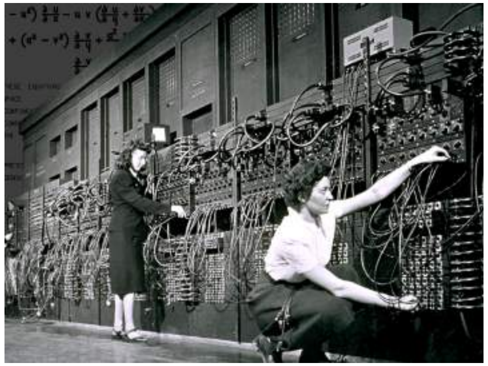
Staging with Class

A Specification of Typed Template Haskell

Ningning Xie

UNIVERSITY OF CAMBRIDGE

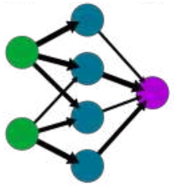
YOW! Lambda Jam
May 18 2022



ENIAC

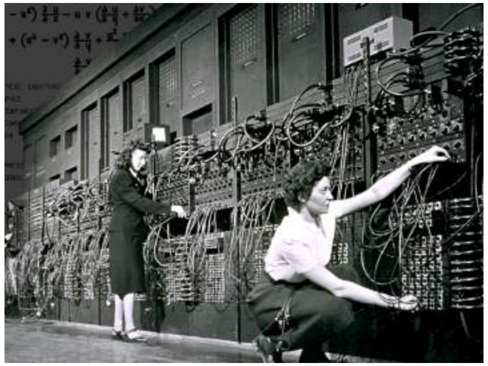
1945 **today**

web development



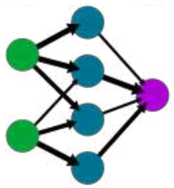
machine learning

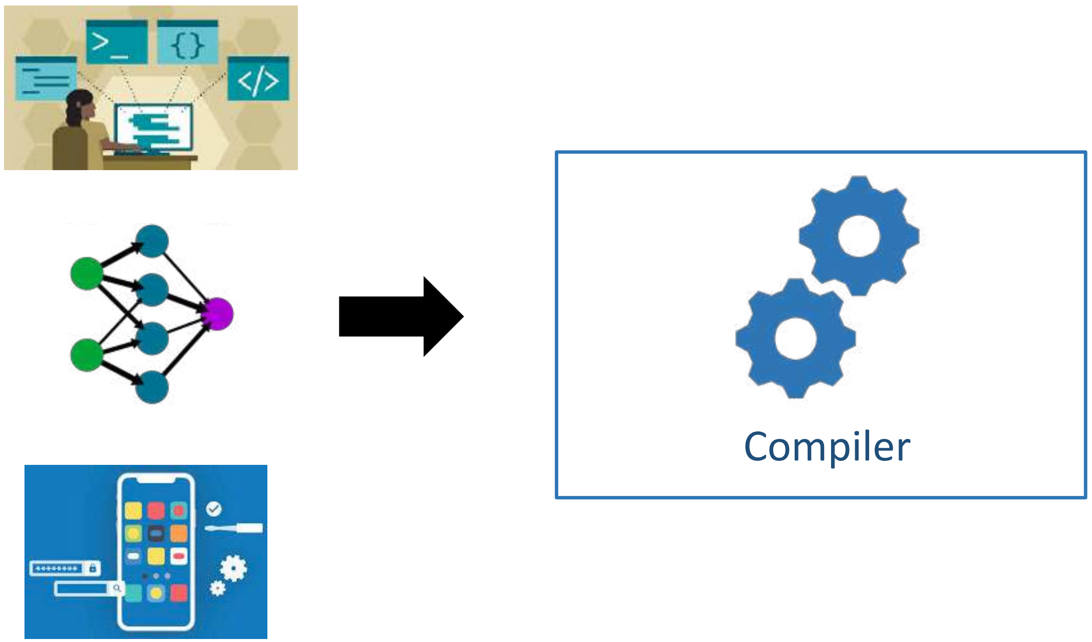
app development

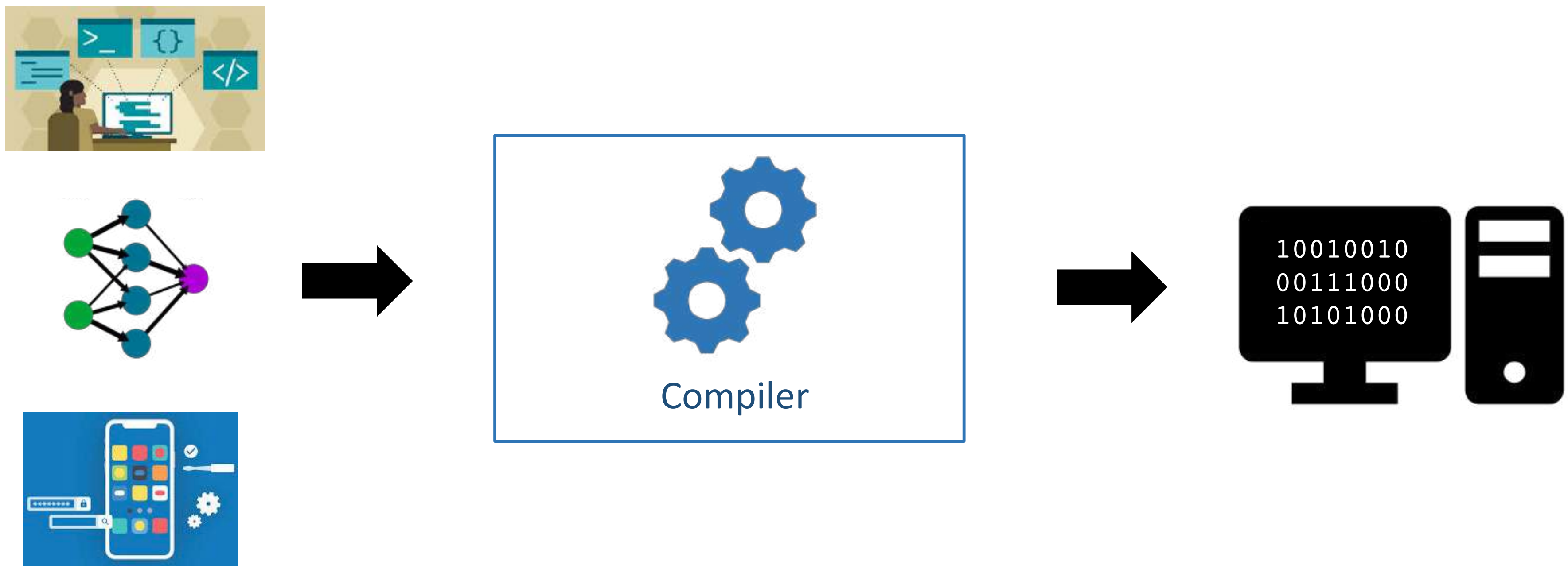


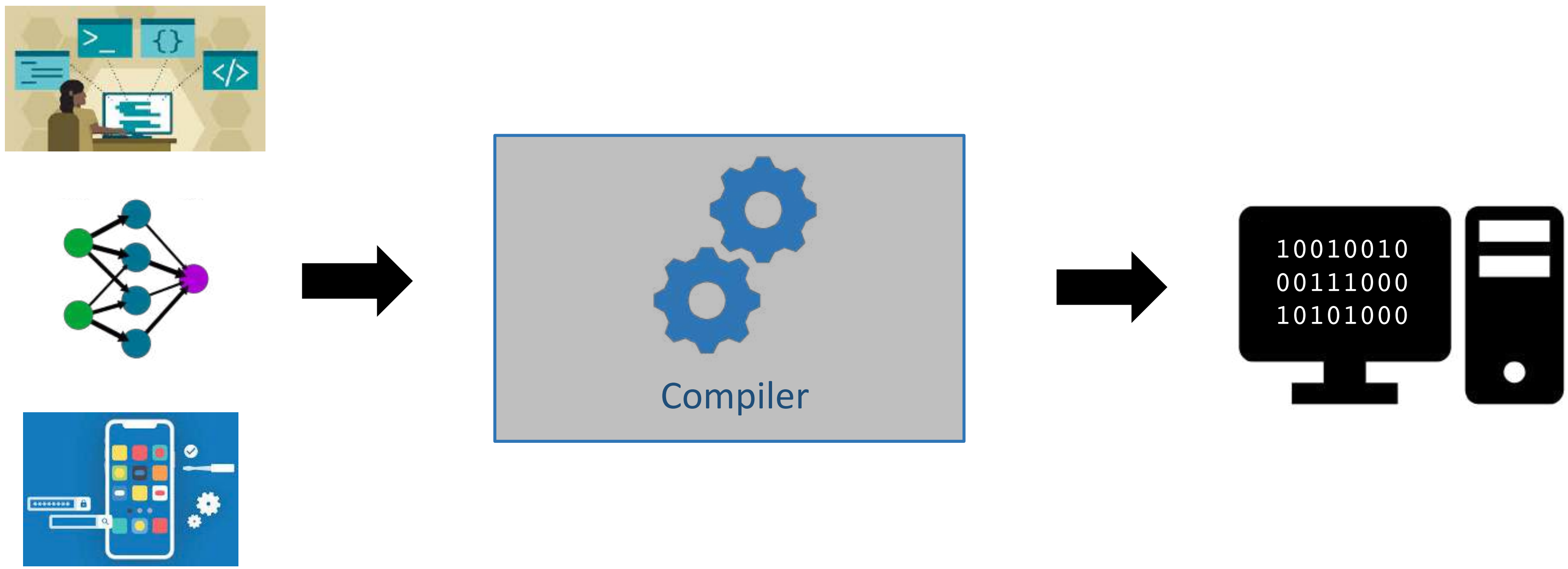
ENIAC

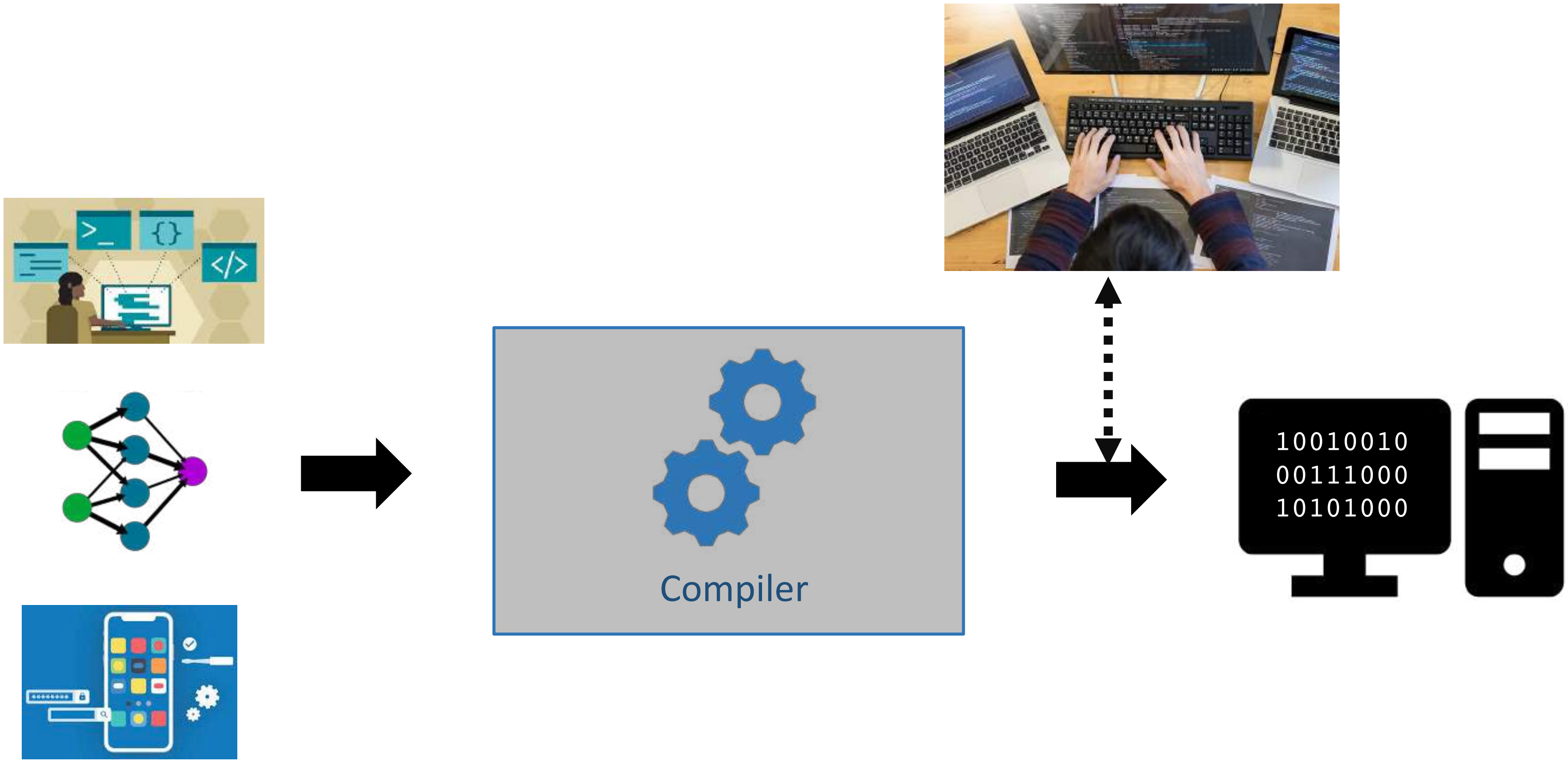
1945 **today**

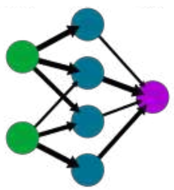


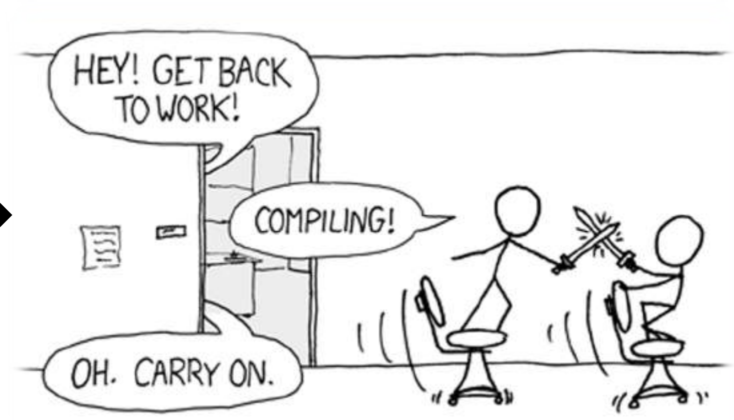

Compiler

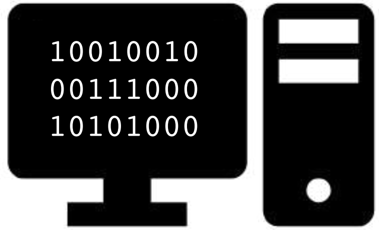

Compiler
10010010
00111000
10101000

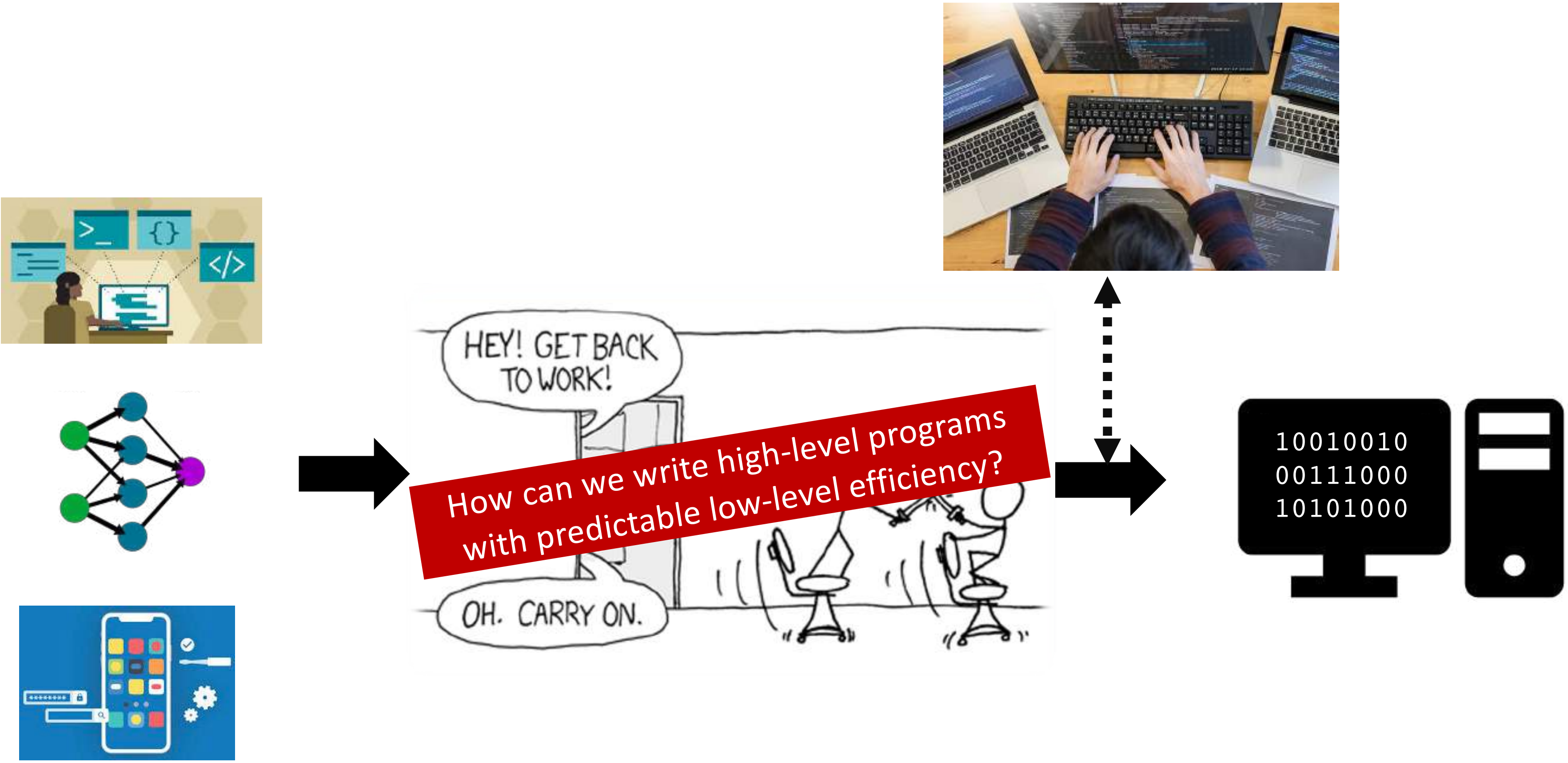

Compiler
10010010
00111000
10101000


Compiler
10010010
00111000
10101000

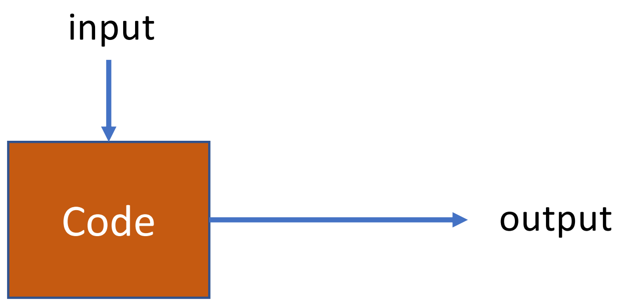



HEY! GET BACK TO WORK!
COMPILING!
OH. CARRY ON.

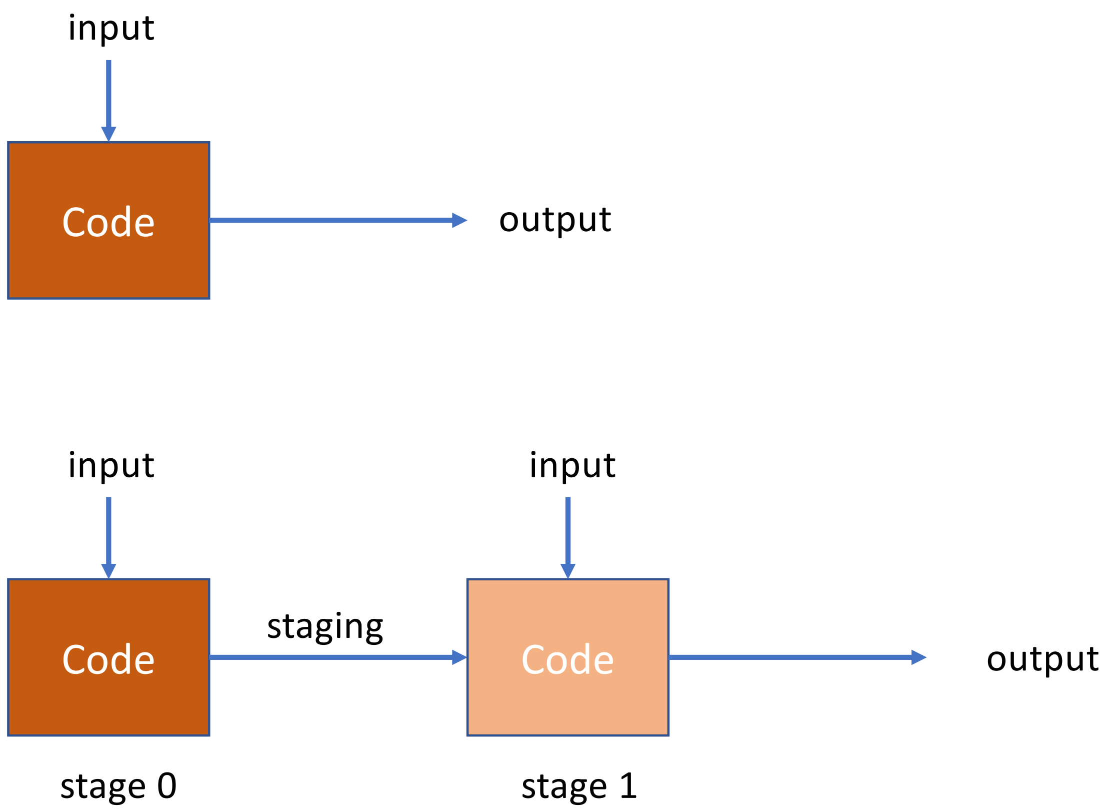

10010010
00111000
10101000


HEY! GET BACK TO WORK!
OH. CARRY ON.
How can we write high-level programs
with predictable low-level efficiency?
10010010
00111000
10101000

Multi-stage programming



Multi-stage programming



TWO-LEVEL SEMANTICS AND CODE GENERATION * [1988]

Flemming NIELSON
Department of Computer Science, Technical University of Denmark, DK-2800 Lyngby, Denmark

Hanne Riis NIELSON
Institute of Electronic Systems, Aalborg University Centre, DK-9000 Aalborg, Denmark

TWO-LEVEL SEMANTICS AND CODE GENERATION* [1988]

Flemming NIELSON
Department of Computer Science, Technical University of Denmark, DK-2800 Lyngby, Denmark

Hanne Riis NIELSON
Institute of Electronic Systems, Aalborg University Centre, DK-9000 Aalborg, Denmark

TWO-LEVEL SEMANTICS AND CODE GENERATION* [1988]

Flemming NIELSON
Department of Computer Science, Technical University of Denmark, DK-2800 Lyngby, Denmark

Hanne Riis NIELSON
Institute of Electronic Systems, Aalborg University Centre, DK-9000 Aalborg, Denmark

— MetaML [Taha and Sheard 1997]

TWO-LEVEL SEMANTICS AND CODE GENERATION* [1988]

Flemming NIELSON
Department of Computer Science, Technical University of Denmark, DK-2800 Lyngby, Denmark

Hanne Riis NIELSON
Institute of Electronic Systems, Aalborg University Centre, DK-9000 Aalborg, Denmark

MetaML [Taha and Sheard 1997]

Template Haskell [Sheard and Peyton Jones 2002]

TWO-LEVEL SEMANTICS AND CODE GENERATION* [1988]

Flemming NIELSON
Department of Computer Science, Technical University of Denmark, DK-2800 Lyngby, Denmark

Hanne Riis NIELSON
Institute of Electronic Systems, Aalborg University Centre, DK-9000 Aalborg, Denmark

MetaML [Taha and Sheard 1997]

Template Haskell [Sheard and Peyton Jones 2002]

MetaOCaml [Calcagno et al. 2003]

TWO-LEVEL SEMANTICS AND CODE GENERATION* [1988]

Flemming NIELSON
Department of Computer Science, Technical University of Denmark, DK-2800 Lyngby, Denmark

Hanne Riis NIELSON
Institute of Electronic Systems, Aalborg University Centre, DK-9000 Aalborg, Denmark

MetaML [Taha and Sheard 1997]

Template Haskell [Sheard and Peyton Jones 2002]

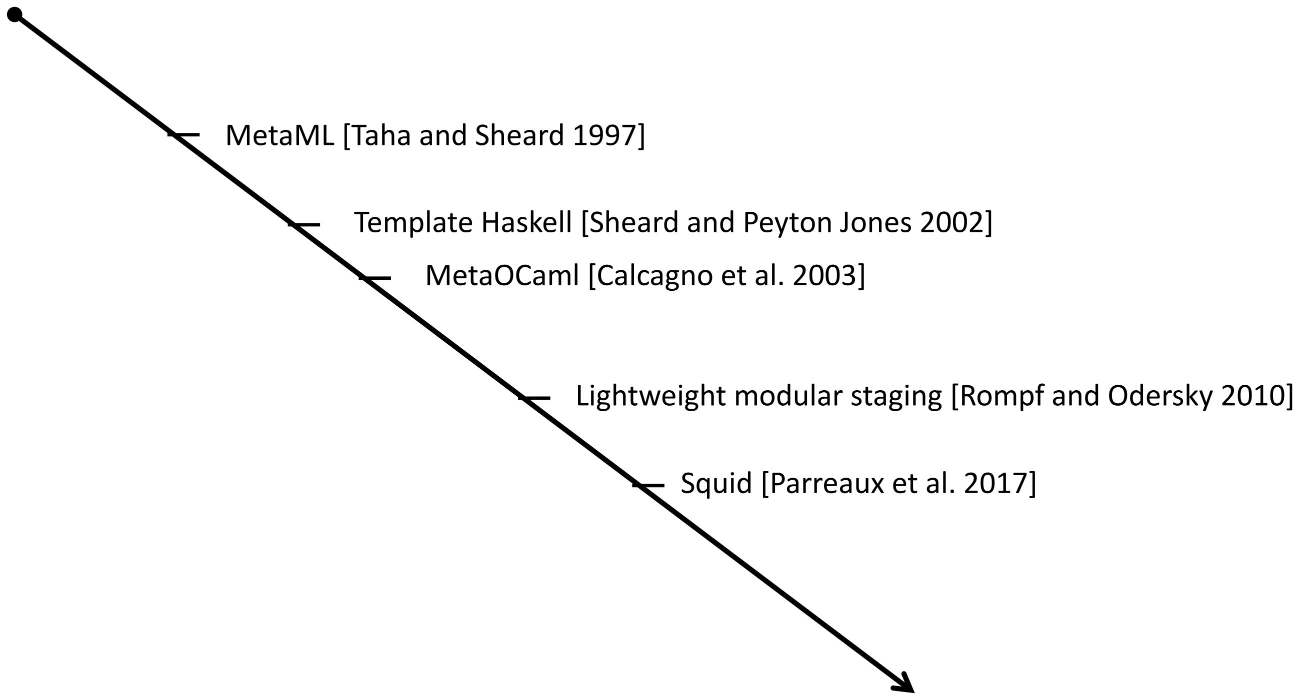
MetaOCaml [Calcagno et al. 2003]

Lightweight modular staging [Rompf and Odersky 2010]

TWO-LEVEL SEMANTICS AND CODE GENERATION * [1988]

Flemming NIELSON
Department of Computer Science, Technical University of Denmark, DK-2800 Lyngby, Denmark

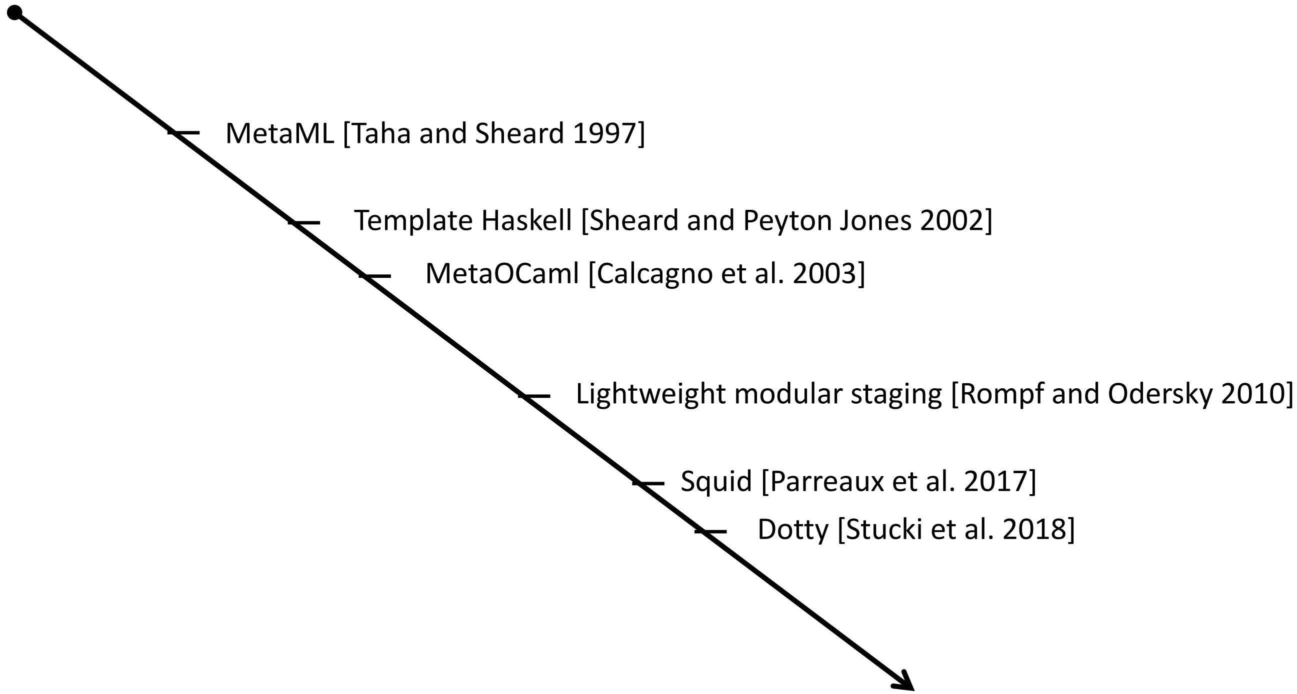
Hanne Riis NIELSON
Institute of Electronic Systems, Aalborg University Centre, DK-9000 Aalborg, Denmark



TWO-LEVEL SEMANTICS AND CODE GENERATION * [1988]

Flemming NIELSON
Department of Computer Science, Technical University of Denmark, DK-2800 Lyngby, Denmark

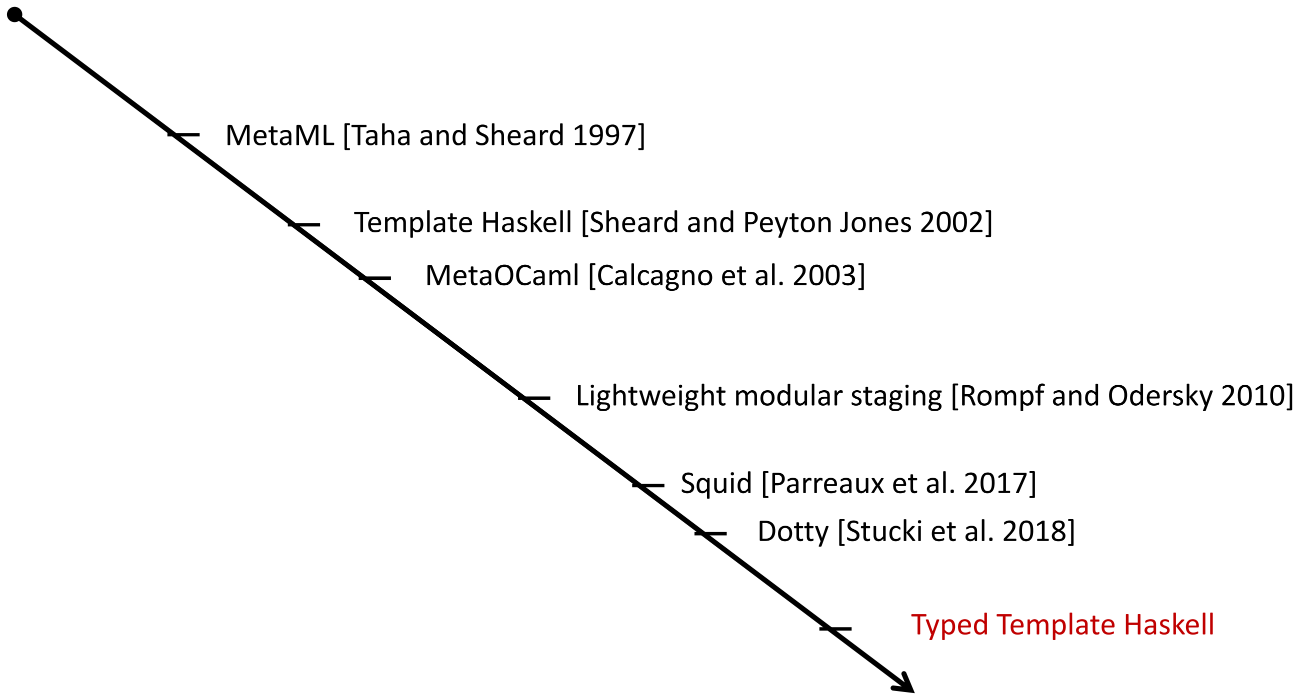
Hanne Riis NIELSON
Institute of Electronic Systems, Aalborg University Centre, DK-9000 Aalborg, Denmark



TWO-LEVEL SEMANTICS AND CODE GENERATION * [1988]

Flemming NIELSON
Department of Computer Science, Technical University of Denmark, DK-2800 Lyngby, Denmark

Hanne Riis NIELSON
Institute of Electronic Systems, Aalborg University Centre, DK-9000 Aalborg, Denmark



TWO-LEVEL SEMANTICS AND CODE GENERATION* [1988]

Flemming NIELSON
Department of Computer Science, Technical University of Denmark, DK-2800 Lyngby, Denmark

Hanne Riis NIELSON
Institute of Electronic Systems, Aalborg University Centre, DK-9000 Aalborg, Denmark

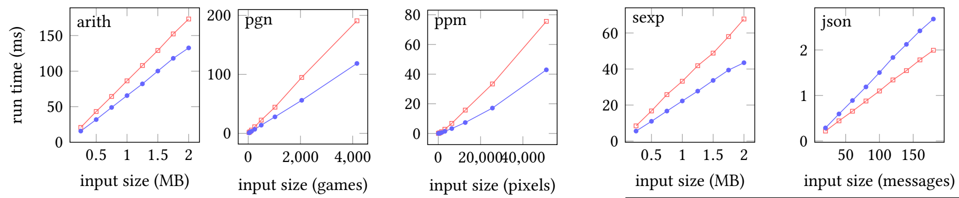


Figure 9. Typed grammar expressions guarantee linear-time parsing

[Krishnaswami and Yallop 2019]

MetaML [Taha and Sheard 1997]

Template Haskell [Sheard and Peyton Jones 2002]

MetaOCaml [Calcagno et al. 2003]

Lightweight modular staging [Rompf and Odersky 2010]

Squid [Parreaux et al. 2017]

Dotty [Stucki et al. 2018]

Typed Template Haskell

TWO-LEVEL SEMANTICS AND CODE GENERATION* [1988]

Flemming NIELSON
Department of Computer Science, Technical University of Denmark, DK-2800 Lyngby, Denmark

Hanne Riis NIELSON
Institute of Electronic Systems, Aalborg University Centre, DK-9000 Aalborg, Denmark

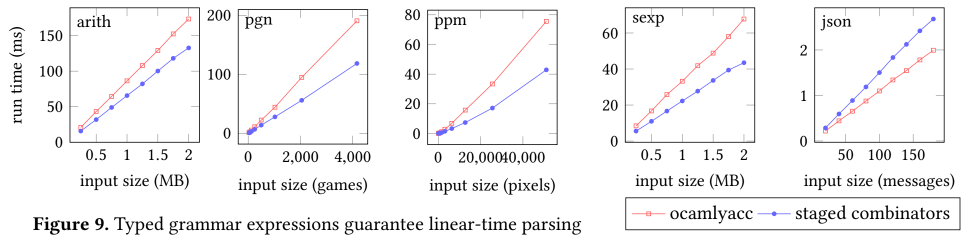
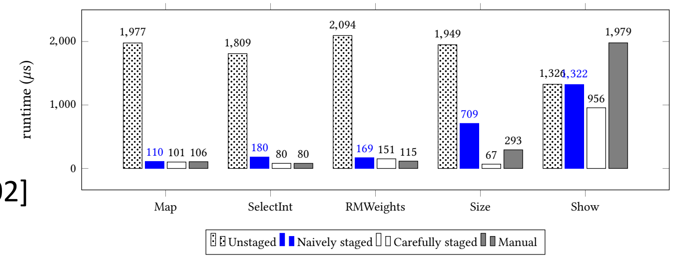


Figure 9. Typed grammar expressions guarantee linear-time parsing

[Krishnaswami and Yallop 2019]



[Yallop 2017]

MetaML [Taha and Sheard 1997]

Template Haskell [Sheard and Peyton Jones 2002]

MetaOCaml [Calcagno et al. 2003]

Lightweight modular staging [Rompf and Odersky 2010]

Squid [Parreaux et al. 2017]

Dotty [Stucki et al. 2018]

Typed Template Haskell

TWO-LEVEL SEMANTICS AND CODE GENERATION* [1988]

Flemming NIELSON
Department of Computer Science, Technical University of Denmark, DK-2800 Lyngby, Denmark

Hanne Riis NIELSON
Institute of Electronic Systems, Aalborg University Centre, DK-9000 Aalborg, Denmark

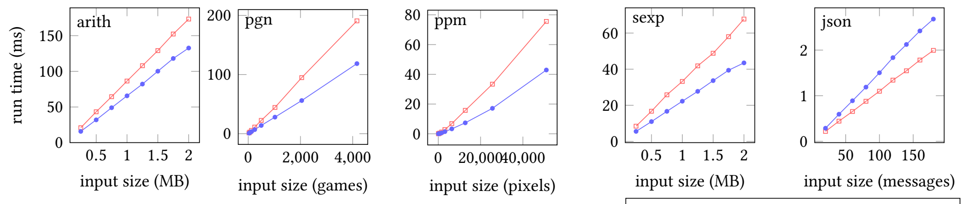


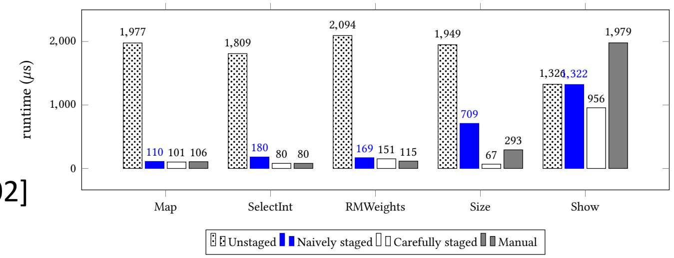
Figure 9. Typed grammar expressions guarantee linear-time parsing

[Krishnaswami and Yallop 2019]

MetaML [Taha and Sheard 1997]

Template Haskell [Sheard and Peyton Jones 2002]

MetaOCaml [Calcagno et al. 2003]



[Yallop 2017]

Lightweight modular staging [Rompf and Odersky 2010]

Squid [Parreaux et al. 2017]

Dotty [Stucki et al. 2018]

Typed Template Haskell

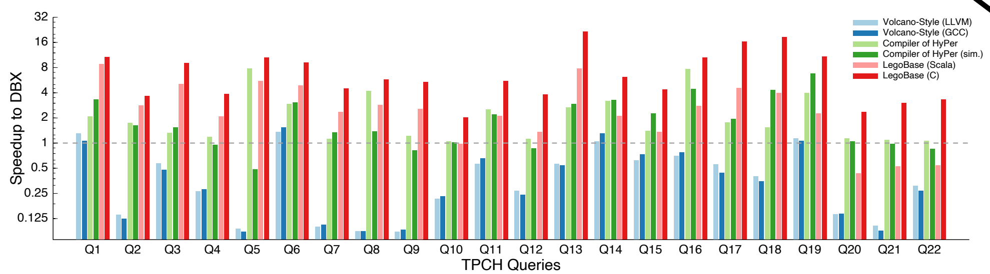


Figure 11: Performance comparison of LegoBase (C and Scala programs) with the code generated by the query compiler of [15].

[Klonatos et al. 2014]

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

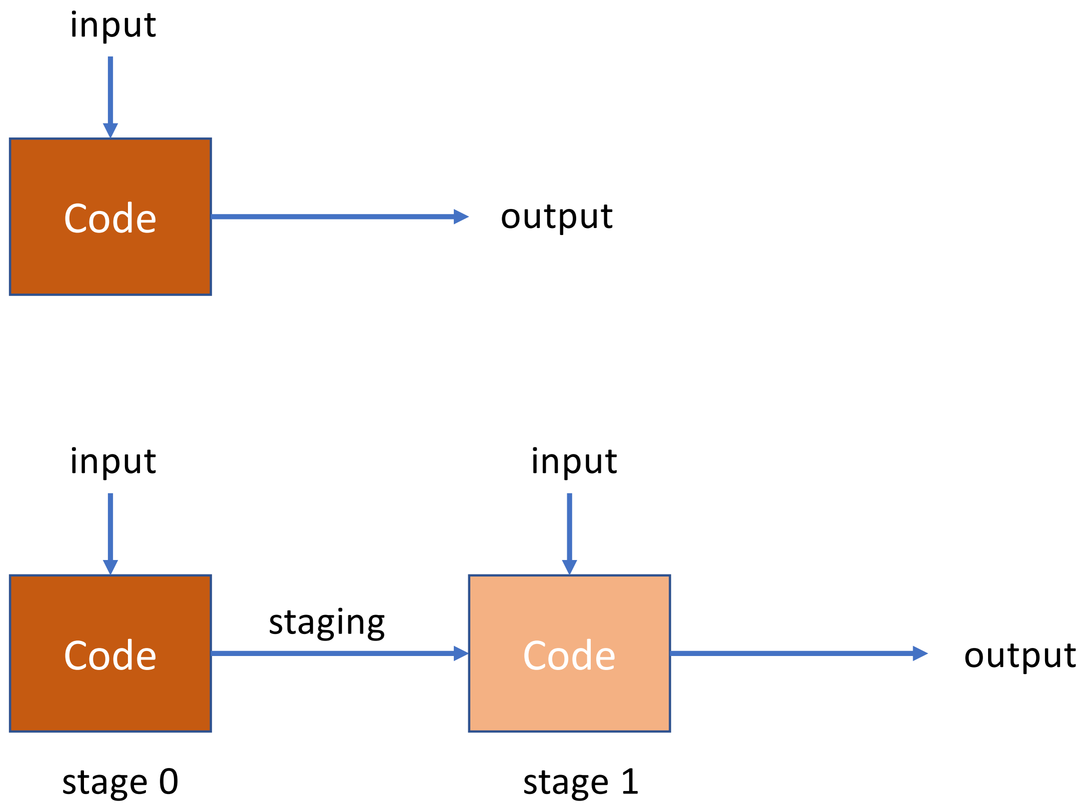
`e :: Int` $\Longrightarrow$ `<e> :: Code Int`

Splice

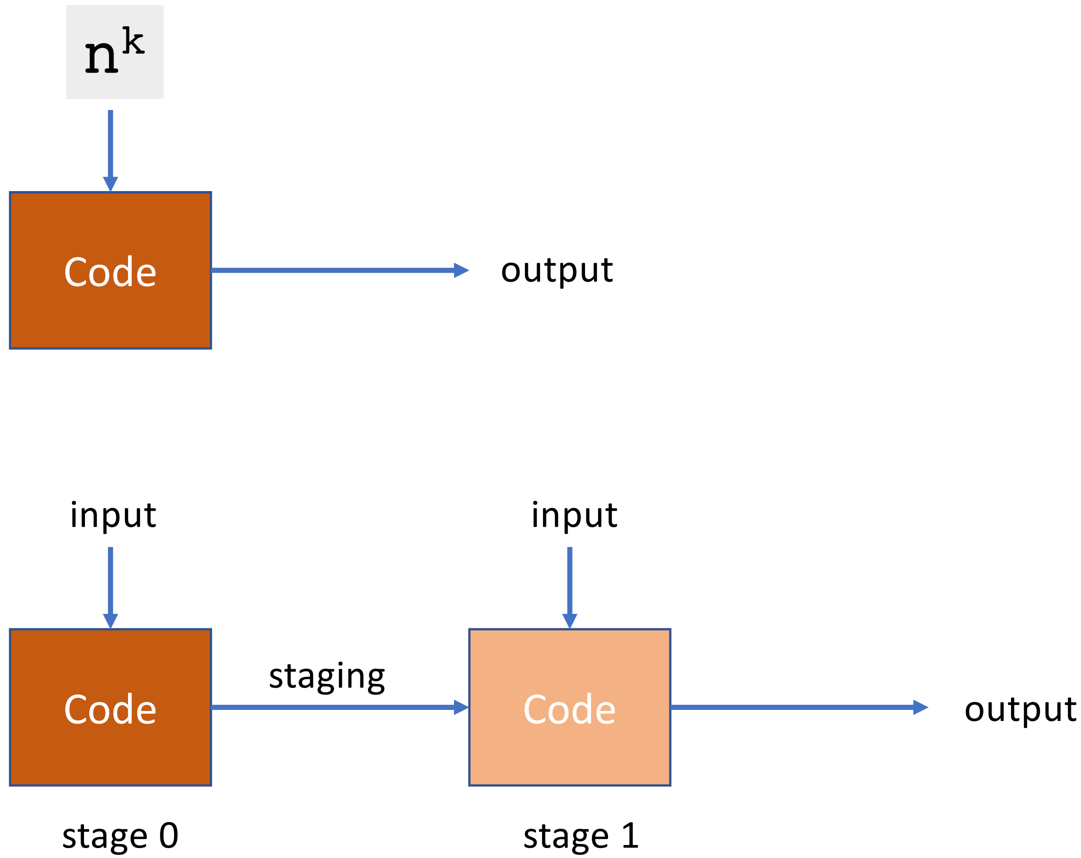
extracts the expression from its representation

`e :: Code Int` $\Longrightarrow$ `$e :: Int`

Multi-stage programming: example

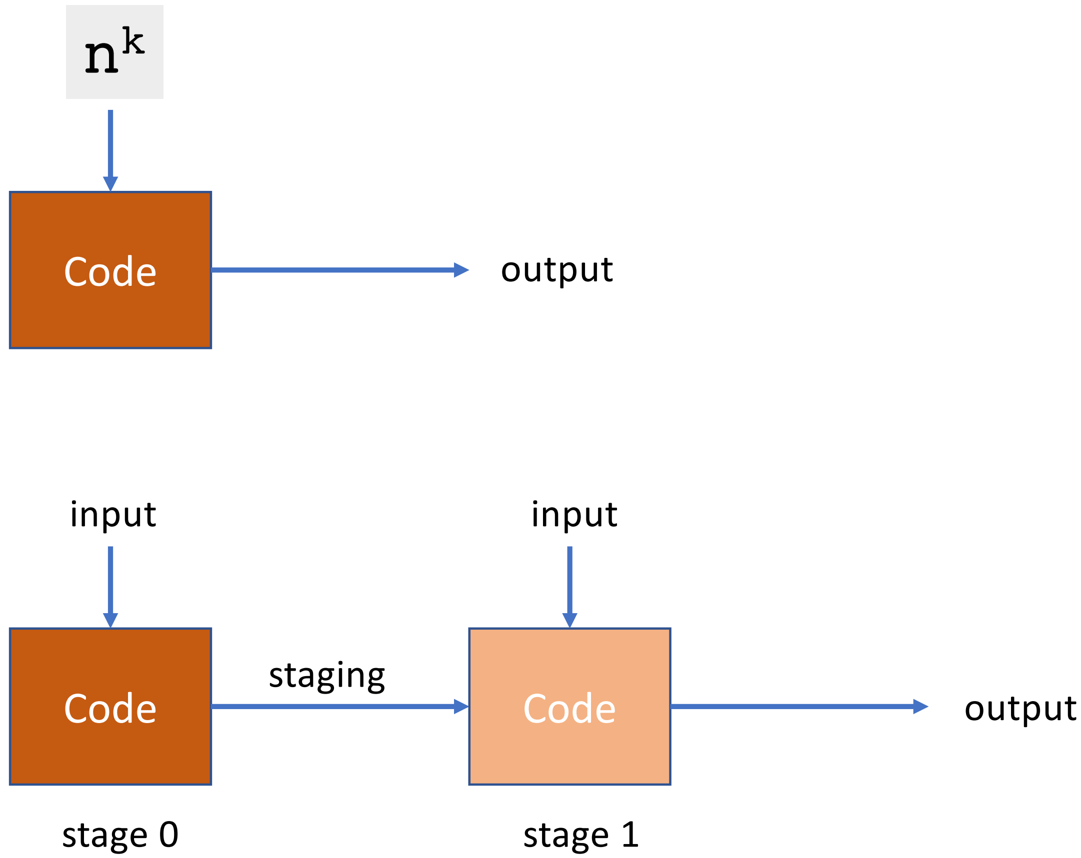


Multi-stage programming: example



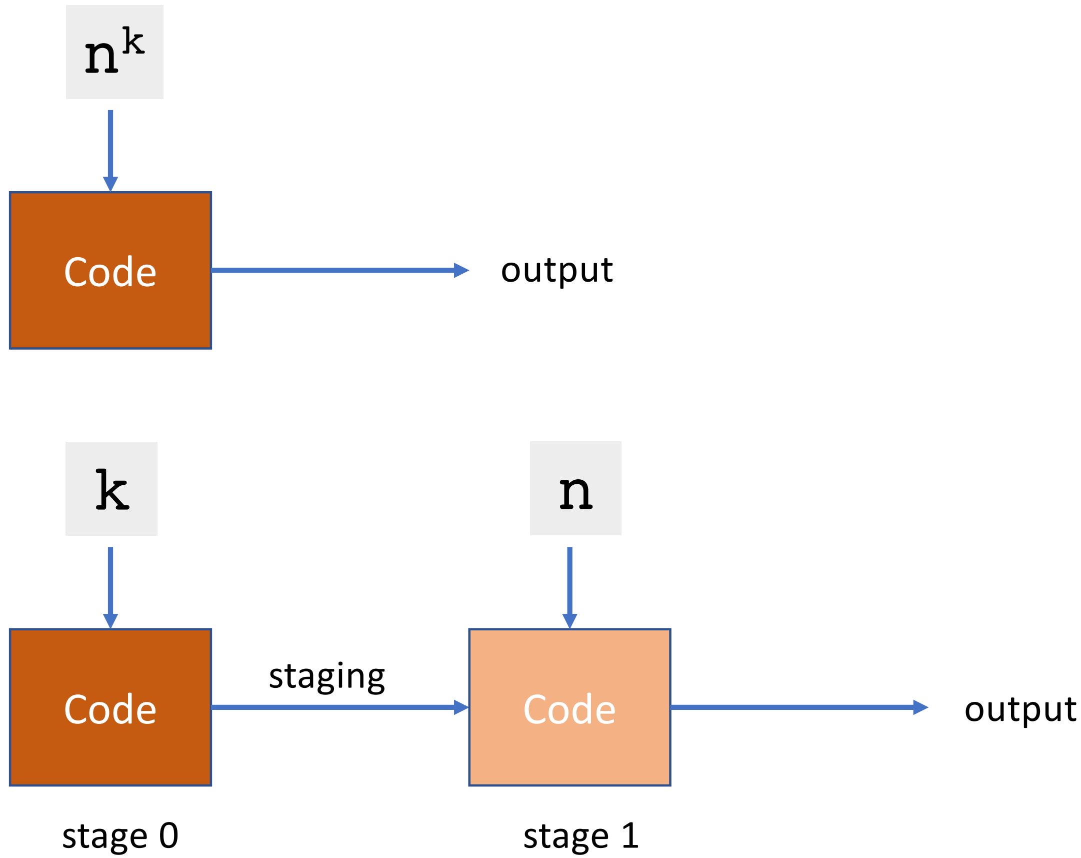
Multi-stage programming: example

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

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



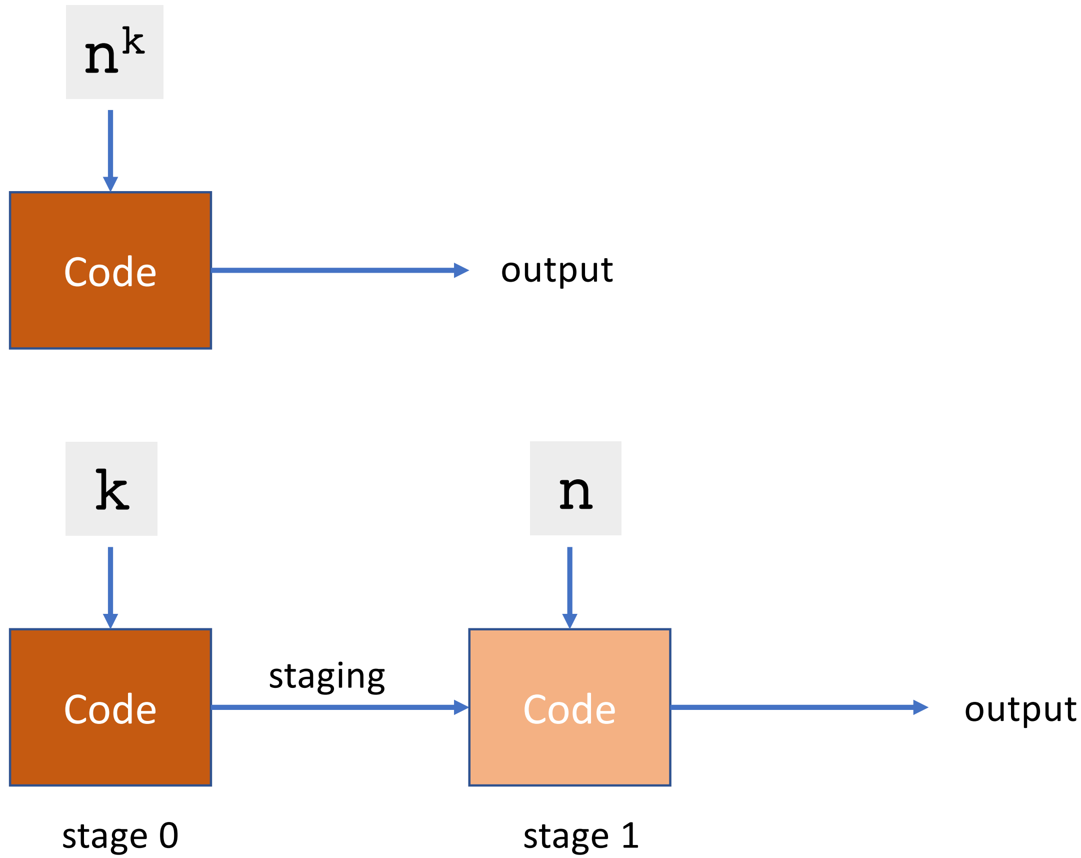
Multi-stage programming: example

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

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



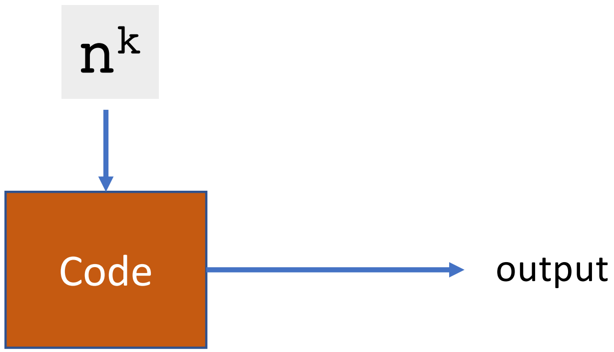
Multi-stage programming: example

```
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
```

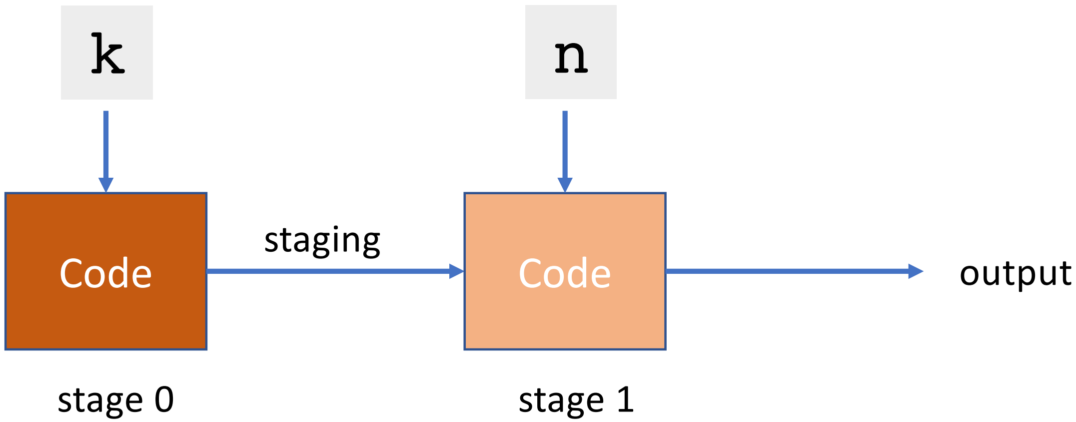


Multi-stage programming: example

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

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

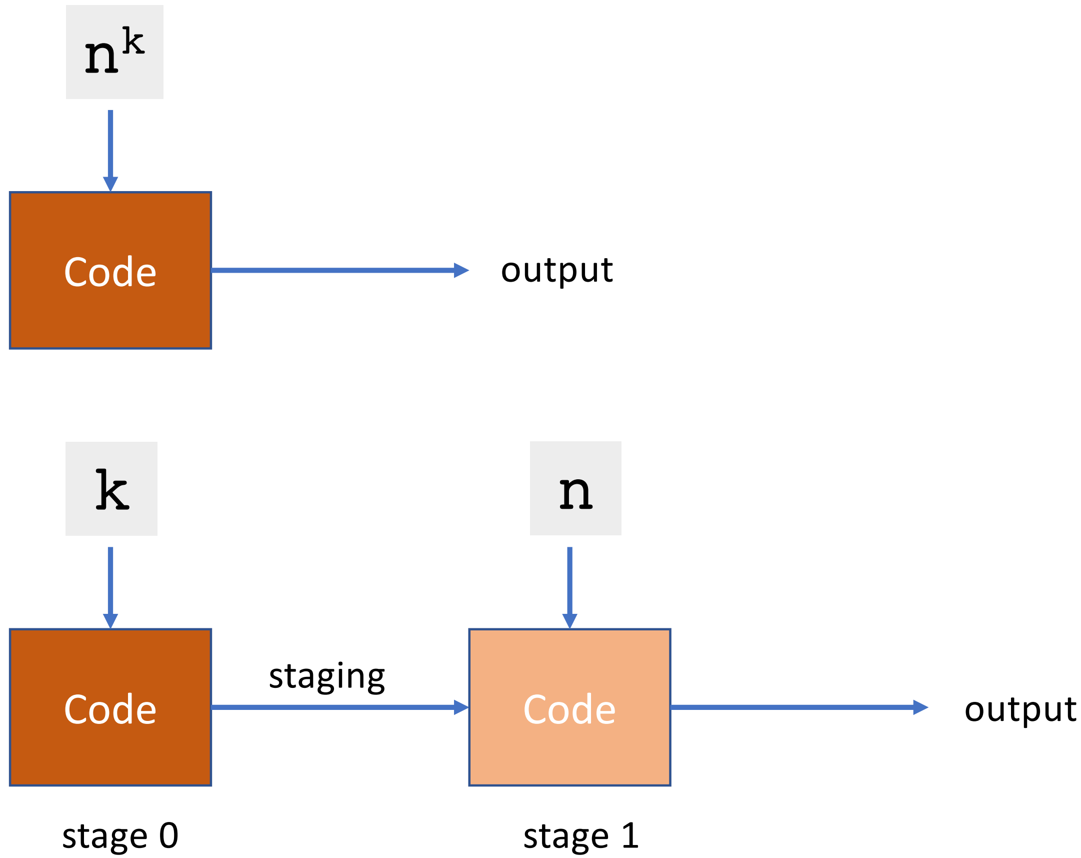



```
qpower :: Int -> Code Int -> Code Int
```



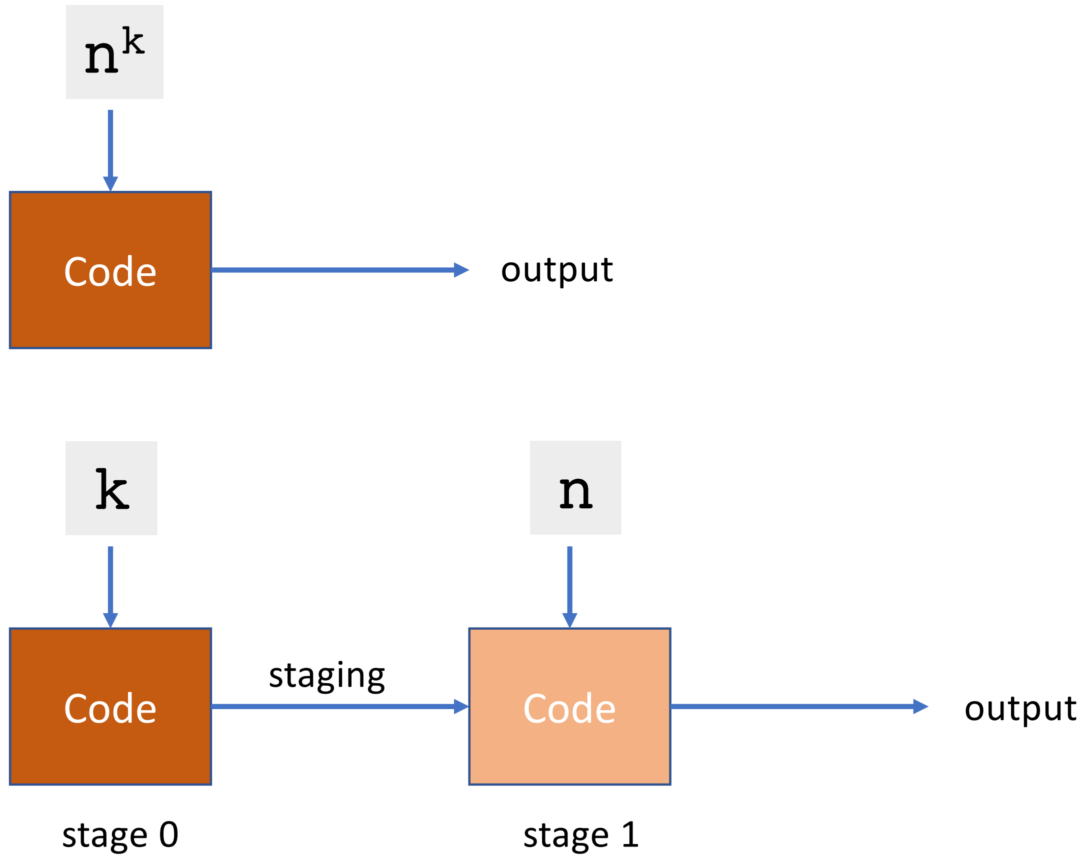
Multi-stage programming: example

```
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
```



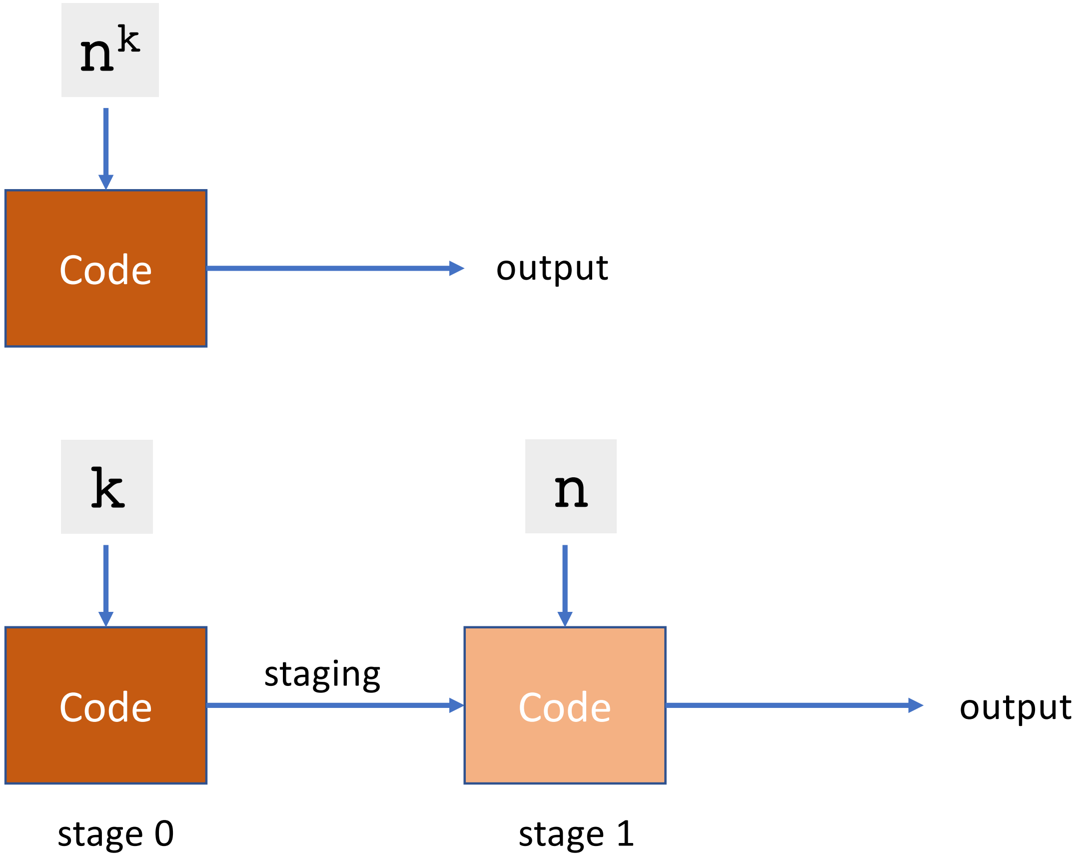
Multi-stage programming: example

```
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>
```



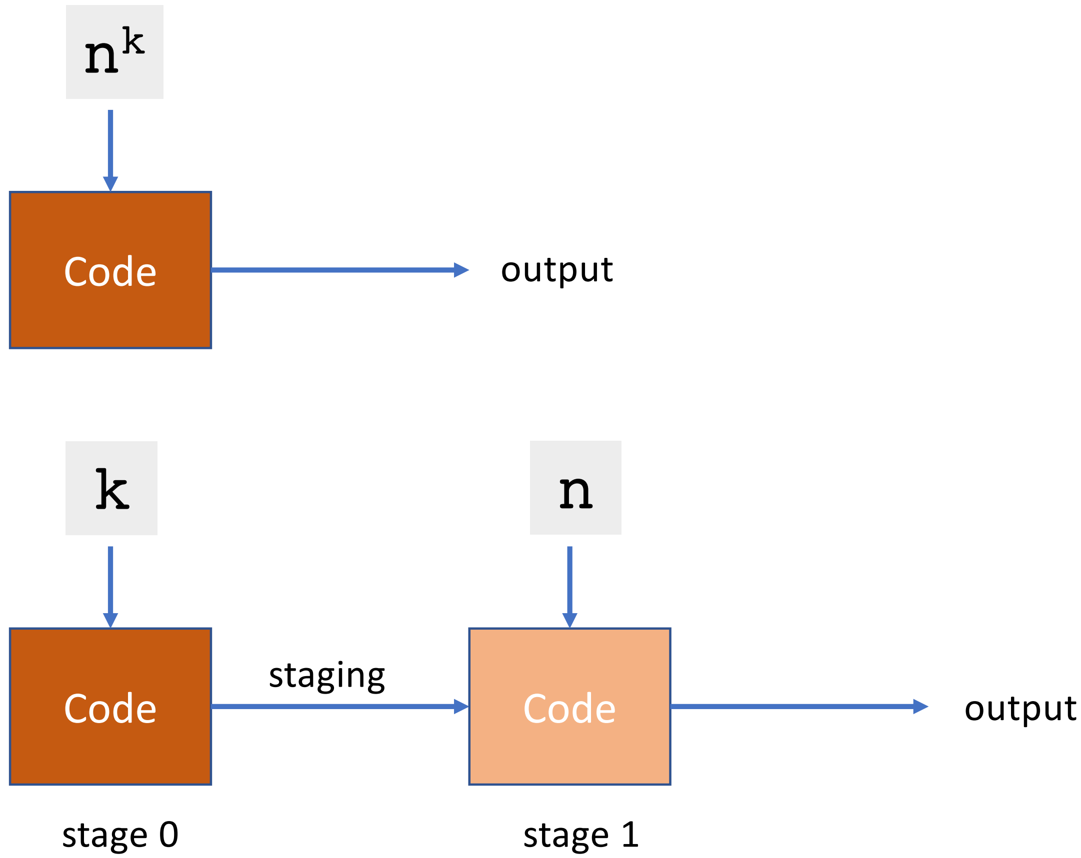
Multi-stage programming: example

```
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>
```



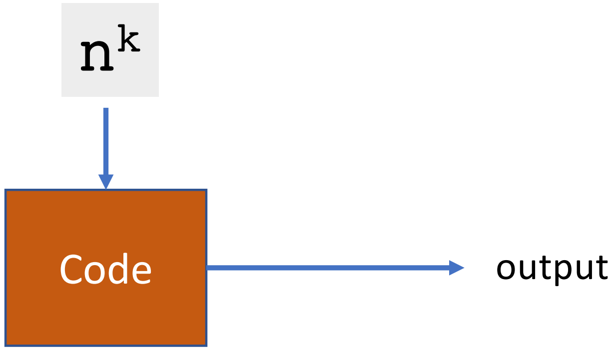
Multi-stage programming: example

```
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>
```

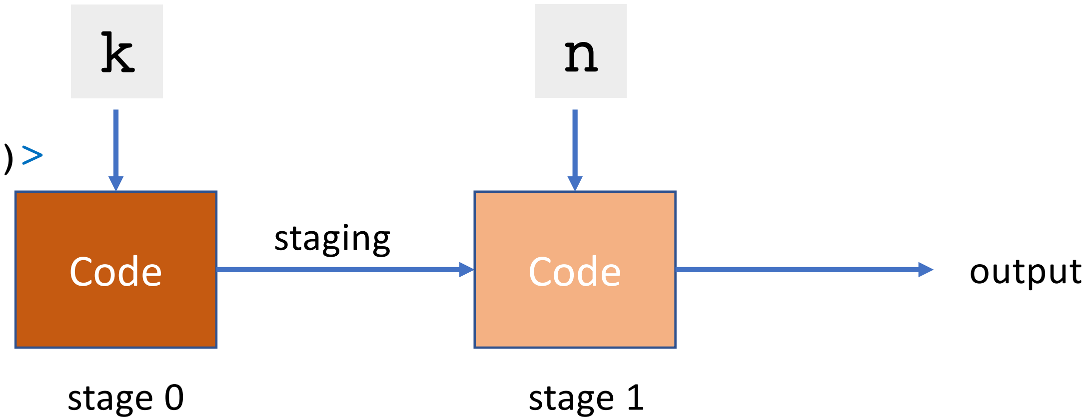


Multi-stage programming: example

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

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

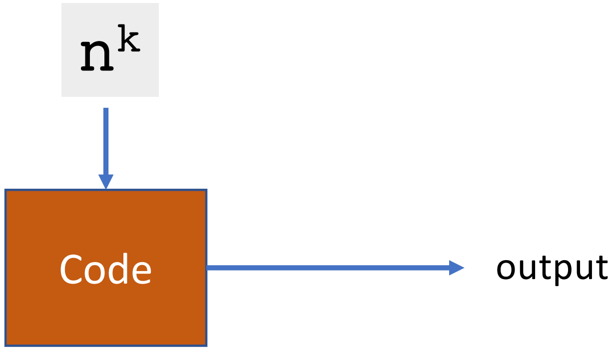



```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k - 1) n)>
```

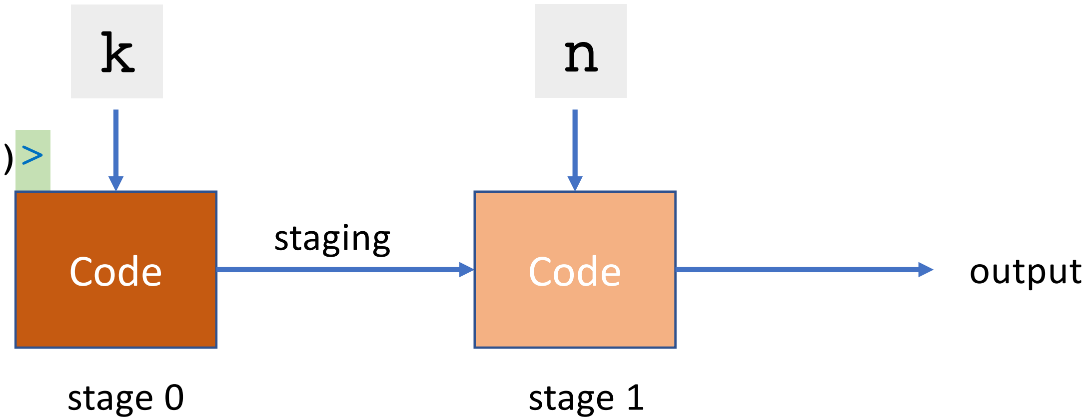


Multi-stage programming: example

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

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

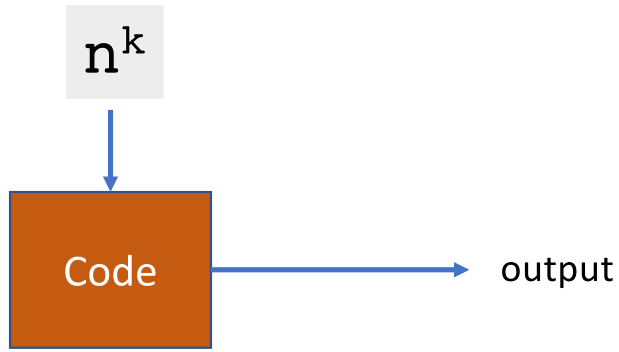



```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k - 1) n)>
```

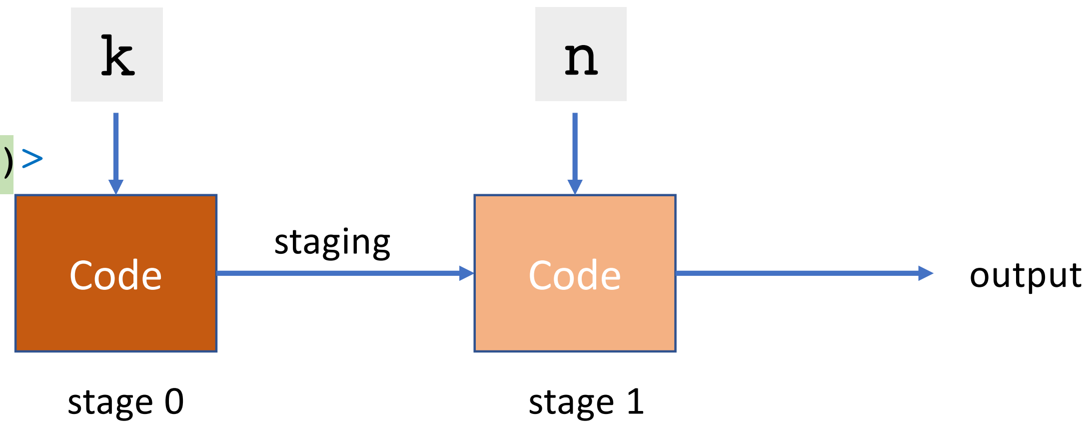


Multi-stage programming: example

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

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

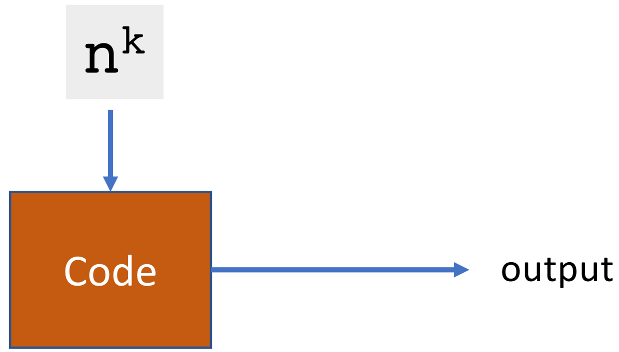



```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k - 1) n)>
```

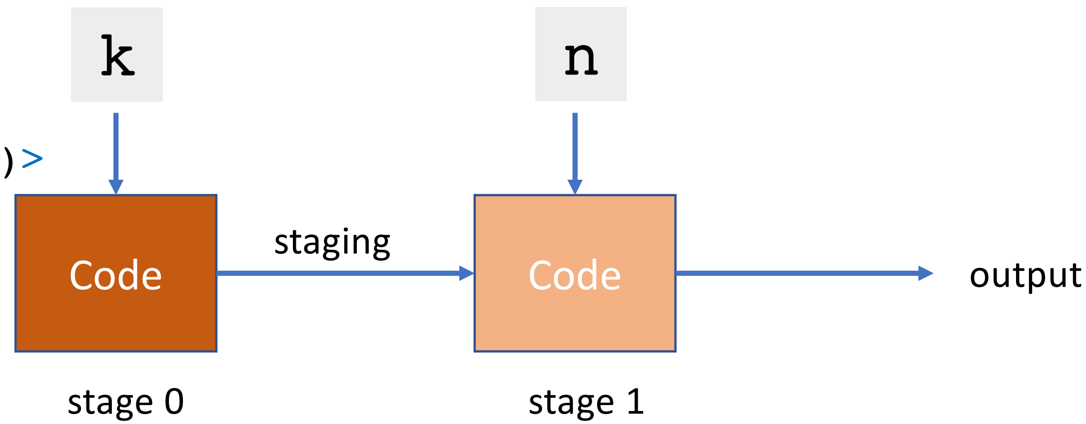


Multi-stage programming: example

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

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

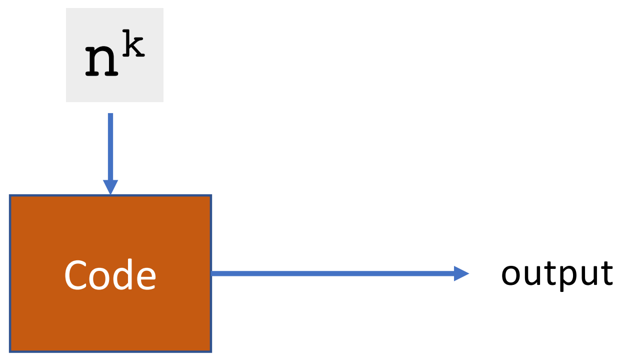



```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>
```

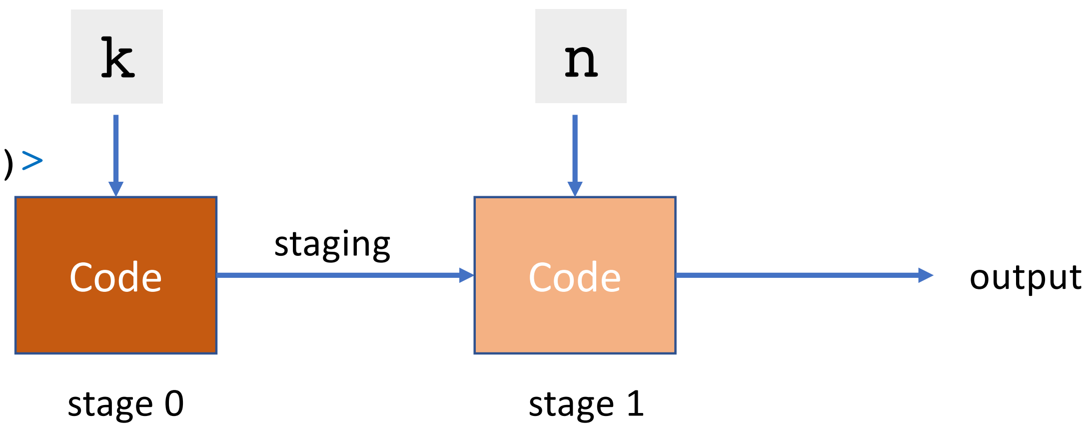


Multi-stage programming: example

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

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

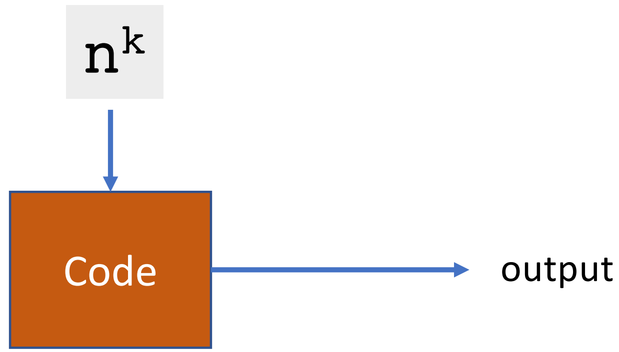



```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

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

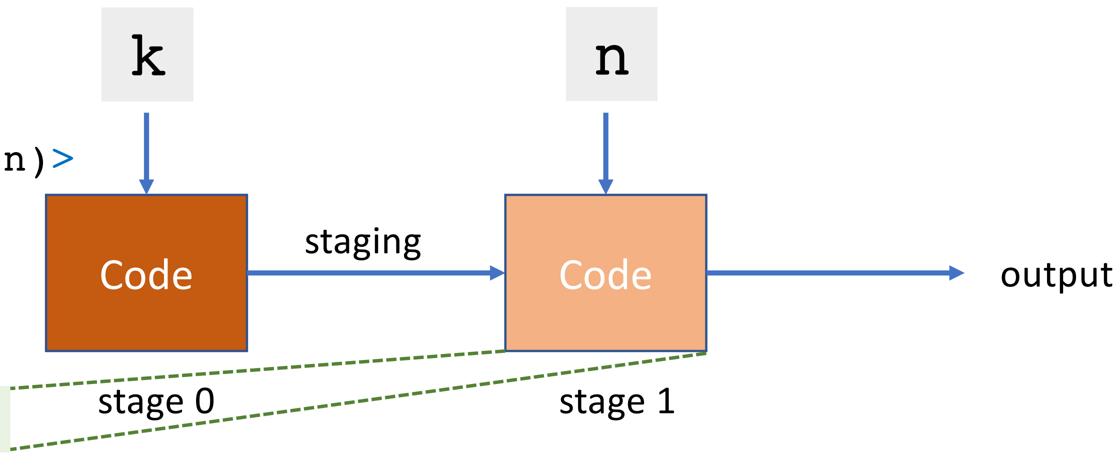


Multi-stage programming: example

```
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…

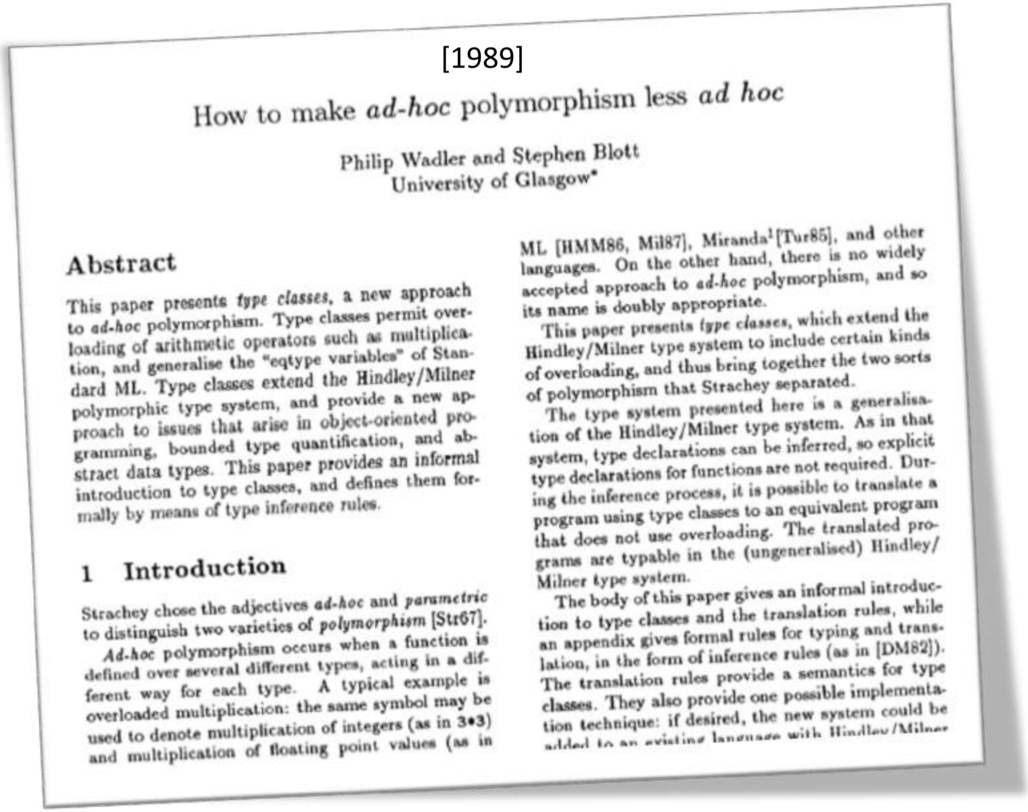
```
qpower :: Int -> Code Int -> Code Int
qpower 0 n = <1>
qpower k n = <$(n) * $(qpower (k – 1) n)>

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>)
```

Type classes

[1989]



How to make *ad-hoc* polymorphism less *ad hoc*

Philip Wadler and Stephen Blott
University of Glasgow*

Abstract

This paper presents *type classes*, a new approach to *ad-hoc* polymorphism. Type classes permit overloading of arithmetic operators such as multiplication, and generalise the "eqtype variables" of Standard ML. Type classes extend the Hindley/Milner polymorphic type system, and provide a new approach to issues that arise in object-oriented programming, bounded type quantification, and abstract data types. This paper provides an informal introduction to type classes, and defines them formally by means of type inference rules.

1 Introduction

Strachey chose the adjectives *ad-hoc* and *parametric* to distinguish two varieties of *polymorphism* [Str67].

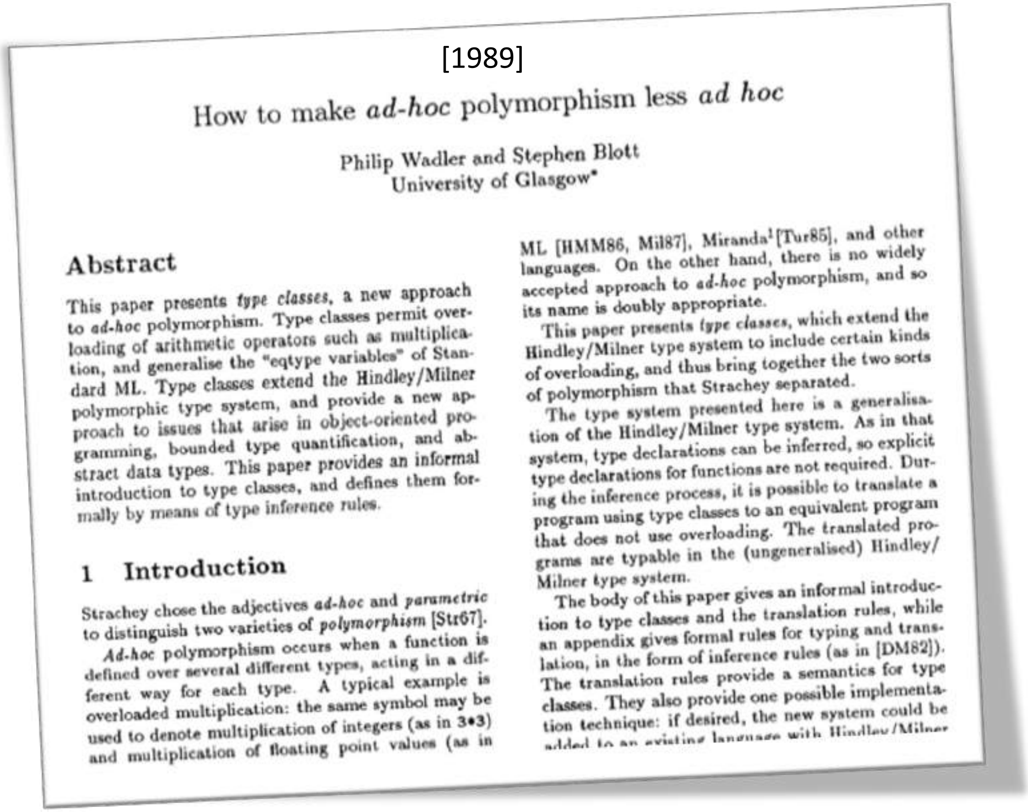
Ad-hoc polymorphism occurs when a function is defined over several different types, acting in a different way for each type. A typical example is overloaded multiplication: the same symbol may be used to denote multiplication of integers (as in 3*3) and multiplication of floating point values (as in ML [HMM86, Mil87], Miranda[1] [Tur85], and other languages. On the other hand, there is no widely accepted approach to *ad-hoc* polymorphism, and so its name is doubly appropriate.

This paper presents *type classes*, which extend the Hindley/Milner type system to include certain kinds of overloading, and thus bring together the two sorts of polymorphism that Strachey separated.

The type system presented here is a generalisation of the Hindley/Milner type system. As in that system, type declarations can be inferred, so explicit type declarations for functions are not required. During the inference process, it is possible to translate a program using type classes to an equivalent program that does not use overloading. The translated programs are typable in the (ungeneralised) Hindley/Milner type system.

The body of this paper gives an informal introduction to type classes and the translation rules, while an appendix gives formal rules for typing and translation, in the form of inference rules (as in [DM82]). The translation rules provide a semantics for type classes. They also provide one possible implementation technique: if desired, the new system could be added to an existing language with Hindley/Milner

Type classes


[1989]

How to make *ad-hoc* polymorphism less *ad hoc*

Philip Wadler and Stephen Blott
University of Glasgow*

Abstract

This paper presents *type classes*, a new approach to *ad-hoc* polymorphism. Type classes permit overloading of arithmetic operators such as multiplication, and generalise the "eqtype variables" of Standard ML. Type classes extend the Hindley/Milner polymorphic type system, and provide a new approach to issues that arise in object-oriented programming, bounded type quantification, and abstract data types. This paper provides an informal introduction to type classes, and defines them formally by means of type inference rules.

1 Introduction

Strachey chose the adjectives *ad-hoc* and *parametric* to distinguish two varieties of *polymorphism* [Str67].

Ad-hoc polymorphism occurs when a function is defined over several different types, acting in a different way for each type. A typical example is overloaded multiplication: the same symbol may be used to denote multiplication of integers (as in 3*3) and multiplication of floating point values (as in

ML [HMM86, Mil87], Miranda[1] [Tur85], and other languages. On the other hand, there is no widely accepted approach to *ad-hoc* polymorphism, and so its name is doubly appropriate.

This paper presents *type classes*, which extend the Hindley/Milner type system to include certain kinds of overloading, and thus bring together the two sorts of polymorphism that Strachey separated.

The type system presented here is a generalisation of the Hindley/Milner type system. As in that system, type declarations can be inferred, so explicit type declarations for functions are not required. During the inference process, it is possible to translate a program using type classes to an equivalent program that does not use overloading. The translated programs are typable in the (ungeneralised) Hindley/Milner type system.

The body of this paper gives an informal introduction to type classes and the translation rules, while an appendix gives formal rules for typing and translation, in the form of inference rules (as in [DM82]). The translation rules provide a semantics for type classes. They also provide one possible implementation technique: if desired, the new system could be added to an existing language with Hindley/Milner

```
class Show a where
  show :: a -> String

instance Show Int where
  show = primShowInt

instance Show Bool where
  show = primShowBool

print :: Show a => a -> String
print x = show x
```

Multi-stage programming and type classes

```
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 constraint

type-directed

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

- Quotations
- Splice environments

This talk

unsound

inspire

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

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

type-directed

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

- Quotations
- Splice environments

This talk

unsound

inspire

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

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

type-directed

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

- Quotations
- Splice environments

This talk

unsound

inspire

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

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

type-directed

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

- Quotations
- Splice environments

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

This talk

unsound

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

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

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

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

- 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 constraint

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

Multi-stage programming: well-typedness

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: well-typedness

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
```

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

Multi-stage programming: well-typedness

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
```

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

Multi-stage programming: well-typedness

Quotation

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

`e :: Int` $\Longrightarrow$ `<e> :: Code Int`

Splice

extracts the expression from its representation

`e :: Code Int` $\Longrightarrow$ `$e :: Int`

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

context

`x : int`

Multi-stage programming: well-typedness

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`

$$\Gamma \vdash e : \tau$$

context

expr

`x : int`

Multi-stage programming: well-typedness

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
```

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

context expr type

```
x : int
```

Multi-stage programming: well-typedness

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 expr type

`x : int`

Multi-stage programming: well-typedness

Quotation

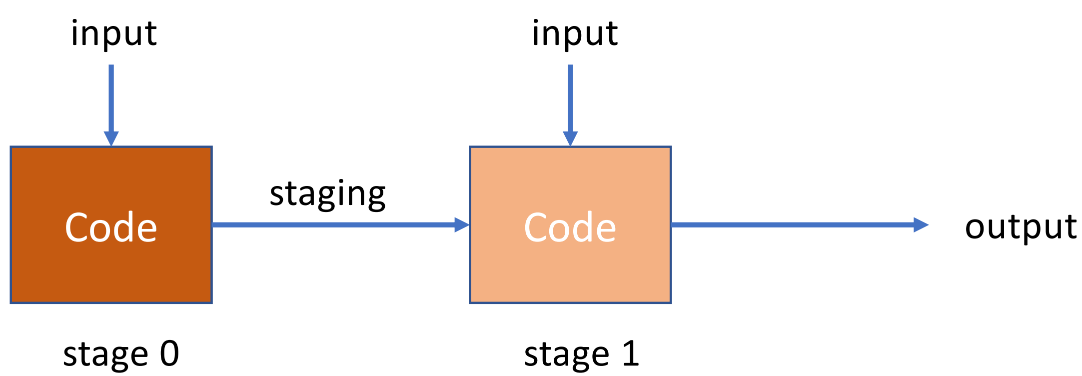
a representation of the expression as program fragment in a future stage

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

Splice

extracts the expression from its representation

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



Multi-stage programming: well-typedness

Quotation

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

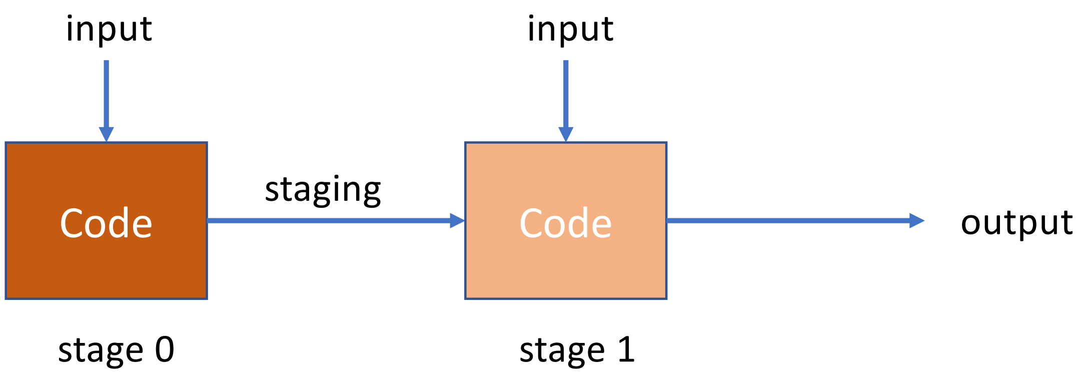
$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \langle e \rangle : \mathrm{Code}\,\tau}$$

Splice

extracts the expression from its representation

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

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



Multi-stage programming: well-typedness

Quotation

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

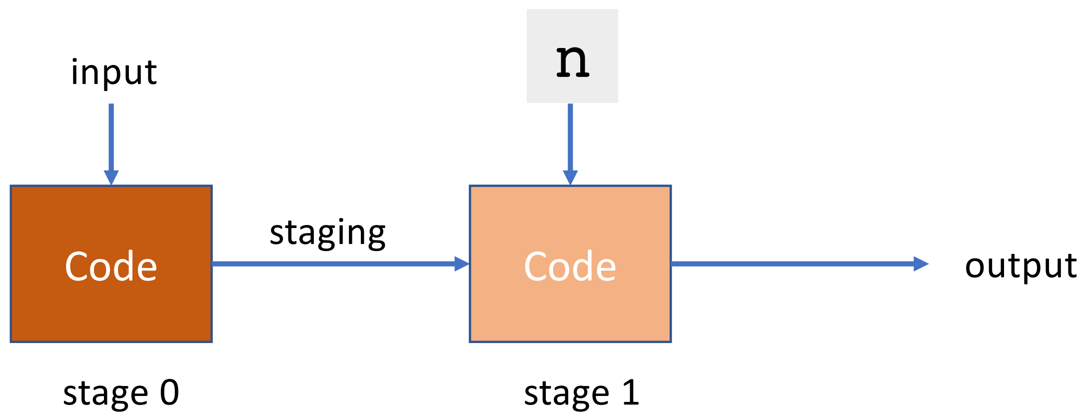
$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \langle e \rangle : \mathrm{Code}\,\tau}$$

Splice

extracts the expression from its representation

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

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



Multi-stage programming: well-typedness

Quotation

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

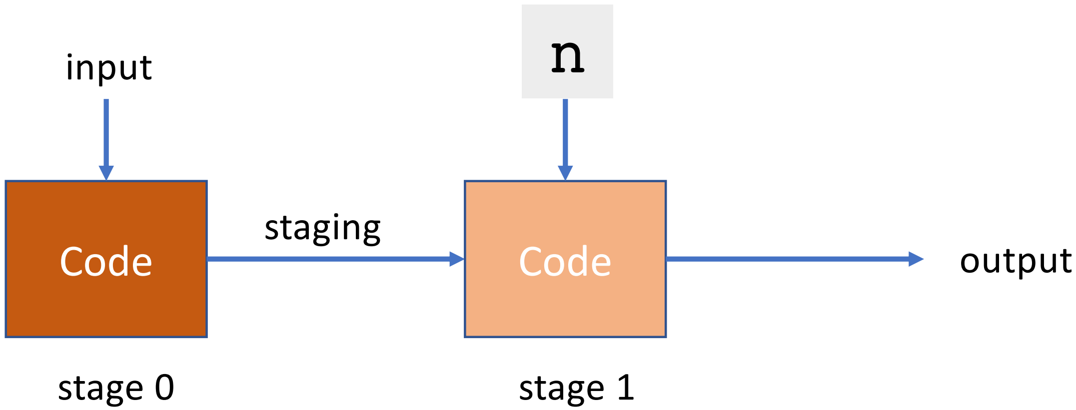
$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \langle e \rangle : \mathrm{Code}\, \tau}$$

Splice

extracts the expression from its representation

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

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

   ⟶   n * $(qpower (n − 1) <n>)
```



Multi-stage programming: well-typedness

Quotation

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

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

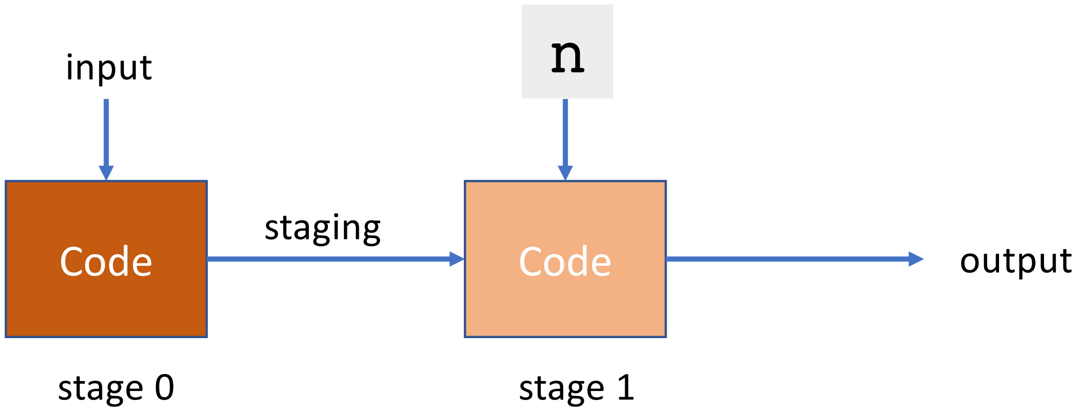
Splice

extracts the expression from its representation

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

```
qpowerN :: Int
qpo            $(qpower n <n>)
   →   n * $(qpower (n − 1) <n>)
```

rejected!



Multi-stage programming: well-typedness

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   expr   type

`x : int`

Well-stagedness: the level of an expression

Quotation

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

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

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

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

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

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

Well-stagedness: the level restriction

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

```
hasty :: Code Int -> Int
hasty = \y -> $(y)

tardy :: Int -> Code Int
tardy = \z -> < z >

timely :: Code (Int -> Int)
timely = < \x -> x >
```

Well-stagedness: the level restriction

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

well-typed? well-staged?

```
hasty :: Code Int -> Int
hasty = \y -> $(y)
```

```
tardy :: Int -> Code Int
tardy = \z -> < z >
```

```
timely :: Code (Int -> Int)
timely = < \x -> x >
```

Well-stagedness: the level restriction

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

	well-typed?	well-staged?
`hasty :: Code Int -> Int` `hasty = \y -> $(y)`	✓	
`tardy :: Int -> Code Int` `tardy = \z -> < z >`		
`timely :: Code (Int -> Int)` `timely = < \x -> x >`		

Well-stagedness: the level restriction

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

	well-typed?	well-staged?
`hasty :: Code Int -> Int` `hasty = \y -> $(y)`	✓	✗
`tardy :: Int -> Code Int` `tardy = \z -> < z >`		
`timely :: Code (Int -> Int)` `timely = < \x -> x >`		

Well-stagedness: the level restriction

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

	well-typed?	well-staged?
`hasty :: Code Int -> Int` `hasty = \y -> $(y)`	✓	✗
`tardy :: Int -> Code Int` `tardy = \z -> < z >`	✓	
`timely :: Code (Int -> Int)` `timely = < \x -> x >`		

Well-stagedness: the level restriction

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

	well-typed?	well-staged?
`hasty :: Code Int -> Int` `hasty = \y -> $(y)`	✓	✗
`tardy :: Int -> Code Int` `tardy = \z -> < z >`	✓	✗
`timely :: Code (Int -> Int)` `timely = < \x -> x >`		

Well-stagedness: the level restriction

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

	well-typed?	well-staged?
`hasty :: Code Int -> Int` `hasty = \y -> $(y)`	✓	✗
`tardy :: Int -> Code Int` `tardy = \z -> < z >`	✓	✗
`timely :: Code (Int -> Int)` `timely = < \x -> x >`	✓	

Well-stagedness: the level restriction

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

	well-typed?	well-staged?
`hasty :: Code Int -> Int` `hasty = \y -> $(y)`	✓	✗
`tardy :: Int -> Code Int` `tardy = \z -> < z >`	✓	✗
`timely :: Code (Int -> Int)` `timely = < \x -> x >`	✓	✓

Well-stagedness: the level restriction

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

	well-typed?	well-staged? (path-based persistence for top-level identifiers)
`hasty :: Code Int -> Int` `hasty = \y -> $(y)`	✓	✗
`tardy :: Int -> Code Int` `tardy = \z -> < z >`	✓	✗
`timely :: Code (Int -> Int)` `timely = < \x -> x >`	✓	✓

Is the problem with qpower well-stageness?

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

```
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-typed? 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-typed? 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-typed?	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

Code	well-typed?	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-typed?	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

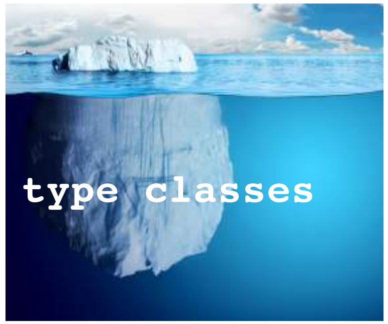
well-typed? well-staged?

```
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>)
```

✓ ?



Type Classes

```haskell
class Show a where
  show :: a -> String

instance Show Int where
  show = primShowInt

instance Show Bool where
  show = primShowBool

print :: Show a => a -> String
print x = show x
```

Type Classes

```
class Show a where
  show :: a -> String

instance Show Int where
  show = primShowInt

instance Show Bool where
  show = primShowBool

print :: Show a => a -> String
print x = show x
```

Type Classes

```
class Show a where
  show :: a -> String

instance Show Int where
  show = primShowInt

instance Show Bool where
  show = primShowBool

print :: Show a => a -> String
print x = show x
```

dictionary-passing elaboration

Type Classes

```
class Show a where
  show :: a -> String

instance Show Int where
  show = primShowInt

instance Show Bool where
  show = primShowBool

print :: Show a => a -> String
print x = show x
```

dictionary-passing elaboration

```
data ShowDict a = ShowDict
  { show :: a -> String }
```

Type Classes

```
class Show a where
  show :: a -> String

instance Show Int where
  show = primShowInt

instance Show Bool where
  show = primShowBool

print :: Show a => a -> String
print x = show x
```

dictionary-passing elaboration

```
data ShowDict a = ShowDict
  { show :: a -> String }

showInt = ShowDict
  { show = primShowInt }

showBool = ShowDict
  { show = primShowBool }
```

Type Classes

```
class Show a where
  show :: a -> String

instance Show Int where
  show = primShowInt

instance Show Bool where
  show = primShowBool

print :: Show a => a -> String
print x = show x
```

dictionary-passing elaboration

```
data ShowDict a = ShowDict
  { show :: a -> String }

showInt = ShowDict
  { show = primShowInt }

showBool = ShowDict
  { show = primShowBool }

print :: ShowDict a -> a -> String
print dShow x = show dShow x
```

Type Classes

```haskell
class Show a where
  show :: a -> String

instance Show Int where
  show = primShowInt

instance Show Bool where
  show = primShowBool

print :: Show a => a -> String
print x = show x
```

dictionary-passing elaboration

```haskell
data ShowDict a = ShowDict
  { show :: a -> String }

showInt = ShowDict
  { show = primShowInt }

showBool = ShowDict
  { show = primShowBool }

print :: ShowDict a -> a -> String
print dShow x = show dShow x
```

Type Classes

```haskell
class Show a where
  show :: a -> String

instance Show Int where
  show = primShowInt

instance Show Bool where
  show = primShowBool

print :: Show a => a -> String
print x = show x
```

dictionary-passing elaboration

```haskell
data ShowDict a = ShowDict
  { show :: a -> String }

showInt = ShowDict
  { show = primShowInt }

showBool = ShowDict
  { show = primShowBool }

print :: ShowDict a -> a -> String
print dShow x = show dShow x
```

Type Classes

```
class Show a where
  show :: a -> String

instance Show Int where
  show = primShowInt

instance Show Bool where
  show = primShowBool

print :: Show a => a -> String
print x = show x
```

dictionary-passing elaboration

```
data ShowDict a = ShowDict
  { show :: a -> String }

showInt = ShowDict
  { show = primShowInt }

showBool = ShowDict
  { show = primShowBool }

print :: ShowDict a -> a -> String
print dShow x = show dShow x
```

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>)
```

dictionary-passing elaboration

well-staged?

```
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

```
print :: Show a => a -> String
print x = show x
```

```
print :: ShowDict a -> a -> String
print dShow x = show dShow x
```

Constraint resolution

```
print :: Show a => a -> String
print x = show x
```

```
print :: ShowDict a -> a -> String
print dShow x = show dShow x
```

Constraint resolution

$$\Gamma \models C \leadsto e$$

```
print :: Show a => a -> String
print x = show x
```

```
print :: ShowDict a -> a -> String
print dShow x = show dShow x
```

Constraint resolution

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

```
print :: Show a => a -> String
print x = show x
```

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

```
print :: ShowDict a -> a -> String
print dShow x = show dShow x
```

Constraint resolution

```
print :: Show a => a -> String
print x = show x
```

```
print :: ShowDict a -> a -> String
print dShow x = show dShow x
```

$$\frac{dShow : Show\ a \in \Gamma}{\Gamma \models Show\ a \rightsquigarrow dShow}$$

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

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

Level-indexed constraint resolution

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

```
print :: Show a => a -> String
print x = show x
```

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

```
print :: ShowDict a -> a -> String
print dShow x = show dShow x
```

$$\frac{dShow : Show\ a \in \Gamma}{\Gamma \models Show\ a \rightsquigarrow dShow}$$

Level-indexed constraint resolution

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

```
print :: Show a => a -> String
print x = show x
```

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

```
print :: ShowDict a -> a -> String
print dShow x = show dShow x
```

$$\frac{dShow : (Show\ a, 0) \in \Gamma}{\Gamma \models^{0} Show\ a \leadsto dShow}$$

Level-indexed constraint resolution

```
print :: Show a => a -> String
print x = show x
```

```
print :: ShowDict a -> a -> String
print dShow x = show dShow x
```

$$\frac{dShow : (Show\ a, 0) \in \Gamma}{\Gamma \models^{0} Show\ a \leadsto dShow}$$

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

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

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

Level-indexed constraint resolution

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

```
print :: Show a => a -> String
print x = show x
```

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

```
print :: ShowDict a -> a -> String
print dShow x = show dShow x
```

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

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

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

This talk

unsound

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

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

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

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

- 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 constraint

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 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

$$\boxed{e_1 \; \langle e_2 \; \$e_3 \rangle}$$

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

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

Level-indexed Evaluation

level 0 1 0

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

Level-indexed Evaluation

level 0 1 0

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

$$\frac{e_1 \xrightarrow{0} v_1 \qquad \dfrac{\dfrac{e_2 \xrightarrow{1} e_2 \qquad \dfrac{e_3 \xrightarrow{0} v_3}{\$ e_3 \xrightarrow{1} v_3}}{e_2 \; \$ e_3 \xrightarrow{1} e_2 \; v_3}}{\langle e_2 \; \$ e_3 \rangle \xrightarrow{0} \langle e_2 \; \$ v_3 \rangle}}{e_1 \; \langle e_2 \; \$ e_3 \rangle \xrightarrow{0} v_1 \; \langle e_2 \; \$ v_3 \rangle}$$

Level-indexed Evaluation

level 0 1 0

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

$$\frac{e_1 \xrightarrow{0} v_1 \qquad \dfrac{\dfrac{e_2 \xrightarrow{1} e_2 \qquad \dfrac{e_3 \xrightarrow{0} v_3}{\$e_3 \xrightarrow{1} v_3}}{e_2 \; \$e_3 \xrightarrow{1} e_2 \; v_3}}{\langle e_2 \; \$e_3 \rangle \xrightarrow{0} \langle e_2 \; \$v_3 \rangle}}{e_1 \; \langle e_2 \; \$e_3 \rangle \xrightarrow{0} v_1 \; \langle e_2 \; \$v_3 \rangle}$$

Adding complexity to implementations

Level-indexed Evaluation

level 0 1 0

$$e_1 \ \langle e_2 \ \$ e_3 \rangle \quad \boxed{\longrightarrow} \quad v_1 \ \langle e_2 \ \$ v_3 \rangle$$

$$\dfrac{e_1 \xrightarrow{0} v_1 \qquad \dfrac{\dfrac{e_2 \xrightarrow{1} e_2 \qquad \dfrac{e_3 \xrightarrow{0} v_3}{\$ e_3 \xrightarrow{1} v_3}}{e_2 \ \$ e_3 \xrightarrow{1} e_2 \ v_3}}{\langle e_2 \ \$ e_3 \rangle \xrightarrow{0} \langle e_2 \ \$ v_3 \rangle}}{e_1 \ \langle e_2 \ \$ e_3 \rangle \xrightarrow{0} v_1 \ \langle e_2 \ \$ v_3 \rangle}$$

Adding complexity to implementations

Requires inspecting and evaluating inside quotations

Key idea: splice environments

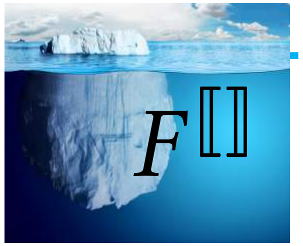
$\lambda^{[\![\Rightarrow]\!]}$

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

Key idea: splice environments

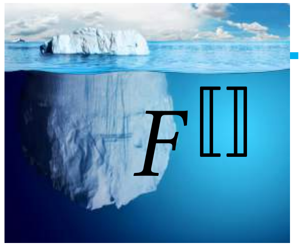
$\lambda^{[\![\Rightarrow]\!]}$

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



Key idea: splice environments

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

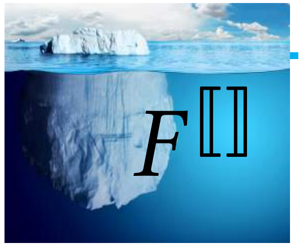


$$e_1 \, \langle e_2 \, \$ e_3 \rangle$$

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

Key idea: splice environments

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



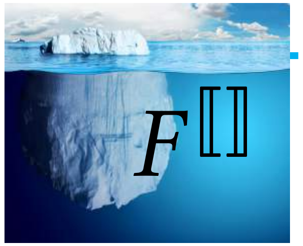
$$e_1\ \langle e_2\ \$e_3\rangle$$

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

Key idea: splice environments

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

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



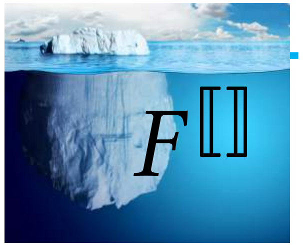
$$e_1 \, \langle e_2 \quad s \quad \rangle_{\bullet \vdash^0 s:\tau = e_3}$$

the spliced expression

Key idea: splice environments

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

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



the spliced expression

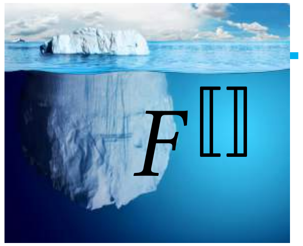
$$e_1 \, \langle e_2 \quad s \quad \rangle_{\bullet \vdash^0 s:\tau = e_3}$$

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

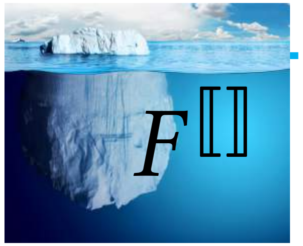
$$e_1 \langle e_2 \quad s \quad \rangle_{\bullet \vdash^0 s : \tau = e_3}$$

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

Key idea: splice environments

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

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



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

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

the spliced expression

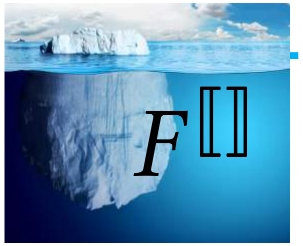
type of s (so the type of e3 is Code τ)

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

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

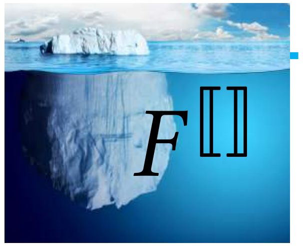
type of s (so the type of e3 is Code τ)

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

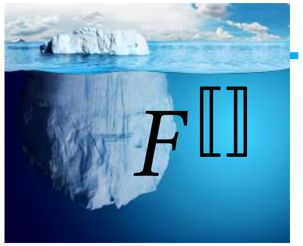
$$e_1 \, \boxed{\langle e_2 \quad s \;\rangle} \bullet \vdash^0 s{:}\tau{=}e_3$$

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

the type context

Key idea: splice environments

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

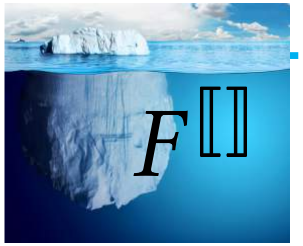


$$e_1\ \langle e_2\ \$e_3\rangle$$

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

Key idea: splice environments

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



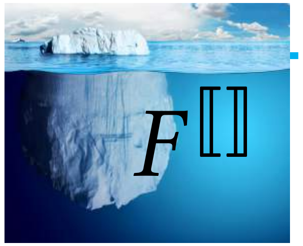
$$e_1 \; \langle e_2 \; \$ e_3 \rangle$$

✓ 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

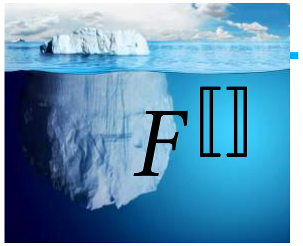
$$\frac{\phi \longrightarrow \phi'}{[\![e]\!]_{\phi} \longrightarrow [\![e]\!]_{\phi'}}$$

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

Key idea: splice environments

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

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



✓ straightforward evaluation

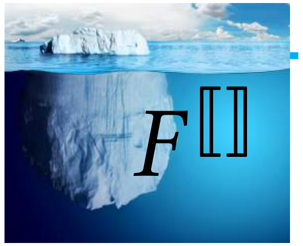
$$\frac{\phi \longrightarrow \phi'}{[\![e]\!]_{\phi} \longrightarrow [\![e]\!]_{\phi'}}$$

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

Key idea: splice environments

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

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



✓ **straightforward evaluation**

$$\frac{\phi \longrightarrow \phi'}{[\![e]\!]_{\phi} \longrightarrow [\![e]\!]_{\phi'}}$$

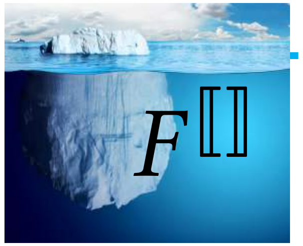
$$e_1 \; \boxed{\langle e_2 \quad s \; \rangle}\bullet \vdash^{0} s{:}\tau{=}e_3$$

✓ **opaque quotations**

Key idea: splice environments

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

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



level of the quotation

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

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

✓ straightforward evaluation

$$\frac{\phi \longrightarrow \phi'}{[\![e]\!]_\phi \longrightarrow [\![e]\!]_{\phi'}}$$

✓ opaque quotations

Type-directed elaboration

Type-directed elaboration

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

Type-directed elaboration

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

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

Type-directed elaboration

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

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

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

This talk

unsound

inspire

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

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

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

This talk

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

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

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

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

- Quotations
- Splice environments

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

✓ easy to implement and stay close to existing implementations

Integration into GHC

Integration into GHC

- Type inference

Integration into GHC

- Type inference
- Local constraints

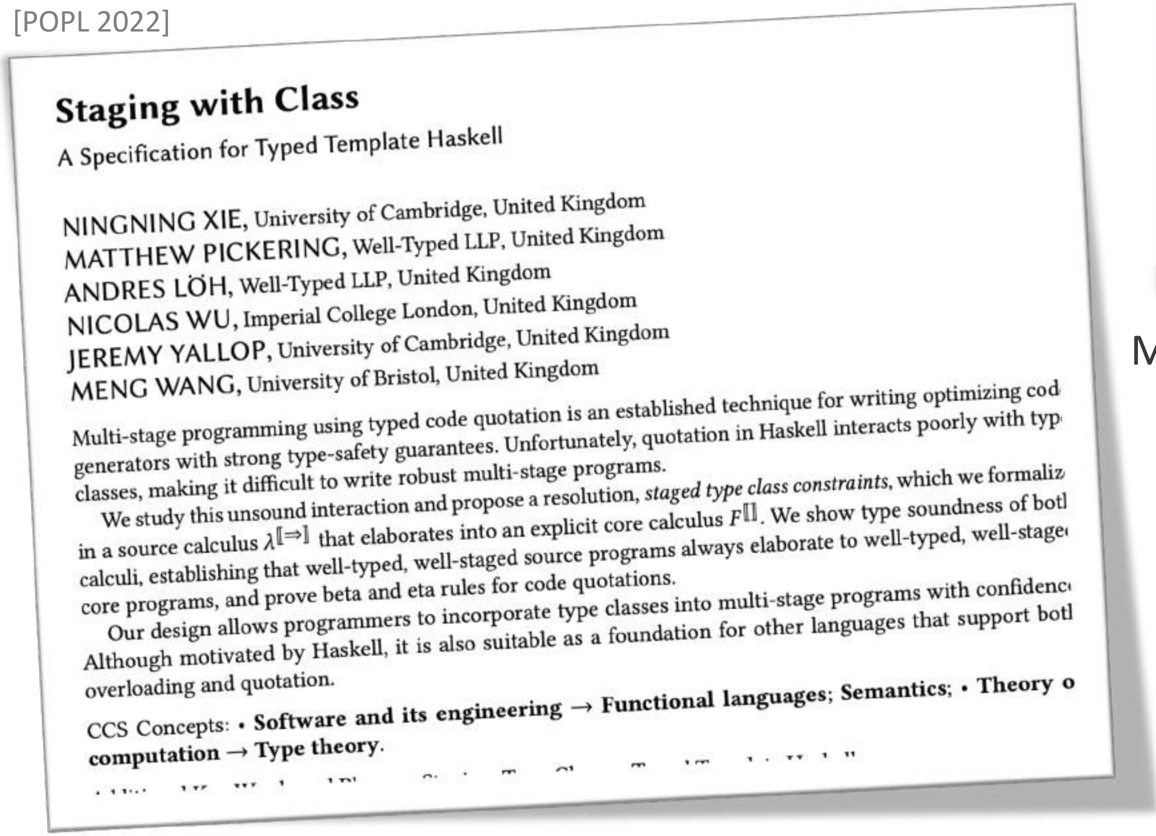
Integration into GHC

- Type inference
- Local constraints
- Quantified constraints

Integration into GHC

- Type inference
- Local constraints
- Quantified constraints
- Representation of quotations

[POPL 2022]



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.**

Matthew Pickering

Andres Löh

Nicolas Wu

Jeremy Yallop

Meng Wang

UNIVERSITY OF CAMBRIDGE

Well-Typed The Haskell Consultants

Imperial College London

University of BRISTOL

Staging with Class

A Specification of Typed Template Haskell

Ningning Xie

UNIVERSITY OF CAMBRIDGE

YOW! Lambda Jam
May 18 2022