Reasoning About Multi-Stage Programs

Jun Inoue[1], Walid Taha[1,2]

[1]Rice University
[2]Halmstad University

WG2.11, June 2012

Abstraction vs. Performance

- matrix_mul : int matrix → int matrix → int matrix
- matrix_mul : $\forall\,\alpha.\ (\alpha \to \alpha \to \alpha) \to (\alpha \to \alpha \to \alpha)$

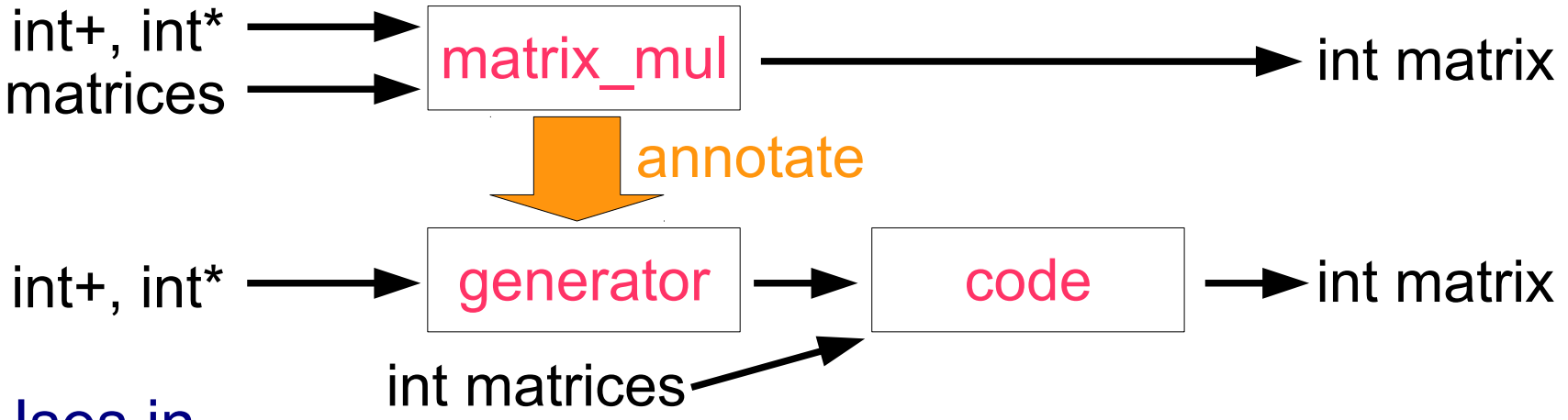
  $\to \alpha$ matrix $\to$ α matrix $\to$ α matrix

The polymorphic one is more *reusable*.

But the monomorphic version is *faster*.

The Idea of MSP

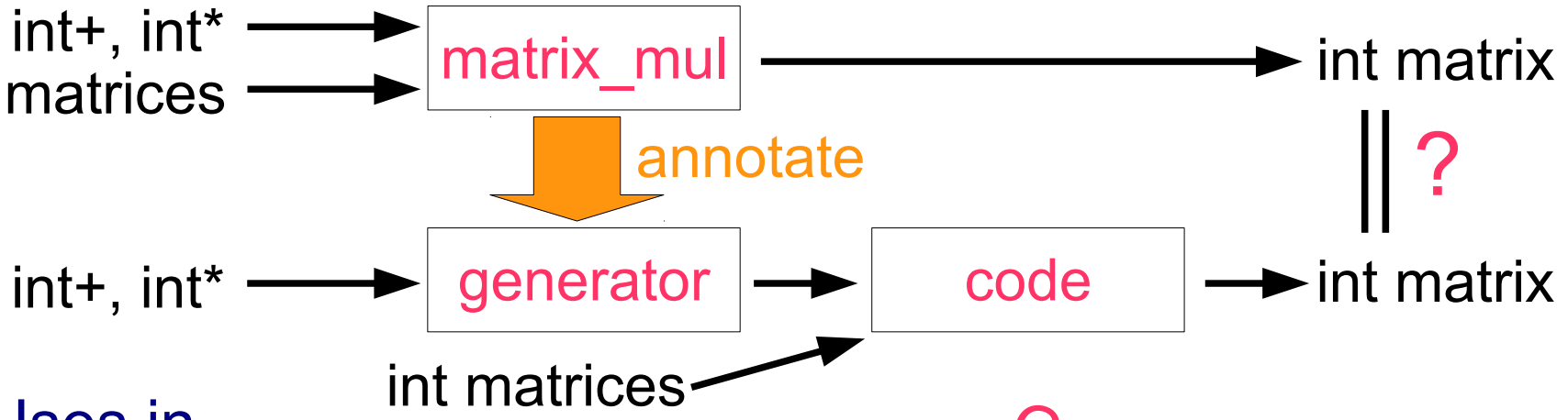
- Staging (multi-stage programming, MSP)



- Uses in
  - Numerical computation [Carette08]
  - DSL implementation [Czarnecki04, Brady06]
  - Circuit generation [Kiselyov04, Chen10]

The Idea of MSP

- Staging (multi-stage programming, MSP)



- Uses in
  - Numerical computation [Carette08]
  - DSL implementation [Czarnecki04, Brady06]
  - Circuit generation [Kiselyov04, Chen10]

Correctness missing

Contributions

In the multi-stage λ calculus λ-U [Taha99], we give:

- Conditions for annotations to preserve semantics

- Applicative bisimulation
- How reasoning in λ-U differs from reasoning in λ
- Case Study: LCS with monadic let-insertion

Running Example: Power

```
let rec  power n x =
  if n = 1 then x
  else   x *  ( power (n-1) x)
```

What we can do

```
let square = λx. power 2 x
let cube   = λx. power 3 x
```

Running Example: Power

```
let rec  power n x =
  if n = 1 then x
  else   x *  ( power (n-1) x)
```

What we want

```
let square = λx. x*x
let cube   = λx. x*x*x
```

Running Example: Power

```
let rec  power n x =
  if n = 1 then x
  else   x *  ( power (n-1) x)
```

Staging annotations:

```
Eval(2+3)    = 5
Eval(<2+3>) = <2+3>

If Eval(f()) = <2+3> then
  Eval(<1 + ~(f())>) = <1 + (2+3)>

Eval(!<2+3>) = 5
```

Running Example: Power

```
let rec genpow n x =
  if n = 1 then x
  else <~x * ~(genpow (n-1) x)>
let stpow n =
  !<λz. ~(genpow n <z>)>
```

Evaluation traces:

```
genpow 1 <z> → <z>
genpow 2 <z> → <z*z>
genpow 3 <z> → <z*z*z>
stpow 2      → !<λz. z*z>  →  λz. z*z
```

Running Example: Power

```
let rec genpow n x =
  if n = 1 then x
  else <~x * ~(genpow (n-1) x)>
let stpow n =
  !<λz. ~(genpow n <z>)>

let rec power n x =
  if n = 1 then x
  else x * power (n-1) x
```

power is the *erasure* of stpow (written ||stpow||)

Running Example: Power

```
let rec genpow n x =
  if n = 1 then x
  else   x *  (genpow (n-1) x)
let stpow n =
    λz.  (genpow n  z )

let rec power n x =
  if n = 1 then x
  else x * power (n-1) x
```

power is the *erasure* of stpow (written ||stpow||)

The Question

- Staging should preserve semantics!
  - stpow *n k* = power *n k*
  - more generally, *e* = ||*e*||
- ...except it doesn't always
  - <loop ()> ≠ loop ()
  - !<λ_. ~(loop ())> ≠ λ_. loop ()

When do we know *e* = ||*e*||?

Staging Changes Evaluation Strategy

with staging

```
stpow 3 5
```

```
!<λz.~(genpow 3 <z>)> 5
```

```
(λz.z*z*z) 5
```

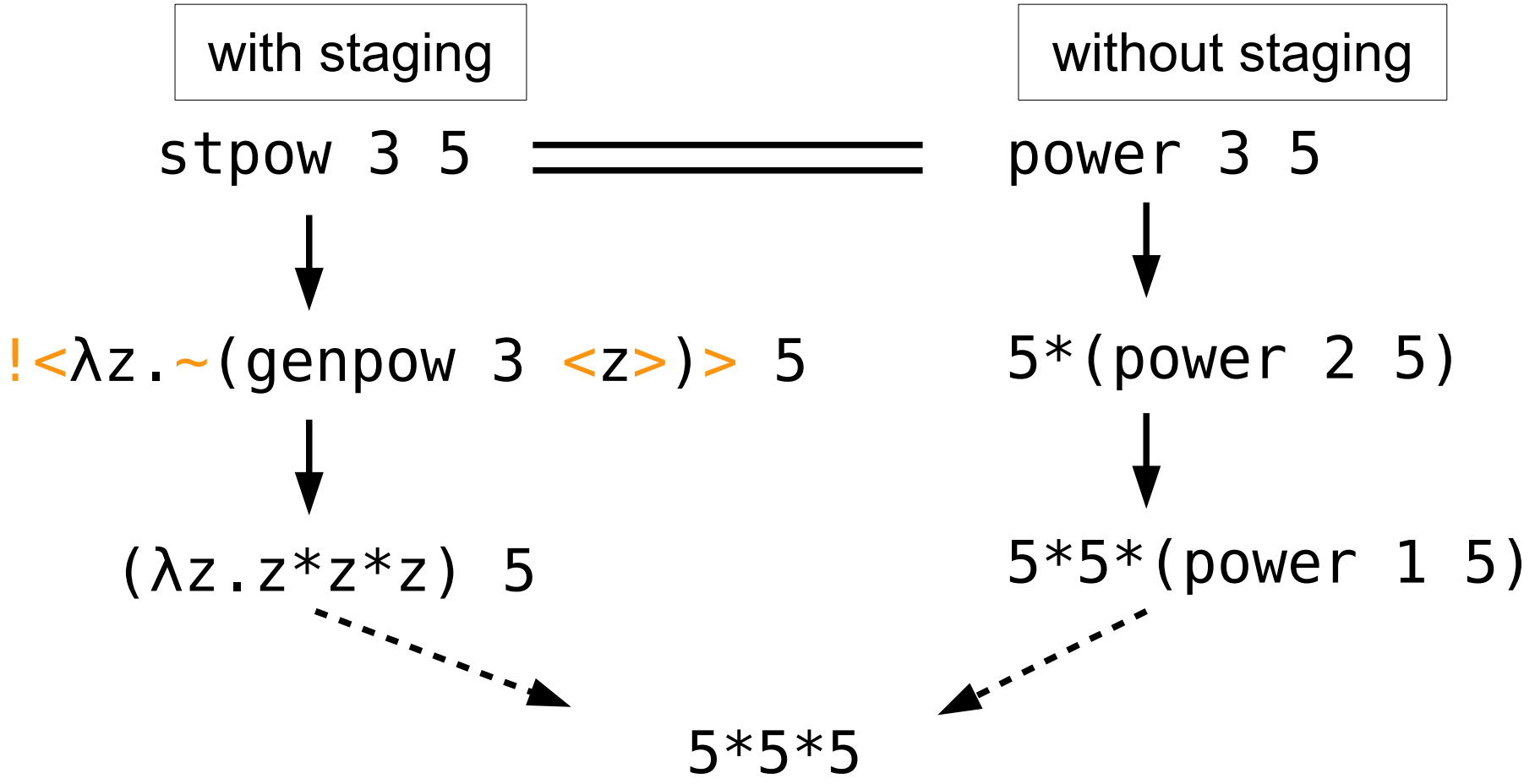
without staging

```
power 3 5
```

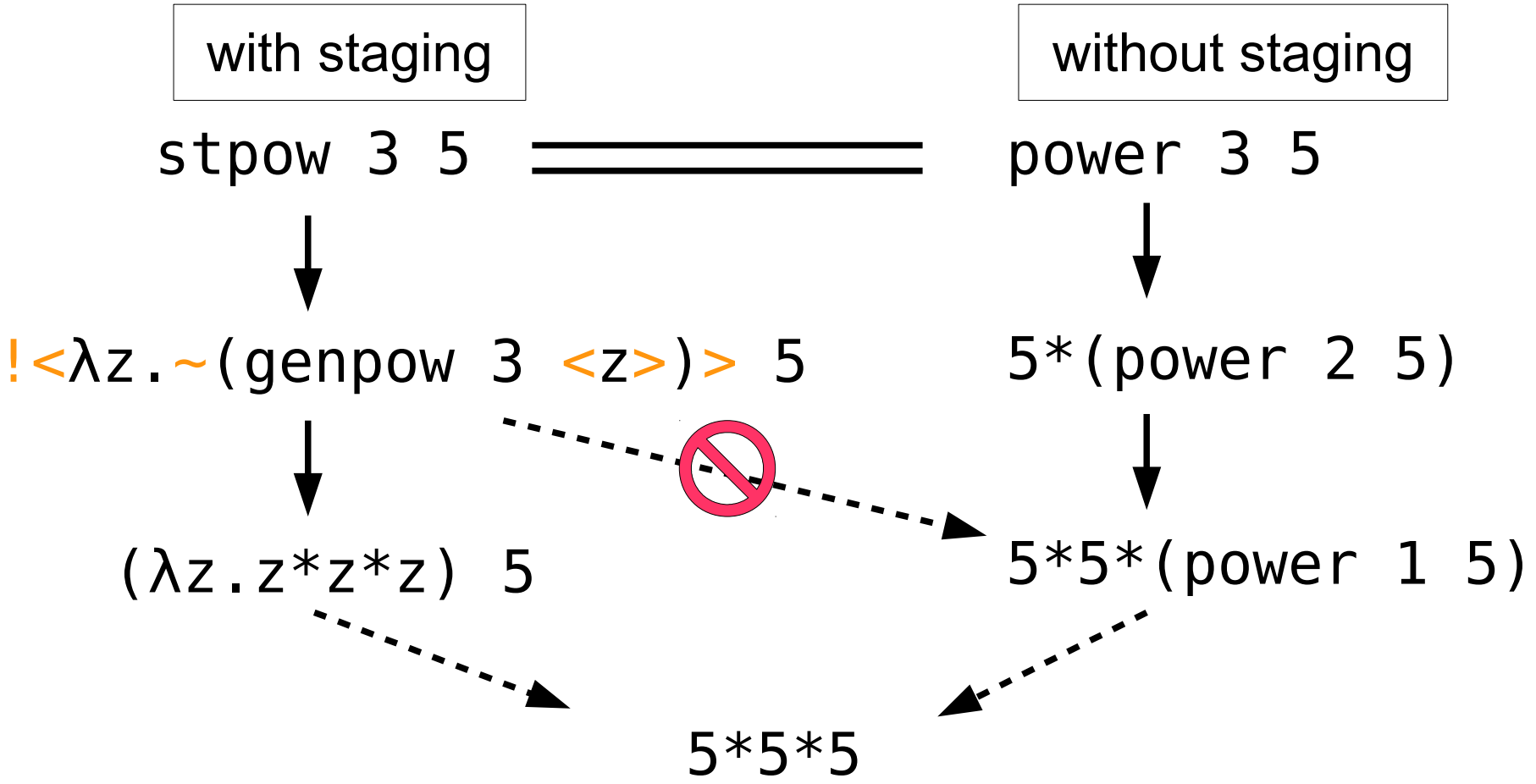
```
5*(power 2 5)
```

```
5*5*(power 1 5)
```

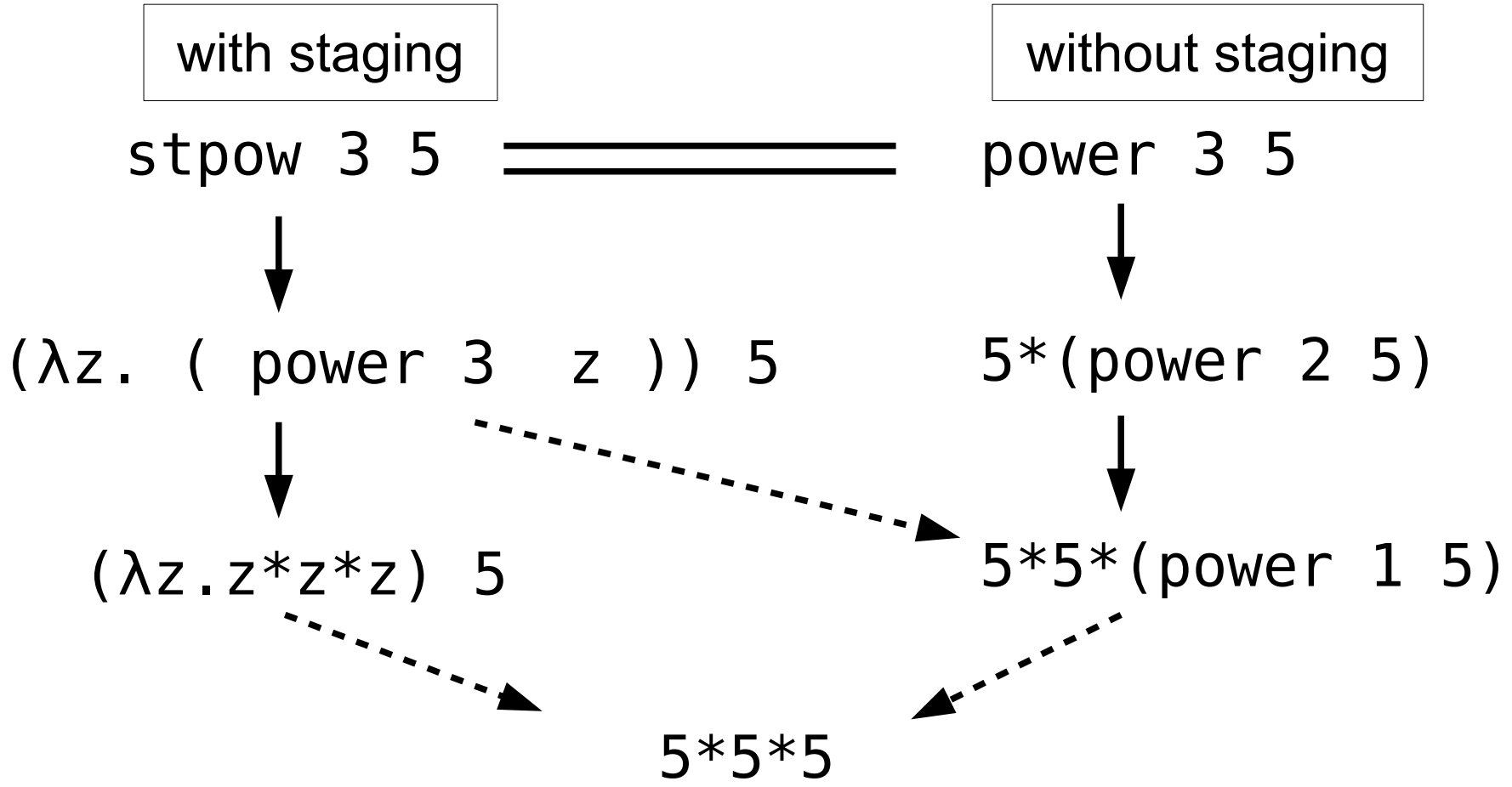
Staging Changes Evaluation Strategy



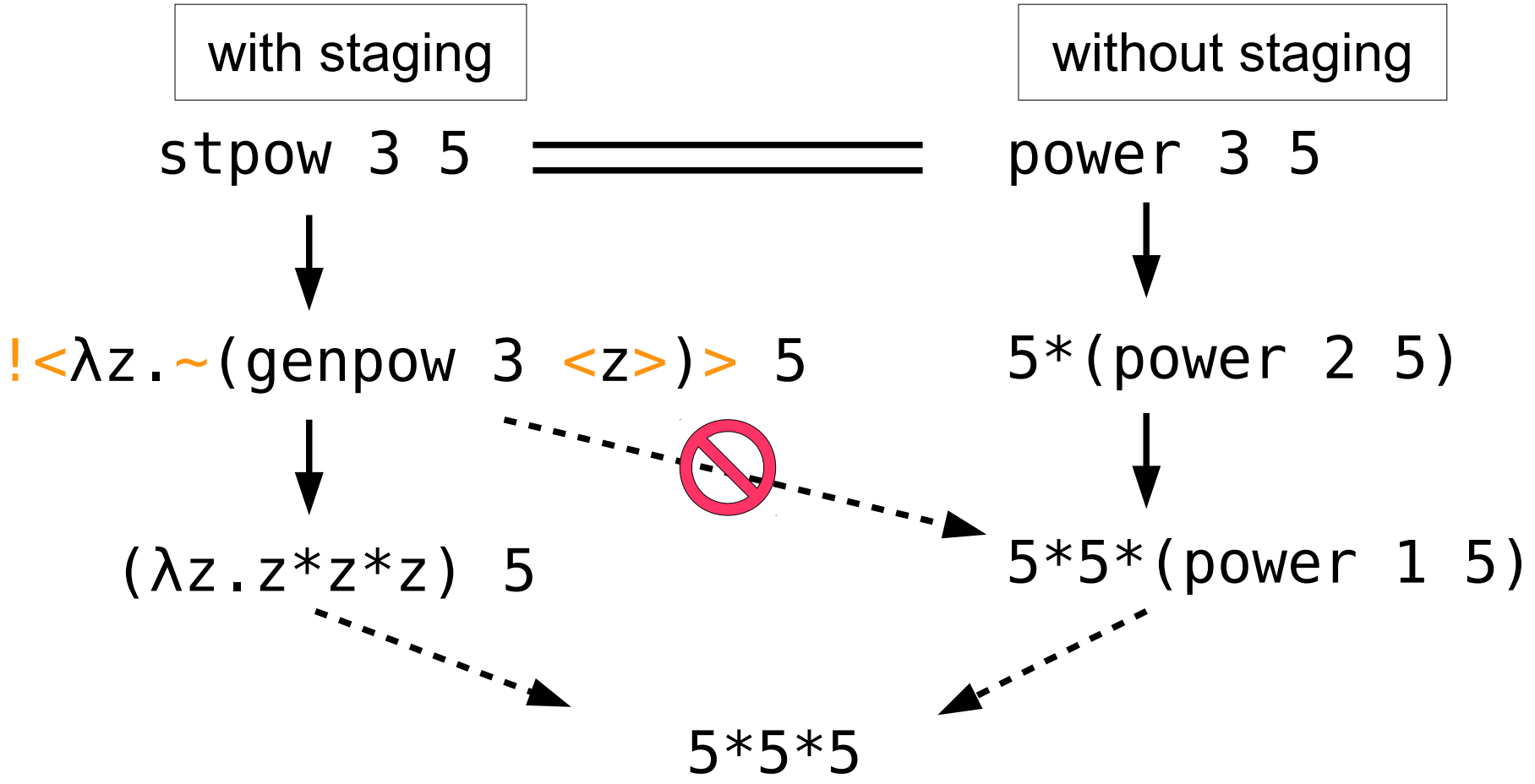
Staging Changes Evaluation Strategy



Staging Changes Evaluation Strategy



Staging Changes Evaluation Strategy



Termination Condition

Theorem. If $e \rightarrow ||t||$ for some $||t||$, then $e = ||e||$

```
let rec genpow n x =
  if n = 1 then x
  else <~x * ~(genpow (n-1) x)>
let stpow n =
  !<λz. ~(genpow n <z>)>
```

Proof sketch of stpow n = power n:

- genpow 1 <z> → <z> call this <||*e*1||>
- genpow 2 <z> → <z * ||*e*1||> call this <||*e*2||> etc.
- stpow n → !<λz. ||*en*||> → λz. ||*en*||

Termination Condition

Theorem. If $e \rightarrow ||t||$ for some $||t||$, then $e = ||e||$

```
let rec genpow n x =
  if n = 1 then x
  else <~x * ~(genpow (n-1) x)>
let stpow n =
  !<λz. ~(genpow n <z>)>
```

Doesn't matter what these $||en||$ look like!

Proof sketch of stpow n = power n:

- genpow 1 <z> → <z> call this <||e1||>
- genpow 2 <z> → <z * ||e1||> call this <||e2||> etc.
- stpow n → !<λz. ||*en*||> → λz. ||*en*||

Termination Condition for CBN

Theorem. If $e \rightarrow ||t||$ for some $||t||$, then $e = ||e||$

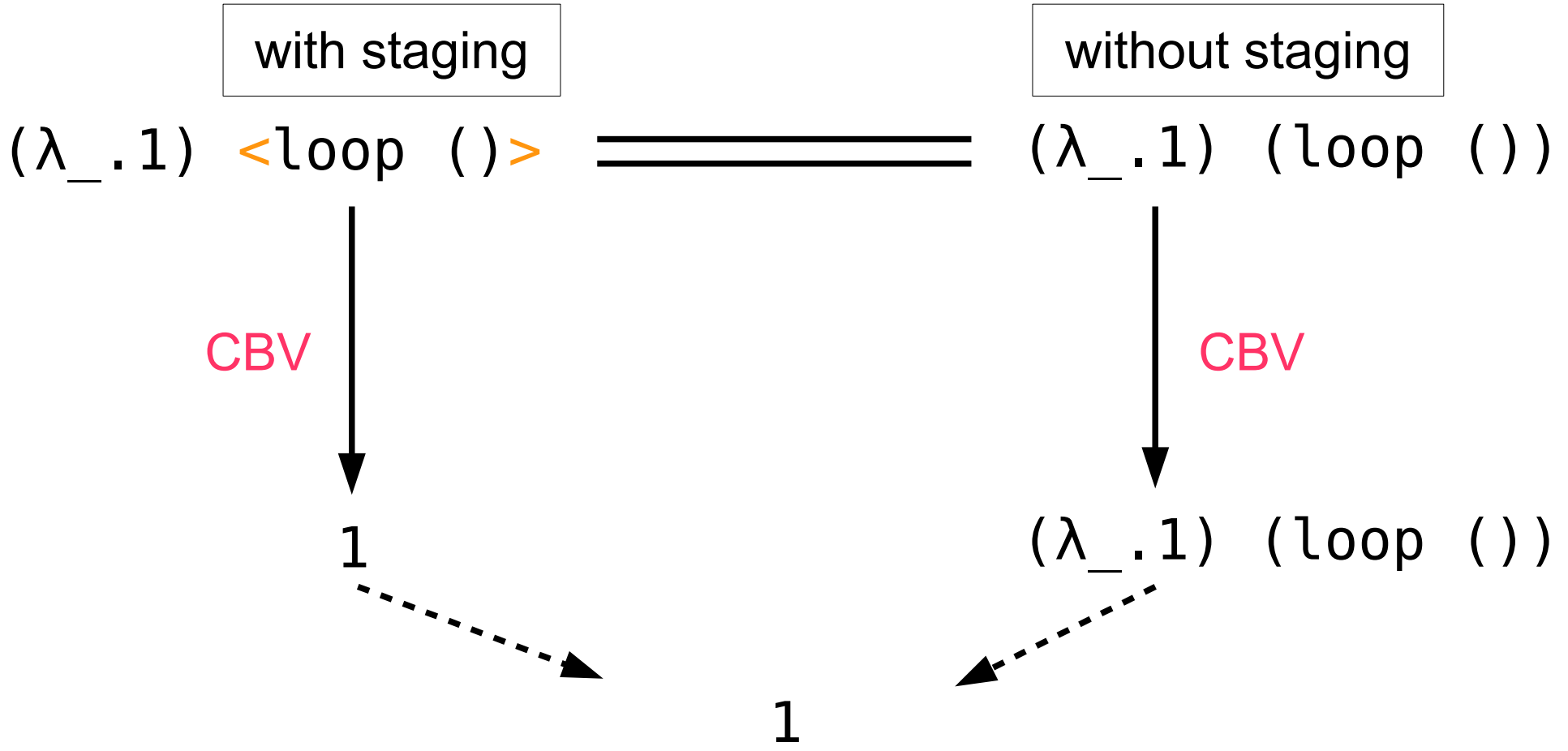
```
let rec genpow n x =
  if n = 1 then x
  else <~x * ~(genpow (n-1) x)>
let stpow n =
  !<λz. ~(genpow n <z>)>
```

Doesn't matter what these $||en||$ look like!

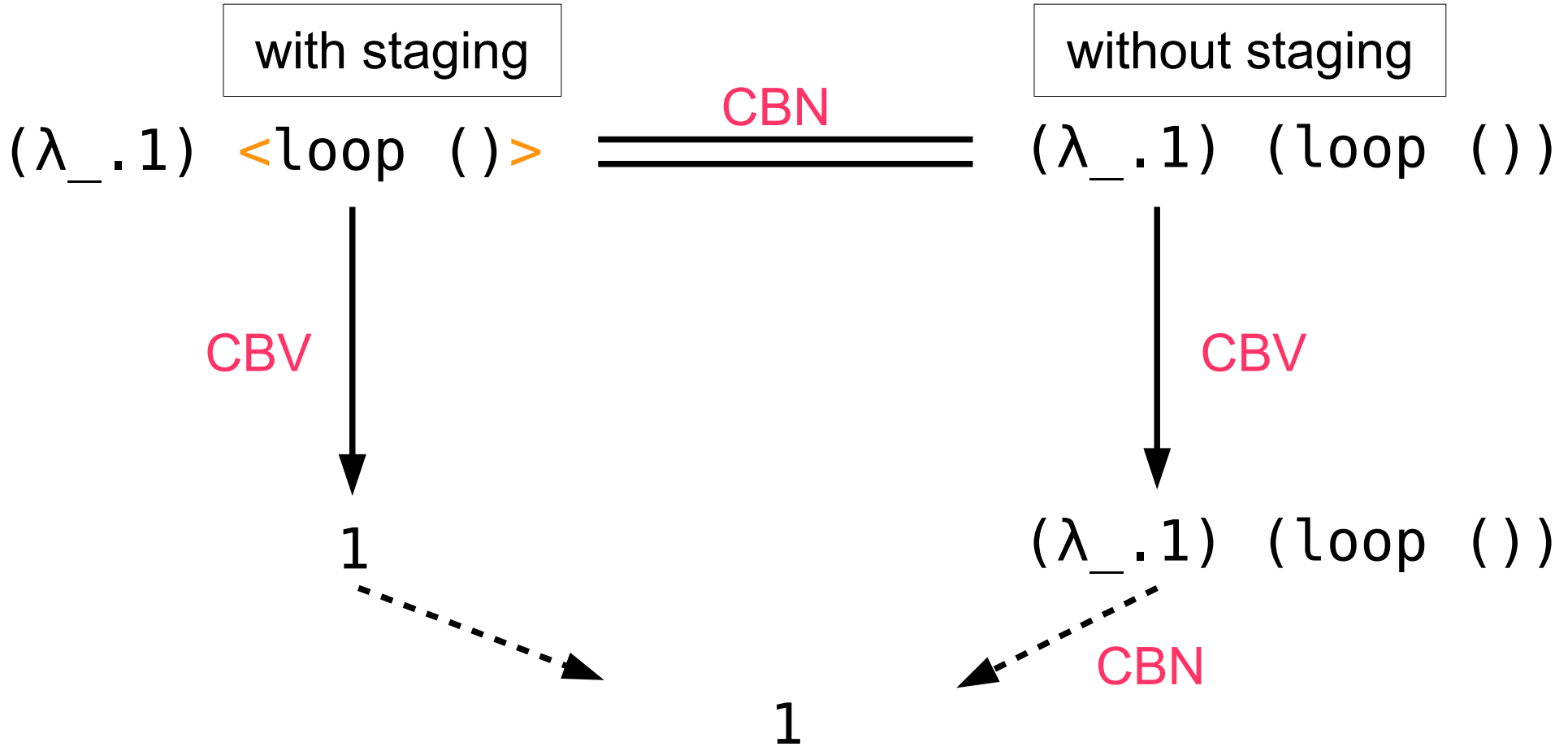
Proof sketch of stpow n = power n:

- genpow 1 <z> → <z> call this <||e1||>
- genpow 2 <z> → <z * ||e1||> call this <||e2||> etc.
- stpow n → !<λz. ||*en*||> → λz. ||*en*||

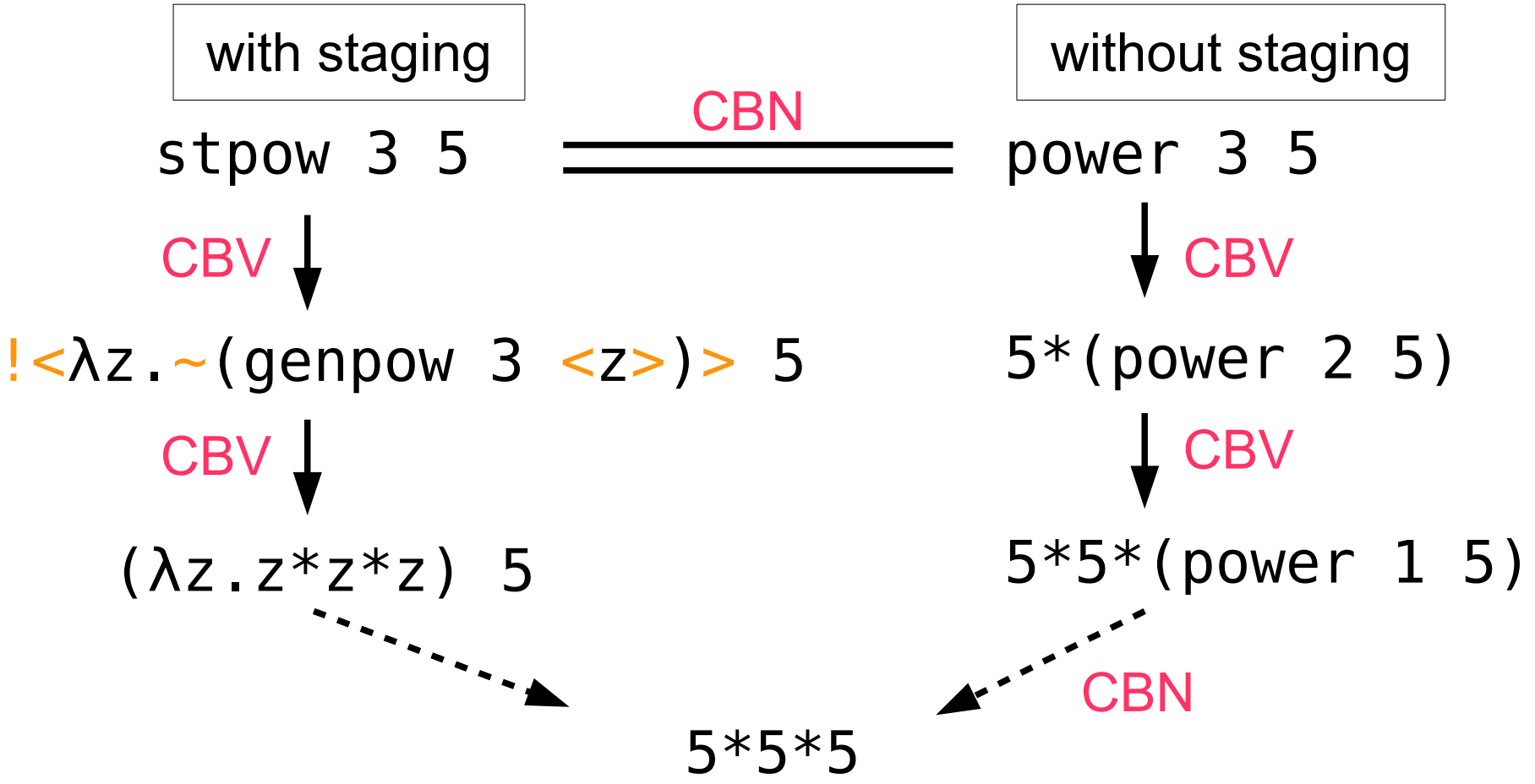
In CBV, Staging Can Enforce CBN



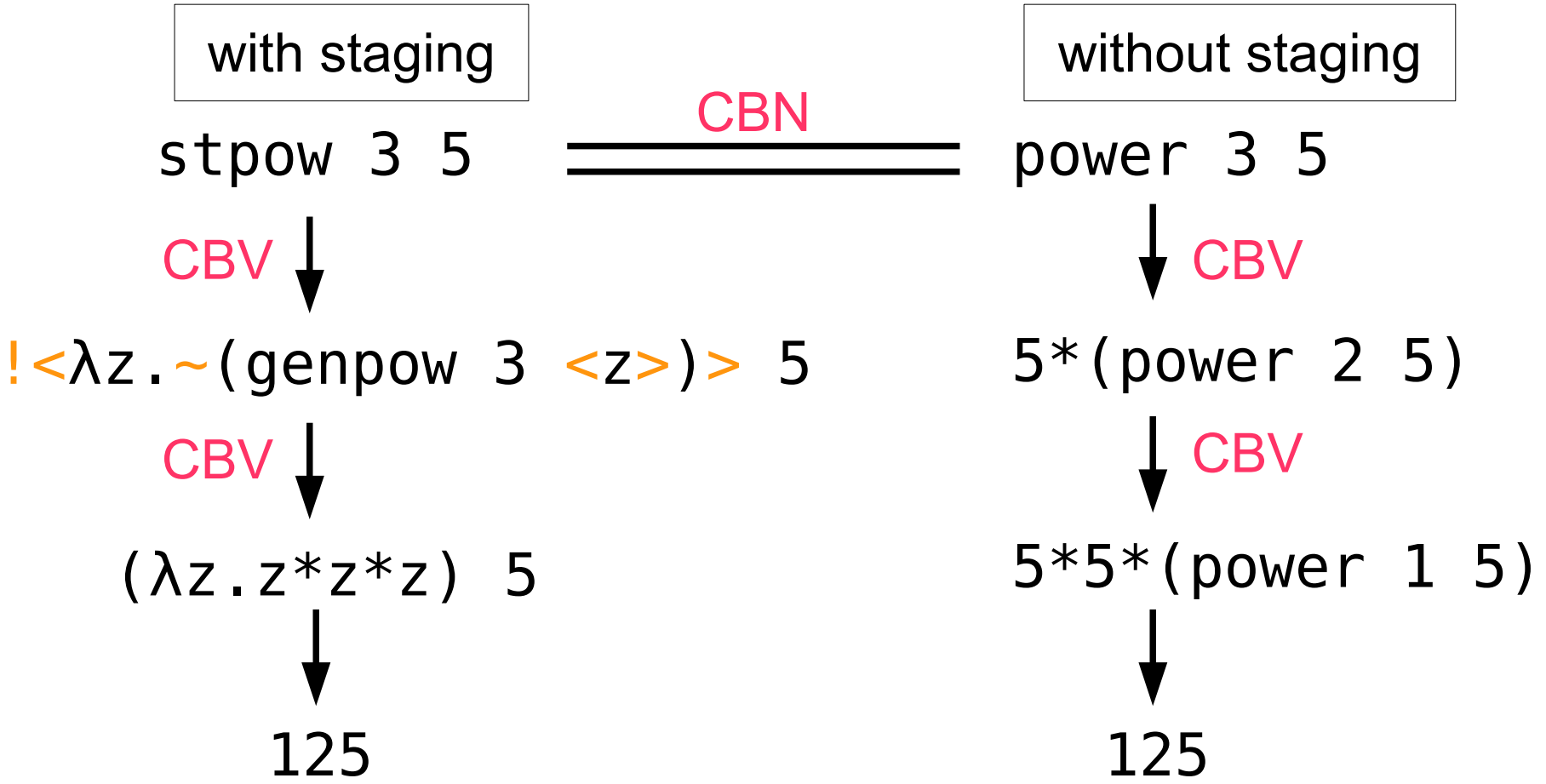
In CBV, Staging Can Enforce CBN



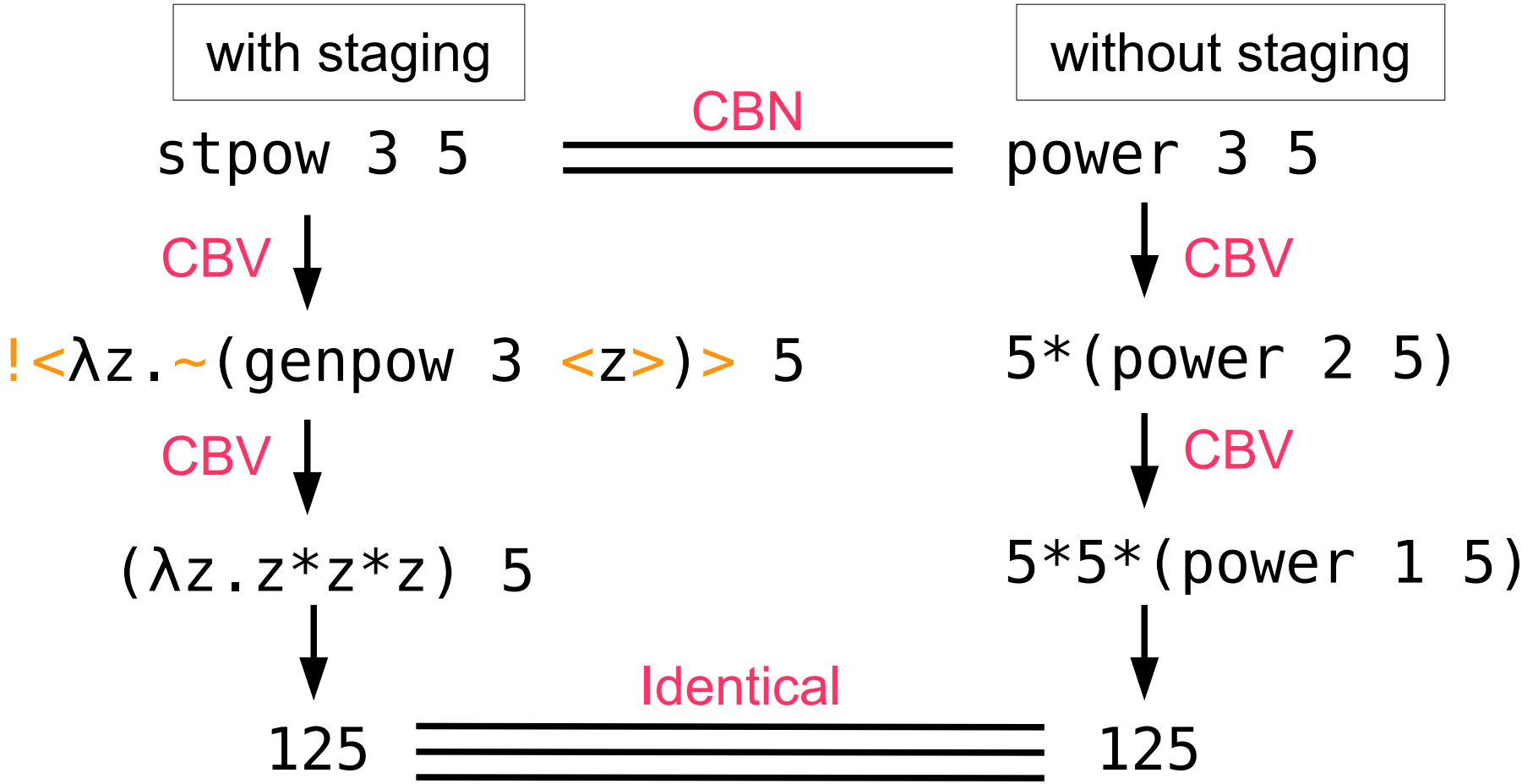
In CBV, Staging Can Enforce CBN



In CBV, Staging Can Enforce CBN



In CBV, Staging Can Enforce CBN



Termination Condition for CBV

Theorem. If $e \to c$ and $||e|| \to d$ for first-order constants c and d, then $e = ||e||$

```
let rec genpow n x =
  if n = 1 then x
  else <~x * ~(genpow (n-1) x)>
let stpow n =
  !<λz. ~(genpow n <z>)>
```

Proof sketch of stpow n k = power n k:

- genpow n <z> = <||*en*||>
- stpow n k = (λz. ||*en*||) k
- Check (λz. ||*en*||) k returns an integer

Conclusion & Future Work

- MSP gives abstraction + performance,

  now also with *correctness*!

  - Make sure your generator terminates

    + has no run-away annotations

  - For CBV, ensure generated code terminates
- Future work: do this with strong effects

  - Have type systems, but not reasoning device
    [Kameyama09, Westbrook10, Rhiger12]
  - Separation logic & geometric semantics?