HASKELL系列教程 I

by 韩冬@滴滴FP

- 环境配置和项目搭建
- 基础语法
- 执行模型
- data与模式匹配

项目搭建 FOR MACOS / BREW

```
brew install ghc cabal-install haskell-stack

把~/.cabal/bin加到$PATH里

常用工具：cabal install hlint ghc-mod

vim推荐安装：syntastic, ghcmod-vim
neocomplete的用户安装neco-ghc即可开启全面的haskell补全

初始化项目：cabal init
使用沙盒环境：cabal sandbox init
在项目环境中运行ghci：cabal repl

cabal文档：https://www.haskell.org/cabal/
          http://cabal.readthedocs.io/en/latest/
```

基础语法

```
-- 这是一行注释
{-
    这是一段注释
-}

1
3 :: Float

addOne :: Int -> Int
addOne x = x + 1
addOne = \ x -> x + 1

zs :: [Int]
zs = sort xs ++ sort ys
-- == (sort xs) ++ (sort ys)

-- 空格代表函数应用，优先级最高
```

基础语法

```
2 + 3              -- 中缀函数调用，  +是加法函数
(+) 1.5 2.5    -- 中缀函数调用的另一种写法

elem 3 [1,2,3]          -- elem x xs判断列表xs中是否出现x
3 `elem` [1,2,3]    -- 普通函数调用的另一种方法
```



```
infixl 6 +
-- + 的优先级是6 左结合
infixl 7 *
-- * 的优先级是7 左结合
infixr 8 ^
-- ^ 的优先级是8 右结合
```

基础语法

```
(++++) :: Int -> Int -> Int
(++++) x y = x ^ 2 + y ^ 2
infixl 5 ++++         -- 左结合，优先级5

3 ++++ 4 ++++ 5 + 6 = (3 ++++ 4) ++++ (5 + 6)
```

```
-- `...`的优先级是9 不能结合
-- 相当于`elem`的默认结合性优先级声明是
-- infix 9 `elem`

x `elem` xs `elem` ys
-- 报错，没有结合性的中缀函数不能这么连着写，必须加括号
-- (x `elem` xs) `elem` ys
-- x `elem` (xs `elem` ys)
```

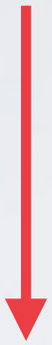
基本单元：函数

```
id :: a -> a -- 直接返回任意类型的参数
id 3         -- 3

const :: a -> b -> a -- 直接返回第一个参数
const "hello" 100    -- "hello"

(&&), (||) :: Bool -> Bool -> Bool -- 逻辑与，逻辑或
True && False          -- False
True || False          -- True

not :: Bool -> Bool    -- 逻辑反
not False              -- True

(==), (/=) :: Eq a => a -> a -> Bool
"hello" == 234         -- 编译报错，类型不符

(<), (<=), (>=), (>) :: Ord a => a -> a -> Bool
10 < 100          -- True
'a' > ‘z'         -- False
```

过程式执行模型

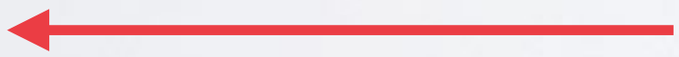
程序由一系列操作序列组成

```
main (..) {
     statement1
     statement2
     statement3
     …
}
```



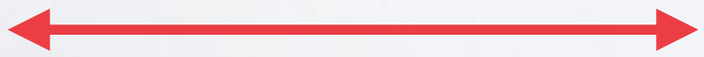
遇到嵌套的表达式如何求值?

```
x = func1( func2( m + n ) );
```



嵌套的表达式有求值顺序么?

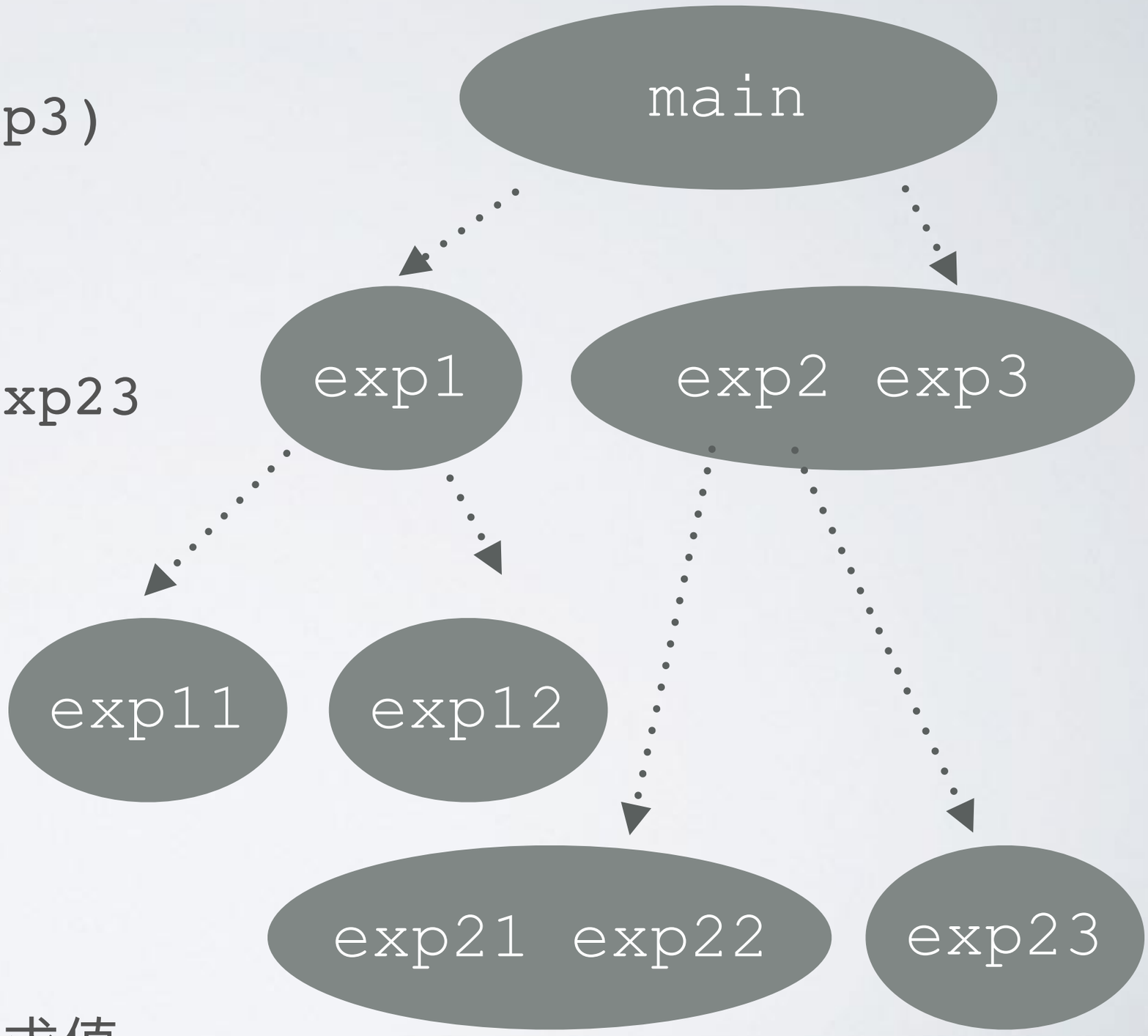
```
y = func1( a(x), b(x), c(x) );
```



HASKELL执行模型

程序由一系列表达式组成

```
main = exp1 (exp2 exp3)

exp1 = exp11 exp12 …

exp2 = exp21 exp22 exp23

exp3 = …

…
```



整个程序是一个大**表达式**

子表达式在**被需要的时候**被求值

DATA

```
data Bool = True | False
True :: Bool
False :: Bool

data Int = I# Int#  -- #表示运行时提供的原始类型
data Int = … | -2 | 0 | 1 | 2 | 3 | …
```

```
data Position = MakePosition Double Double
MakePosition 1.5 2 :: Position

MakePosition :: Double -> Double -> Position

data Position = Double :+ Double
-- 中缀构造函数以:开头
1.5 :+ 2 :: Position
(:+) 0.1 0.2 :: Position
```

模式匹配

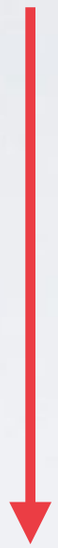
```
data Position = MakePosition Double Double

distance :: Position -> Position -> Double
distance p1 p2 =
    case p1 of MakePosition x1 y1 ->
        case p2 of MakePosition x2 y2 ->
            sqrt ((x1 - x2) ^ 2 + (y1 - y2) ^ 2)
```

模式匹配

```
pointA = MakePosition 0 0
case pointA of MakePosition x y -> … -- x = 0, y =0

pointB = MakePosition 3 4
distance pointA pointB
-- 5
```

模式匹配

```
case x of
     pattern1 -> expression1
     pattern2 -> expression2
     .
     .
     .
     patternN -> expressionN
```



```
-- if … then … else … 也是模式匹配的语法糖
if x then …
     else …

case x of
     True  -> …
     False -> …
```

模式匹配

```
-- 绑定左侧直接对参数进行模式匹配
distance (MakePosition x1 y1) (MakePosition x2 y2) =
    sqrt ((x1 - x2) ^ 2 + (y1 - y2) ^ 2)

-- let … in … 的时候进行模式匹配
distance p1 p2 =
    let MakePosition x1 y1 = p1
        MakePosition x2 y2 = p2
    in sqrt ((x1 - x2) ^ 2 + (y1 - y2) ^ 2)

-- where关键字添加函数内部的辅助绑定
distance p1 p2 =
    sqrt ((x1 - x2) ^ 2 + (y1 - y2) ^ 2)
  where
    MakePosition x1 y1 = p1
    MakePosition x2 y2 = p2
```

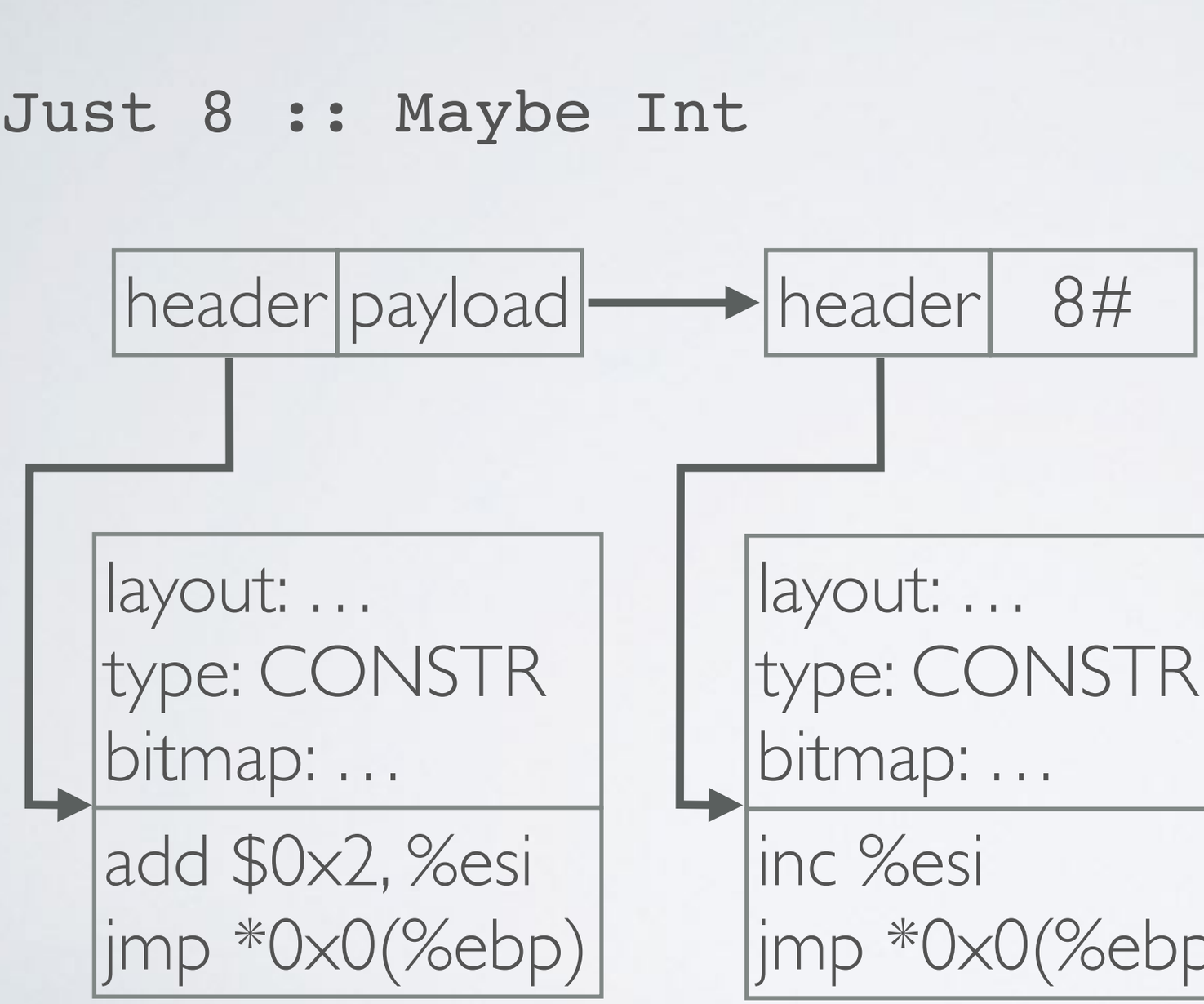
RECORD 记录语法

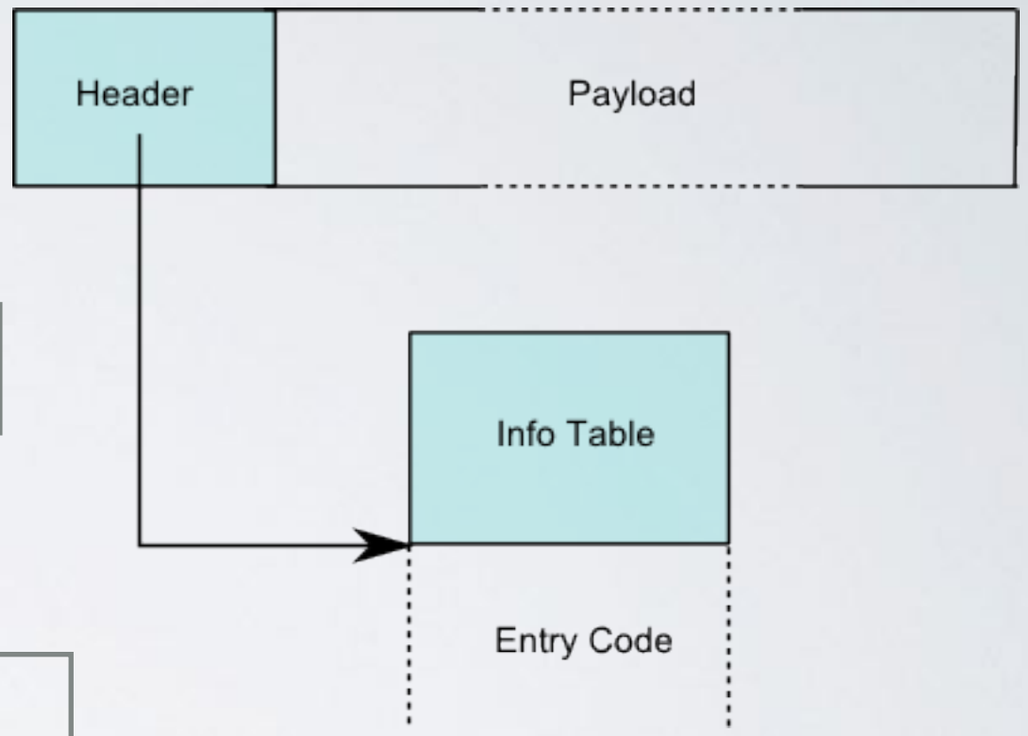
```
data User = User
    { userID   :: Int
    , userName :: String
    }

x :: User -> Int  -- \ u -> case u of User id _ -> id
y :: User -> String
```

```
peter :: User
peter = User 13 “Peter”
peter = User {userName = “Peter”, userID = 13}

x peter -- 13; y peter -- Peter

peterNew = peter {userID = 11} -- User 11 “Peter”

case peter of User id name -> …
case peter of User{userID = id, userName = name} -> …
```

HEAP OBJECT

```
data Maybe a =
    Nothing | Just a

Just 8 :: Maybe Int
```



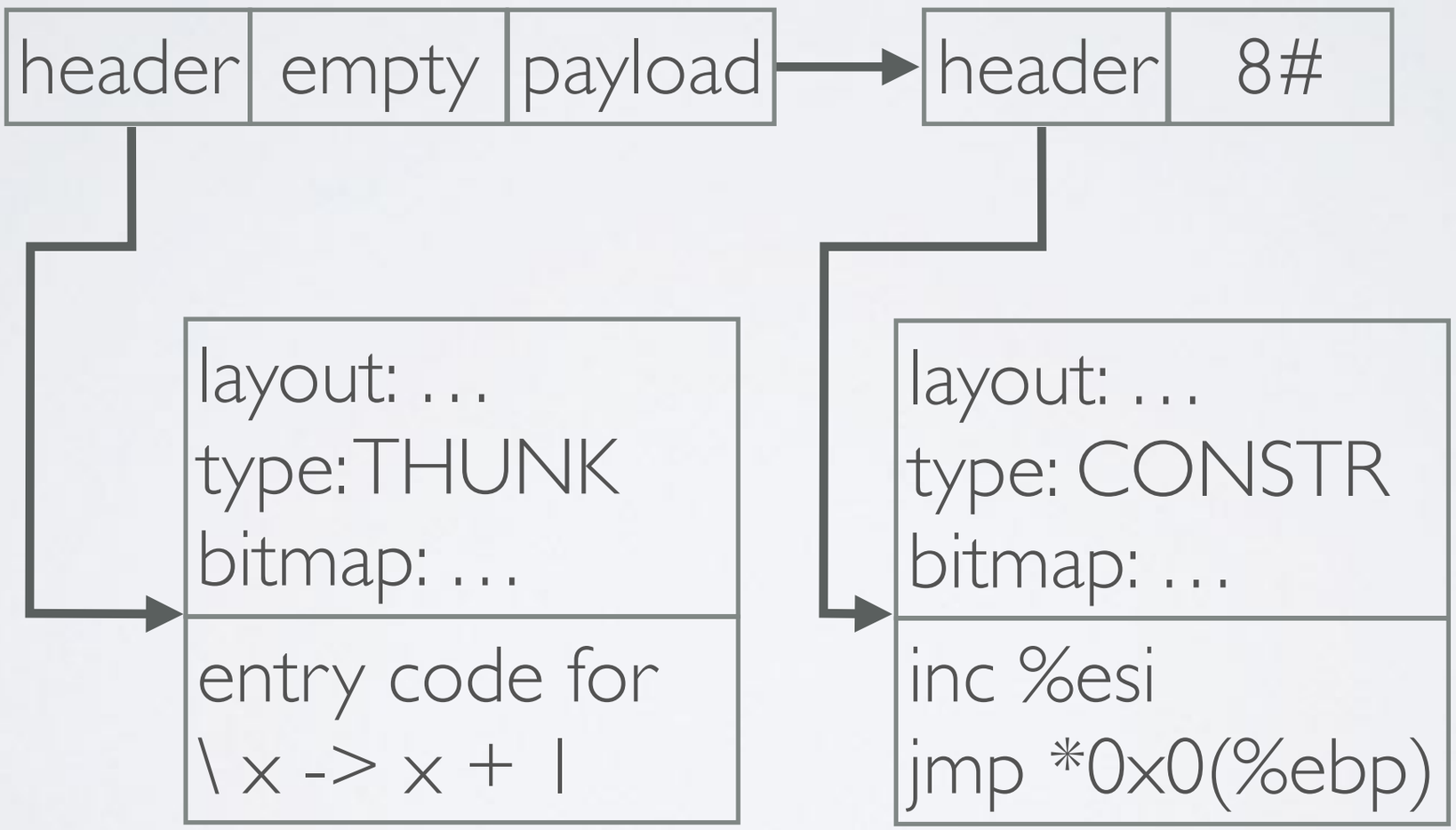


HEAP OBJECT

任务盒：`thunk`

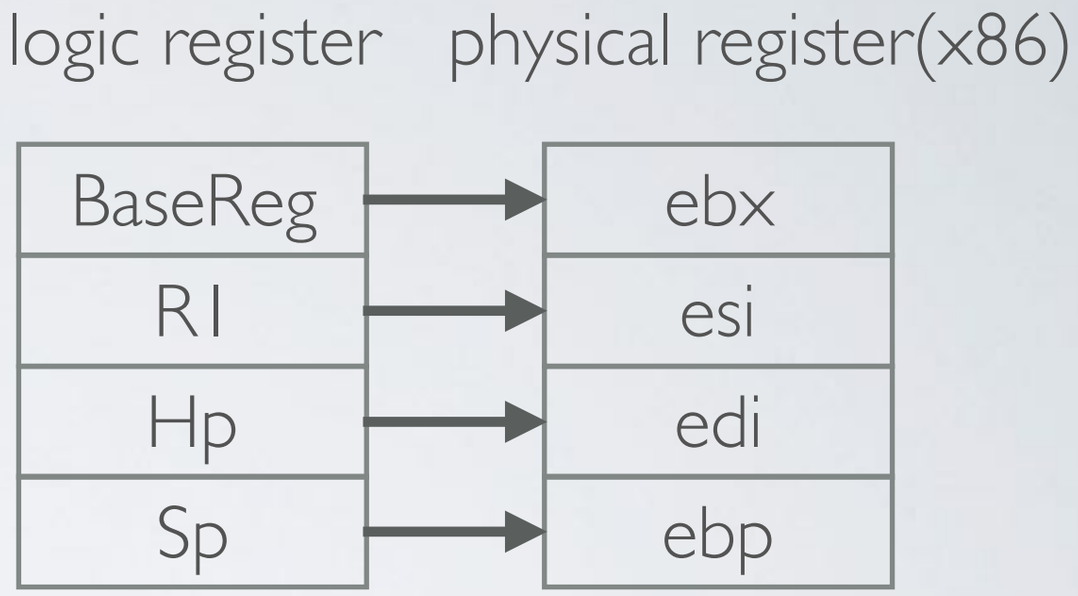
`x + 1 :: Int`

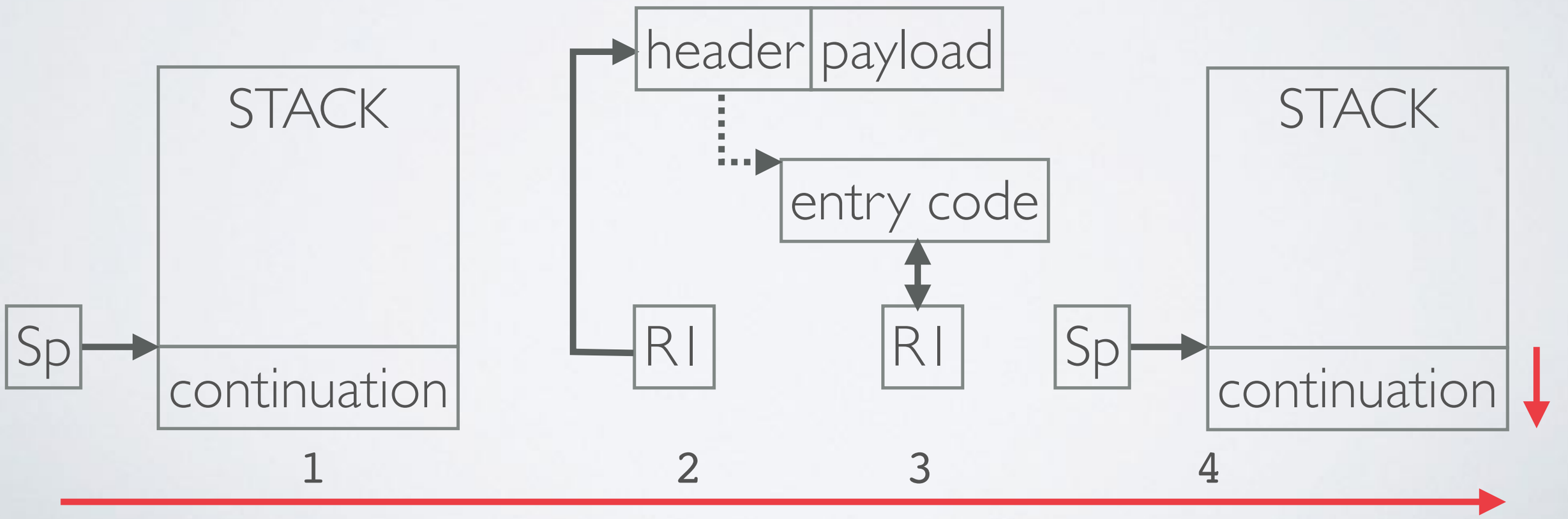
自由变量: `x = 8`



STG执行模型

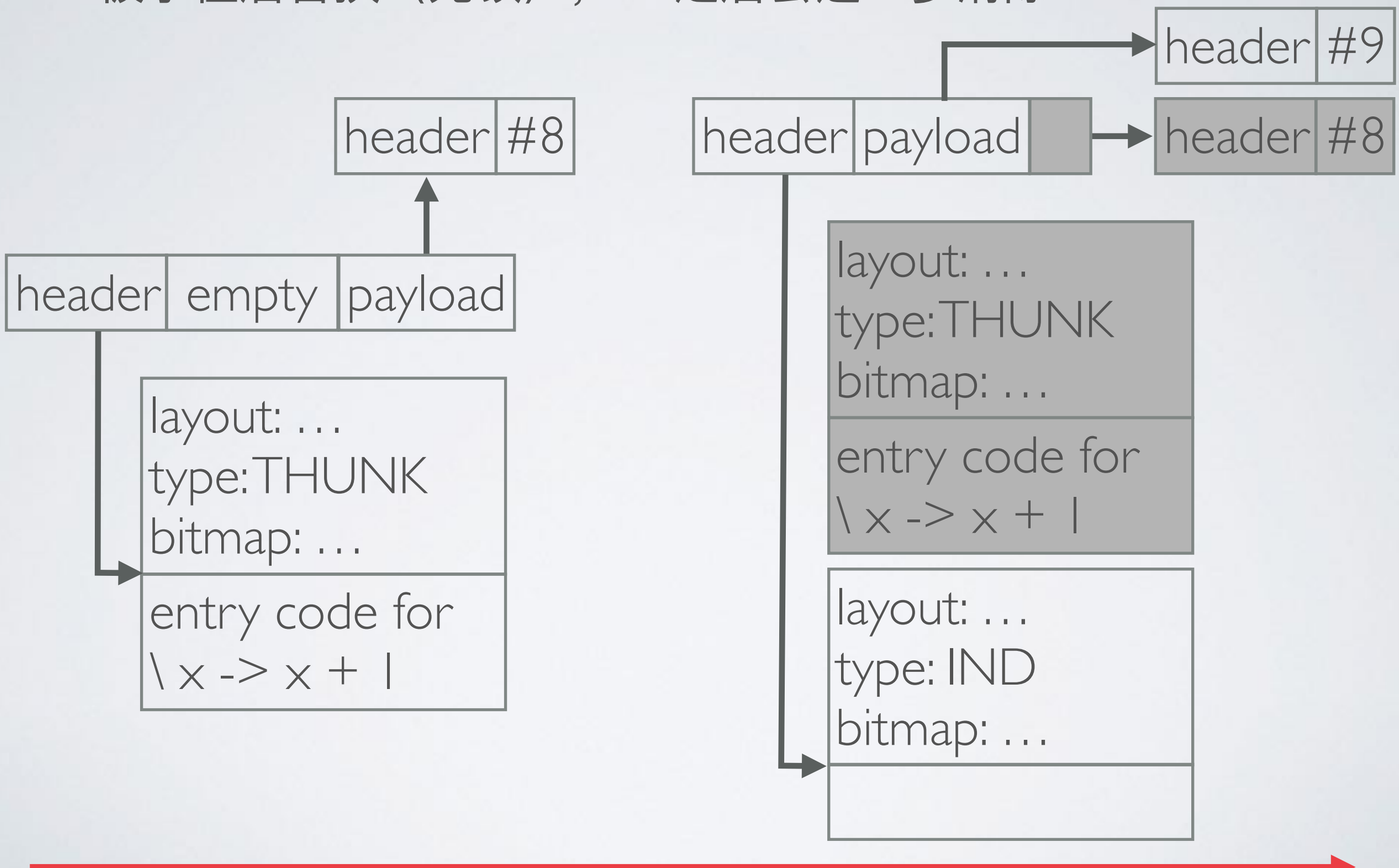
1. 把接下来要执行的代码压入stack
2. 进入R1指向的heap object
3. 执行infotable里的entry code
4. 执行stack最顶端的代码
5. 回到1





STG执行模型

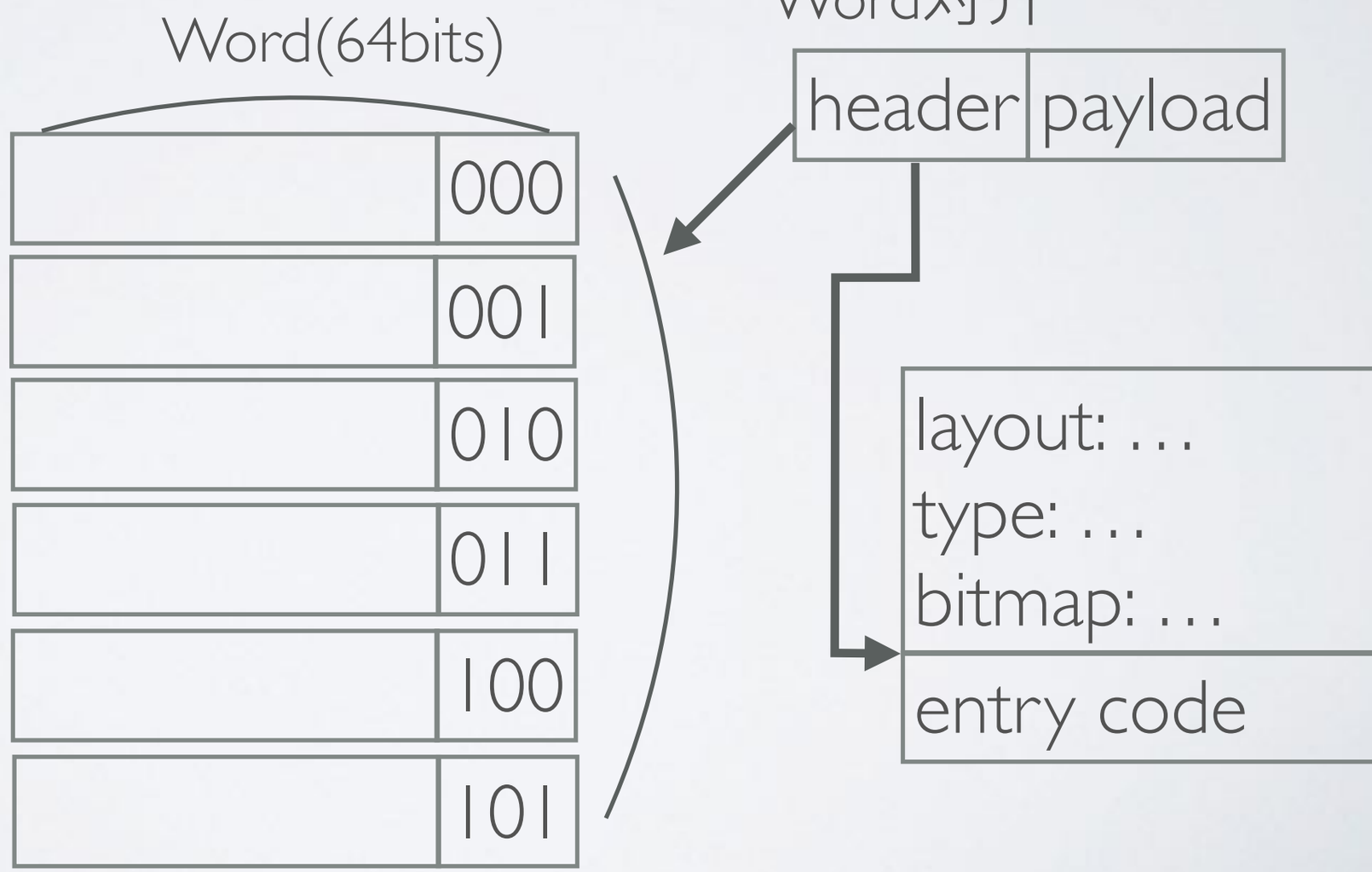
Thunk被求值后替换（无锁），GC之后会进一步清除IND



POINTER TAGGING

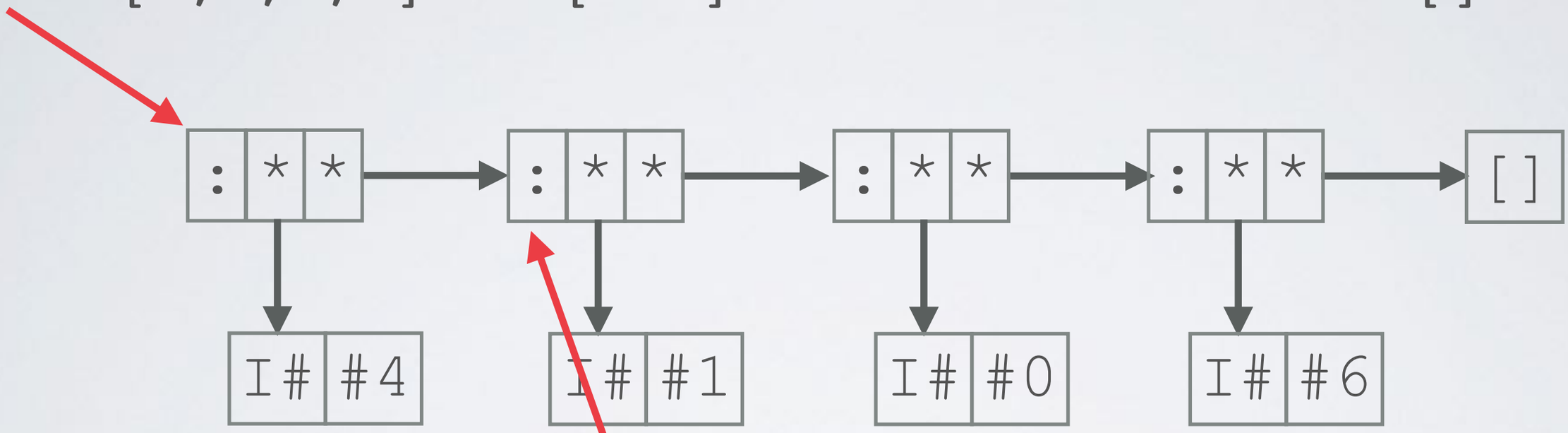
```
data JSONValue
     = JSONText Text
     | JSONNumber Scientific
     | JSONObject (HashMap String JSONValue)
     | JSONArray [JSONValue]
     | JSONNull
```

- 未求值的thunk
- 指向JSONText
- 指向JSONNumber
- 指向JSONObject
- 指向JSONArray
- 指向JSONNull



列表

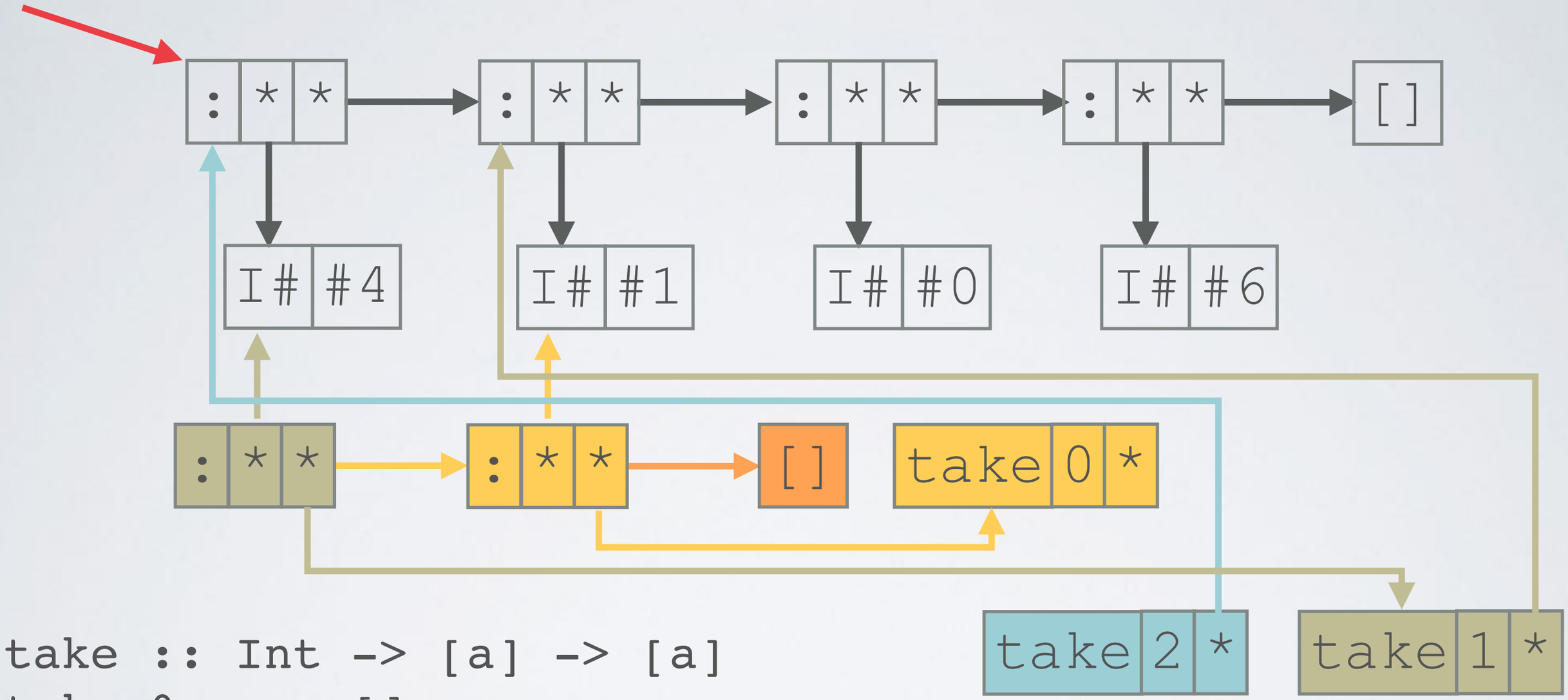
```
data [a] = [] | a : [a]
[1] -- 1 : []
z = [4,1,0,6] :: [Int] -- 4 : 1 : 0 : 6 : []
```




```
tail :: [a] -> [a]
tail (x:xs) = xs
tail _   = error “…”

tail z -- 1 : 0 : 6 : []
```

列表

```
z = [4,1,0,6] :: [Int] -- 4 : 1 : 0 : 6 : []
```

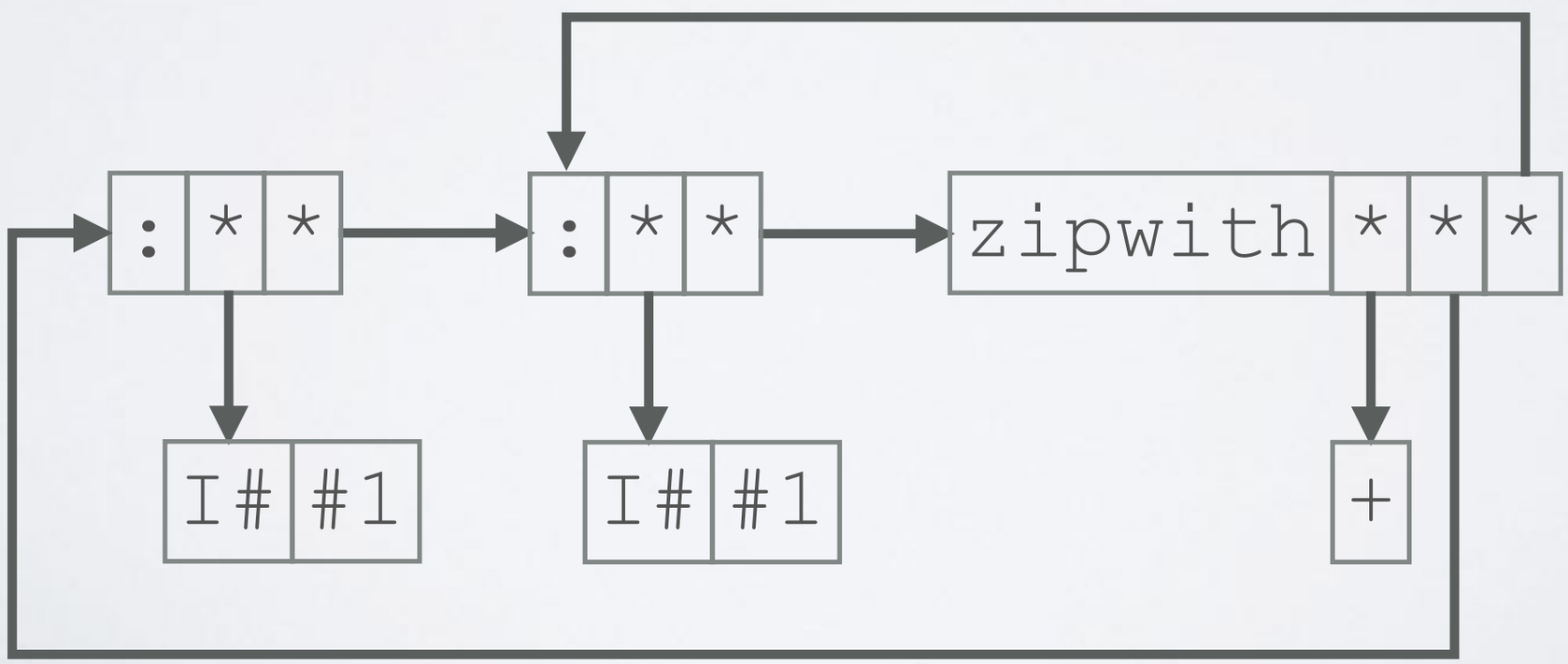



```
take :: Int -> [a] -> [a]
take 0 xs = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs

take 2 z -- 4 : 1 : []
```

FIBONACCI

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith _ [] _ = []
zipWith _ _ [] = []

zipWith (-) [10, 9 ,8] [3, 4] -- [7, 5]
```

```
xs = 1 : 1 : zipWith (+) xs (tail xs)
```



FIBONACCI

```
xs = 1 : 1 : zipWith (+) xs (tail xs)
take 5 xs -- [1,1,2,3,5]
```