Staging with Class

Ningning Xie Matthew Pickering Andres Löh Nicolas Wu Jeremy Yallop Meng Wang

UNIVERSITY OF CAMBRIDGE

Well-Typed
The Haskell Consultants

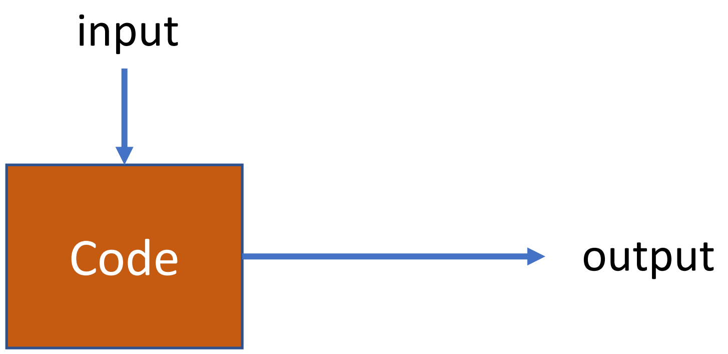
Imperial College London

University of BRISTOL

POPL 2022

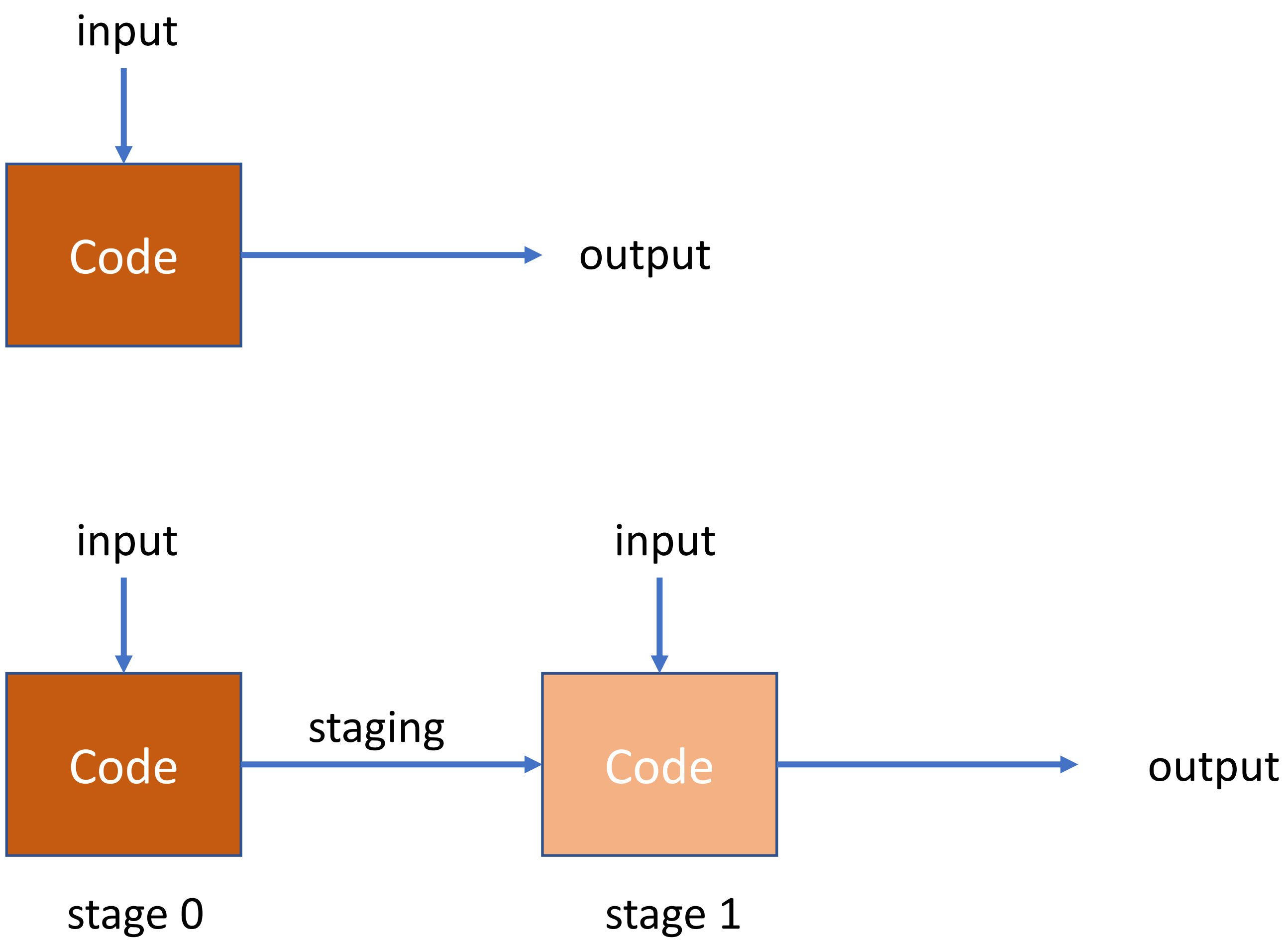
Multi-stage programming

generate efficient code with predictable performance



Multi-stage programming

generate efficient code with predictable performance



Quotations and splices

Quotations and splices

`Code`: program fragment in a future stage

Quotations and splices

`Code`: program fragment in a future stage

Quotations and splices

`Code`: program fragment in a future stage

Quotation

a representation of the expression as program fragment in a future stage

`e :: Int` ⟹ `<e> :: Code Int`

Quotations and splices

`Code`: program fragment in a future stage

Quotation

a representation of the expression as program fragment in a future stage

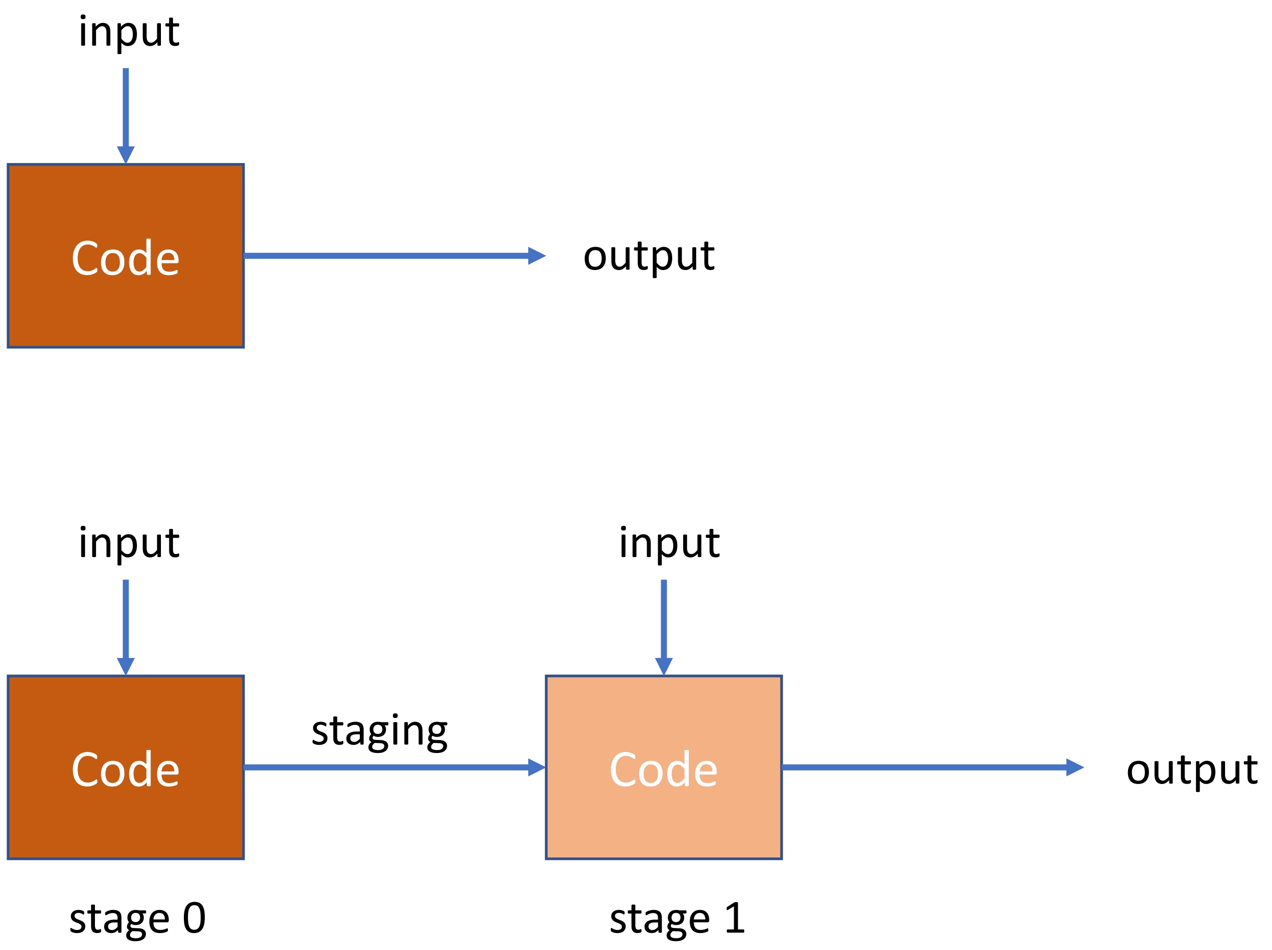
`e :: Int` ⟹ `<e> :: Code Int`

Splice

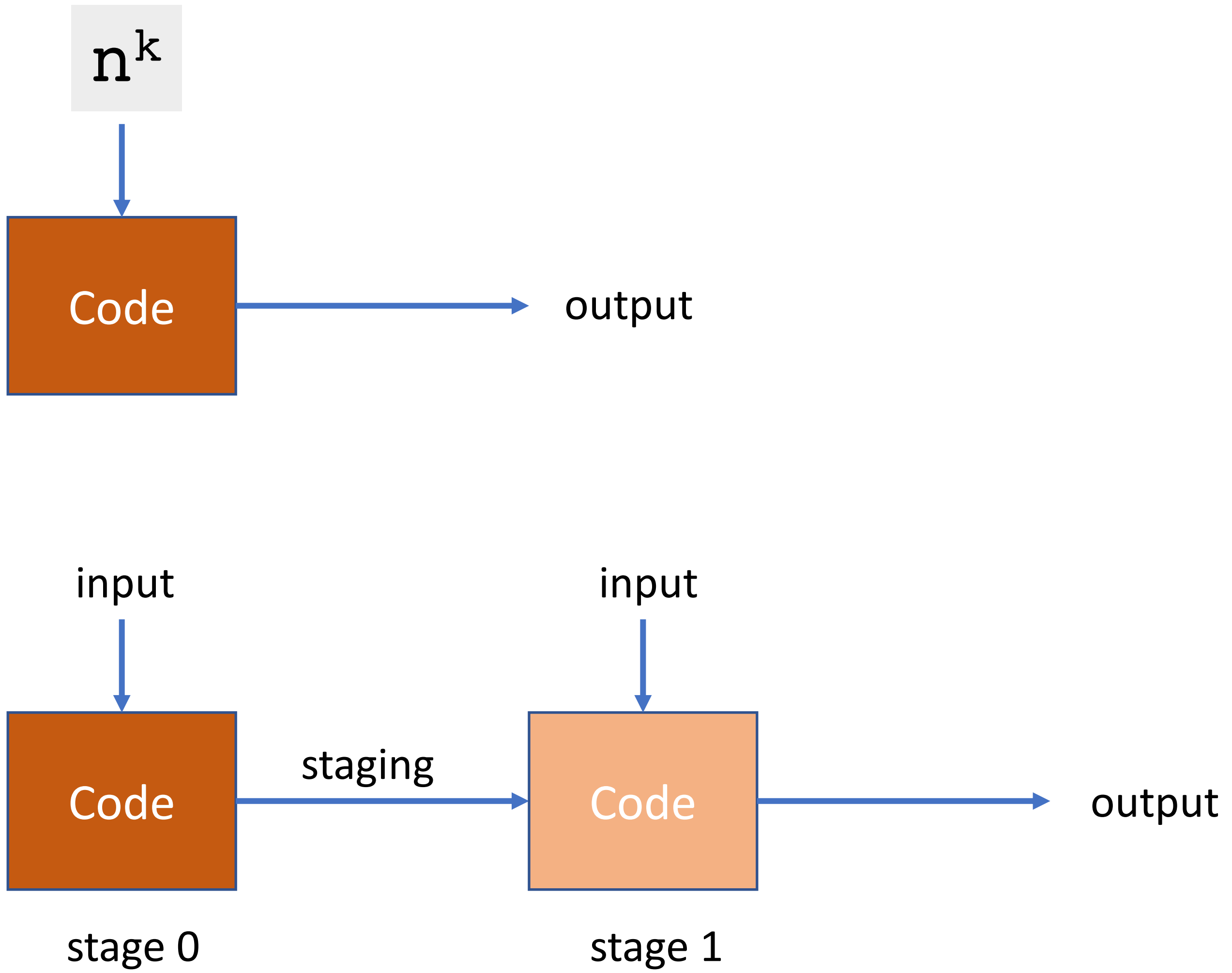
extracts the expression from its representation

`e :: Code Int` ⟹ `$e :: Int`

Multi-stage programming: example

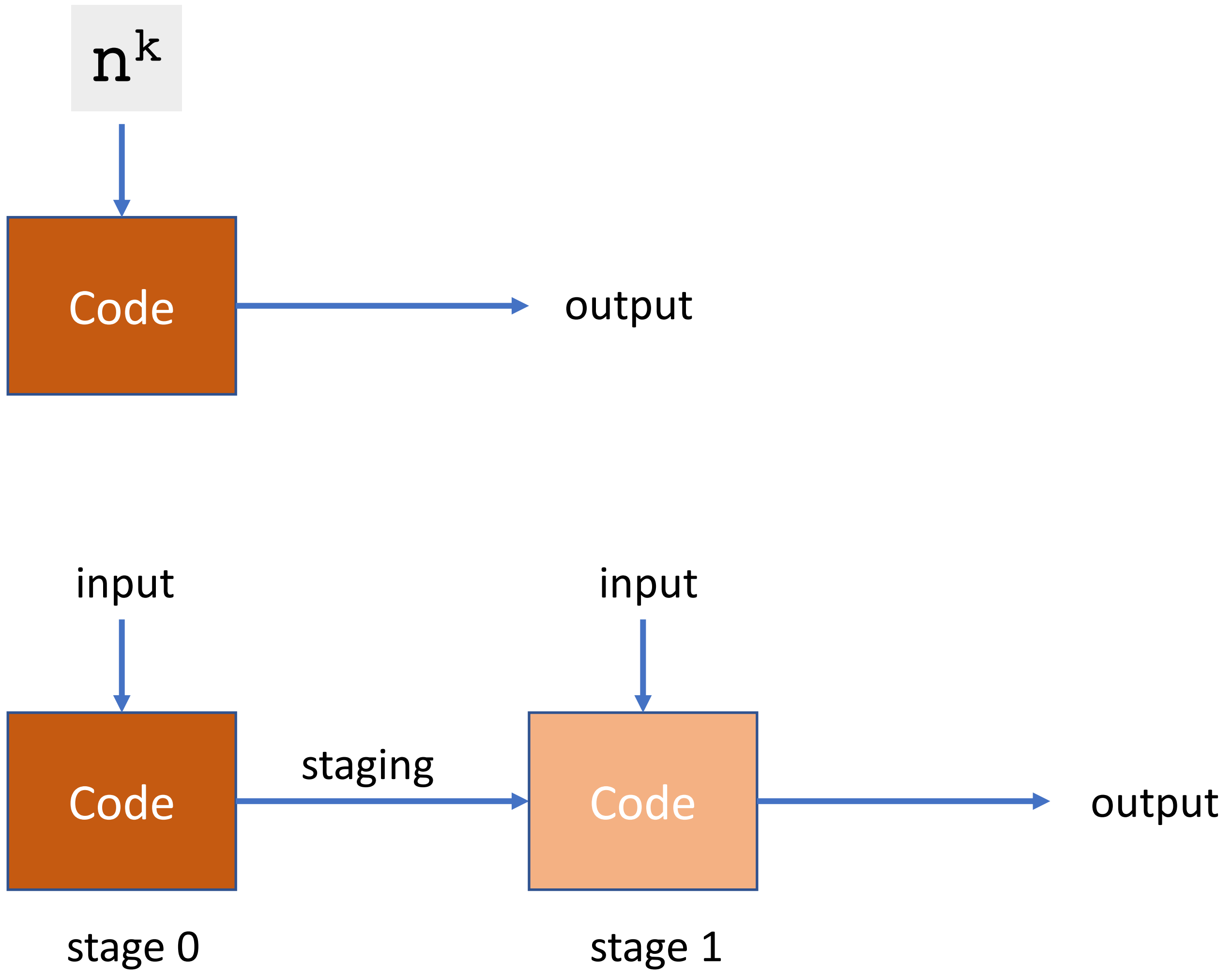


Multi-stage programming: example



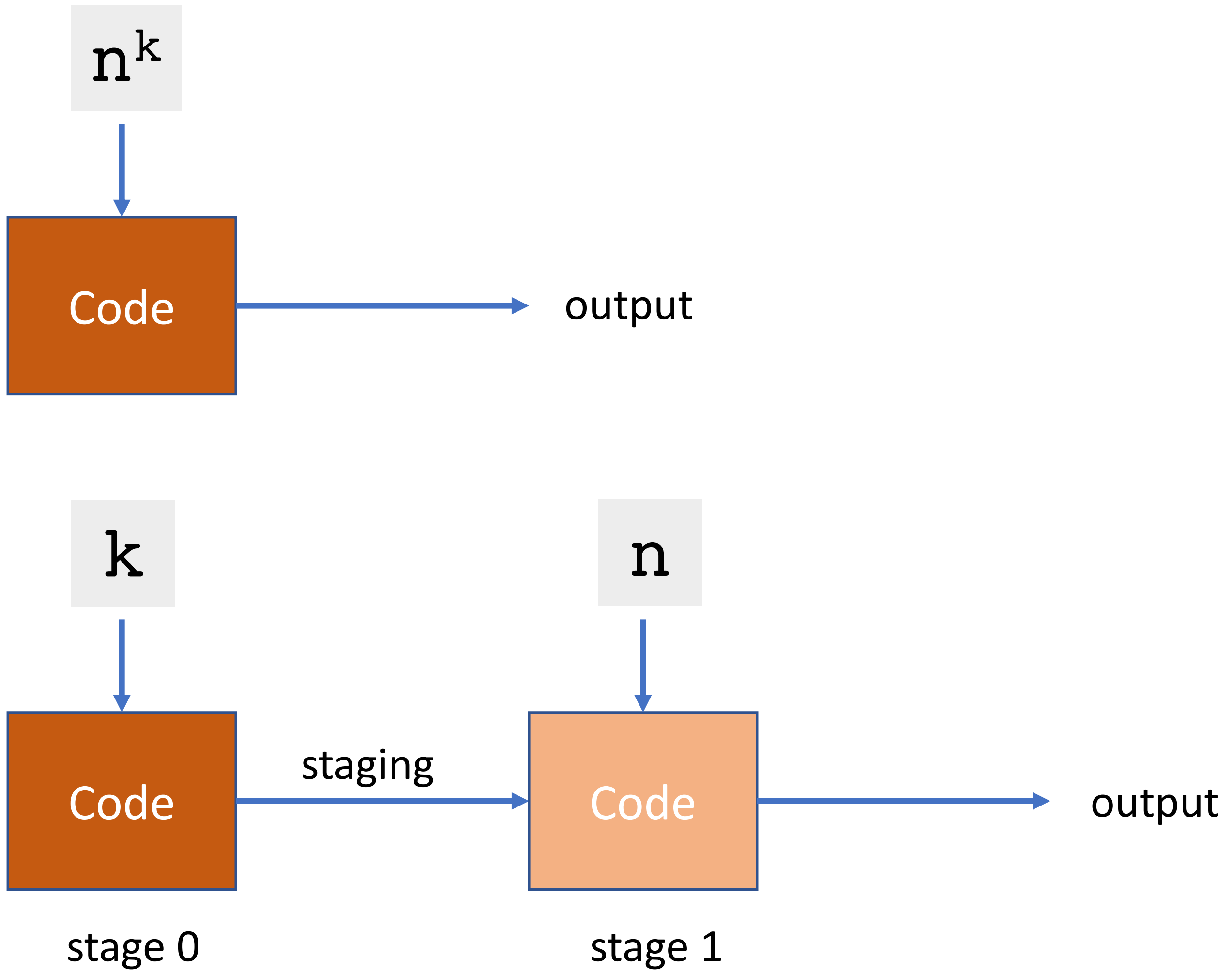
Multi-stage programming: example

```
power :: Int -> Int -> Int
power 0 n = 1
power k n = n * power (k – 1) n

powerFive :: Int -> Int
powerFive n = power 5 n
```



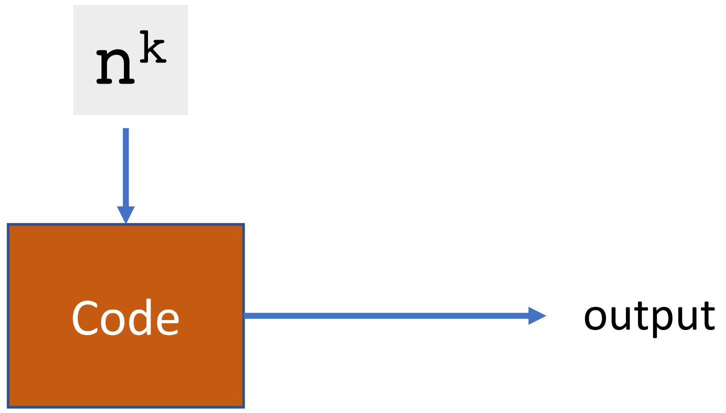
Multi-stage programming: example

```
power :: Int -> Int -> Int
power 0 n = 1
power k n = n * power (k – 1) n

powerFive :: Int -> Int
powerFive n = power 5 n
```

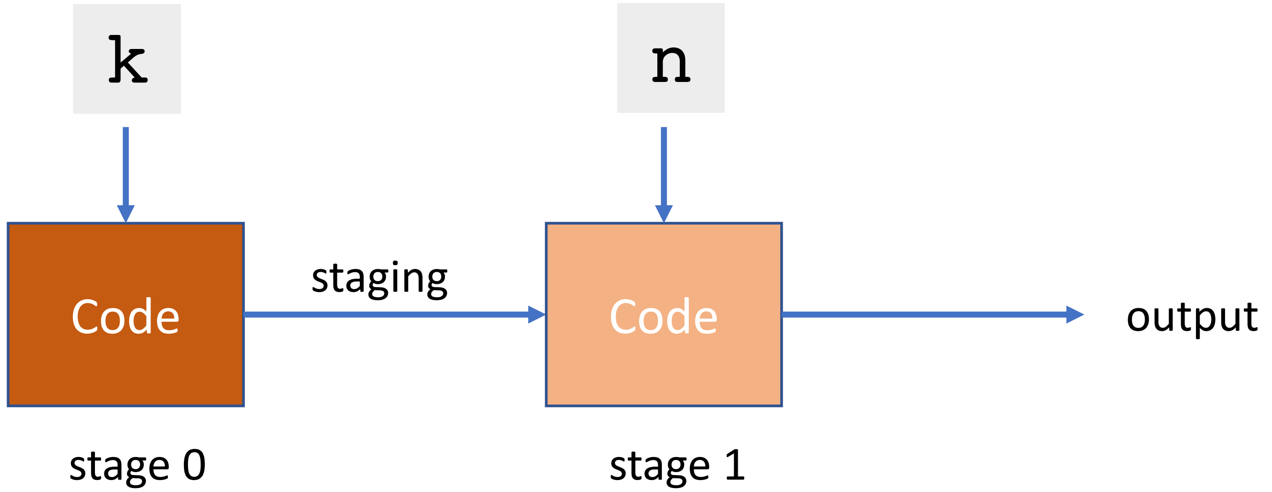


Multi-stage programming: example

```
power :: Int -> Int -> Int
power 0 n = 1
power k n = n * power (k – 1) n

powerFive :: Int -> Int
powerFive n = power 5 n
```

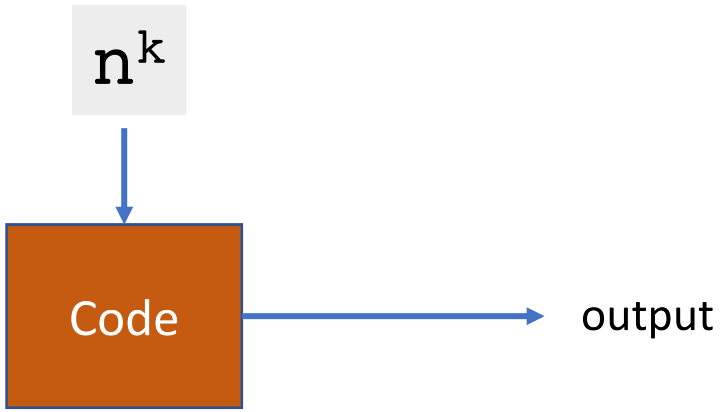


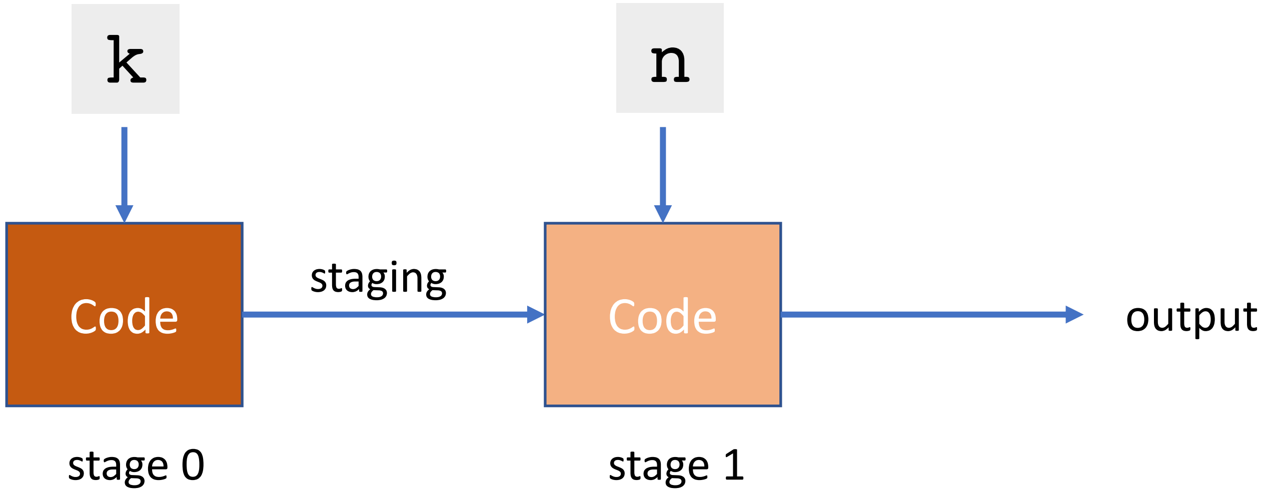

```
qpower :: Int -> Code Int -> Code Int
```



Multi-stage programming: example

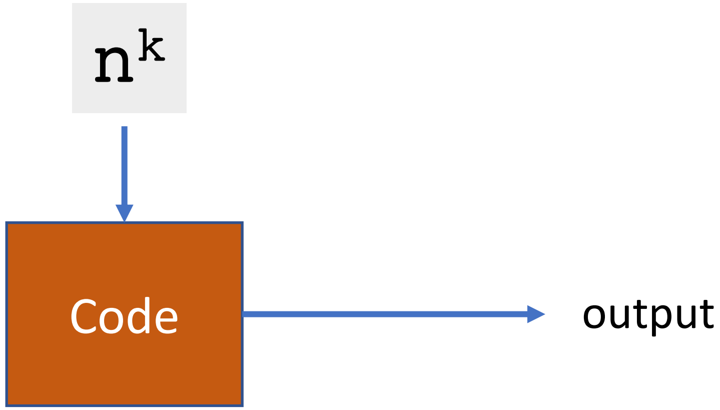
```
power :: Int -> Int -> Int
power 0 n = 1
power k n = n * power (k – 1) n

powerFive :: Int -> Int
powerFive n = power 5 n


qpower :: Int -> Code Int -> Code Int
```



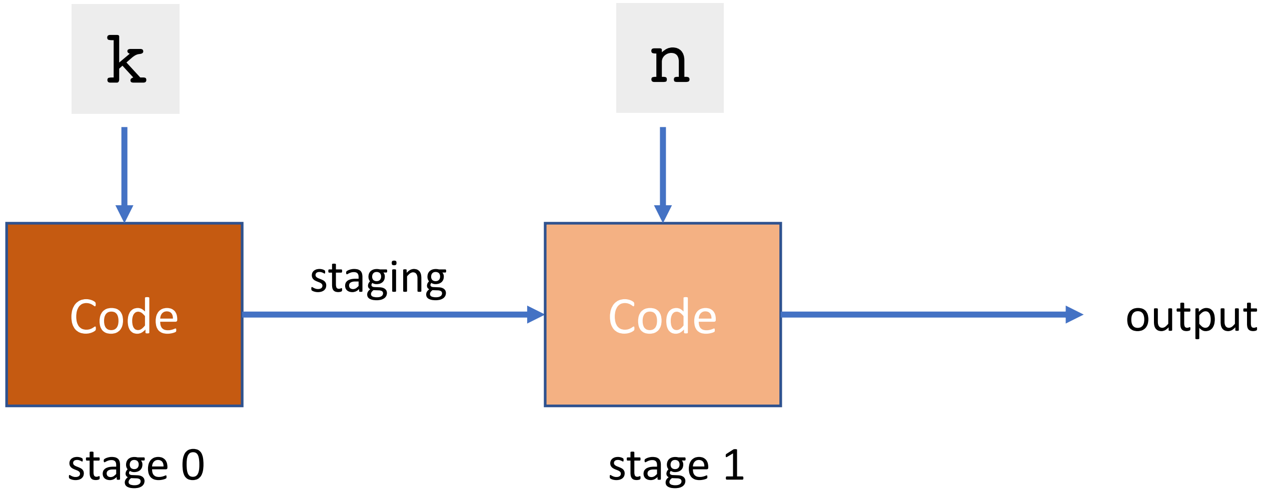


Multi-stage programming: example

```
power :: Int -> Int -> Int
power 0 n = 1
power k n = n * power (k – 1) n

powerFive :: Int -> Int
powerFive n = power 5 n
```

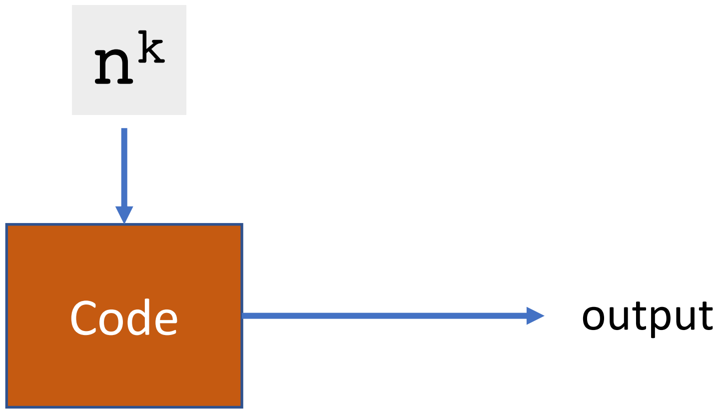



```
qpower :: Int -> Code Int -> Code Int
```

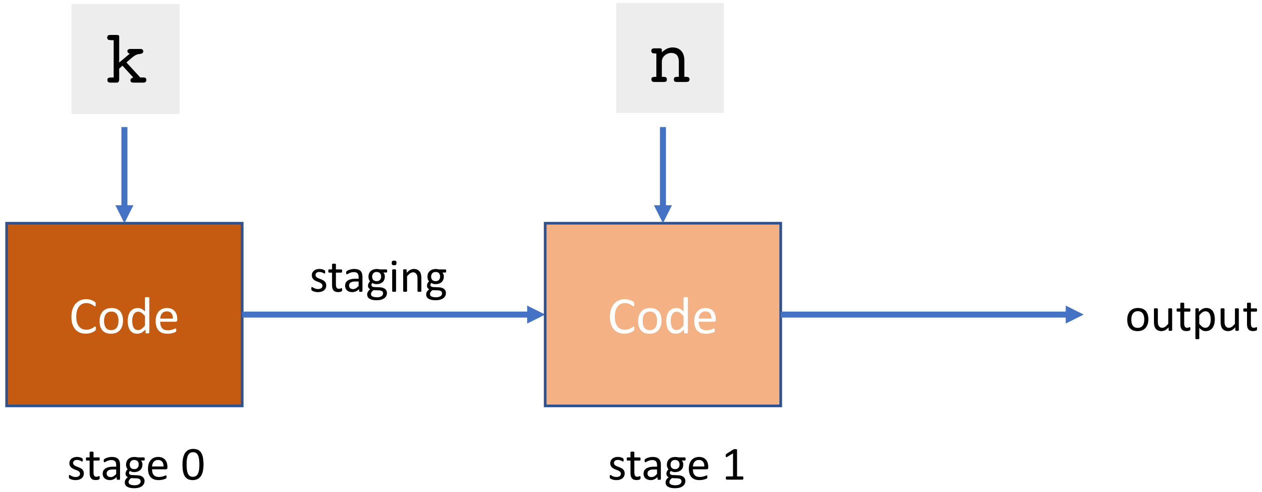


Multi-stage programming: example

```
power :: Int -> Int -> Int
power 0 n = 1
power k n = n * power (k – 1) n

powerFive :: Int -> Int
powerFive n = power 5 n
```

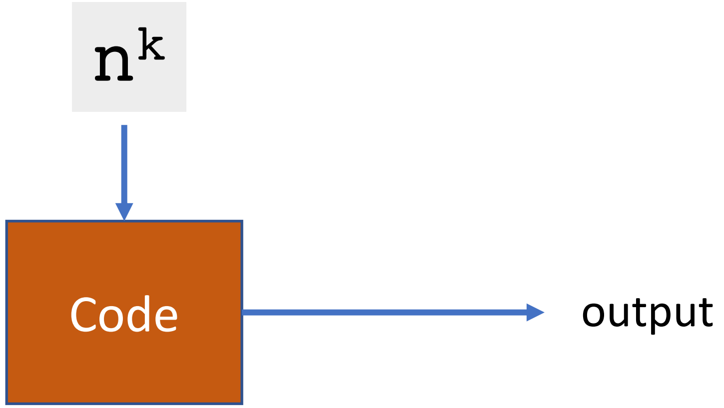



```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
```

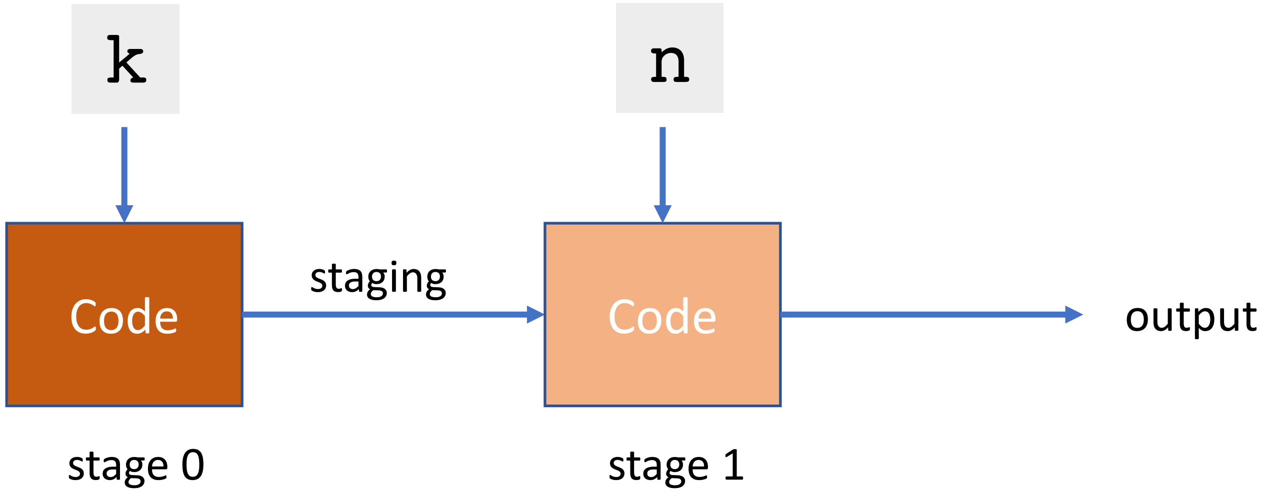


Multi-stage programming: example

```
power :: Int -> Int -> Int
power 0 n = 1
power k n = n * power (k – 1) n

powerFive :: Int -> Int
powerFive n = power 5 n
```

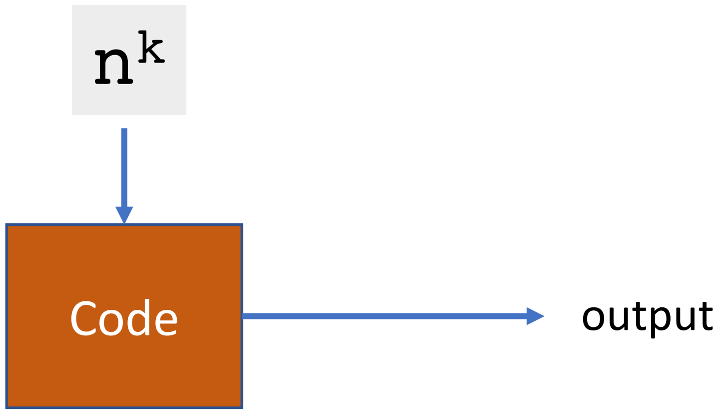



```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
```

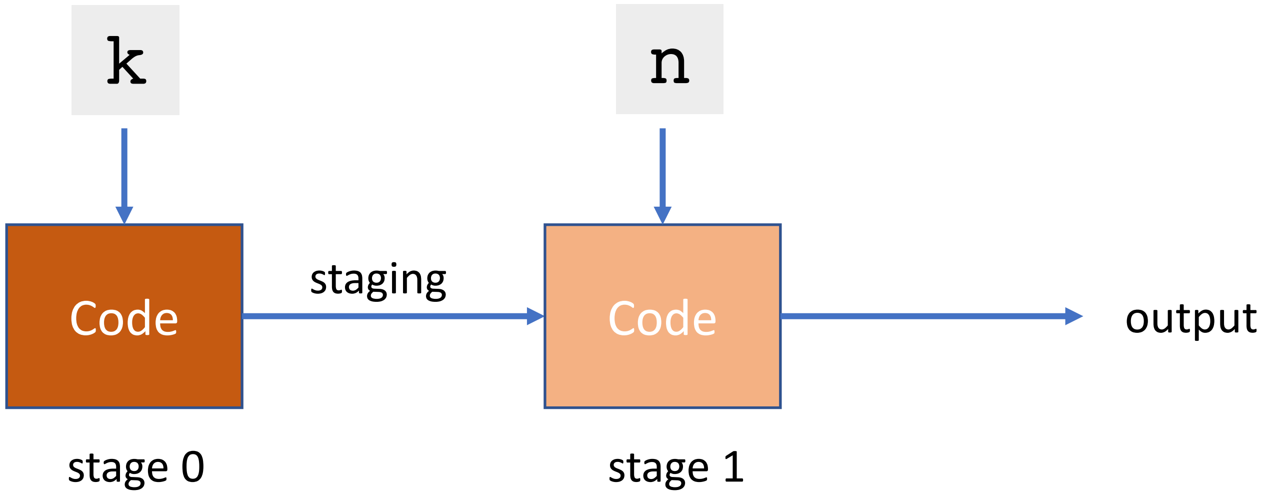


Multi-stage programming: example

```
power :: Int -> Int -> Int
power 0 n = 1
power k n = n * power (k – 1) n

powerFive :: Int -> Int
powerFive n = power 5 n
```

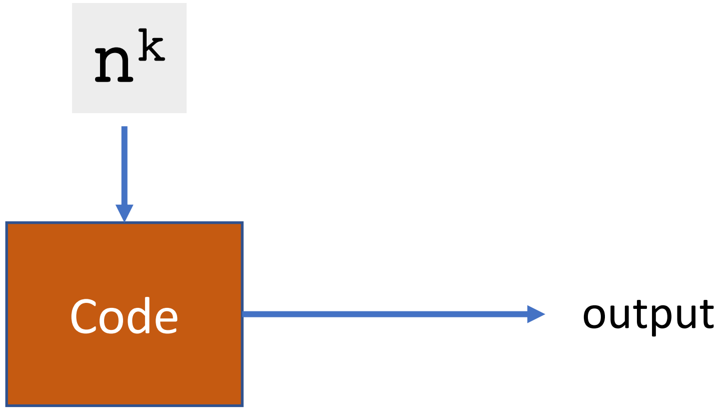



```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
```

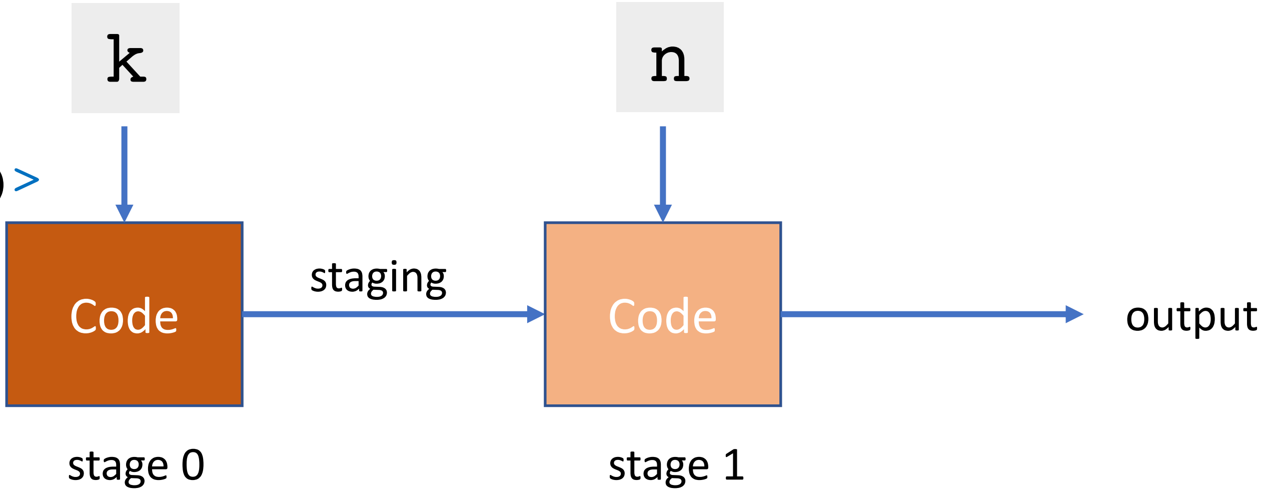


Multi-stage programming: example

```
power :: Int -> Int -> Int
power 0 n = 1
power k n = n * power (k – 1) n

powerFive :: Int -> Int
powerFive n = power 5 n
```

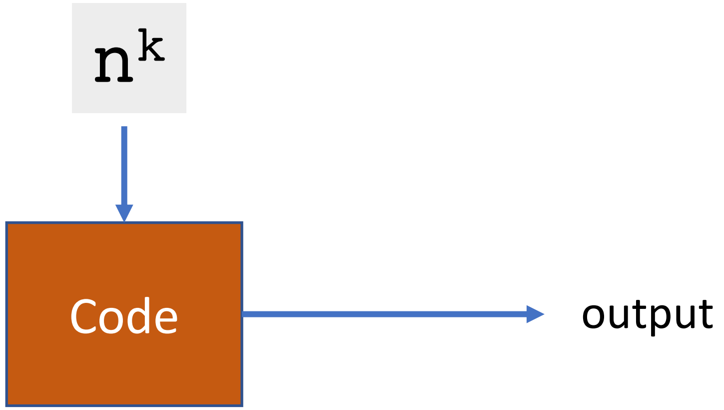



```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>
```

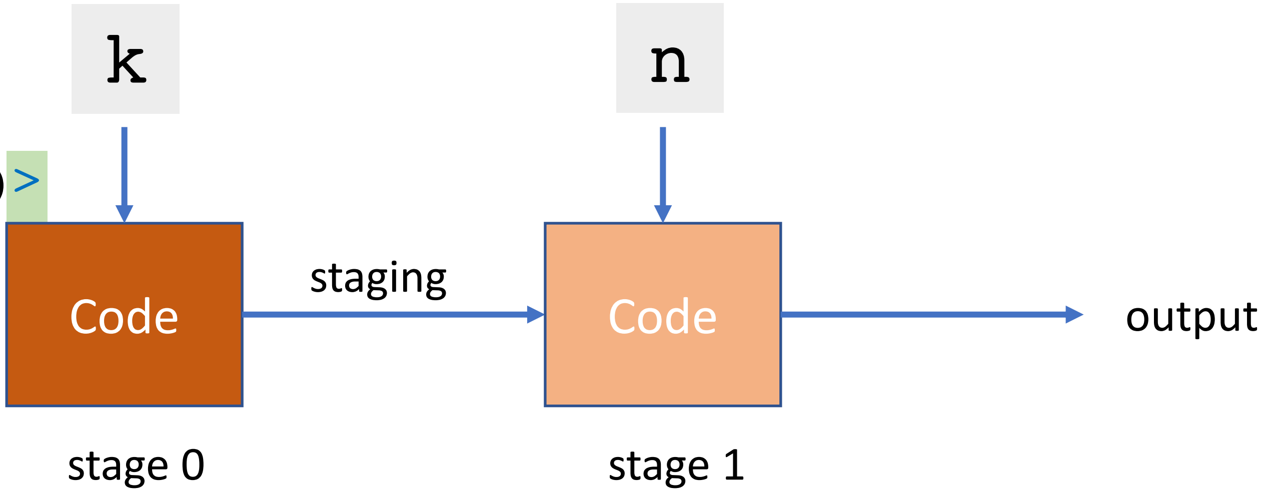


Multi-stage programming: example

```
power :: Int -> Int -> Int
power 0 n = 1
power k n = n * power (k – 1) n

powerFive :: Int -> Int
powerFive n = power 5 n
```

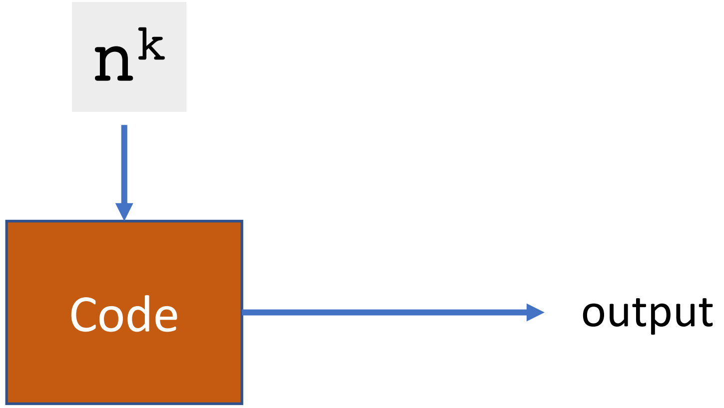



```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>
```

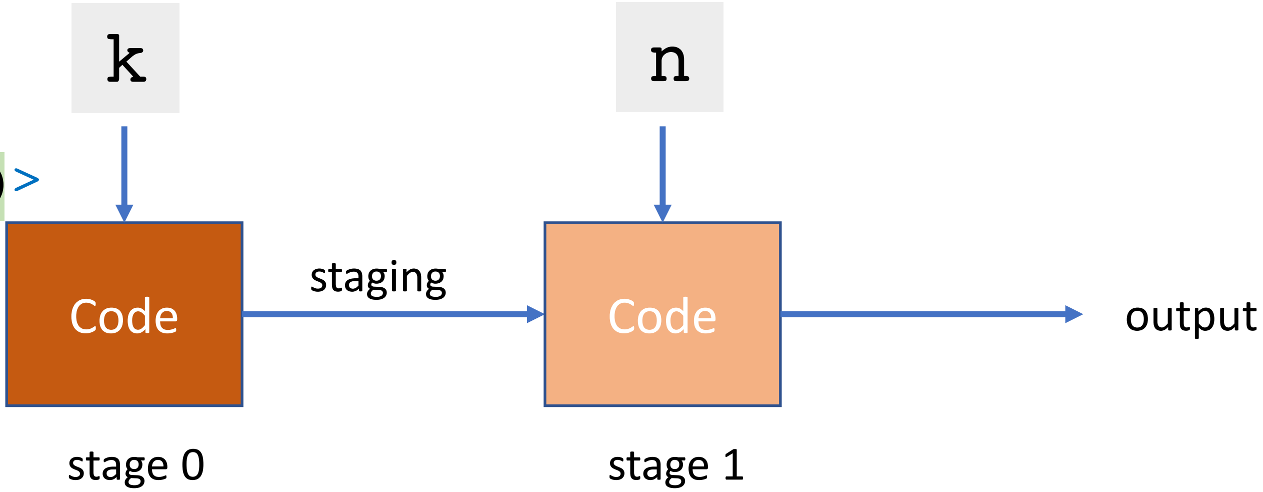


Multi-stage programming: example

```
power :: Int -> Int -> Int
power 0 n = 1
power k n = n * power (k – 1) n

powerFive :: Int -> Int
powerFive n = power 5 n
```

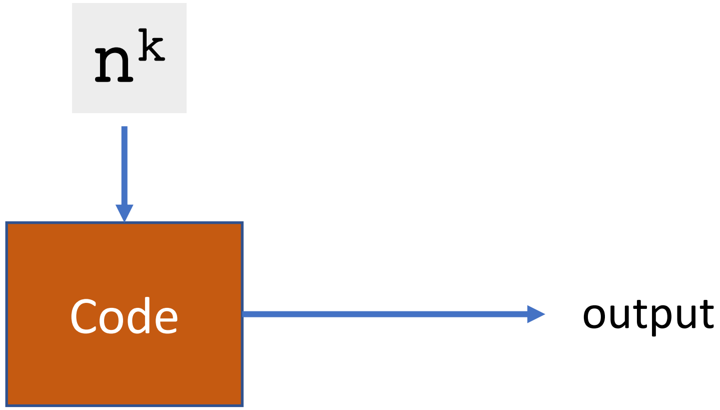



```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>
```

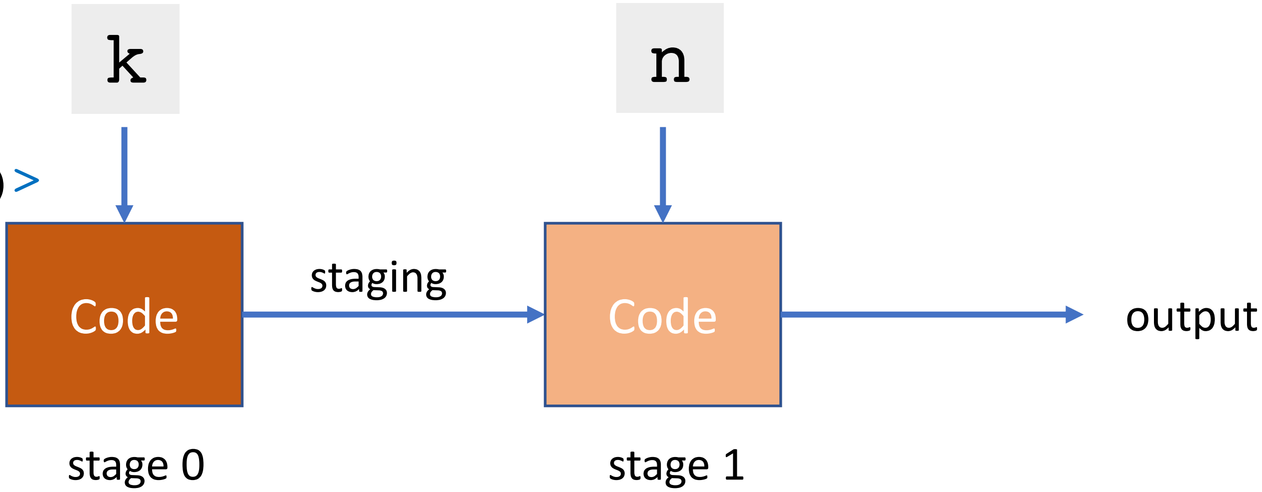


Multi-stage programming: example

```
power :: Int -> Int -> Int
power 0 n = 1
power k n = n * power (k – 1) n

powerFive :: Int -> Int
powerFive n = power 5 n
```

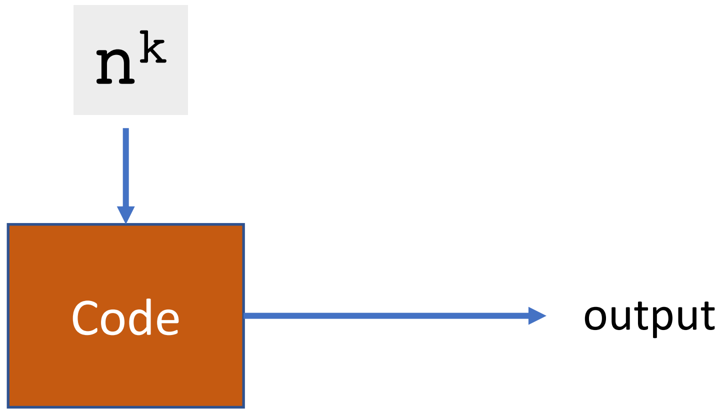



```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>
```

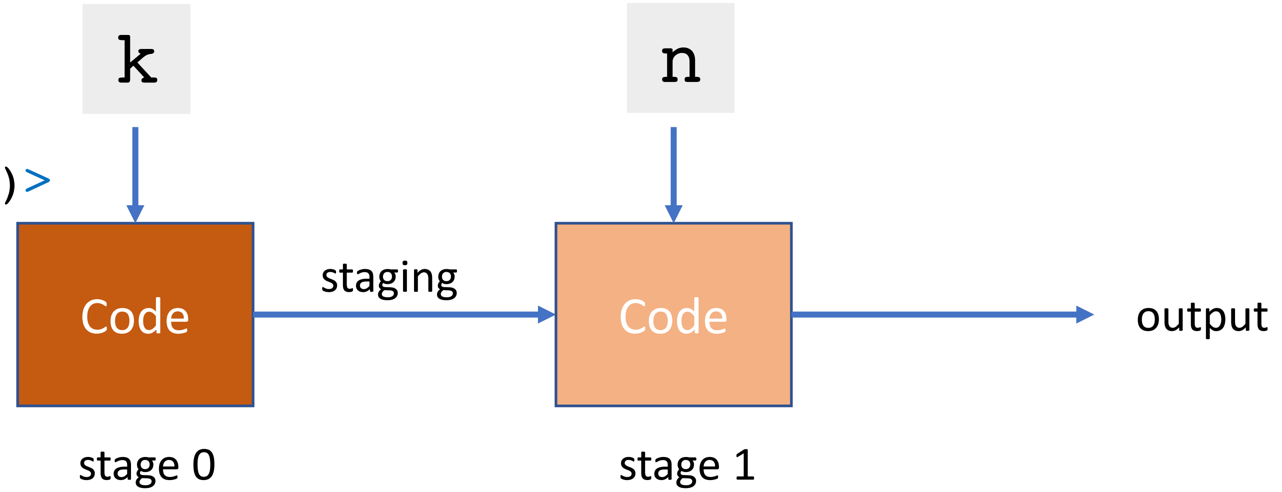


Multi-stage programming: example

```haskell
power :: Int -> Int -> Int
power 0 n = 1
power k n = n * power (k – 1) n

powerFive :: Int -> Int
powerFive n = power 5 n
```

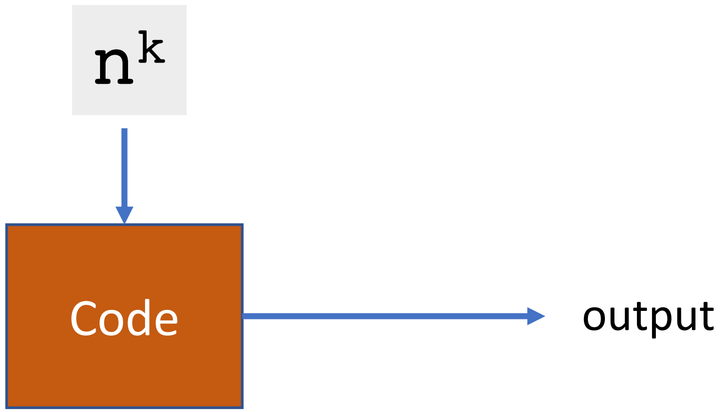



```haskell
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Int -> Int
qpowerFive n = $(qpower 5 <n>)
```

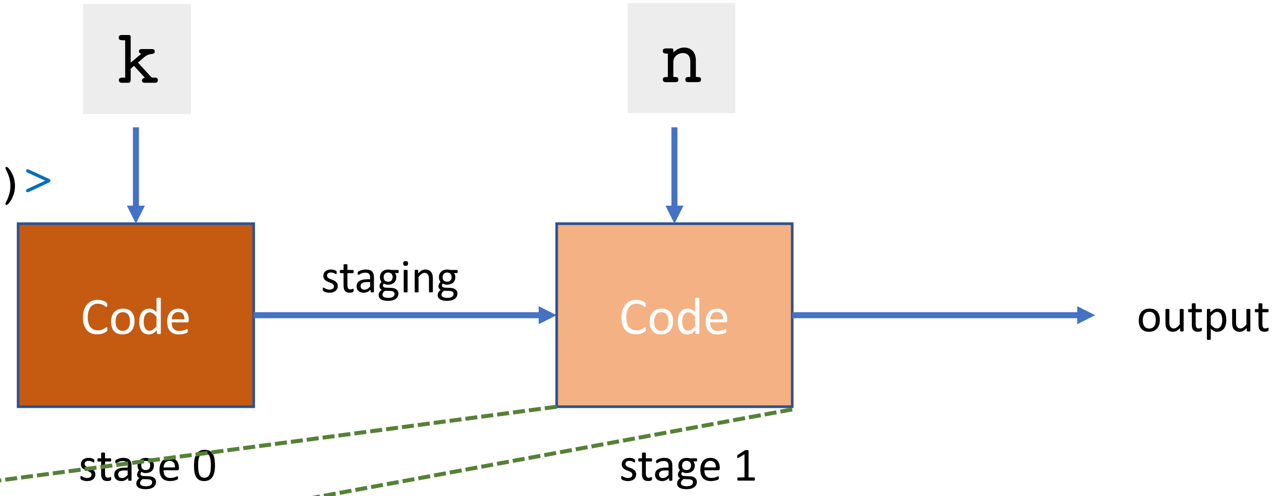


Multi-stage programming: example

```
power :: Int -> Int -> Int
power 0 n = 1
power k n = n * power (k – 1) n

powerFive :: Int -> Int
powerFive n = power 5 n
```




```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Int -> Int
qpowerFive n = $(qpower 5 <n>)
```




```
qpowerFive n = n * n * n * n * n * 1
```

Code generation

```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Int -> Int
qpowerFive n = $(qpower 5 <n>)
```

Code generation

```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Int -> Int
qpowerFive n = $(qpower 5 <n>)
```

⟶

```
n * n * n * n * n * 1
```

Code generation

```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Int -> Int
qpowerFive n = $(qpower 5 <n>)
```

⟶
```
n * n * n * n * n * 1
```

Code generation

```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Int -> Int
qpowerFive n = $(qpower 5 <n>)
           ⟶  $(                                  )


           ⟶  n * n * n * n * n * 1
```

Code generation

```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Int -> Int
qpowerFive n = $(qpower 5 <n>)
          ⟶  $(<$(<n>) * $(qpower (5 – 1) <n>)>)

          ⟶  n * n * n * n * n * 1
```

Code generation

```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Int -> Int
qpowerFive n = $(qpower 5 <n>)
             ⟶ $(<$(<n>) * $(qpower (5 – 1) <n>)>)

             ⟶ n * n * n * n * n * 1
```

Code generation

```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Int -> Int
qpowerFive n = $(qpower 5 <n>)
          ⟶  $(<$(<n>) * $(qpower (5 – 1) <n>)>)

          ⟶  $(<n>) * $(qpower (5 – 1) <n>)



          ⟶  n * n * n * n * n * 1
```

Code generation

```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Int -> Int
qpowerFive n = $(qpower 5 <n>)
          ⟶  $(<$(<n>) * $(qpower (5 – 1) <n>)>)

          ⟶  $(<n>) * $(qpower (5 – 1) <n>)



          ⟶  n * n * n * n * n * 1
```

Code generation

```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Int -> Int
qpowerFive n = $(qpower 5 <n>)
```

```
      ⟶  $(<$(<n>) * $(qpower (5 – 1) <n>)>)

      ⟶  $(<n>) * $(qpower (5 – 1) <n>)

      ⟶  n * n * n * n * n * 1
```

Code generation

```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Int -> Int
qpowerFive n = $(qpower 5 <n>)
```

```
⟶ $(<$(<n>) * $(qpower (5 – 1) <n>)>)

⟶ $(<n>) * $(qpower (5 – 1) <n>)

⟶ n * $(qpower (5 – 1) <n>)



⟶ n * n * n * n * n * 1
```

Code generation

```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Int -> Int
qpowerFive n = $(qpower 5 <n>)
           ⟶  $(<$(<n>) * $(qpower (5 – 1) <n>)>)

           ⟶  $(<n>) * $(qpower (5 – 1) <n>)

           ⟶  n * $(qpower (5 – 1) <n>)



           ⟶  n * n * n * n * n * 1
```

Code generation

```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Int -> Int
qpowerFive n = $(qpower 5 <n>)
          ⟶  $(<$(<n>) * $(qpower (5 – 1) <n>)>)

          ⟶  $(<n>) * $(qpower (5 – 1) <n>)

          ⟶  n * $(qpower (5 – 1) <n>)

          ⟶  n * $(qpower 4 <n>)


          ⟶  n * n * n * n * n * 1
```

Code generation

```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Int -> Int
qpowerFive n = $(qpower 5 <n>)
          ⟶  $(<$(<n>) * $(qpower (5 – 1) <n>)>)

          ⟶  $(<n>) * $(qpower (5 – 1) <n>)

          ⟶  n * $(qpower (5 – 1) <n>)

          ⟶  n * $(qpower 4 <n>)


          ⟶  n * n * n * n * n * 1
```

Code generation

```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Int -> Int
qpowerFive n = $(qpower 5 <n>)
            ⟶ $(<$(<n>) * $(qpower (5 – 1) <n>)>)

            ⟶ $(<n>) * $(qpower (5 – 1) <n>)

            ⟶ n * $(qpower (5 – 1) <n>)

            ⟶ n * $(qpower 4 <n>)


            ⟶ n * n * n * n * n * 1
```

Code generation

```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Int -> Int
qpowerFive n = $(qpower 5 <n>)
             ⟶  $(<$(<n>) * $(qpower (5 – 1) <n>)>)

             ⟶  $(<n>) * $(qpower (5 – 1) <n>)

             ⟶  n * $(qpower (5 – 1) <n>)

             ⟶  n * $(qpower 4 <n>)

             ⟶  ......

             ⟶  n * n * n * n * n * 1
```

But...

```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Int -> Int
qpowerFive n = $(qpower 5 <n>)
```

But...

```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Int -> Int
qpowerFive n = $(qpower 5 <n>)
```

Multi-stage programming and type classes

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

Multi-stage programming and type classes

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

Multi-stage programming and type classes

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

Multi-stage programming and type classes

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

Multi-stage programming and type classes

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

rejected:
No instance for (Num a) arising from a use of 'qpower’
In the expression: qpower 5 <n>

Multi-stage programming and type classes

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

rejected:
No instance for (Num a) arising from a use of 'qpower'
In the expression: qpower 5 <n>

Multi-stage programming and type classes

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

?

rejected:
No instance for (Num a) arising from a use of 'qpower'
In the expression: qpower 5 <n>

Multi-stage programming and type classes

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

rejected:
No instance for (Num a) arising from a use of 'qpower'
In the expression: qpower 5 <n>

Multi-stage programming and type classes

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

?

rejected:
No instance for (Num a) arising from a use of 'qpower’
In the expression: qpower 5 <n>

Multi-stage programming and type classes

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

rejected:
No instance for (Num a) arising from a use of 'qpower'
In the expression: qpower 5 <n>

This talk

unsound

inspire

$\lambda^{[\![\Rightarrow]\!]}$

- Type Classes
- Quotations/Splicing
- Staged type class constraints

type-directed

$F^{[\![\,]\!]}$

- Quotations
- Splice environments

This talk

unsound

inspire

$\lambda^{[\![\Rightarrow]\!]}$

- Type Classes
- Quotations/Splicing
- Staged type class constraints

type-directed

$F^{[\![\,]\!]}$

- Quotations
- Splice environments

This talk

unsound

inspire $\longrightarrow$ $\lambda^{[\![\Rightarrow]\!]}$ type-directed $\longrightarrow$ $F^{[\![\,]\!]}$

$\lambda^{[\![\Rightarrow]\!]}$:

- Type Classes
- Quotations/Splicing
- Staged type class constraints

$F^{[\![\,]\!]}$:

- Quotations
- Splice environments

This talk

unsound

inspire

$\lambda^{[\![\Rightarrow]\!]}$

- Type Classes
- Quotations/Splicing
- Staged type class constraints

type-directed

$F^{[\![]\!]}$

- Quotations
- Splice environments

✓ A solid theoretical foundation for integrating type classes into multi-stage programs

This talk

unsound $\xrightarrow{\text{inspire}}$ $\lambda^{[\![\Rightarrow]\!]}$ $\xrightarrow{\text{type-directed}}$ $F^{[\![]\!]}$

$\lambda^{[\![\Rightarrow]\!]}$:

- Type Classes
- Quotations/Splicing
- Staged type class constraints

$F^{[\![]\!]}$:

- Quotations
- Splice environments

✓ A solid theoretical foundation for integrating type classes into multi-stage programs

✓ Easy to implement and stay close to existing implementations

This talk

unsound

inspire

$\lambda^{[\![\Rightarrow]\!]}$

- Type Classes
- Quotations/Splicing
- Staged type class constraints

type-directed

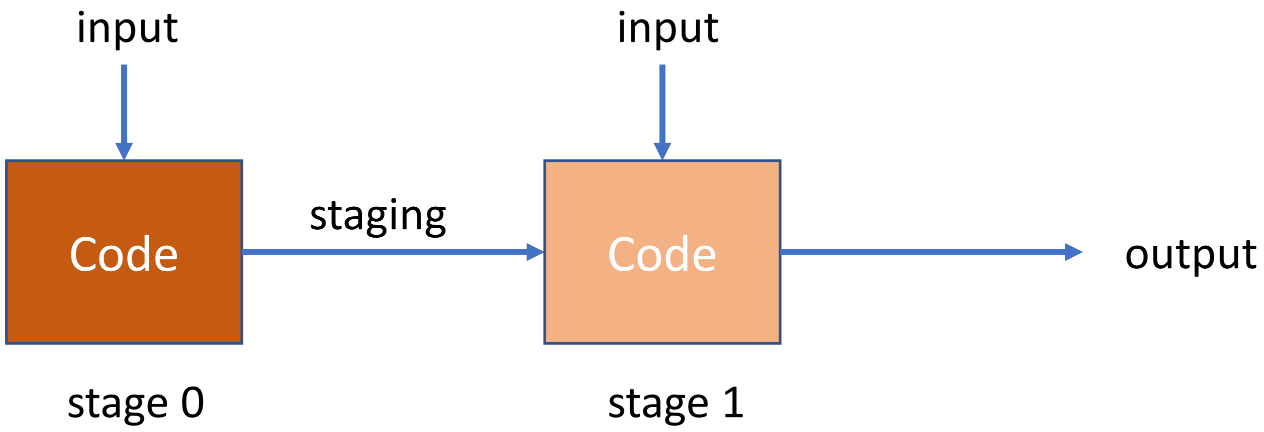
$F^{[\![\,]\!]}$

- Quotations
- Splice environments

✓ A solid theoretical foundation for integrating type classes into multi-stage programs

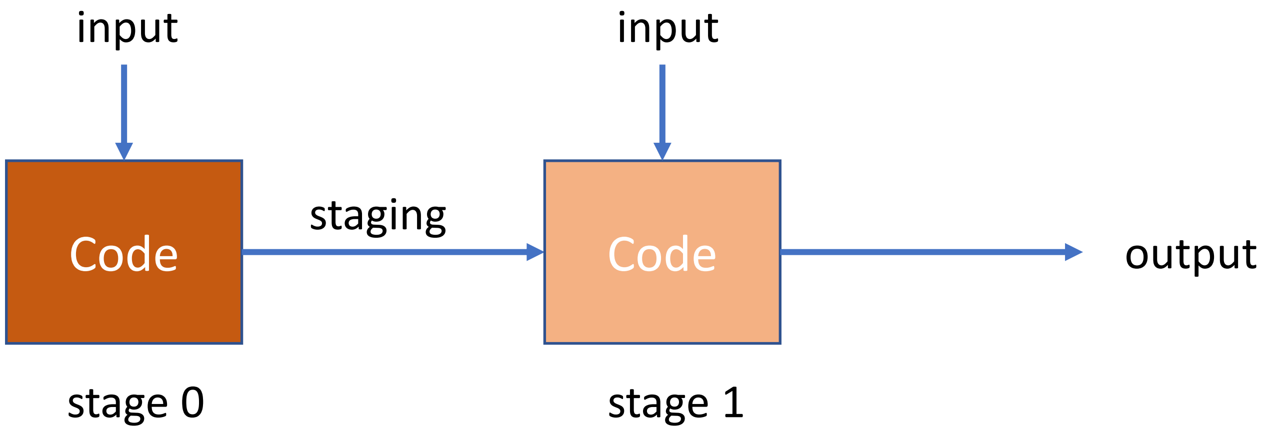
✓ Easy to implement and stay close to existing implementations

How does multi-stage programming ensure type safety?



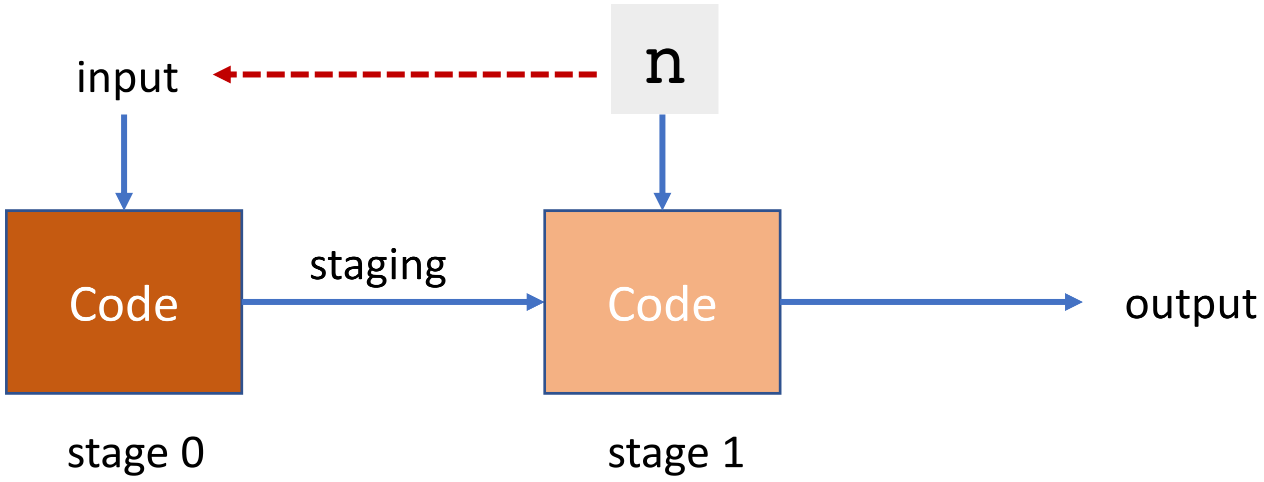
How does multi-stage programming ensure type safety?

```
qpowerN :: Int -> Int
qpowerN n = $(qpower n <n>)
```



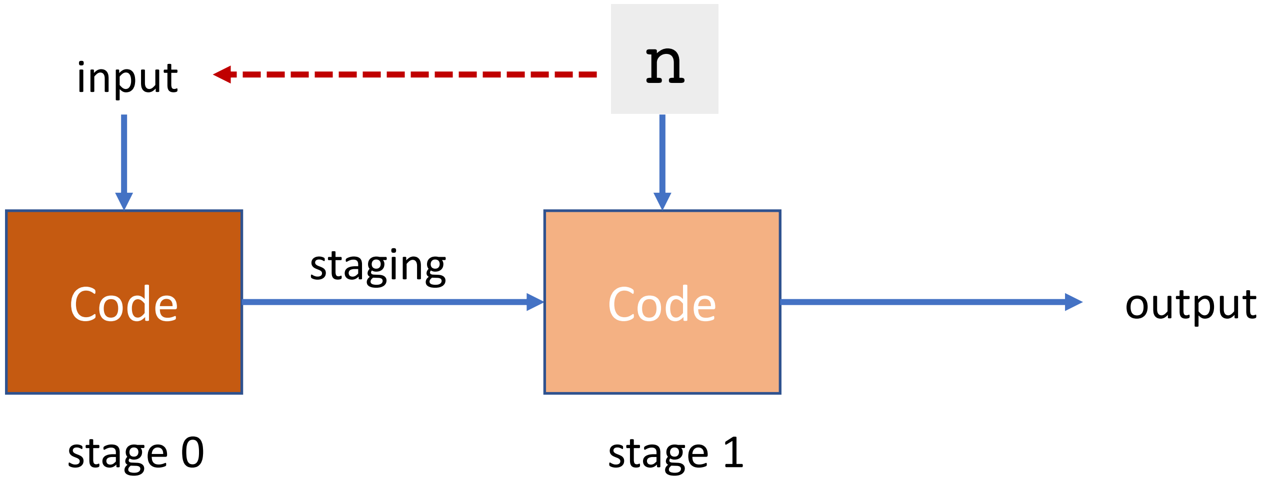
How does multi-stage programming ensure type safety?

```
qpowerN :: Int -> Int
qpowerN n = $(qpower n <n>)
```



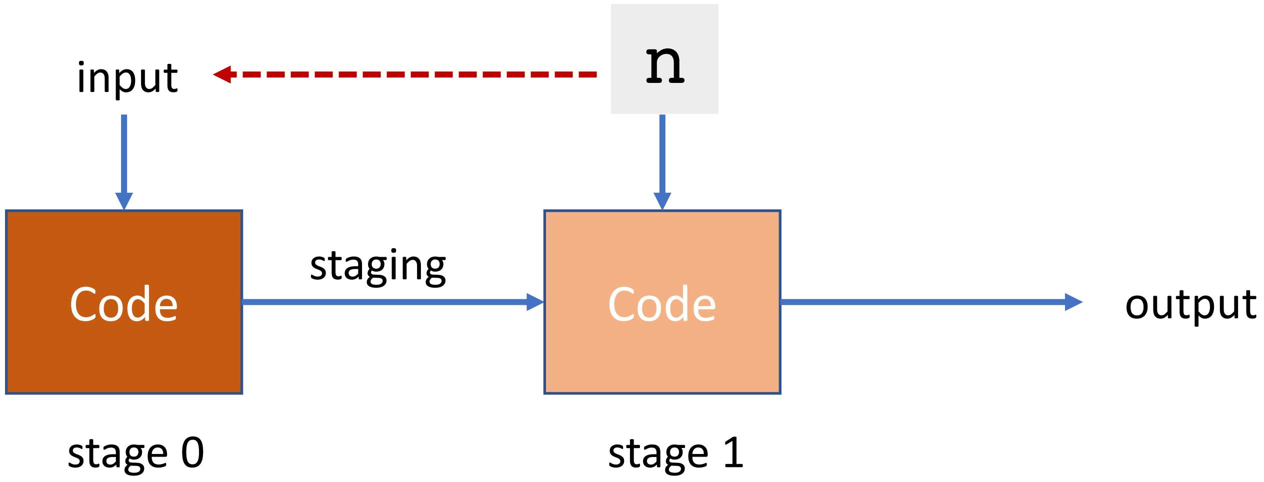
How does multi-stage programming ensure type safety?

```
qpowerN :: Int -> Int
qpowerN n = $(qpower n <n>)
    →  ……
    →  n * $(qpower (n – 1) <n>)
```



How does multi-stage programming ensure type safety?

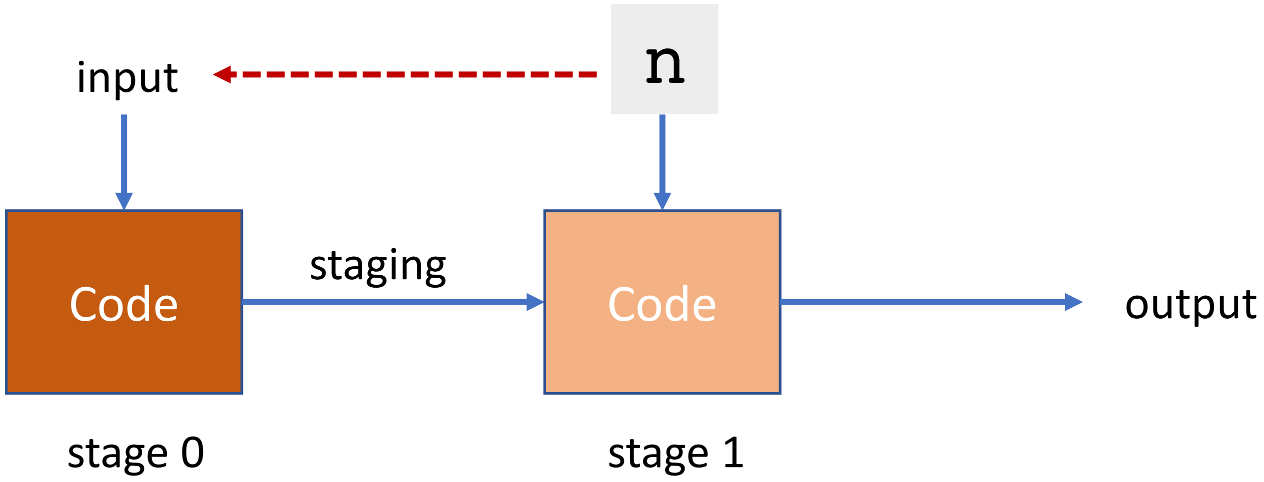
```
qpowerN :: Int -> Int
qpowerN n = $(qpower n <n>)
    ⟶  ......
    ⟶  n * $(qpower (n – 1) <n>)
```



How does multi-stage programming ensure type safety?

```
qpowerN :: Int -> Int
qpowerN n                    n>)

  → n * $(qpower (n – 1) <n>)
```

rejected!



Well-stagedness: the level of an expression

Quotation

a representation of the expression as program fragment in a future stage

```
e :: Int    ⟹   <e> :: Code Int
```

Splice

extracts the expression from its representation

```
e :: Code Int  ⟹   $e :: Int
```

Well-stagedness: the level of an expression

Quotation

a representation of the expression as program fragment in a future stage

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \langle e \rangle : \mathrm{Code}\ \tau}$$

Splice

extracts the expression from its representation

$$\frac{\Gamma \vdash e : \mathrm{Code}\ \tau}{\Gamma \vdash \$e : \tau}$$

$$\boxed{\Gamma \vdash e : \tau}$$

Well-stagedness: the level of an expression

Quotation

a representation of the expression as program fragment in a future stage

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \langle e \rangle : \mathrm{Code}\ \tau}$$

Splice

extracts the expression from its representation

$$\frac{\Gamma \vdash e : \mathrm{Code}\ \tau}{\Gamma \vdash \$e : \tau}$$

$$\boxed{\Gamma \vdash e : \tau}$$

context

Well-stagedness: the level of an expression

Quotation

a representation of the expression as program fragment in a future stage

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \langle e \rangle : \mathrm{Code}\,\tau}$$

Splice

extracts the expression from its representation

$$\frac{\Gamma \vdash e : \mathrm{Code}\,\tau}{\Gamma \vdash \$e : \tau}$$

$$\boxed{\Gamma \vdash e : \tau}$$

context

`x : int`

Well-stagedness: the level of an expression

Quotation

a representation of the expression as program fragment in a future stage

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \langle e \rangle : \mathrm{Code}\ \tau}$$

Splice

extracts the expression from its representation

$$\frac{\Gamma \vdash e : \mathrm{Code}\ \tau}{\Gamma \vdash \$e : \tau}$$

$$\boxed{\underbrace{\Gamma}_{\text{context}} \vdash \underbrace{e}_{\text{expr}} : \tau}$$

`x : int`

Well-stagedness: the level of an expression

Quotation

a representation of the expression as program fragment in a future stage

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \langle e \rangle : \mathrm{Code}\ \tau}$$

Splice

extracts the expression from its representation

$$\frac{\Gamma \vdash e : \mathrm{Code}\ \tau}{\Gamma \vdash \$e : \tau}$$

$$\underbrace{\Gamma}_{\text{context}} \vdash \underbrace{e}_{\text{expr}} : \underbrace{\tau}_{\text{type}}$$

context — `x : int`

Well-stagedness: the level of an expression

Quotation

a representation of the expression as program fragment in a future stage

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \langle e \rangle : \mathrm{Code}\ \tau}$$

Splice

extracts the expression from its representation

$$\frac{\Gamma \vdash e : \mathrm{Code}\ \tau}{\Gamma \vdash \$e : \tau}$$

level: evaluation order of expressions

$$\boxed{\Gamma \vdash^{n} e : \tau}$$

context expr type

`x : int`

Well-stagedness: the level of an expression

Quotation

a representation of the expression as program fragment in a future stage

$$\frac{\Gamma \vdash^{n+1} e : \tau}{\Gamma \vdash^{n} \langle e \rangle : \mathrm{Code}\,\tau}$$

Splice

extracts the expression from its representation

$$\frac{\Gamma \vdash^{n-1} e : \mathrm{Code}\,\tau}{\Gamma \vdash^{n} \$e : \tau}$$

level: evaluation order of expressions

$$\boxed{\Gamma \vdash^{n} e : \tau}$$

context expr type

`x : int`

Well-stagedness: the level of an expression

Quotation

a representation of the expression as program fragment in a future stage

$$\frac{\Gamma \vdash^{n+1} e : \tau}{\Gamma \vdash^{n} \langle e \rangle : \mathrm{Code}\,\tau}$$

Splice

extracts the expression from its representation

$$\frac{\Gamma \vdash^{n-1} e : \mathrm{Code}\,\tau}{\Gamma \vdash^{n} \$e : \tau}$$

level: evaluation order of expressions

$$\Gamma \vdash^{n} e : \tau$$

leveled context expr type

`x : int`

Well-stagedness: the level of an expression

Quotation

a representation of the expression as program fragment in a future stage

$$\frac{\Gamma \vdash^{n+1} e : \tau}{\Gamma \vdash^{n} \langle e \rangle : \mathrm{Code}\ \tau}$$

Splice

extracts the expression from its representation

$$\frac{\Gamma \vdash^{n-1} e : \mathrm{Code}\ \tau}{\Gamma \vdash^{n} \$e : \tau}$$

level: evaluation order of expressions

$$\boxed{\Gamma \vdash^{n} e : \tau}$$

leveled context expr type

`x : (int , 0)`

Well-stagedness: the level restriction

Quotation

a representation of the expression as program fragment in a future stage

$$\frac{\Gamma \vdash^{n+1} e : \tau}{\Gamma \vdash^{n} \langle e \rangle : \mathrm{Code}\,\tau}$$

Splice

extracts the expression from its representation

$$\frac{\Gamma \vdash^{n-1} e : \mathrm{Code}\,\tau}{\Gamma \vdash^{n} \$e : \tau}$$

level: evaluation order of expressions

The level restriction: each variable is used only at the level in which it is bound

leveled context expr type

x : (int , 0)

Well-stagedness: the level restriction

Quotation

a representation of the expression as program fragment in a future stage

$$\frac{\Gamma \vdash^{n+1} e : \tau}{\Gamma \vdash^{n} \langle e \rangle : \mathrm{Code}\,\tau}$$

Splice

extracts the expression from its representation

$$\frac{\Gamma \vdash^{n-1} e : \mathrm{Code}\,\tau}{\Gamma \vdash^{n} \$e : \tau}$$

The level restriction: each variable is used only at the level in which it is bound

```
qpowerN :: Int -> Int
qpowerN n = $(qpower n <n>)
```

Well-stagedness: the level restriction

Quotation

a representation of the expression as program fragment in a future stage

$$\frac{\Gamma \vdash^{n+1} e : \tau}{\Gamma \vdash^{n} \langle e \rangle : \mathrm{Code}\, \tau}$$

Splice

extracts the expression from its representation

$$\frac{\Gamma \vdash^{n-1} e : \mathrm{Code}\, \tau}{\Gamma \vdash^{n} \$e : \tau}$$

level: evaluation order of expressions

The level restriction: each variable is used only at the level in which it is bound

```
qpowerN ::
qpow          (qpower n <n>)
```

rejected!

Is the problem with qpower well-stageness?

The level restriction: each variable is used only at the level in which it is bound

well-staged?

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

Is the problem with qpower well-stageness?

The level restriction: each variable is used only at the level in which it is bound

well-staged?

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>
```

?

```
qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

?

Is the problem with qpower well-stageness?

The level restriction: each variable is used only at the level in which it is bound

well-staged?

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>
```

?

```
qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

?

Is the problem with qpower well-stageness?

The level restriction: each variable is used only at the level in which it is bound

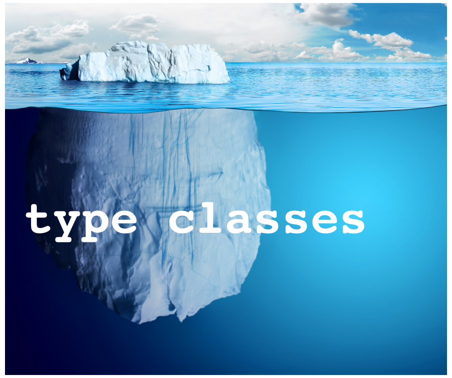
well-staged?

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>
```

?

```
qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

?



Well-staged type classes

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

Well-staged type classes

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>
```

```
qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

Well-staged type classes

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

dictionary-passing elaboration

Well-staged type classes

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

dictionary-passing elaboration

```
qpower :: NumDict a -> Int -> Code a -> Code a
qpower dNum 0 n = <1>
qpower dNum k n = < (*) dNum $(n) $(qpower dNum (k – 1)) n >

qpowerFive :: NumDict a -> a -> a
qpowerFive dNum n = $(qpower dNum 5 <n>)
```

Well-staged type classes

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

dictionary-passing elaboration

```
qpower :: NumDict a -> Int -> Code a -> Code a
qpower dNum 0 n = <1>
qpower dNum k n = < (*) dNum $(n) $(qpower dNum (k – 1)) n >

qpowerFive :: NumDict a -> a -> a
qpowerFive dNum n = $(qpower dNum 5 <n>)
```

Well-staged type classes

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

dictionary-passing elaboration

```
qpower :: NumDict a -> Int -> Code a -> Code a
qpower dNum 0 n = <1>
qpower dNum k n = < (*) dNum $(n) $(qpower dNum (k – 1)) n >

qpowerFive :: NumDict a -> a -> a
qpowerFive dNum n = $(qpower dNum 5 <n>)
```

Well-staged type classes

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

dictionary-passing elaboration

```
qpower :: NumDict a -> Int -> Code a -> Code a
qpower dNum 0 n = <1>
qpower dNum k n = < (*) dNum $(n) $(qpower dNum (k – 1)) n >

qpowerFive :: NumDict a -> a -> a
qpowerFive dNum n = $(qpower dNum 5 <n>)
```

Well-staged type classes

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

dictionary-passing elaboration

```
qpower :: NumDict a -> Int -> Code a -> Code a
qpower dNum 0 n = <1>
qpower dNum k n = < (*) dNum $(n) $(qpower dNum (k – 1)) n >

qpowerFive :: NumDict a -> a -> a
qpowerFive dNum n = $(qpower dNum 5 <n>)
```

Well-staged type classes

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

dictionary-passing elaboration

```
qpower :: NumDict a -> Int -> Code a -> Code a
qpower dNum 0 n = <1>
qpower dNum k n = < (*) dNum $(n) $(qpower dNum (k – 1)) n >

qpowerFive :: NumDict a -> a -> a
qpowerFive dNum n = $(qpower dNum 5 <n>)
```

Well-staged type classes

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

dictionary-passing elaboration

```
qpower :: NumDict a -> Int -> Code a -> Code a
qpower dNum 0 n = <1>
qpower dNum k n = < (*) dNum $(n) $(qpower dNum (k – 1)) n >

qpowerFive :: NumDict a -> a -> a
qpowerFive dNum n = $(qpower dNum 5 <n>)
```

Well-staged type classes

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

dictionary-passing elaboration

```
qpower :: NumDict a -> Int -> Code a -> Code a
qpower dNum 0 n = <1>
qpower dNum k n = < (*) dNum $(n) $(qpower dNum (k – 1)) n >

qpowerFive :: NumDict a -> a -> a
qpowerFive dNum n = $(qpower dNum 5 <n>)
```

Well-staged type classes

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

dictionary-passing elaboration

```
qpower :: NumDict a -> Int -> Code a -> Code a
qpower dNum 0 n = <1>
qpower dNum k n = < (*) dNum $(n) $(qpower dNum (k – 1)) n >

qpowerFive :: NumDict a -> a -> a
qpowerFive dNum n = $(qpower dNum 5 <n>)
```

Well-staged type classes

```
qpower :: Num a => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

well-staged?

dictionary-passing elaboration

```
qpower :: NumDict a -> Int -> Code a -> Code a
qpower dNum 0 n = <1>
qpower dNum k n = < (*) dNum $(n) $(qpower dNum (k – 1)) n >
```

✗

```
qpowerFive :: NumDict a -> a -> a
qpowerFive dNum n = $(qpower dNum 5 <n>)
```

✗

Key idea: staged type class constraints

$$\lambda^{[\![\Rightarrow]\!]}$$

Key idea: staged type class constraints

unstaged	**staged**
Int	Code Int
Num a	

Key idea: staged type class constraints

unstaged	**staged**
Int	Code Int
Num a	CodeC (Num a)

Key idea: staged type class constraints

Key idea: staged type class constraints

```
qpower :: CodeC (Num a) => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1)) n>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

Key idea: staged type class constraints

```
qpower :: CodeC (Num a) => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1)) n>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

Key idea: staged type class constraints

```
qpower :: CodeC (Num a) => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1)) n>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

dictionary-passing elaboration

```
qpower :: Code (NumDict a) -> Int -> Code a -> Code a
qpower dNum 0 n = <1>
qpower dNum k n = < (*) $(dNum) $(n) $(qpower dNum (k – 1)) n>

qpowerFive :: NumDict a -> a -> a
qpowerFive dNum n = $(qpower <dNum> 5 <n>)
```

Key idea: staged type class constraints

```
qpower :: CodeC (Num a) => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1)) n>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

dictionary-passing elaboration

```
qpower :: Code (NumDict a) -> Int -> Code a -> Code a
qpower dNum 0 n = <1>
qpower dNum k n = < (*) $(dNum) $(n) $(qpower dNum (k – 1)) n>

qpowerFive :: NumDict a -> a -> a
qpowerFive dNum n = $(qpower <dNum> 5 <n>)
```

Key idea: staged type class constraints

```
qpower :: CodeC (Num a) => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1)) n>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

dictionary-passing elaboration

```
qpower :: Code (NumDict a) -> Int -> Code a -> Code a
qpower dNum 0 n = <1>
qpower dNum k n = < (*) $(dNum) $(n) $(qpower dNum (k – 1)) n>

qpowerFive :: NumDict a -> a -> a
qpowerFive dNum n = $(qpower <dNum> 5 <n>)
```

Key idea: staged type class constraints

```
qpower :: CodeC (Num a) => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1)) n>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

dictionary-passing elaboration

```
qpower :: Code (NumDict a) -> Int -> Code a -> Code a
qpower dNum 0 n = <1>
qpower dNum k n = < (*) $(dNum) $(n) $(qpower dNum (k – 1)) n>

qpowerFive :: NumDict a -> a -> a
qpowerFive dNum n = $(qpower <dNum> 5 <n>)
```

Key idea: staged type class constraints

```
qpower :: CodeC (Num a) => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1)) n>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

dictionary-passing elaboration

```
qpower :: Code (NumDict a) -> Int -> Code a -> Code a
qpower dNum 0 n = <1>
qpower dNum k n = < (*) $(dNum) $(n) $(qpower dNum (k – 1)) n>

qpowerFive :: NumDict a -> a -> a
qpowerFive dNum n = $(qpower <dNum> 5 <n>)
```

Key idea: staged type class constraints

```
qpower :: CodeC (Num a) => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1)) n>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

dictionary-passing elaboration

```
qpower :: Code (NumDict a) -> Int -> Code a -> Code a
qpower dNum 0 n = <1>
qpower dNum k n = < (*) $(dNum) $(n) $(qpower dNum (k – 1)) n>

qpowerFive :: NumDict a -> a -> a
qpowerFive dNum n = $(qpower <dNum> 5 <n>)
```

Key idea: staged type class constraints

```
qpower :: CodeC (Num a) => Int -> Code a -> Code a
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1)) n>

qpowerFive :: Num a => a -> a
qpowerFive n = $(qpower 5 <n>)
```

dictionary-passing elaboration

```
qpower :: Code (NumDict a) -> Int -> Code a -> Code a
qpower dNum 0 n = <1>
qpower dNum k n = < (*) $(dNum) $(n) $(qpower dNum (k – 1)) n>

qpowerFive :: NumDict a -> a -> a
qpowerFive dNum n = $(qpower <dNum> 5 <n>)
```

well-staged!

Constraint resolution

```
incr :: Num a => a -> a
incr x = x + 1
```

```
incr :: NumDict a -> a ->a
incr dNum x = (+) dNum x
```

Constraint resolution

```
incr :: Num a => a -> a
incr x = x + 1
```

```
incr :: NumDict a -> a ->a
incr dNum x = (+) dNum x
```

Constraint resolution

$$\Gamma \models C \rightsquigarrow e$$

```
incr :: Num a => a -> a
incr x = x + 1
```

```
incr :: NumDict a -> a ->a
incr dNum x = (+) dNum x
```

Constraint resolution

$$\Gamma \models C \rightsquigarrow e$$

```
incr :: Num a => a -> a
incr x = x + 1
```

$$\frac{ev : C \in \Gamma}{\Gamma \models C \rightsquigarrow ev}$$

```
incr :: NumDict a -> a ->a
incr dNum x = (+) dNum x
```

Constraint resolution

```
incr :: Num a => a -> a
incr x = x + 1
```

$$\Gamma \models C \rightsquigarrow e$$

$$\frac{ev : C \in \Gamma}{\Gamma \models C \rightsquigarrow ev}$$

```
incr :: NumDict a -> a ->a
incr dNum x = (+) dNum x
```

$$\frac{dNum : Num\ a \in \Gamma}{\Gamma \models Num\ a \rightsquigarrow dNum}$$

Level-indexed constraint resolution

$$\Gamma \models^{n} C \rightsquigarrow e$$

```
incr :: Num a => a -> a
incr x = x + 1
```

$$\frac{ev : (C, n) \in \Gamma}{\Gamma \models^{n} C \rightsquigarrow ev}$$

```
incr :: NumDict a -> a ->a
incr dNum x = (+) dNum x
```

$$\frac{dNum : Num\ a \in \Gamma}{\Gamma \models Num\ a \rightsquigarrow dNum}$$

Level-indexed constraint resolution

$$\Gamma \models^{n} C \rightsquigarrow e$$

```
incr :: Num a => a -> a
incr x = x + 1
```

$$\frac{ev : (C, n) \in \Gamma}{\Gamma \models^{n} C \rightsquigarrow ev}$$

```
incr :: NumDict a -> a ->a
incr dNum x = (+) dNum x
```

$$\frac{dNum : (Num\ a, 0) \in \Gamma}{\Gamma \models^{0} Num\ a \rightsquigarrow dNum}$$

Level-indexed constraint resolution

```
incr :: Num a => a -> a
incr x = x + 1
```

```
incr :: NumDict a -> a ->a
incr dNum x = (+) dNum x
```

$$\frac{dNum : (Num\ a, 0) \in \Gamma}{\Gamma \models^{0} Num\ a \rightsquigarrow dNum}$$

$$\boxed{\Gamma \models^{n} C \rightsquigarrow e}$$

$$\frac{ev : (C, n) \in \Gamma}{\Gamma \models^{n} C \rightsquigarrow ev}$$

$$\frac{\Gamma \models^{n+1} C \rightsquigarrow e}{\Gamma \models^{n} \mathrm{CodeC}\ C \rightsquigarrow \langle e \rangle}$$

Level-indexed constraint resolution

$$\Gamma \models^{n} C \rightsquigarrow e$$

```
incr :: Num a => a -> a
incr x = x + 1
```

```
incr :: NumDict a -> a ->a
incr dNum x = (+) dNum x
```

$$\frac{dNum : (Num\ a, 0) \in \Gamma}{\Gamma \models^{0} Num\ a \rightsquigarrow dNum}$$

$$\frac{ev : (C, n) \in \Gamma}{\Gamma \models^{n} C \rightsquigarrow ev}$$

$$\frac{\Gamma \models^{n+1} C \rightsquigarrow e}{\Gamma \models^{n} \mathrm{CodeC}\, C \rightsquigarrow \langle e \rangle}$$

$$\frac{\Gamma \models^{n-1} \mathrm{CodeC}\, C \rightsquigarrow e}{\Gamma \models^{n} C \rightsquigarrow \$e}$$

This talk

unsound

inspire $\longrightarrow$ $\lambda^{[\![\Rightarrow]\!]}$ type-directed $\longrightarrow$ $F^{[\![\,]\!]}$

- Type Classes
- Quotations/Splicing
- Staged type class constraints

- Quotations
- Splice environments

✓ A solid theoretical foundation for integrating type classes into multi-stage programs

✓ easy to implement and stay close to existing implementations

This talk

unsound

inspire

$\lambda^{\llbracket \Rightarrow \rrbracket}$

- Type Classes
- Quotations/Splicing
- Staged type class constraints

type-directed

$F^{\llbracket\rrbracket}$

- Quotations
- Splice environments

✓ A solid theoretical foundation for integrating type classes into multi-stage programs

✓ easy to implement and stay close to existing implementations

How to evaluate staged programs?

How to evaluate staged programs?

$$e_1 \; \langle e_2 \; \$ e_3 \rangle$$

How to evaluate staged programs?

level

$$e_1 \; \langle e_2 \; \$ e_3 \rangle$$

How to evaluate staged programs?

level **0**

$$e_1 \; \langle e_2 \, \$ e_3 \rangle$$

How to evaluate staged programs?

level **0** **1**

$$e_1 \; \langle e_2 \; \$ e_3 \rangle$$

How to evaluate staged programs?

level 0 1 0

$$e_1 \ \langle e_2 \ \$ e_3 \rangle$$

How to evaluate staged programs?

level 0 1 0

$$e_1 \; \langle e_2 \; \$ e_3 \rangle \quad \longrightarrow \quad v_1 \; \langle e_2 \; \$ v_3 \rangle$$

Key idea: splice environments

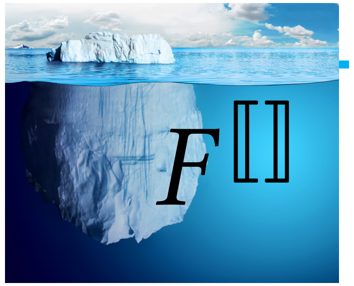
$\lambda^{[\![\Rightarrow]\!]}$

$$e_1 \; \langle e_2 \, \$ e_3 \rangle$$

Key idea: splice environments

$\lambda^{[\![\Rightarrow]\!]}$

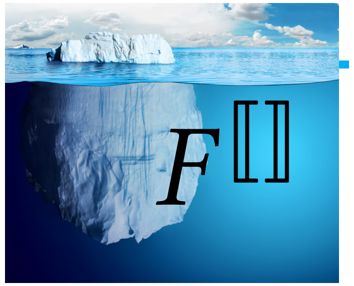
$$e_1 \; \langle e_2 \, \$ e_3 \rangle$$



Key idea: splice environments

$\lambda^{[\![\Rightarrow]\!]}$

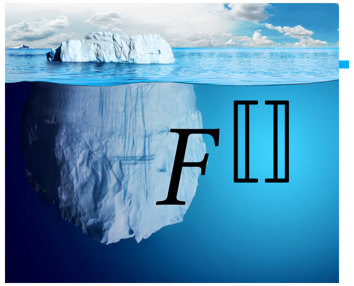
$$e_1 \, \langle e_2 \, \$ e_3 \rangle$$



$$e_1 \, \langle e_2 \quad s \quad \rangle$$

Key idea: splice environments

$\lambda^{[\![\Rightarrow]\!]}$



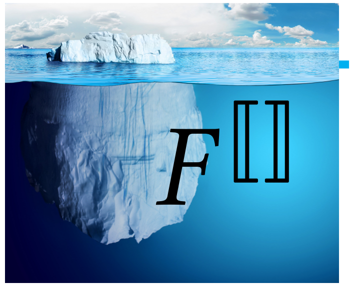
$$e_1 \, \langle e_2 \, \$ e_3 \rangle$$

$$e_1 \, \langle e_2 \quad s \quad \rangle_{\bullet \vdash^0 s : \tau = e_3}$$

Key idea: splice environments

$\lambda^{\llbracket\Rightarrow\rrbracket}$

$$e_1 \; \langle e_2 \; \$ e_3 \rangle$$

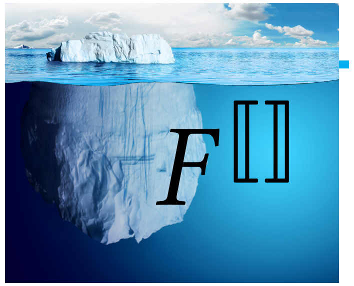


the spliced expression

$$e_1 \; \langle e_2 \quad s \; \rangle_{\bullet \vdash^0 s : \tau = e_3}$$

Key idea: splice environments

$\lambda^{[\![\Rightarrow]\!]}$



$$e_1 \; \langle e_2 \; \$ e_3 \rangle$$

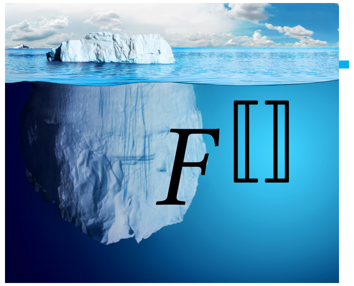
the spliced expression

$$e_1 \; \langle e_2 \quad s \; \rangle_{\bullet \vdash^0 s : \tau = e_3}$$

type of s (so the type of e3 is Code τ)

Key idea: splice environments

$\lambda^{[\![\Rightarrow]\!]}$



$$e_1 \, \langle e_2 \; \$ e_3 \rangle$$

level of the e3 (so level of s is 0 + 1 = 1)

the spliced expression

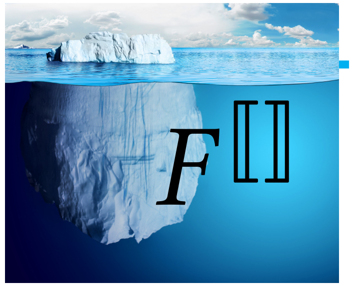
$$e_1 \, \langle e_2 \quad s \; \rangle_{\bullet \vdash^0 s : \tau = e_3}$$

type of s (so the type of e3 is Code τ)

Key idea: splice environments

$\lambda^{[\![\Rightarrow]\!]}$

$$e_1 \ \langle e_2 \ \$ e_3 \rangle$$



level of the e3 (so level of s is 0 + 1 = 1)

the spliced expression

$$e_1 \ \langle e_2 \quad s \ \rangle_{\bullet \vdash^{0} s : \tau = e_3}$$

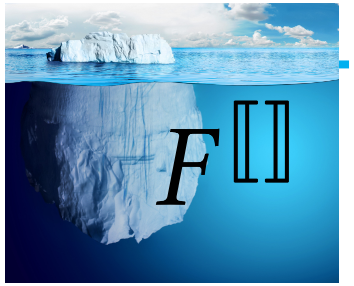
type of s (so the type of e3 is Code τ)

the type context

Key idea: splice environments

$\lambda^{[\![\Rightarrow]\!]}$

$$e_1\ \langle e_2\ \$e_3 \rangle$$



level of the quotation

level of the e3 (so level of s is 0 + 1 = 1)

the spliced expression

$$e_1\ \boxed{\langle e_2 \quad s\ \rangle}_{\bullet \vdash^{0} s:\tau = e_3}$$

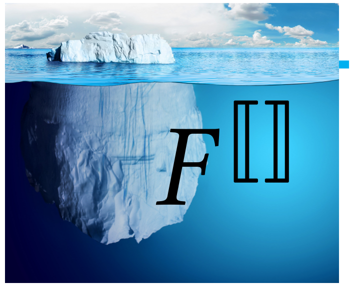
type of s (so the type of e3 is Code τ)

the type context

Key idea: splice environments

$\lambda^{[\![\Rightarrow]\!]}$

$$e_1 \langle e_2 \,\$e_3 \rangle$$



level of the quotation

a splice is bound to the innermost surrounding quotation at the same level

level of the e3 (so level of s is 0 + 1 = 1)

the spliced expression

$$e_1 \boxed{\langle e_2 \quad s \;\rangle} \bullet \vdash^0 s : \tau = e_3$$

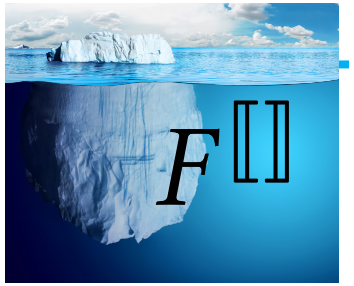
type of s (so the type of e3 is Code τ)

the type context

Key idea: splice environments

$\lambda^{[\![\Rightarrow]\!]}$

$$e_1 \; \langle e_2 \; \$e_3 \rangle$$

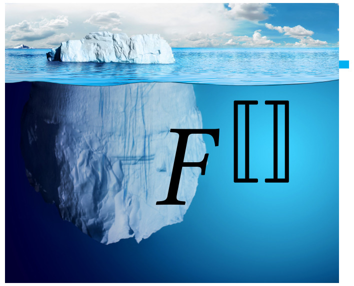


$$e_1 \; \boxed{\langle e_2 \quad s \; \rangle} \bullet \vdash^0 s{:}\tau{=}e_3$$

Key idea: splice environments

$\lambda^{[\![\Rightarrow]\!]}$

$$e_1 \, \langle e_2 \, \$ e_3 \rangle$$

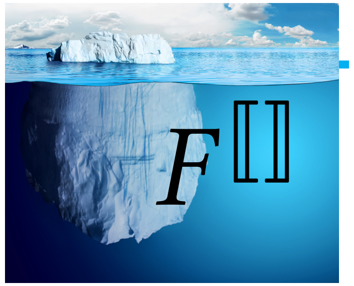


✓ Straightforward evaluation

$$e_1 \, \boxed{\langle e_2 \quad s \; \rangle} \bullet \vdash^0 s : \tau = e_3$$

Key idea: splice environments

$\lambda^{[\![\Rightarrow]\!]}$



$$e_1 \; \langle e_2 \; \$e_3 \rangle$$

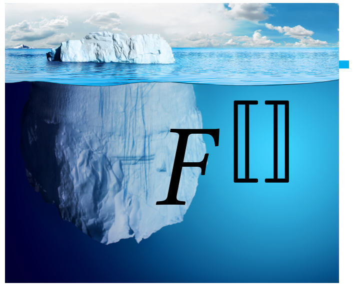
✓ **Straightforward evaluation**

$$\frac{\phi \longrightarrow \phi'}{\langle e \rangle_{\phi} \longrightarrow \langle e \rangle_{\phi'}}$$

$$e_1 \; \boxed{\langle e_2 \quad s \; \rangle} \bullet \vdash^{0} s{:}\tau = e_3$$

Key idea: splice environments

$\lambda^{[\![\Rightarrow]\!]}$



$e_1 \, \langle e_2 \, \$ e_3 \rangle$

✓ **Straightforward evaluation**

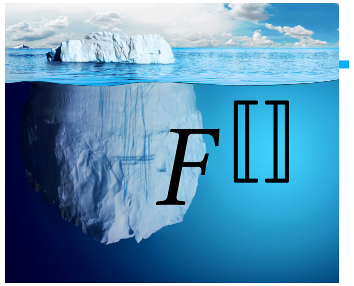
$$\frac{\phi \longrightarrow \phi'}{\langle e \rangle_{\phi} \longrightarrow \langle e \rangle_{\phi'}}$$

$e_1 \, \boxed{\langle e_2 \quad s \; \rangle} \bullet \vdash^0 s{:}\tau{=}e_3$

Key idea: splice environments

$\lambda^{[\![\Rightarrow]\!]}$

$$e_1 \, \langle e_2 \, \$ e_3 \rangle$$



✓ **Straightforward evaluation**

$$\frac{\phi \longrightarrow \phi'}{\langle e \rangle_{\phi} \longrightarrow \langle e \rangle_{\phi'}}$$

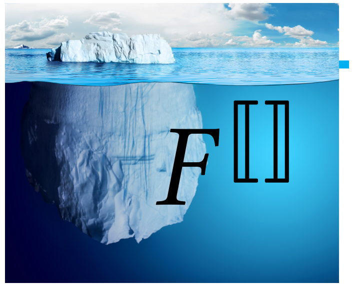
$$e_1 \, \boxed{\langle e_2 \quad s \; \rangle}_{\bullet \vdash^0 s : \tau = e_3}$$

✓ **Opaque quotations**

Key idea: splice environments

$\lambda^{[\![\Rightarrow]\!]}$

$$e_1\ (e_2\ \$e_3)$$

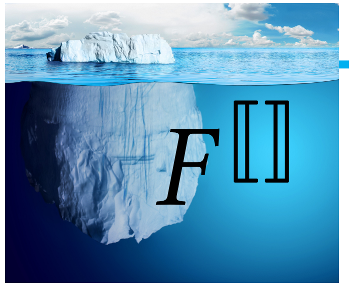


$$e_1\ \boxed{\langle e_2 \quad s\ \rangle}\ \bullet \vdash^0 s:\tau = e_3$$

Key idea: splice environments

$\lambda^{[\![\Rightarrow]\!]}$

$$e_1 \; (e_2 \; \$ e_3 \;)$$

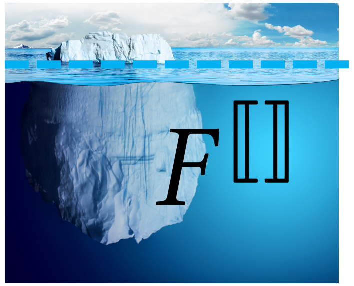


$$e_1 \; \boxed{(e_2 \quad s \quad)}$$

Negative levels and top-level splice definitions

$\lambda^{[\![\Rightarrow]\!]}$

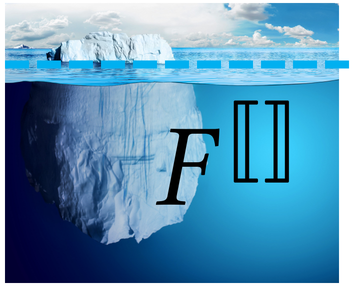
$$e_1\ (e_2\ \$e_3\ )$$



$$e_1\ \boxed{(\,e_2\quad s\ \,)}$$

Negative levels and top-level splice definitions

$\lambda^{[\![\Rightarrow]\!]}$



$$e_1\ (e_2\ \$e_3\ )$$

compile-time evaluation

$$\textbf{spdef}\ \bullet \vdash^{-1} s : \tau = e_3\ ;$$

$$e_1\ \boxed{(e_2\ \ s\ )}$$

Type-directed elaboration

Type-directed elaboration

$$\boxed{\Gamma \vdash^{n} \lambda^{[\![\Rightarrow]\!]} \rightsquigarrow F^{[\![]\!]} \mid \phi}$$

Type-directed elaboration

$$\boxed{\Gamma \vdash^{n} \lambda^{[\![\Rightarrow]\!]} \rightsquigarrow F^{[\![]\!]} \mid \phi}$$

$$\frac{\Gamma \vdash^{n-1} e : \mathsf{Code}\,\tau \rightsquigarrow e' \mid \phi \qquad \Gamma \vdash \tau \rightsquigarrow \tau' \qquad \text{fresh } s}{\Gamma \vdash^{n} \$e : \tau \rightsquigarrow s \mid \phi, (\bullet \vdash^{n-1} s : \tau' = e')}$$

$$\frac{\Gamma \vdash^{n+1} e : \tau \rightsquigarrow e' \mid \phi}{\Gamma \vdash^{n} \langle e \rangle : \mathsf{Code}\,\tau \rightsquigarrow \langle e' \rangle_{\phi.n} \mid \lfloor \phi \rfloor^{n}}$$

Type soundness

$F^{[\![\,]\!]}$

(1) *If* $\bullet \vdash^n e : \tau$, *then either* e *is a value, or* $e \longrightarrow e'$ *for some* e'.

(2) *If* $\Delta \vdash^n e : \tau$, *and* $e \longrightarrow e'$, *then* $\Delta \vdash^n e' : \tau$.

Type soundness

$\lambda^{[\![\Rightarrow]\!]}$

If $\Gamma \vdash^{n} e : \tau \rightsquigarrow e \mid \phi$, *and* $\Gamma \rightsquigarrow \Delta$, *and* $\Gamma \vdash \tau \rightsquigarrow \tau$,
then $\Delta, \phi^{\Gamma} \vdash^{n} e : \tau$.

$F^{[\![\,]\!]}$

(1) *If* $\bullet \vdash^{n} e : \tau$, *then either* e *is a value, or* $e \longrightarrow e'$ *for some* e'.

(2) *If* $\Delta \vdash^{n} e : \tau$, *and* $e \longrightarrow e'$, *then* $\Delta \vdash^{n} e' : \tau$.

Duality

$$\lambda^{[\![\Rightarrow]\!]}$$

$$\begin{array}{rcl} \langle \$e \rangle & =_{ax} & e \\ \$\langle e \rangle & =_{ax} & e \end{array}$$

Duality

$\lambda^{[\![\Rightarrow]\!]}$

$$
\begin{array}{lcl}
\hline
\langle \$e \rangle & =_{ax} & e \\
\$\langle e \rangle & =_{ax} & e \\
\hline
\end{array}
$$

$F^{[\![\,]\!]}$

$$
\begin{array}{llll}
\hline
\langle s \rangle_{\bullet \Vdash^n s:\tau=e} & =_{ax} & e & \\
\langle e_1 \rangle_{\phi_1, \Delta \Vdash^n s:\tau=\langle e \rangle_\phi, \phi_2} & =_{ax} & \langle e_1[s \mapsto e] \rangle_{\phi_1, \phi', \phi_2} & \textit{where } \phi \mathbin{+\!\!+} \Delta \rightsquigarrow \phi' \\
\textbf{spdef } \Delta \Vdash^n s : \tau = \langle e \rangle_\phi ; \rho gm & =_{ax} & \textbf{spdef } \phi' ; \rho gm[s \mapsto e] & \textit{where } \phi \mathbin{+\!\!+} \Delta \rightsquigarrow \phi' \\
\hline
\end{array}
$$

Duality

$$\lambda^{[\![\Rightarrow]\!]}$$

$$\begin{array}{lcl} \langle \$e \rangle & =_{ax} & e \\ \$\langle e \rangle & =_{ax} & e \end{array}$$

$$F^{[\![]\!]}$$

$$\begin{array}{llll} \langle s \rangle_{\bullet \Vdash^n s:\tau=e} & =_{ax} & e & \\ \langle e_1 \rangle_{\phi_1, \Delta \Vdash^n s:\tau=\langle e \rangle_\phi, \phi_2} & =_{ax} & \langle e_1[s \mapsto e] \rangle_{\phi_1, \phi', \phi_2} & \textit{where } \phi +\!\!+ \Delta \rightsquigarrow \phi' \\ \textbf{spdef } \Delta \Vdash^n s : \tau = \langle e \rangle_\phi ; \rho gm & =_{ax} & \textbf{spdef } \phi' ; \rho gm[s \mapsto e] & \textit{where } \phi +\!\!+ \Delta \rightsquigarrow \phi' \end{array}$$

$$\Theta \vdash \rho gm_1 \simeq_{ctx} \rho gm_2 : \tau$$

Duality

$$\lambda^{[\![\Rightarrow]\!]}$$

$$\begin{aligned} \langle \$e \rangle &=_{ax} e \\ \$\langle e \rangle &=_{ax} e \end{aligned}$$

$$F^{[\![]\!]}$$

$$\begin{array}{lll} \langle s \rangle_{\bullet \Vdash^n s:\tau=e} & =_{ax} & e \\ \langle e_1 \rangle_{\phi_1, \Delta \Vdash^n s:\tau=\langle e \rangle_\phi, \phi_2} & =_{ax} & \langle e_1[s \mapsto e] \rangle_{\phi_1, \phi', \phi_2} \quad \textit{where } \phi +\!\!+ \Delta \rightsquigarrow \phi' \\ \textbf{spdef } \Delta \Vdash^n s : \tau = \langle e \rangle_\phi ; \rho gm & =_{ax} & \textbf{spdef } \phi' ; \rho gm[s \mapsto e] \quad \textit{where } \phi +\!\!+ \Delta \rightsquigarrow \phi' \end{array}$$

$$\Theta \vdash \rho gm_1 \simeq_{ctx} \rho gm_2 : \tau$$

Duality

$\lambda^{[\![\Rightarrow]\!]}$

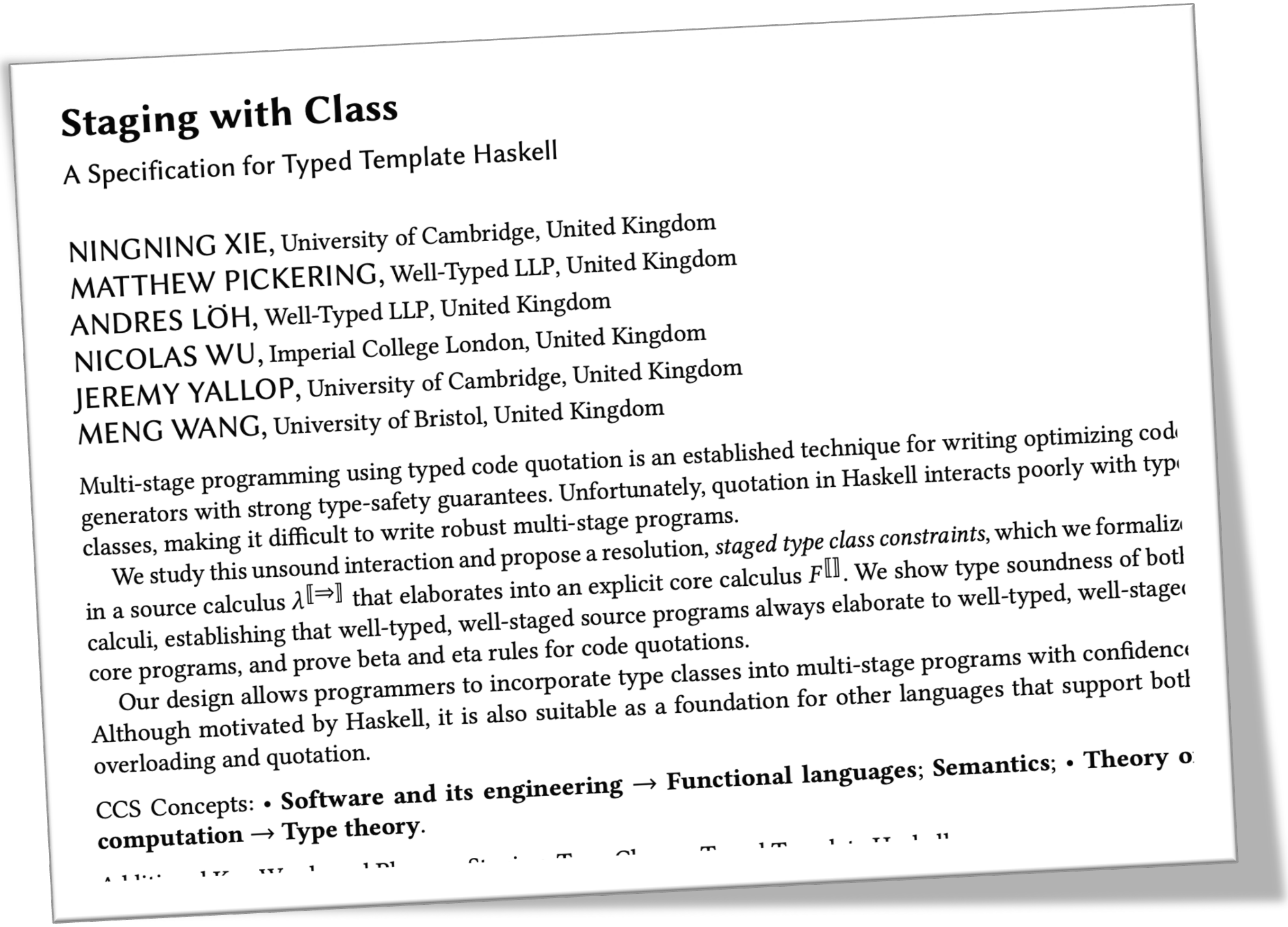
$$\begin{aligned} \langle \$ e \rangle &=_{ax} e \\ \$\langle e \rangle &=_{ax} e \end{aligned}$$

$F^{[\![\,]\!]}$

$$\begin{array}{lll l} \langle s \rangle_{\bullet \Vdash^n s:\tau=e} & =_{ax} & e & \\ \langle e_1 \rangle_{\phi_1, \Delta \Vdash^n s:\tau=\langle e \rangle_\phi, \phi_2} & =_{ax} & \langle e_1[s \mapsto e] \rangle_{\phi_1, \phi', \phi_2} & \textit{where } \phi +\!\!+ \Delta \rightsquigarrow \phi' \\ \textbf{spdef } \Delta \Vdash^n s : \tau = \langle e \rangle_\phi ; \rho gm & =_{ax} & \textbf{spdef } \phi' ; \rho gm[s \mapsto e] & \textit{where } \phi +\!\!+ \Delta \rightsquigarrow \phi' \end{array}$$

$$\Theta \vdash \rho gm_1 \simeq_{ctx} \rho gm_2 : \tau$$

More in paper



Staging with Class

A Specification for Typed Template Haskell

NINGNING XIE, University of Cambridge, United Kingdom
MATTHEW PICKERING, Well-Typed LLP, United Kingdom
ANDRES LÖH, Well-Typed LLP, United Kingdom
NICOLAS WU, Imperial College London, United Kingdom
JEREMY YALLOP, University of Cambridge, United Kingdom
MENG WANG, University of Bristol, United Kingdom

Multi-stage programming using typed code quotation is an established technique for writing optimizing cod generators with strong type-safety guarantees. Unfortunately, quotation in Haskell interacts poorly with typ classes, making it difficult to write robust multi-stage programs.

We study this unsound interaction and propose a resolution, *staged type class constraints*, which we formaliz in a source calculus $\lambda^{[\![\Rightarrow]\!]}$ that elaborates into an explicit core calculus $F^{[\![]\!]}$. We show type soundness of botl calculi, establishing that well-typed, well-staged source programs always elaborate to well-typed, well-stage core programs, and prove beta and eta rules for code quotations.

Our design allows programmers to incorporate type classes into multi-stage programs with confidenc Although motivated by Haskell, it is also suitable as a foundation for other languages that support botl overloading and quotation.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Semantics**; • **Theory o computation** → **Type theory**.

- Full typing rules
- Metatheory development
- Comparison between GHC and $\lambda^{[\![\Rightarrow]\!]}$
- Challenges of integration into GHC

This talk

unsound

inspire $\longrightarrow$ $\lambda^{[\![\Rightarrow]\!]}$ type-directed $\longrightarrow$ $F^{[\![\,]\!]}$

$\lambda^{[\![\Rightarrow]\!]}$:

- Type Classes
- Quotations/Splicing
- Staged type class constraints

$F^{[\![\,]\!]}$:

- Quotations
- Splice environments

✓ A solid theoretical foundation for integrating type classes into multi-stage programs

✓ easy to implement and stay close to existing implementations

Thank you!

I am on the academic job market!

https://xnning.github.io/