Lazy evaluation illustrated

for Haskellers

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
  - Lazy evaluation

2. Expressions
  - Expression and value
  - Expressions in Haskell
  - Classification by value
  - Classification by form
  - WHNF

3. Internal representation of expressions
  - Constructor
  - Thunk
  - Uniform representation
  - let, case expression
  - WHNF

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

5. Implementation of evaluator
  - Lazy graph reduction
  - STG-machine

6. Semantics
  - Bottom
  - Non-strict Semantics
  - Strict analysis

7. Appendix
  - References

1. Introduction

Basic mental models

How to evaluate program in your brain ?

program code

```
code
code
code
  :
```

プログラムは、どの順で評価される？

どういうステップ、どういう順で evaluation (execution, reduction) される？

What are these mental models?

What “mental model” do you have?

References : [1]

One of the mental models for C program

文の並び

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

?

入れ子の構造

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

?

引数の並び

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

?

関数と引数

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

?

どのように評価される？
あなたの頭の中の、評価メンタルモデルは？

References : [1]

One of the mental models for C program

プログラムは、statement の集まり

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

(1) 文は基本的に、
上から下へ評価
downward

statement order

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

(2) 内側の関数評価が先
（内から外へ。）

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

(3) 同階層では、左側の
（左から右へ。）

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

(4) 引数評価が先
（引数評価から関数評価へ。

Each programmers have some mental models in their brain.

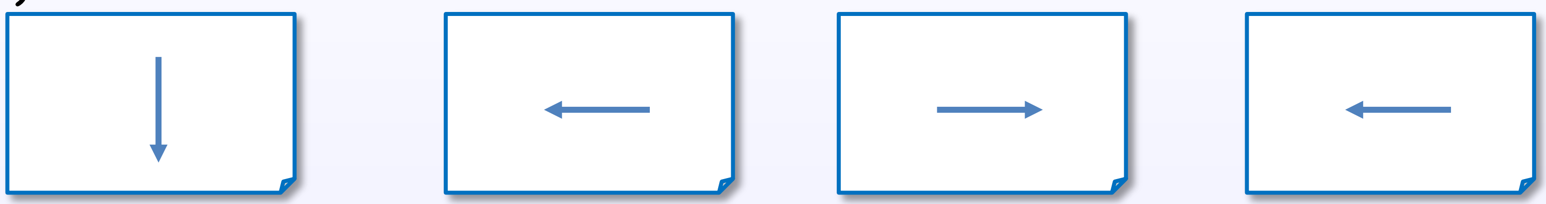
References : [1]

One of the mental models for C program

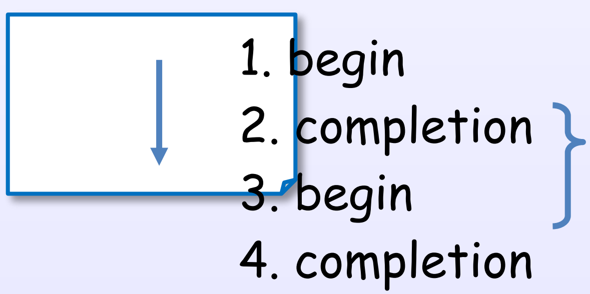
Maybe, You have some implicit mental model in your brain for C program.

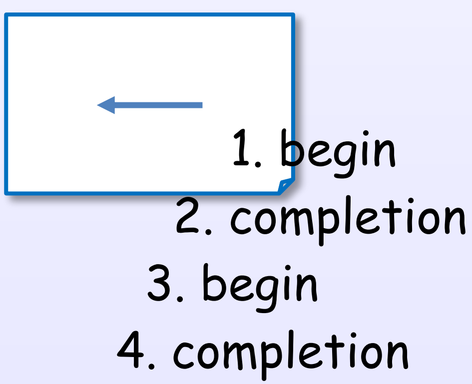
(1) program is collection of statements

(2) an order between evaluations of elements



(3) an order between completion and begin of evaluations





This is an example of an implicit sequential order model for programming languages.

References : [1]

One of the mental models for Haskell program

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

$exp_{13} = exp_{131}\ \ exp_{132}$

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

:

どのように評価される？
あなたの頭の中の、評価メンタルモデルは？

References : [1]

One of the mental models for Haskell program

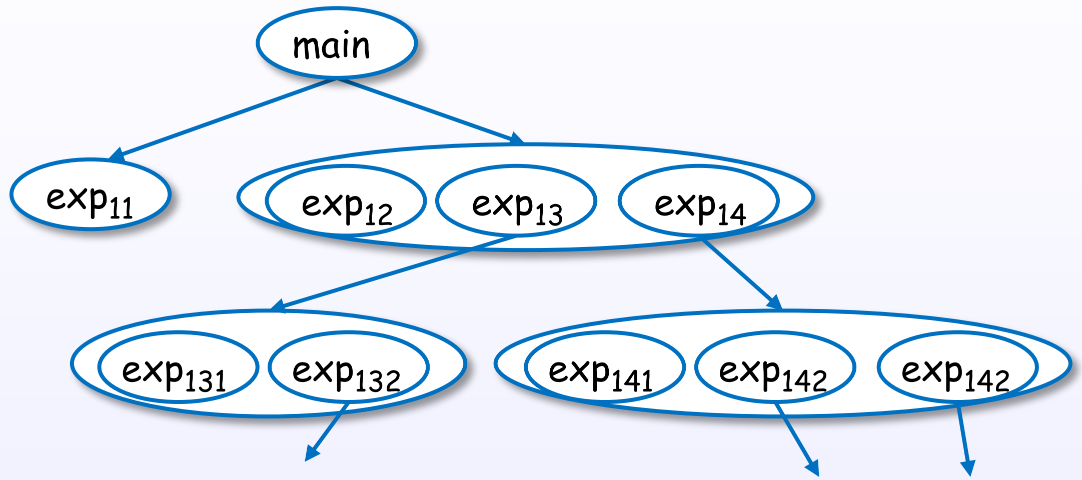
プログラムは、式の集まり

$\mathrm{main} \;=\; \mathrm{exp}_{11}\ (\mathrm{exp}_{12}\ \ \mathrm{exp}_{13}\ \mathrm{exp}_{14}\ )$

$\mathrm{exp}_{13} \;=\; \mathrm{exp}_{131}\ \ \mathrm{exp}_{132}$

$\mathrm{exp}_{14} \;=\; \mathrm{exp}_{141}\ \ \mathrm{exp}_{142}\ \mathrm{exp}_{143}$

:



$$\mathrm{main} \;=\; \mathrm{exp}_{11}\ (\mathrm{exp}_{12}\ \ (\mathrm{exp}_{131}\ \mathrm{exp}_{132}\ )\ (\mathrm{exp}_{141}\ \mathrm{exp}_{142}\ \mathrm{exp}_{143})\ )$$

(1) プログラム全体を1つの式と見立てて

(2) 部分式をある順で評価(簡約)していく

(3) 評価は置換により行う

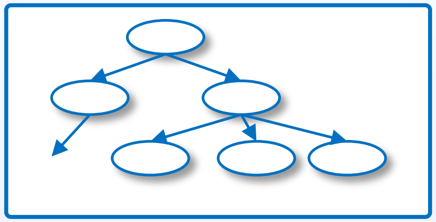
References : [1]

One of the mental models for Haskell program

(1) program is collection of expression's declaration

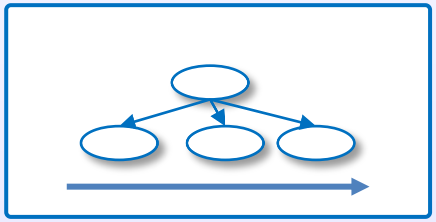
(2) プログラム全体が階層をもった1つの式

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

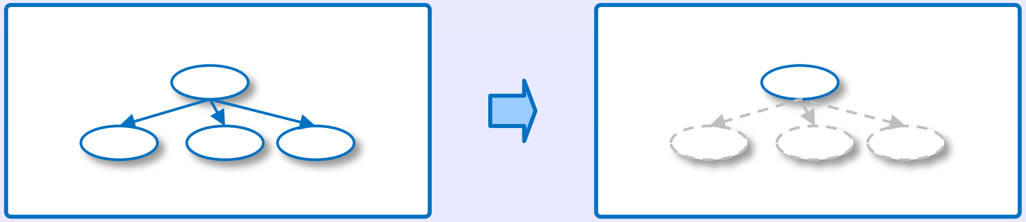


(3) 部分式を、ある順序で評価していく

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



(4) 評価は置換により行われる

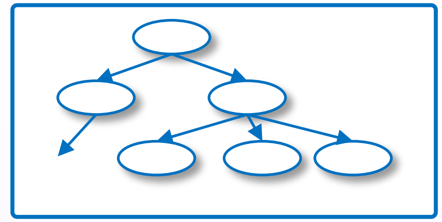


References : [1]

Lazy evaluation

では、具体的にはどうやって評価される？

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



Haskellは purely functional language

↓ no side effect　[slpj-book-1987], p.193

order free (so, potentially hi-level optimization and parallelism

↓ call-by-need 可能

GHC chosen lazy evaluation to implement non-strict semantics.

[slpj-book-1987], p.33

[CIS194]

References : [1]

GHC chosen lazy evaluation

必要な時に、必要な箇所のみを評価する

(STG p.11)

↓

・引数評価を先送る （case式が来るまで評価しない） call-by-need

・部分式を完全評価しない （caseのパターンマッチで参照するところのみを評価する）WHNF

これは、計算量を最小化する戦略（メモリ量でなく）

References : [1]

GHC chosen lazy evaluation

必要になるまで計算しない

無駄な計算をしないように

to avoid unnecessary computation

(performance)

無限構造を扱えるように

to manipulate infinite and huge
data structure
naturally

(abstraction)

非同期事象も

References : [1]

GHC chosen lazy evaluation

計算を後回しにして、「性能」と「表現力」を高めるアプローチ

↓ 無駄な計算をしないように

↓ 大きなものを自然に扱える

References : [1]

Haskell(GHC) 's lazy evaluation

ingredient of Haskell's "lazy evaluation"

when needed

evaluate only if needed

postpone the evaluation until it is needed

+

only to WHNF

必要な部分のみ

+

only once

evaluate only once

only be evaluated once

---

normal order reduction
(= leftmost + outermost reduction)

call-by-need

[slpj-book-1987], 194

lazy constructor

stop at WHNF

[slpj-book-1987], 197

[slpj-book-1987], Cha

substitute pointers

update redex root with result

call-by-need (sharing)

[slpj-book-1987], p.198, 23, 194

graph reduction

[Bird, Chap 7]

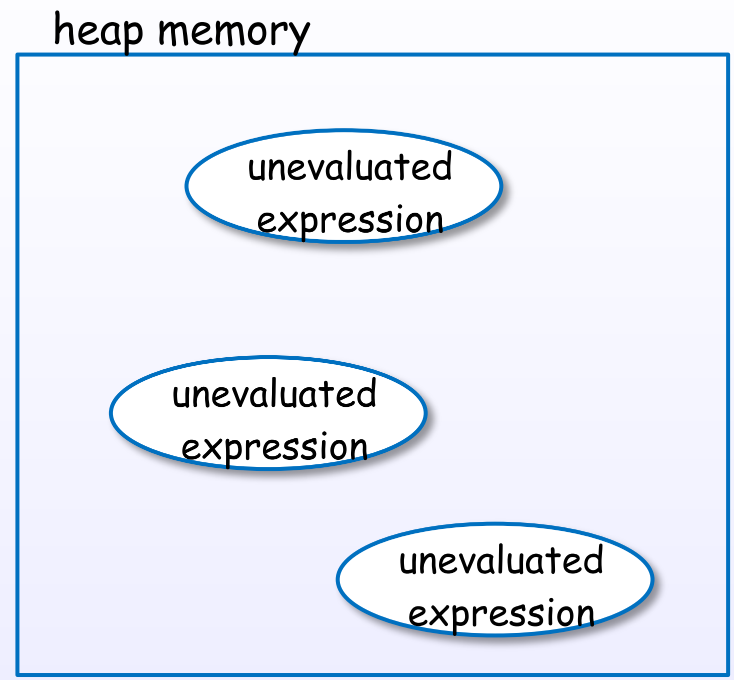
self updating model

call-by-needは、狭義のlazy eval

References : [1]

では、必要な時までどこに置いておく？

postpone



stackでなく、heap。
なので、sequential アクセスでなくて良い。

heapに置いておく

References : [1]

では、必要になるのは、いつ？

必要になるのはいつか？

↓

要素が取り出されるとき
(case, built-in)

“give me your components”

↓

forcing要求のとき
明示的に指示があったとき

“I need you”

References : [1]

では、必要になるのは、いつ？

case式か、関数定義のパターンマッチで、取りだされるときが、必要なとき

```
f = case  (g x) of
        [] -> a
        _   -> b


g  (x:xs) = ...
g  []      = ...
```

HERE!

pattern match via
case expression and function definition
will {cause, trigger} the evaluation

References : [1]

では、必要な部分とはどこ？

パターンマッチで明示された部分

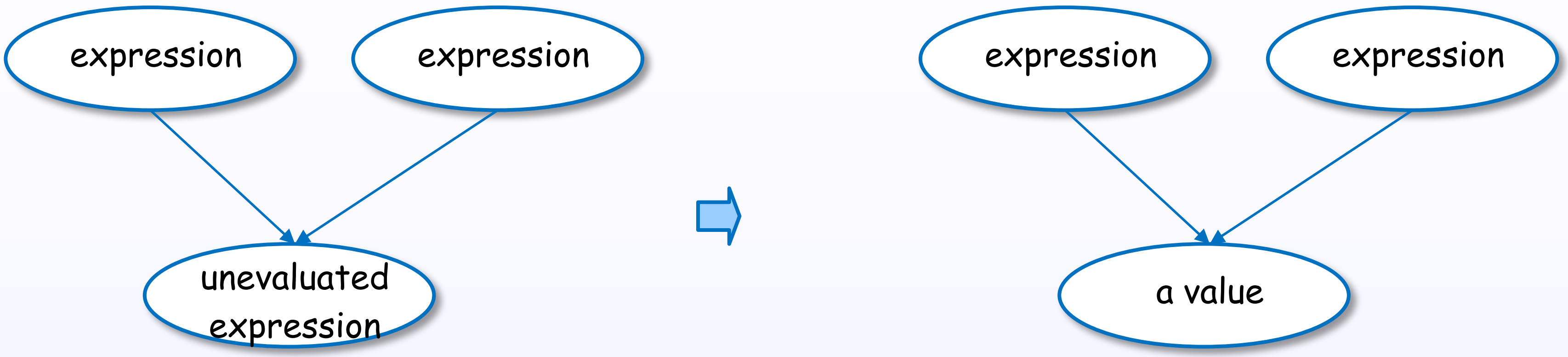
```
f = case  (g a) of
        x : y : _ -> k x y
        []        -> False

f = case  (g a) of
        Just   _   -> True
        Nothing    -> False
```

HERE!

there are components which you need.

References : [1]

どうやって、一度だけ評価する？



self updating

shared term

repeat call

References : [1]

Why lazy evaluation?

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

[slpj-book-1987], p.25

pursue normal order reduction, but stop at WHNF.
This is an essential ingredient of lazy evaluation

(2) lazy evaluation implements non-strict semantics

infinite data structure and stream

[slpj-book-1987], p.194

(3) 不要な評価を避ける

References : [1]

Lazy evaluationの注意点

(1) realtime タイミングが分かりにくい（計算量でなく）
code と 実行が同期していない

(2) 後回しにするための性能コスト。
性能が良くなるのは、「後回しコスト ＜ 抑制効果」のとき

(3) 後回しにするためのメモリコスト （ヒープに隠れスペースリーク）

-> lazy と eager をうまくバランスとれば、good

References : [1]

Lazy evaluationの注意点 1

実行タイミングがずれる

code と 実行が同期していない

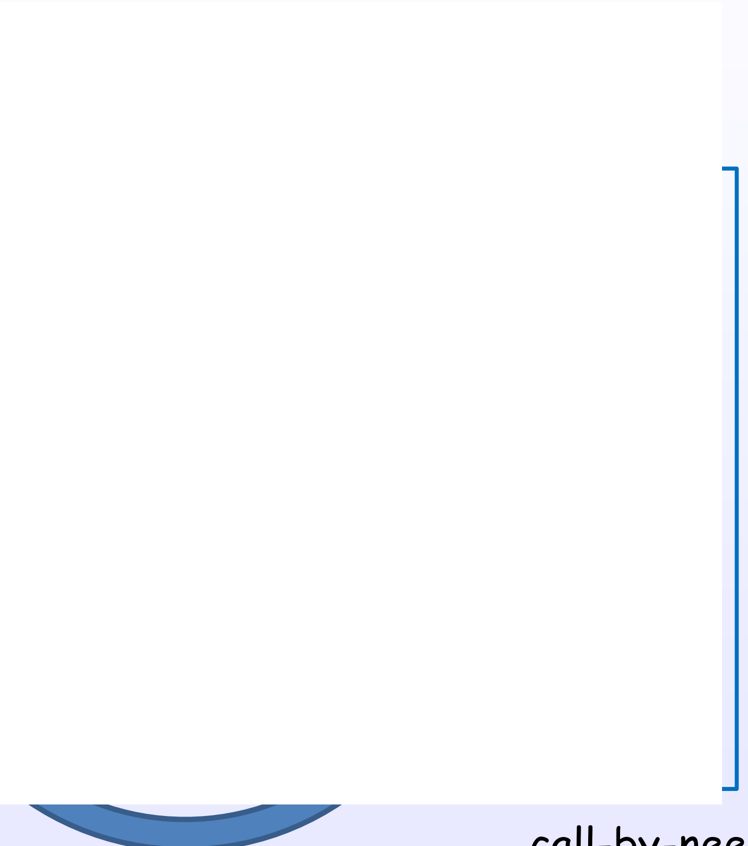
References : [1]

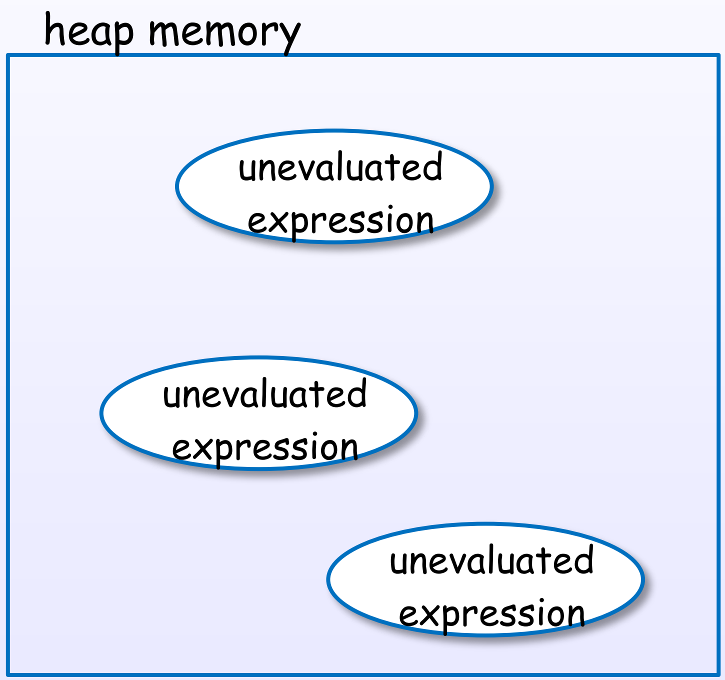
Lazy evaluationの注意点 2

ヒープの使用

ヒープにたまっていく

[slpj-book-1987], p.194





call-by-needは、スタックベースでは実装が難しい。 [hack.hands]

[CIS194]

コントロールが必要

space leak

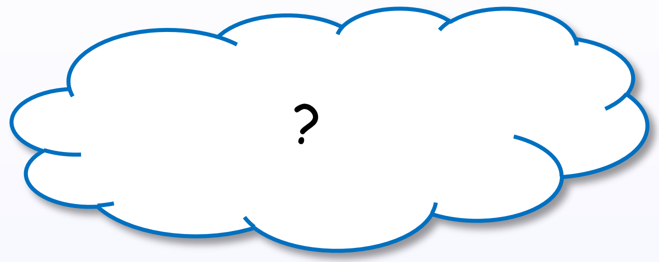
References : [1]

2. Expressions

Expression and value

What is an expression?

An expression



References : [1]

An expression denotes a value

An expression

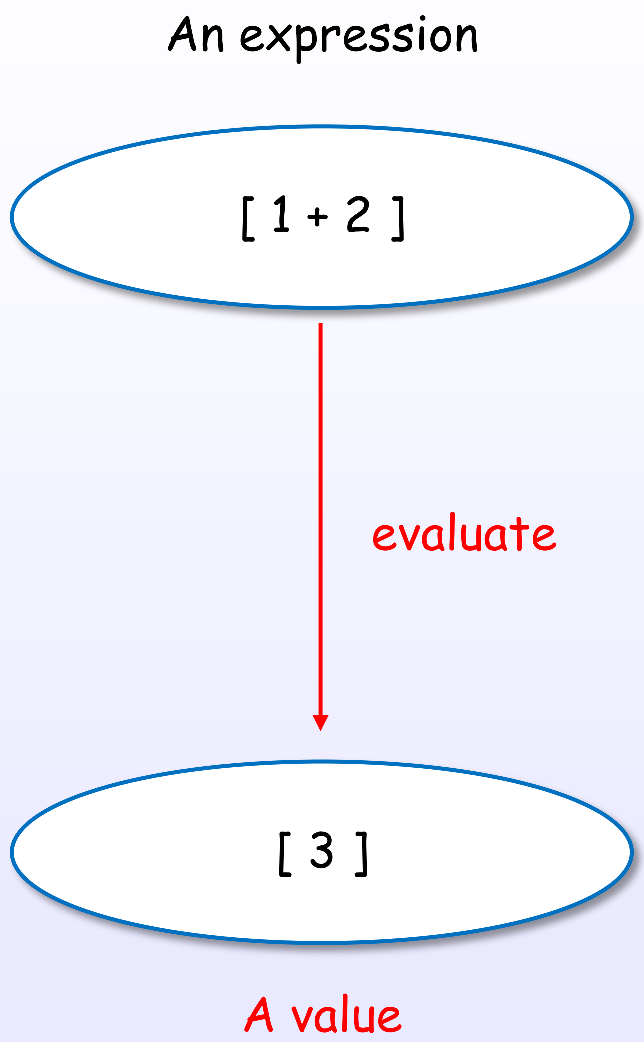
[1 + 2]

[HR2010]

[Bird, Chapter 2]

References : [1]

An expression evaluates to a value



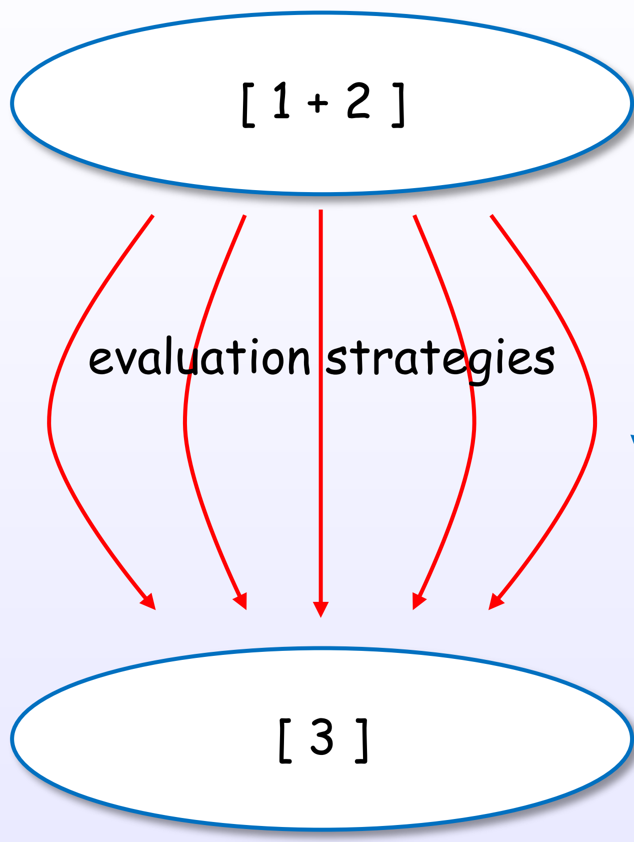
[HR2010]

[Bird, Chapter 2]

References : [1]

There are many evaluation approaches

An expression



A value

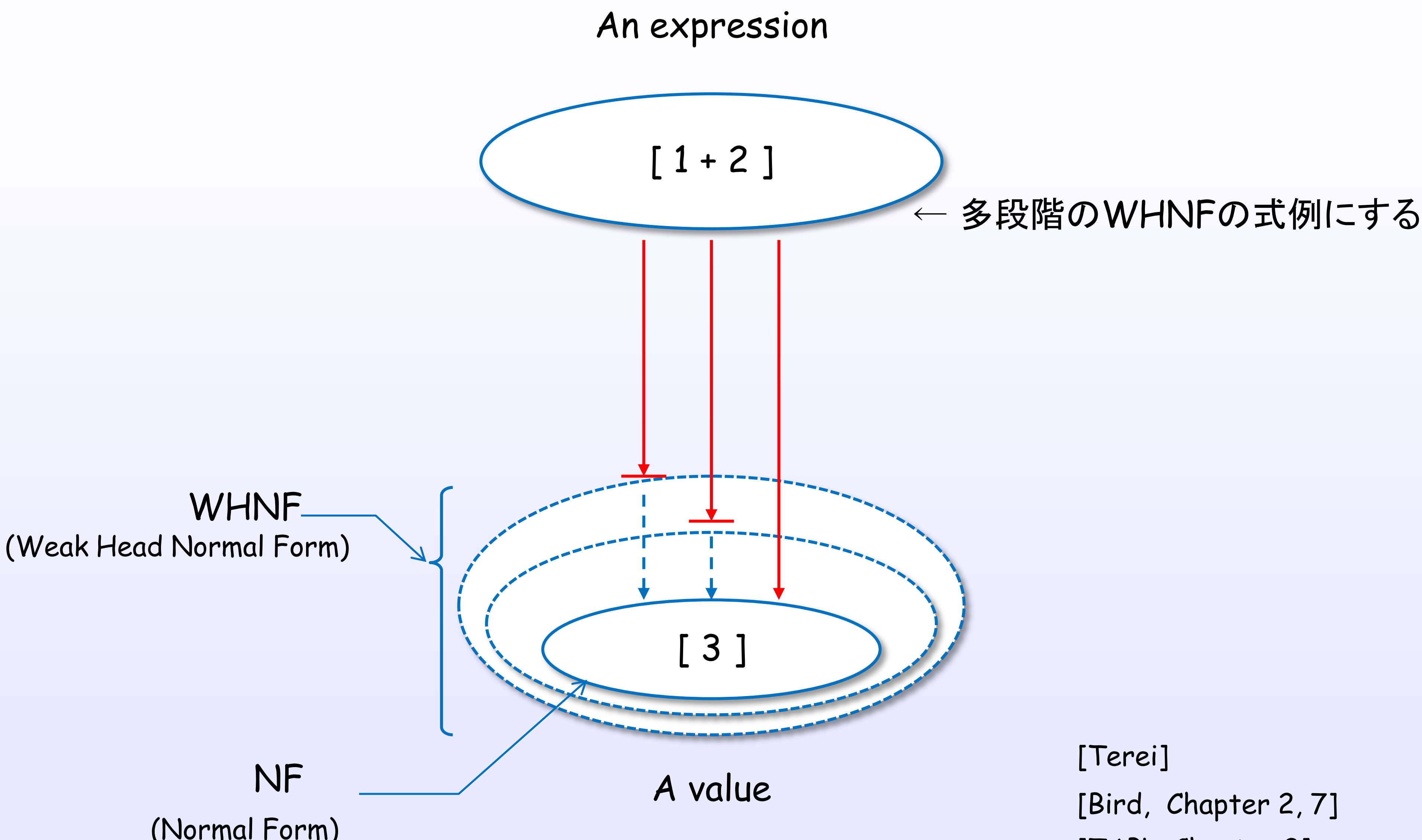
- Strict, Non-strict evaluation
- Eager, Lazy evaluation
- Call-by-value, Call-by-name, Call-by-need, …
- Innermost, Outermost
- Normal order, Applicative order
- …

[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

References : [1]

There are some evaluation levels



[Terei]

[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

References : [1]

2. Expressions

Expressions in Haskell

An expression denotes a value

An expression

[1 + 2]

[HR2010]

[Bird, Chapter 2]

References : [1]

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'
- if b then 1 else 0
- map f xs
- 7
- ¥x -> x + 1
- x : xs
- fun arg
- case x of _ -> 0
- do {x <- get; put x}
- (¥x -> x + 1) 3
- xs

categorizing

[HR2010]

[Bird, Chapter 2]

References : [1]

Expression categories in Haskell

WHNF(a value)、
unevaluated expression
との関連づけを
PAPもWHNFなので注意

lambda abstraction

- ¥x -> x + 1

let expression

- let x = 1 in x + y

variable

- xs

conditional

- if b then 1 else 0

case expression

- case x of _ -> 0

do expression

- do {x <- get; put x}

general constructor, literal and some forms

- 7
- [1, 2, 3]
- (1, 2)
- 'a'
- x : xs
- Just 5

function application

- take 5 xs
- (¥x -> x + 1) 3
- 1 + 2
- map f xs
- fun arg

[HR2010]
[Bird, Chapter 2

References : [1]

Specification is defined in Haskell 2010 Language Report

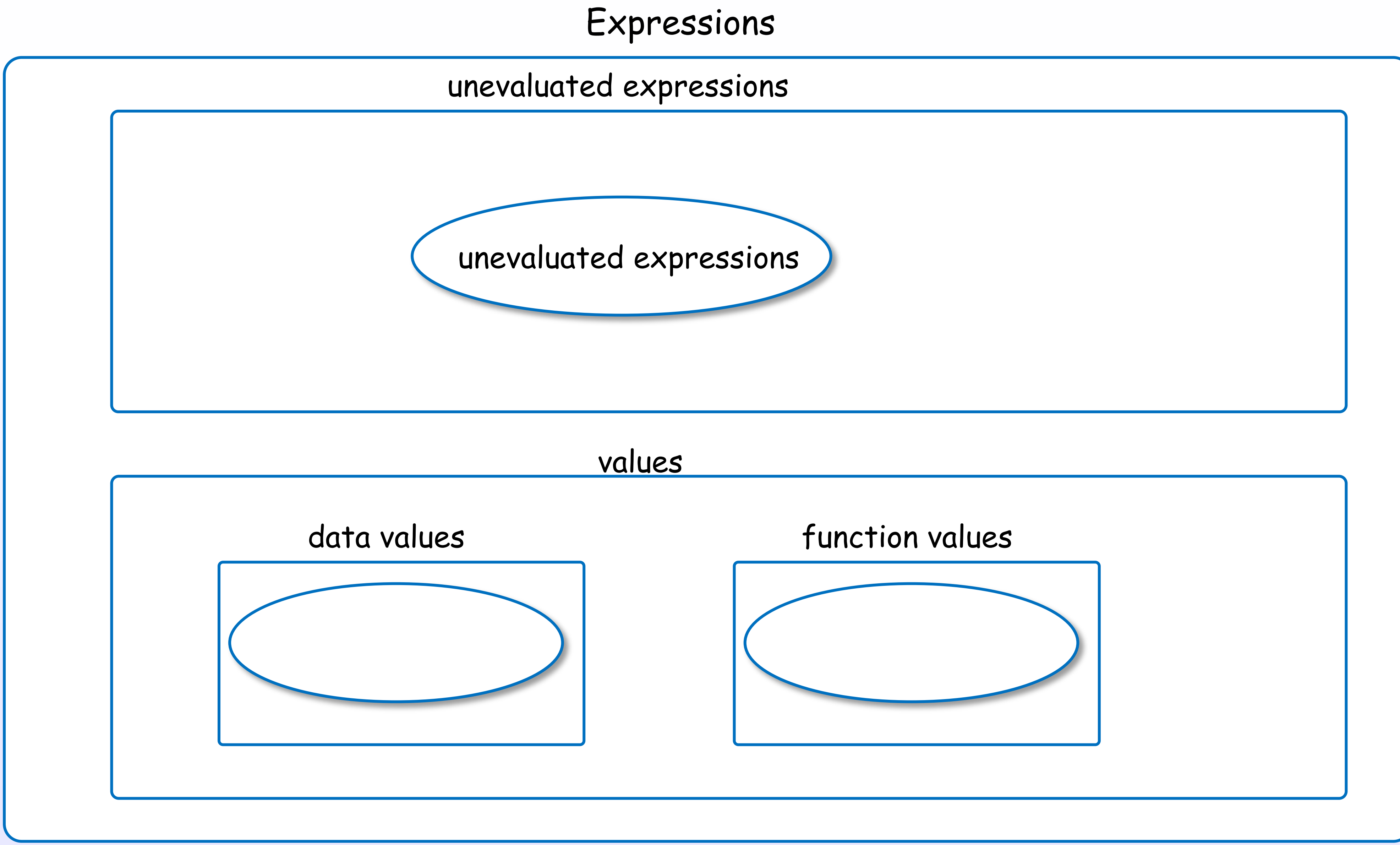
Haskell 2010 Language Report, Chapter 3 Expressions [1]

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 : [1]

Classification by value

A value or an unevaluated expression



値か否か。値は2種。

[STG]

References : [1]

実例との対応付け

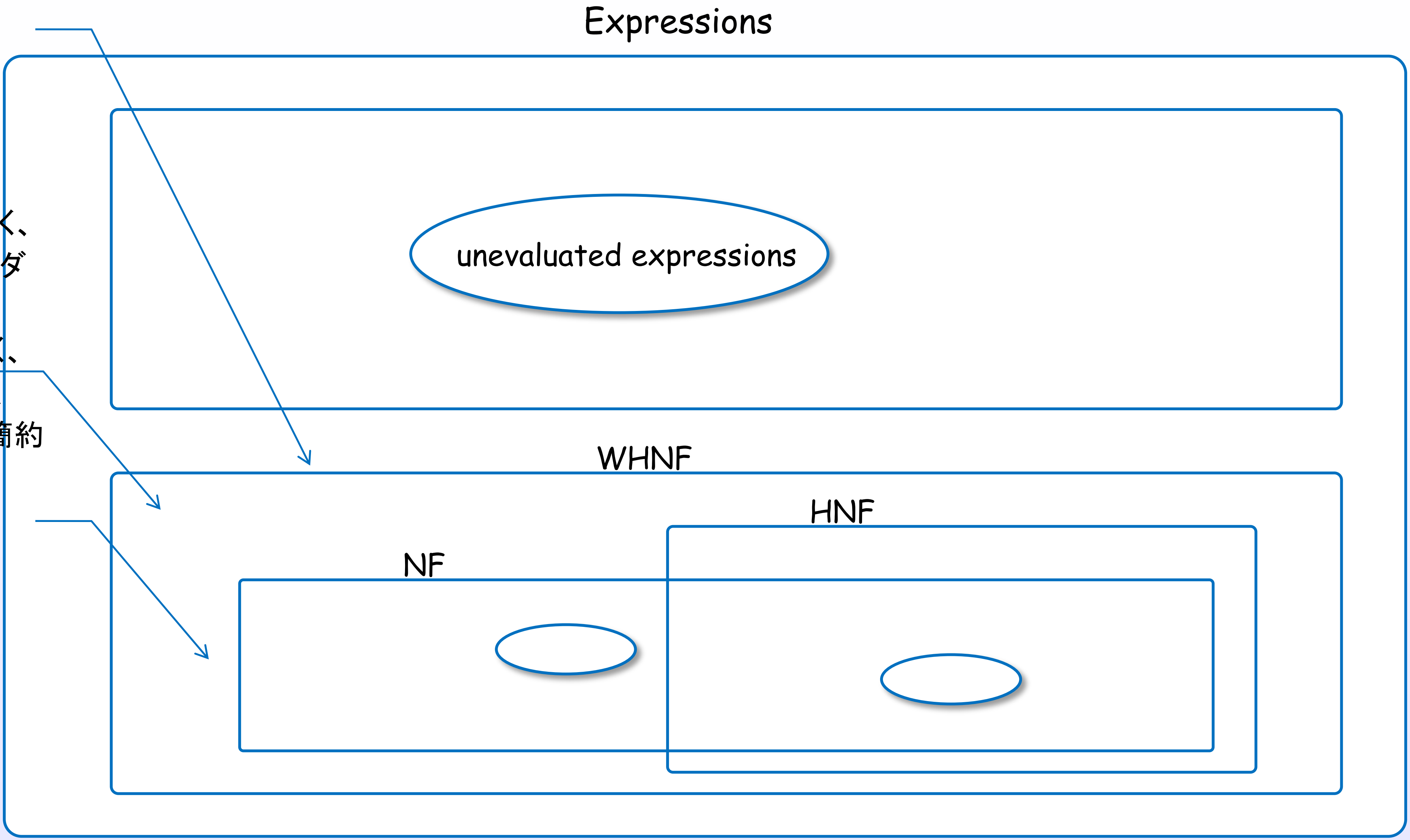
[STG]

References : [1]

Classification by form

evaluation level

でなく、
ラムダ
でなく、
合は、
も簡約
い。
）



値には、評価レベルがある。

[STG]

References : [1]

実例との対応付け

[STG]

References : [1]

2. Expressions

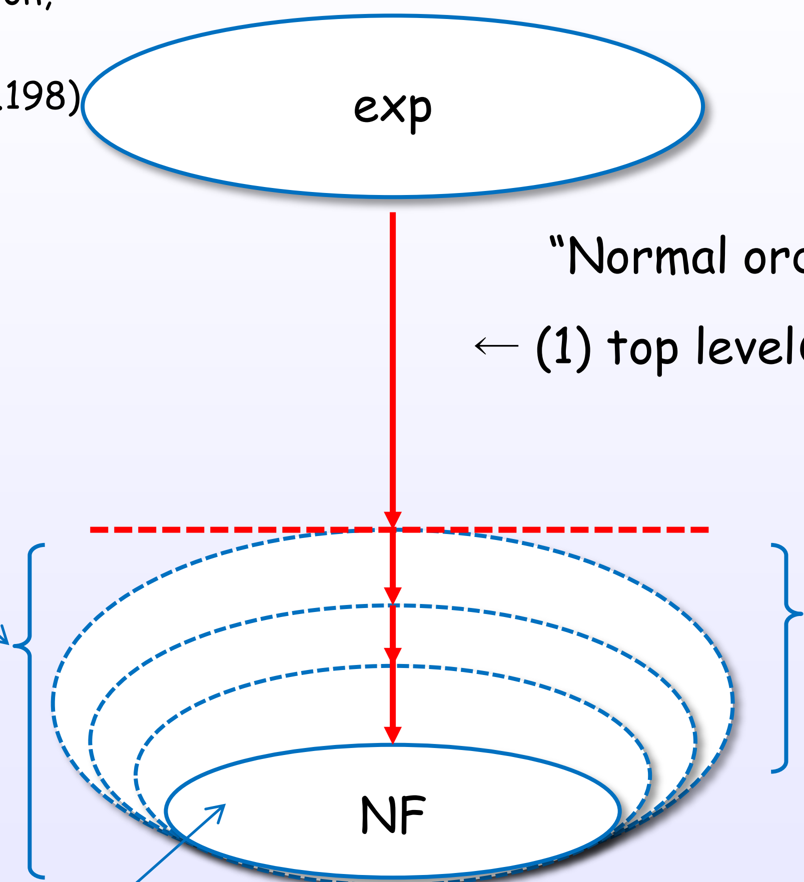
WHNF

evaluation step (GHC)

Our reduction order is therefore to reduce the top-level redex until weak head normal form is reached. (p.198)

An expression

We pursue normal order reduction, but stop at WHNF rather than proceeding all the way to NF. (p.198)



“Normal order reduction of top-level rede

← (1) top levelのreduction

[slpj-book-1987]

WHNF
(Weak Head Normal Form)

← (2) innerのreduction

“Normal order reduction of

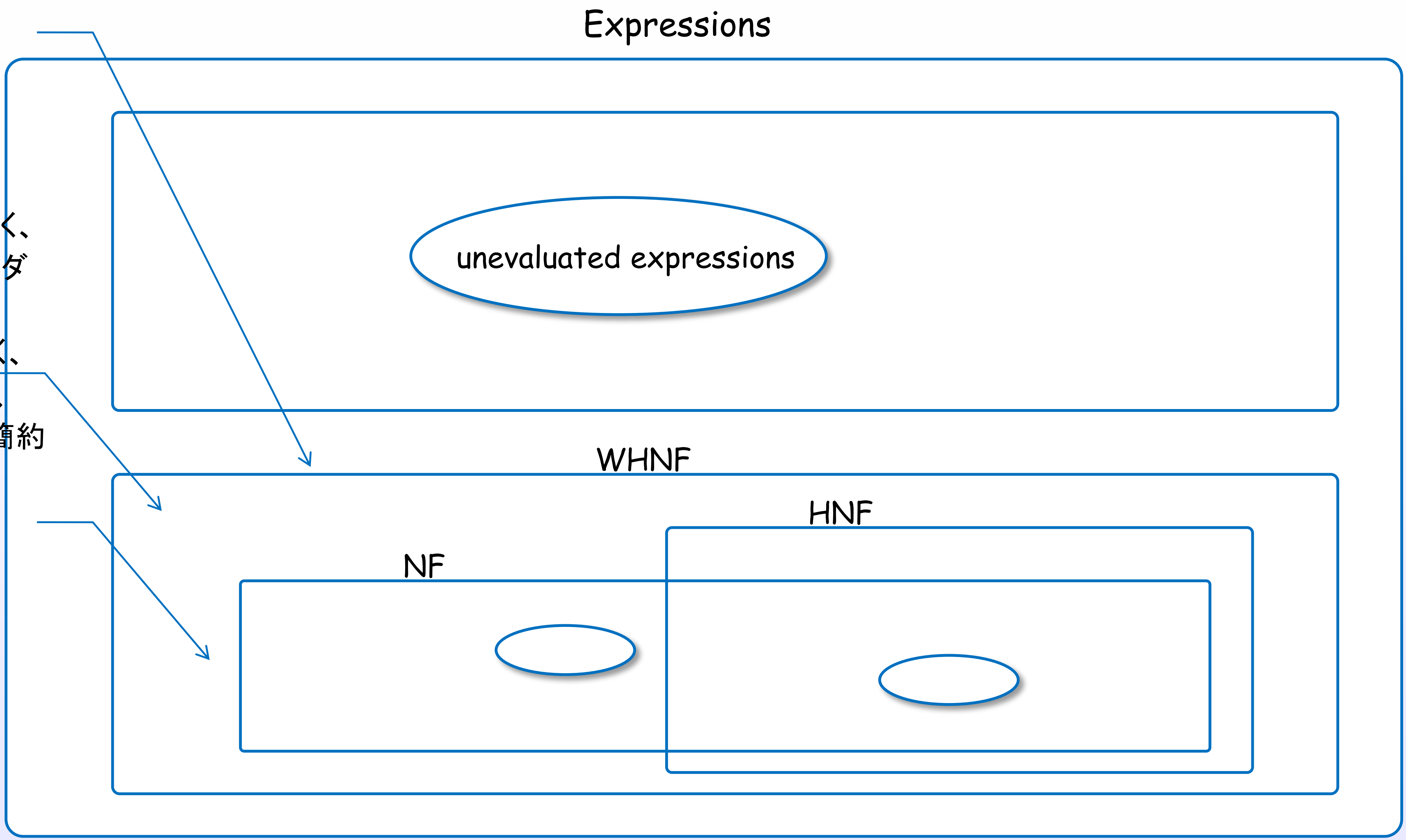
NF
(Normal Form)
no more evaluation (reduction)

A value

[Terei] [stephen]
[Bird, Chapter 2, 7]
[TAPL, Chapter 3]
[Haskell/Laziness]

References : [1]

Evaluation level



値には、評価レベルがある。

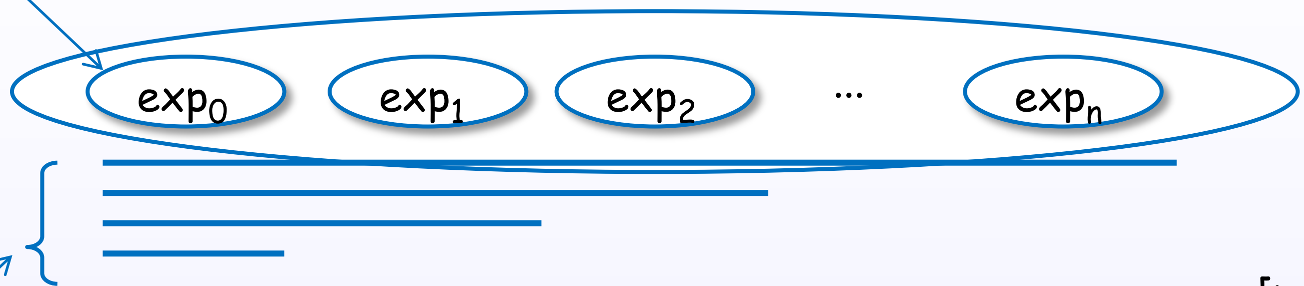
[STG]

References : [1]

WHNF

ダ抽象、ビルトイン

an expression



[parconc, Ch.2]

more

An expression has no top level redex, if it is in WHNF.

[slpj-book-1987]

[stephen]

These are in weak head normal form,
but not in normal form, since they contain inner redex. (p.198)

[hack.hands]

[Terei]

[Bird, Chapter 2, 7]

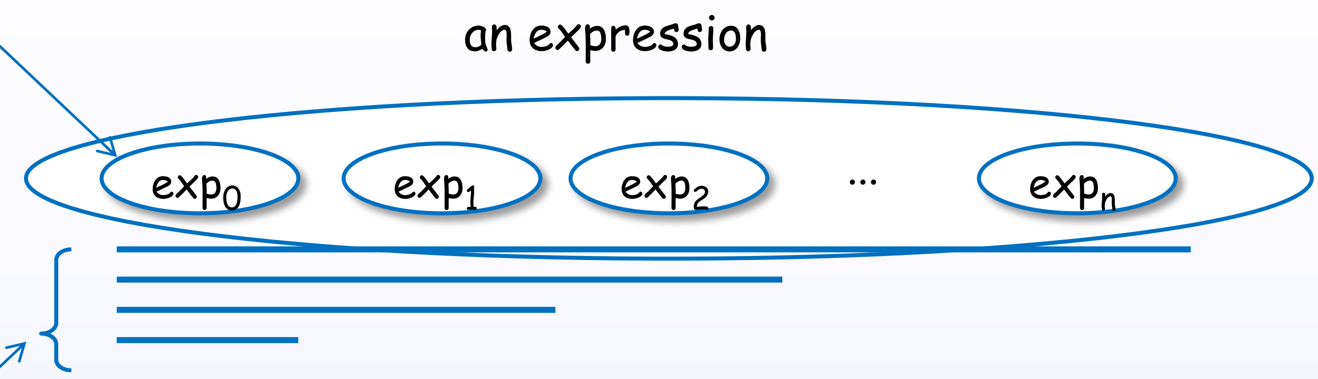
[TAPL, Chapter 3]

[Terei]

References : [1]

Examples of WHNF

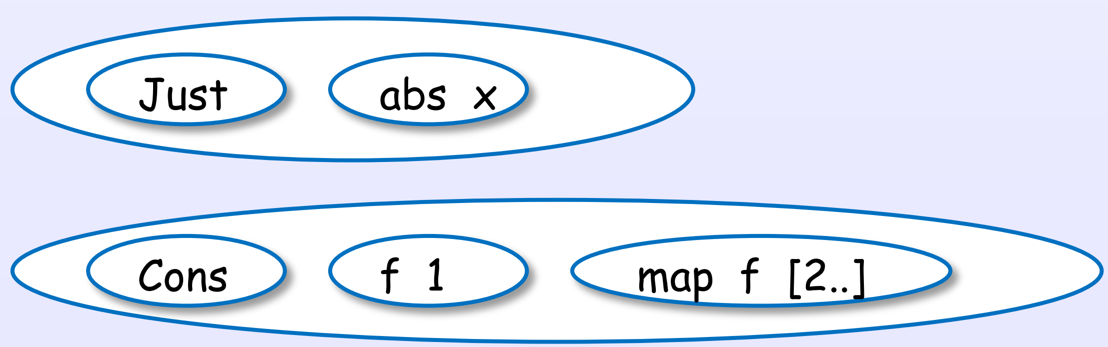
ダ抽象、ビルトイン



Just (abs x)

more

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



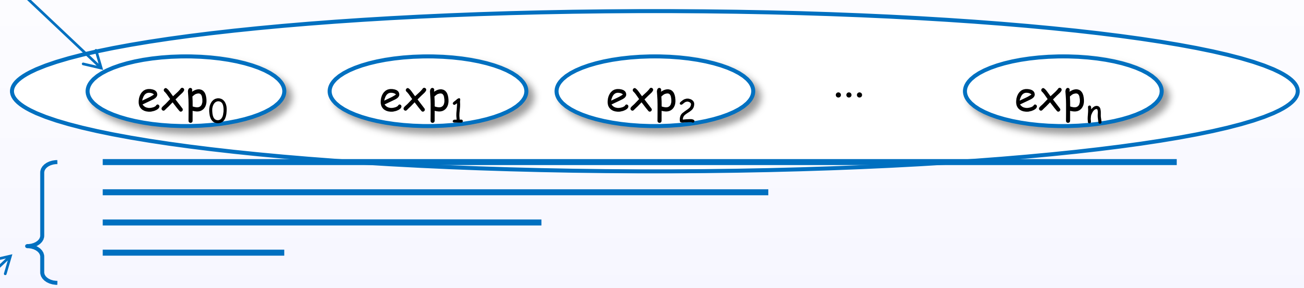
[slpj-book-1987]

References : [1]

HNF

ダ抽象、ビルトイン

an expression



内側(body)が、箇

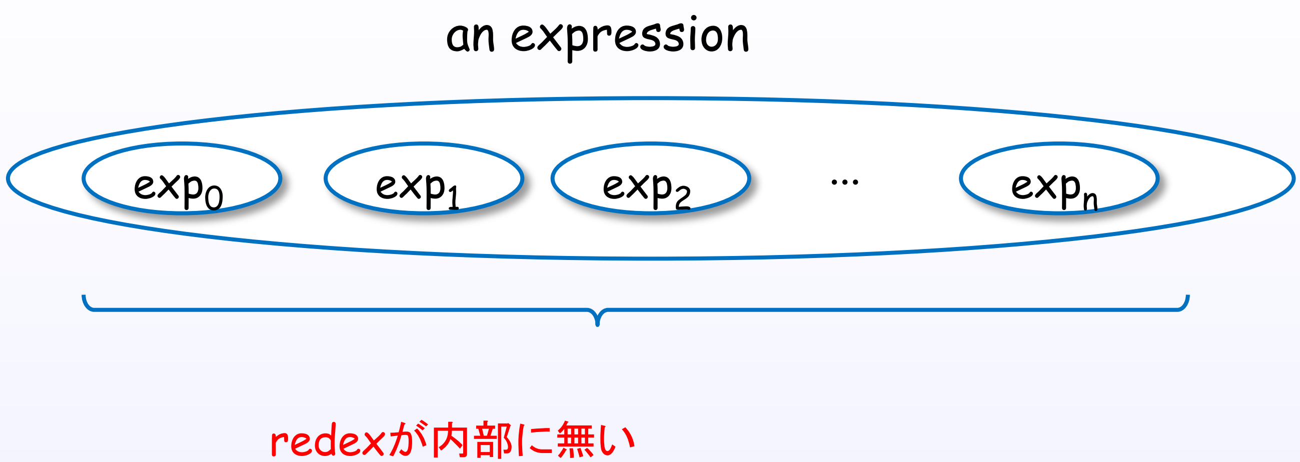
more

[slpj-book-1987]

[Terei]

References : [1]

NF



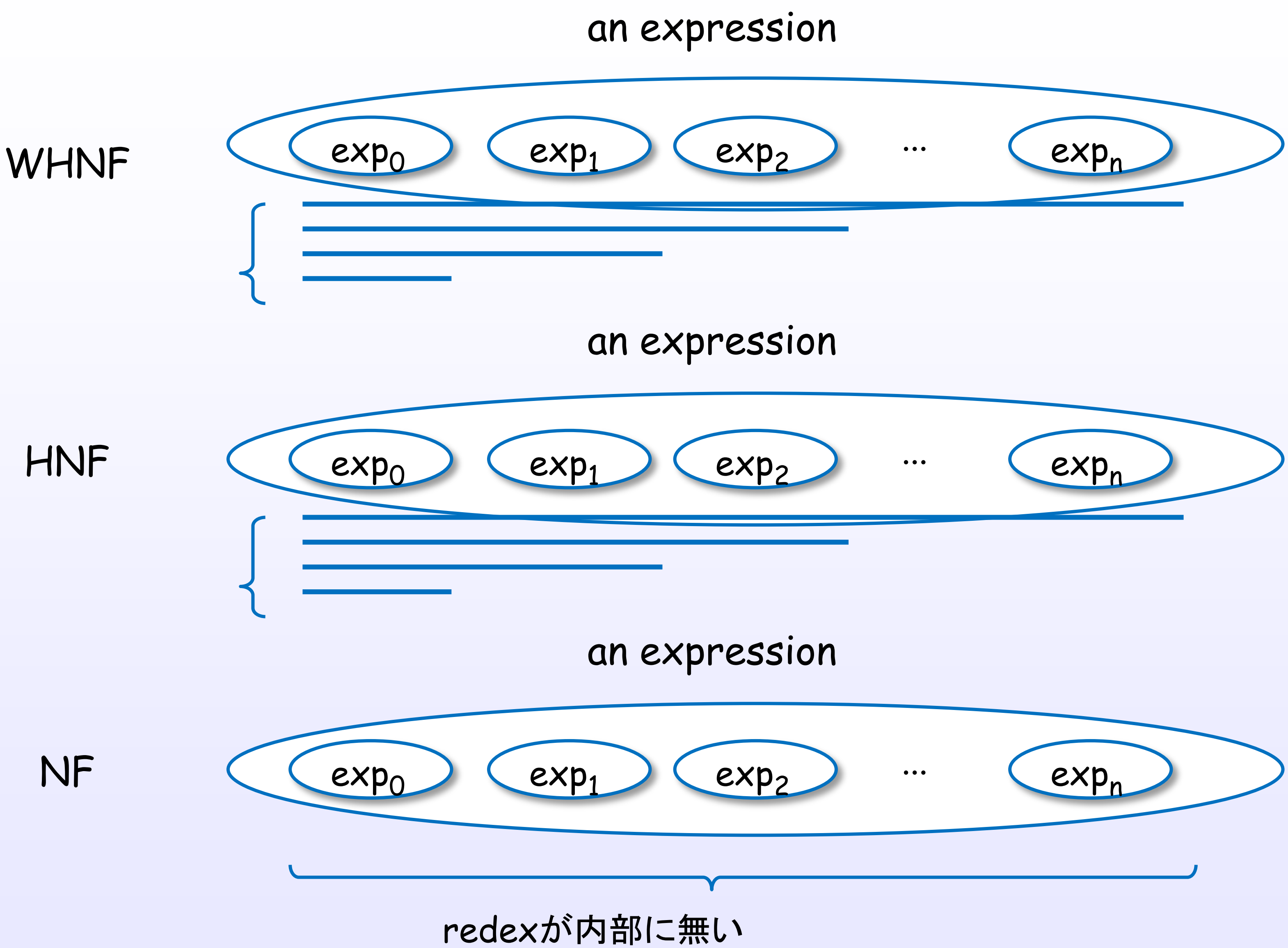
[slpj-book-1987]

[Terei]
[Bird, Chapter 2, 7]
[TAPL, Chapter 3]

[Terei]

References : [1]

WHNF, HNF, NF



[slpj-book-1987]

References : [1]

definition of WHNF and HNF

The implementation of functional programming languages [19]

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 confus
some discussion. The content of
since for most purposes head no
form. Nevertheless, we will stick 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 $p \leq m$.

[slpj-book-1987]

References : [1]

3. Internal representation of expressions

Constructor

Constructor

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

References : [1]

Algebraic data type and value

data文で宣言する代数的データ型とその値

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

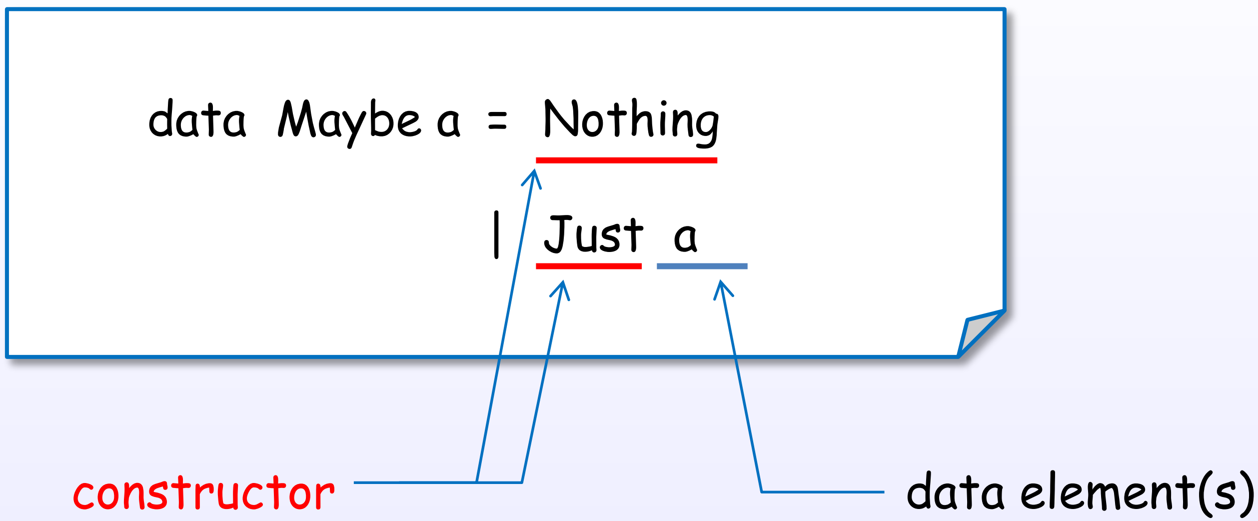
Algebraic Data Type

Data Values

References : [1]

Constructors are defined by data declaration

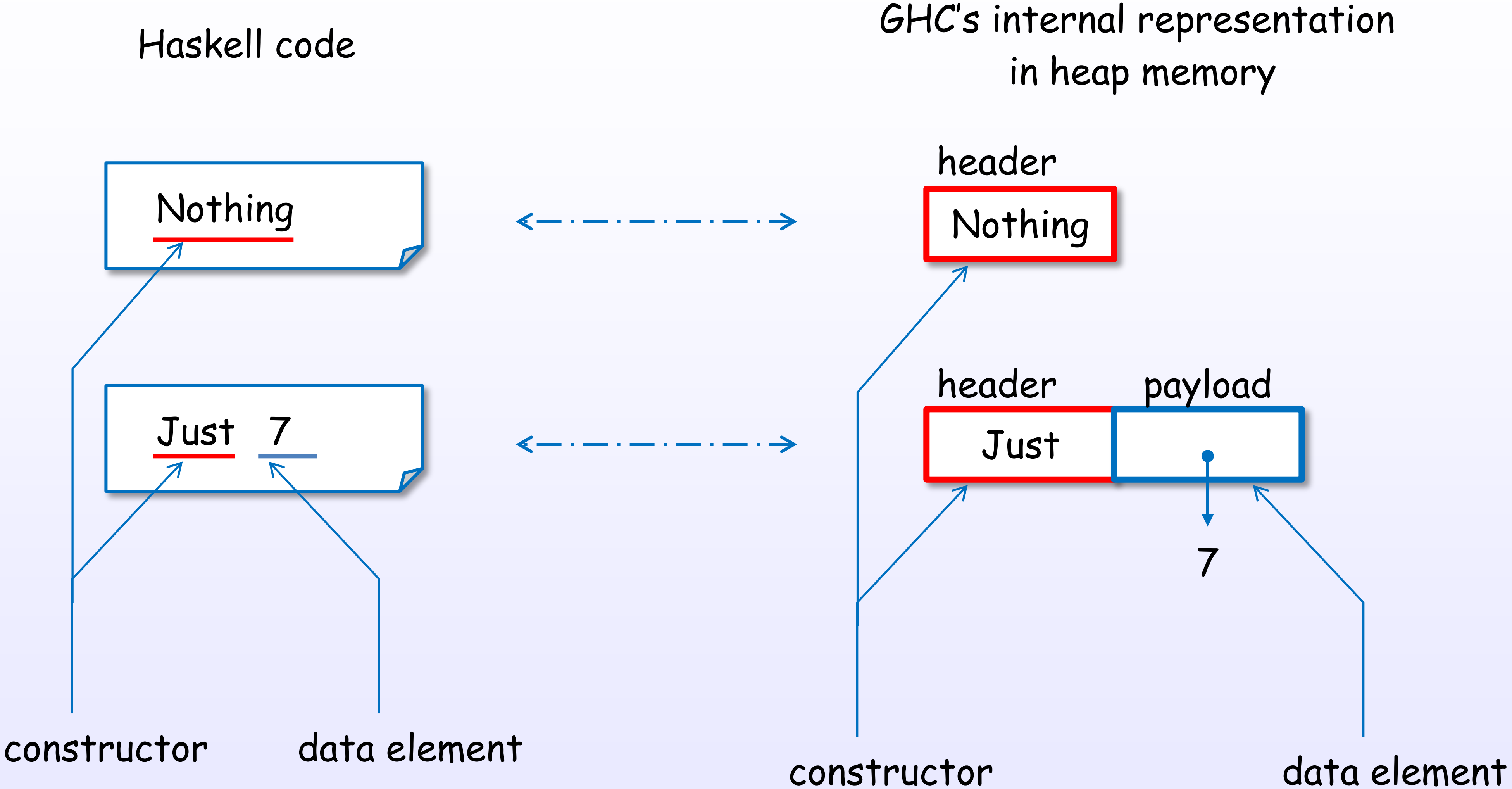
Constructorはdata文で宣言する代数的データ値



References : [1]

Internal representation of Constructors for data values

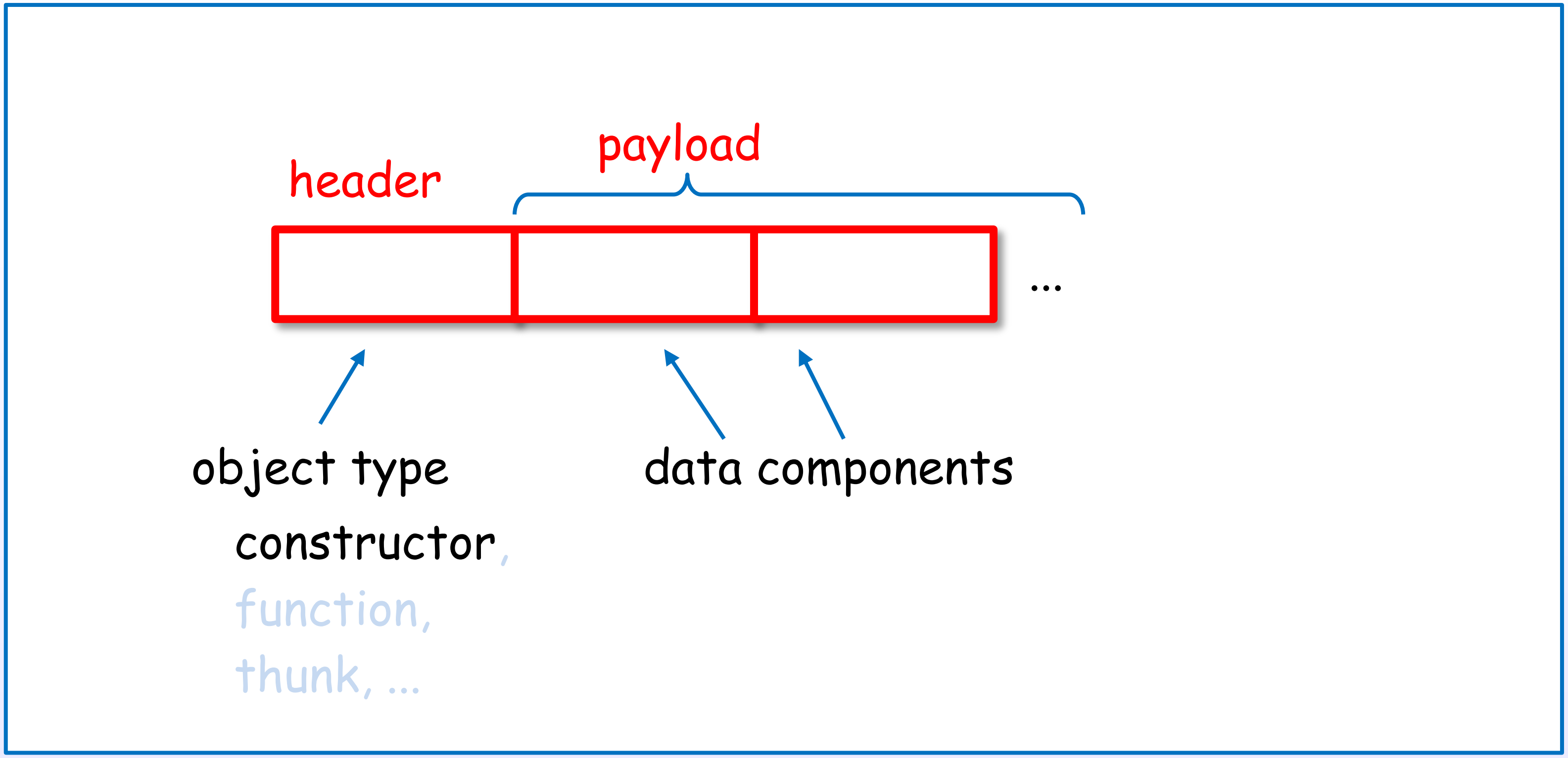
↑ data values



[STG]

References : [1]

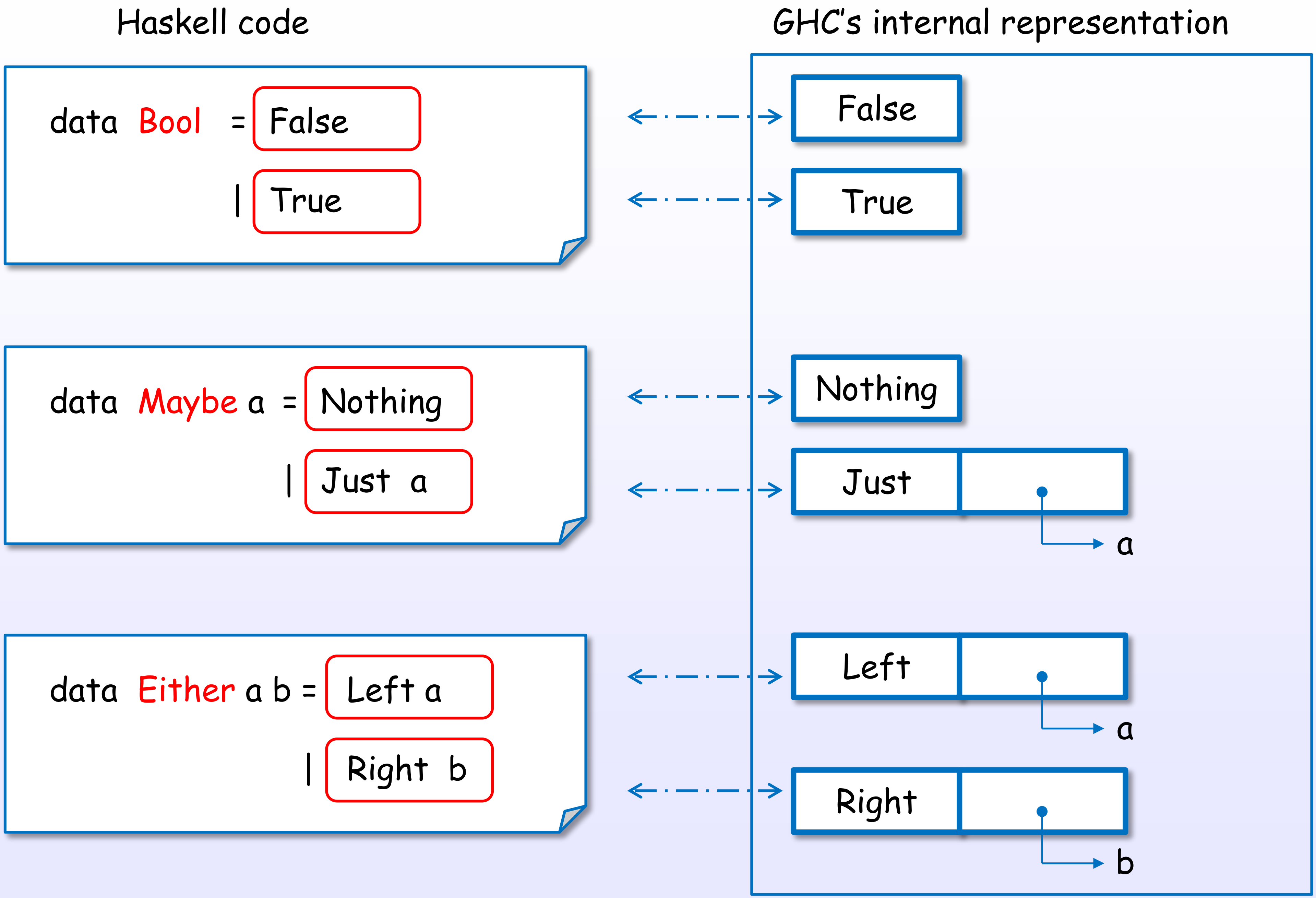
Constructors are represented uniformly



in heap memory, stack, registers or static memory

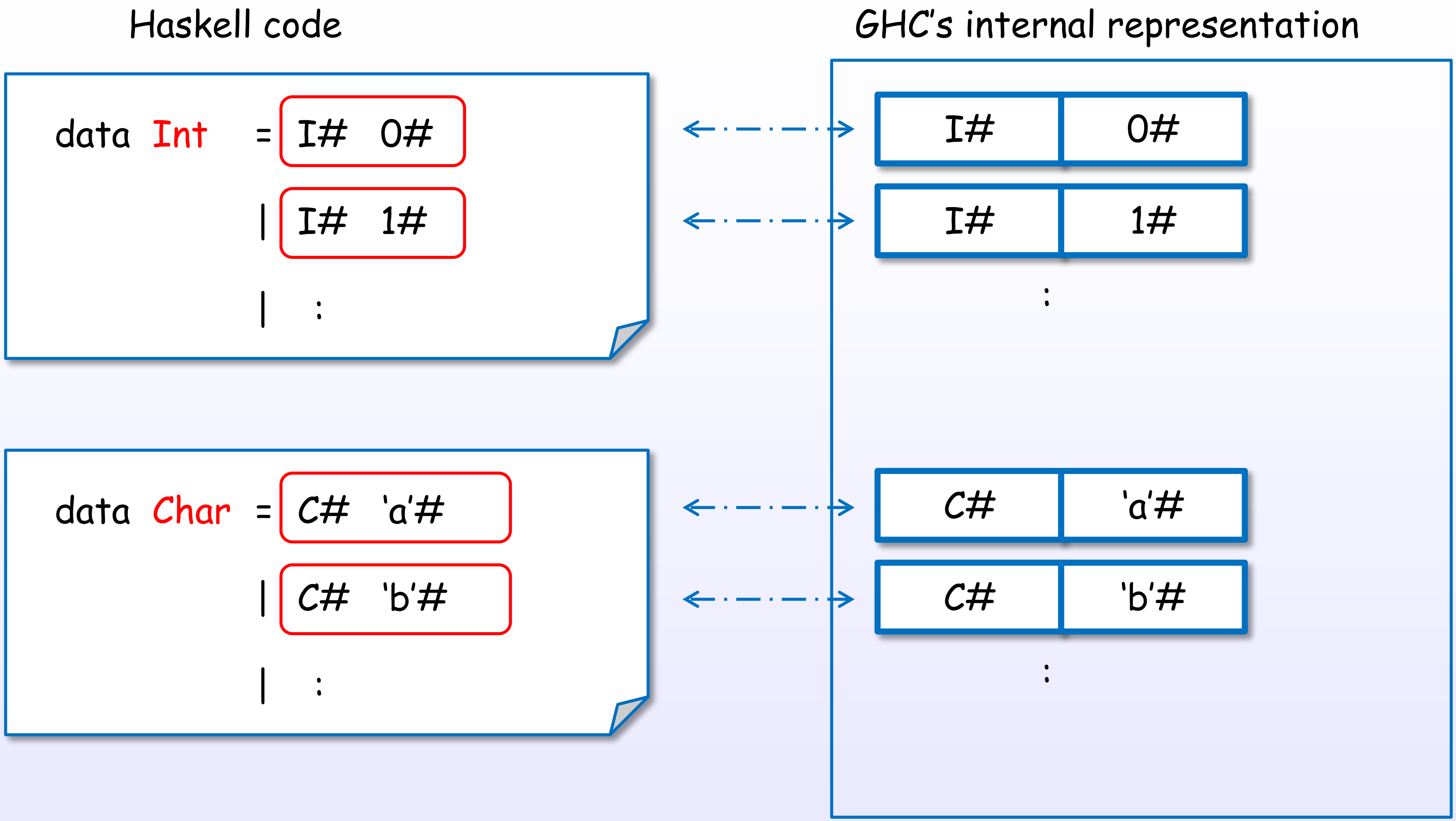
References : [1]

Representation of various constructors



References : [1]

Primitive data types are also represented with constructor



[Terei]

References : [1]

Lists are 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

References : [1]

Lists are 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

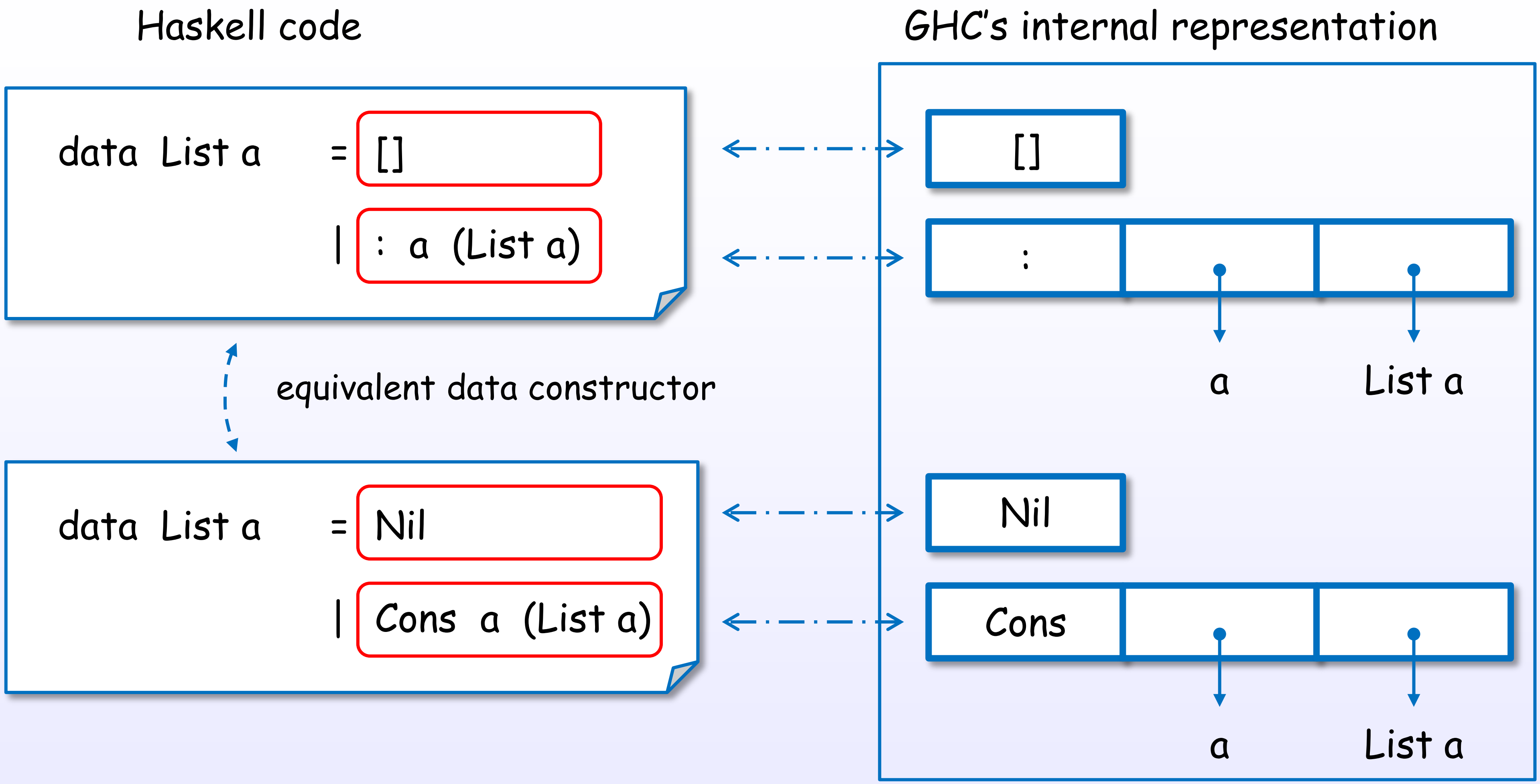
** pseudo code*

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

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

References : [1]

Lists are also represented with constructor



References : [1]

Lists are also represented with constructor

Haskell code

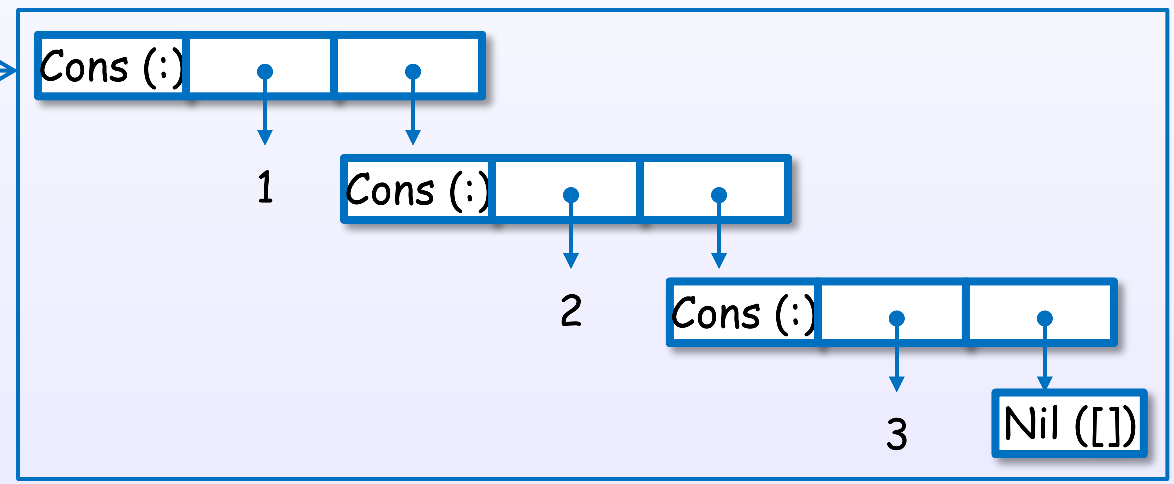
```
[ 1, 2, 3 ]
```

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

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

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

GHC's internal representation



References : [1]

Tuples are 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
```

References : [1]

Tuples are also represented with constructor

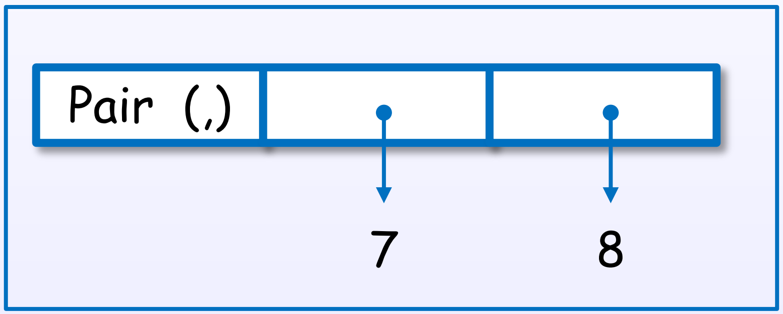
Haskell code

```
( 7, 8 )
```

```
(,) 7 8
```

```
Pair  7  8
```

GHC's internal representation

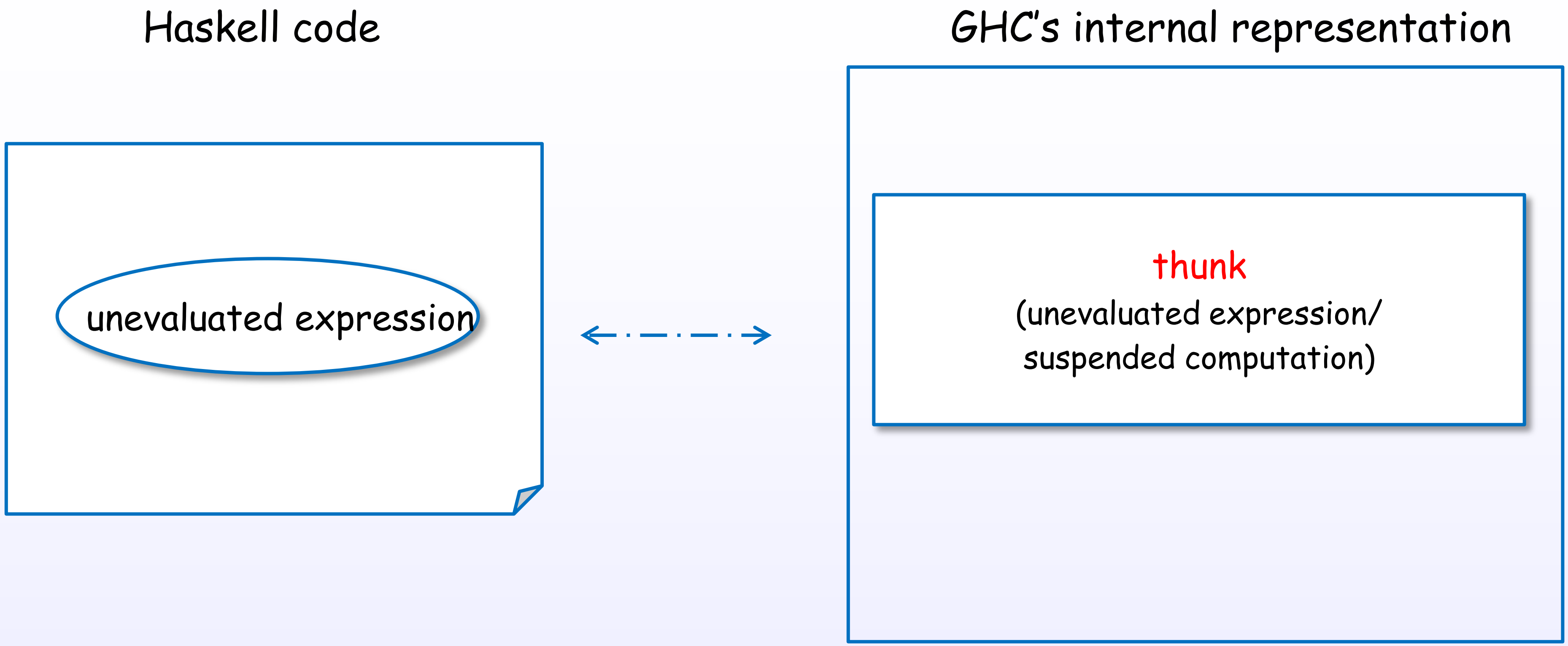


References : [1]

3. Internal representation of expressions

Thunk

Thunk



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

[parconc, Ch.2]

[hack.hands]

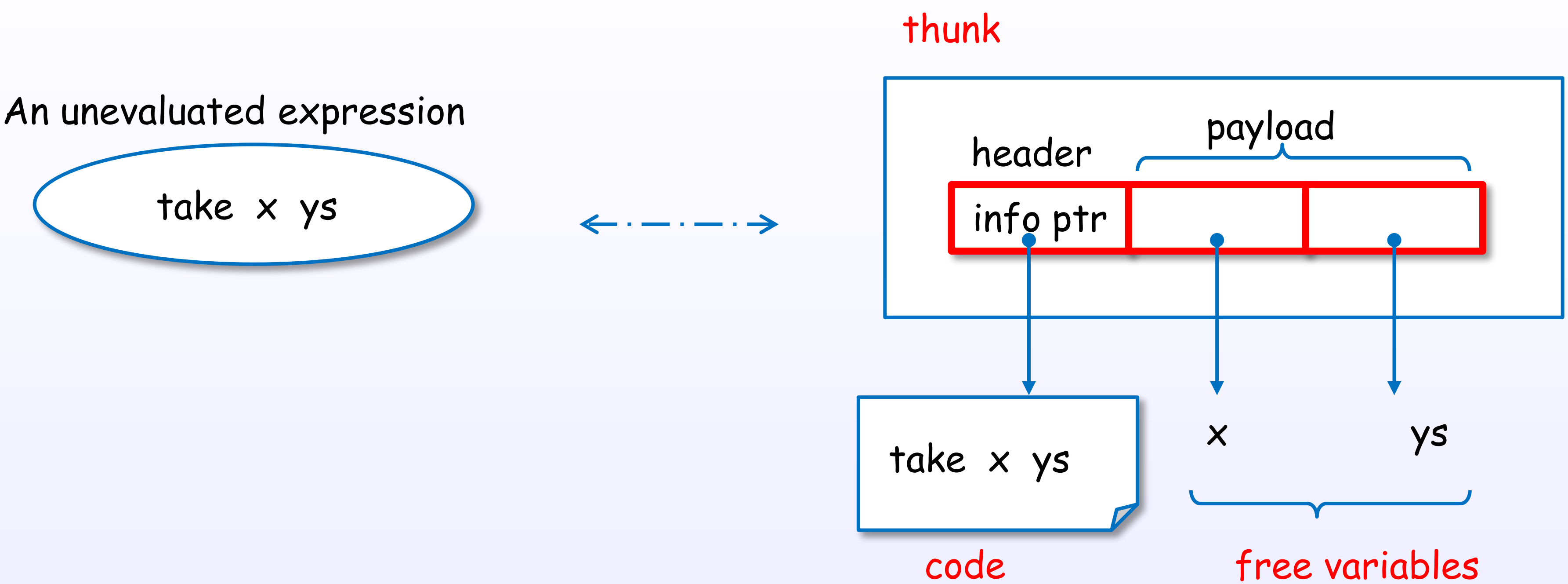
[Haskell/Laziness]

References : [1]

Internal representation of thunk

Haskell code

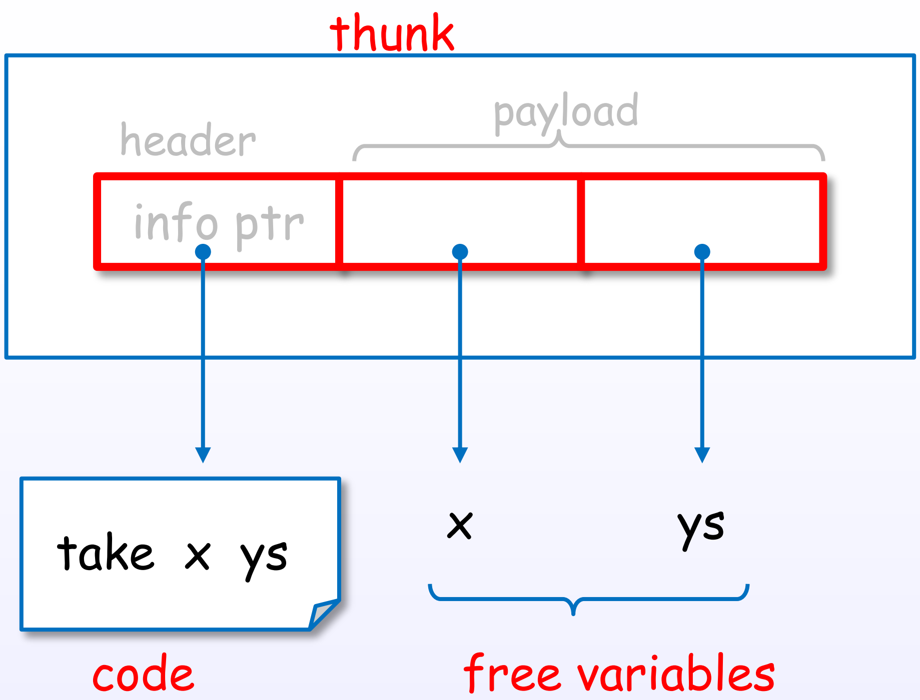
GHC's internal representation



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

References : [1]

A thunk is a package of code and free variables

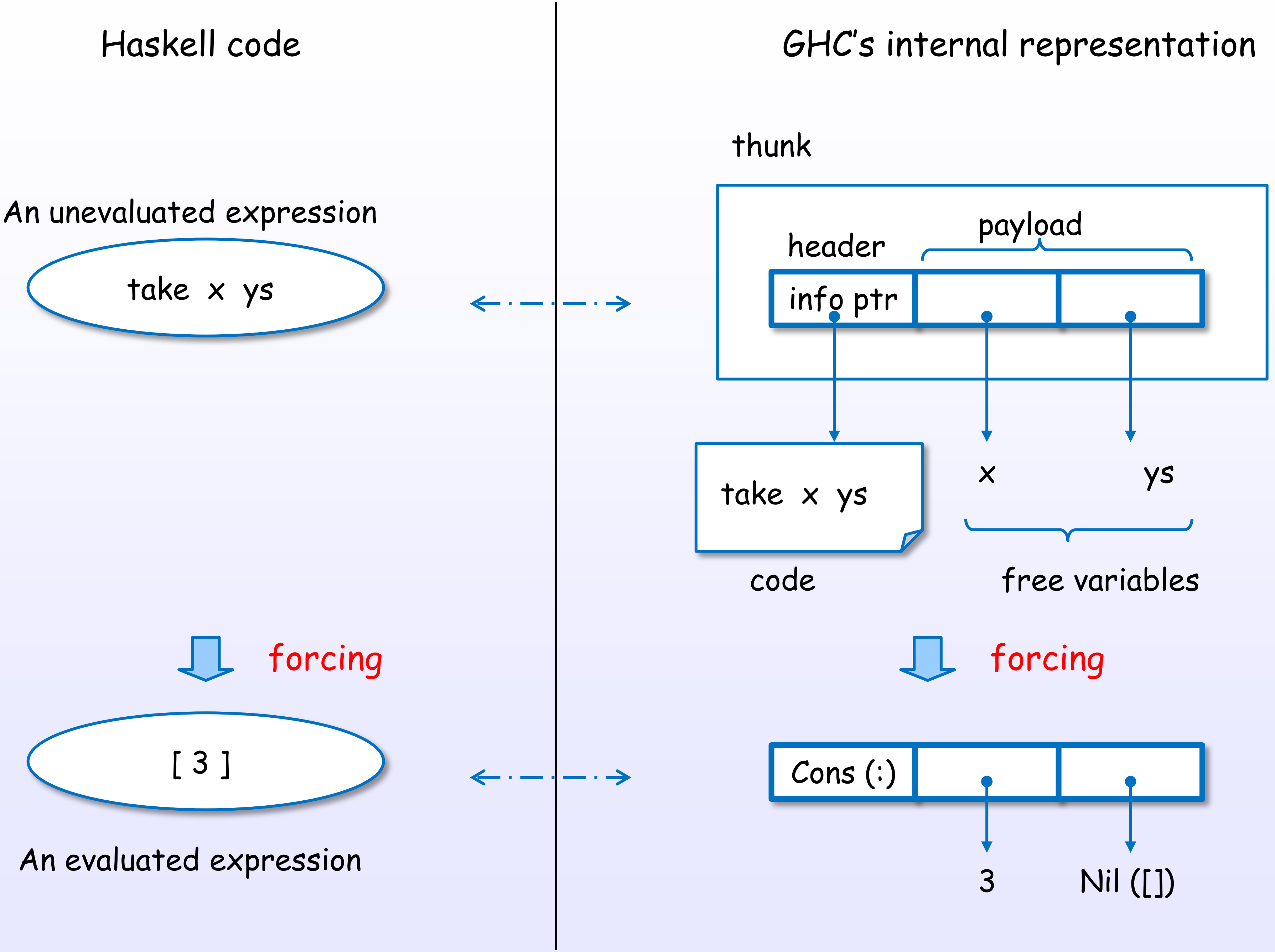


A thunk is a package of code + free variables.

[CIS194]

References : [1]

A thunk is evaluated by forcing request

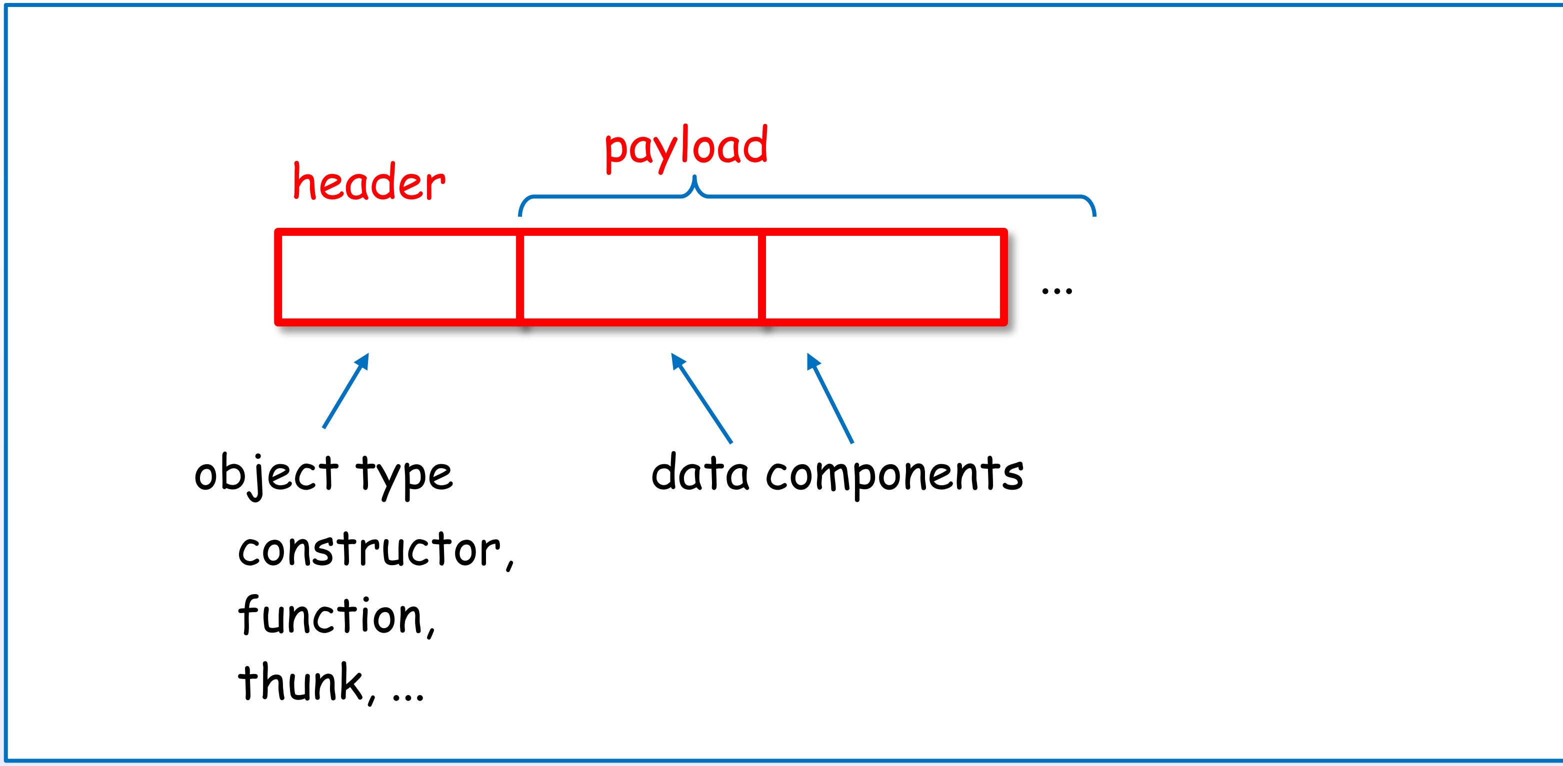


References : [1]

3. Internal representation of expressions

Uniform representation

Every object is represented uniformly

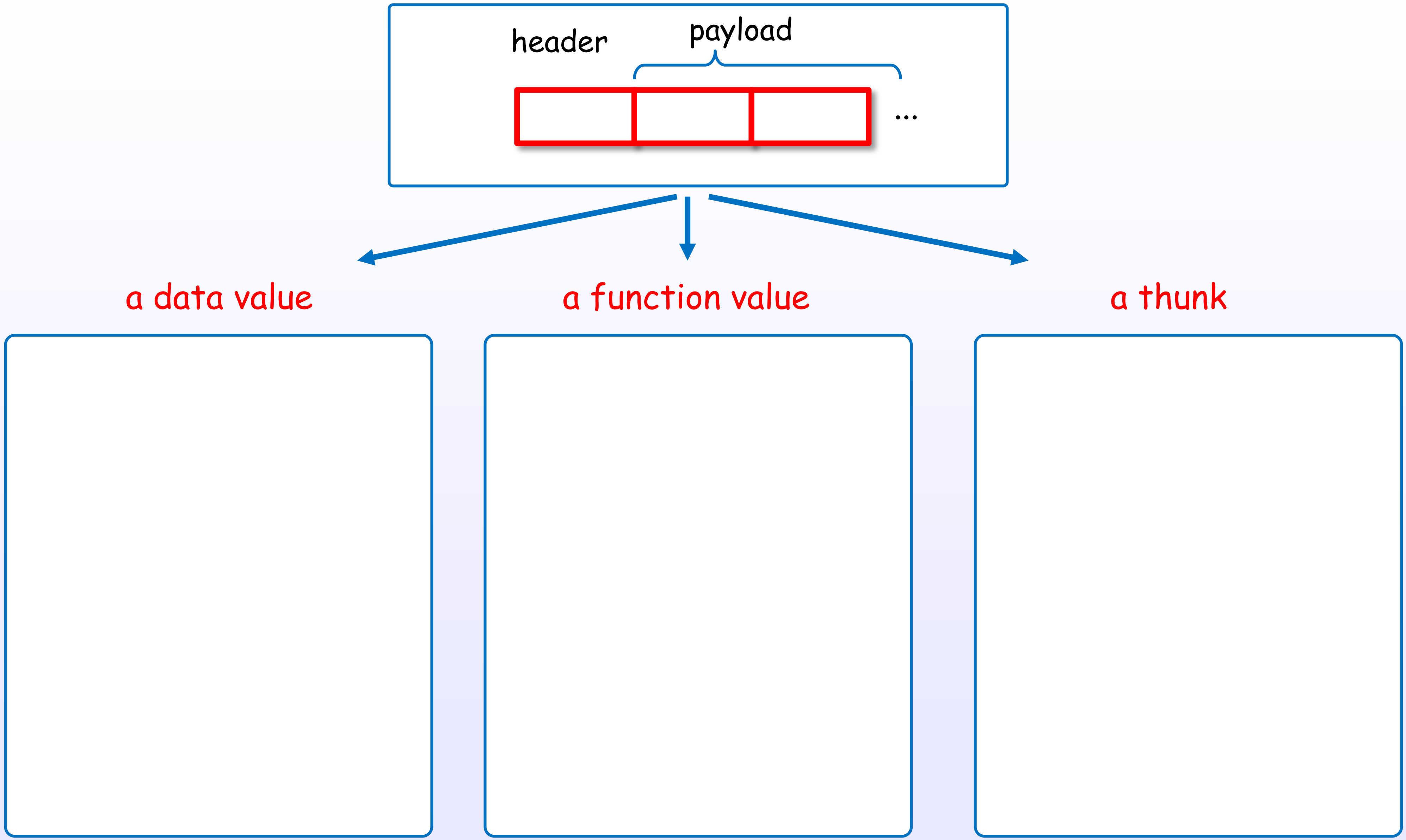


in heap memory, stack, registers or static memory

[STG]

References : [1]

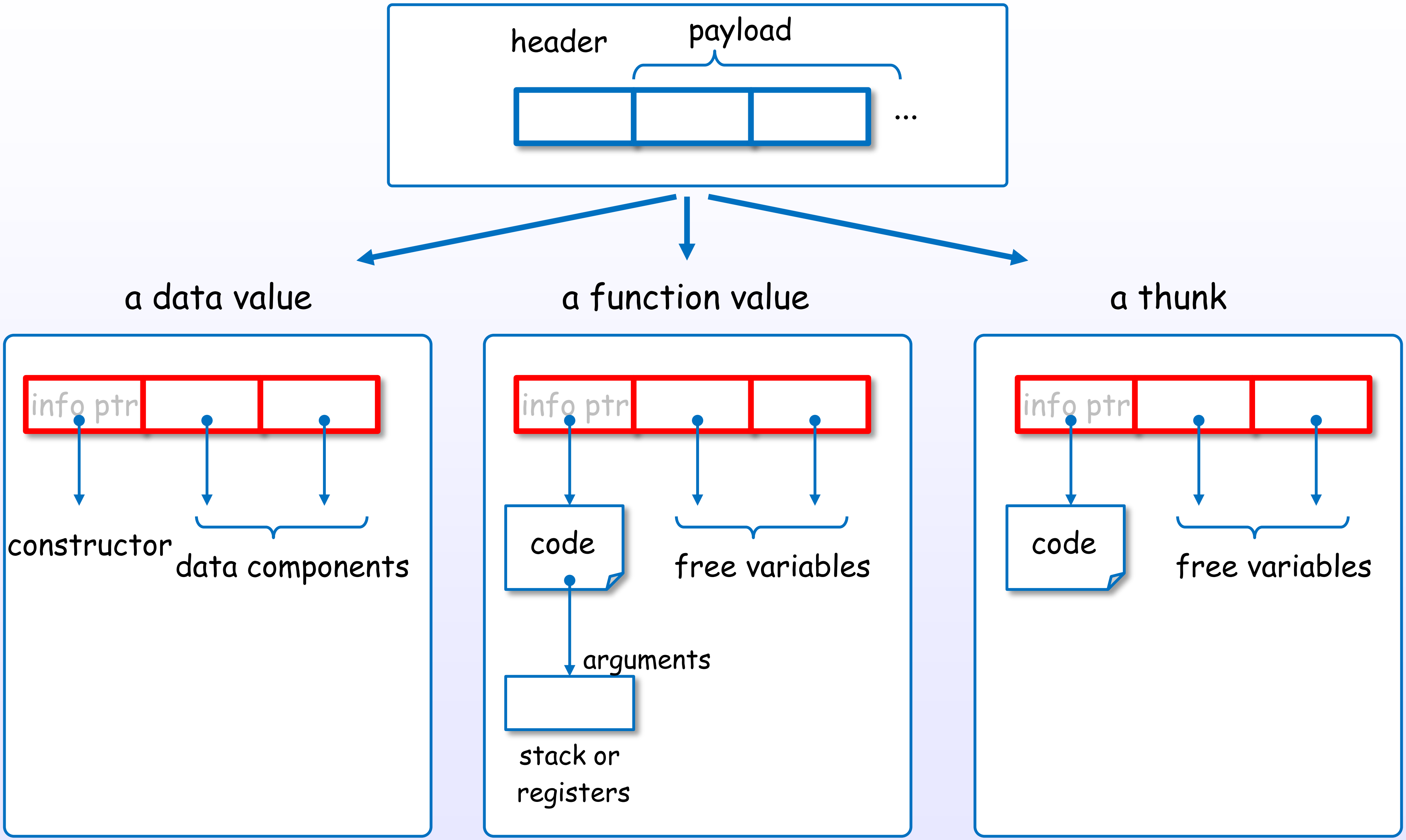
Every object is represented uniformly



[STG]

References : [1]

Every object is represented uniformly

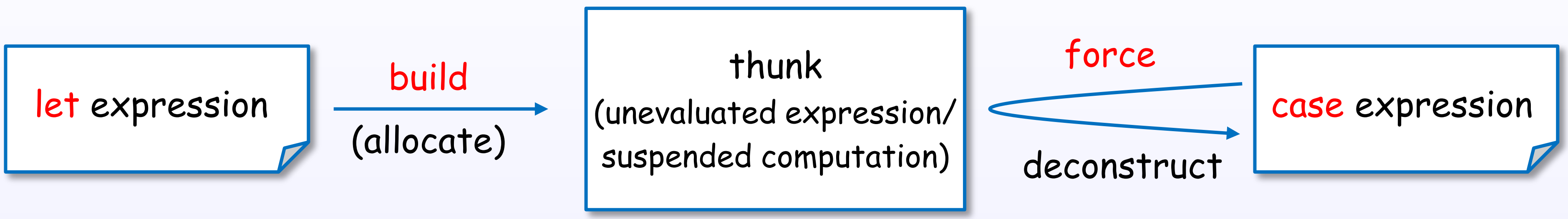


いずれも、広義の、"closure" (= code + environment(free variables))

References : [1]

let, case expression

let/case expressions and thunk

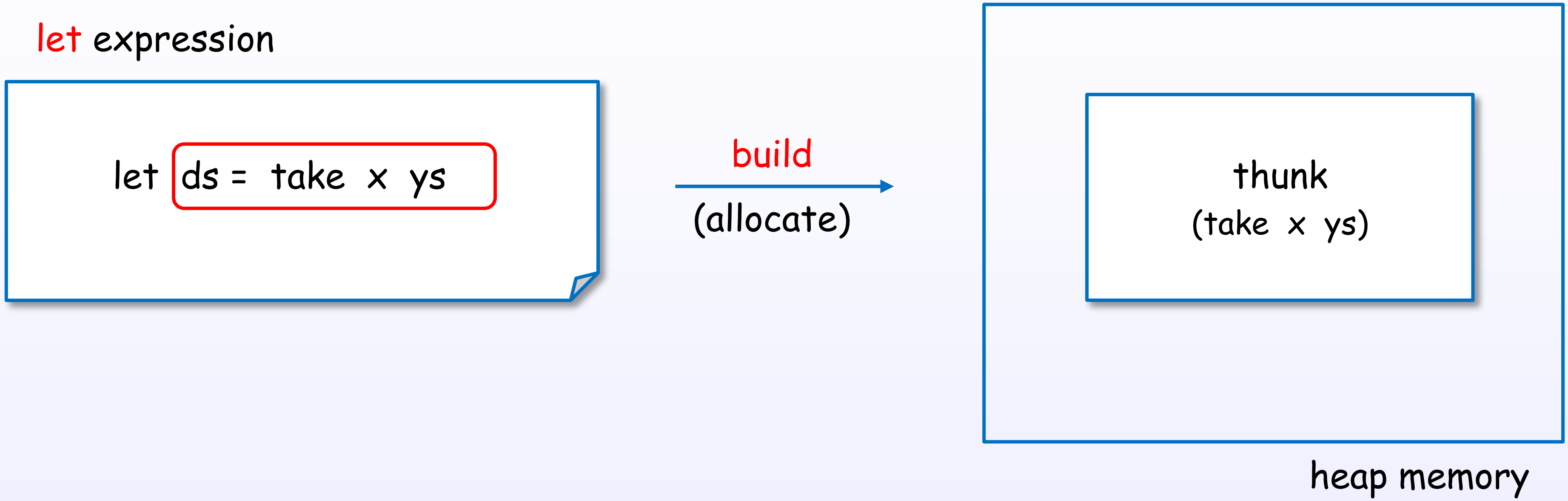


A let expression may build a thunk.
A case expression forces and deconstructs the thunk.

[STG]

References : [1]

A let expression builds a thunk



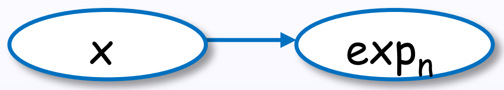
References : [1]

various representation level

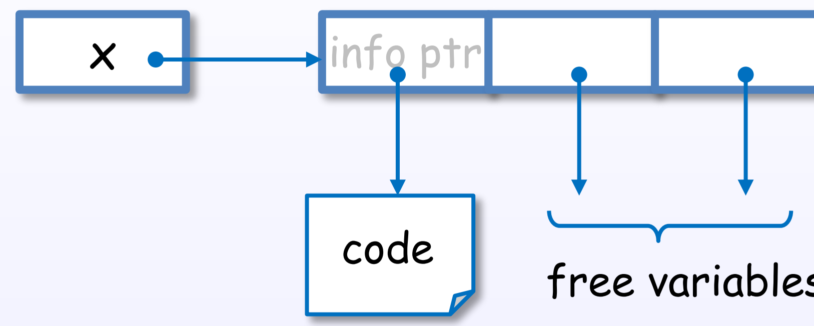
Haskell code

let x = exp_n

Expression graph

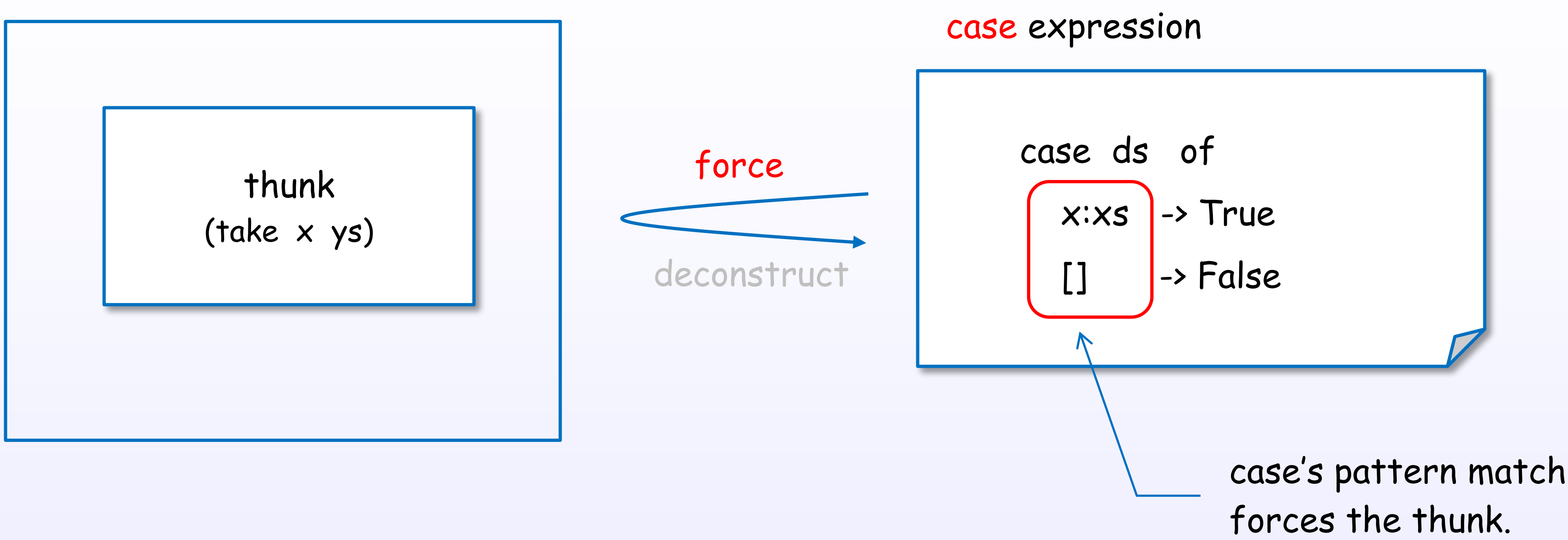


GHC internal representation



References : [1]

A case expression forces a thunk

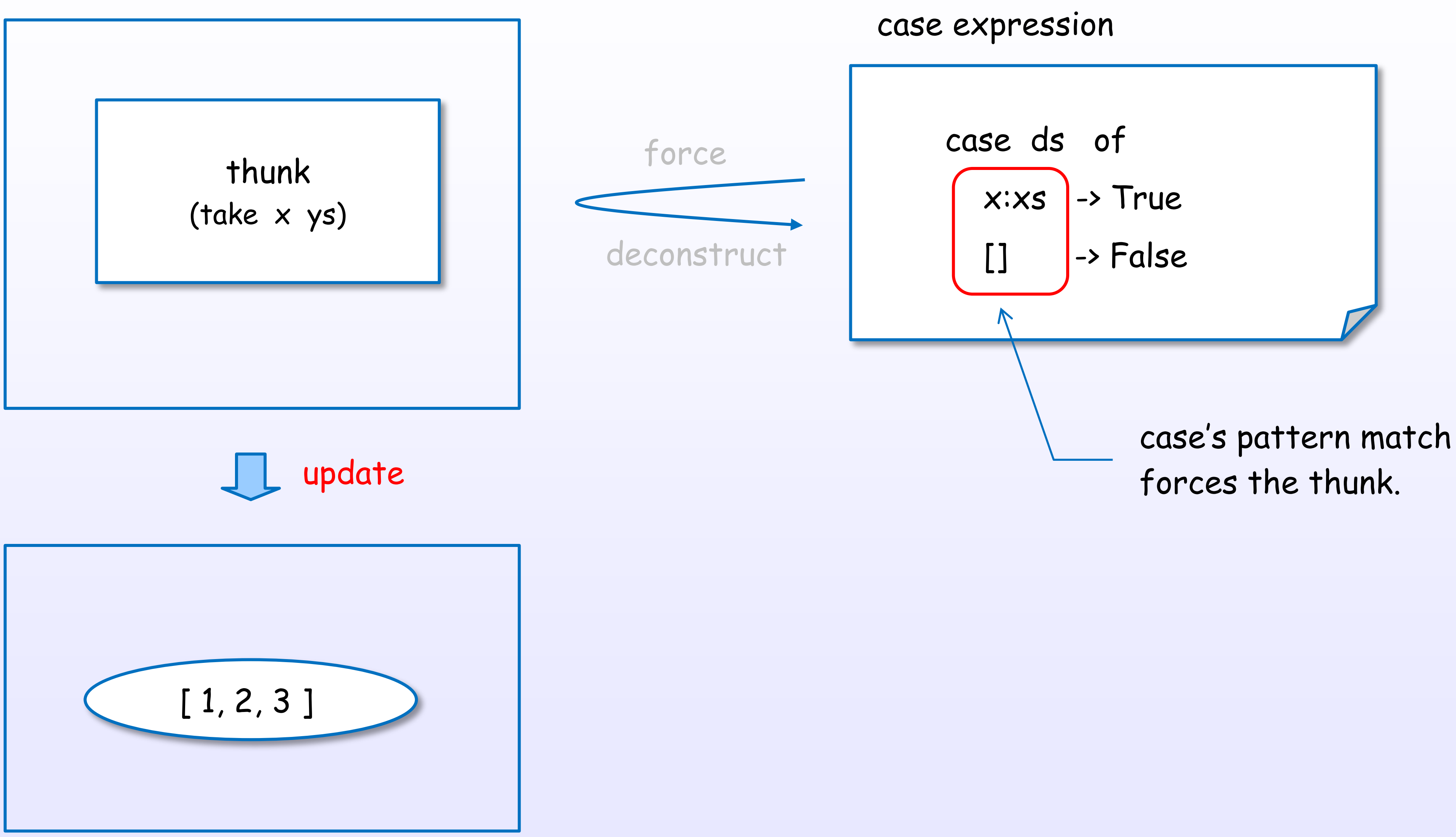


[Terei] [CIS194]

[STG]

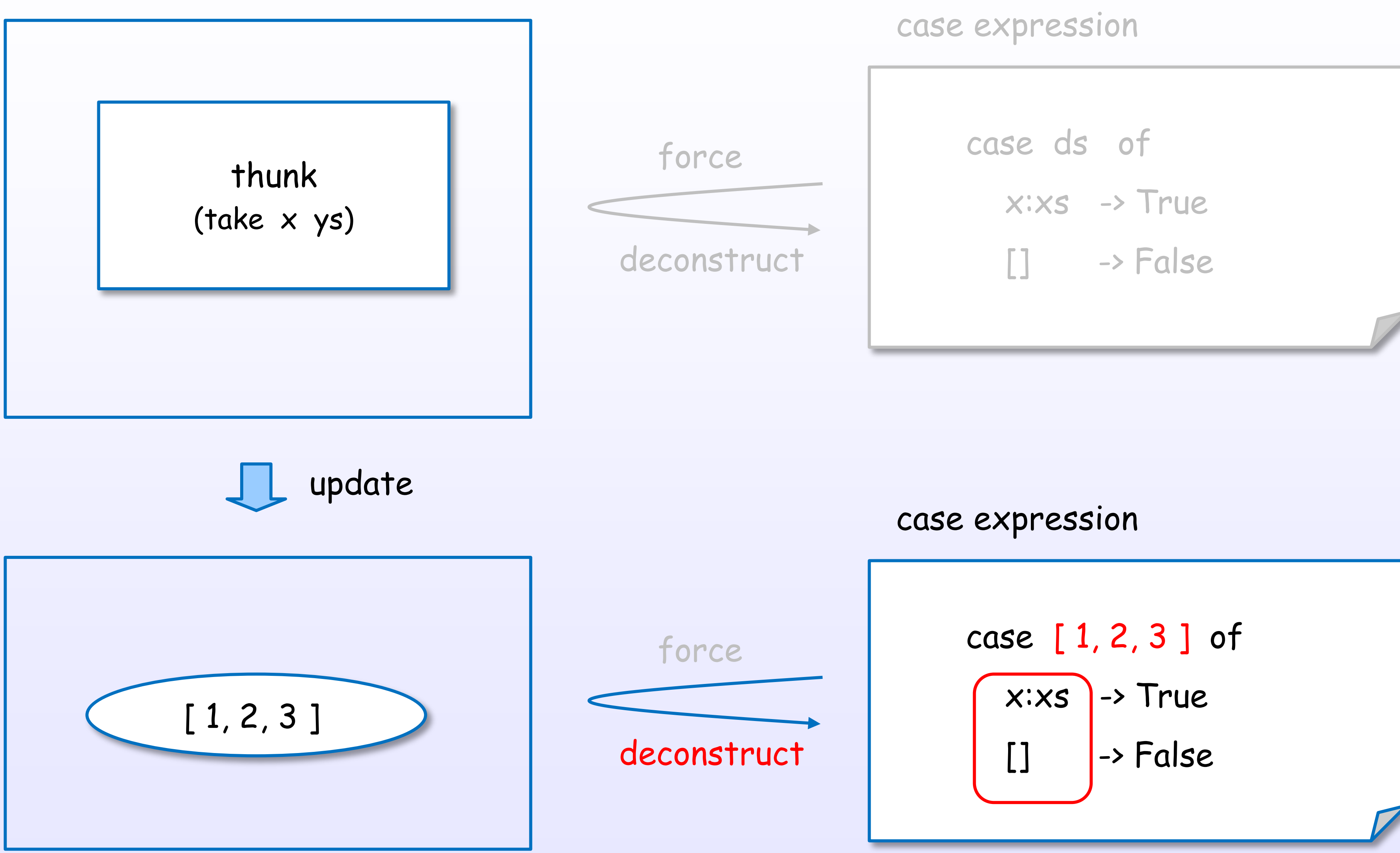
References : [1]

A case expression forces a thunk



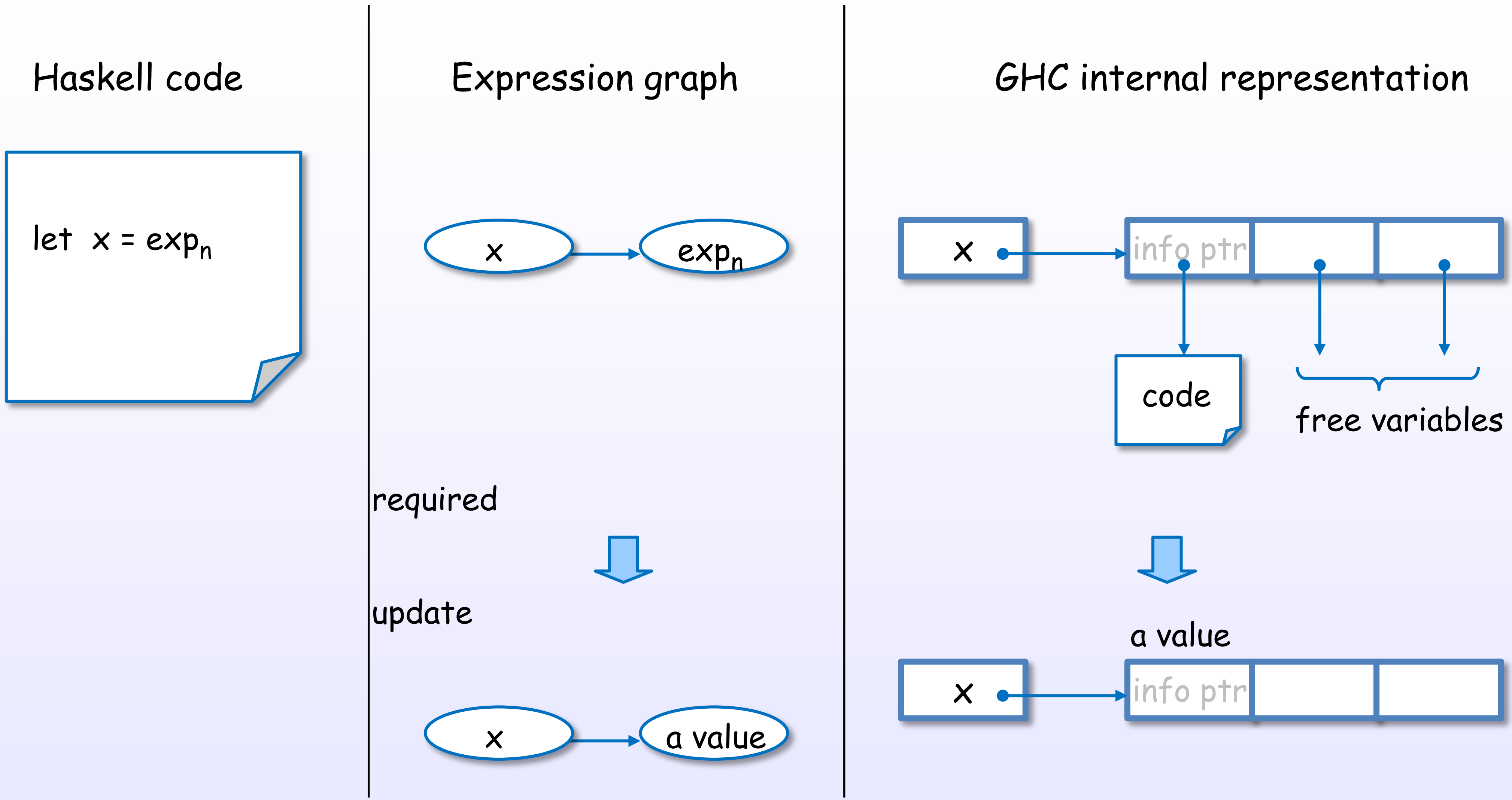
References : [1]

A case expression forces a thunk



References : [1]

Forcing and update



when a variable is bound
it is generally bound to an unevaluated closure allocated in the heap

このイメージを伝える

References : [1]

WHNF

internal representation of WHNF

heap objectイメージ

Constructor   XX XX XX

evaluated   unevaluated

Just   (map f   xs)

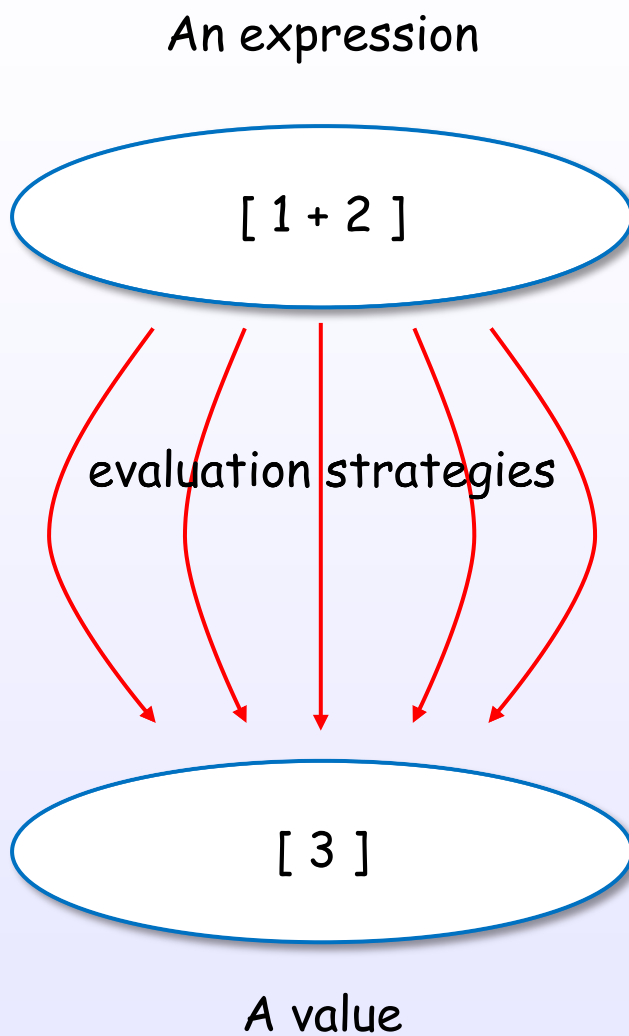
evaluated   unevaluated

References : [1]

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
- …

[Bird, Chapter 2, 7]

[TAPL, Chapter 3]

References : [1]

Evaluation concept layer

Denotational semantics

Operational semantics
(Evaluation strategies / Reduction strategies)

Implementation techniques

[Bird, Chapter 7]

[Hutton, Chapter 8]

[TAPL, Chapter 3]

References : [1]

Evaluation layer for GHC's Haskell

Denotational semantics

- Strict semantics
- Non-strict semantics

[Terei]

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
- ...

[Terei]

[Haskell/Laziness]

[Bird, Chapter 7] [stephen]

[Hutton, Chapter 8]

[Lazy Eval] [CIS194]

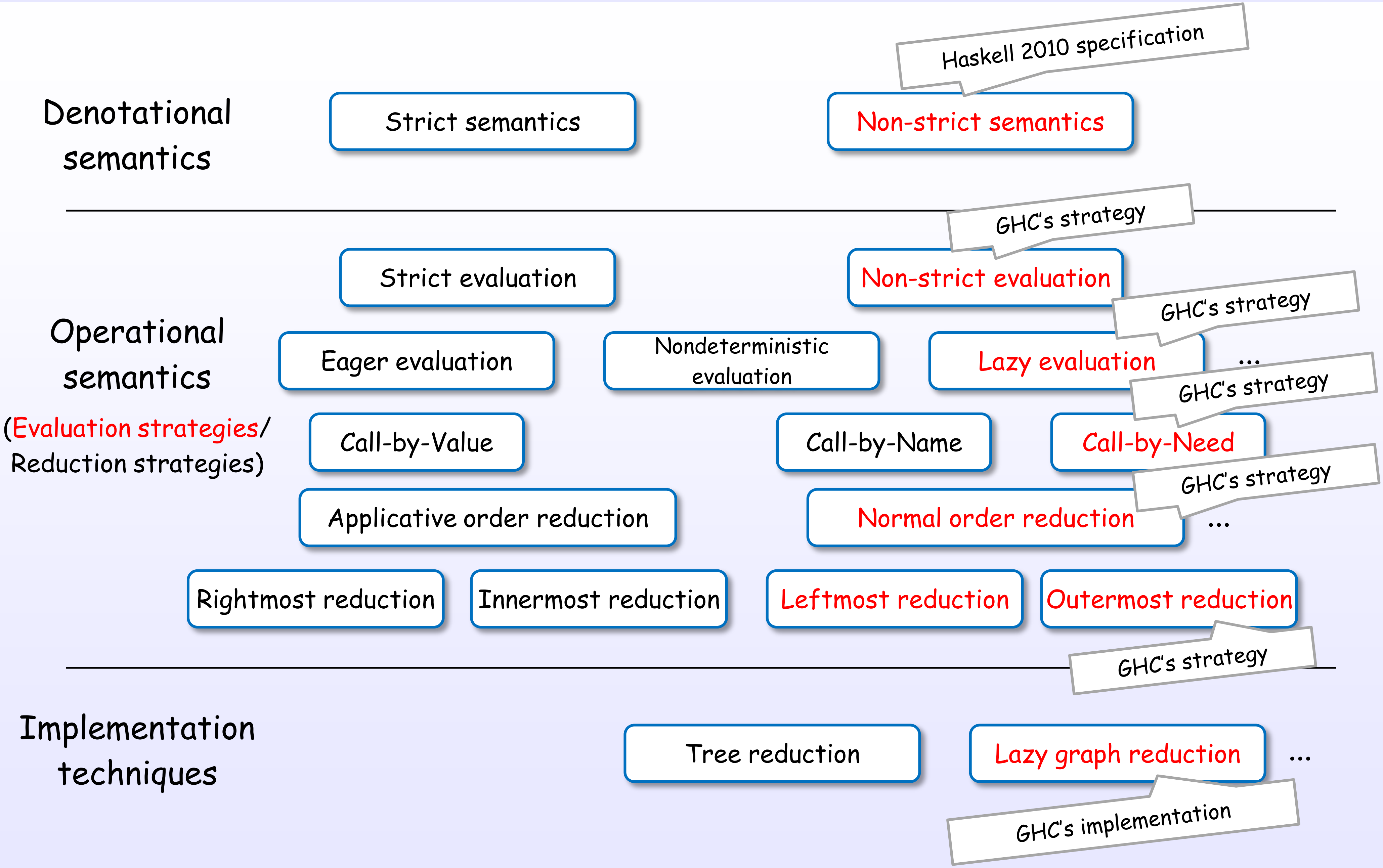
[Lazy vs NS]

[TAPL, Chapter 3]

References : [1]

eva
call
call
call
eva
no-
non
eva
fun
lazy
sha
[Bir

Evaluation layer for GHC's Haskell



References : [1]

Evaluation strategies and order

a (b c) + d (e (f g))

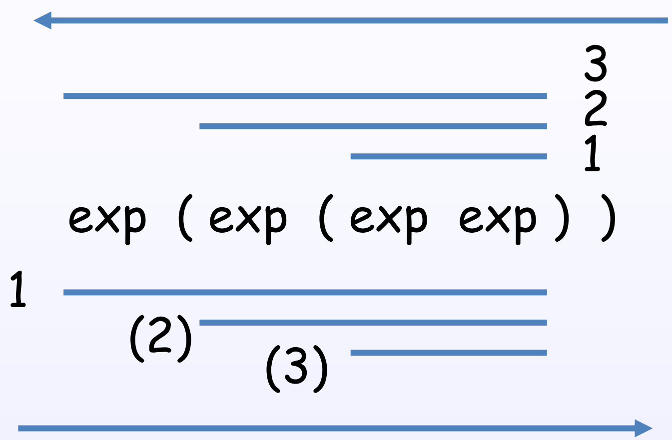
order

[Bird]
[Hutton]

References : [1]

Evaluation strategies and order

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



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

[Bird]
[Hutton]

References : [1]

Evaluation strategies and order

data flow

exp (exp (exp exp))

control flow

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

References : [1]

Simple example of typical evaluations

Eager evaluation
(Strict evaluation)

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

square (1 + 2)

argument evaluation first

square (3)

3 * 3

9

Lazy evaluation
(Non-strict evaluation)

default: Haskell (GHC), ...

square (1 + 2)

apply first

(1 + 2) * (1 + 2)

(3) * (3)

9

[Bird]
[Hutton]

References : [1]

Simple example of typical evaluations

Eager evaluation
(Strict evaluation)

square (1 + 2)

square (3)

argument evaluated

3 * 3

9

Lazy evaluation
(Non-strict evaluation)

square (1 + 2)

(1 + 2) * (1 + 2)

argument evaluation delayed !

(3) * (3)

9

[Bird]
[Hutton]

References : [1]

Evaluation in Haskell (GHC)

GHC chosen lazy evaluation

必要な時に、必要な箇所のみを評価する

(STG p.11)

↓

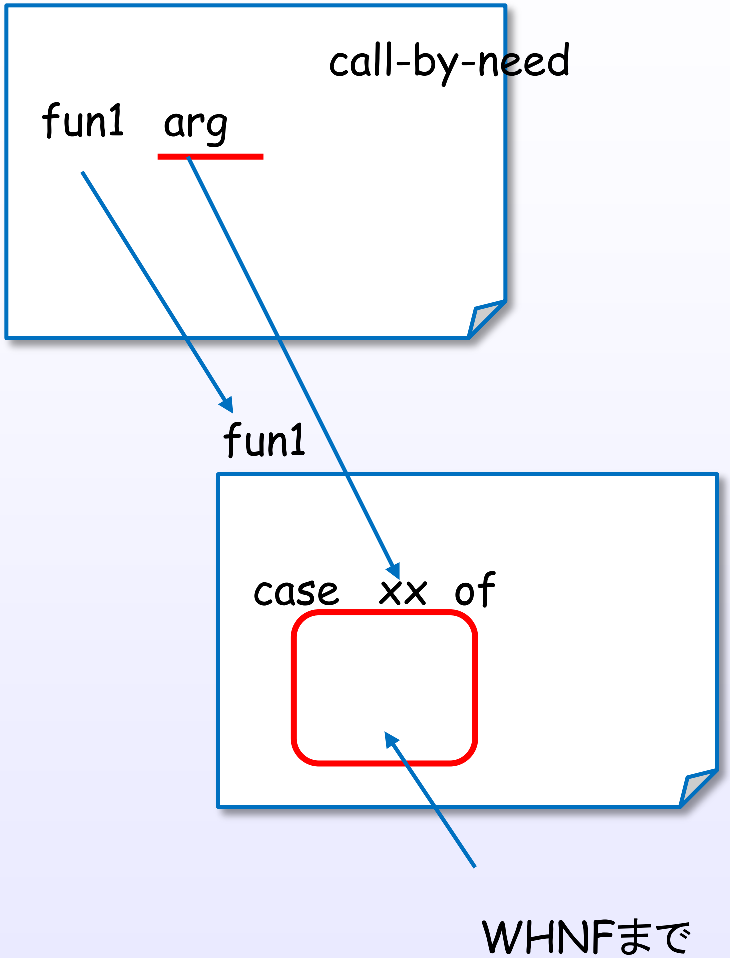
・引数評価を先送る （case式が来るまで評価しない） call-by-need

・部分式を完全評価しない （caseのパターンマッチで参照するところのみを評価する）WHNF

これは、計算量を最小化する戦略（メモリ量でなく）

References : [1]

eval 全体のイメージ



・call-by-need
・WHNFまで
・caseで駆動
・パターンマッチ

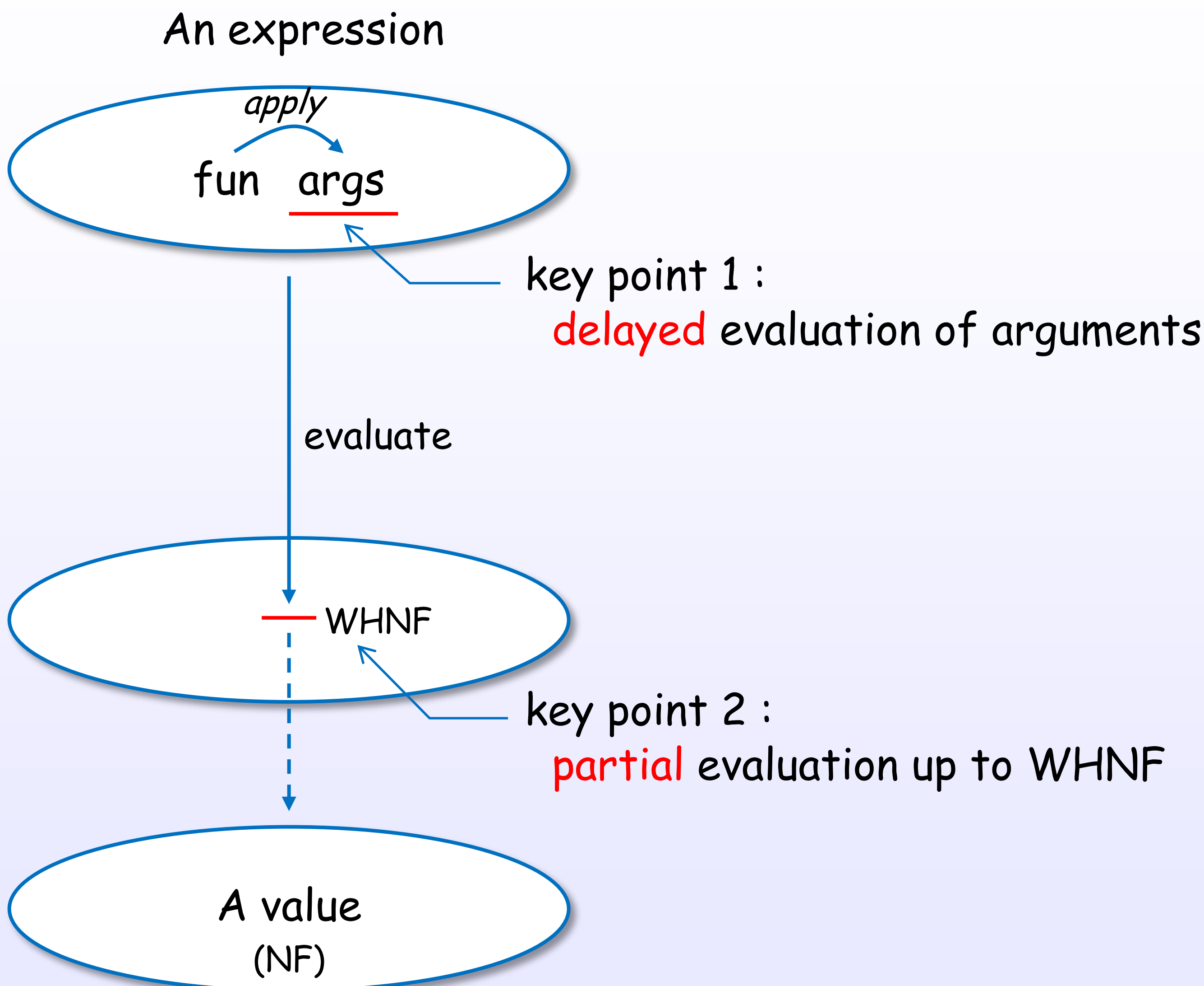
を、1枚の絵に

ページ順番の変更を

[STG]

References : [1]

Key concept of Haskell's lazy evaluation



References : [1]

key point 1 : delayed evaluation of arguments

apply

```
head  (map f xs)
```

internal transformation by GHC

```
let  thunk0 =  map f xs
in  head  thunk0
```

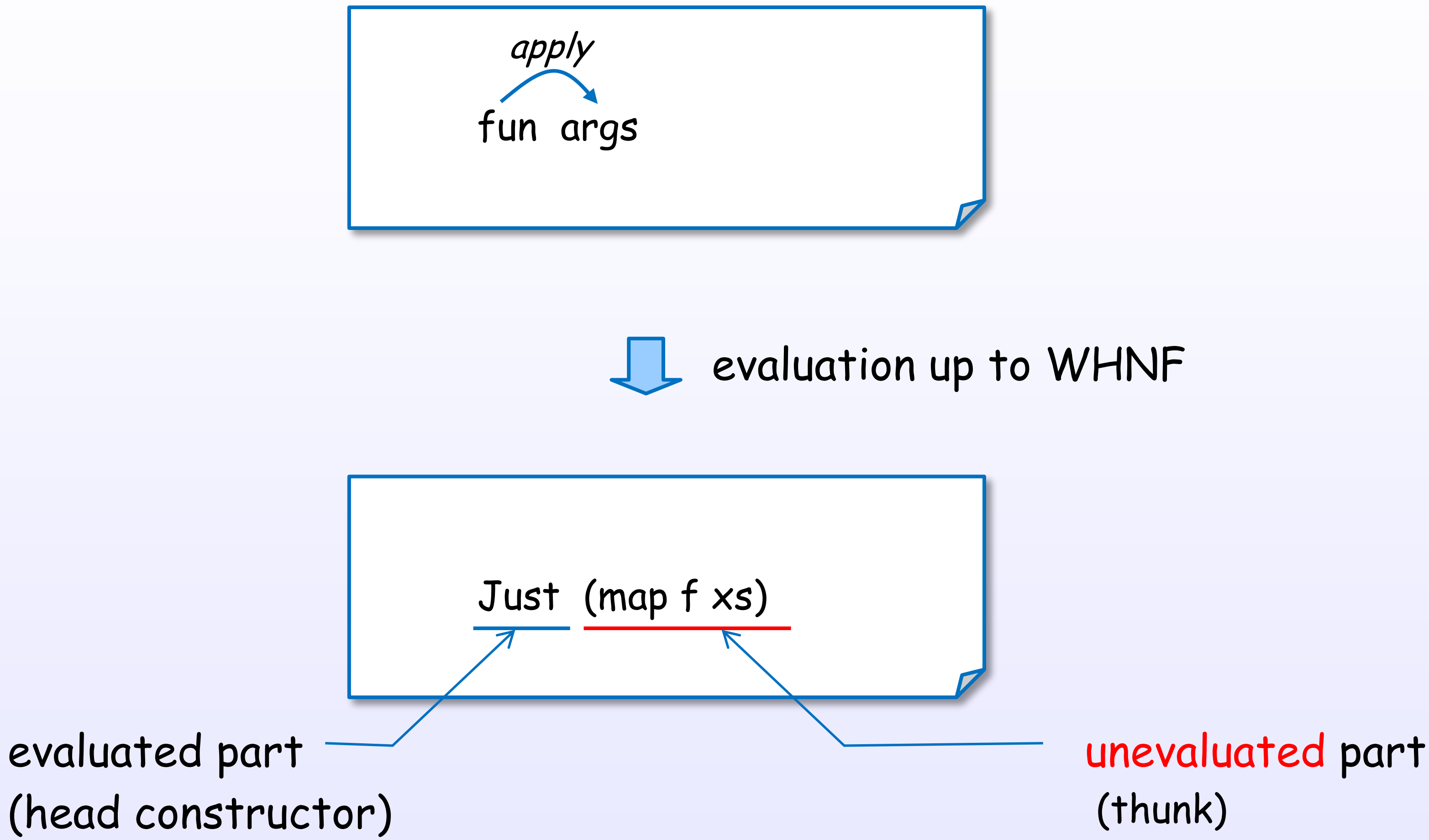
apply

build a thunk
in heap memory

GHC implements lazy evaluation using the thunk.
Evaluation of arguments is delayed with the thunk.

References : [1]

key point 2 : partial evaluation up to WHNF



GHC can partially evaluate a expression.
Constructor can hold an unevaluated expression (a thunk).

References : [1]

では、必要なときはいつか？

References : [1]

では、必要なときはいつか？

Haskell code

```
f = case  (g x) of
          [] -> a
          _  -> b

g  (x:xs) = ...
g  []     = ...
```

HERE!

pattern match via
case expression and function definition
will {cause, trigger} the evaluation

[Terei] [CIS194]

[STG]

References : [1]

Pattern match

[CIS194]

References : [1]

Pattern match

strict pattern

lazy pattern

case expression
function definition

let bounding pattern
Irrefutable Patterns

[stephen]

References : [1]

Examples of evaluation steps

Example of repeat

repeat 1

1 : repeat 1

1 : 1 : repeat 1

1 : 1 : 1 : repeat 1

References : [1]

Example of repeat

repeat 1

1 : repeat 1

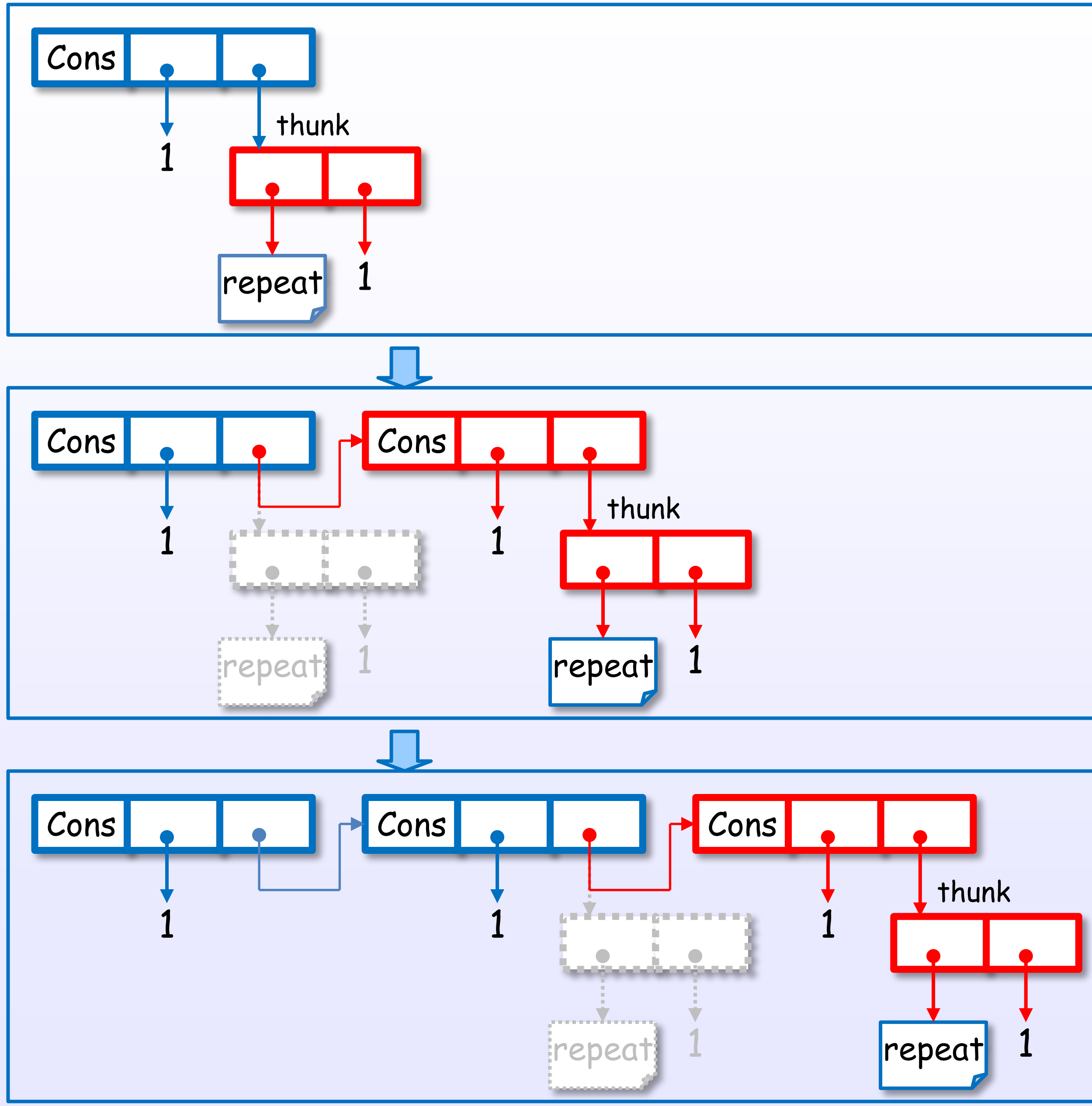
Cons 1 (repeat 1)

1 : 1 : repeat 1

Cons 1 (Cons 1 (repeat 1))

1 : 1 : 1 : repeat 1

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



References : [1]

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

⇓

...

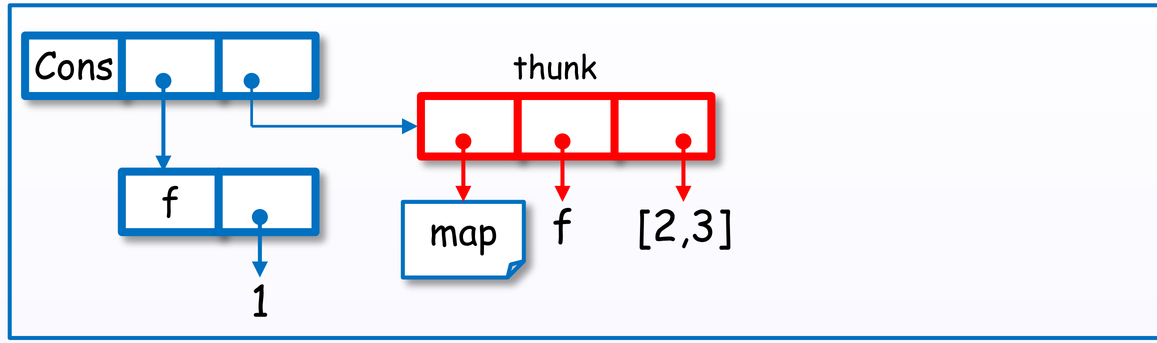
References : [1]

Example of map

map f [1, 2, 3]

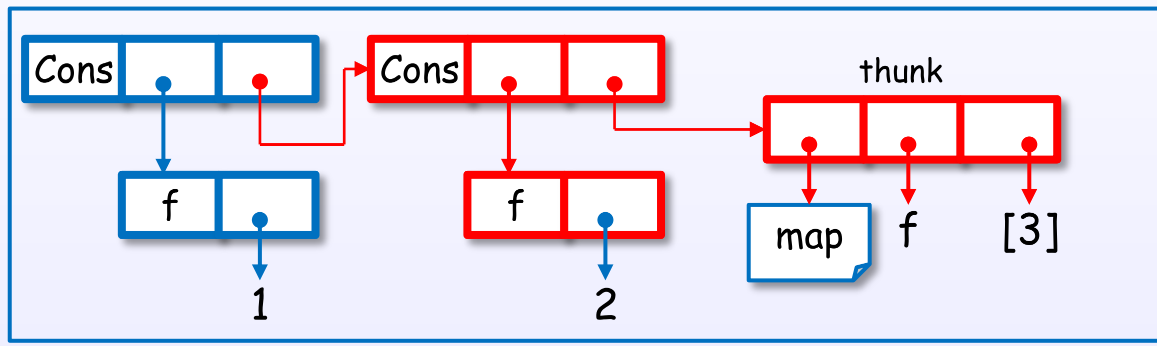
f 1 : map f [2, 3]

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



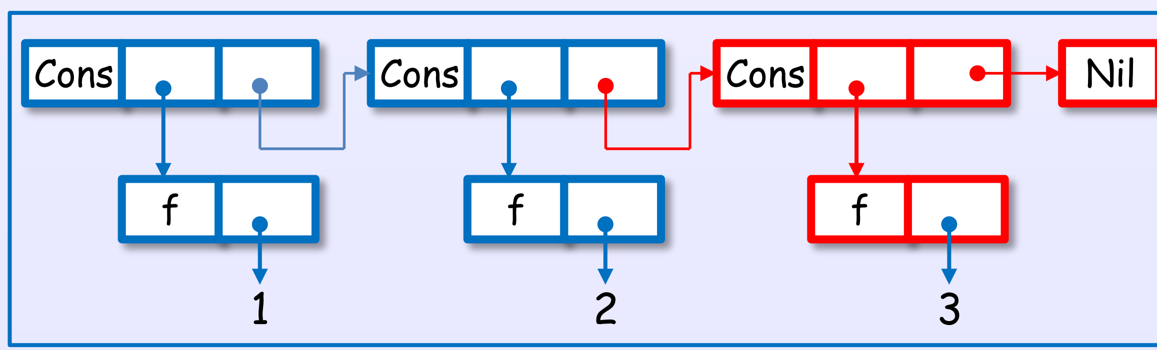
f 1 : f 2 : map f [3]

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 : [1]

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 : [1]

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]
```

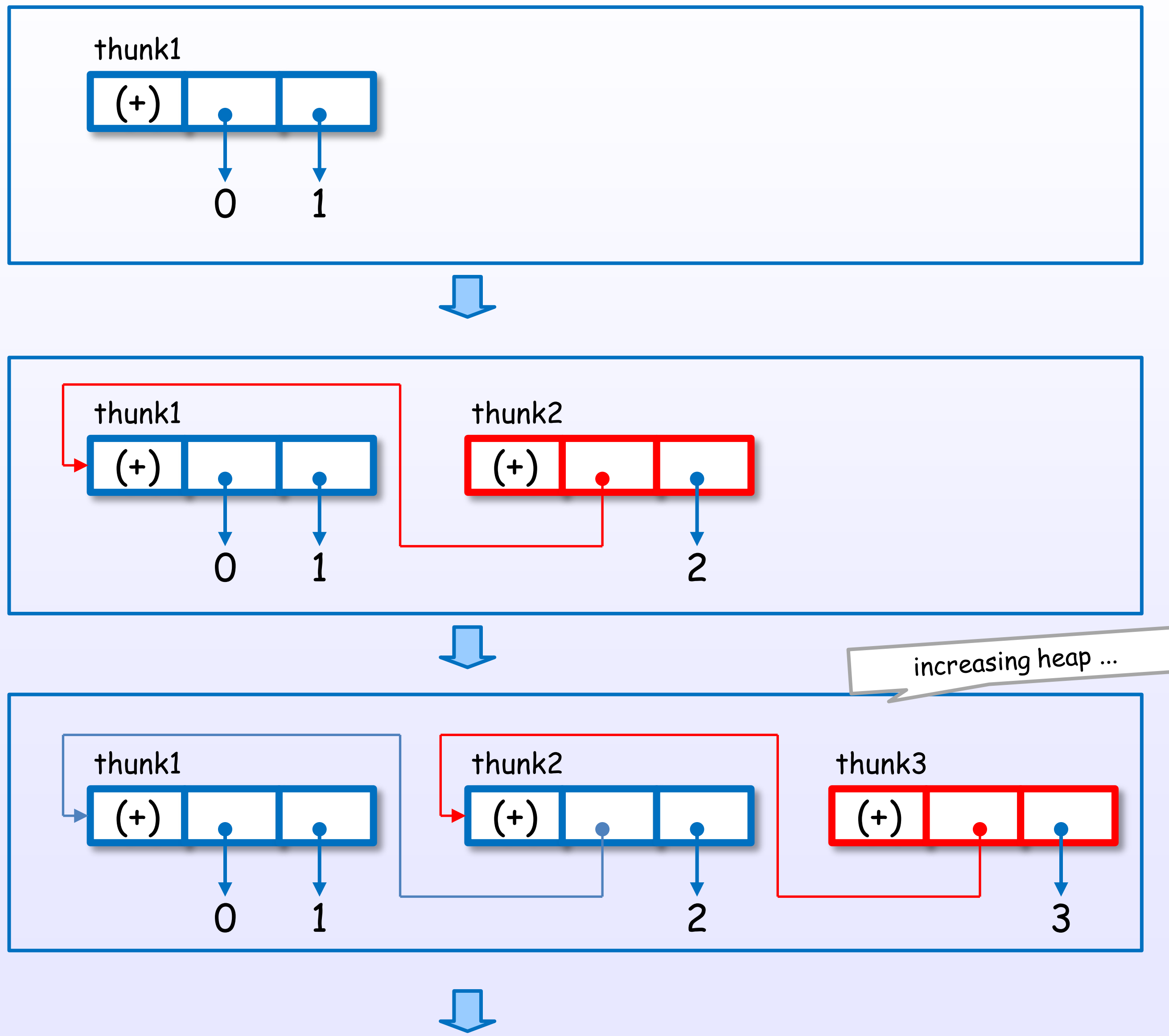
```
let  thunk2 = (thunk1 + 2)
in foldl (+) thunk2 [3 .. 100]
```

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

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

...

heap memory



References : [1]

Example of foldl' (strict)

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

⇩

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

⇩

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

⇩

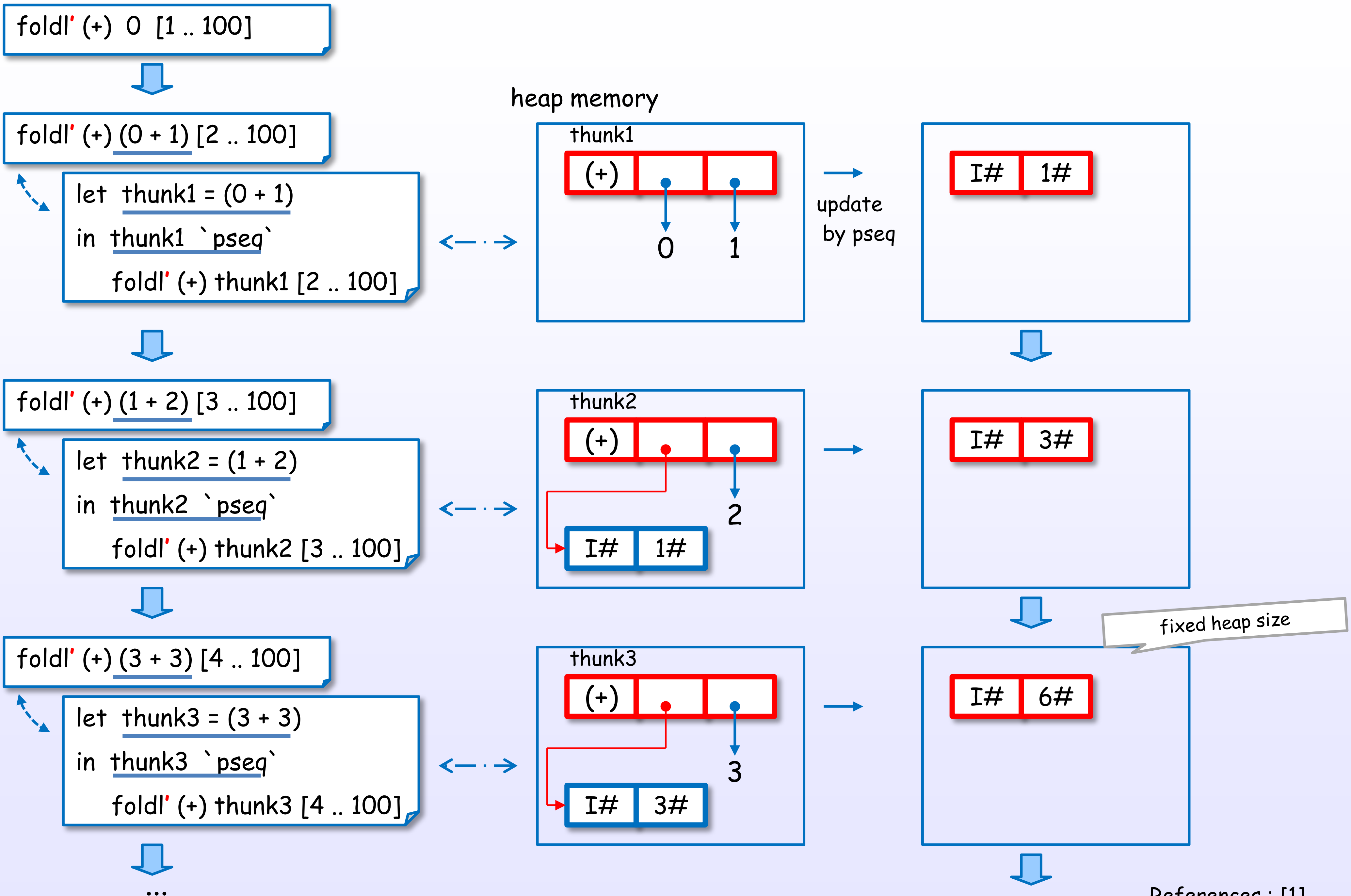
```
foldl' (+) (3 + 3) [4 .. 100]
```

⇩

...

References : [1]

Example of foldl' (strict)



References : [1]

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]

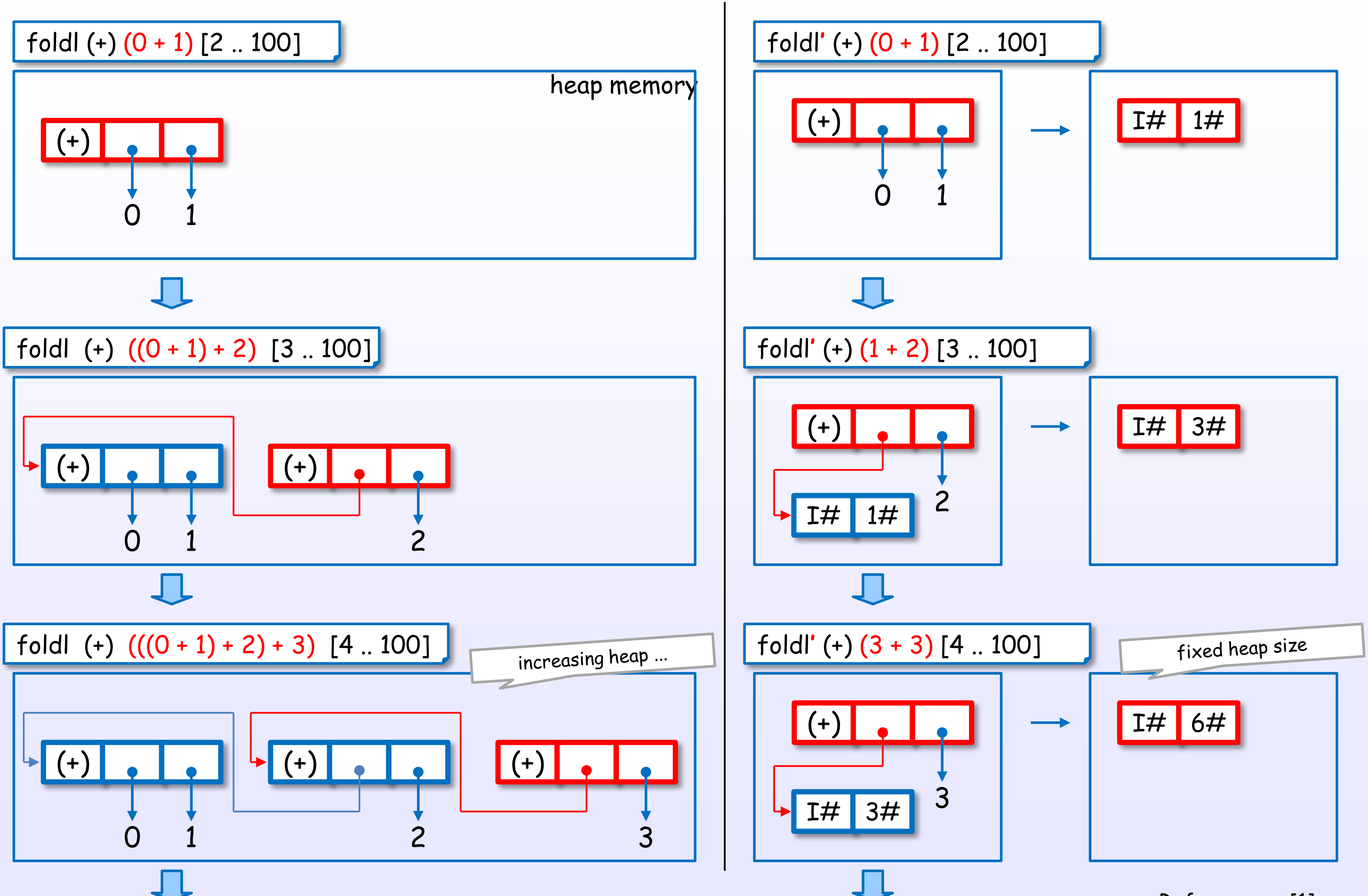
foldl' (+) (0 + 1) [2 .. 100]

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

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

References : [1]

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



References : [1]

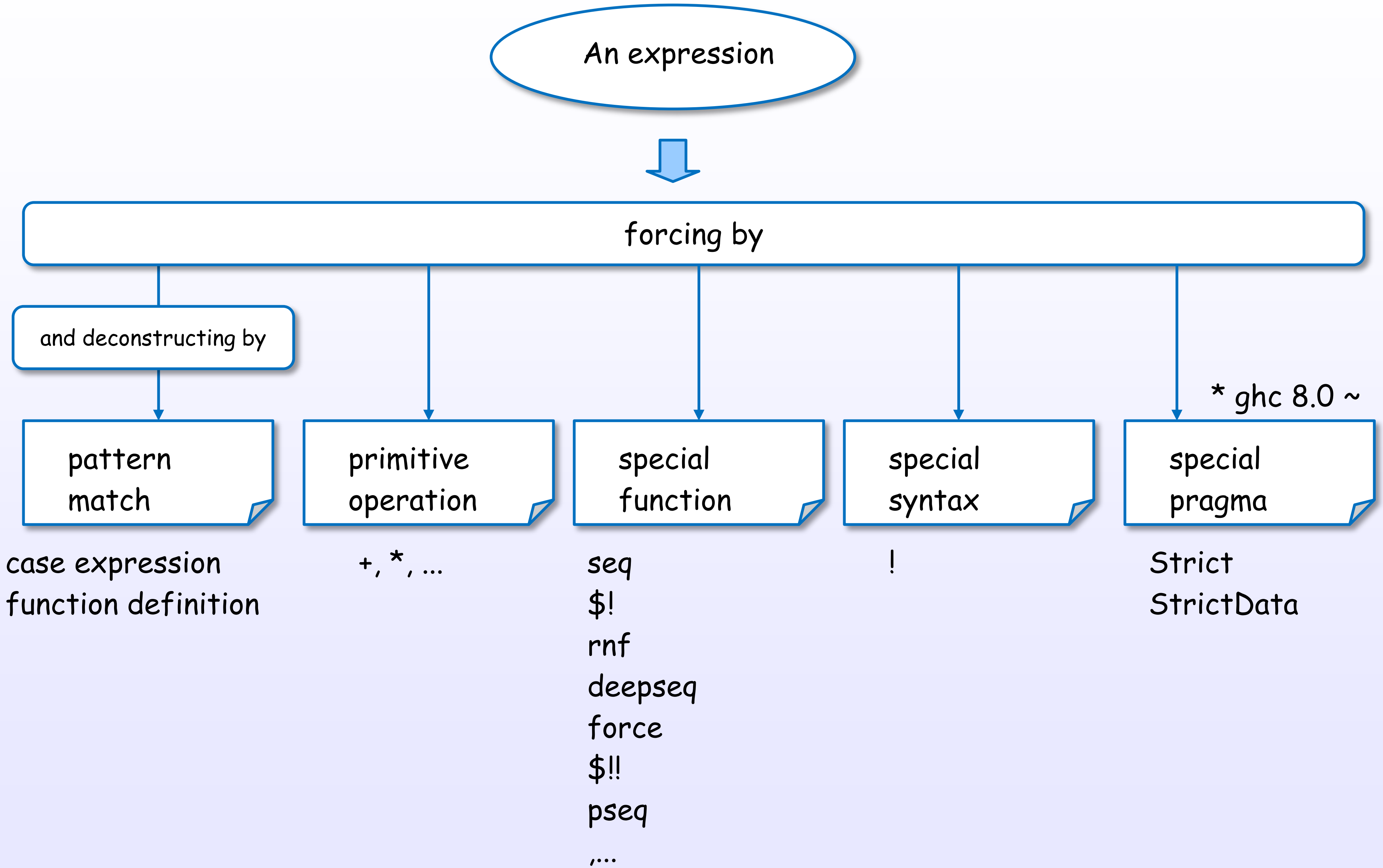
Example of nest-function

```
take 5 ( map f xs )
```

References : [1]

Controlling the evaluation

How to drive evaluation



References : [1]

Example of the evaluation by pattern match

case expression

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

case expression
function definition

References : [1]

Example of the evaluation by primitive operation

primitive operation

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

+, *, ...

References : [1]

Example of the evaluation by special function

special function

```
f x y = seq x y
```

seq
$!
rnf
deepseq
force
$!!
pseq
,...

to WHNF
to NF

[parconc, Ch.2]

[RWH, Ch.24-25]

[stephen]
[hack.hands]

[Bird, Chapter 7]

[CIS194]

Please refer the document more detail. [xx]
hoogle or hackage

References : [1]

Example of the evaluation by special function

seq のobject図イメージ force rnf rwhnf

deepseq のobject図イメージ

References : [1]

Example of the evaluation by special function

表で整理

to WHNF	seq $!	rwhnf	pseq
to NF	force deepseq $!!	rnf	

References : [1]

Example of the evaluation by special syntax

special syntax

```
{-# LANGUAGE BangPatterns #-}

f !xs = g xs
```

```
{-# LANGUAGE BangPatterns #-}

data ...
```

BangPattern

[RWH, Ch.25]

[stephen]

Please refer the document more detail. [xx]

[user guide, 7.19]

References : [1]

Example of the evaluation by special pragma

special pragma

```
{-# LANGUAGE Strict #-}

f xs = g xs
```

* ghc 8.0 ~

```
{-# LANGUAGE StrictData #-}

f xs = g xs
```

Strict
StrictData

Please refer the document more detail. [xx] [wiki]

References : [1]

5. Implementation of evaluator

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
- …

[Terei]

[Haskell/Laziness] [Bird, Chapter 7] [stephen]

[Lazy vs NS] [Lazy Eval] [Hutton, Chapter 8] [CIS194]

[TAPL, Chapter 3]

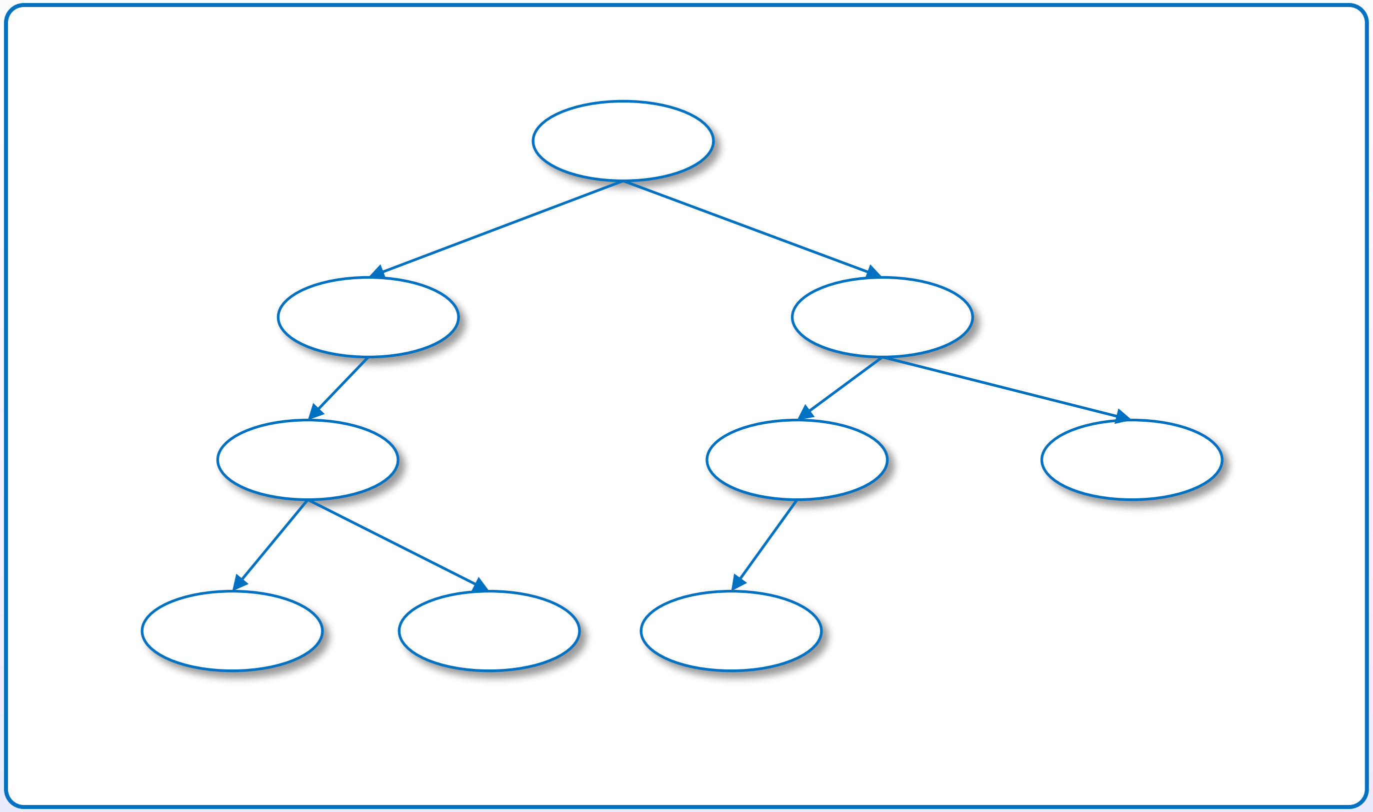
[Terei]
eva
call
call
call
eva
no-
non
eva
fun
lazy
sha
[Bir

References : [1]

Lazy graph reduction

Tree

AST represents an expression

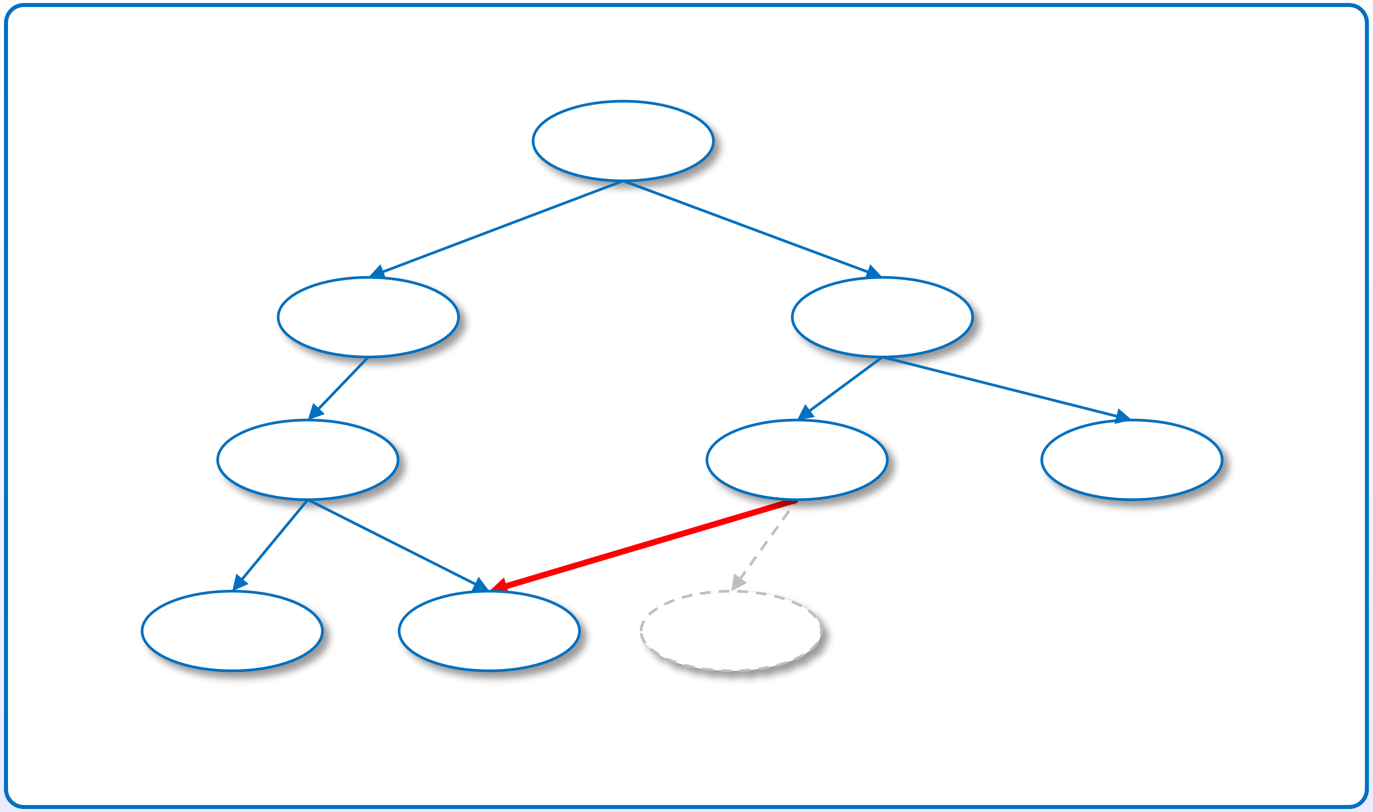


Stack base

References : [1]

Graph

Share the term, looped
not Tree, but Graph



Heap base

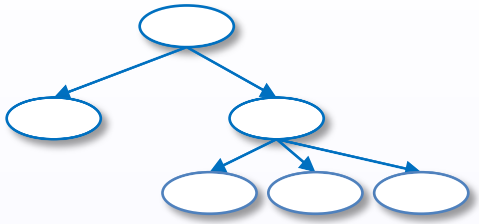
[Terei]

[hack.hands]

[CIS194]

References : [1]

Tree and graph reduction



Tree reduction

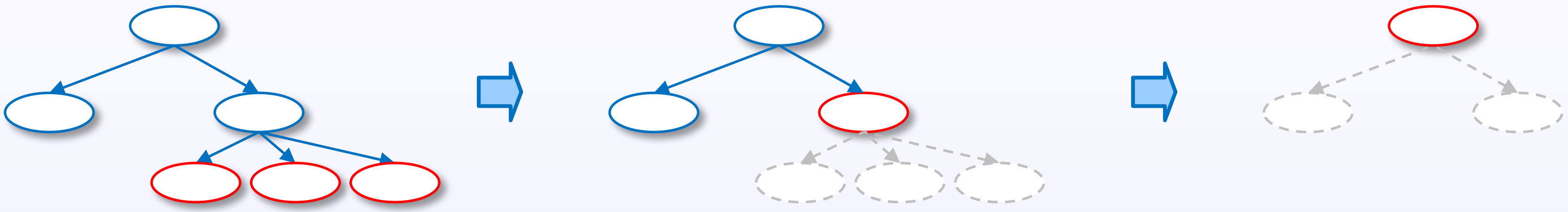
Graph reduction

copy arguments

share arguments by pointers

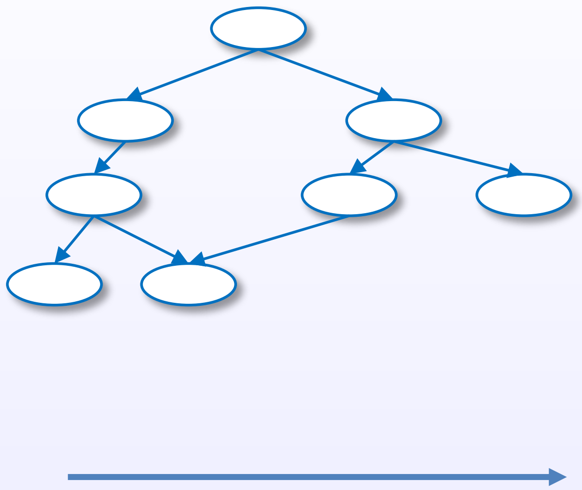
References : [1]

Graph reduction



References : [1]

Graph reduction and lazy



References : [1]

STG-machine

Abstract machine

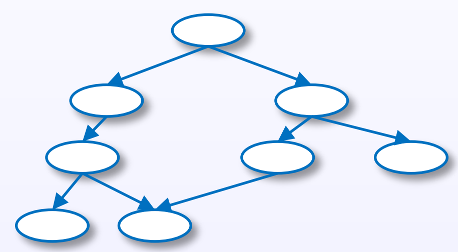
References : [1]

Layer

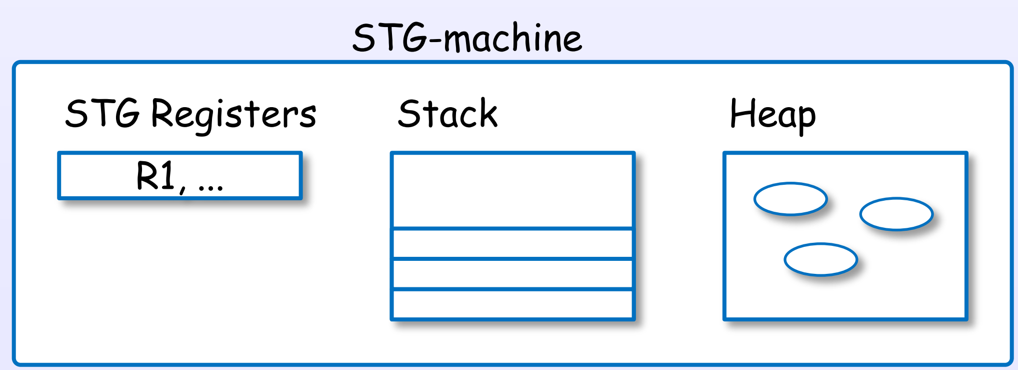
Haskell code

take 5 [1..10]

Internal representation
by graph

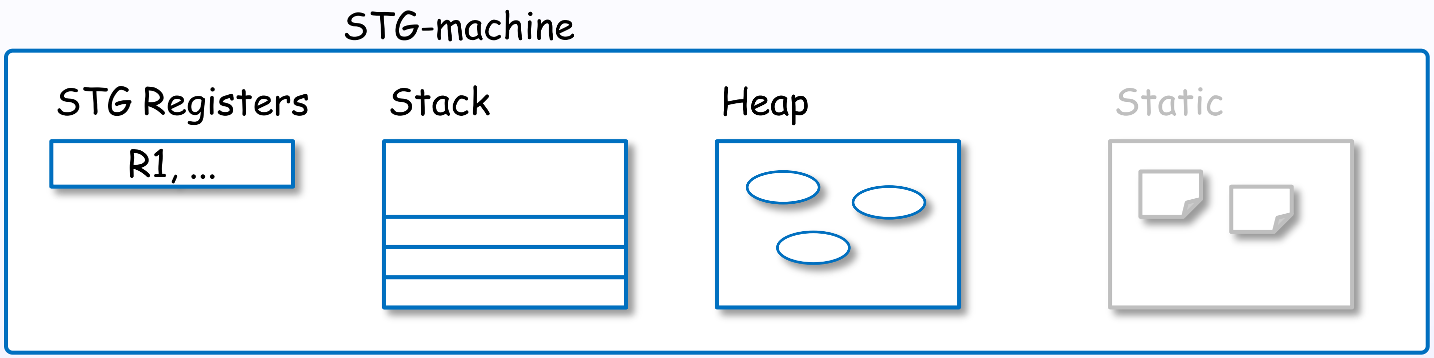


Evaluation (execution, reduction)
by STG-machine



References : [1]

STG-machine

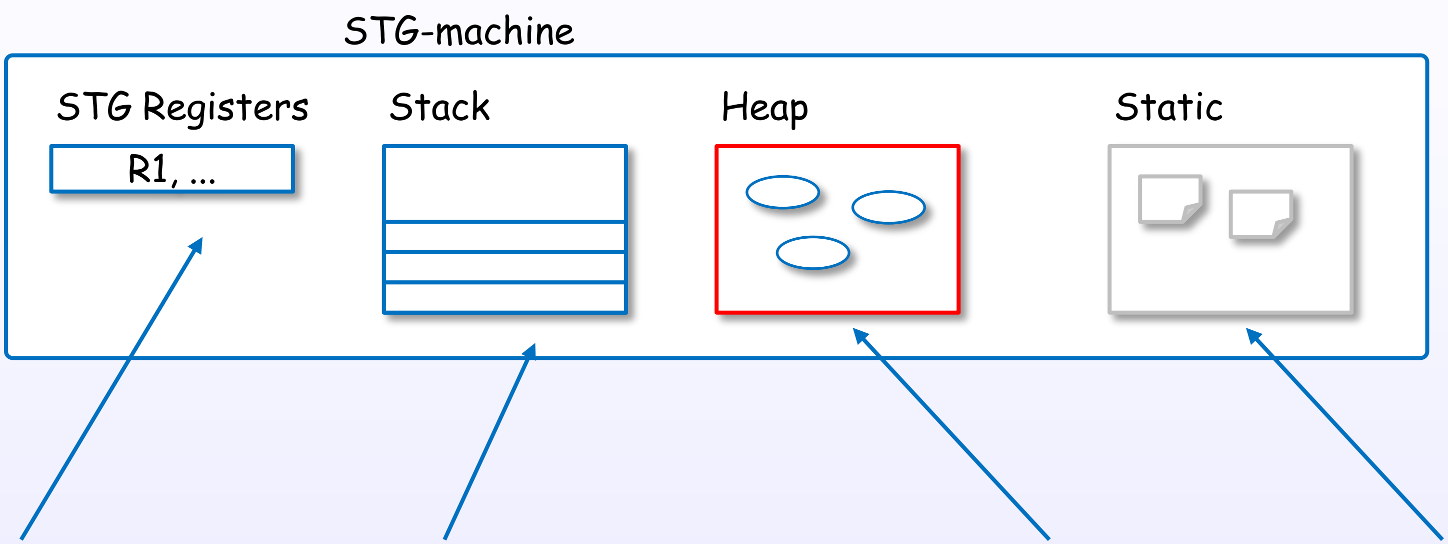


STG-machine is abstraction machine
which is defined by operational semantics.

STG-machine efficiently performs lazy graph reduction.

References : [1]

STG-machine



mainly using for
call/return convention
various control

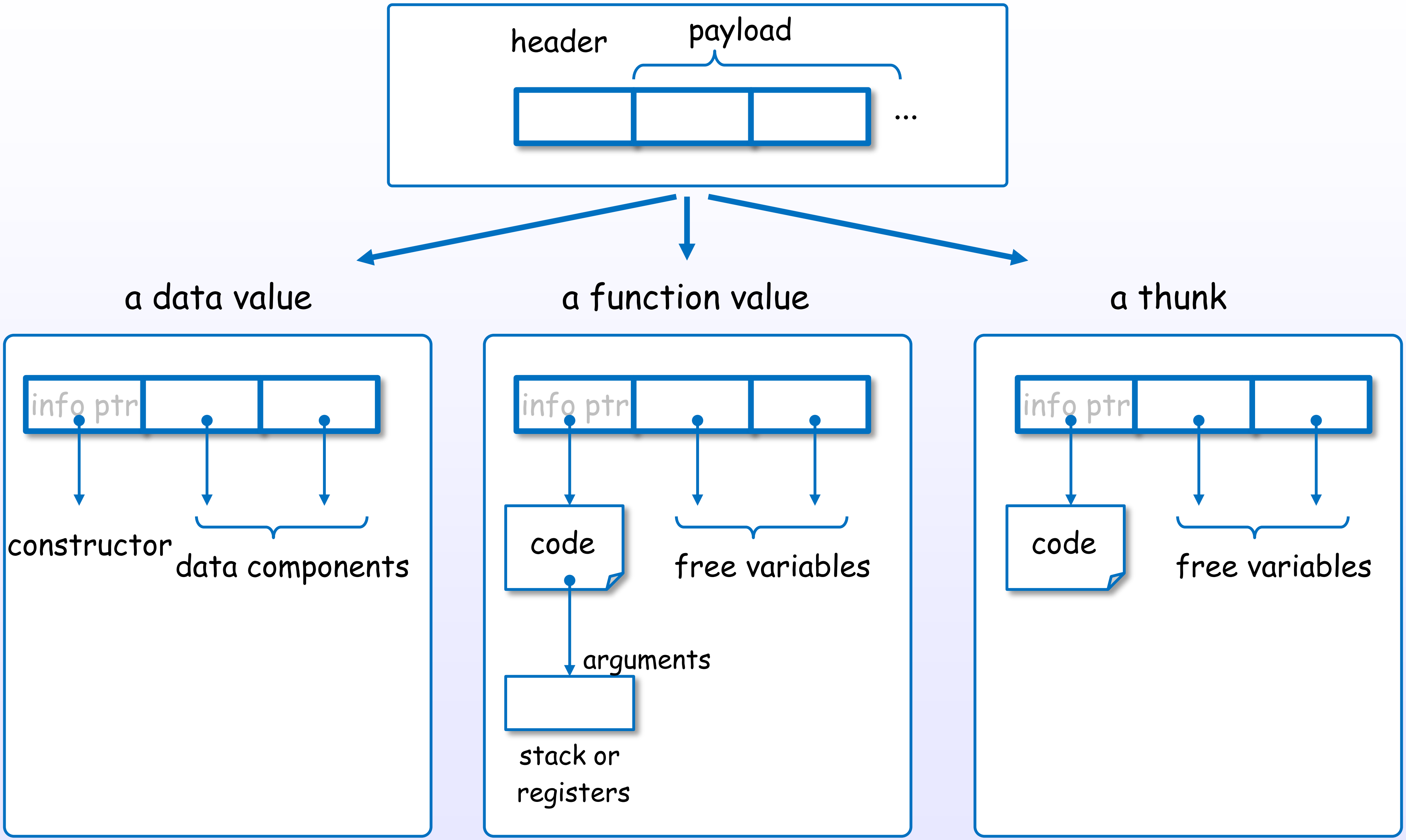
mainly using for
nest continuation
argument passing

mainly using for
allocating thunks

mainly using for
code
static closure

References : [1]

an unified representation in {heap, stack, static} memory

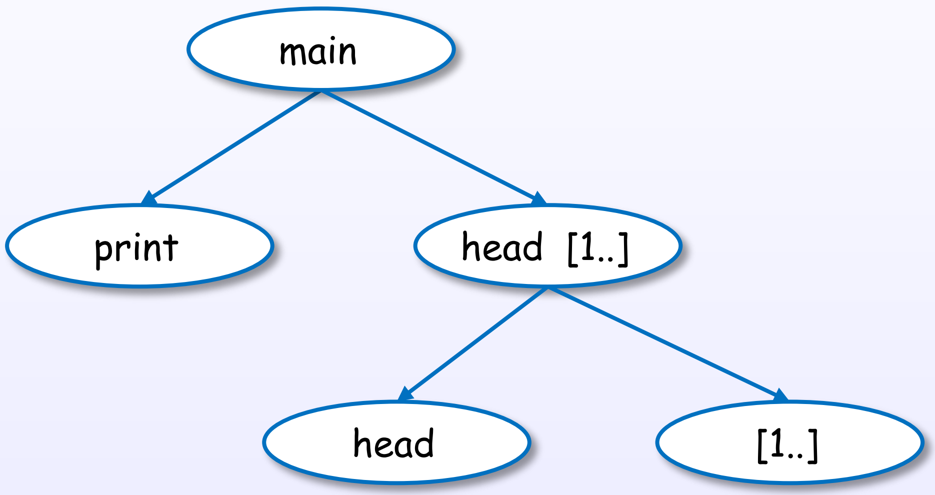


いずれも、広義の、"closure" (= code + environment(free variables))

References : [1]

Mapping the graph to the code

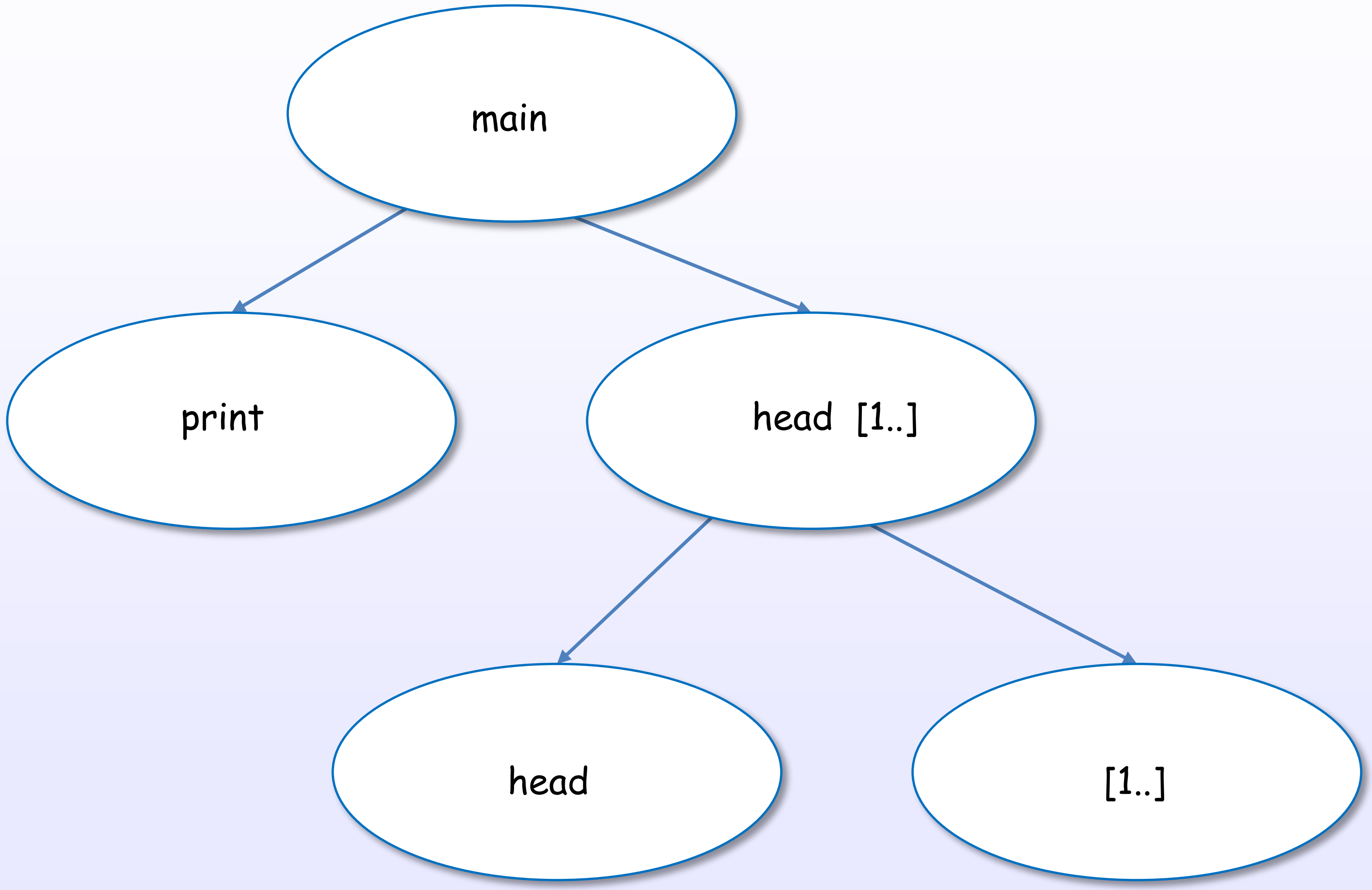
```
main = print (head [1..])
```



References : [1]

Mapping the graph to the code

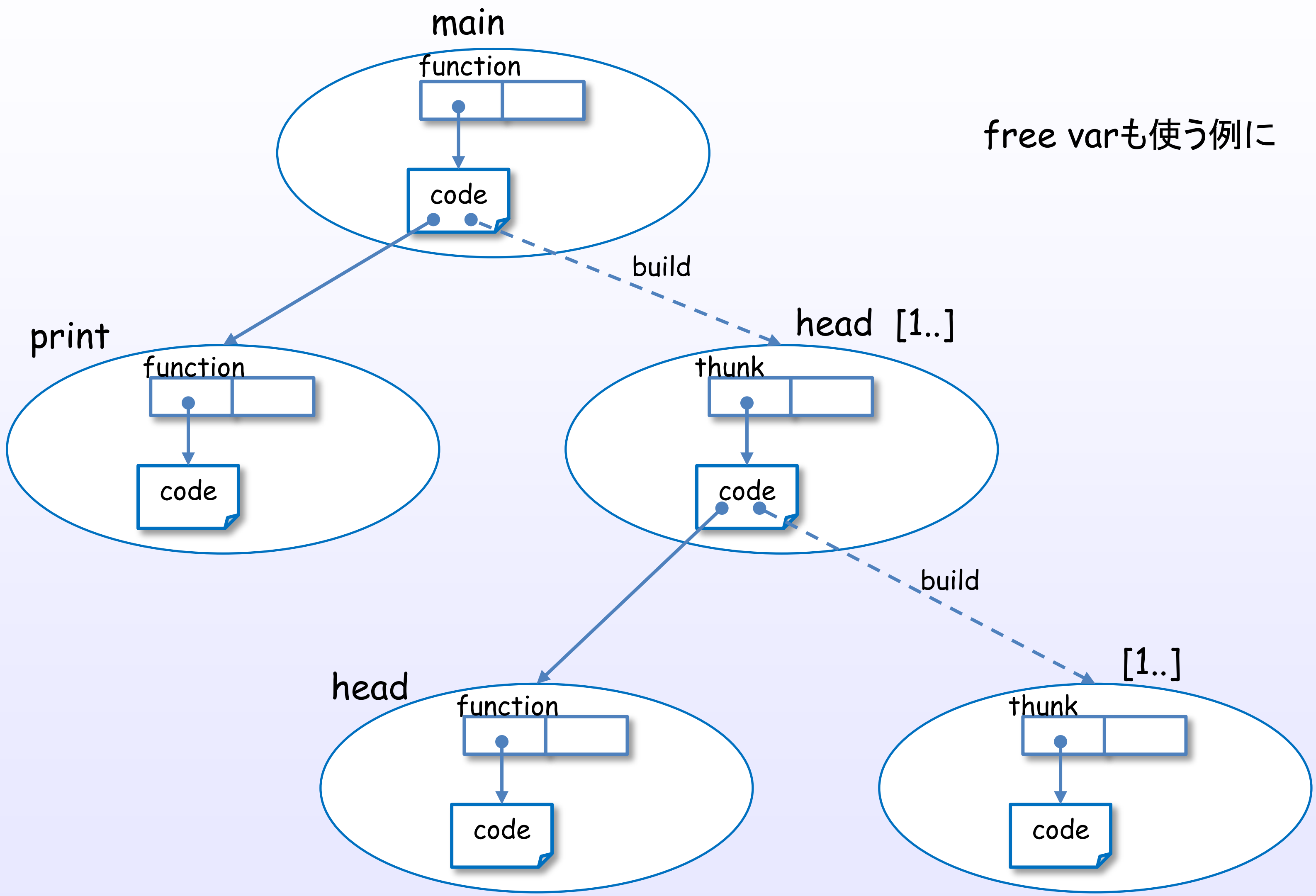
main = print (head [1..])



References : [1]

Mapping the graph to the code

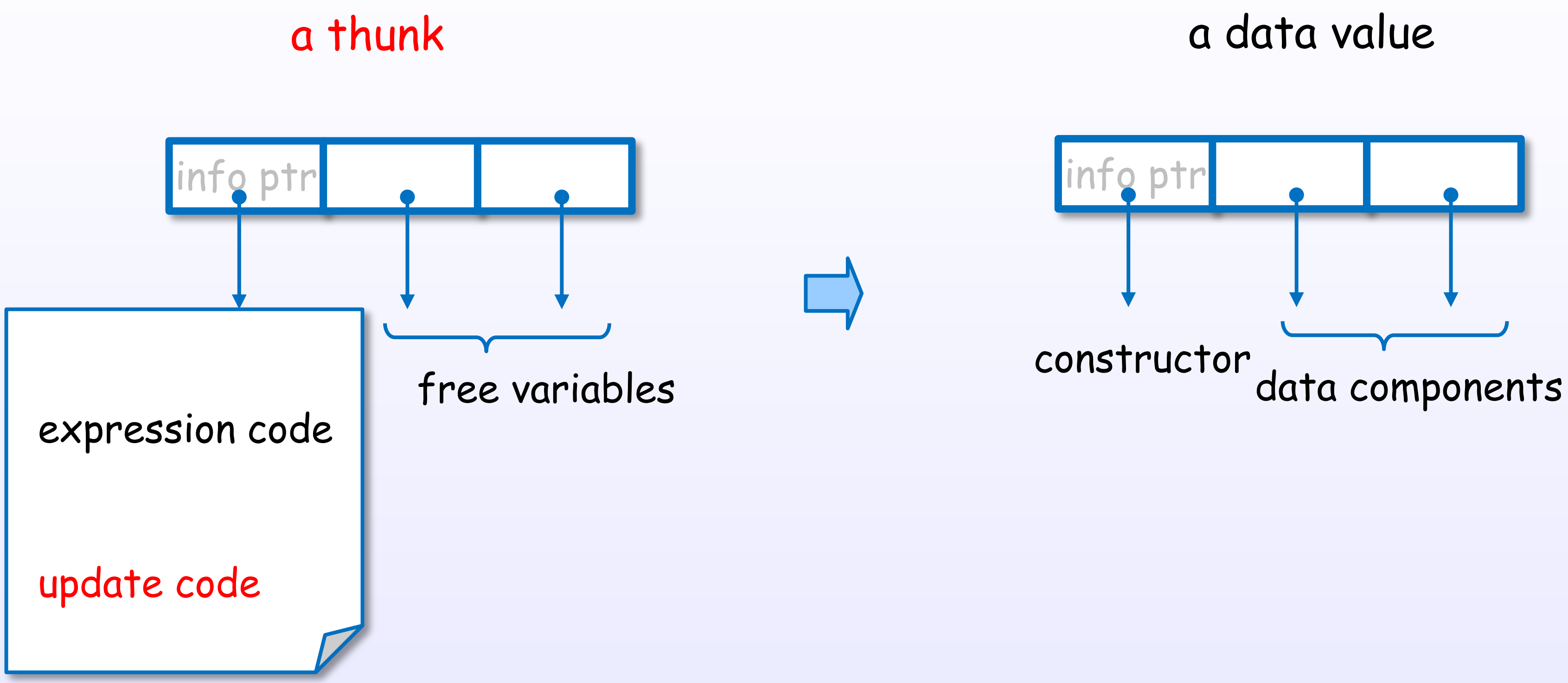
main = print (head [1..])



free varも使う例に

References : [1]

self-updating model



References : [1]

6. Semantics

Evaluation layer for GHC's Haskell

Denotational semantics

Strict semantics | Non-strict semantics

[Terei]

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

eva
call
call
call
eva
no-
non
eva
fun
lazy
sha

Implementation techniques

Tree reduction | Lazy graph reduction | ...

[Bir

[Terei]

[Haskell/Laziness]

[Bird, Chapter 7] [stephen]

[Hutton, Chapter 8]

[Lazy vs NS] [Lazy Eval]

[CIS194]

[TAPL, Chapter 3]

References : [1]

Bottom

Well formed expression has a value

An expression

[1 + 2]

$\downarrow$ evaluate

Well formed expression has a value

[3]

A value

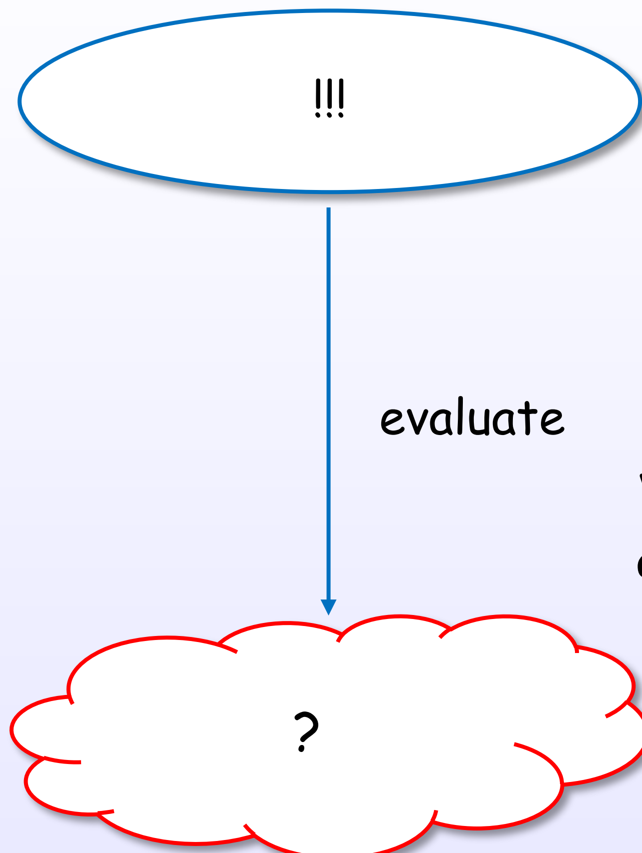
[Bird, Chapter 2]

References : [1]

Well formed expression has a value

An expression

infinite loop
partial defined function



evaluate

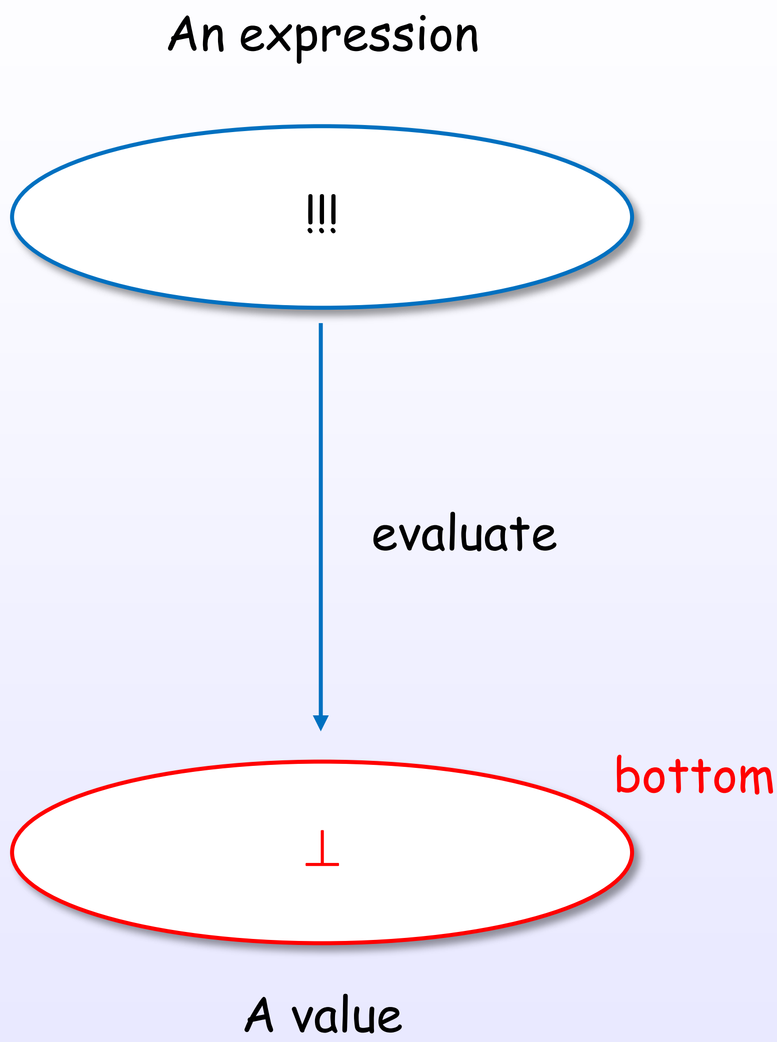
Well formed expression has a value

A value ?

[Bird, Chapter 2]

References : [1]

Well formed expression has a value



[Bird, Chapter 2]

References : [1]

Bottom

[Bird, Chapter 2]

References : [1]

Non-strict Semantics

Strictness

$$f \perp = \perp$$

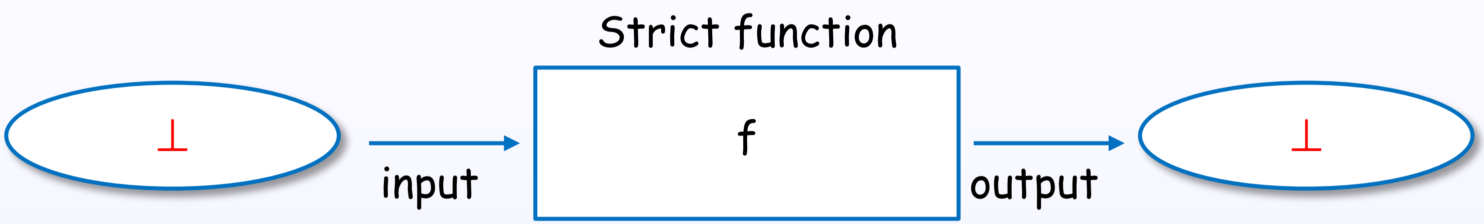
Strictness is attribution of the function.

[Bird, Chapter 2]

References : [1]

Strictness

$$f \perp = \perp$$



Strictness is attribution of the function.

[Bird, Chapter 2]

References : [1]

Strictness and Non-strictness

Strict

$$f \perp = \perp$$

Non-strict

$$f \perp \neq \perp$$

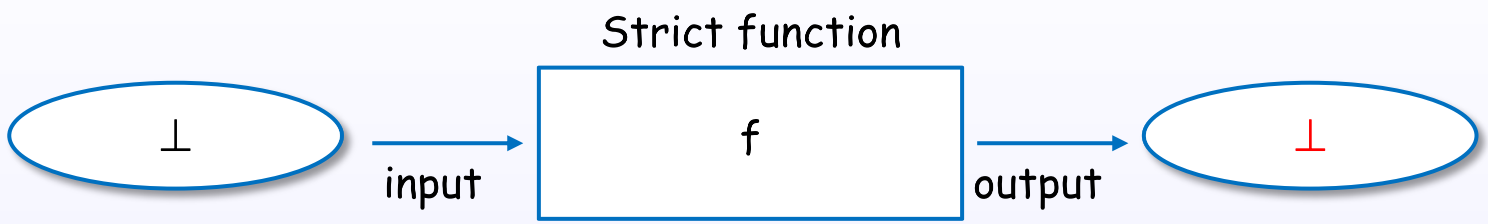
[Bird, Chapter 2]

References : [1]

Strictness and Non-strictness

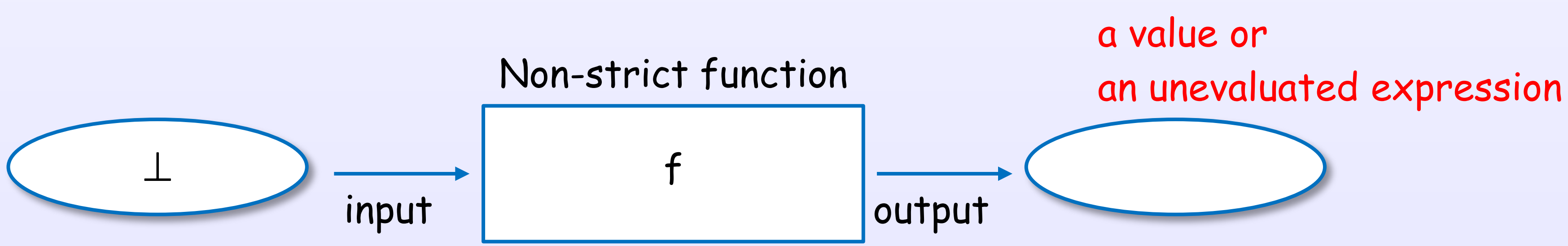
Strict

$$f \perp = \perp$$



Non-strict

$$f \perp \neq \perp$$



[Bird, Chapter 2]

References : [1]

Layer

Non-strictness	$f \perp = \perp$
Lazy evaluation	GHC chosen lazy evaluation to implement non-strict semantics.
Graph reduction	GHC chosen graph reduction to implement lazy evaluation.
STG-machine	GHC implements graph reduction by STG-machine.

References : [1]

seq and pseq

$$\text{seq } a \; b = \begin{cases} \perp, & \text{if } a = \perp \\ b, & \text{otherwise} \end{cases} \qquad \text{pseq } a \; b = \begin{cases} \perp, & \text{if } a = \perp \\ b, & \text{otherwise} \end{cases}$$

$\text{seq } a \perp = \perp$ — a is strict

$\text{seq } \perp b = \perp$ — b is strict

$\text{pseq } a \perp = \perp$ — a is strict

$\text{pseq } \perp b \neq \perp$ — b is non-strict

[Runtime Support for Multicore Haskell]

[Snoyman]

References : [1]

Strict analysis

Strict analysis

References : [1]

7. Appendix

References

References

[1] Haskell 2010 Language Report
https://www.haskell.org/definition/haskell2010.pdf

[2] The Glorious Glasgow Haskell Compilation System (GHC user's guide)
https://downloads.haskell.org/~ghc/latest/docs/users_guide.pdf

[3] A History of Haskell: Being Lazy With Class
http://haskell.cs.yale.edu/wp-content/uploads/2011/02/history.pdf

[4] The implementation of functional programming languages
http://research.microsoft.com/en-us/um/people/simonpj/papers/slpj-book-1987/slpj-book-1987.pdf

[5] 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

[6] 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

[7] Runtime Support for Multicore Haskell
http://community.haskell.org/~simonmar/papers/multicore-ghc.pdf

[8] I know kung fu: learning STG by example
https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/GeneratedCode

[9] GHC Commentary: The Layout of Heap Objects
https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/HeapObjects

[10] GHC Commentary: Strict & StrictData
https://ghc.haskell.org/trac/ghc/wiki/StrictPragma

References

[11] 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

[12] Thinking Functionally with Haskell (IFPH 3rd edition)
http://www.cs.ox.ac.uk/publications/books/functional/

[13] Programming in Haskell
https://www.cs.nott.ac.uk/~gmh/book.html

[14] Real World Haskell
http://book.realworldhaskell.org/

[15] Parallel and Concurrent Programming in Haskell
http://chimera.labs.oreilly.com/books/1230000000929

[16] Types and Programming Languages (TAPL)
https://mitpress.mit.edu/books/types-and-programming-languages

References

[17] Laziness
http://dev.stephendiehl.com/hask/#laziness

[18] Being Lazy with Class
http://www.seas.upenn.edu/~cis194/lectures/06-laziness.html

[19] 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

[20] Incomplete Guide to e Lazy Evaluation (in Haskell)
https://hackhands.com/guide-lazy-evaluation-haskell

[21] Evaluation on the Haskell Heap
http://blog.ezyang.com/2011/04/evaluation-on-the-haskell-heap

[22] Fixing foldl
http://www.well-typed.com/blog/2014/04/fixing-foldl

[23] How to force a list
https://ro-che.info/articles/2015-05-28-force-list

[24] Evaluation order and state tokens
https://www.fpcomplete.com/user/snoyberg/general-haskell/advanced/evaluation-order-and-state-tokens

[25] GHC illustrated
http://takenobu-hs.github.io/downloads/haskell_ghc_illustrated.pdf

References

[26] Haskell/Laziness
https://en.wikibooks.org/wiki/Haskell/Laziness

[27] Lazy evaluation
https://wiki.haskell.org/Lazy_evaluation

[28] Lazy vs. non-strict
https://wiki.haskell.org/Lazy_vs._non-strict

[29] Haskell/Denotational semantics
https://en.wikibooks.org/wiki/Haskell/Denotational_semantics

[30] Haskell/Graph reduction
https://en.wikibooks.org/wiki/Haskell/Graph_reduction

References

[31] Hackage
https://hackage.haskell.org

[32] Hoogle
https://www.haskell.org/hoogle

Lazy,... zzz