Reusing Thunks for Recursive Data Structures in Lazy Functional Programs

Yasunao TAKANO (Coma-systems Co., Ltd.)
Hideya IWASAKI (The University of Electro-Communications)
Tomoharu UGAWA (The University of Electro-Communications)

Thunk (promise, suspension)

- A *thunk* is created to delay the evaluation of an expression
  - A thunk contains the expression and the environment (a collection of pairs of bound variables and values)
- The process of evaluating the expression in a thunk is called "forcing"

n+1 ➡ T{n+1}{n=2} ➡ 3

delay — force

Our idea - *Thunk Reuse*

- Lazy evaluation has significant run-time overheads
  - Allocating many thunks (space-consuming task)

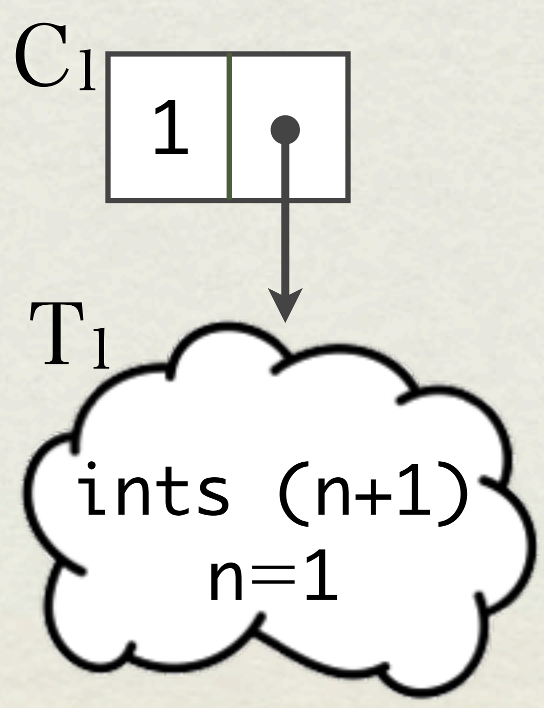
- We suppress thunk allocations by reusing the thunk that has been just forced
  - Our target is a thunk at the tail part of cons cell
  - We destructively update the environment of the thunk

Without thunk reuse

- The data constructor Cons ":" delays its arguments

```
ints n = n : ints (n + 1)
```

ints 1 ⇒ 1 : T_1{ints (n+1)}{n=1}

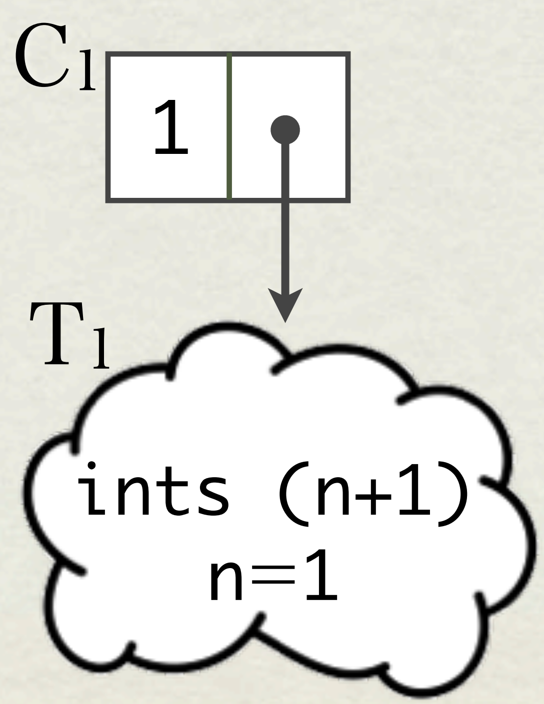


Without thunk reuse

- The data constructor Cons ":" delays its arguments

```
ints n = n : ints (n + 1)
```

ints 1 ⇒ 1 : $\underline{T_1\{\text{ints (n+1)}\}\{n=1\}}$



Forcing T_1

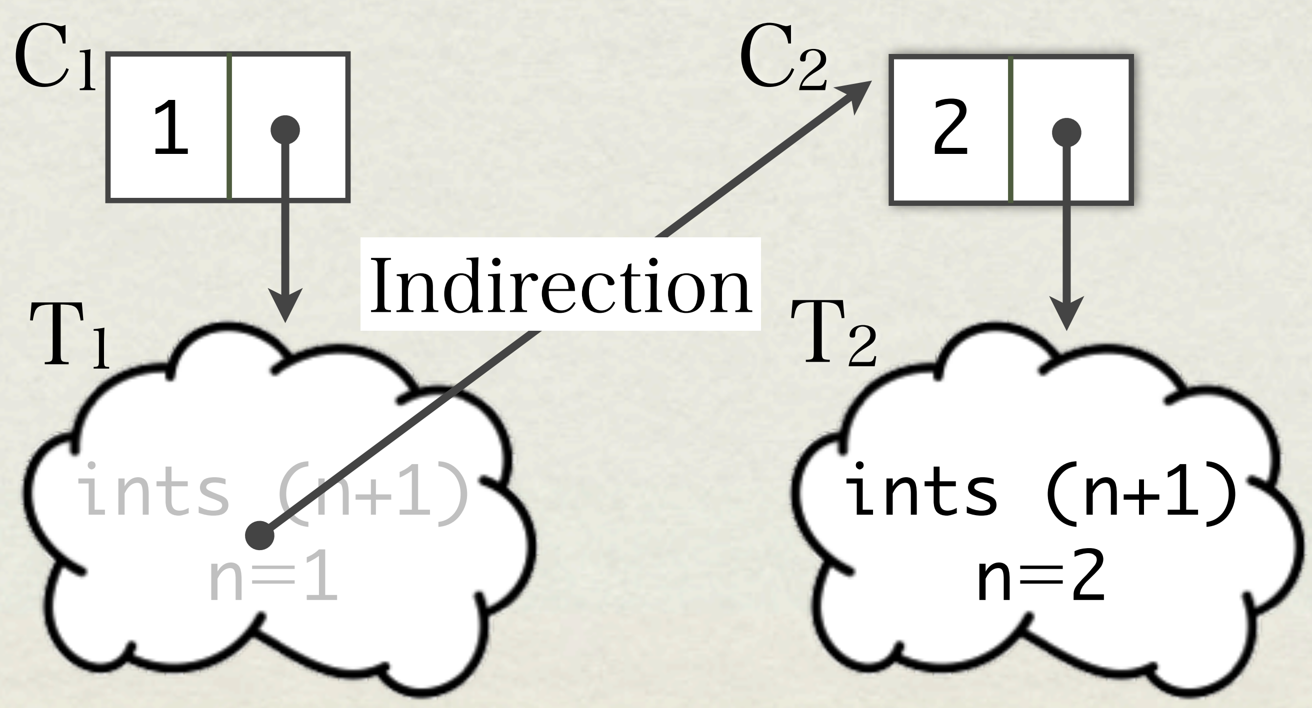
Without thunk reuse

- The data constructor Cons ":" delays its arguments

```
ints n = n : ints (n + 1)
```

ints 1 ⇒ 1 : T_1{ints (n+1)}{n=1}

⇒ 1 : 2 : T_2{ints (n+1)}{n=2}



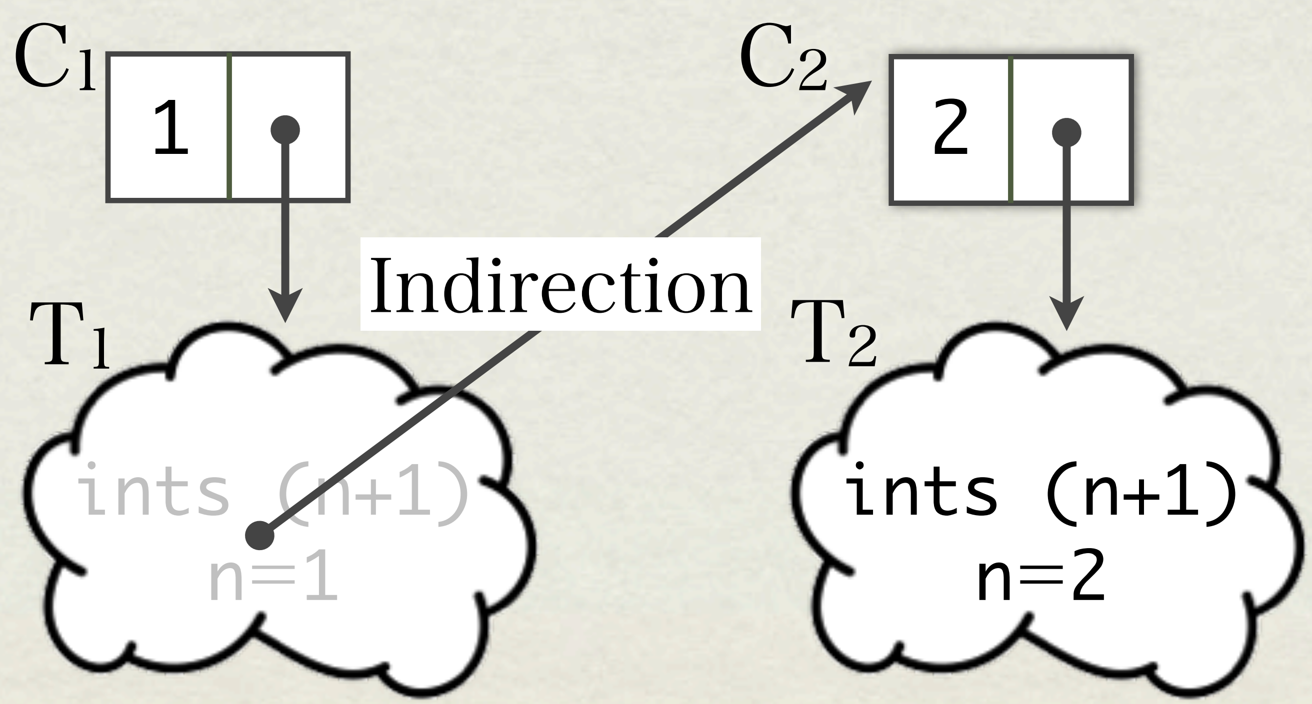
Without thunk reuse

- The data constructor Cons ":" delays its arguments

```
ints n = n : ints (n + 1)
```

ints 1 ⇒ 1 : T_1{ints (n+1)}{n=1}

⇒ 1 : 2 : T_2{ints (n+1)}{n=2}

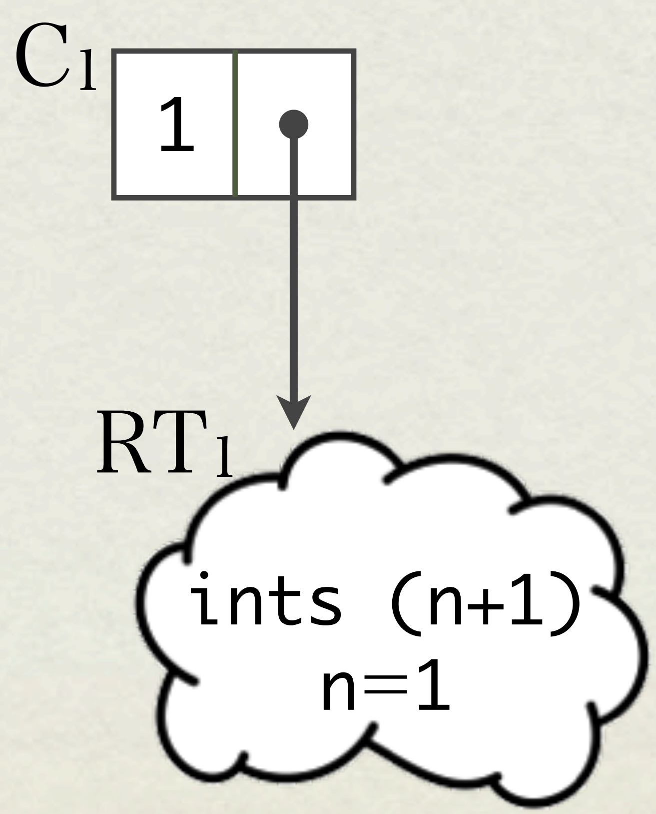


Structures of T_1 and T_2 are almost the same.

Thunk reuse

```
ints n = n : ints (n+1)
```

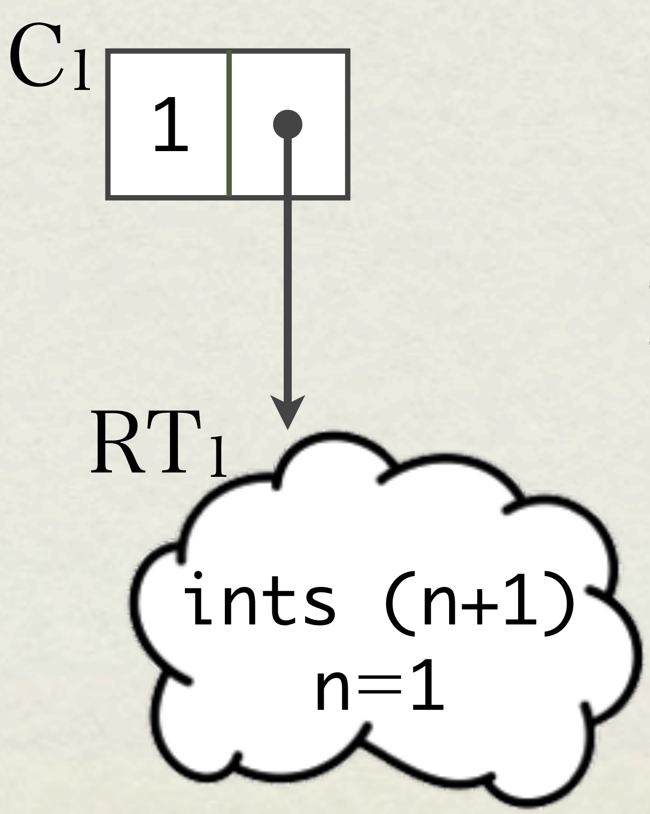
ints 1 ⇒ 1 : RT_1\{ints (n+1)\}\{n=1\}



Thunk reuse

```
ints n = n : ints (n+1)
```

`ints 1` ⇒ `1 :` $\underline{RT_1\{\texttt{ints (n+1)}\}\{n=1\}}$

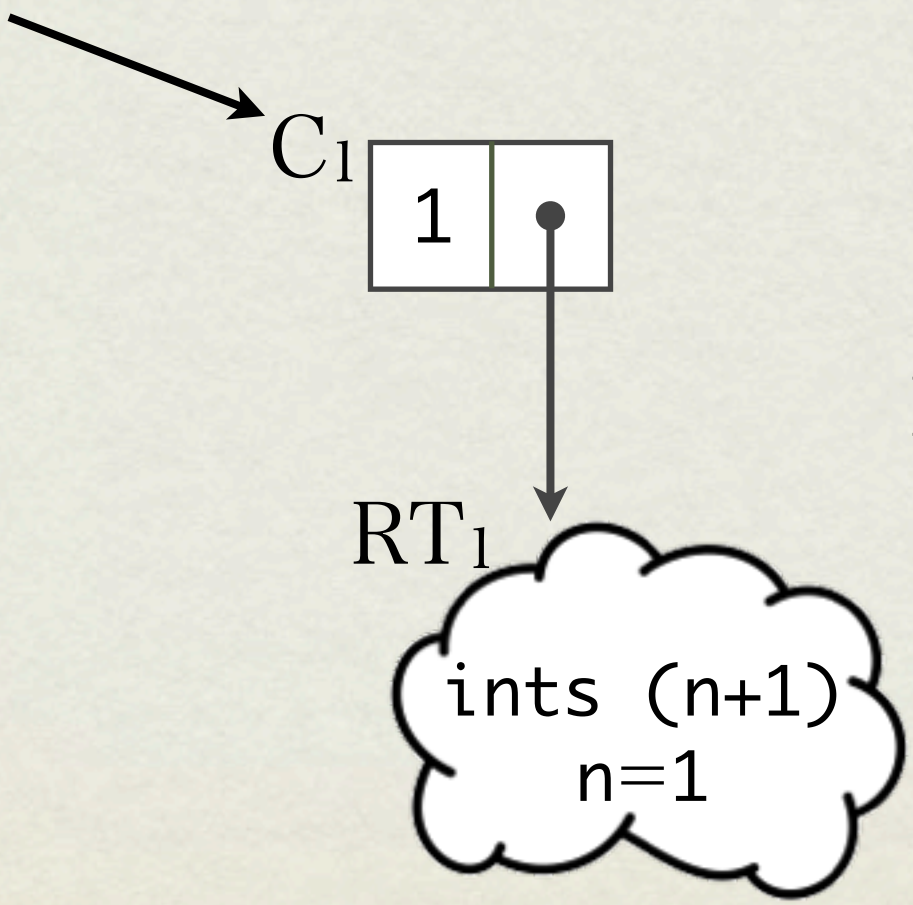


Forcing RT_1

Thunk reuse

```
ints n = n : ints (n+1)
```

`ints 1` ⇒ `1 :` $\underline{RT_1\{\texttt{ints (n+1)}\}\{\texttt{n=1}\}}$



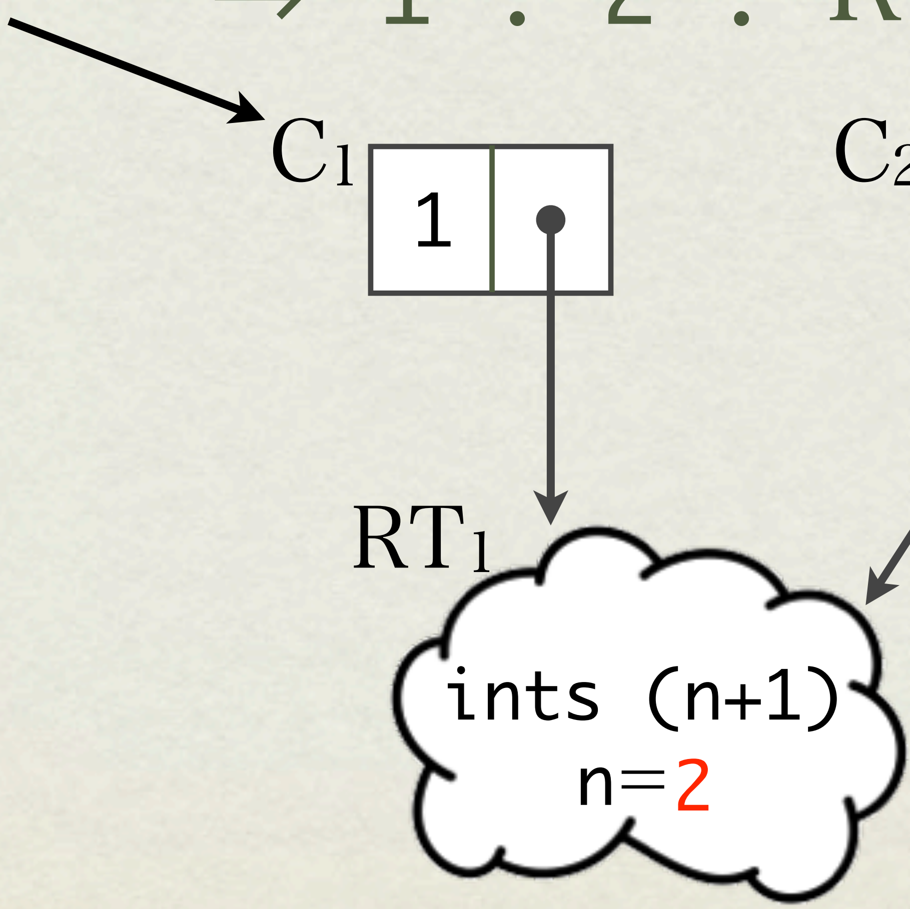
Forcing RT_1

Thunk reuse

`ints n = n : ints (n+1)`

ints 1 ⇒ 1 : RT_1{ints (n+1)}{n=1}

⇒ 1 : 2 : RT_1{ints(n+1)}{n=2}



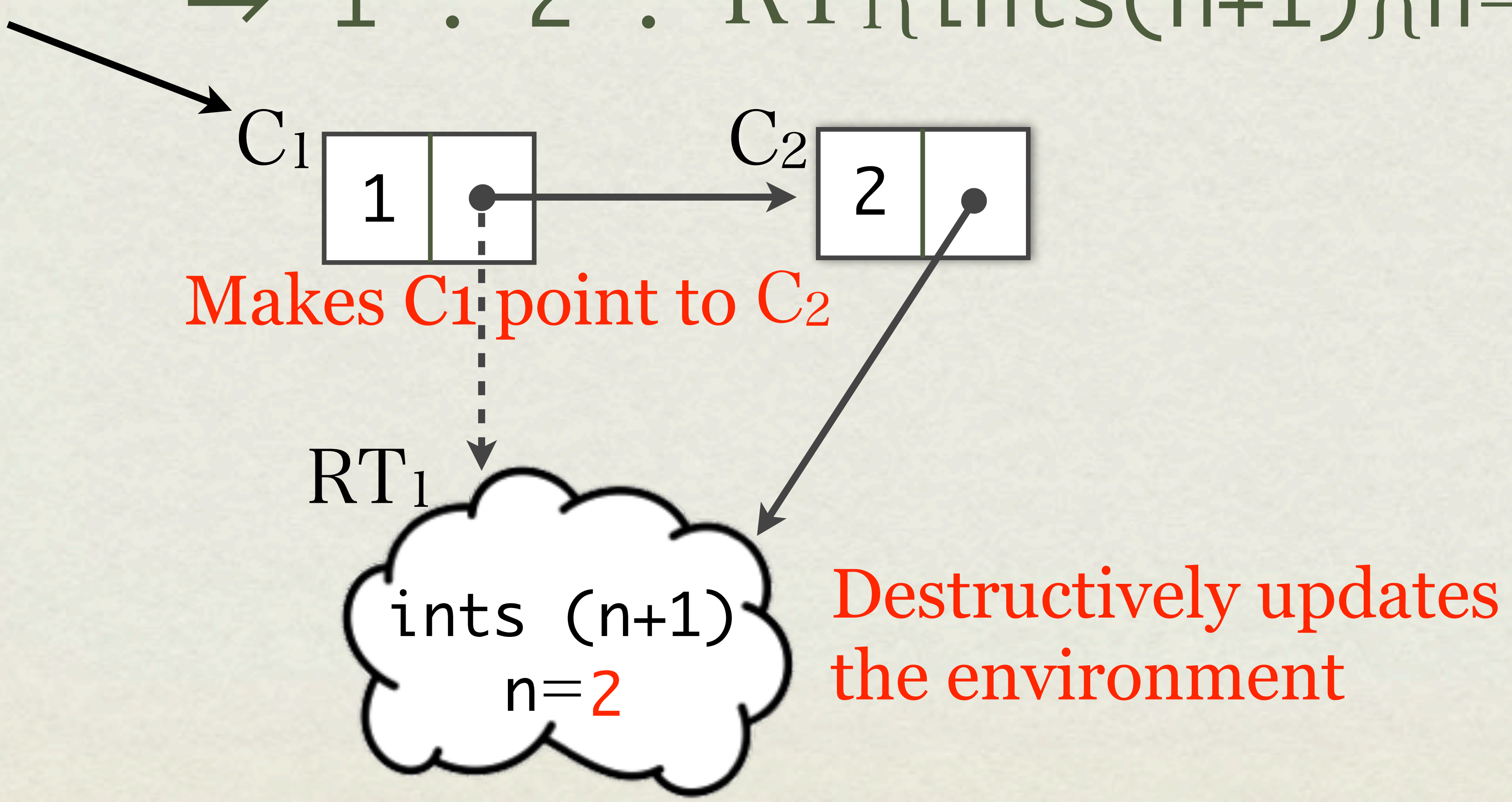
Destructively updates the environment

Thunk reuse

```
ints n = n : ints (n+1)
```

ints 1 ⇒ 1 : RT_1{ints (n+1)}{n=1}

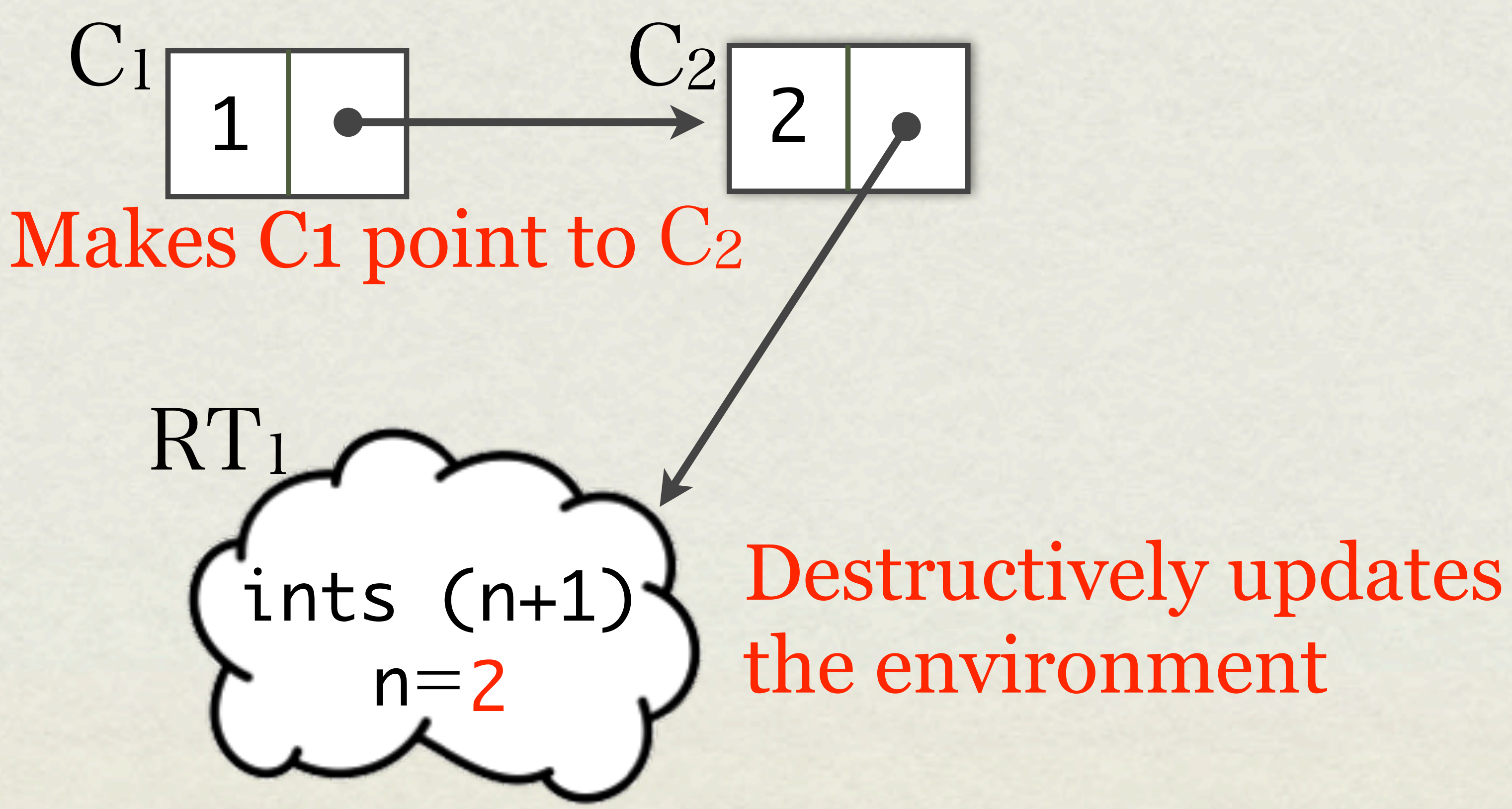
⇒ 1 : 2 : RT_1{ints(n+1)}{n=2}



Thunk reuse

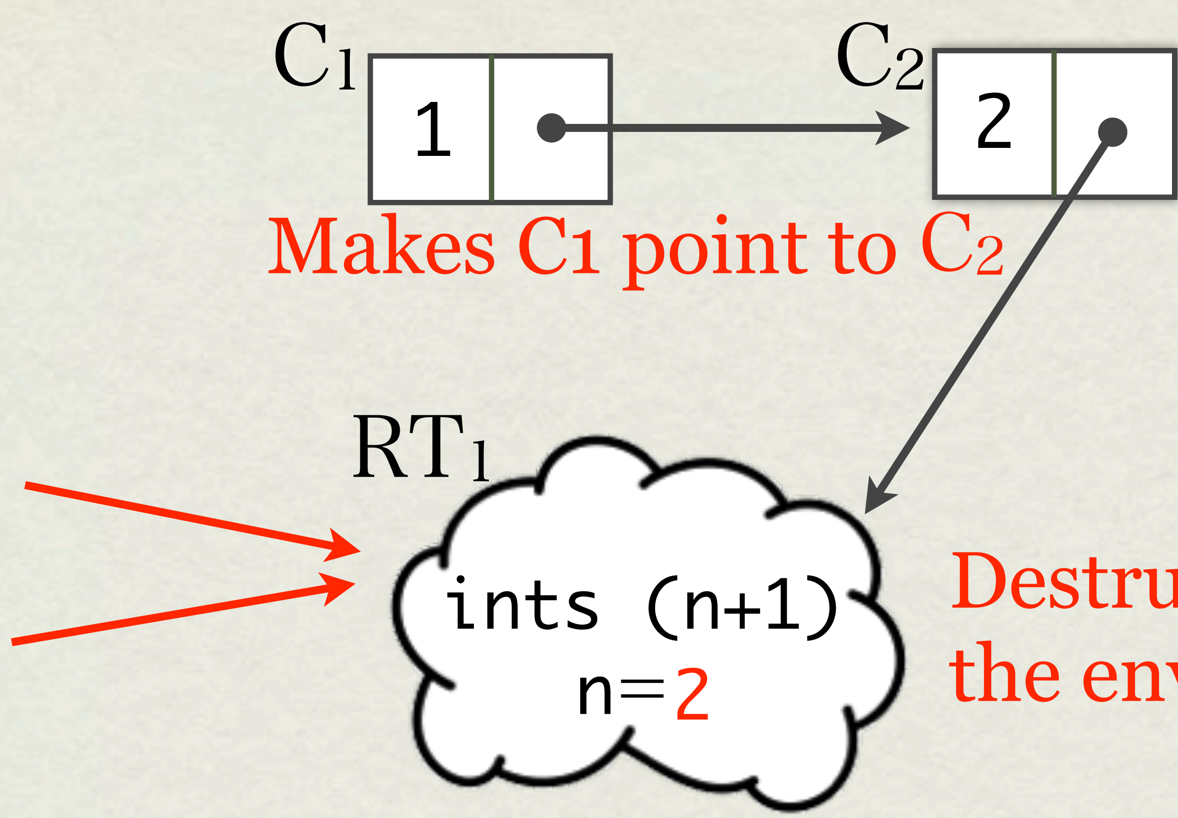
```
ints n = n : ints (n+1)
```

Suppresses the allocation of a new thunk



Singly referred condition

```
ints n = n : ints (n+1)
```



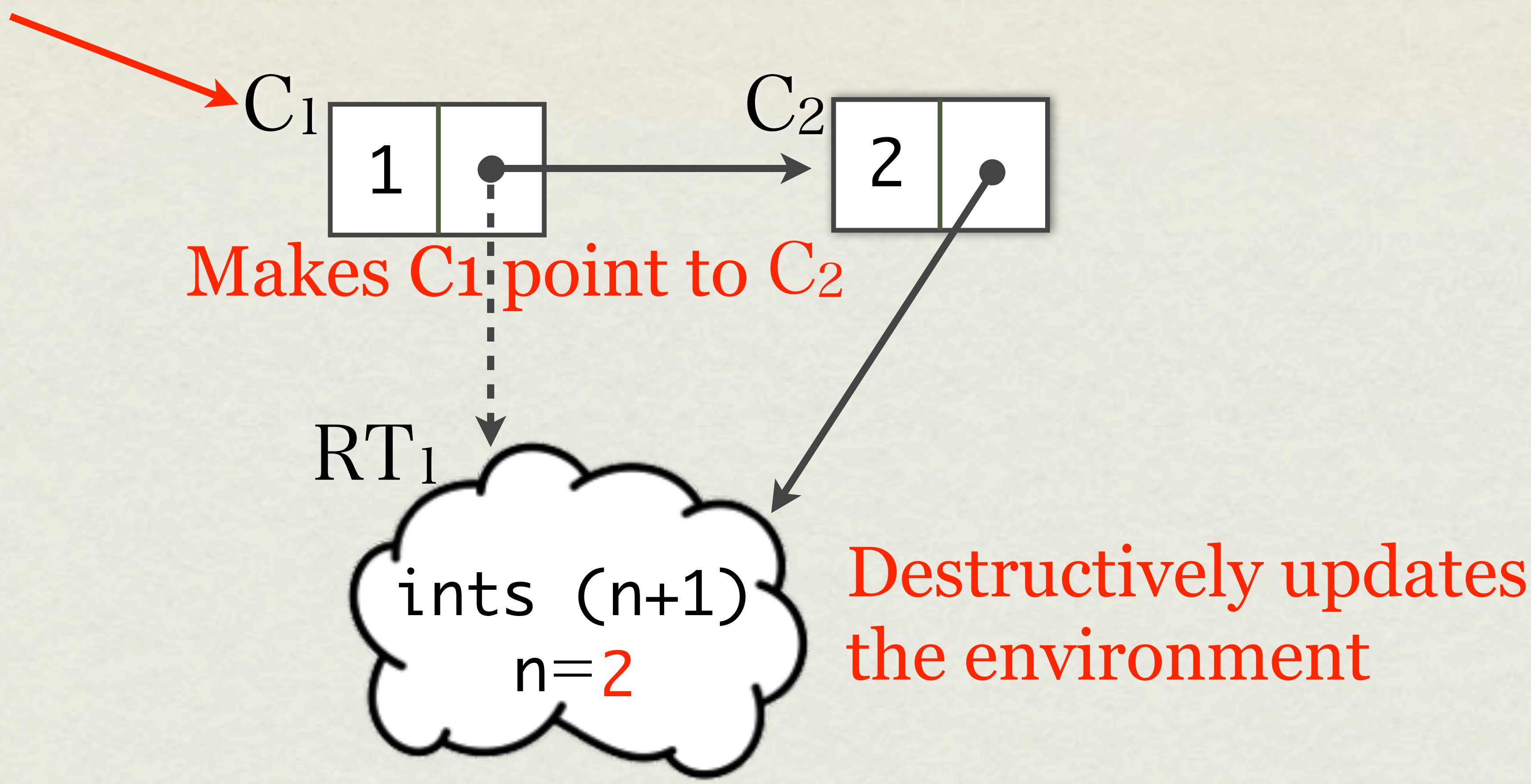
Destructively updates the environment

Singly referred condition

RT_1 should be referred to only by the tail part of C_2

Remembering the reference of C_1

```
ints n = n : ints (n+1)
```



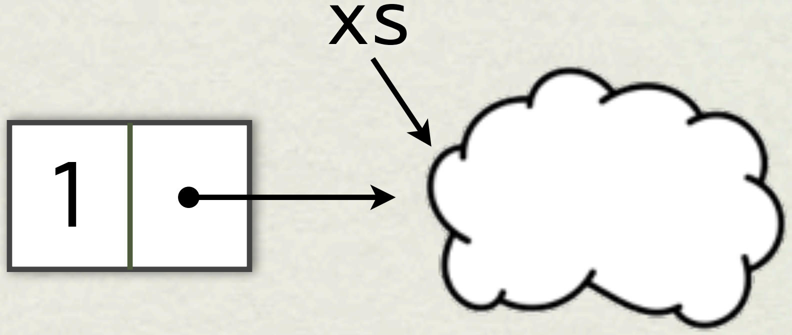
Before forcing RT1, we have to remember the reference of C_1,

because we are going to destructively update the C_1's tail

Our observation

Pattern matching can increase the number of references to a thunk

```
case (ints 1) of
  x:xs -> .. xs ..
```



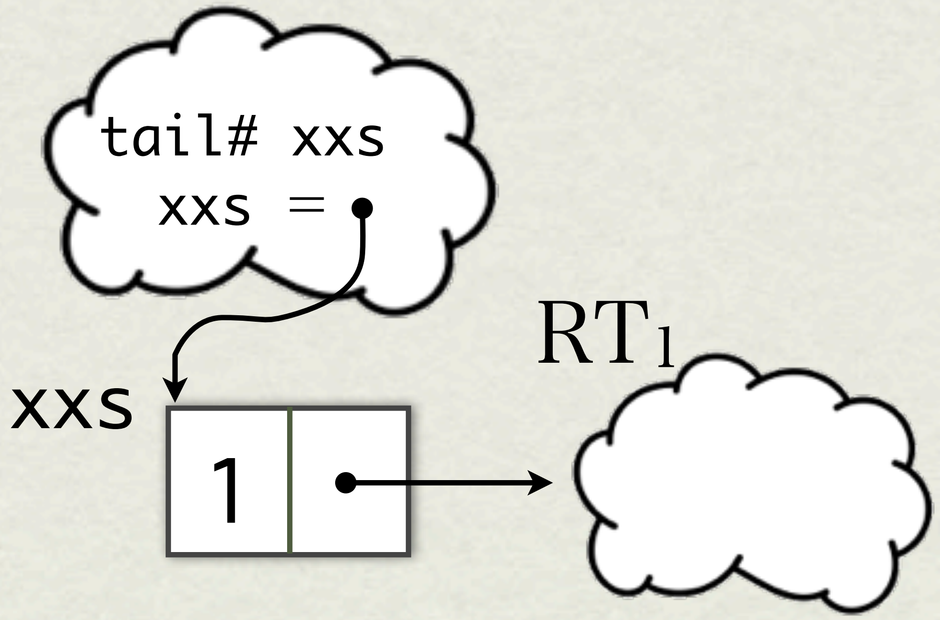
Transforming pattern matching

We replace each occurrence of `xs` with `(tail# xxs)` to avoid the duplication of references

```
case (ints 1) of
 x:xs -> .. xs ..
```

➡

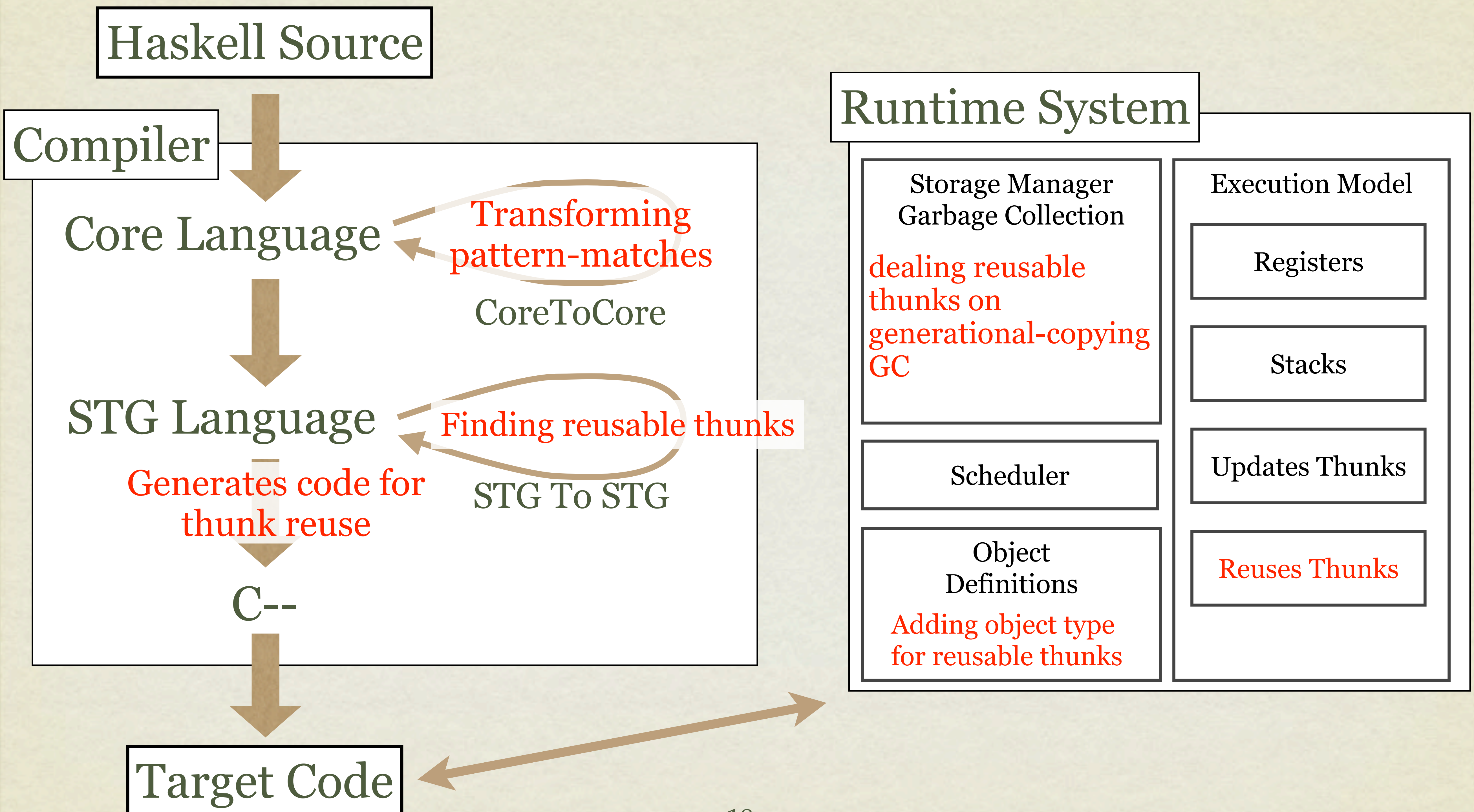
```
case (ints 1) of
  xxs@(x:_) -> .. (tail# xxs) ..
```



Evaluation of `(tail# xxs)` leads to forcing RT1

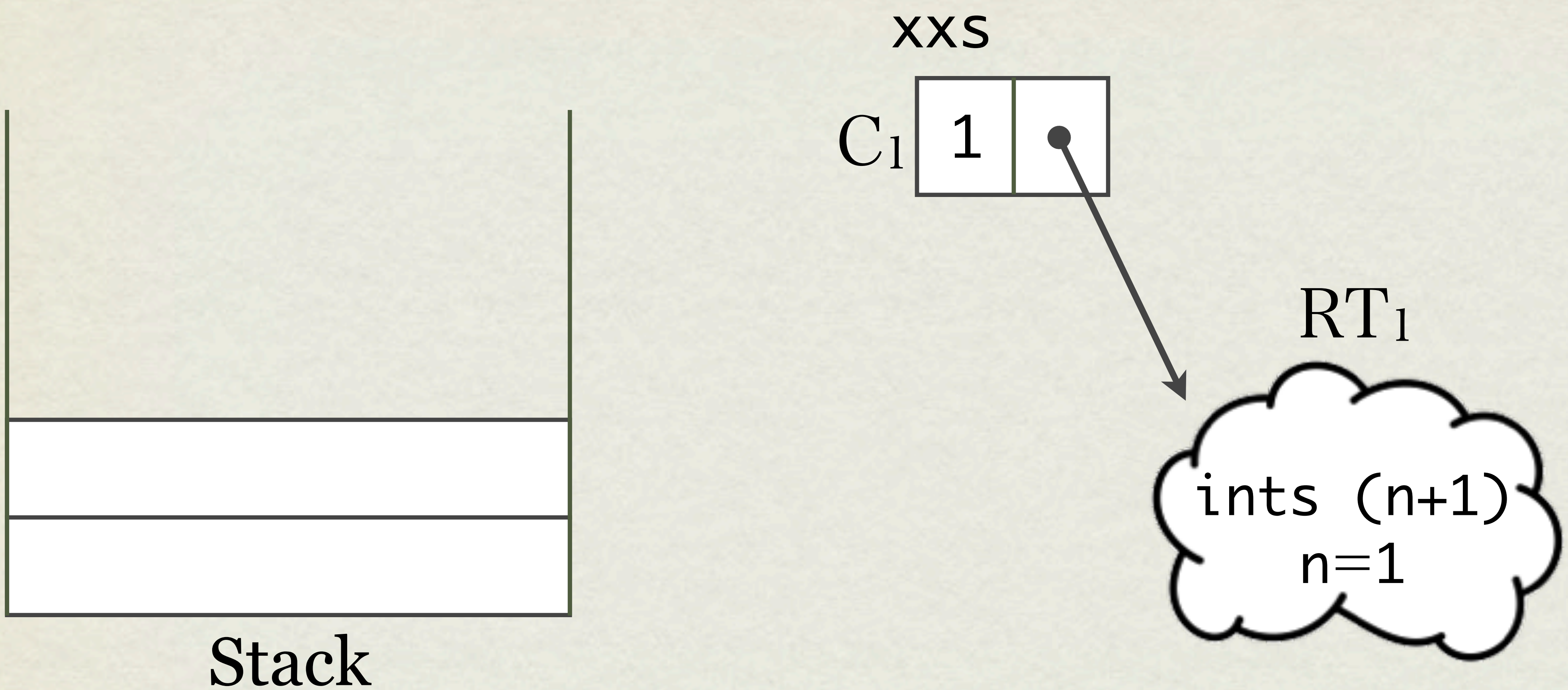
`(tail# xxs)` is almost the same as `(tail xxs)` except that `(tail# xxs)` remembers the address of `xxs`

Implementing our Idea to GHC



Thunk reuse in GHC execution model

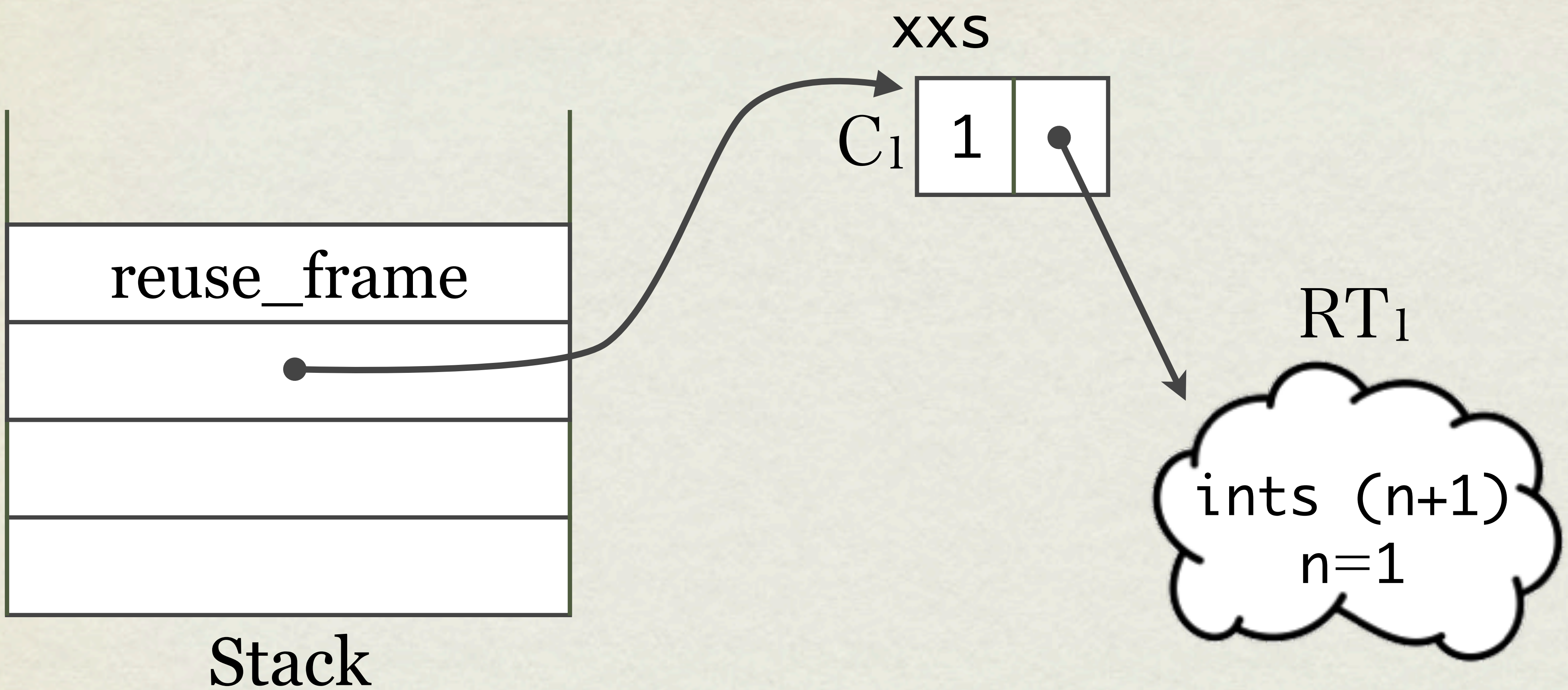
```
case (ints 1) of
  xxs@(x:_) -> .. (tail# xxs) ..
```



This process resembles updating thunks.

Thunk reuse in GHC execution model

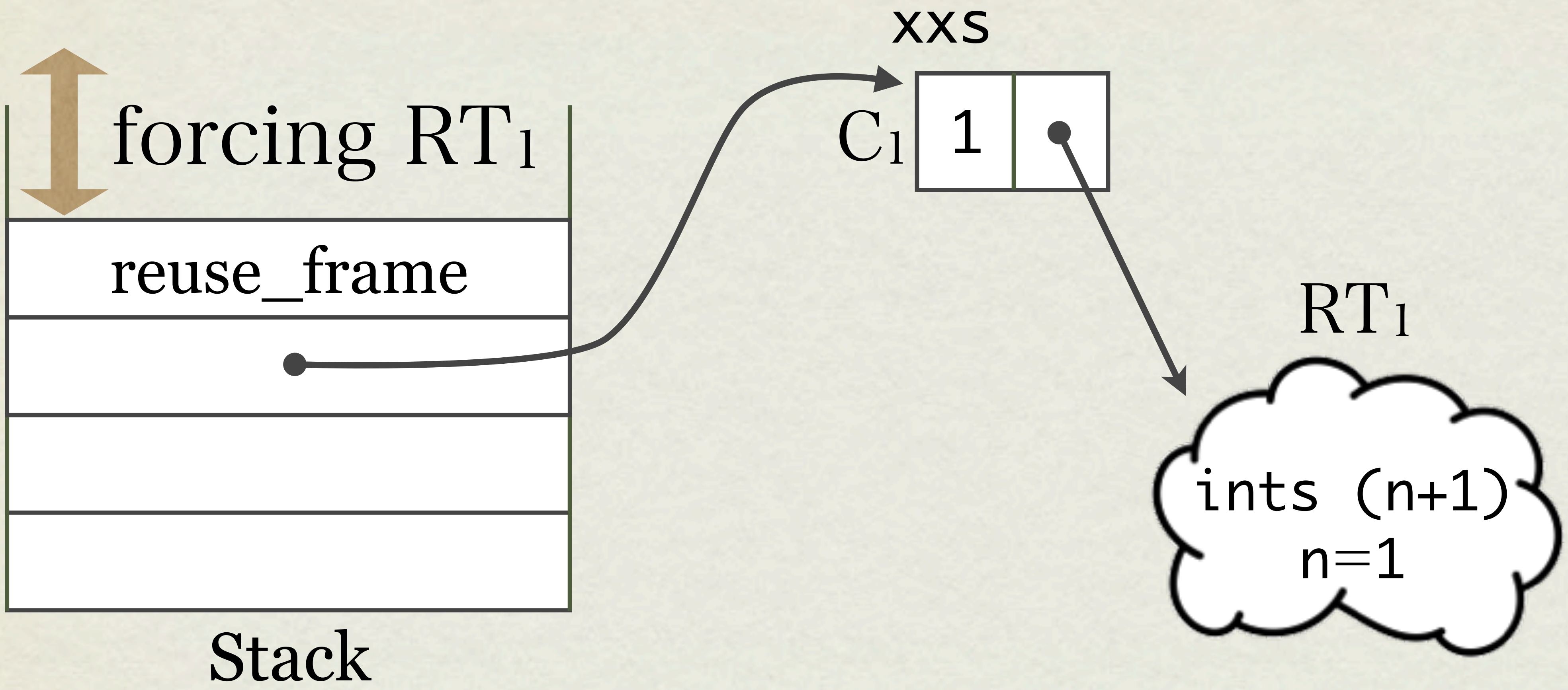
```
case (ints 1) of
  xxs@(x:_) -> .. (tail# xxs) ..
```



`tail#` pushes `xxs` and reuse_frame onto the stack.

Thunk reuse in GHC execution model

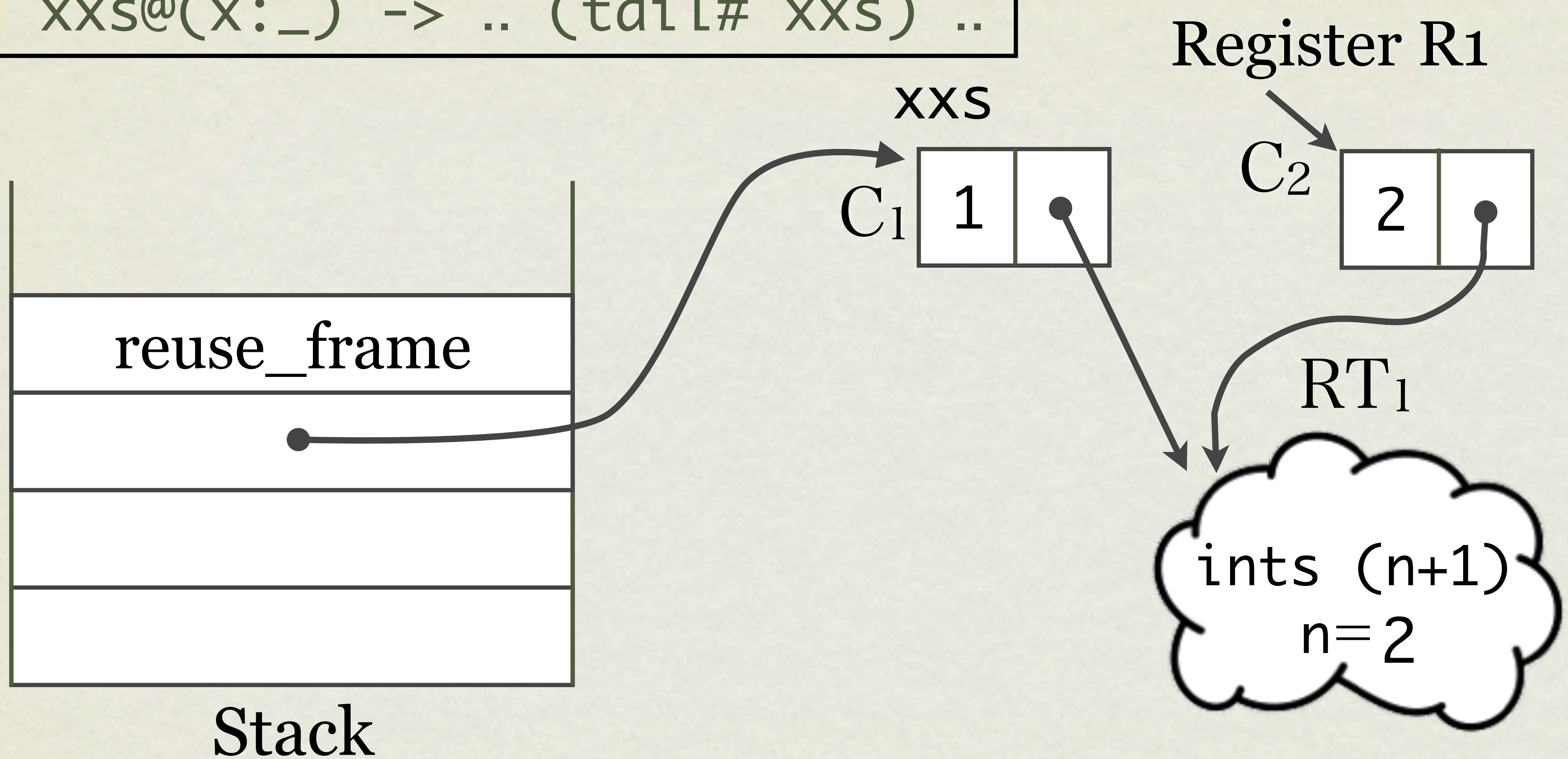
```
case (ints 1) of
  xxs@(x:_) -> .. (tail# xxs) ..
```



RT_1 is forced

Thunk reuse in GHC execution model

```
case (ints 1) of
  xxs@(x:_) -> .. (tail# xxs) ..
```

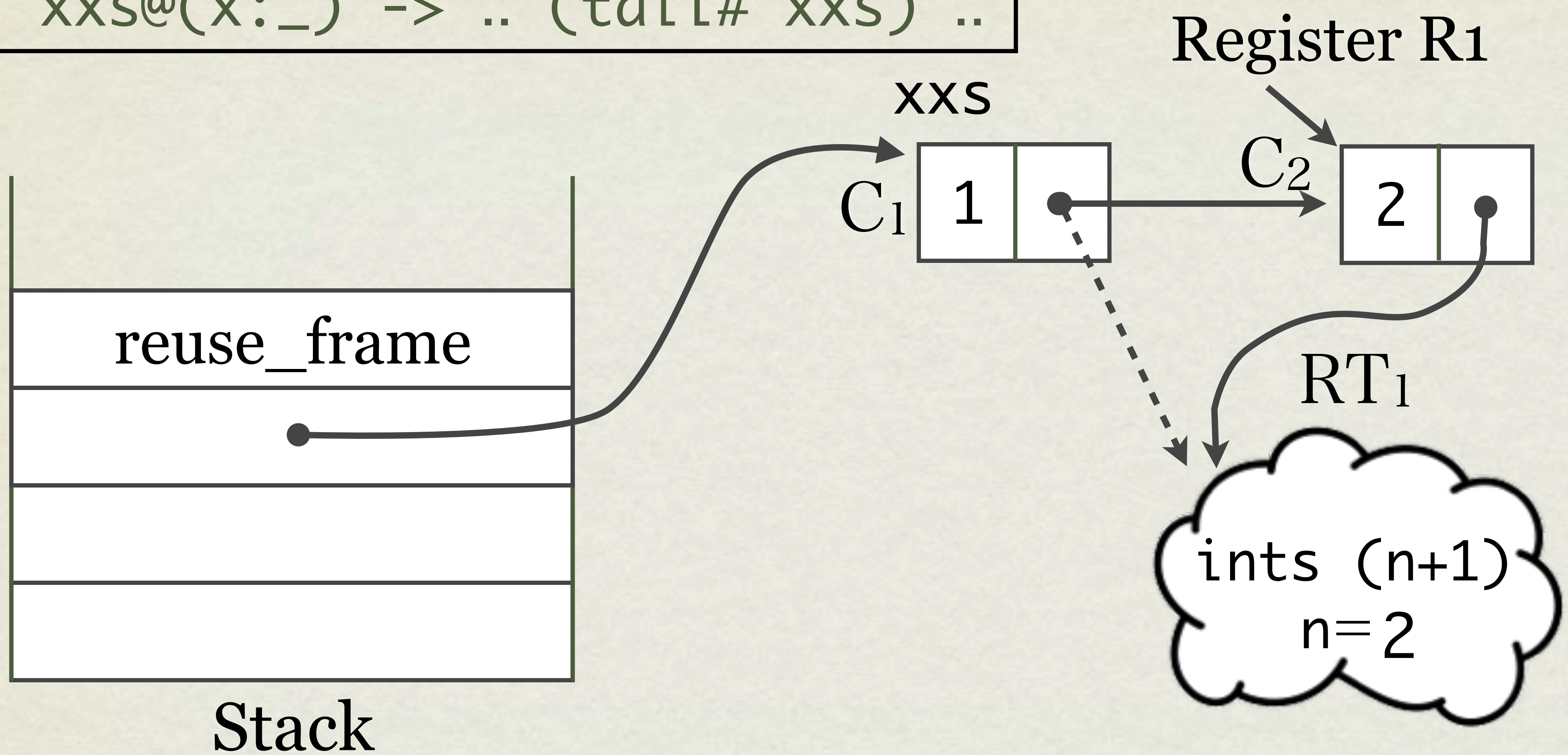


RT_1 is forced and as a result C_2 is obtained.

RT_1 is reused as the delayed computation at the tail of C_2

Thunk reuse in GHC execution model

```
case (ints 1) of
  xxs@(x:_) -> .. (tail# xxs) ..
```

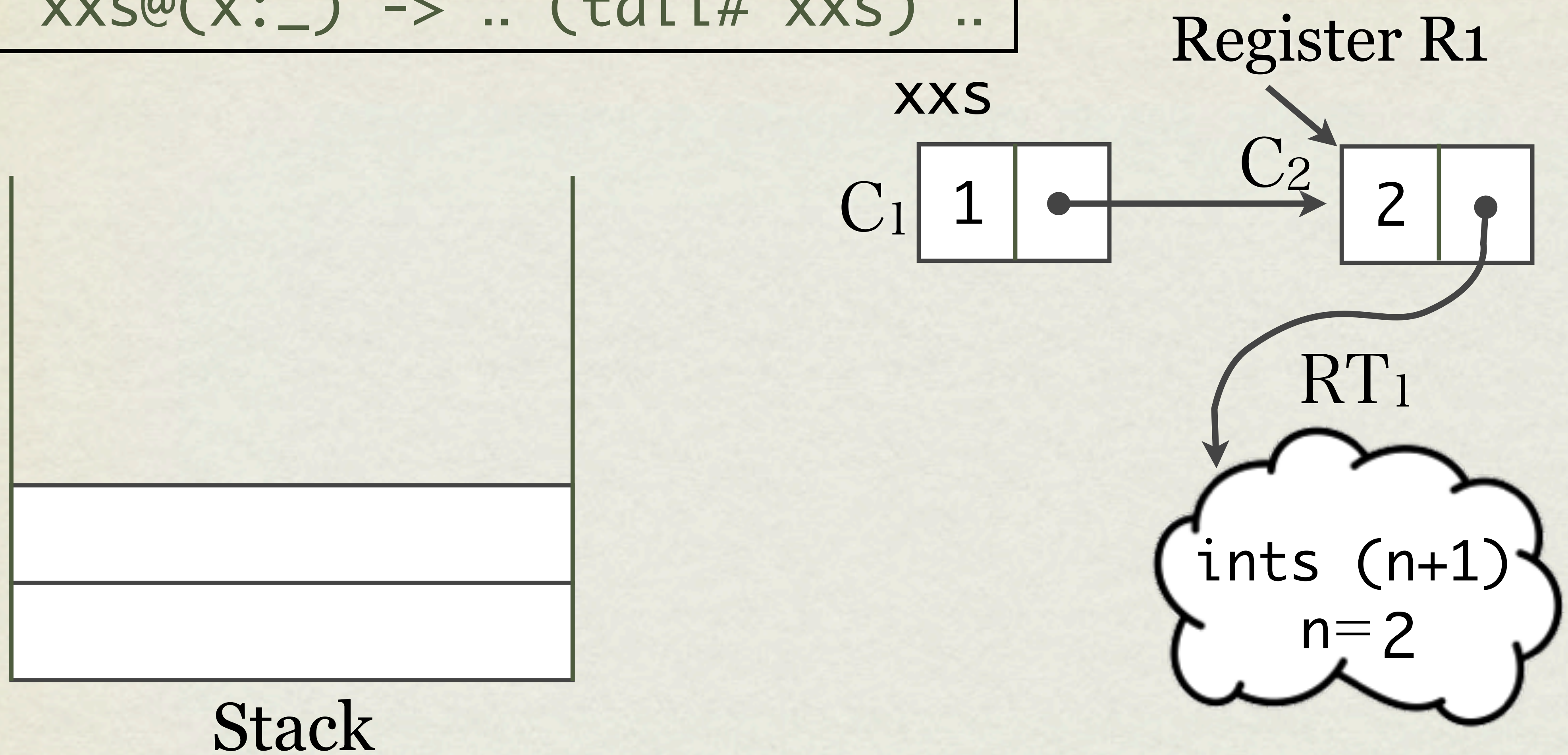


reuse_frame overwrites the tail of C_1 with a pointer to C_2.

C_1's address can be obtained from the stack.

Thunk reuse in GHC execution model

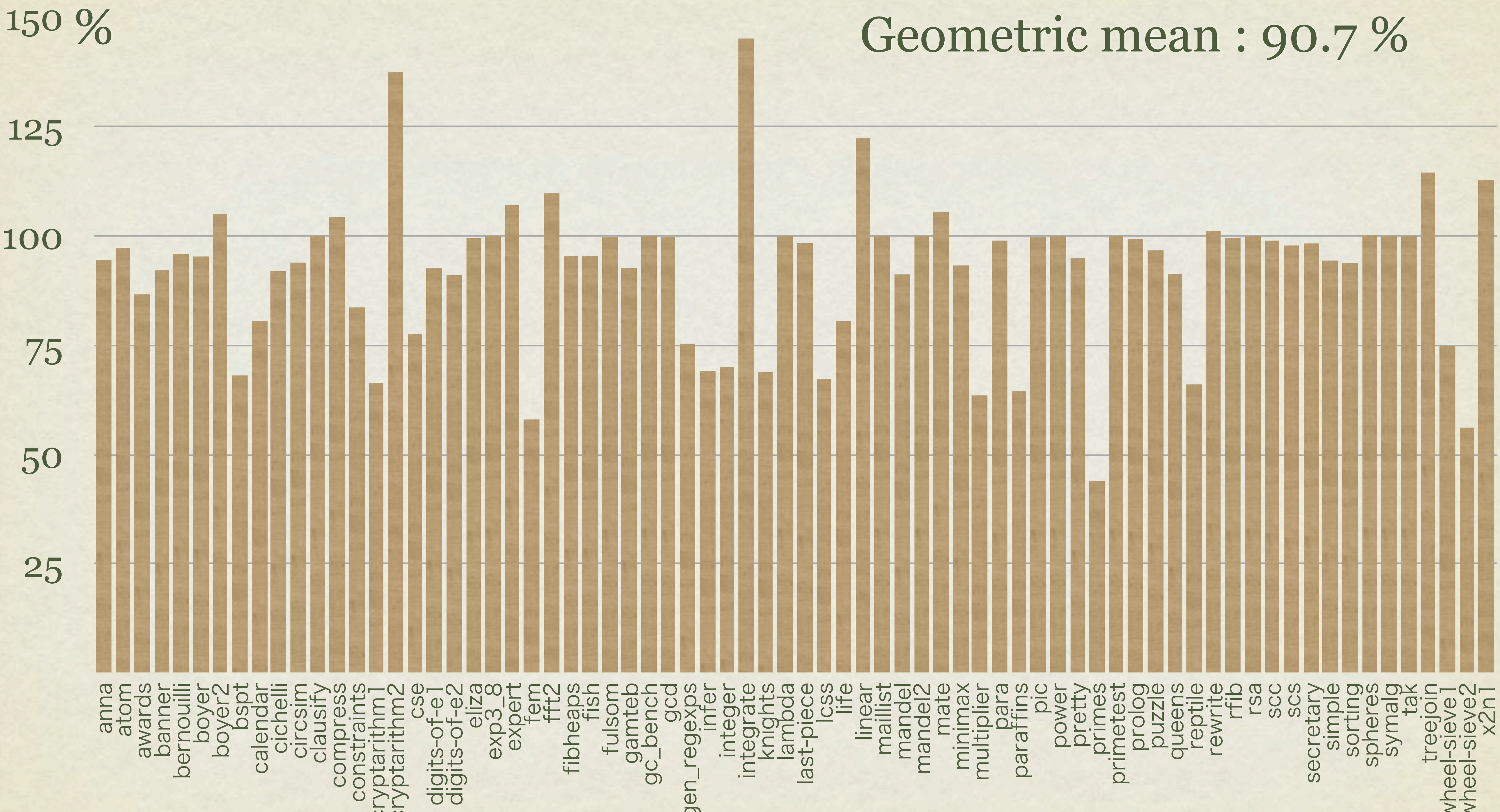
```
case (ints 1) of
  xxs@(x:_) -> .. (tail# xxs) ..
```



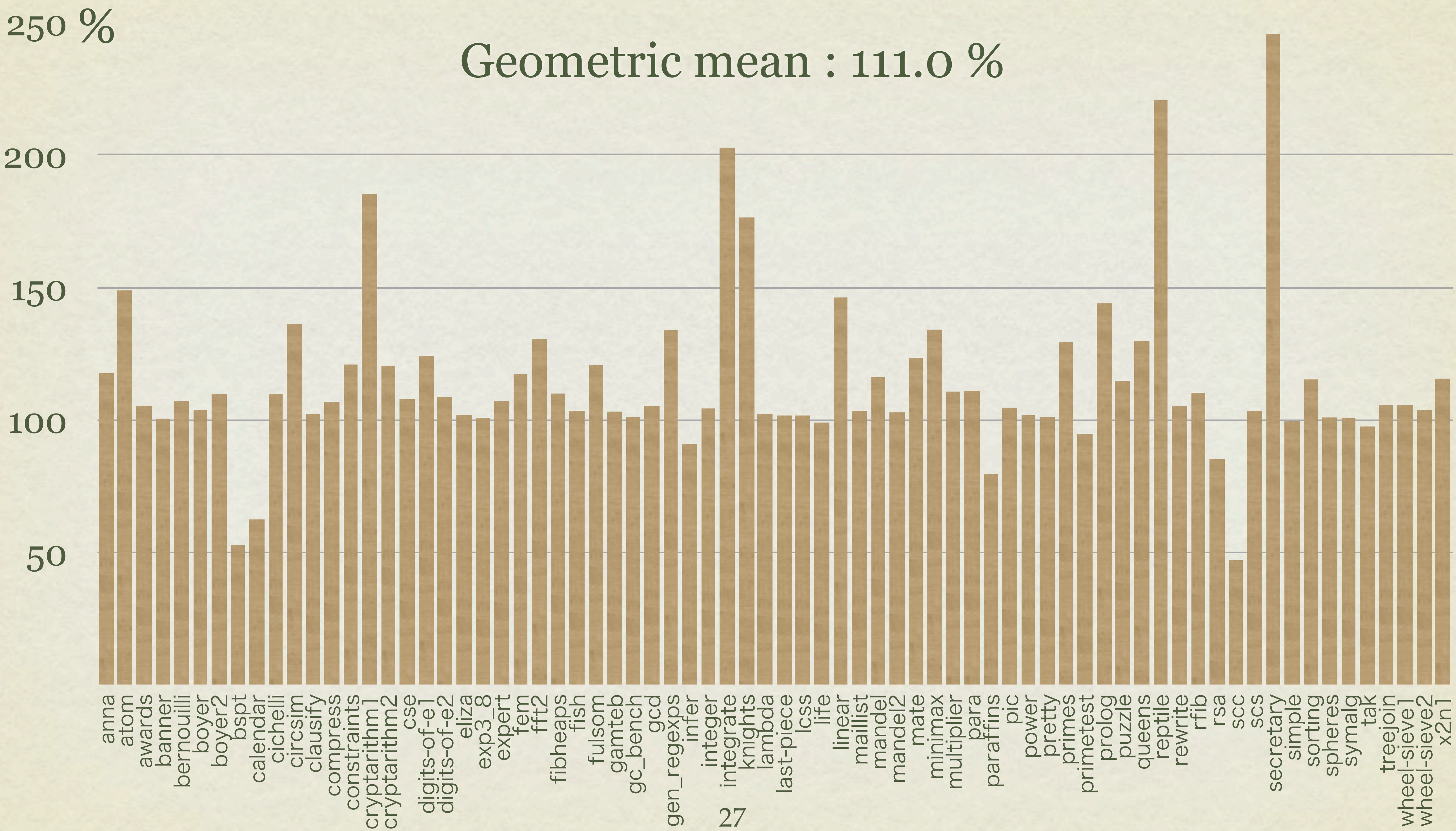
Experiments

- `nofib` benchmark
  - imaginary, spectral, real
- GHC 7.0.3
- AMD Opteron CPU, 8GB main memory, Linux 2.6.32
- Compiled with -O2 flag
- Measured by GHC's statistic option -S

Total memory allocations



Execution time



Result

- Total memory allocations
  - Thunk reuse is effective in many programs except programs which allocate thunks for `tail#`
- Execution time
  - In many programs, the execution time is between 100% and 110%, compared to the original GHC

Analysis on execution time

- Advantage
  - Time for memory allocations
  - The number of GC cycles
- Disadvantage
  - Overhead of `tail#`
  - Overhead of checking reusability of thunks

Summary

- We have proposed a new implementation technique to suppress memory allocations by reusing thunks
- On current our implementation, total allocation is reduced in many case, while extra execution time is necessary

We need advices

- We should improve execution time
  - Elimination of the overhead of `tail#`
    - Can we use the technique of *pointer tagging* instead of allocating a thunk for `tail#` ?
  - Further optimization for self recursive functions such as `map`

```
map f [] = []
map f (x:xs) = f x : map' f xs
  where map' f [] = []
        map' f (x:xs) = f x : map' f xs
```

    - We have to add new functions in STGtoSTG path, but we don't know how to do that

- Modifying GHC is a very hard task for me

takano@coma-systems.com