Qurts: Automatic Quantum Uncomputation by Affine Types with Lifetime

Kengo Hirata, Chris Heunen (University of Edinburgh)

Quantum Programming Language

Classical Computation	Quantum Computation
Bits $00\ldots0,\ 00\ldots1,\ \ldots\ 11\ldots1$	Qubits $\alpha_0\lvert 00\ldots0\rangle + \alpha_1\lvert 00\ldots1\rangle + \ldots + \alpha_{2^n-1}\lvert 11\ldots1\rangle$
Program = (computable) function	Unitary + Measurement
✓ Copy & Discard	Linear Type (✗ copy & discard)

Quantum Programming Language

Linear Types

- ✦Copy

✗

```
let y := U(x);
let z := V(x);
return (y, z)
```

- ✦Discarding

✗

```
{
  let y := U(x);
  // forget y
}
```

Quantum Computation

↔ No-cloning theorem

↔ Principle of implicit measurement

Example: No discarding?

Given:

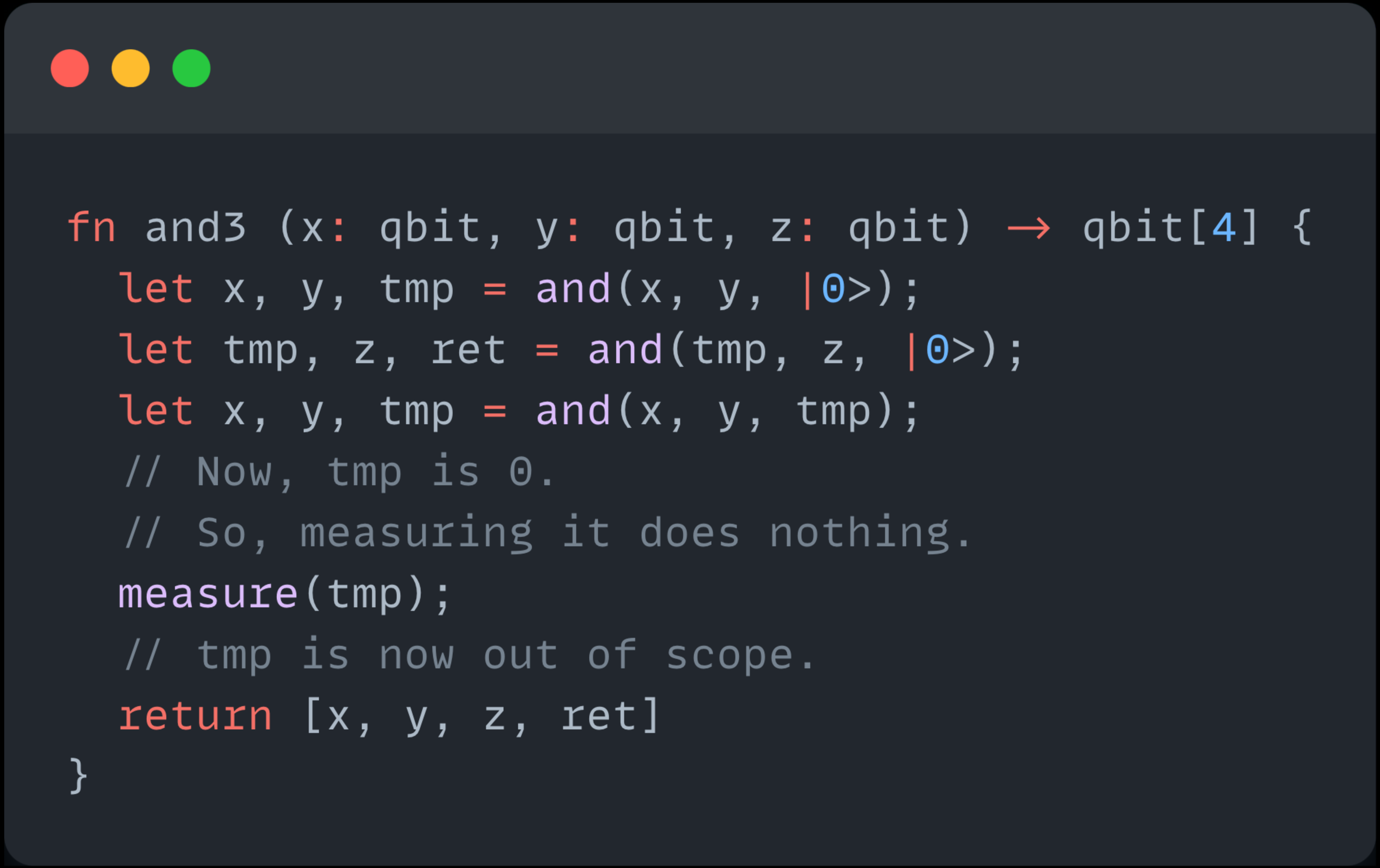
$$\text{and}(q_1,\ \ q_2,\ \ q_3) \quad := \quad (q_1,\ \ q_2,\ \ q_3 \oplus (q_1 \land q_2))$$

$$\begin{matrix} |000\rangle \\ |110\rangle \end{matrix} \quad \mapsto \quad \begin{matrix} |000\rangle \\ |111\rangle \end{matrix}$$

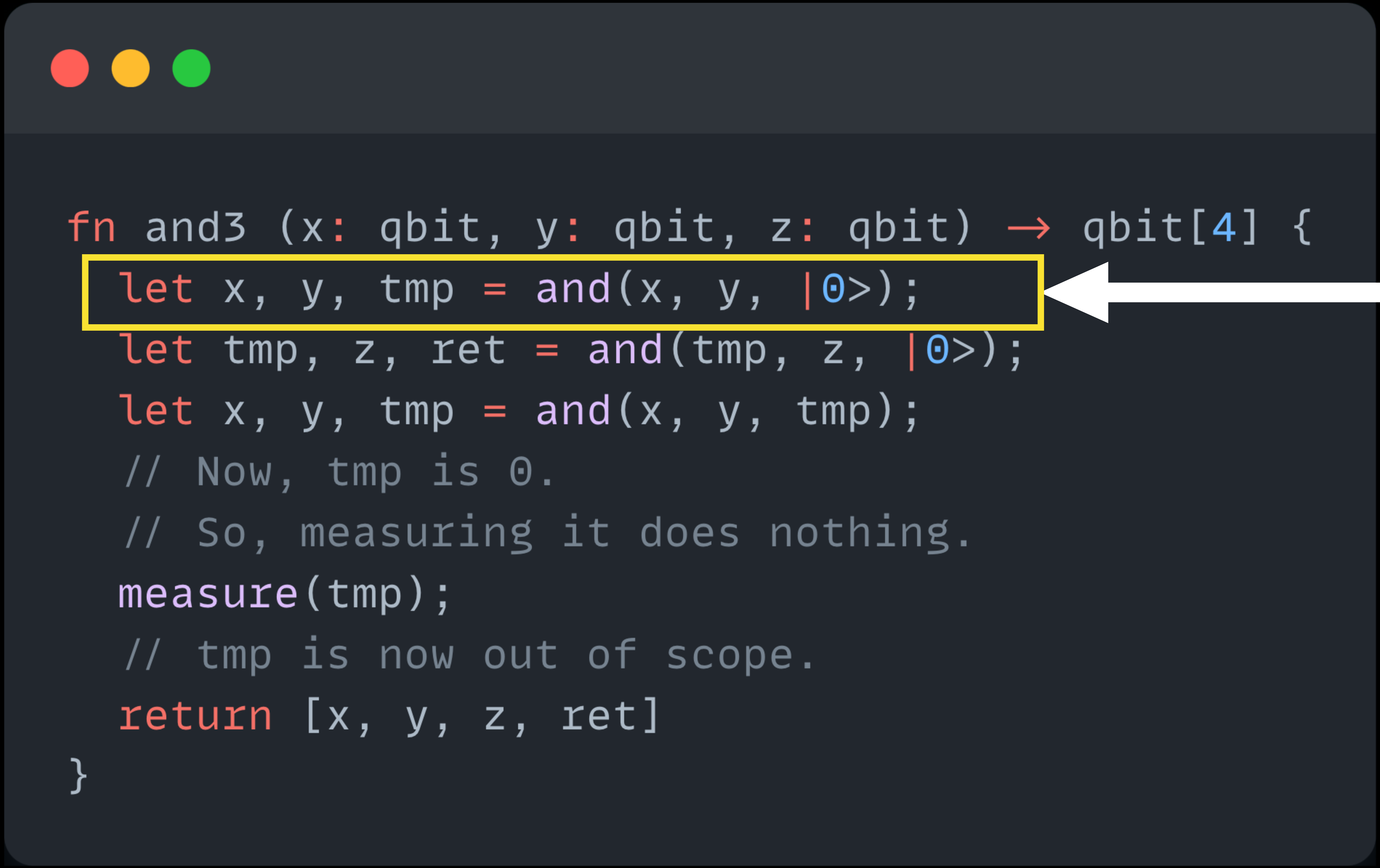
Goal:

$$\text{and3}(q_1,\ \ q_2,\ \ q_3) \quad := \quad (q_1,\ \ q_2,\ \ q_3,\ \ q_1 \land q_2 \land q_3)$$

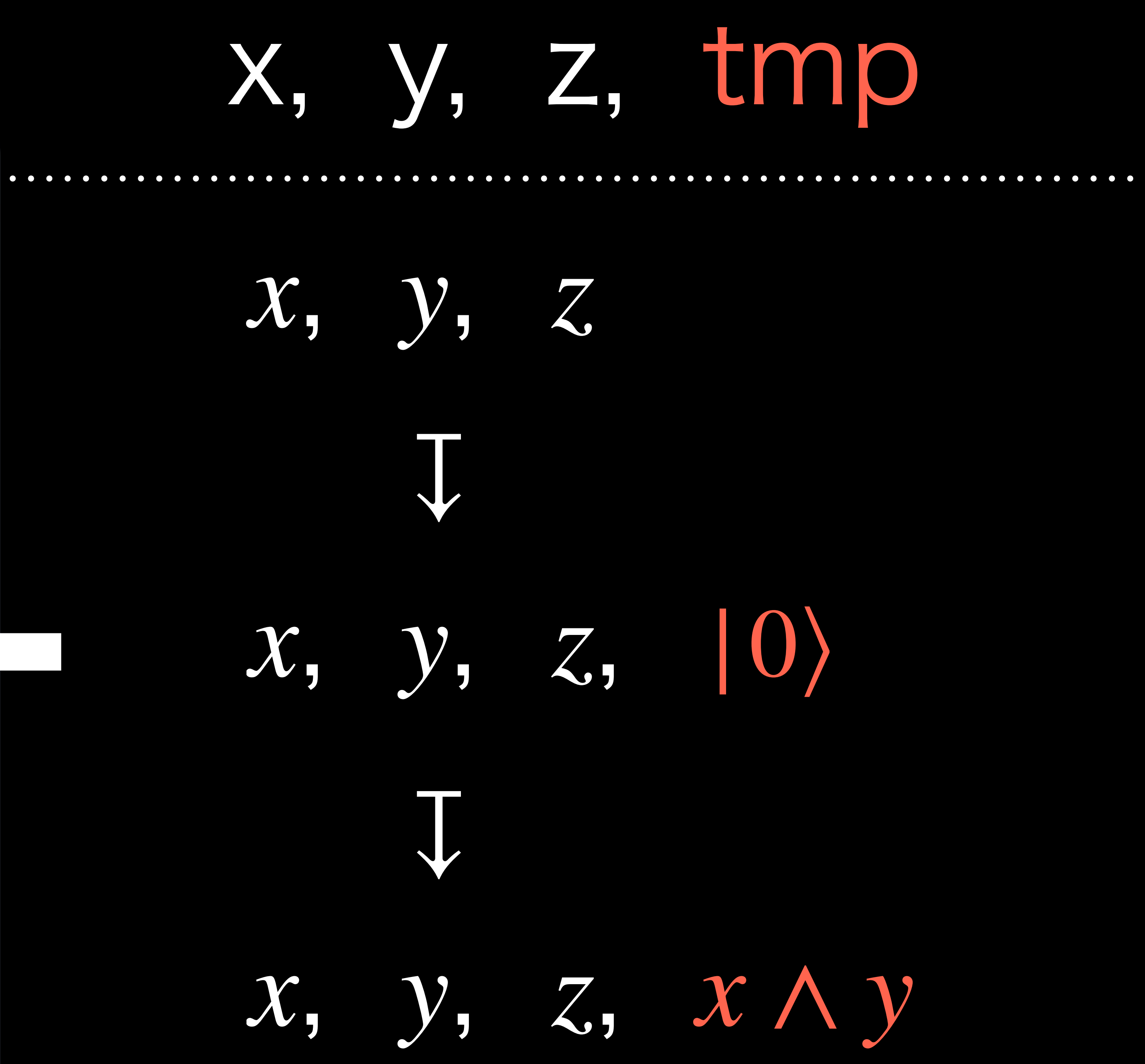
$$|111\rangle \quad \mapsto \quad |1111\rangle$$



```
fn and3 (x: qbit, y: qbit, z: qbit) -> qbit[4] {
  let x, y, tmp = and(x, y, |0>);
  let tmp, z, ret = and(tmp, z, |0>);
  let x, y, tmp = and(x, y, tmp);
  // Now, tmp is 0.
  // So, measuring it does nothing.
  measure(tmp);
  // tmp is now out of scope.
  return [x, y, z, ret]
}
```



```
fn and3 (x: qbit, y: qbit, z: qbit) → qbit[4] {
  let x, y, tmp = and(x, y, |0>);
  let tmp, z, ret = and(tmp, z, |0>);
  let x, y, tmp = and(x, y, tmp);
  // Now, tmp is 0.
  // So, measuring it does nothing.
  measure(tmp);
  // tmp is now out of scope.
  return [x, y, z, ret]
}
```



x, y, z, tmp

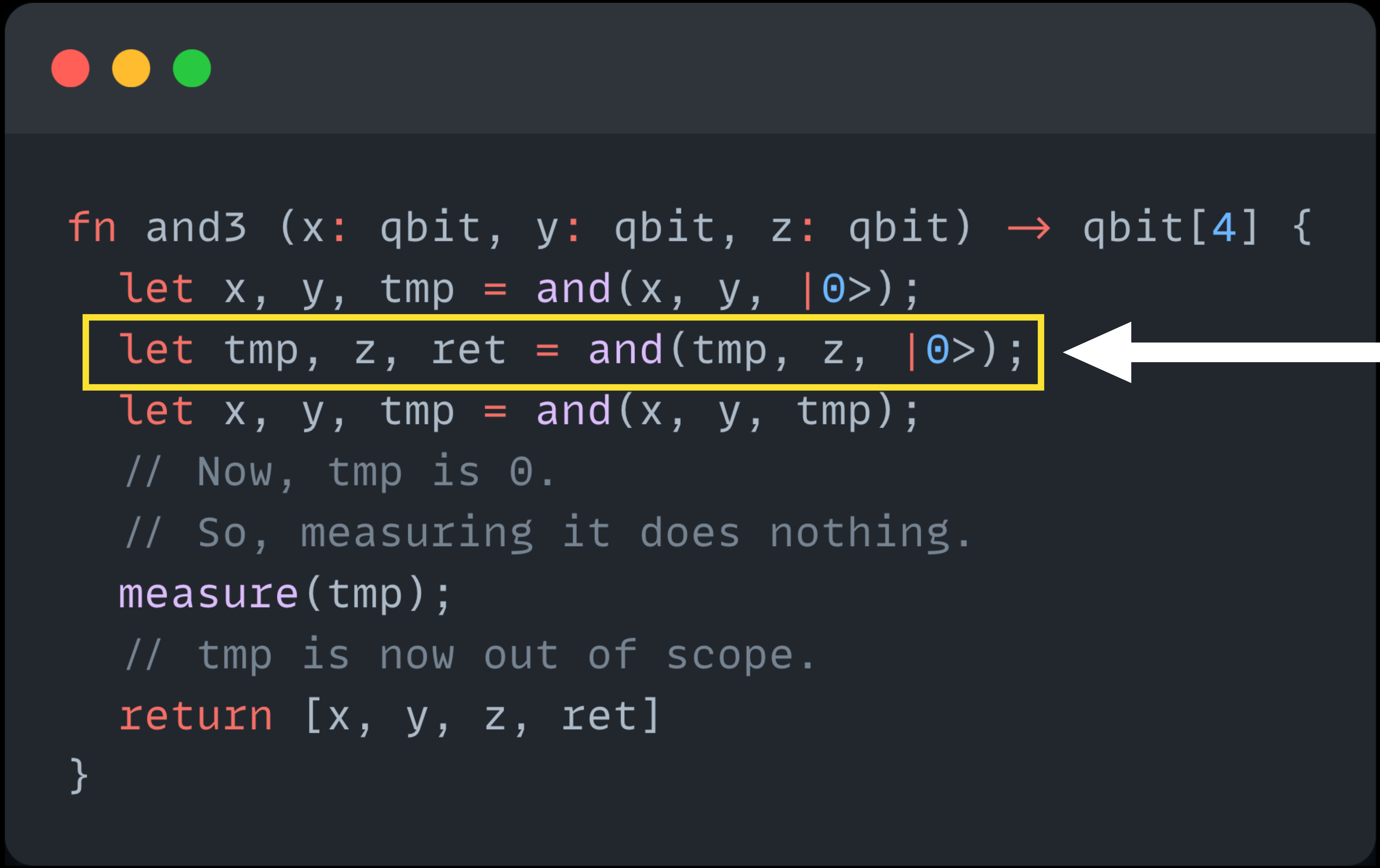
x, y, z

$\downarrow$

$x, y, z, |0\rangle$

$\downarrow$

$x, y, z, x \wedge y$



```
fn and3 (x: qbit, y: qbit, z: qbit) → qbit[4] {
  let x, y, tmp = and(x, y, |0>);
  let tmp, z, ret = and(tmp, z, |0>);
  let x, y, tmp = and(x, y, tmp);
  // Now, tmp is 0.
  // So, measuring it does nothing.
  measure(tmp);
  // tmp is now out of scope.
  return [x, y, z, ret]
}
```

x,y,z, tmp, ret

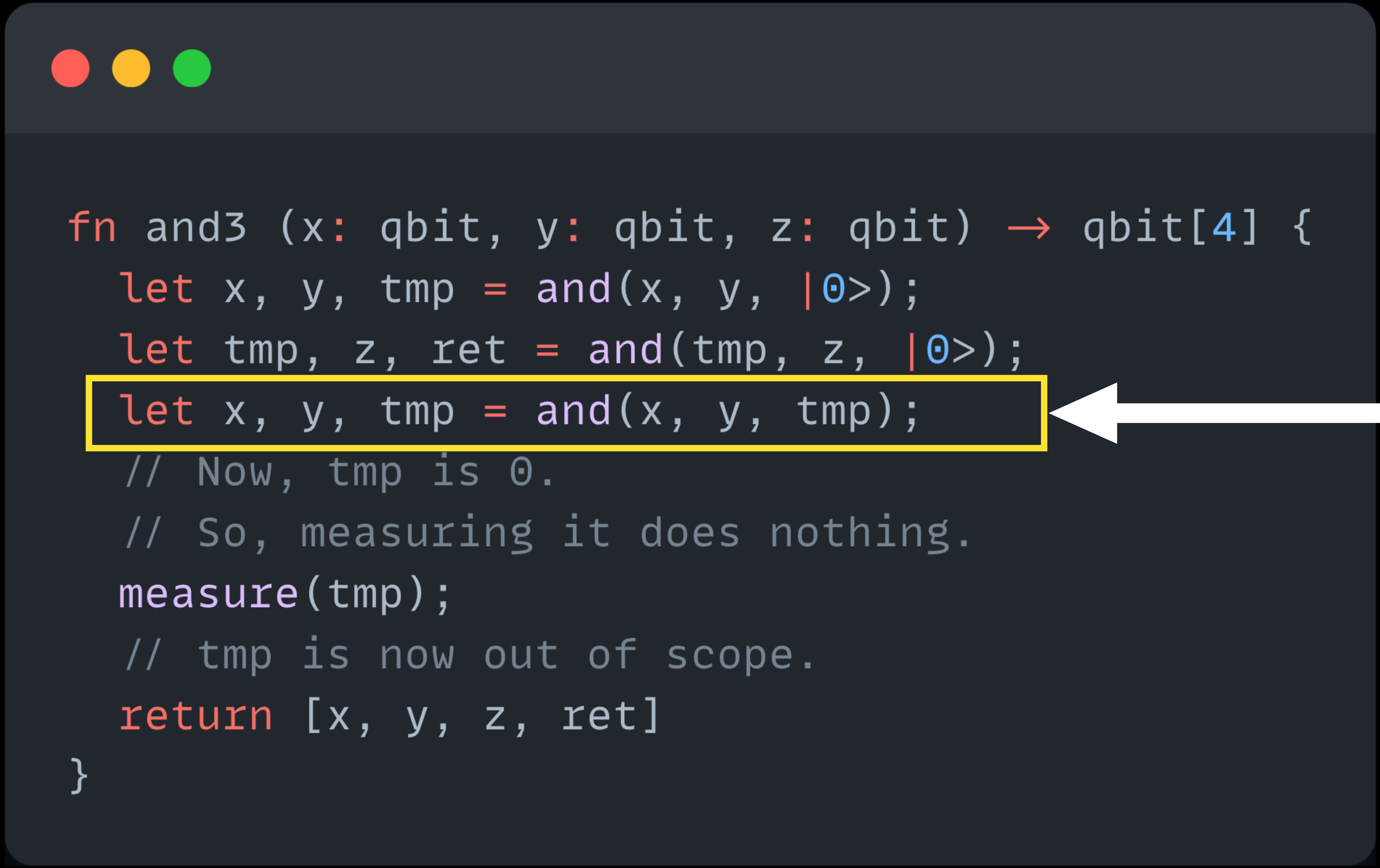
〃, $x \wedge y$

$\mapsto$

〃, $x \wedge y$, $|0\rangle$

$\mapsto$

〃, $x \wedge y$, $x \wedge y \wedge z$



```
fn and3 (x: qbit, y: qbit, z: qbit) → qbit[4] {
  let x, y, tmp = and(x, y, |0>);
  let tmp, z, ret = and(tmp, z, |0>);
  let x, y, tmp = and(x, y, tmp);
  // Now, tmp is 0.
  // So, measuring it does nothing.
  measure(tmp);
  // tmp is now out of scope.
  return [x, y, z, ret]
}
```

x,y,z, tmp, ret

$〃, \; x \wedge y$

$\downarrow$

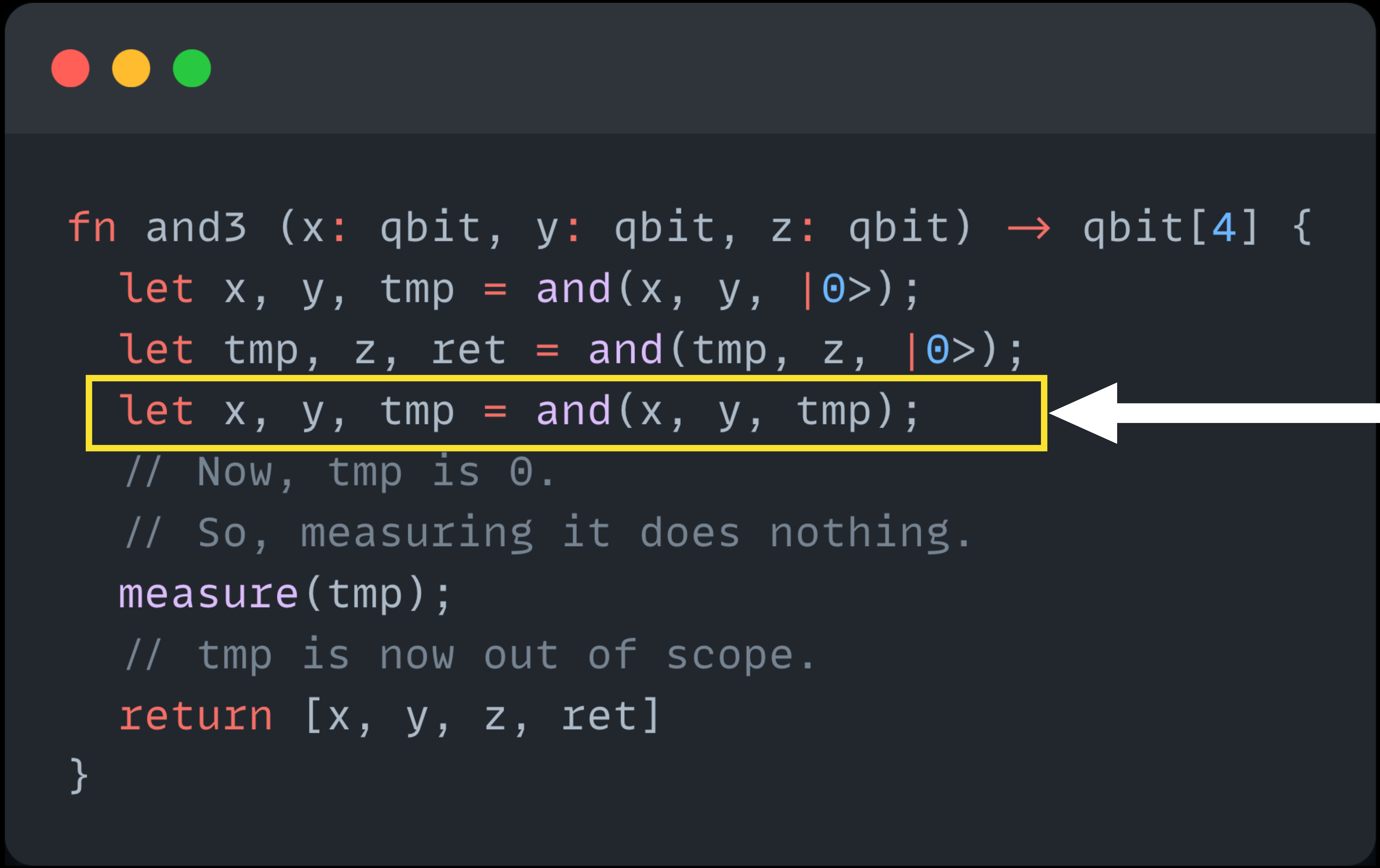
$〃, \; x \wedge y, \; |0\rangle$

$\downarrow$

$〃, \; x \wedge y, \; x \wedge y \wedge z$

$\downarrow$

$〃, \; (x \wedge y) \oplus (x \wedge y), \; x \wedge y \wedge z$



```
fn and3 (x: qbit, y: qbit, z: qbit) → qbit[4] {
  let x, y, tmp = and(x, y, |0>);
  let tmp, z, ret = and(tmp, z, |0>);
  let x, y, tmp = and(x, y, tmp);
  // Now, tmp is 0.
  // So, measuring it does nothing.
  measure(tmp);
  // tmp is now out of scope.
  return [x, y, z, ret]
}
```

x,y,z, tmp, ret

$〃, \quad x \wedge y$

$\downarrow$

$〃, \quad x \wedge y, \quad |0\rangle$

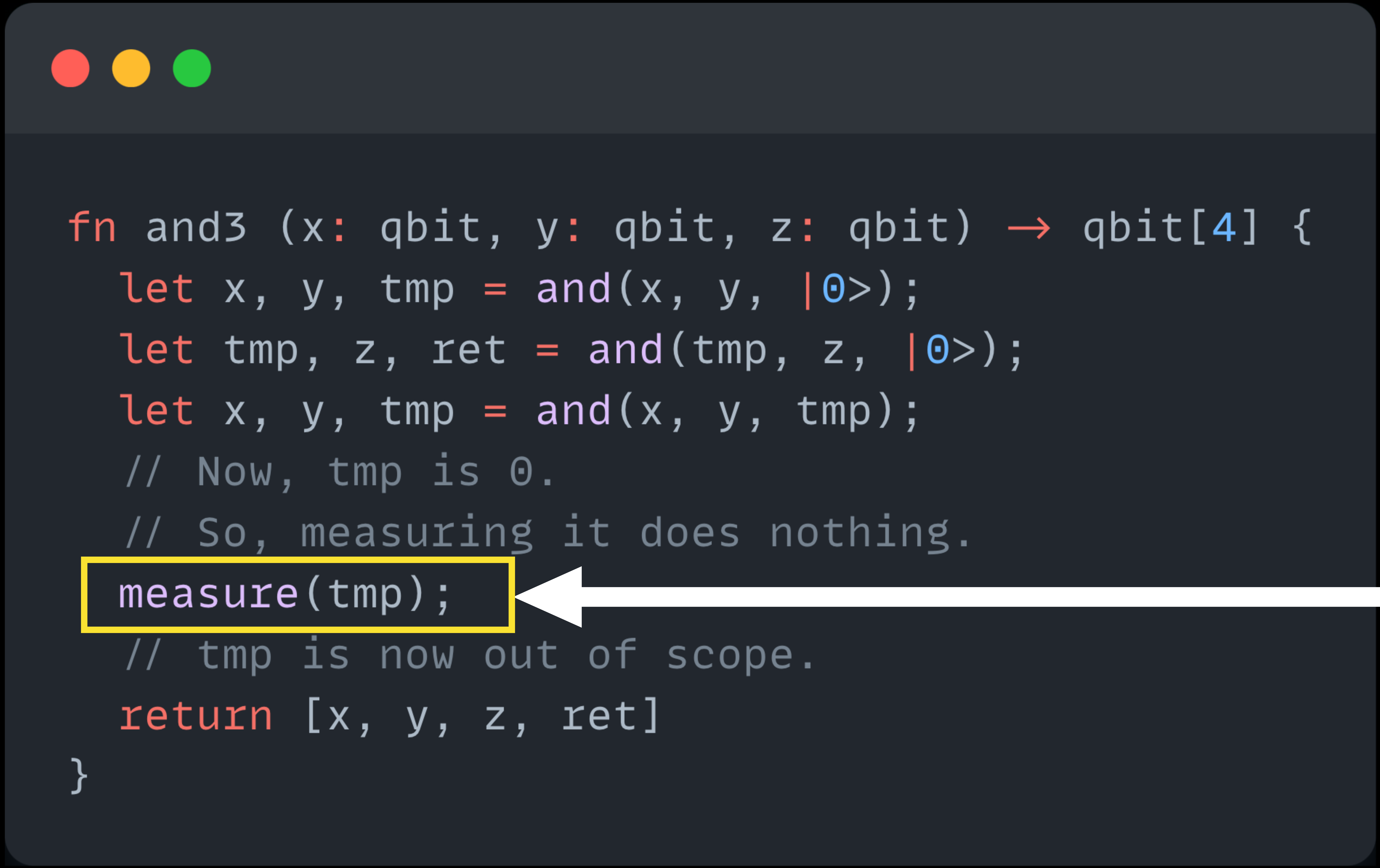
$\downarrow$

$〃, \quad x \wedge y, \quad x \wedge y \wedge z$

$\downarrow$

$〃, \quad |0\rangle,$

$x \wedge y \wedge z$



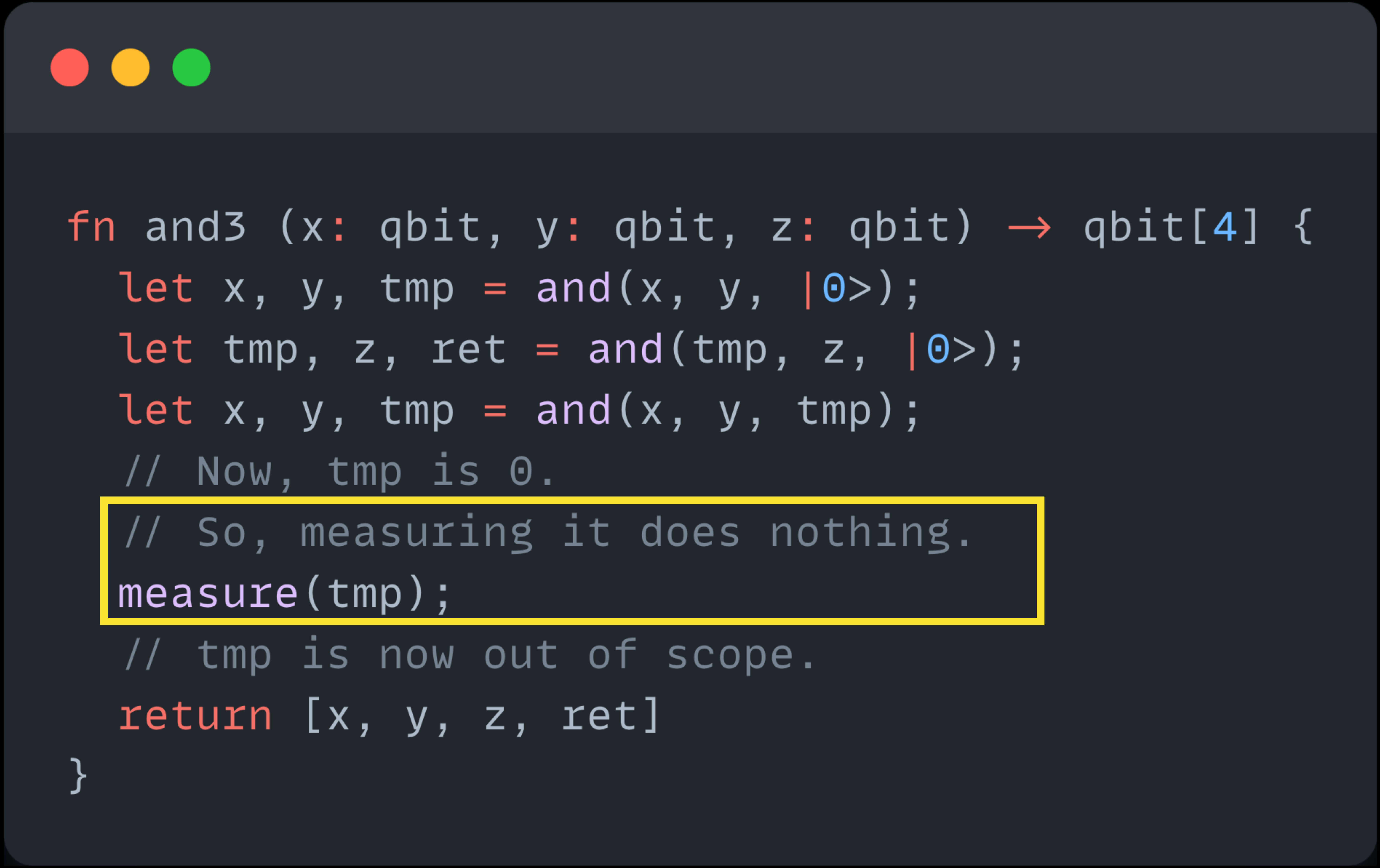
```
fn and3 (x: qbit, y: qbit, z: qbit) → qbit[4] {
  let x, y, tmp = and(x, y, |0>);
  let tmp, z, ret = and(tmp, z, |0>);
  let x, y, tmp = and(x, y, tmp);
  // Now, tmp is 0.
  // So, measuring it does nothing.
  measure(tmp);
  // tmp is now out of scope.
  return [x, y, z, ret]
}
```

x,y,z, tmp, ret

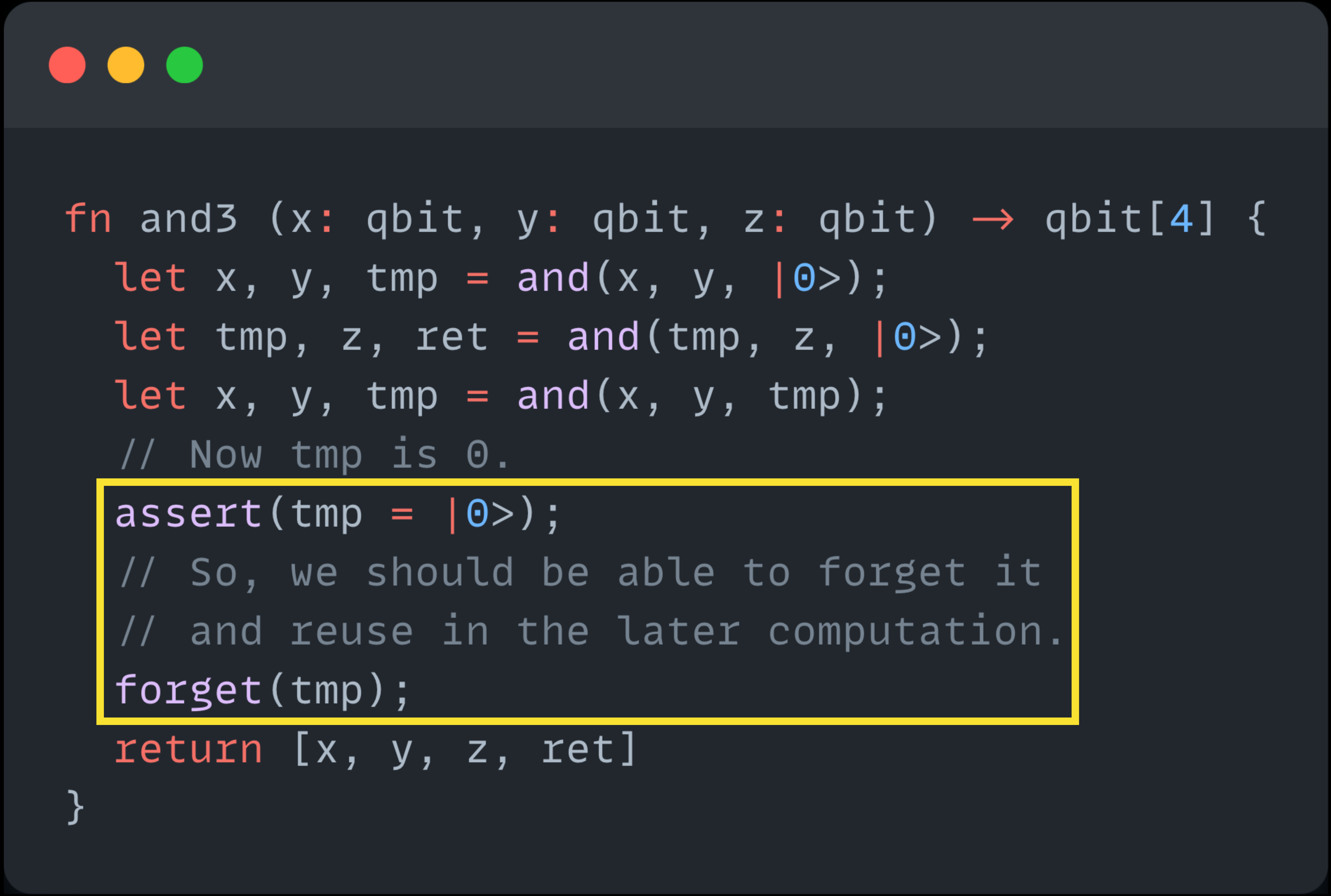
〃, $|0\rangle$, $x \wedge y \wedge z$

↧ measure
+ forget (bit)

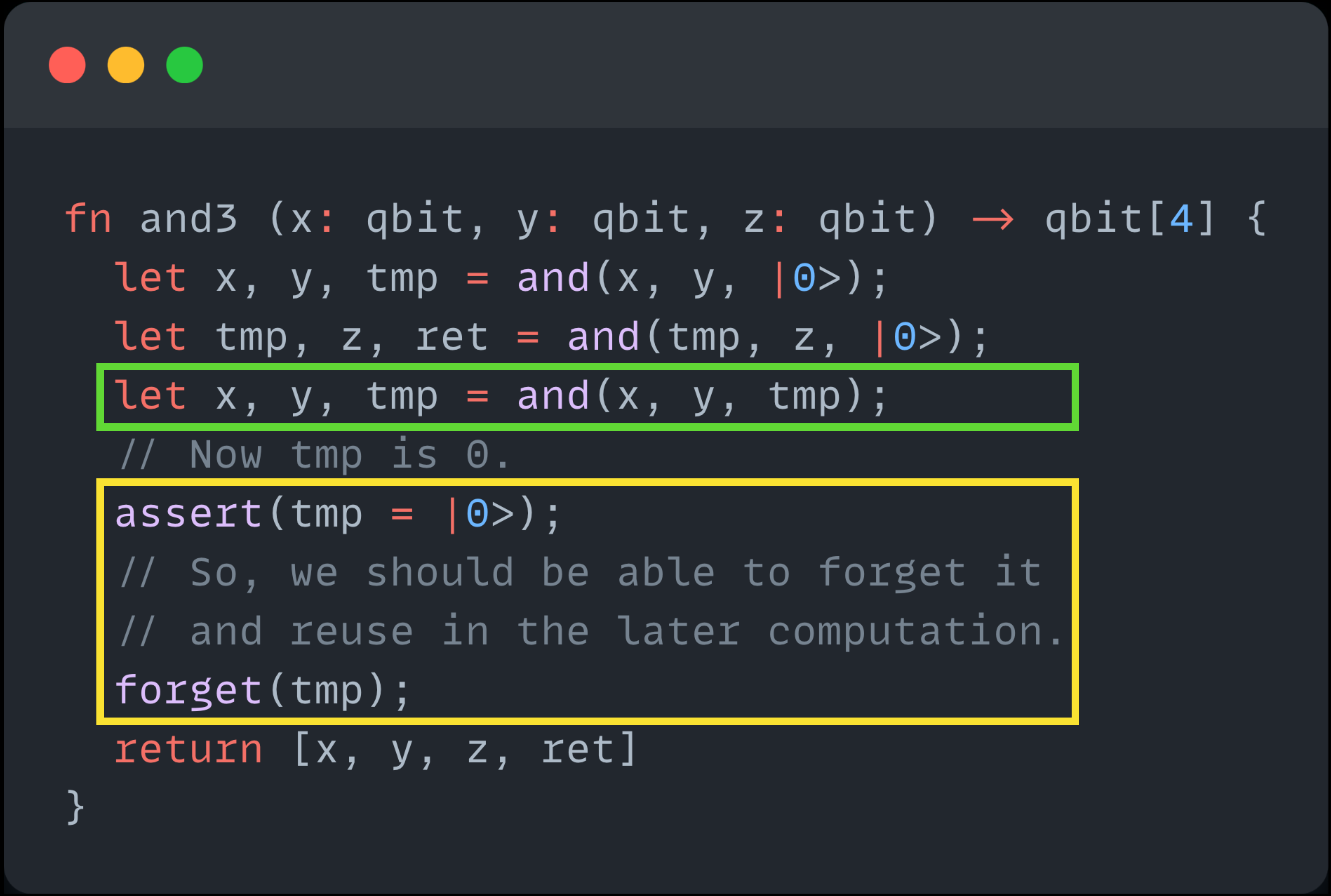
〃, $x \wedge y \wedge z$



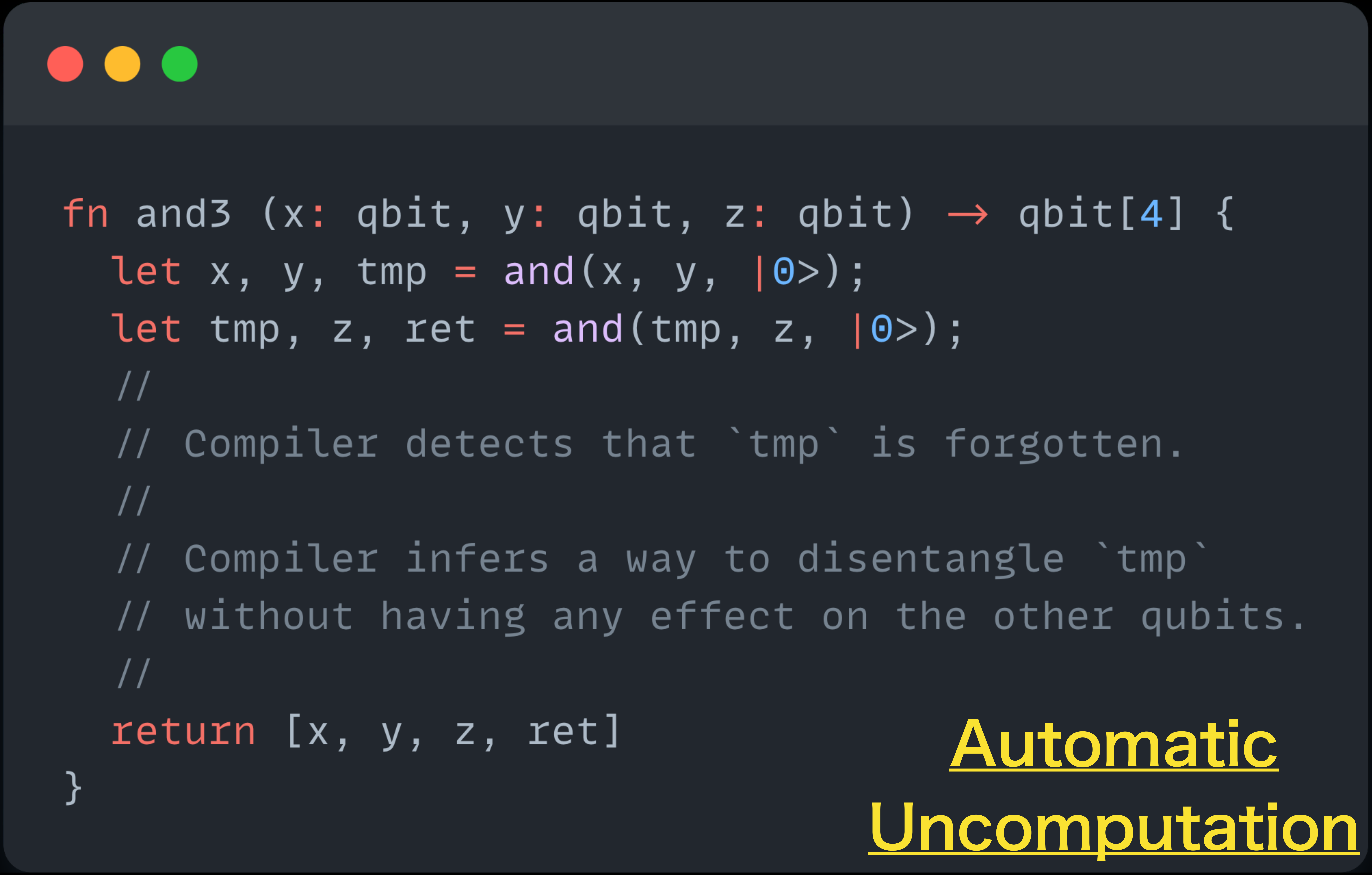
```
fn and3 (x: qbit, y: qbit, z: qbit) → qbit[4] {
  let x, y, tmp = and(x, y, |0>);
  let tmp, z, ret = and(tmp, z, |0>);
  let x, y, tmp = and(x, y, tmp);
  // Now, tmp is 0.
  // So, measuring it does nothing.
  measure(tmp);
  // tmp is now out of scope.
  return [x, y, z, ret]
}
```



```
fn and3 (x: qbit, y: qbit, z: qbit) → qbit[4] {
  let x, y, tmp = and(x, y, |0>);
  let tmp, z, ret = and(tmp, z, |0>);
  let x, y, tmp = and(x, y, tmp);
  // Now tmp is 0.
  assert(tmp = |0>);
  // So, we should be able to forget it
  // and reuse in the later computation.
  forget(tmp);
  return [x, y, z, ret]
}
```



```
fn and3 (x: qbit, y: qbit, z: qbit) -> qbit[4] {
  let x, y, tmp = and(x, y, |0>);
  let tmp, z, ret = and(tmp, z, |0>);
  let x, y, tmp = and(x, y, tmp);
  // Now tmp is 0.
  assert(tmp = |0>);
  // So, we should be able to forget it
  // and reuse in the later computation.
  forget(tmp);
  return [x, y, z, ret]
}
```



```
fn and3 (x: qbit, y: qbit, z: qbit) -> qbit[4] {
  let x, y, tmp = and(x, y, |0>);
  let tmp, z, ret = and(tmp, z, |0>);
  //
  // Compiler detects that `tmp` is forgotten.
  //
  // Compiler infers a way to disentangle `tmp`
  // without having any effect on the other qubits.
  //
  return [x, y, z, ret]
}
```

<u>Automatic Uncomputation</u>

Automatic Uncomputation

- ✦ Uncomputation: A way to drop resources “safely”
- ✦ Automatic Uncomputation

```
let x, y, tmp = and(x, y, tmp);
```

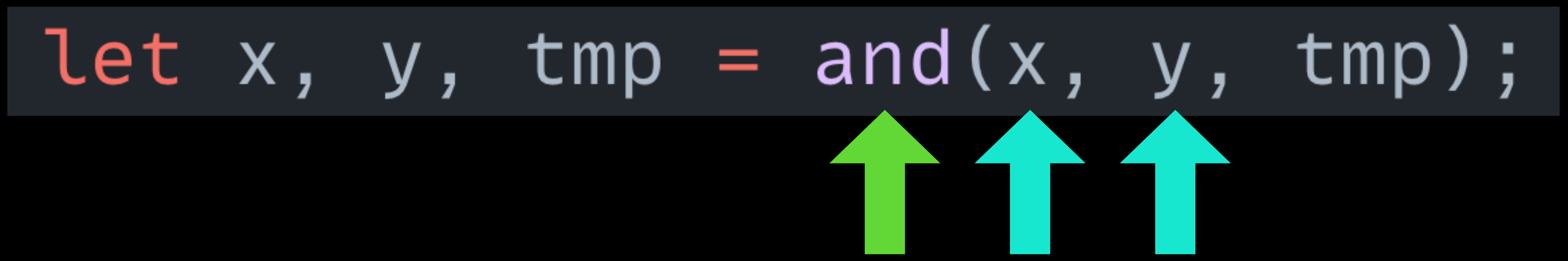
1. Detect drop
2. Check easily “uncompute-able”?
3. Do uncomputation or give up

Automatic Uncomputation

✦ Uncomputation: A way to drop resources “safely”

✦ Automatic Uncomputation

1. Detect drop
2. Check easily “uncompute-able”?
    - **x**, **y** still there?
    - Lift of classical computation?
3. Do uncomputation or give up



```
let x, y, tmp = and(x, y, tmp);
```

Idea: Type System for Automatic Uncomputation

✦Special type #T: “droppable T”

and’(x: qbit, y: qbit) -> (qbit, qbit, #qbit)

✦Allows drops only until x, y change

Affine Type ≤ #T ≤ Linear Type

Use **lifetime** α in Rust

Our Work: Type System for Automatic Uncomputation

- $\#^{\alpha}$ T: “T droppable until α” + $\&^{\alpha}$ T: “immutable reference until α”

  and’(x: $\&^{\alpha}$ qbit, y: $\&^{\alpha}$ qbit) -> $\#^{\alpha}$ qbit

- Allows drops only until α ends

  Affine Type $\leq$ $\#^{\alpha}$ T $\leq$ Linear Type

$$\begin{cases} \alpha = \text{static} & \Rightarrow \quad \#^{\alpha}\ \mathrm{T} = \text{affine} \\ \alpha = \text{empty} & \Rightarrow \quad \#^{\alpha}\ \mathrm{T} = \text{linear} \end{cases}$$

Our Work: Type System for Automatic Uncomputation

✦ $\#^{\alpha}$ T: “T droppable until α” + $\&^{\alpha}$ T: “immutable reference until α”

and’(x: $\&^{\alpha}$ qbit, y: $\&^{\alpha}$ qbit) -> $\#^{\alpha}$ qbit

✦ Allows drops only until α ends

Affine Type ≤ $\#^{\alpha}$ T ≤ Linear Type

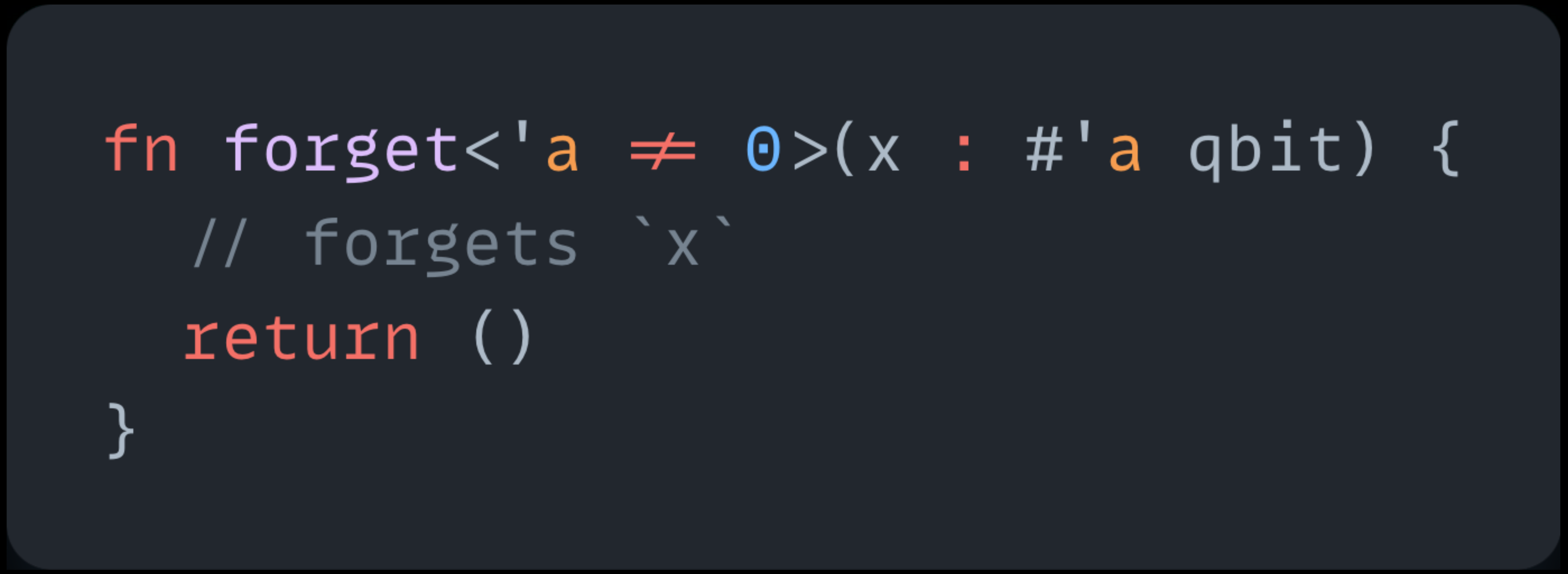
Automatic Uncomputation

⇒ Rust + affine type with lifetime $\#^{\alpha}$ T

Two Operational Semantics

✦Simulation Semantics

- Naive semantics on quantum & classical memory
- Circuit compilation ✗

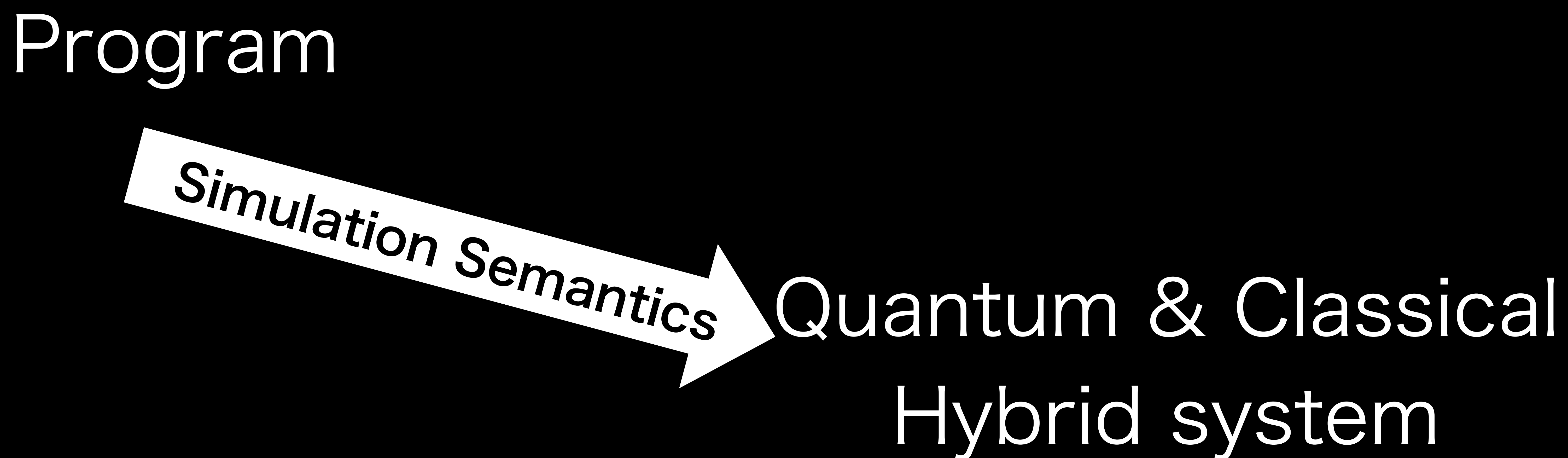


```
fn forget<'a ≠ 0>(x : #'a qbit) {
  // forgets `x`
  return ()
}
```

Two Operational Semantics

✦Simulation Semantics

- Naive semantics on quantum & classical memory
- Circuit compilation ✗



Program

Simulation Semantics

Quantum & Classical Hybrid system

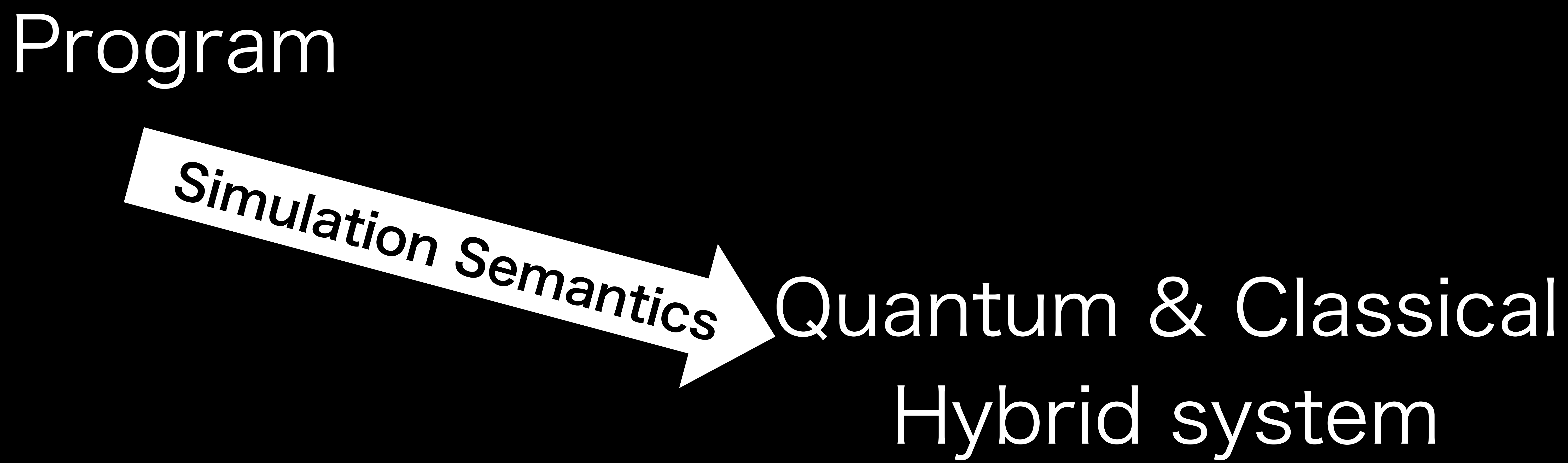
Two Operational Semantics

✦Simulation Semantics

- Naive semantics on quantum & classical memory
- Circuit compilation ✗

✦Uncomputation Semantics

- Program ⇒ circuit graphs
- Pebble game on graph = execution


Program

Simulation Semantics

Quantum & Classical Hybrid system

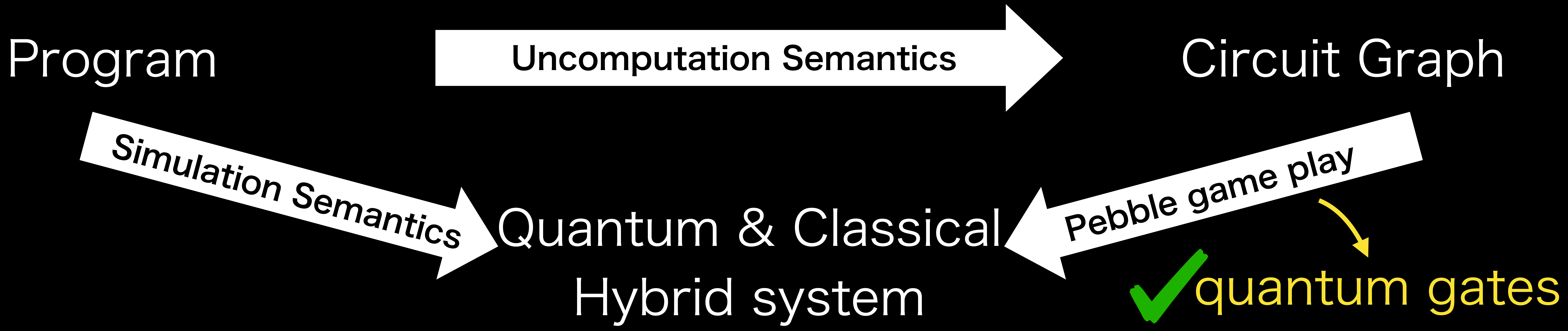
Two Operational Semantics

✦Simulation Semantics

- Naive semantics on quantum & classical memory
- Circuit compilation ✗

✦Uncomputation Semantics

- Program ⇒ circuit graphs
- Pebble game on graph = execution



Program

Uncomputation Semantics

Circuit Graph

Simulation Semantics

Quantum & Classical Hybrid system

Pebble game play

✓quantum gates

Related Work

✦Type system for Automatic Uncomputation

	Silq [Bichsel et al. PLDI ’20]	Qurts (Our Language)
Type system	Ad hoc	Rust + $\#^{\alpha}$T
Implementation	✓ CPU simulator	✗
Lifetime	✗	✓
forget	✗	✓

```
fn forget<'a ≠ 0>(x : #'a qbit) {
  // forgets `x`
  return ()
}
```

Summary of Contribution

✦New type system for Automatic Uncomputation

- Rust + $\#^{\alpha}$ T (affine type with lifetime)
- Many examples

✦Two semantics

- Proved equivalence of the two semantics
- Uncomputation semantics uses pebble games