

Lazy evaluation illustrated

for Haskell divers

exploring some mental models and implementations

Takenobu T.

Lazy,... zzz

..., It's fun!

NOTE

- Meaning of terms are different by communities.
- There are a lot of good documents. Please see also references.
- This is written for GHC's Haskell.

Contents

1. Introduction

- Basic mental models
- Lazy evaluation
- Simple questions

2. Expressions

- Expression and value
- Expressions in Haskell
- Classification by values and forms
- WHNF

3. Internal representation of expressions

- Constructor
- Thunk
- Uniform representation
- WHNF
- let, case expression

4. Evaluation

- Evaluation strategies
- Evaluation in Haskell (GHC)
- Examples of evaluation steps
- Examples of evaluations
- Controlling the evaluation

5. Implementation of evaluator

- Lazy graph reduction
- STG-machine

6. Semantics

- Bottom
- Strict/Non-strict
- Lifted and boxed types
- Strictness analysis
- Sequential order

7. Appendix

- References

1. Introduction

1. Introduction

Basic mental models

How to evaluate a program in your brain ?

a program

```
code  
code  
code  
:  
?
```

How to evaluate (execute, reduce) the program in your brain?

What "mental model" do you have?

One of the mental models for C program

C program

A sequence of statements

```
main (...) {  
    code..  
    code.. } ?  
    code..  
    code..  
    code.. }
```

A nested structure

```
x = func1( func2( a ) );  
_____ ?
```

A sequence of arguments

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

A function and arguments

```
z = func1( m + n );  
_____ ? _____
```

How to evaluate (execute, reduce) the program in your brain?

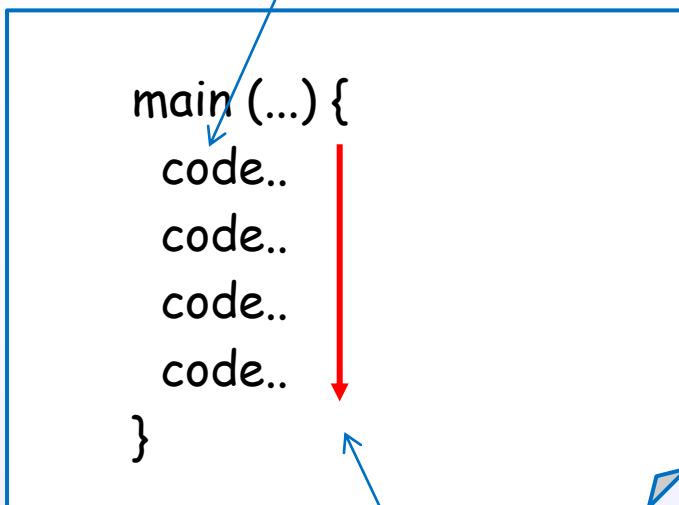
What step, what order, ... ?

One of the mental models for C program

C program

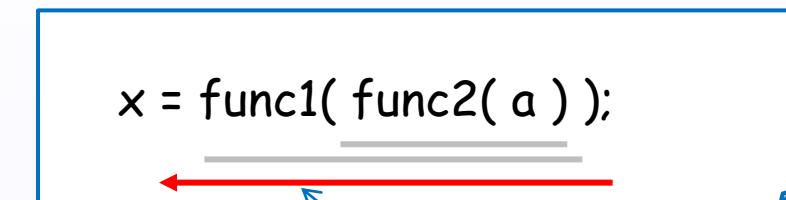
A sequence of statements

A program is a collection of statements.

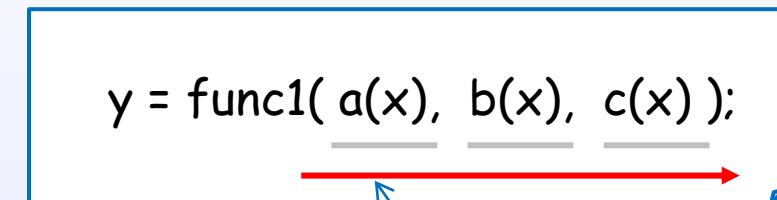


Statements are executed downward.

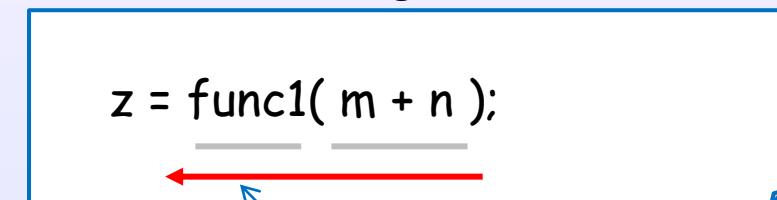
A nested structure



A sequence of arguments



A function and arguments



arguments first
apply second

Each programmer has some mental models in their brain.

One of the mental models for C program

Maybe, You have some implicit mental model in your brain for C program.

(1) A program is **a collection of statements**.

(2) There is the **order** between evaluations of elements.



(3) There is the **order** between termination and start of evaluations.



This is a **syntactically straightforward** model for programming languages.
(an implicit sequential order model)

One of the mental models for Haskell program

Haskell program

```
main = expaa (expab expac expad)
```

```
expac = expaca expacb
```

```
expad = expada expadb expadc
```

```
:
```



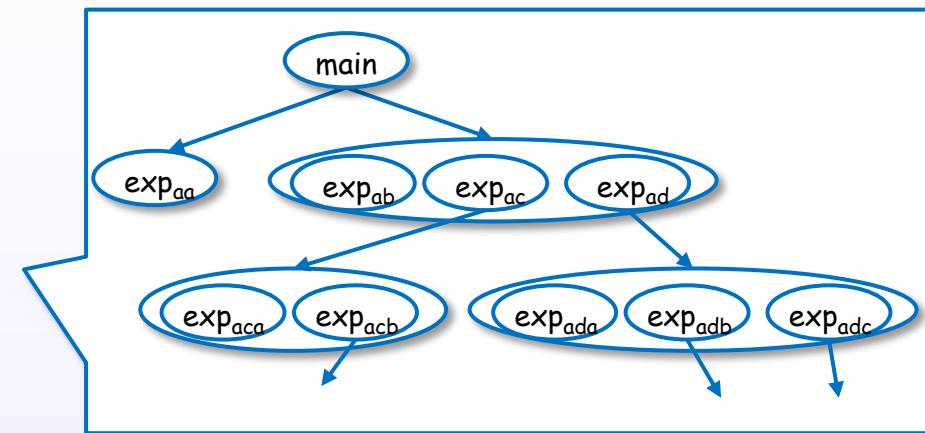
How to evaluate (execute, reduce) the program in your brain?
What step, what order, ... ?

One of the mental models for Haskell program

Haskell program

A program is a collection of expressions.

```
main = expaa (expab expac expad)
          ^_____
          |_____
          |_____
expac = expaca expacb
          _____
          |_____
expad = expada expadb expadc
          _____
          |_____
          :
          :
```



```
main = expaa (expab (expaca expacb) (expada expadb expadc))
```

A entire program is regarded as a single expression.

The subexpression is evaluated (reduced) in some order.

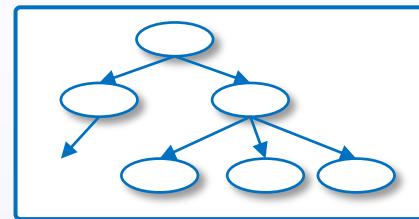
The evaluation is performed by replacement.

One of the mental models for Haskell program

(1) A program is a collection of expressions.

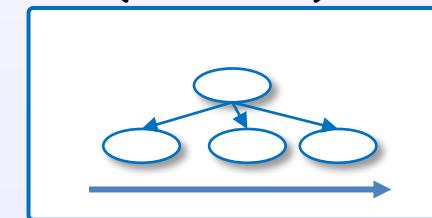
(2) A entire program is regarded as a single expression.

```
main = e (e (e (e e) e (e e e)) )
```

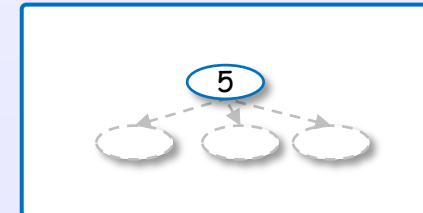
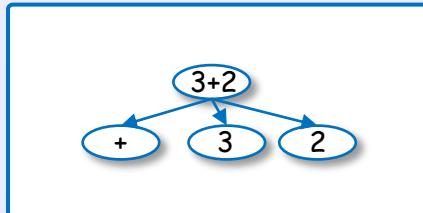


(3) The subexpressions are evaluated (reduced) in some order.

```
f = e (e (e (e e) e (e e e)) )
```



(4) The evaluation is performed by replacement.



This is an example of an expression reduction model for Haskell.

1. Introduction

Lazy evaluation

Why lazy evaluation?

To avoid unnecessary computation

To manipulate infinite data structures

modularity

To manipulate streams

abstraction

pure is order free

amortizing

To manipulate huge data structures

potentially parallelism

2nd Church-Rosser theorem

out-of-order optimization

To implement non-strict semantics

asynchronization

fun

reactive

...

There are various reasons ☺

Haskell(GHC) 's lazy evaluation

Lazy evaluation

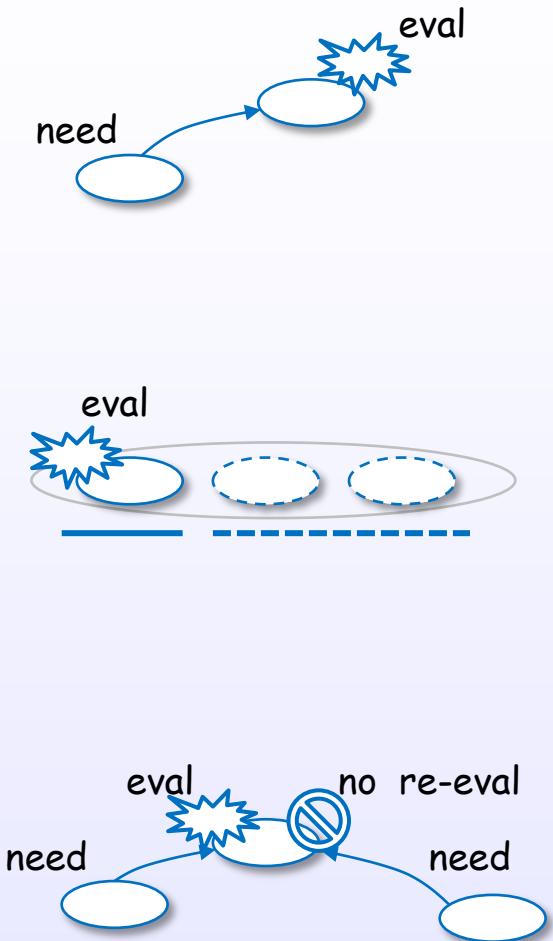
evaluate **only when needed**

+

evaluate **only enough**

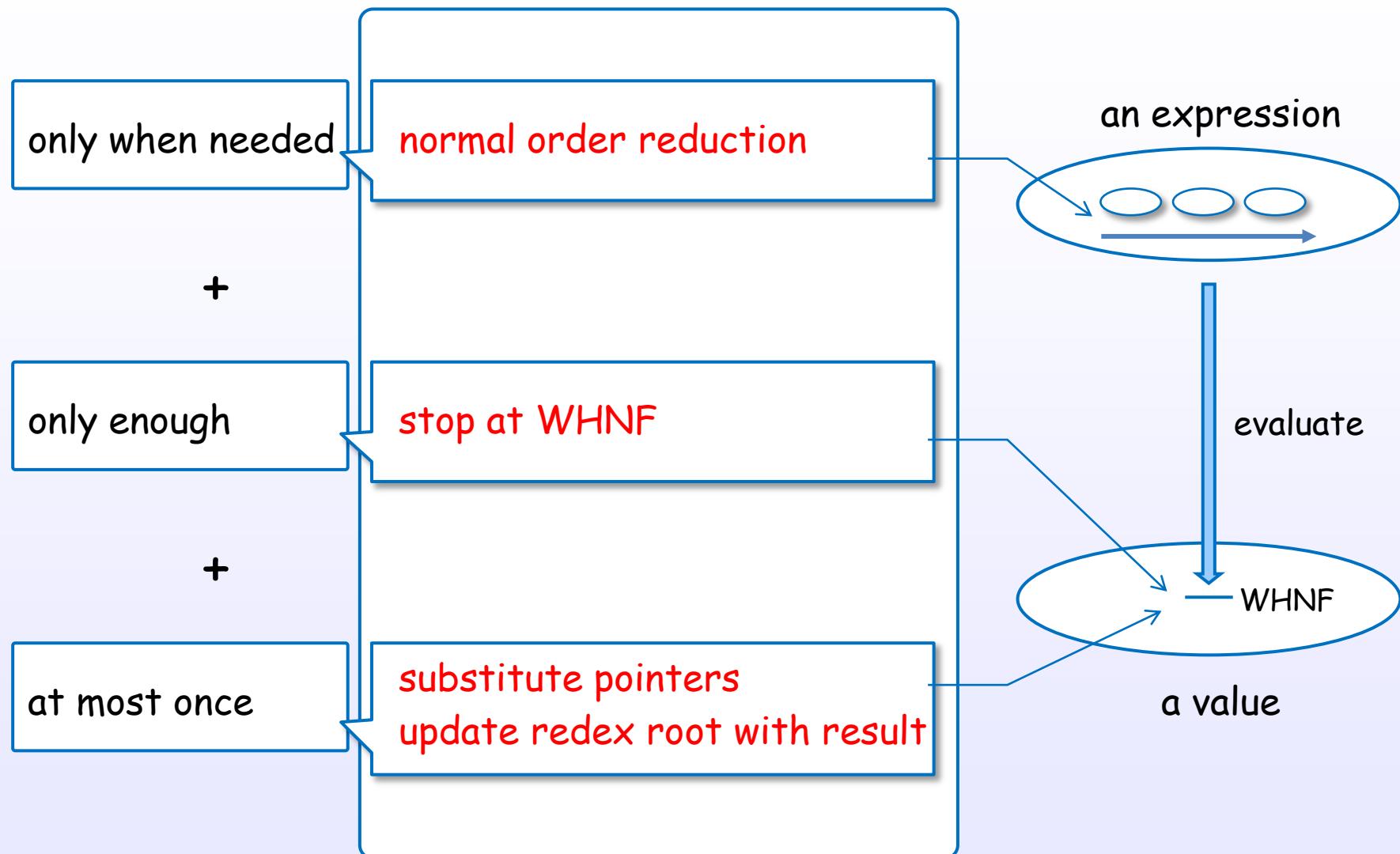
+

evaluate **at most once**



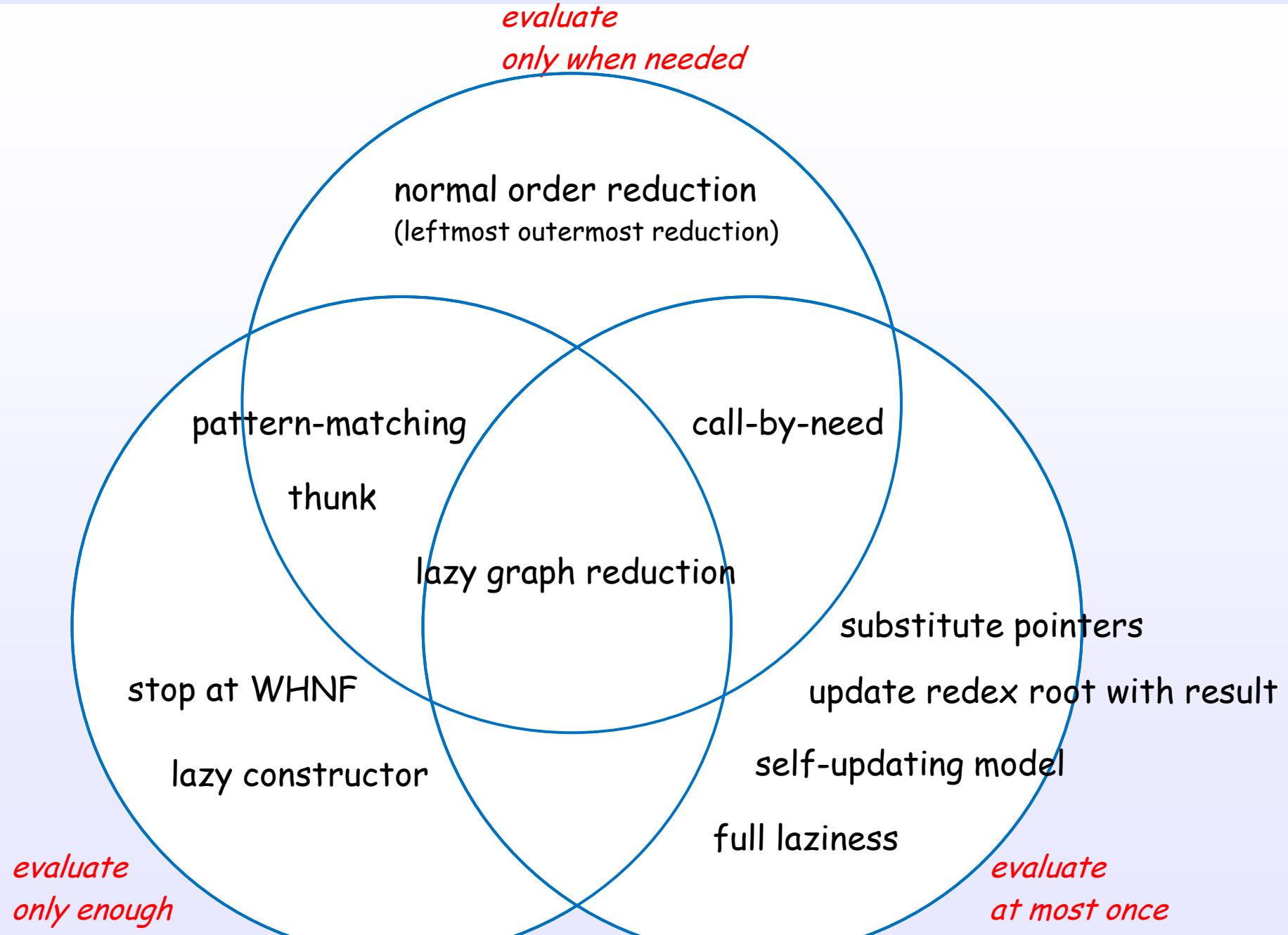
"Lazy" is "**delay** and **avoidance**" rather than "delay".

Ingredient of Haskell(GHC) 's lazy evaluation



This strategy is implemented by lazy graph reduction.

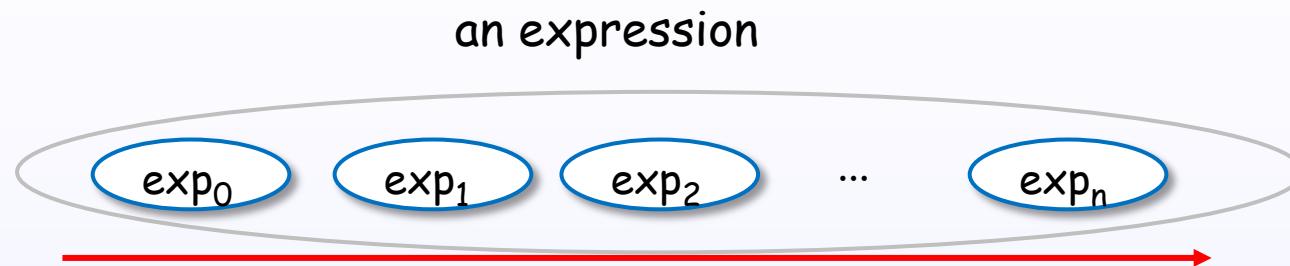
Techniques of Haskell(GHC) 's lazy evaluation



1. Introduction

Simple questions

What order?



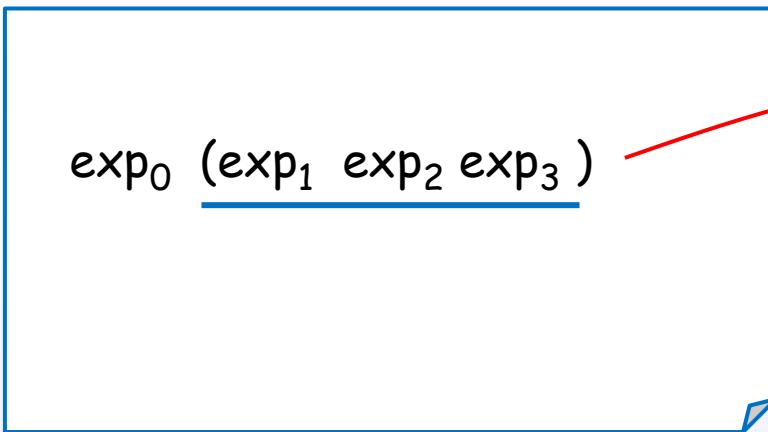
An expression is evaluated by normal order (leftmost outermost redex first).

Normal order reduction guarantees to find a normal form (if one exists).

To avoid unnecessary computation, normal order reduction chooses to apply the function rather than first evaluating the argument.

How to postpone?

Haskell code



build/allocate

heap memory

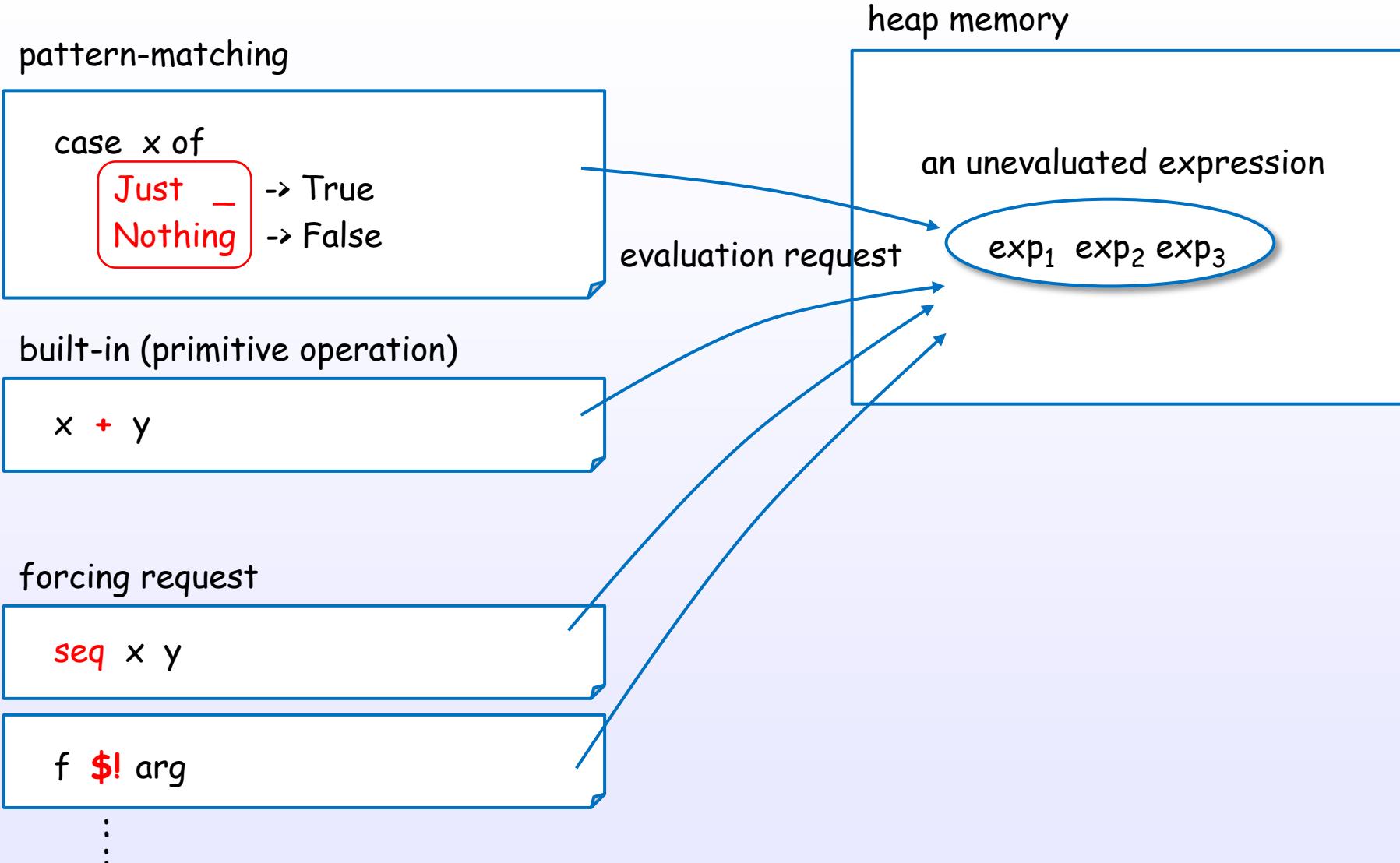
an unevaluated expression

`exp1 exp2 exp3`

thunk

To postpone the evaluation, an unevaluated expression is built in the heap memory.

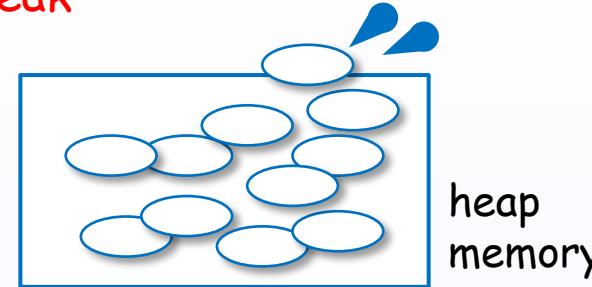
When needed?



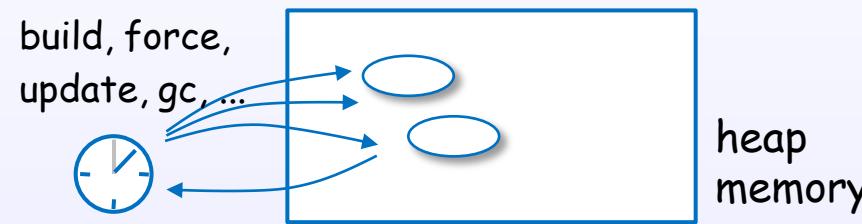
Pattern-matching or forcing request drive the evaluation.

What to be careful about?

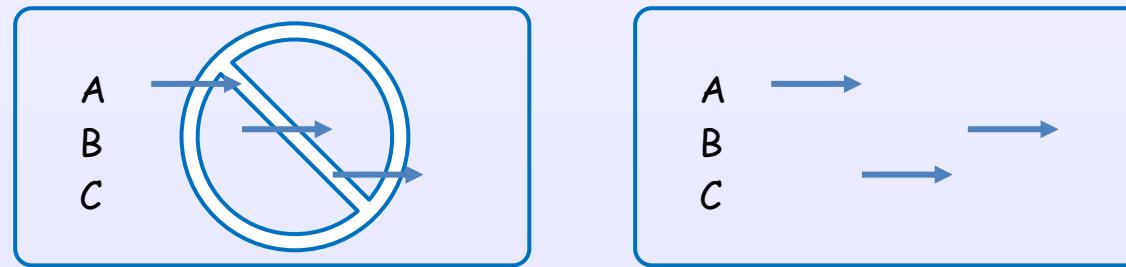
To consider hidden **space leak**



To consider **performance cost** to postpone unevaluated expressions



To consider evaluation (execution) **order** and **timing** in real world



You can avoid the pitfalls by controlling the evaluation.

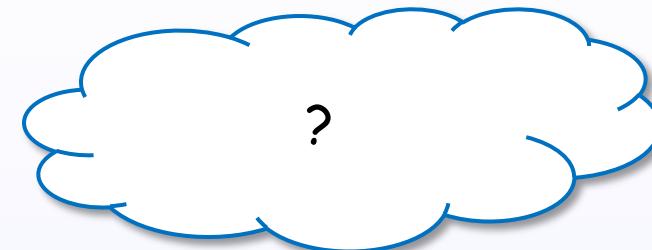
2. Expressions

2. Expressions

Expression and value

What is an expression?

An expression

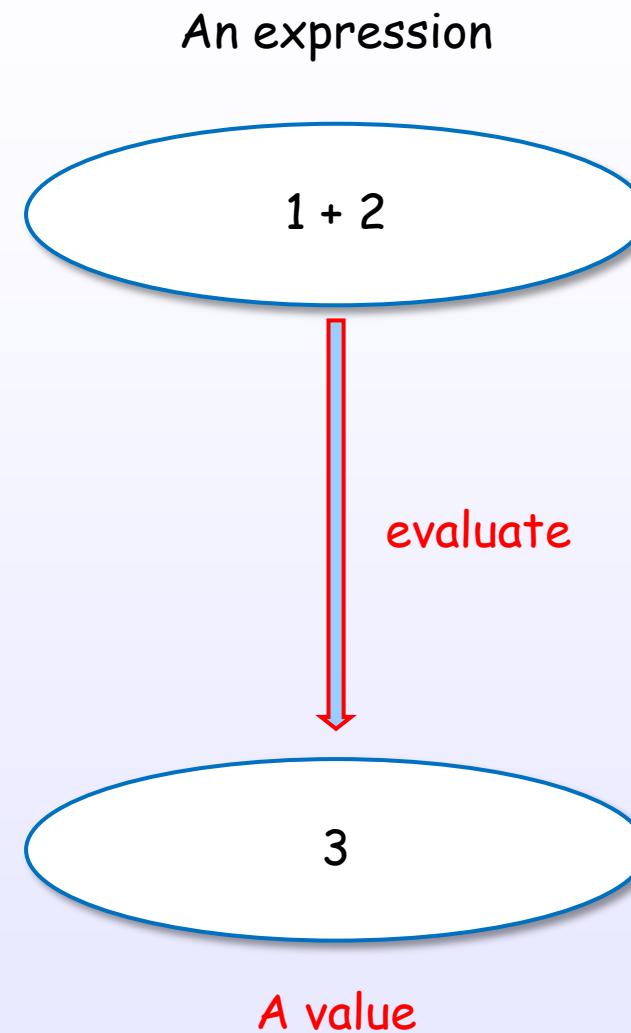


An expression denotes a value

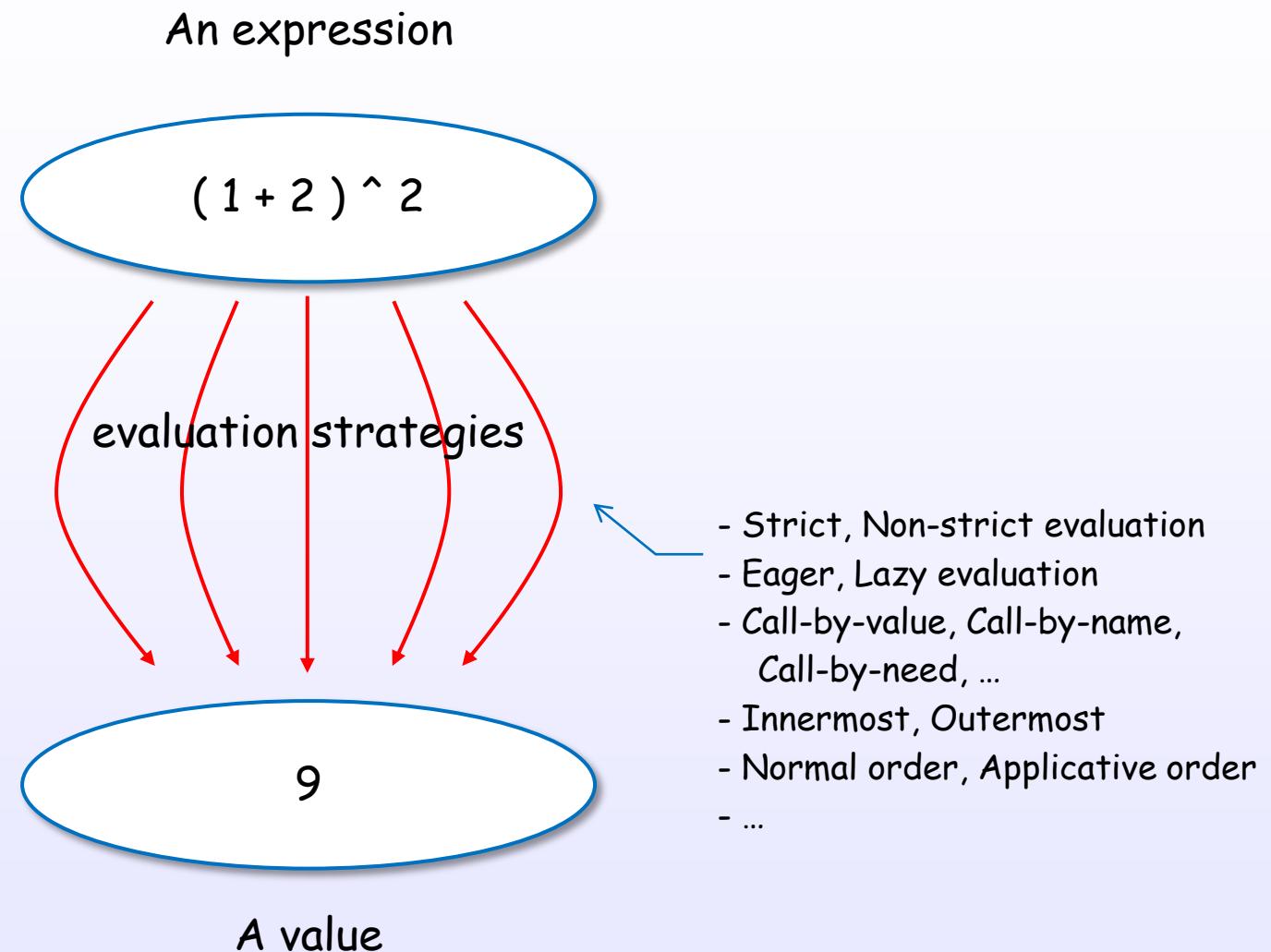
An expression

$$1 + 2$$

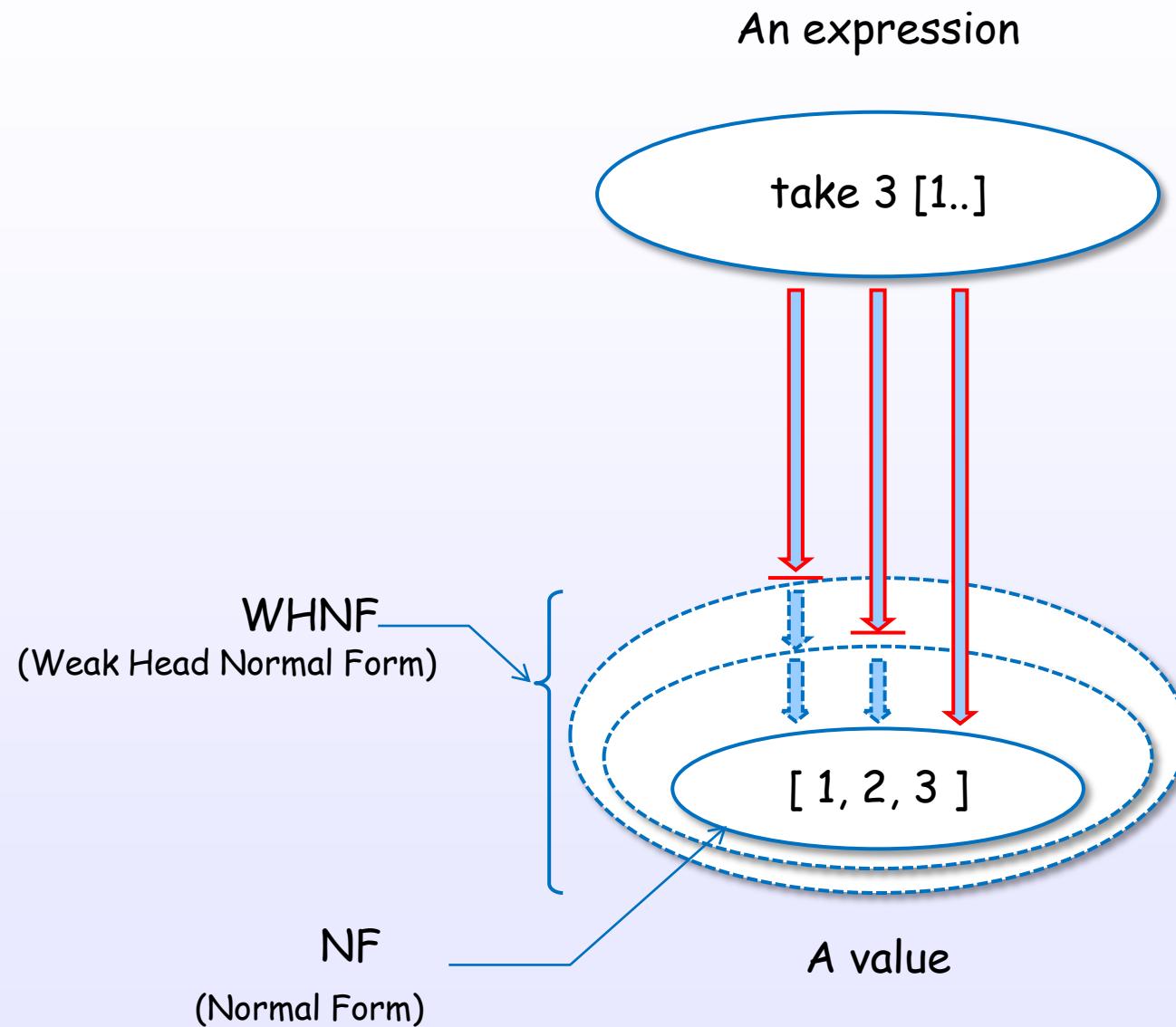
An expression evaluates to a value



There are many evaluation approaches



There are some evaluation levels

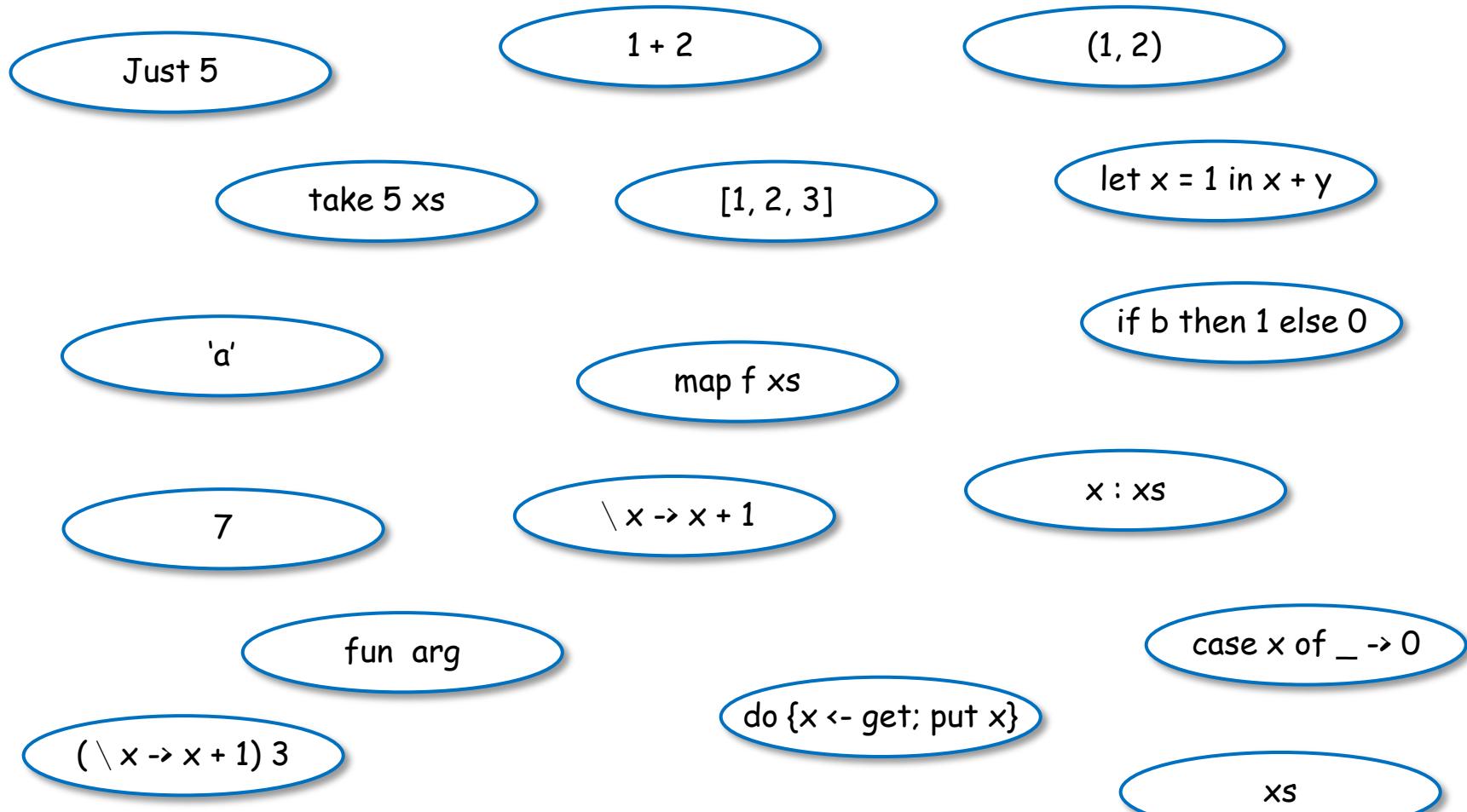


2. Expressions

Expressions in Haskell

There are many expressions in Haskell

Expressions



categorizing

Expression categories in Haskell

lambda abstraction

 $\lambda x \rightarrow x + 1$

let expression

 $\text{let } x = 1 \text{ in } x + y$

conditional

 $\text{if } b \text{ then } 1 \text{ else } 0$

case expression

 $\text{case } x \text{ of } _ \rightarrow 0$

do expression

 $\text{do } \{x \leftarrow \text{get}; \text{put } x\}$

function application

 $\text{take } 5 \text{ xs}$
 $(\lambda x \rightarrow x + 1) 3$
 $1 + 2$
 $\text{map } f \text{ xs}$

fun arg

general constructor, literal and some forms

 7
 $[1, 2, 3]$
 $(1, 2)$
 $'a'$
 $x : xs$
 $\text{Just } 5$

variable

 xs

Specification is defined in Haskell 2010 Language Report

"Haskell 2010 Language Report, Chapter 3 Expressions" [H1]

<i>exp</i>	\rightarrow	<i>infixexp</i> :: [<i>context</i> =>] <i>type</i>	(expression type signature)
		<i>infixexp</i>	
<i>infixexp</i>	\rightarrow	<i>lexp qop infixexp</i>	(infix operator application)
		<i>- infixexp</i>	(prefix negation)
		<i>lexp</i>	
<i>lexp</i>	\rightarrow	$\lambda \ a_1 \dots \ a_n \rightarrow \ exp$	(lambda abstraction, $n \geq 1$)
		<i>let decls in exp</i>	(let expression)
		<i>if exp [;] then exp [;] else exp</i>	(conditional)
		<i>case exp of { alts }</i>	(case expression)
		<i>do { stmts }</i>	(do expression)
		<i>fexp</i>	
<i>fexp</i>	\rightarrow	<i>[fexp] aexp</i>	(function application)
<i>aexp</i>	\rightarrow	<i>qvar</i>	(variable)
		<i>gcon</i>	(general constructor)
		<i>literal</i>	
		<i>(exp)</i>	(parenthesized expression)
		<i>(exp₁ , ... , exp_k)</i>	(tuple, $k \geq 2$)
		<i>[exp₁ , ... , exp_k]</i>	(list, $k \geq 1$)
		<i>[exp₁ , exp₂] ... [exp₃]</i>	(arithmetic sequence)
		<i>[exp qual₁ , ... , qual_n]</i>	(list comprehension, $n \geq 1$)
		<i>(infixexp qop)</i>	(left section)
		<i>(qop(-) infixexp)</i>	(right section)
		<i>qcon { fbind₁ , ... , fbind_n }</i>	(labeled construction, $n \geq 0$)
		<i>aexp_(qcon) { fbind₁ , ... , fbind_n }</i>	(labeled update, $n \geq 1$)

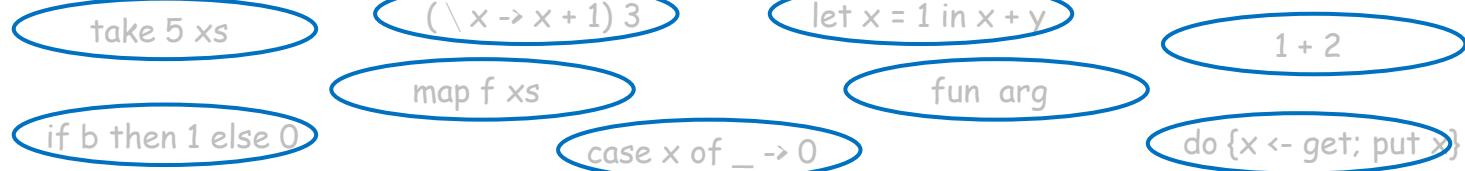
2. Expressions

Classification by values and forms

Classification by values

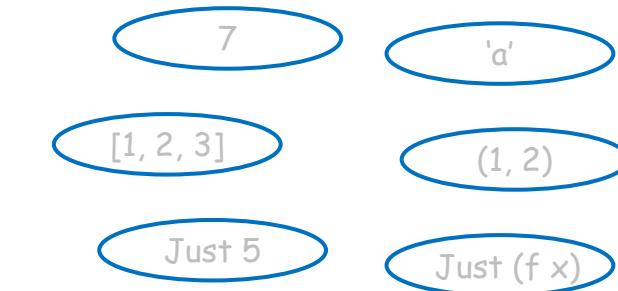
Expressions

unevaluated expressions



values

data values



function values



bottom

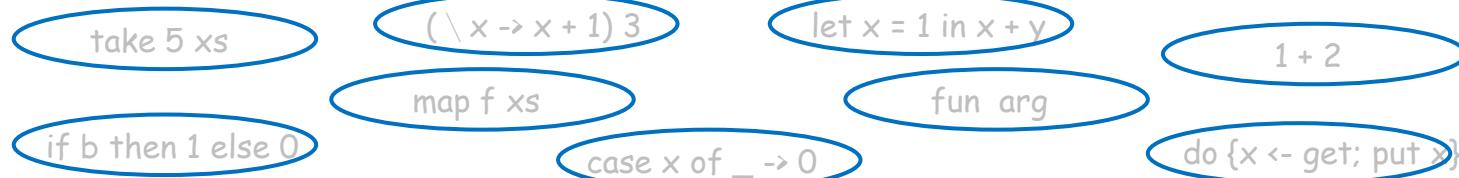


Values are data values or function values.

Classification by forms

Expressions

unevaluated expressions



values

WHNF

HNF

NF

$\lambda x \rightarrow \text{abs } 1$

$\lambda x \rightarrow x + (\text{abs } 1)$

$\lambda x \rightarrow x$

Just (f x)

[f x, g y]

7

'a'

[1, 2, 3]

(1, 2)

Just 5

bottom

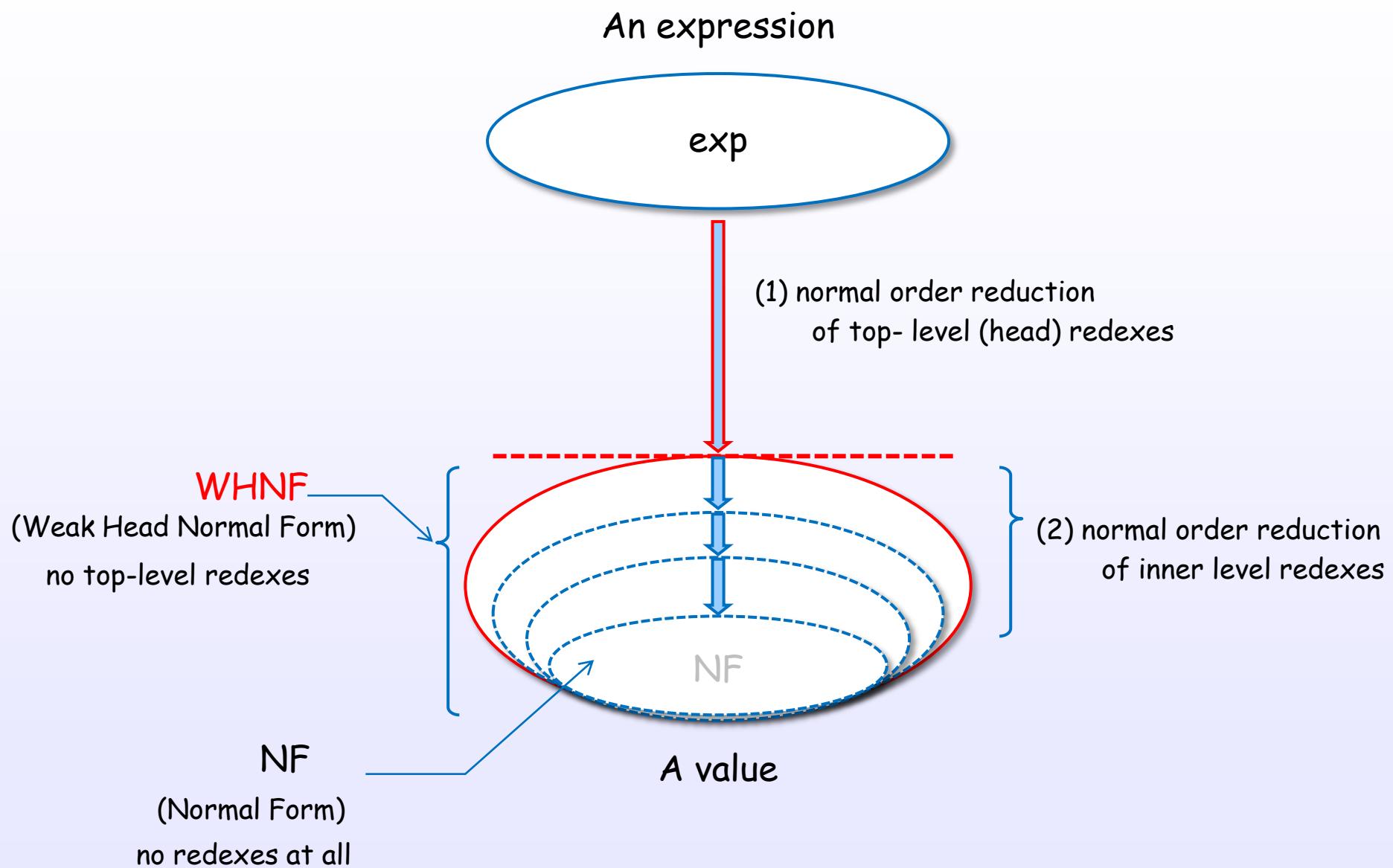
\perp

Values are WHNF, HNF or NF.

2. Expressions

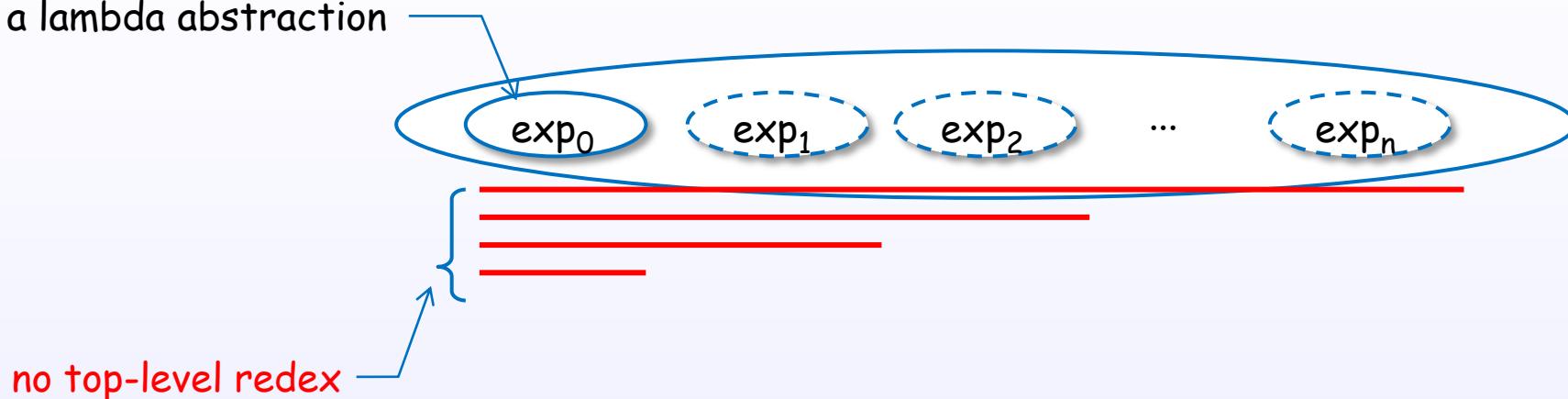
WHNF

WHNF is one of the form for the evaluated values



WHNF

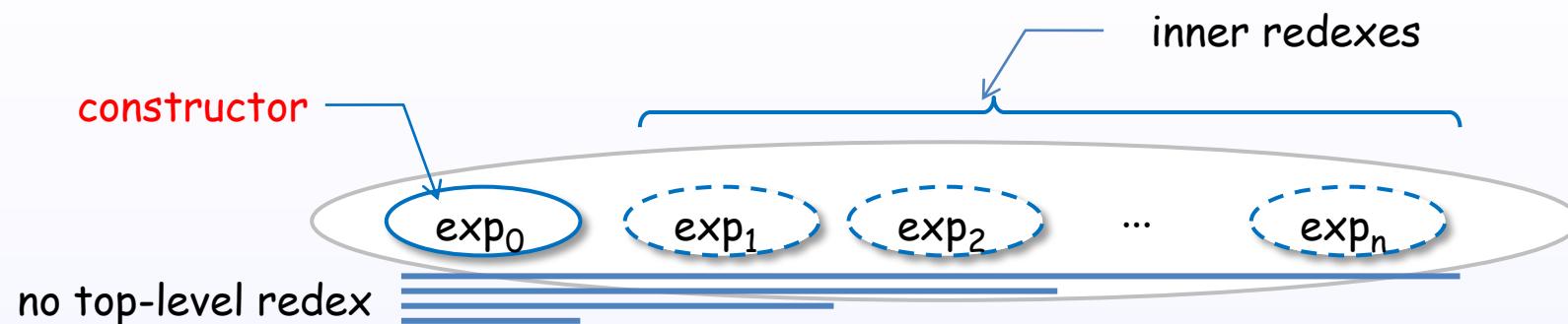
top-level (head) is
a constructor or
a lambda abstraction



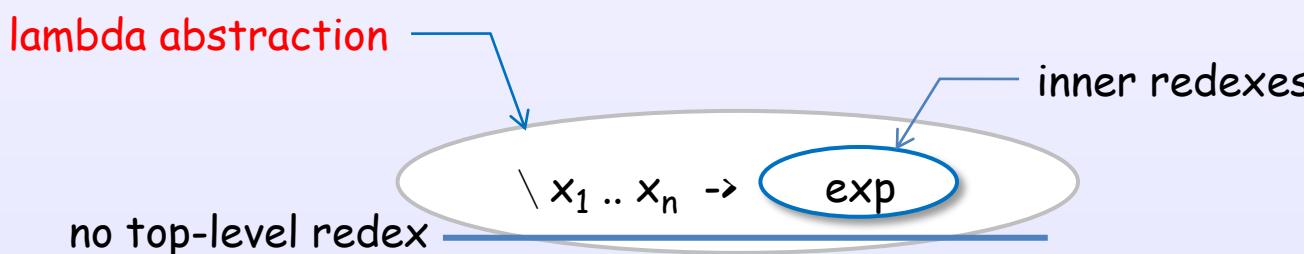
WHNF is a value which has evaluated top-level

WHNF for a data value and a function value

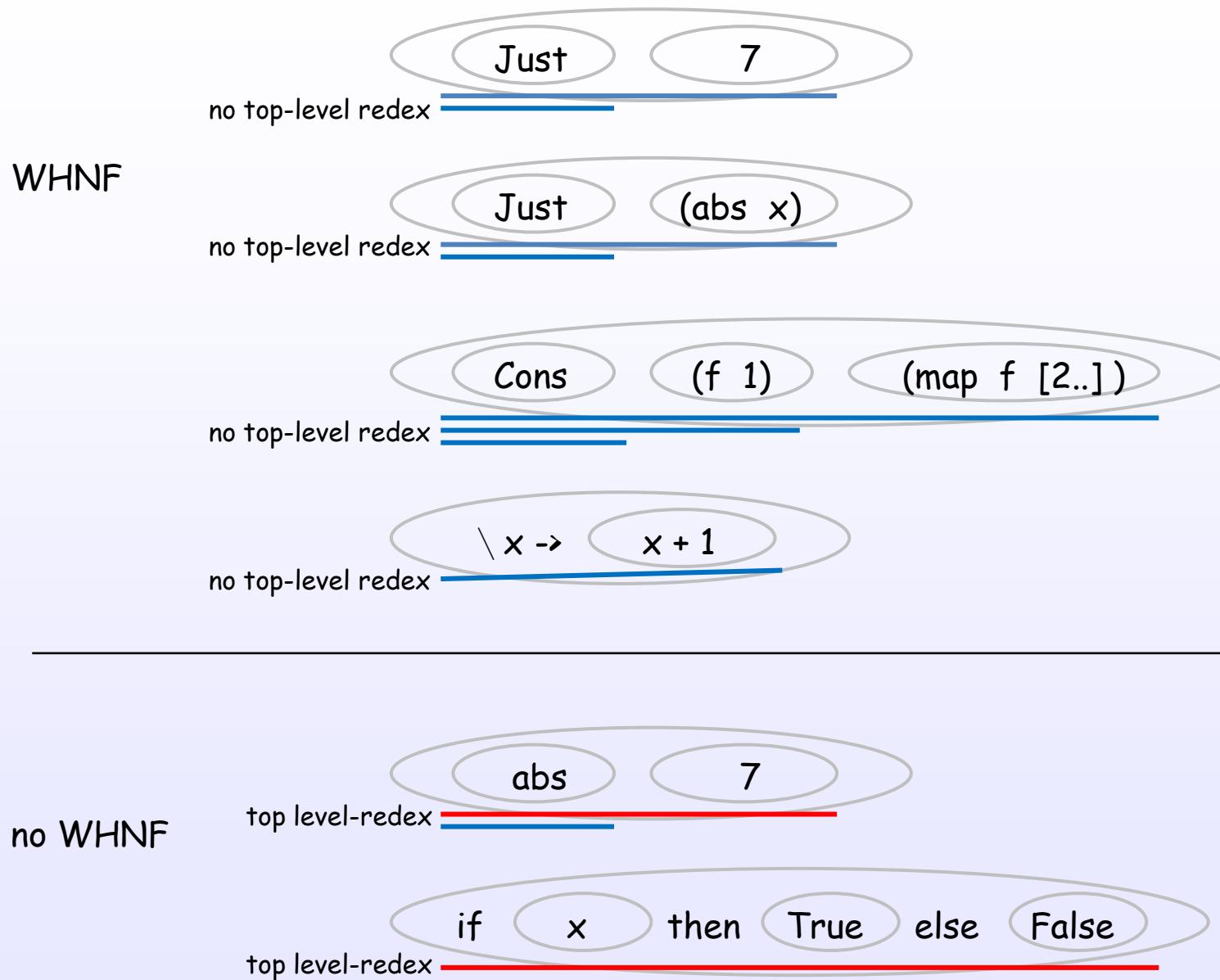
a data value in WHNF



a function value in WHNF

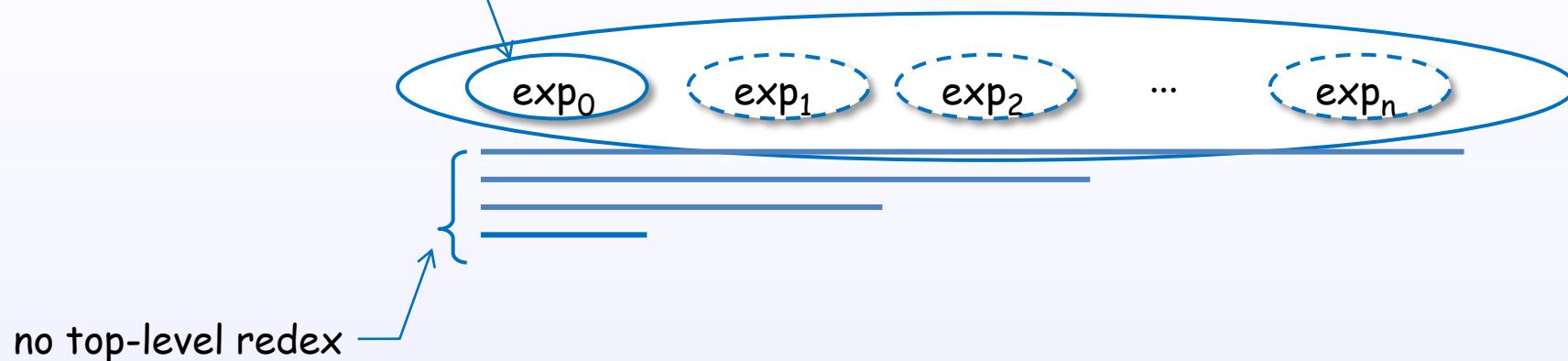


Examples of WHNF



HNF

top-level (head) is
a constructor or
a lambda abstraction with no top-level redex

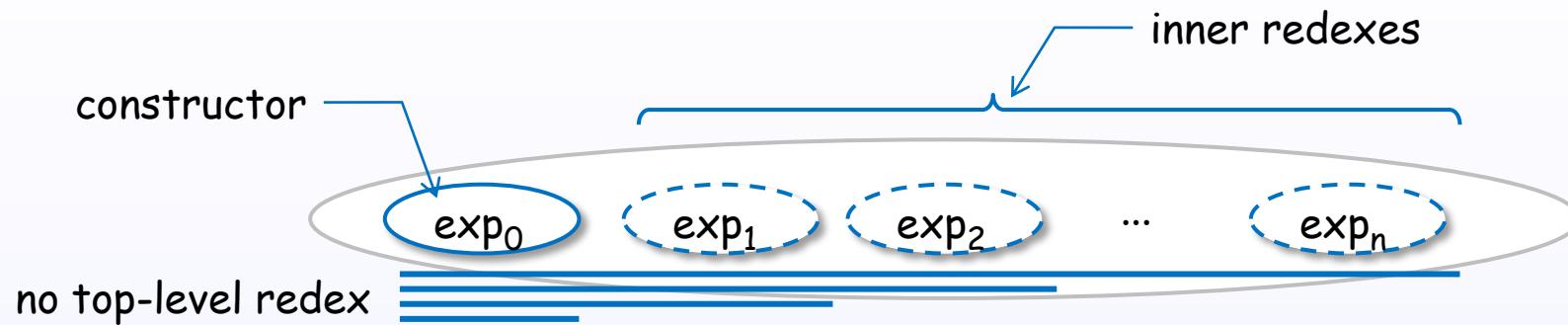


HNF is a value which has evaluated top-level

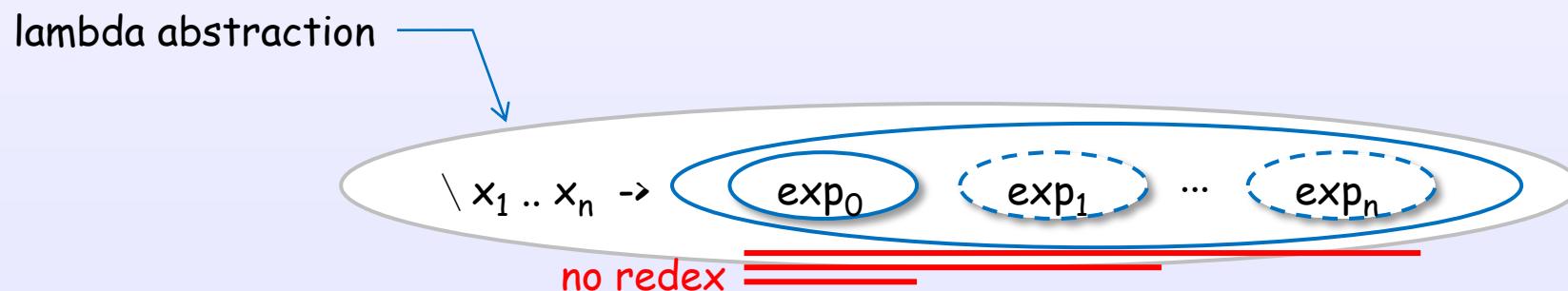
* GHC uses WHNF rather than HNF.

HNF for a data value and a function value

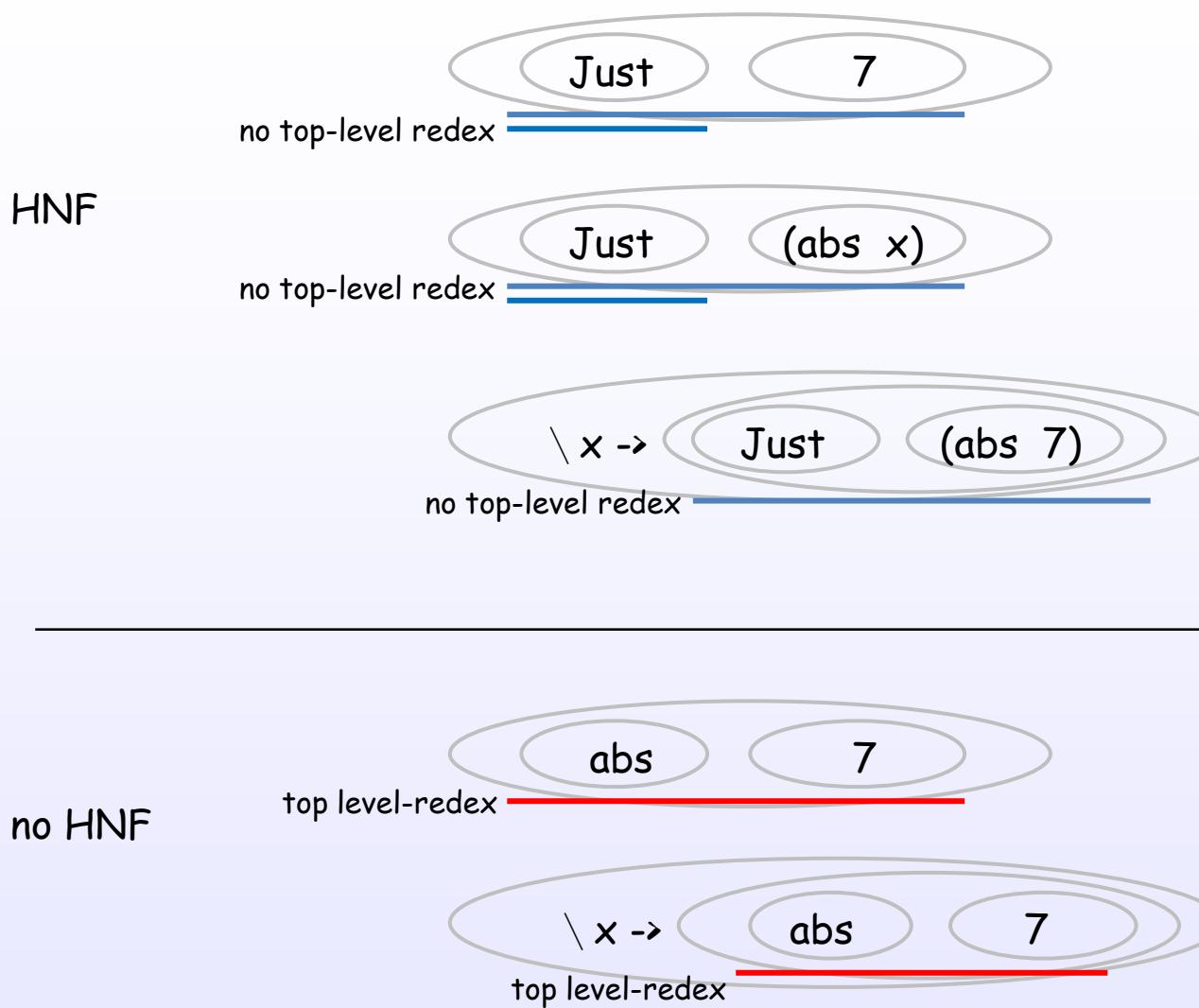
a data value in HNF (same as WHNF)



a function value in HNF

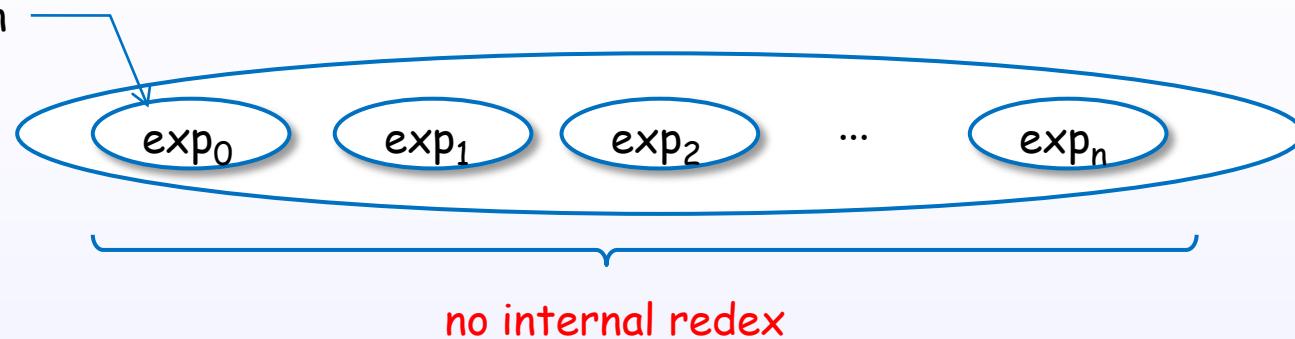


Examples of HNF



NF

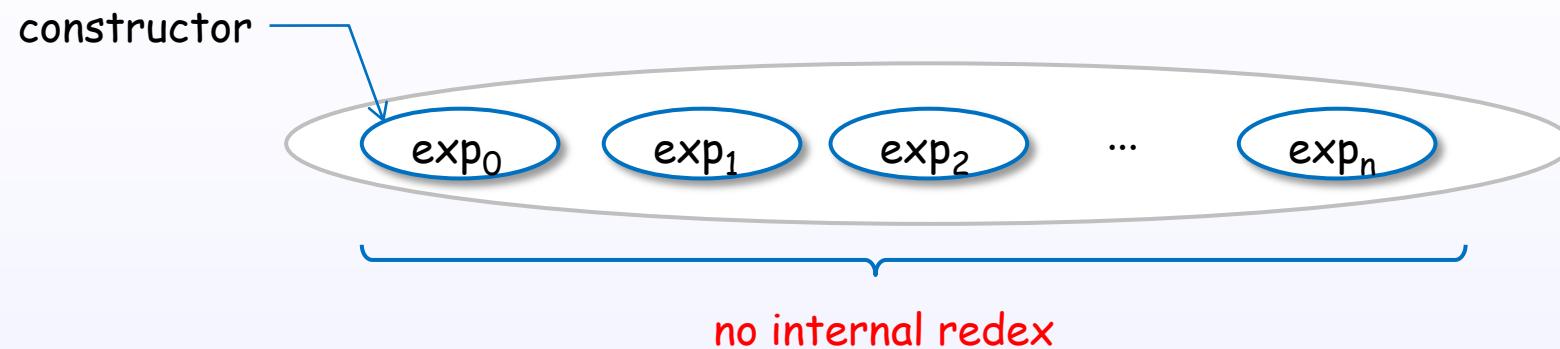
top-level (head) is
a constructor or
a lambda abstraction



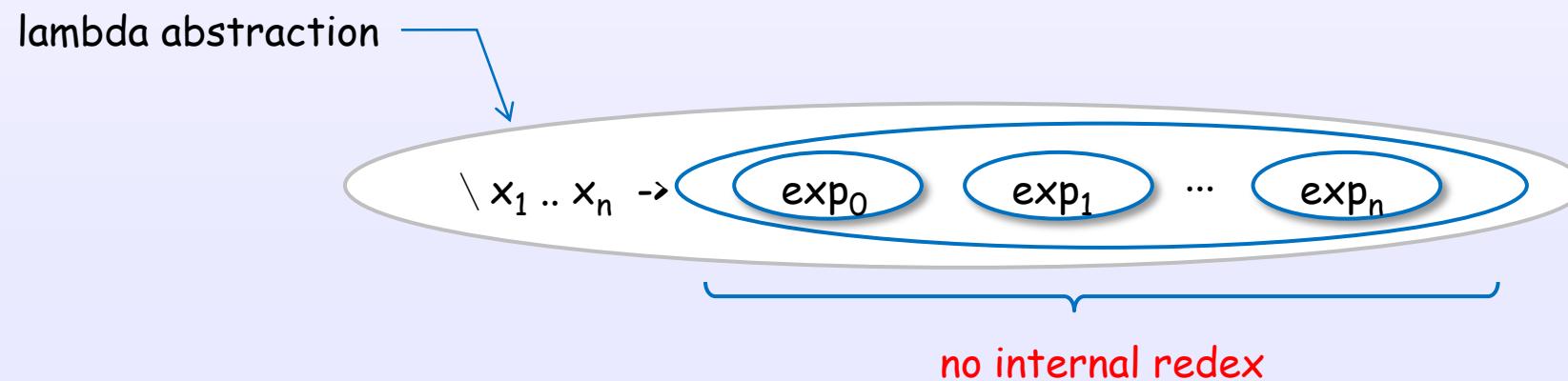
NF is a value which has no redex.

NF for a data value and a function value

a data value in NF

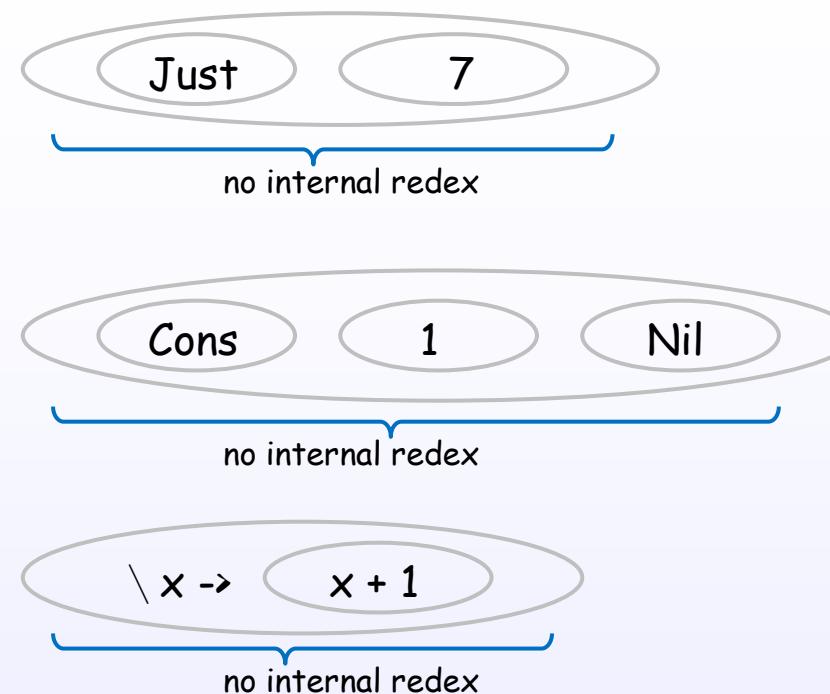


a function value in NF

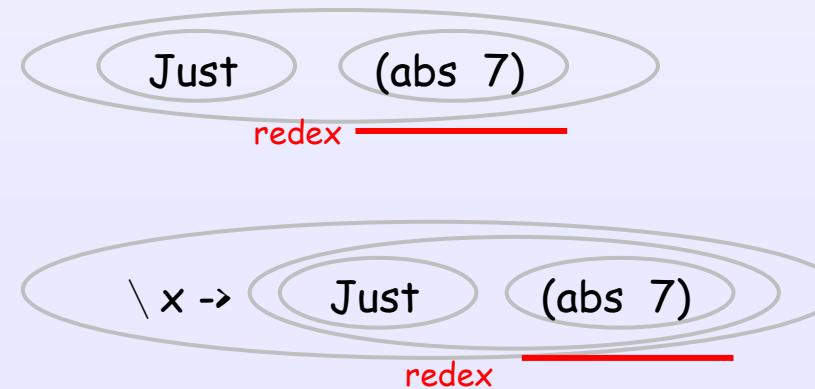


Examples of NF

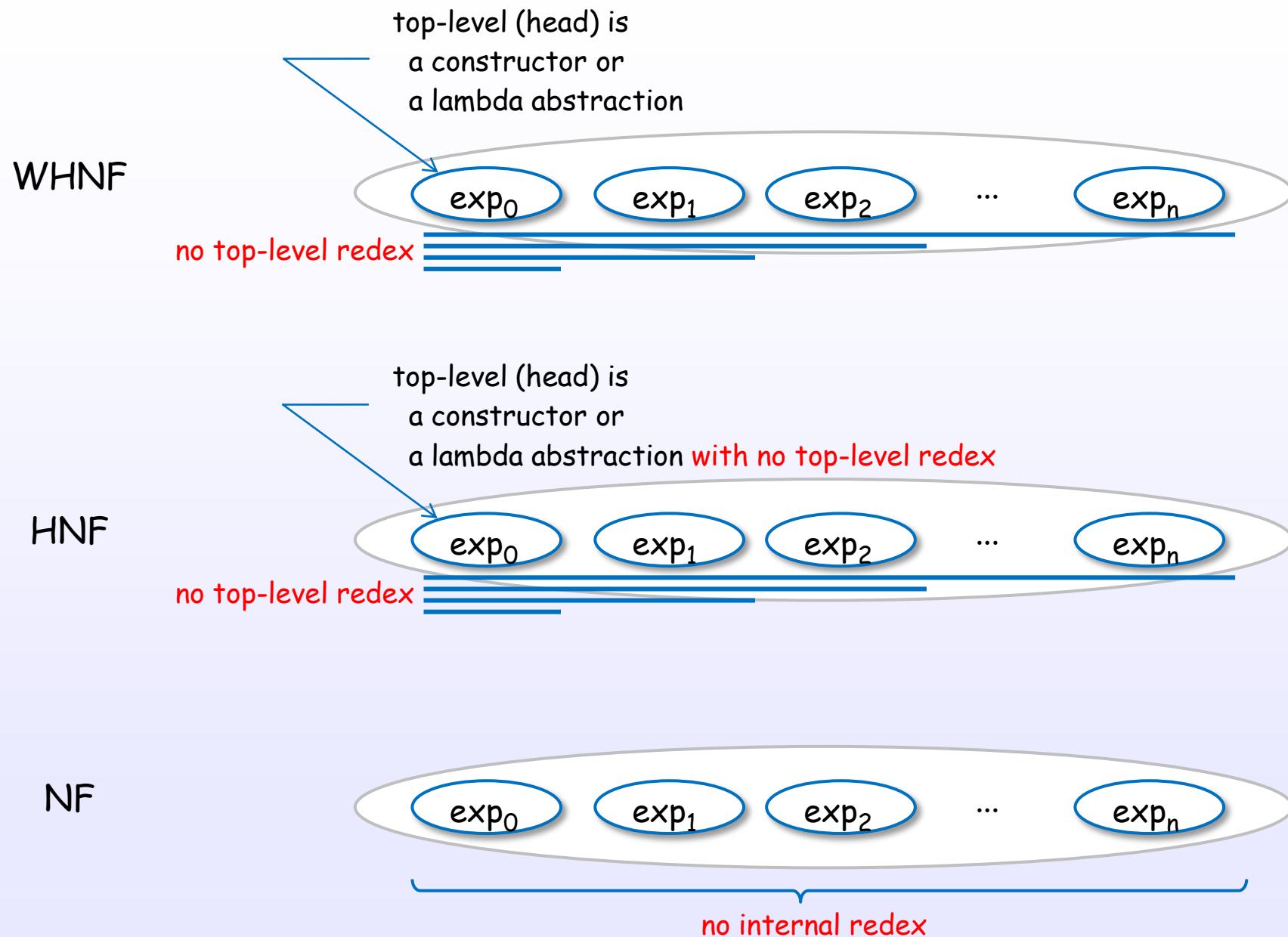
NF



no NF



WHNF, HNF, NF



Definition of WHNF and HNF

"The implementation of functional programming languages" [H4]

11.3.1 Weak Head Normal Form

To express this idea precisely we need to introduce a new definition:

DEFINITION

A lambda expression is in *weak head normal form* (WHNF) if and only if it is of the form

$F E_1 E_2 \dots E_n$

where $n \geq 0$;

and either F is a variable or data object

or F is a lambda abstraction or built-in function

and $(F E_1 E_2 \dots E_m)$ is not a redex for any $m \leq n$.

An expression has no *top-level redex* if and only if it is in weak head normal form.

DEFINITION

A lambda expression is in *head normal form* (HNF) if and only if it is of the form

$\lambda x_1. \lambda x_2. \dots \lambda x_n. (v M_1 M_2 \dots M_m)$

where $n, m \geq 0$;

v is a variable (x_i), a data object, or a built-in function;

and $(v M_1 M_2 \dots M_p)$ is not a redex for any $p \leq m$.

3. Internal representation of expressions

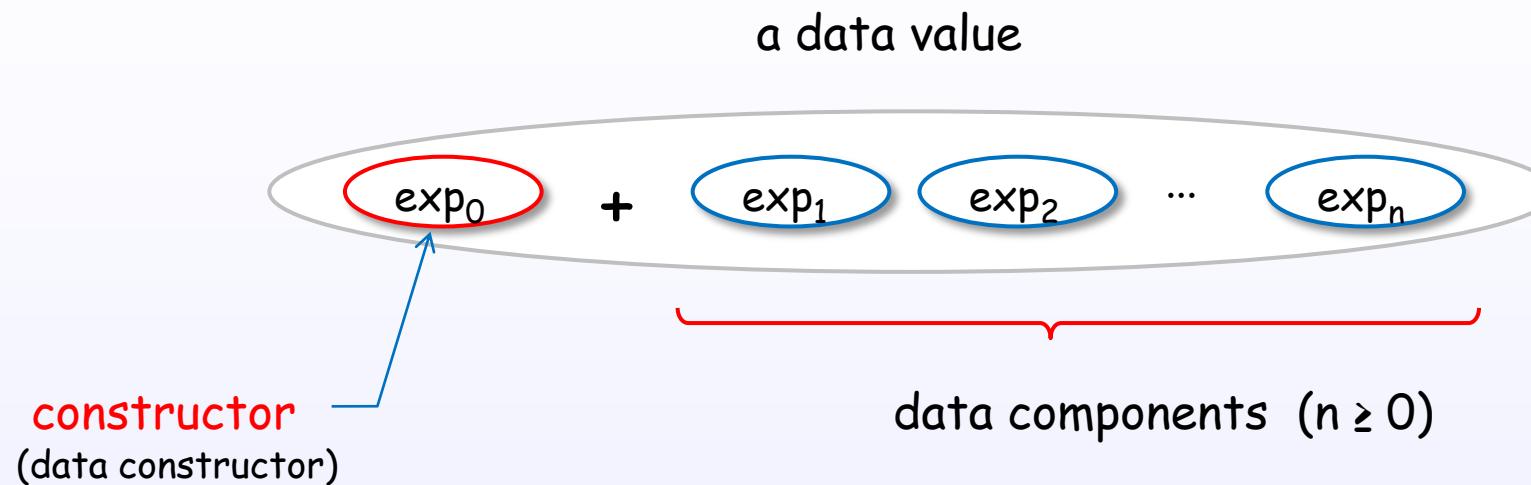
3. Internal representation of expressions

Constructor

Constructor

Constructor is one of the key elements
to understand WHNF and lazy evaluation in Haskell.

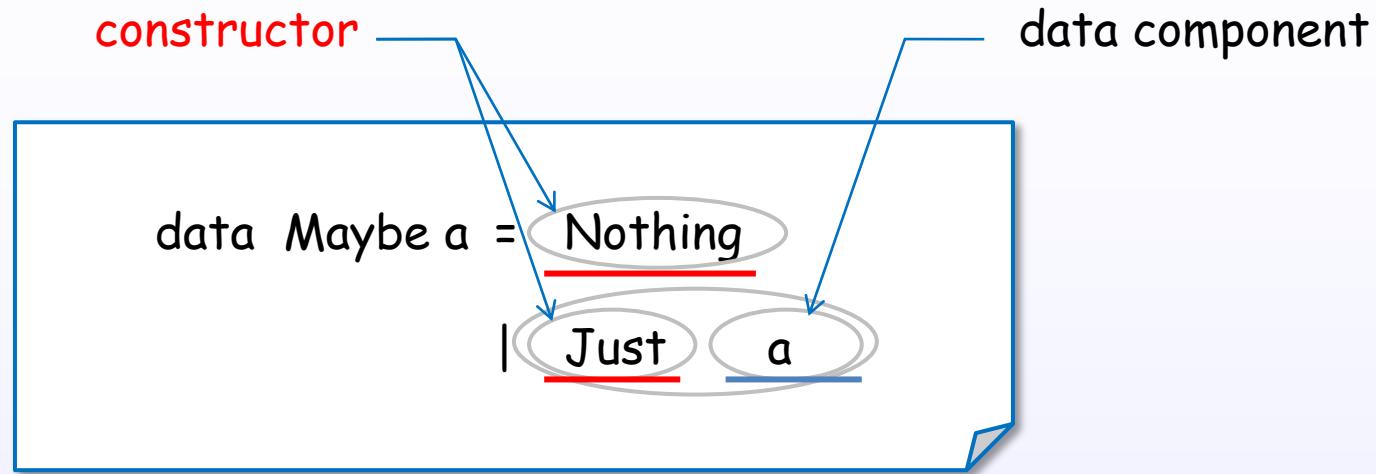
Constructor



A constructor builds a structured data value.

A constructor distinguishes the data value in expressions.

Constructors and data declaration



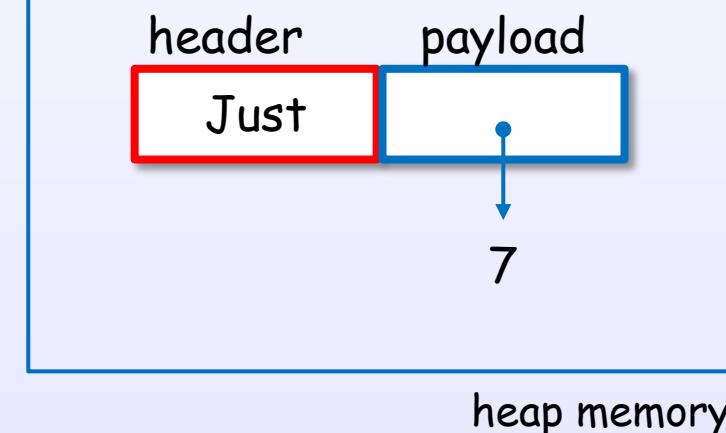
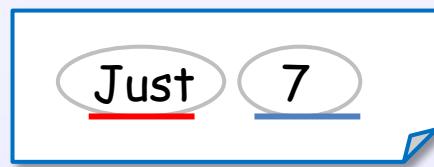
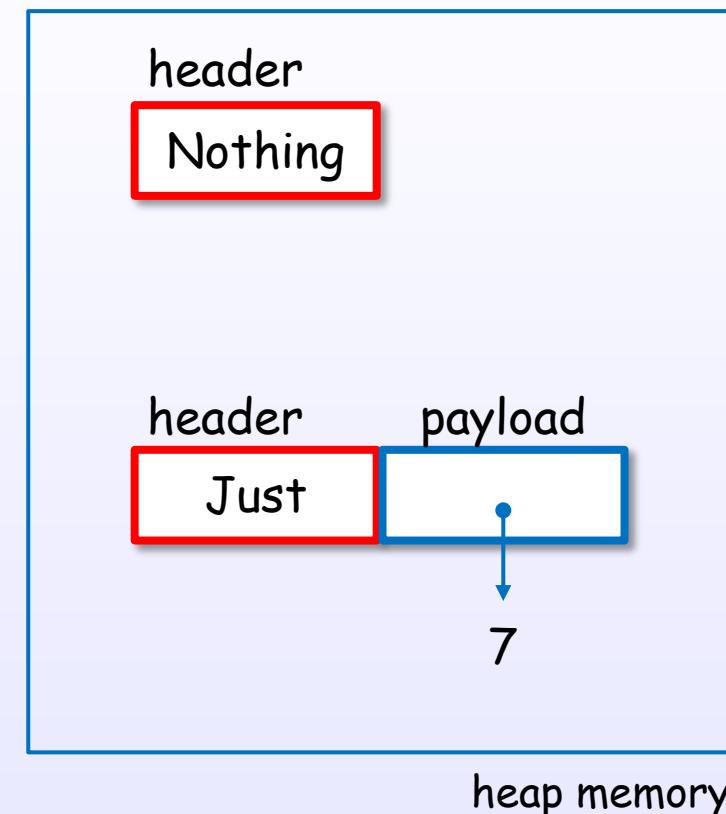
Constructors are defined by data declaration.

Internal representation of Constructors for data values

Haskell code

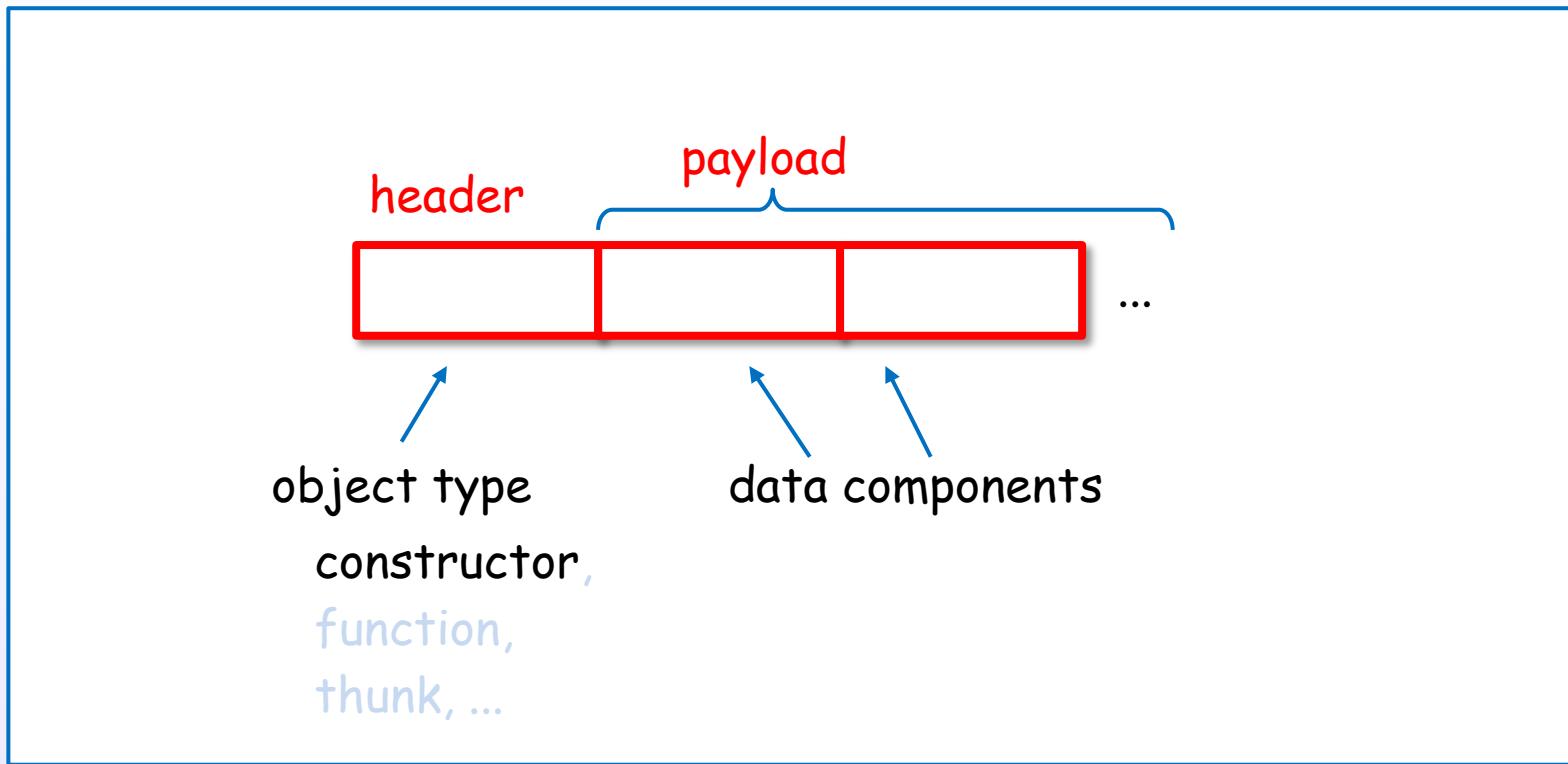


GHC's internal representation



Constructors are represented uniformly

GHC's internal representation



A data value is represented with header(constructor) + payload(components).

Representation of various constructors

Haskell code

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

```
data Maybe a = Nothing
              | Just a
```

```
data Either a b = Left a
                  | Right b
```

GHC's internal representation

False

True

Nothing

Just

Left

Right

Primitive data types are also represented with constructors

Haskell code

```
data Int = I# Int#
```

boxed integer

unboxed integer

```
data Char = C# Char#
```

I# 0#

I# 1#

⋮ 1 :: Int

C# 'a'#

C# 'b'#

⋮ 'a' :: Char

heap memory

List is also represented with constructors

List

```
[ 1, 2, 3 ]
```

syntactic desugar

```
1 : ( 2 : ( 3 : [] ) )
```

prefix notation by section

```
(:) 1 ( (:) 2 ( (:) 3 [] ) )
```

equivalent data constructor

```
Cons 1 ( Cons 2 ( Cons 3 Nil ) )
```

constructor

List is also represented with constructors

List

[1, 2, 3]

syntactic desugar

1 : (2 : (3 : []))

prefix notation by section

(:) 1 ((:) 2 ((:) 3 []))

equivalent data constructor

Cons 1 (Cons 2 (Cons 3 Nil))

type declaration

* pseudo code

```
data List a = [] | a : (List a)
```

```
data List a = Nil | Cons a (List a)
```

List is also represented with constructors

Haskell code

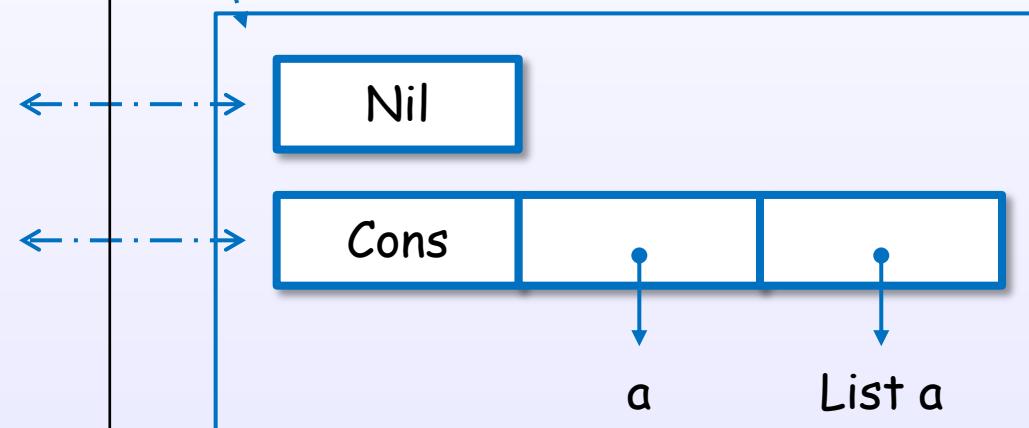
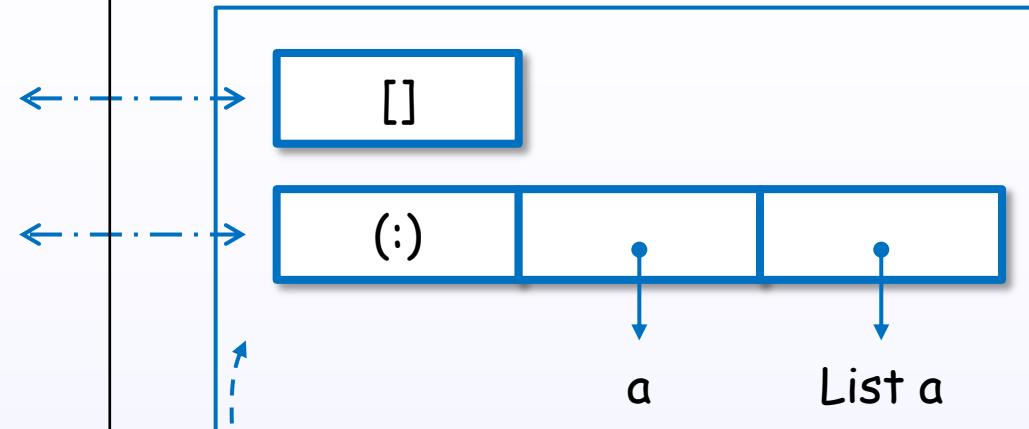
* pseudo code

```
data List a = []  
| : a (List a)
```

equivalent data constructor

```
data List a = Nil  
| Cons a (List a)
```

GHC's internal representation



heap memory

List is also represented with constructors

Haskell code

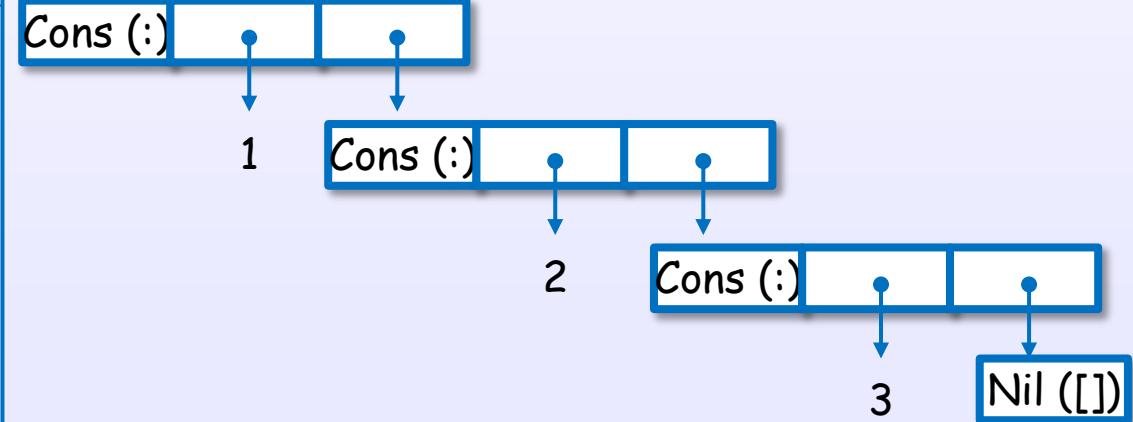
```
[ 1, 2, 3 ]
```

```
1 : ( 2 : ( 3 : [] ) )
```

```
(:) 1 ( (:) 2 ( (:) 3 [] ) )
```

```
Cons 1 ( Cons 2 ( Cons 3 Nil ) )
```

GHC's internal representation



Tuple is also represented with constructor

Tuple (Pair)

(7, 8)

prefix notation by section

(.) 7 8

equivalent data constructor

Pair 7 8

constructor

type declaration

* pseudo code

data Pair a = (.) a a

data Pair a = Pair a a

Tuple is also represented with constructor

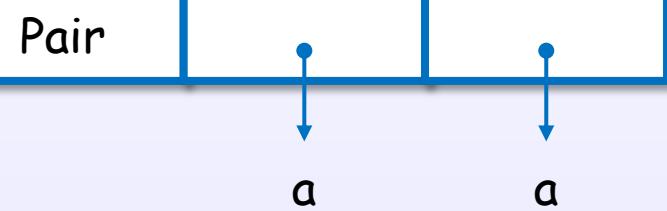
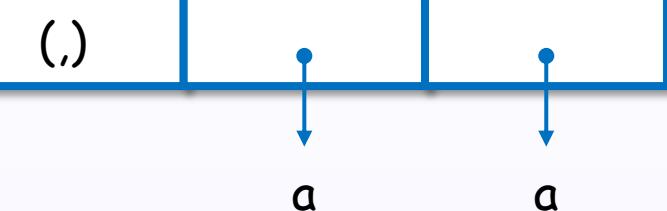
Haskell code

```
data Pair a = (,) a a
```

equivalent data constructor

```
data Pair a = Pair a a
```

GHC's internal representation



heap memory

Tuple is also represented with constructor

Haskell code

(7, 8)

(.,) 7 8

Pair 7 8

GHC's internal representation

Pair (.)

7

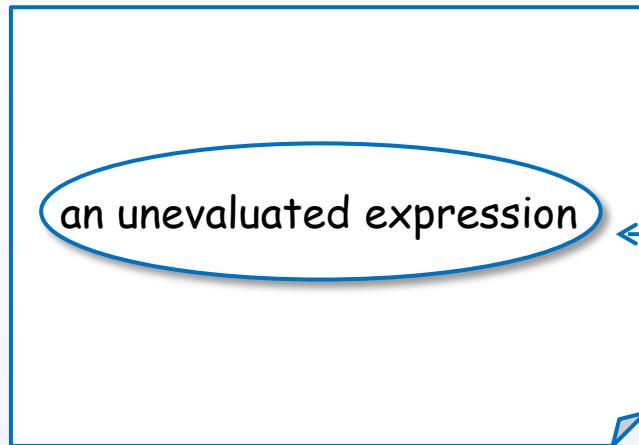
8

3. Internal representation of expressions

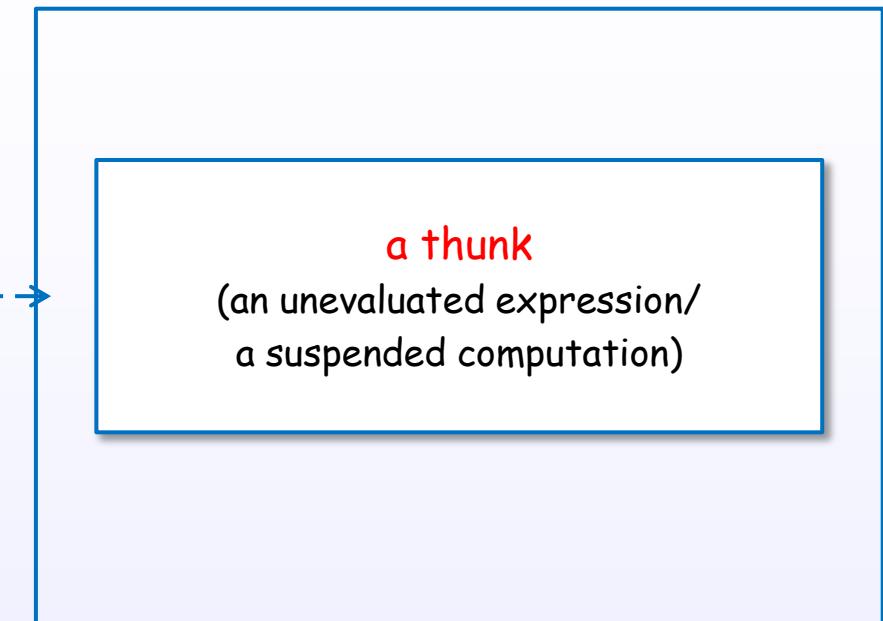
Thunk

Thunk

Haskell code



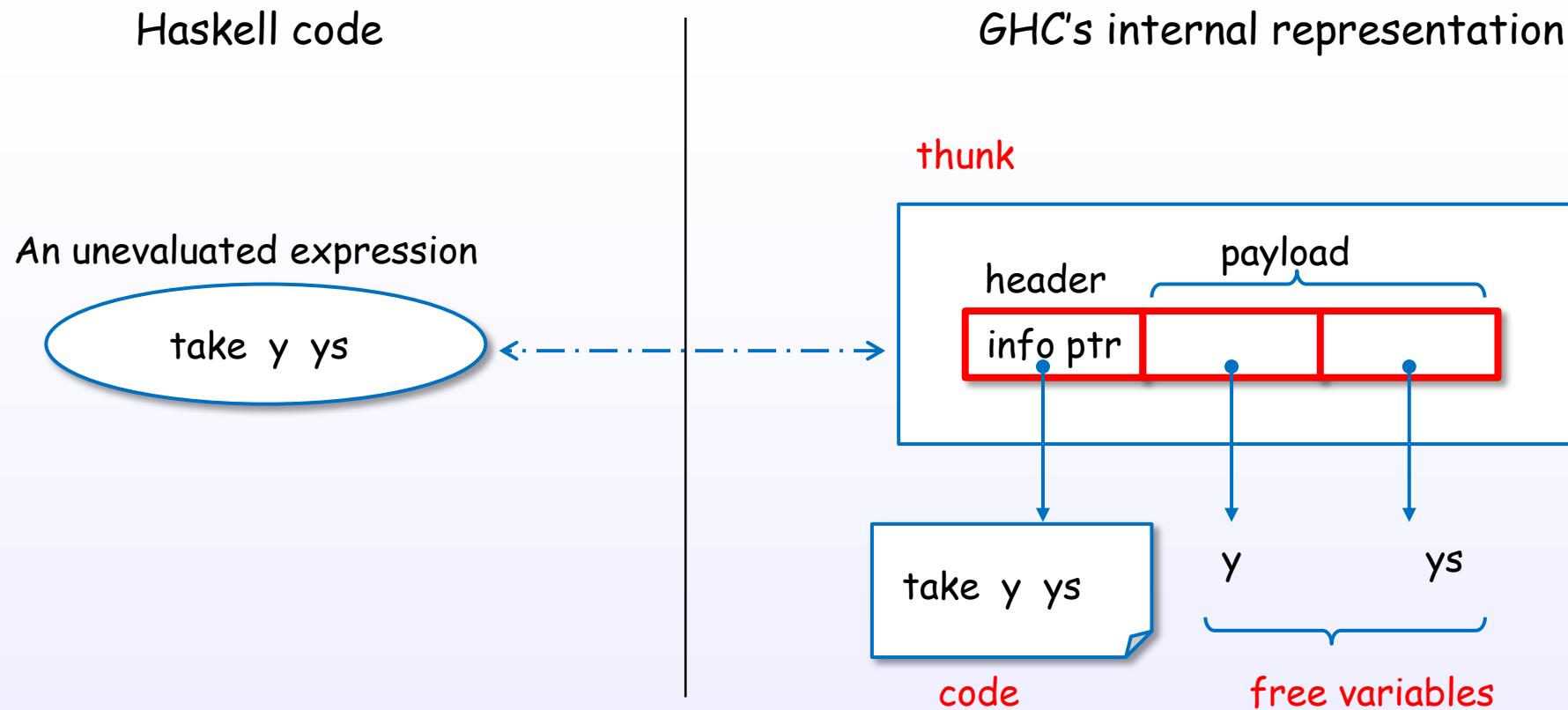
GHC's internal representation



heap memory

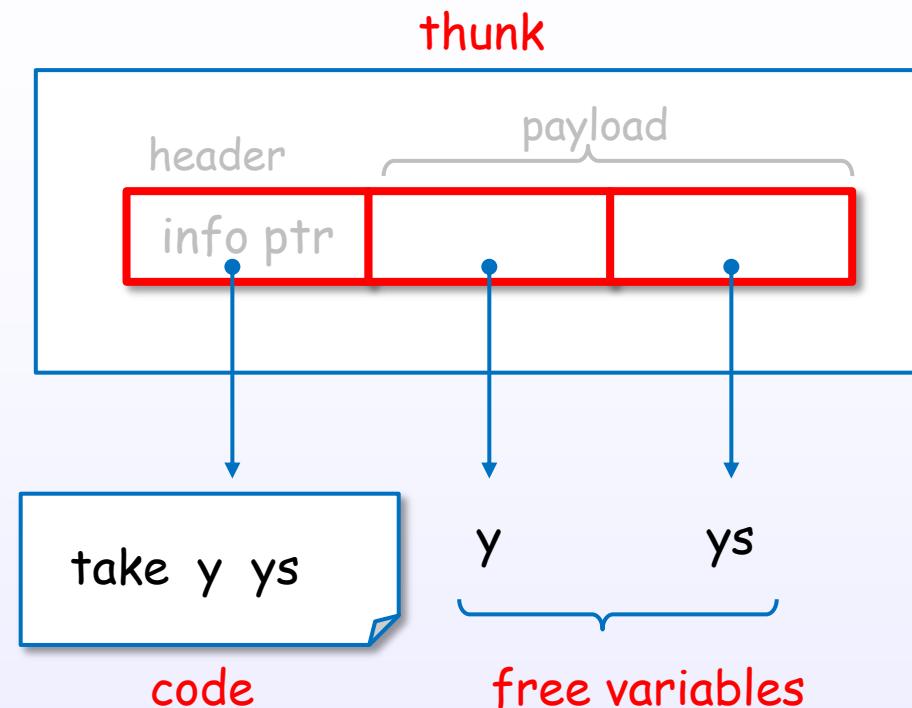
A thunk is an **unevaluated** expression in heap memory.
A thunk is built to **postpone** the evaluation.

Internal representation of thunk



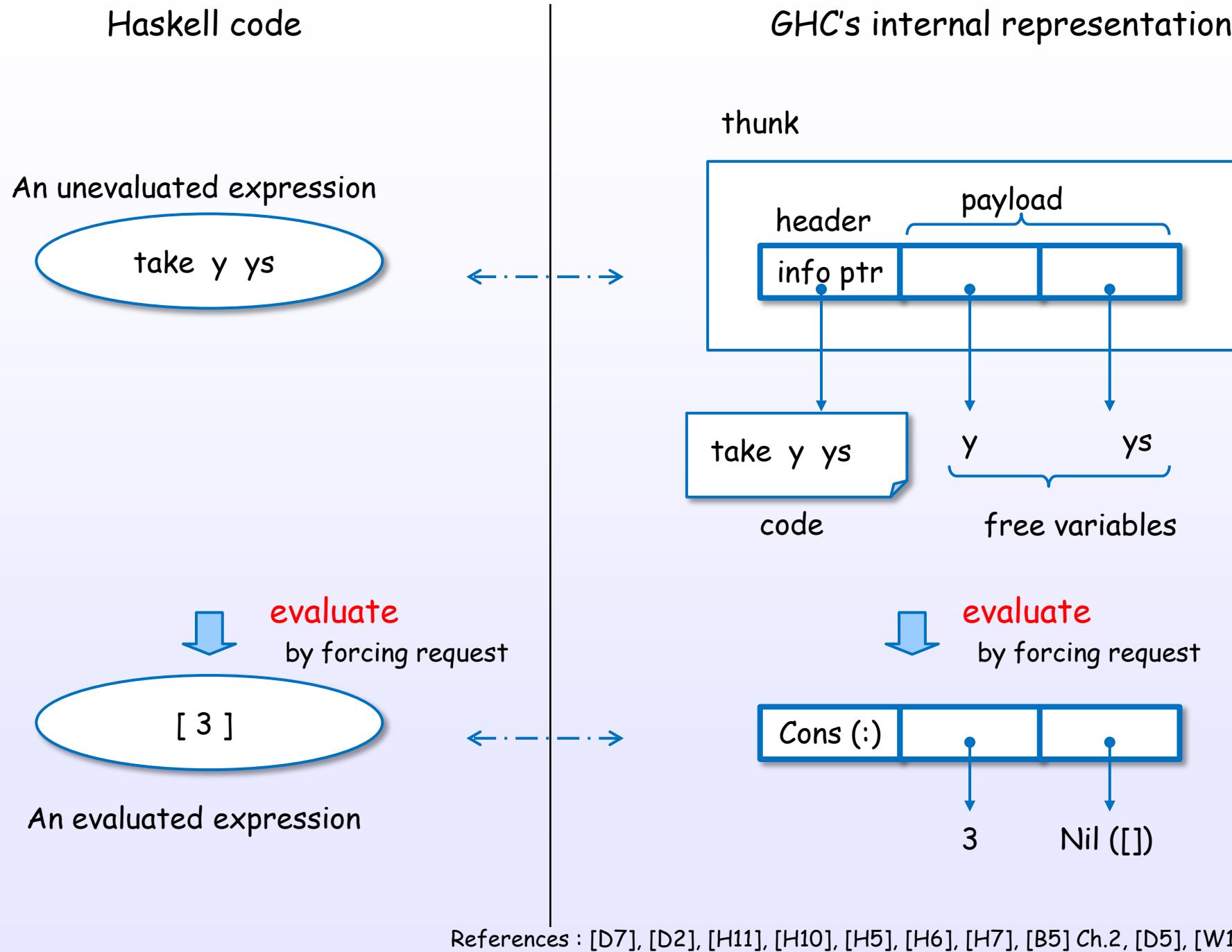
A thunk is represented with header(code) + payload(free variables).

A thunk is a package



A thunk is a package of code + free variables.

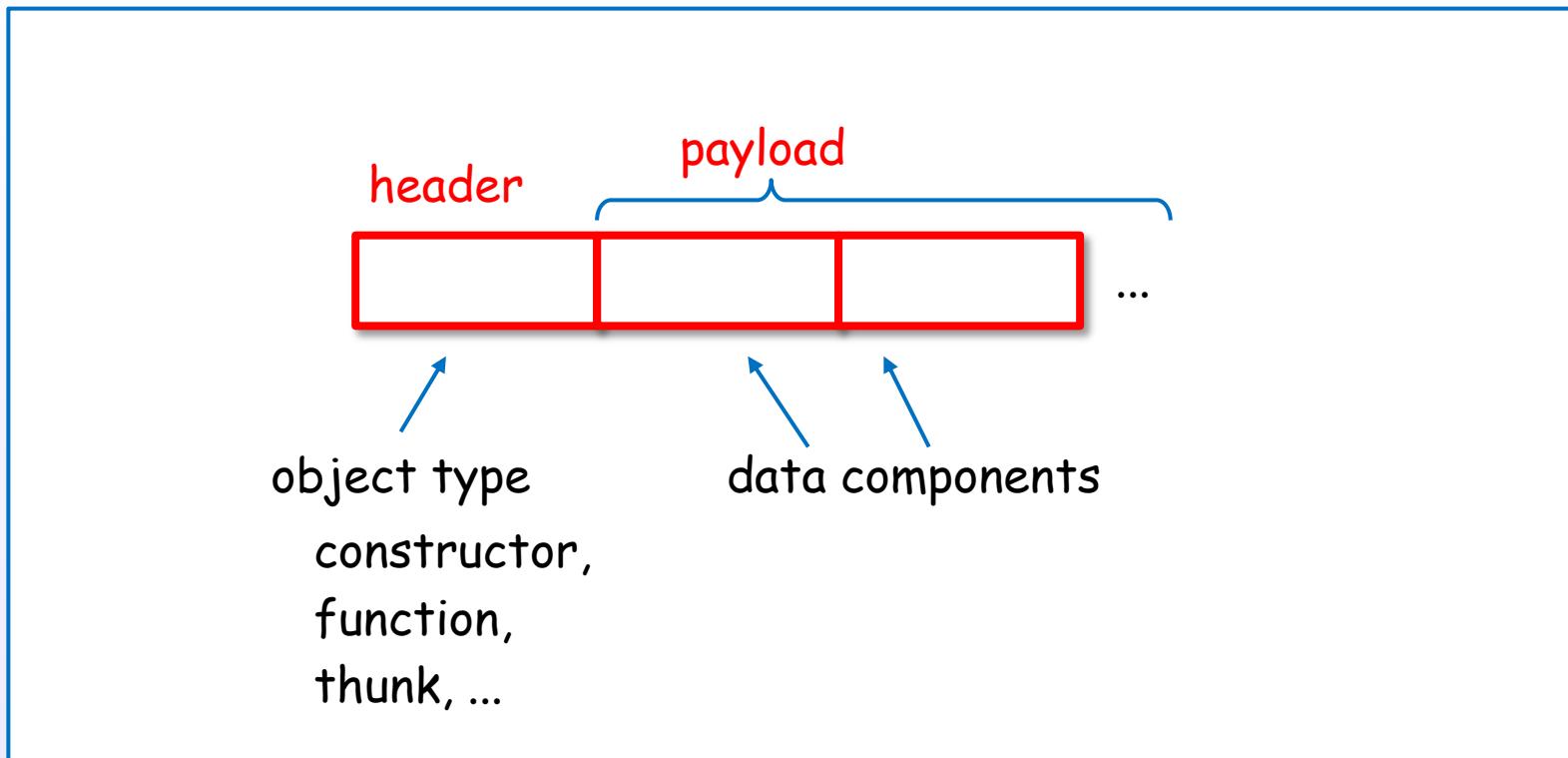
A thunk is evaluated by forcing request



3. Internal representation of expressions

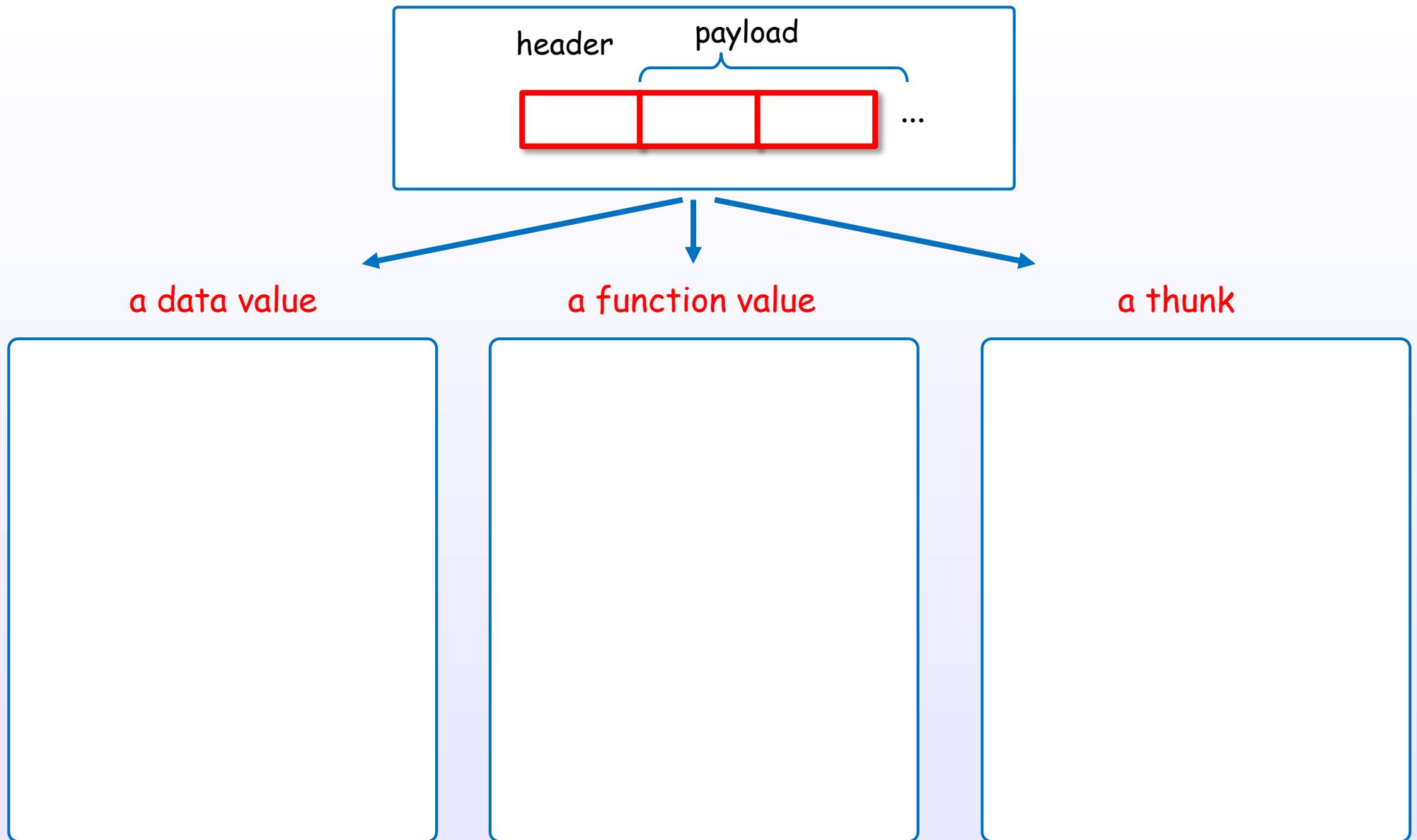
Uniform representation

Every object is uniformly represented in memory

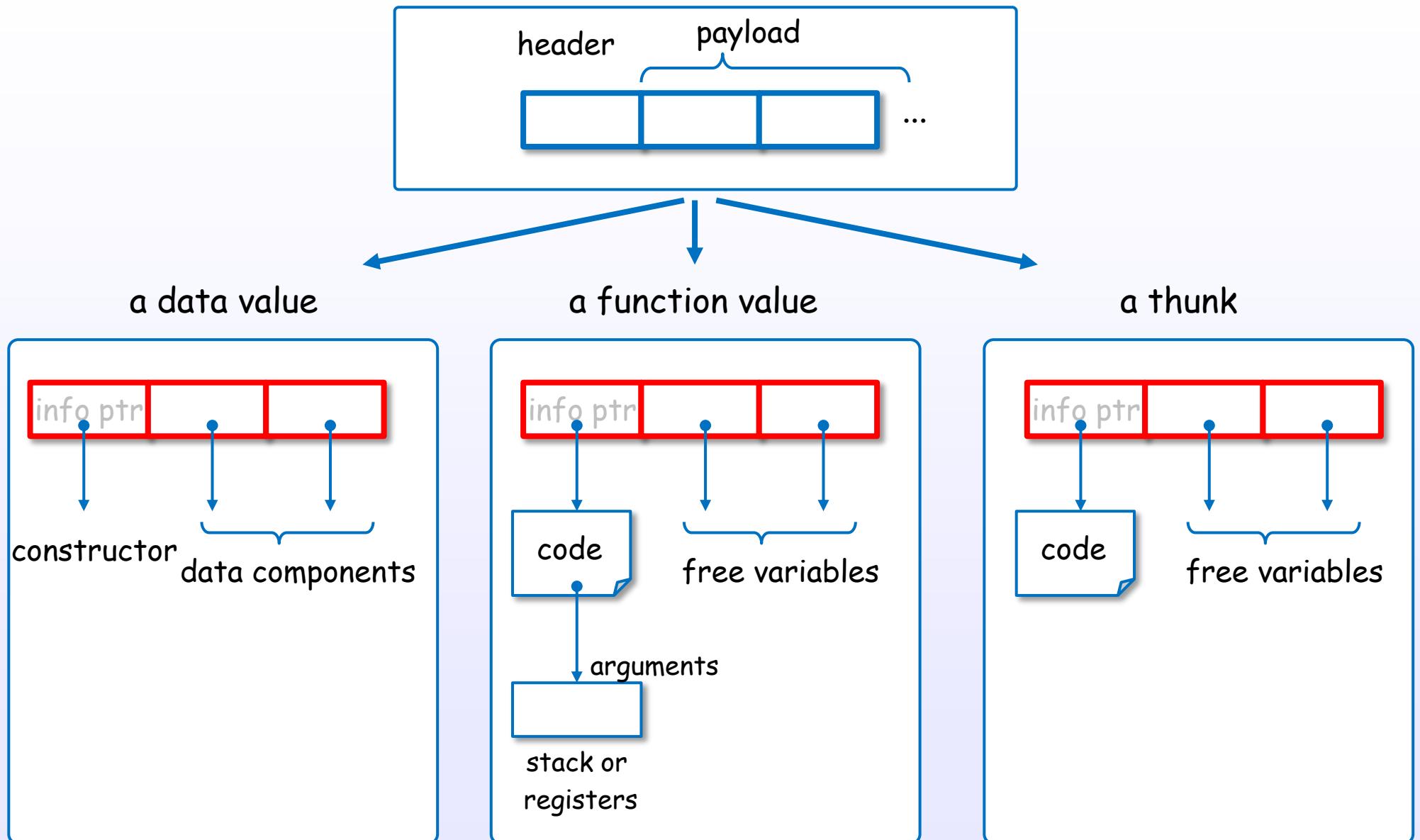


in heap memory, stack or static memory

Every object is uniformly represented in memory



Every object is uniformly represented in memory



* At exactly, a thunk object has a reserved field in second.

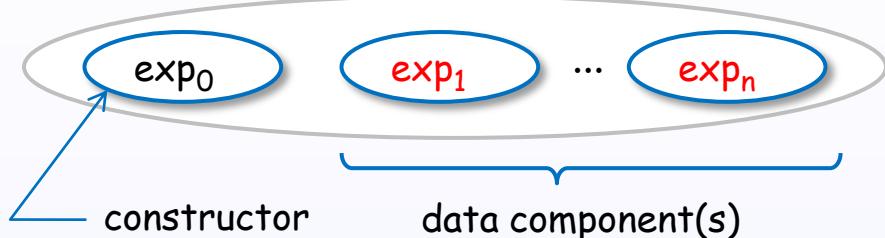
3. Internal representation of expressions

WHNF

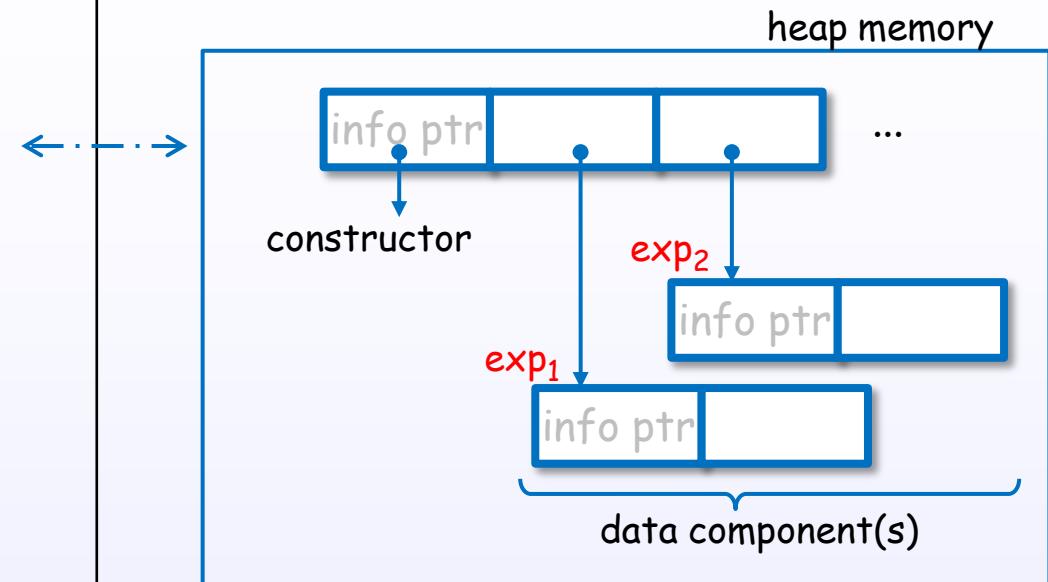
Internal representation of WHNF

Haskell code

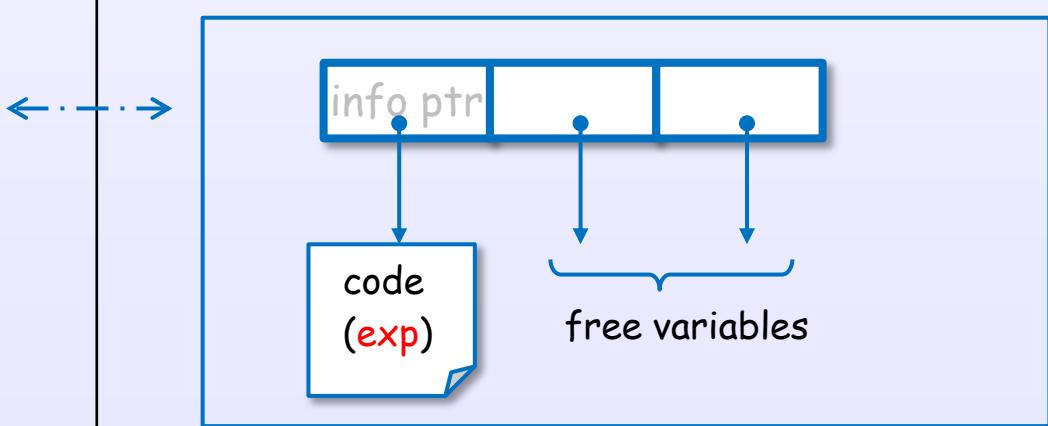
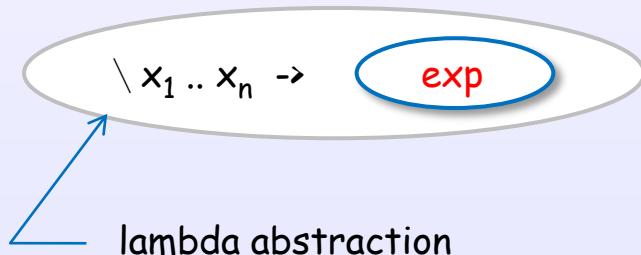
a data value in WHNF



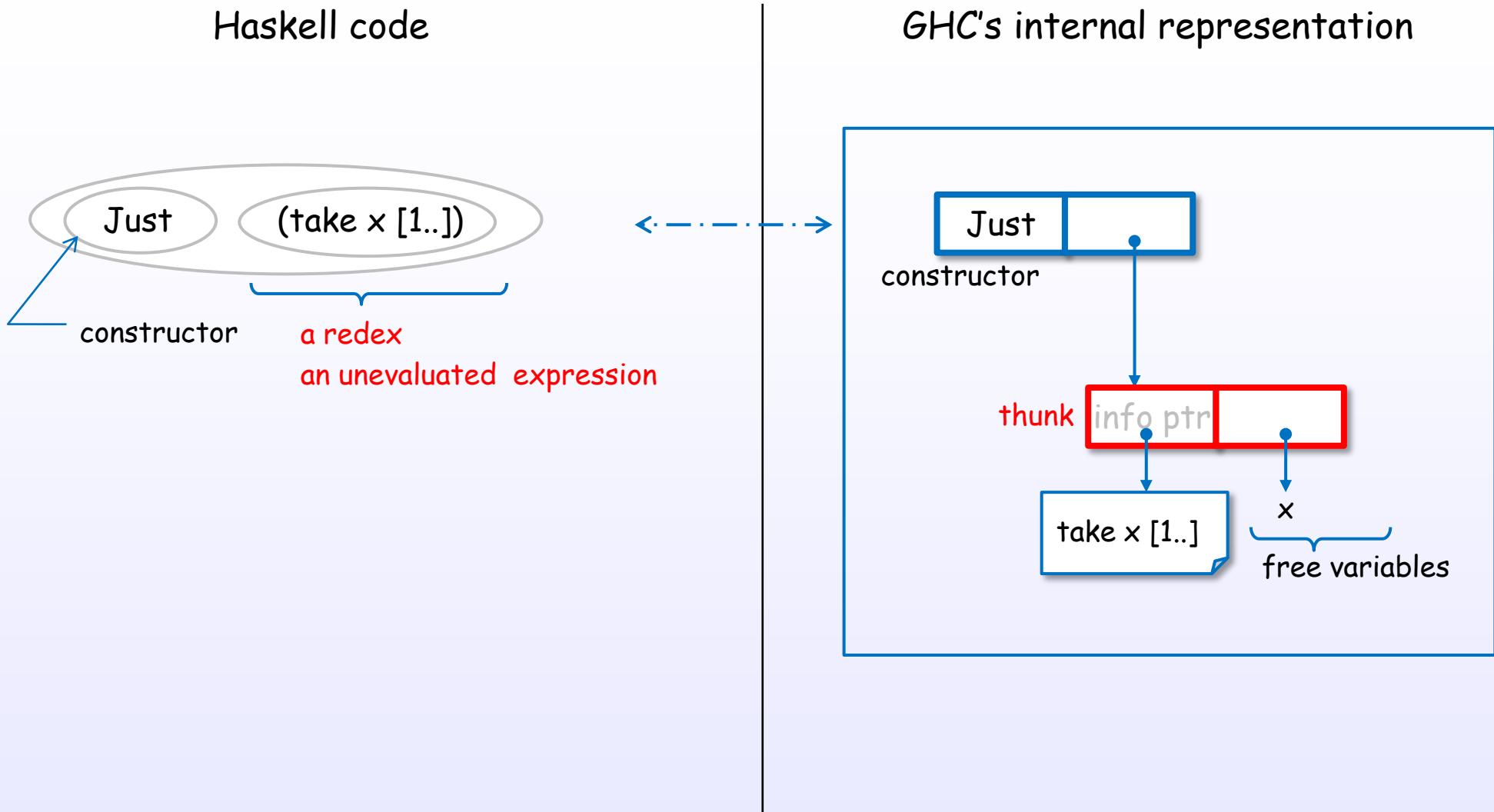
GHC's internal representation



a function value in WHNF



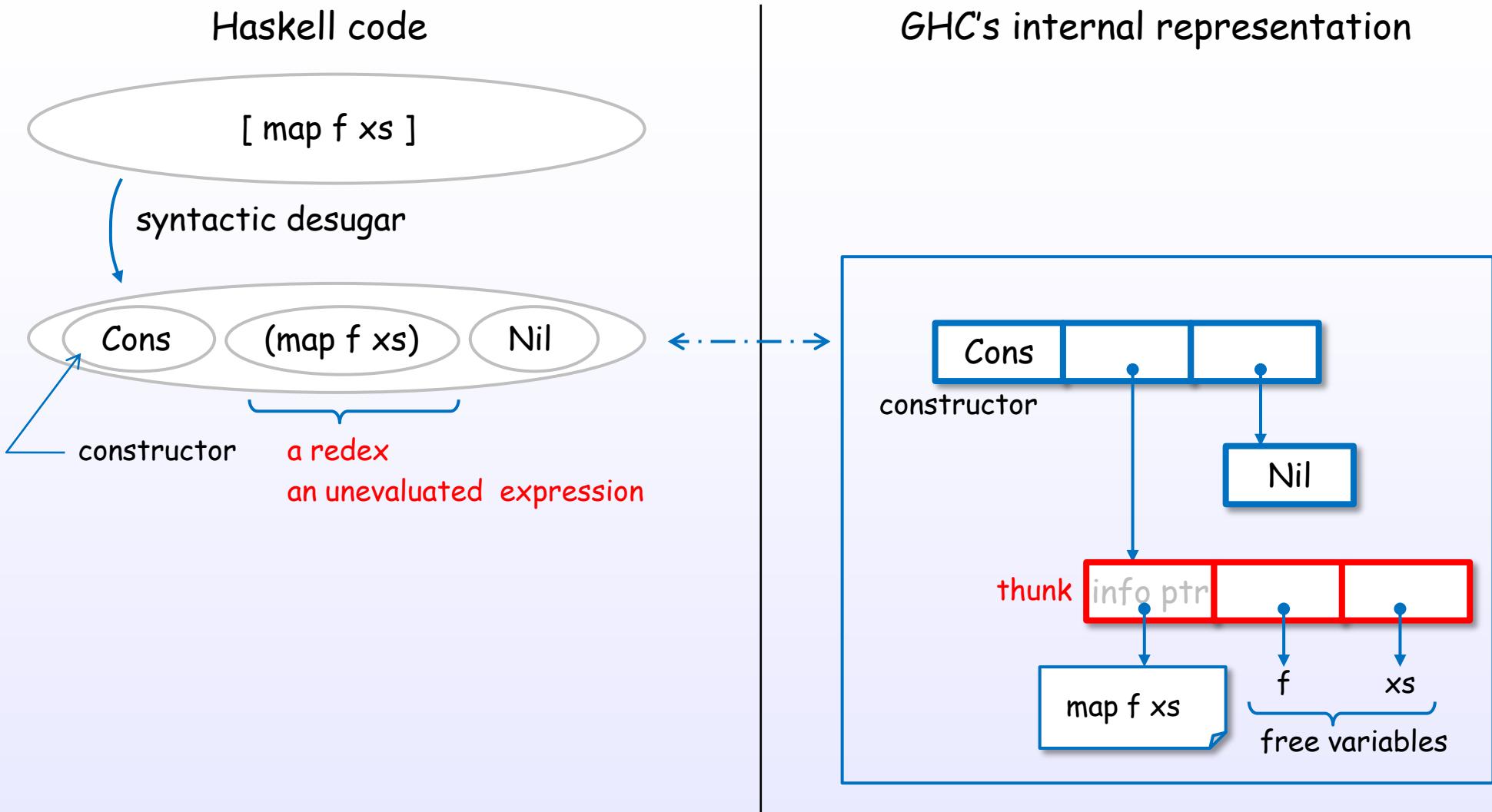
Example of WHNF for a data value



Constructors can contain unevaluated expressions by thunks.

Haskell's constructors are lazy constructors.

Example of WHNF for a data value



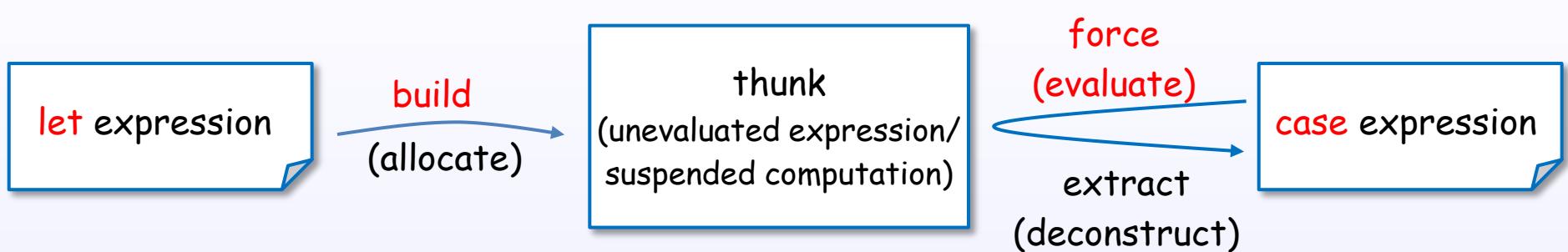
3. Internal representation of expressions

let, case expression

let, case expression

let and case expressions are special role in the evaluation

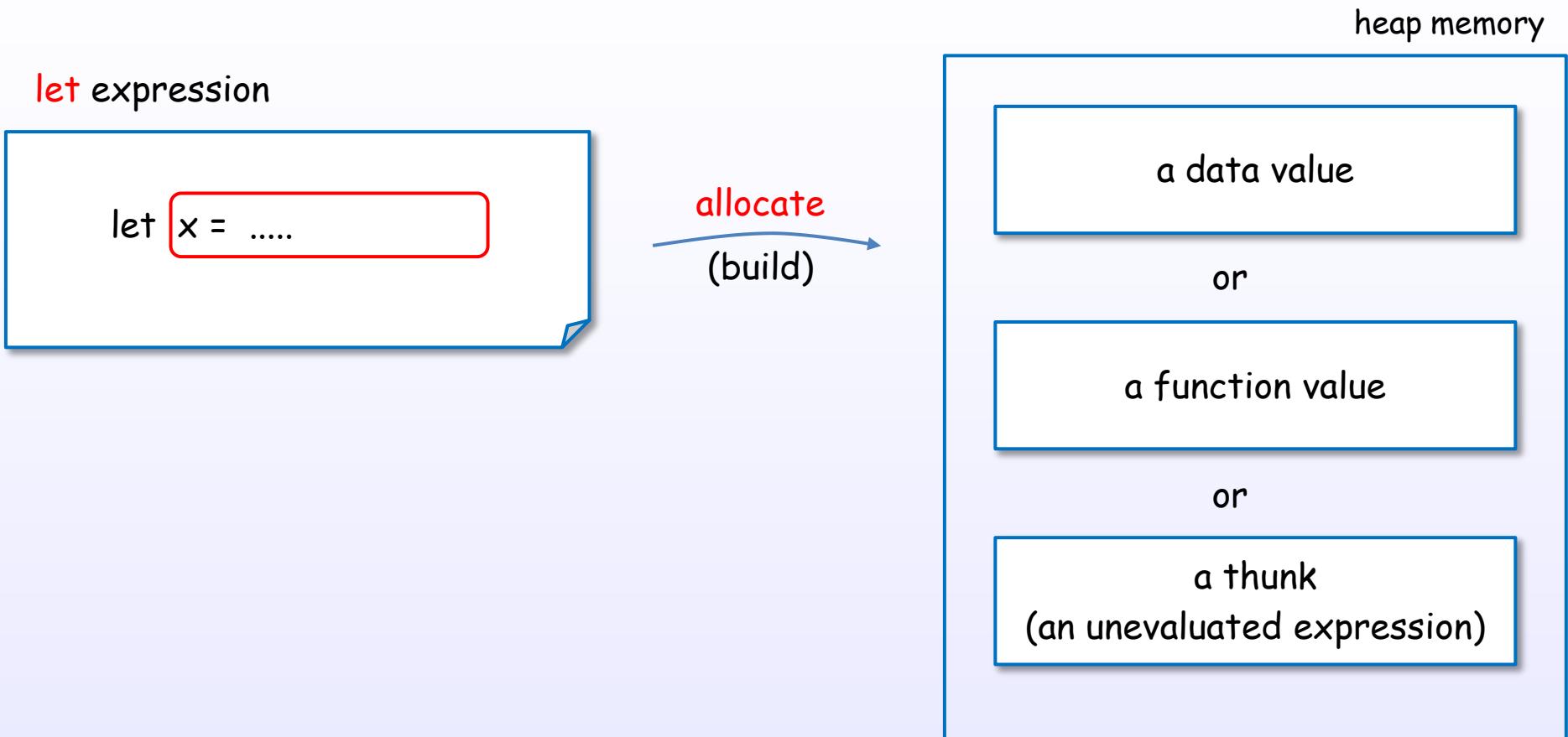
let/case expressions and thunk



A let expression may build a thunk.

A case expression evaluates (forces) and deconstructs the thunk.

A let expression may allocates a heap object



A let expression may allocates an object in the heap.
(If GHC can optimize it, the let expression may not allocate.)

* At exactly, STG language's let expression rather than Haskell's let expression

Example of let expressions

Haskell code

```
let x = Just 5
```

allocate

```
let x = \y -> y + z
```

allocate

```
let x = take y ys
```

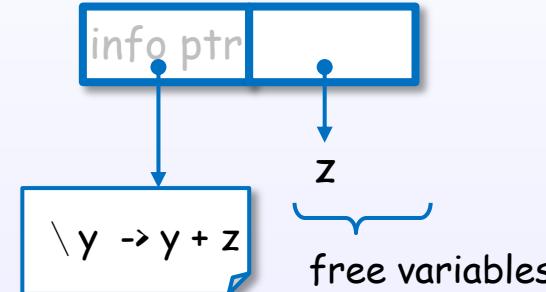
allocate
(build)

GHC's internal representation

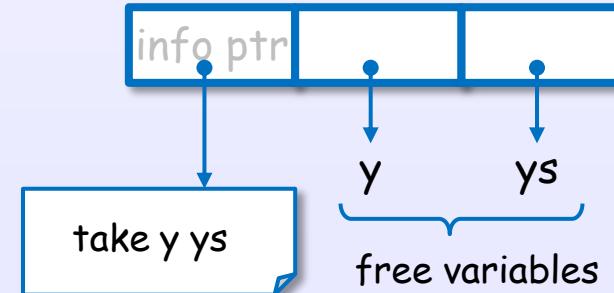
a data value



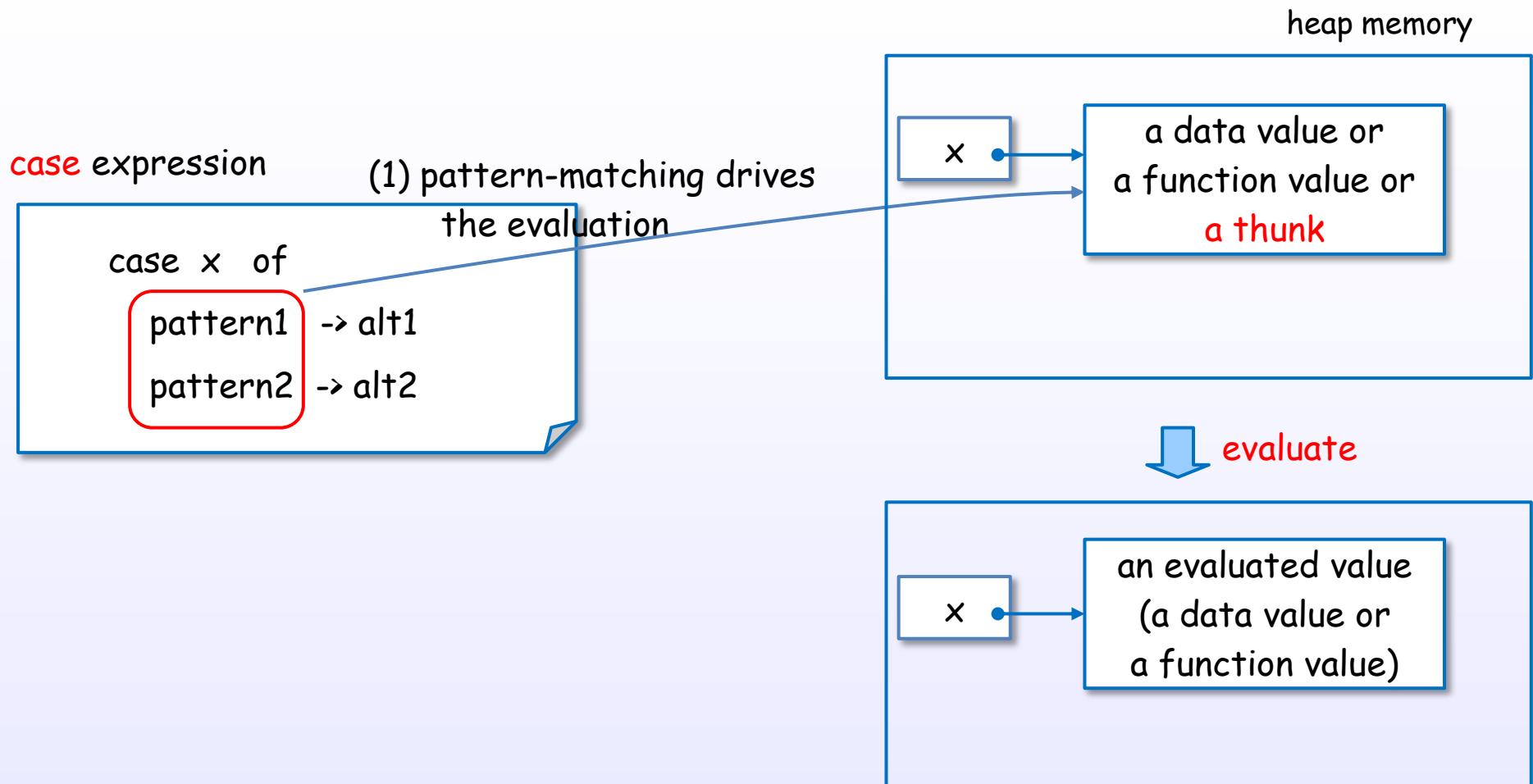
a function value



a thunk



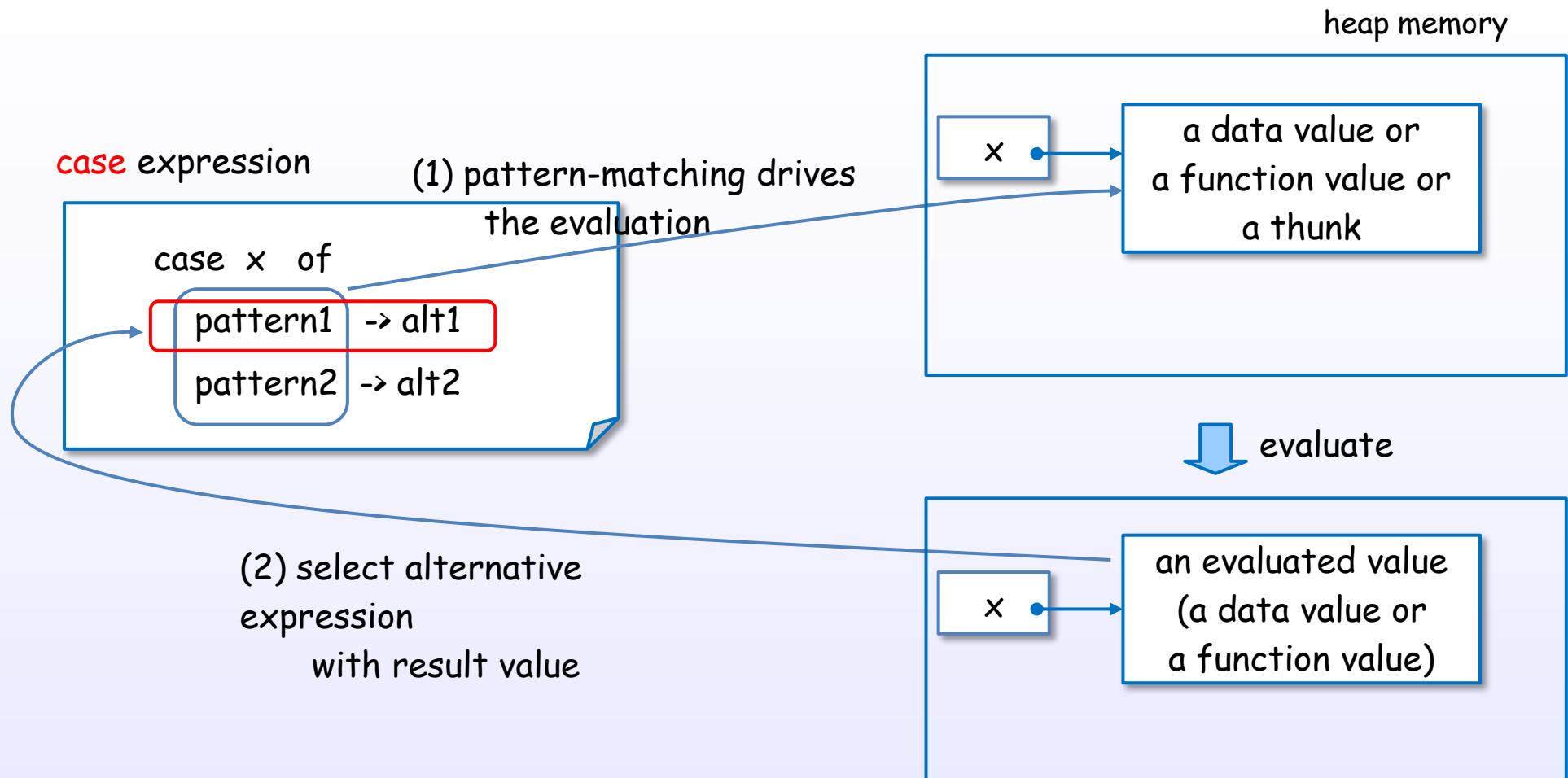
A case expression evaluates a subexpression



Pattern-matching drives the evaluation.

* At exactly, STG language's case expression rather than Haskell's case expression

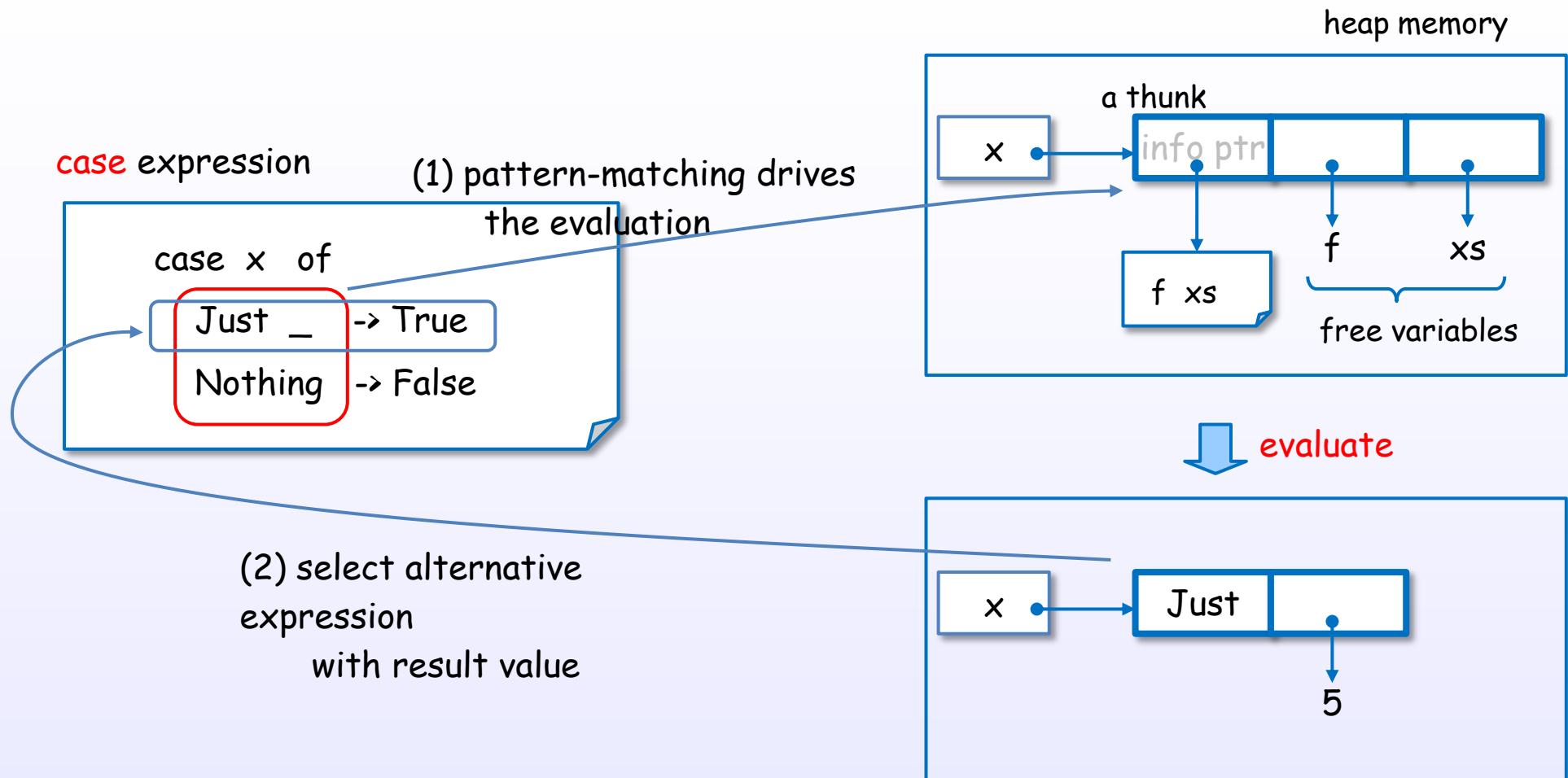
A case expression also performs case analysis



A case expression evaluates a subexpression and optionally performs case analysis on its value.

* At exactly, STG language's case expression rather than Haskell's case expression

Example of a case expression



A case expression's pattern-matching says "I **need** the value".

Pattern-matching in function definition

pattern-matching in **function definition**

$$\begin{aligned} f \text{ } \boxed{\text{Just } _} &= \text{True} \\ f \text{ } \boxed{\text{Nothing}} &= \text{False} \end{aligned}$$

pattern-matching in **case expression**

$$\begin{aligned} f \text{ } x = \text{case } x \text{ of} \\ \boxed{\text{Just } _} &\rightarrow \text{True} \\ \boxed{\text{Nothing}} &\rightarrow \text{False} \end{aligned}$$

syntactic desugar

A function's pattern-matching is syntactic sugar of case expression.

A function's pattern-matching also drives the evaluation.

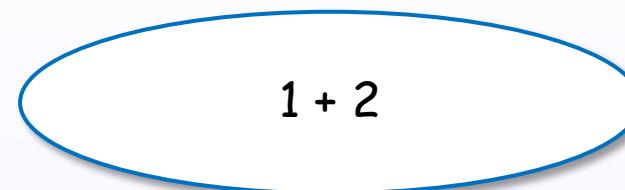
4. Evaluation

4. Evaluation

Evaluation strategies

Evaluation

An expression



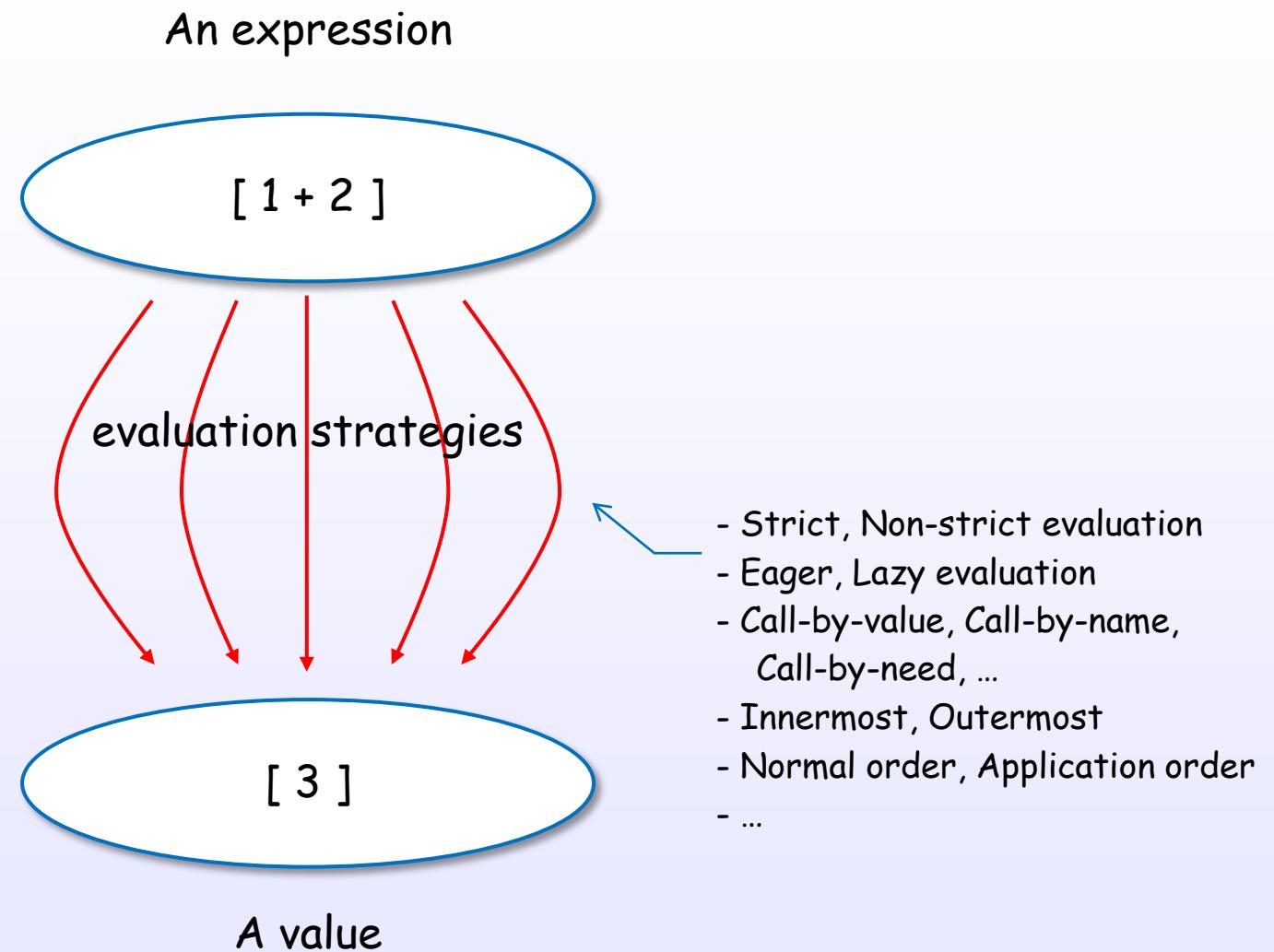
evaluate

3

A value

The evaluation produces a value from an expression.

There are many evaluation approaches



Evaluation concept layer

Denotational semantics

Operational semantics
(Evaluation strategies / Reduction strategies)

Implementation techniques

Evaluation layer for GHC's Haskell

Denotational
semantics

Strict semantics

Non-strict semantics

Operational
semantics

Strict evaluation

Non-strict evaluation

Eager evaluation

Nondeterministic
evaluation

Lazy evaluation

...

Call-by-Value

Call-by-Name

Call-by-Need

...

Applicative order reduction

Normal order reduction

...

Rightmost reduction

Innermost reduction

Leftmost reduction

Outermost reduction

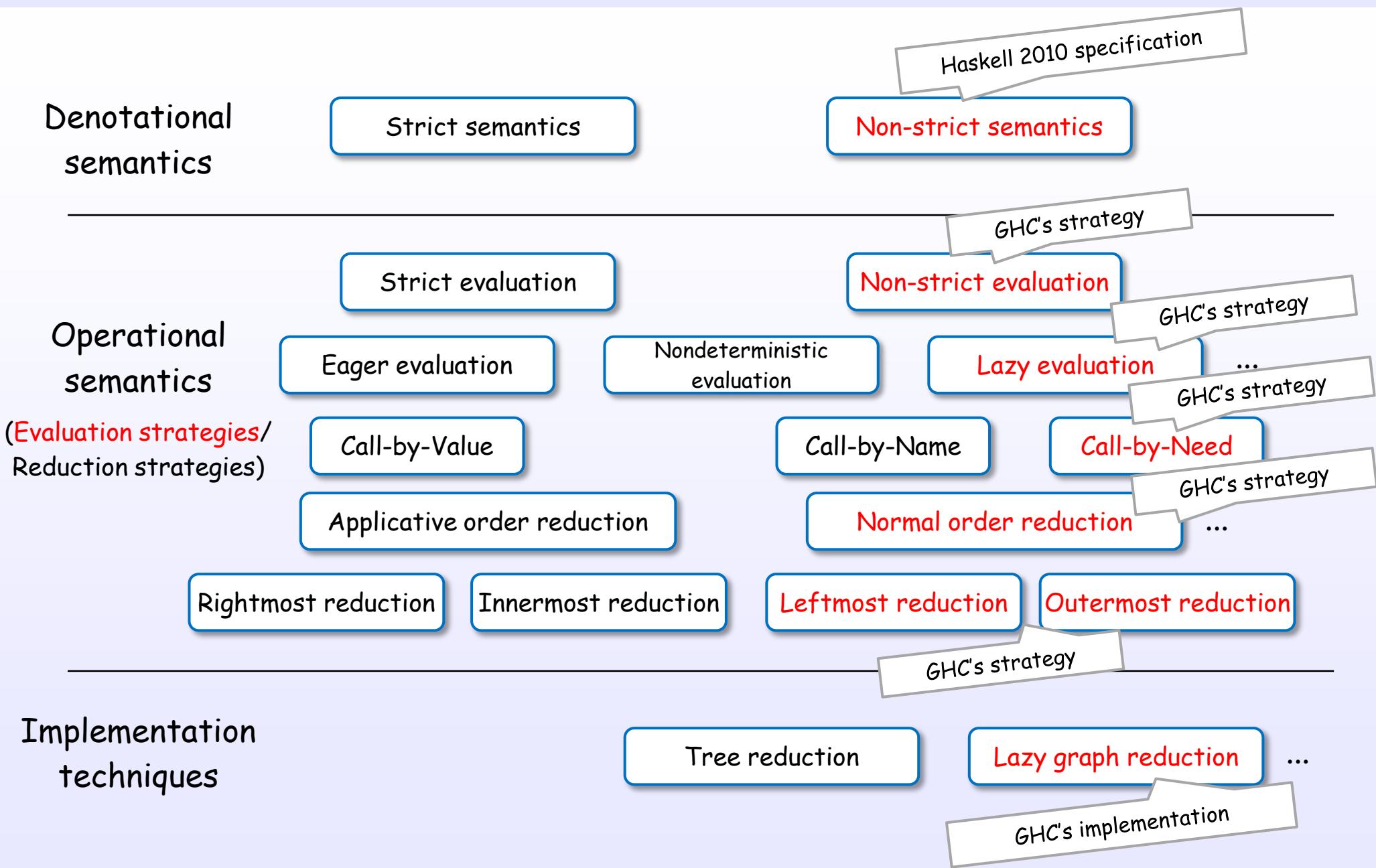
Implementation
techniques

Tree reduction

Lazy graph reduction

...

Evaluation layer for GHC's Haskell

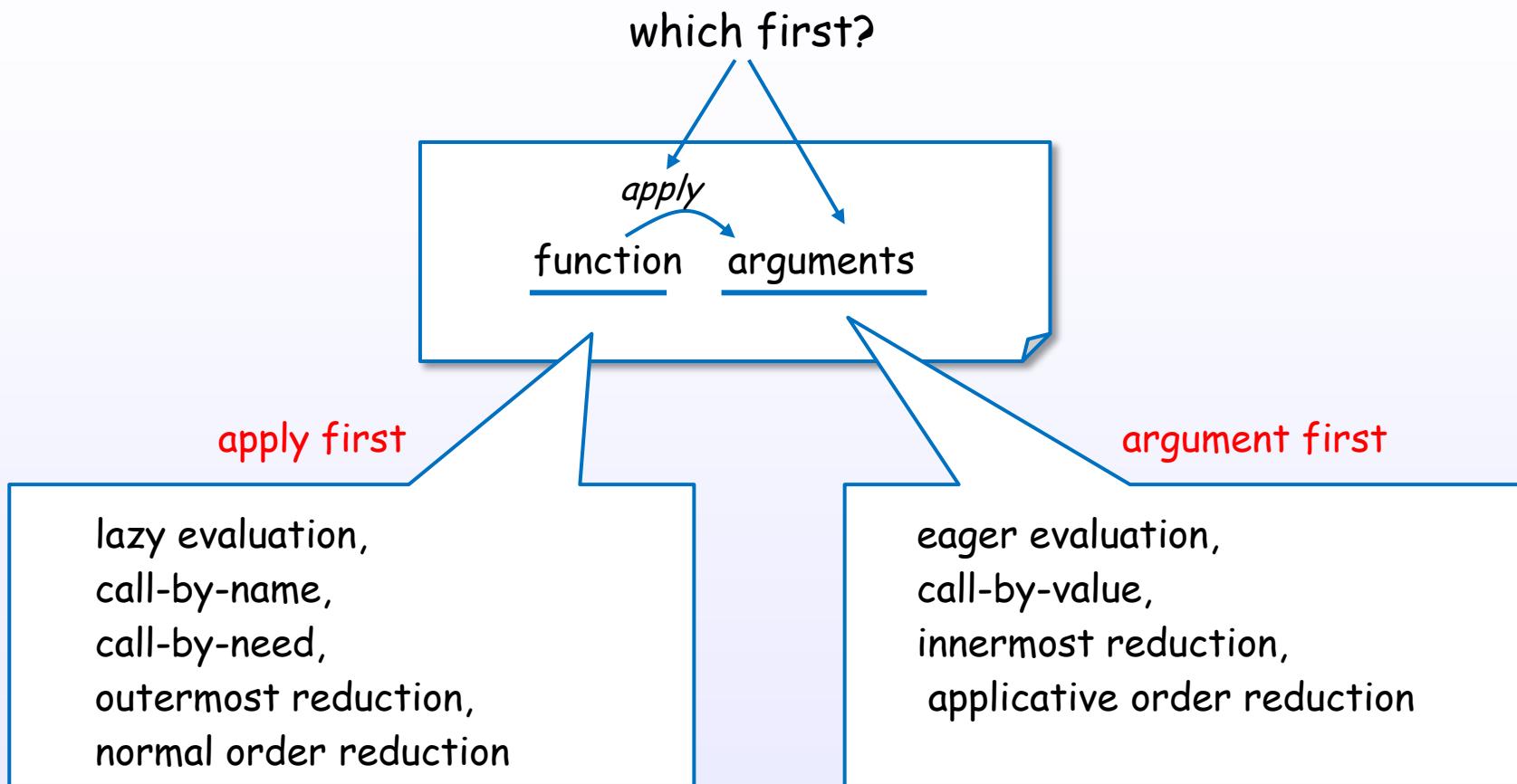


Evaluation strategies

Each evaluation strategy decides how to operate the evaluation about

- ordering,
- region,
- trigger condition,
- termination condition,
- re-evaluation, ...

One of the important points is the order



Simple example of typical evaluations

call-by-value

default

C, Java, JavaScript,
Python, OCaml, Scheme, ...

square (1 + 2)

argument
evaluation
first



call-by-need

default

Haskell (GHC), ...

square (1 + 2)

apply
first



Simple example of typical evaluations

call-by-value

square (1 + 2)



square (3)



3 * 3



9

call-by-need

square (1 + 2)



(1 + 2) * (1 + 2)



(3) * (3)



9

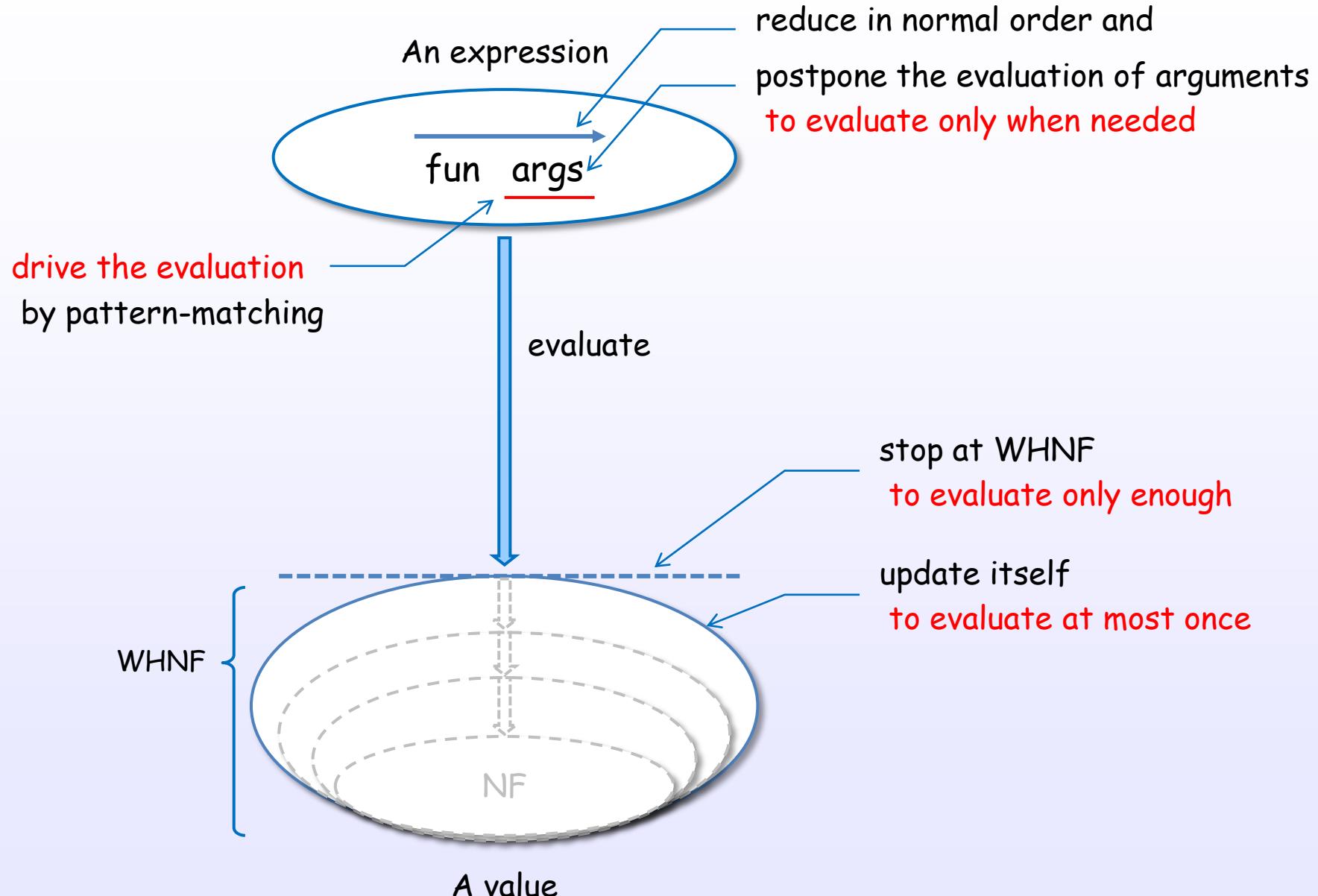
evaluation is
performed

evaluation is
delayed !

4. Evaluation

Evaluation in Haskell (GHC)

Key concepts of Haskell's lazy evaluation



Postpone the evaluation of arguments

Haskell code

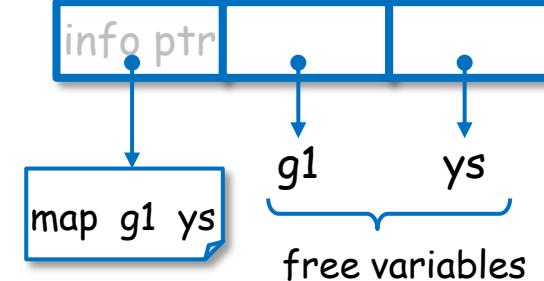
```
fun (map g1 ys)
```

internal translation

```
let thunk0 = map g1 ys
in fun thunk0
```

postpone
(build)

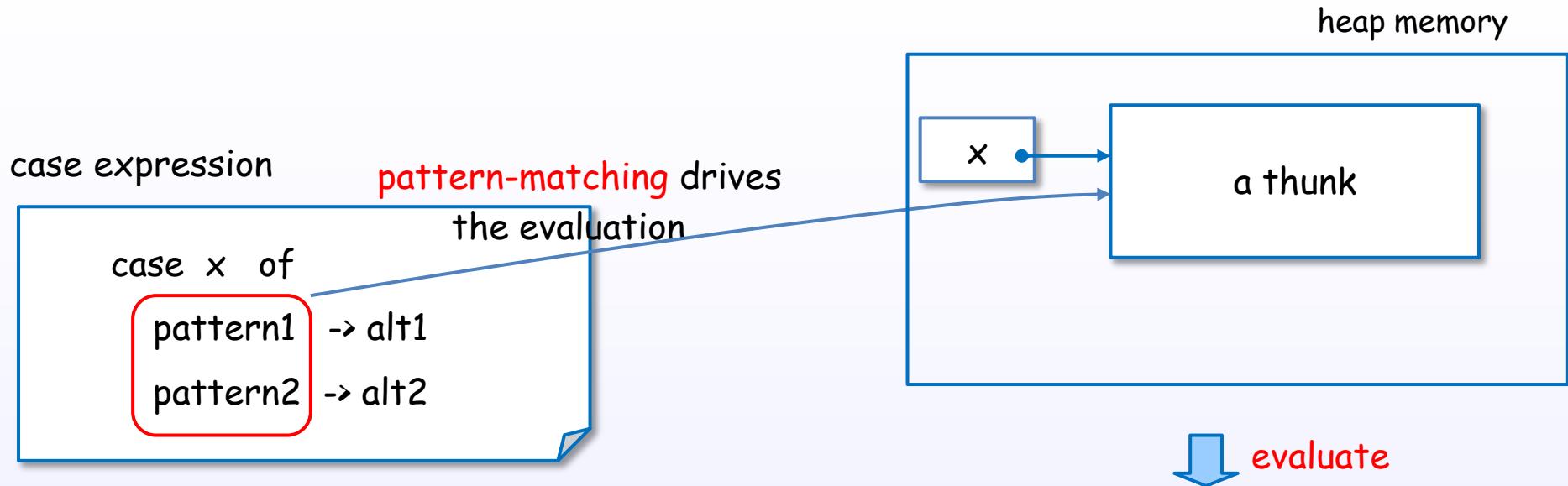
a thunk



heap memory

postpone the evaluation by a thunk which build with let expression
(When GHC can optimize it by analysis, the thunk may not be build.)

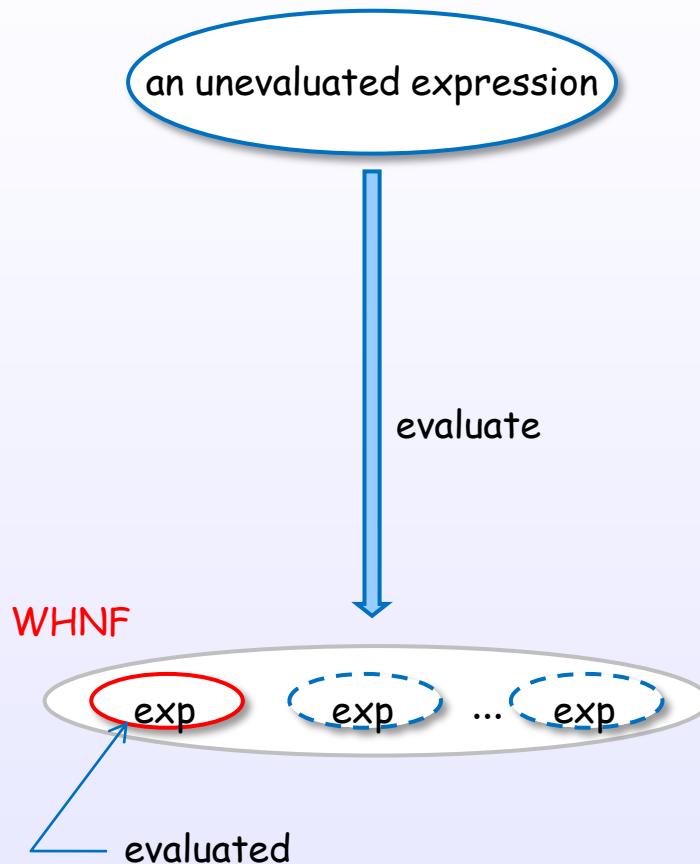
Pattern-matching drives the evaluation



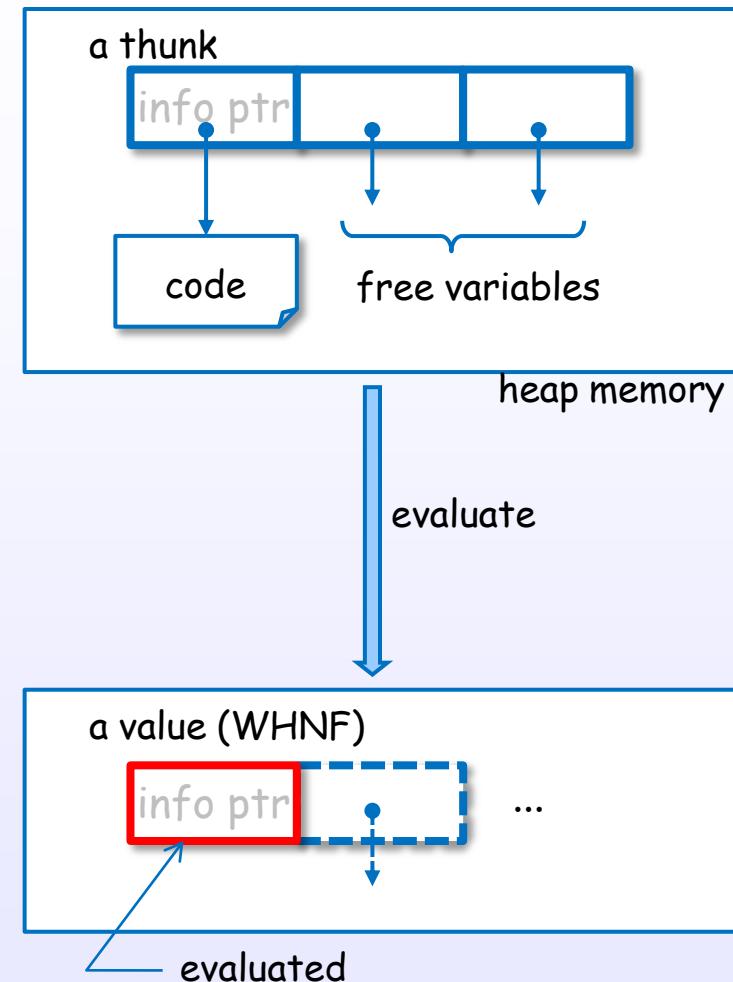
drive the evaluation by pattern-matching

Stop at WHNF

Haskell code



GHC's internal representation



stop the evaluation at WHNF

4. Evaluation

Examples of evaluation steps

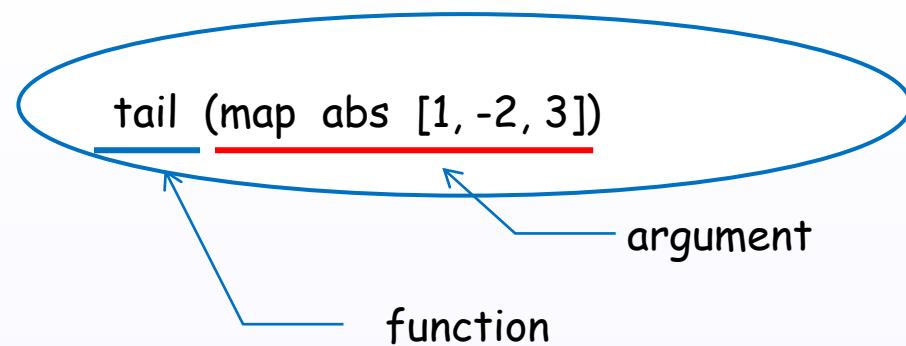
(1) Example of GHC's evaluation

tail (map abs [1, -2, 3])

Let's evaluate. It's time to magic!

* no optimizing case (without -O)

(2) How to postpone the evaluation of arguments?



(3) GHC internally translates the expression

tail (map abs [1, -2, 3])

internal translation

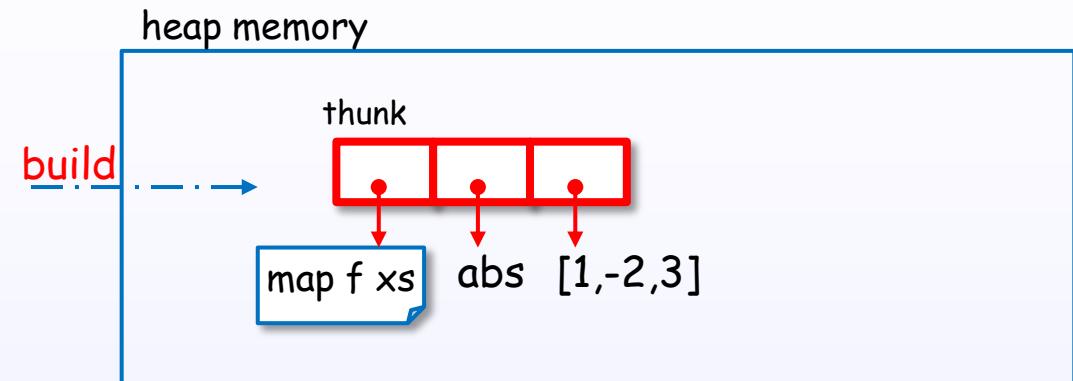
let thunk0 = map abs [1, -2, 3]
in tail thunk0

(4) a let expression builds a thunk

tail (map abs [1, -2, 3])

internal translation

let thunk0 = map abs [1, -2, 3]
in tail thunk0



(5) function apply to argument

tail (map abs [1, -2, 3])

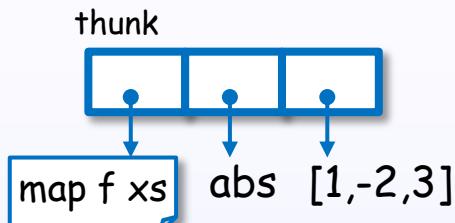
internal translation

let thunk0 = map abs [1, -2, 3]

in tail thunk0

apply

heap memory



(6) tail is defined here

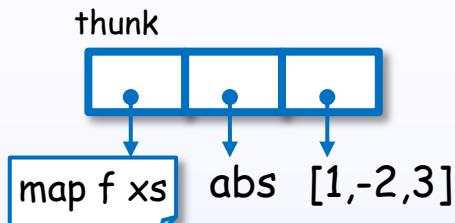
tail (map abs [1, -2, 3])

internal translation

let thunk0 = map abs [1, -2, 3]
in tail thunk0

tail (_:xs) = xs *definition*

heap memory



(7) function is syntactic sugar

tail (map abs [1, -2, 3])

internal translation

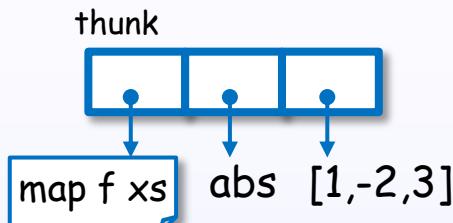
let thunk0 = map abs [1, -2, 3]
in tail thunk0

syntactic
desugar

tail (_:xs) = xs

tail y = case y of
(_:xs) -> xs

heap memory



(8) substitute function body (beta reduction)

tail (map abs [1, -2, 3])

internal translation

let thunk0 = map abs [1, -2, 3]
in tail thunk0

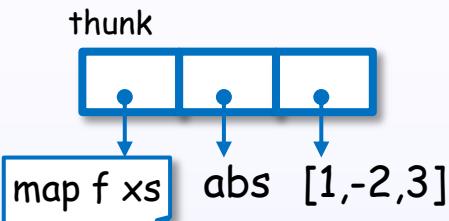
tail (_:xs) = xs

tail y = case y of
(_:xs) -> xs

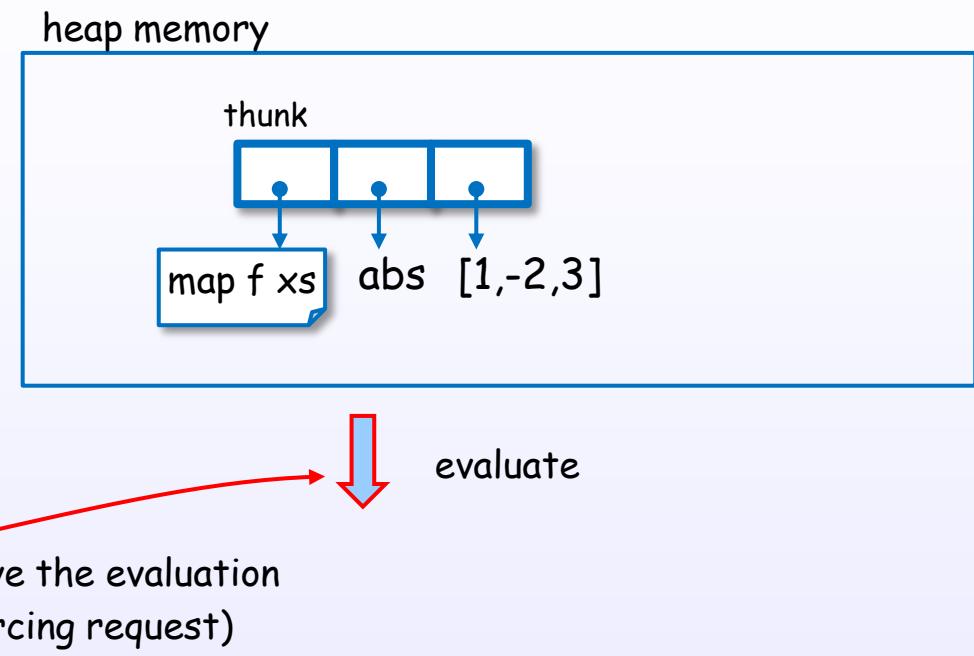
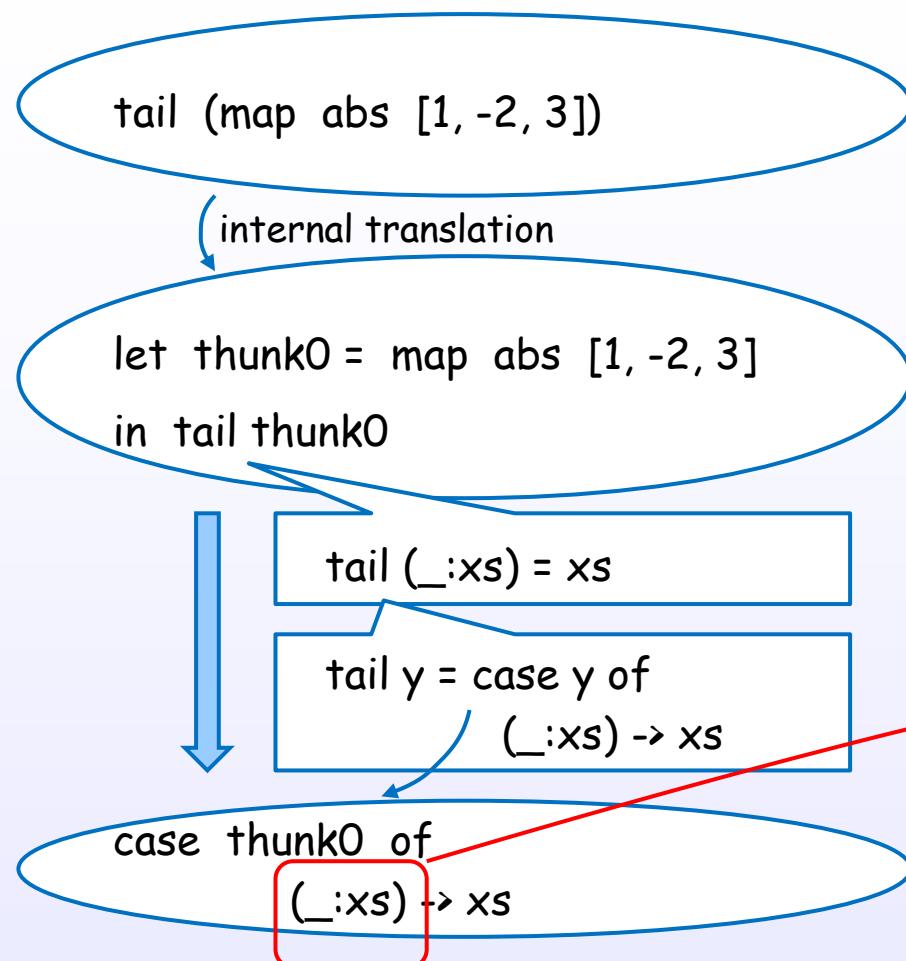
case thunk0 of
(_:xs) -> xs

reduction

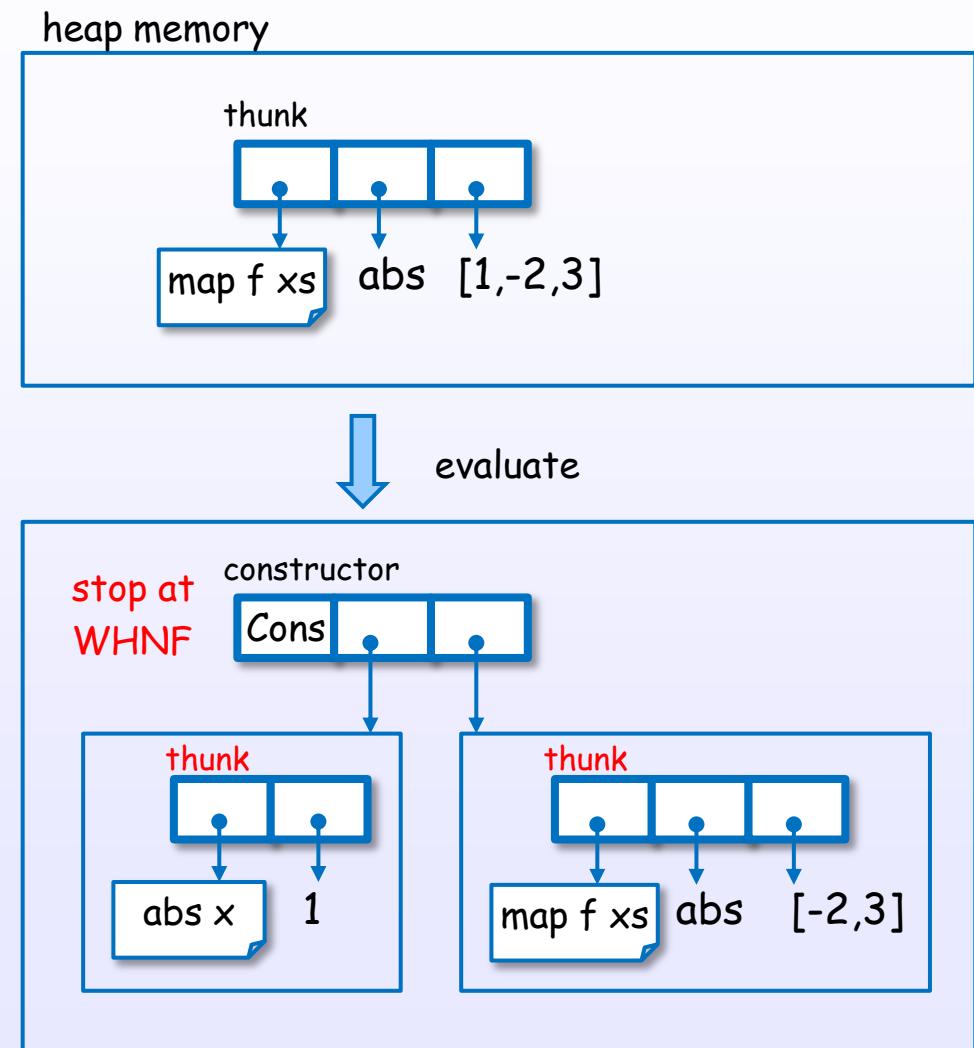
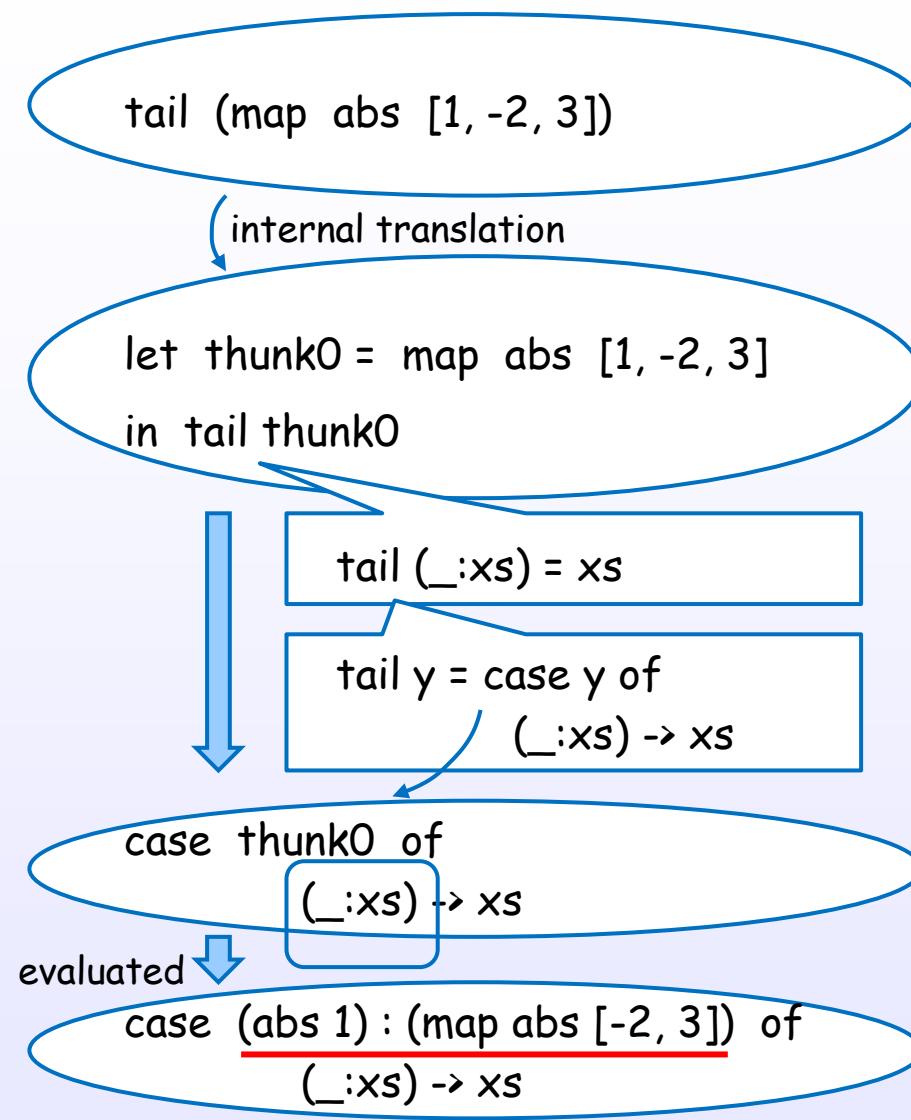
heap memory



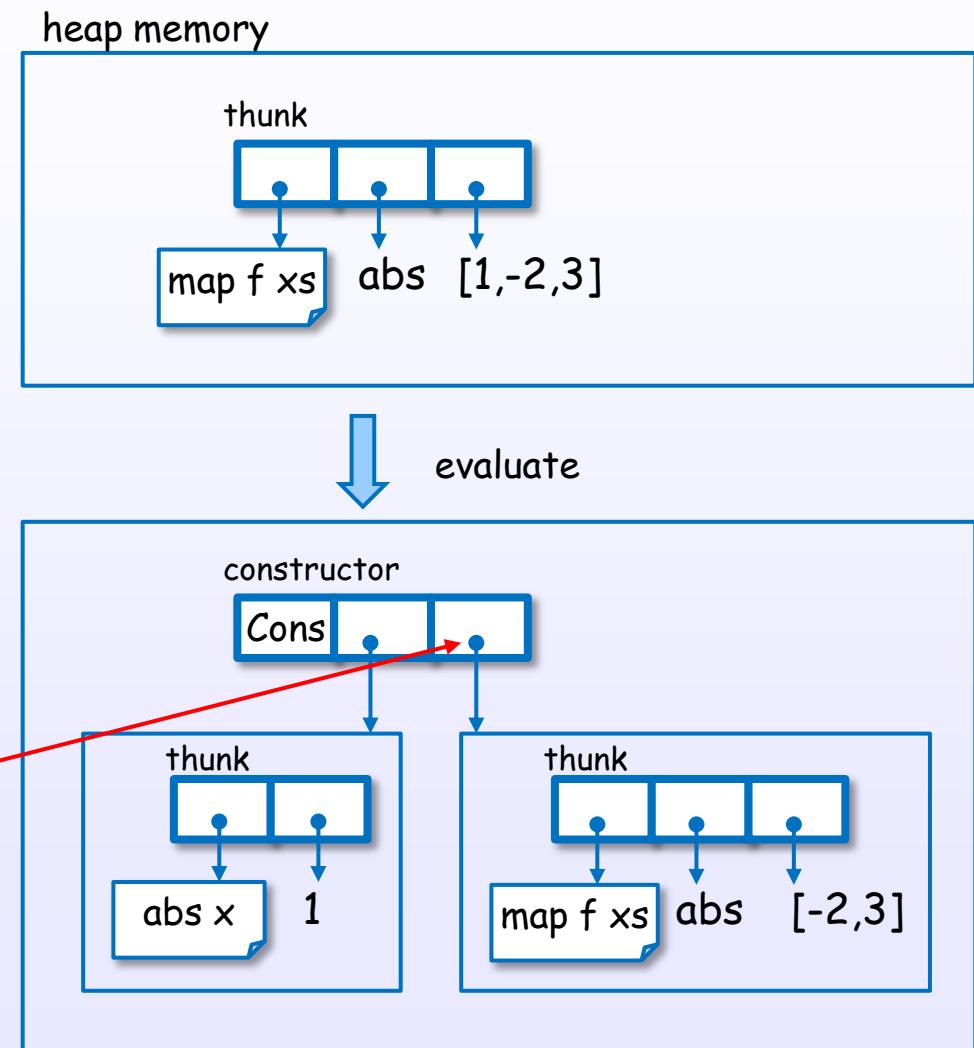
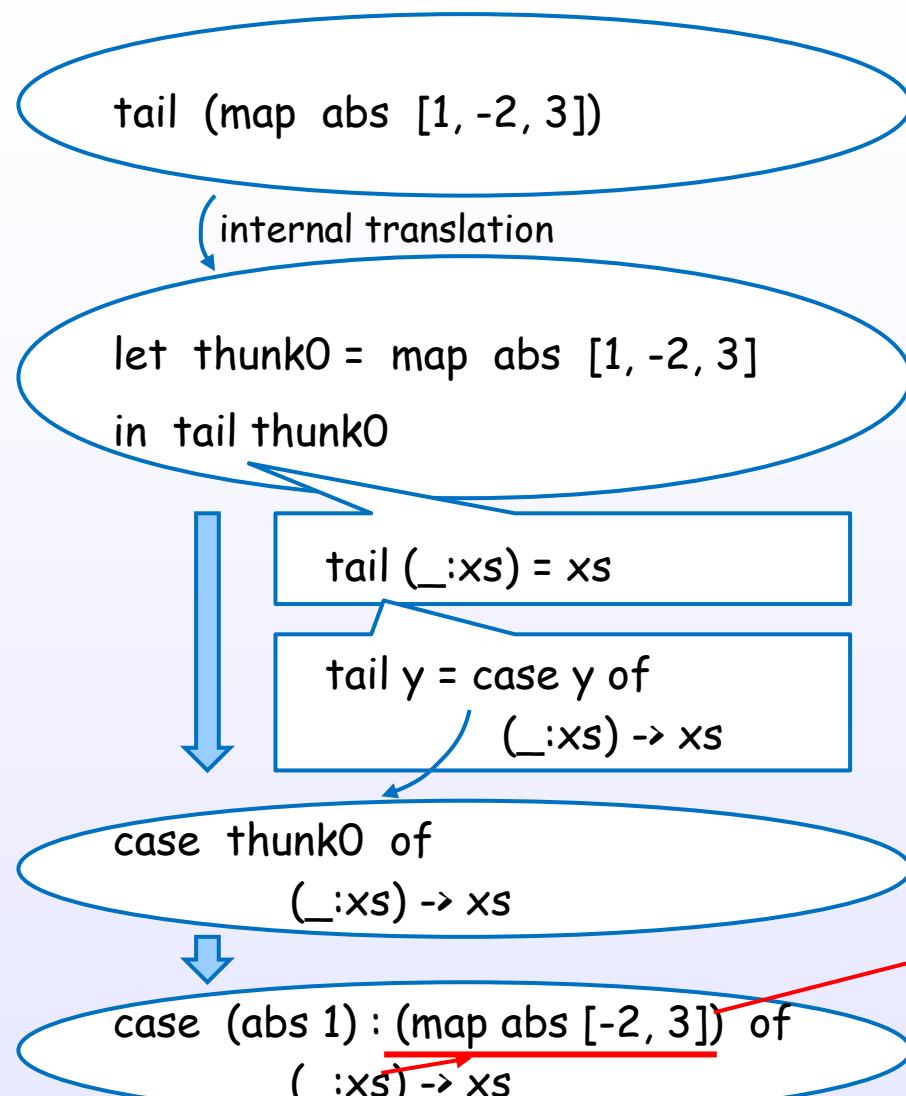
(9) case pattern-matching drives the evaluation



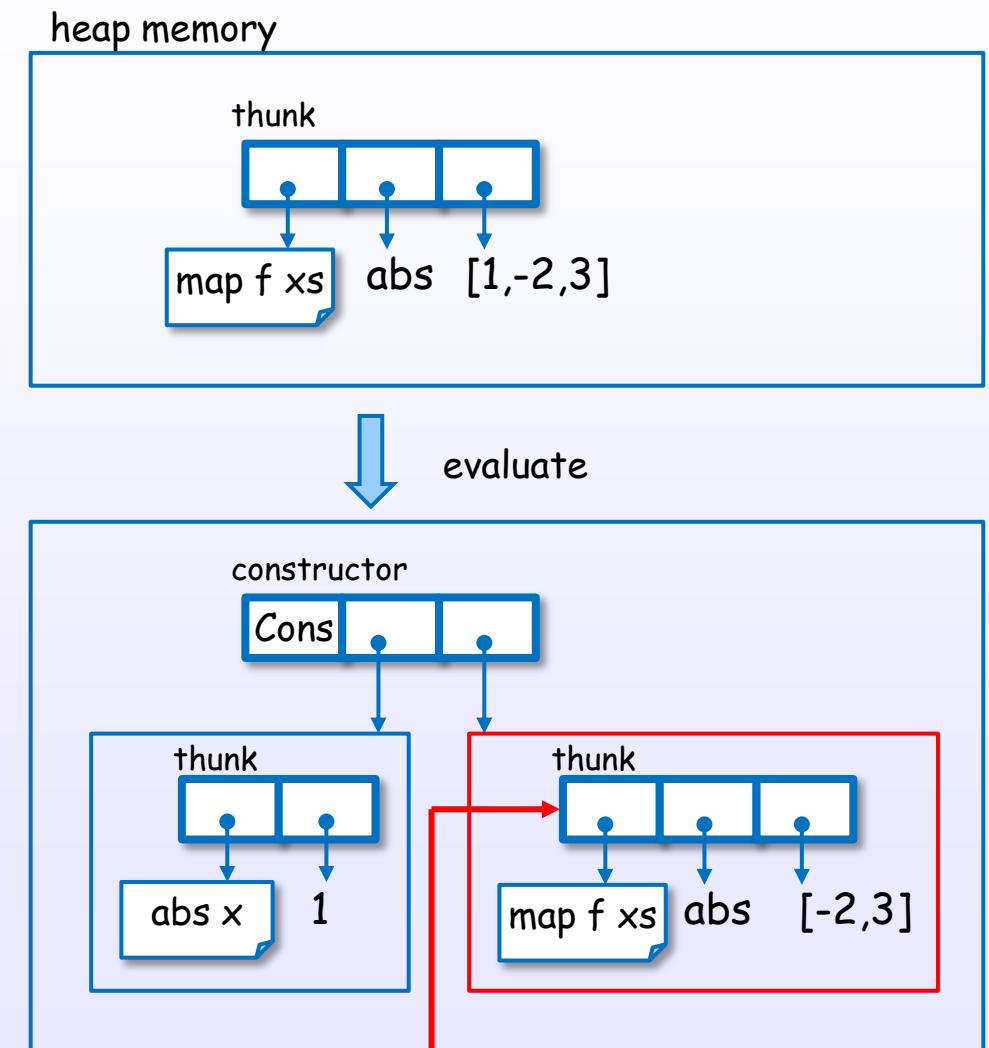
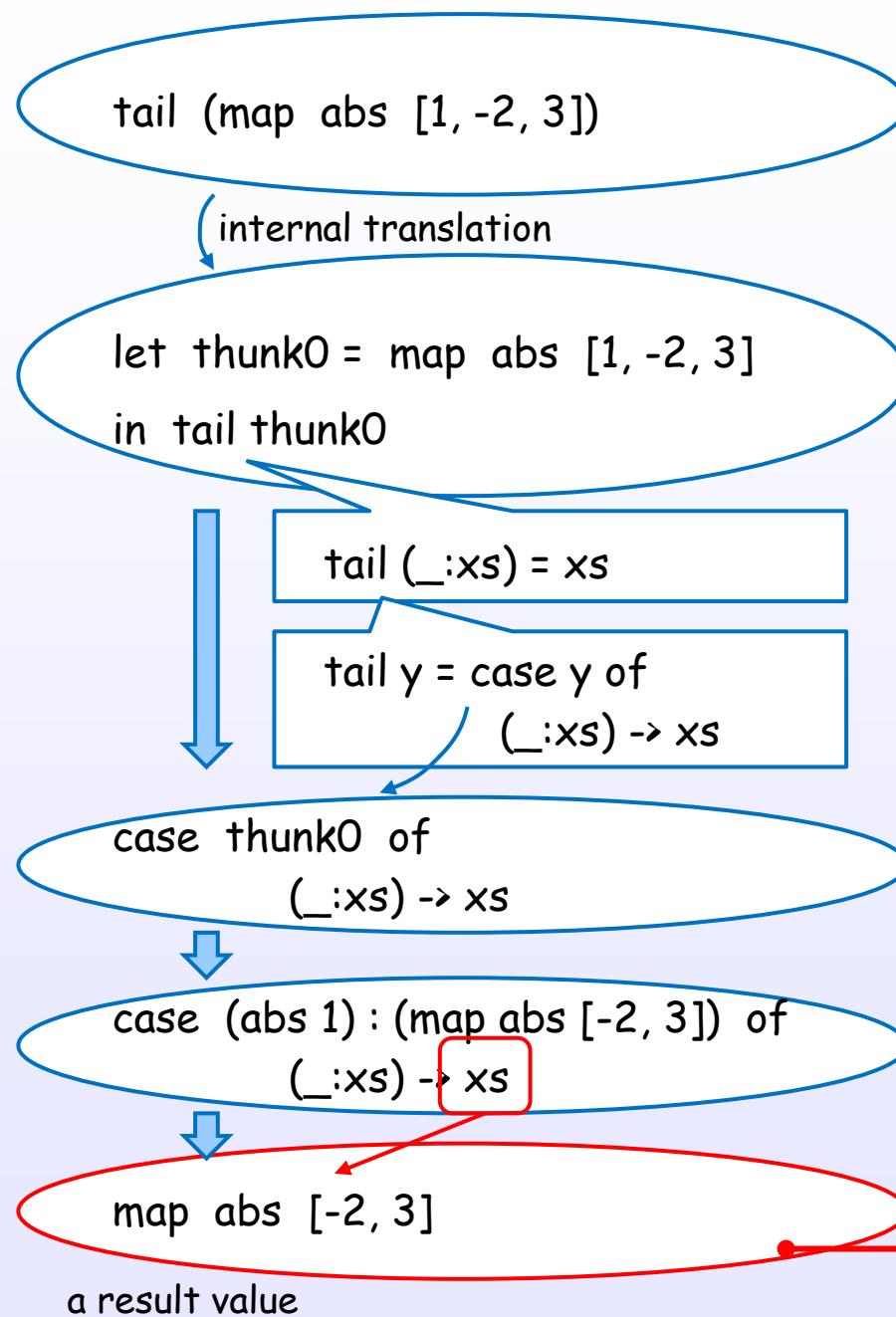
(10) but, stop at WHNF



(11) bind variables to result



(12) return the value



Key points

tail (map abs [1, -2, 3])

internal translation

postpone by thunk

let thunk0 = map abs [1, -2, 3]
in tail thunk0

tail (_:xs) = xs

tail y = case y of
(_:xs) -> xs

case thunk0 of
(_:xs) -> xs

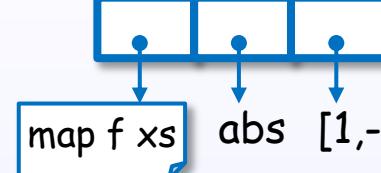
case (abs 1) : (map abs [-2, 3]) of
(_:xs) -> xs

map abs [-2, 3]

a result value

to memory

thunk



map f xs abs [1,-2,3]

evaluate

stop at WHNF

constructor

Cons

thunk



abs x 1

thunk



map f xs abs [-2,3]

4. Evaluation

Examples of evaluations

* no optimizing case (without -O)

Example of repeat

repeat 1



1 : repeat 1



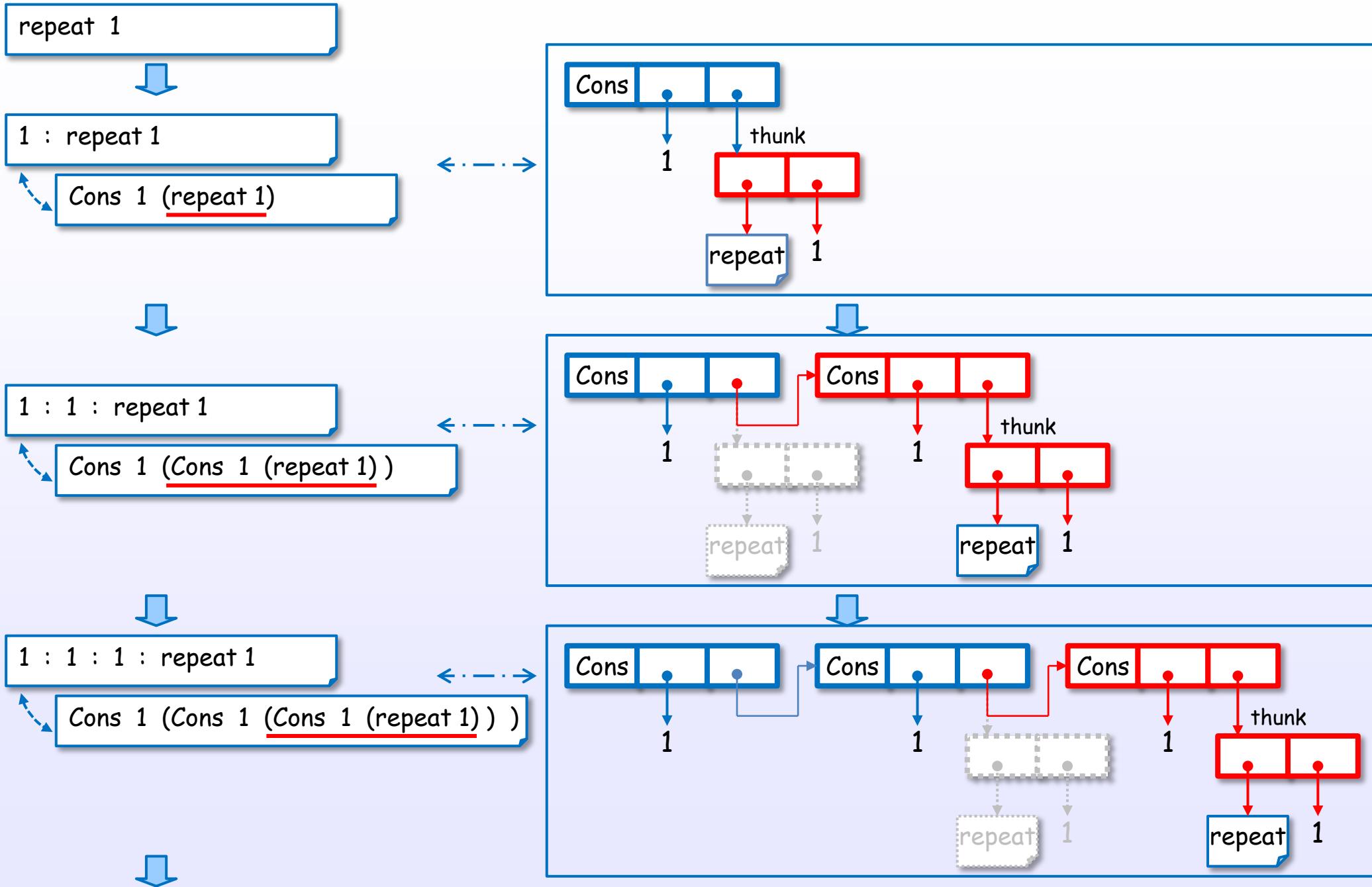
1 : 1 : repeat 1



1 : 1 : 1 : repeat 1



Example of repeat



Example of map

```
map f [1, 2, 3]
```



```
f 1 : map f [2, 3]
```



```
f 1 : f 2 : map f [3]
```

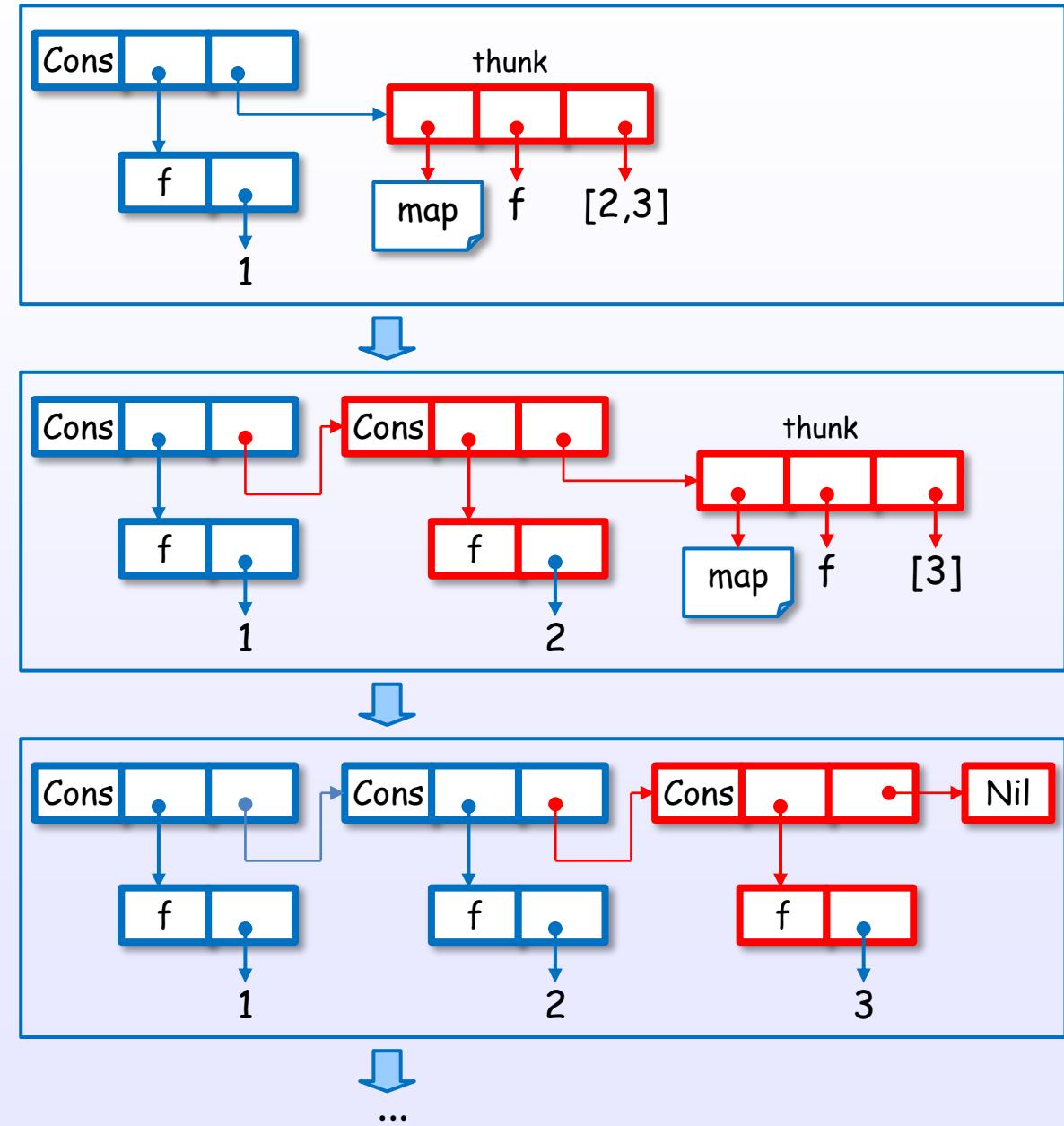
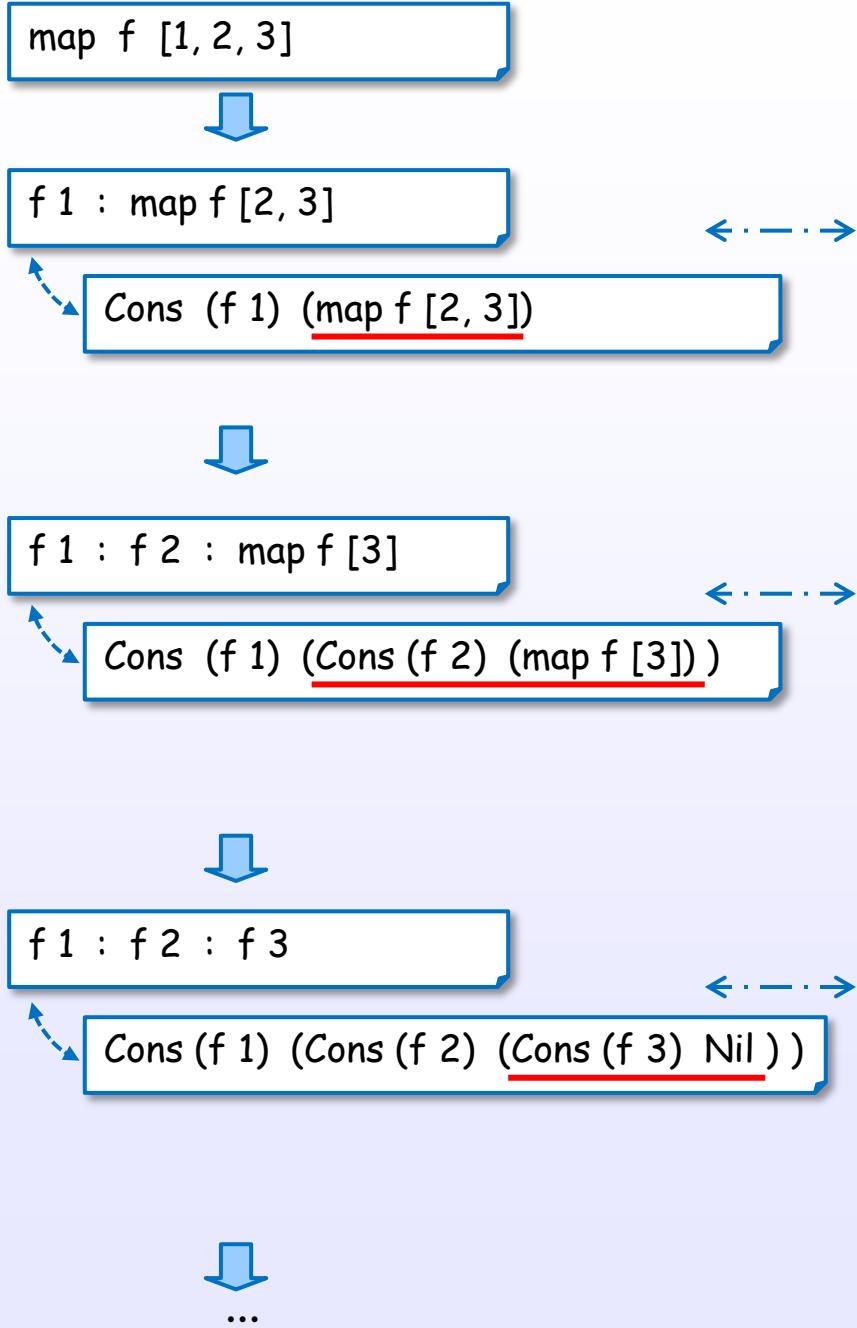


```
f 1 : f 2 : f 3
```



...

Example of map



Example of foldl (non-strict)

```
foldl (+) 0 [1 .. 100]
```



```
foldl (+) (0 + 1) [2 .. 100]
```



```
foldl (+) (((0 + 1) + 2) [3 .. 100]
```



```
foldl (+) (((0 + 1) + 2) + 3) [4 .. 100]
```



...

Example of foldl (non-strict)

`foldl (+) 0 [1 .. 100]`



`foldl (+) (0 + 1) [2 .. 100]`

`let thunk1 = (0 + 1)
in foldl (+) thunk1 [2 .. 100]`



`foldl (+) ((0 + 1) + 2) [3 .. 100]`

`let thunk2 = (thunk1 + 2)
in foldl (+) thunk2 [3 .. 100]`



`foldl (+) (((0 + 1) + 2) + 3) [4 .. 100]`

`let thunk3 = (thunk2 + 3)
in foldl (+) thunk3 [4 .. 100]`

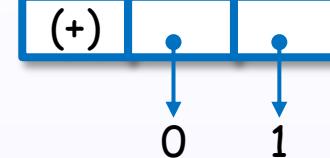


...

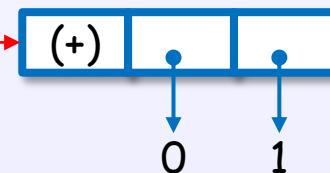
heap memory

*show only accumulation value

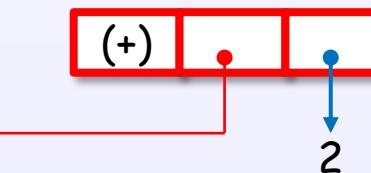
thunk1



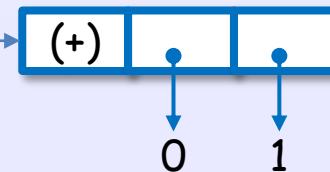
thunk1



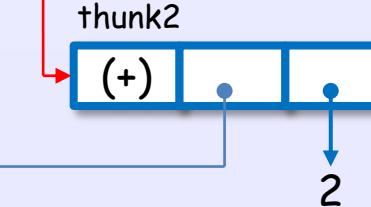
thunk2



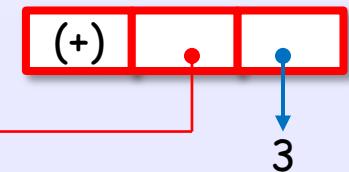
thunk1



thunk2



thunk3



increasing heap ...



References : [D5], [D6], [D8], [D9], [D10], [H10]

Example of foldl' (strict)

```
foldl' (+) 0 [1 .. 100]
```



```
foldl' (+) (0 + 1) [2 .. 100]
```



```
foldl' (+) (1 + 2) [3 .. 100]
```



```
foldl' (+) (3 + 3) [4 .. 100]
```



...

Example of foldl' (strict)

`foldl' (+) 0 [1 .. 100]`



`foldl' (+) (0 + 1) [2 .. 100]`

```
let thunk1 = (0 + 1)
in thunk1 `seq`
  foldl' (+) thunk1 [2 .. 100]
```



`foldl' (+) (1 + 2) [3 .. 100]`

```
let thunk2 = (1 + 2)
in thunk2 `seq`
  foldl' (+) thunk2 [3 .. 100]
```

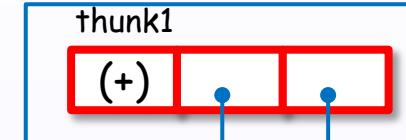


`foldl' (+) (3 + 3) [4 .. 100]`

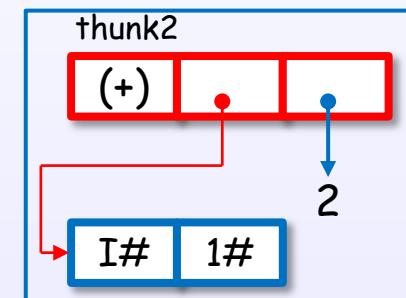
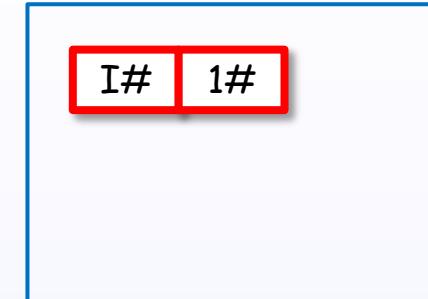
```
let thunk3 = (3 + 3)
in thunk3 `seq`
  foldl' (+) thunk3 [4 .. 100]
```



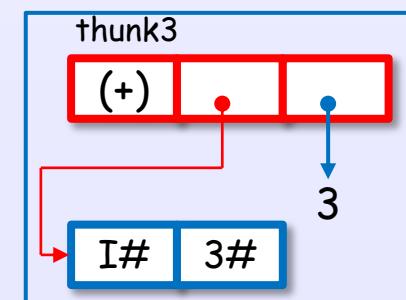
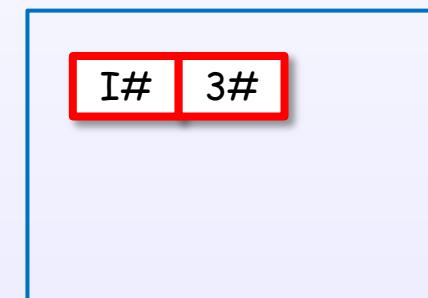
heap memory



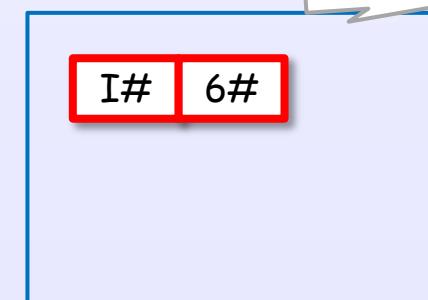
→
update
by seq



→



→



fixed heap size

...

References : [D5], [D6], [D8], [D9], [D10], [H10]

Example of foldl (non-strict) and foldl' (strict)

foldl (+) (0 + 1) [2 .. 100]



foldl' (+) (0 + 1) [2 .. 100]

foldl (+) ((0 + 1) + 2) [3 .. 100]



foldl' (+) (1 + 2) [3 .. 100]



foldl (+) (((0 + 1) + 2) + 3) [4 .. 100]



foldl' (+) (3 + 3) [4 .. 100]

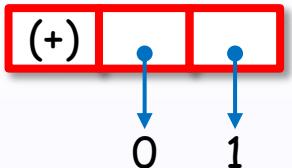


References : [D5], [D6], [D8], [D9], [D10], [H10]

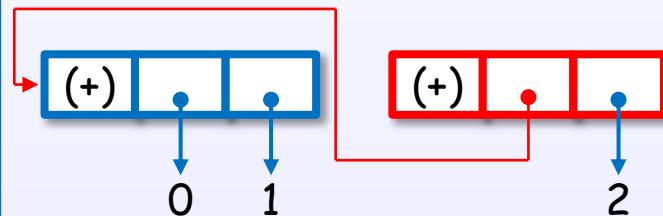
Example of foldl (non-strict) and foldl' (strict)

foldl (+) (0 + 1) [2 .. 100]

heap memory

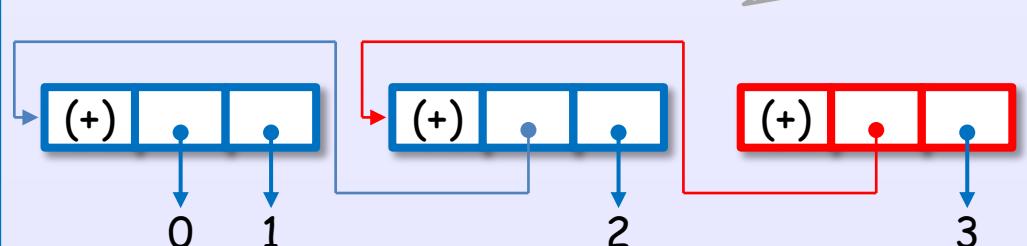


foldl (+) ((0 + 1) + 2) [3 .. 100]

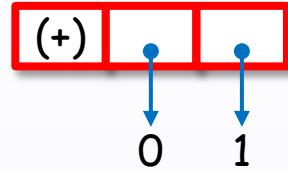


foldl (+) (((0 + 1) + 2) + 3) [4 .. 100]

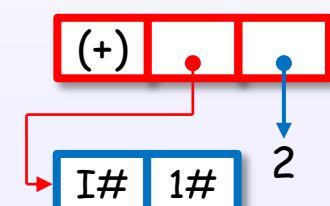
increasing heap ...



foldl' (+) (0 + 1) [2 .. 100]

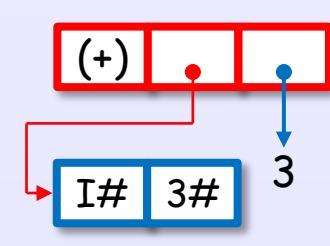


foldl' (+) (1 + 2) [3 .. 100]



foldl' (+) (3 + 3) [4 .. 100]

fixed heap size

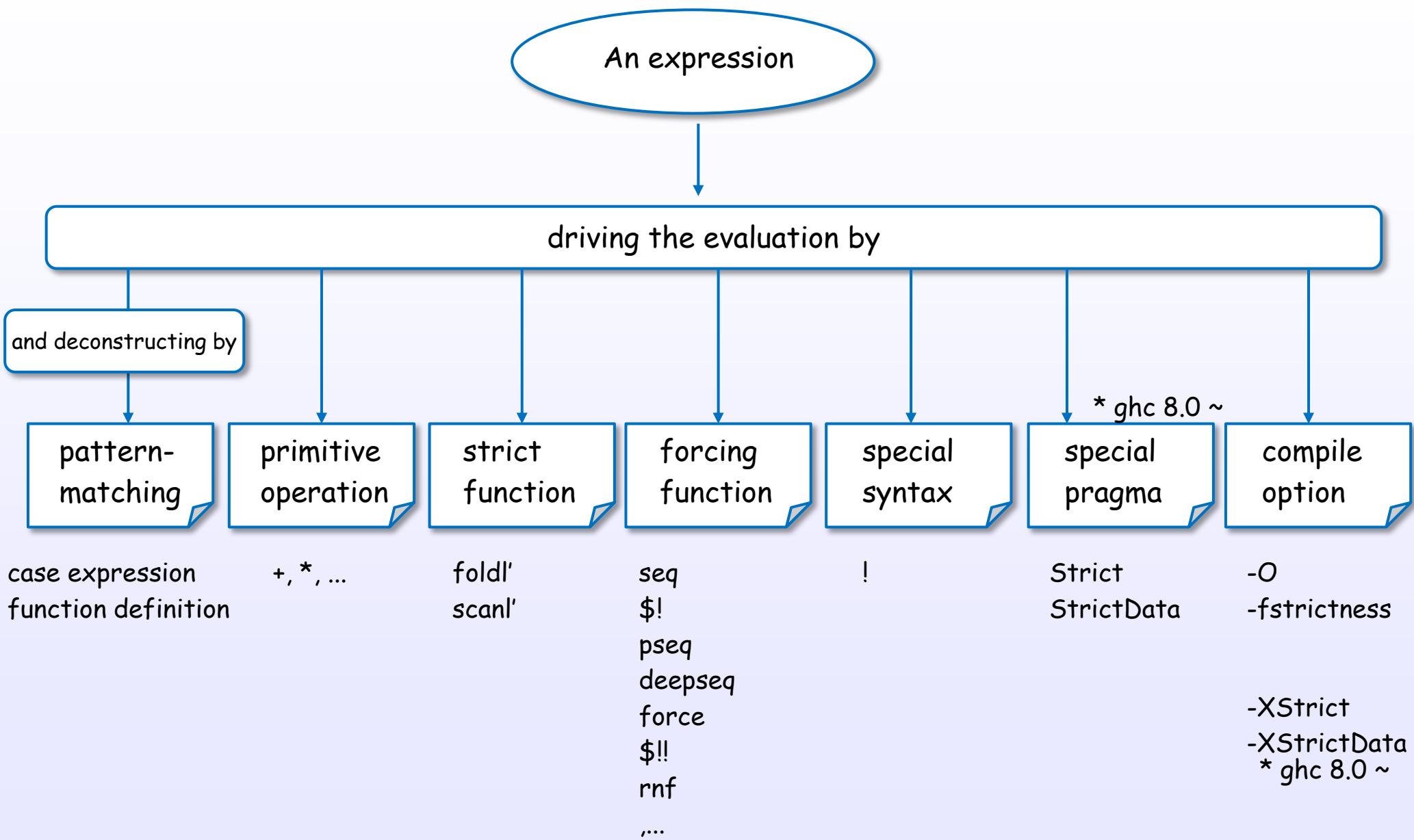


References : [D5], [D6], [D8], [D9], [D10], [H10]

4. Evaluation

Controlling the evaluation

How to drive the evaluation



(1) Evaluation by pattern-matching

pattern-matching in **case expression**

```
case ds of
  x:xs -> f x xs
  []      -> False
```

forcing
(drive the evaluation of the thunk)

pattern-matching in **function definition**

```
f Just _ = True
f Nothing = False
```

forcing
(drive the evaluation of the thunk)

(1) Evaluation by pattern-matching

Strict patterns drive the evaluation

case expression

```
case ds of
  x:xs -> f x xs
  []      -> False
```

Lazy patterns postpone the evaluation.

let binding pattern

```
let (x:xs) = fun args
```

function definition

```
f Just _ = True
f Nothing = False
```

irrefutable patterns [H1] 3.17

```
f ~(Just _) = True
f ~(Nothing) = False
```

There are two kinds of pattern-matching.

(2) Evaluation by primitive operation

primitive (built-in) operation

$$f \ x \ y = x + y$$

$+, *, \dots$

forcing x and y
(drive the evaluation of the thunks)

primitive operations are defined such as

* pseudo code

$$(+)(I\# a)(I\# b) = I\# (a+b)$$

pattern-matching

(3) Evaluation by strict version function

strict version function

foldl' (+) 0 xs

strict application of the operator

scanl' (+) 0 xs

(4) Evaluation by forcing function

forcing functions to **WHNF**

`seq x y`

`f $! x`

`pseq x y`

forcing
(drive the evaluation of the thunk)

forcing functions to **NF**

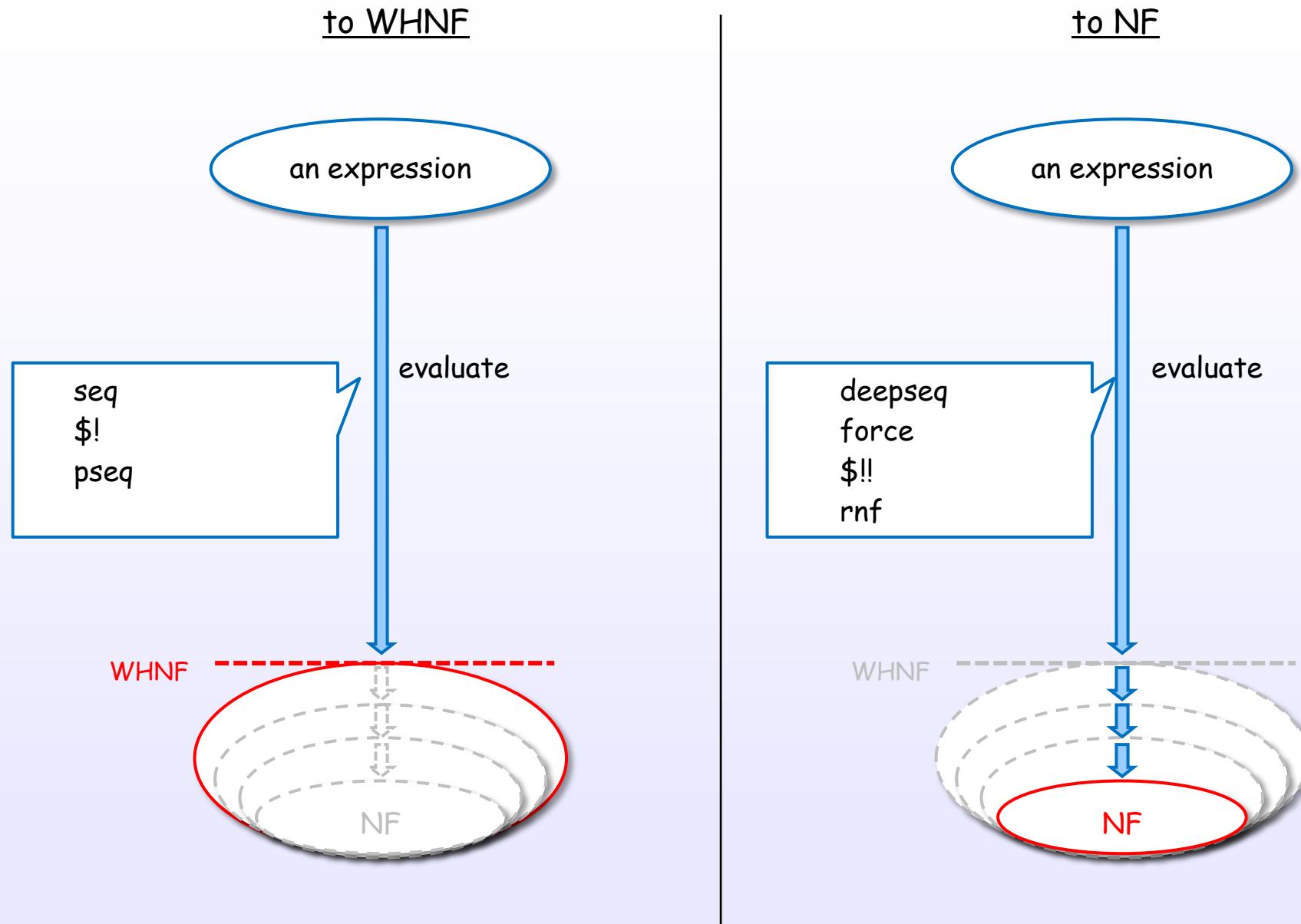
`deepseq x y`

`f $!! x`

`force x`

`rnf x`

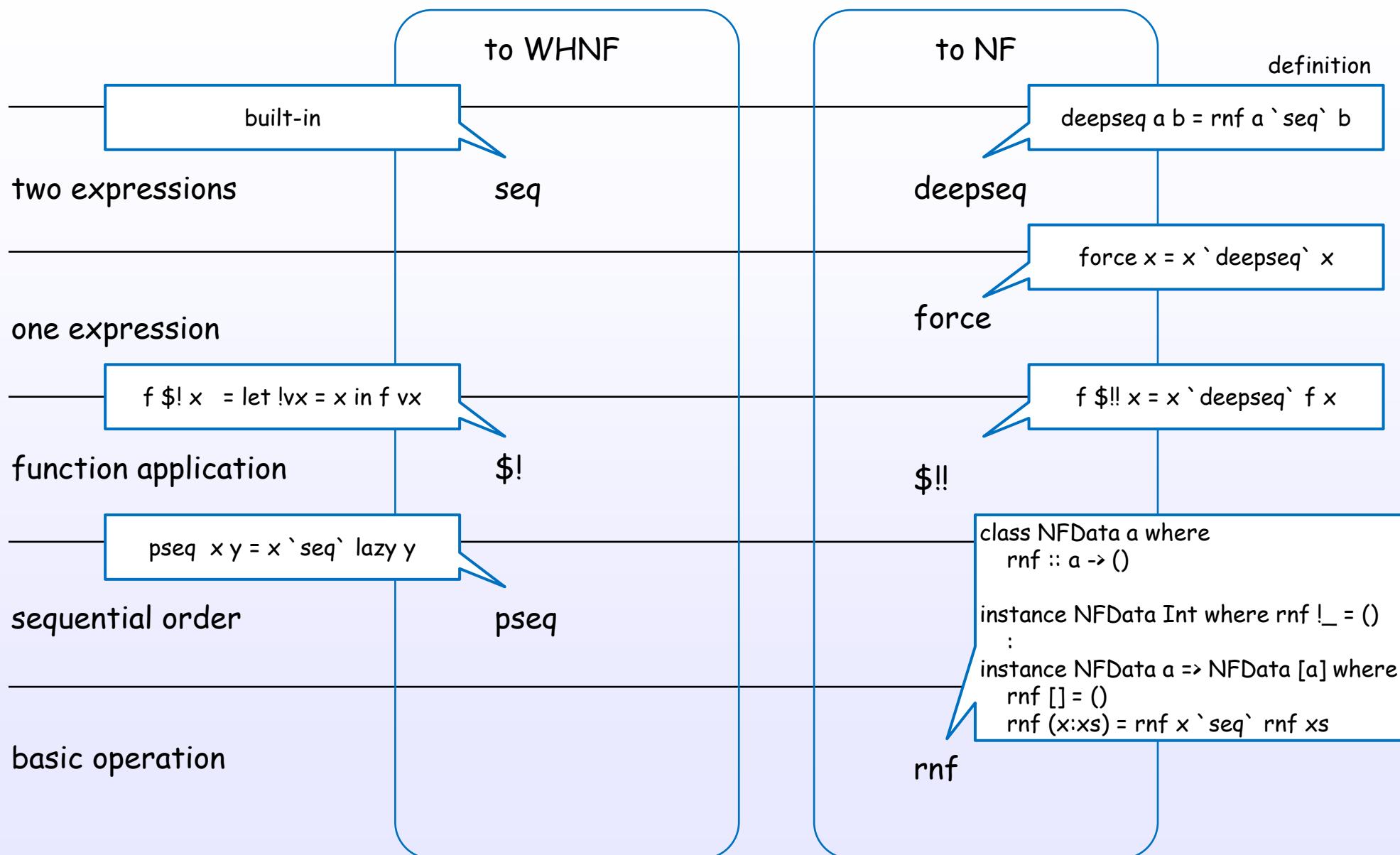
(4) Evaluation by forcing function



(4) Evaluation by forcing function

	to WHNF	to NF
two arguments	seq	deepseq
one argument		force
function application	\$!	\$!!
sequential order	pseq	
basic operation		rnf

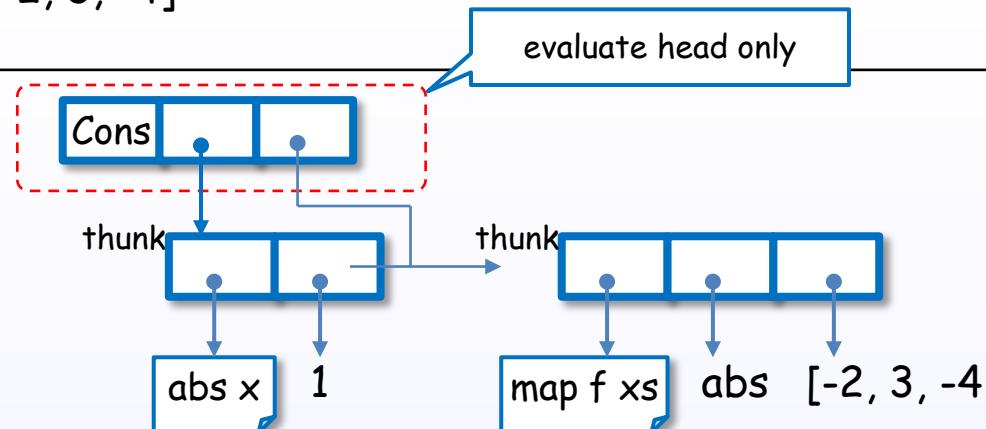
(4) Evaluation by forcing function



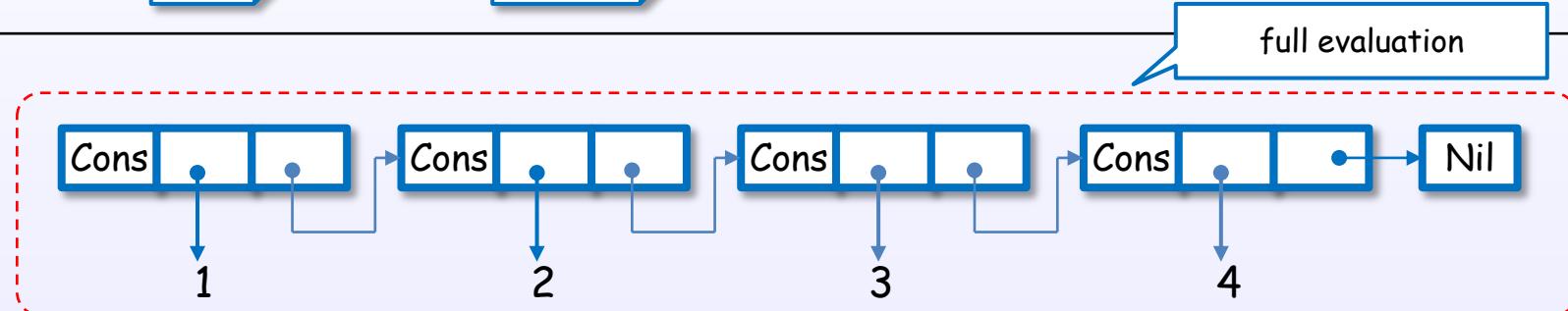
(4) Evaluation by forcing function

`a = map abs [1, -2, 3, -4]`

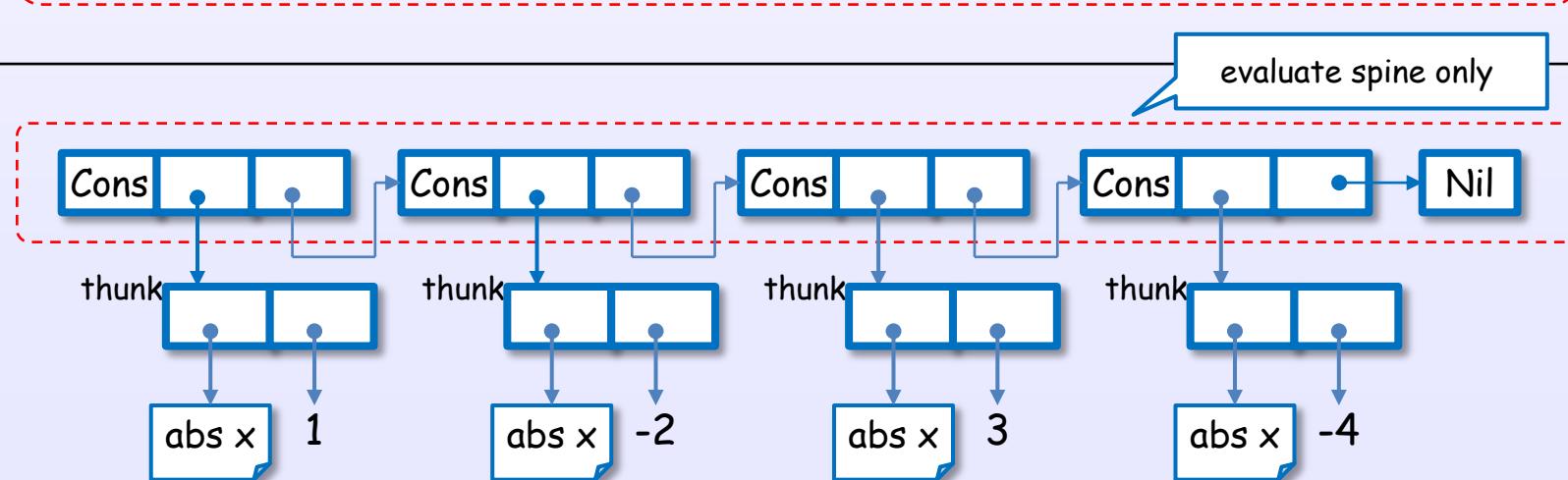
`seq a ()`



`deepseq a ()`



`length a`



(5) Evaluation by special syntax

Strictness annotation

Bang pattern [H2] 7.19

see also Strict pragma

{-# LANGUAGE BangPatterns #-}

f !xs = g xs

arguments are evaluated
before function application

Strictness flag [H1] 4.2.1

see also StrictData and Strict pragma

data Pair = Pair !a !b

arguments are evaluated
before constructor application

Strictness annotations assist strictness analysis.

(6) Evaluation by special pragma

Special pragma for strictness language extension

Strict pragma

* ghc 8.0 ~

see also bang pattern and strictness flag

{-# LANGUAGE Strict #-}

let f xs = g xs in f ys

data Pair = Pair a b

arguments are evaluated
before application

StrictData pragma

* ghc 8.0 ~

see also strictness flag

{-# LANGUAGE StrictData #-}

data Pair = Pair a b

Strict and StrictData pragmas are module level control.

These can use in ghc 8.0 or later.

(7) Evaluation by compile option

Compile option

strictness analysis

```
$ ghc -O
```

Turn on optimization.
Implies "-fstrictness".

```
$ ghc -fstrictness
```

Turn on strictness analysis.
Implied by "-O".

strictness language extension * ghc 8.0 ~

```
$ ghc -XStrict
```

apply Strict pragma

```
$ ghc -XStrictData
```

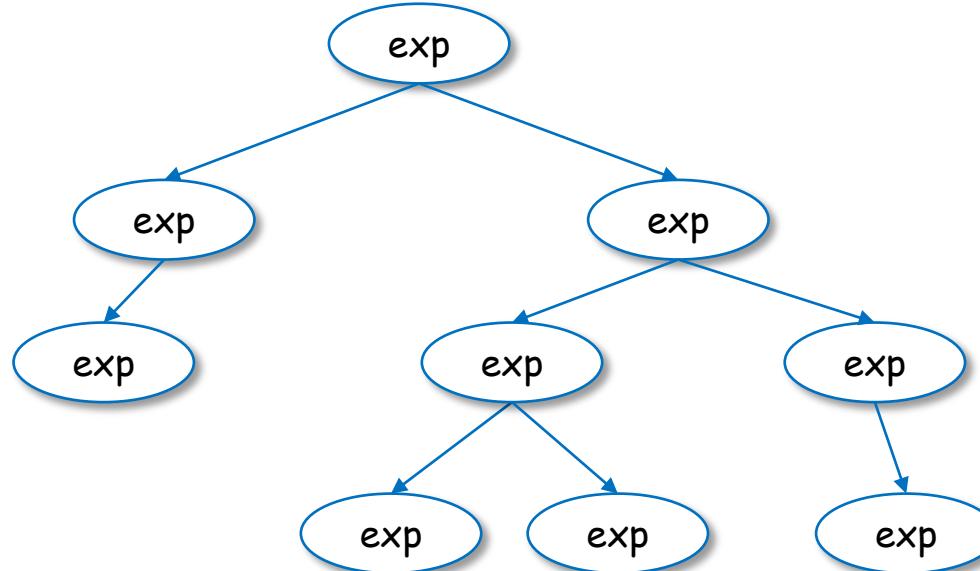
apply StrictData pragma

5. Implementation of evaluator

5. Implementation of evaluator

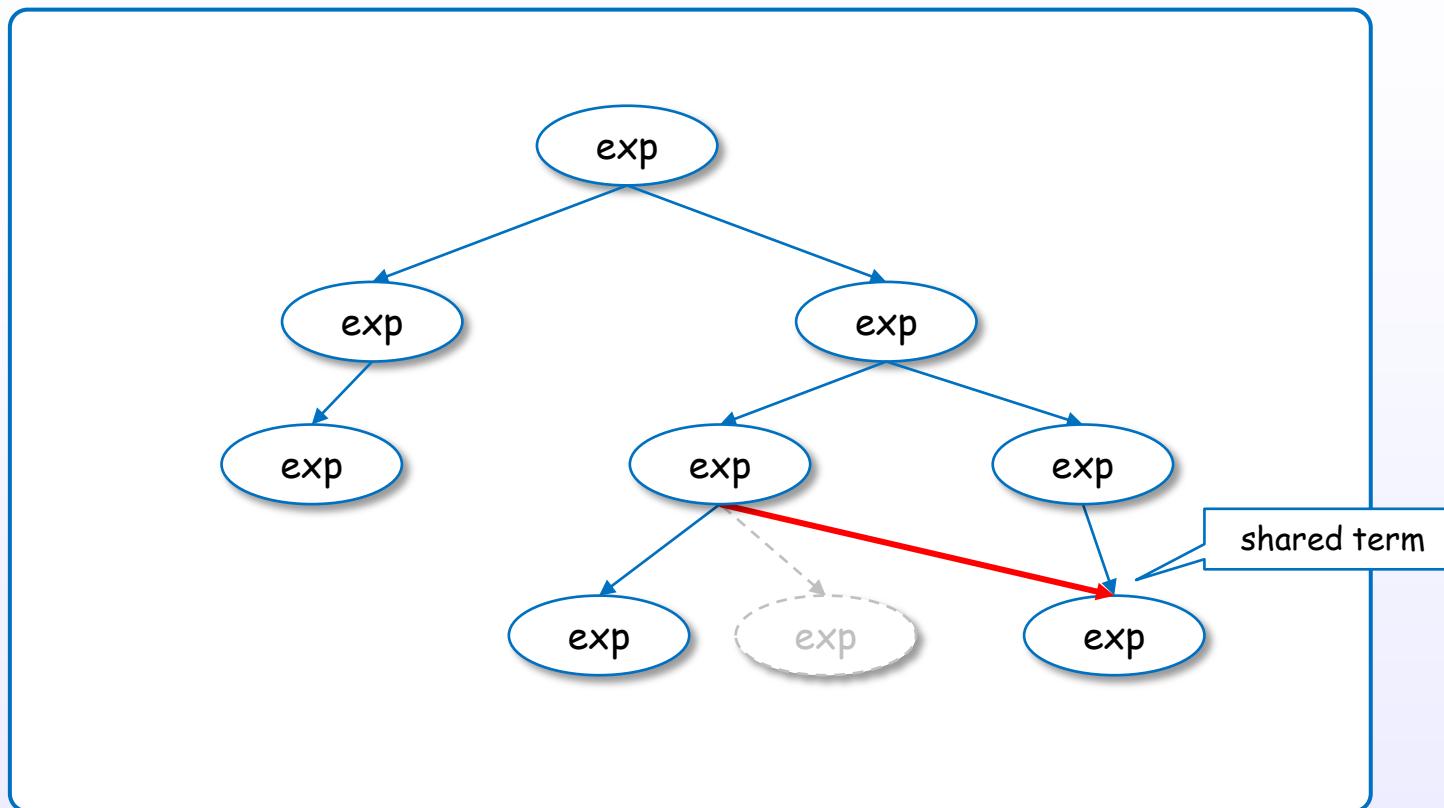
Lazy graph reduction

Tree



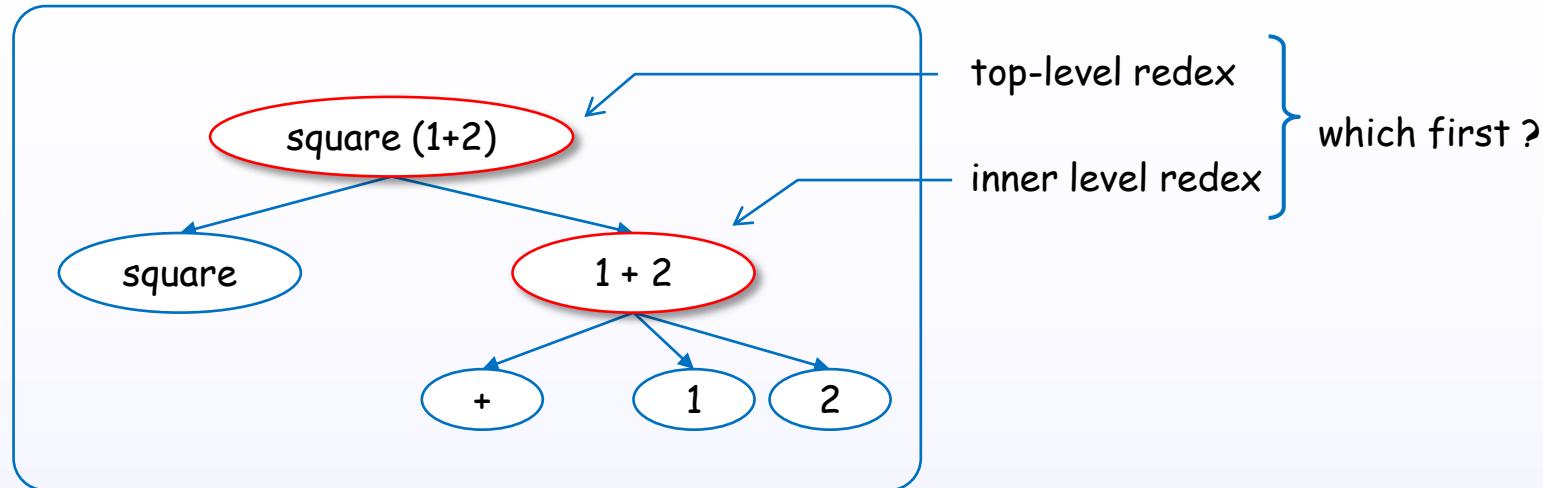
An expression can be represented in the form of Abstract Syntax **Tree** (AST). AST is reduced using stack (sequential access memory).

Graph



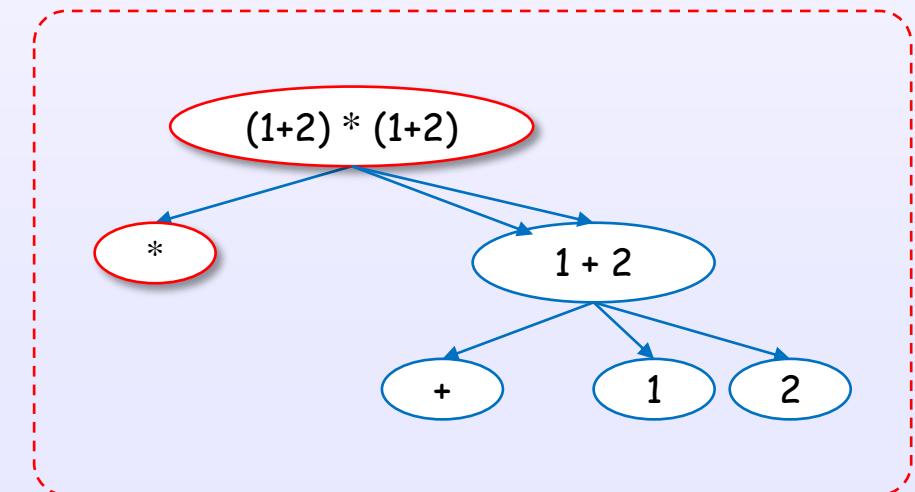
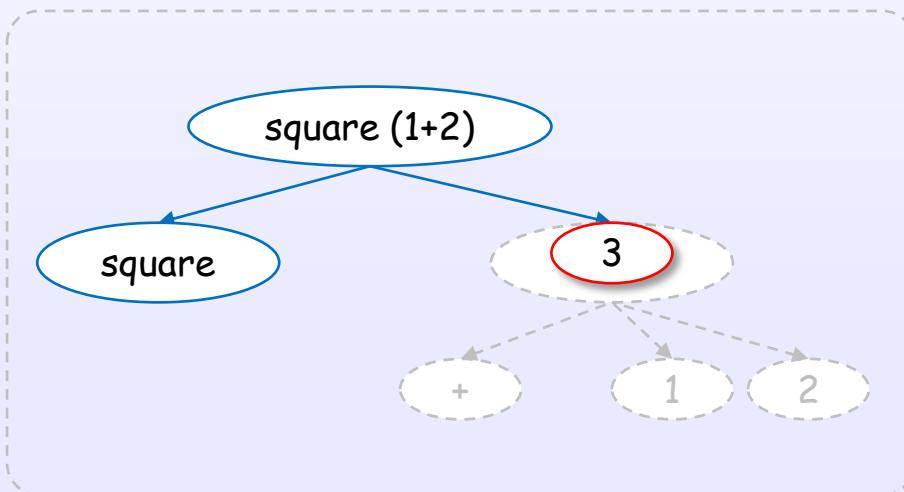
An expression can be also represented in the form of Graph.
Graph can share subexpressions to evaluate at once.
So, graph is reduced using heap (random access memory) rather than stack.

Graph can be reduced some order



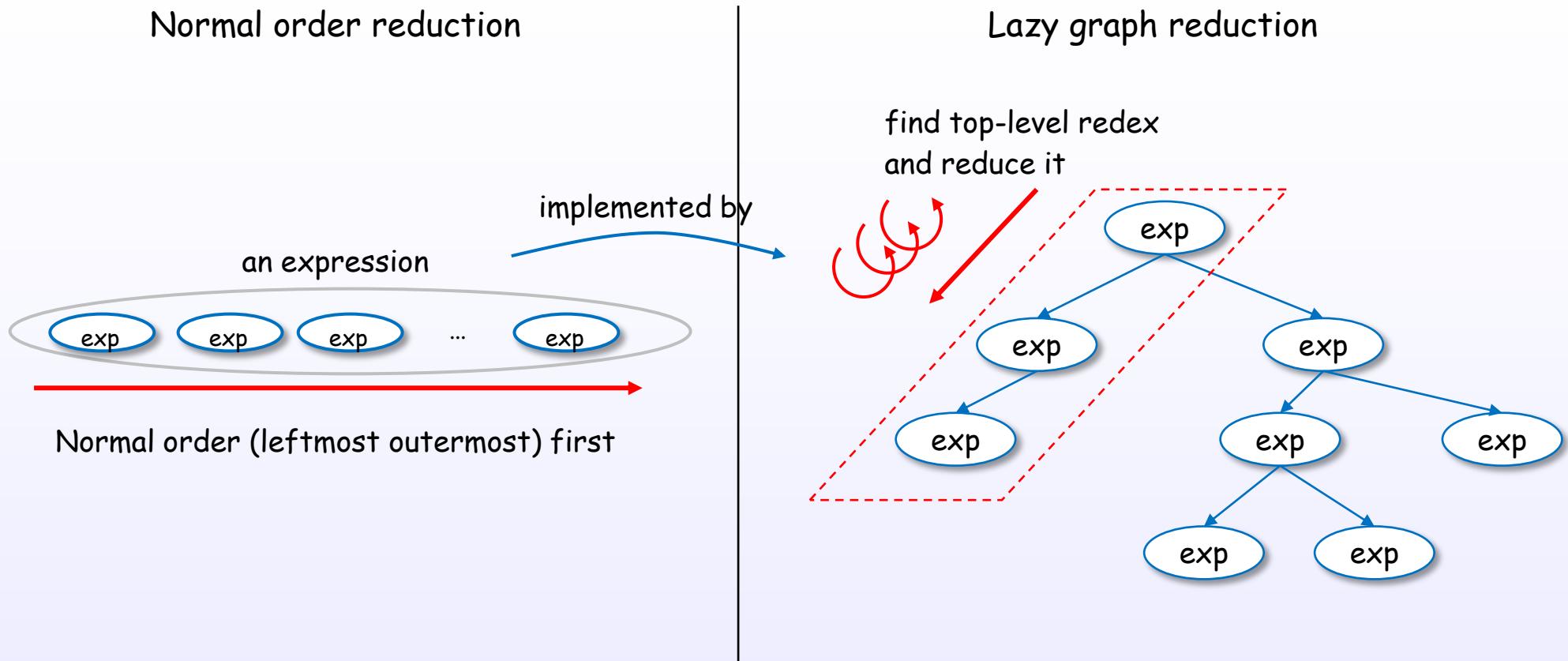
reduce inner level first
for call-by-value

reduce top-level (outermost) first
for call-by-need



To select top-level redex first, the evaluation of arguments can be postponed.

Normal order reduction is implemented by lazy graph reduction



Normal order (leftmost outermost) reduction is implemented by lazy graph reduction to select top-level redex first.

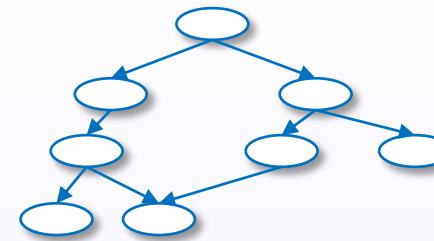
Given an application of a function, the outermost redex is the function application itself.

5. Implementation of evaluator

STG-machine

Abstract machine

Graph
(expression)



evaluate
(reduce / execute)

STG-machine

Evaluator
(abstract machine)

GHC uses abstract machine to reduce the expression.
It's called "STG-machine".

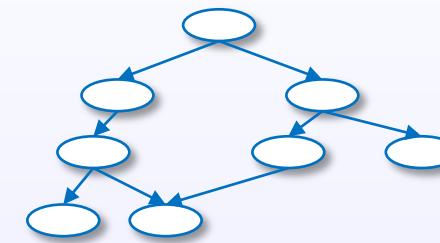
Concept layer

Haskell code

`take 5 [1..10]`

:

Graph
(internal representation
of the expression)



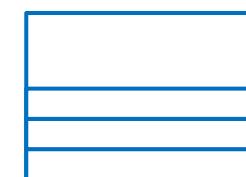
Evaluator (reducer, executer)
(abstract machine)

STG-machine

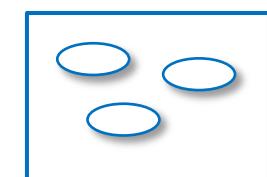
STG Registers

R1, ...

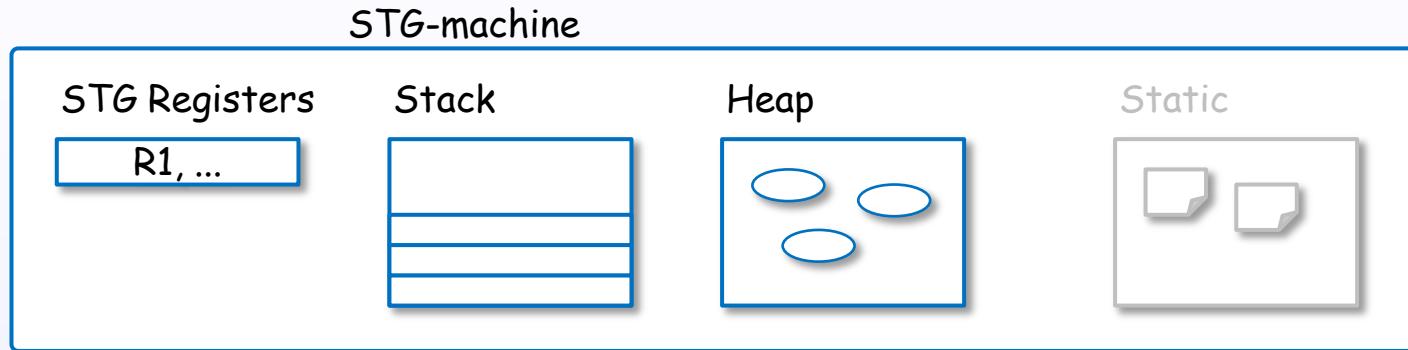
Stack



Heap



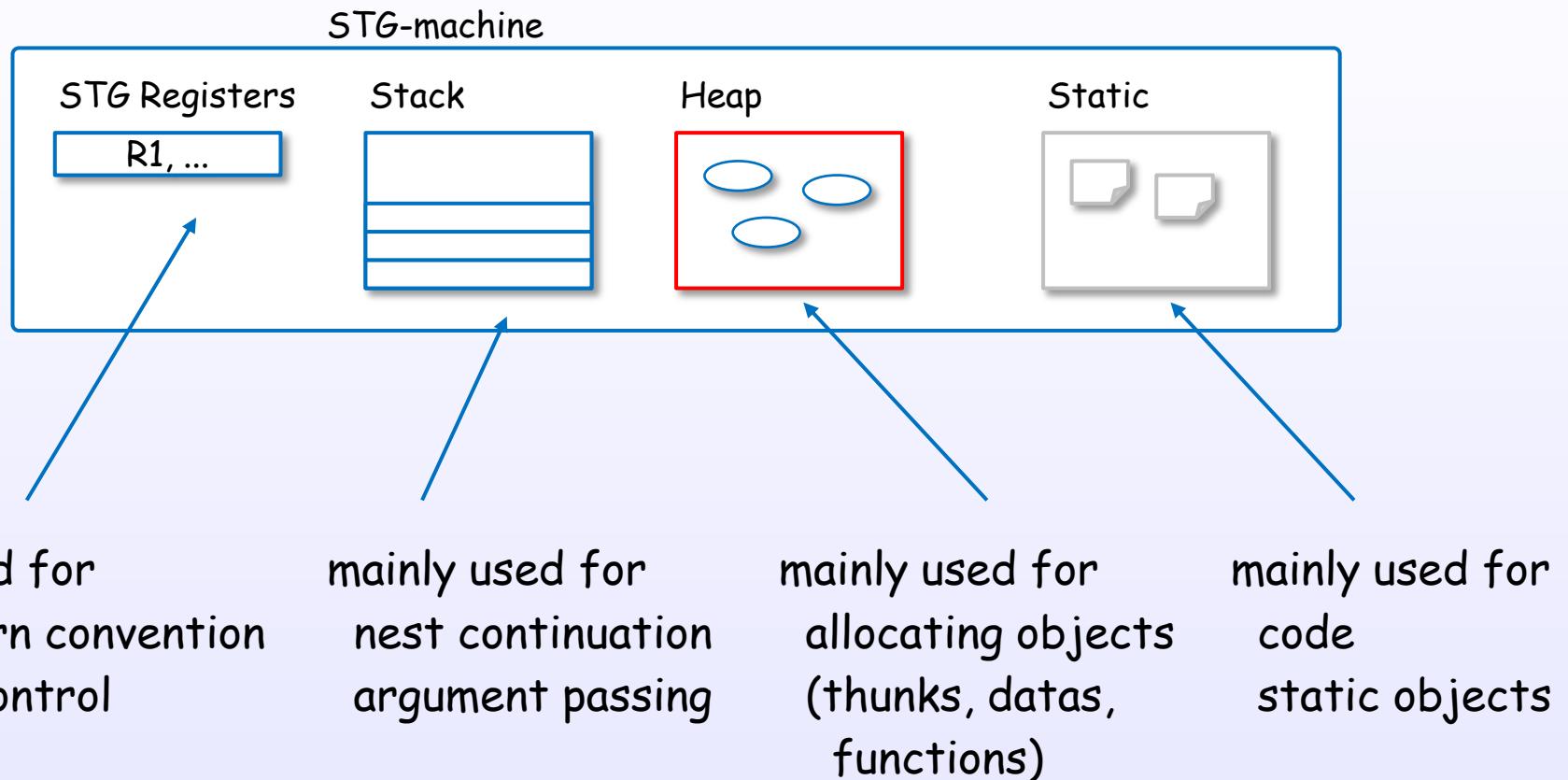
STG-machine



STG-machine is abstraction machine
which is defined by operational semantics.

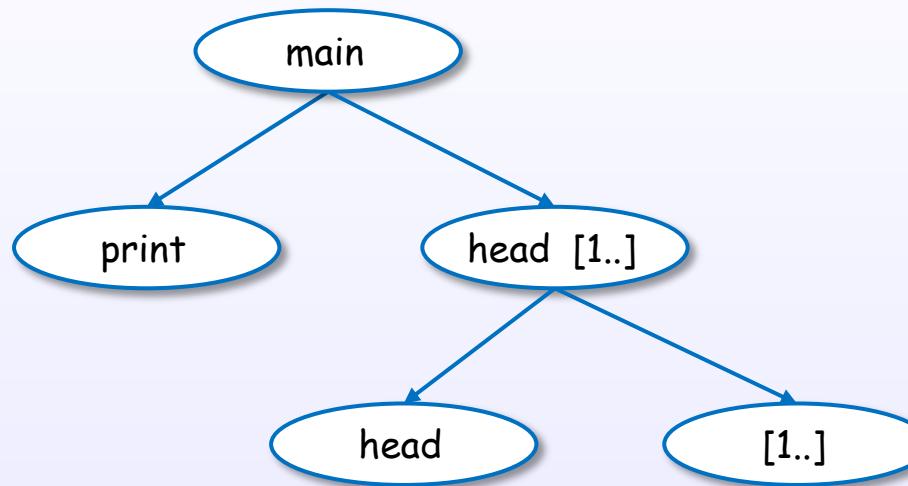
STG-machine efficiently performs lazy graph reduction.

STG-machine



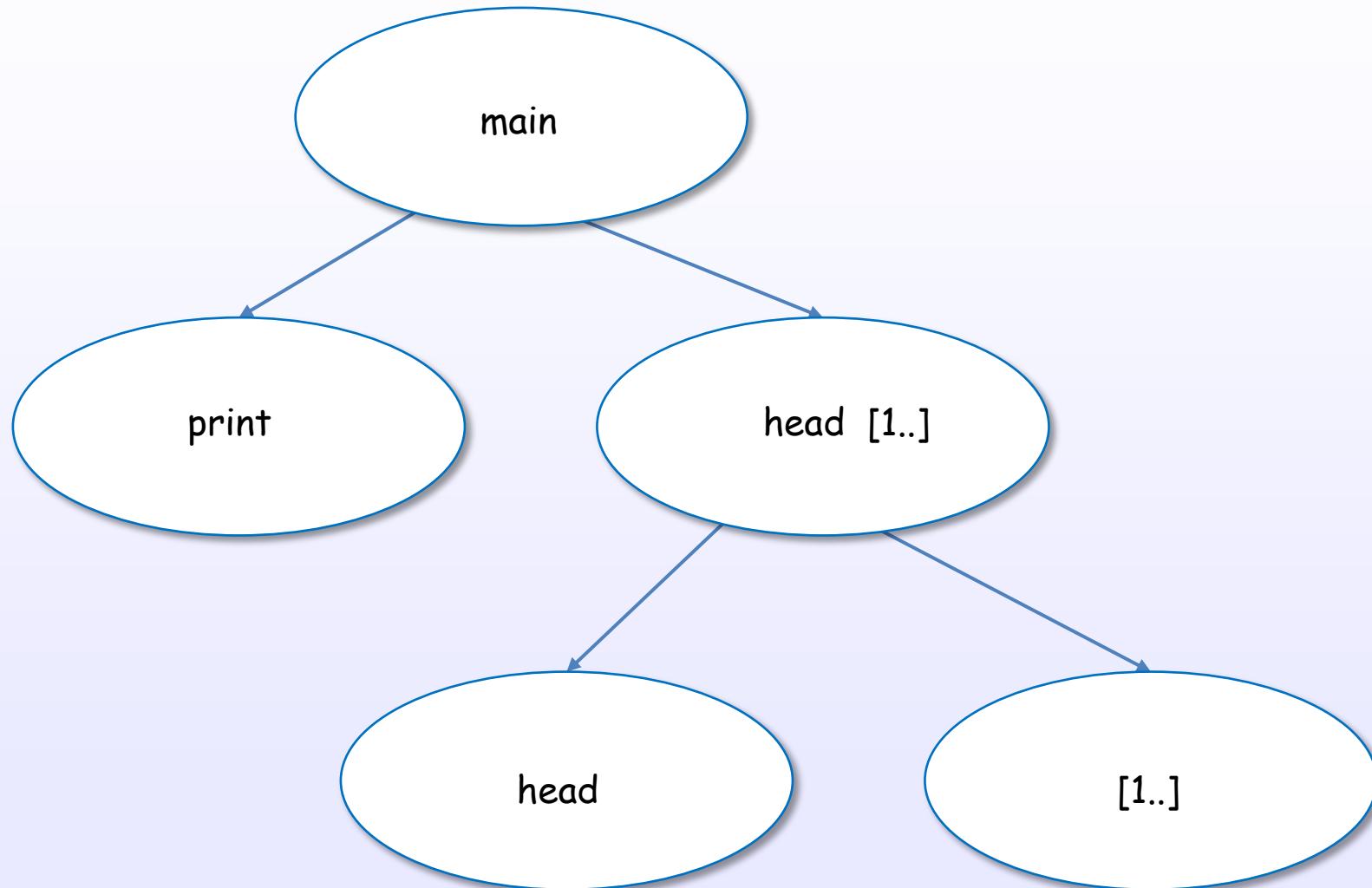
Example of mapping a code to a graph

main = print (head [1..])



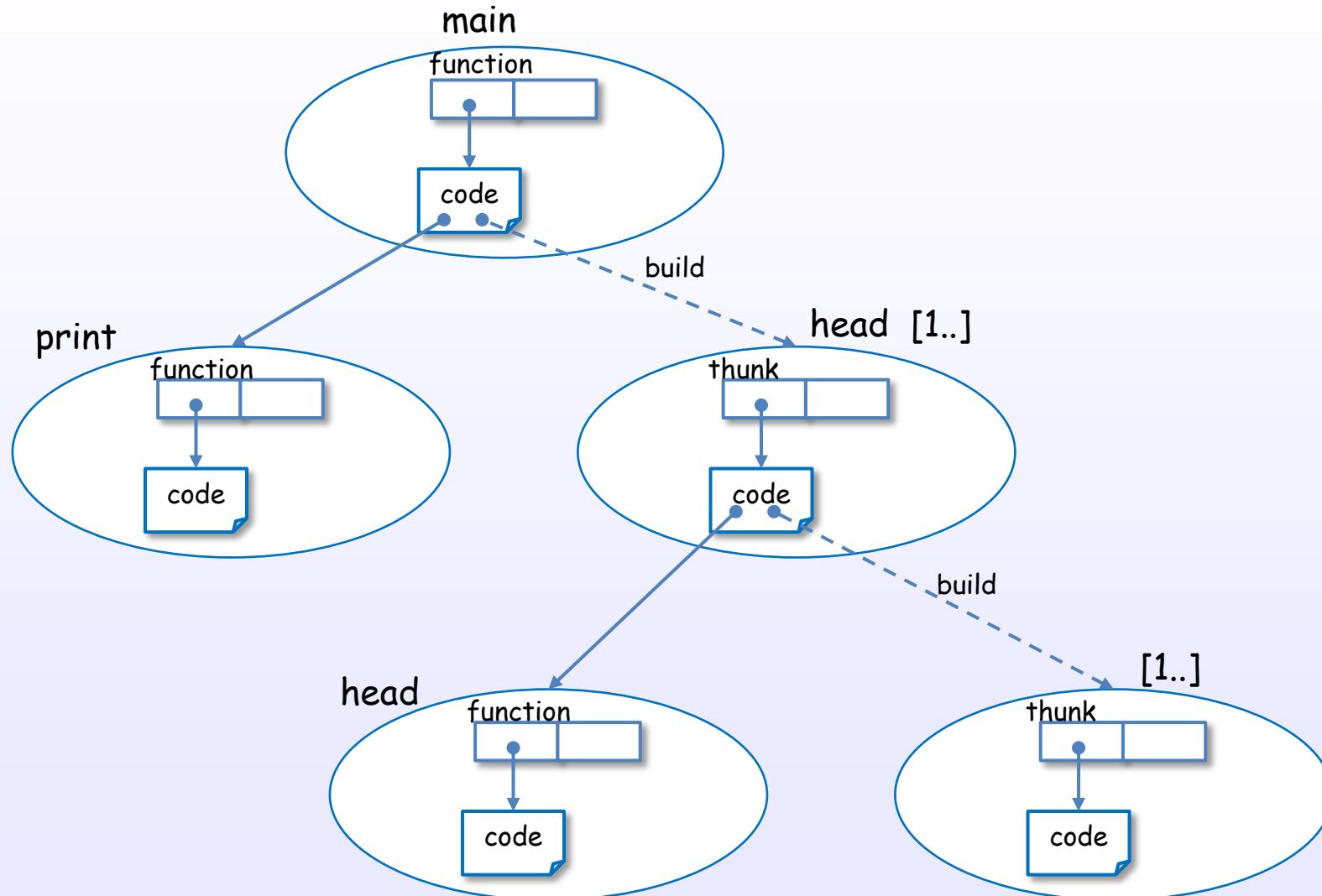
Example of mapping a code to a graph

main = print (head [1..])



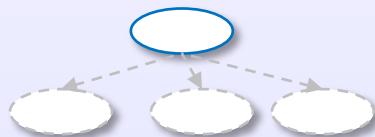
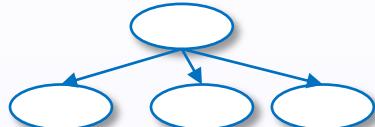
Example of mapping a code to a graph

main = print (head [1..])



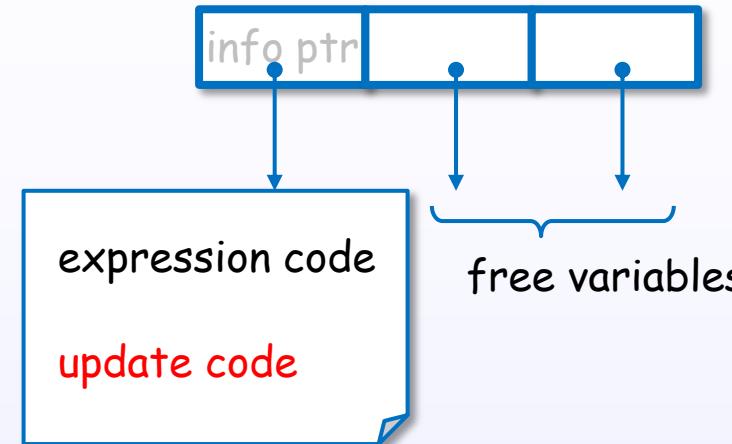
Self-updating model

Expression



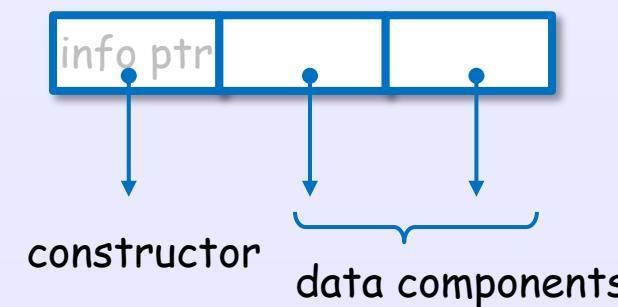
GHC's internal representation

a thunk



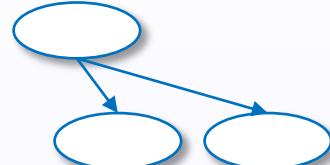
evaluate and update
(replace myself to result value)

a data value

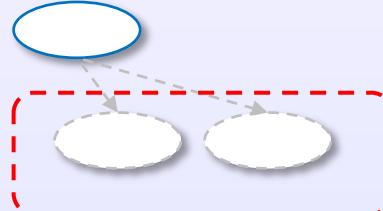


Unreferenced expressions (objects) will be removed by GC

Expression

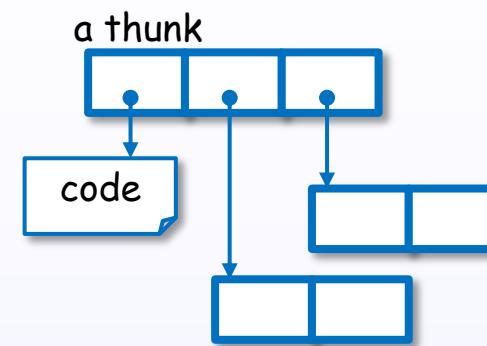


evaluate and update



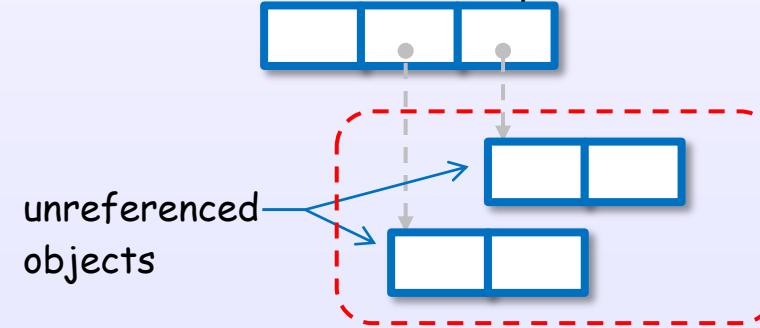
collected by GC(Garbage Collection)

GHC's internal representation



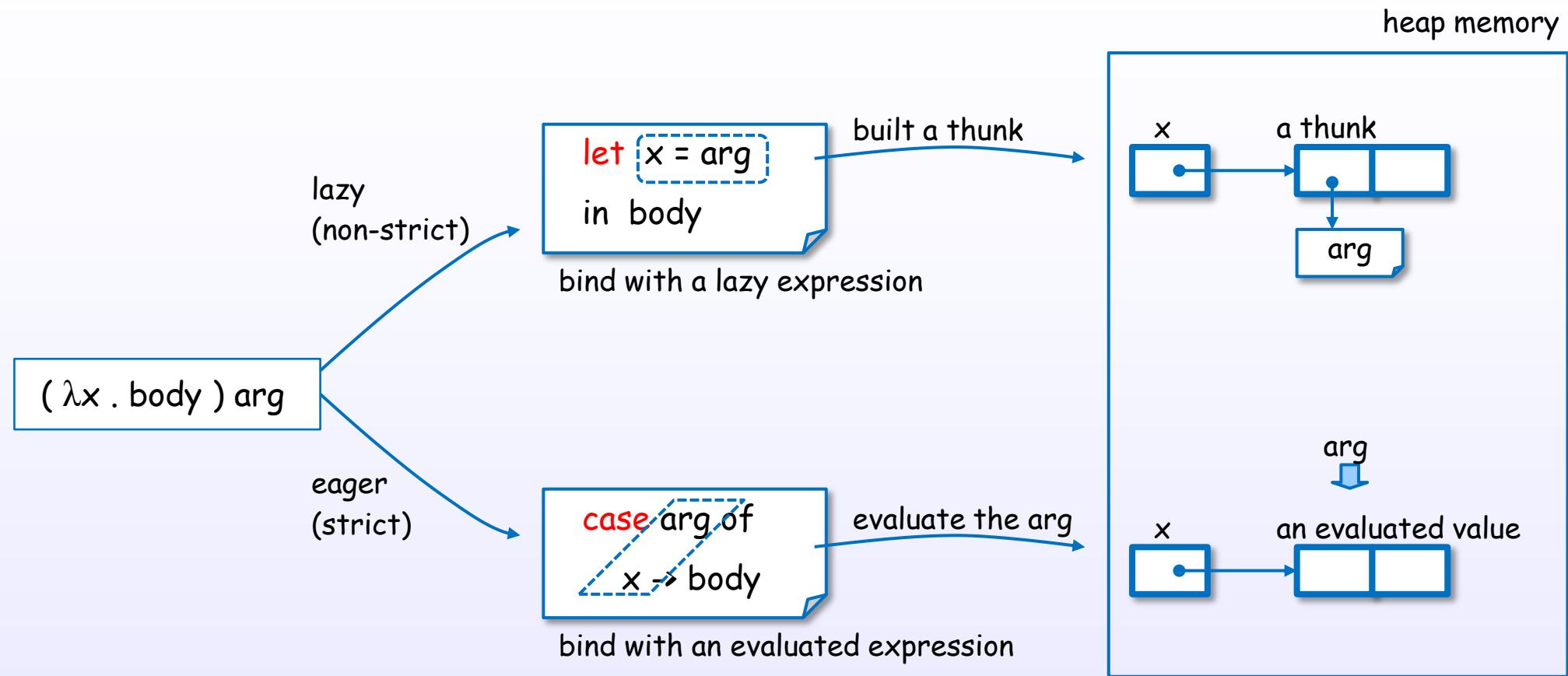
evaluate and update

an evaluated expression



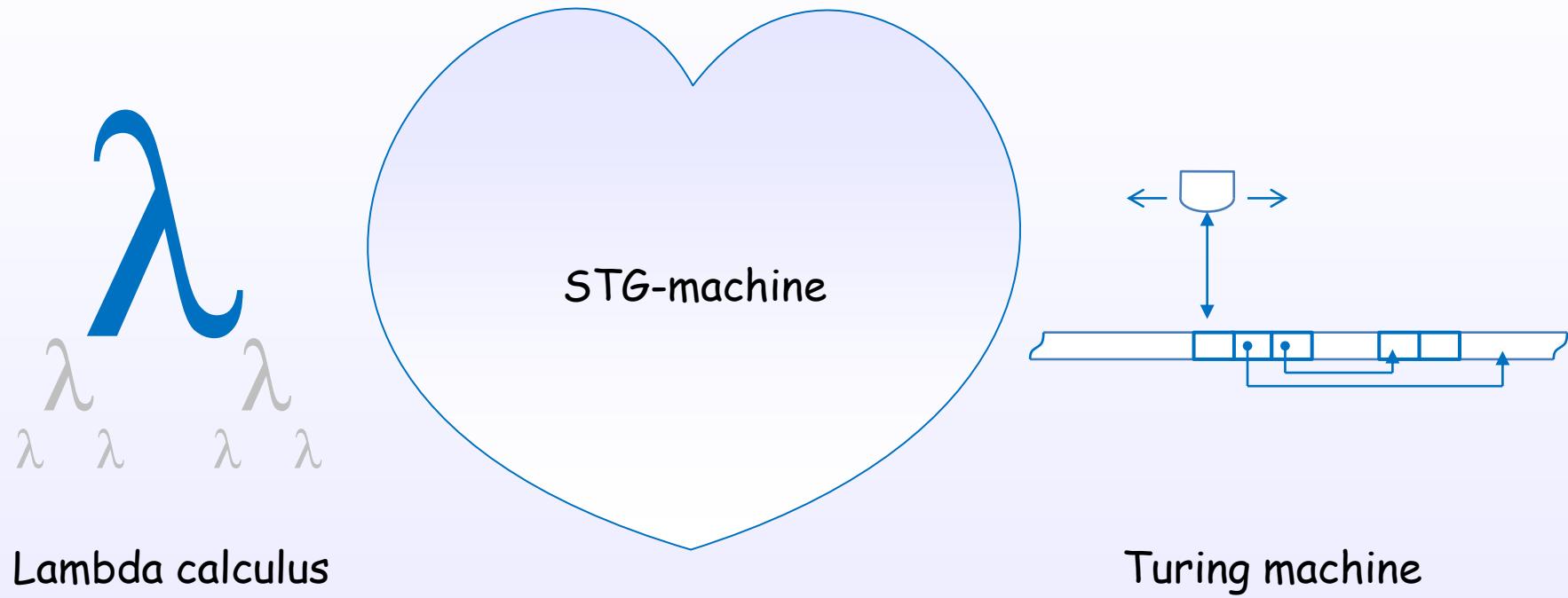
collected by GC(Garbage Collection)

STG-machine associates directly ...



STG-machine associates directly lambda calculus and physical machine.

The STG-machine is ...



The STG-machine is the marriage of Lambda calculus and Turing machine.

STG-dump shows which expression is built as thunks

[Example.hs]

```
module Example where
  fun f1 n = take 1 f1 n
```

STG code dump
by "\$ ghc -O -ddump-stg Example.hs"

Example.fun

```
:: forall a_aME t_aMF. (t_aMF -> [a_aME]) -> t_aMF ->
[a_aME]
[GblId,
Arity=2,
Caf=NoCafRefs,
Str=DmdType <L,1*C1(U)><L,U>,
Unf=OtherCon []]=
\ r srt:SRT:[] [f1_sQT n_sQU]
let {
  sat_sQV [Occ=Once, Dmd=<L,1*U>] :: [a_aMH]
  [LclId, Str=DmdType] =
    \ s srt:SRT:[] [] f1_sQT n_sQU;
} in GHC.List.take_unsafe_UInt 1 sat_sQV;
```

thunk

f1_sQT n_sQU

f1_sQT n_sQU

heap memory

build/allocate

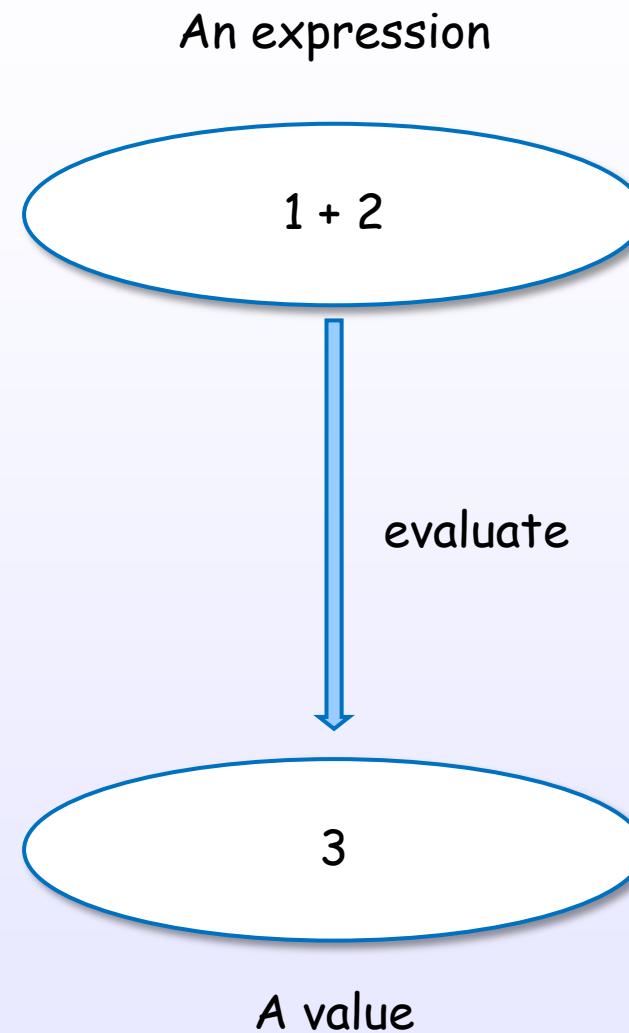
let expression in STG language

6. Semantics

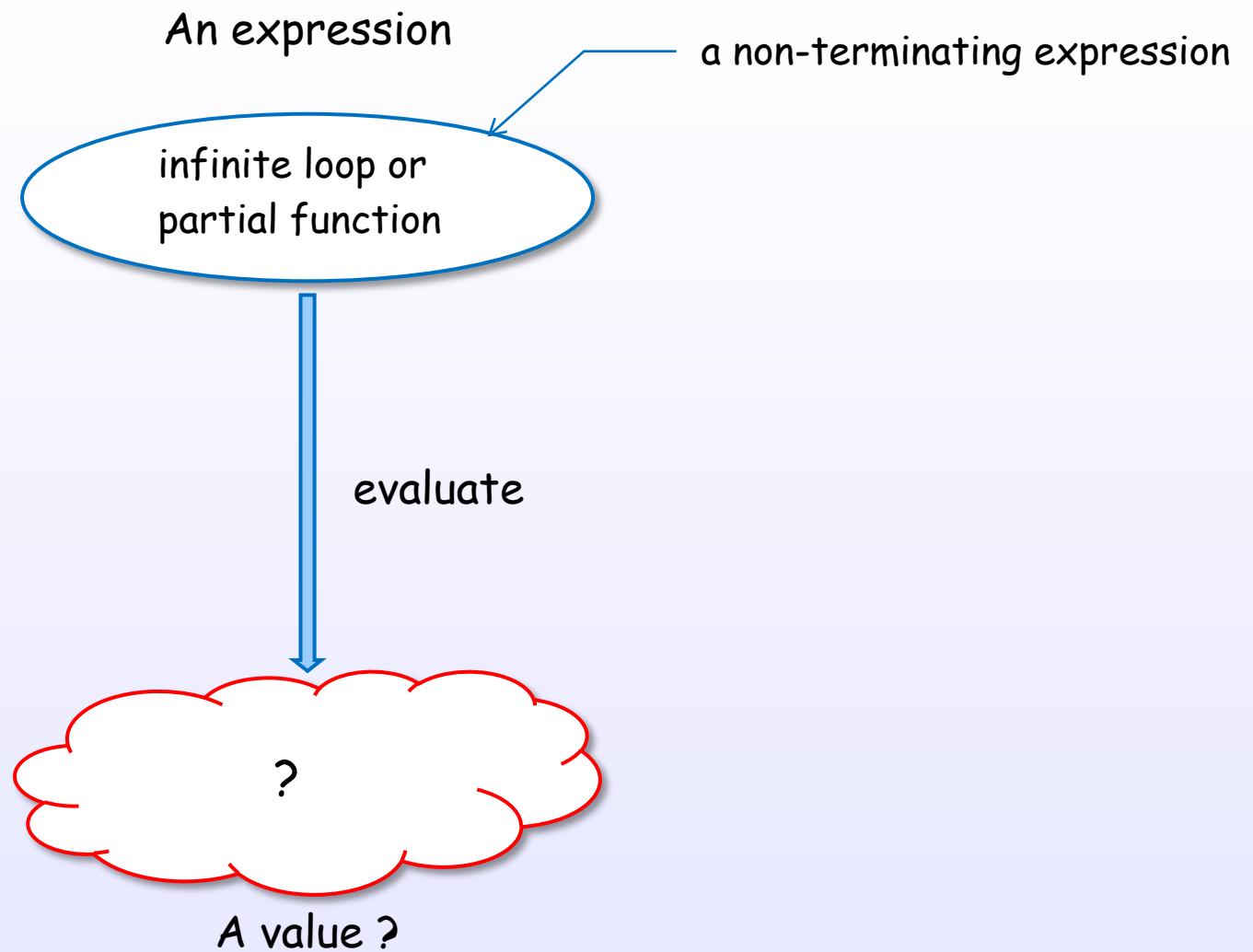
6. Semantics

Bottom

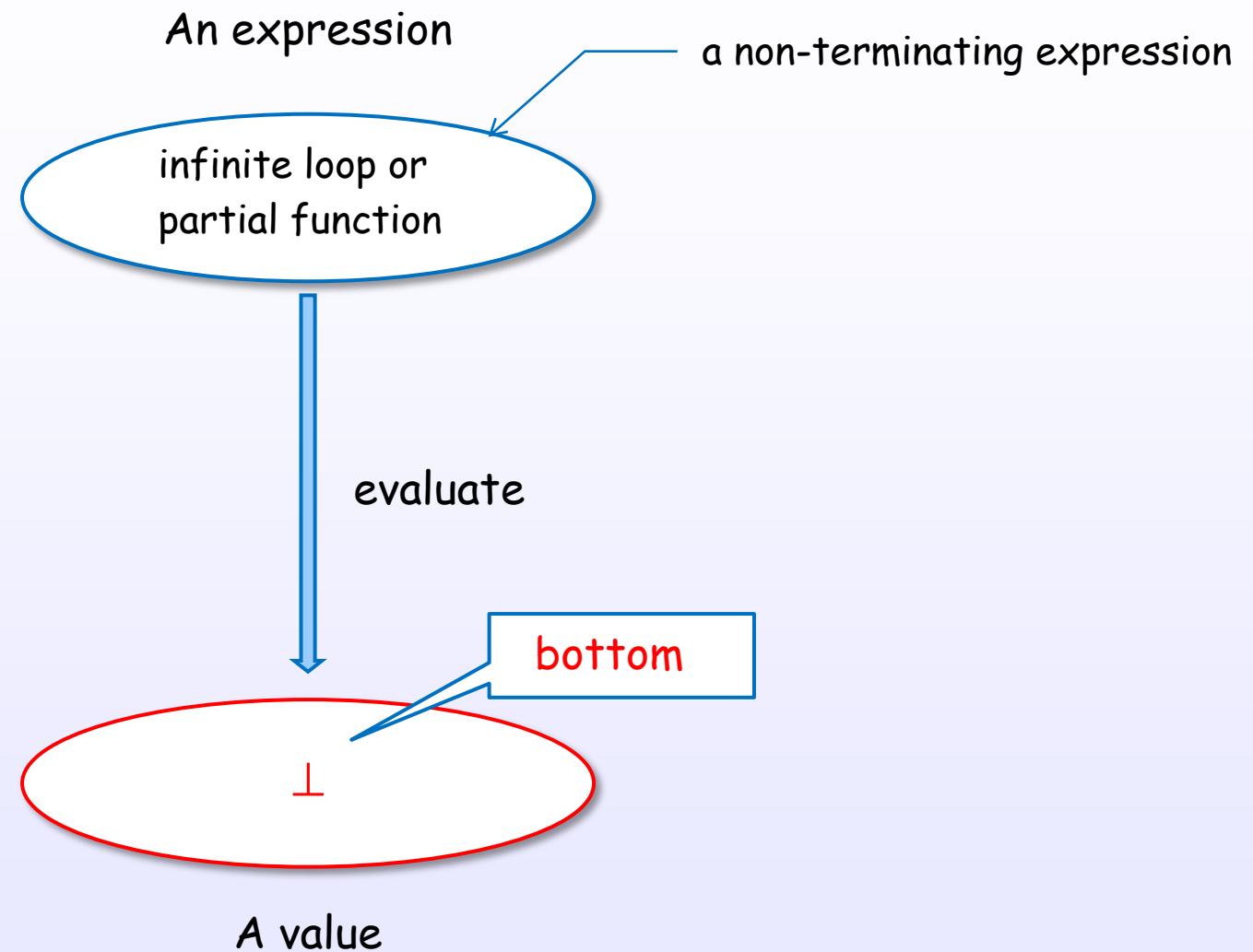
Well formed expression should have a value



What is a value in this case?

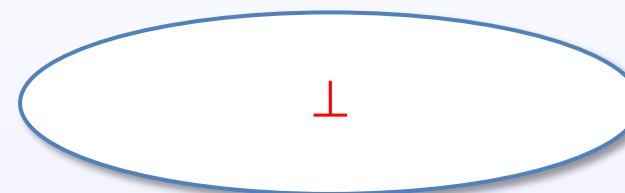


A value “bottom” is introduced



Bottom

A value



Bottom (\perp) is “an undefined value”.

Bottom (\perp) is “a non-terminating value”.

Bottom is also represented by “undefined” in Haskell

Haskell code

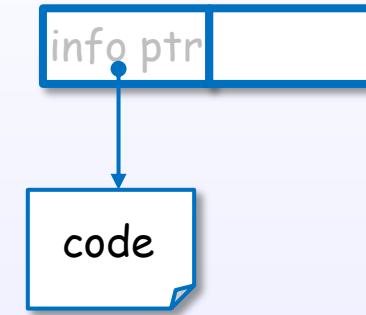
undefined :: a

Expression

\perp

GHC's internal representation

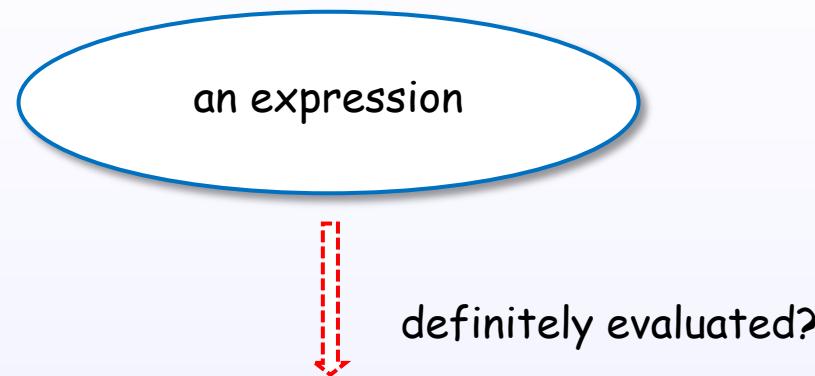
GHC.Err.undefined



6. Semantics

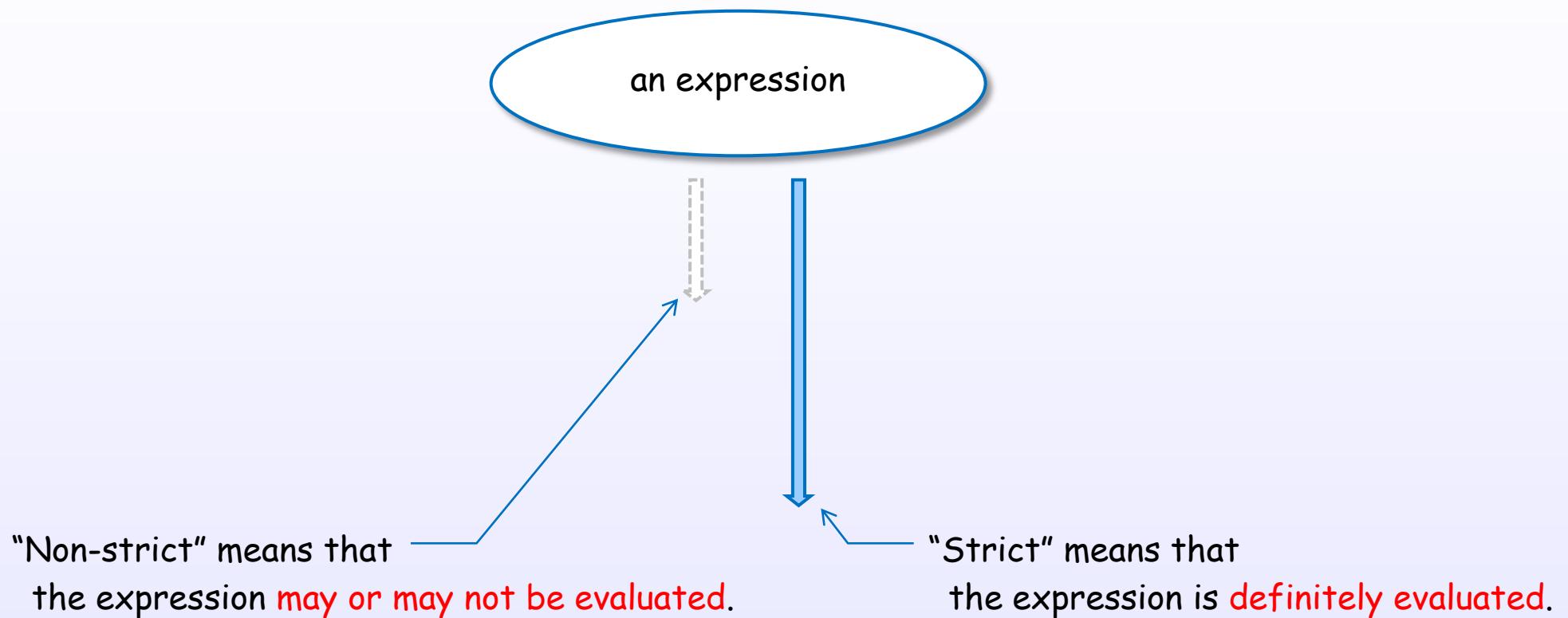
Strict/Non-strict

Strictness

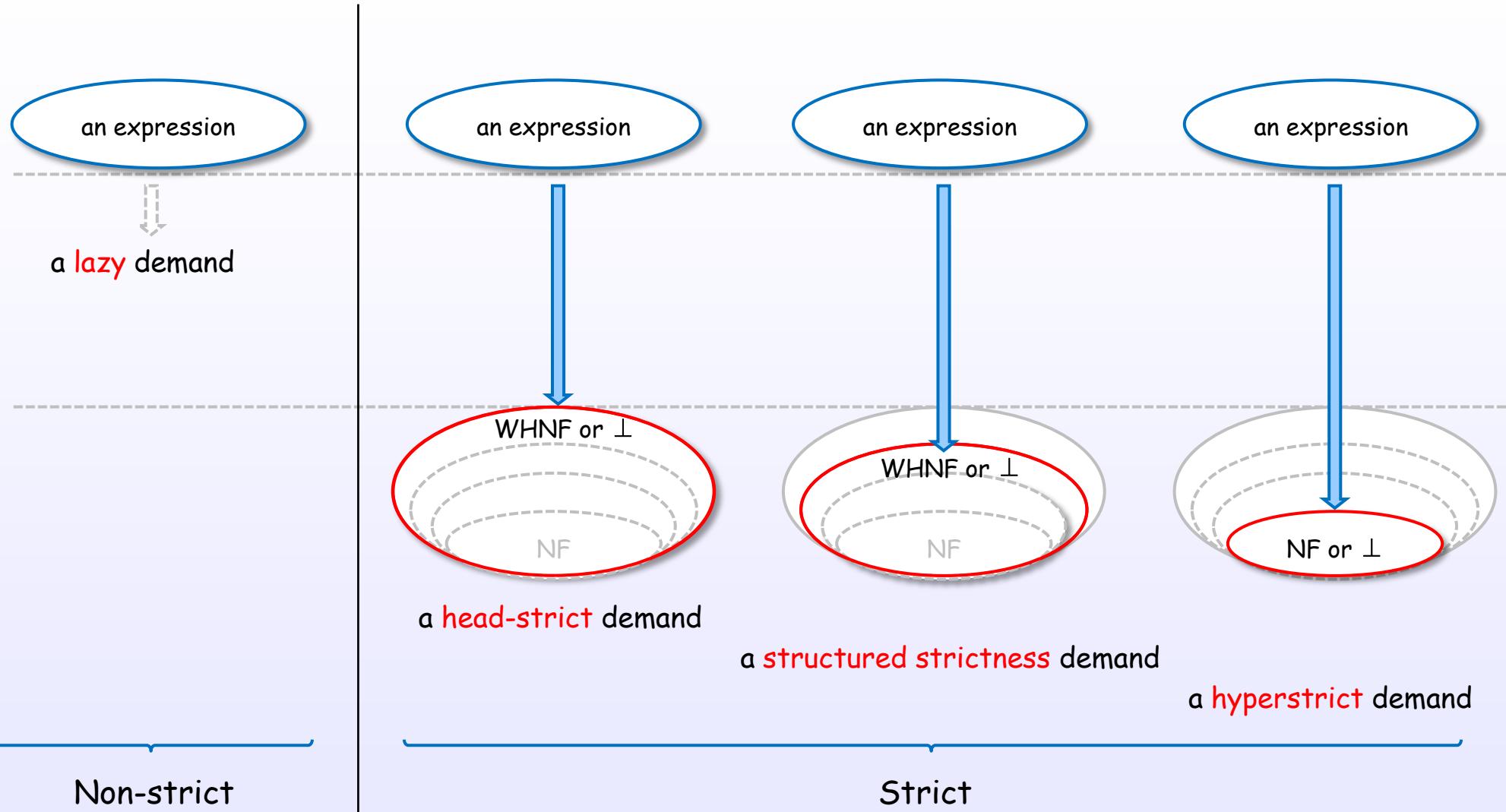


Strictness is “evaluation demand” of the expression.

Strict and non-strict

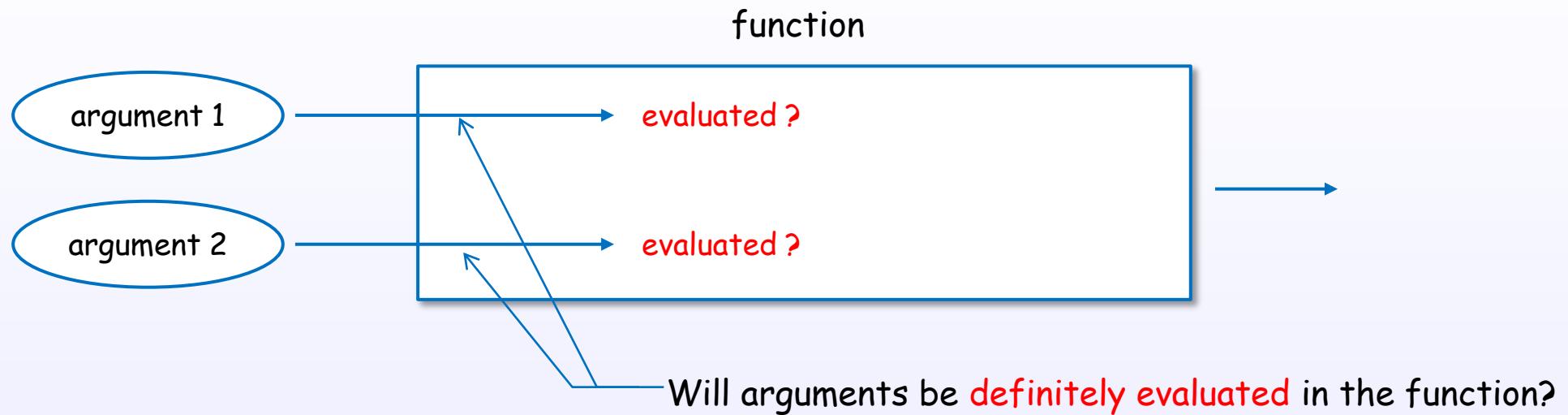


GHC has the lattice of strictness



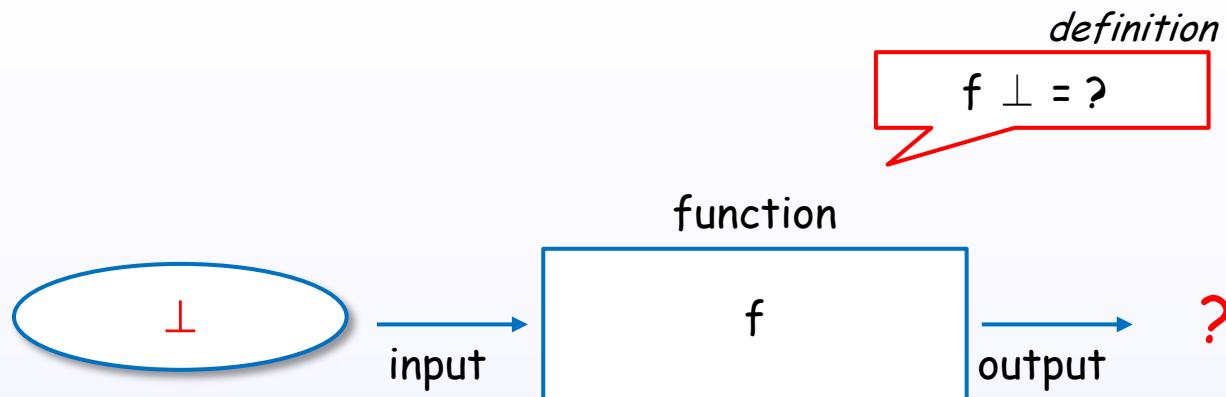
There are multiple levels in strict.

Strictness of a function



A function places "strictness demands" on each of its arguments.

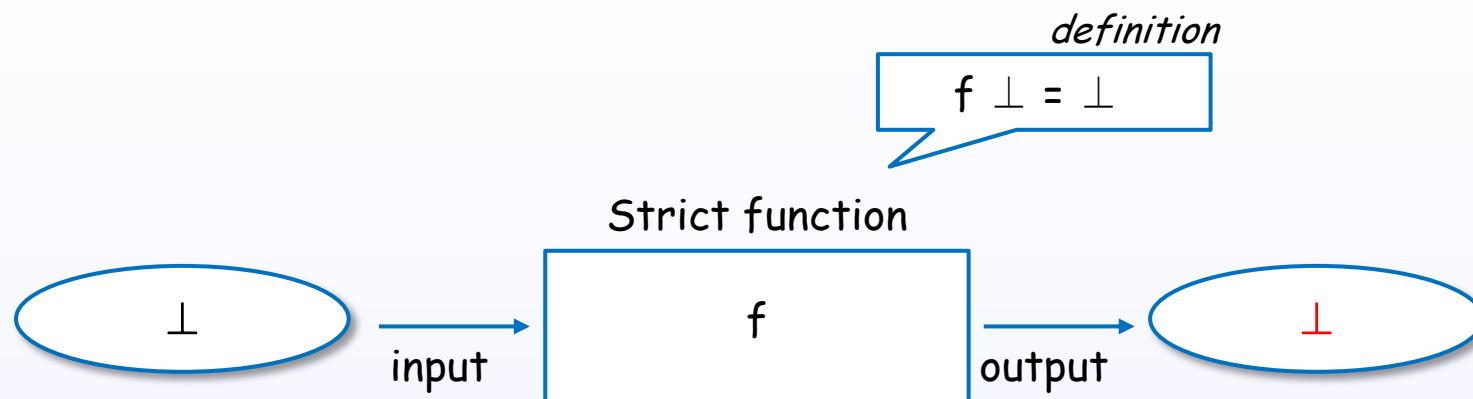
Strictness of a function is defined formally



Strictness of a function can be defined with the relationship between input and output.

"given a non-terminating arguments, the function will terminate?"

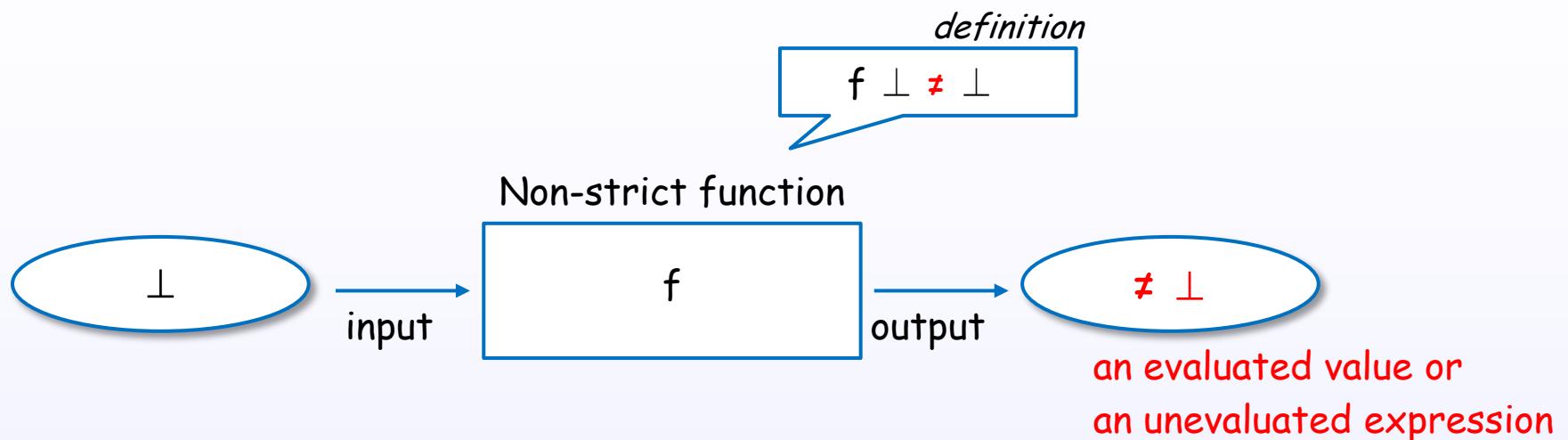
Definition of strict function



Strict function's output is bottom when input is bottom.

given a non-terminating arguments, strict function will **not** terminate.

Definition of non-strict function

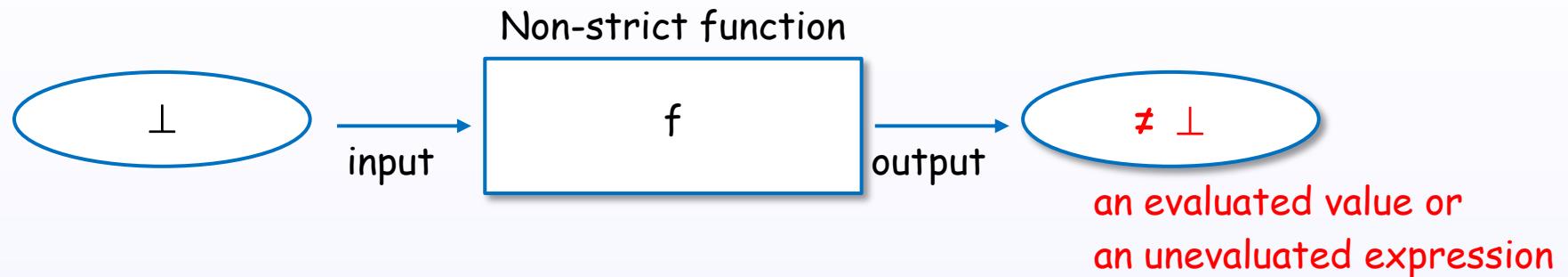


Non-strict function's output is **not** bottom when input is bottom.

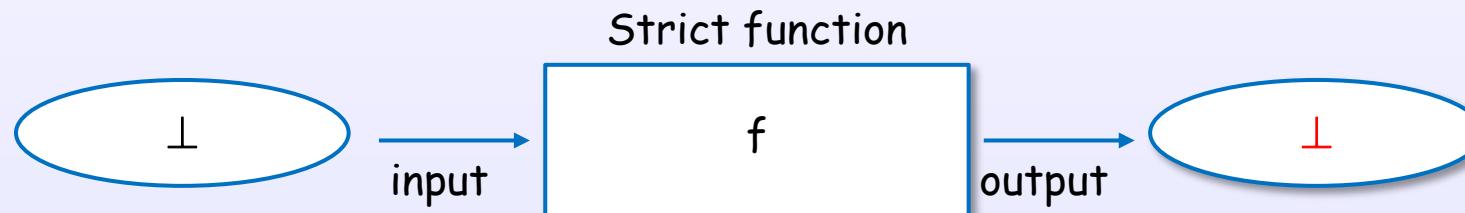
given a non-terminating arguments, non-strict function will terminate.

Strict and Non-strict functions

Non-strict

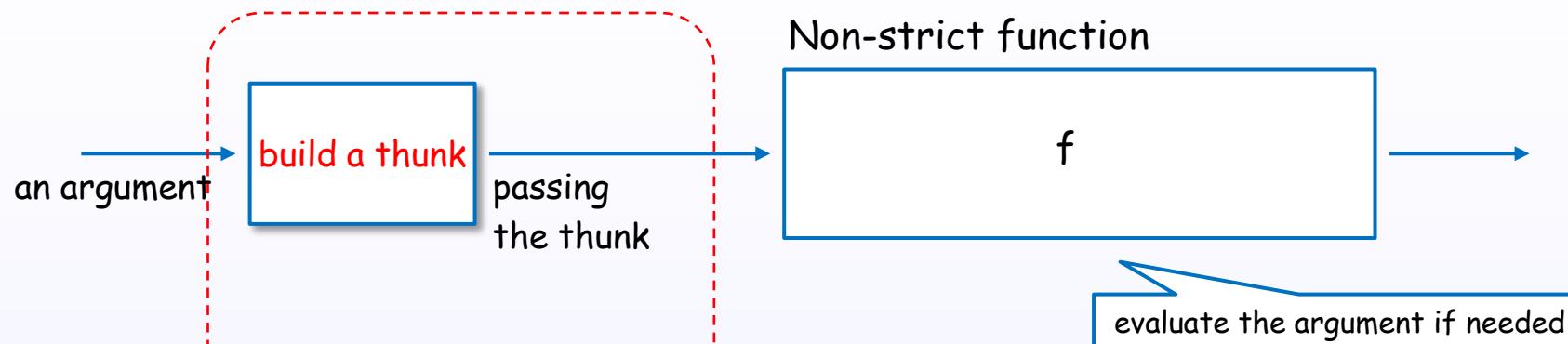


Strict

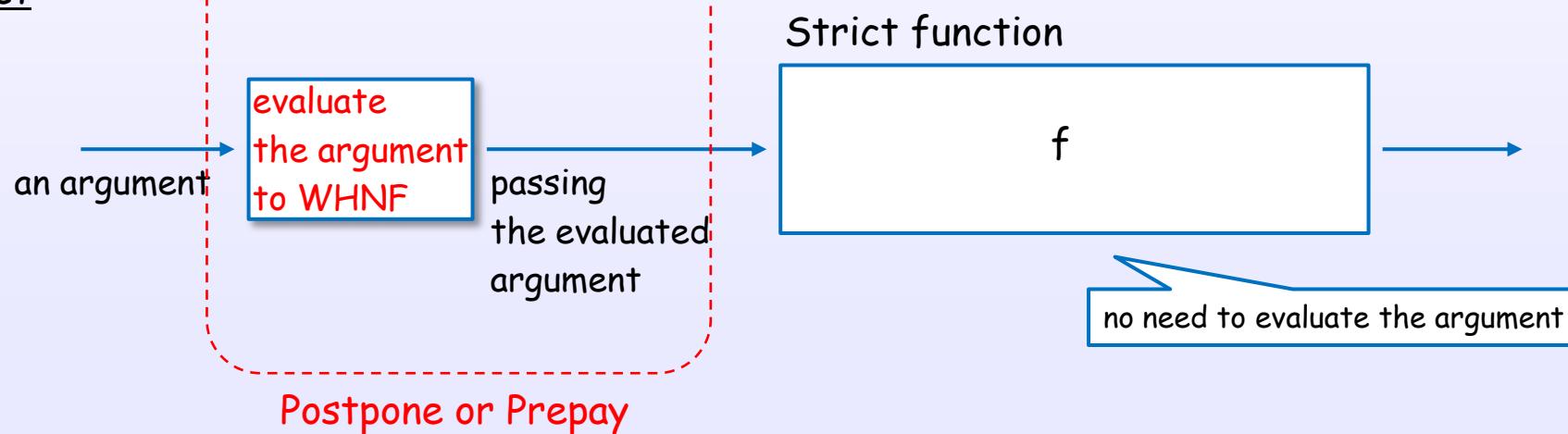


Function application and strictness

Non-strict



Strict



The front stage is also important.

Strict and normal form

Example of function application

	to WHNF	to NF
Non-strict	$f \$ \text{arg}$	
Strict	$f \$! \text{arg}$ (seq)	$f \$!! \text{arg}$ (deepseq)

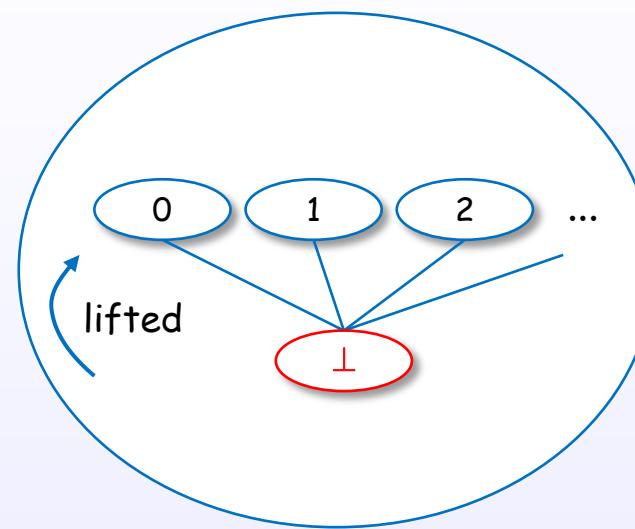
Strict \neq Normal form

6. Semantics

Lifted and boxed types

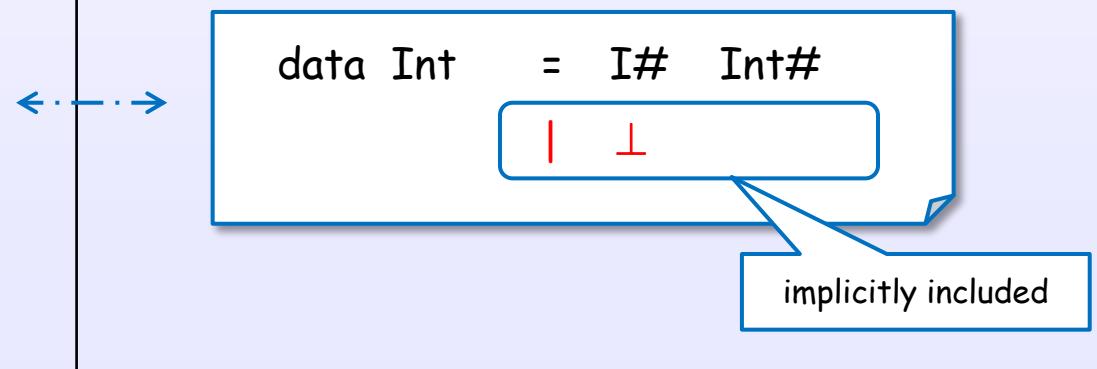
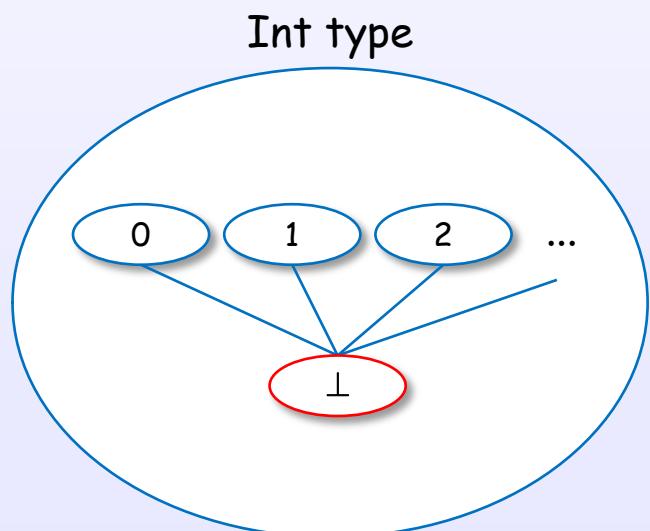
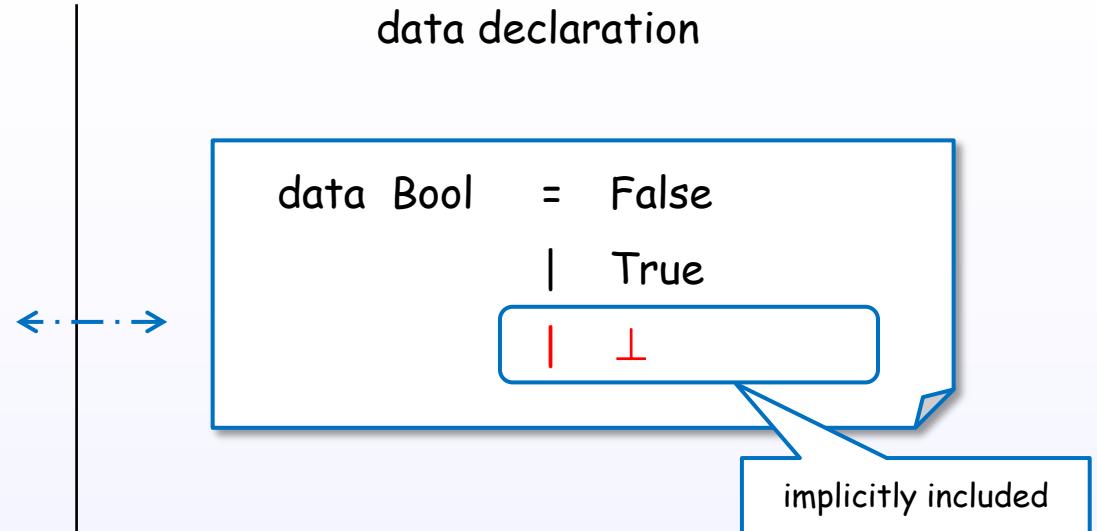
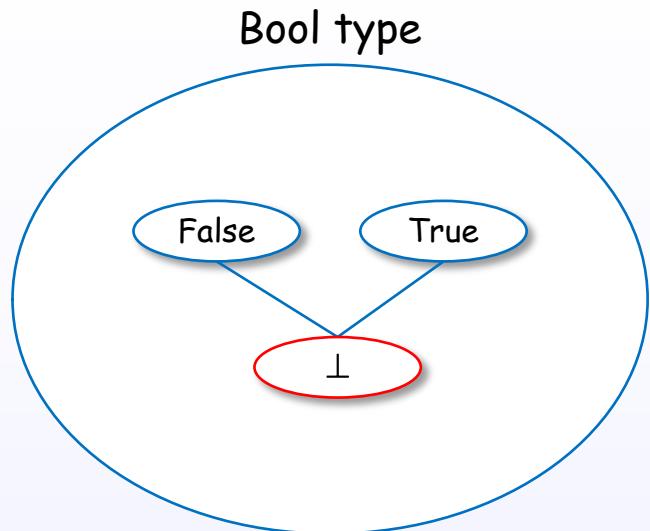
Lifted types

Lifted type



Lifted types include bottom as an element.

Lifted type's declaration implicitly include bottom

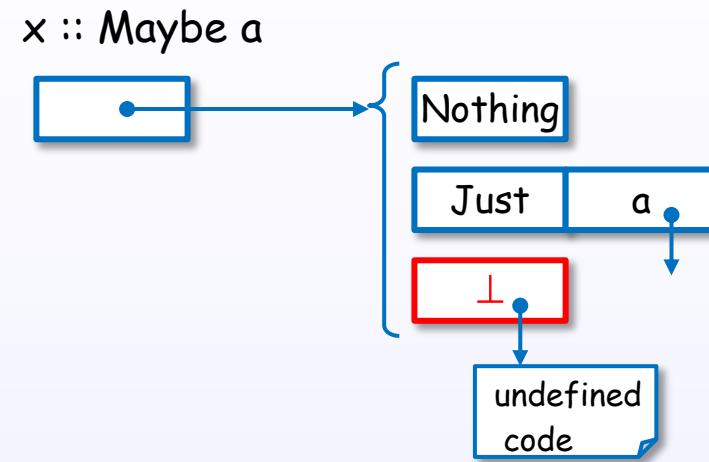


Lifted type are also implemented by uniform representation

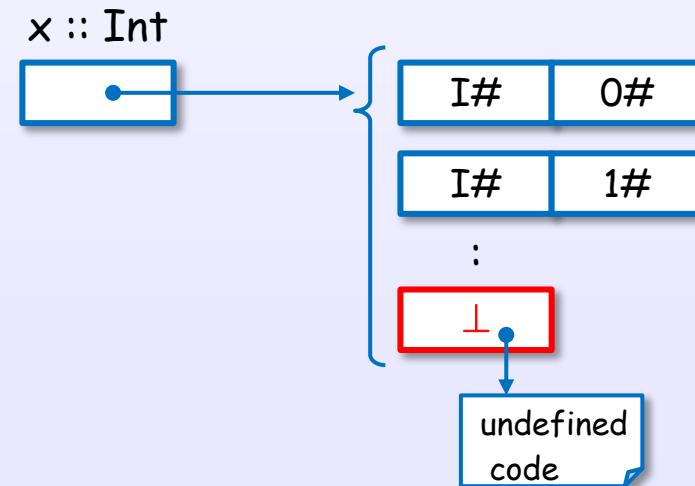
data declaration

```
data Maybe a = Nothing
             | Just a
             | ⊥
```

GHC's internal representation



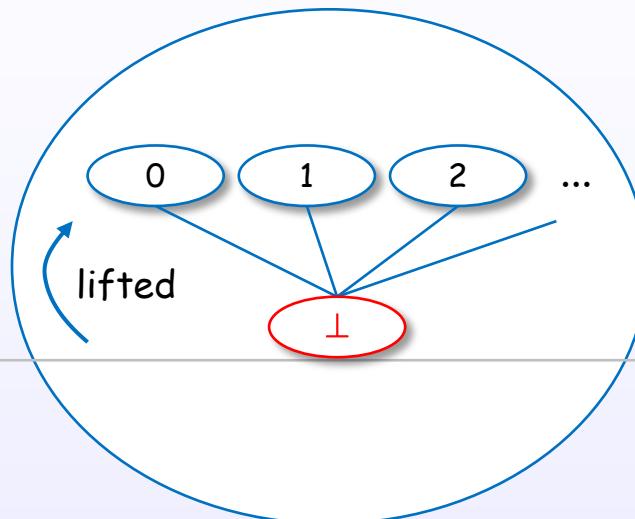
```
data Int = I# Int#
          | ⊥
```



Lifted and unlifted types

Lifted types

Int type

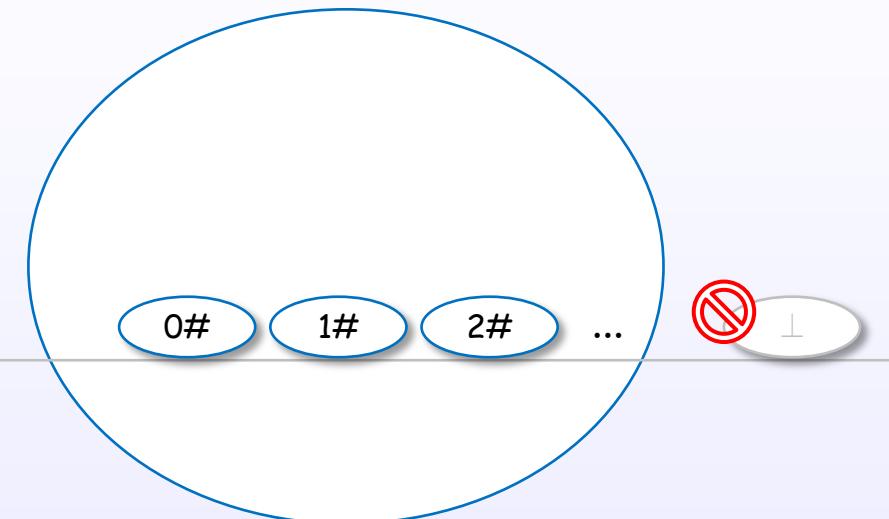


Lifted types include bottom.

(Bool, Int, Char, Maybe, List, ...)

Unlifted types

Int# type

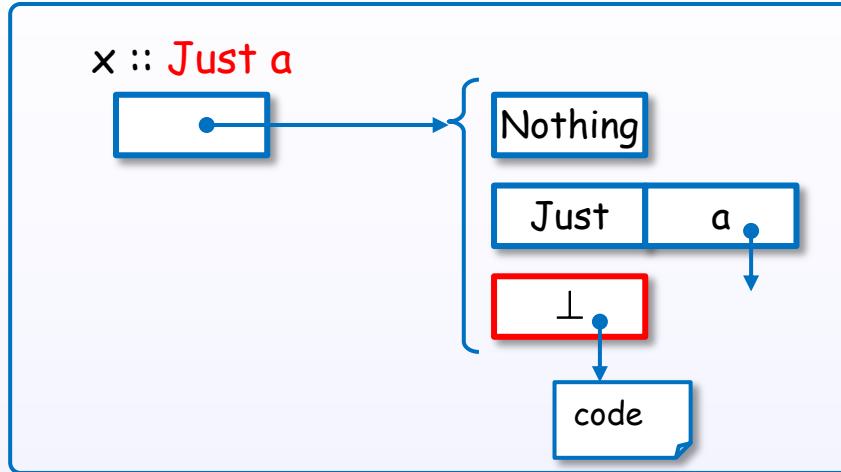


Unlifted types do not include bottom.

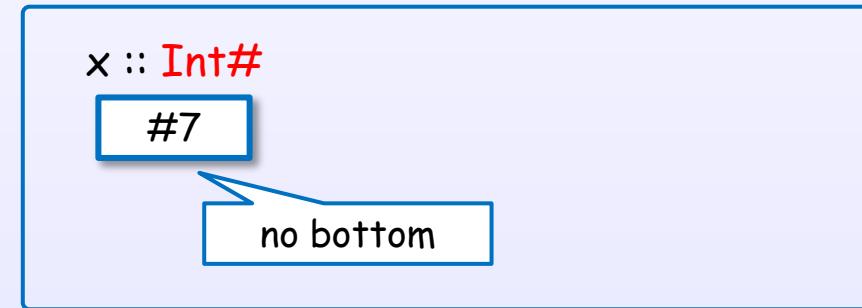
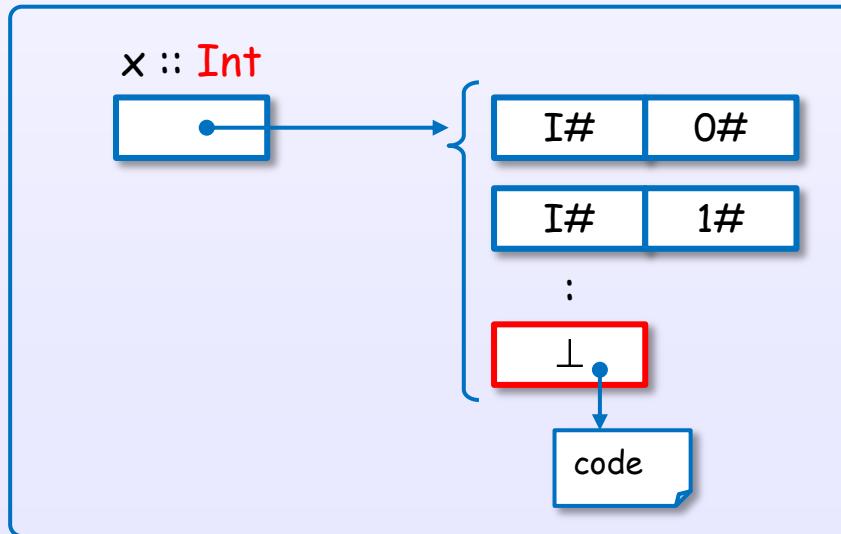
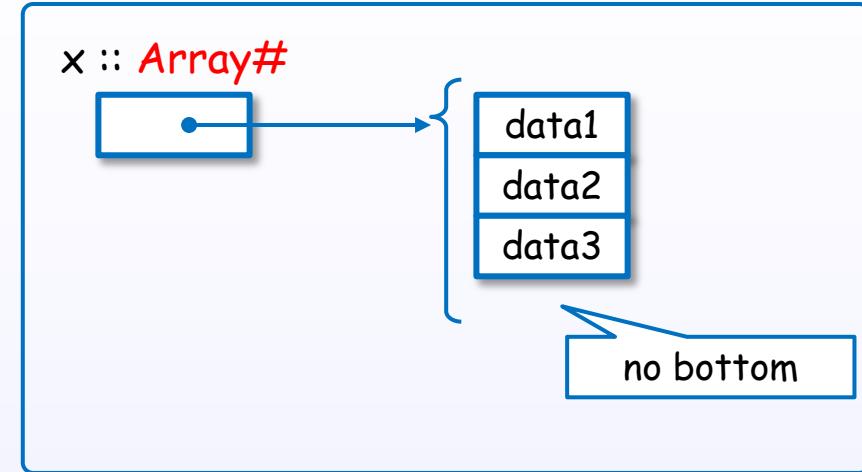
(Int#, Char#, Addr#, Array#, ByteArray#, ...)

Example of lifted and unlifted types

Lifted types



Unlifted types



Boxed and unboxed type

Boxed
types



Boxed types are represented as a pointer.

Unboxed
types

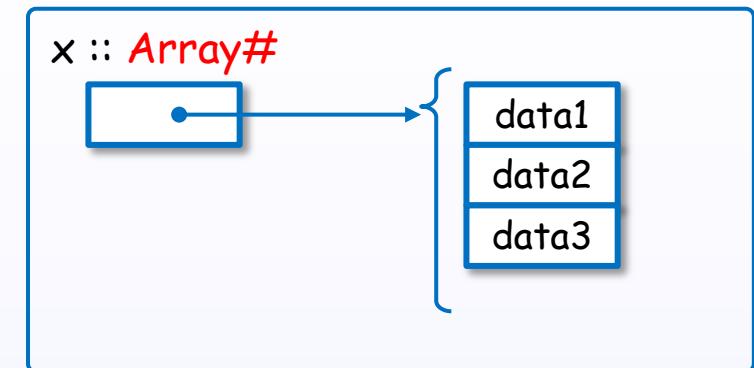
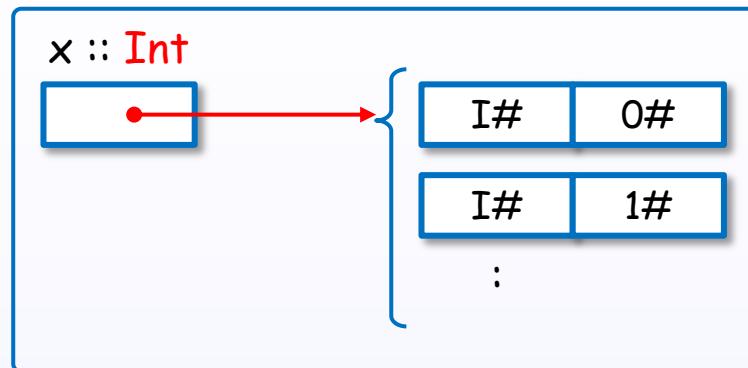


Unboxed types are represented other than a pointer.

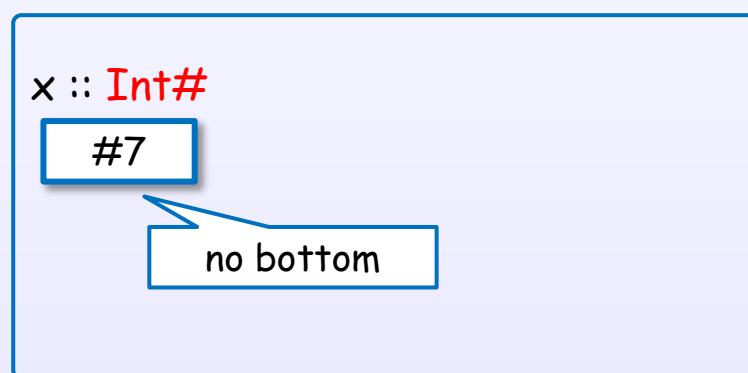
- no bottom (can't be lifted)
- no thunk (can't be postponed)
- no polymorphism (non-uniform size)
- + low cost memory size (no pointer)
- + high performance (no wrap/unwrap)

Example of boxed and unboxed types

Boxed
types



Unboxed
types



Lifted and boxed type

Boxed
types

Lifted types

Int
Char
Float
Maybe
:

Unlifted types

Array#
ByteArray#
:

no bottom

Unboxed
types

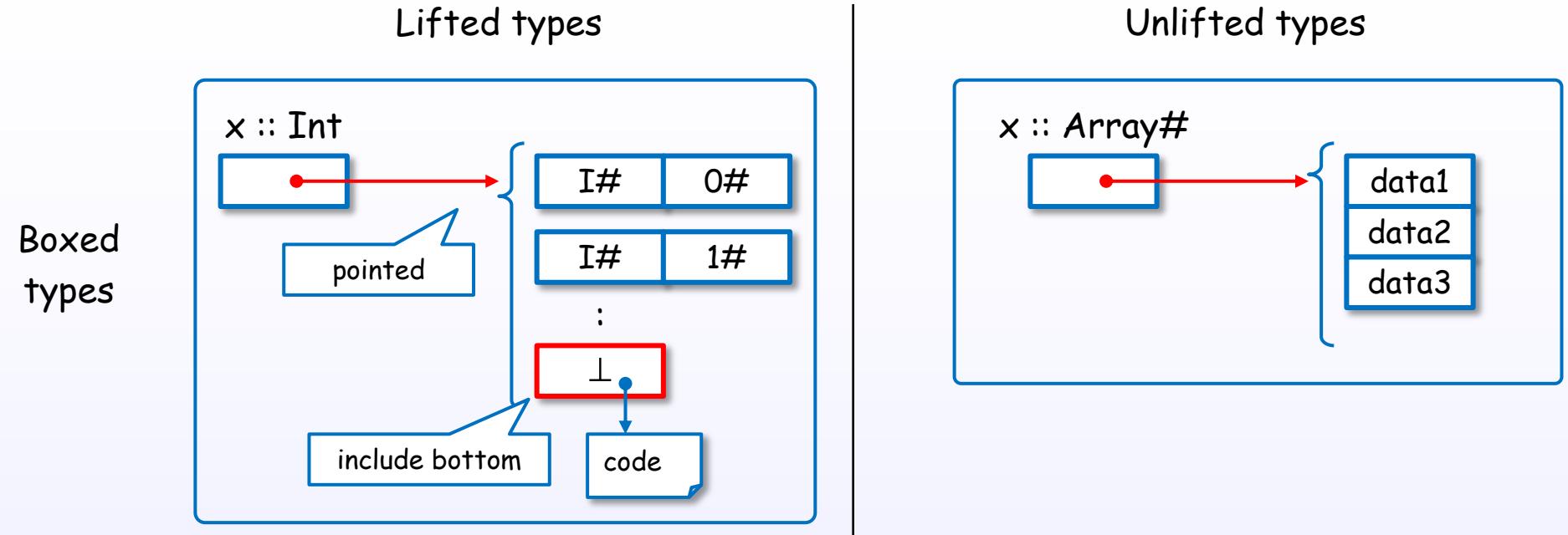


unboxed can't be lifted

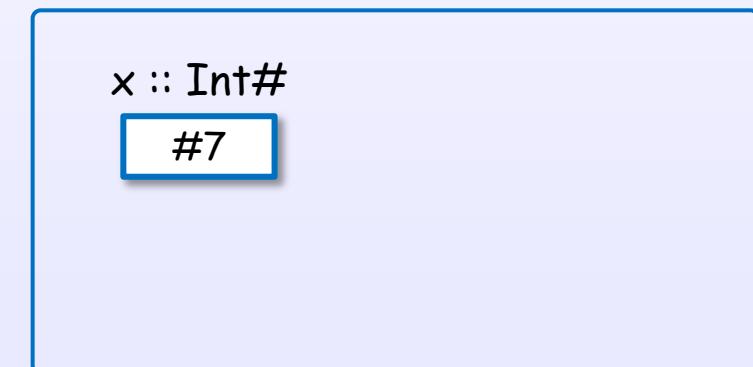
Int#
Char#
Float#
:

no bottom
no packed

Example of lifted and boxed type



Unboxed
types



Types and kinds

Boxed
types

Lifted types

kind '*' 

Int
Char
Float
Maybe
:

Unlifted types

kind '#' 

Array#
ByteArray#
:

Unboxed
types



Int#
Char#
Float#
:

Note:

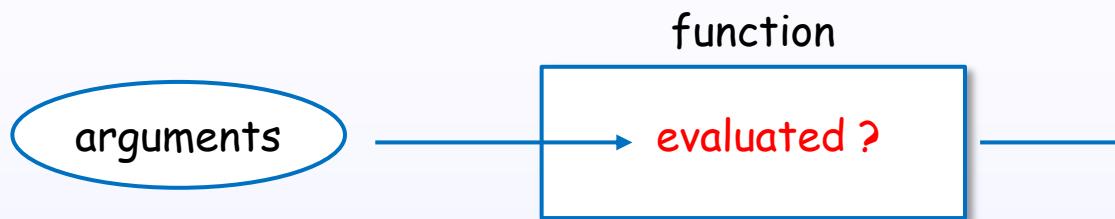
Identifier's '#' customarily means "primitive" rather than "unboxed" or "unlifted".

Kind's '#' means "unlifted".

6. Semantics

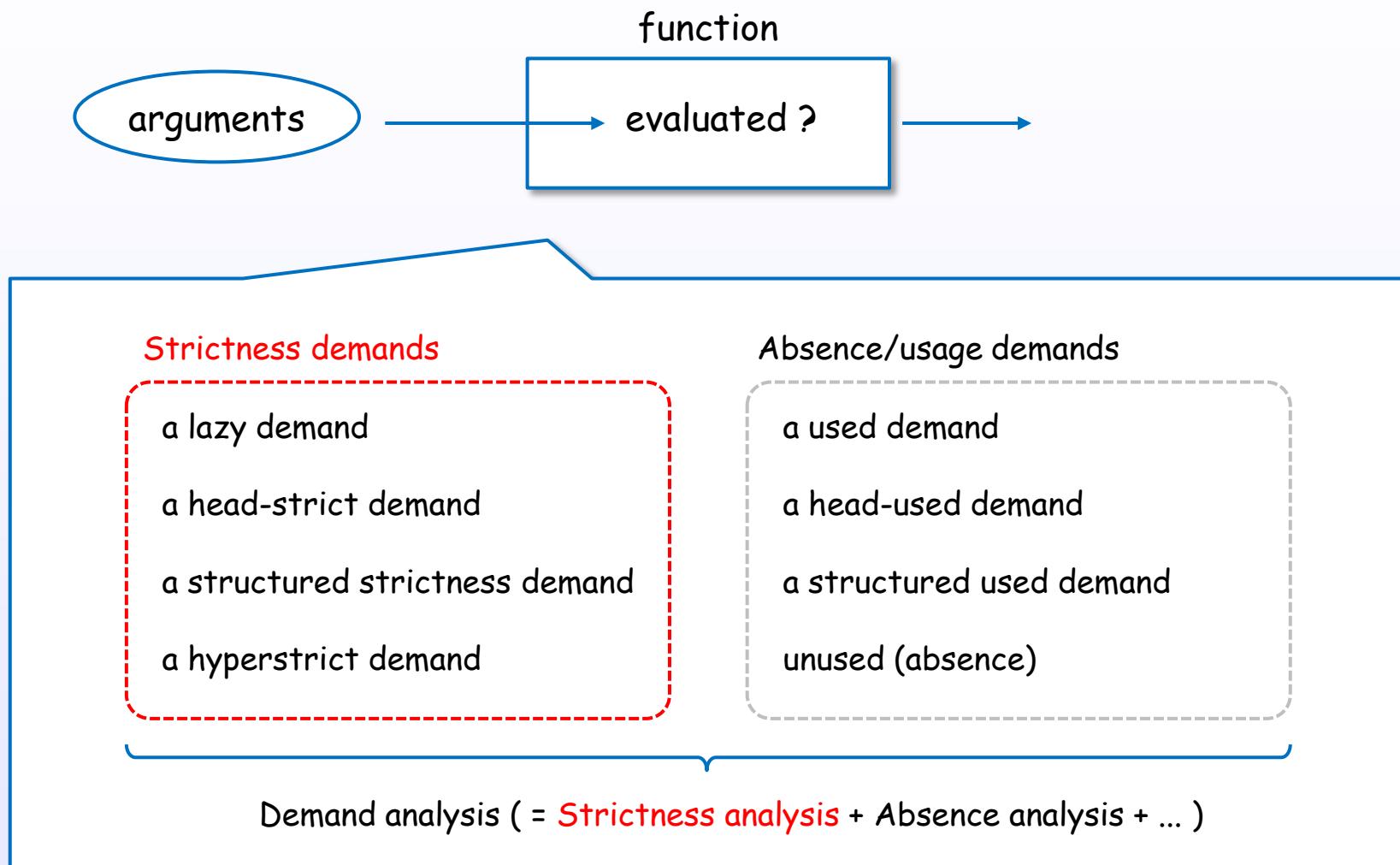
Strictness analysis

Strictness analysis



Strictness analysis analyzes whether a function is sure to evaluate its argument.

Strictness analysis in GHC



GHC shows strictness analysis information with “-ddump-strsigs” and “-ddump-stranal”.

Example of strictness analysis information in GHC

[Example.hs]

```
module Example where

f1 :: Bool -> a -> Maybe a
f1 c x = case c of
    True -> Just x
    False -> Nothing
```

Strictness analysis dump
by "\$ ghc -O -ddump-strsigs Example.hs"

===== Strictness signatures =====

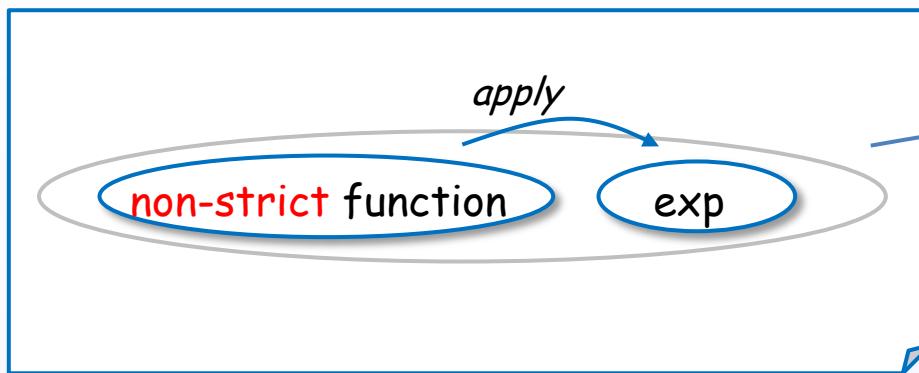
Example.f1: <S,1*U><L,U>

L -- second argument is "Lazy"

S -- first argument is "head-Strict"

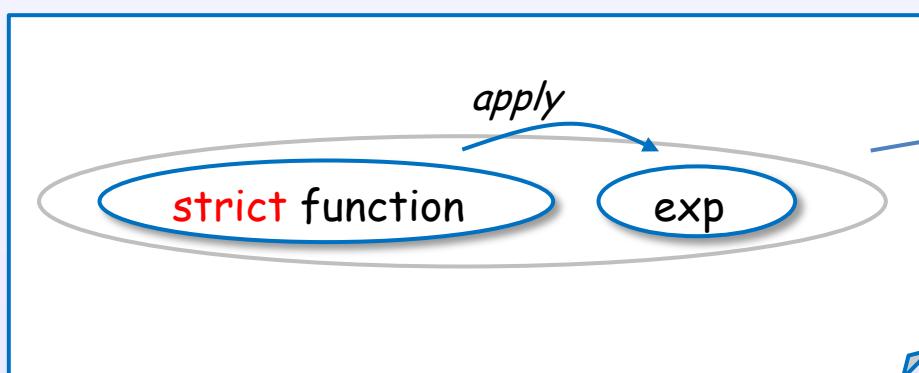
(1) Strictness analysis are used to avoid the thunk

non-strict function



heap memory

strict function



heap memory



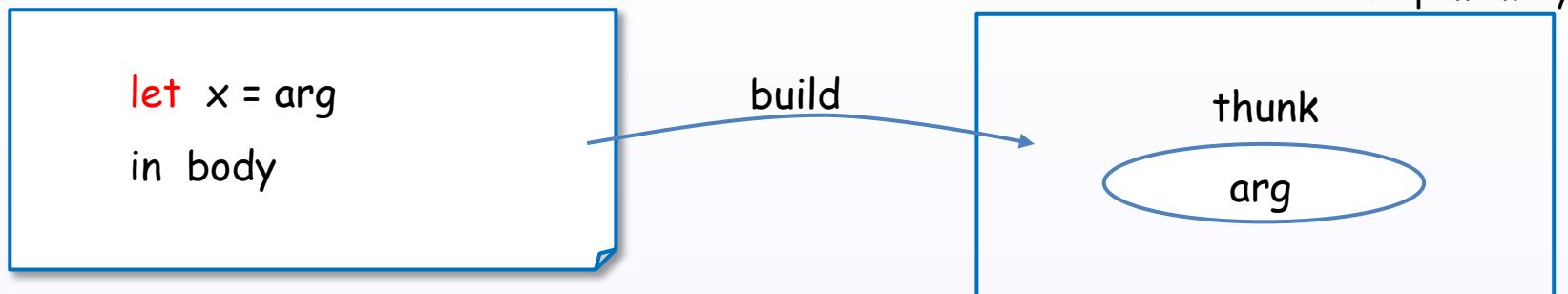
no build
no force
no update
no GC

If GHC knows that a function is strict, arguments are evaluated before application.

GHC finds strict functions by "strictness analysis (demand analysis)".

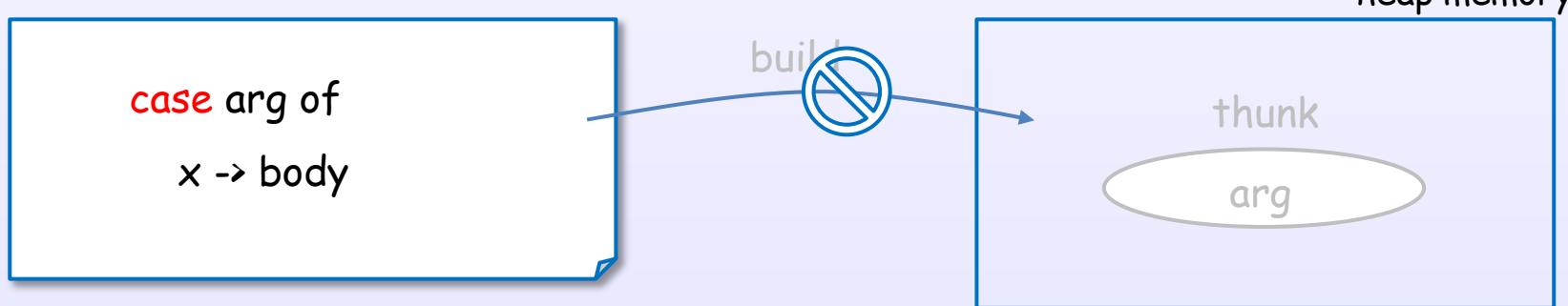
(1) Strictness analysis are used to avoid the thunk

non-strict



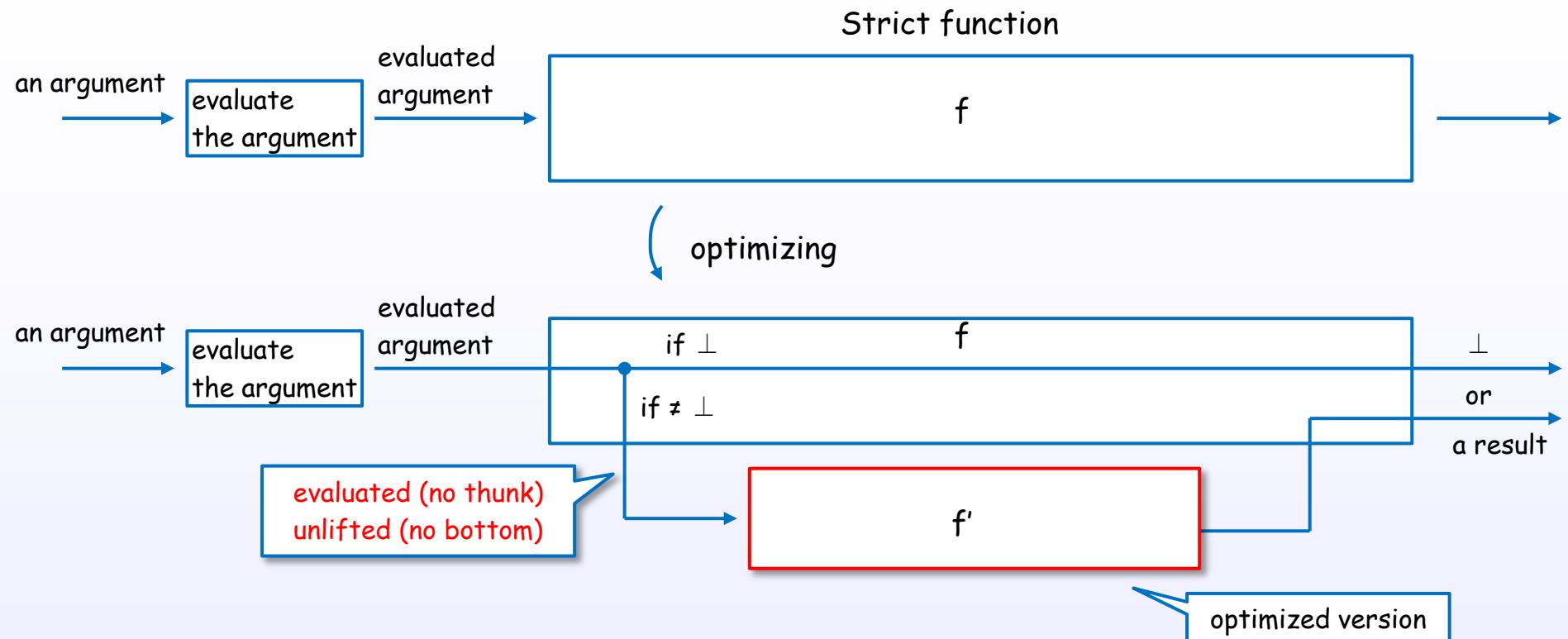
let-to-case transformation

strict



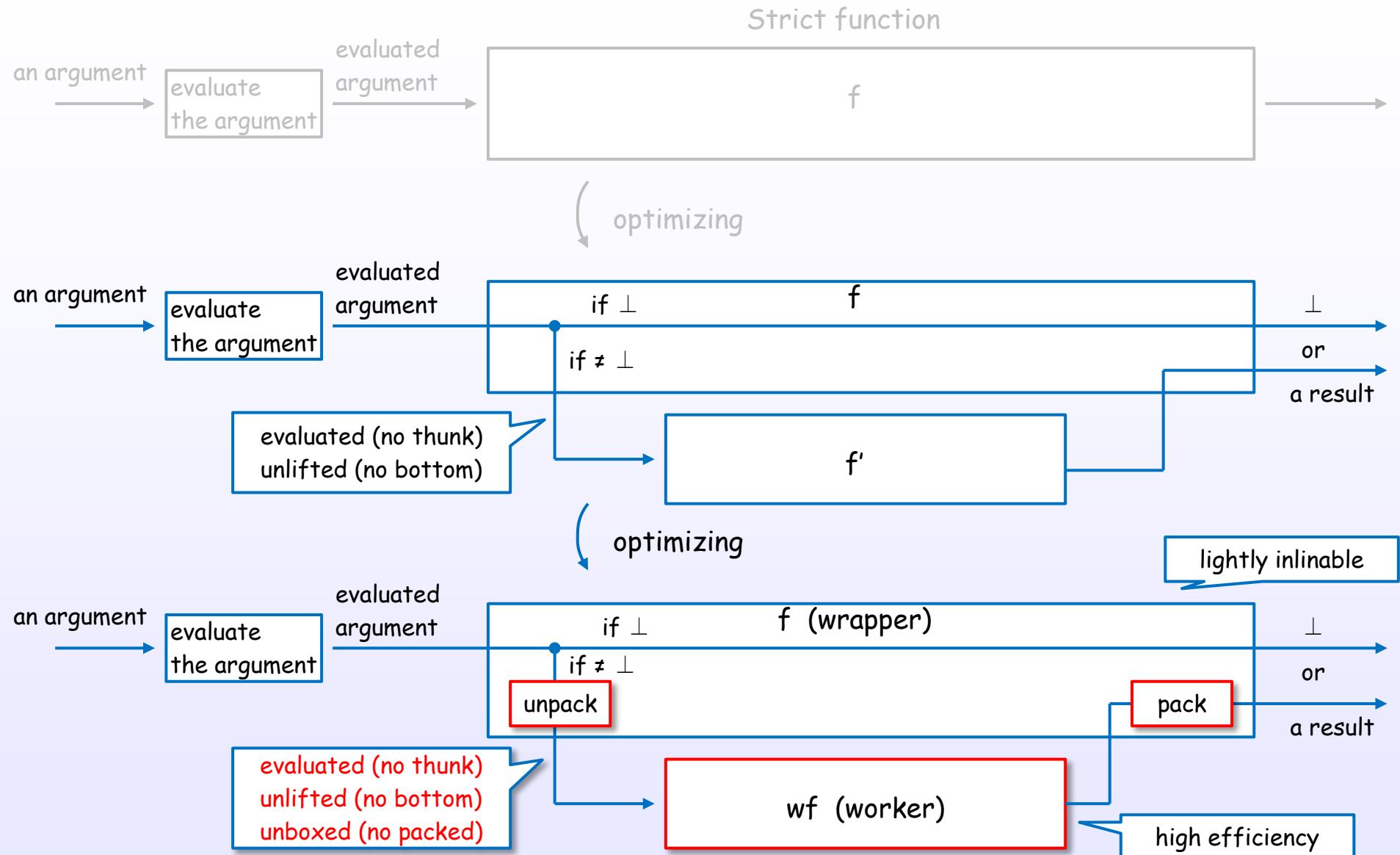
If GHC knows that a function is strict, GHC performs let-to-case transformation.

(2) Strictness analysis are also used to optimize



Strictness function can be optimized to assume no thunk, no bottom.

(2) Strictness analysis are also used to optimize



Strictness function can be optimized to assume no thunk, no bottom, no packed.

6. Semantics

Sequential order

"seq" doesn't guarantee the evaluation order

specification

```
seq a b = ⊥, if a = ⊥  
      = b, otherwise
```

strictness for each arguments

```
seq ⊥ b = ⊥ // a is strict  
seq a ⊥ = ⊥ // b is strict
```

"seq" function only guarantee that it is strict in both arguments.

This semantics property makes **no operational guarantee** about **order** of evaluation.

"seq" and "pseq"

specification

```
seq a b = ⊥, if a = ⊥  
= b, otherwise
```

```
seq ⊥ b = ⊥ // a is strict  
seq a ⊥ = ⊥ // b is strict
```

specification

```
pseq a b = ⊥, if a = ⊥  
= b, otherwise
```

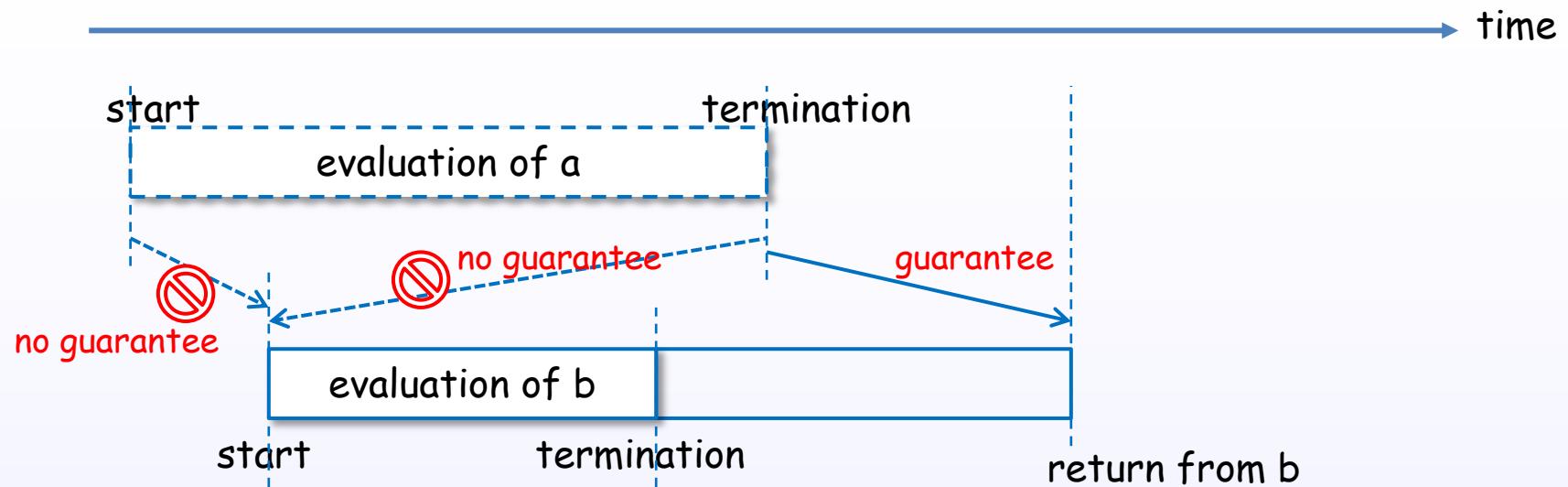
```
pseq ⊥ b = ⊥ // a is strict  
pseq a ⊥ = ⊥ // b is strict
```

Both of denotational semantics are the same.

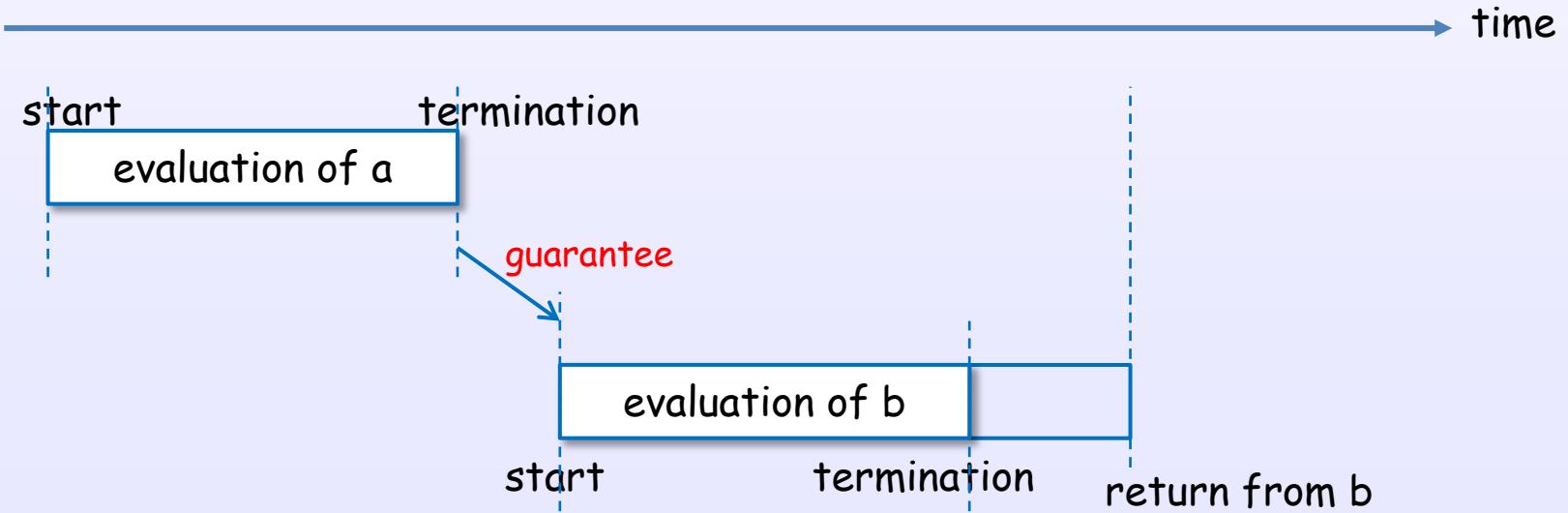
But "pseq" makes operational guarantee about order of evaluation.

Evaluation order of "seq" and "pseq"

seq a b



pseq a b



Implementation of "seq" and "pseq"

specification

```
seq a b = ⊥, if a = ⊥
          = b, otherwise
```

specification

```
pseq a b = ⊥, if a = ⊥
          = b, otherwise
```

Haskell's built-in

`pseq x y = x `seq` lazy y`

GHC's "lazy" function restrains
the strictness analysis.

"seq" is built-in function.

"pseq" is implemented by built-in functions ("seq" and "lazy").

7. Appendix

7. Appendix

References

References

- [H1] Haskell 2010 Language Report
<https://www.haskell.org/definition/haskell2010.pdf>
- [H2] The Glorious Glasgow Haskell Compilation System (GHC user's guide)
https://downloads.haskell.org/~ghc/latest/docs/users_guide.pdf
- [H3] A History of Haskell: Being Lazy With Class
<http://haskell.cs.yale.edu/wp-content/uploads/2011/02/history.pdf>
- [H4] The implementation of functional programming languages
<http://research.microsoft.com/en-us/um/people/simonpj/papers/slpj-book-1987/slpj-book-1987.pdf>
- [H5] Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine Version 2.5
<http://research.microsoft.com/en-us/um/people/simonpj/Papers/spineless-tagless-gmachine.ps.gz>
- [H6] Making a Fast Curry Push/Enter vs Eval/Apply for Higher-order Languages
<http://research.microsoft.com/en-us/um/people/simonpj/papers/eval-apply>
- [H7] Faster Laziness Using Dynamic Pointer Tagging
<http://research.microsoft.com/en-us/um/people/simonpj/papers/ptr-tag/ptr-tagging.pdf>
- [H8] Measuring the effectiveness of a simple strictness analyser
<http://research.microsoft.com/~simonpj/papers/simple-strictnes-analyser.ps.gz>
- [H9] Runtime Support for Multicore Haskell
<http://community.haskell.org/~simonmar/papers/multicore-ghc.pdf>
- [H10] I know kung fu: learning STG by example
<https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/GeneratedCode>

References

- [H11] *GHC Commentary: The Layout of Heap Objects*
<https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/HeapObjects>
- [H12] *The ghc-prim package*
<https://hackage.haskell.org/package/ghc-prim>
- [H13] *GHC Commentary: Strict & StrictData*
<https://ghc.haskell.org/trac/ghc/wiki/StrictPragma>
- [H14] *The data type Type and its friends*
<https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/TypeType>
- [H15] *Demand analyser in GHC*
<https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/Demand>
- [H16] *Demand analysis*
<http://research.microsoft.com/en-us/um/people/simonpj/papers/demand-anal/demand.ps>
- [H17] *Core-to-Core optimization pipeline*
<https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/Core2CorePipeline>
- [H18] *The GHC reading list*
<https://ghc.haskell.org/trac/ghc/wiki/ReadingList>
- [H19] *The GHC Commentary*
<https://ghc.haskell.org/trac/ghc/wiki/Commentary>

References

- [B1] Introduction to Functional Programming using Haskell (IFPH 2nd edition)
<http://www.cs.ox.ac.uk/publications/books/functional/bird-1998.jpg>
<http://www.pearsonhighered.com/educator/product/Introduction-Functional-Programming/9780134843469.page>
- [B2] Thinking Functionally with Haskell (IFPH 3rd edition)
<http://www.cs.ox.ac.uk/publications/books/functional/>
- [B3] Programming in Haskell
<https://www.cs.nott.ac.uk/~gmh/book.html>
- [B4] Real World Haskell
<http://book.realworldhaskell.org/>
- [B5] Parallel and Concurrent Programming in Haskell
<http://chimera.labs.oreilly.com/books/1230000000929>
- [B6] Types and Programming Languages (TAPL)
<https://mitpress.mit.edu/books/types-and-programming-languages>
- [B7] Purely Functional Data Structures
<http://www.cambridge.org/us/academic/subjects/computer-science/programming-languages-and-applied-logic/purely-functional-data-structures>
- [B8] Algorithms: A Functional Programming Approach
<http://catalogue.pearsoned.co.uk/catalog/academic/product/0,1144,0201596040,00.html>

References

- [D1] Laziness
<http://dev.stephendiehl.com/hask/#laziness>
- [D2] Being Lazy with Class
<http://www.seas.upenn.edu/~cis194/lectures/06-laziness.html>
- [D3] A Haskell Compiler
<http://www.scs.stanford.edu/14sp-cs240h/slides/ghc-compiler-slides.html>
<http://www.scs.stanford.edu/11au-cs240h/notes/ghc-slides.html>
- [D4] Evaluation
http://dev.stephendiehl.com/fun/005_evaluation.html
- [D5] Incomplete Guide to Lazy Evaluation (in Haskell)
<https://hackhands.com/guide-lazy-evaluation-haskell>
- [D6] Laziness
<https://www.fpcomplete.com/school/starting-with-haskell/introduction-to-haskell/6-laziness>
- [D7] Evaluation on the Haskell Heap
<http://blog.ezyang.com/2011/04/evaluation-on-the-haskell-heap>
- [D8] Lazy Evaluation of Haskell
<http://www.vex.net/~trebla/haskell/lazy.xhtml>
- [D9] Fixing foldl
<http://www.well-typed.com/blog/2014/04/fixing-foldl>
- [D10] How to force a list
<https://ro-che.info/articles/2015-05-28-force-list>

References

- [D11] Evaluation order and state tokens
<https://www.fpcomplete.com/user/snoyberg/general-haskell/advanced/evaluation-order-and-state-tokens>
- [D12] Reasoning about laziness
<http://blog.johantibell.com/2011/02/slides-from-my-talk-on-reasoning-about.html>
- [D13] Some History of Functional Programming Languages
http://www-fp.cs.st-andrews.ac.uk/tifp/TFP2012/TFP_2012/Turner.pdf
- [D14] Why Functional Programming Matters
<https://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf>
- [D15] GHC illustrated
http://takenobu-hs.github.io/downloads/haskell_ghc_illustrated.pdf

References

[W1] Haskell/Laziness

<https://en.wikibooks.org/wiki/Haskell/Laziness>

[W2] Lazy evaluation

https://wiki.haskell.org/Lazy_evaluation

[W3] Lazy vs. non-strict

https://wiki.haskell.org/Lazy_vs._non-strict

[W4] Haskell/Denotational semantics

https://en.wikibooks.org/wiki/Haskell/Denotational_semantics

[W5] Haskell/Graph reduction

https://en.wikibooks.org/wiki/Haskell/Graph_reduction

[W6] Performance/Strictness

<https://wiki.haskell.org/Performance/Strictness>

References

- [S1] Hackage
<https://hackage.haskell.org>

- [S2] Hoogle
<https://www.haskell.org/hoogle>

Lazy,... zzz

to be as lazy as possible...