Polynomial Time Inverse Computation for Accumulative Functions with Multiple Data Traversals

Kazutaka Matsuda (Tohoku University)
Kazuhiro Inaba (National Institute of Informatics*)
Keisuke Nakano (the University of Electro-Communications)

*Currently he is working at Google

Jan. 23rd, 2012 @ PEPM 2012

Inverse Computation

Problem: Inverse Computation

Given a program `f` and its output `t`, find all `s` such that `f(s) = t`.

- For `add(_,Z)` and `S(Z)`, inv-comp returns `S(Z)`

```
add(Z,y)      = y
add(S(x),y) = S(add(x,y))
```

Applications

- Deriving deserializer from serializer
- Supporting Undo
- Efficient test-case generation

[Runciman+ 08, Christiansen&Fischer 08]

- What x makes pred(x) true?

This Talk

- We propose an inv-comp method ...
  - works for a class of functions called parameter-linear macro tree transducers
    - **accumulative functions**
    - **multiple data traversals**

    } **difficult to handle**

- runs in time **polynomial** to the size of a given output

This Talk

- We propose an inv-comp method ...
  - works for a class of functions called parameter-linear macro tree transducers
    - **accumulative functions**
    - **multiple data traversals**

  } **difficult to handle**

But, why?

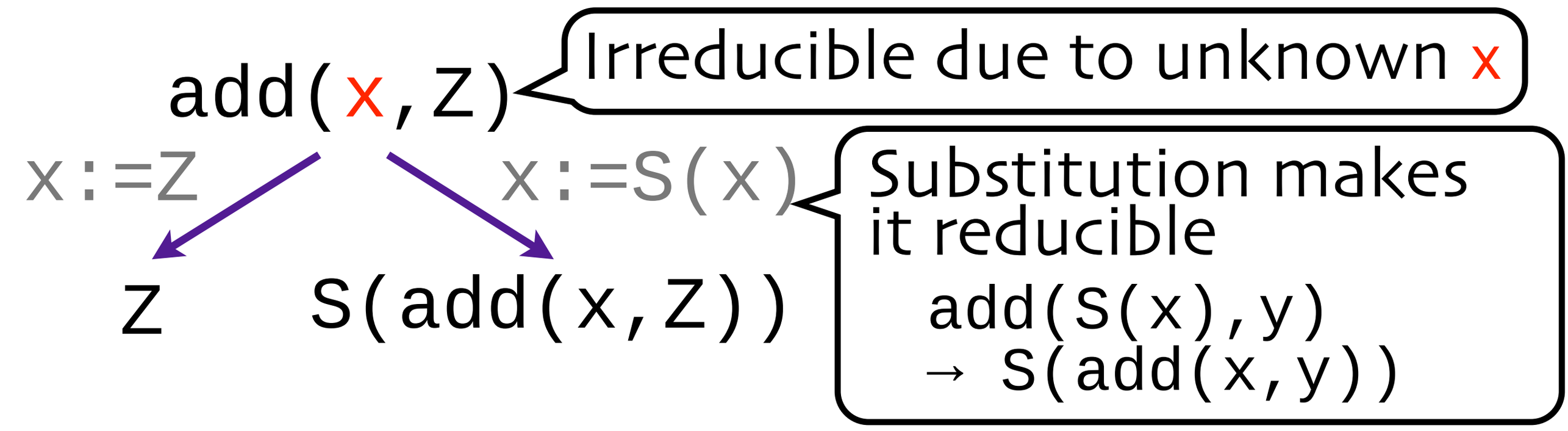
- runs in time **polynomial** to the size of a given output

Review: Existing Method

- An existing inv-comp method
  - known as
    - (Needed) **Narrowing** [Antoy+ 00]
    - URA [Abramov&Glück 02]
    - SLD resolution
  - used in logic programming languages
    - Curry, Prolog
  - based on symbolic computation

(Needed) Narrowing

- A symbolic computation
  - substitution followed by reduction




```
add(Z,y)    = y
add(S(x),y) = S(add(x,y))
```

InvComp = Narrowing+EqChk

```
add(Z,y)      = y
add(S(x),y) = S(add(x,y))
```

```
add(x,Z)==S(Z)
```

InvComp = Narrowing+EqChk

```
add(Z,y)      = y
add(S(x),y) = S(add(x,y))
```

add(x,Z)==S(Z)

x:=Z

InvComp = Narrowing+EqChk

```
add(Z,y)      = y
add(S(x),y) = S(add(x,y))
```

add(x,Z)==S(Z)

x:=Z

Z==S(Z)

InvComp = Narrowing+EqChk

```
add(Z,y)      = y
add(S(x),y) = S(add(x,y))
```

add(x,Z)==S(Z)

x:=Z

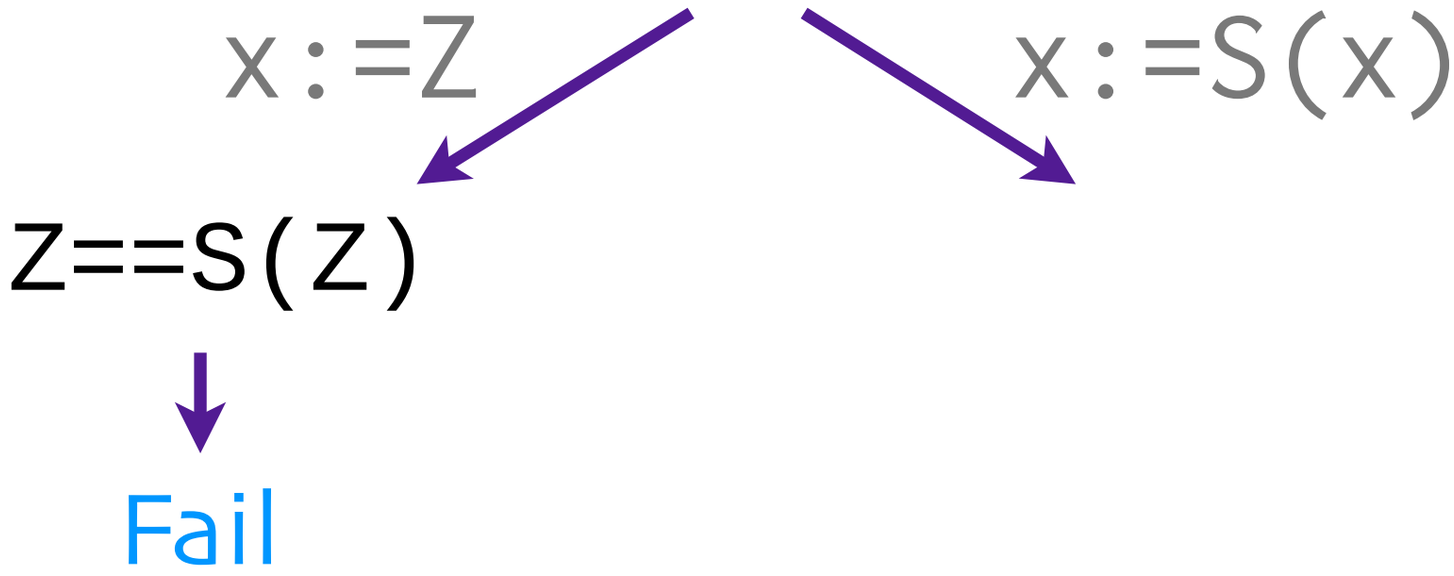
Z==S(Z)

↓

Fail

InvComp = Narrowing+EqChk

```
add(Z,y)      = y
add(S(x),y) = S(add(x,y))
```



InvComp = Narrowing+EqChk

```
add(Z,y)      = y
add(S(x),y) = S(add(x,y))
```

add(x,Z)==S(Z)

x:=Z

Z==S(Z)

↓

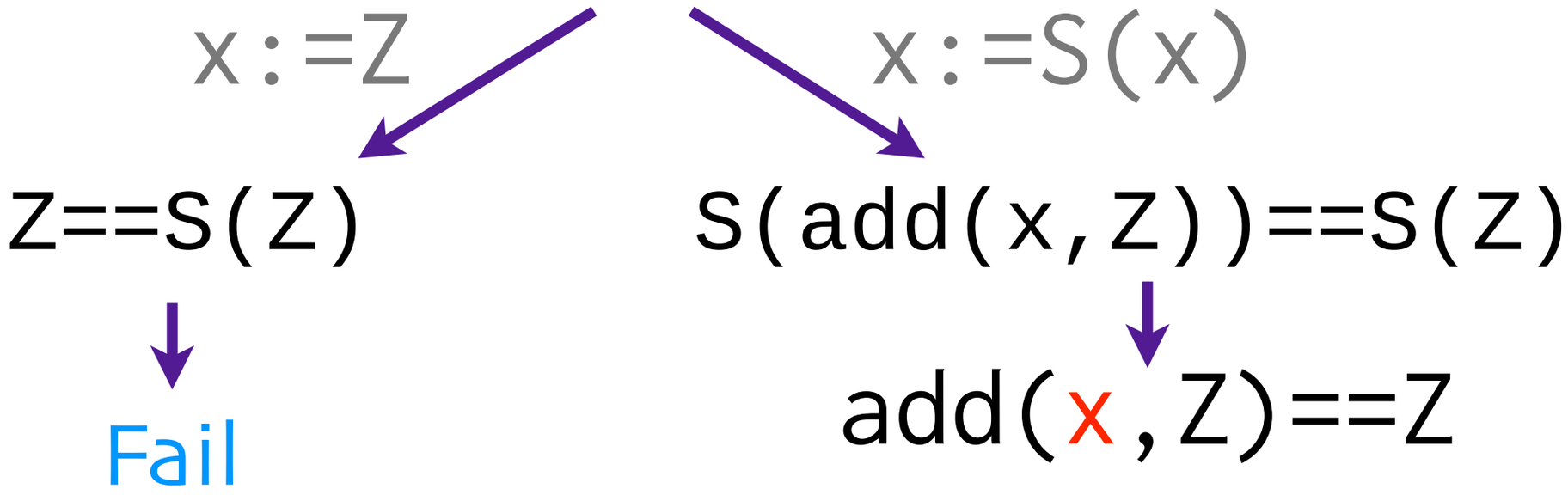
Fail

x:=S(x)

S(add(x,Z))==S(Z)

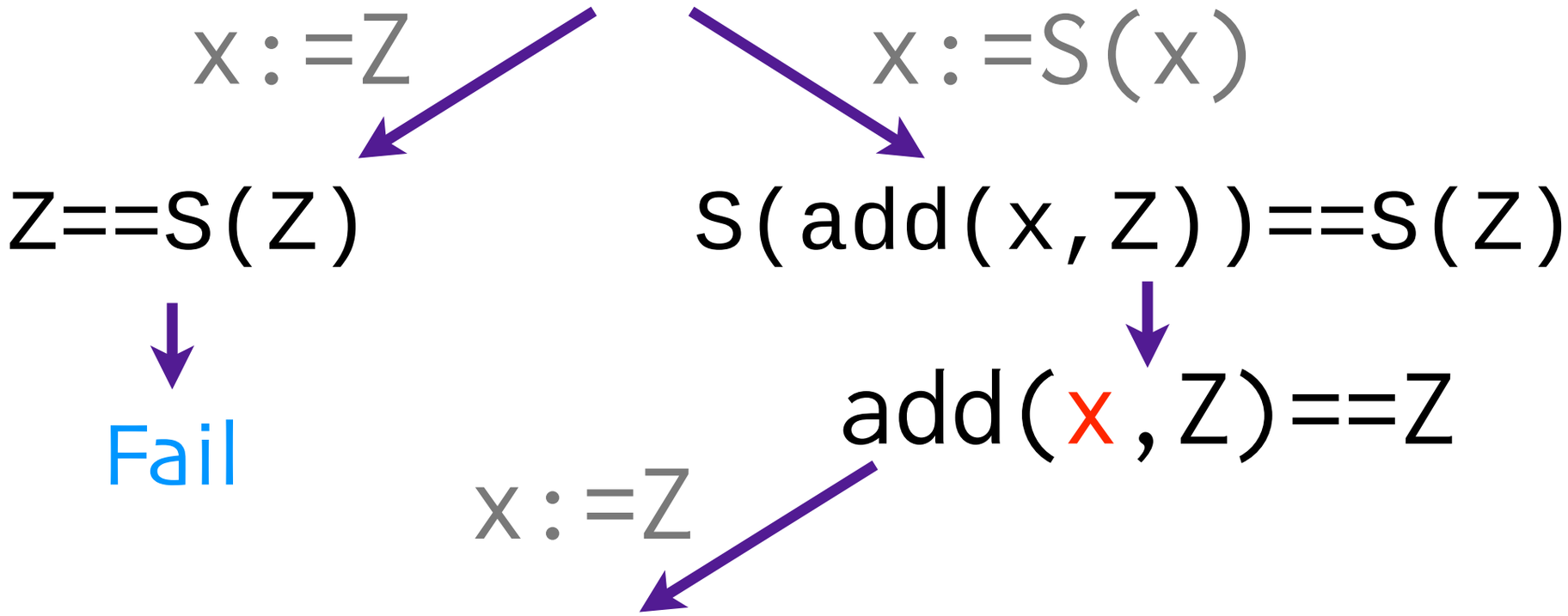
InvComp = Narrowing+EqChk

```
add(Z,y)      = y
add(S(x),y) = S(add(x,y))
```



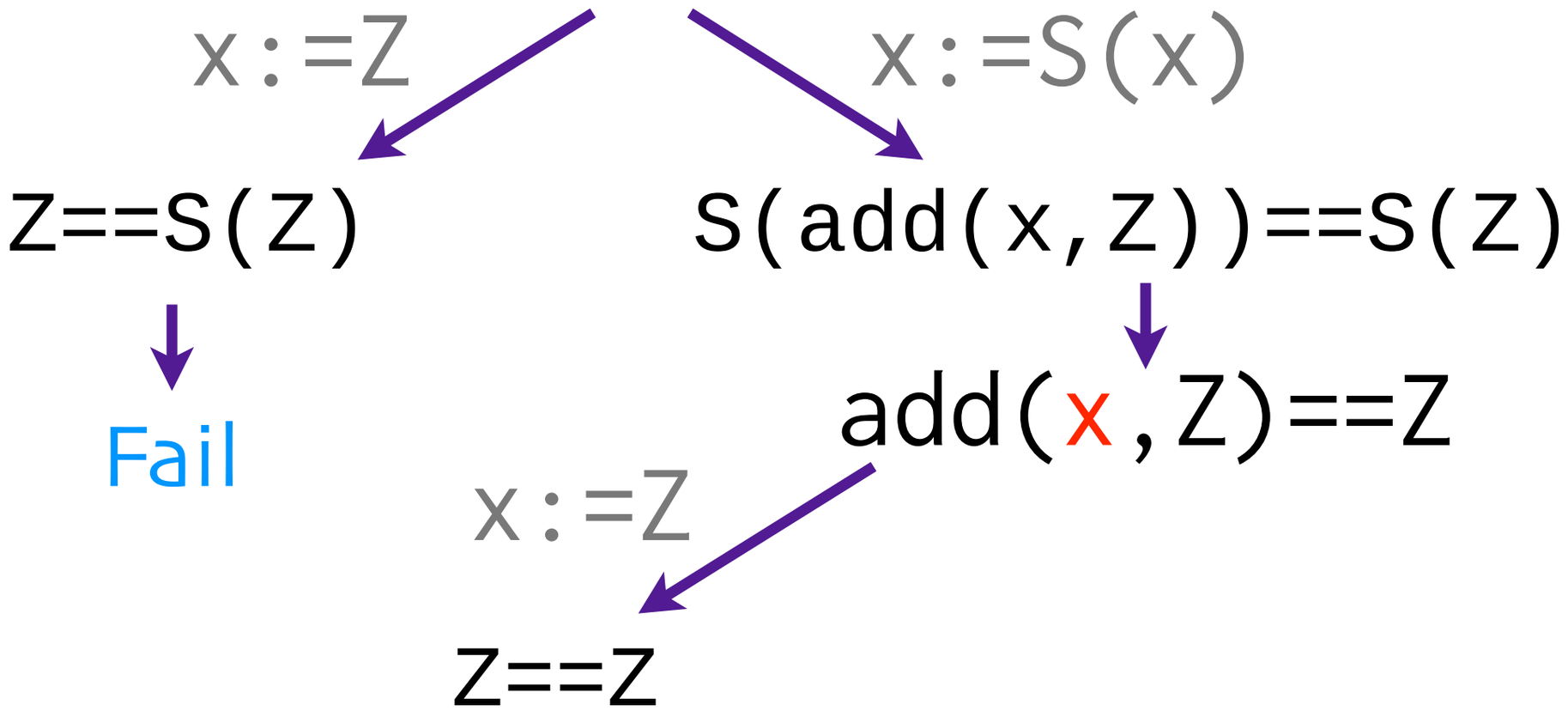
InvComp = Narrowing+EqChk

```
add(Z,y)      = y
add(S(x),y) = S(add(x,y))
```



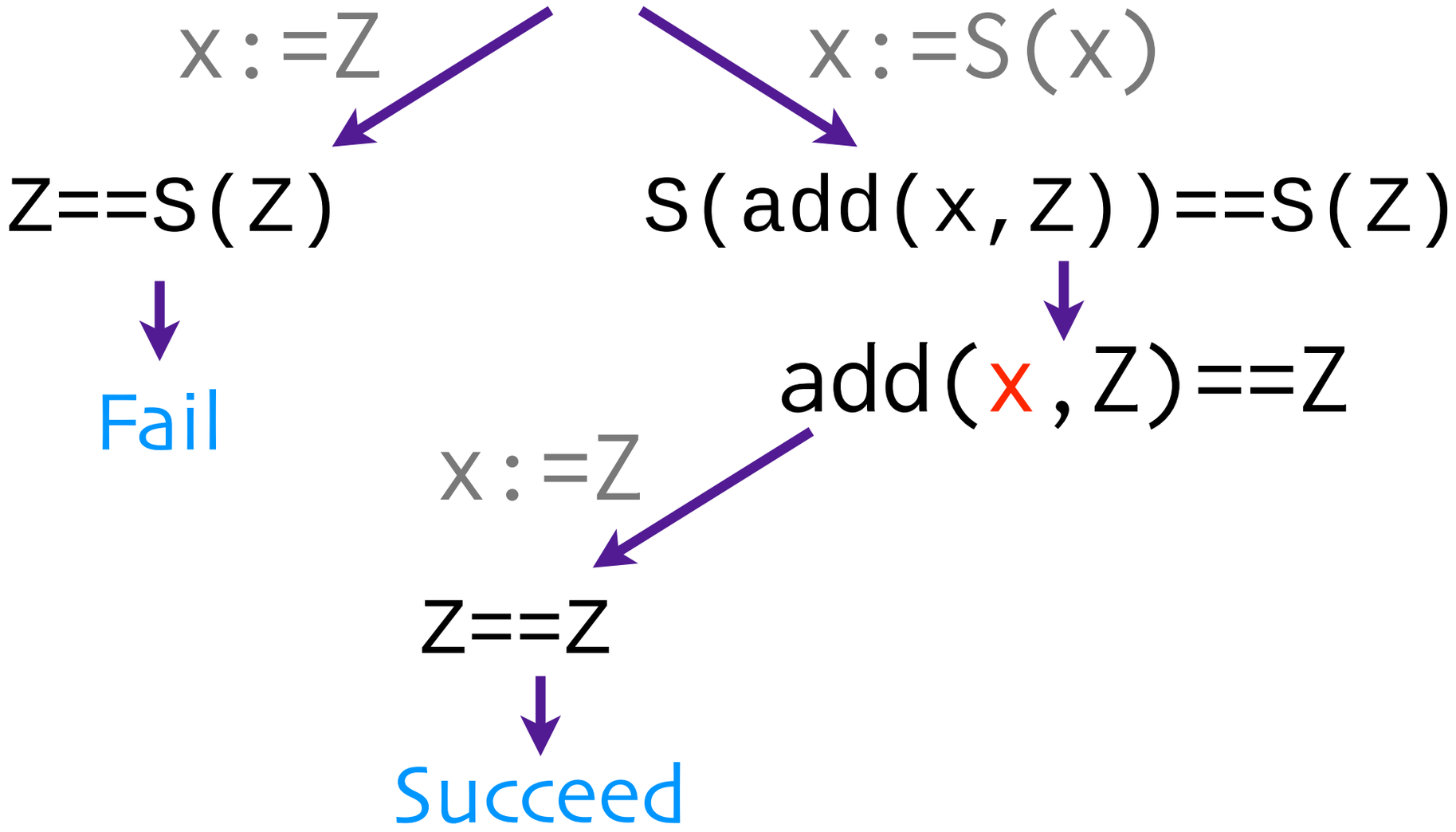
InvComp = Narrowing+EqChk

```
add(Z,y)      = y
add(S(x),y) = S(add(x,y))
```



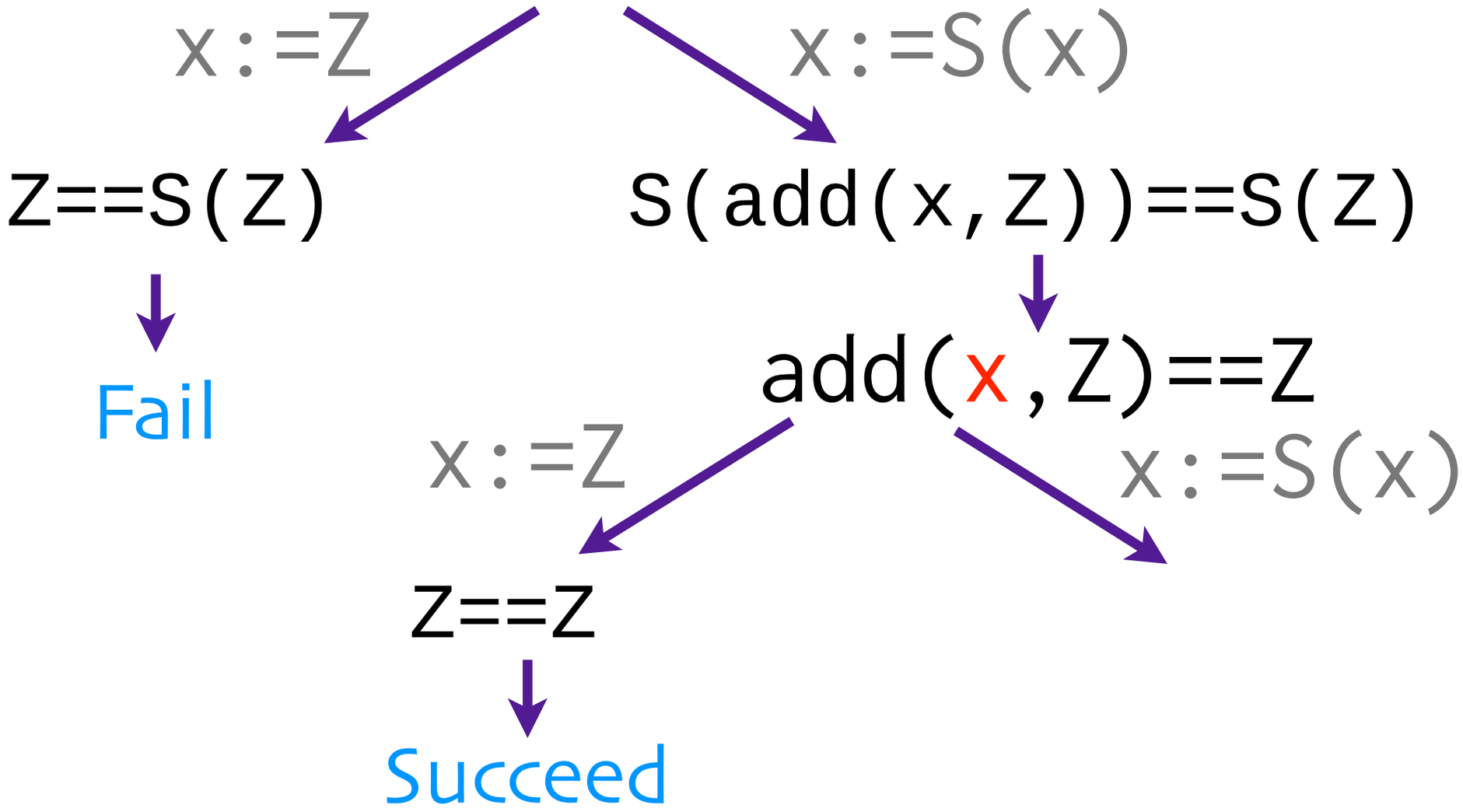
InvComp = Narrowing+EqChk

```
add(Z,y)      = y
add(S(x),y) = S(add(x,y))
```



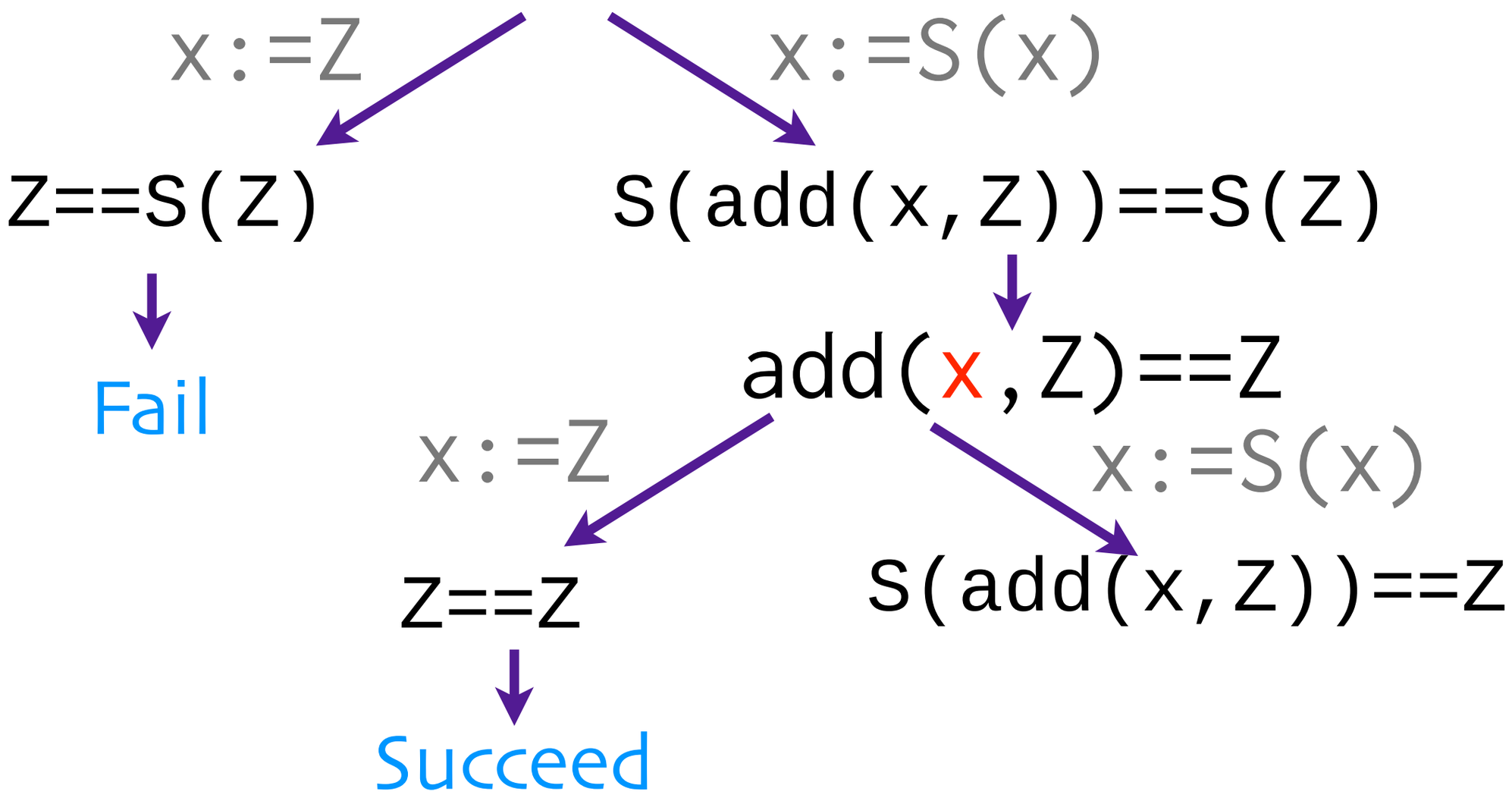
InvComp = Narrowing+EqChk

```
add(Z,y)      = y
add(S(x),y) = S(add(x,y))
```



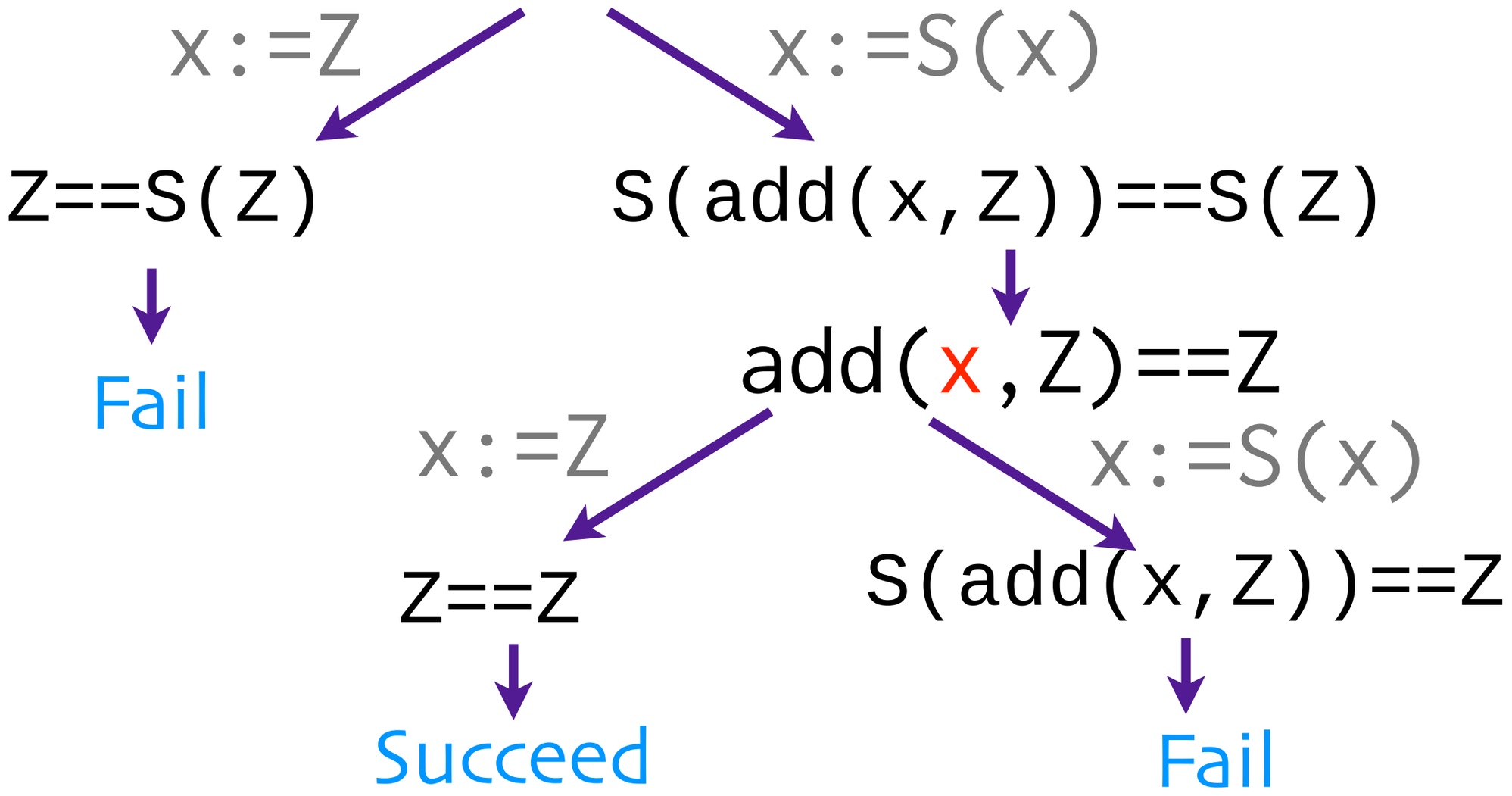
InvComp = Narrowing+EqChk

```
add(Z,y)      = y
add(S(x),y) = S(add(x,y))
```



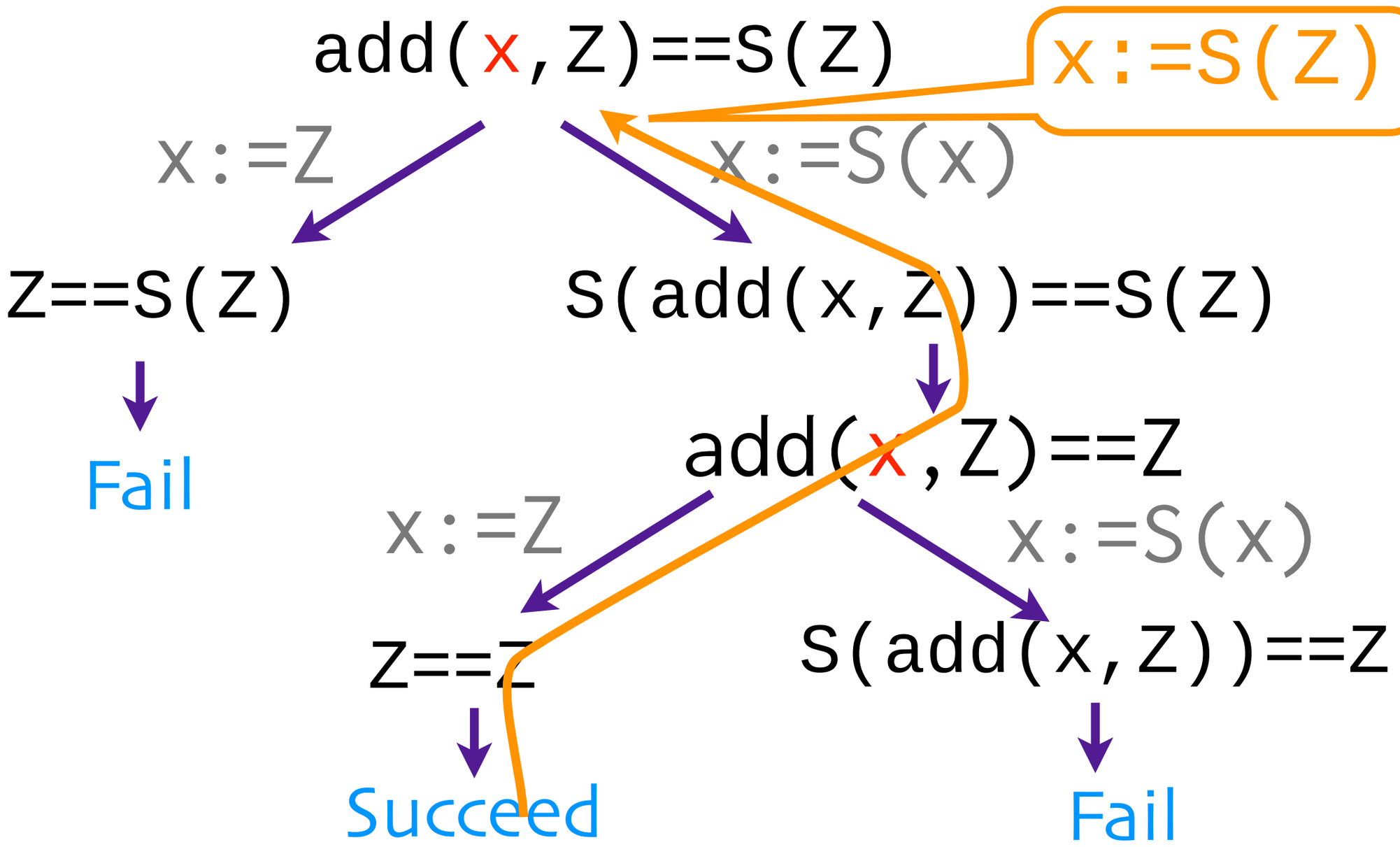
InvComp = Narrowing+EqChk

```
add(Z,y)      = y
add(S(x),y) = S(add(x,y))
```



InvComp = Narrowing+EqChk

```
add(Z,y)    = y
add(S(x),y) = S(add(x,y))
```



Problem Overview

- This simple inverse computation usually runs infinitely for programs ...
  - with accumulations and multiple data traversals

```
exp(x) = ex(x,Z)
ex(Z,y)      = S(y)
ex(S(x),y) = ex(x,ex(x,y))
```

multiple data traversal accumulation

NB: $exp(S^n(Z)) = S^{2^n}(Z)$

Non-Terminating InvComp

```
exp(x) = ex(x,Z)
ex(Z,y)      = S(y)
ex(S(x),y) = ex(x,ex(x,y))
```

NB:
$exp(S^n(Z))=S^{2^n}(Z)$

exp(x)==Z (expected to report "no ans")

Non-Terminating InvComp

```
exp(x) = ex(x,Z)
ex(Z,y)      = S(y)
ex(S(x),y) = ex(x,ex(x,y))
```

NB:
$exp(S^n(Z))=S^{2^n}(Z)$

exp(x)==Z (expected to report "no ans")

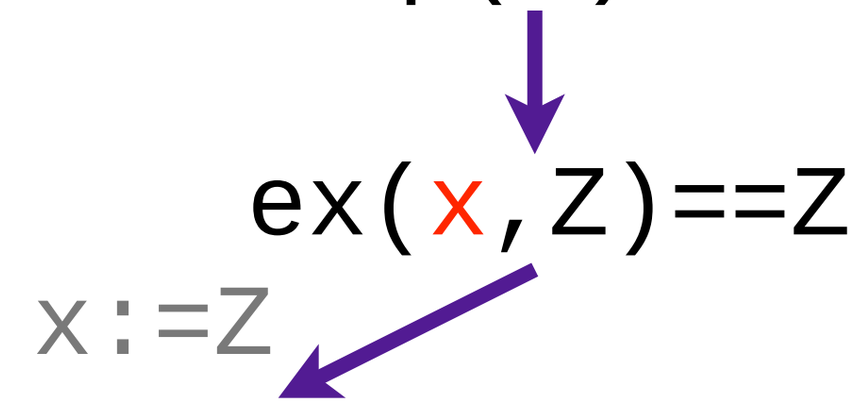
ex(x,Z)==Z

Non-Terminating InvComp

```
exp(x) = ex(x,Z)
ex(Z,y)      = S(y)
ex(S(x),y) = ex(x,ex(x,y))
```

NB:
$exp(S^n(Z))=S^{2^n}(Z)$

exp(x)==Z (expected to report "no ans")

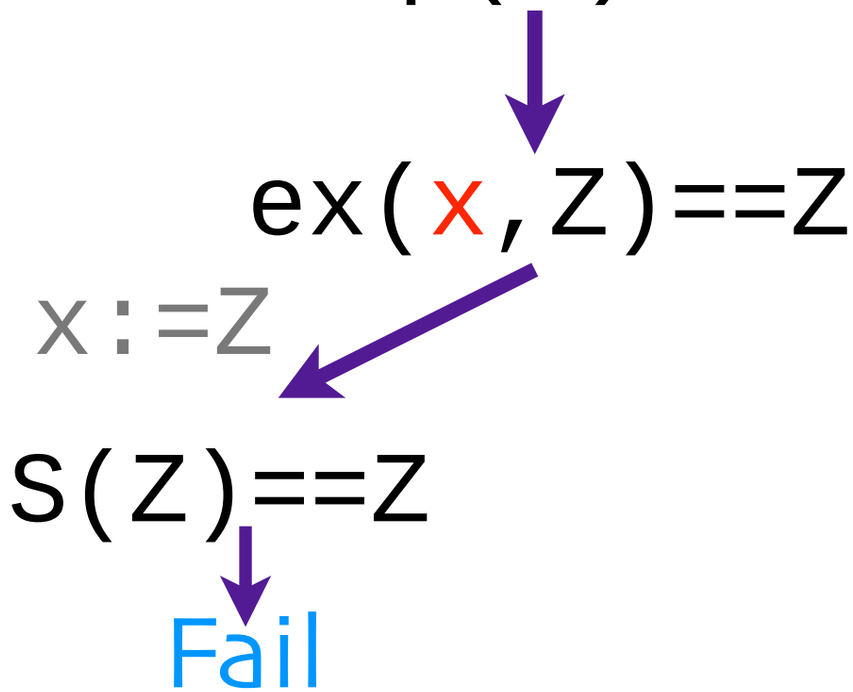


Non-Terminating InvComp

```
exp(x) = ex(x,Z)
ex(Z,y)      = S(y)
ex(S(x),y) = ex(x,ex(x,y))
```

NB:
$exp(S^n(Z))=S^{2^n}(Z)$

exp(x)==Z (expected to report "no ans")

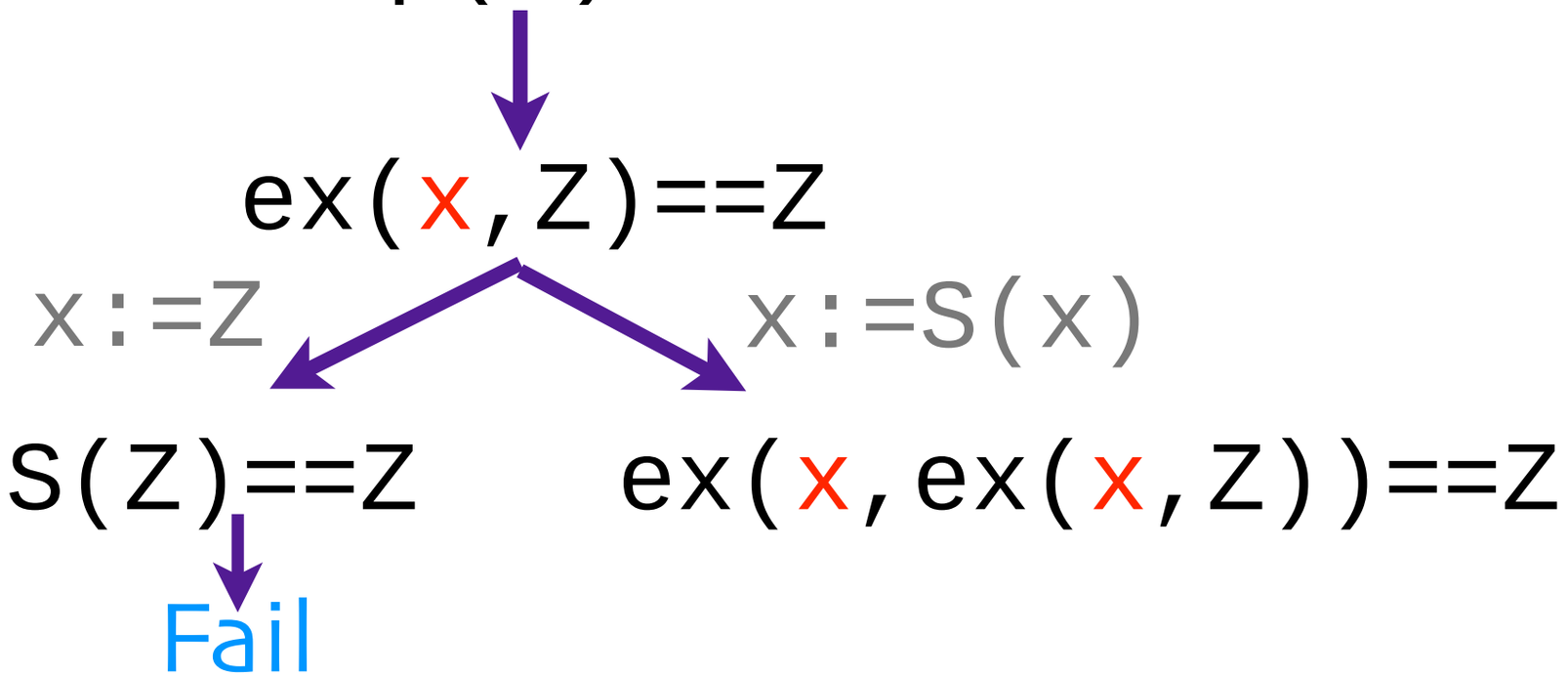


Non-Terminating InvComp

```
exp(x) = ex(x,Z)
ex(Z,y)      = S(y)
ex(S(x),y) = ex(x,ex(x,y))
```

NB:
$\exp(S^n(Z))=S^{2^n}(Z)$

exp(x)==Z (expected to report "no ans")

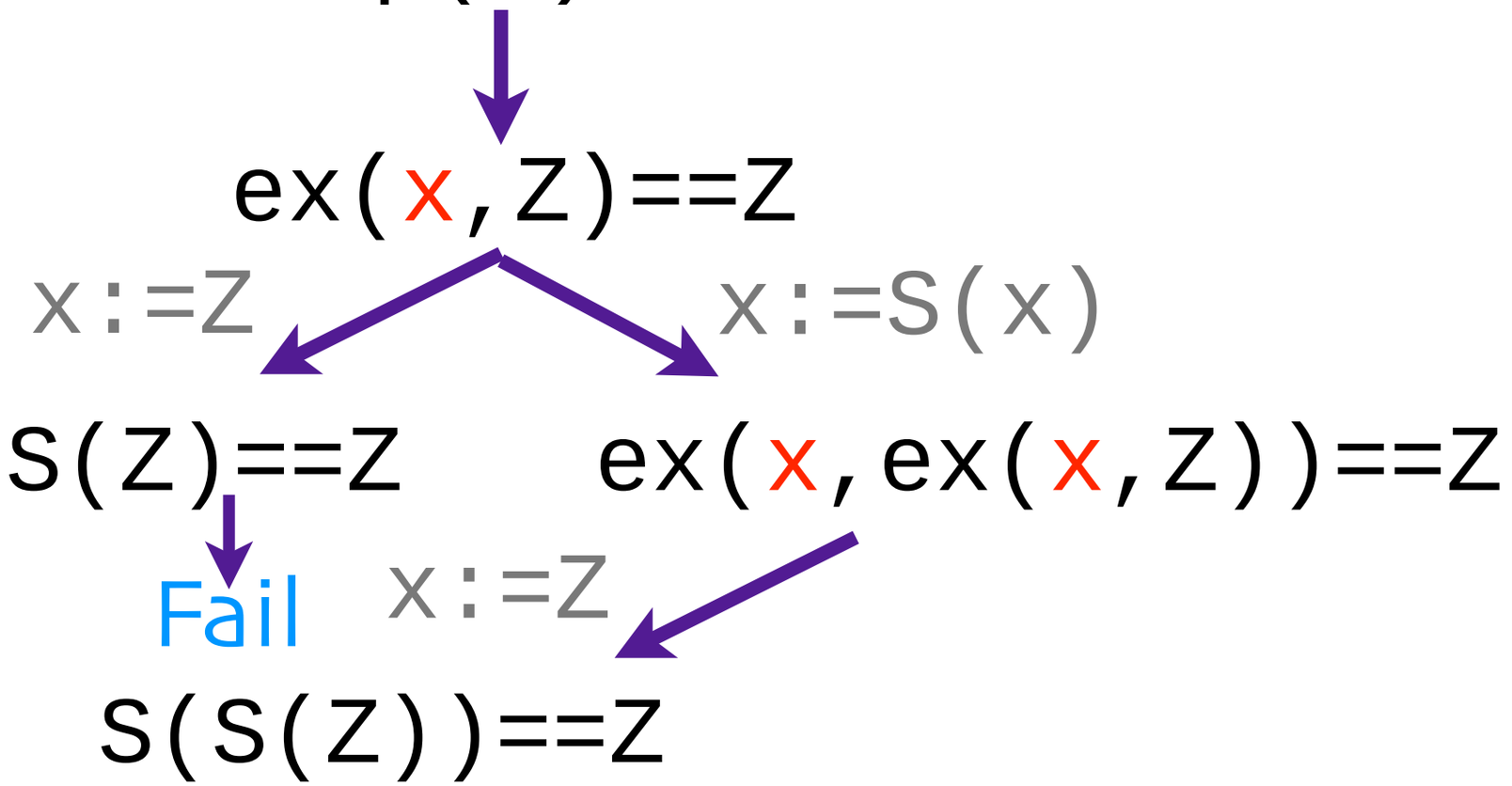


Non-Terminating InvComp

```
exp(x) = ex(x,Z)
ex(Z,y)      = S(y)
ex(S(x),y) = ex(x,ex(x,y))
```

NB:
$exp(S^n(Z))=S^{2^n}(Z)$

exp(x)==Z (expected to report "no ans")

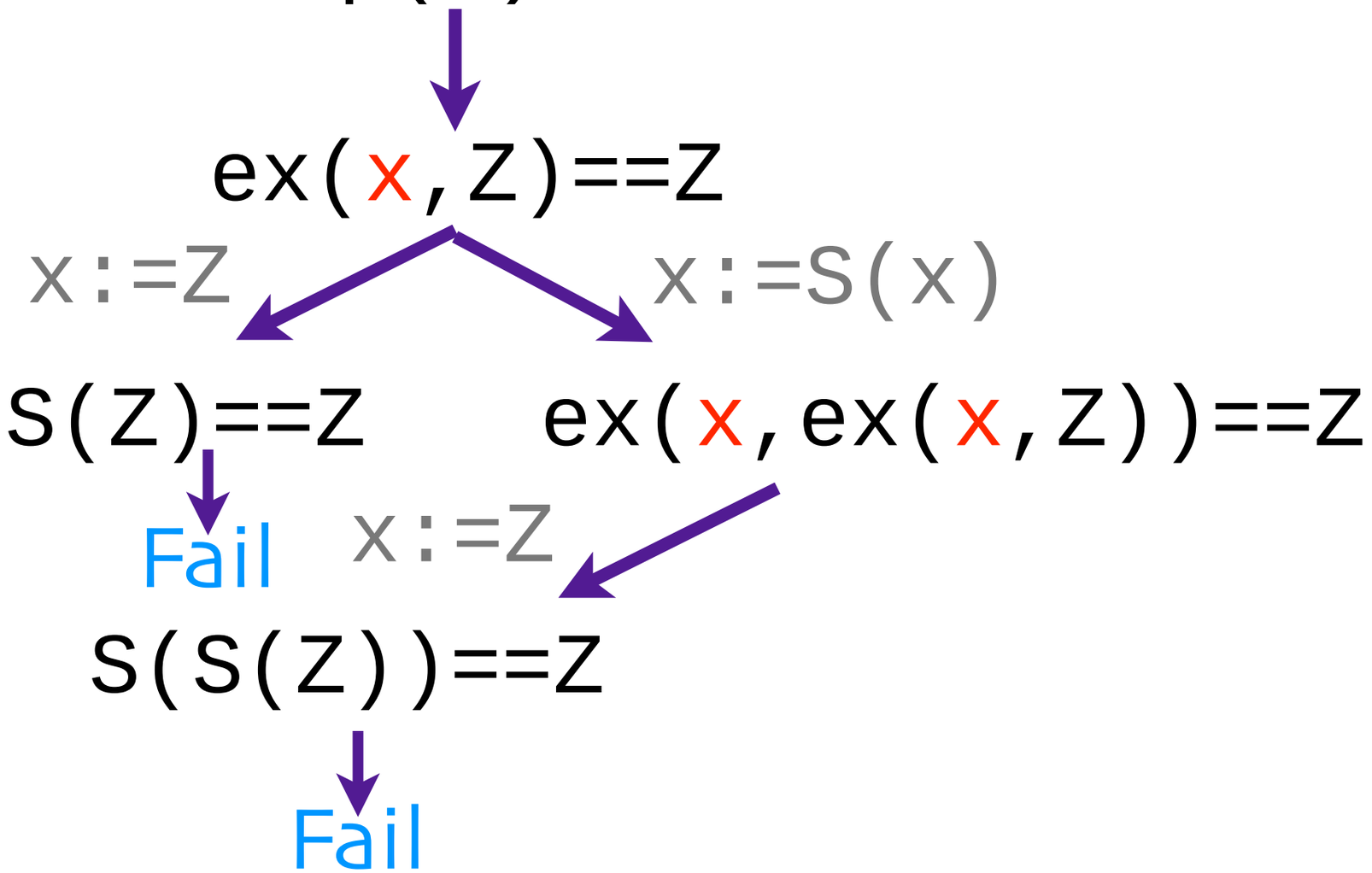


Non-Terminating InvComp

```
exp(x) = ex(x,Z)
ex(Z,y)      = S(y)
ex(S(x),y) = ex(x,ex(x,y))
```

NB:
$exp(S^n(Z))=S^{2^n}(Z)$

exp(x)==Z (expected to report "no ans")

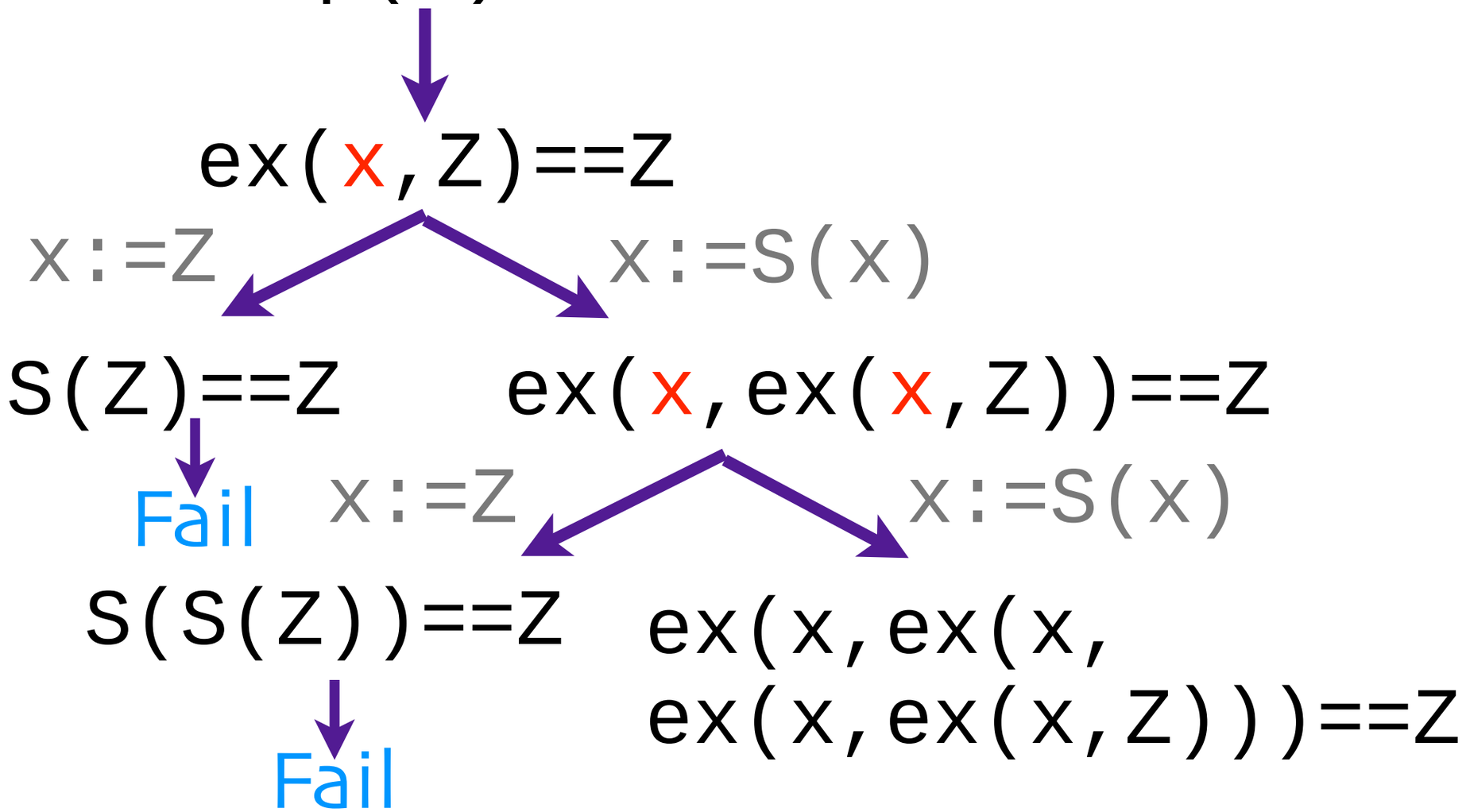


Non-Terminating InvComp

```
exp(x) = ex(x,Z)
ex(Z,y)      = S(y)
ex(S(x),y) = ex(x,ex(x,y))
```

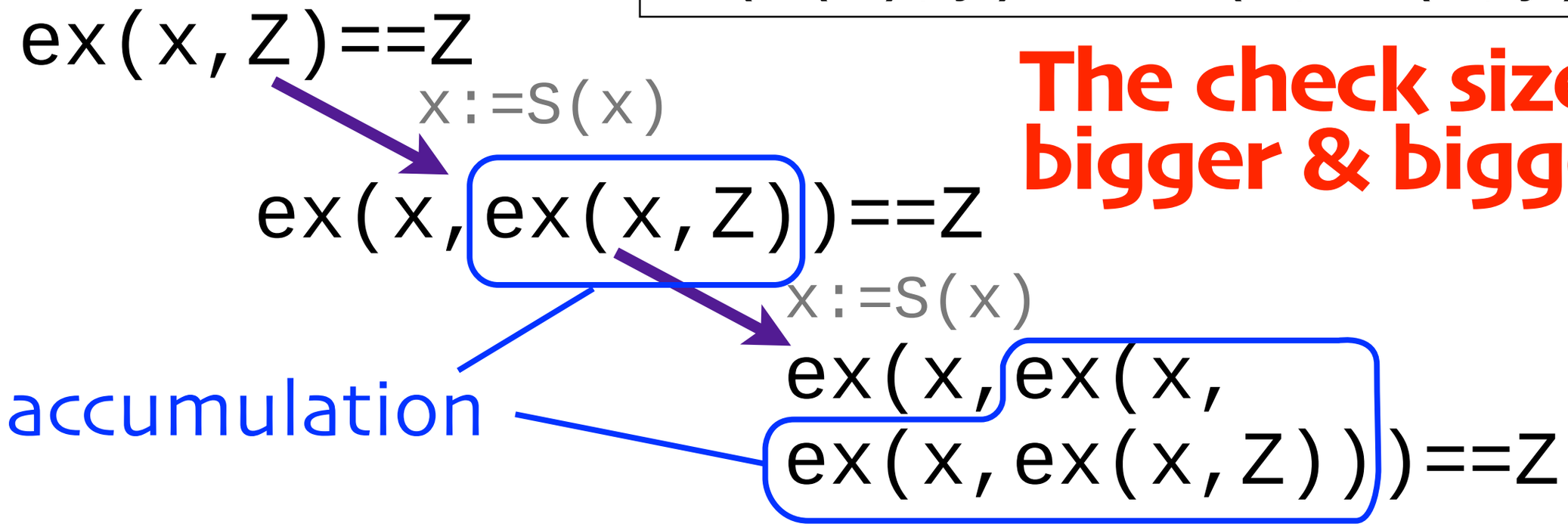
NB:
$exp(S^n(Z))=S^{2^n}(Z)$

exp(x)==Z (expected to report "no ans")



Problem

```
exp(x) = ex(x,Z)
ex(Z,y)      = S(y)
ex(S(x),y) = ex(x,ex(x,y))
```



The check size gets bigger & bigger

- **Accumulations** cause the size increase of equivalence checks
- **Multiple data traversals** bring in-term dependency

Our Contribution

- An inverse computation method
  - targets parameter-linear Macro-Tree Transducers (MTTs)
    - **Accumulations**
    - **Multiple Data Traversals**

  - terminates in polynomial-time
    - Idea: transformation of a program in the class to a **linear non-accumulative context-generating** program

Our Method

A Param-Linear MTT Prog

↓

Non-Accumulative
Context-Generating Prog

↓

Linear Non-Accumulative
Context-Generating Tupled Prog

↓

An Output → Narrowing-based Inv-comp (memozied) → Inputs

Target Language

A Param-Linear MTT Prog

Non-Accumulative
Context-Generating Prog

Linear Non-Accumulative
Context-Generating Tupled Prog

An Output → Narrowing-based Inv-comp (memozied) → Inputs

Language: MTT (1/2)

- (stay) macro-tree transducers (MTT):
  - A first-order FP language where ...
    - inputs can only be destructed
    - outputs cannot be destructed
    - a function takes one input and multiple outputs

```
exp(x) = ex(x,Z)
ex(Z,y)      = S(y)
ex(S(x),y) = ex(x,ex(x,y))
```

[Engelfriet&Voger 85]

Language: MTT (2/2)

- (stay) macro-tree transducers (MTT):
  - A first-order FP language where ...
    - inputs can only be destructed
    - outputs cannot be destructed
    - a function takes one input and multiple outputs

$$
\begin{array}{lll}
prog & ::= & rule_1,\ldots,rule_n \\
rule & ::= & f(p,y_1,\ldots,y_m)=e \\
p & ::= & C(x_1,\ldots,x_n) \mid x \\
e & ::= & C(e_1,\ldots,e_n) \mid y \mid f(x,e_1,\ldots,e_m)
\end{array}
$$

[Engelfriet&Voger 85]

Parameter Linear MTT

- An MTT such that
  - each parameter is used exactly once
    - parameter: a variable bind outputs

OK

```
exp(x) = ex(x,Z)
ex(Z,y)      = S(y)
ex(S(x),y) = ex(x,ex(x,y))
```

NG

```
completeBinT(x) = cb(x,L)
cb(Z,y)      = y
cb(S(x),y) = cb(x,N(y,y))
```

Elimination of Acc

A Param-Linear MTT Prog

↓

Non-Accumulative
Context-Generating Prog

↓

Linear Non-Accumulative
Context-Generating Tupled Prog

An Output → Narrowing-based Inv-comp (memozied) → Inputs

Key Observation

- In MTT, outputs cannot be destructed
  → they appear in the result as-is

e.g.: `ex(S(Z),Z)` = `S(S(Z))`

`ex(S(Z),S(Z))` = `S(S(S(Z)))`

context: tree w/ holes

`ex(S(Z),•)` = `S(S(•))`

```
ex(Z,y)    = S(y)
ex(S(x),y) = ex(x,ex(x,y))
```

Key Observation

- In MTT, outputs cannot be destructed
  → they appear in the result as-is

e.g.: `ex(S(Z),Z)` = `S(S(Z))`

`ex(S(Z),S(Z))` = `S(S(S(Z)))`

context: tree w/ holes

`ex(S(Z),•)` = `S(S(•))`

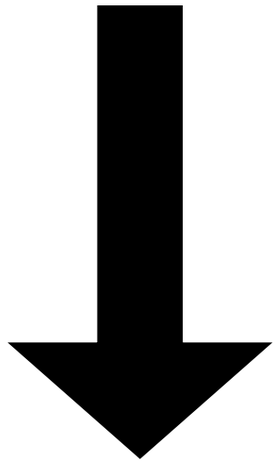
An MTT generates contexts rather than trees

```
ex(Z,y)    = S(y)
ex(S(x),y) = ex(x,ex(x,y))
```

Make it Explicit

- By program transformation

```
exp(x) = ex(x,Z)
ex(Z,y)      = S(y)
ex(S(x),y) = ex(x,ex(x,y))
```



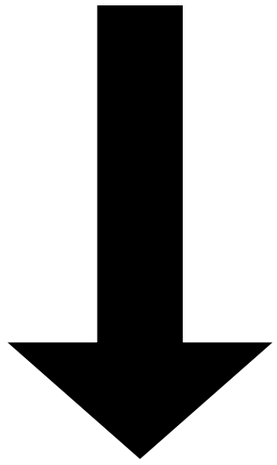
replace f with f_c where
$f_c(x) = f(x,\bullet_1,\ldots,\bullet_m)$

Make it Explicit

- By program transformation

```
exp(x) = ex(x,Z)
ex(Z,y)      = S(y)
ex(S(x),y) = ex(x,ex(x,y))
```

exp(x) = ex(x,•)[Z]



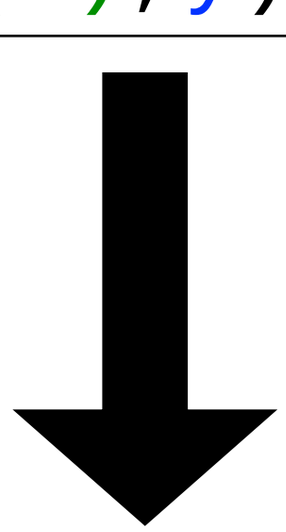
replace f with f_c where
$f_c(x) = f(x,•_1,\ldots,•_m)$

Make it Explicit

- By program transformation

```
exp(x) = ex(x,Z)
ex(Z,y)      = S(y)
ex(S(x),y) = ex(x,ex(x,y))
```

exp(x) = ex(x,•)[Z]



replace f with f_c where
$f_c(x) = f(x,•_1,\ldots,•_m)$

```
expc(x)    = k[Z] where k = exc(x)
```

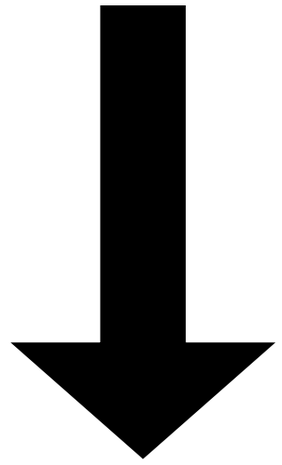
Make it Explicit

- By program transformation

```
exp(x) = ex(x,Z)
ex(Z,y)      = S(y)
ex(S(x),y) = ex(x,ex(x,y))
```

```
exp(x) = ex(x,•)[Z]
```

```
ex(x,•) = S(•)
```



replace f with f_c where
$f_c(x) = f(x,•_1,\ldots,•_m)$

```
expc(x)    = k[Z] where k = exc(x)
```

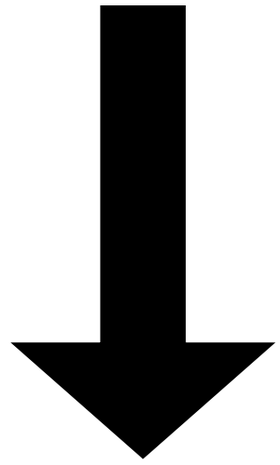
Make it Explicit

- By program transformation

```
exp(x) = ex(x,Z)
ex(Z,y)      = S(y)
ex(S(x),y) = ex(x,ex(x,y))
```

```
exp(x) = ex(x,•)[Z]
```

```
ex(x,•) = S(•)
```



replace f with f_c where
$f_c(x) = f(x,•_1,\ldots,•_m)$

```
exp_c(x)    = k[Z] where k = ex_c(x)
ex_c(Z)     = S(•)
```

Make it Explicit

- By program transformation

```
exp(x) = ex(x,Z)
ex(Z,y)      = S(y)
ex(S(x),y) = ex(x,ex(x,y))
```

```
exp(x) = ex(x,•)[Z]
```

```
ex(x,•) = S(•)
```

```
ex(x,•) = ex(x,•)[ex(x,•)[•]]
```

replace f with f_c where
$f_c(x) = f(x,•_1,\ldots,•_m)$

```
expc(x)    = k[Z] where k = exc(x)
exc(Z)     = S(•)
```

Make it Explicit

- By program transformation

```
exp(x) = ex(x,Z)
ex(Z,y)      = S(y)
ex(S(x),y) = ex(x,ex(x,y))
```

```
exp(x) = ex(x,•)[Z]
```

```
ex(x,•) = S(•)
```

```
ex(x,•) = ex(x,•)[ex(x,•)[•]]
```

replace f with f_c where
$f_c(x) = f(x,•_1,\ldots,•_m)$

```
expc(x)    = k[Z] where k = exc(x)
exc(Z)     = S(•)
exc(S(x)) = k[k[•]] where k = exc(x)
```

Make it Explicit

- By program transformation

```
exp(x) = ex(x,Z)
ex(Z,y)      = S(y)
ex(S(x),y) = ex(x,ex(x,y))
```

```
exp(x) = ex(x,•)[Z]
```

```
ex(x,•) = S(•)
```

```
ex(x,•) = ex(x,•)[ex(x,•)[•]]
```

replace f with f_c where

$f_c(x) = f(x,•_1,\ldots,•_m)$

No Accumulations!

```
expc(x)    = k[Z] where k = exc(x)
exc(Z)     = S(•)
exc(S(x)) = k[k[•]] where k = exc(x)
```

Elimination of MDT

A Param-Linear MTT Prog

Non-Accumulative
Context-Generating Prog

Linear Non-Accumulative
Context-Generating Tupled Prog

An Output → Narrowing-based Inv-comp (memozied) → Inputs

Just Apply Tupling

A Param-Linear

Non-Accumu
Context-Genera

Linear Non-Acc
Context-Generating

An Output → Narrowing-
Inv-con
(memozi

Tupling eliminates multiple data traversals [Hu+97, Chin+93,06]!

```
h(x) = …f(x)…g(x)…
f(x) = …
g(x) = …
```

fg(x)=(f(x),g(x))

```
h(x) = …s…t…
    where (s,t)=fg(x)
fg(x) = …
```

After Tupling

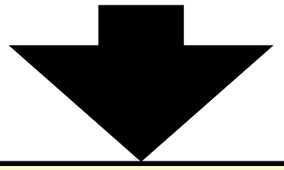
- Tupling always succeeds,
  resulting in a **linear non-accumulative context-genenerating** program
  - **no** multiple data traversals
  - **no** accumulations

  → **no problems** on the inv-comp

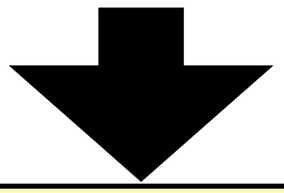
Now, we can apply the existing narrowing-based inv-comp method!!

(Memoized) Inv-Comp

A Param-Linear MTT Prog



Non-Accumulative
Context-Generating Prog



Linear Non-Accumulative
Context-Generating Tupled Prog

An Output

Narrowing-based
Inv-comp
(memozied)

Inputs

Example

```
expc(x)    = k[Z] where k = exc(x)
exc(Z)     = S(•)
exc(S(x)) = k[k[•]] where k = exc(x)
```

```
expc(x) == S(S(Z))
```

Example

```
exp_c(x)    = k[Z] where k = ex_c(x)
ex_c(Z)     = S(•)
ex_c(S(x)) = k[k[•]] where k = ex_c(x)
```

$$exp_c(x) == S(S(Z))$$

Example

```
expc(x)    = k[Z] where k = exc(x)
exc(Z)     = S(•)
exc(S(x)) = k[k[•]] where k = exc(x)
```

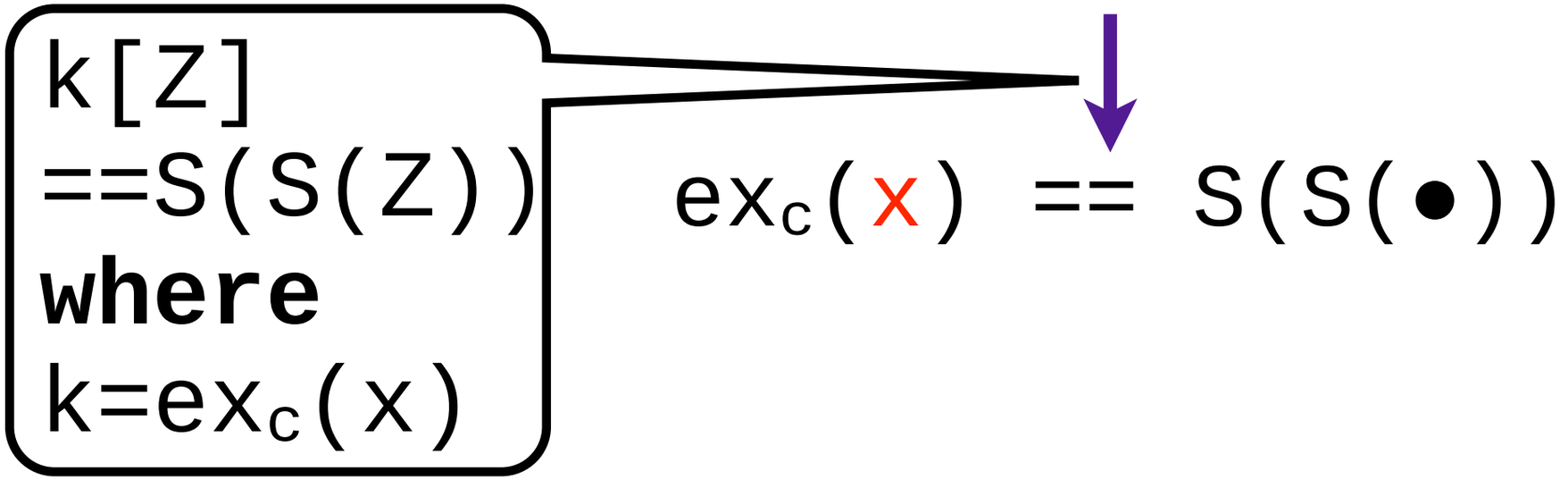
```
expc(x) == S(S(Z))
```

```
k[Z]
==S(S(Z))
where
k=exc(x)
```

Example

```
expc(x)    = k[Z] where k = exc(x)
exc(Z)     = S(•)
exc(S(x)) = k[k[•]] where k = exc(x)
```

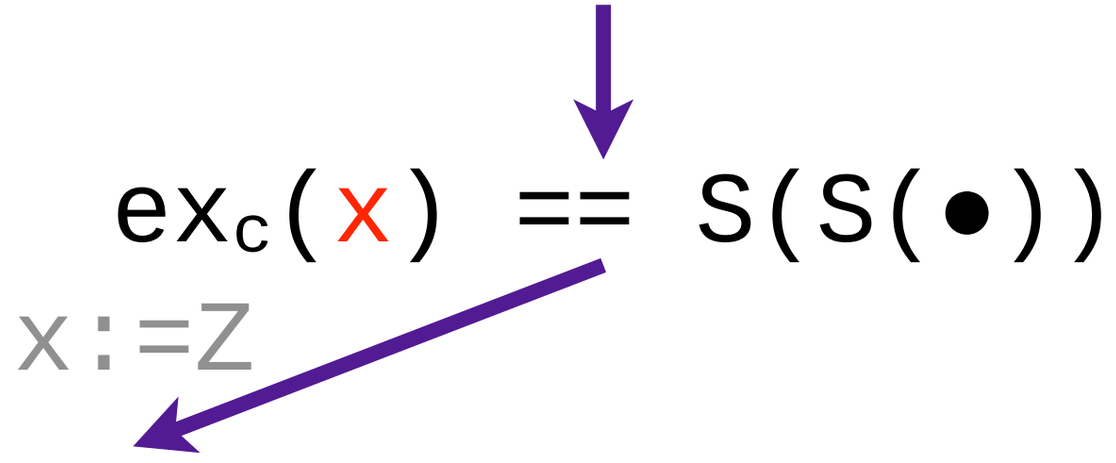
$$exp_c(x) == S(S(Z))$$



Example

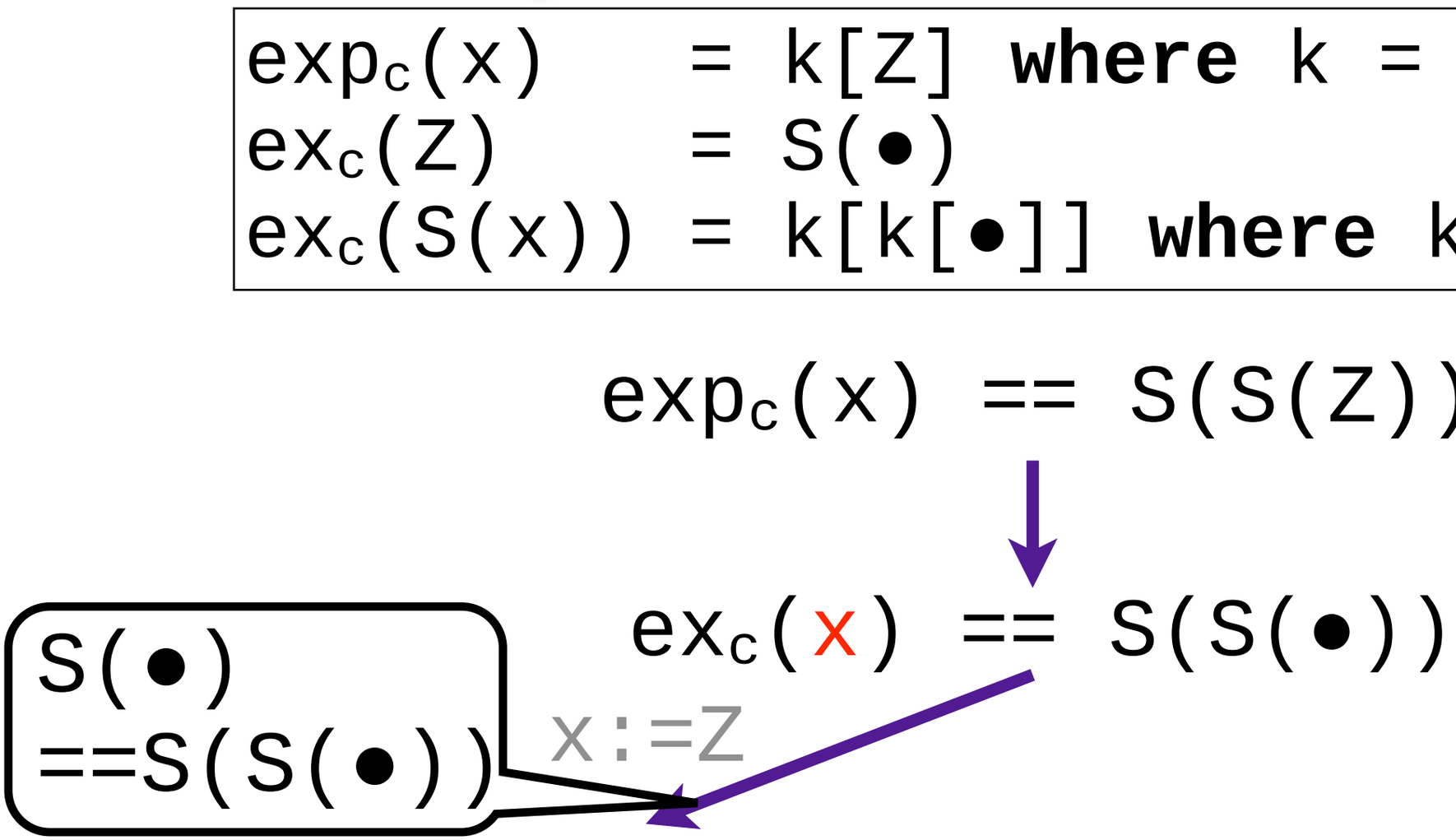
$$
\begin{array}{l}
\mathrm{exp}_c(x) \;\; = k[Z] \textbf{ where } k = \mathrm{ex}_c(x) \\
\mathrm{ex}_c(Z) \;\; = S(\bullet) \\
\mathrm{ex}_c(S(x)) = k[k[\bullet]] \textbf{ where } k = \mathrm{ex}_c(x)
\end{array}
$$

$\mathrm{exp}_c(x) == S(S(Z))$



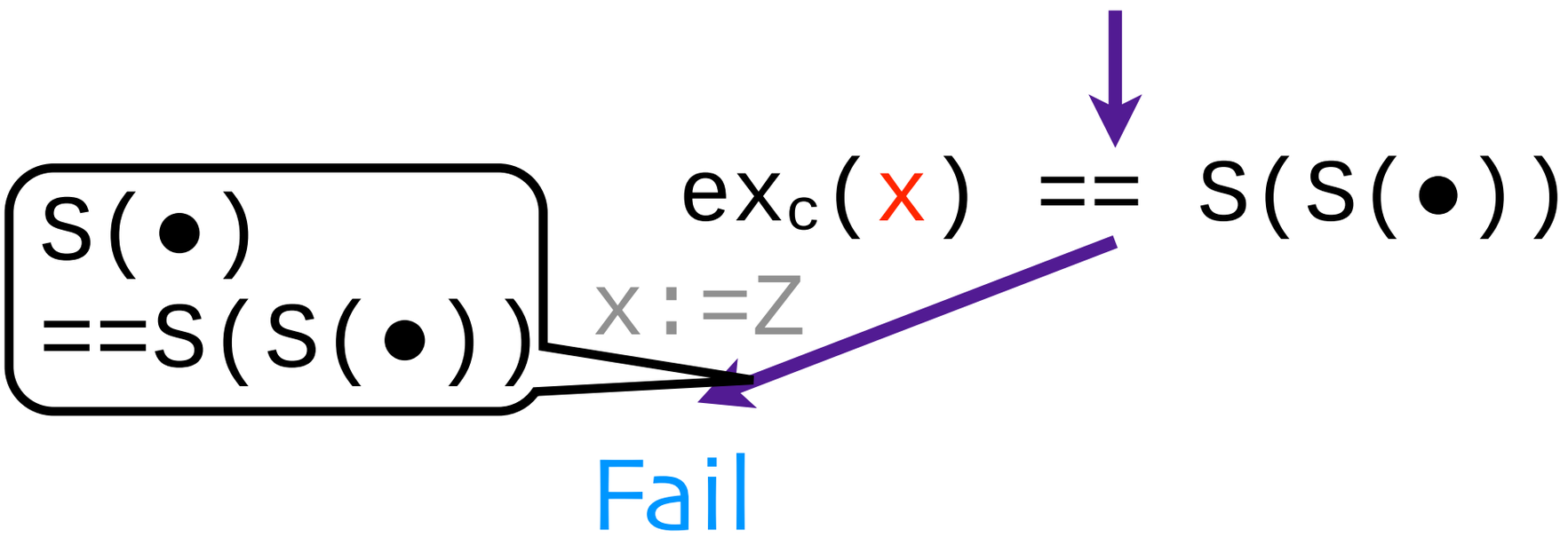
Example

```
expc(x)    = k[Z] where k = exc(x)
exc(Z)     = S(•)
exc(S(x)) = k[k[•]] where k = exc(x)
```



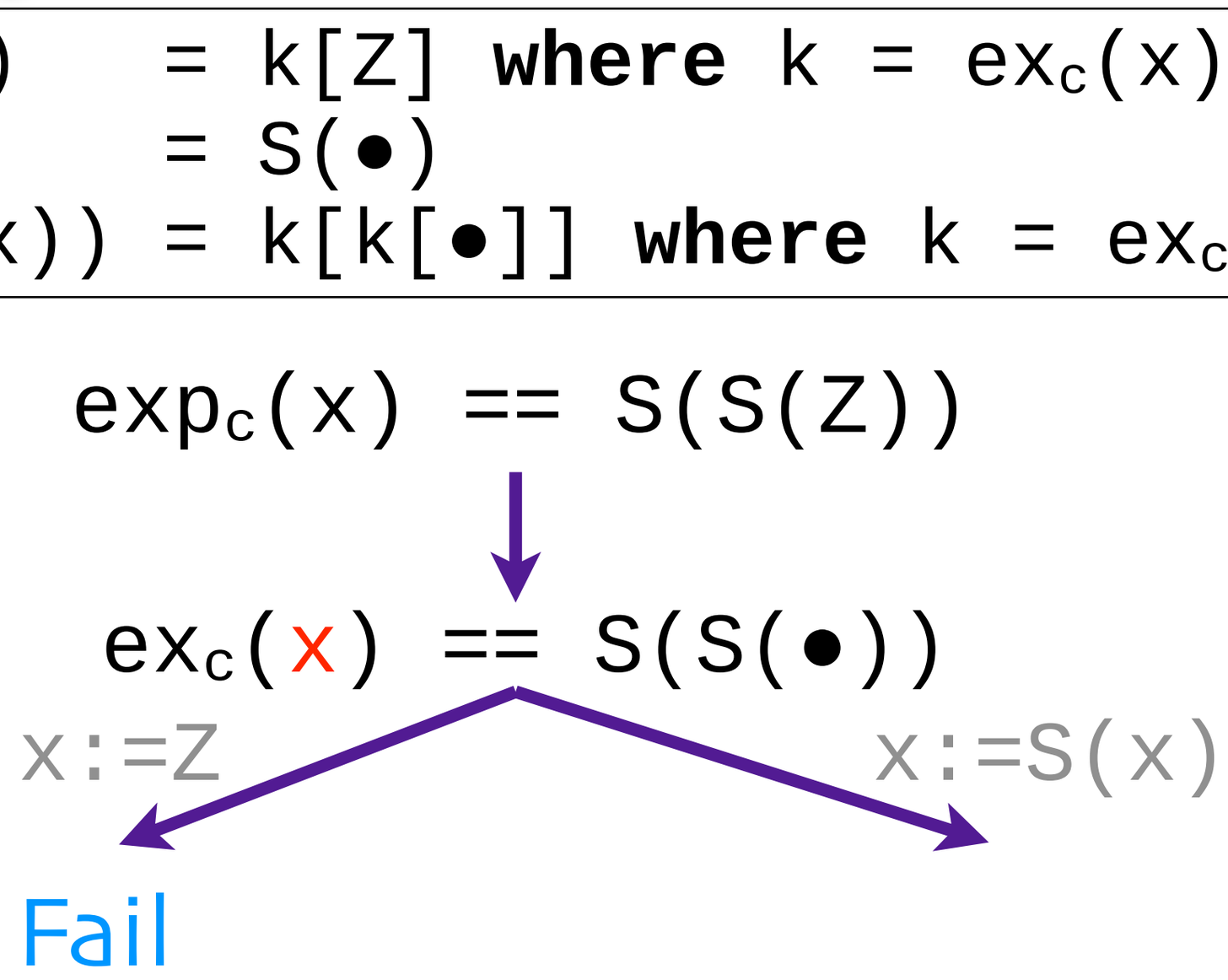
Example

```
expc(x)    = k[Z] where k = exc(x)
exc(Z)     = S(•)
exc(S(x)) = k[k[•]] where k = exc(x)
```



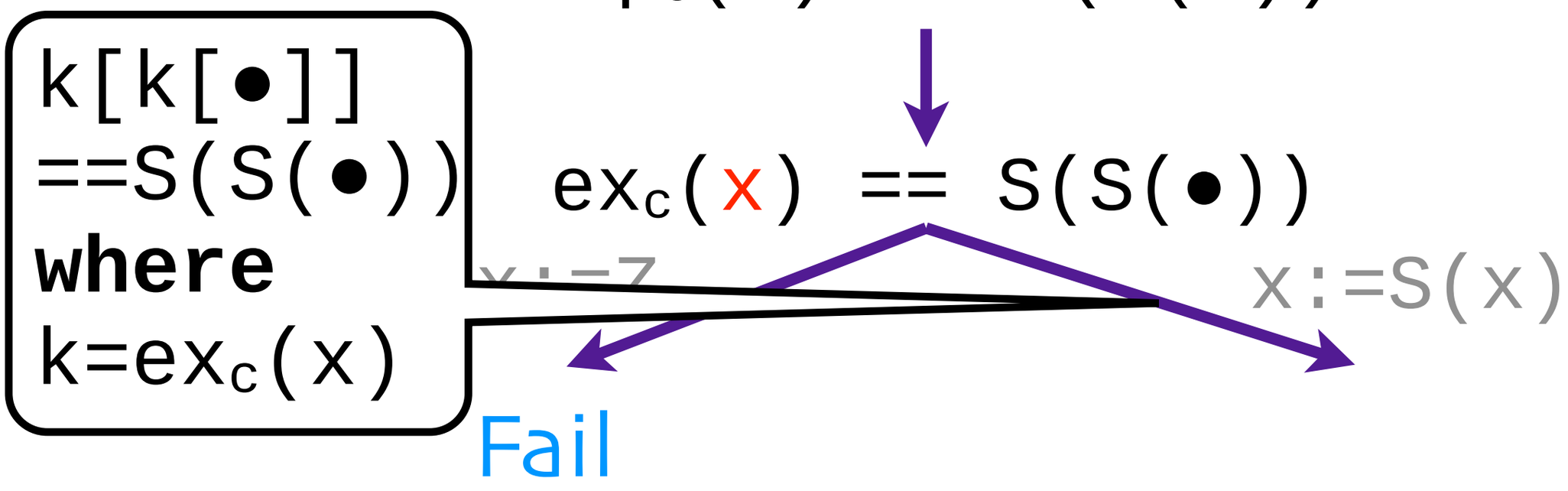
Example

```
expc(x)    = k[Z] where k = exc(x)
exc(Z)     = S(•)
exc(S(x)) = k[k[•]] where k = exc(x)
```



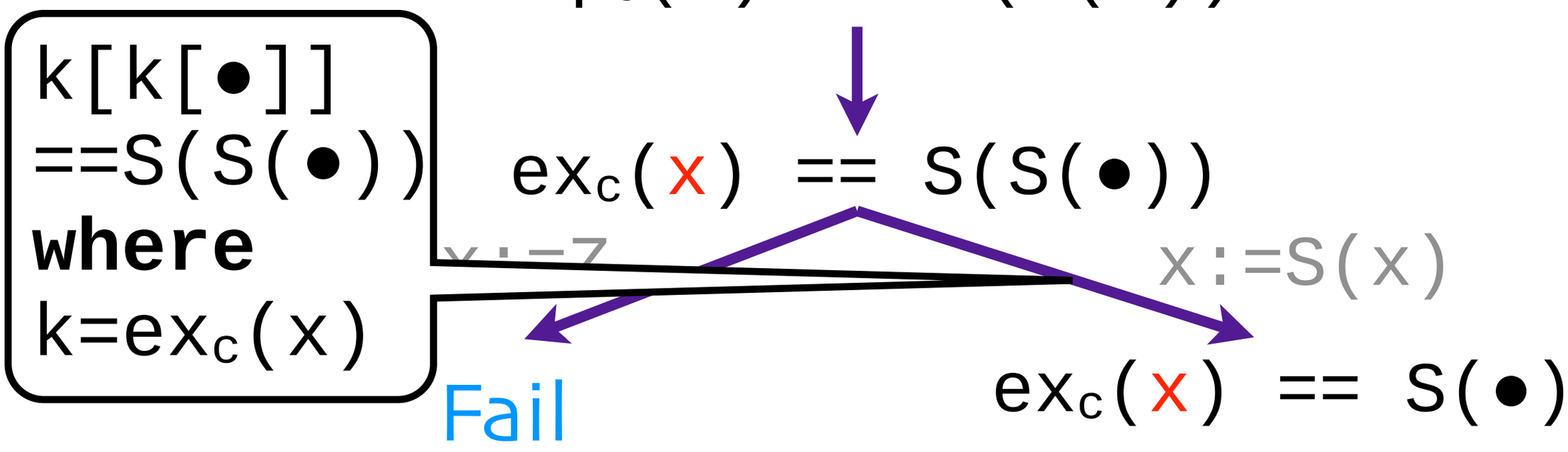
Example

```
expc(x)    = k[Z] where k = exc(x)
exc(Z)     = S(•)
exc(S(x)) = k[k[•]] where k = exc(x)
```



Example

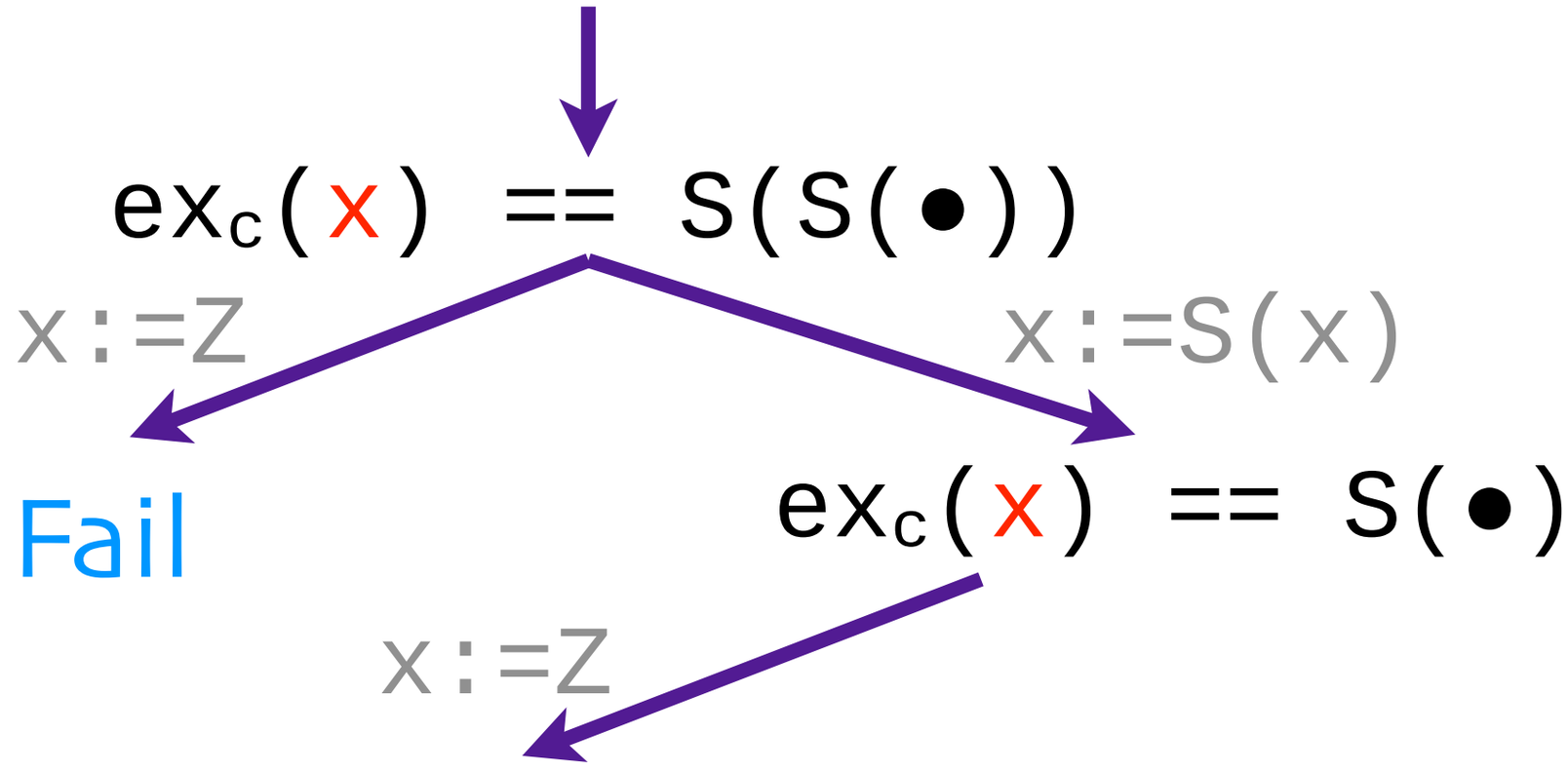
$exp_c(x) = k[Z]$ **where** $k = ex_c(x)$
$ex_c(Z) = S(\bullet)$
$ex_c(S(x)) = k[k[\bullet]]$ **where** $k = ex_c(x)$



Example

```
expc(x)    = k[Z] where k = exc(x)
exc(Z)     = S(•)
exc(S(x)) = k[k[•]] where k = exc(x)
```

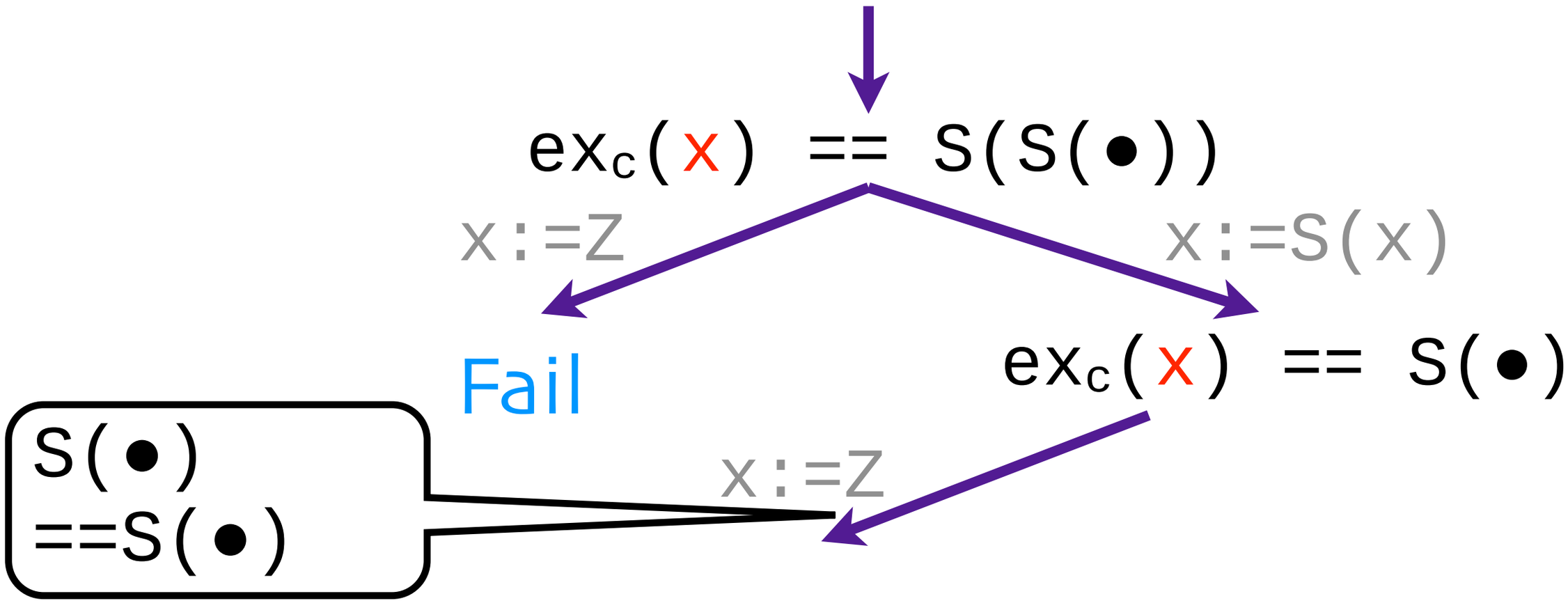
$exp_c(x) == S(S(Z))$



Example

```
expc(x)    = k[Z] where k = exc(x)
exc(Z)     = S(•)
exc(S(x)) = k[k[•]] where k = exc(x)
```

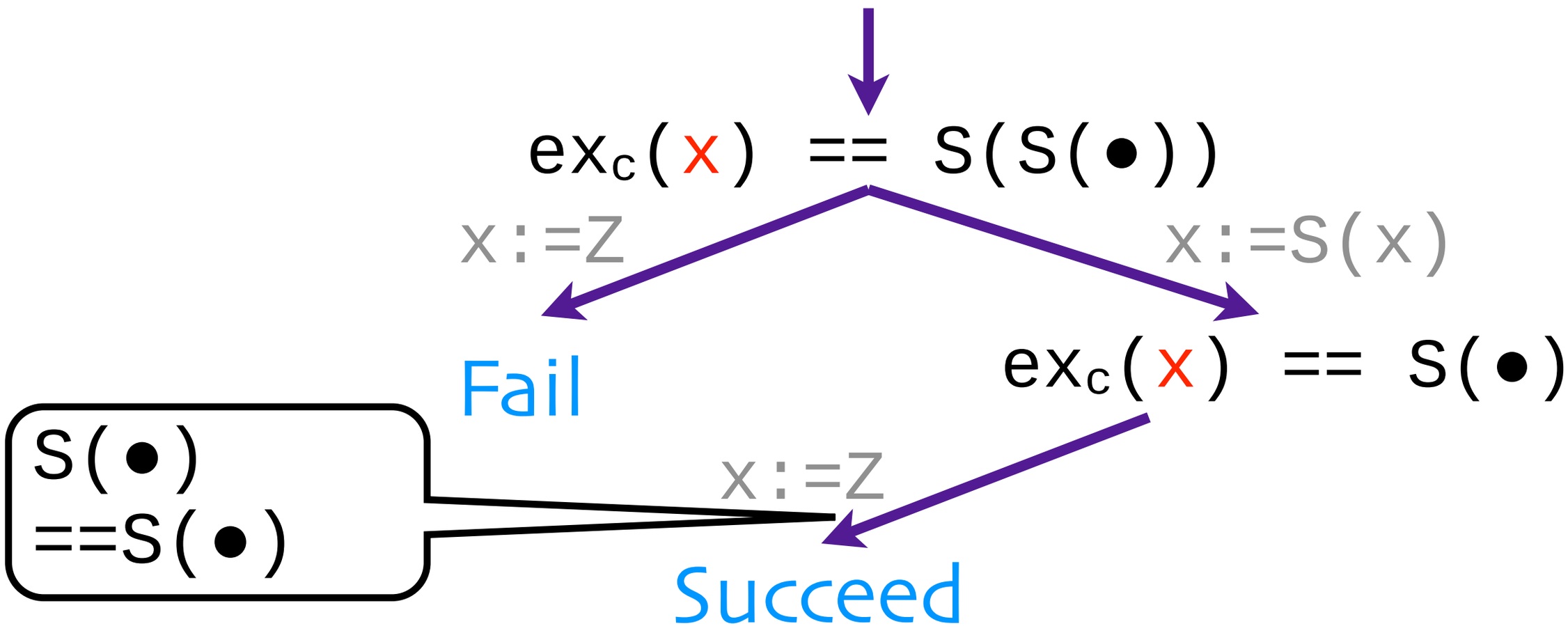
$exp_c(x) == S(S(Z))$



Example

```
expc(x)    = k[Z] where k = exc(x)
exc(Z)     = S(•)
exc(S(x)) = k[k[•]] where k = exc(x)
```

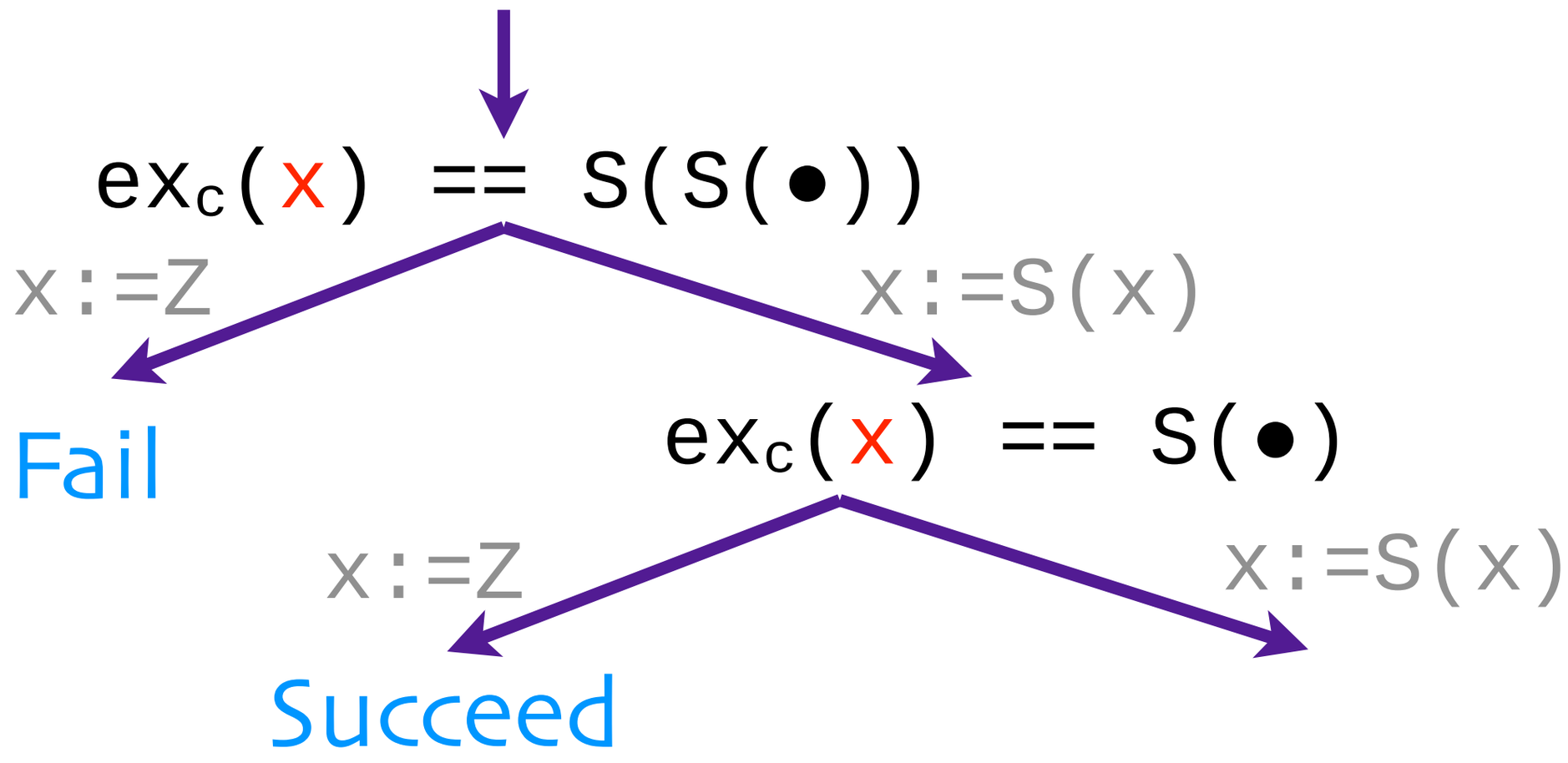
$exp_c(x) == S(S(Z))$



Example

```
exp_c(x)    = k[Z] where k = ex_c(x)
ex_c(Z)     = S(•)
ex_c(S(x)) = k[k[•]] where k = ex_c(x)
```

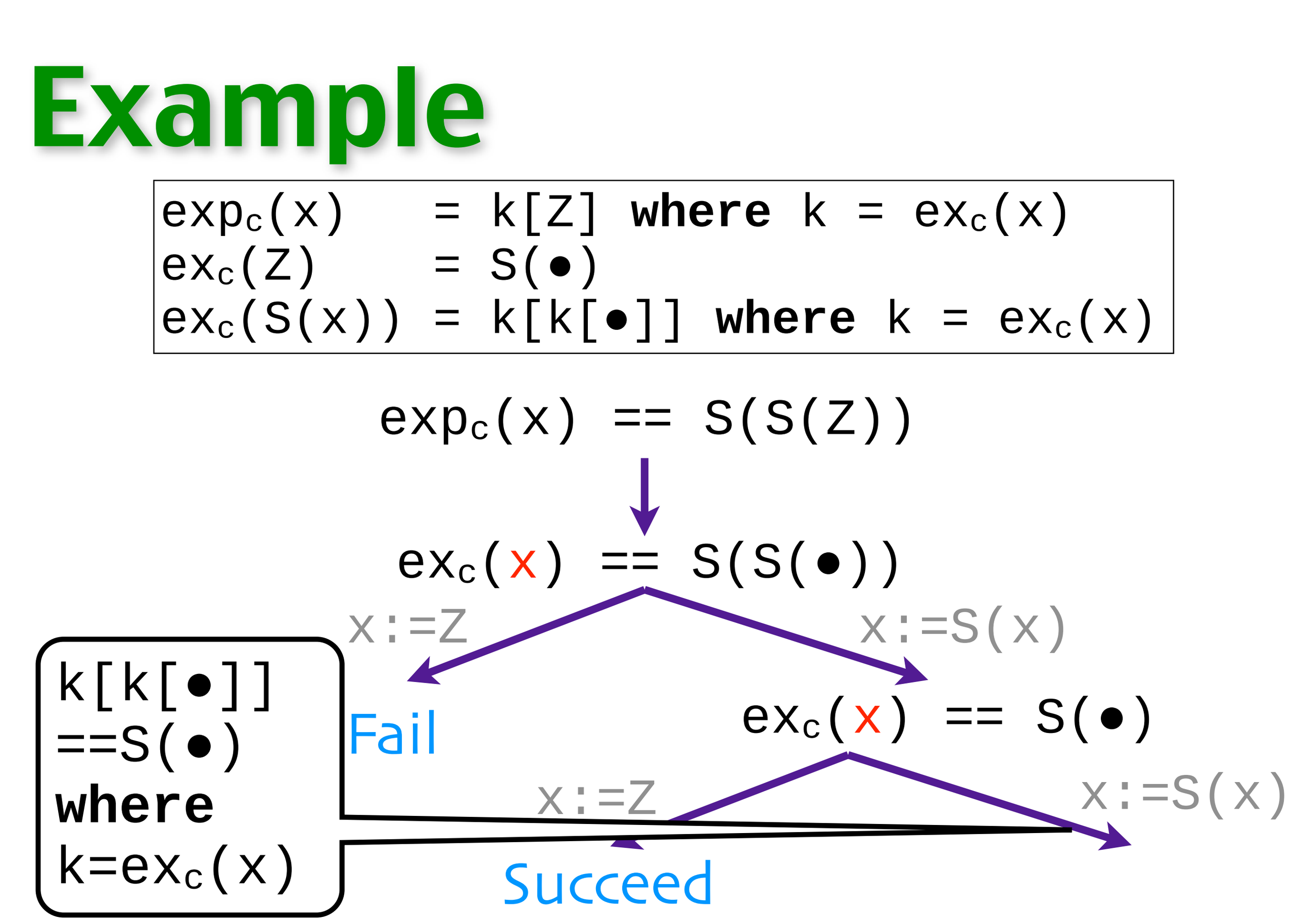
$exp_c(x)$ == S(S(Z))



Example

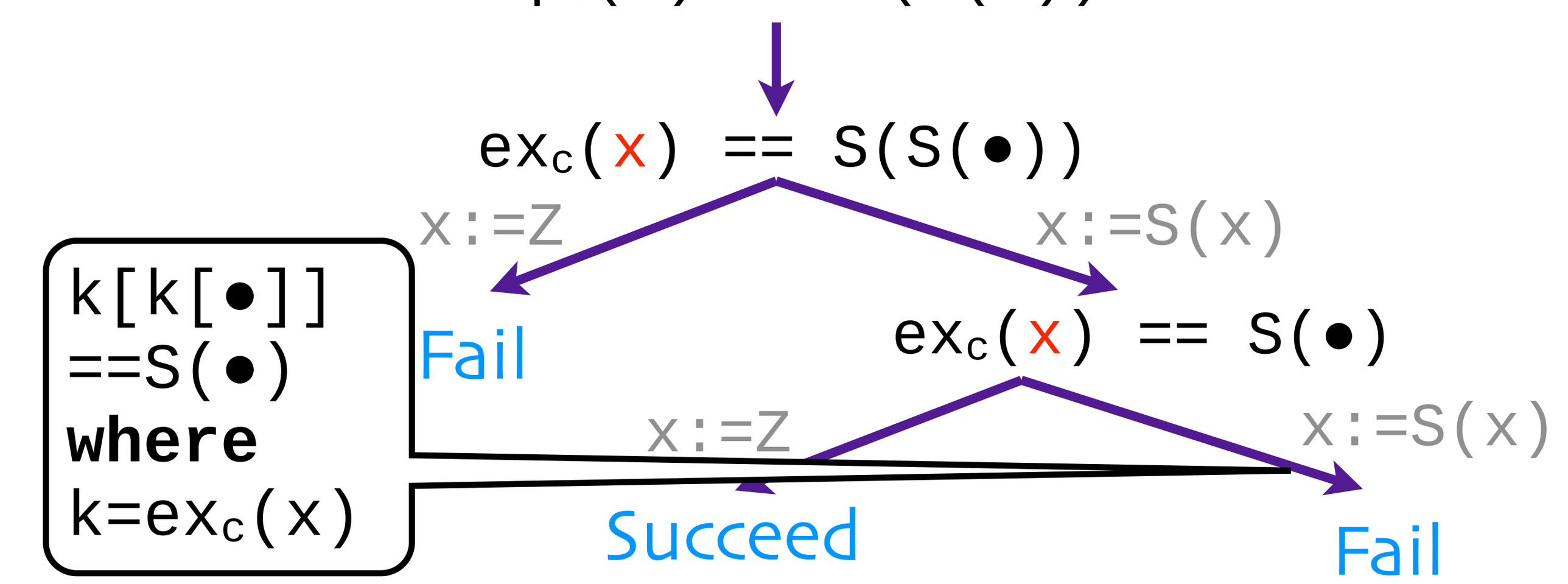
```
expc(x)    = k[Z] where k = exc(x)
exc(Z)     = S(•)
exc(S(x)) = k[k[•]] where k = exc(x)
```

$exp_c(x) == S(S(Z))$



Example

```
expc(x)    = k[Z] where k = exc(x)
exc(Z)     = S(•)
exc(S(x)) = k[k[•]] where k = exc(x)
```



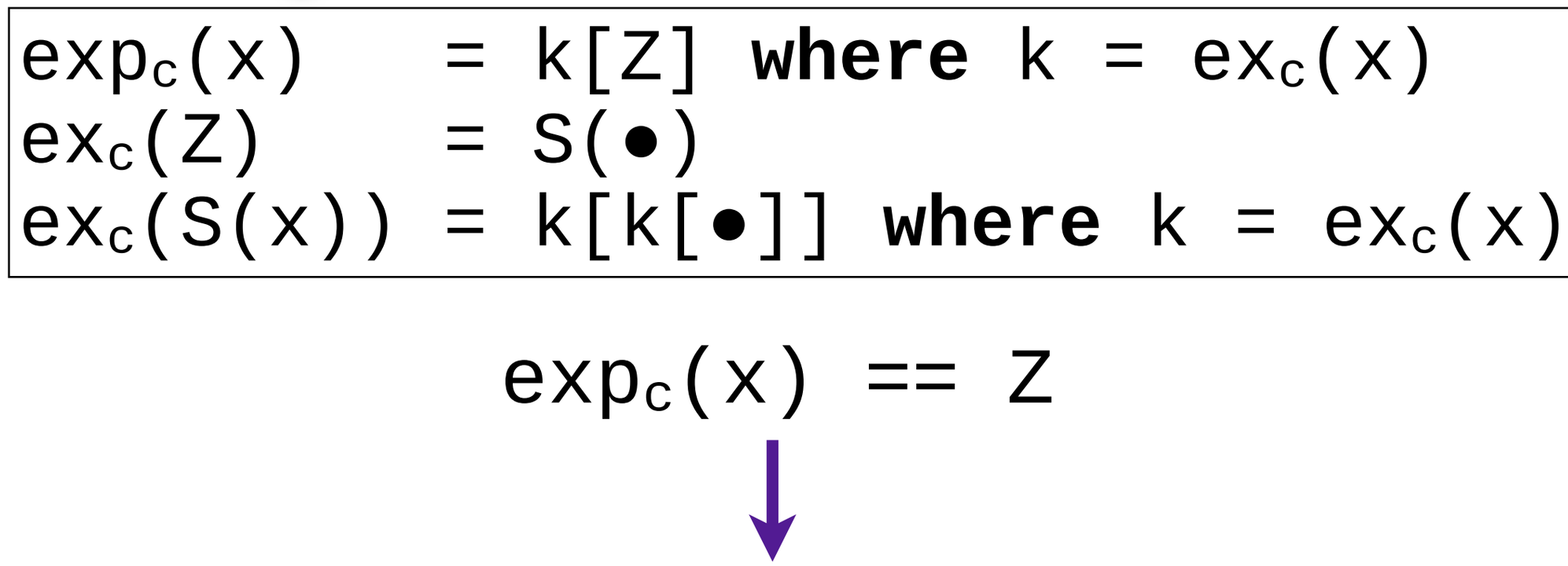
Example

```
expc(x)    = k[Z] where k = exc(x)
exc(Z)     = S(•)
exc(S(x)) = k[k[•]] where k = exc(x)
```

$$exp_c(x) == Z$$

Example

```
expc(x)    = k[Z] where k = exc(x)
exc(Z)     = S(•)
exc(S(x)) = k[k[•]] where k = exc(x)
```



Example

```
expc(x)    = k[Z] where k = exc(x)
exc(Z)     = S(•)
exc(S(x)) = k[k[•]] where k = exc(x)
```

$exp_c(x) == Z$

```
k[Z]
==Z
where
k=exc(x)
```

Example

$$
\begin{aligned}
\mathrm{exp}_c(x) &= k[Z] \textbf{ where } k = \mathrm{ex}_c(x) \\
\mathrm{ex}_c(Z) &= S(\bullet) \\
\mathrm{ex}_c(S(x)) &= k[k[\bullet]] \textbf{ where } k = \mathrm{ex}_c(x)
\end{aligned}
$$

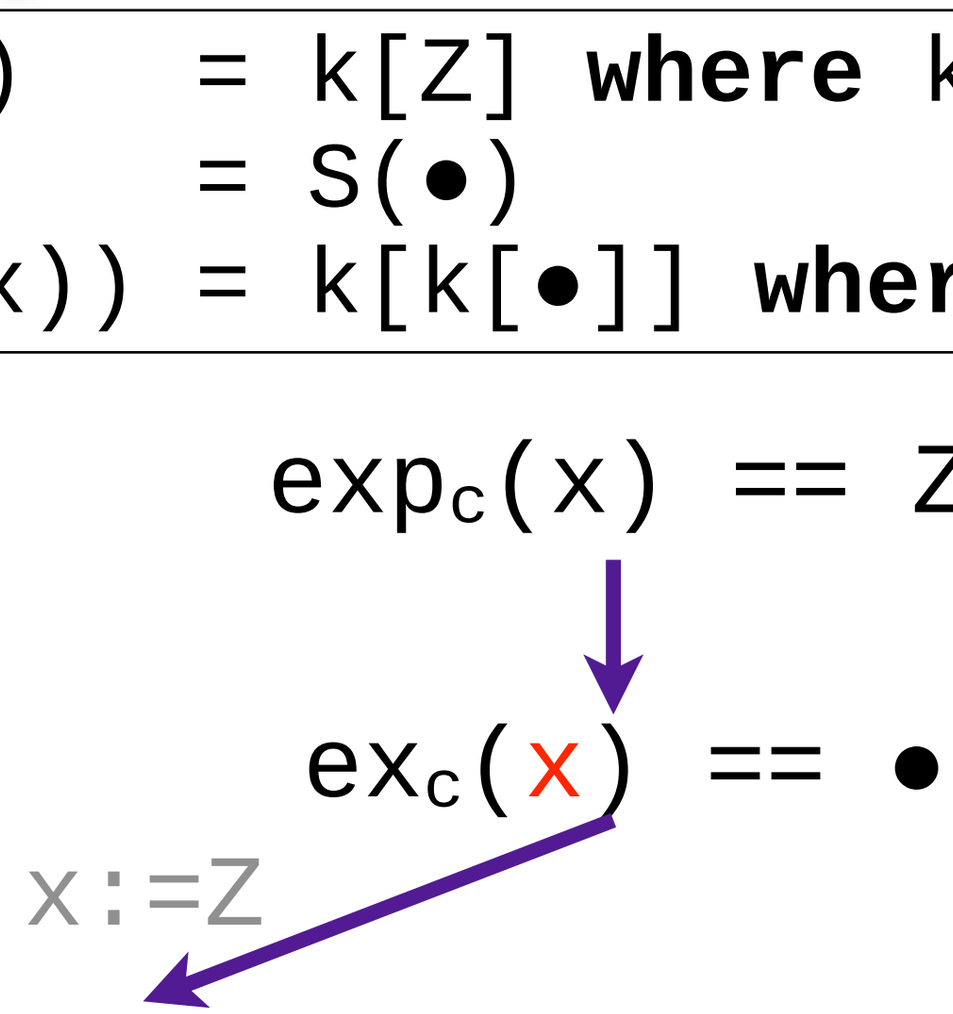
$$\mathrm{exp}_c(x) == Z$$

$$\mathrm{ex}_c(x) == \bullet$$

k[Z]
==Z
where
k=$\mathrm{ex}_c(x)$

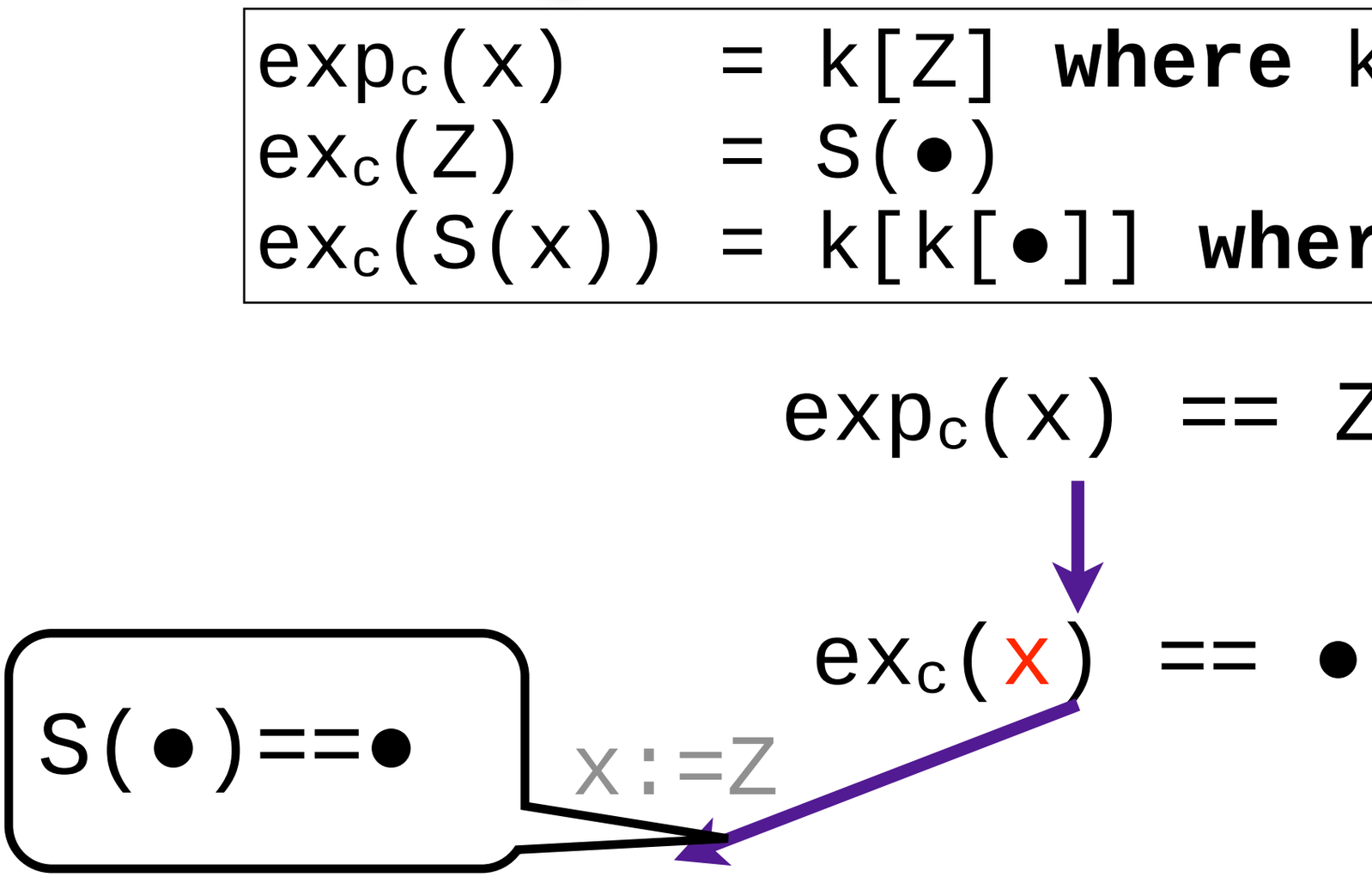
Example

```
expc(x)    = k[Z] where k = exc(x)
exc(Z)     = S(•)
exc(S(x)) = k[k[•]] where k = exc(x)
```



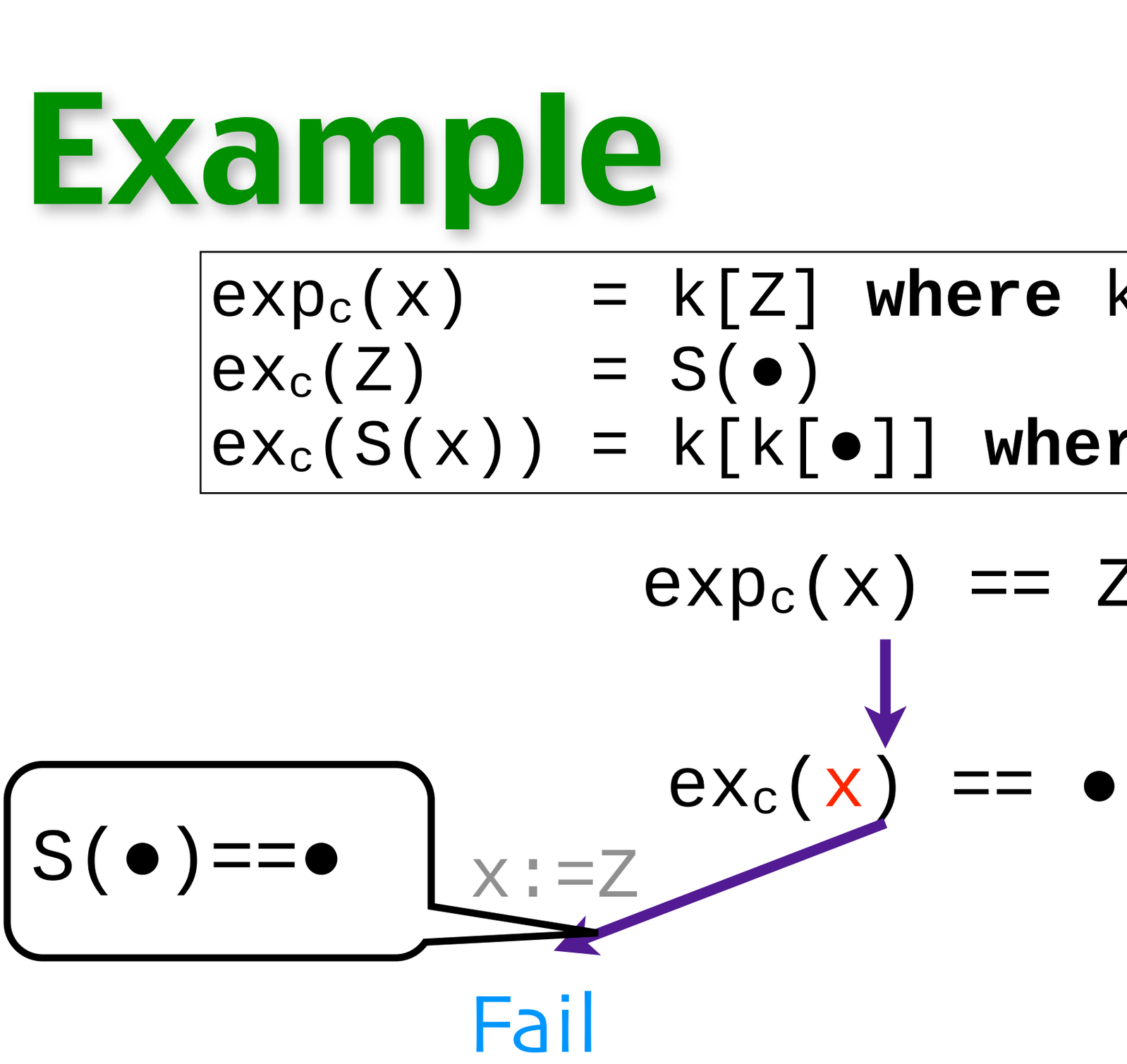
Example

```
expc(x)    = k[Z] where k = exc(x)
exc(Z)     = S(•)
exc(S(x)) = k[k[•]] where k = exc(x)
```



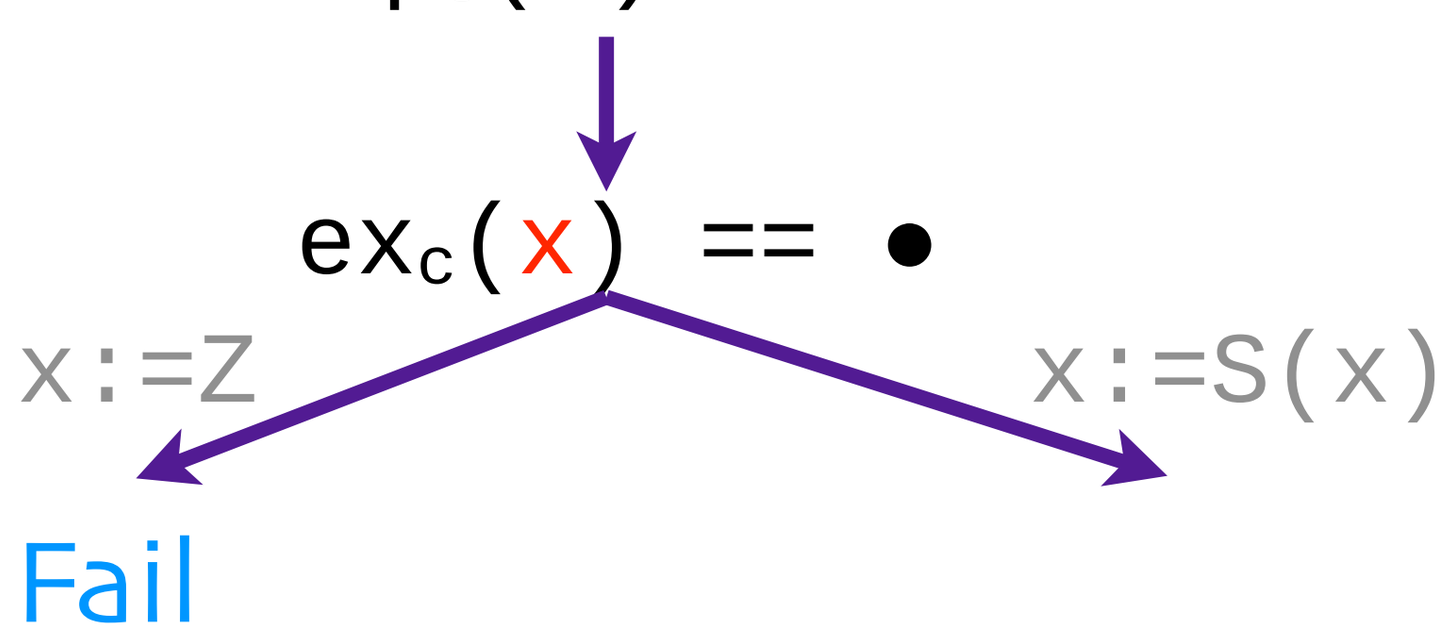
Example

```
exp_c(x)    = k[Z] where k = ex_c(x)
ex_c(Z)     = S(•)
ex_c(S(x)) = k[k[•]] where k = ex_c(x)
```



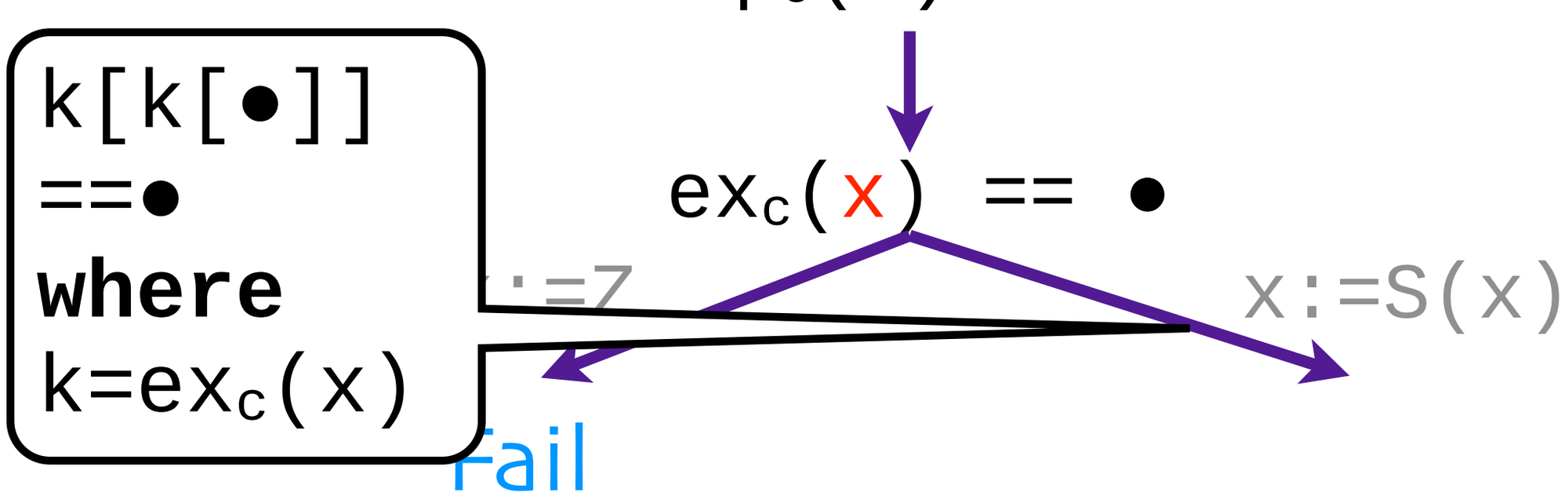
Example

```
expc(x)    = k[Z] where k = exc(x)
exc(Z)     = S(•)
exc(S(x)) = k[k[•]] where k = exc(x)
```



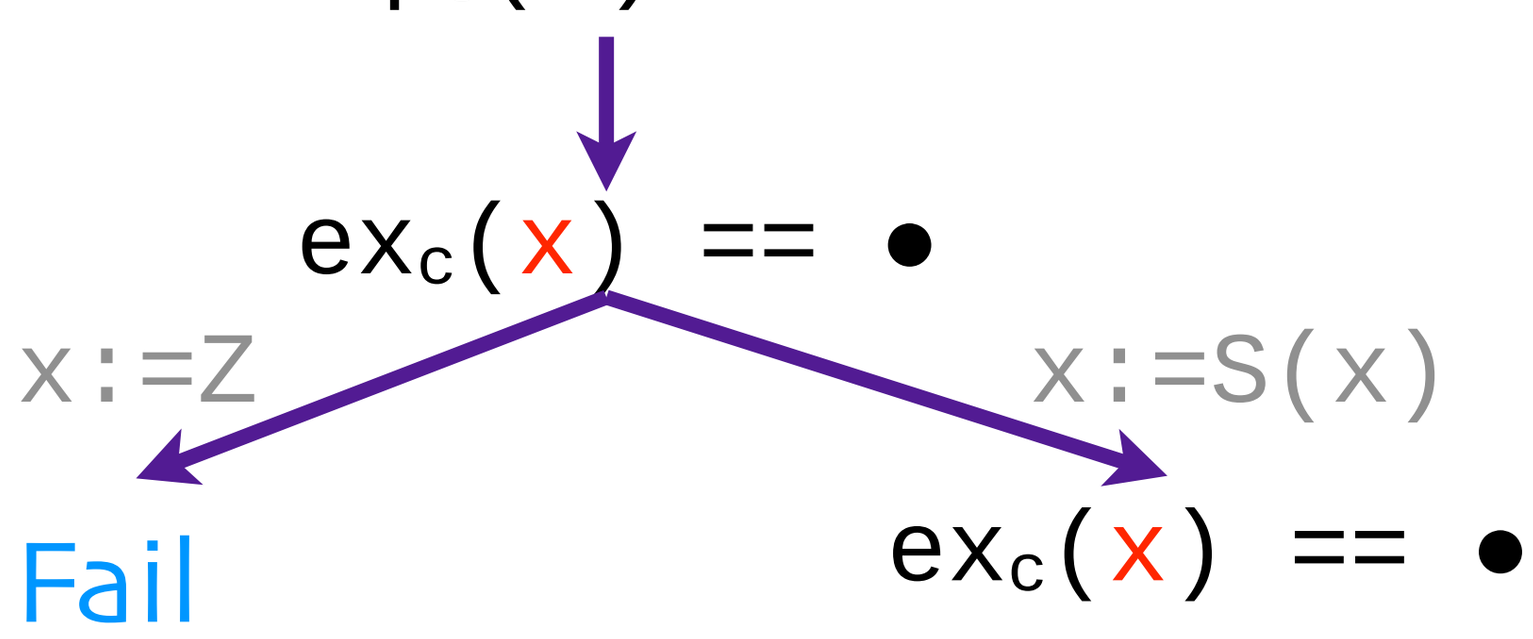
Example

```
exp_c(x)    = k[Z] where k = ex_c(x)
ex_c(Z)     = S(•)
ex_c(S(x)) = k[k[•]] where k = ex_c(x)
```



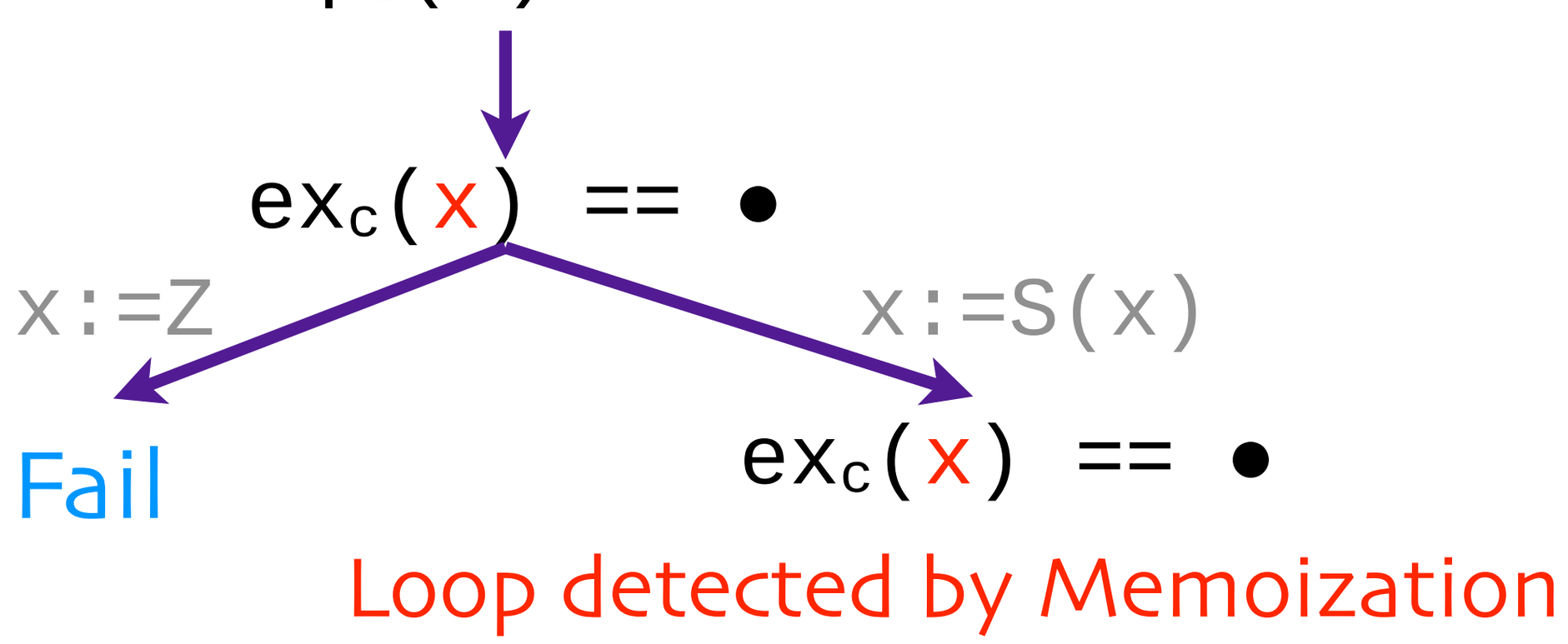
Example

```
exp_c(x)    = k[Z] where k = ex_c(x)
ex_c(Z)     = S(•)
ex_c(S(x)) = k[k[•]] where k = ex_c(x)
```



Example

```
exp_c(x)    = k[Z] where k = ex_c(x)
ex_c(Z)     = S(•)
ex_c(S(x)) = k[k[•]] where k = ex_c(x)
```



Complexity

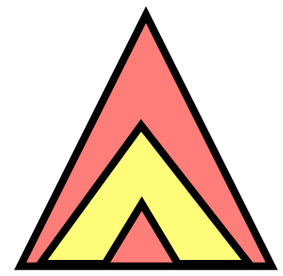
$ex_c(x) == S(S(\bullet))$

In general, each eq-chk has the form of:

$h_c(x) == (K_1, ..., K_m)$

from parameter-linearity

Each K_i is a **linear subcontext** of the original output (fed to inverse computation)



$\Rightarrow \#K = O(n^{\#hole+1}) \leqq$

(n: the size of the original output)

Complexity

$ex_c(x) == S(S(\bullet))$

In general, each eq-chk has the form of:

$h_c(x) == (K_1, ..., K_m)$

from parameter-linearity

Each K_i is a **linear subcontext** of the original output (fed to i

Recall: $f_c(x) = f(x, \bullet_1, .., \bullet_n)$

$\Rightarrow$ $\#K = O(n^{\#hole+1}) \leqq$

(n: the size of the original output)

Complexity

$ex_c(x) == S(S(\bullet))$

In general, each eq-chk has the form of:

$h_c(x) == (K_1, ..., K_m)$

from parameter-linearity

Each K_i is a **linear subcontext** of the original output (fed to i

Recall: $f_c(x) = f(x, \bullet_1, .., \bullet_n)$

$\Rightarrow \#K = O(n^{\#hole+1}) \leqq O(n^{maxArity})$

(n: the size of the original output)

Complexity

$ex_c(x) == S(S(\bullet))$

In general, each eq-chk has the form of:

$h_c(x) == (K_1, ..., K_m)$

from parameter-linearity

Each K_i is a **linear subcontext** of the original output (fed to i

Recall: $f_c(x) = f(x, \bullet_1, .., \bullet_n)$

$\Rightarrow \#K = O(n^{\#hole+1}) \leqq O(n^{maxArity})$

$\Rightarrow \#(EqChk) \leqq$

(n: the size of the original output)

Complexity

$ex_c(x) == S(S(\bullet))$

In general, each eq-chk has the form of:

$h_c(x) == (K_1, ..., K_m)$

$\#h_c \leqq 2^{\#fun}$ and $m \leqq \#fun$ by tupling from parameter-linearity

Each K_i is a **linear subcontext** of the original output (fed to i

Recall: $f_c(x) = f(x, \bullet_1, .., \bullet_n)$

$\Rightarrow \#K = O(n^{\#hole-1}) \leqq O(n^{maxArity})$

$\Rightarrow \#(EqChk) \leqq$

(n: the size of the original output)

Complexity

$ex_c(x) == S(S(\bullet))$

In general, each eq-chk has the form of:

$h_c(x) == (K_1, ..., K_m)$

$\#h_c \leqq 2^{\#fun}$ and $m \leqq \#fun$ by tupling from parameter-linearity

Each K_i is a **linear subcontext** of the original output (fed to i

Recall: $f_c(x) = f(x, \bullet_1, .., \bullet_n)$

$\Rightarrow \#K = O(n^{\#hole-1}) \leqq O(n^{maxArity})$

$\Rightarrow \#(EqChk) \leqq O(n^{maxArity \times \#fun + c})$

(n: the size of the original output)

Complexity

$ex_c(x) == S(S(\bullet))$

In general, each eq-chk has the form of:

$h_c(x) == (K_1, ..., K_m)$

$\#h_c \leqq 2^{\#fun}$ and $m \leqq \#fun$ by tupling

from parameter-linearity

Each K_i is a **linear subcontext** of the original output (fed to i

Recall: $f_c(x) = f(x, \bullet_1, .., \bullet_n)$

$\Rightarrow \#K = O(n^{\#hole-1}) \leqq O(n^{maxArity})$

$\Rightarrow \#(EqChk) \leqq O(n^{maxArity \times \#fun + c})$

(n: the size of the original output)

Polynomial!!

Short Summary

- This talk
  - Idea of a polynomial time inverse computation for parameter-linear MTT
- In the paper
  - A tree automaton to represent infinite results of (memoized) inv-comp
  - Extensions
    - pattern guards
    - bounded parameter copying

Related Work

- Polynomial-time inverse-image comp for a class of MTT [Frisch&Hosoya 07]
  - Inverse computation is a subproblem of inverse-image computation

	ours	**theirs**
multiple data traversals	no restriction	bounded
parameter copies	bounded (see paper)	no restriction

See our paper what causes the difference

Related Work

- Polynomial-time inverse-image comp

```
exp(x) = ex(x,Z)
ex(Z,y)    = S(y)
ex(S(x),y) = ex(x,ex(x,y))
```

```
completeBinTree(x)=cb(x,L)
cb(Z,y)    = y
cb(S(x),y) = cb(x,N(y,y))
```

inverse-image computation

	ours	**theirs**
multiple data traversals	no restriction	bounded
parameter copies	bounded (see paper)	no restriction

See our paper what causes the difference

Conclusion

- An inverse computation method
  - Polynomial-time for parameter-linear Macro-Tree Transducers (MTTs)
    - **Accumulations**
    - **Multiple Data Traversals**

  - Idea: transformation of a program in the class to a **linear non-accumulative context-generating** program

Future Work

1. Range-analysis-based inversion

Overlap

```
rev([], y) = y
rev(a:x,y) = rev(x,a:y)
```

disjoint

```
rev([]) = •
rev(a:x) = k[a:•]
  where k = rev(x)
```

2. More practical copying

- join-like operations in DB query

```
[ (x,y,z) |
   (x,y) <- as, (y',z) <- bs,
   y==y' ]
```