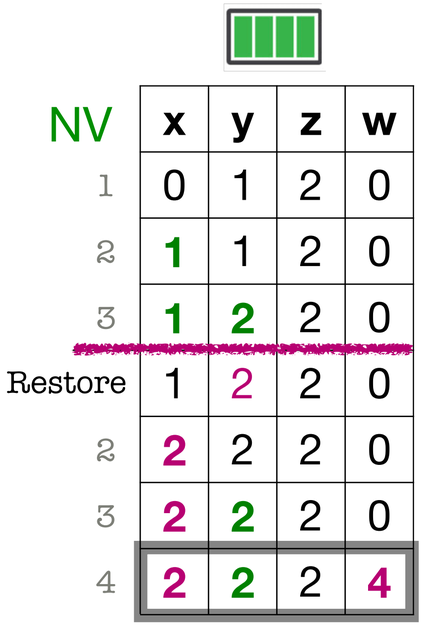
Checkpoint nothing

```
1 start_atom(_)
2     x := y;
3     y := z;
4     w := x + y
5  end_atom
```

NV	x	y	z	w
1	0	1	2	0
2	**1**	1	2	0
3	**1**	**2**	2	0
Restore	1	2	2	0
2	**2**	2	2	0
3	**2**	**2**	2	0
4	**2**	**2**	2	**4**

Checkpoint nothing

```
1 start_atom(_)
2     x := y;
3     y := z;
4     w := x + y
5 end_atom
```



NV	x	y	z	w
1	0	1	2	0
2	**1**	1	2	0
3	**1**	**2**	2	0
Restore	1	2	2	0
2	**2**	2	2	0
3	**2**	**2**	2	0
4	**2**	**2**	2	**4**

NV	x	y	z	w
1	0	1	2	0
2	**1**	1	2	0
3	**1**	**2**	2	0
4	**1**	**2**	2	**3**

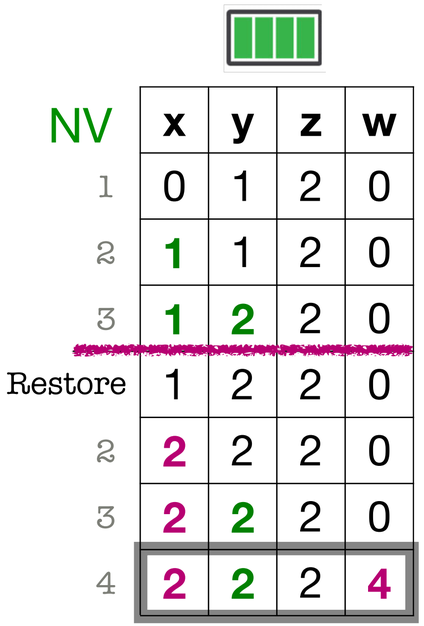
inconsistent memory!

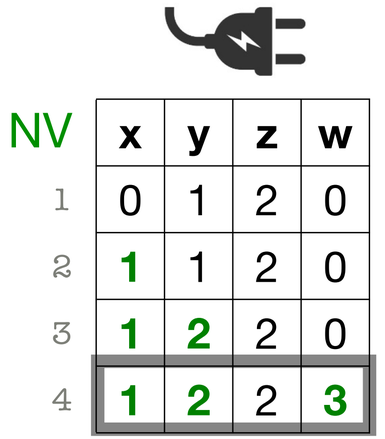
nothing

↳ consistency bugs!

Checkpoint nothing

```
1 start_atom(_)
2     x := y;
3     y := z;
4     w := x + y
5 end_atom
```





inconsistent memory!

nothing

↬ consistency bugs!

“Nonvolatile memory is a broken time machine.”

- Benjamin Ransford and Brandon Lucia, 2014

Checkpoint everything! (undo logging)

```
1 start_atom(x,y,z,w)
2     x := y;
3     y := z;
4     w := x + y
5  end_atom
```

NV

x	y	z	w
0	**1**	**2**	**0**

NV	x	y	z	w
1	0	1	2	0
2	**1**	1	2	0
3	**1**	**2**	2	0
4	**1**	**2**	2	**3**

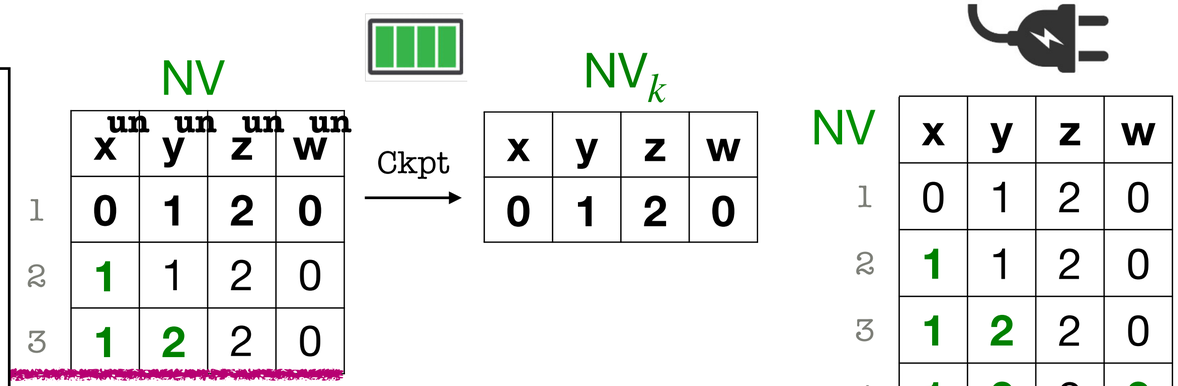
nothing

↳ consistency bugs!

everything

Checkpoint everything! (undo logging)

```
1 start_atom(x,y,z,w)
2     x := y;
3     y := z;
4     w := x + y
5  end_atom
```



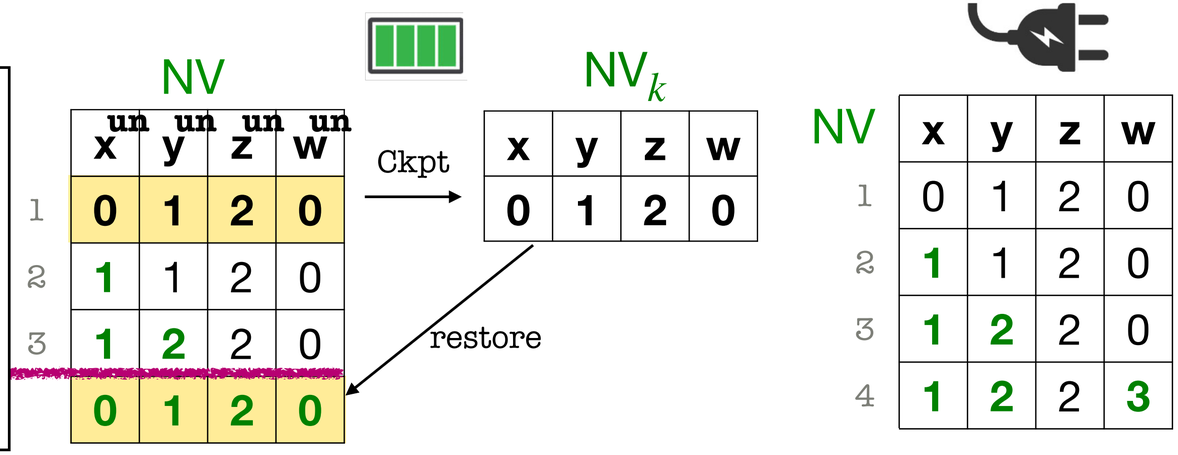
nothing

↬ consistency bugs!

everything

Checkpoint everything! (undo logging)

```
1 start_atom(x,y,z,w)
2     x := y;
3     y := z;
4     w := x + y
5  end_atom
```



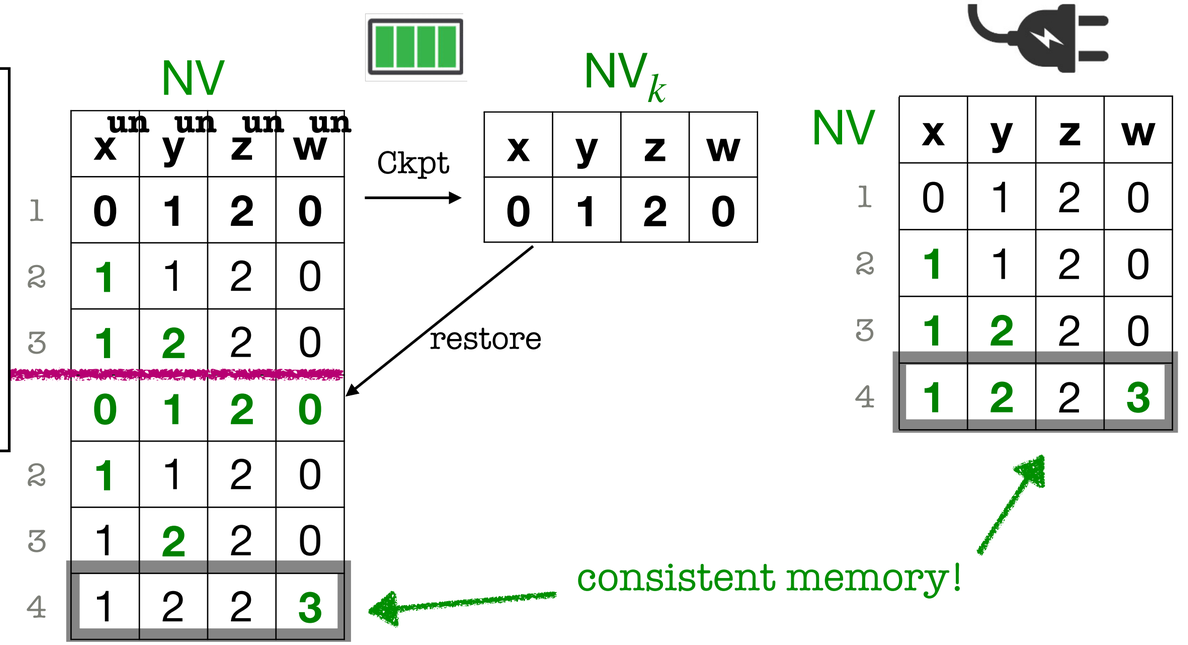
nothing

↬ consistency bugs!

everything

Checkpoint everything! (undo logging)

```
1 start_atom(x,y,z,w)
2       x := y;
3       y := z;
4       w := x + y
5  end_atom
```



nothing

↳ consistency bugs!

everything

↳ inefficient!

Do we really need to checkpoint everything?

```
1  start_atom(x,y,z,w)
2      x := y;
3      y := z;
4      w := x + y
5  end_atom
```

Do we really need to checkpoint everything?

```
1  start_atom(x,y,z,w)
2      x := y;
3      y := z;
4      w := x + y
5  end_atom
```

- z is never updated

Do we really need to checkpoint everything?

written before read

```
1  start_atom(x,y,z,w)
2     x := y;
3     y := z;
4     w := x + y
5  end_atom
```

- z is never updated
- x,w are written before being read

Do we really need to checkpoint everything?

```
1  start_atom(x,y,z,w)
2      x := y;
3      y := z;
4      w := x + y
5  end_atom
```

- z is never updated
- x,w are written before being read

but what about y?

Do we really need to checkpoint everything?

```
1  start_atom(x,y,z,w)
2      x := y;
3      y := z;
4      w := x + y
5  end_atom
```

- z is never updated
- x,w are written before being read

y is a “write-after-read (WAR) variable”

“No, only the WAR variables”

- Brandon Lucia and Benjamin Ransford, 2015

Checkpoint WAR variables (undo logging)

```
1 start_atom(y)
2     x := y;
3     y := z;
4     w := x + y
5  end_atom
```

NV

x	y	z	w
0	1	2	0

NV	x	y	z	w
1	0	1	2	0
2	**1**	1	2	0
3	**1**	**2**	2	0
4	**1**	**2**	2	**3**

nothing

↳ consistency bugs!

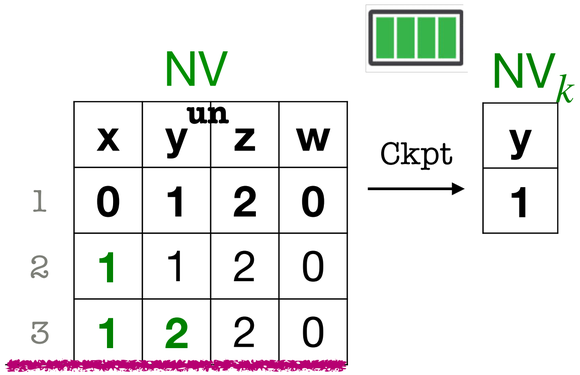
everything

↳ inefficient!

write-after-read variables

Checkpoint WAR variables (undo logging)

```
1 start_atom(y)
2     x := y;
3     y := z;
4     w := x + y
5 end_atom
```



nothing

↳ consistency bugs!

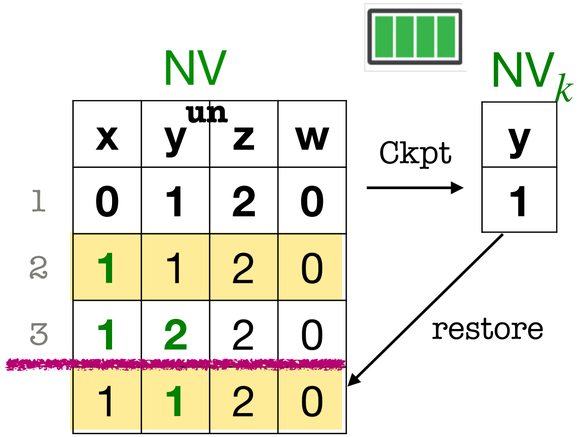
everything

↳ inefficient!

write-after-read variables

Checkpoint WAR variables (undo logging)

```
1 start_atom(y)
2     x := y;
3     y := z;
4     w := x + y
5 end_atom
```



NV	x	y	z	w
1	0	1	2	0
2	1	1	2	0
3	1	2	2	0
4	1	2	2	3

nothing

↳ consistency bugs!

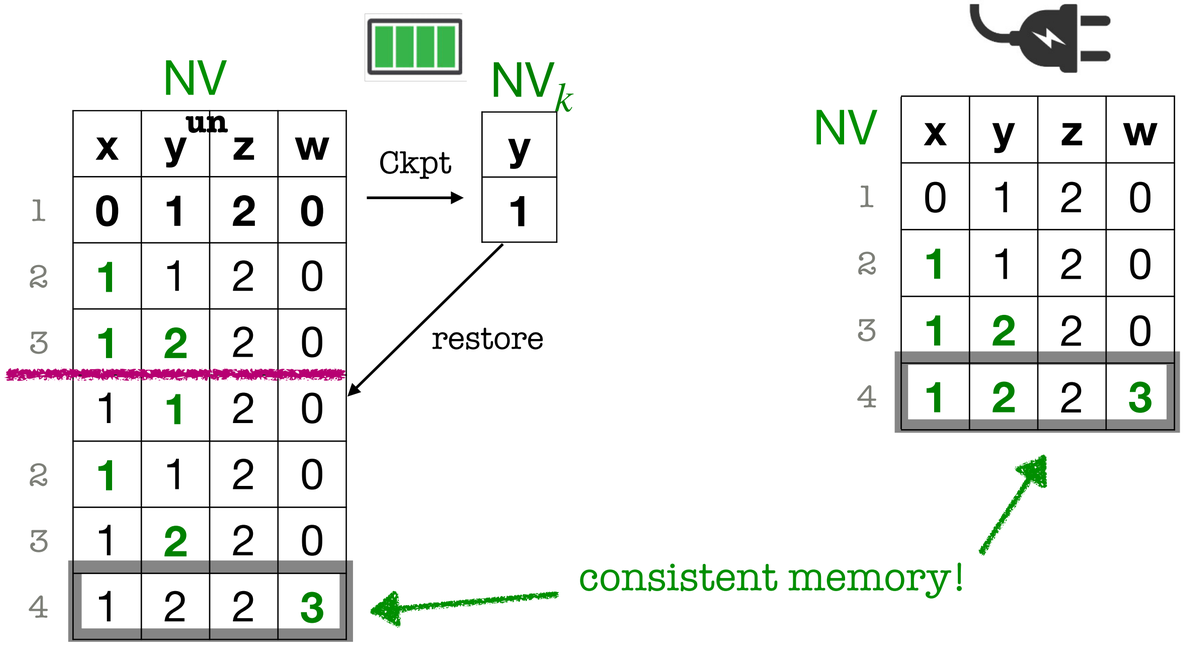
everything

↳ inefficient!

write-after-read variables

Checkpoint WAR variables (undo logging)

```
1 start_atom(y)
2     x := y;
3     y := z;
4     w := x + y
5 end_atom
```



nothing

↬ consistency bugs!

everything

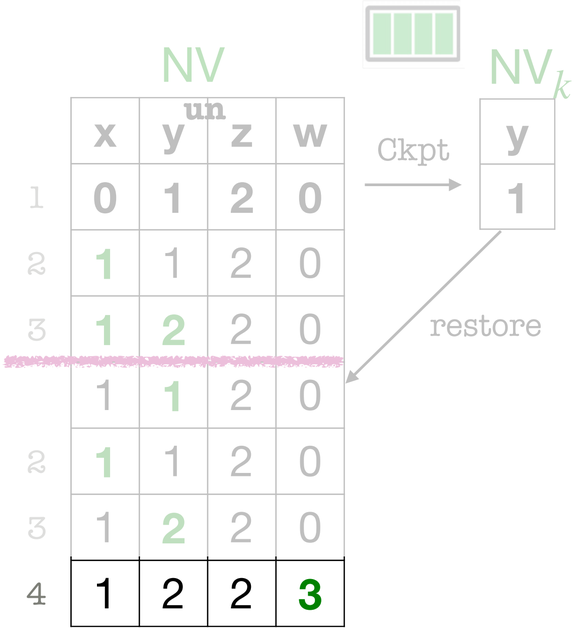
↬ inefficient!

write-after-read variables

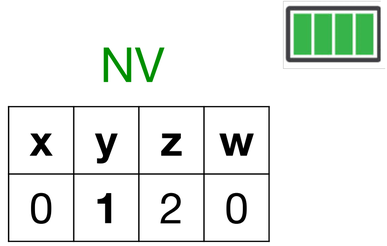
✓ state-of-the-art systems

Checkpoint WAR variables (undo vs redo logging)

```
1 start_atom(y)
2     x := y;
3     y := z;
4     w := x + y
5  end_atom
```



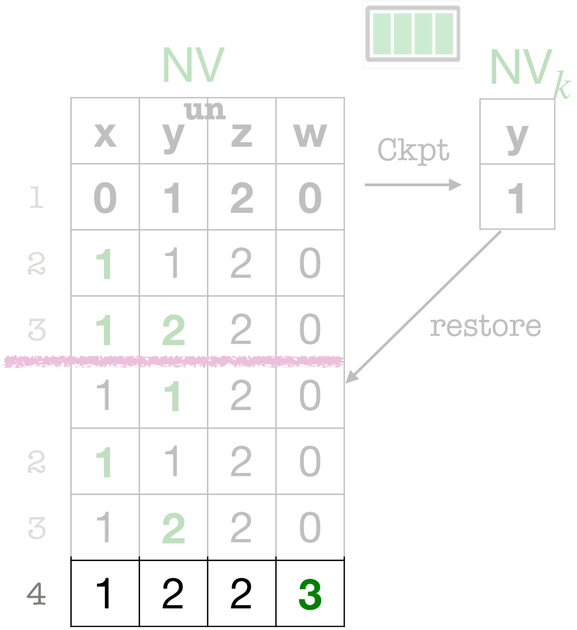
undo logging



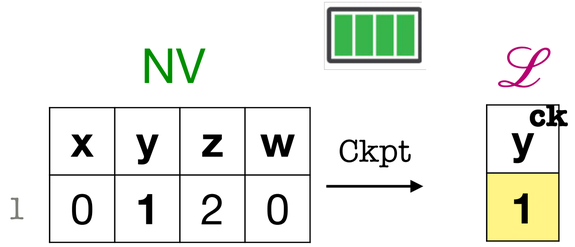
redo logging

Checkpoint WAR variables (undo vs redo logging)

```
1 start_atom(y)
2     x := y;
3     y := z;
4     w := x + y
5  end_atom
```



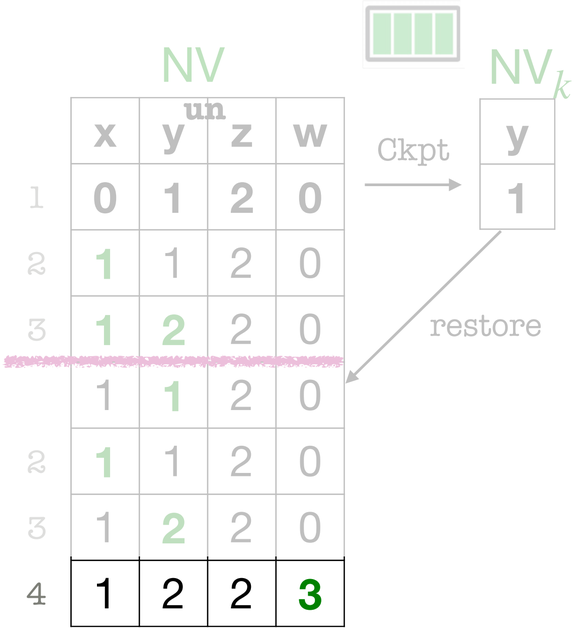
undo logging



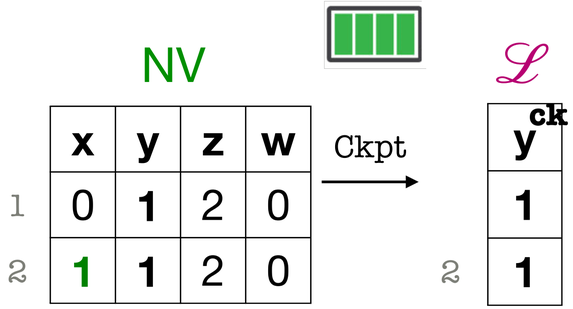
redo logging

Checkpoint WAR variables (undo vs redo logging)

```
1 start_atom(y)
2     x := y;
3     y := z;
4     w := x + y
5  end_atom
```



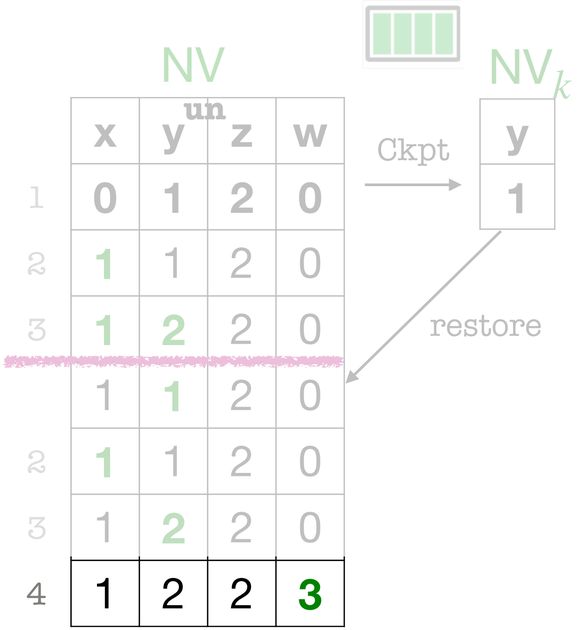
undo logging



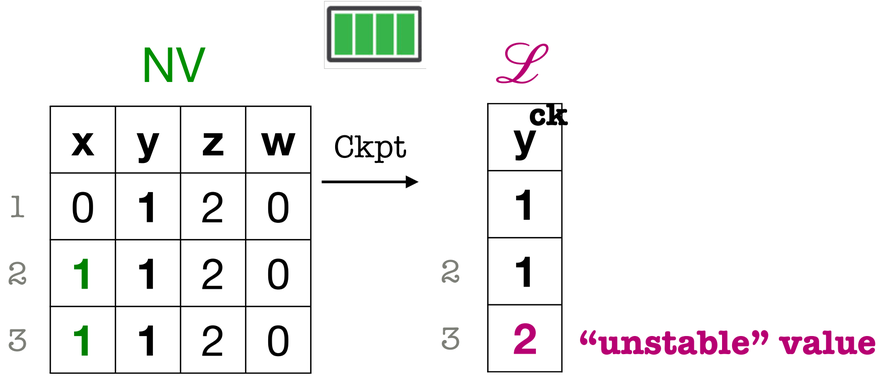
redo logging

Checkpoint WAR variables (undo vs redo logging)

```
1 start_atom(y)
2     x := y;
3     y := z;
4     w := x + y
5  end_atom
```



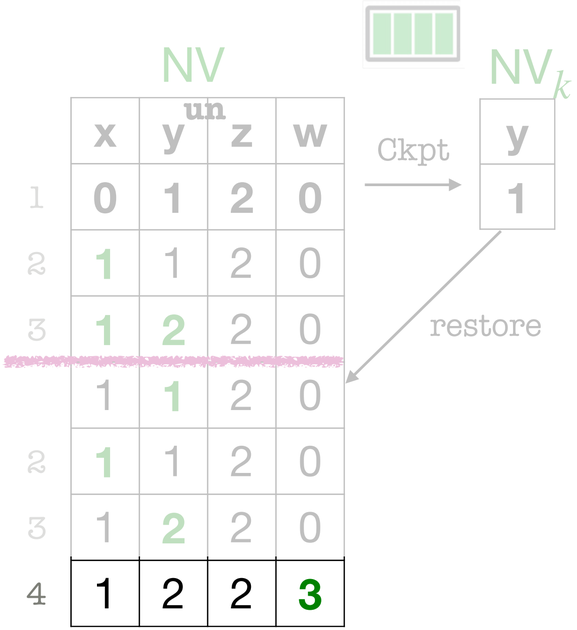
undo logging



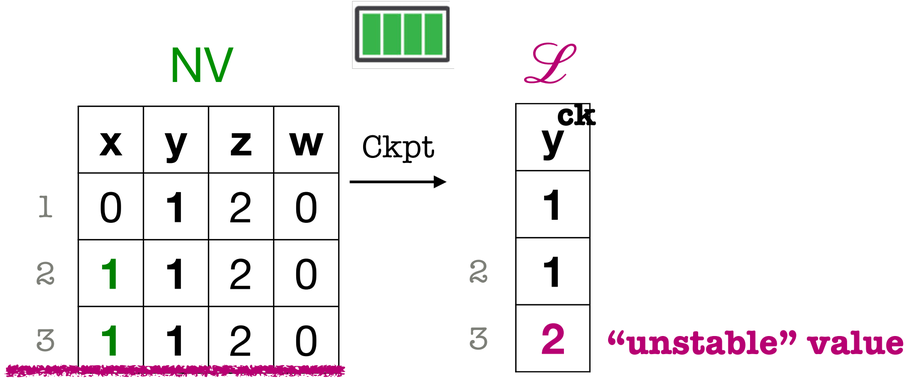
redo logging

Checkpoint WAR variables (undo vs redo logging)

```
1 start_atom(y)
2     x := y;
3     y := z;
4     w := x + y
5  end_atom
```



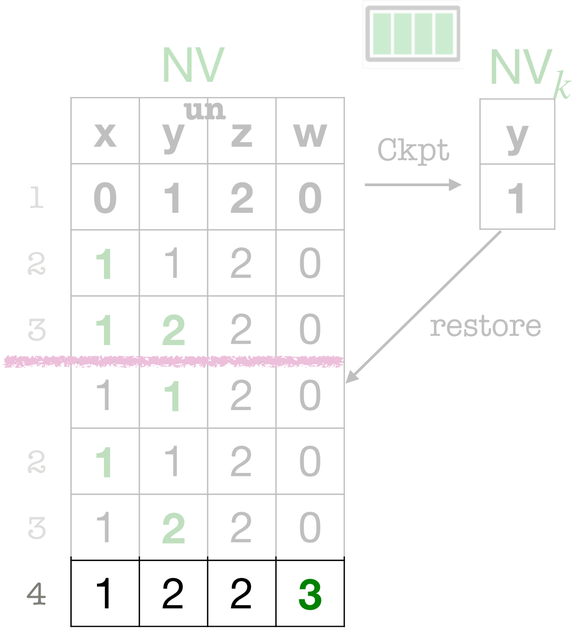
undo logging



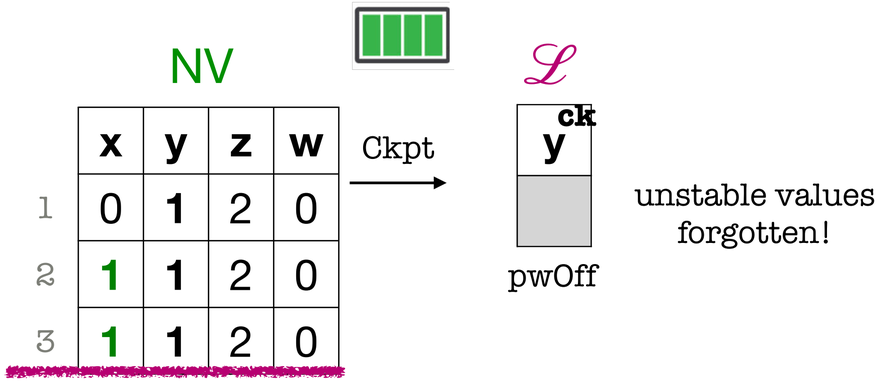
redo logging

Checkpoint WAR variables (undo vs redo logging)

```
1 start_atom(y)
2     x := y;
3     y := z;
4     w := x + y
5 end_atom
```



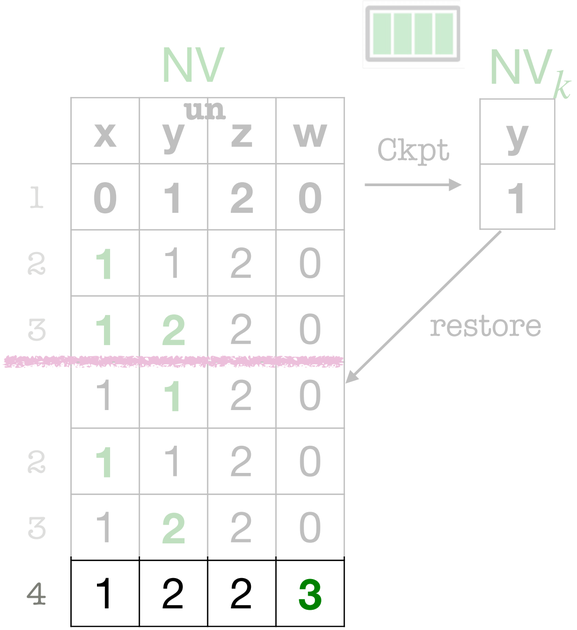
undo logging



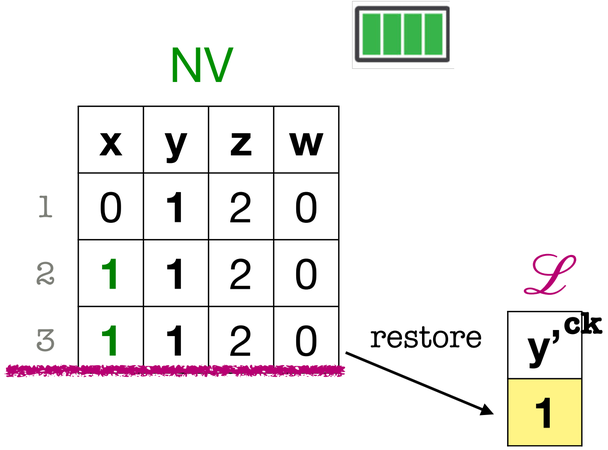
redo logging

Checkpoint WAR variables (undo vs redo logging)

```
1 start_atom(y)
2     x := y;
3     y := z;
4     w := x + y
5  end_atom
```



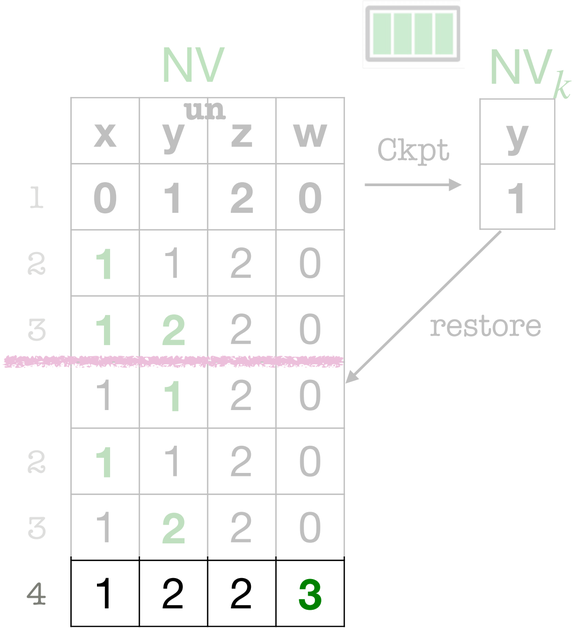
undo logging



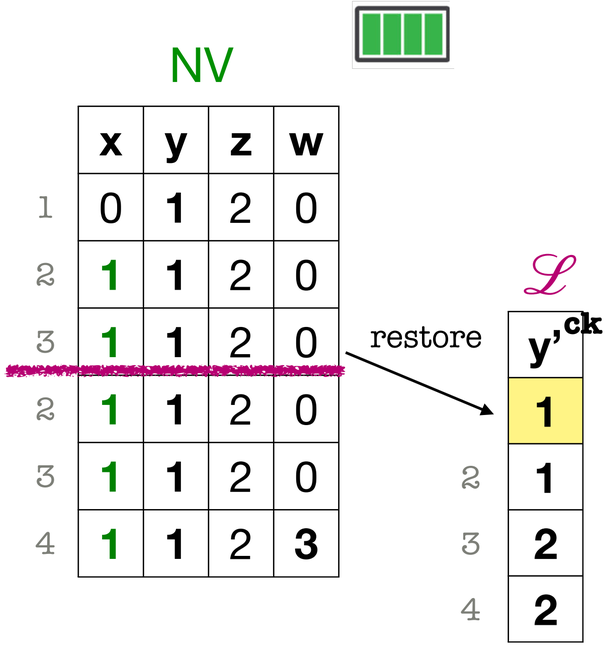
redo logging

Checkpoint WAR variables (undo vs redo logging)

```
1 start_atom(y)
2     x := y;
3     y := z;
4     w := x + y
5  end_atom
```



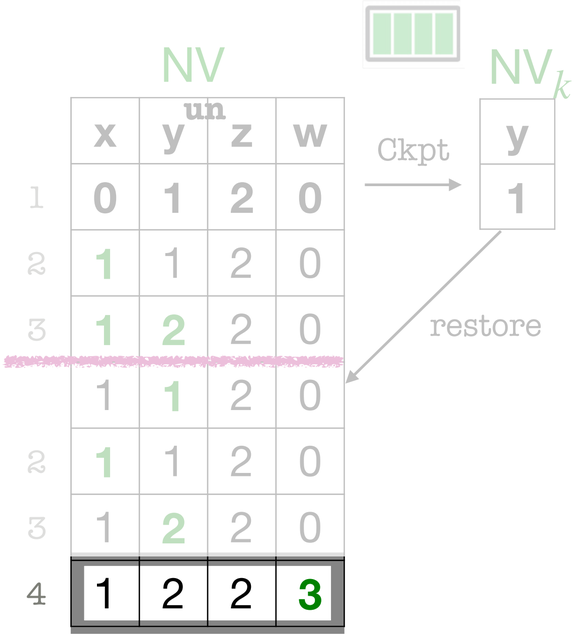
undo logging



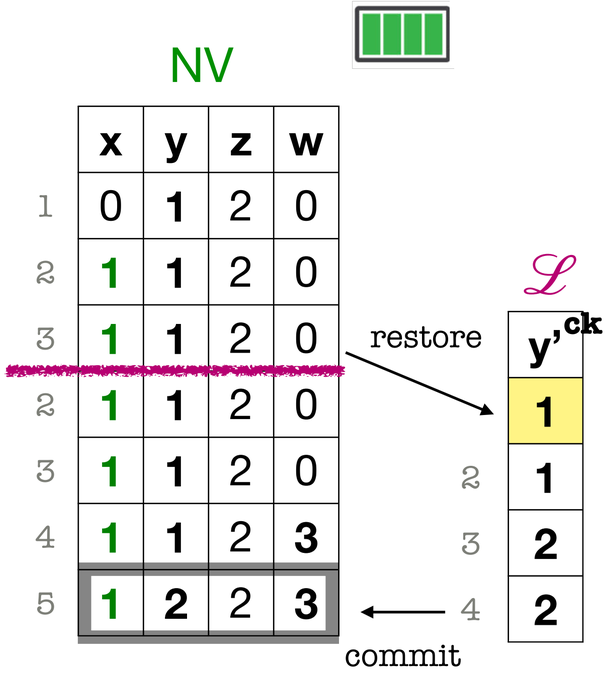
redo logging

Checkpoint WAR variables (undo vs redo logging)

```
1 start_atom(y)
2     x := y;
3     y := z;
4     w := x + y
5  end_atom
```



undo logging



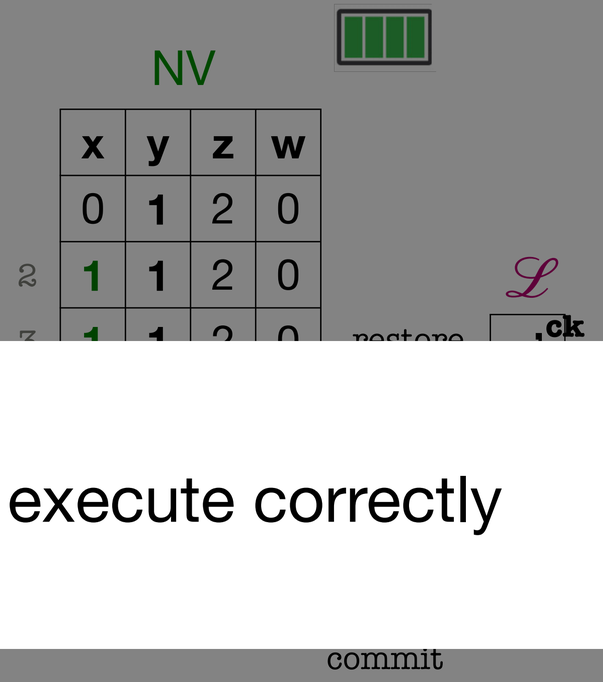
redo logging

“Undo and redo logging are equivalent!”

- Surbatovich et al., 2020

Checkpoint WAR variables (undo vs redo logging)

```
1 start_atom(y)
2     x := y;
3     y := z;
4     w := x + y
5 end_atom
```



Question:

How can we **prove** that programs execute correctly intermittently?

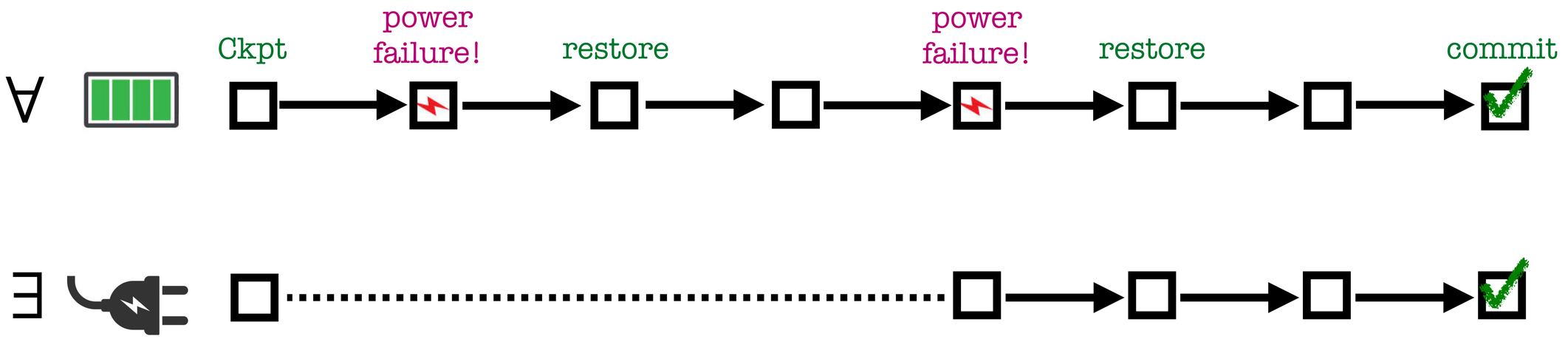
undo logging

redo logging

“Undo and redo logging are equivalent!”

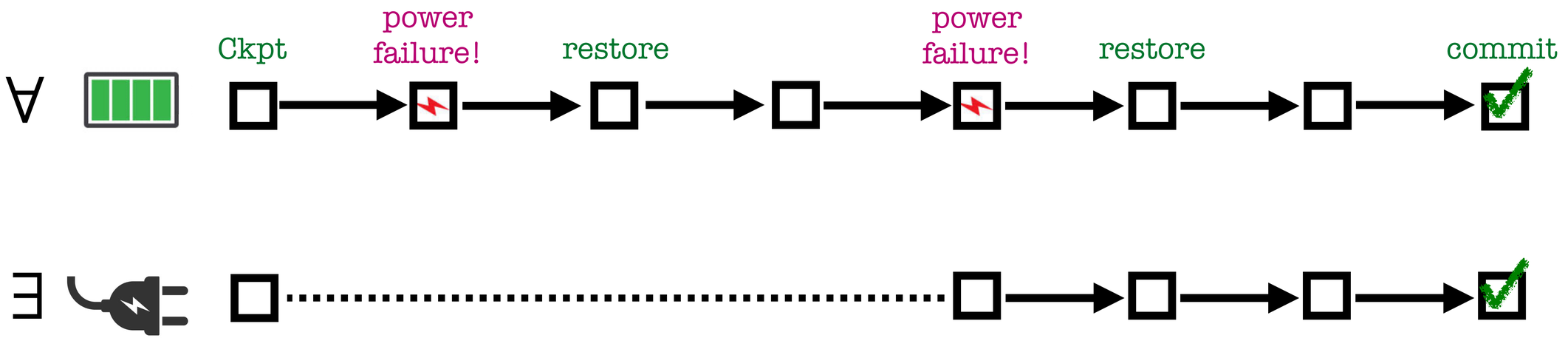
- Surbatovich et al., 2020

Ensuring the *correctness* of intermittent systems



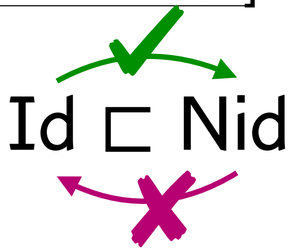
- first formal framework for reasoning about intermittent execution [Surbatovich et al. '20]

Ensuring the *correctness* of intermittent systems

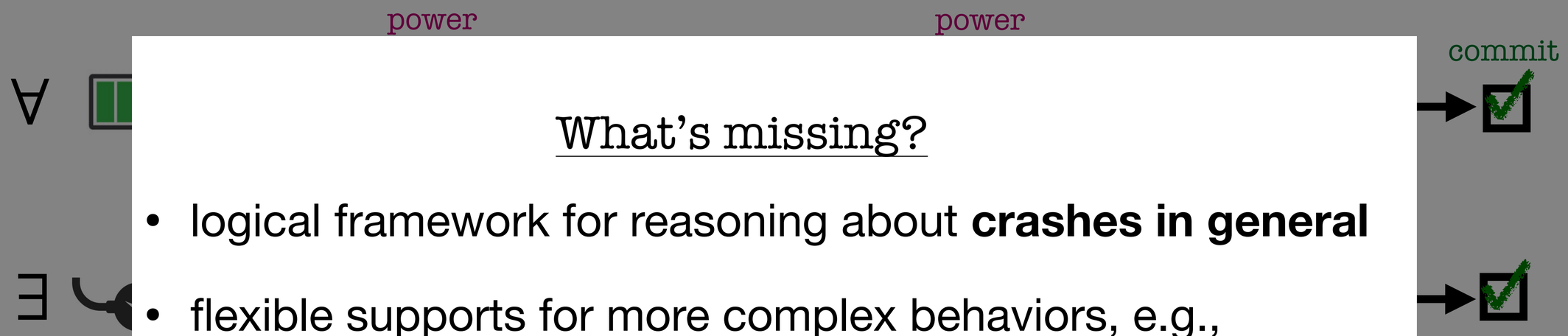


- first formal framework for reasoning about intermittent execution [Surbatovich et al. '20]
- information flow type system w/ correctness as noninterference [Surbatovich et al. '23]

Non-idempotent data (Nid) should not flow to idempotent variables (Id)!



Ensuring the *correctness* of intermittent systems

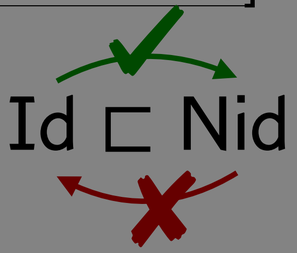


What's missing?

- logical framework for reasoning about **crashes in general**
- flexible supports for more complex behaviors, e.g., **outputs, concurrency, etc.**

- first form et al. '20]
- information flow type system w/ correctness as noninterference [Surbatovich et al. '23]

Non-idempotent data (Nid) should not flow to idempotent variables (Id)!



Thesis statement

“Type-based abstractions rooted in logic provide a uniform way to statically reason about the correctness of intermittent computation that scales to more realistic scenarios.”

- Me, today

Agenda

- Chapters 3 and 4: Modal crash types for intermittent computing
  [Derakhshan et al., ESOP '23]**[Dotzel et al., TOPLAS '25]**
- Chapter 5: Crash types with undo-logging
- Chapter 6: Crash types with I/O, nonidempotence, and flexible branching
- Chapter 7: Formal Foundations of Intermittent Concurrency

Modal Crash Types

[Derakhshan et al., ESOP ’23][Dotzel et al., TOPLAS ’25]

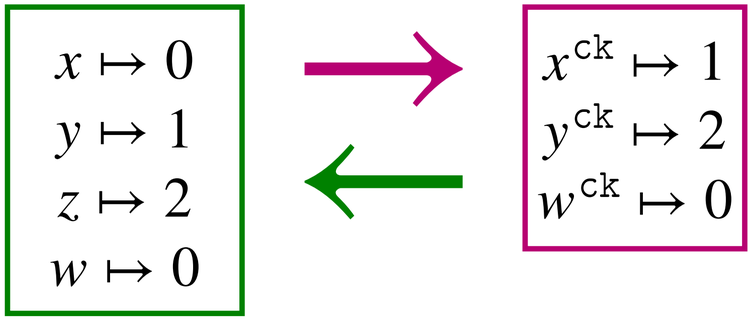
“First logical interpretation of intermittent computing”

power failure, restore, commit.

“First logical interpretation of intermittent computing”

power failure, restore, commit.

<u>One approach:</u> checkpoint everything not read-only [Derakhshan et al. ’23]




```
start_atom(x,y,w)
    x := y;
    y := z;
    w := x + y
end_atom
```

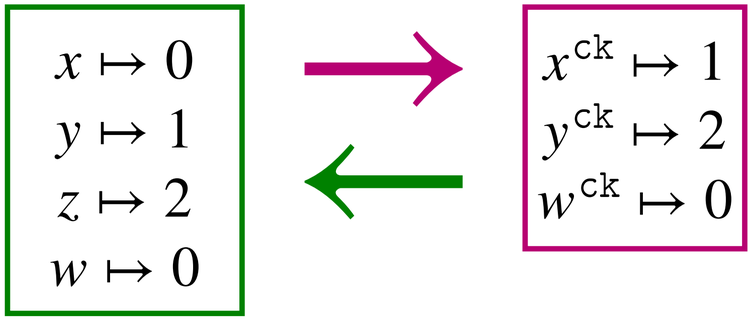
redo logging

$\mathscr{L}$ combined in V

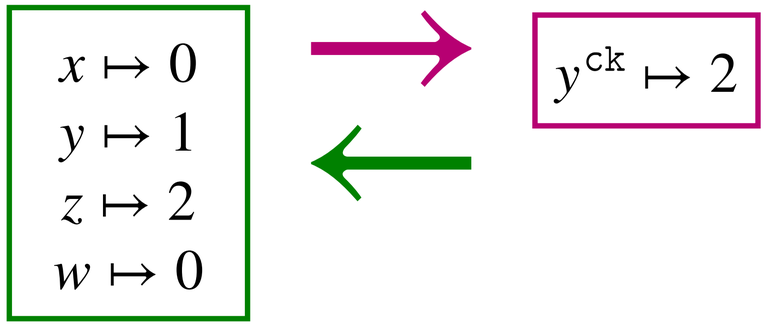
“First logical interpretation of intermittent computing”

power failure, restore, commit.

<u>One approach:</u> checkpoint everything not read-only [Derakhshan et al. ’23]



<u>More efficient:</u> checkpoint only write-after-read variables [Dotzel et al. ‘25]




```
start_atom(y)
    x := y;
    y := z;
    w := x + y
end_atom
```

redo logging

$\mathscr{L}$ combined in V

Types allow us to automatically check that programs will execute **correctly intermittently**.

Basic types and access qualifiers

basic types $T := \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$

access qualifiers $qAcc := \texttt{CK} \mid \texttt{RD} \mid \texttt{MtWt} \mid \texttt{Wtn}$

unstable types $\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid v_t$

stable types $\tau^s := \uparrow \tau^u \mid$  $\rightsquigarrow \tau^s$

CK — checkpointed

RD — read only

MtWt — must-first-write

Wtn — written

```
start_atom(y)
    x := y;
    y := z;
    w := x + y
end_atom
```

Basic types and access qualifiers

basic types $T := \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$

access qualifiers $qAcc := \texttt{CK} \mid \texttt{RD} \mid \texttt{MtWt} \mid \texttt{Wtn}$

unstable types $\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid v_t$

stable types $\tau^s := \uparrow \tau^u \mid$  $\rightsquigarrow \tau^s$

CK — checkpointed

RD — read only

MtWt — must-first-write

Wtn — written

```
start_atom(y)
    x := y;
    y := z;
    w := x + y
end_atom
```

- y is checkpointed y : CK

Basic types and access qualifiers

basic types $T := \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$

access qualifiers $qAcc := \texttt{CK} \mid \texttt{RD} \mid \texttt{MtWt} \mid \texttt{Wtn}$

unstable types $\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid v_t$

stable types $\tau^s := \uparrow \tau^u \mid$  $\rightsquigarrow \tau^s$

CK — checkpointed

RD — read only

MtWt — must-first-write

Wtn — written

```
start_atom(y)
    x := y;
    y := z;
    w := x + y
end_atom
```

- y is checkpointed — y : CK
- z is never updated — z : RD

Basic types and access qualifiers

basic types $T := \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$

access qualifiers $qAcc := \texttt{CK} \mid \texttt{RD} \mid \texttt{MtWt} \mid \texttt{Wtn}$

unstable types $\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid v_t$

stable types $\tau^s := \uparrow \tau^u \mid$  $\rightsquigarrow \tau^s$

CK — checkpointed

RD — read only

MtWt — must-first-write

Wtn — written

```
start_atom(y)
    x := y;
    y := z;
    w := x + y
end_atom
```

- y is checkpointed — y : CK
- z is never updated — z : RD
- x,w are written before being read — x,w : MtWt

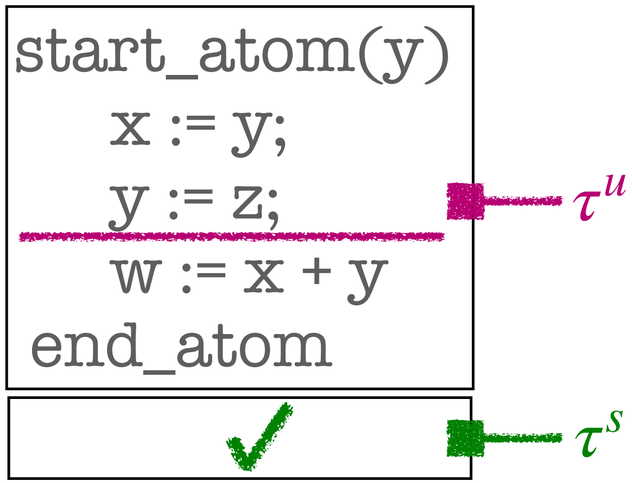
Stable and unstable types

basic types $T := \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$

access qualifiers $qAcc := \texttt{CK} \mid \texttt{RD} \mid \texttt{MtWt} \mid \texttt{Wtn}$

unstable types $\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid v_t$

stable types $\tau^s := \uparrow \tau^u \mid$  $\rightsquigarrow \tau^s$



τ^u **in-progress execution** /

values in **volatile memory** (e.g., logs)

τ^s **successful execution** /

values in **nonvolatile memory**

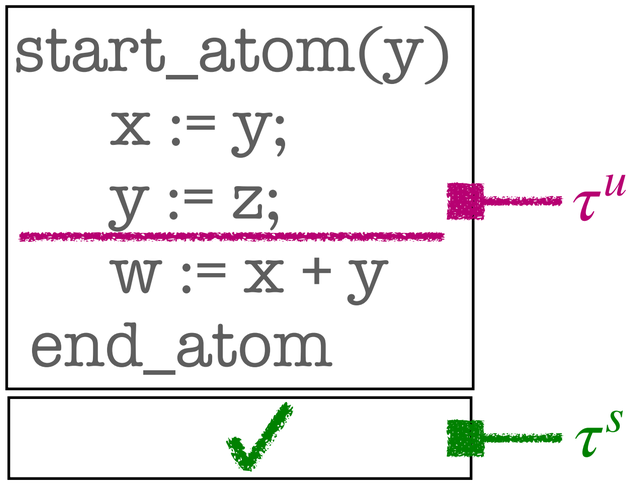
Adjoint Logic*

basic types $T := \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$

access qualifiers $qAcc := \texttt{CK} \mid \texttt{RD} \mid \texttt{MtWt} \mid \texttt{Wtn}$

unstable types $\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid v_t$

stable types $\tau^s := \uparrow \tau^u \mid$  $\rightsquigarrow \tau^s$



Independence principle: validity does not depend on truth.

* [Pruiksma et al. '21, Pfenning et al. '01, Benton et al. '97]

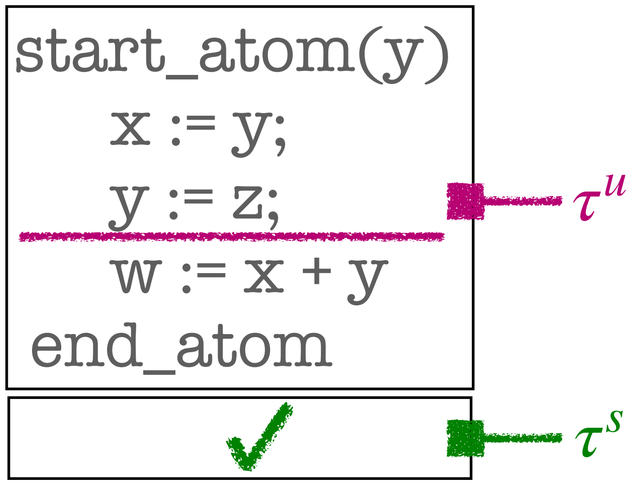
Adjoint Logic*

basic types $T := \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$

access qualifiers $qAcc := \texttt{CK} \mid \texttt{RD} \mid \texttt{MtWt} \mid \texttt{Wtn}$

unstable types $\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid \nu_t$

stable types $\tau^s := \uparrow \tau^u \mid$  $\rightsquigarrow \tau^s$



Independence principle: validity does not depend on truth.

"Zena the cat *always* steals my hair tie"

"Zena the cat stole my hair tie *today*"

* [Pruiksma et al. '21, Pfenning et al. '01, Benton et al. '97]

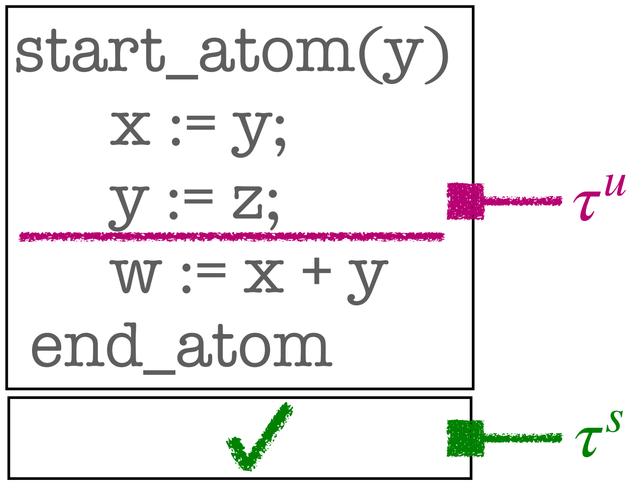
Adjoint Logic*

basic types $T := \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$

access qualifiers $qAcc := \texttt{CK} \mid \texttt{RD} \mid \texttt{MtWt} \mid \texttt{Wtn}$

unstable types $\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid \nu_t$

stable types $\tau^s := \uparrow \tau^u \mid$  $\rightsquigarrow \tau^s$



Independence principle: validity does not depend on truth.

"Zena the cat *always* steals my hair tie"

↑ ↓

"Zena the cat stole my hair tie *today*"

* [Pruiksma et al. '21, Pfenning et al. '01, Benton et al. '97]

Adjoint Logic*

basic types $T := \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$

access qualifiers $qAcc := \texttt{CK} \mid \texttt{RD} \mid \texttt{MtWt} \mid \texttt{Wtn}$

unstable types $\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid \nu_t$

stable types $\tau^s := \uparrow \tau^u \mid$  $\rightsquigarrow \tau^s$

```
start_atom(y)
    x := y;
    y := z;
    w := x + y
end_atom
```

τ^u

τ^s

Independence principle: validity does not depend on truth.

"Zena the cat *always* steals my hair tie"

↑ ↓

"Zena the cat stole my hair tie *today*"

Here: a stable (*s*) computation must not depend on unstable (*u*) values.

* [Pruiksma et al. '21, Pfenning et al. '01, Benton et al. '97]

Adjoint Logic*

basic types $T := \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$

access qualifiers $qAcc := \texttt{CK} \mid \texttt{RD} \mid \texttt{MtWt} \mid \texttt{Wtn}$

unstable types $\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid \nu_t$

stable types $\tau^s := \uparrow \tau^u \mid$  $\rightsquigarrow \tau^s$

```
start_atom(y)
    x := y;
    y := z;
    w := x + y
end_atom
```

τ^u

τ^s

Independence principle: validity does not depend on truth.

"Zena the cat *always* steals my hair tie"

$\uparrow \downarrow$

"Zena the cat stole my hair tie *today*"

Here: a stable (*s*) computation must not depend on unstable (*u*) values.

τ^s

Restore $\uparrow$ $\downarrow$ PwOff/Commit

τ^u

* [Pruiksma et al. '21, Pfenning et al. '01, Benton et al. '97]

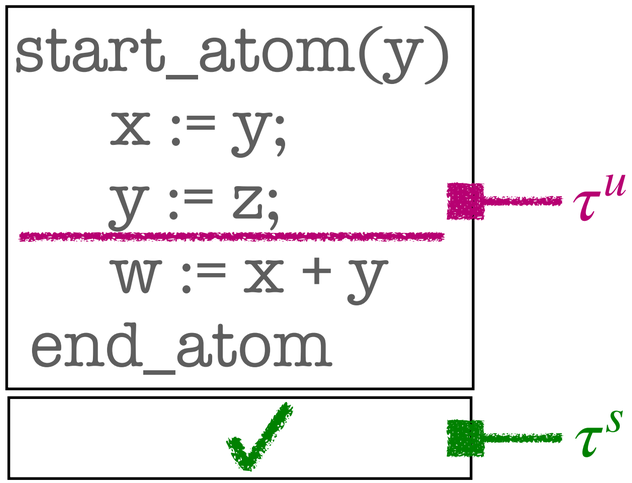
Crash Type

basic types $T := \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$

access qualifiers $qAcc := \texttt{CK} \mid \texttt{RD} \mid \texttt{MtWt} \mid \texttt{Wtn}$

unstable types $\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid v_t$

stable types $\tau^s := \uparrow \tau^u \mid$  $\rightsquigarrow \tau^s$



$$C = \underbrace{\qquad}_{\text{Successful execution}} \vee \underbrace{\qquad}_{\text{Failed execution}}$$

Crash Type

basic types $T := \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$

access qualifiers $qAcc := \texttt{CK} \mid \texttt{RD} \mid \texttt{MtWt} \mid \texttt{Wtn}$

unstable types $\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid v_t$

stable types $\tau^s := \uparrow \tau^u \mid$  $\rightsquigarrow \tau^s$

```
start_atom(y)
    x := y;
    y := z;
    w := x + y
end_atom
```

τ^u

τ^s

intermittent execution

$C = \downarrow$ Successful execution $\vee$ Failed execution

Crash Type

basic types $T := \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$

access qualifiers $qAcc := \texttt{CK} \mid \texttt{RD} \mid \texttt{MtWt} \mid \texttt{Wtn}$

unstable types $\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid v_t$

stable types $\tau^s := \uparrow \tau^u \mid$  $\rightsquigarrow \tau^s$

```
start_atom(y)
    x := y;
    y := z;
    w := x + y
end_atom
```

τ^u

τ^s

Commit

$C = \downarrow \uparrow \texttt{unit} \vee$ ~~_____~~

Successful execution

Failed execution

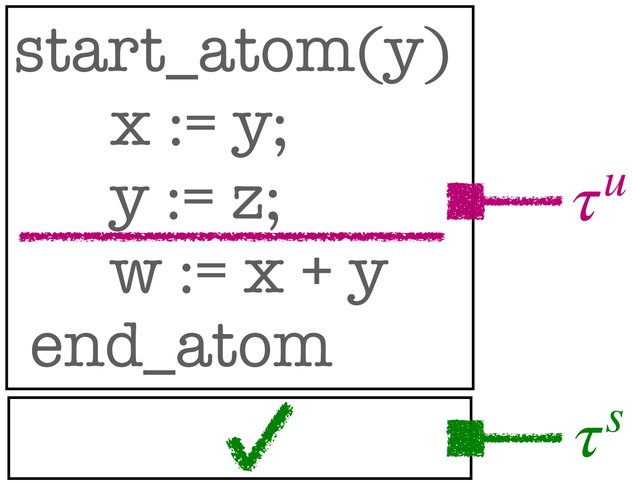
Crash Type

basic types $T := \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$

access qualifiers $qAcc := \texttt{CK} \mid \texttt{RD} \mid \texttt{MtWt} \mid \texttt{Wtn}$

unstable types $\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid v_t$

stable types $\tau^s := \uparrow \tau^u \mid$  $\rightsquigarrow \tau^s$



$$C = \downarrow \uparrow \texttt{unit} \vee \overset{\texttt{PwOff}}{\downarrow}$$

Successful execution

Failed execution

Crash Type

basic types $T := \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$

access qualifiers $qAcc := \texttt{CK} \mid \texttt{RD} \mid \texttt{MtWt} \mid \texttt{Wtn}$

unstable types $\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid v_t$

stable types $\tau^s := \uparrow \tau^u \mid$  $\rightsquigarrow \tau^s$

```
start_atom(y)
    x := y;
    y := z;
    w := x + y
end_atom
```

τ^u

τ^s

Charge

$C = \downarrow \uparrow \texttt{unit} \vee \downarrow ($  $\rightsquigarrow \_\_\_)$

Successful execution

Failed execution

Crash Type

basic types $T := \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$

access qualifiers $qAcc := \texttt{CK} \mid \texttt{RD} \mid \texttt{MtWt} \mid \texttt{Wtn}$

unstable types $\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid v_t$

stable types $\tau^s := \uparrow \tau^u \mid$  $\rightsquigarrow \tau^s$

```
start_atom(y)
    x := y;
    y := z;
    w := x + y
end_atom
```

τ^u

τ^s

Restore

$$C = \downarrow \uparrow \texttt{unit} \vee \downarrow (\text{  } \rightsquigarrow \uparrow \_)$$

Successful execution

Failed execution

Crash Type

basic types $T := \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$

access qualifiers $qAcc := \texttt{CK} \mid \texttt{RD} \mid \texttt{MtWt} \mid \texttt{Wtn}$

unstable types $\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid v_t$

stable types $\tau^s := \uparrow \tau^u \mid$  $\rightsquigarrow \tau^s$

```
start_atom(y)
    x := y;
    y := z;
    w := x + y
end_atom
```

τ^u

τ^s

Re-execute

$$C = \downarrow \uparrow \texttt{unit} \vee \downarrow (\text{  } \rightsquigarrow \uparrow C)$$

Successful execution

Failed execution

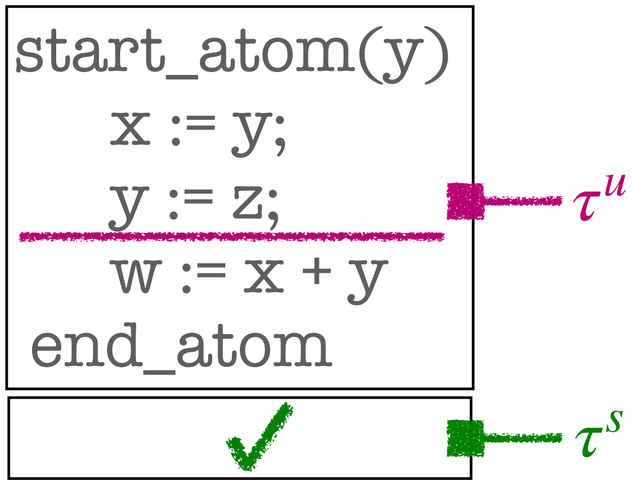
Typing judgments

basic types $T := \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$

access qualifiers $qAcc := \texttt{CK} \mid \texttt{RD} \mid \texttt{MtWt} \mid \texttt{Wtn}$

unstable types $\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid v_t$

stable types $\tau^s := \uparrow \tau^u \mid$  $\rightsquigarrow \tau^s$



Unstable typing judgment

$$\frac{}{} \vdash \frac{}{}$$

Stable typing judgment

$$\frac{}{} \vdash \frac{}{}$$

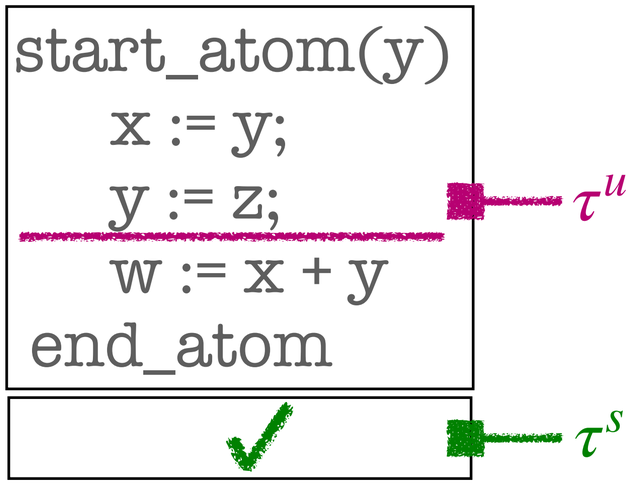
Typing judgments

basic types $T := \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$

access qualifiers $qAcc := \texttt{CK} \mid \texttt{RD} \mid \texttt{MtWt} \mid \texttt{Wtn}$

unstable types $\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid v_t$

stable types $\tau^s := \uparrow \tau^u \mid$  $\rightsquigarrow \tau^s$



Unstable typing judgment

$$\frac{}{\vdash \texttt{x} := \texttt{y} : \texttt{C}}$$

Command is well-typed

Stable typing judgment

$$\frac{\vdash \quad}{\vdash \quad}$$

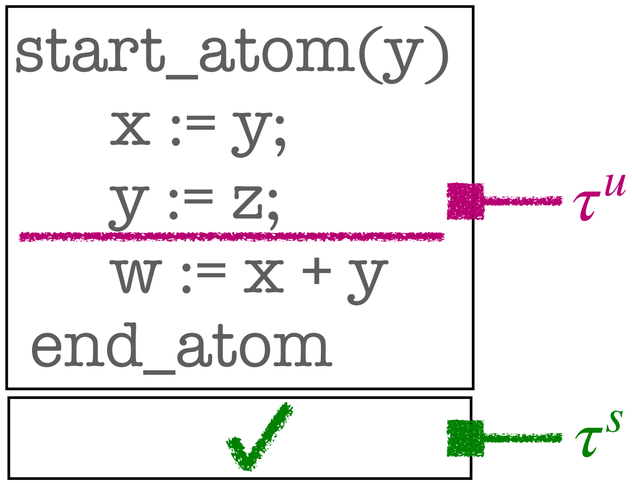
Typing judgments

basic types $T := \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$

access qualifiers $qAcc := \texttt{CK} \mid \texttt{RD} \mid \texttt{MtWt} \mid \texttt{Wtn}$

unstable types $\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid v_t$

stable types $\tau^s := \uparrow \tau^u \mid$  $\rightsquigarrow \tau^s$



Unstable typing judgment

 $\mid$ _____ $\vdash$ x := y : C

Energy buffer

Command is well-typed

Stable typing judgment

$\vdash$

_____ _____

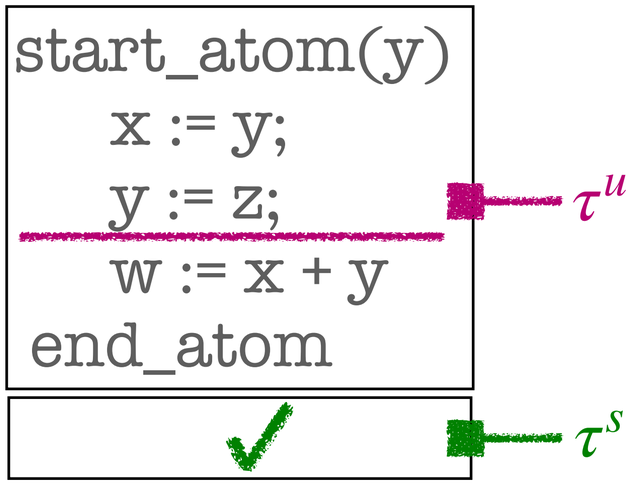
Typing judgments

basic types $T := \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$

access qualifiers $qAcc := \texttt{CK} \mid \texttt{RD} \mid \texttt{MtWt} \mid \texttt{Wtn}$

unstable types $\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid \nu_t$

stable types $\tau^s := \uparrow \tau^u \mid$  $\rightsquigarrow \tau^s$



Ω : NV

Σ : V

Unstable typing judgment

$x : \tau^s$ $y : \tau^u$

 $\mid \Omega \mid \Sigma \vdash \texttt{x} := \texttt{y} : \texttt{C}$

Energy buffer

stable/ unstable typing contexts

Command is well-typed

Stable typing judgment

$\vdash$

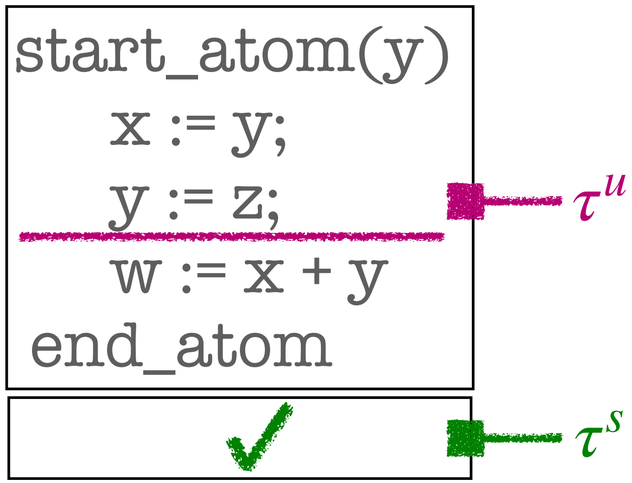
Typing judgments

basic types $T := \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$

access qualifiers $qAcc := \texttt{CK} \mid \texttt{RD} \mid \texttt{MtWt} \mid \texttt{Wtn}$

unstable types $\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid v_t$

stable types $\tau^s := \uparrow \tau^u \mid$  $\rightsquigarrow \tau^s$



Unstable typing judgment

$x : \tau^s$ $y : \tau^u$

 $\mid \Omega \mid \Sigma \vdash \texttt{x := y : C} \dashv \Omega'$

Energy buffer

stable/ unstable typing contexts

Command is well-typed

Post-context

Stable typing judgment

$\vdash$

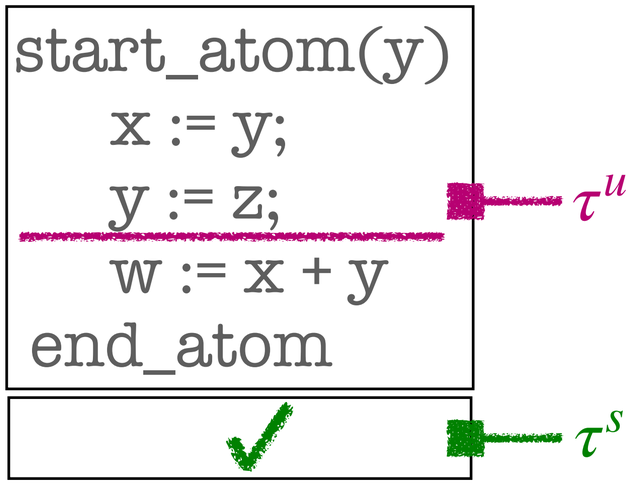
Typing judgments

basic types $T := \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$

access qualifiers $qAcc := \texttt{CK} \mid \texttt{RD} \mid \texttt{MtWt} \mid \texttt{Wtn}$

unstable types $\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid v_t$

stable types $\tau^s := \uparrow \tau^u \mid$  $\rightsquigarrow \tau^s$



Unstable typing judgment

$x : \tau^s$ $y : \tau^u$

 $\mid \Omega \mid \Sigma \vdash \texttt{x := y} : \texttt{C} \dashv \Omega'$

Energy buffer

stable/ unstable typing contexts

Command is well-typed

Post-context

Stable typing judgment

$x : \tau^s$

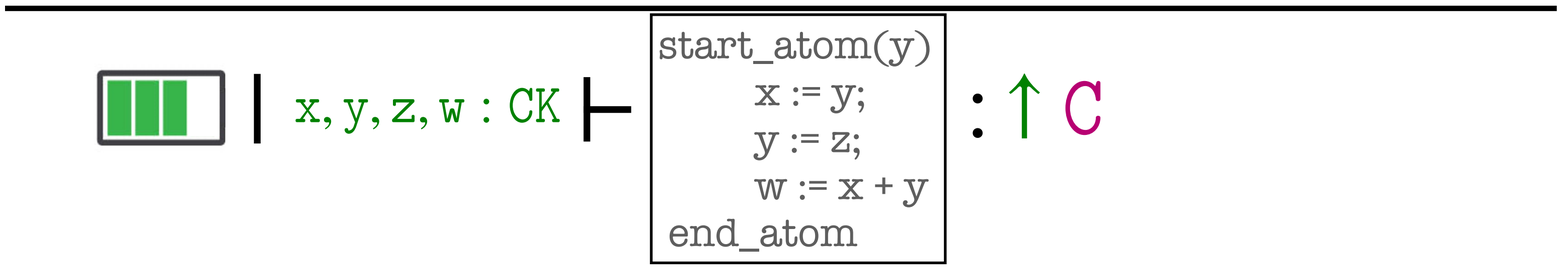
 $\mid \Omega \vdash$

```
start_atom(y)
    x := y;
    y := z;
    w := x + y
end_atom
```

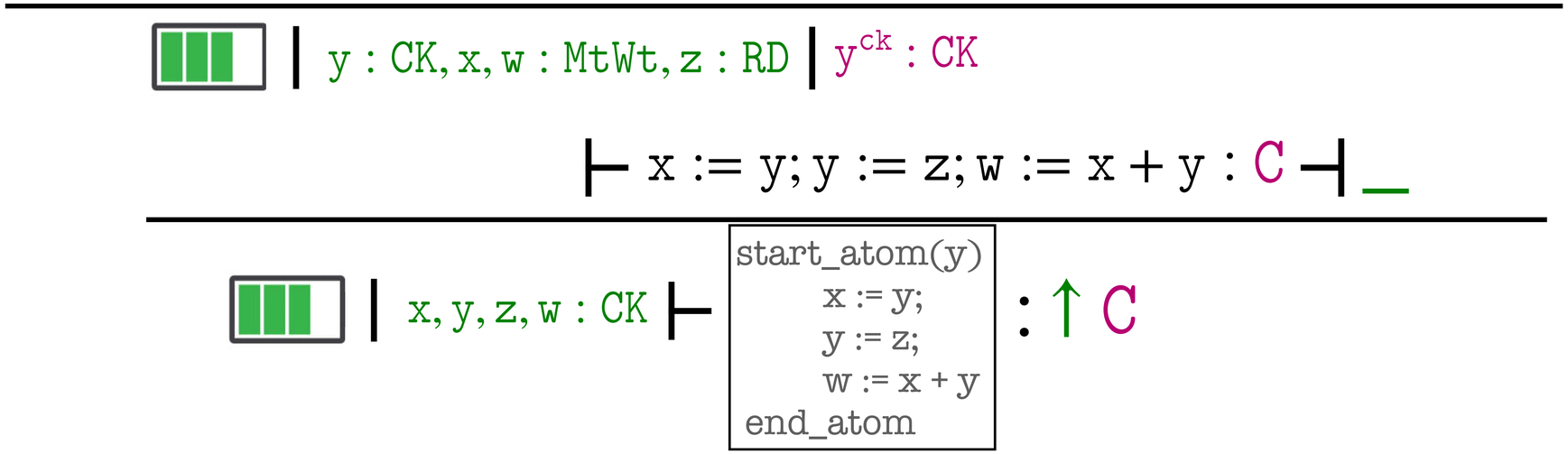
$: \uparrow \texttt{C}$

stable computation independent of unstable values!

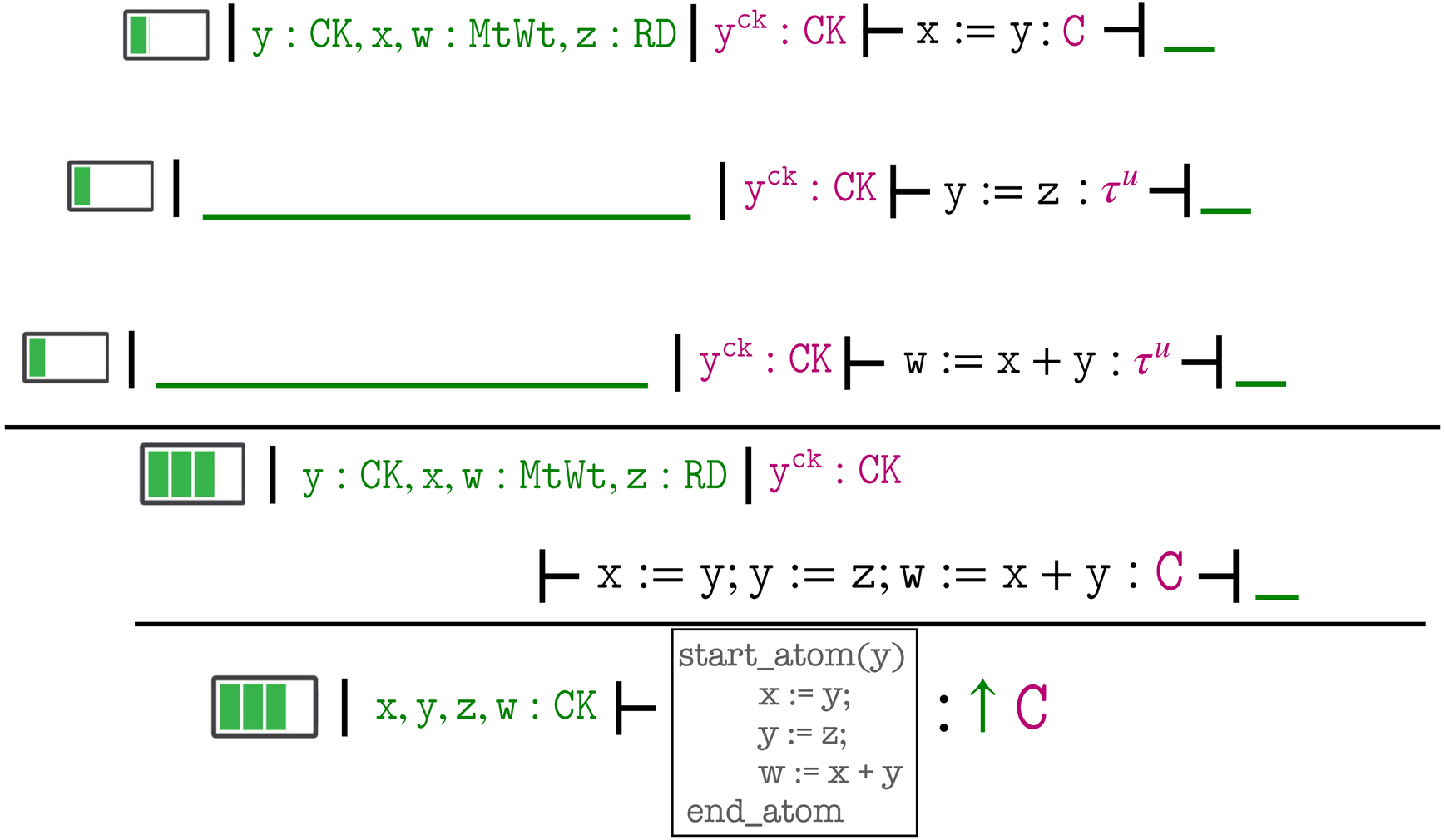
Typing a program bottom-up



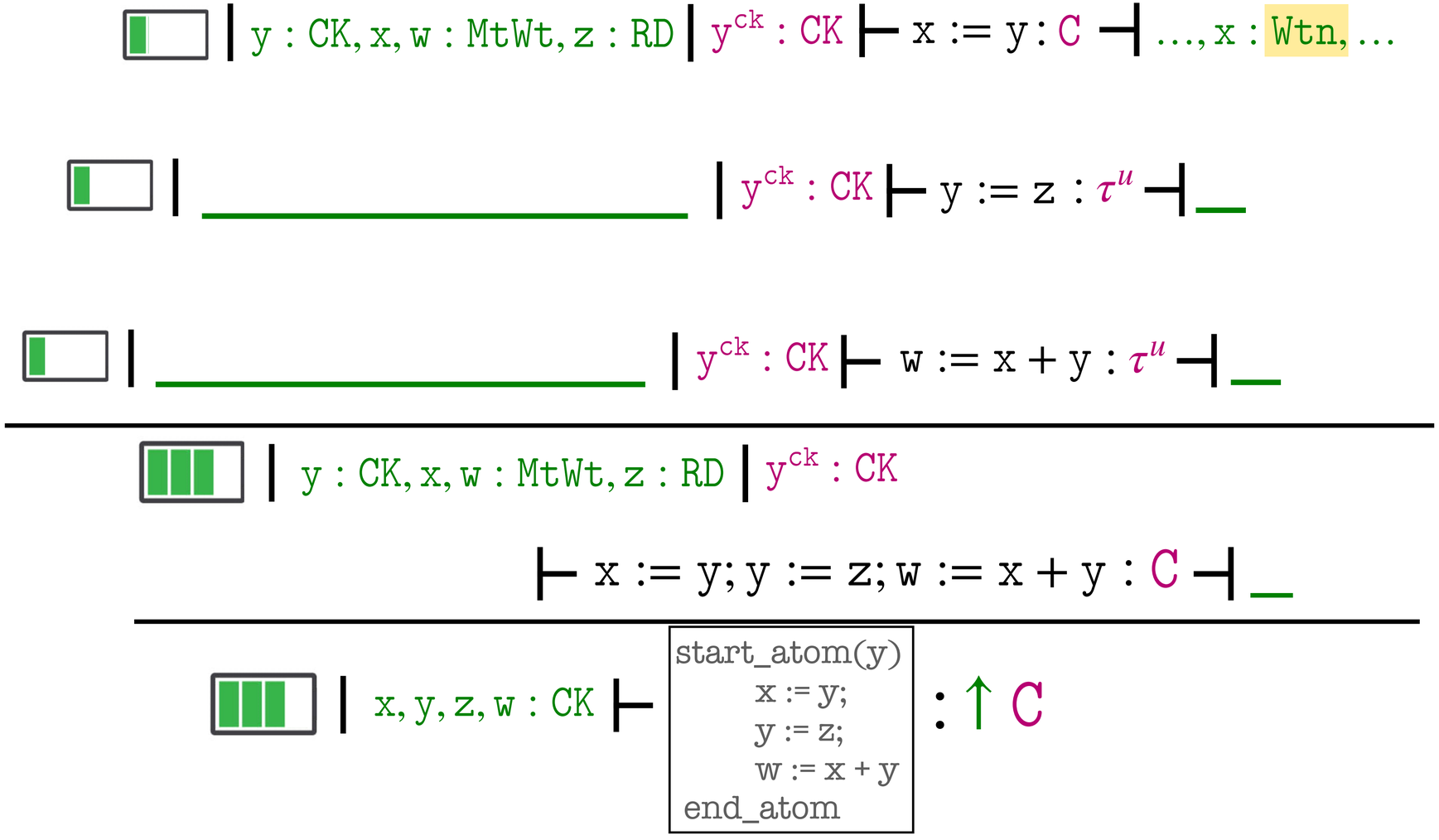
Typing a program bottom-up



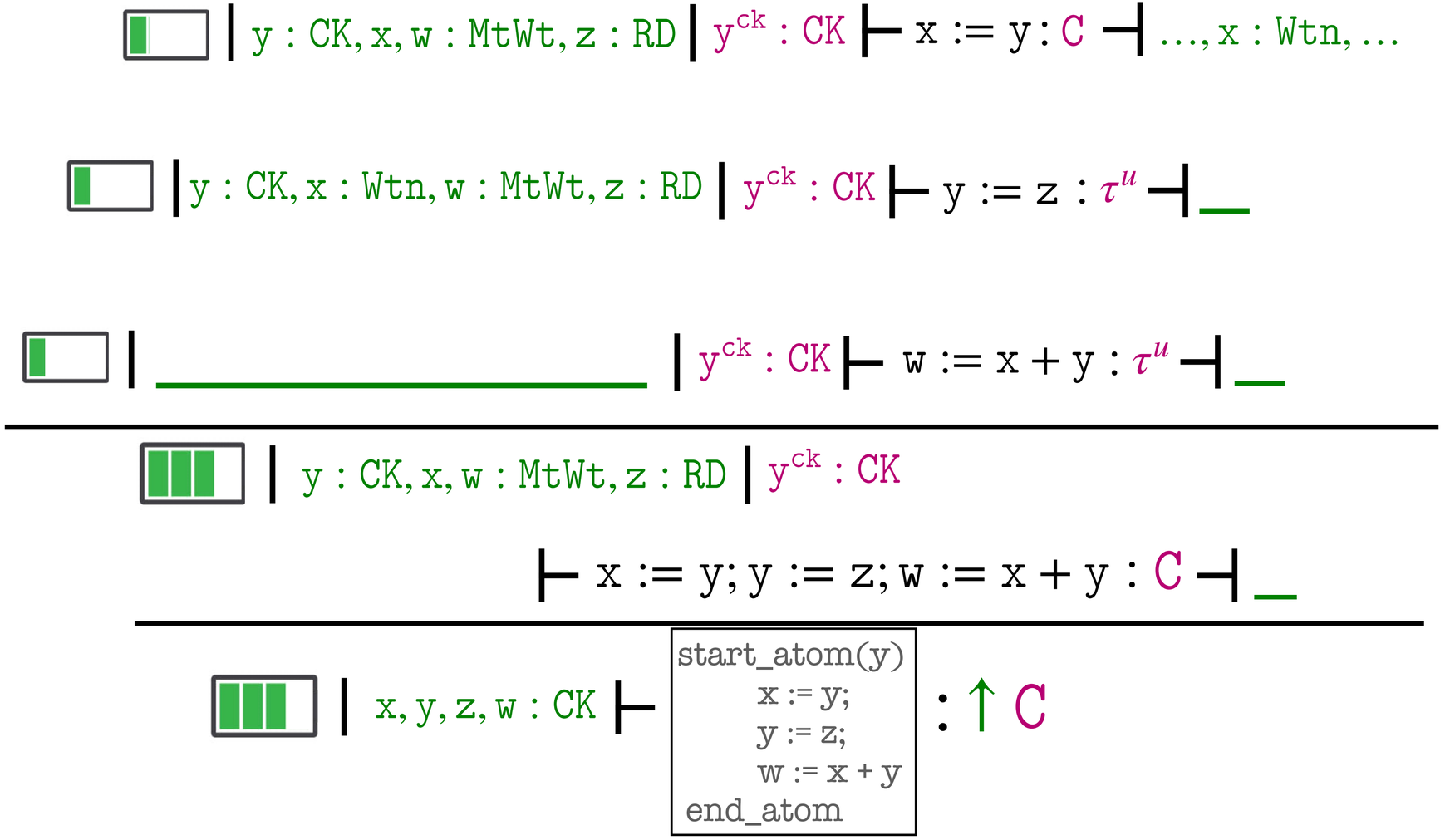
Typing a program bottom-up



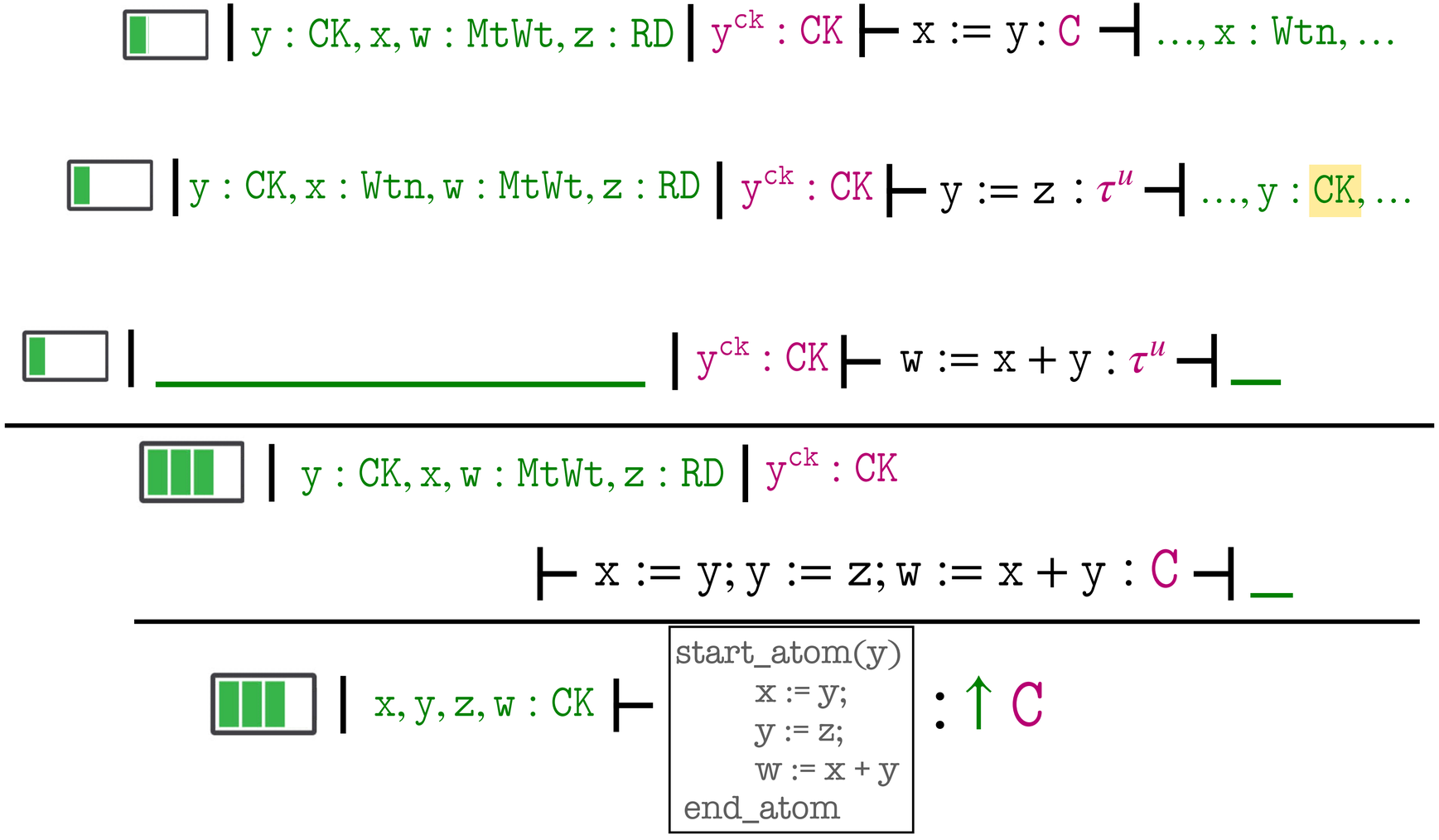
Typing a program bottom-up



Typing a program bottom-up



Typing a program bottom-up



Typing a program bottom-up

$$\mid \mathtt{y} : \mathtt{CK}, \mathtt{x}, \mathtt{w} : \mathtt{MtWt}, \mathtt{z} : \mathtt{RD} \mid \mathtt{y}^{\mathtt{ck}} : \mathtt{CK} \vdash \mathtt{x} := \mathtt{y} : \mathtt{C} \dashv \ldots, \mathtt{x} : \mathtt{Wtn}, \ldots$$

$$\mid \mathtt{y} : \mathtt{CK}, \mathtt{x} : \mathtt{Wtn}, \mathtt{w} : \mathtt{MtWt}, \mathtt{z} : \mathtt{RD} \mid \mathtt{y}^{\mathtt{ck}} : \mathtt{CK} \vdash \mathtt{y} := \mathtt{z} : \tau^{u} \dashv \ldots, \mathtt{y} : \mathtt{CK}, \ldots$$

$$\mid \mathtt{y} : \mathtt{CK}, \mathtt{x} : \mathtt{Wtn}, \mathtt{w} : \mathtt{MtWt}, \mathtt{z} : \mathtt{RD} \mid \mathtt{y}^{\mathtt{ck}} : \mathtt{CK} \vdash \mathtt{w} := \mathtt{x} + \mathtt{y} : \tau^{u} \dashv \_$$

$$\mid \mathtt{y} : \mathtt{CK}, \mathtt{x}, \mathtt{w} : \mathtt{MtWt}, \mathtt{z} : \mathtt{RD} \mid \mathtt{y}^{\mathtt{ck}} : \mathtt{CK}$$

$$\vdash \mathtt{x} := \mathtt{y}; \mathtt{y} := \mathtt{z}; \mathtt{w} := \mathtt{x} + \mathtt{y} : \mathtt{C} \dashv \_$$

$$\mid \mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{w} : \mathtt{CK} \vdash$$

```
start_atom(y)
    x := y;
    y := z;
    w := x + y
end_atom
```

$$: \uparrow \mathtt{C}$$

Typing a program bottom-up

$$\mid \mathtt{y} : \mathtt{CK}, \mathtt{x}, \mathtt{w} : \mathtt{MtWt}, \mathtt{z} : \mathtt{RD} \mid \mathtt{y}^{\mathtt{ck}} : \mathtt{CK} \vdash \mathtt{x} := \mathtt{y} : \mathtt{C} \dashv \ldots, \mathtt{x} : \mathtt{Wtn}, \ldots$$

$$\mid \mathtt{y} : \mathtt{CK}, \mathtt{x} : \mathtt{Wtn}, \mathtt{w} : \mathtt{MtWt}, \mathtt{z} : \mathtt{RD} \mid \mathtt{y}^{\mathtt{ck}} : \mathtt{CK} \vdash \mathtt{y} := \mathtt{z} : \tau^{u} \dashv \ldots, \mathtt{y} : \mathtt{CK}, \ldots$$

$$\mid \mathtt{y} : \mathtt{CK}, \mathtt{x} : \mathtt{Wtn}, \mathtt{w} : \mathtt{MtWt}, \mathtt{z} : \mathtt{RD} \mid \mathtt{y}^{\mathtt{ck}} : \mathtt{CK} \vdash \mathtt{w} := \mathtt{x} + \mathtt{y} : \tau^{u} \dashv \mathtt{y} : \mathtt{CK}, \mathtt{x} : \mathtt{Wtn}, \mathtt{w} : \mathtt{Wtn}, \mathtt{z} : \mathtt{RD}$$

$$\mid \mathtt{y} : \mathtt{CK}, \mathtt{x}, \mathtt{w} : \mathtt{MtWt}, \mathtt{z} : \mathtt{RD} \mid \mathtt{y}^{\mathtt{ck}} : \mathtt{CK}$$

$$\vdash \mathtt{x} := \mathtt{y}; \mathtt{y} := \mathtt{z}; \mathtt{w} := \mathtt{x} + \mathtt{y} : \mathtt{C} \dashv \_$$

$$\mid \mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{w} : \mathtt{CK} \vdash$$

```
start_atom(y)
    x := y;
    y := z;
    w := x + y
end_atom
```

$$: \uparrow \mathtt{C}$$

Typing a program bottom-up

$$\mid \mathtt{y} : \mathtt{CK}, \mathtt{x}, \mathtt{w} : \mathtt{MtWt}, \mathtt{z} : \mathtt{RD} \mid \mathtt{y}^{\mathtt{ck}} : \mathtt{CK} \vdash \mathtt{x} := \mathtt{y} : \mathtt{C} \dashv \ldots, \mathtt{x} : \mathtt{Wtn}, \ldots$$

$$\mid \mathtt{y} : \mathtt{CK}, \mathtt{x} : \mathtt{Wtn}, \mathtt{w} : \mathtt{MtWt}, \mathtt{z} : \mathtt{RD} \mid \mathtt{y}^{\mathtt{ck}} : \mathtt{CK} \vdash \mathtt{y} := \mathtt{z} : \tau^{u} \dashv \ldots, \mathtt{y} : \mathtt{CK}, \ldots$$

$$\mid \mathtt{y} : \mathtt{CK}, \mathtt{x} : \mathtt{Wtn}, \mathtt{w} : \mathtt{MtWt}, \mathtt{z} : \mathtt{RD} \mid \mathtt{y}^{\mathtt{ck}} : \mathtt{CK} \vdash \mathtt{w} := \mathtt{x} + \mathtt{y} : \tau^{u} \dashv \mathtt{y} : \mathtt{CK}, \mathtt{x} : \mathtt{Wtn}, \mathtt{w} : \mathtt{Wtn}, \mathtt{z} : \mathtt{RD}$$

$$\mid \mathtt{y} : \mathtt{CK}, \mathtt{x}, \mathtt{w} : \mathtt{MtWt}, \mathtt{z} : \mathtt{RD} \mid \mathtt{y}^{\mathtt{ck}} : \mathtt{CK}$$

$$\vdash \mathtt{x} := \mathtt{y}; \mathtt{y} := \mathtt{z}; \mathtt{w} := \mathtt{x} + \mathtt{y} : \mathtt{C} \dashv \mathtt{y} : \mathtt{CK}, \mathtt{x}, \mathtt{w} : \mathtt{Wtn}, \mathtt{z} : \mathtt{RD}$$

$$\mid \mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{w} : \mathtt{CK} \vdash$$

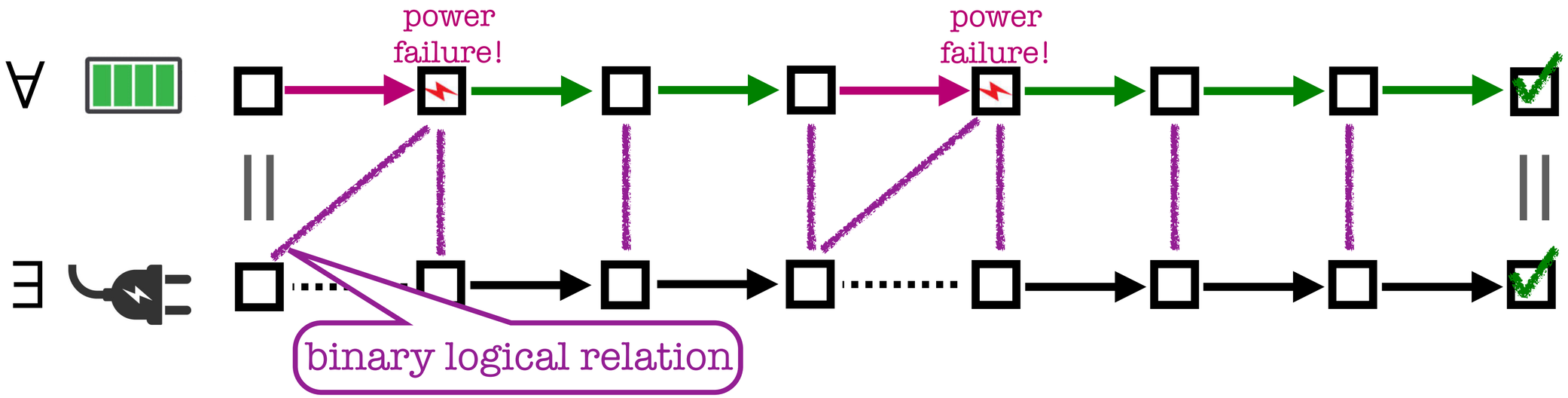
```
start_atom(y)
    x := y;
    y := z;
    w := x + y
end_atom
```

$$: \uparrow \mathtt{C}$$

no remaining MtWt!

Proving Correctness

Every intermittent execution of a **well-typed** program can be simulated by its continuous execution.



Binary Logical Relation

Every intermittent execution of a **well-typed** program can be simulated by its continuous execution.

Term Relation

where c_1 is...

$\texttt{x} := \texttt{y}$

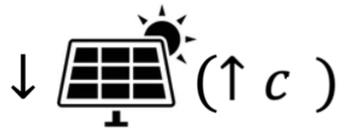
$\texttt{if}\ v\ \texttt{then}\ c\ \texttt{else}\ c'$

$\texttt{let}\ x = e\ \texttt{in}\ c$

$c; c'$

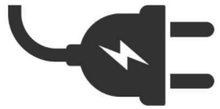
Value Relation

where v_1 is... n true false skip



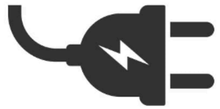
↑ c

Logical relation for atomic regions

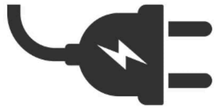
$(\ $  $\ \mid \mathsf{NV}_1 \mid \mathsf{V} \mid c,$  $\mid \mathsf{NV}_2 \mid \mathsf{V} \mid c) \in \mathscr{E}[\![\mathbf{C}_{Unit}]\!]^0$

no remaining attempts

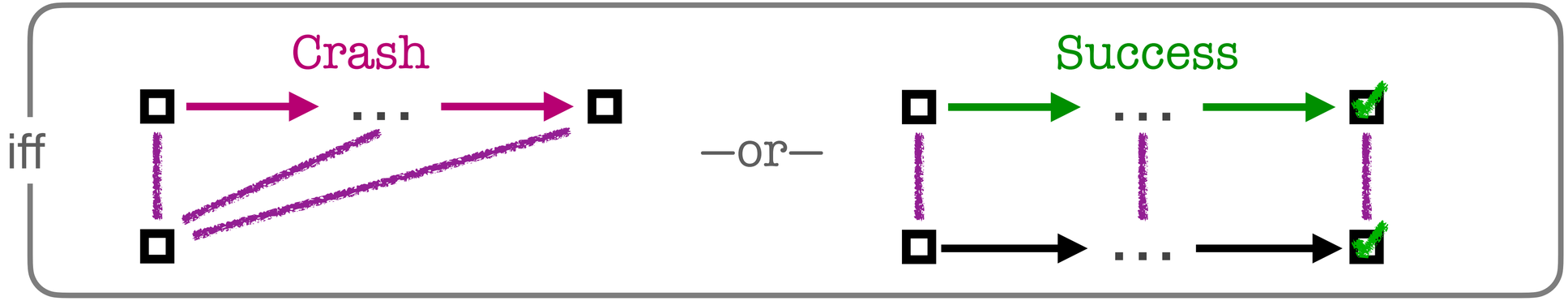
Logical relation for atomic regions


$\mid \mathrm{NV}_1 \mid \mathrm{V} \mid c,$

$\mid \mathrm{NV}_2 \mid \mathrm{V} \mid c) \in \mathscr{E}[\![\mathrm{C}_{Unit}]\!]^0$

no remaining attempts


$\mid \mathrm{NV}_1 \mid \mathrm{V} \mid c,$

$\mid \mathrm{NV}_2 \mid \mathrm{V} \mid c) \in \mathscr{E}[\![\mathrm{C}_{Unit}]\!]^{k+1}$

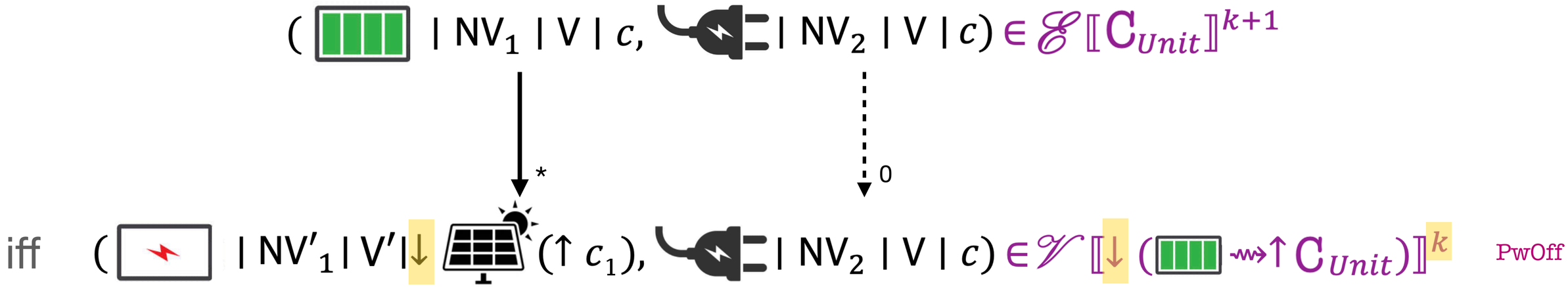
at least one remaining attempt



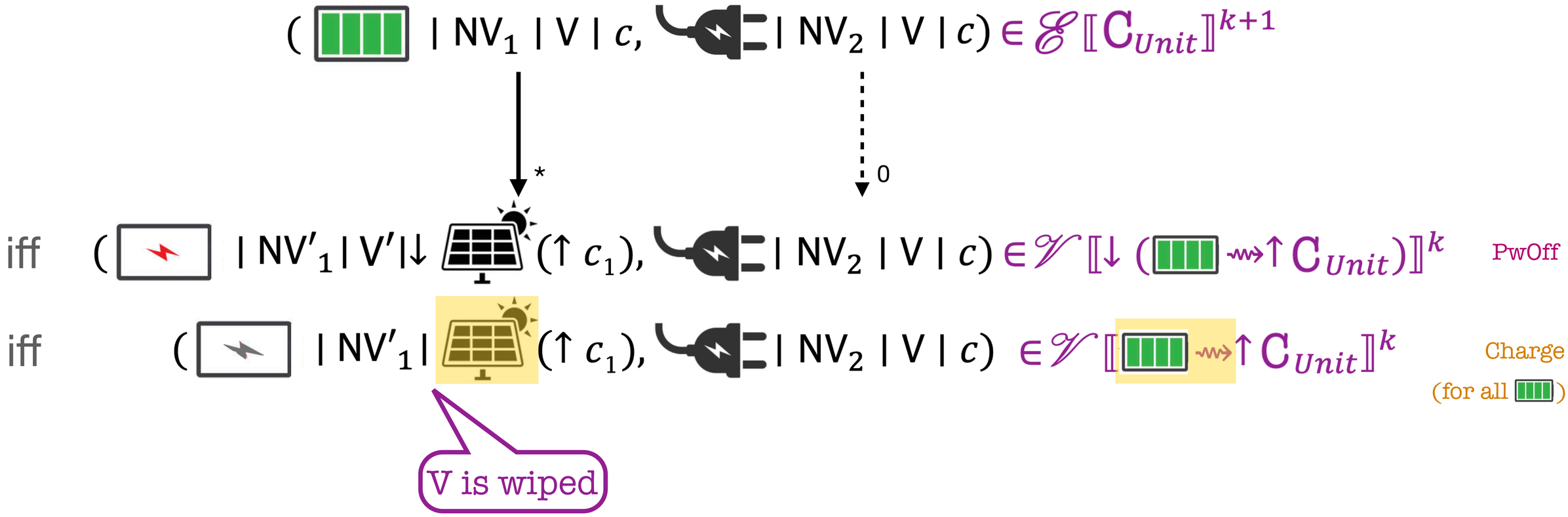
If there *is* a power failure…

$$(\ \text{  }\ \mid \mathrm{NV}_1 \mid \mathrm{V} \mid c,\ \text{  }\ \mid \mathrm{NV}_2 \mid \mathrm{V} \mid c) \in \mathscr{E}[\![\mathbf{C}_{Unit}]\!]^{k+1}$$

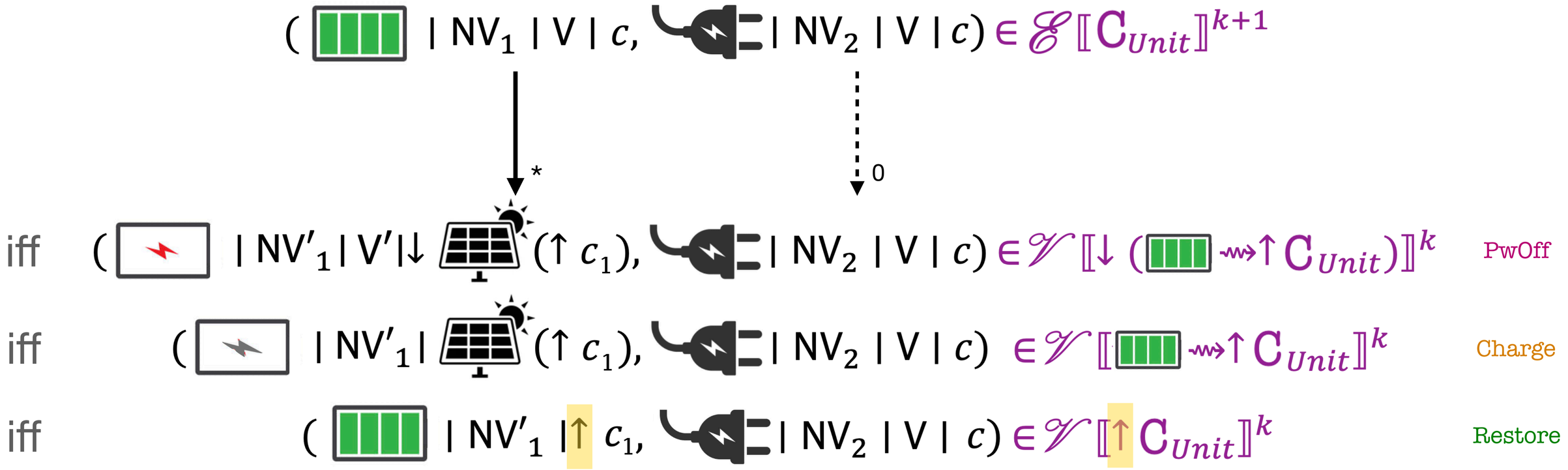
Device powers off



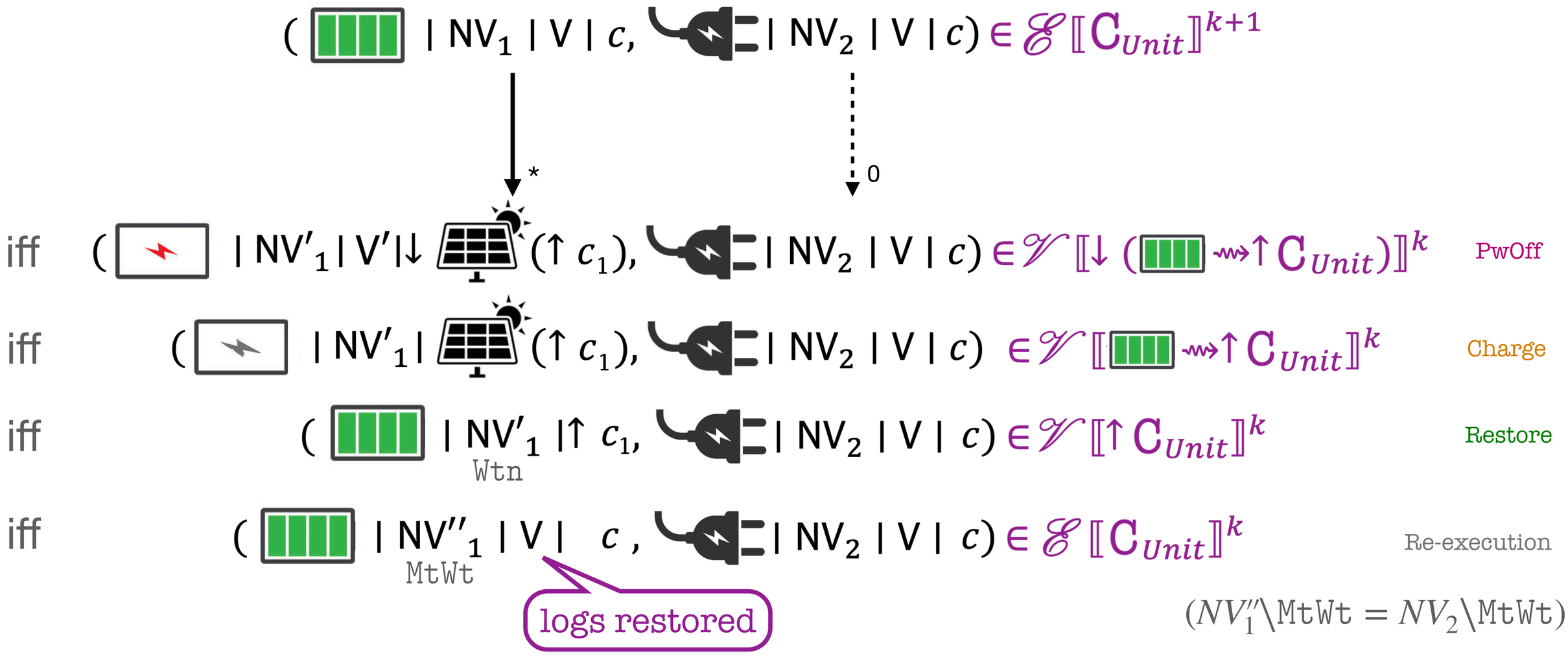
Energy buffer charges



Execution state is restored



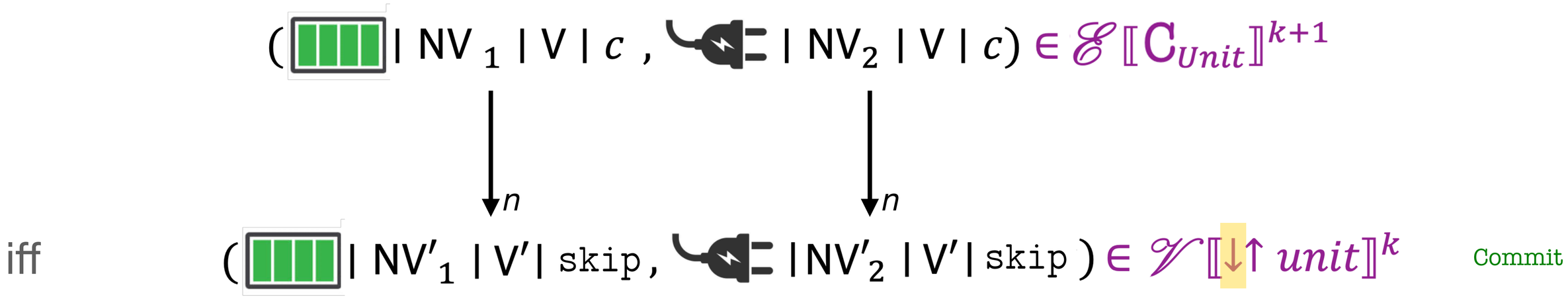
Program re-executes



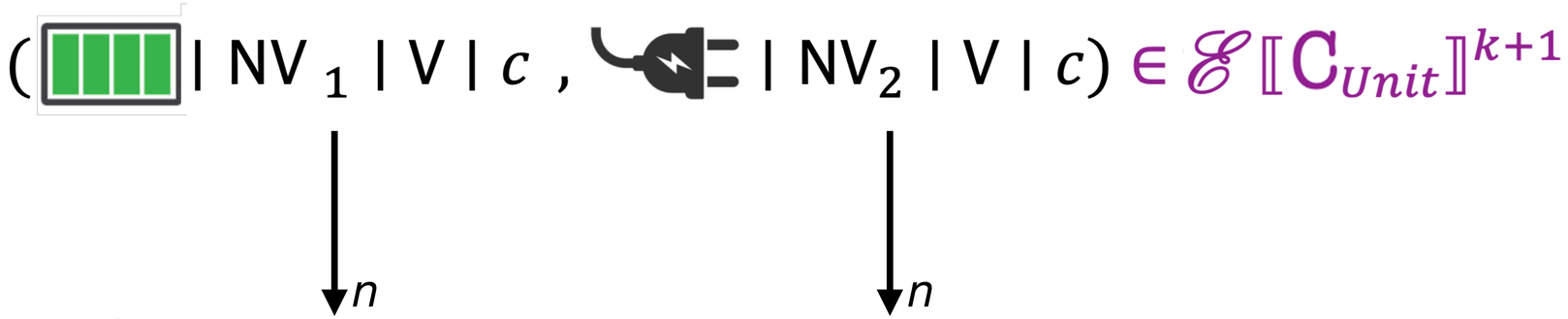
If there is *no* power failure…

$$(\text{  } \mid \mathrm{NV}_1 \mid \mathrm{V} \mid c \;,\; \text{  } \mid \mathrm{NV}_2 \mid \mathrm{V} \mid c) \in \mathscr{E}[\![\mathbb{C}_{Unit}]\!]^{k+1}$$

Stable values are committed



We are done!



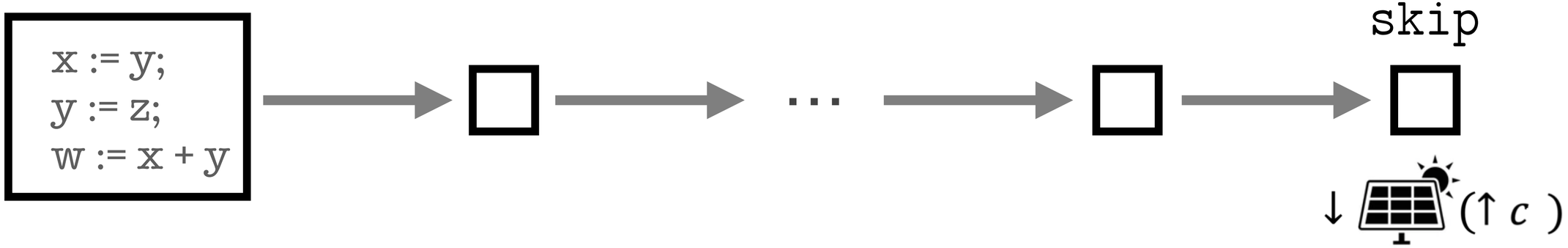
iff  Commit

iff  Success!

$(NV''_1 = NV''_2)$

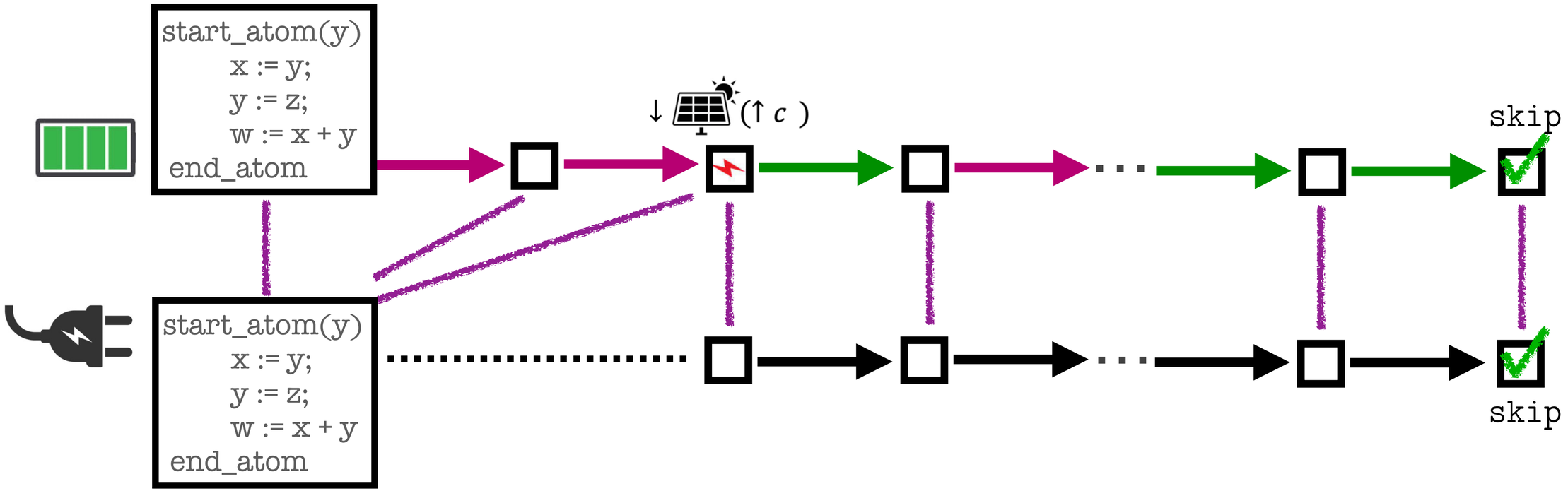
all MtWt variables are written (Wtn)
to by last execution

Progress and Preservation



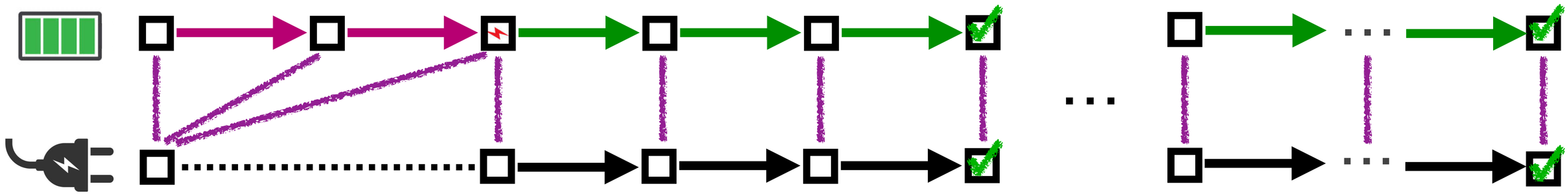
Well-typed commands don’t get stuck and stay well-typed

Fundamental Theorem of Logical Relation



Well-typed code regions are self-related

Adequacy



Every intermittent execution of a well-typed program can be simulated by its continuous execution

In summary

- first logical interpretation of key operations of intermittent execution
- a novel logical relation to characterize correctness
- supports and reasoning for JIT and atomic regions (see thesis!)

In summary

- first logical interpretation of key operations of intermittent execution
- a novel logical relation to characterize correctness
- supports and reasoning for JIT and atomic regions (see thesis!)

Limitations of crash types

- tied specifically to redo-logging, which is not that common

In summary

- first logical interpretation of key operations of intermittent execution
- a novel logical relation to characterize correctness
- supports and reasoning for JIT and atomic regions (see thesis!)

Limitations of crash types

- tied specifically to redo-logging, which is not that common
- support for I/O? flexible branching?

In summary

- first logical interpretation of key operations of intermittent execution
- a novel
- support

Can we use crash types to reason about common system features, e.g., **undo logging**, **I/O**, **flexible branching**?

Limitatio

- tied specifically to redo-logging, which is not that common
- support for I/O? flexible branching?

In summary

- first logical interpretation of key operations of intermittent execution
- a novel
- support

Yes we can!

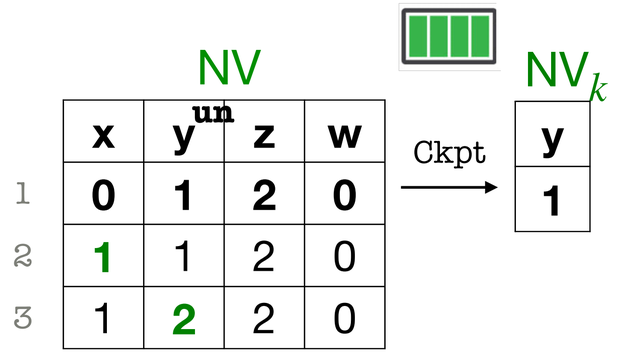
Limitatio

- tied specifically to redo-logging, which is not that common
- support for I/O? flexible branching?

Crash types for undo-logging

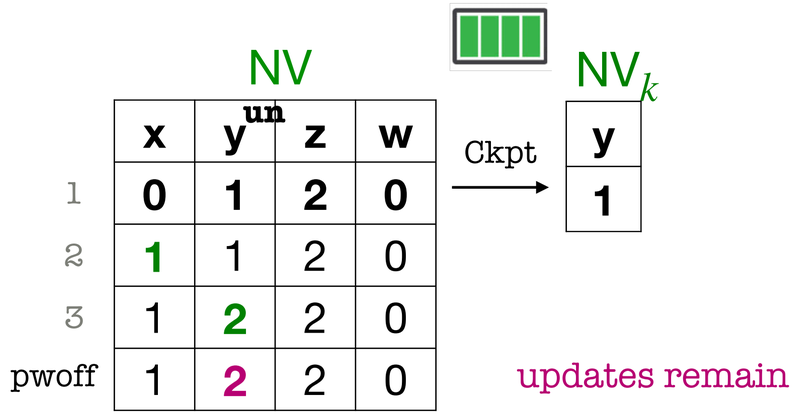
Undo-logging writes directly on NV

```
1 start_atom(_)
2       x := y;
3       y := z;
4       w := x + y
5  end_atom
```



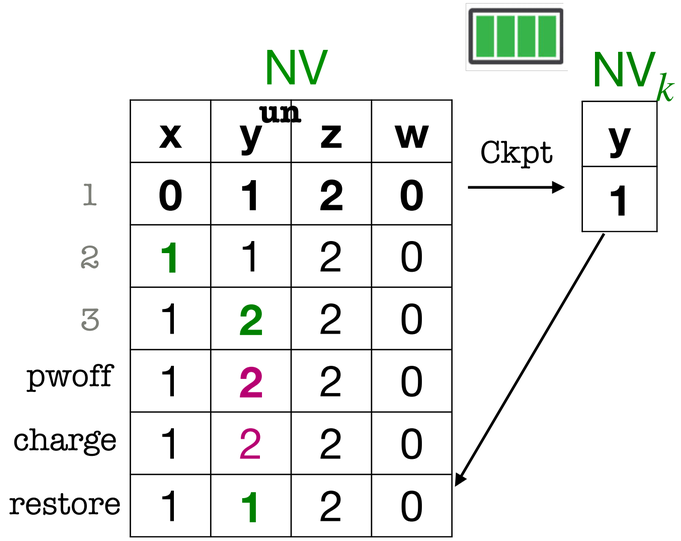
Undo-logging writes directly on NV

```
1 start_atom(_)
2       x := y;
3       y := z;
4       w := x + y
5  end_atom
```



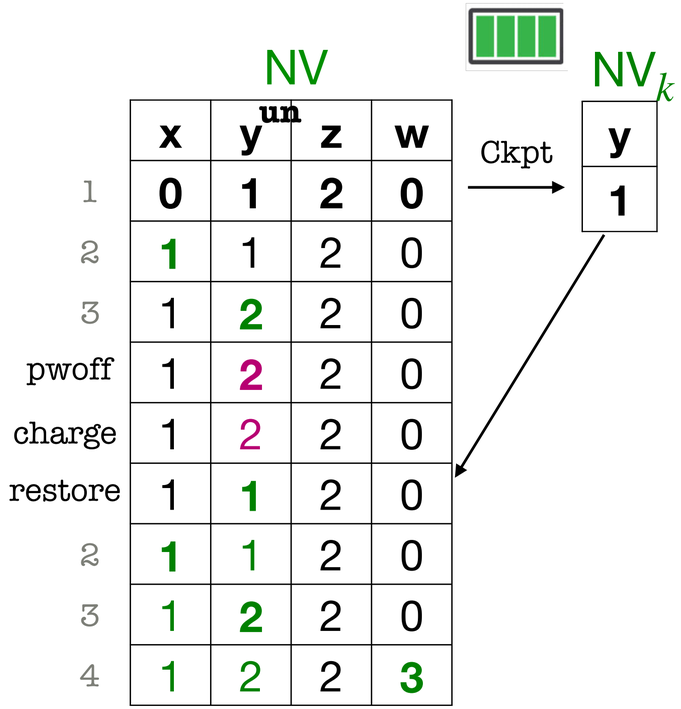
Undo-logging writes directly on NV

```
1 start_atom(_)
2       x := y;
3       y := z;
4       w := x + y
5  end_atom
```



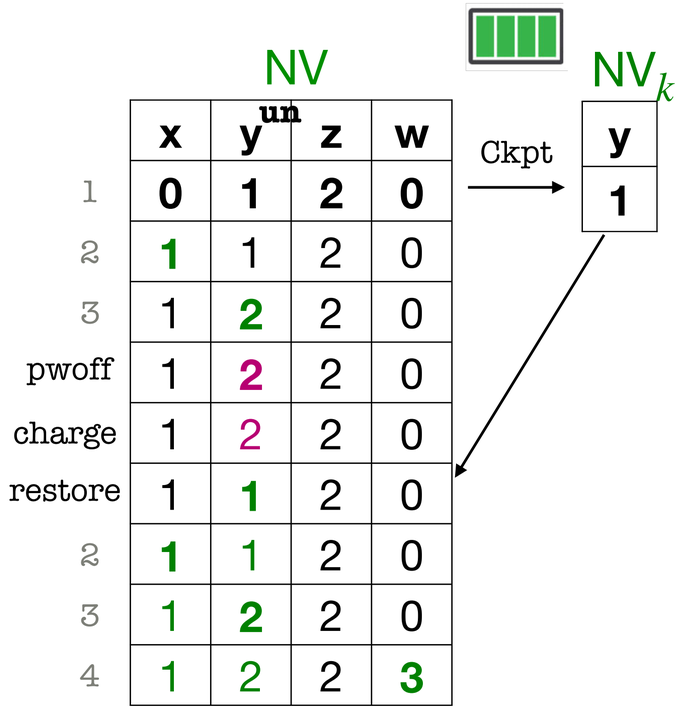
Undo-logging writes directly on NV

```
1 start_atom(_)
2       x := y;
3       y := z;
4       w := x + y
5  end_atom
```



Undo-logging writes directly on NV

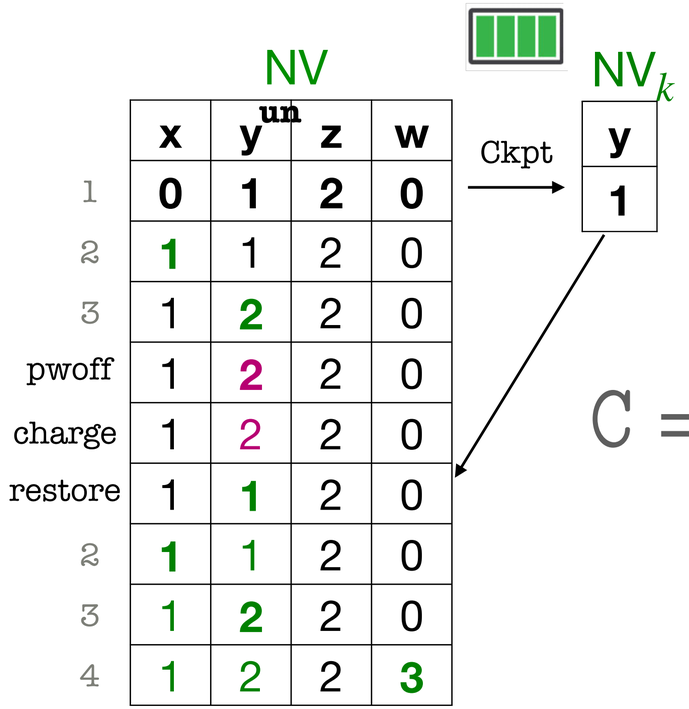
```
1 start_atom(_)
2       x := y;
3       y := z;
4       w := x + y
5  end_atom
```



nonvolatile state is unstable through power failure and restore!

Undo-logging crash type

```
1 start_atom(_)
2       x := y;
3       y := z;
4       w := x + y
5  end_atom
```



nonvolatile state is unstable through power failure and restore!

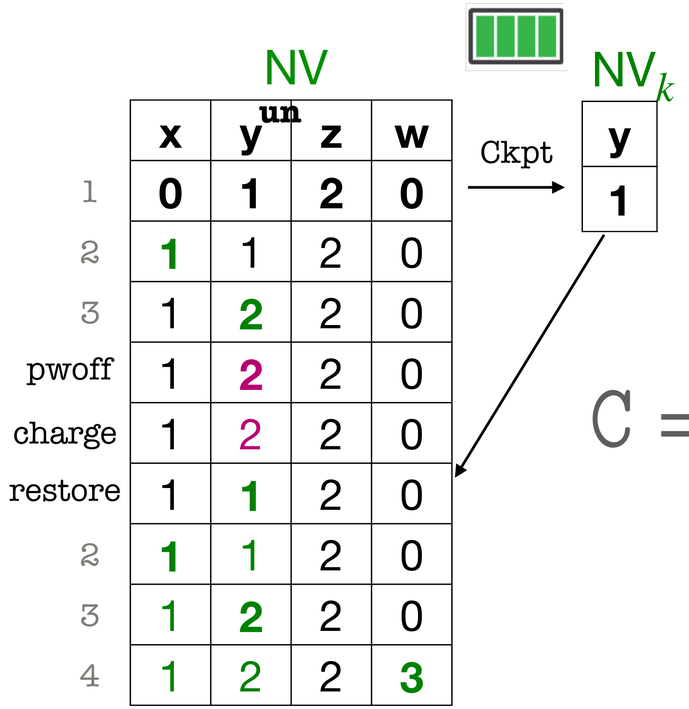
$$C = \downarrow \uparrow \mathtt{unit} \vee \underline{\qquad}$$

Successful execution

Failed execution

Undo-logging crash type

```
1 start_atom(_)
2       x := y;
3       y := z;
4       w := x + y
5  end_atom
```



nonvolatile state is unstable through power failure and restore!

$$C = \downarrow\uparrow \mathtt{unit} \vee (\rightsquigarrow \_\_\_)$$

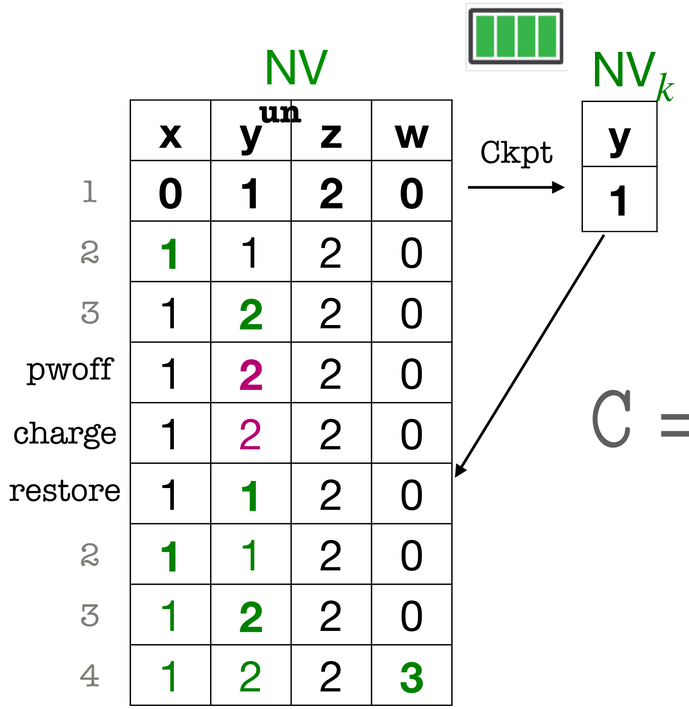
Charge

Successful execution

Failed execution

Undo-logging crash type

```
1 start_atom(_)
2       x := y;
3       y := z;
4       w := x + y
5  end_atom
```



nonvolatile state is unstable through power failure and restore!

$$C = \downarrow\uparrow \mathtt{unit} \vee (\rightsquigarrow \downarrow\uparrow \_)$$

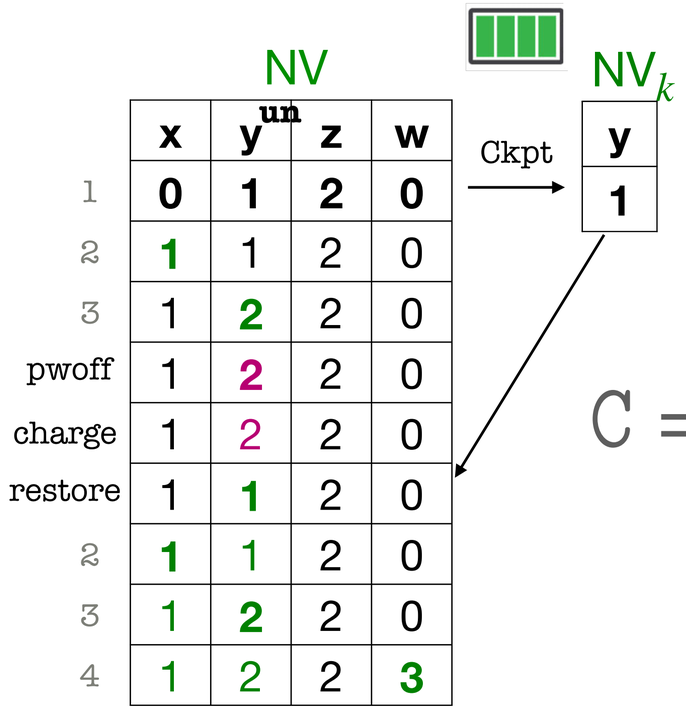
Restore

Successful execution

Failed execution

Undo-logging crash type

```
1 start_atom(_)
2       x := y;
3       y := z;
4       w := x + y
5  end_atom
```



nonvolatile state is unstable through power failure and restore!

$$C = \downarrow\uparrow \mathtt{unit} \vee (\rightsquigarrow \downarrow\uparrow C)$$

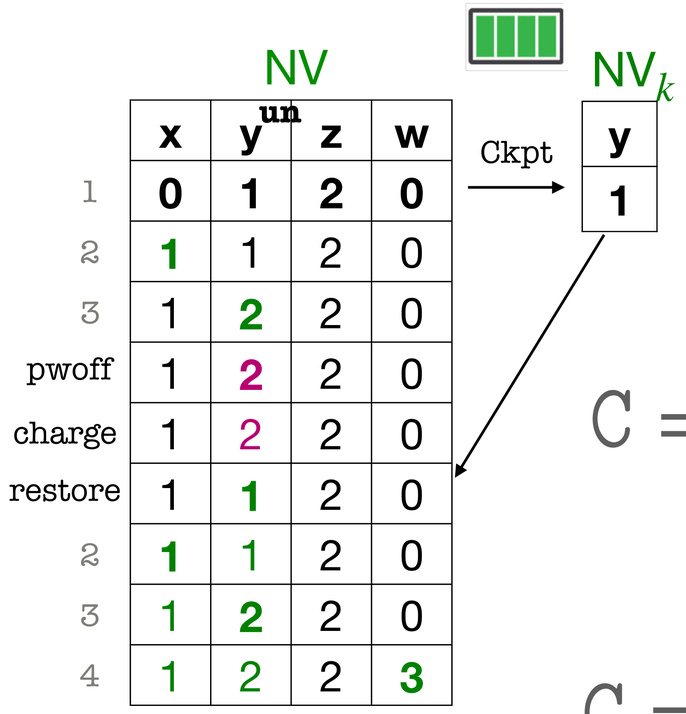
Re-execute

Successful execution

Failed execution

Undo-logging crash type

```
1 start_atom(_)
2       x := y;
3       y := z;
4       w := x + y
5  end_atom
```



nonvolatile state is unstable through power failure and restore!

restore NV

$$C = \downarrow\uparrow \mathtt{unit} \vee (\rightsquigarrow \downarrow\uparrow C)$$

undo-logging crash type

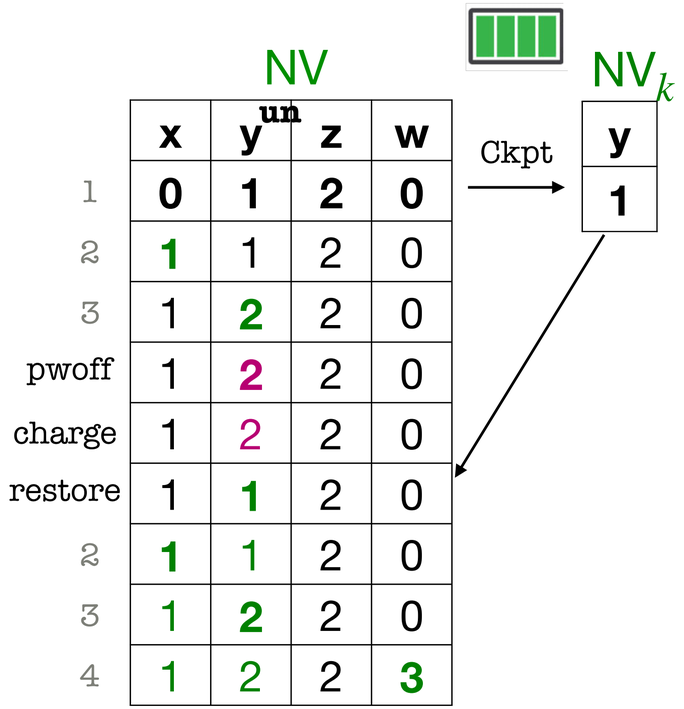
$$C = \downarrow\uparrow \mathtt{unit} \vee \downarrow(\rightsquigarrow \uparrow C)$$

redo-logging crash type

Undo-logging crash type

$$C = \downarrow\uparrow \texttt{unit} \vee (\blacksquare \rightsquigarrow \downarrow\uparrow C)$$

```
1 start_atom(_)
2       x := y;
3       y := z;
4       w := x + y
5  end_atom
```

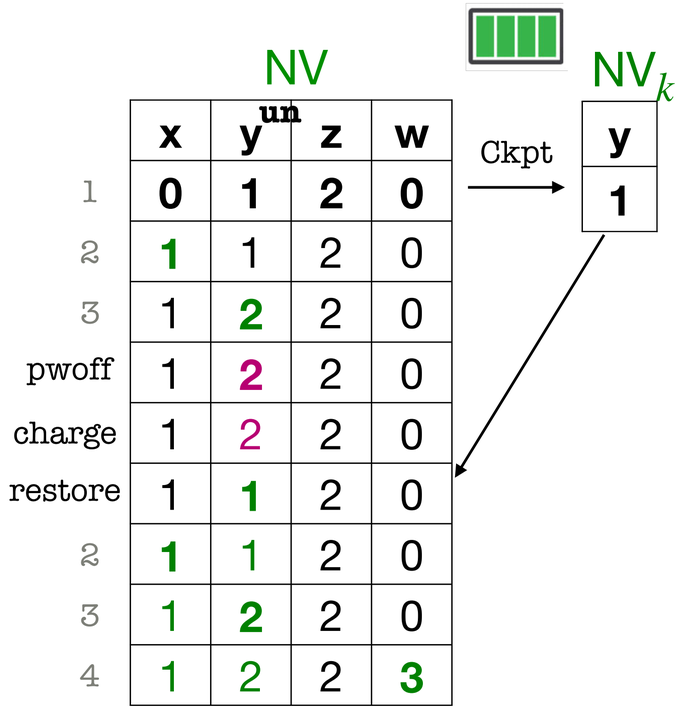


basic types	$T := \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$
access qualifiers	$qAcc := \texttt{CK} \mid \texttt{RD} \mid \texttt{MtWt} \mid \texttt{Wtn}$
unstable types	$\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid C \mid \blacksquare \rightsquigarrow \tau^u$
stable types	$\tau^s := \uparrow \tau^u$

Undo-logging crash type

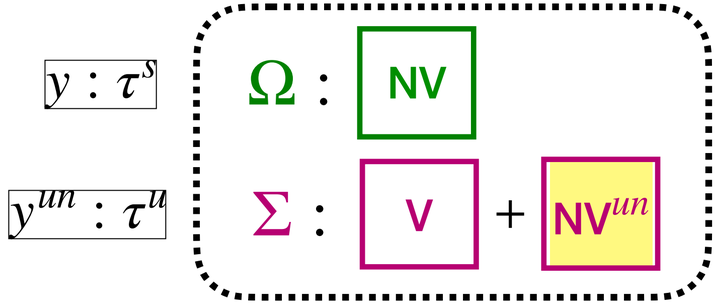
$$\mathtt{C} = \downarrow \uparrow \mathtt{unit} \vee (\blacksquare\!\!\blacksquare\!\!\blacksquare\!\!\blacksquare \rightsquigarrow \downarrow \uparrow \mathtt{C})$$

```
1 start_atom(_)
2       x := y;
3       y := z;
4       w := x + y
5  end_atom
```



basic types	$T := \mathtt{int} \mid \mathtt{bool} \mid \mathtt{unit}$
access qualifiers	$qAcc := \mathtt{CK} \mid \mathtt{RD} \mid \mathtt{MtWt} \mid \mathtt{Wtn}$
unstable types	$\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid \mathtt{C} \mid \blacksquare\!\!\blacksquare\!\!\square\!\!\square \rightsquigarrow \tau^u$
stable types	$\tau^s := \uparrow \tau^u$

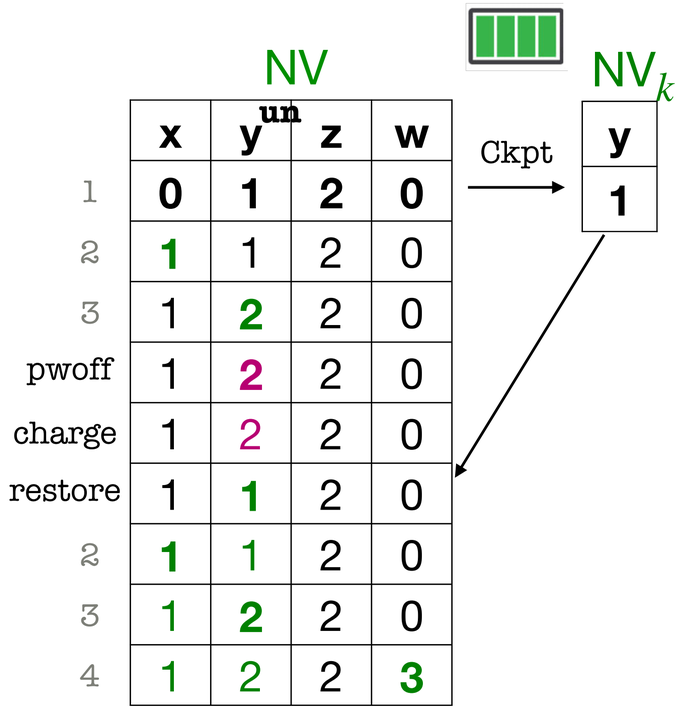
Σ types a portion of nonvolatile memory



Undo-logging crash type

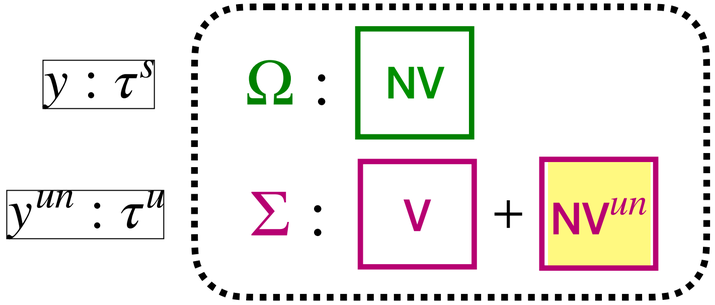
$$\mathtt{C} = \downarrow\uparrow \mathtt{unit} \vee (\blacksquare \rightsquigarrow \downarrow\uparrow \mathtt{C})$$

```
1 start_atom(_)
2       x := y;
3       y := z;
4       w := x + y
5  end_atom
```



basic types	$T := \mathtt{int} \mid \mathtt{bool} \mid \mathtt{unit}$
access qualifiers	$qAcc := \mathtt{CK} \mid \mathtt{RD} \mid \mathtt{MtWt} \mid \mathtt{Wtn}$
unstable types	$\tau^u := T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid \mathtt{C} \mid \blacksquare \rightsquigarrow \tau^u$
stable types	$\tau^s := \uparrow \tau^u$

Σ types a portion of nonvolatile memory



undo-logging operations also have interpretations in adjoint logic! (see thesis)

Metatheory

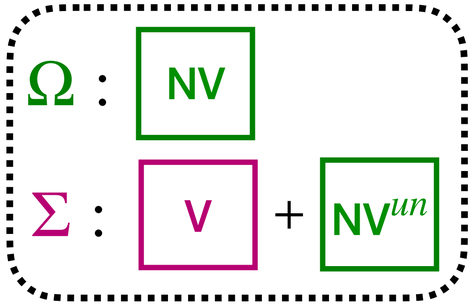
- progress and preservation
  - similar to redo-logging

- fundamental theorem and adequacy (future work!)
  - need to define a new logical relation for undo-logging

(|NV | V | c_1, | NV| V | c_2) $\in \mathscr{V}[\![\downarrow\uparrow \mathbb{C}_{unit}]\!]^m$ *iff* ???

In summary

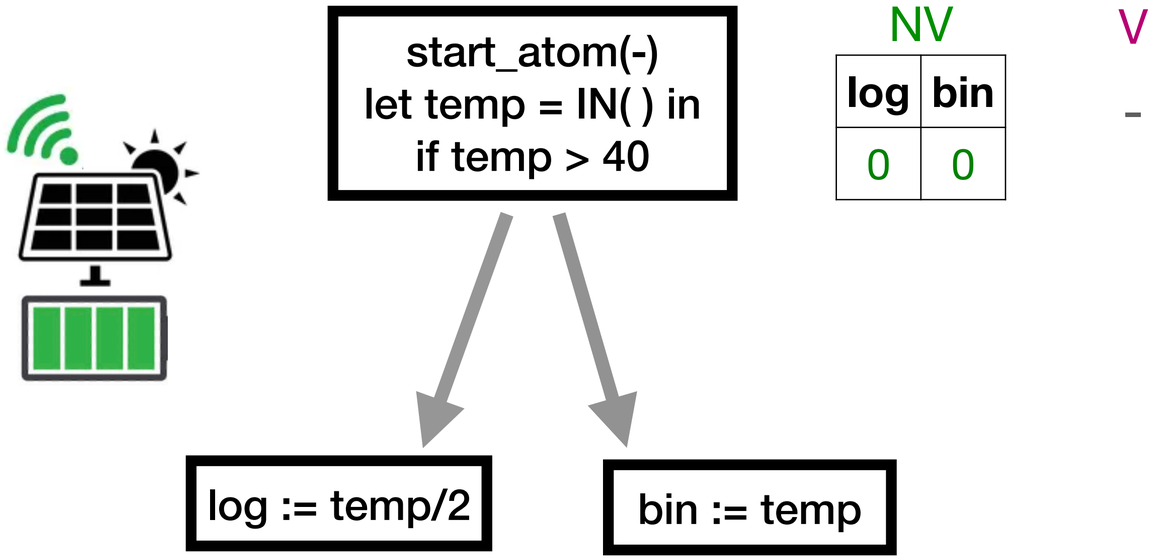
- an interpretation of crash types for undo-logging atomic regions
- unstable values can live in nonvolatile memory



Modal Crash Types w/ I/O, nonidempotence, flexible branching

Embedded applications rely on *non-deterministic* sensor inputs

input-dependent = tainted



NV

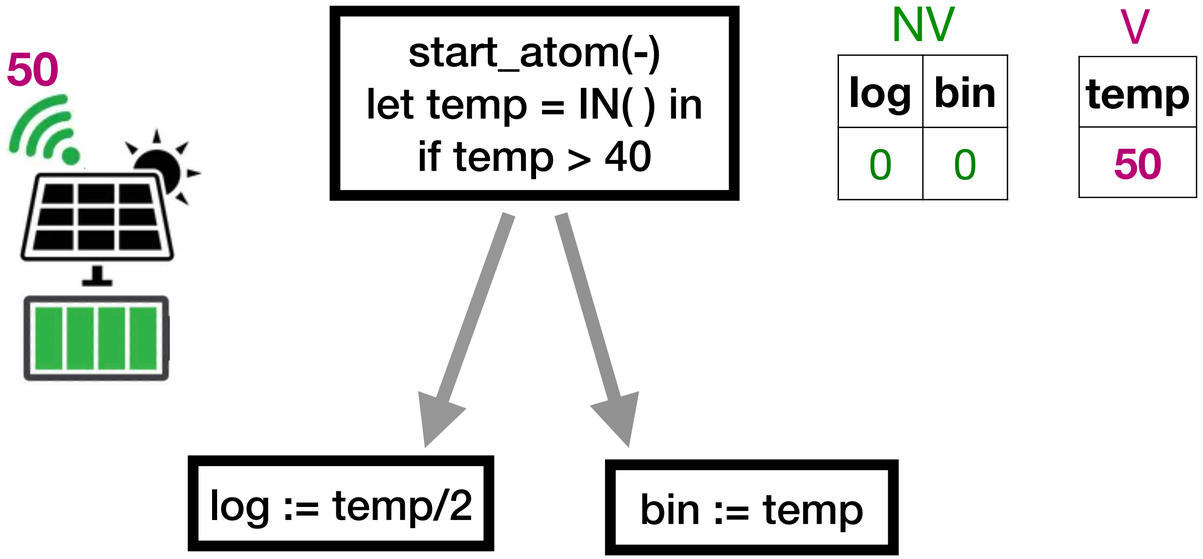
log	bin
0	0

V

-

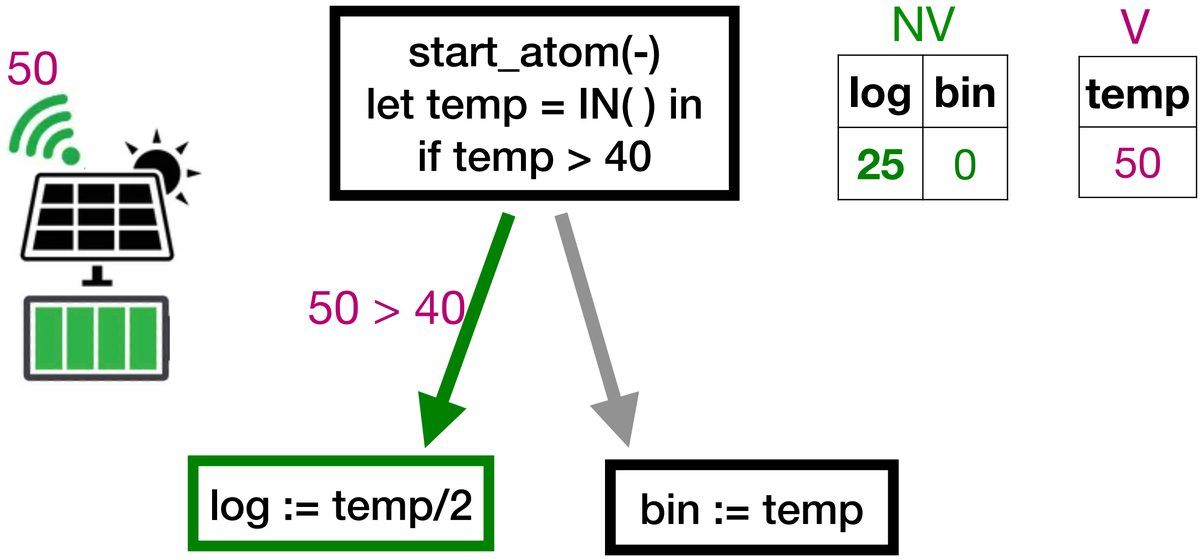
Embedded applications rely on *non-deterministic* sensor inputs

input-dependent = tainted



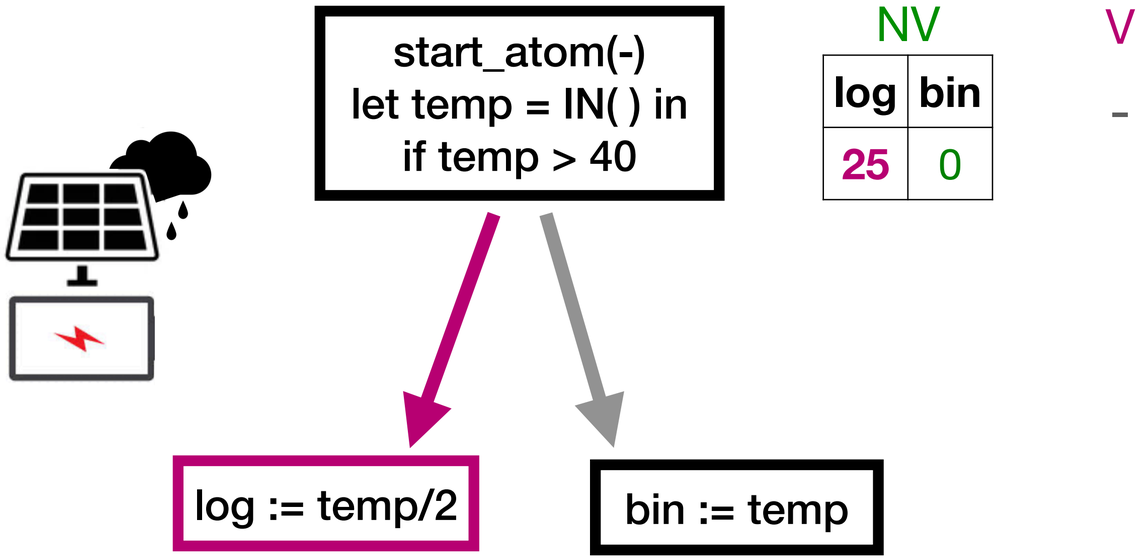
Embedded applications rely on *non-deterministic* sensor inputs

input-dependent = tainted



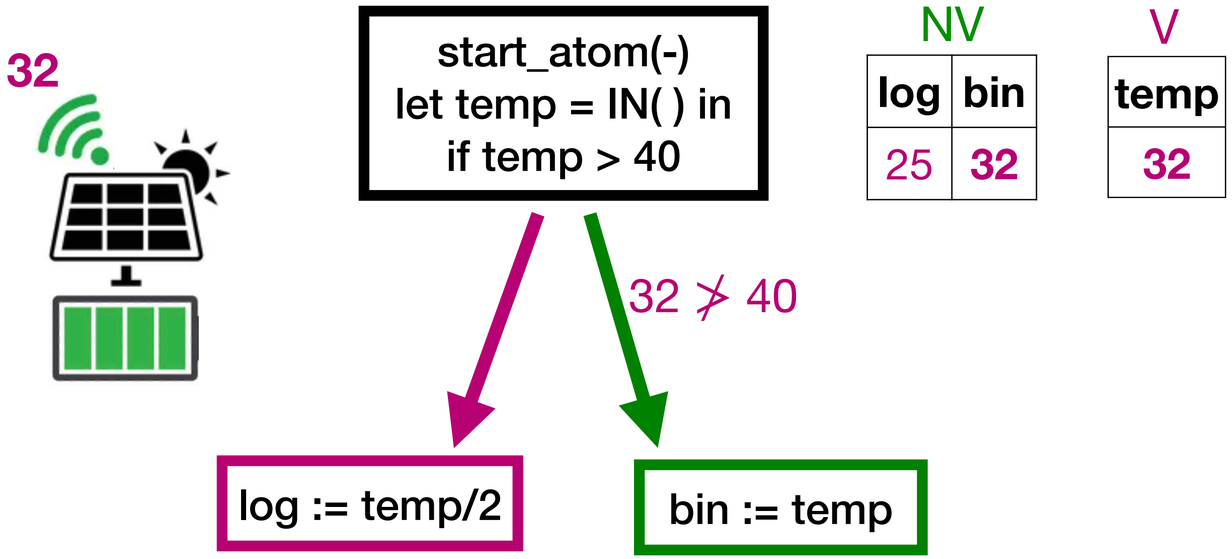
Embedded applications rely on *non-deterministic* sensor inputs

input-dependent = tainted



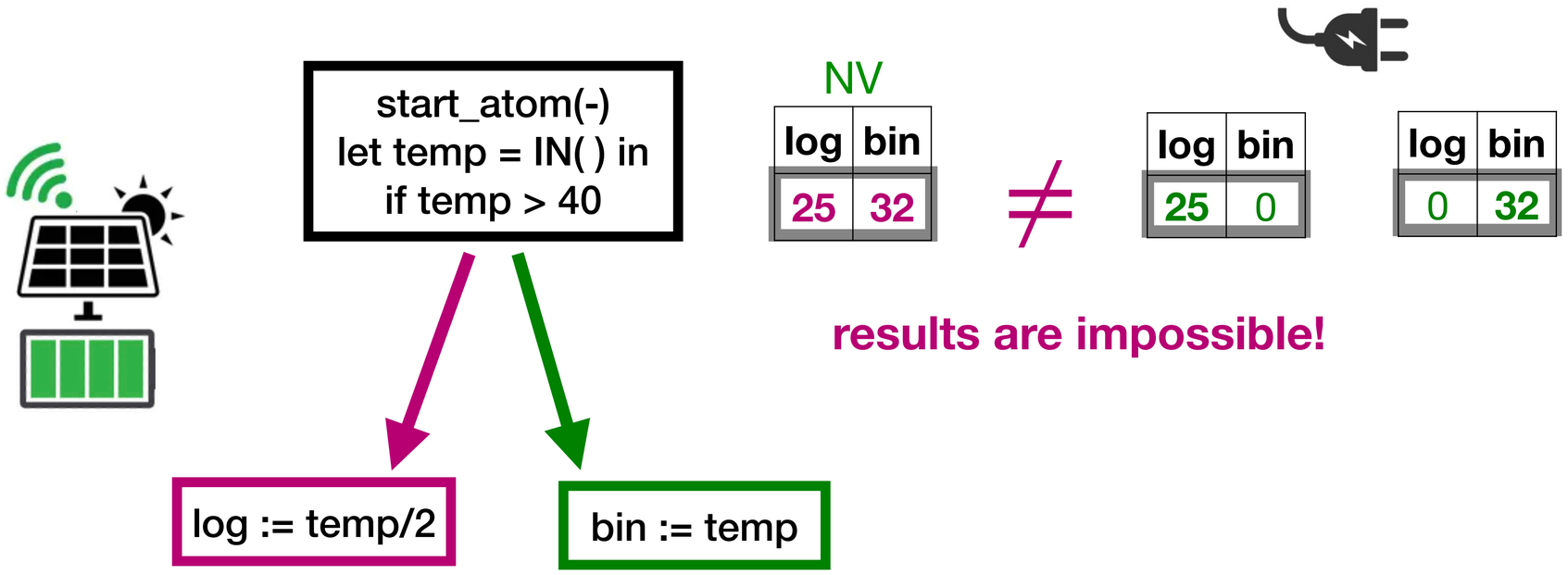
Embedded applications rely on *non-deterministic* sensor inputs

input-dependent = tainted



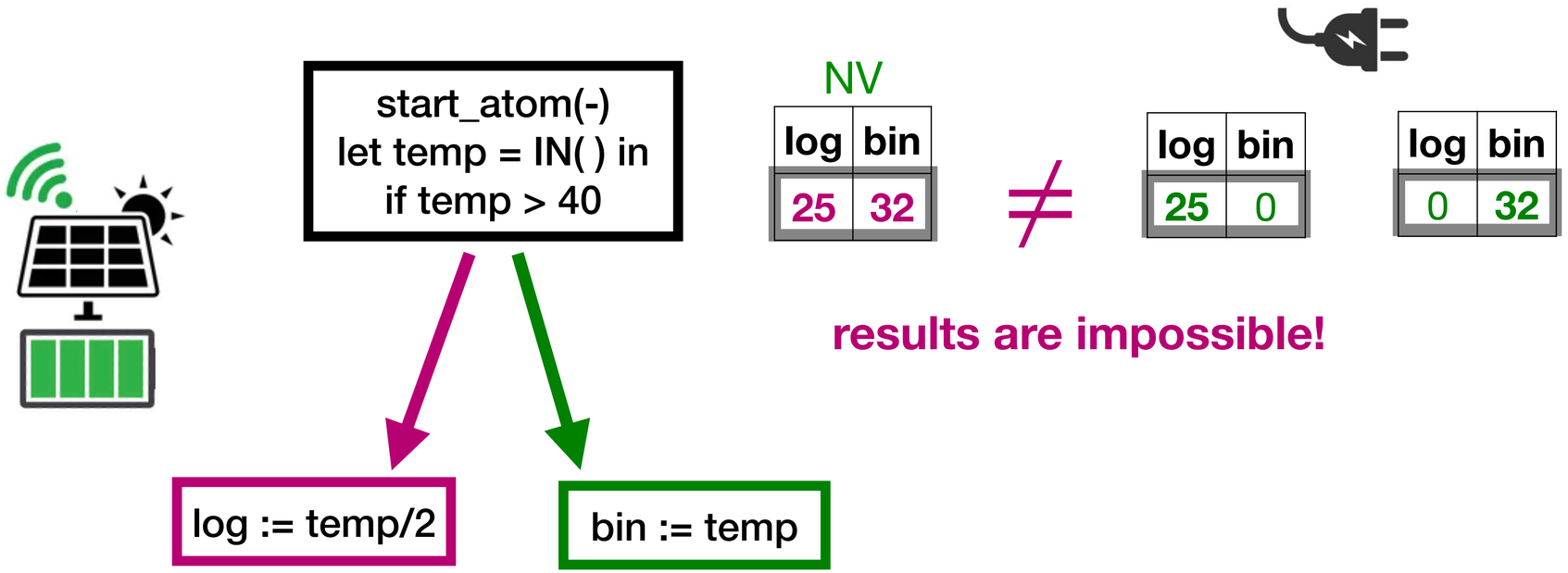
Embedded applications rely on *non-deterministic* sensor inputs

input-dependent = tainted



Another class of memory consistency bug: repeated I/O (RIO)

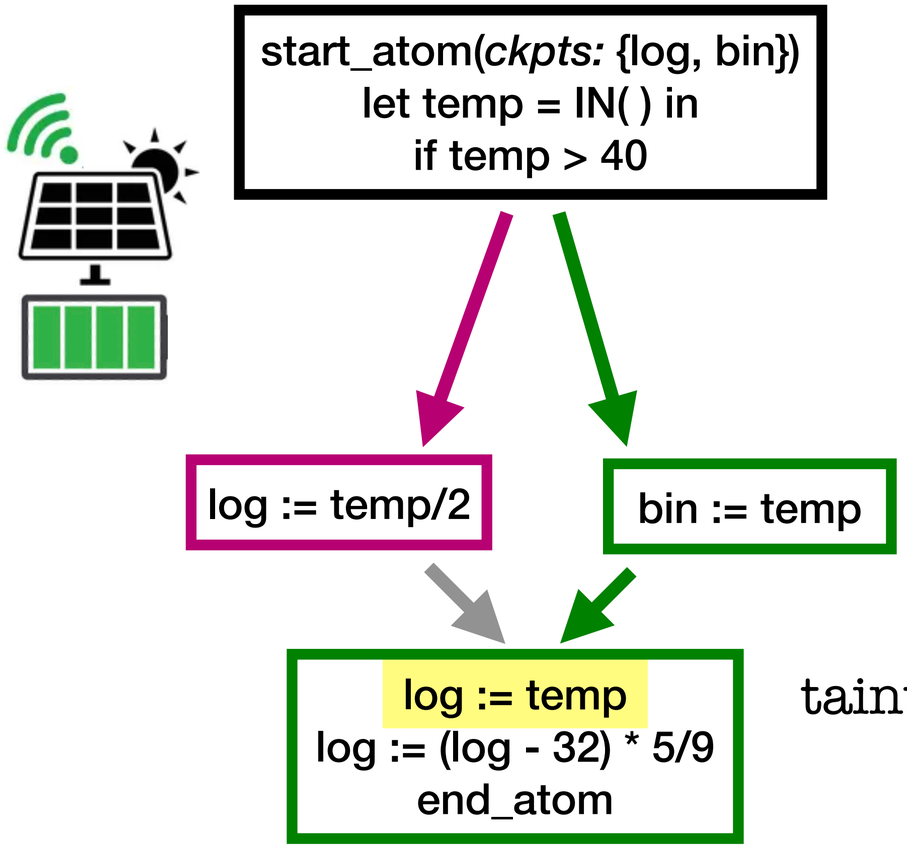
input-dependent = tainted



“checkpoint variables written in one branch but not all (e.g., log and bin)”

- Surbatovich et al., 2020

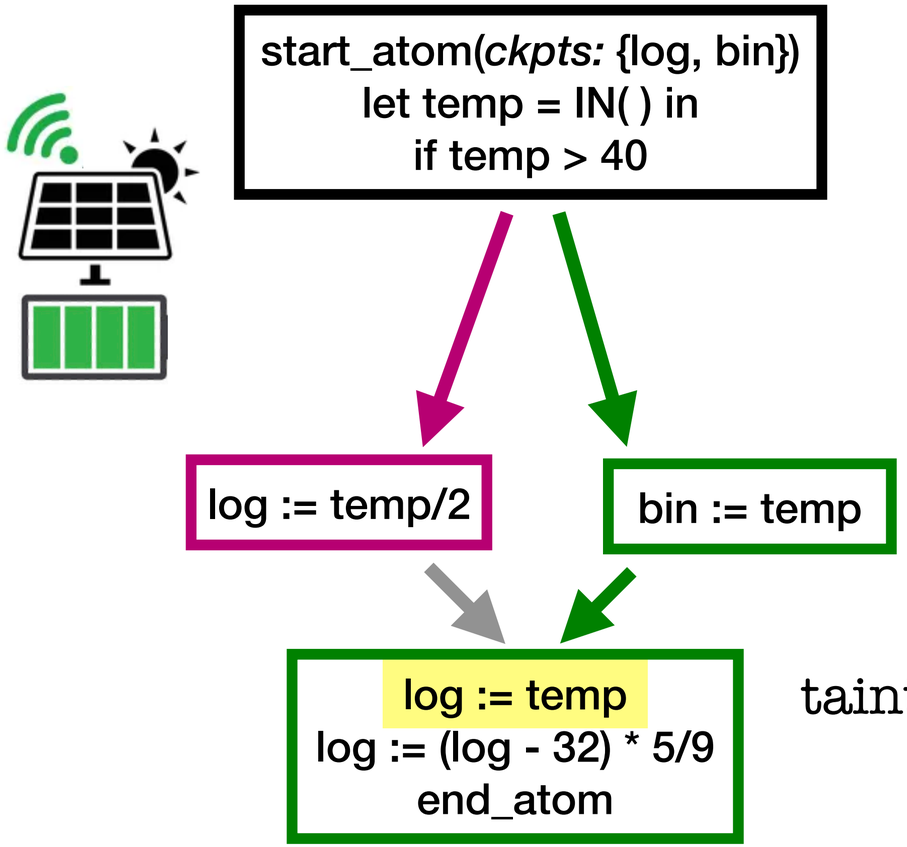
Prior RIO checking is too strict



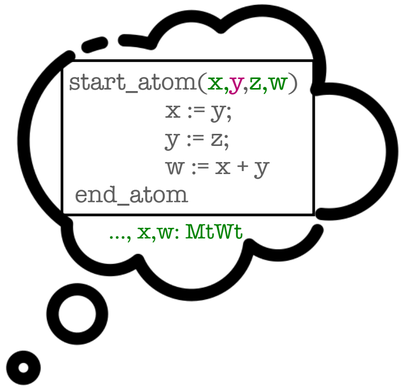
"log and bin must be checkpointed!"

tainted value of log is **overwritten before it is read**!

Prior RIO checking is too strict

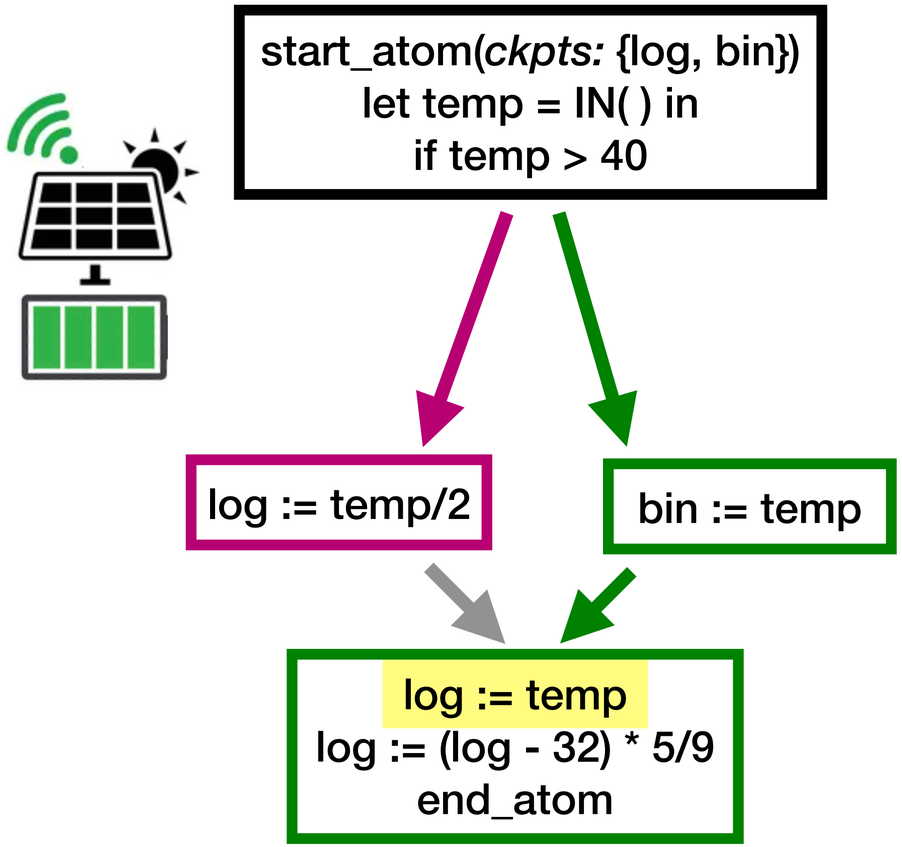


"log and bin must be checkpointed!"



tainted value of log is **overwritten before it is read**!

Prior RIO checking is too strict

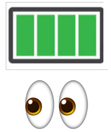


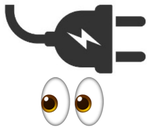
"log and bin must be checkpointed!"

Can we do better? i.e., develop a type checking that requires fewer checkpointed variables?

Another class of RIO bug

Repeated outputs cause duplicate effects



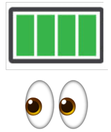


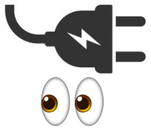
water_crop 💧💧

```
start_atom(-)
   water_crop
end_atom
```

Another class of RIO bug

Repeated outputs cause duplicate effects



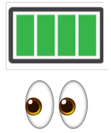


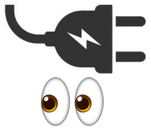
```
start_atom(-)
   water_crop
end_atom
```

pwoff

Another class of RIO bug

Repeated outputs cause duplicate effects



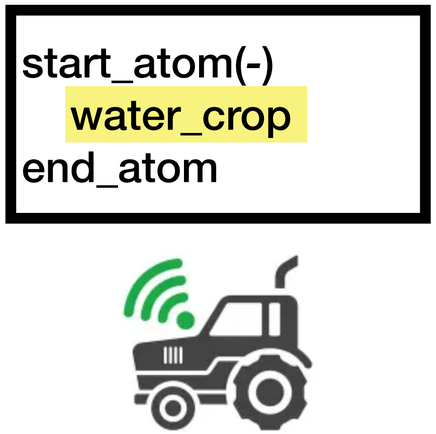


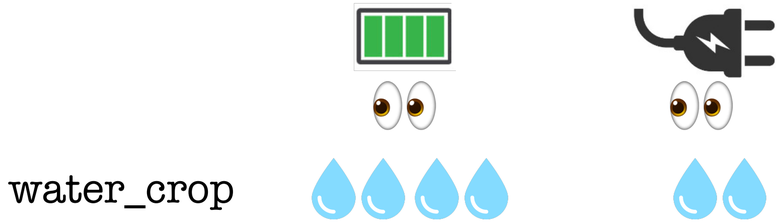
water_crop 

```
start_atom(-)
   water_crop
end_atom
```

Another class of RIO bug

Repeated outputs cause duplicate effects

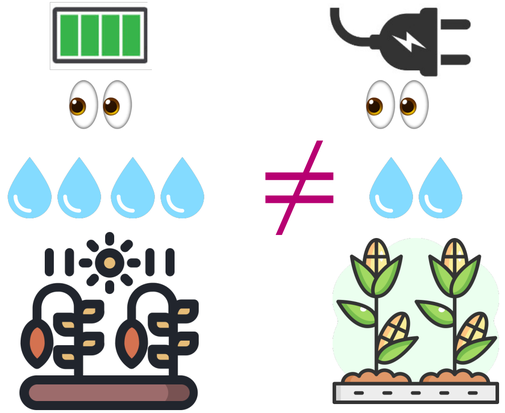




Another class of RIO bug

Repeated outputs cause duplicate effects

```
start_atom(-)
   water_crop
end_atom
```

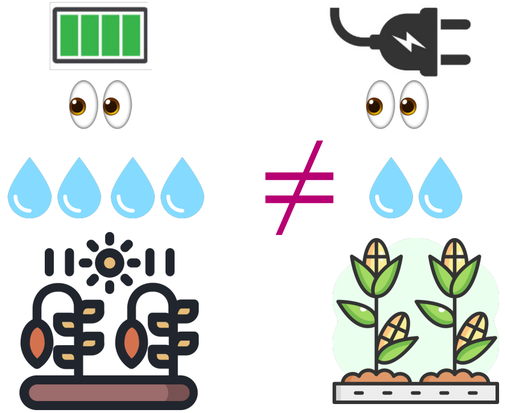


repeated outputs can cause irreversible and harmful effects!

Another class of RIO bug

Repeated outputs cause duplicate effects

```
start_atom(-)
   water_crop
end_atom
```

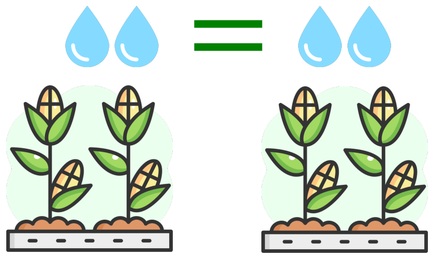


repeated outputs can cause irreversible and harmful effects!

no **provably correct** intermittent supports for outputs!

More flexible RIO supports

```
start_atom(-)
   water_crop
end_atom
```



idempotent outputs are not repeated

outputs

- idempotent or nonidempotent

  e.g., **water_crop** **calibrate_wheel**

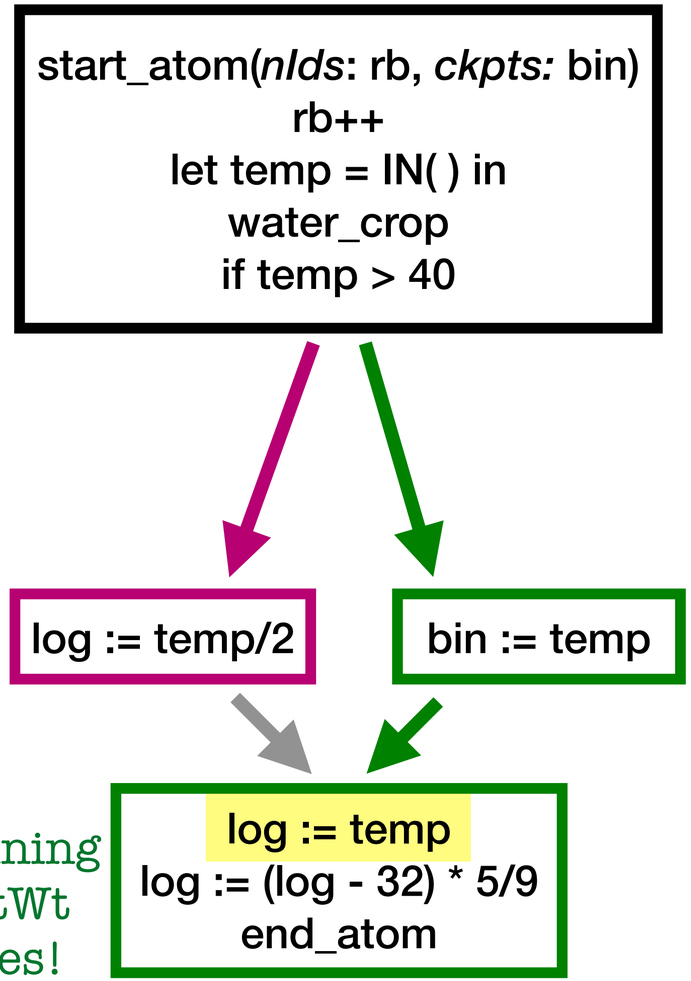
  

  0

- delay idempotent outputs until end of atomic region

More flexible RIO supports



<u>outputs</u>

- idempotent or nonidempotent

  e.g., **water_crop** **calibrate_wheel**

  0

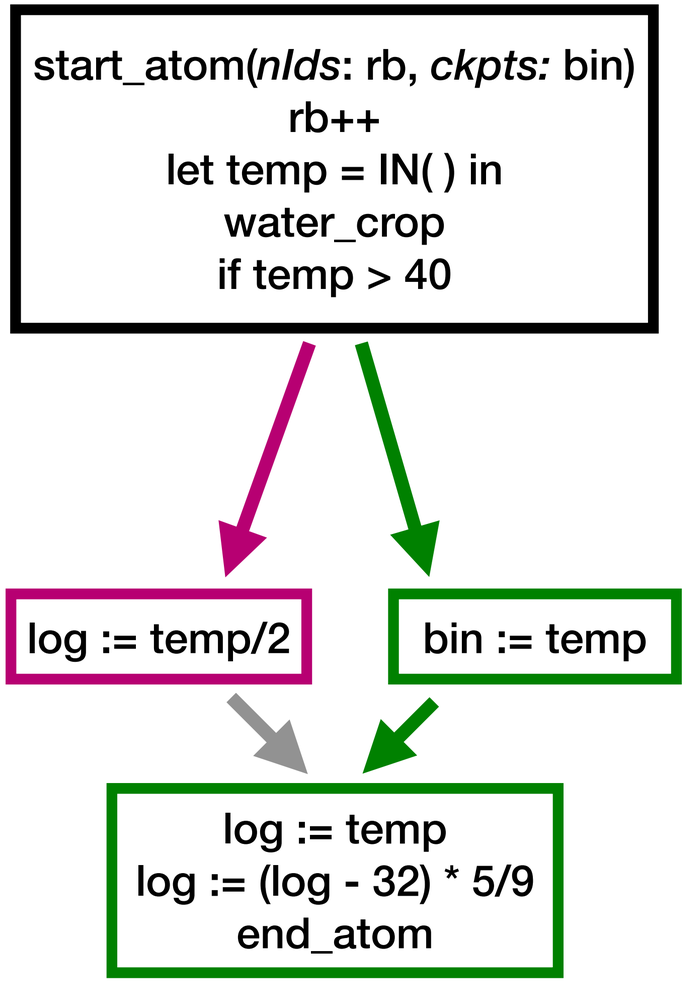
- delay idempotent outputs until end of atomic region

<u>inputs</u>

- fine-grained qualifiers to delay failing

  e.g., **mayTntWt** "written on one tainted branch but not on all"

Qualifiers for inputs and (non-)idempotence



$$q := \langle qId, qIO \rangle$$

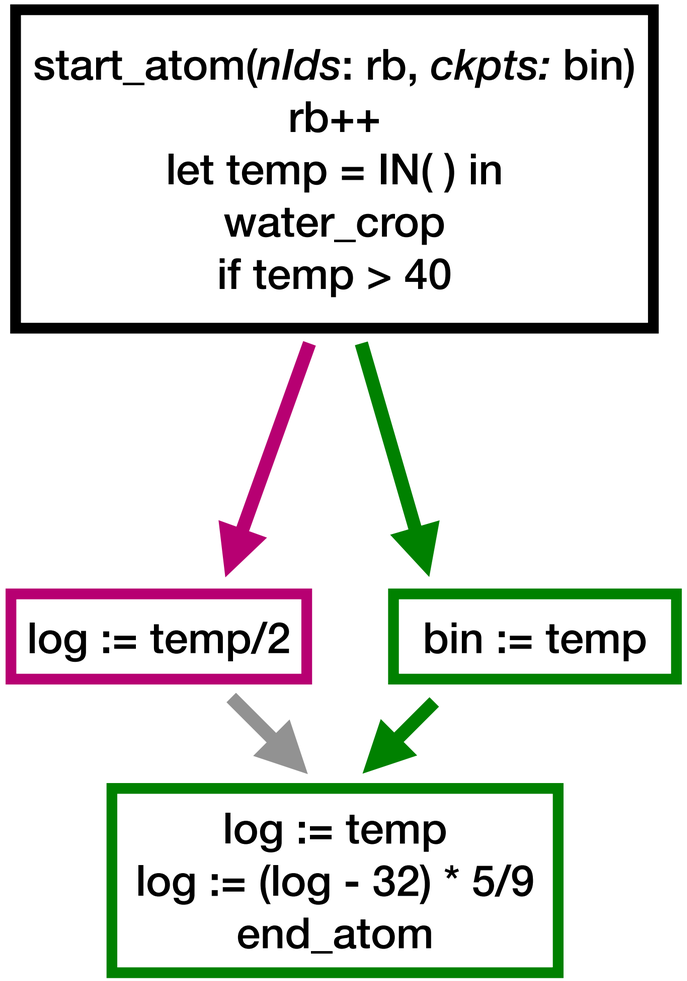
idem. $qId := \texttt{Id} \mid \texttt{Id}(qAcc) \mid \texttt{Nid}$

taint $qIO := \texttt{Tnt} \mid \texttt{Nt}$

access $qAcc := \texttt{CK} \mid \texttt{Wtn} \mid \texttt{NRD} \mid \texttt{mayFstRD} \mid \texttt{MTntWt} \mid \texttt{mayTntWt}$

see paper for join semi-lattices

Qualifiers for inputs and (non-)idempotence



$$q := \langle qId, qIO \rangle$$

idem. $qId := \texttt{Id} \mid \texttt{Id}(qAcc) \mid \texttt{Nid}$

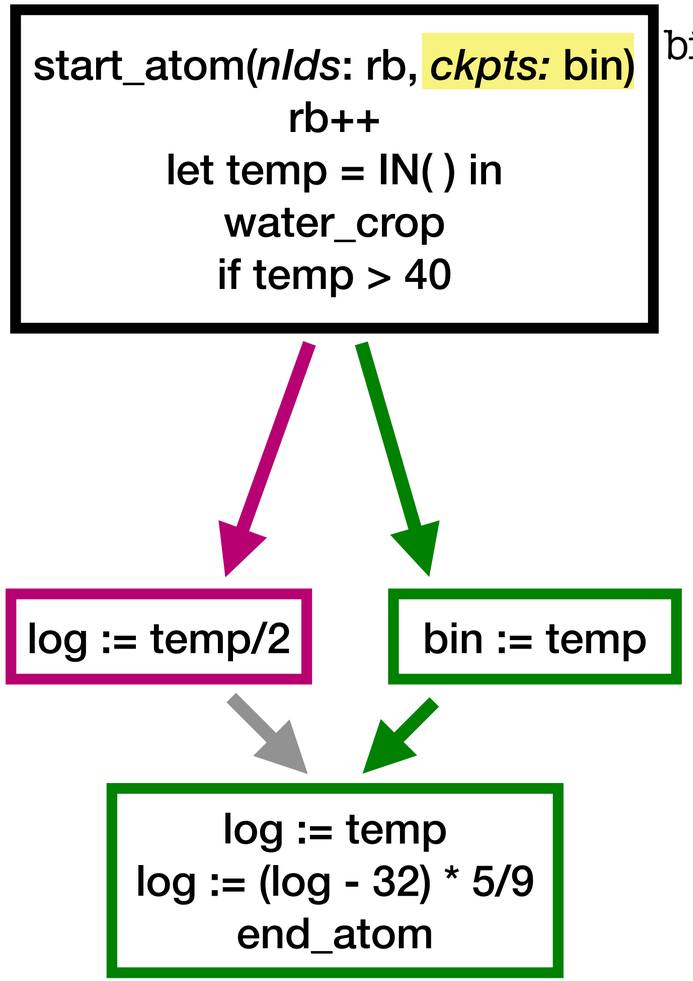
taint $qIO := \texttt{Tnt} \mid \texttt{Nt}$

access $qAcc := \texttt{CK} \mid \texttt{Wtn} \mid \texttt{NRD} \mid \texttt{mayFstRD} \mid \texttt{MTntWt} \mid \texttt{mayTntWt}$

see paper for join semi-lattices

access qualifiers allow more flexible type checking

Qualifiers for inputs and (non-)idempotence



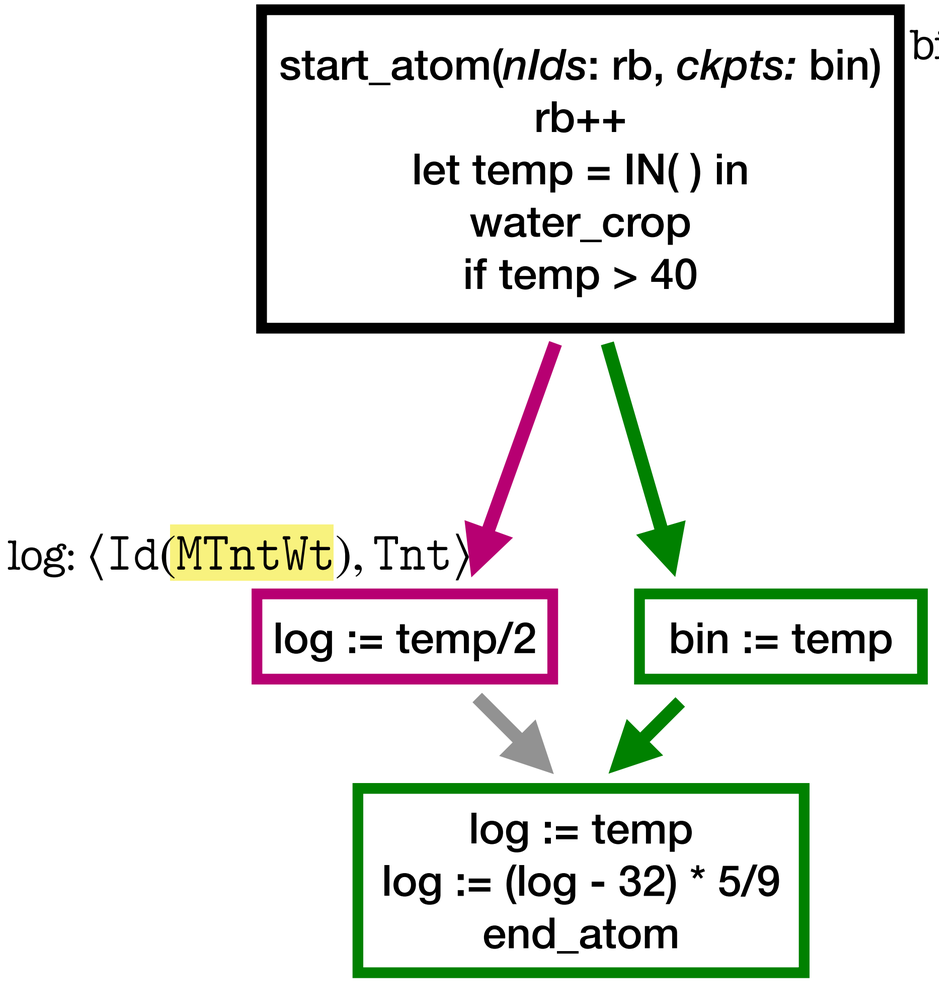
bin: $\langle \texttt{Id}(\texttt{CK}), \texttt{Nt} \rangle$, log: $\langle \texttt{Id}(\texttt{NRD}), \texttt{Nt} \rangle$

$$
\begin{array}{rr}
 & q := \langle qId, qIO \rangle \\
\text{idem.} & qId := \texttt{Id} \mid \texttt{Id}(qAcc) \mid \texttt{Nid} \\
\text{taint} & qIO := \texttt{Tnt} \mid \texttt{Nt} \\
\text{access} & qAcc := \texttt{CK} \mid \texttt{Wtn} \mid \texttt{NRD} \mid \texttt{mayFstRD} \\
 & \quad \mid \texttt{MTntWt} \mid \texttt{mayTntWt}
\end{array}
$$

CK — checkpointed

NRD — not read yet

Qualifiers for inputs and (non-)idempotence



bin: ⟨Id(CK), Nt⟩, log: ⟨Id(NRD), Nt⟩

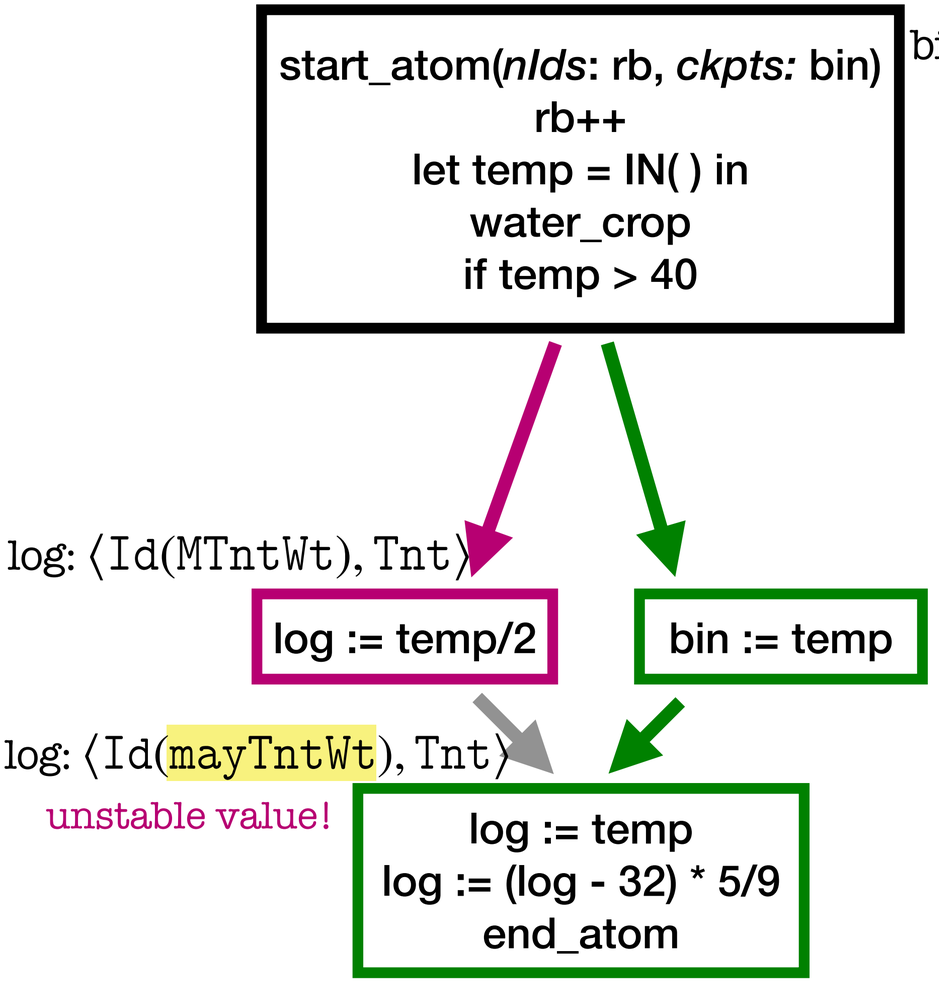
$$
\begin{array}{rr}
 & q := \langle qId, qIO \rangle \\
\text{idem.} & qId := \texttt{Id} \mid \texttt{Id}(qAcc) \mid \texttt{Nid} \\
\text{taint} & qIO := \texttt{Tnt} \mid \texttt{Nt} \\
\text{access} & qAcc := \texttt{CK} \mid \texttt{Wtn} \mid \texttt{NRD} \mid \texttt{mayFstRD} \\
 & \qquad \mid \texttt{MTntWt} \mid \texttt{mayTntWt}
\end{array}
$$

CK — checkpointed

NRD — not read yet

MTntWt — written on all paths (in Tnt context)

Qualifiers for inputs and (non-)idempotence



bin: ⟨Id(CK), Nt⟩, log: ⟨Id(NRD), Nt⟩

$$
\begin{array}{rr}
 & q := \langle qId, qIO \rangle \\
\text{idem.} & qId := \texttt{Id} \mid \texttt{Id}(qAcc) \mid \texttt{Nid} \\
\text{taint} & qIO := \texttt{Tnt} \mid \texttt{Nt} \\
\text{access} & qAcc := \texttt{CK} \mid \texttt{Wtn} \mid \texttt{NRD} \mid \texttt{mayFstRD} \\
 & \qquad \mid \texttt{MTntWt} \mid \texttt{mayTntWt}
\end{array}
$$

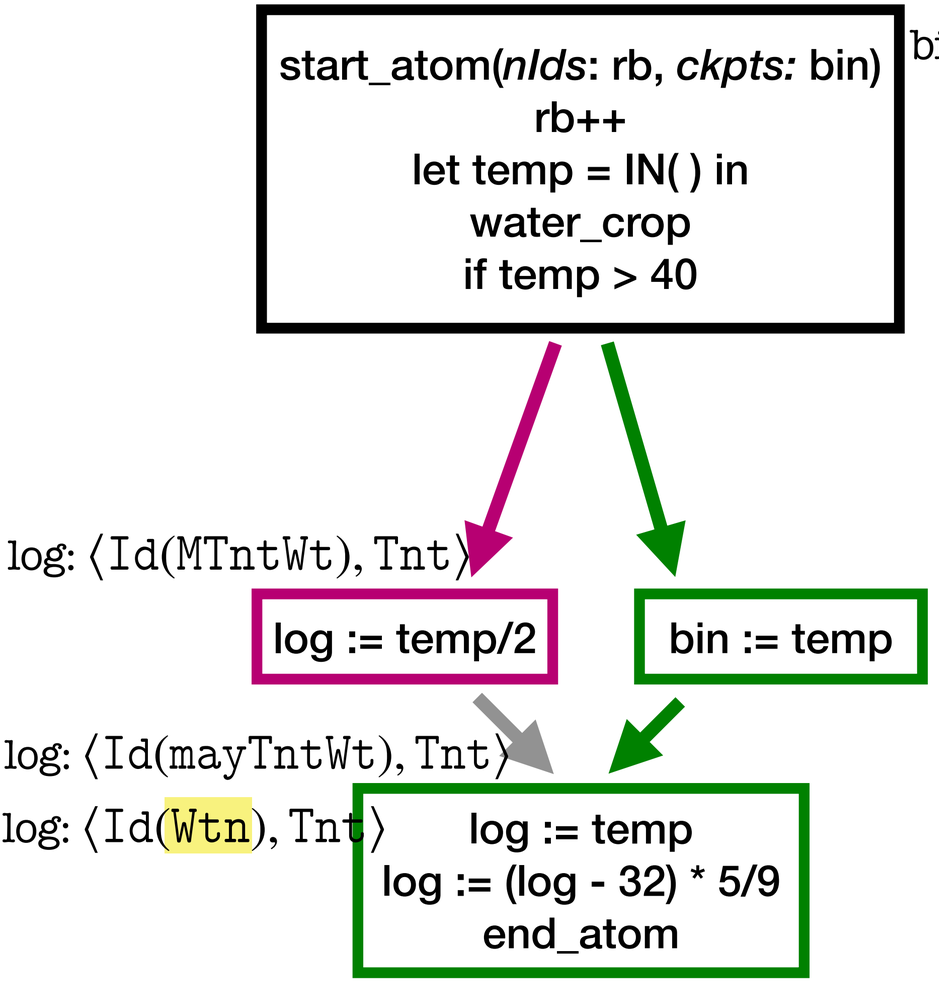
CK — checkpointed

NRD — not read yet

MTntWt — written on all paths (in Tnt context)

mayTntWt — written on one tainted path, but not on all

Qualifiers for inputs and (non-)idempotence



bin: ⟨Id(CK), Nt⟩, log: ⟨Id(NRD), Nt⟩

$$
\begin{array}{rr}
 & q := \langle qId, qIO \rangle \\
\text{idem.} & qId := \texttt{Id} \mid \texttt{Id}(qAcc) \mid \texttt{Nid} \\
\text{taint} & qIO := \texttt{Tnt} \mid \texttt{Nt} \\
\text{access} & qAcc := \texttt{CK} \mid \texttt{Wtn} \mid \texttt{NRD} \mid \texttt{mayFstRD} \\
 & \qquad \mid \texttt{MTntWt} \mid \texttt{mayTntWt}
\end{array}
$$

CK — checkpointed

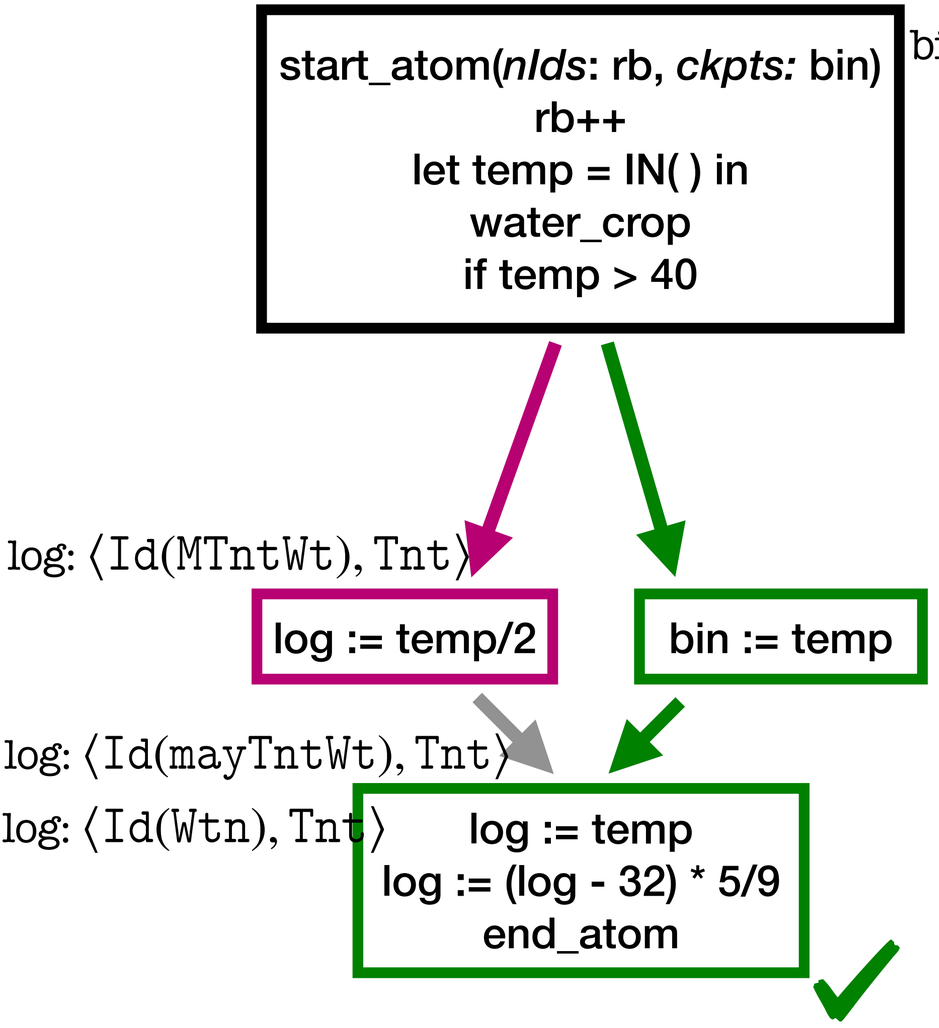
NRD — not read yet

MTntWt — written on all paths (in Tnt context)

mayTntWt — written on one tainted path, but not on all

Wtn — written on all paths (in Nt context)

Qualifiers for inputs and (non-)idempotence



bin: ⟨Id(CK), Nt⟩, log: ⟨Id(NRD), Nt⟩

$$
\begin{array}{rr}
 & q := \langle qId, qIO \rangle \\
\text{idem.} & qId := \texttt{Id} \mid \texttt{Id}(qAcc) \mid \texttt{Nid} \\
\text{taint} & qIO := \texttt{Tnt} \mid \texttt{Nt} \\
\text{access} & qAcc := \texttt{CK} \mid \texttt{Wtn} \mid \texttt{NRD} \mid \texttt{mayFstRD} \\
 & \qquad \mid \texttt{MTntWt} \mid \texttt{mayTntWt}
\end{array}
$$

CK — checkpointed

NRD — not read yet

MTntWt — written on all paths (in Tnt context)

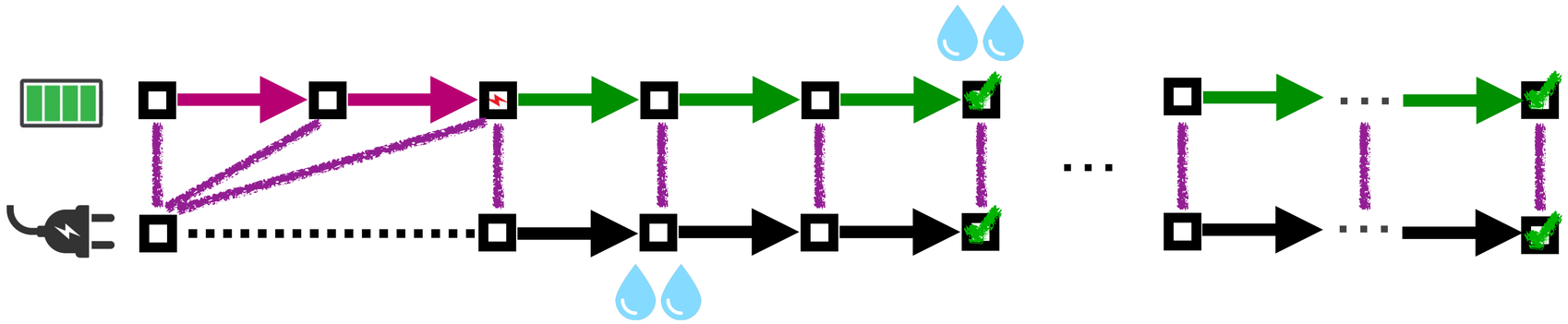
mayTntWt — written on one tainted path, but not on all

Wtn — written on all paths (in Nt context)

no remaining mayTntWt variables!

Metatheory

- progress and preservation
  - like undo-logging but accounting for new access qualifiers and post-contexts
- fundamental theorem and adequacy (future work!)
  - reasoning about output orderings and nonidempotence



Conclusion and Future Work

Summary

- Modal crash types for intermittent computing [Derakhshan et al., ESOP '23] [Dotzel et al., TOPLAS '25]
- Undo-logging
- I/O, flexible branching, nonidempotence
- Intermittent shared memory concurrency

Summary

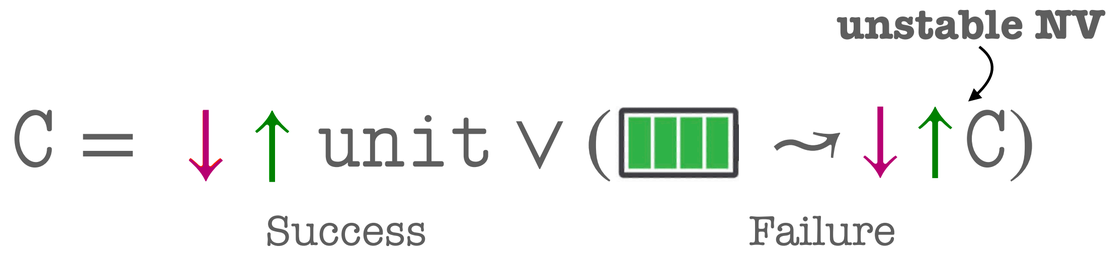
- **Modal crash types for intermittent computing** [Derakhshan et al., ESOP '23] [Dotzel et al., TOPLAS '25]
- Undo-logging
- I/O, flexible branching, nonidempotence
- Intermittent shared memory concurrency

"Every intermittent execution of a correct program has a corresponding continuously-powered execution of it."

(via a **logical relation**)

Summary

- Modal crash types for intermittent computing [Derakhshan et al., ESOP '23] [Dotzel et al., TOPLAS '25]
- **Undo-logging**
- I/O, flexible branching, nonidempotence
- Intermittent shared memory concurrency



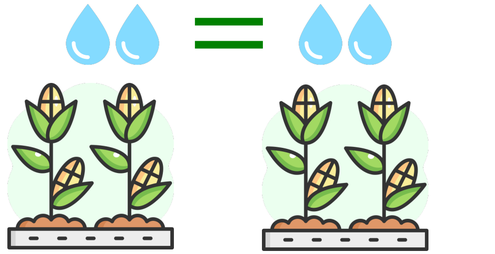
"Every intermittent execution of a correct program has a corresponding continuously-powered execution of it."

(via a **logical relation**)

Summary

- Modal crash types for intermittent computing [Derakhshan et al., ESOP '23] [Dotzel et al., TOPLAS '25]
- Undo-logging
- **I/O, flexible branching, nonidempotence**
- Intermittent shared memory concurrency

delay failing until the end of an atomic region



idempotent outputs are not repeated

"Every intermittent execution of a correct program has a corresponding continuously-powered execution of it."

(via a **logical relation**)

Summary

- Modal crash types for intermittent computing [Derakhshan et al., ESOP '23] [Dotzel et al., TOPLAS '25]
- Undo-logging
- I/O, flexible branching, nonidempotence
- **Intermittent shared memory concurrency**

*The **first** provably correct supports for intermittent concurrency.*

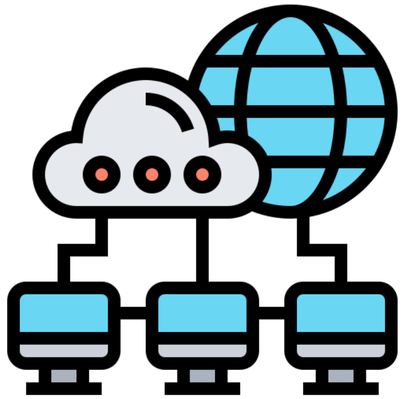
> *"Every intermittent execution of a correct **concurrent** program has a corresponding continuously-powered execution ~~of it~~."*

(via an **equivalence relation**)

processes that do not successfully finish executing should not affect idempotent results!

Future Work

shared memory concurrency in crash types

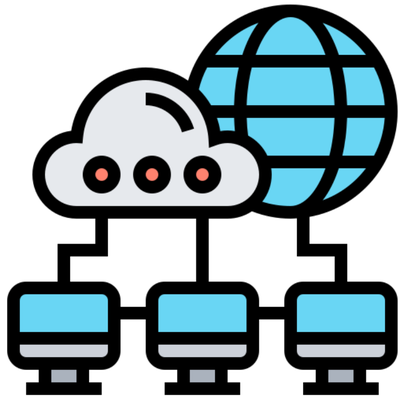


distributed intermittent systems

fault-resilient cyberphysical systems

Future Work

shared memory concurrency in crash types

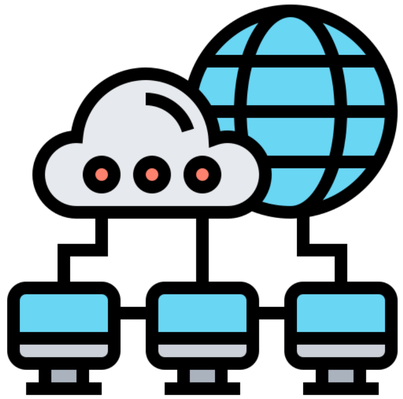


distributed intermittent systems

fault-resilient cyberphysical systems

Future Work

shared memory concurrency in crash types

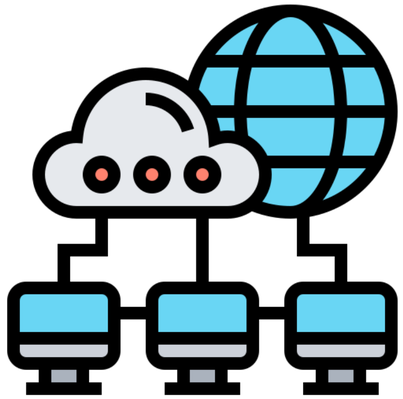


distributed intermittent systems

fault-resilient cyberphysical systems

Future Work

shared memory concurrency in crash types



distributed intermittent systems

fault-resilient cyberphysical systems

Logical Foundations of Intermittent Computing

Myra Dotzel

PhD Thesis Defense