Lazy evaluation illustrated

for Haskell divers

exploring some mental models and implementations

Takenobu T.

Rev. 0.00.0
WIP

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

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..
}
```

?

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 program is a collection of statements.

A sequence of statements

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

Statements are executed downward.

A nested structure

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

from inner to outer

A sequence of arguments

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

from left to right

A function and arguments

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

arguments first
apply second

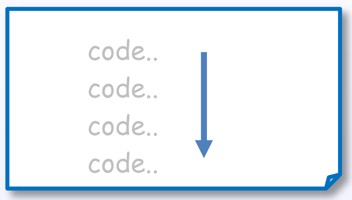
Each programmer has some mental models in their brain.

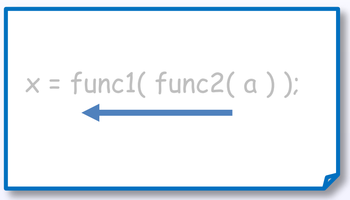
One of the mental models for C program

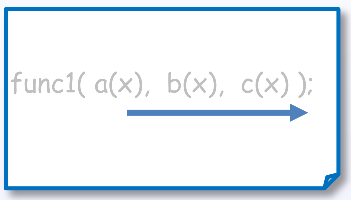
Maybe, You have some implicit mental model in your brain for C program.

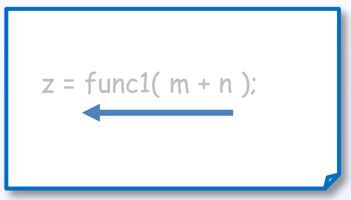
(1) A program is a collection of statements.

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

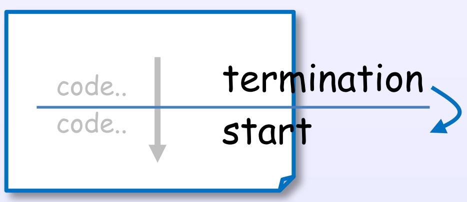


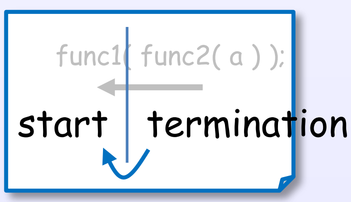






(3) There is the order between termination and start of evaluations.





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  =  exp11  (exp12  exp13 exp14 )

exp13  =  exp131  exp132

exp14  =  exp141  exp142 exp143
   :
```

?

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

One of the mental models for Haskell program

Haskell program

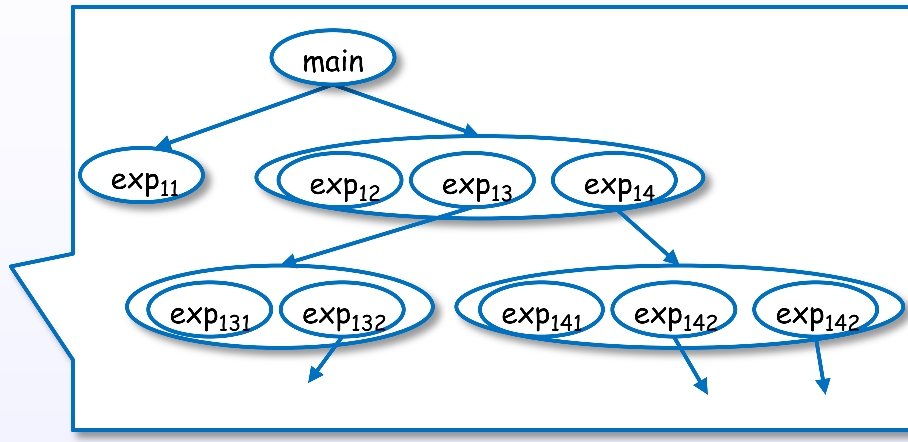
A program is a collection of expressions.

main = exp_{11} (exp_{12} exp_{13} exp_{14})

exp_{13} = exp_{131} exp_{132}

exp_{14} = exp_{141} exp_{142} exp_{143}

:



main = exp_{11} (exp_{12} (exp_{131} exp_{132}) (exp_{141} exp_{142} exp_{143}))

A entire program is regarded as a single expression.

The subexpression is evaluated (reduced) in some order.

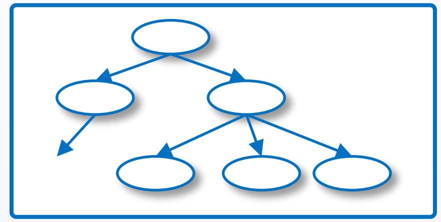
The evaluation is performed by replacement.

One of the mental models for Haskell program

(1) A program is a collection of expressions.

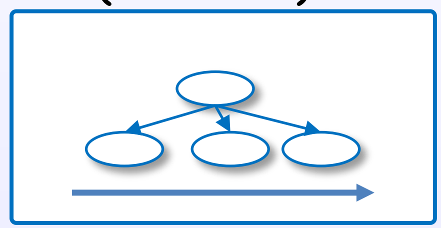
(2) A entire program is regarded as a single expression.

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

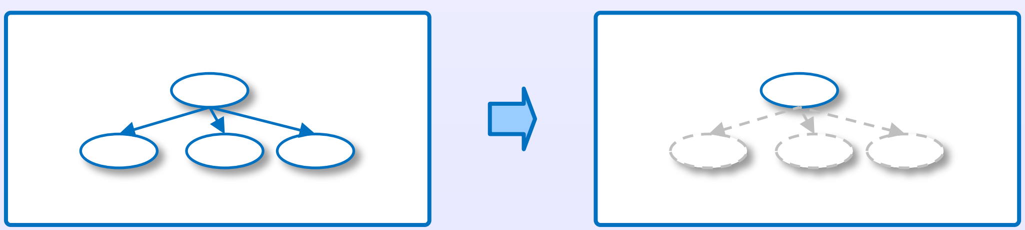


(3) The subexpressions are evaluated (reduced) in some order.

```
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.

Lazy evaluation

Why lazy evaluation?

To manipulate infinite data structures

To avoid unnecessary computation

To manipulate streames

modularity

pure is order free

abstraction

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 ☺

References : [H4], [H3], [B2], [B7], [B8], [D2], [D12], [D13], [D14]

Haskell(GHC) 's lazy evaluation

Lazy evaluation

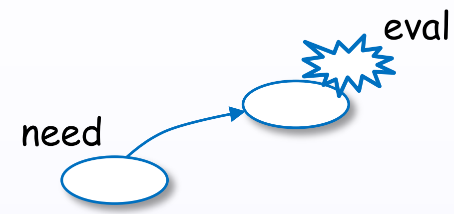
evaluate only if needed

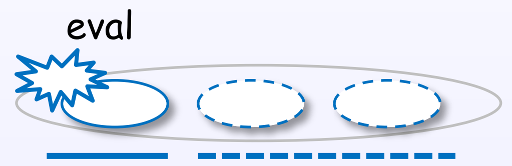
+

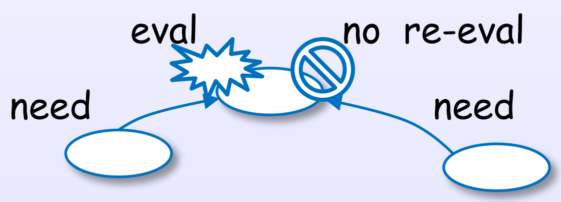
evaluate only enough

+

evaluate at most once







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

References : [B2] Ch.7, [H4] Ch.11, 12, [D2]

Ingredient of Haskell(GHC) 's lazy evaluation

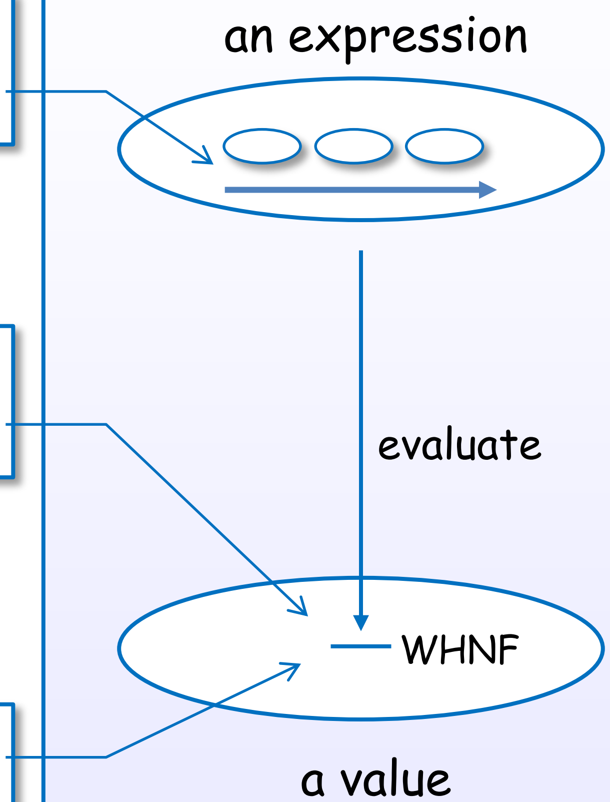
only if needed — normal order reduction

+

only enough — stop at WHNF

+

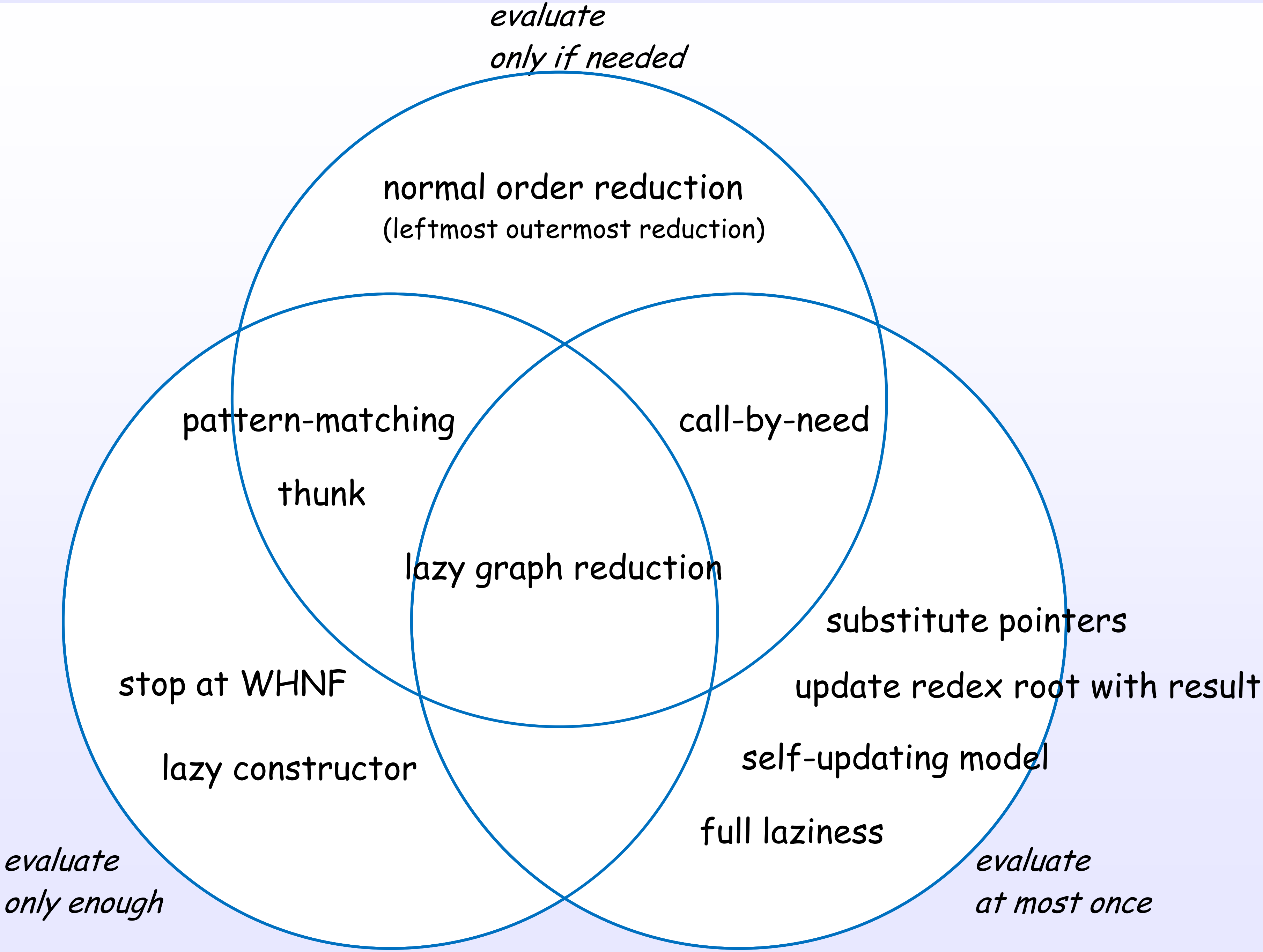
at most once — substitute pointers, update redex root with result



This strategy is implemented by lazy graph reduction.

References : [B2] Ch.7, [H4] Ch.11, 12, [D2]

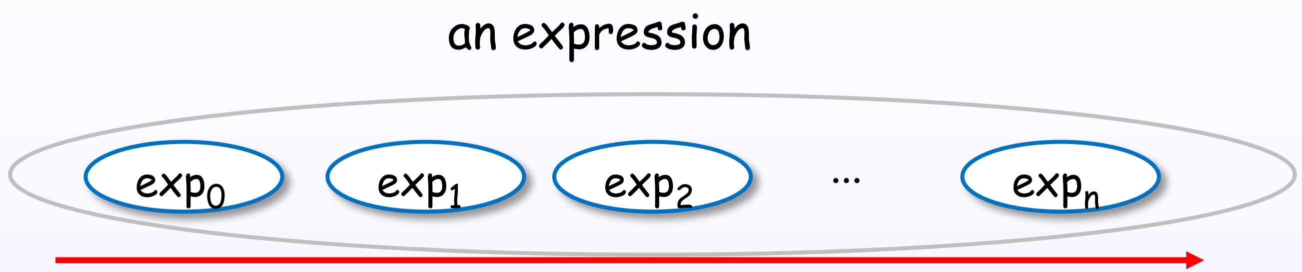
Techniques of Haskell(GHC) 's lazy evaluation



References : [B2] Ch.7, [H4] Ch.2, 11, 12, 15, [H5], [D2]

Simple questions

What order?



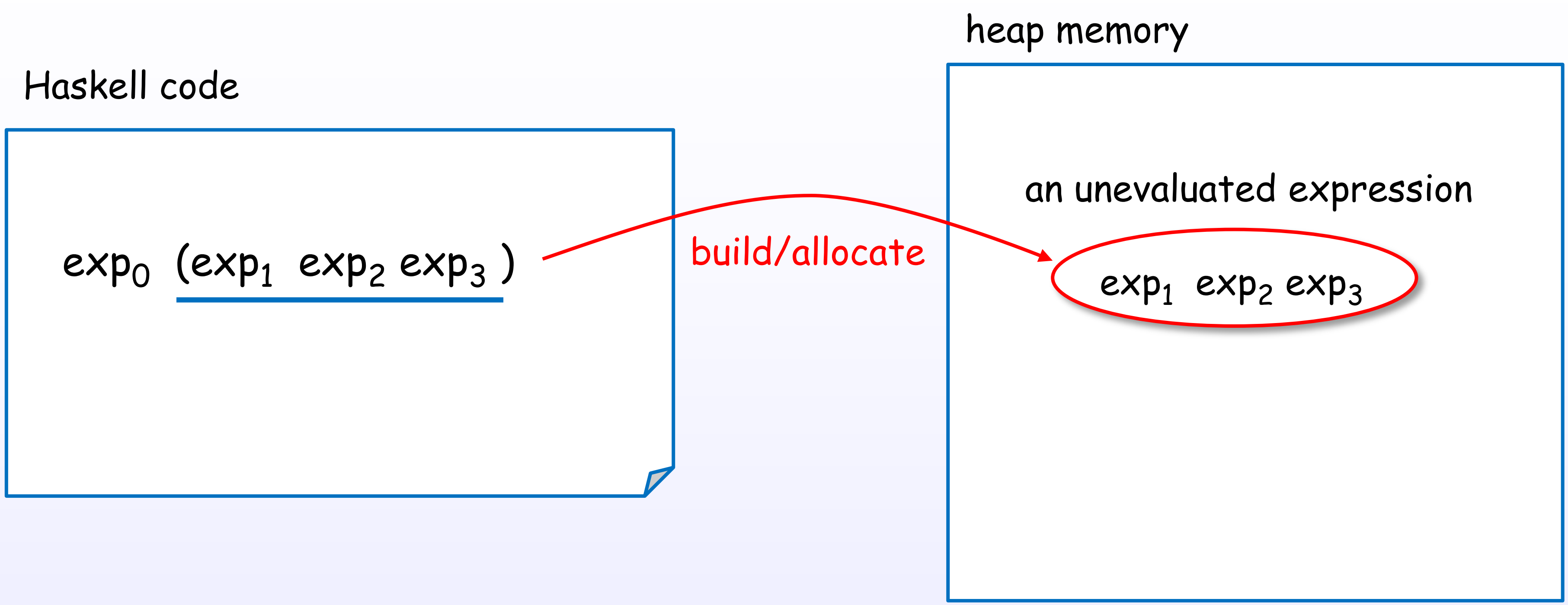
An expression is evaluated by normal order (leftmost outermost redex first).

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.

References : [H4] Ch.2, 11, [B6] Ch.5

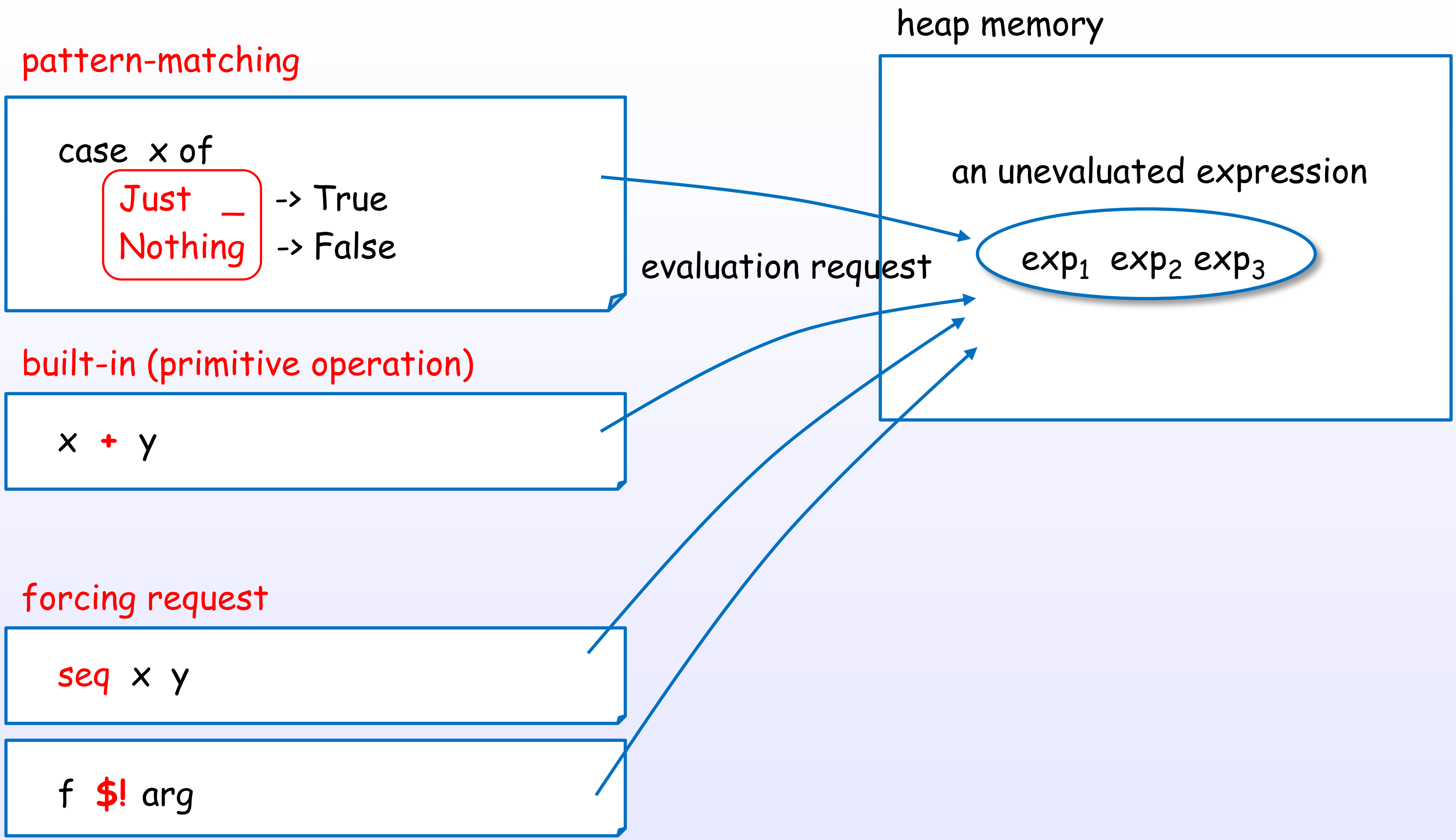
How to postpone?



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

References : [H4], [H5]

When needed?

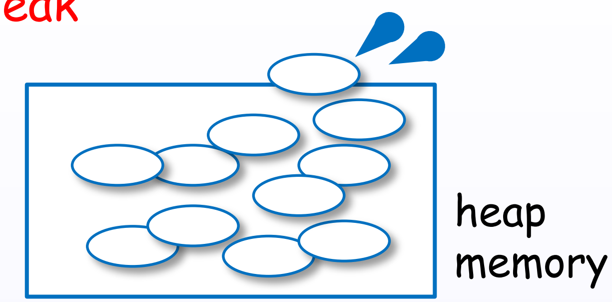


Pattern-matching or forcing request drive the evaluation.

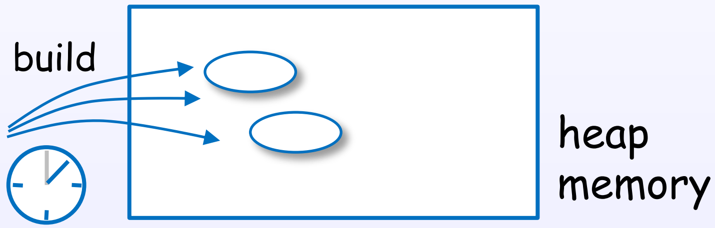
References : [H4], [D2], [D5]

What to be careful about?

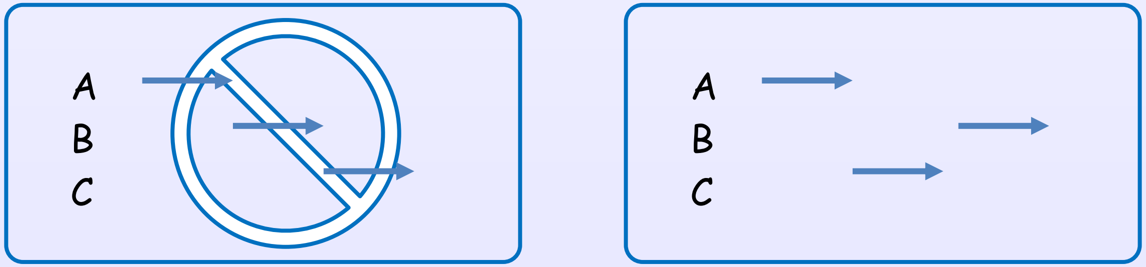
To consider hidden space leak



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.

References : [H4], [D2], [D5]

2. Expressions

Expression and value

What is an expression?

An expression

?

References : [B1] Ch.1, [B2] Ch.2, [B6] Ch.3

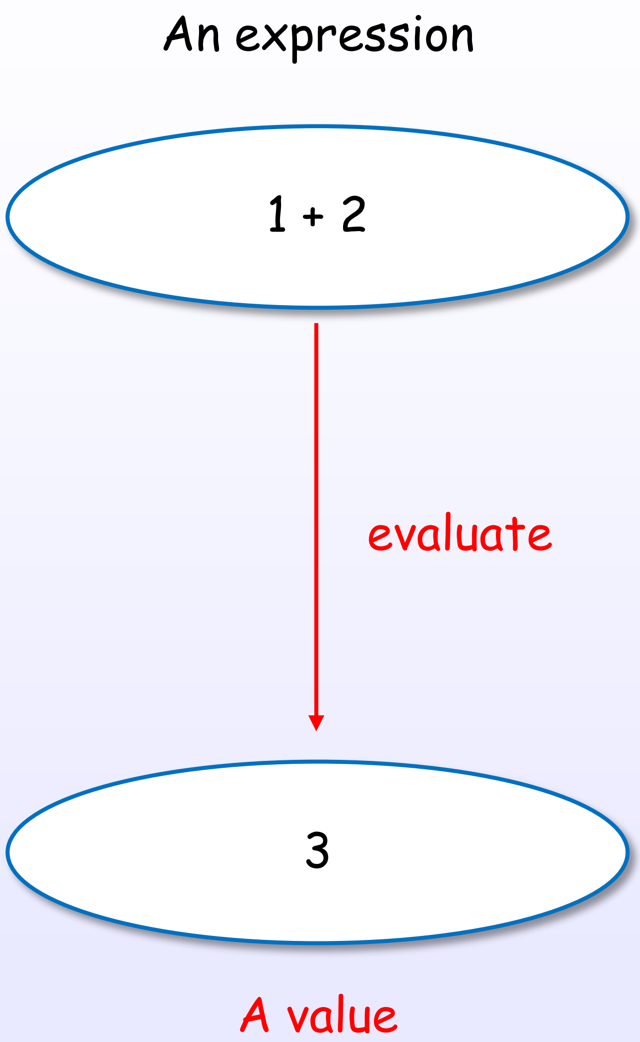
An expression denotes a value

An expression

1 + 2

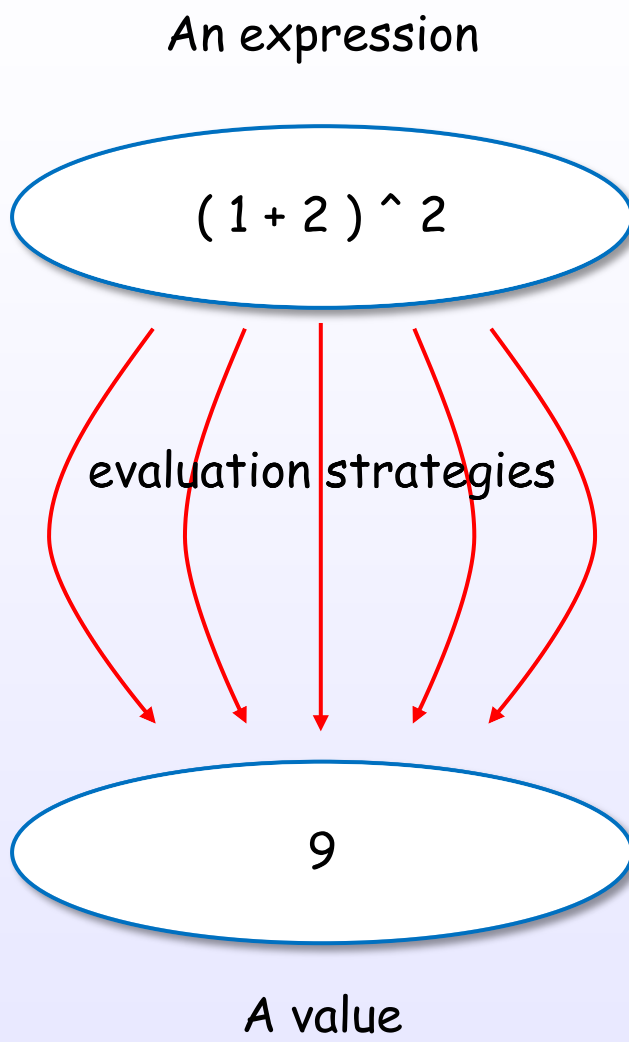
References : [B1] Ch.1, [B2] Ch.2, [H1] Ch.1, [B6] Ch.3

An expression evaluates to a value



References : [B1] Ch.1, [B2] Ch.2, [H1] Ch.1, [B6] Ch.3

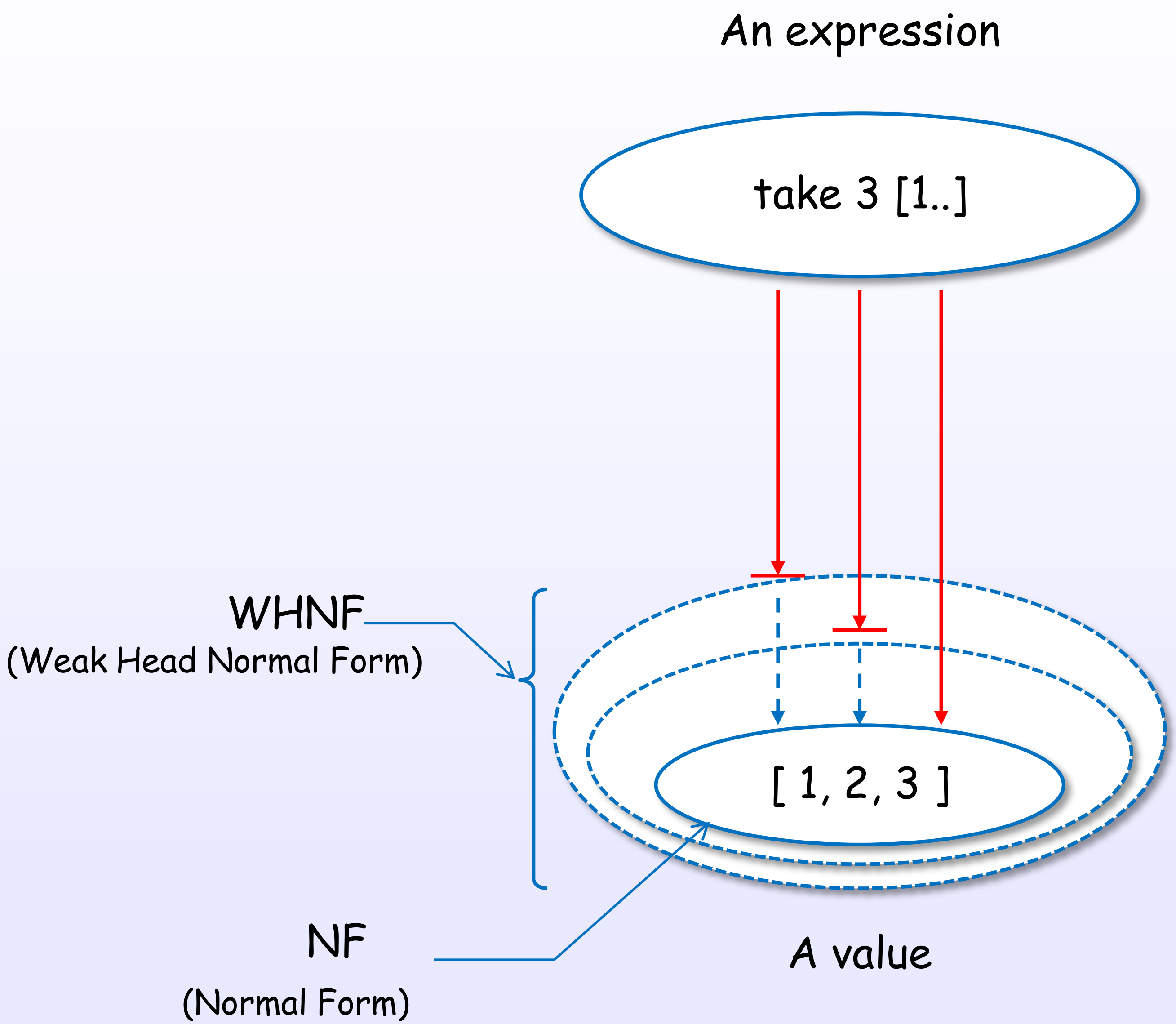
There are many evaluation approaches



- Strict, Non-strict evaluation
- Eager, Lazy evaluation
- Call-by-value, Call-by-name, Call-by-need, …
- Innermost, Outermost
- Normal order, Applicative order
- …

References : [B2] Ch.2, 7, [B6] Ch.3, [D1]

There are some evaluation levels



References : [D3], [B2] Ch.2, 7, [B6] Ch.3, [D1]

2. Expressions

Expressions in Haskell

There are many expressions in Haskell

Expressions

Just 5

1 + 2

(1, 2)

take 5 xs

[1, 2, 3]

let x = 1 in x + y

'a'

map f xs

if b then 1 else 0

7

¥x -> x + 1

x : xs

fun arg

case x of _ -> 0

(¥x -> x + 1) 3

do {x <- get; put x}

xs

categorizing

References : [B2] Ch.2, [H1] Ch.3

Expression categories in Haskell

lambda abstraction

`¥x -> x + 1`

let expression

`let x = 1 in x + y`

conditional

`if b then 1 else 0`

case expression

`case x of _ -> 0`

do expression

`do {x <- get; put x}`

function application

`take 5 xs`

`(¥x -> x + 1) 3`

`1 + 2`

`map f xs`

`fun arg`

general constructor, literal and some forms

`7`

`[1, 2, 3]`

`(1, 2)`

`'a'`

`x : xs`

`Just 5`

variable

`xs`

References : [H1] Ch.3, [B2] Ch.2

Specification is defined in Haskell 2010 Language Report

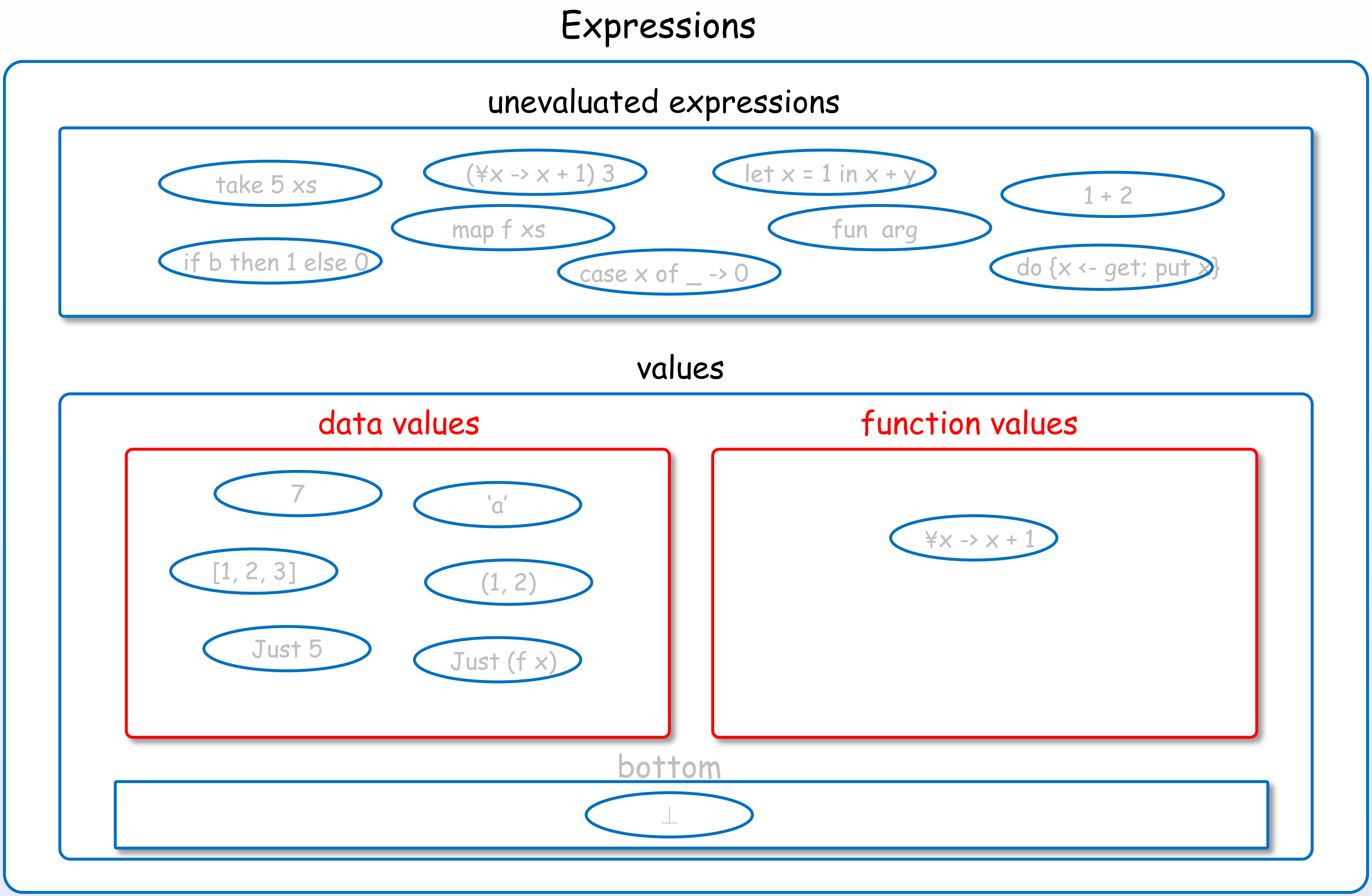
"Haskell 2010 Language Report, Chapter 3 Expressions" [H1]

exp	→	*infixexp* `::` [*context* `=>`] *type*	(expression type signature)
	|	*infixexp*	
infixexp	→	*lexp qop infixexp*	(infix operator application)
	|	`-` *infixexp*	(prefix negation)
	|	*lexp*	
lexp	→	`\` $apat_1$ … $apat_n$ `->` *exp*	(lambda abstraction, $n \geq 1$)
	|	`let` *decls* `in` *exp*	(let expression)
	|	`if` *exp* [`;`] `then` *exp* [`;`] `else` *exp*	(conditional)
	|	`case` *exp* `of` `{` *alts* `}`	(case expression)
	|	`do` `{` *stmts* `}`	(do expression)
	|	*fexp*	
fexp	→	[*fexp*] *aexp*	(function application)
aexp	→	*qvar*	(variable)
	|	*gcon*	(general constructor)
	|	*literal*	
	|	`(` *exp* `)`	(parenthesized expression)
	|	`(` exp_1 `,` … `,` exp_k `)`	(tuple, $k \geq 2$)
	|	`[` exp_1 `,` … `,` exp_k `]`	(list, $k \geq 1$)
	|	`[` exp_1 [`,` exp_2] `..` [exp_3] `]`	(arithmetic sequence)
	|	`[` *exp* `|` $qual_1$ `,` … `,` $qual_n$ `]`	(list comprehension, $n \geq 1$)
	|	`(` *infixexp qop* `)`	(left section)
	|	`(` $qop_{\langle - \rangle}$ *infixexp* `)`	(right section)
	|	*qcon* `{` $fbind_1$ `,` … `,` $fbind_n$ `}`	(labeled construction, $n \geq 0$)
	|	$aexp_{\langle qcon \rangle}$ `{` $fbind_1$ `,` … `,` $fbind_n$ `}`	(labeled update, $n \geq 1$)

References : [H1] Ch.3, [B2] Ch.2

Classification by values and forms

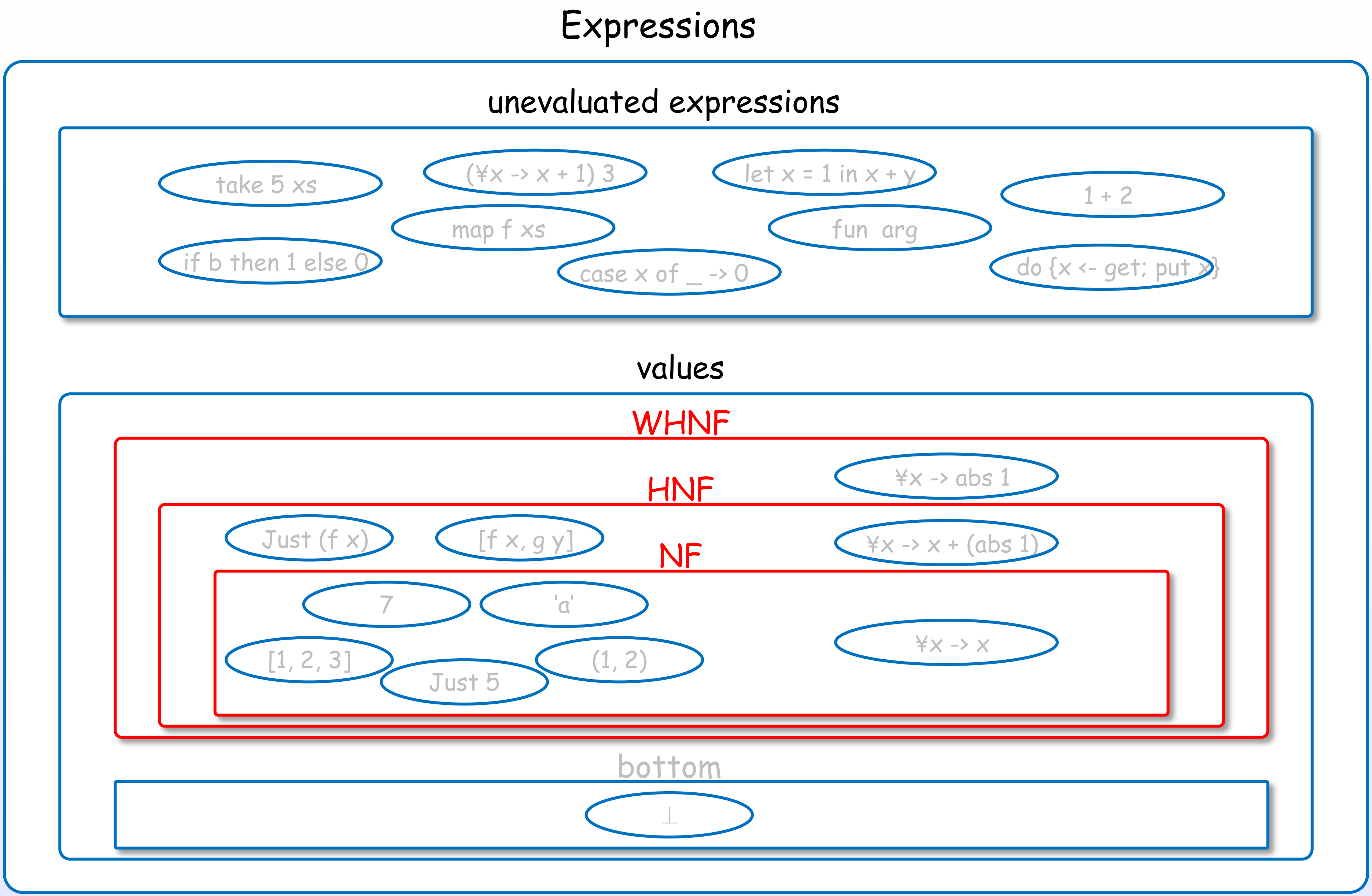
Classification by values



Values are data values or function values.

References : [H5]

Classification by forms



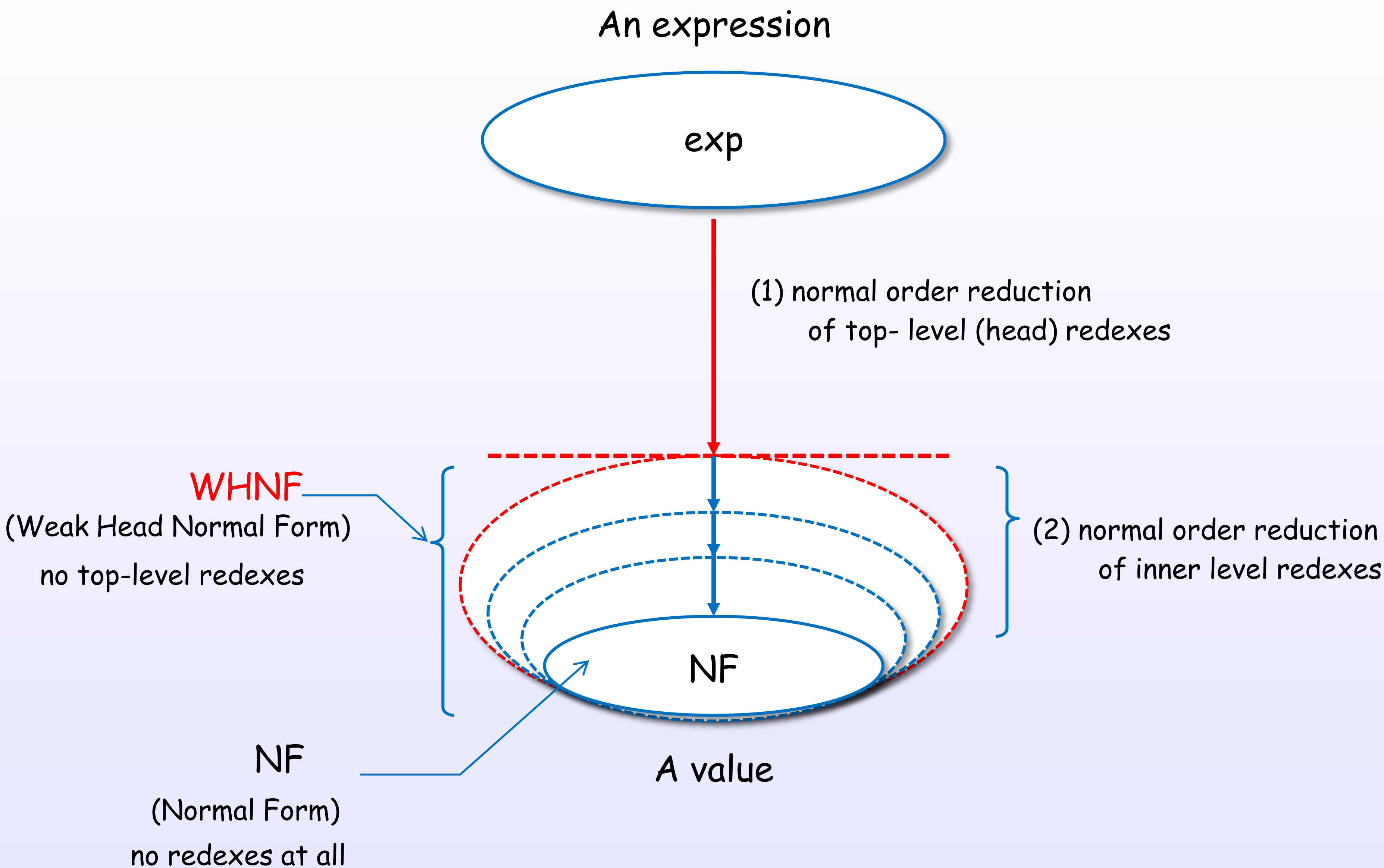
Values are NF, HNF or WHNF.

References : [H5]

2. Expressions

WHNF

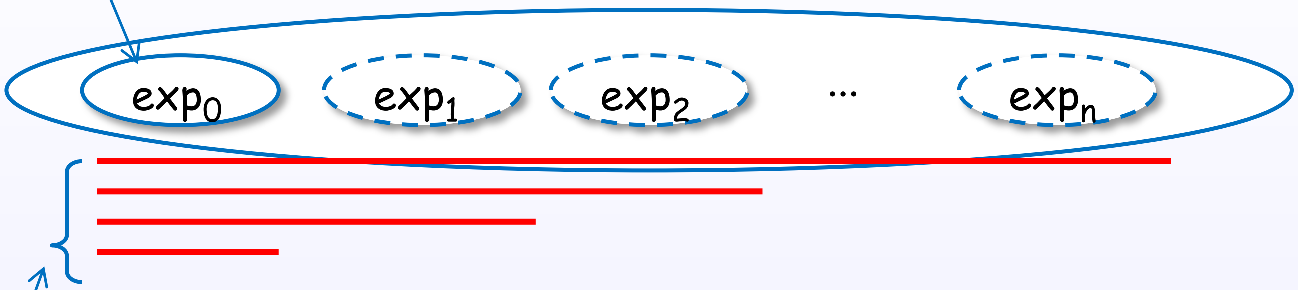
WHNF is one of the evaluated values.



References : [H4] Ch.11, [D3], [B6] Ch.3, [B2] Ch.2, 7, [D1], [W1]

WHNF

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



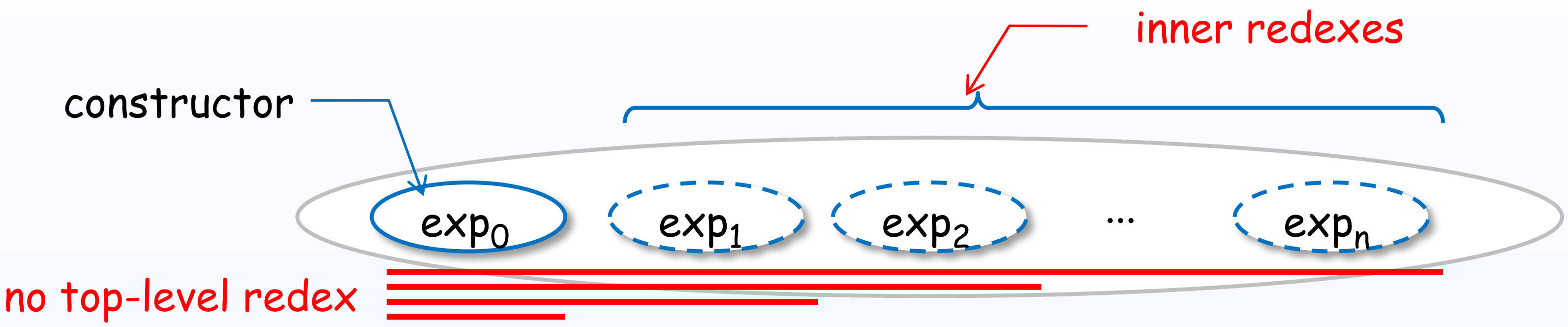
no top-level redex

WHNF is a value which has evaluated top-level

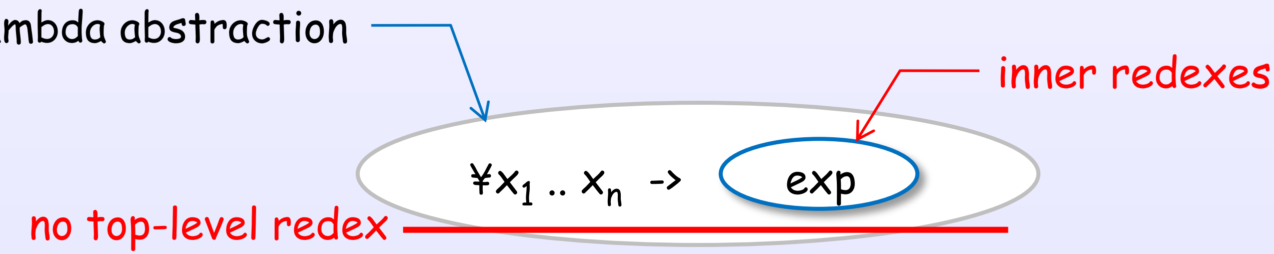
References : [H4] Ch.11, [D3], [B5] Ch.2, [B4] Ch.24, [B6] Ch.3, [B2] Ch.2, 7, [D5], [D1]

WHNF for a data value and a function value

a data value in WHNF



a function value in WHNF



References : [H4] Ch.11, [D3], [B5] Ch.2, [B4] Ch.24, [B6] Ch.3, [B2] Ch.2, 7, [D5], [D1]

Examples of WHNF

WHNF

Just 7
no top-level redex

Just (abs x)
no top-level redex

Cons (f 1) (map f [2..])
no top-level redex

¥x -> x + 1
no top-level redex

---

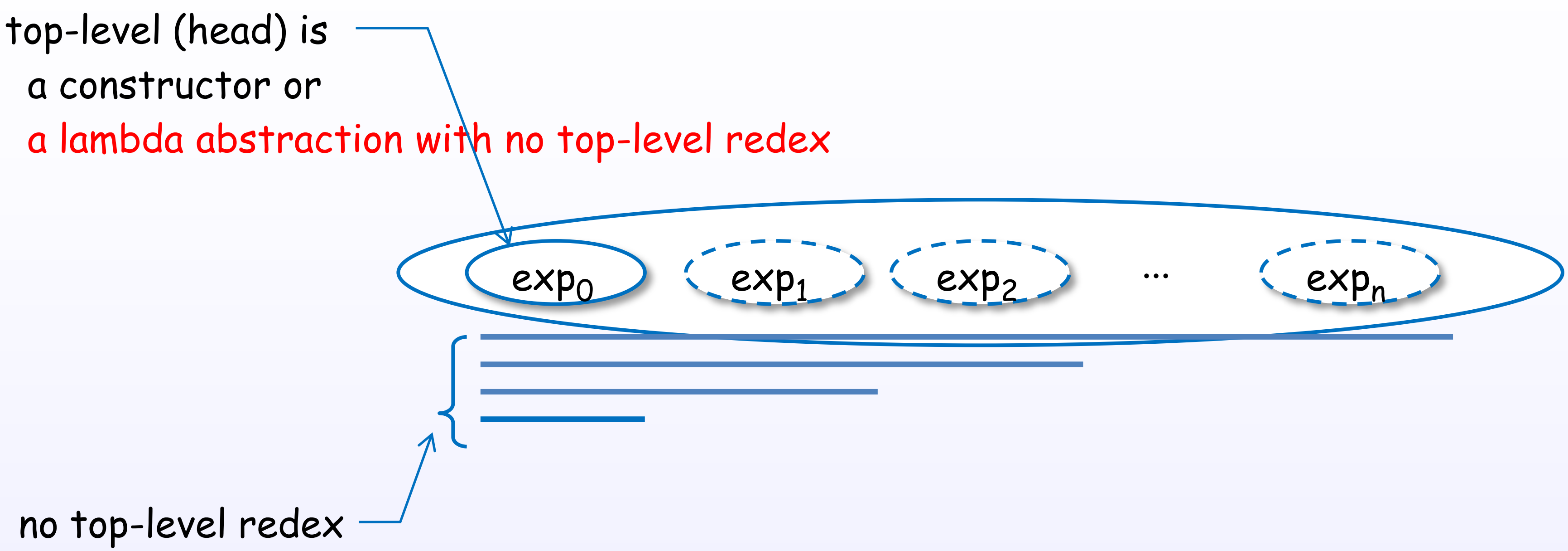
no WHNF

abs 7
top level-redex

if x then True else False
top level-redex

References : [H4] Ch.11, [D3], [B5] Ch.2, [B4] Ch.24, [B6] Ch.3, [B2] Ch.2, 7, [D5], [D1]

HNF



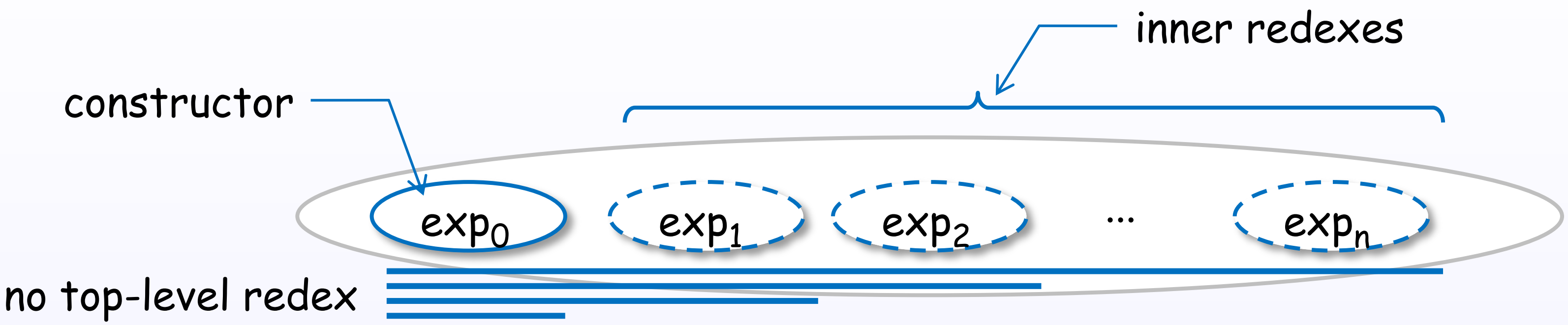
HNF is a value which has evaluated top-level

* GHC uses WHNF rather than HNF.

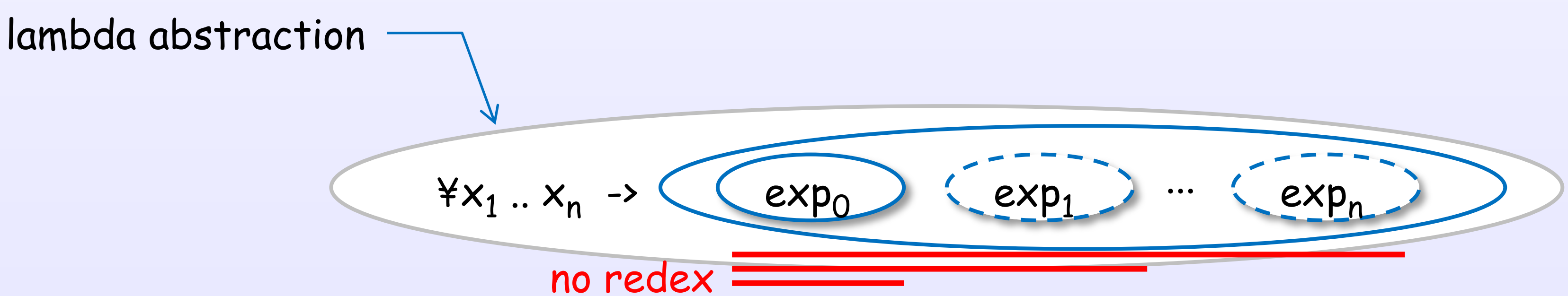
References : [H4] Ch.11, [D3], [B3]

HNF for a data value and a function value

a data value in HNF (same as WHNF)



a function value in HNF



References : [H4] Ch.11, [D3], [B3]

Examples of HNF

HNF

Just 7

no top-level redex

Just (abs x)

no top-level redex

¥x -> Just (abs 7)

no top-level redex

---

no HNF

abs 7

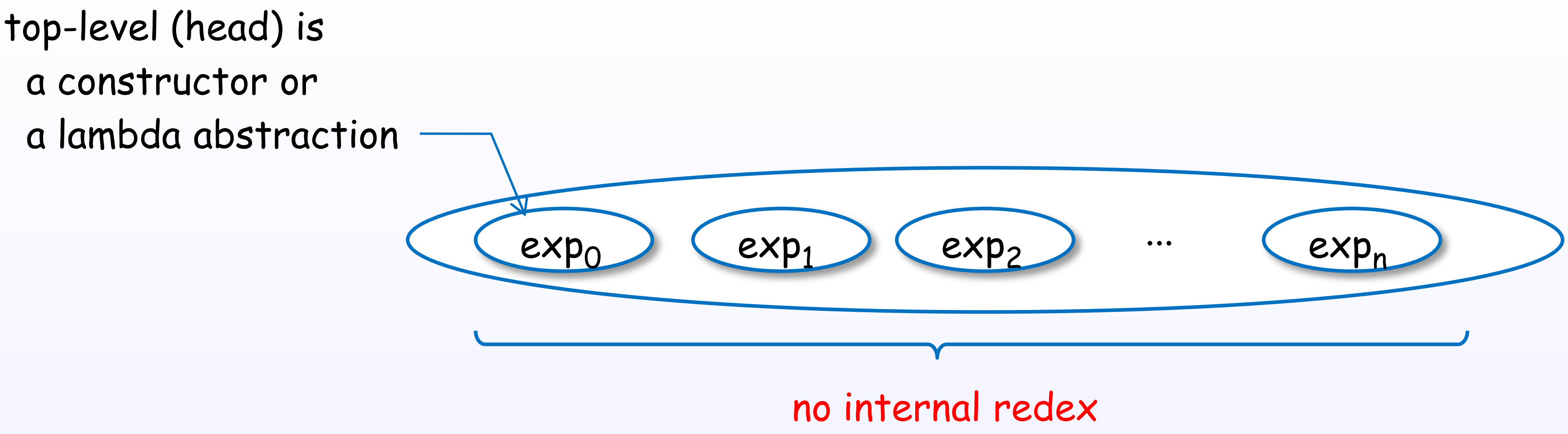
top level-redex

¥x -> abs 7

top level-redex

References : [H4] Ch.11, [D3], [B3]

NF

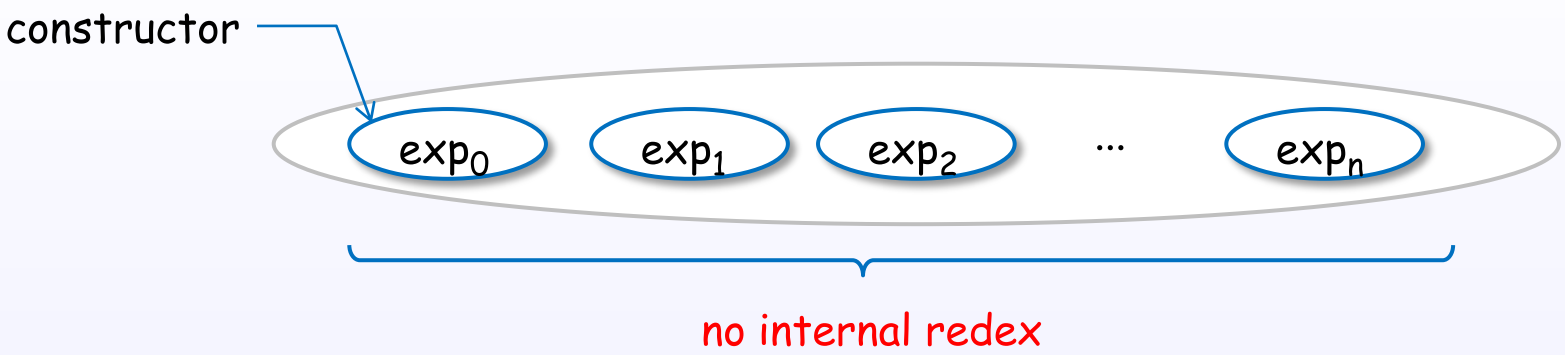


NF is a value which has no redex.

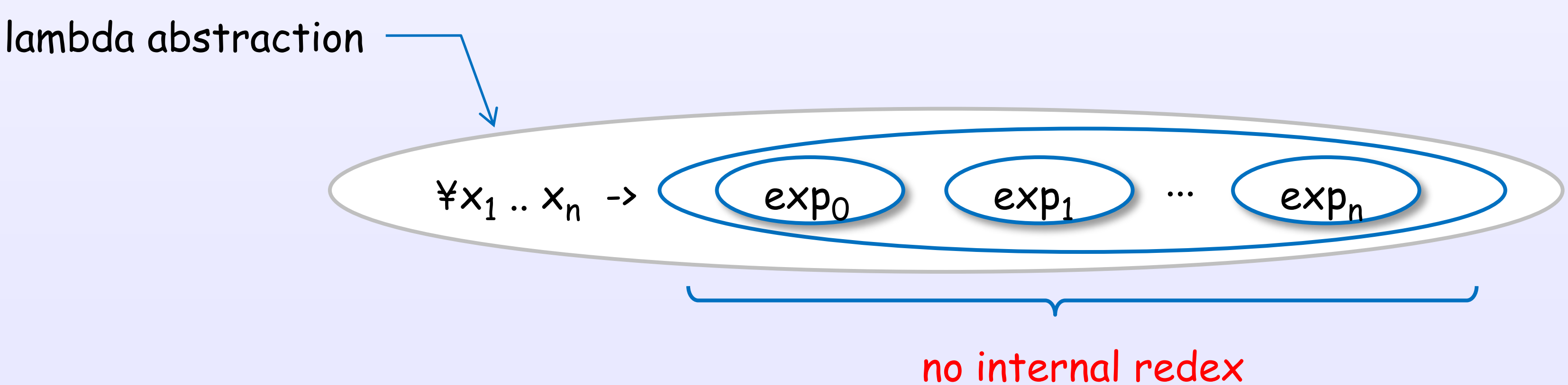
References : [H4] Ch.11, [D3], [B5] Ch.2, [B4] Ch.24, [B6] Ch.3, [B2] Ch.2, 7, [D5], [D1]

NF for a data value and a function value

a data value in NF



a function value in NF



References : [H4] Ch.11, [D3], [B5] Ch.2, [B4] Ch.24, [B6] Ch.3, [B2] Ch.2, 7, [D5], [D1]

Examples of NF

NF

Just 7

no internal redex

Cons 1 Nil

no internal redex

¥x -> x + 1

no internal redex

---

no NF

Just (abs 7)

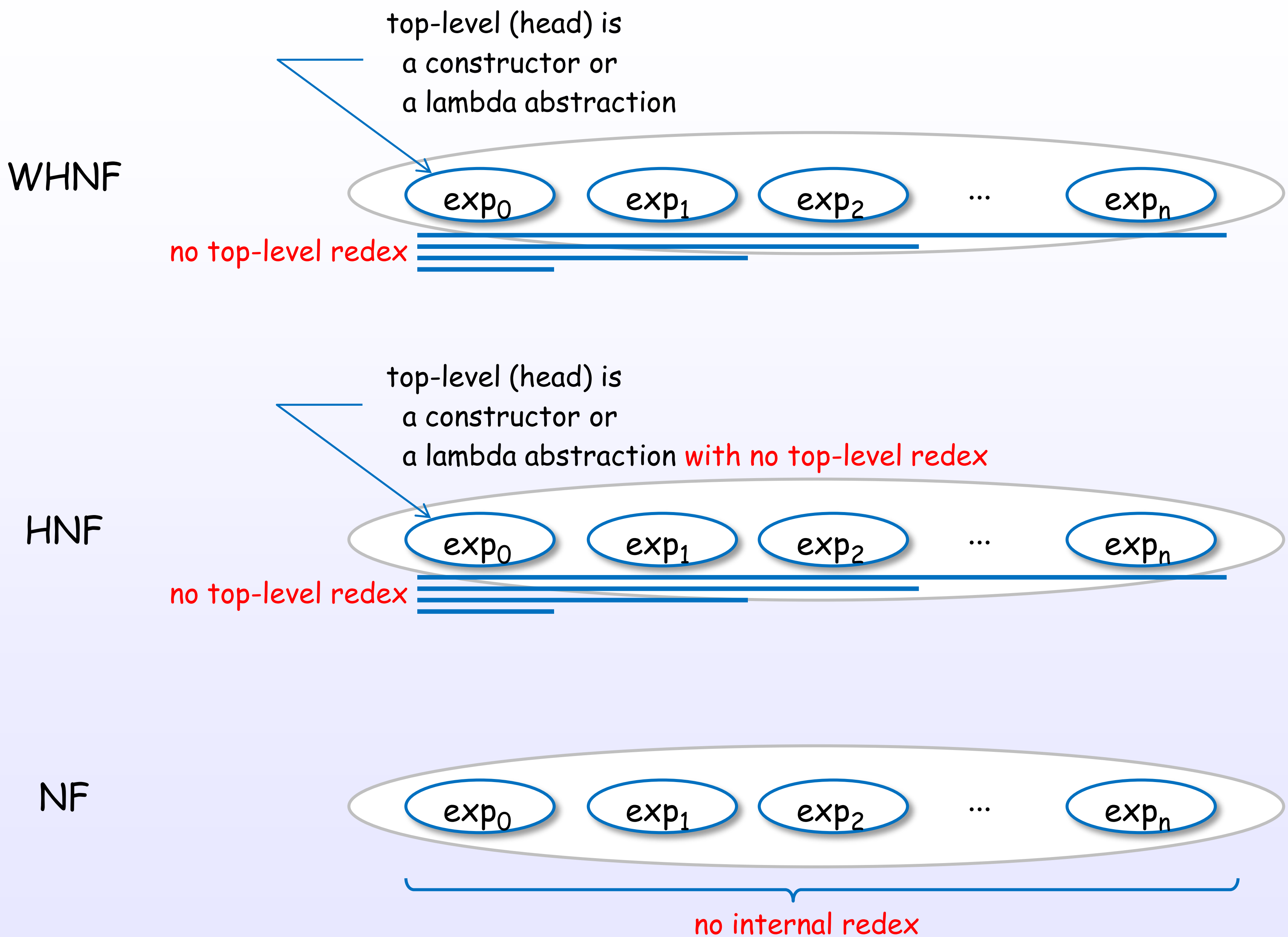
redex

¥x -> Just (abs 7)

redex

References : [H4] Ch.11, [D3], [B5] Ch.2, [B4] Ch.24, [B6] Ch.3, [B2] Ch.2, 7, [D5], [D1]

WHNF, HNF, NF



References : [H4] Ch.11, [D3], [B5] Ch.2, [B4] Ch.24, [B6] Ch.3, [B2] Ch.2, 7, [D5], [D1]

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

$$F\ E_1\ E_2\ \ldots\ E_n$$

where $n \geq 0$;
and either F is a variable or data object
or F is a lambda abstraction or built-in function
and $(F\ E_1\ E_2\ \ldots\ E_m)$ 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.

11.3.3 Head Normal Form

Head normal form is often confused
some discussion. The content of th
since for most purposes head norm
form. Nevertheless, we will stick to t

DEFINITION

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

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

where $n, m \geq 0$;
v is a variable (x_i), a data object, or a built-in function;
and $(v\ M_1\ M_2\ \ldots\ M_p)$ is not a redex for any $p \leq m$.

References : [H4] Ch.11

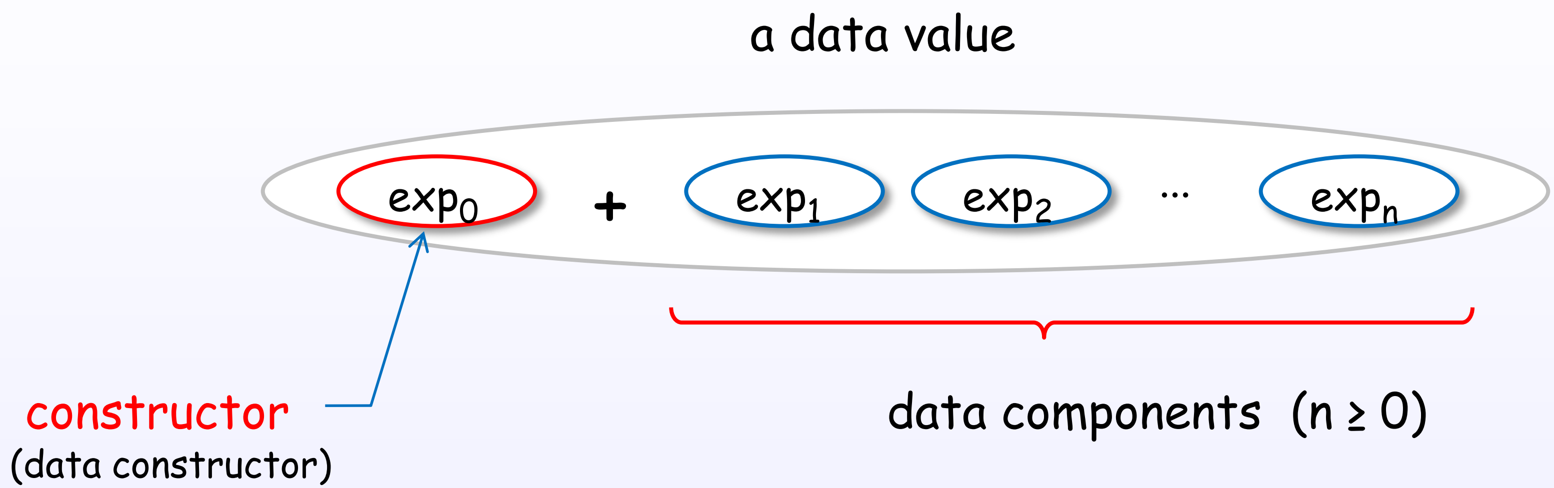
3. Internal representation of expressions

Constructor

Constructor

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

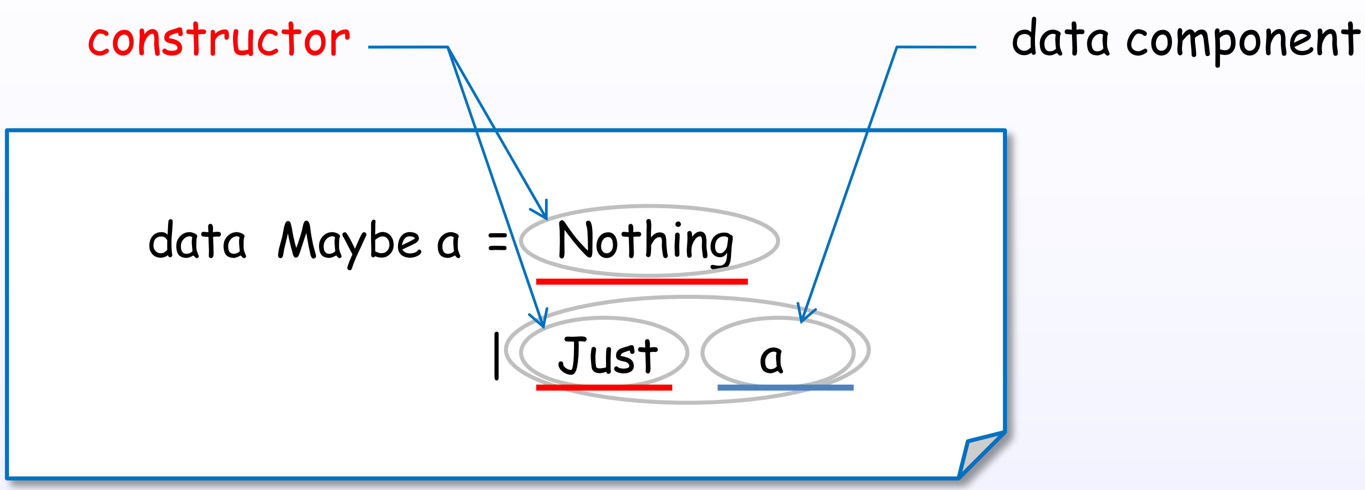
Constructor



A constructor builds a structured data value.

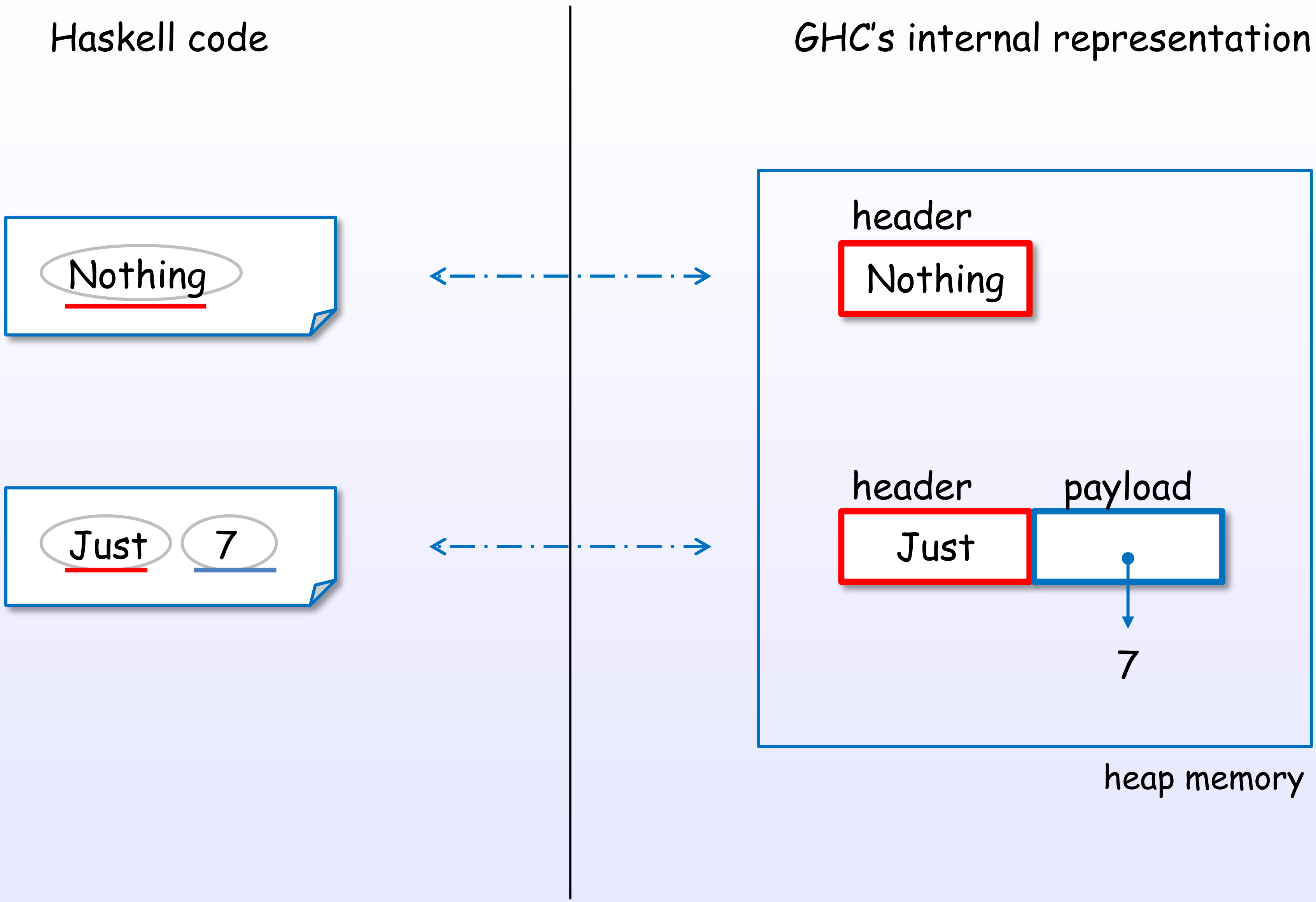
A constructor distinguishes the data value in expressions.

Constructors are defined by data declaration



Constructors are defined by data declaration.

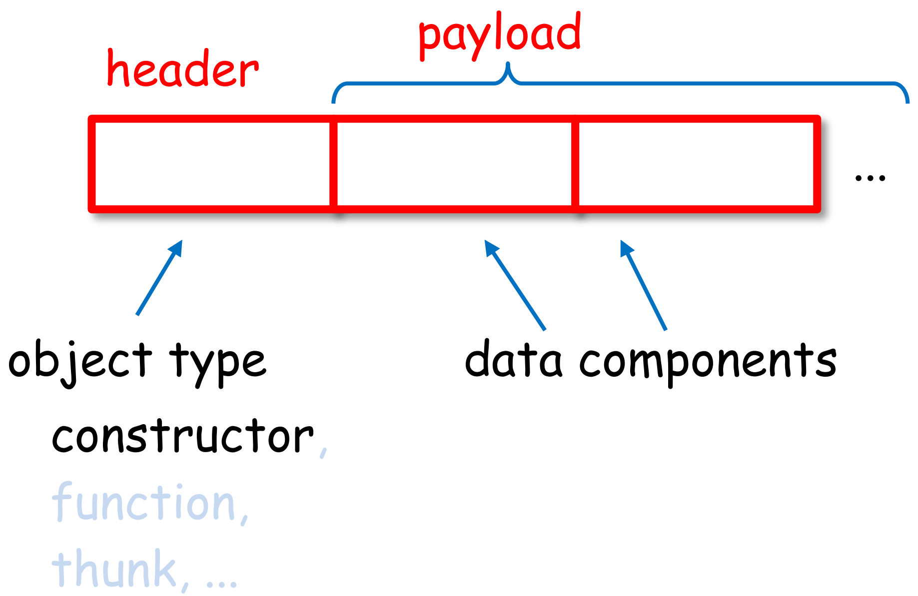
Internal representation of Constructors for data values



References : [H11], [H10], [H5], [H6], [H7]

Constructors are represented uniformly

GHC's internal representation

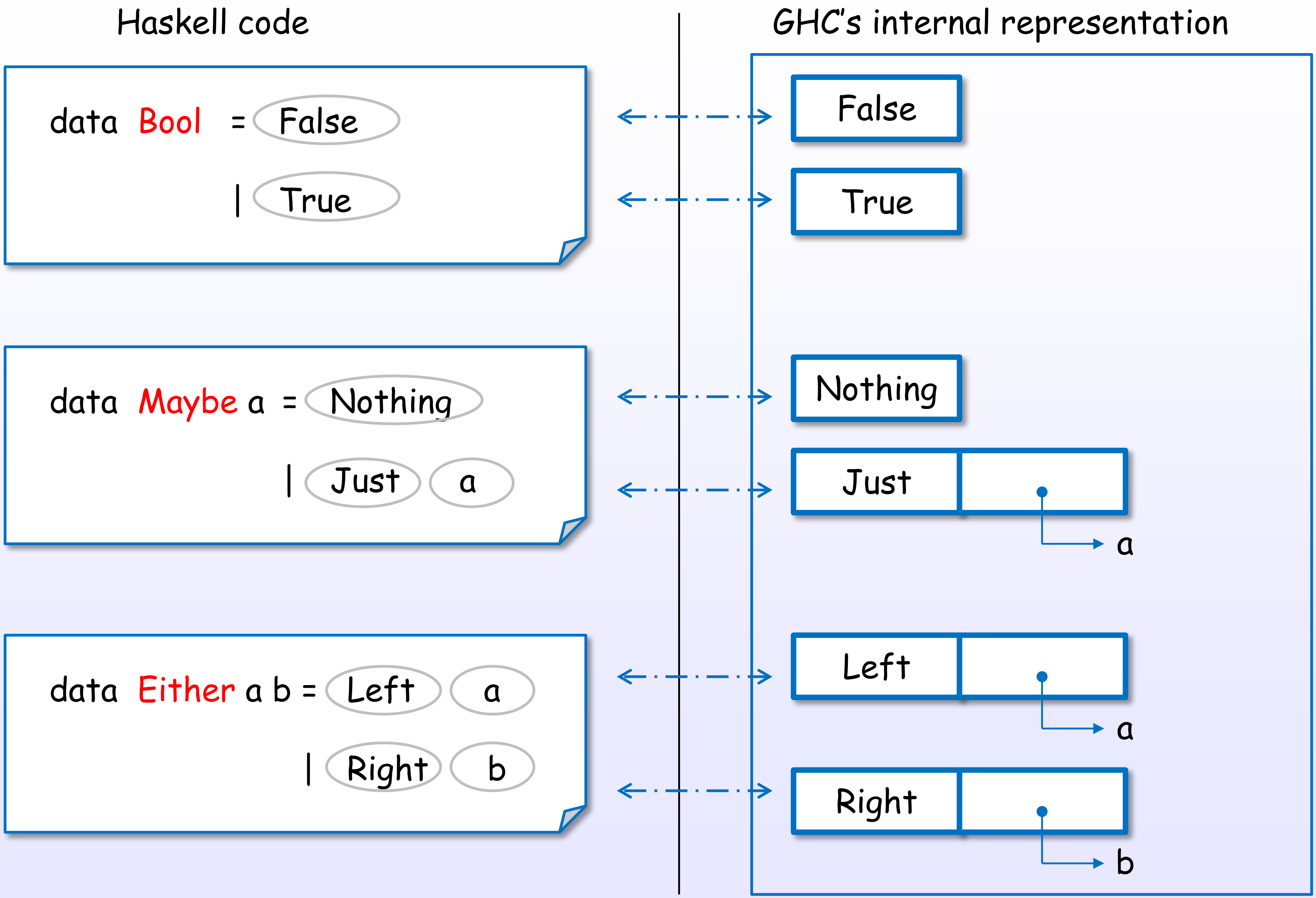


in heap memory, stack or static memory

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

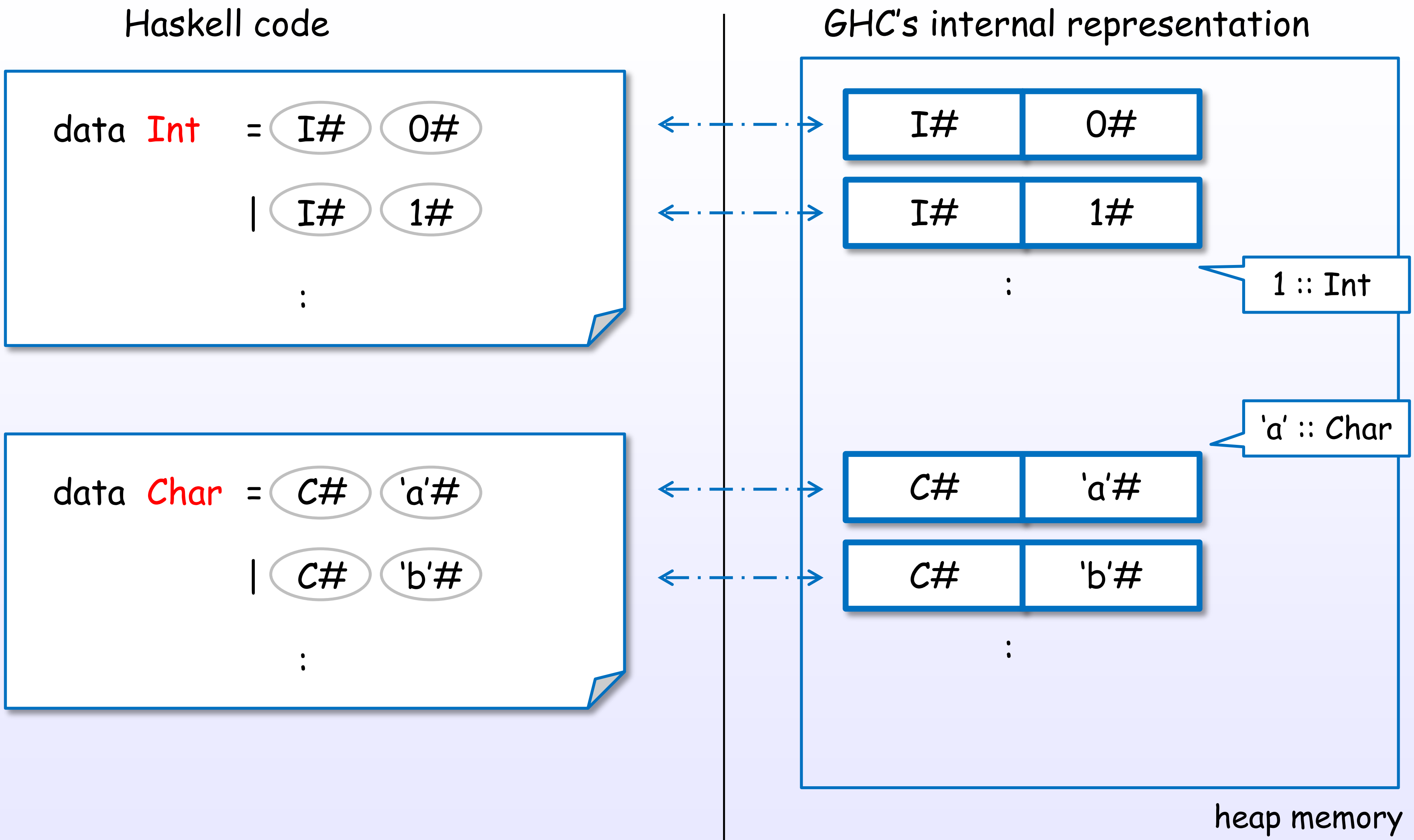
References : [H11], [H10], [H5], [H6], [H7], [D15]

Representation of various constructors



References : [H11], [H10], [H5], [H6], [H7]

Primitive data types are also represented with constructor



References : [H11], [H10], [H5], [H6], [H7]

List is also represented with constructor

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 constructor

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 ) )
```

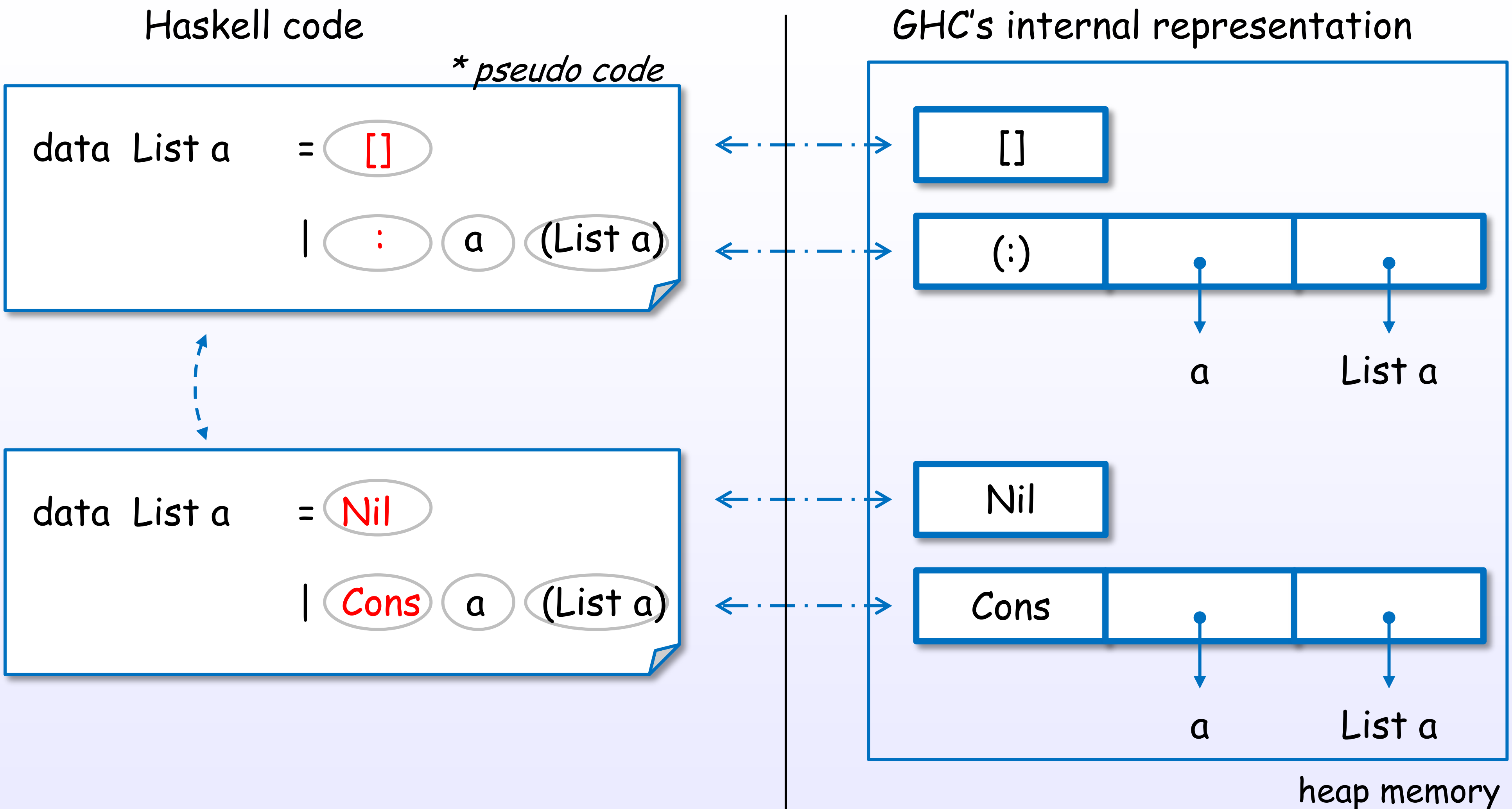
type declaration

** pseudo code*

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

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

List is also represented with constructor



References : [H11], [H10], [H5], [H6], [H7]

List is also represented with constructor

Haskell code

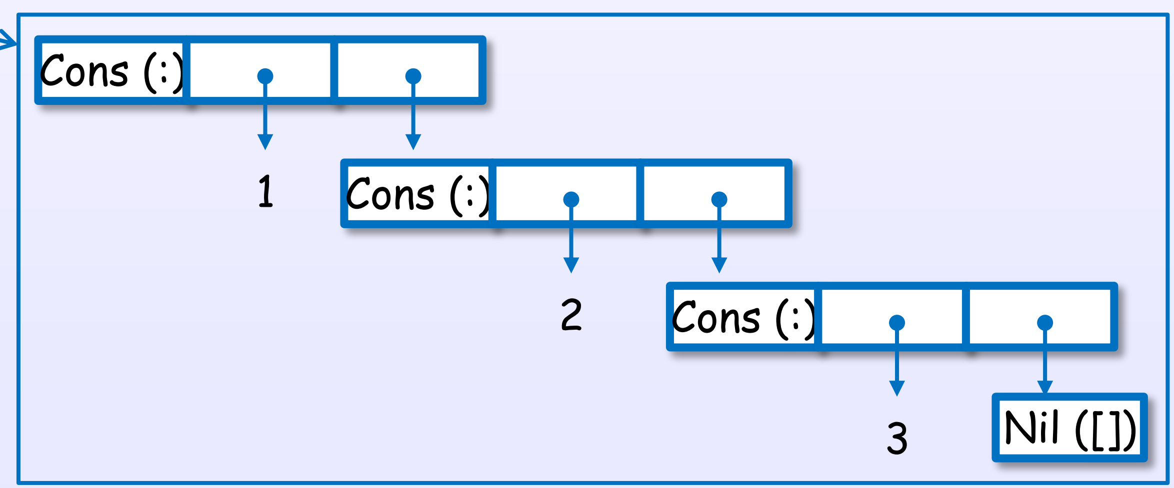
```
[ 1, 2, 3 ]
```

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

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

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

GHC's internal representation



References : [H11], [H10], [H5], [H6], [H7]

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
```

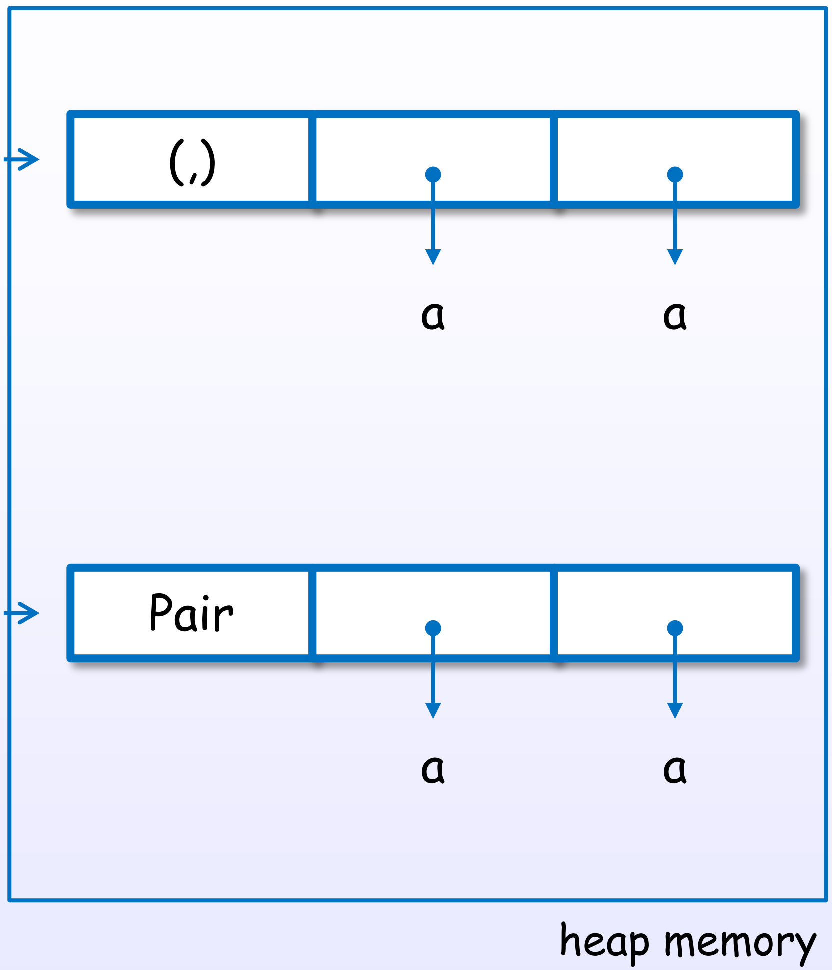
Tuple is also represented with constructor

Haskell code

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

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

GHC's internal representation



References : [H11], [H10], [H5], [H6], [H7]

Tuple is also represented with constructor

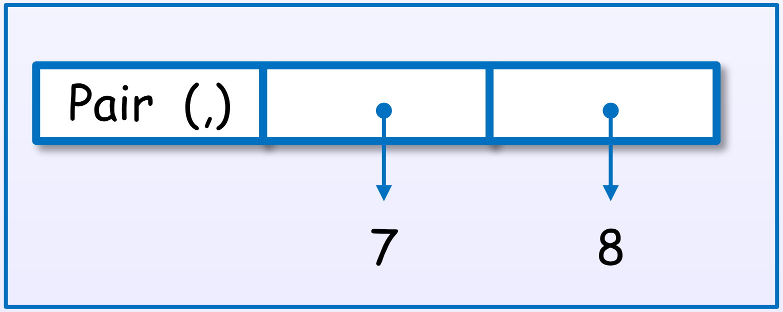
Haskell code

```
( 7, 8 )
```

```
(,) 7 8
```

```
Pair  7  8
```

GHC's internal representation

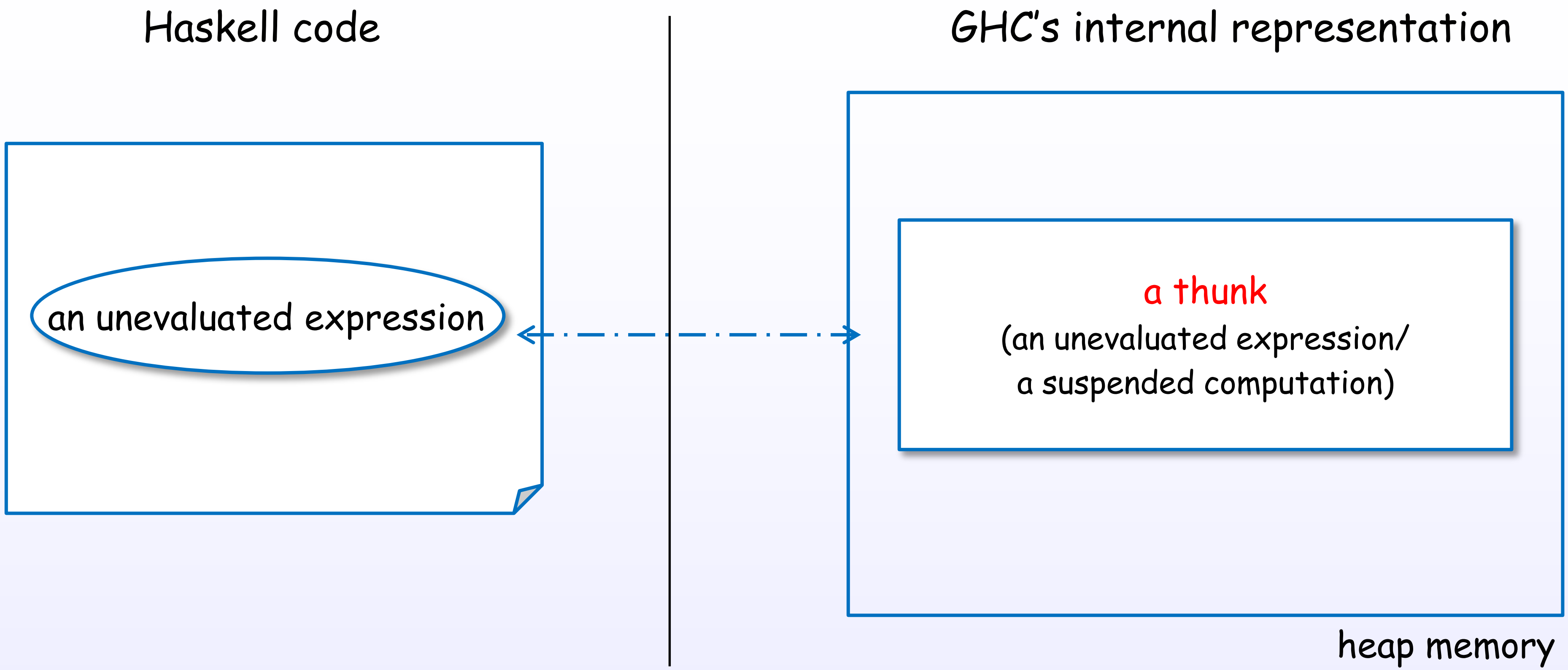


References : [H11], [H10], [H5], [H6], [H7]

3. Internal representation of expressions

Thunk

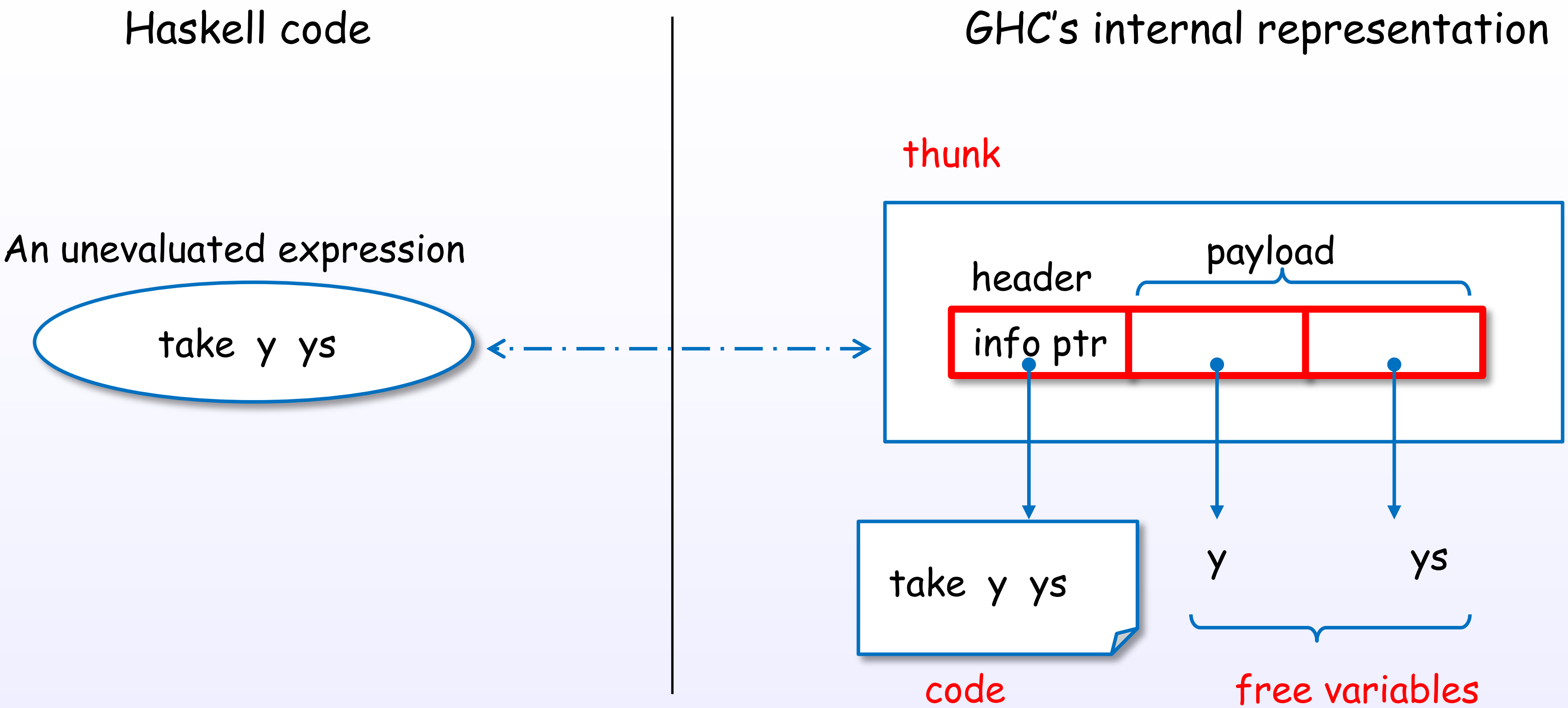
Thunk



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

References : [B5] Ch.2, [D5], [W1], [H10], [H5], [D7]

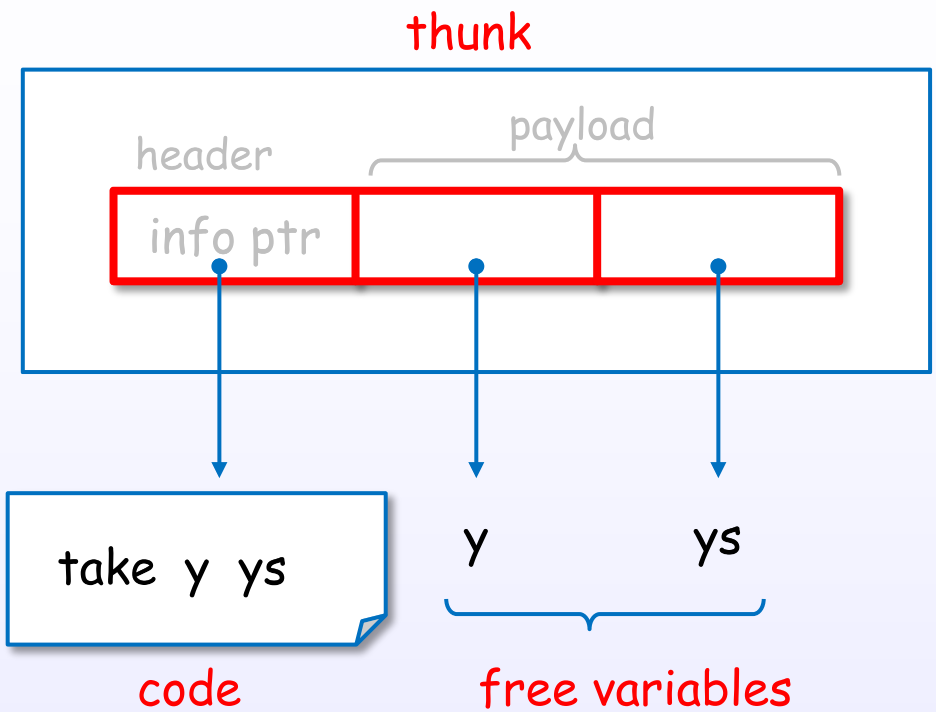
Internal representation of thunk



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

References : [H11], [H10], [D2], [H5], [H6], [H7], [B5] Ch.2, [D5], [W1]

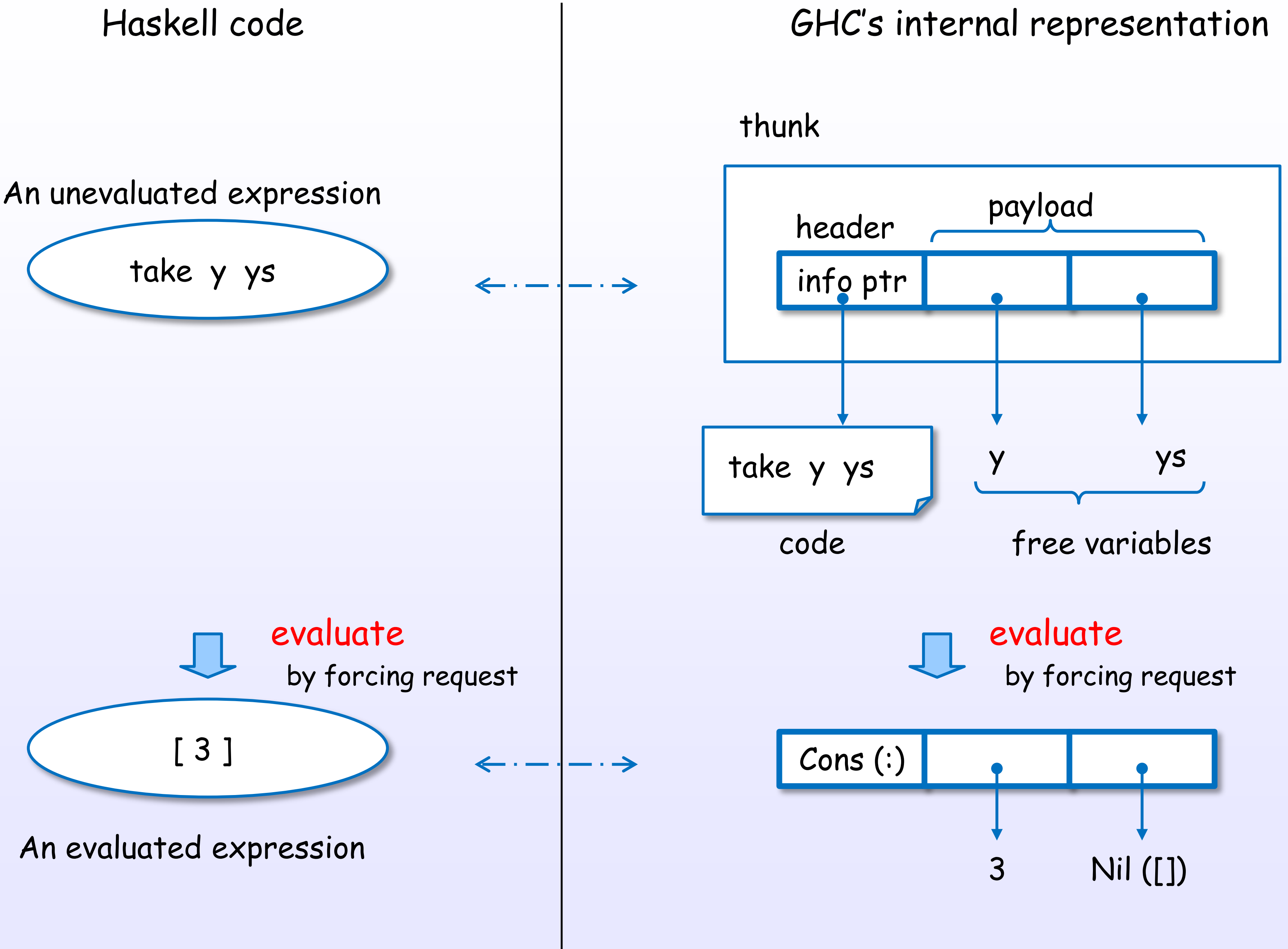
A thunk is a package of code and free variables



A thunk is a package of code + free variables.

References : [D2], [H11], [H10], [H5], [H6], [H7], [B5] Ch.2, [D5], [W1]

A thunk is evaluated by forcing request

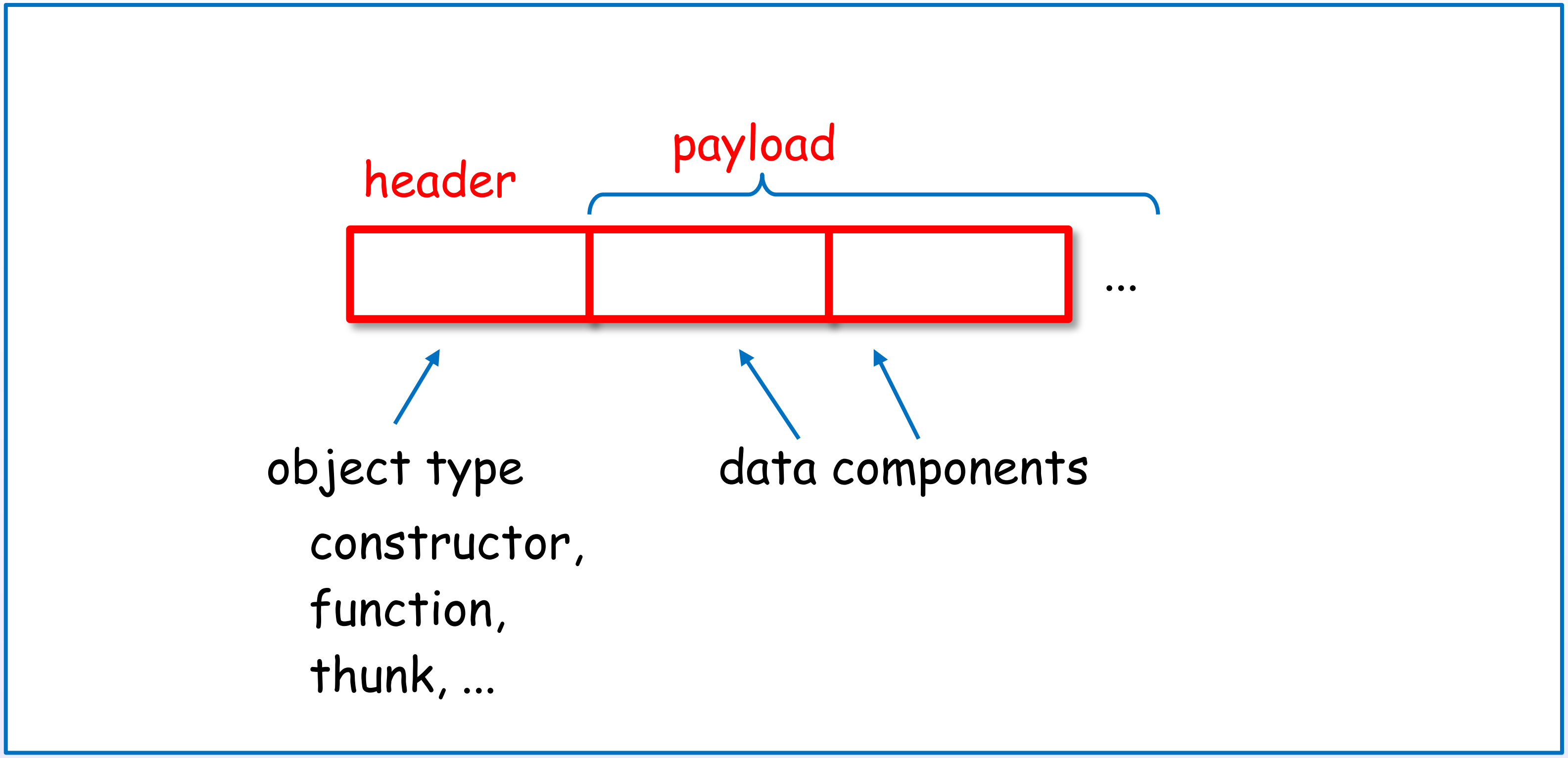


References : [D7], [D2], [H11], [H10], [H5], [H6], [H7], [B5] Ch.2, [D5], [W1], [D15]

3. Internal representation of expressions

Uniform representation

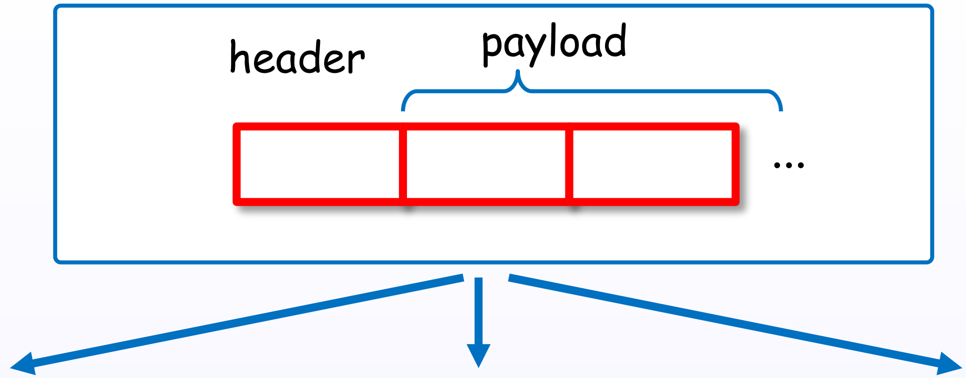
Every object is represented uniformly in memory



in heap memory, stack or static memory

References : [H11], [H10], [H5], [H6], [H7], [D15]

Every object is represented uniformly



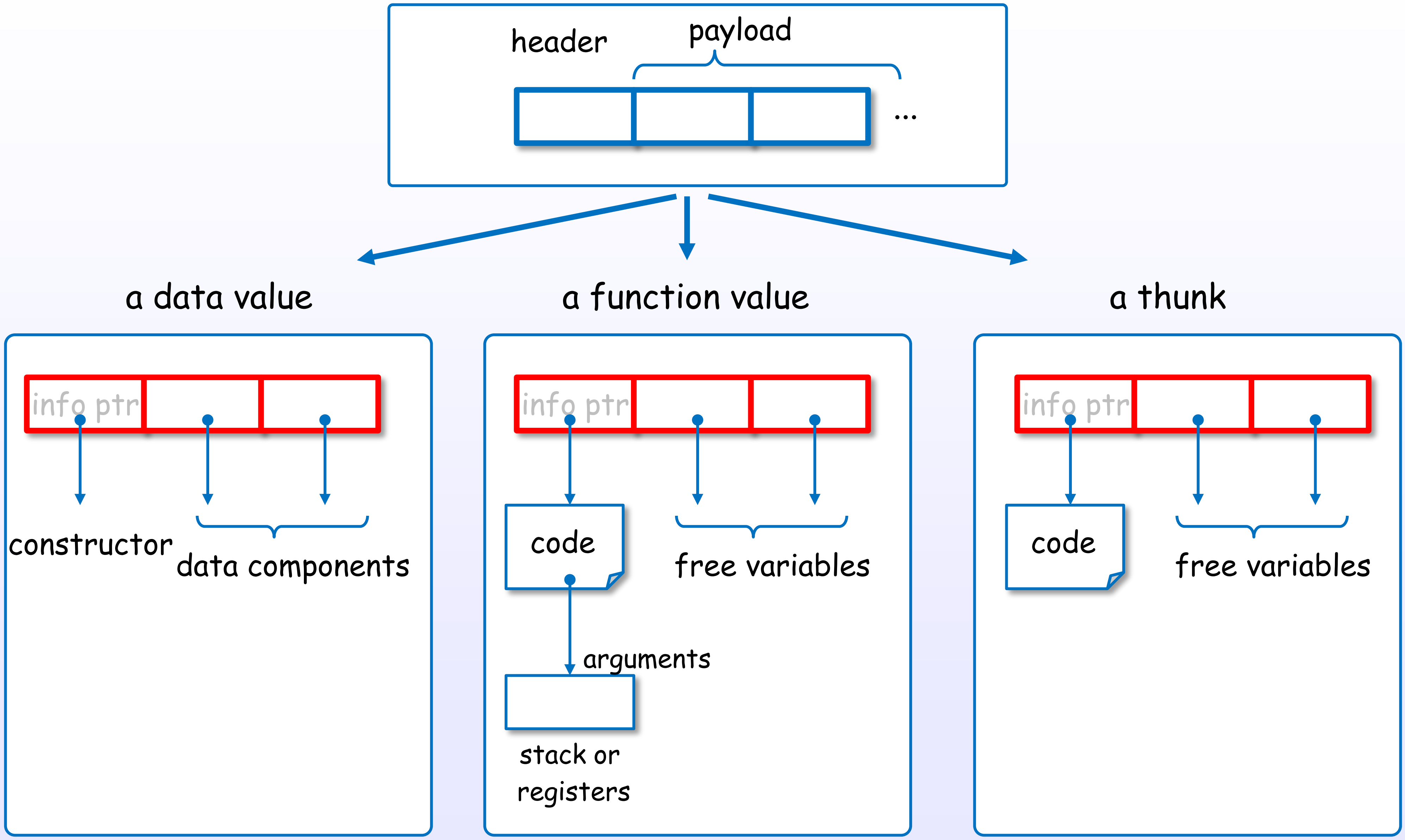
a data value

a function value

a thunk

References : [H11], [H10], [H5], [H6], [H7], [D15]

Every object is represented uniformly



References : [H11], [H10], [H5], [H6], [H7], [D15]

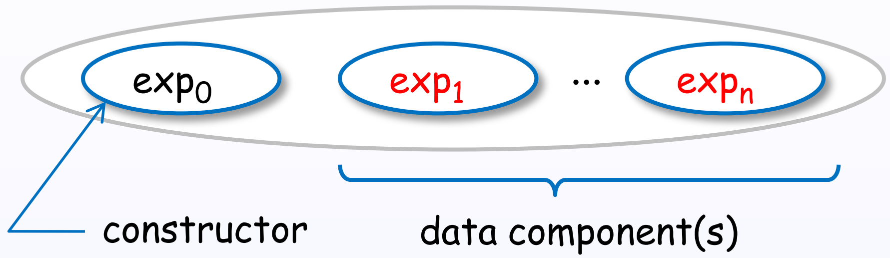
WHNF

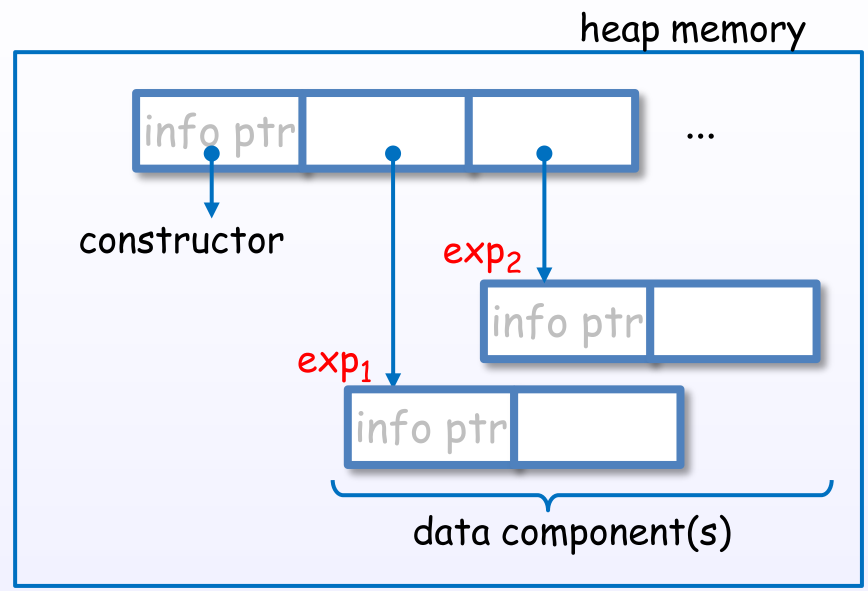
Internal representation of WHNF

Haskell code

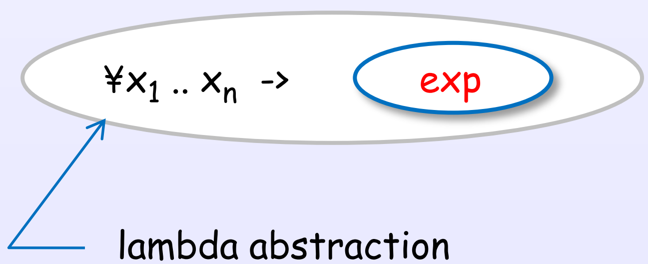
GHC's internal representation

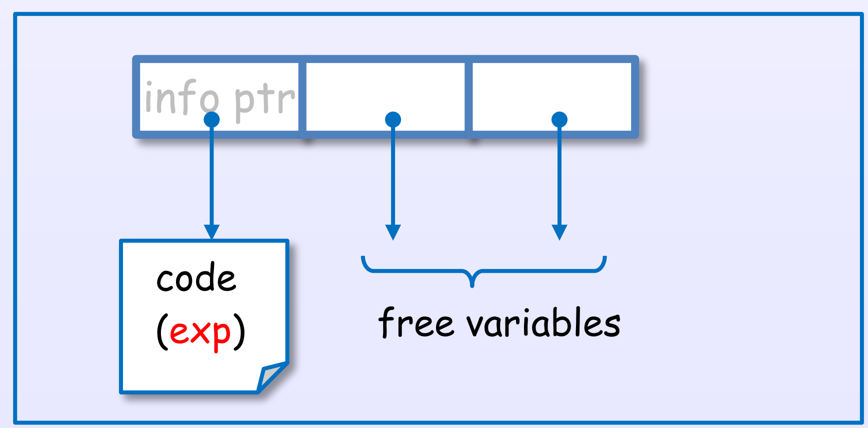
a data value in WHNF





a function value in WHNF



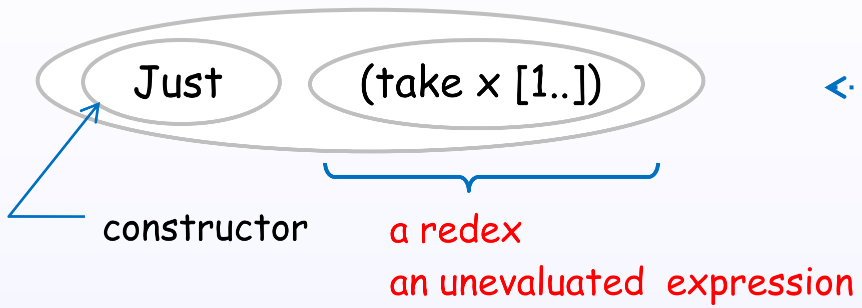


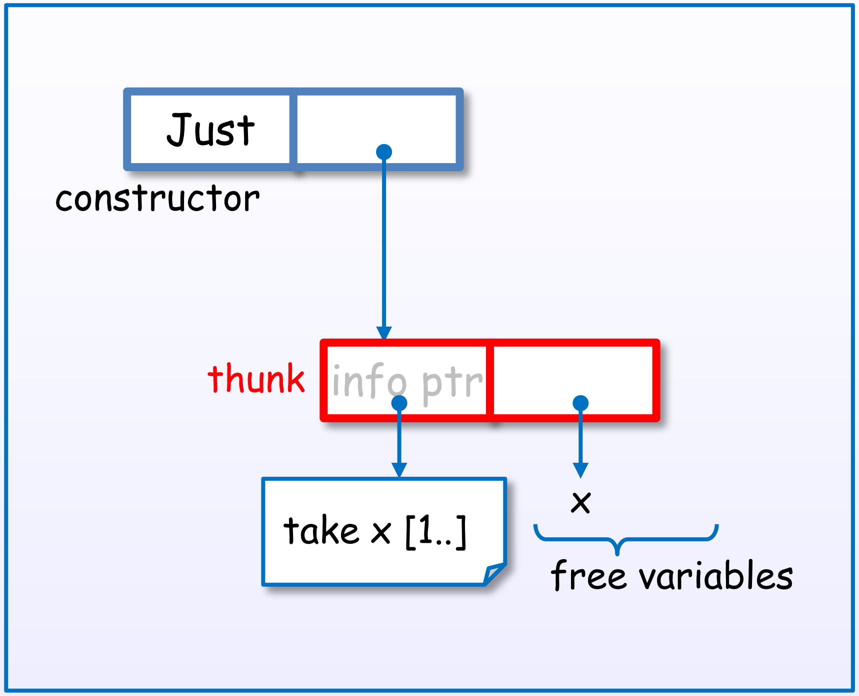
References : [H11], [H5], [H6], [H7], [H10]

Example of WHNF for a data value

Haskell code

GHC's internal representation



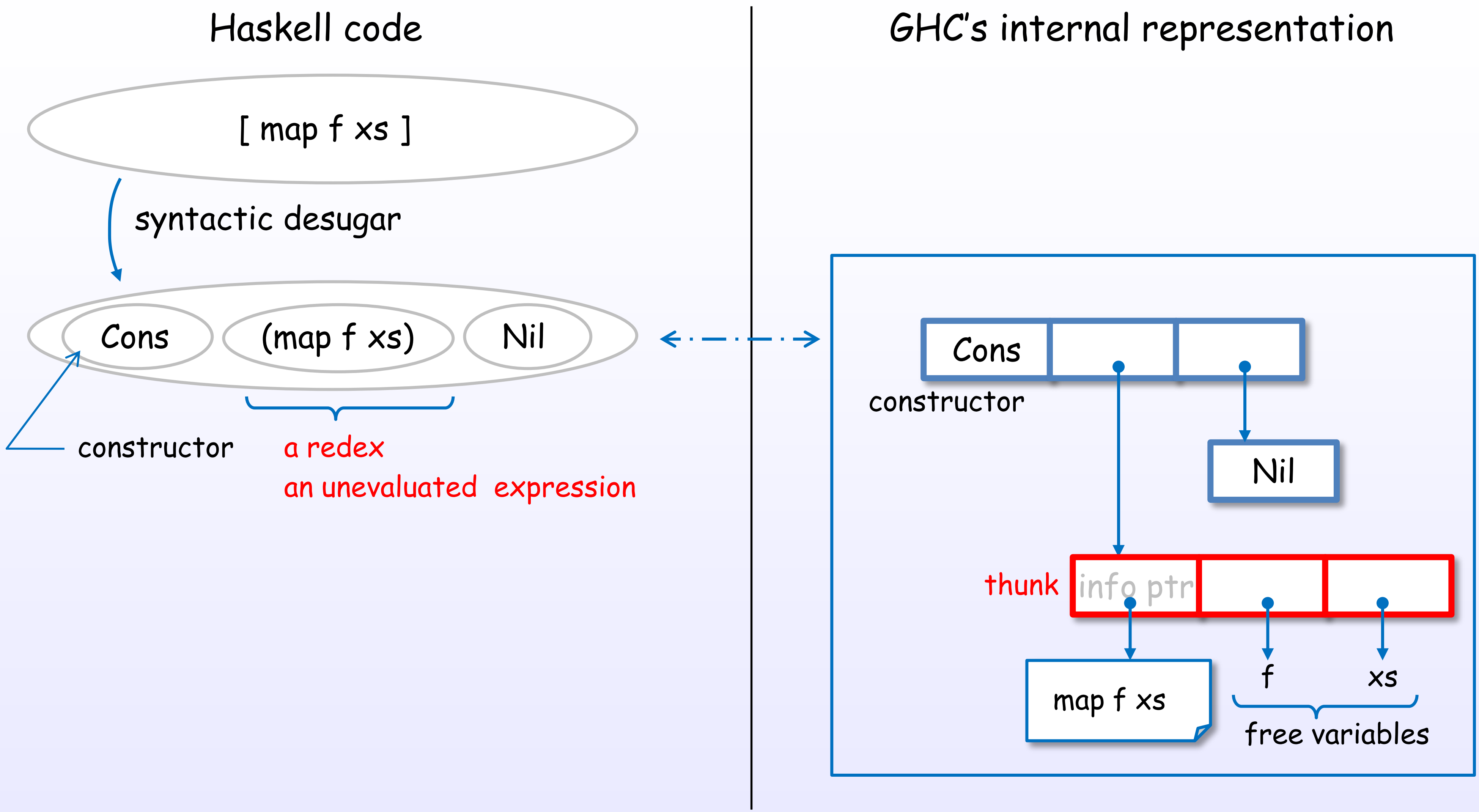


Constructors can contain unevaluated expressions by thunks.

Haskell's constructors are lazy constructors.

References : [H11], [H5], [H6], [H7], [H10]

Example of WHNF for a data value



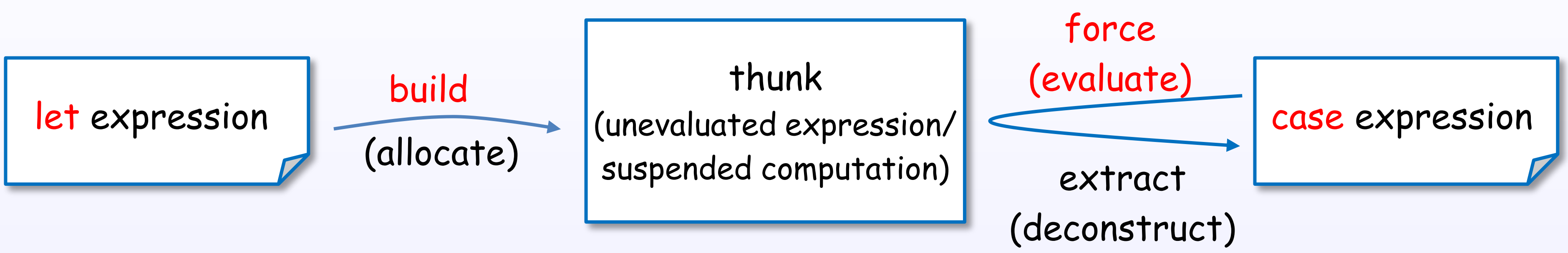
References : [H11], [H5], [H6], [H7], [H10]

let, case expression

let, case expression

let and case expressions are special role in the evaluation

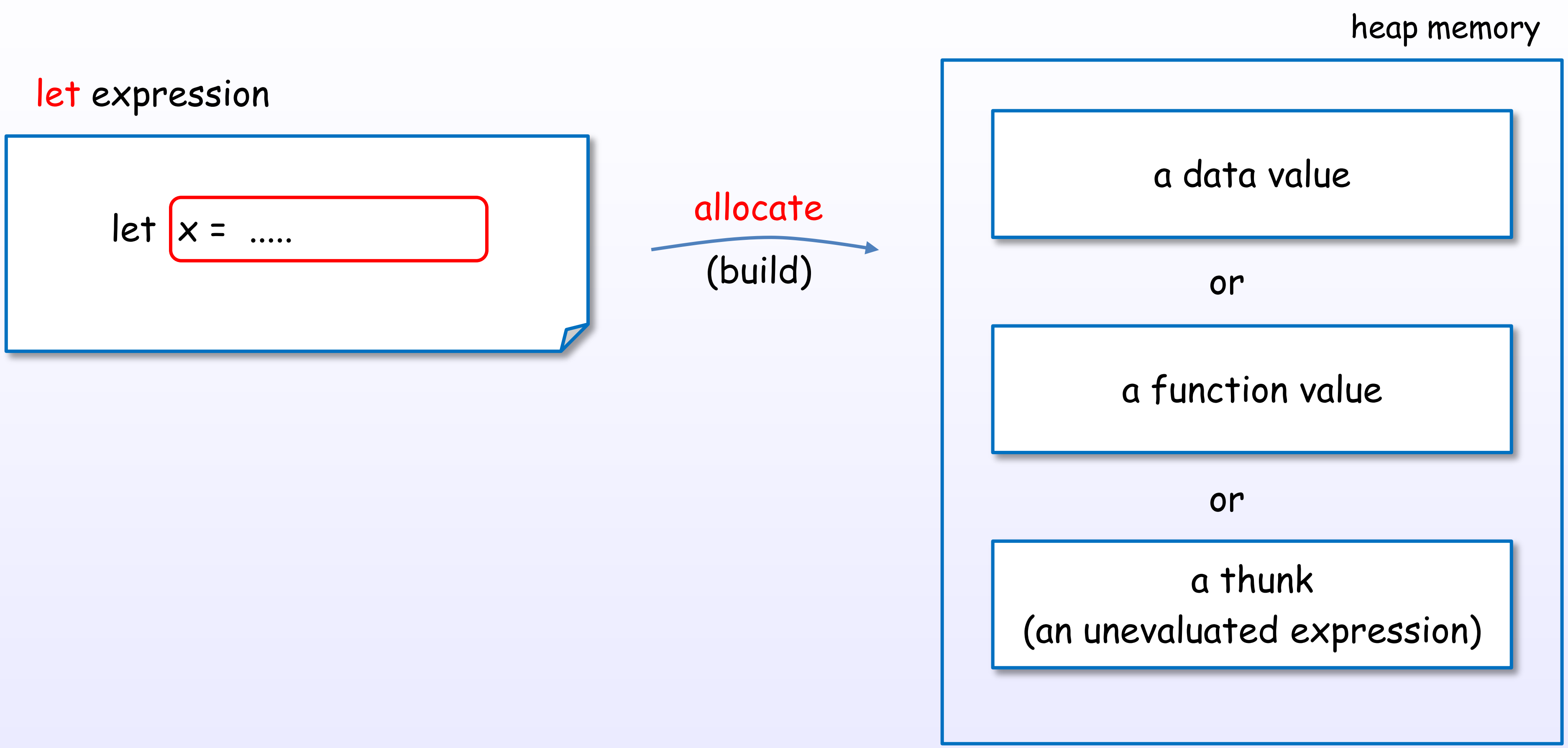
let/case expressions and thunk



A let expression may build a thunk.
A case expression evaluates (forces) and deconstructs the thunk.

References : [H5], [H6], [H7], [H10]

A let expression may allocates a heap object



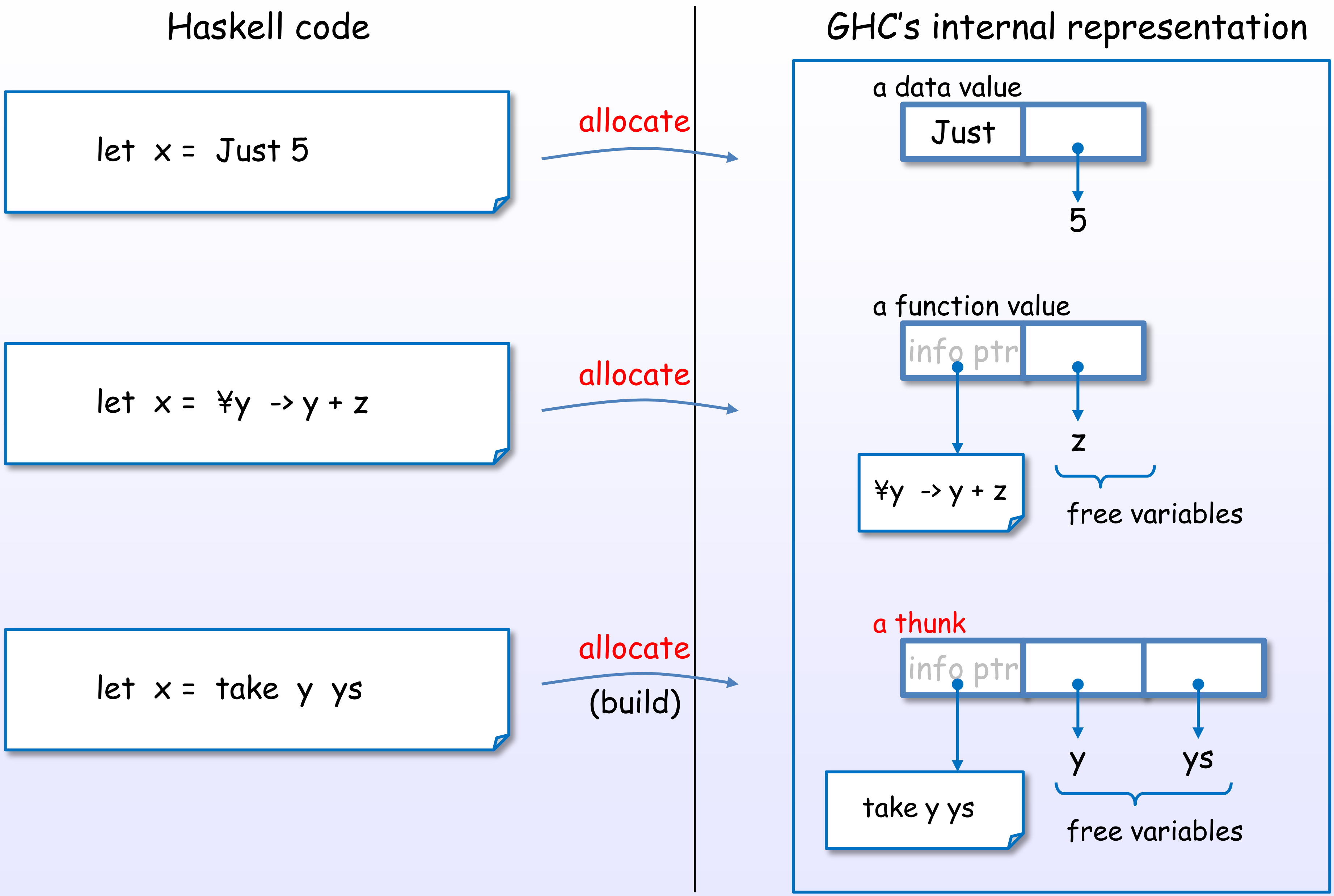
A let expression may allocates an object in the heap.

(If GHC can optimize it, the let expression may not allocate.)

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

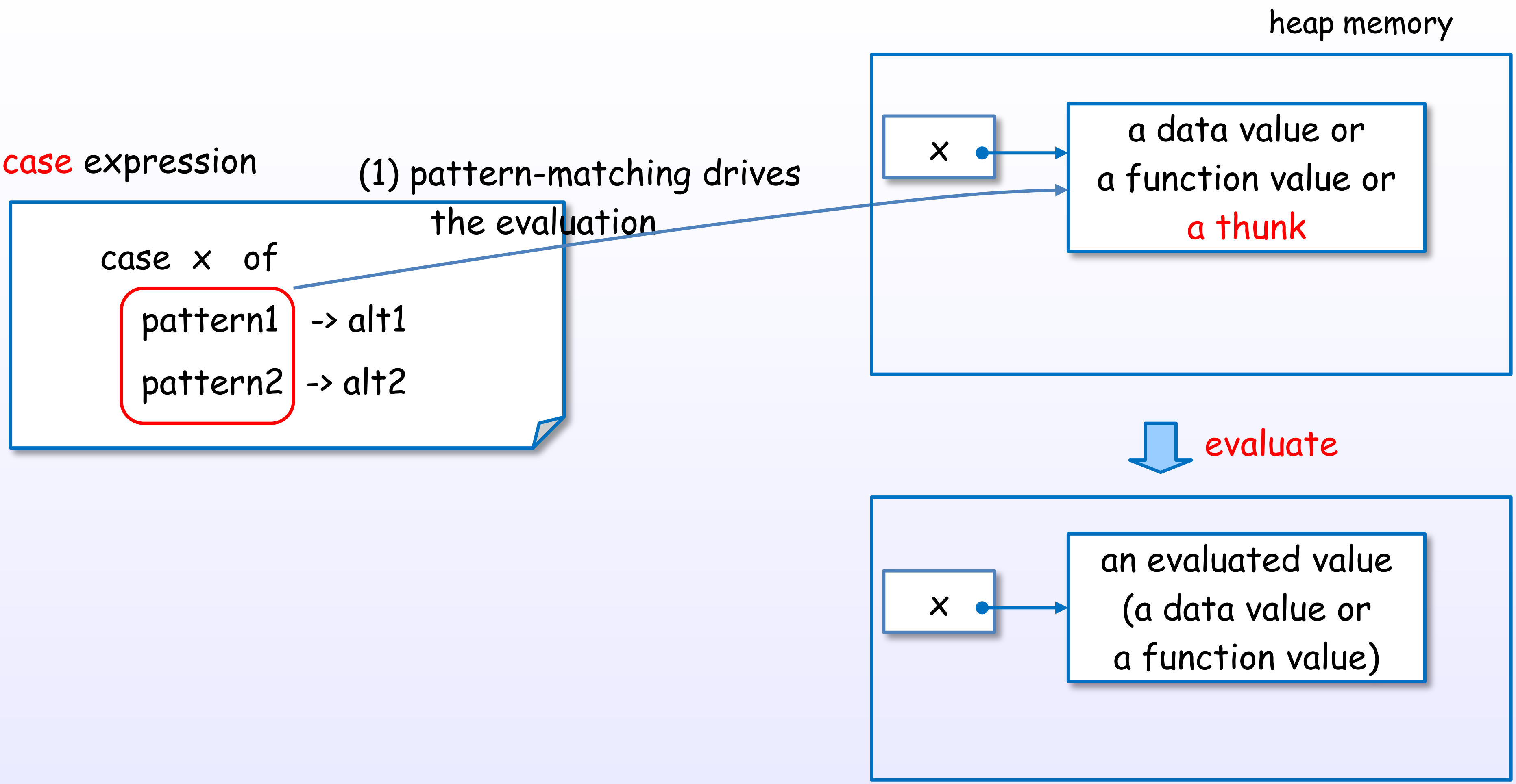
References : [H5], [H6], [H7], [H10]

Example of let expressions



References : [H5], [H6], [H7], [H10]

A case expression evaluates a subexpression

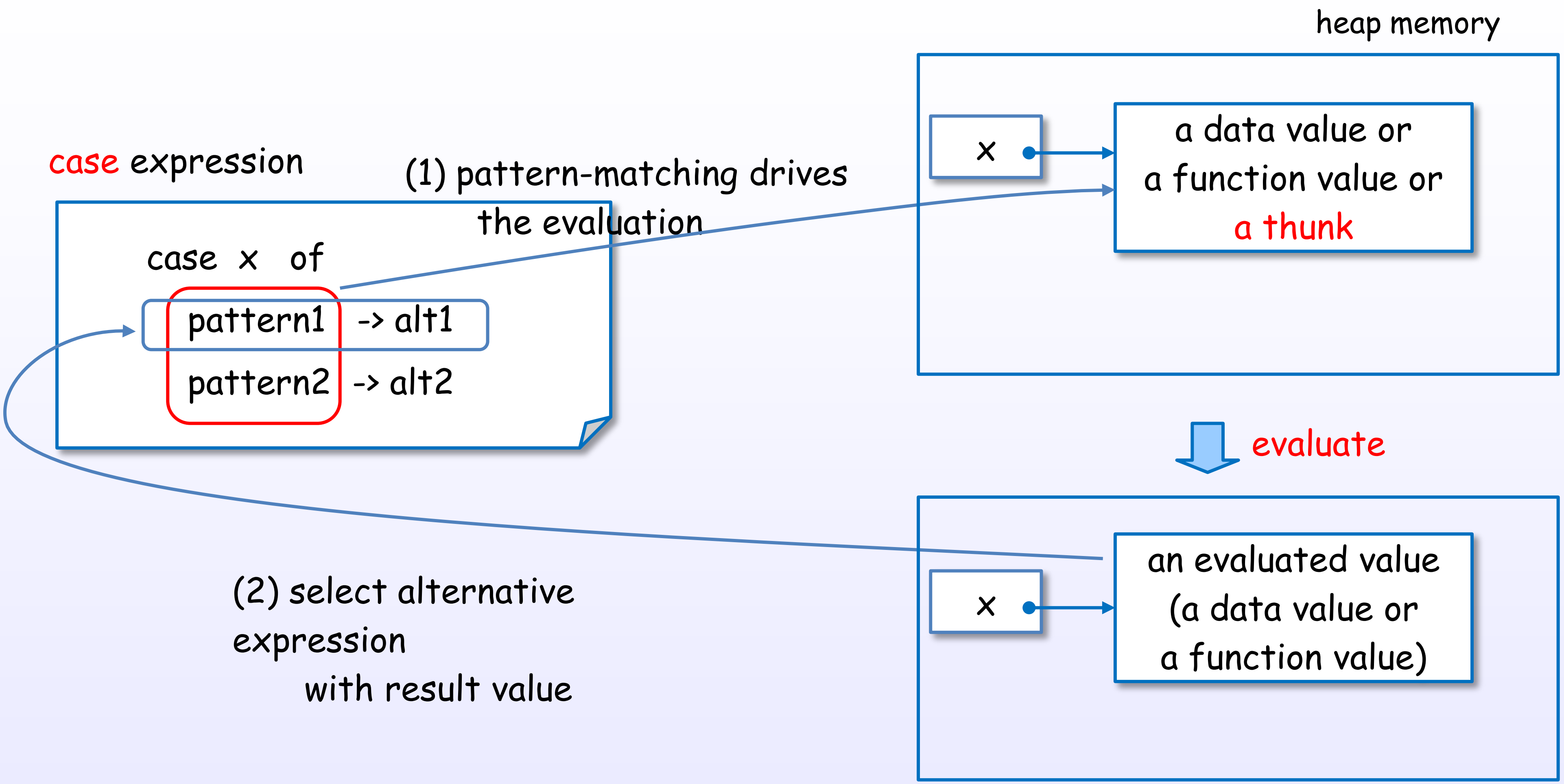


Pattern-matching drives the evaluation

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

References : [H5], [H6], [H7], [H10], [D2]

A case expression also perform case analysis

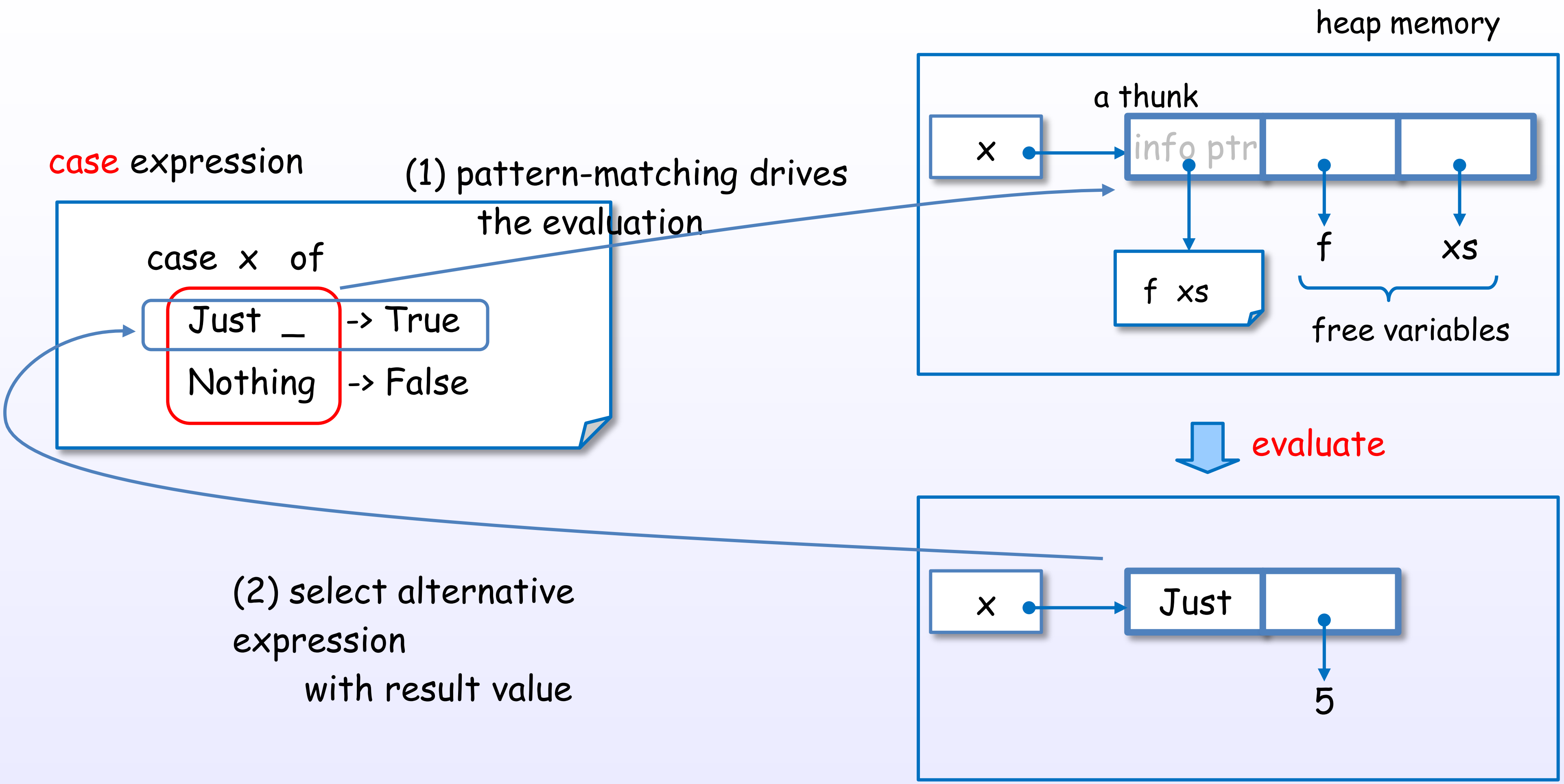


A case expression evaluates a subexpression
and optionally performs case analysis on its value.

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

References : [H5], [H6], [H7], [H10], [D2]

Example of a case expression



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

References : [H5], [H6], [H7], [H10], [D2]

Pattern-matching in function definition

pattern-matching in function definition

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

syntactic desugar

pattern-matching in case expression

```
f x = case  x  of
        Just _    -> True
        Nothing   -> False
```

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

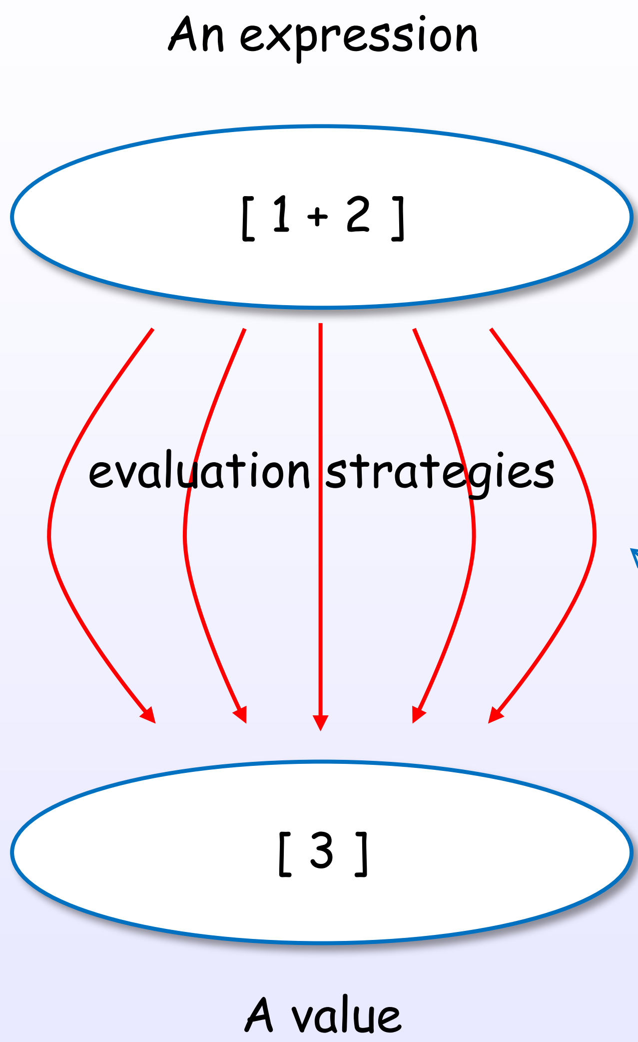
A function's pattern-matching also drives the evaluation.

References : [H1]

4. Evaluation

Evaluation strategies

There are many evaluation approaches



- Strict, Non-strict evaluation
- Eager, Lazy evaluation
- Call-by-value, Call-by-name, Call-by-need, …
- Innermost, Outermost
- Normal order, Application order
- …

References : [B3] Ch.2, 7, [B6] Ch.3, [D1]

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 (Evaluation strategies/ Reduction strategies)	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		…

References : [D3], [D1], [D2], [D5], [D4], [B2] Ch.7, [B3] Ch.8, [B6] Ch.5, [W1], [W2], [W3], [B7], [B8]

Evaluation layer for GHC's Haskell

Denotational semantics

Strict semantics

Non-strict semantics

Haskell 2010 specification

Operational semantics
(Evaluation strategies/ Reduction strategies)

Strict evaluation

Non-strict evaluation

GHC's strategy

Eager evaluation

Nondeterministic evaluation

Lazy evaluation

GHC's strategy

...

Call-by-Value

Call-by-Name

Call-by-Need

GHC's strategy

Applicative order reduction

Normal order reduction

GHC's strategy

...

Rightmost reduction

Innermost reduction

Leftmost reduction

Outermost reduction

GHC's strategy

Implementation techniques

Tree reduction

Lazy graph reduction

...

GHC's implementation

References : [D3], [D1], [D2], [D5], [D4], [B2] Ch.7, [B3] Ch.8, [B6] Ch.5, [W1], [W2], [W3], [B7], [B8]

Evaluation strategies

Each evaluation strategy decides how to operate the evaluation about

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

References : [D3], [D1], [D2], [D5], [D4], [B2] Ch.7, [B3] Ch.8, [B6] Ch.5, [W1], [W2], [W3], [B7], [B8]

One of the important points is the order

which first?

function arguments

apply first

- lazy evaluation,
- call-by-name,
- call-by-need,
- outermost reduction,
- normal order reduction

argument first

- eager evaluation,
- call-by-value,
- innermost reduction,
- applicative order reduction

References : [D3], [D1], [D2], [D5], [D4], [B2] Ch.7, [B3] Ch.8, [B6] Ch.5, [W1], [W2], [W3], [B7], [B8]

Simple example of typical evaluations

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

call-by-value

square (1 + 2)

argument evaluation first

default
Haskell (GHC), ...

call-by-need

square (1 + 2)

apply first

References : [B2] Ch.7, [B3] Ch.8, [D4], [B6] Ch.5

Simple example of typical evaluations

<u>call-by-value</u>

square (1 + 2)

square (3)

evaluation is **performed**

3 * 3

9

<u>call-by-need</u>

square (1 + 2)

(1 + 2) * (1 + 2)

evaluation is **delayed !**

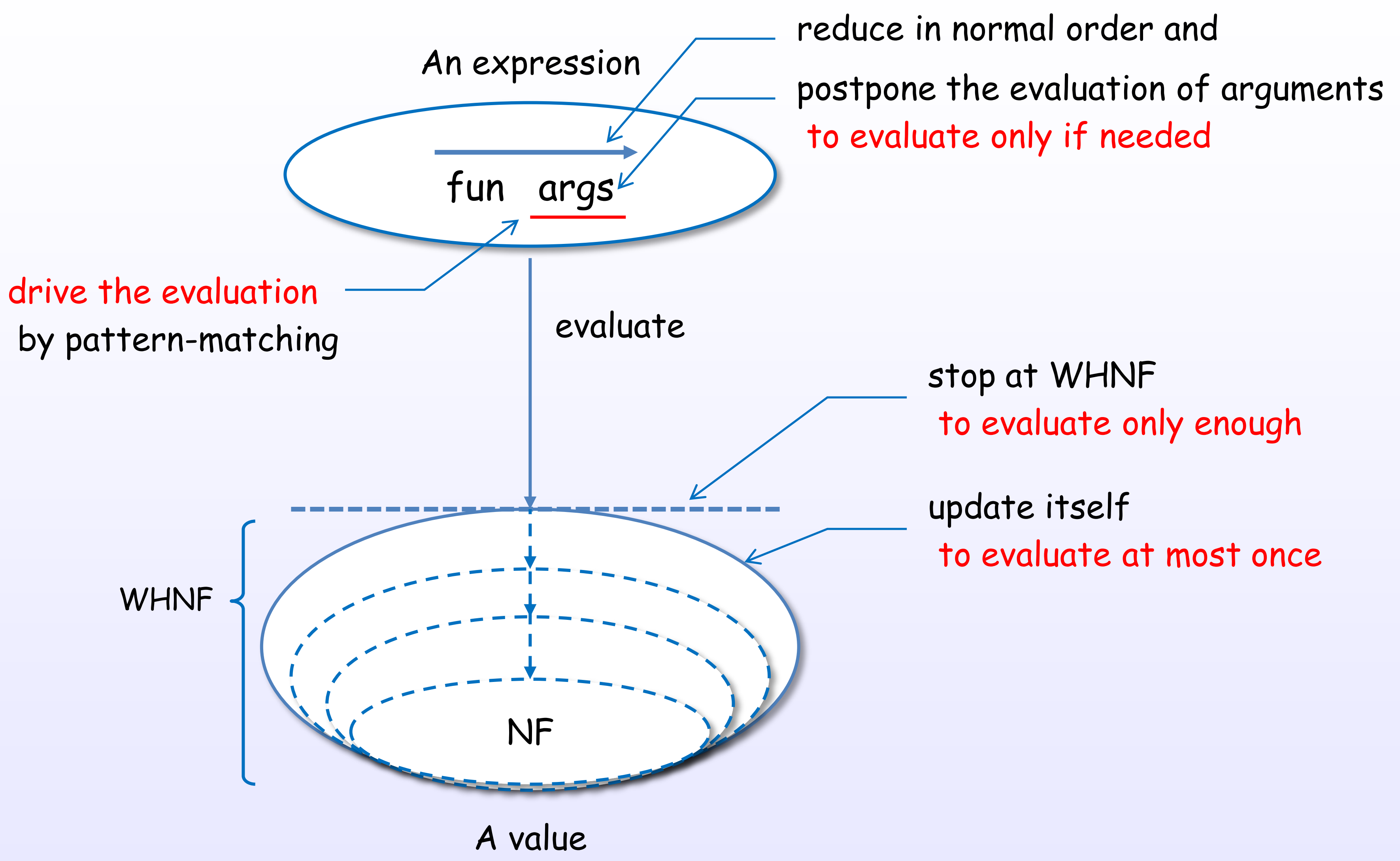
(3) * (3)

9

References : [B2] Ch.7, [B3] Ch.8, [D4], [B6] Ch.5

Evaluation in Haskell (GHC)

Key concepts of Haskell's lazy evaluation



References : [H4] Ch.11, 12, [H5], [H6], [D2]

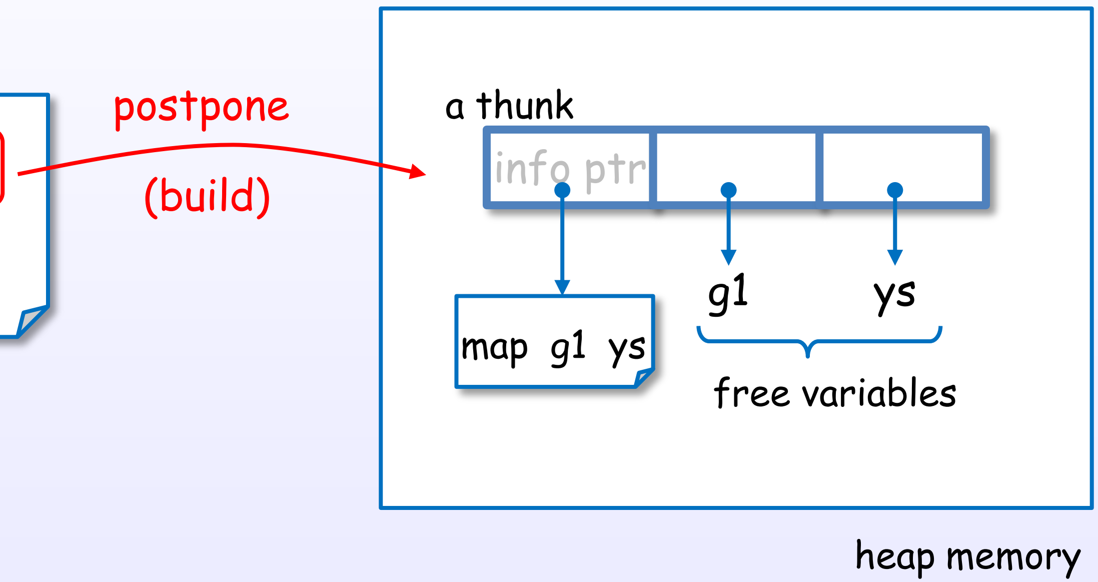
Postpone the evaluation of arguments

Haskell code

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

internal translation

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

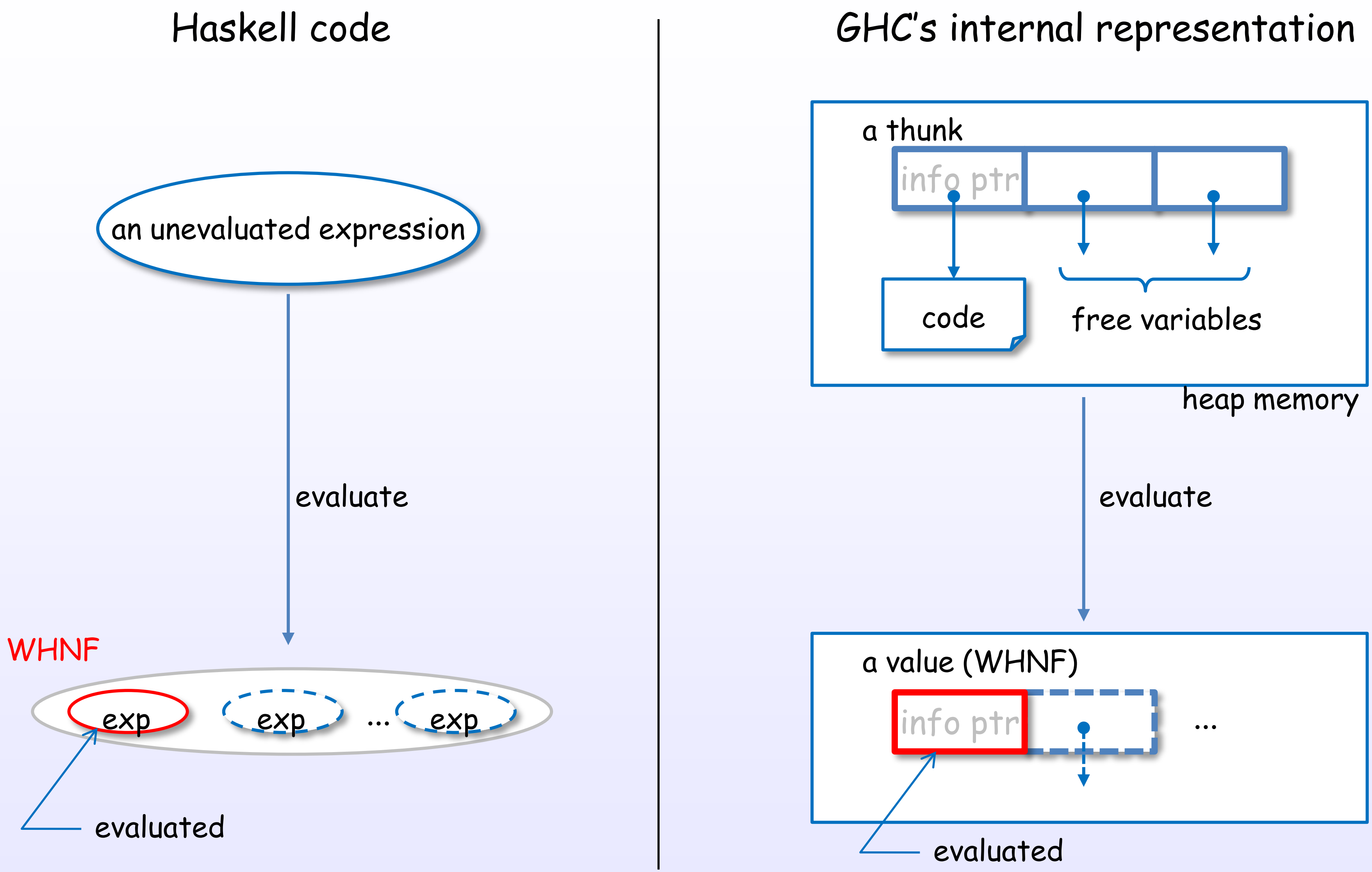


postpone the evaluation by a thunk which build with let expression

(When GHC can optimize it by analysis, the thunk may not be build.)

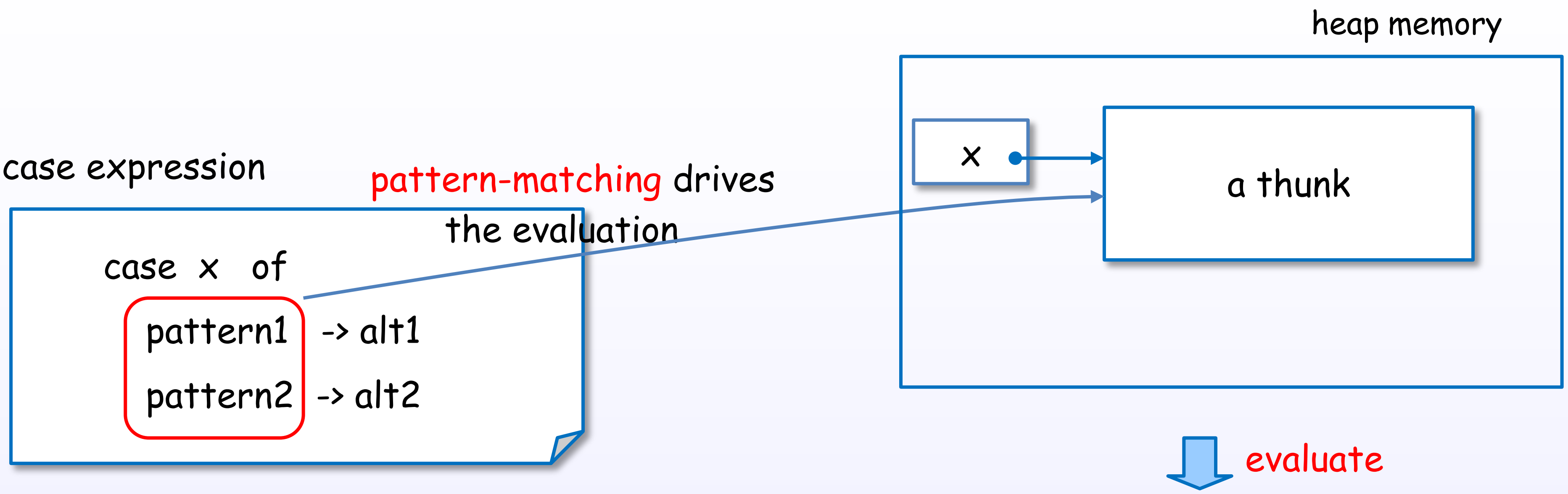
References : [H5], [H6], [H10]

Stop at WHNF



References : [H5], [H6], [H10]

Pattern-matching drives the evaluation



References : [H5], [H6], [H7], [H10], [D2]

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?

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

argument

function

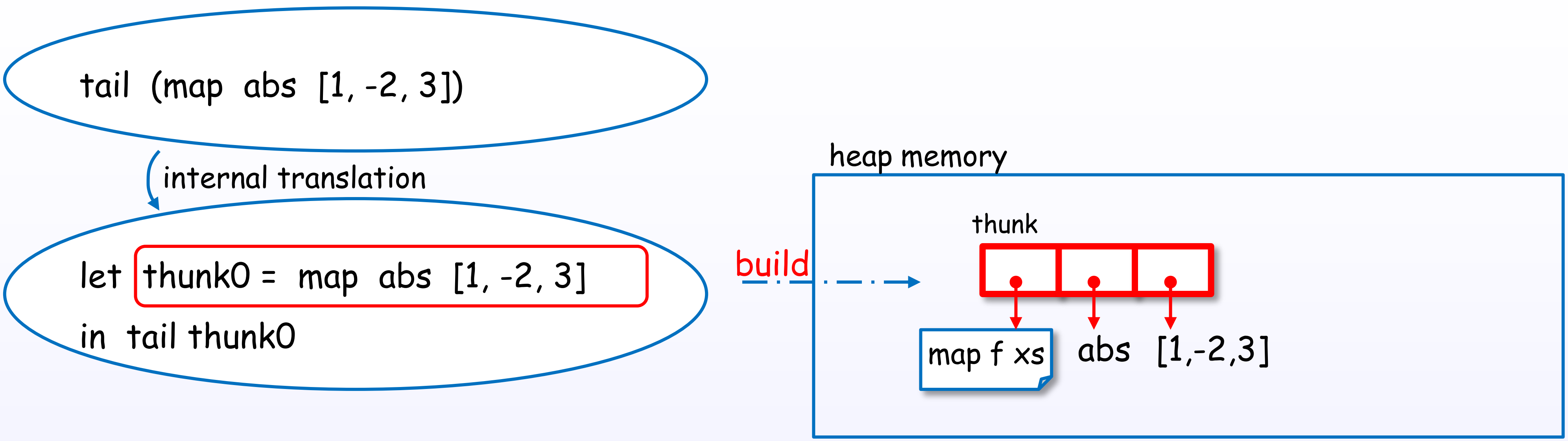
(3) GHC internally translates the expression

```
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



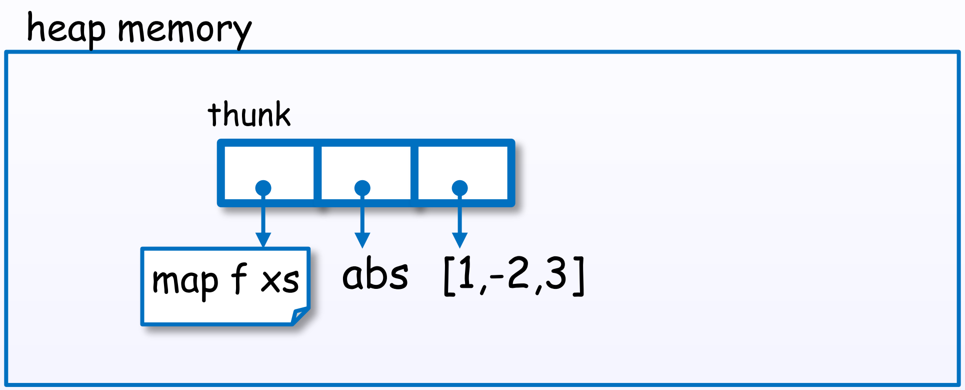
(5) function apply to argument

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

internal translation

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

apply



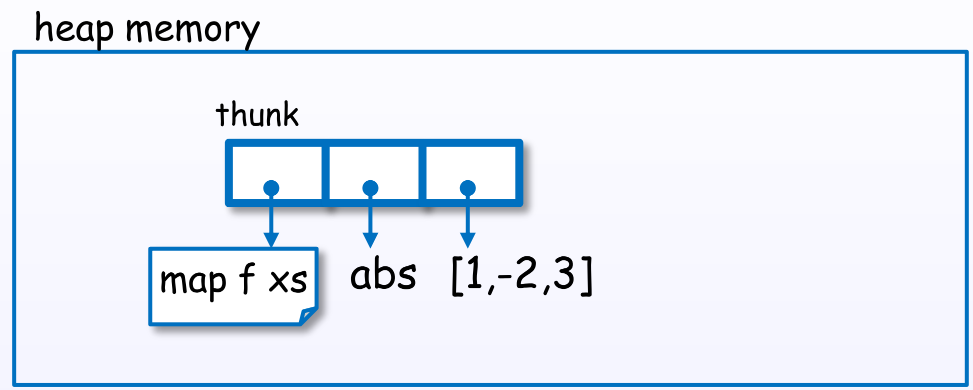
(6) tail is defined here

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

internal translation

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

```
tail (_:xs) = xs
```



(7) function is syntactic sugar

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

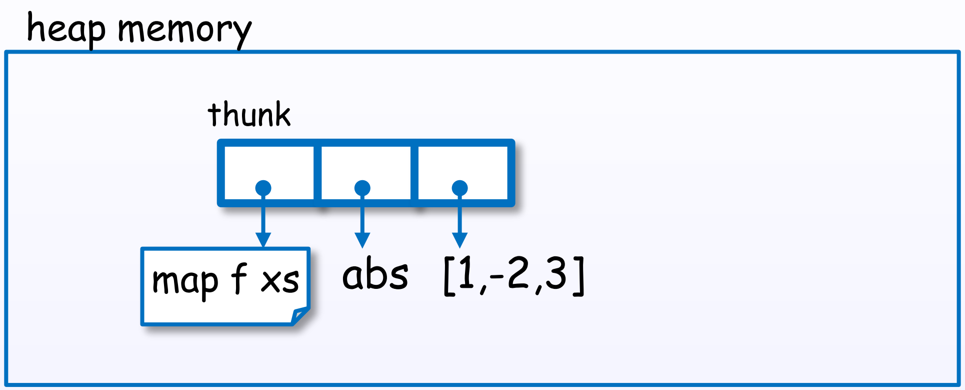
internal translation

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

```
tail (_:xs) = xs
```

syntactic desugar

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



(8) substitute function body (beta reduction)

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

internal translation

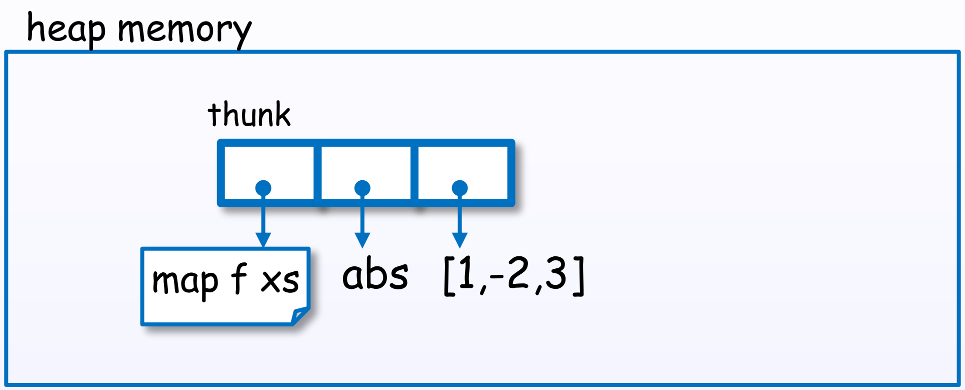
```
let  thunk0 =  map  abs  [1, -2, 3]
in  tail thunk0
```

reduction

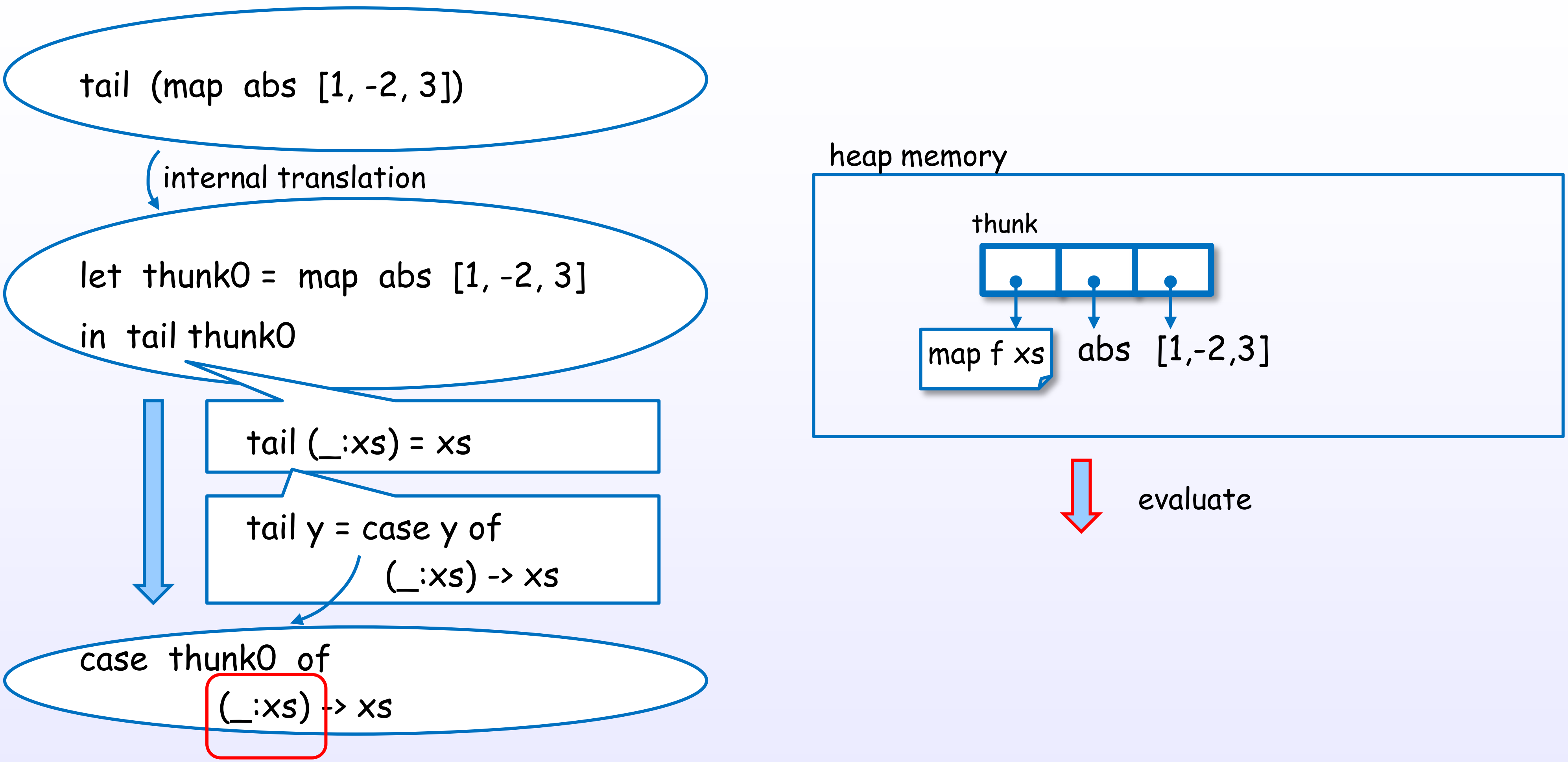
```
tail (_:xs) = xs
```

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

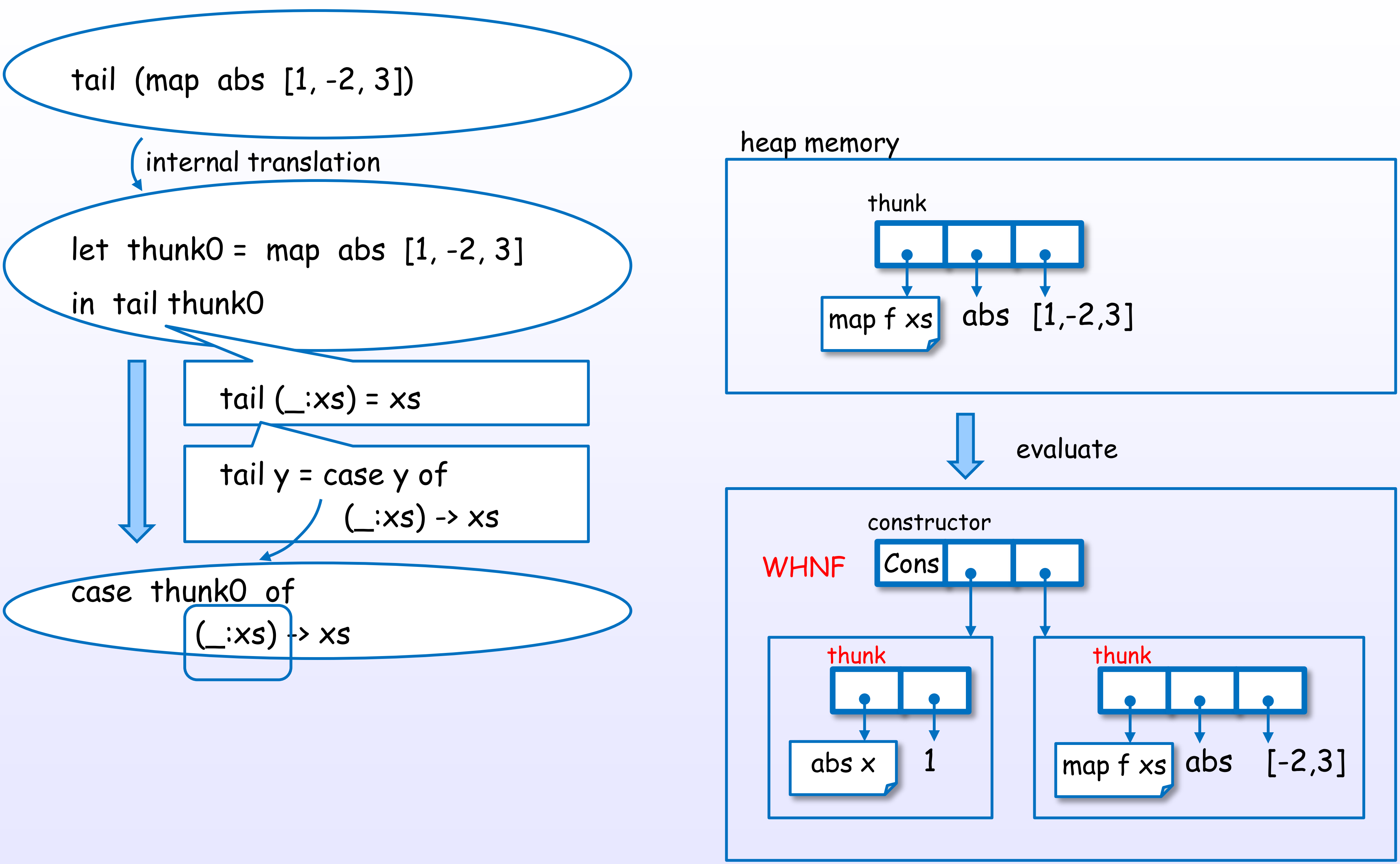
```
case  thunk0  of
         (_:xs) -> xs
```



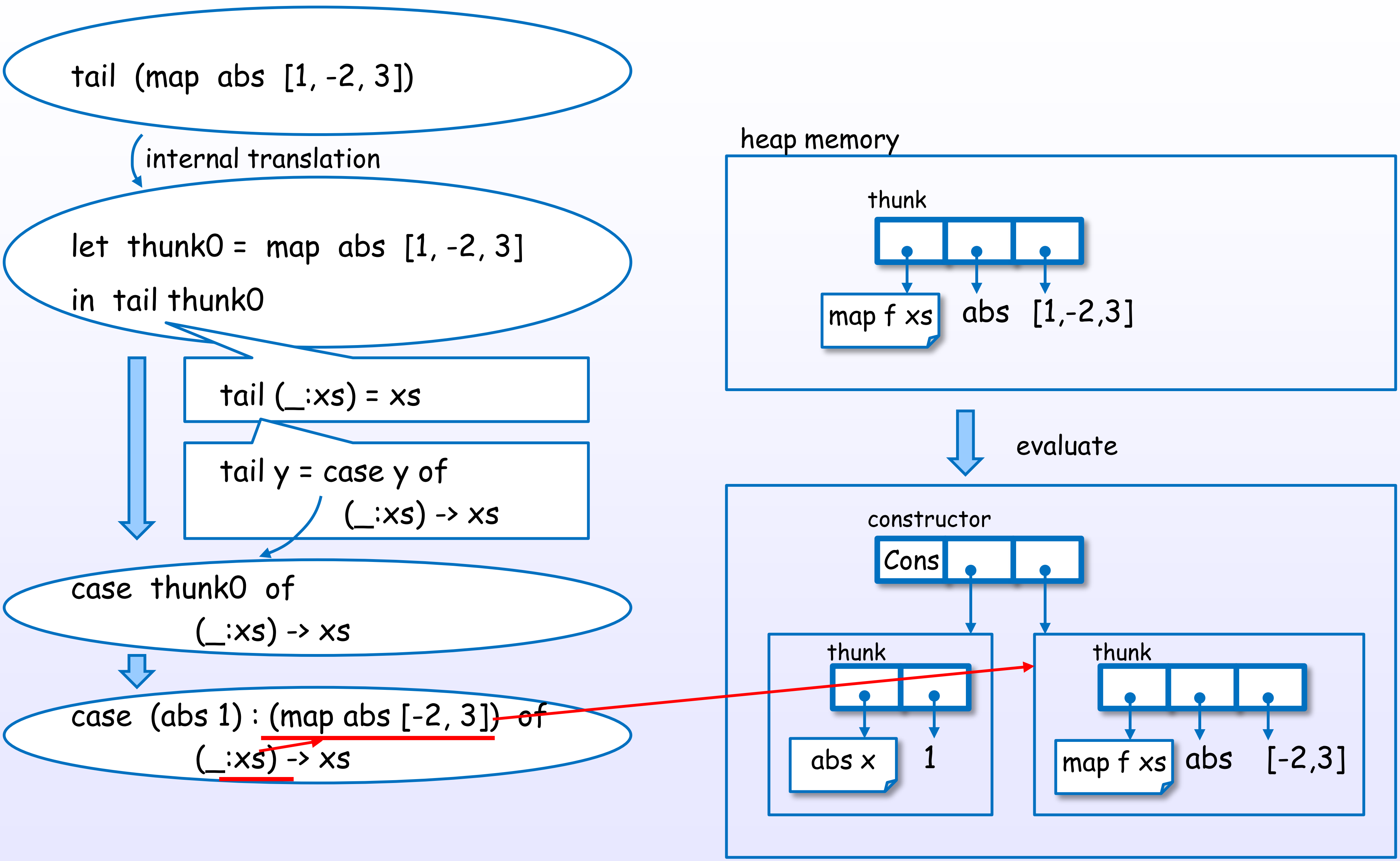
(9) case pattern-matching drives the evaluation



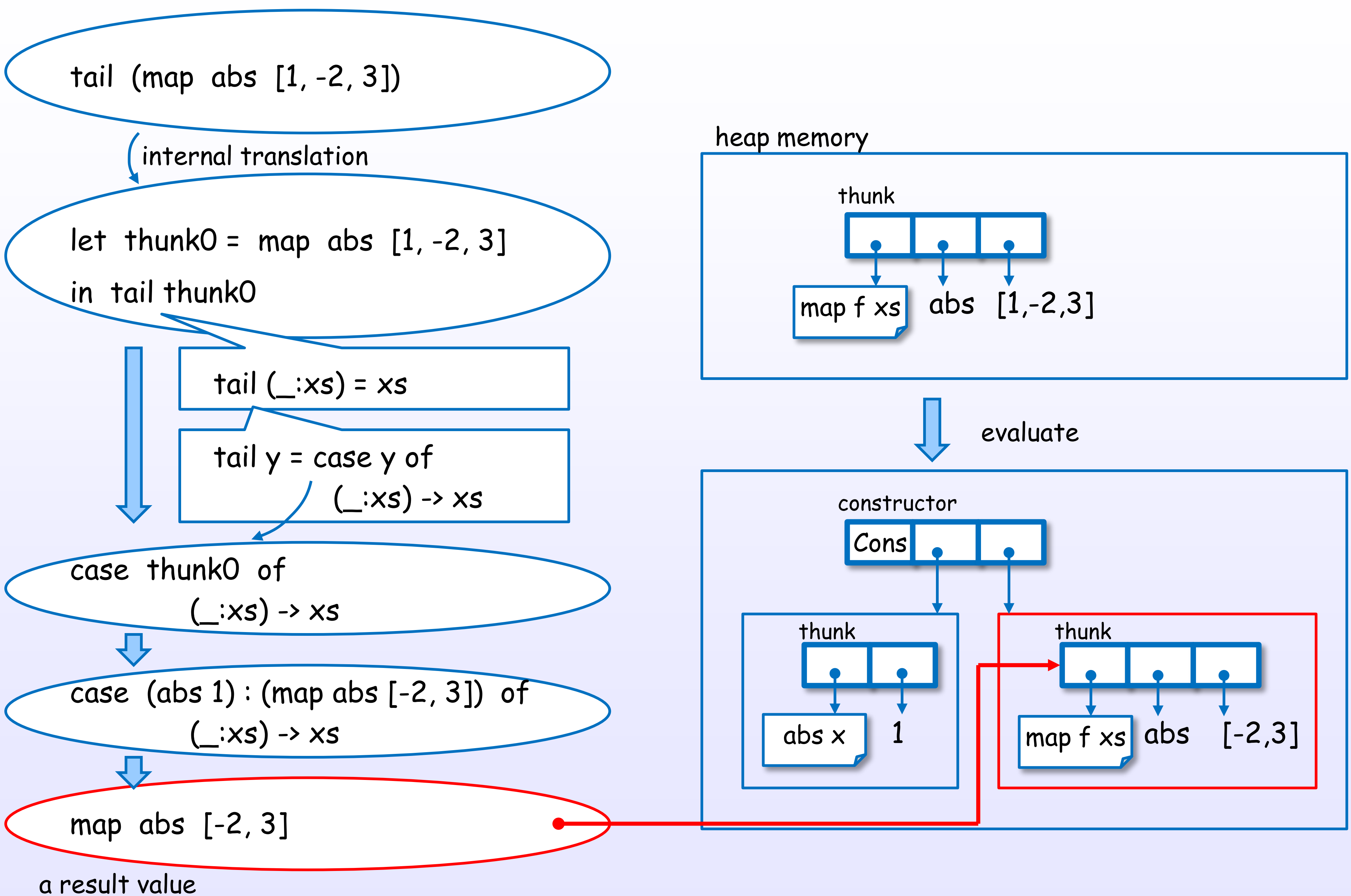
(10) but, stop at WHNF



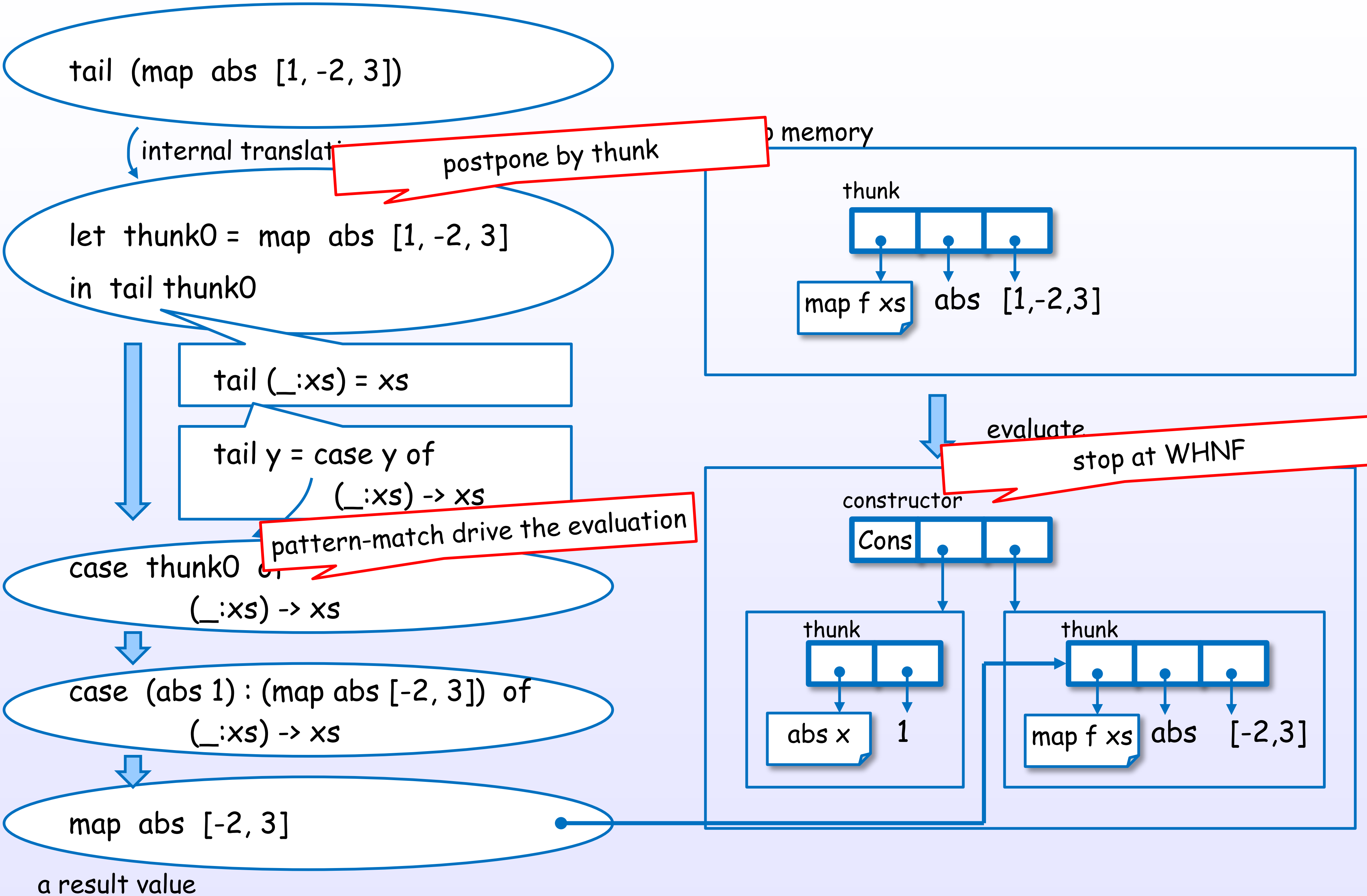
(11) bind variables to result



(12) return the value



Key points


tail (map abs [1, -2, 3])
internal translat
postpone by thunk
memory
let thunk0 = map abs [1, -2, 3]
in tail thunk0
thunk
map f xs
abs
[1,-2,3]
tail (_:xs) = xs
tail y = case y of
(_:xs) -> xs
evaluate
stop at WHNF
constructor
Cons
pattern-match drive the evaluation
case thunk0 of
(_:xs) -> xs
thunk
abs x
1
thunk
map f xs
abs
[-2,3]
case (abs 1) : (map abs [-2, 3]) of
(_:xs) -> xs
map abs [-2, 3]
a result value

Examples of evaluations

Example of repeat

```
repeat 1
```

```
1 : repeat 1
```

```
1 : 1 : repeat 1
```

```
1 : 1 : 1 : repeat 1
```

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

Example of repeat

repeat 1

⇓

1 : repeat 1

Cons 1 (repeat 1)

⇓

1 : 1 : repeat 1

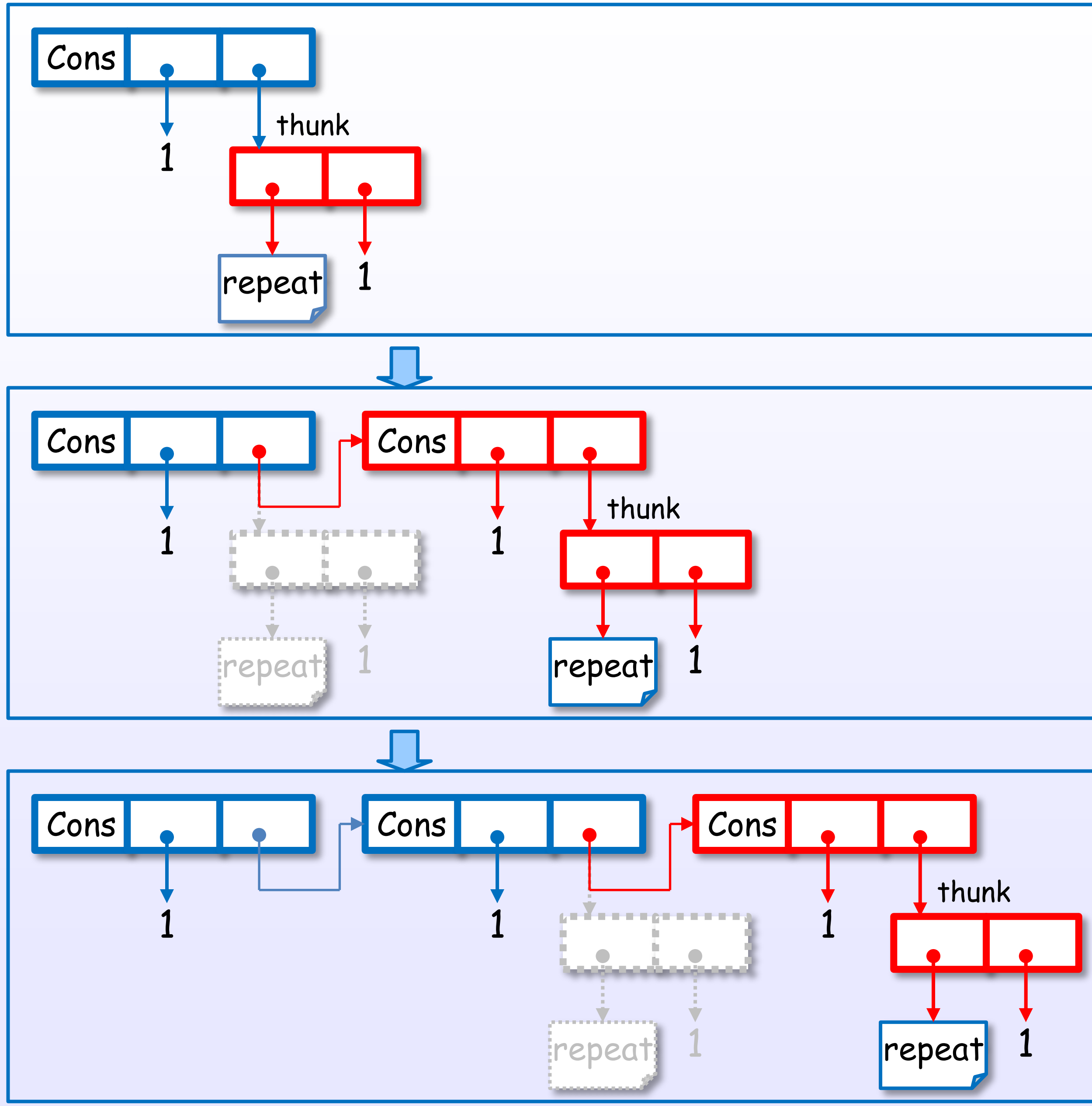
Cons 1 (Cons 1 (repeat 1))

⇓

1 : 1 : 1 : repeat 1

Cons 1 (Cons 1 (Cons 1 (repeat 1)))

⇓



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

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

⇓

...

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

Example of map

map f [1, 2, 3]

⇓

f 1 : map f [2, 3]

Cons (f 1) (map f [2, 3])

⇓

f 1 : f 2 : map f [3]

Cons (f 1) (Cons (f 2) (map f [3]))

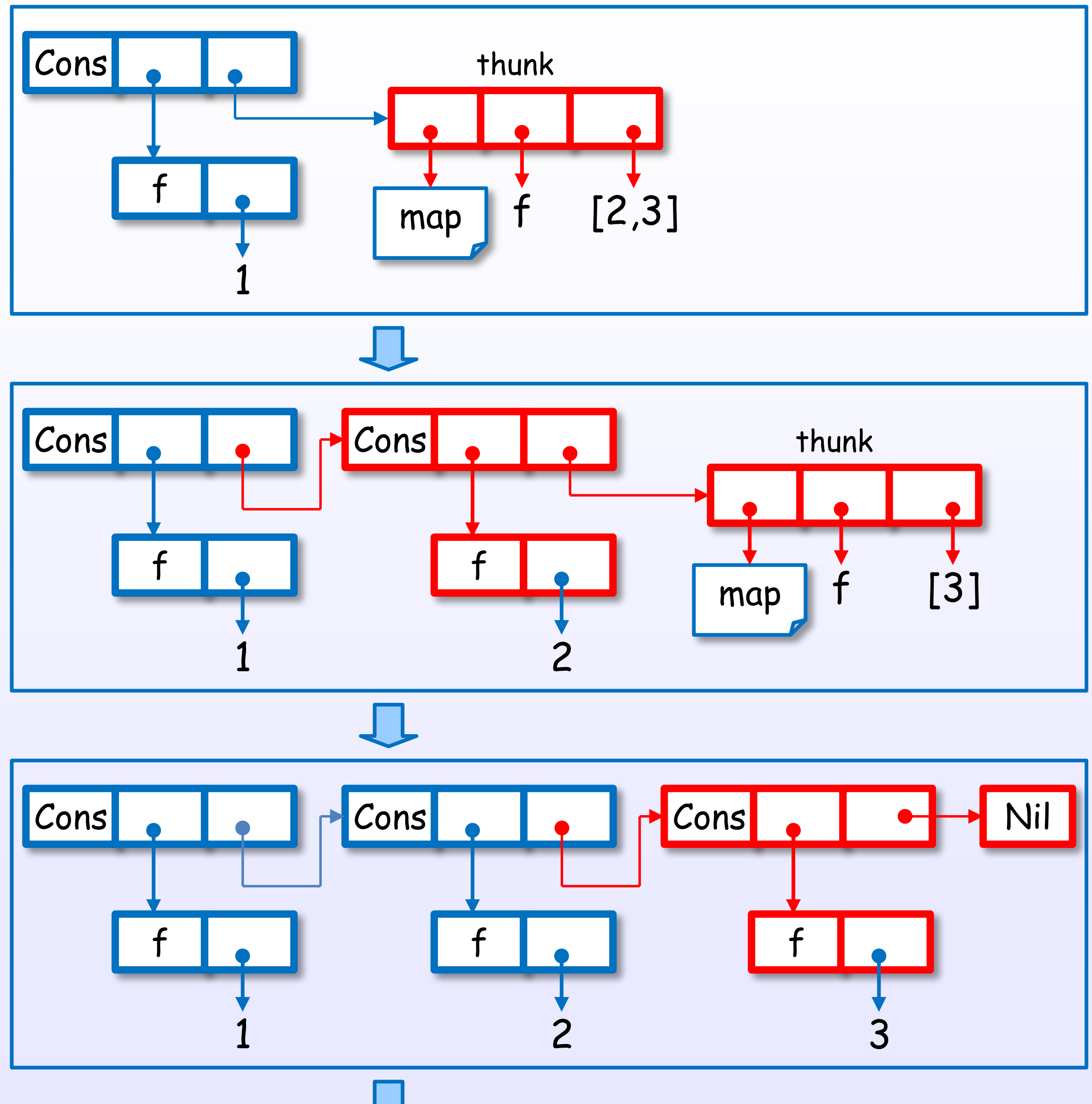
⇓

f 1 : f 2 : f 3

Cons (f 1) (Cons (f 2) (Cons (f 3) Nil))

⇓

...



...

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

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]
```

⇓

...

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

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]
```

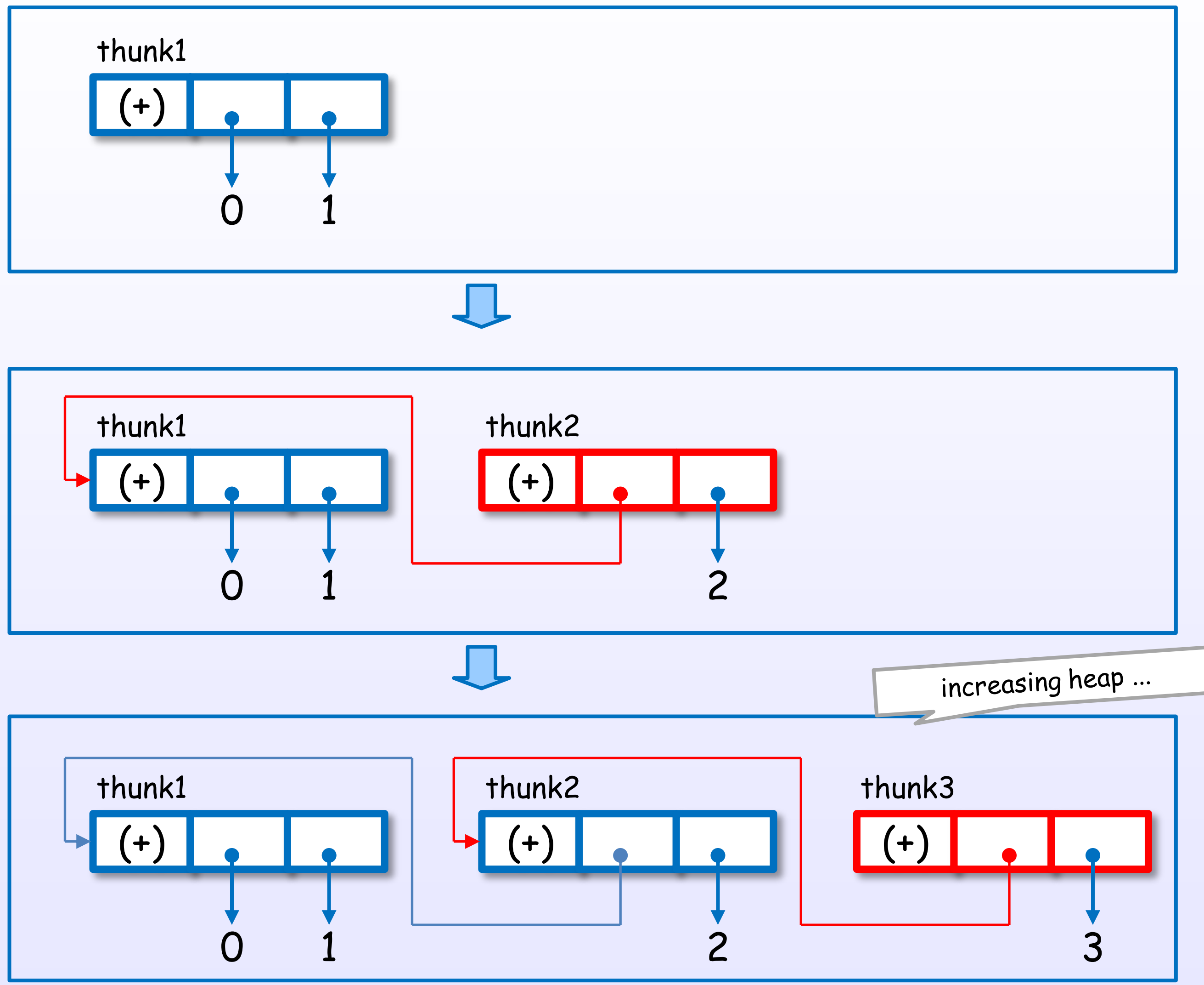
```
foldl (+) (((0 + 1) + 2) + 3) [4 .. 100]
```

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

...

heap memory

*show only accumulation value



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]

⇩

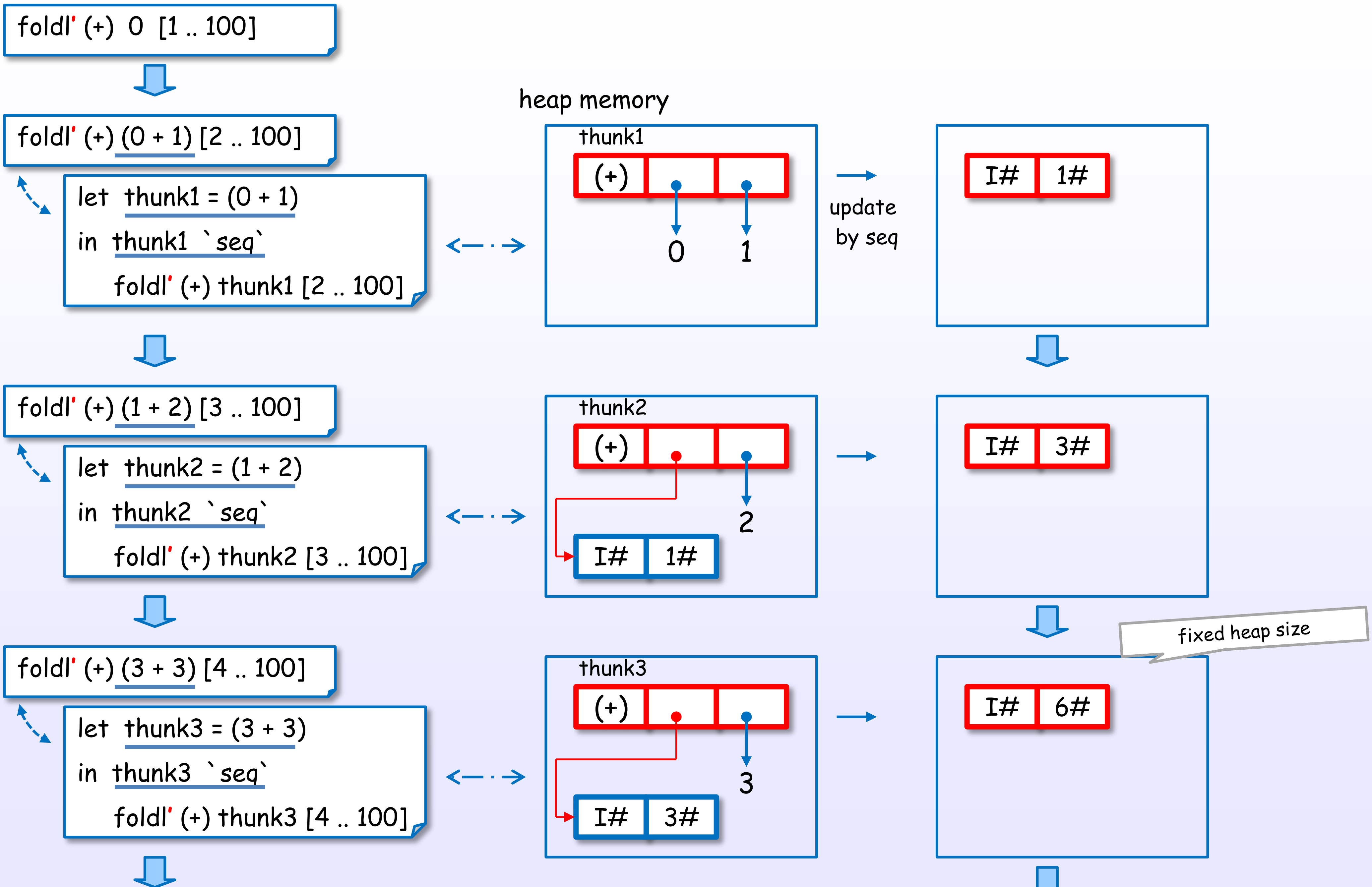
foldl' (+) (3 + 3) [4 .. 100]

⇩

...

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

Example of foldl' (strict)



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

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

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

⇩

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

⇩

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

⇩

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

⇩

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

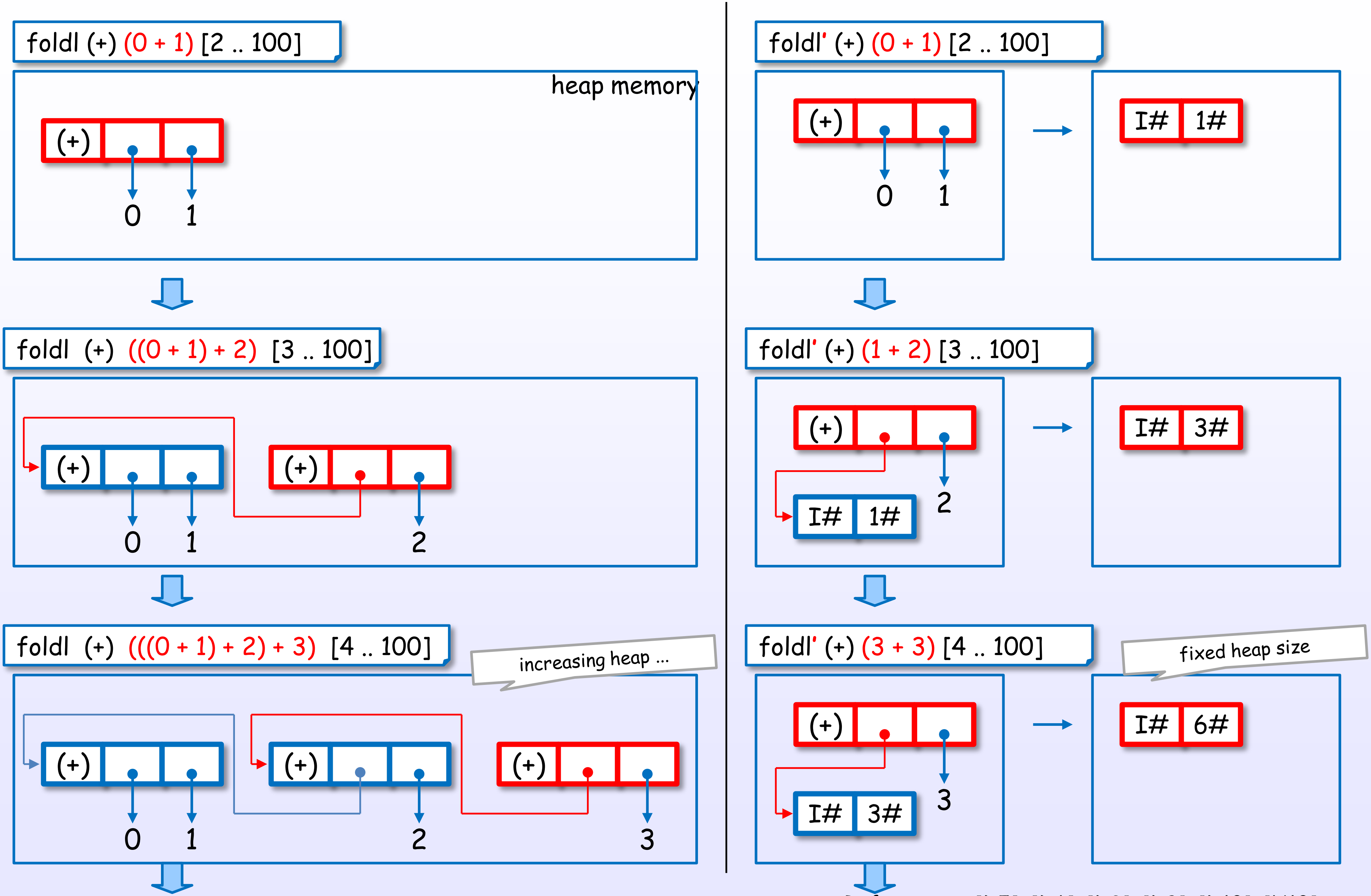
⇩

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

⇩

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

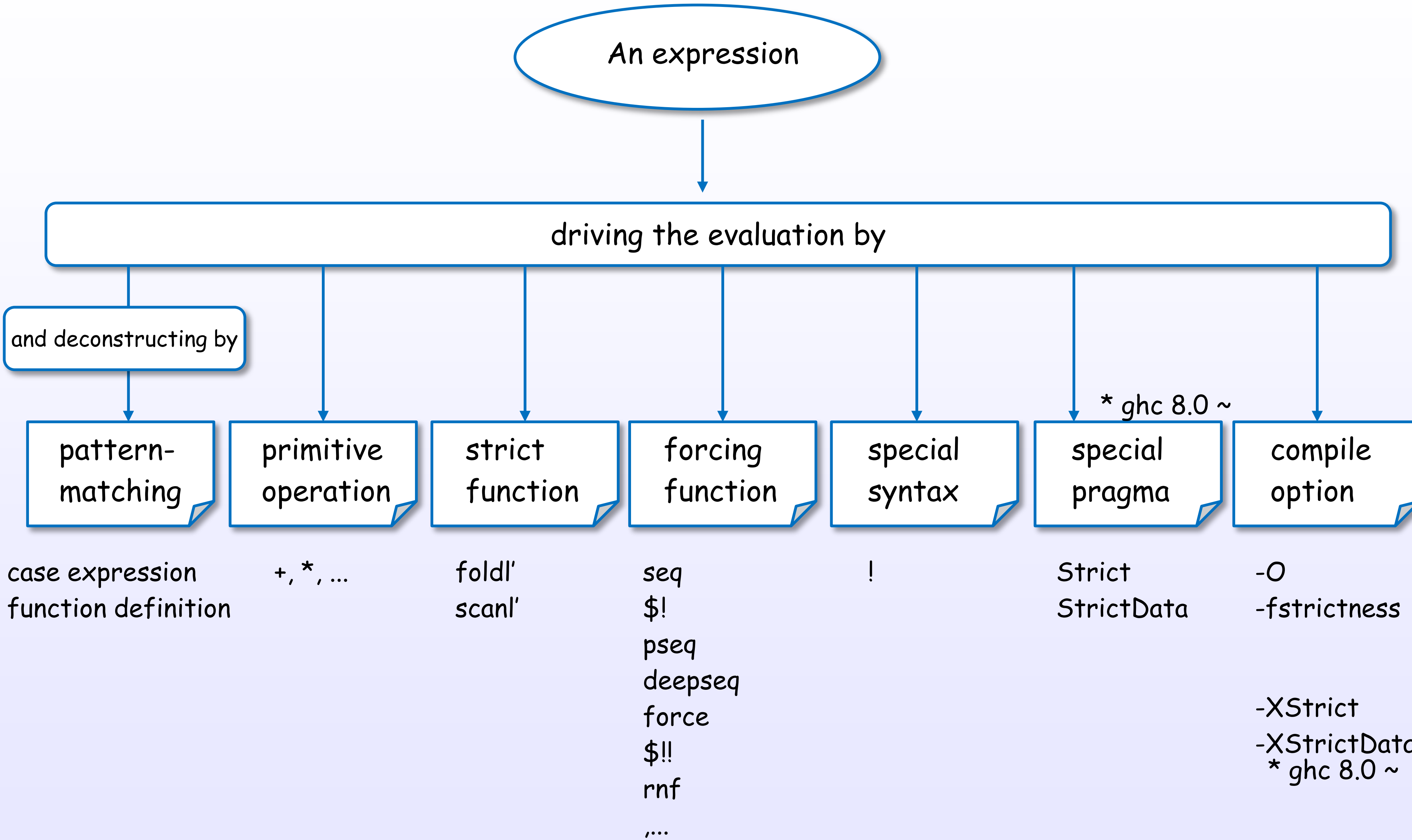
Example of foldl (non-strict) and foldl' (strict)



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

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)

References : [H1] Ch.3, [D2], [D1], [H5], [W1]

(1) Evaluation by pattern-matching

Strict patterns drive the evaluation

case expression

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

function definition

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

Lazy patterns postpone the evaluation.

let binding pattern

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

irrefutable patterns [H1] 3.17

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

There are two kinds of pattern-matching.

References : [H1] Ch.3, [D2], [D1], [H5], [W1]

(2) Evaluation by primitive operation

primitive (built-in) operation

```
f x y = x + y
```

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

+, *, ...

primitive operations are defined like as

pseudo code

```
(+) (I# a) (I# b) = I# (a+b)
```

pattern-matching

References : [D3], [H5], [H12]

(3) Evaluation by strict version function

strict version function

strict application of the operator

```
foldl' (+) 0 xs
```

```
scanl' (+) 0 xs
```

References : [B4] Ch.25, [D9], [B6] Ch.7, [S1], [S2]

(4) Evaluation by forcing function

forcing function to WHNF

```
seq x y
```

forcing
(drive the evaluation of the thunk)

```
f $! x
```

```
pseq x y
```

forcing function to NF

```
deepseq x y
```

```
f $!! x
```

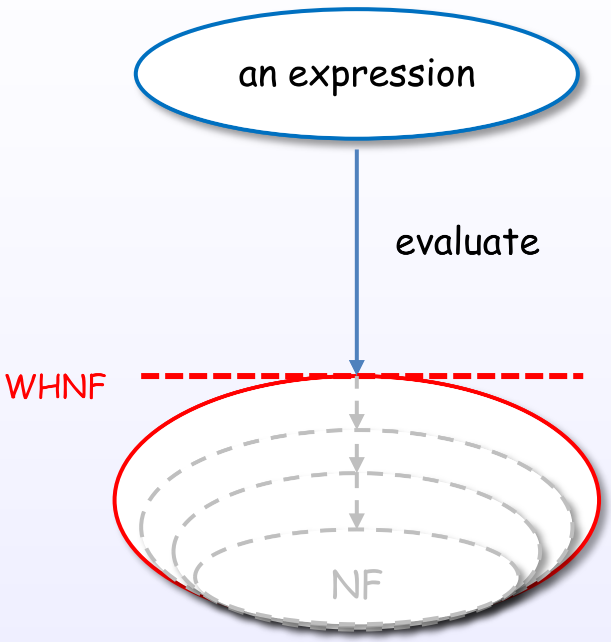
```
force x
```

```
rnf x
```

References : [B5] Ch.2, [B4] Ch.24, 25, [B6] Ch.7, [H1] Ch.6, [D2], [B2], [D1], [D2]. [S1], [S2]

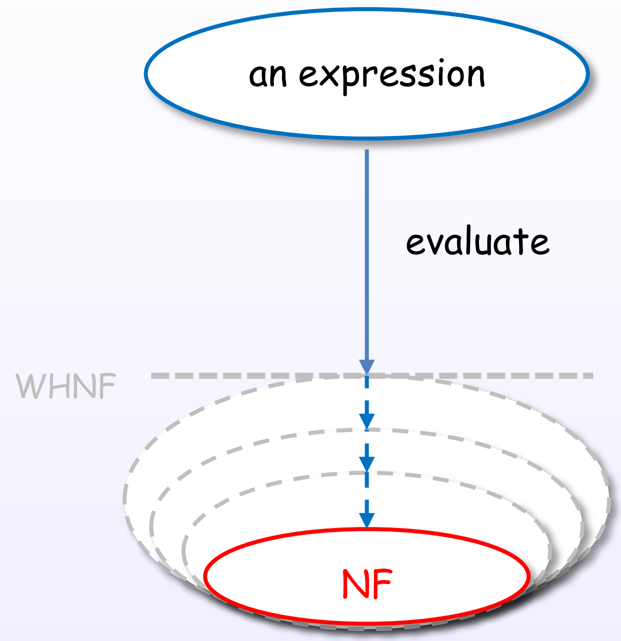
(4) Evaluation by forcing function

to WHNF



seq
$!
pseq

to NF



deepseq
force
$!!
rnf

References : [B5] Ch.2, [B4] Ch.24, 25, [H1] Ch.6, [D2], [B2], [D1], [D2]. [S1], [S2]

(4) Evaluation by forcing function

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

References : [B5] Ch.2, [B4] Ch.24, 25, [H1] Ch.6, [D2], [B2], [D1], [D2]. [S1], [S2]

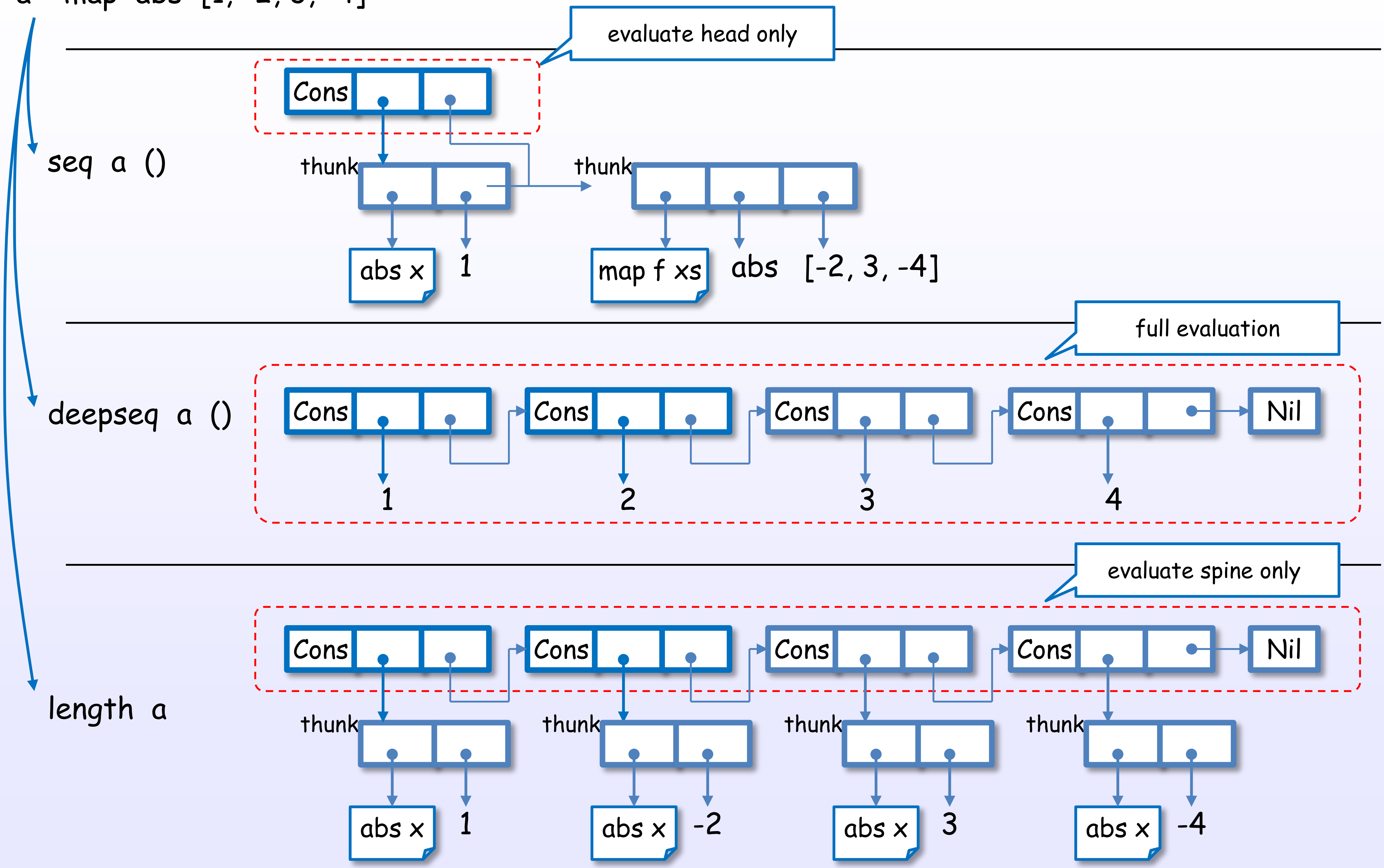
(4) Evaluation by forcing function

	to WHNF	definition	to NF	definition
two expressions	seq	built-in	deepseq	deepseq a b = rnf a `seq` b
one expression			force	force x = x `deepseq` x
function application	$!	f $! x = let !vx = x in f vx	$!!	f $!! x = x `deepseq` f x
sequential order	pseq	pseq x y = x `seq` lazy y		
basic operation			rnf	class NFData a where rnf :: a -> () instance NFData Int where rnf !_ = () : instance NFData a => NFData [a] where rnf [] = () rnf (x:xs) = rnf x `seq` rnf xs

References : [B5] Ch.2, [B4] Ch.24, 25, [H1] Ch.6, [D2], [B2], [D1], [D2]. [S1], [S2]

(4) Evaluation by forcing function

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



References : [B5] Ch.2, [B4] Ch.24, 25, [H1] Ch.6, [D2], [B2], [D1], [D10]. [S1], [S2]

(5) Evaluation by special syntax

Strictness annotation

Bang pattern[H2] 7.19

see also Strict pragma

```
{-# LANGUAGE BangPatterns #-}

f !xs = g xs
```

Strictness flag [H1] 4.2.1

see also StrictData and Strict pragma

```
data Pair = Pair !a !b
```

Strictness annotations assist strictness analysis.

References : [D1], [H2] Ch.7, [H1] Ch.4, [B4] Ch.25, [B2] Ch.7, [W6], [H4] Ch.22, [W4]

(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
```

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.

References : [H13]

(7) Evaluation by compile option

Compile option

strictness analysis

```
$ ghc -O
```

Turn on strictness analysis.

```
$ 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

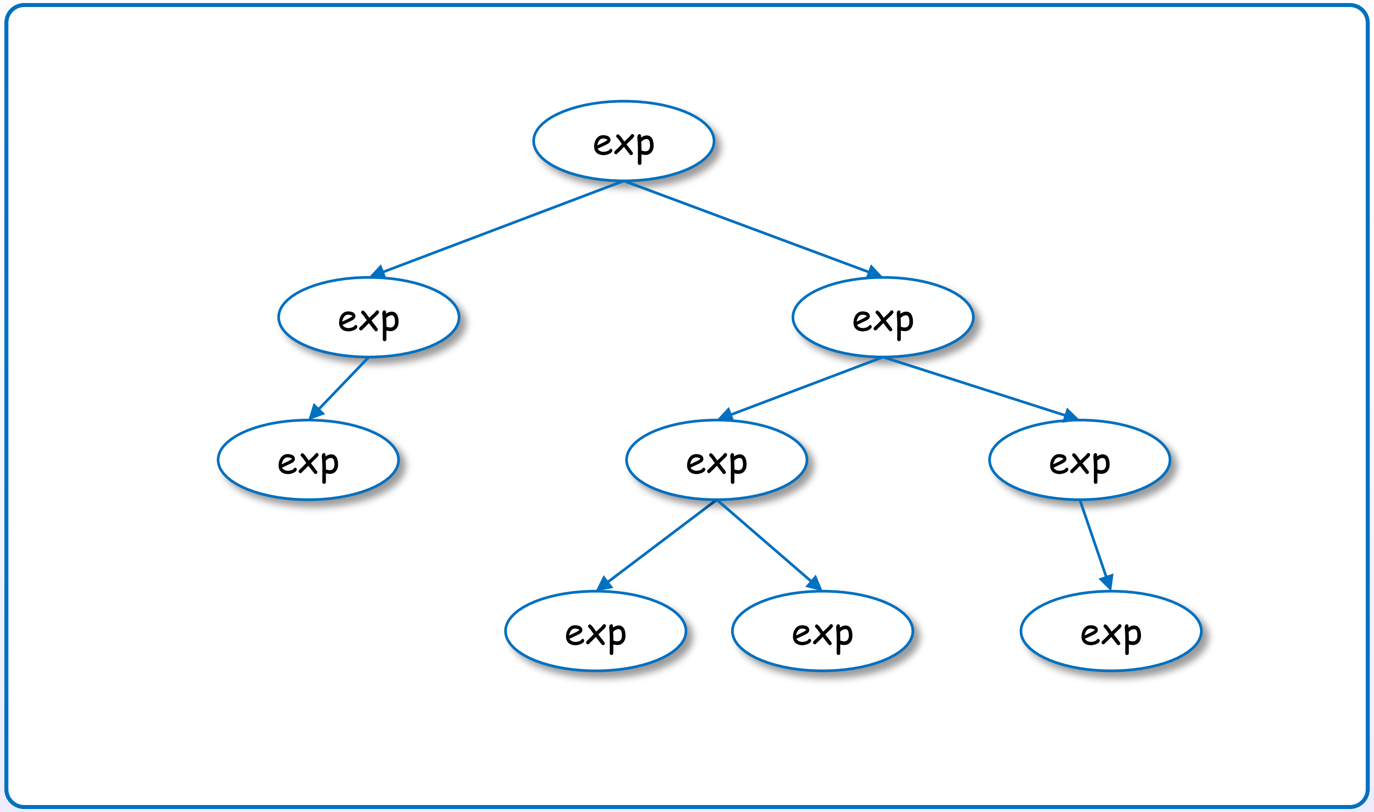
References : [H2], [H13]

5. Implementation of evaluator

5. Implementation of evaluator

Lazy graph reduction

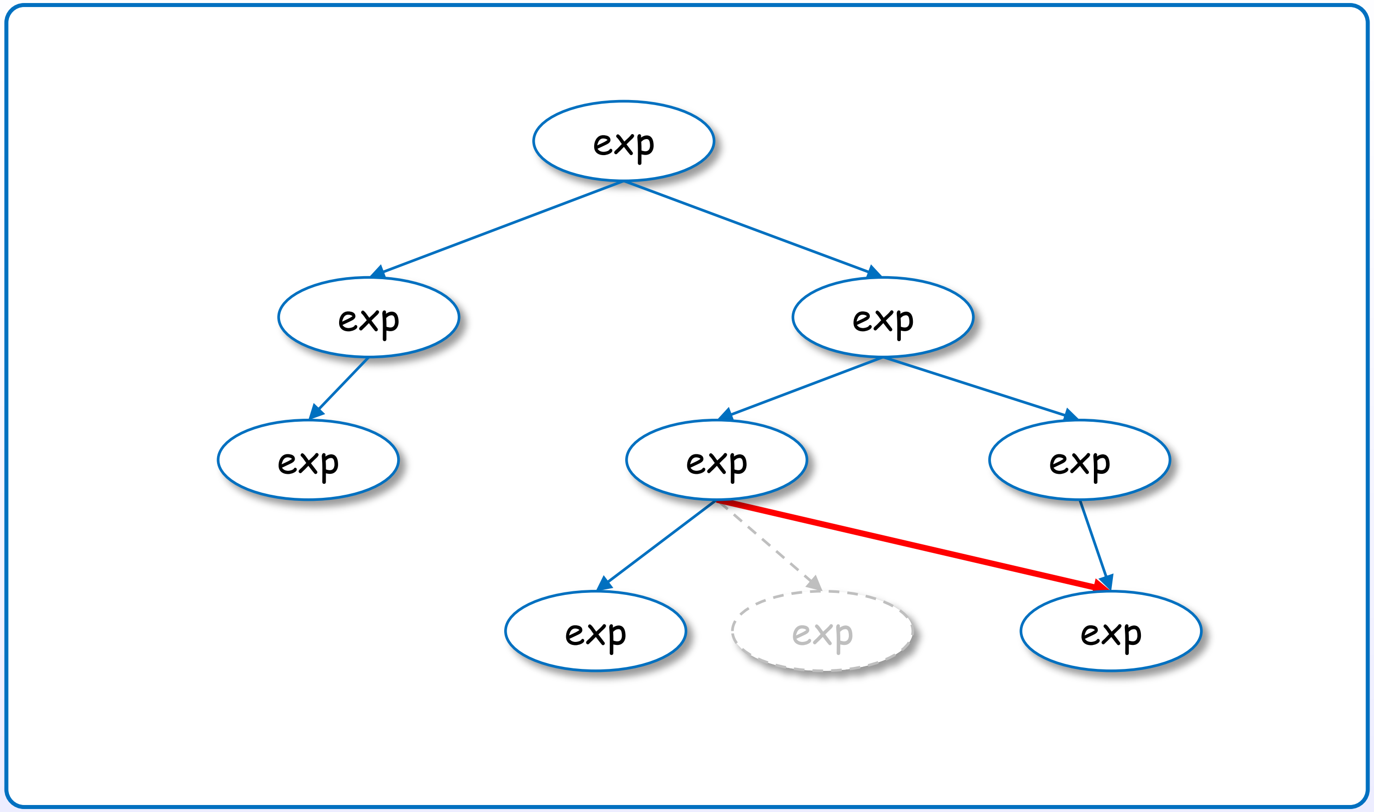
Tree



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

References : [D3], [D2], [D5], [W5], [H4] Ch.12, [D8] Ch.3

Graph



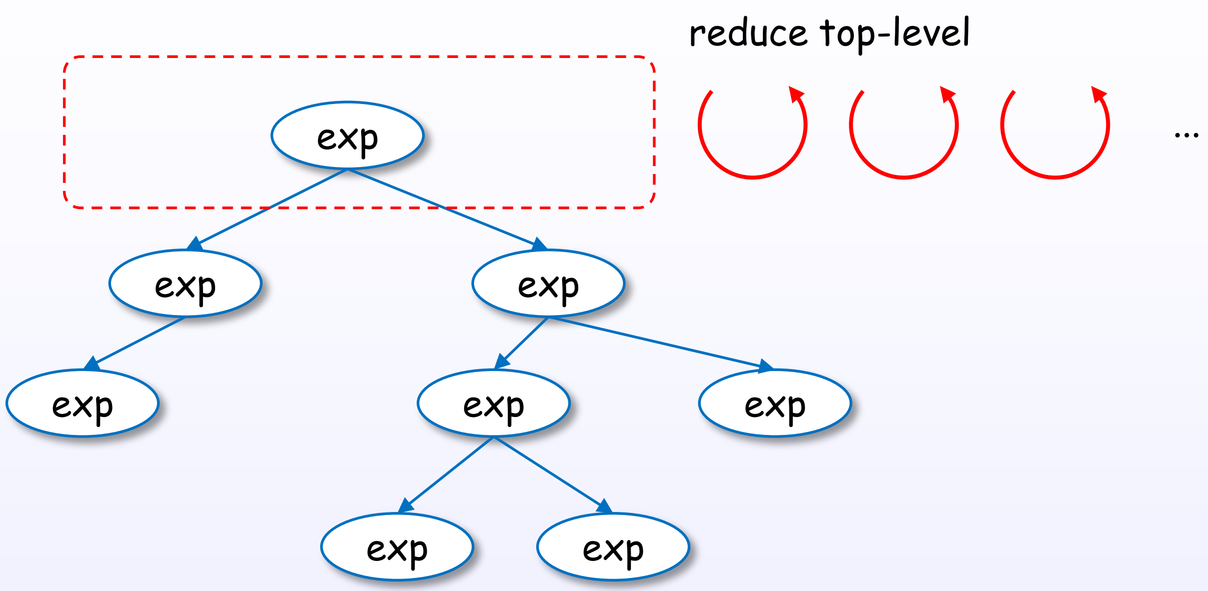
An expression can be also represented with the form of Graph.

Graph can share subexpressions to evaluate at once.

So, graph is reduced using heap (random access memory) rather than stack.

References : [D3], [D2], [D5], [W5], [H4] Ch.12, [D8] Ch.3

Normal order graph reduction



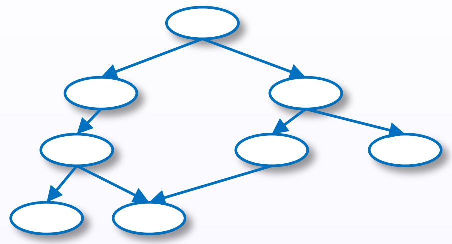
Normal order graph reduction reduces from top-level, thus outermost redex.
Normal order applies first rather than evaluating arguments.

References : [D3], [D2], [D5], [W5], [H4] Ch.12, [D8] Ch.3

STG-machine

Abstract machine

Graph
(expression)



Evaluator
(abstract machine)

STG-machine

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

References : [H5], [H6], [H7], [D15]

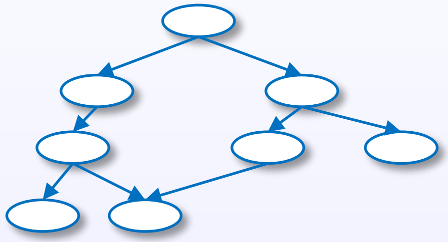
Concept layer

Haskell code

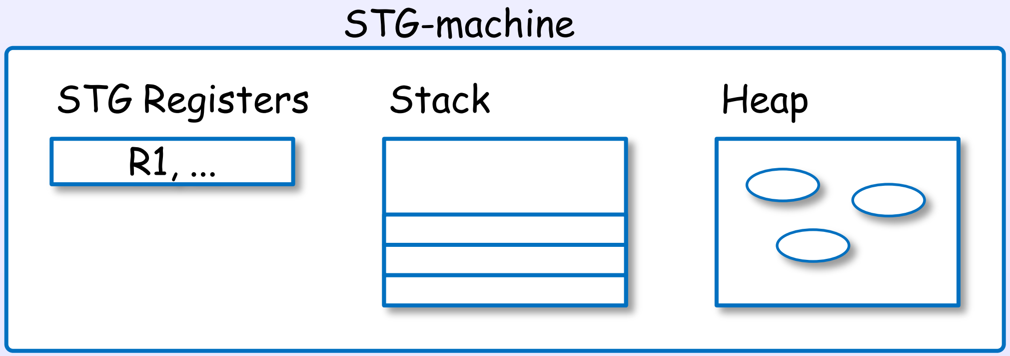
take 5 [1..10]

:

Graph
(internal representation
of expression)

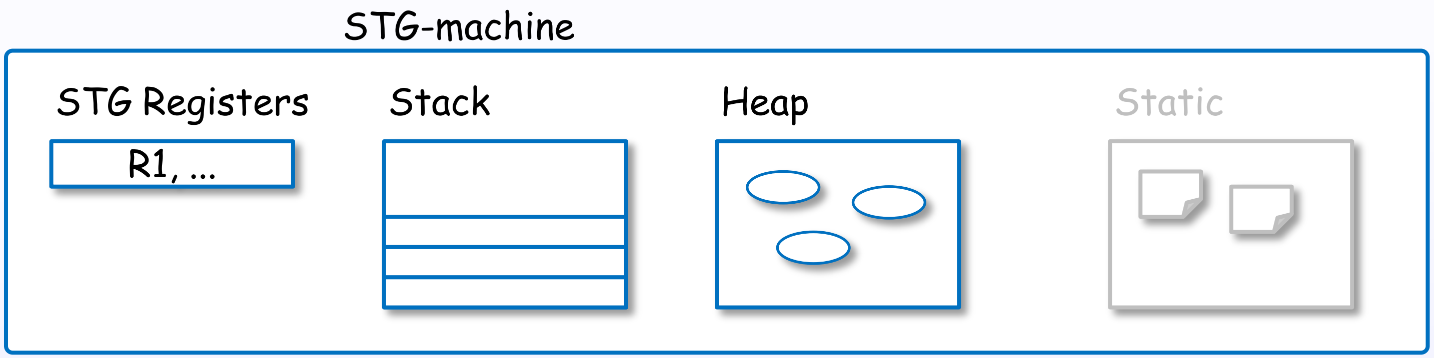


Evaluator (reducer, executer)
(abstract machine)



References : [H5], [H6], [H7], [D15]

STG-machine

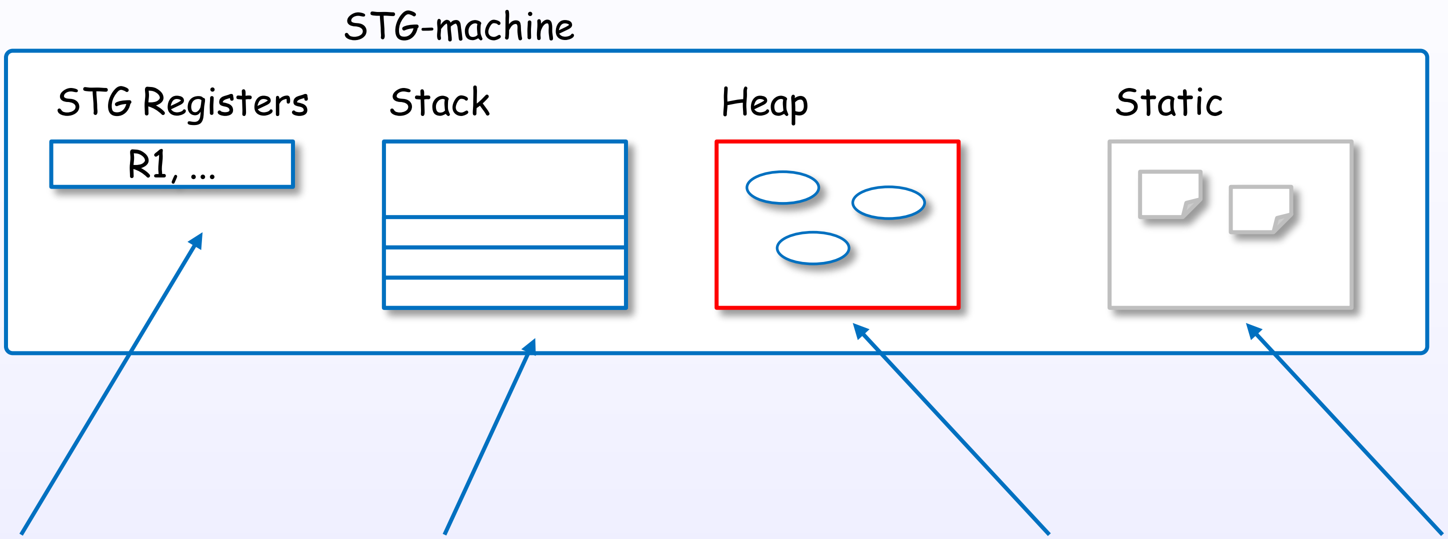


STG-machine is abstraction machine
which is defined by operational semantics.

STG-machine efficiently performs lazy graph reduction.

References : [H5], [H6], [H7], [D15]

STG-machine



mainly used for
call/return convention
various control

mainly used for
nest continuation
argument passing

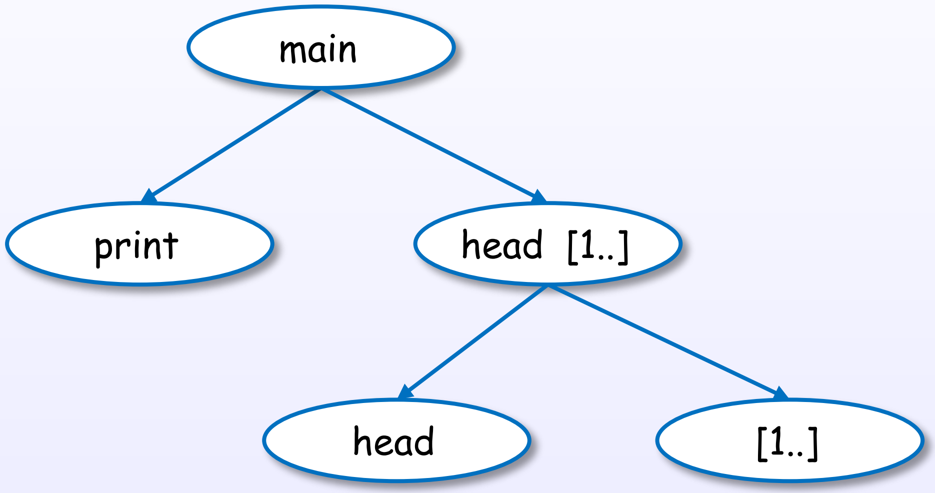
mainly used for
allocating thunks

mainly used for
code
static closure

References : [H5], [H6], [H7], [D15]

Example of mapping a code to a graph

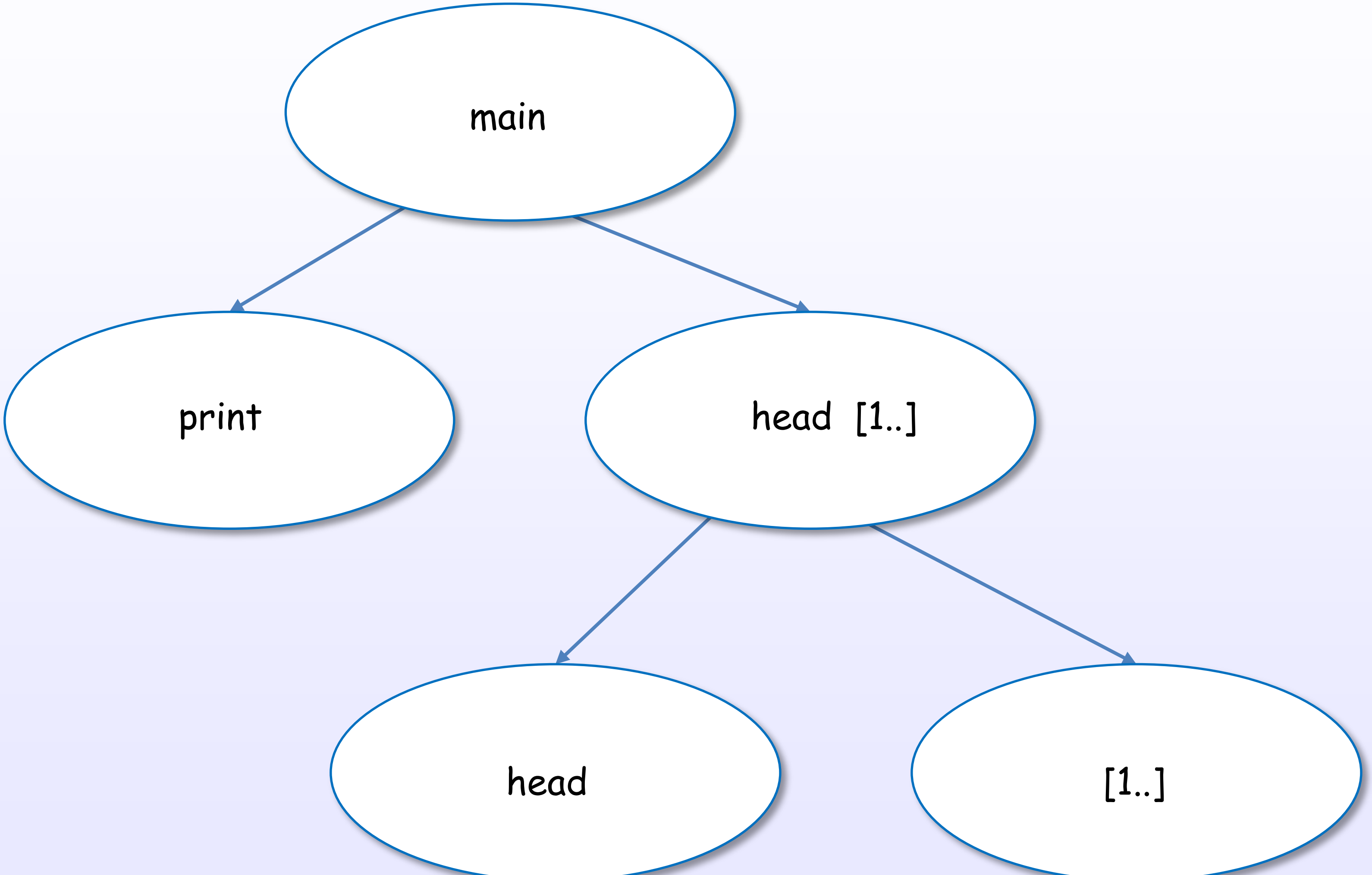
main = print (head [1..])



References : [1]

Example of mapping a code to a graph

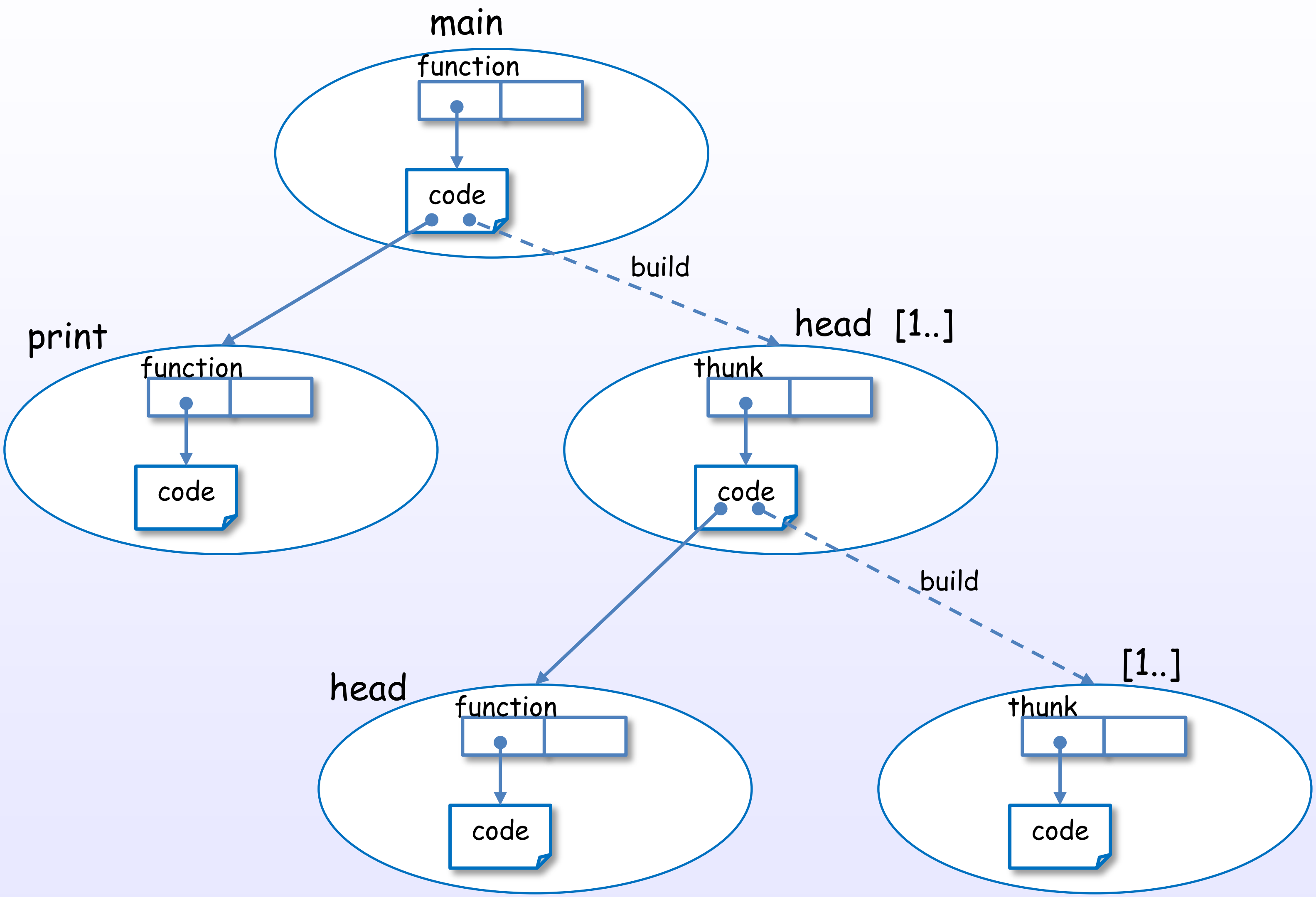
main = print (head [1..])



References : [1]

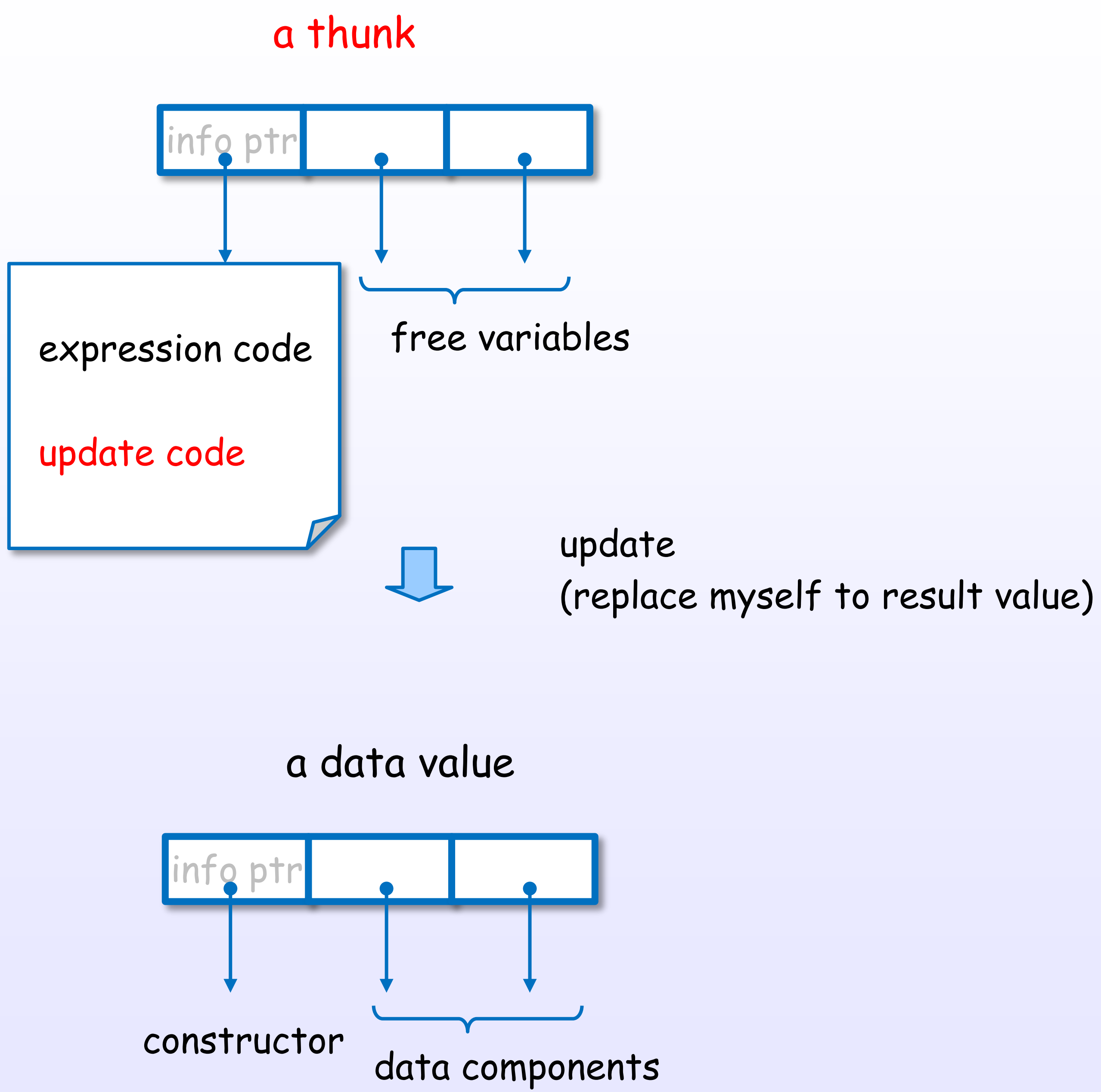
Example of mapping a code to a graph

main = print (head [1..])



References : [1]

Self-updating model



References : [H5], [H6], [H7], [D15]

STG-dump shows which expression build thunks

[Example.hs]

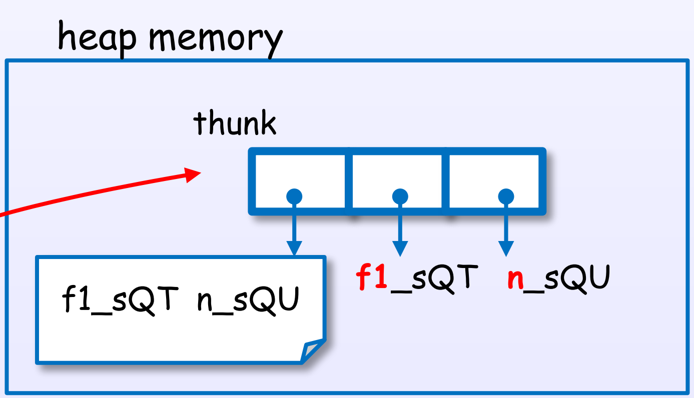
```
module Example where

fun  f1  n = take  1  (f1  n)
```

STG code dump

[$ 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;
```

build/allocate



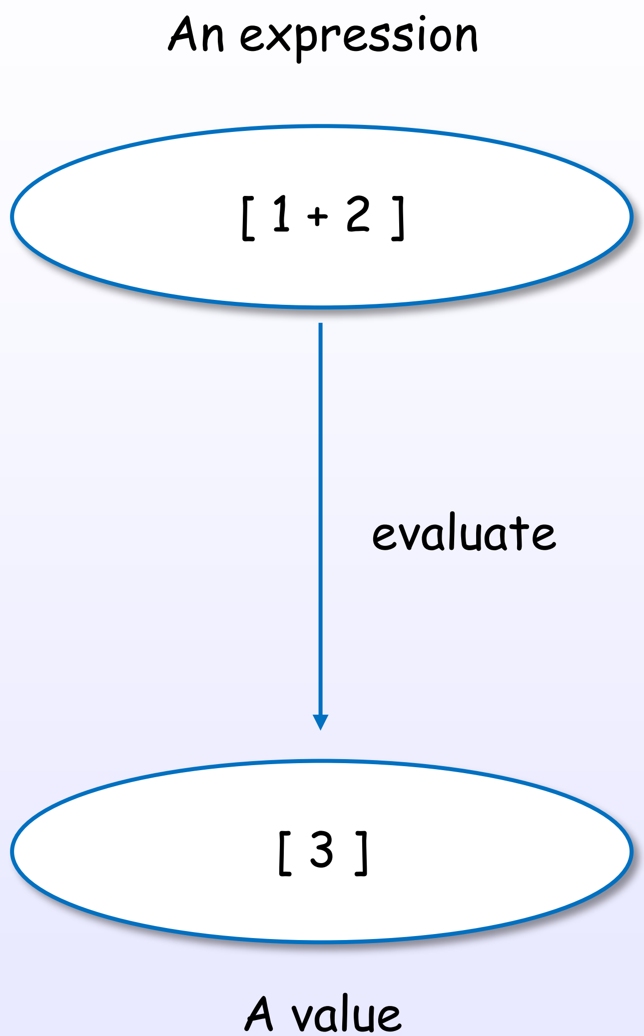
let expression of STG language

References : [H5], [H6], [H7], [D15]

6. Semantics

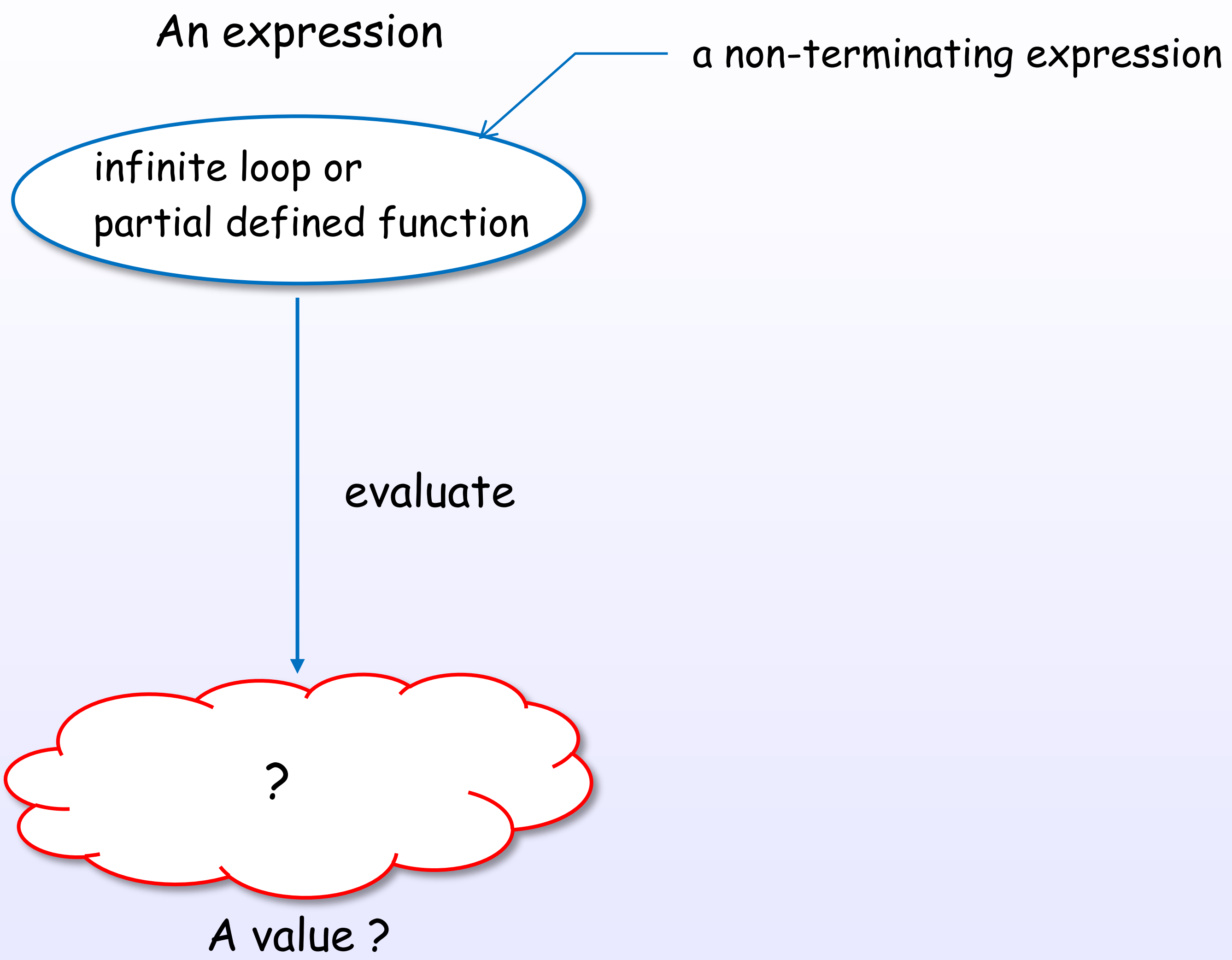
Bottom

Well formed expression should have a value



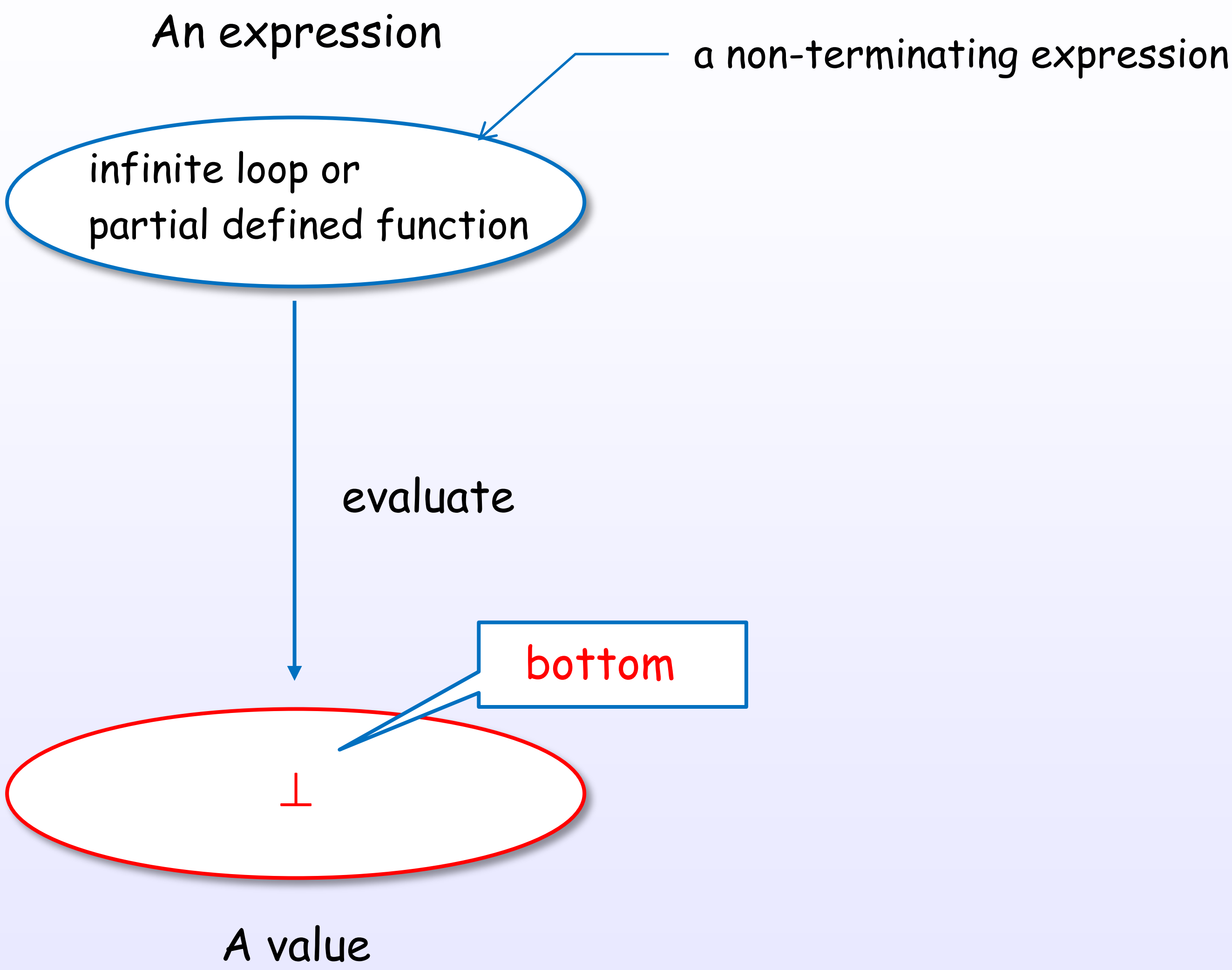
References : [B2] Ch.2, [W4]

What is a value in this case?



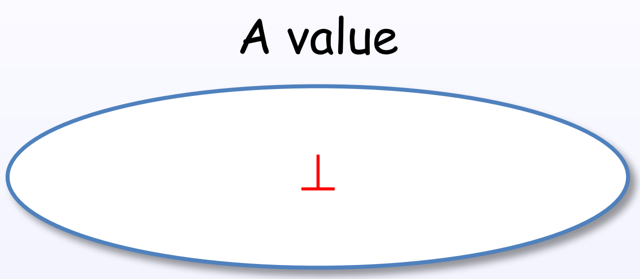
References : [B2] Ch.2, [H1] Ch.3, [W4], [H4] Ch.2, 22

A value "bottom" is introduced



References : [B2] Ch.2, [H1] Ch.3, [W4], [H4] Ch.2, 22

Bottom



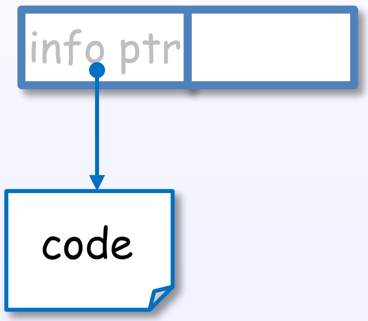
Bottom ($\perp$) is "an undefined value".

Bottom ($\perp$) is "a non-terminating value".

References : [B2] Ch.2, 9, [H1] Ch.3, [W4], [H4] Ch.2, 22

"undefined" represents bottom in Haskell

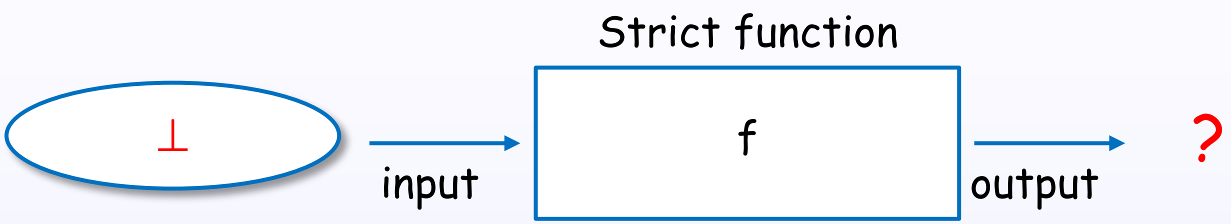
Haskell code	Expression	GHC's internal representation
undefined :: a	$\perp$	GHC.Err.undefined



References : [B2] Ch.2, 9, [H1] Ch.3, [W4], [H4] Ch.2, 22

Non-strict Semantics

Strictness

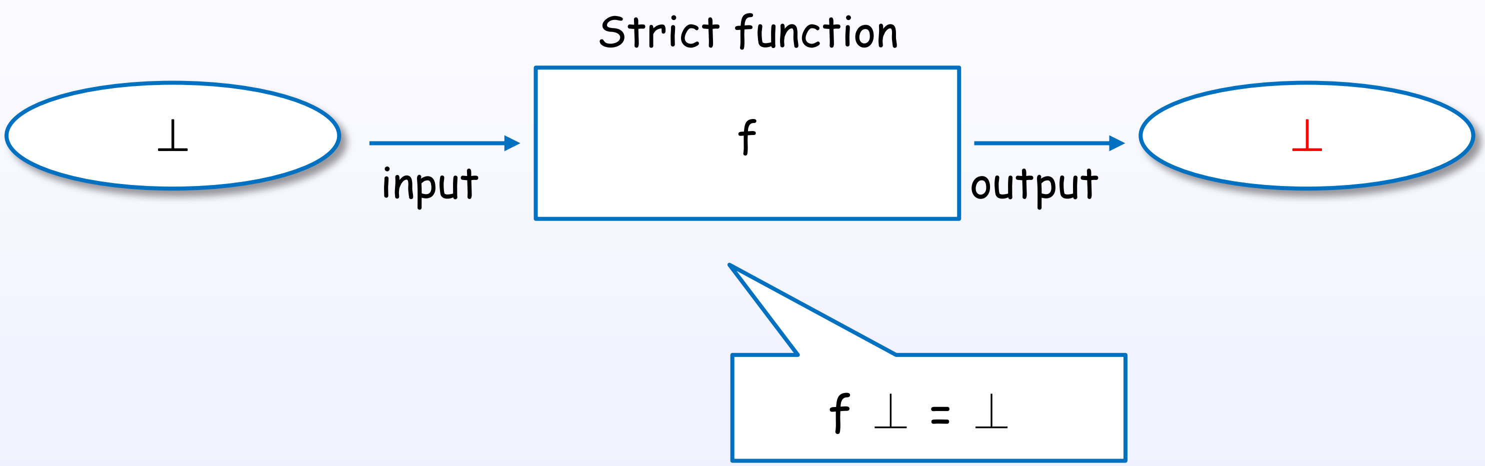


Strictness is property of the function.

"given a non-terminating arguments, f will terminate?"

References : [B2] Ch.2, [W1], [W4], [H4] Ch.2, 22

Strict function

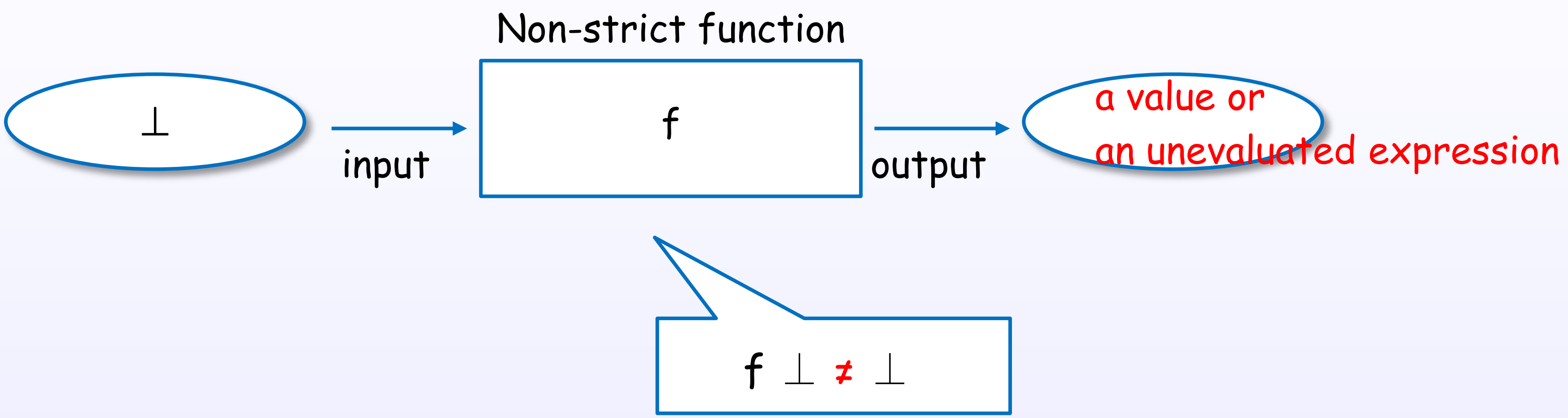


Strict function's output is bottom if input is bottom.

given a non-terminating arguments, f will not terminate.

References : [B2] Ch.2, [W1], [W4], [H4] Ch.2, 22

Non-strict function



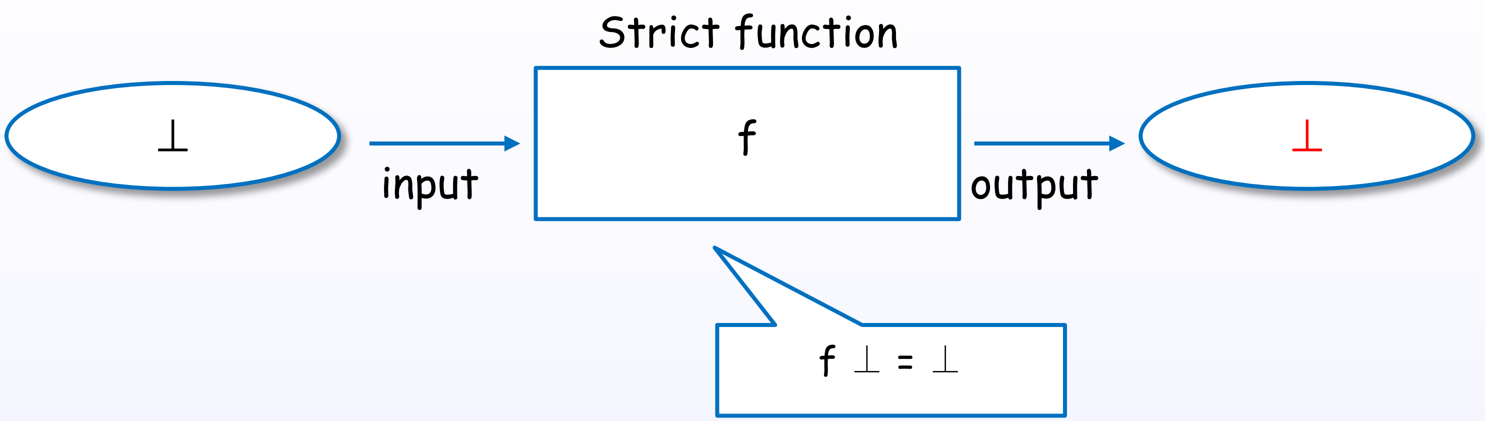
Non-strict function's output is not bottom if input is bottom.

given a non-terminating arguments, f will terminate.

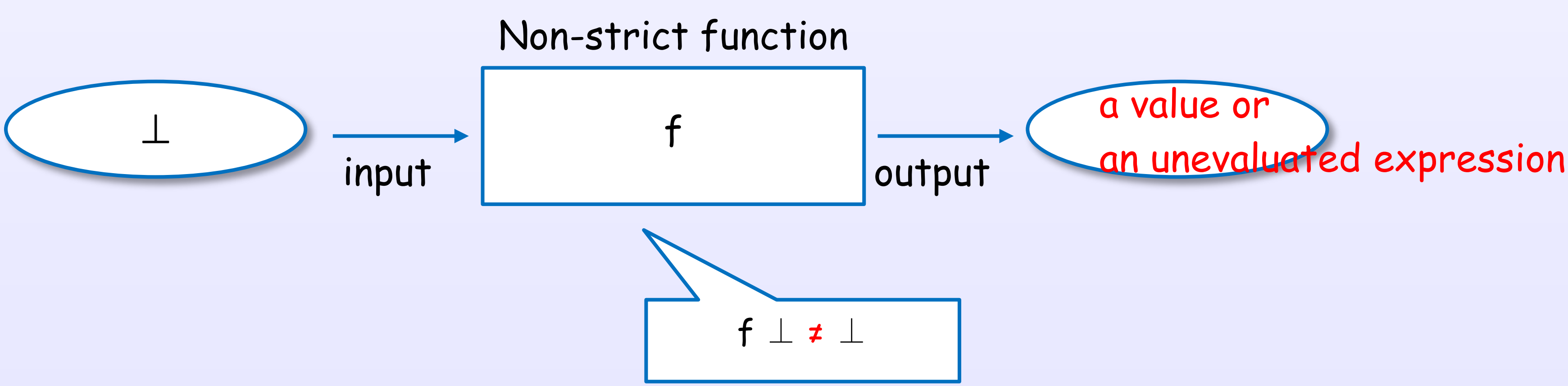
References : [B2] Ch.2, [W1], [W4], [H4] Ch.2, 22

Strictness and Non-strictness

Strict



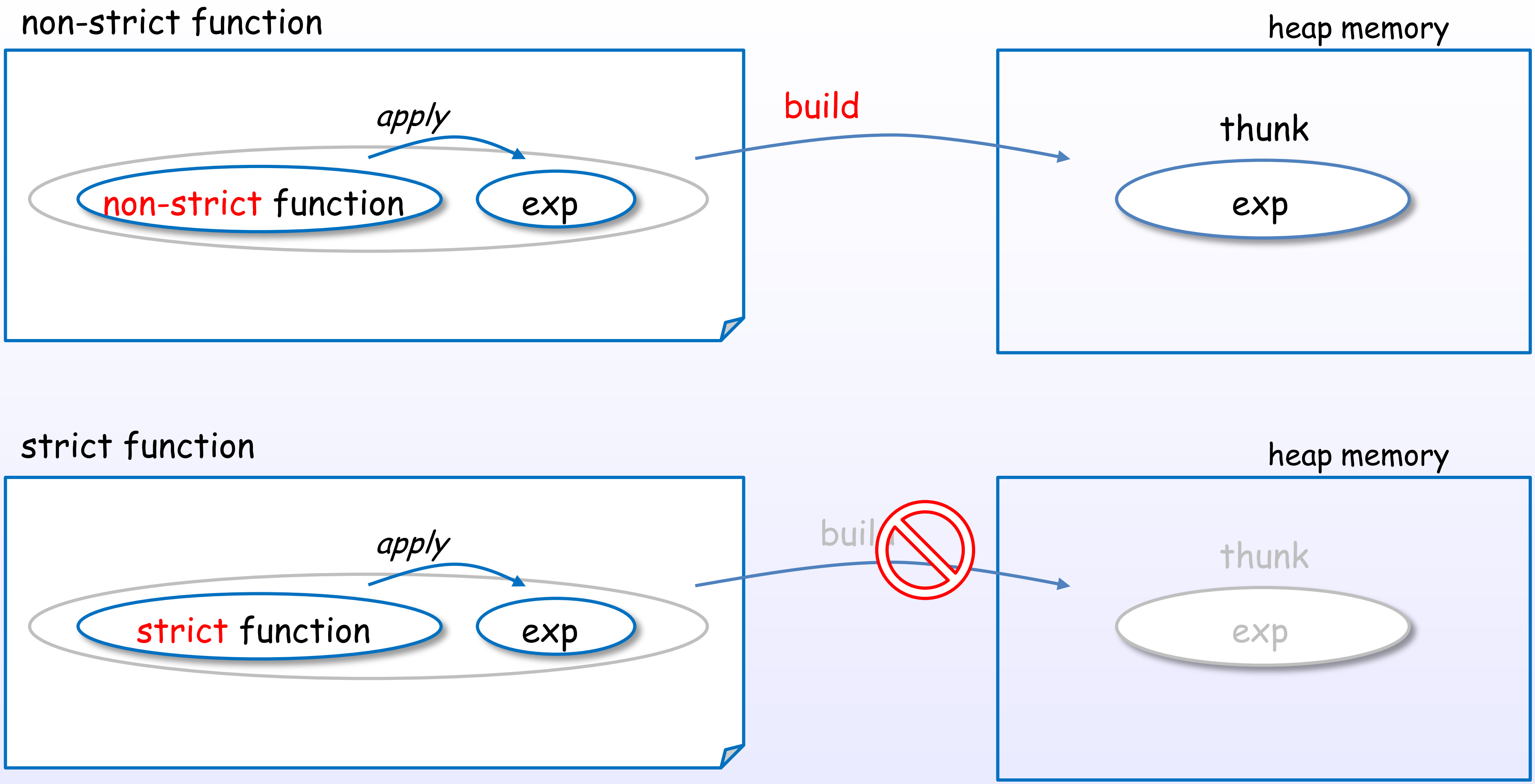
Non-strict



References : [B2] Ch.2, [W1], [W4], [H4] Ch.2, 22

Strict analysis

Strict function's application doesn't build a thunk



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

GHC find strict function by "strictness analysis".

References : [H4] Ch.22, [H8], [W6], [W3], [H14], [H15], [H13]

Strict analysis

Strict analysis can determine which arguments a function is sure to evaluate.

“Does this function always need the value of its argument?”

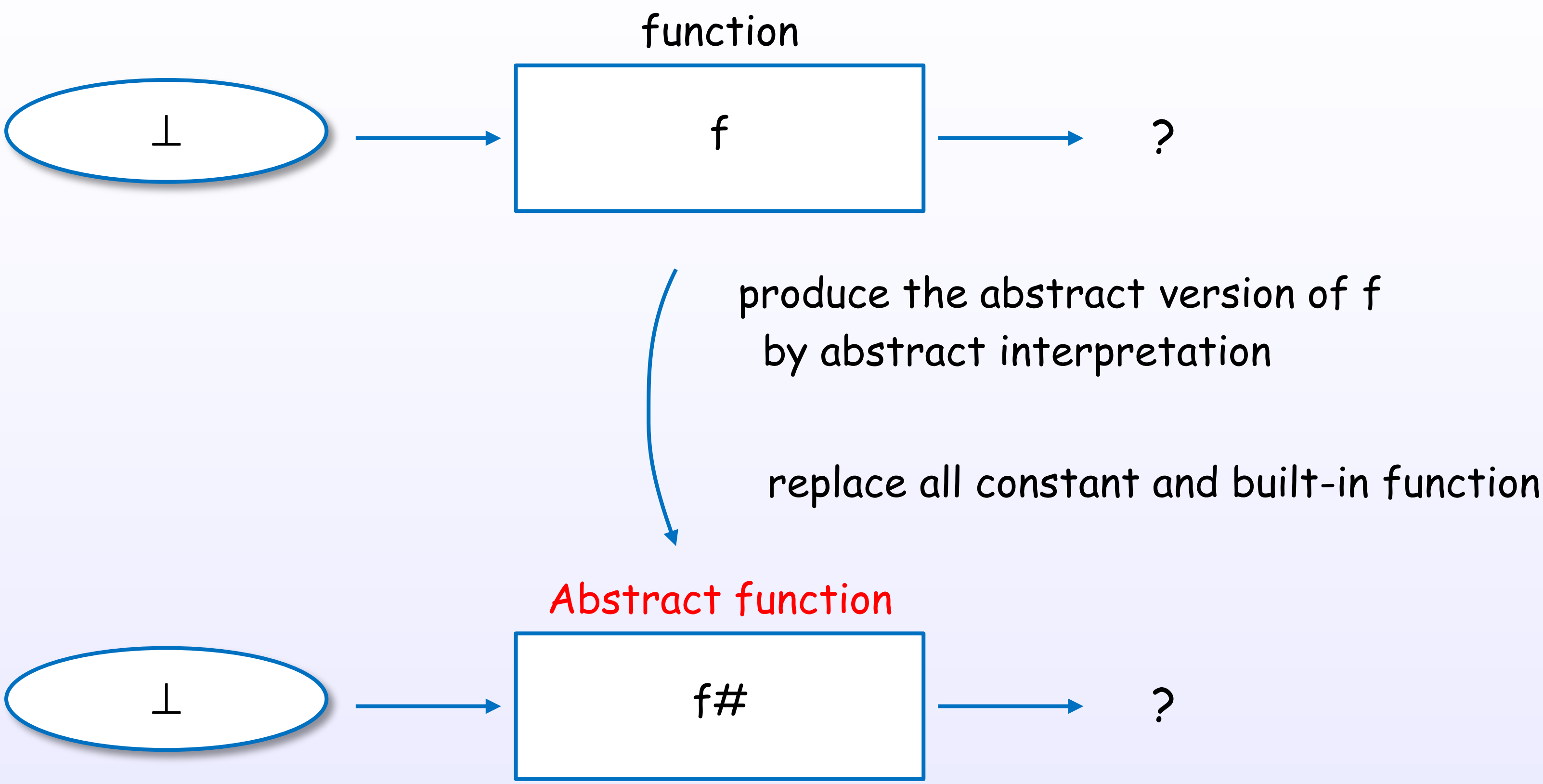
given a function f, does “f $\perp = \perp$” ?

```
$ ghc  -ddump-stranal
$ ghc  -ddump-strsigs
```

[user_guide] -fstrictness

References : [H4] Ch.22, [H8], [W6], [W3], [H14], [H15], [H13]

Strict analysis using abstract function



f# gives a short-cut result.

Abstract function can decide output without going via full evaluation.

References : [H4] Ch.22, [H8], [W6], [W3], [H14], [H15], [H13]

Sequential order

"seq" doesn't guarantee the evaluation order

specification

$$
\begin{aligned}
\text{seq } a\ b &= \bot, && \text{if } a = \bot \\
&= b, && \text{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
\text{seq } \bot\ b &= \bot && \text{// a is strict} \\
\text{seq } a\ \bot &= \bot && \text{// b is strict}
\end{aligned}
$$

The only guarantee made by seq is that it is strict in both arguments.
This semantics property makes no operational guarantee about order of evaluation.

References : [H9], [D11], [H1] Ch.6, [S1]

seq and pseq

specification

$$\begin{aligned} \textbf{seq}\ a\ b &= \perp, && \text{if } a = \perp \\ &= b, && \text{otherwise} \end{aligned}$$

$$\begin{aligned} \text{seq}\ \perp\ b &= \perp && \text{// a is strict} \\ \text{seq}\ a\ \perp &= \perp && \text{// b is strict} \end{aligned}$$

specification

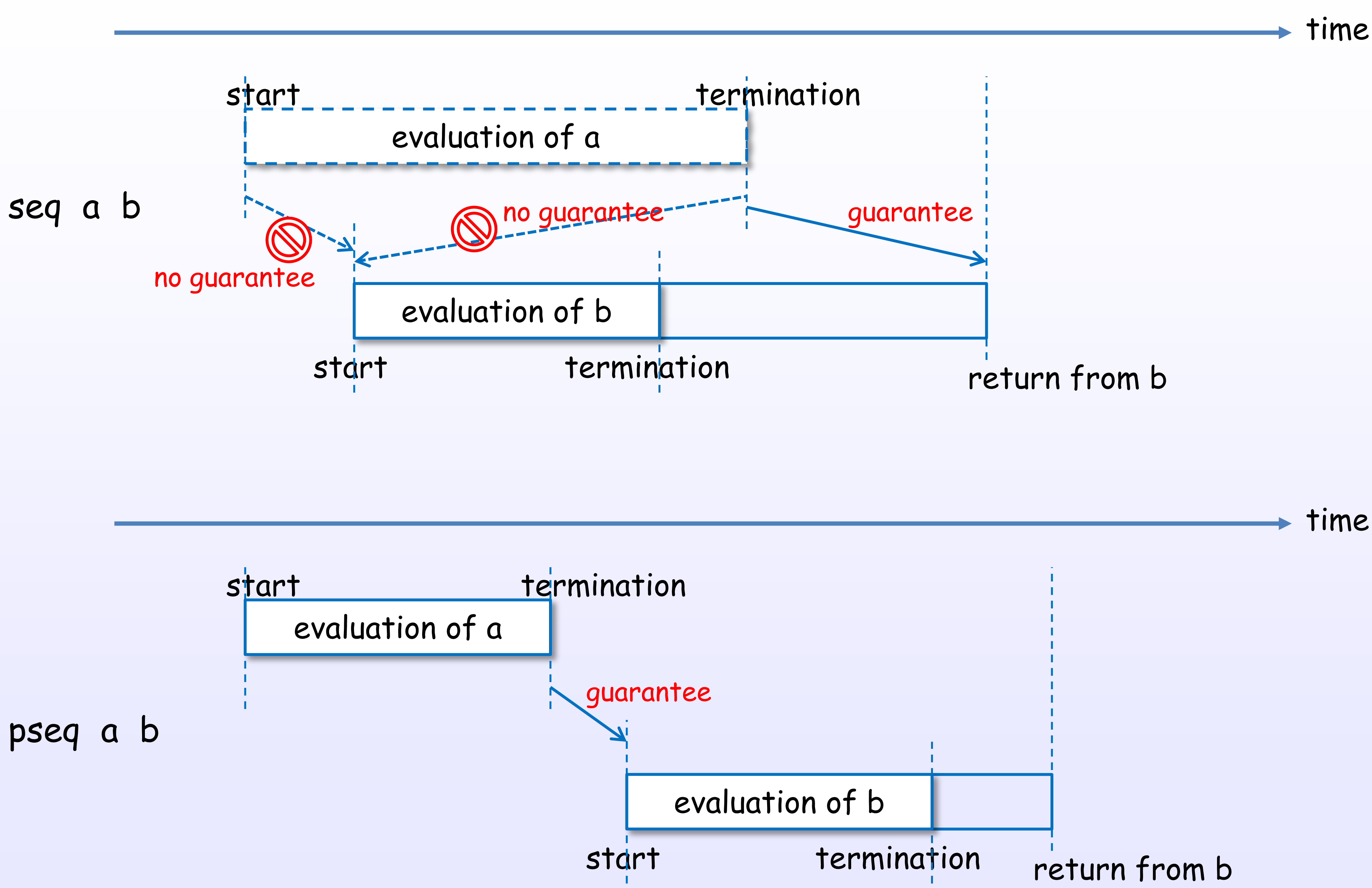
$$\begin{aligned} \textbf{pseq}\ a\ b &= \perp, && \text{if } a = \perp \\ &= b, && \text{otherwise} \end{aligned}$$

$$\begin{aligned} \text{pseq}\ \perp\ b &= \perp && \text{// a is strict} \\ \text{pseq}\ a\ \perp &= \perp && \text{// b is strict} \end{aligned}$$

Both denotational semantics are same.
But pseq makes operational guarantee about order of evaluation.

References : [H9], [D11], [H1] Ch.6, [S1]

Evaluation order of seq and pseq



References : [H9], [D11], [H1] Ch.6, [S1]

Implementation of seq and pseq

specification

$$\text{seq } a\ b = \perp, \quad \text{if } a = \perp$$
$$= b, \quad \text{otherwise}$$

Haskell's built-in

specification

$$\text{pseq } a\ b = \perp, \quad \text{if } a = \perp$$
$$= b, \quad \text{otherwise}$$

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

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

References : [H9], [D11], [H1] Ch.6, [S1]

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] A transformation-based optimiser for Haskell
http://research.microsoft.com/pubs/67064/comp-by-trans-scp.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] Demand analyser in GHC
https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/Demand

[H15] Core-to-Core optimization pipeline
https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/Core2CorePipeline

[H16] The GHC reading list
https://ghc.haskell.org/trac/ghc/wiki/ReadingList

[H17] 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...