# Counting Immutable Beans

**Reference Counting Optimized for Purely Functional Programming**

Sebastian Ullrich - KIT - Germany
Leonardo de Moura - MSR - USA

Karlsruhe Institute of Technology

# LEAN THEOREM PROVER

- Pure functional language; Strict; Dependent types
- Meta programming: extend Lean using Lean
- Applications:
  - Formal Abstracts Project - Tom Hales
  - Perfectoid Spaces Project

    Kevin Buzzard, Johan Commelin, and Patrick Massot
  - Education (CMU, Imperial College, …)
  - Lean Forward - Jasmin Blanchette
  - Protocol Verification (Galois)
  - SQL query equivalence (UW)
  - IMO Grand Challenge (MSR)
  - AliveInLean (MSR)
  - 6 papers at ITP 2019

LEAN

# an extensible compiler

**Programming language**

Lean3 users write metaprograms/tactics in Lean

Examples: ring solver, conductive predicates, superposition prover, transfer tactic, …

We are implementing Lean4 in Lean itself.

All subsystems can be extended: parser, elaborator, compiler, …

New compiler is already outperforming Haskell and OCaml.

Proofs for performance and profit.

A better value proposition: use proofs for obtaining more efficient code.

# The **return of reference counting**

- Most compilers for functional languages (OCaml, GHC, …) use tracing GC
- RC is simple to implement.
- Easy to support multi-threading programs.
- Destructive updates when reference count = 1.
  - It is a known optimization for big objects (e.g., arrays).
    Array.set : Array a -> Index -> a -> Array a
  - We demonstrate it is also relevant for small objects.
- In languages like Coq and **Lean**, we do not have cycles.
- Easy to interface with C, C++ and Rust.

# Resurrection hypothesis

**Many objects die just before the creation of an object of the same kind.**

Examples:

- List.map : List a -> (a -> b) -> List b
- Compiler applies transformations to expressions.
- Proof assistant rewrites/simplifies formulas.
- Updates to functional data structures such as red black trees.
- List zipper

$$goForward\,([], bs) = ([], bs)$$
$$goForward\,(x : xs, bs) = (xs, x : bs)$$

# Contributions

- Approach for reusing memory: **small** and big values.
  Big values are often nested into small ones.

- Inference procedure for borrowed references (à la Swift)

- Simple and efficient scheme for performing atomic RC updates in multi-threaded programs.

- Implementation and experimental evaluation.

- https://github.com/leanprover/lean4

# Reference counts

- Each heap-allocated object has a reference count.
- We can view the counter as a collection of tokens.
- The **inc** instruction creates a new token.
- The **dec** instruction consumes a token.
- When a function takes an argument as an **owned** reference, it must consume one of its tokens.
- A function may consume an owned reference by using dec, passing it to another function, or storing it in a newly allocated value.

# Owned references: examples

$$id\ x = \mathbf{ret}\ x$$

$$mkPairOf\ x = \mathbf{inc}\ x;\ \mathbf{let}\ p = Pair\ x\ x;\ \mathbf{ret}\ p$$

$$fst\ x\ y = \mathbf{dec}\ y;\ \mathbf{ret}\ x$$

# Borrowed references

- If *xs* is an owned reference

$$\begin{aligned}&isNil\ xs = \textbf{case}\ xs\ \textbf{of}\\&\quad(Nil \longrightarrow \textbf{dec}\ xs\,;\ \textbf{ret}\ true)\\&\quad(Cons \longrightarrow \textbf{dec}\ xs\,;\ \textbf{ret}\ false)\end{aligned}$$

- If *xs* is a borrowed reference

$$isNil\ xs = \textbf{case}\ xs\ \textbf{of}\ (Nil \longrightarrow \textbf{ret}\ true)\ (Cons \longrightarrow \textbf{ret}\ false)$$

# Owned vs Borrowed

- Transformers and constructors **own** references.
- Inspectors and visitors **borrow** references.
- Remark: it is not safe to destructively update borrowed references even when RC = 1

# Reusing small objects

$$map\ f\ [] = []$$
$$map\ f\ (x : xs) = (f\ x) : (map\ f\ xs)$$

**First attempt**

$$
\begin{array}{l}
map\ f\ xs = \mathbf{case}\ xs\ \mathbf{of} \\
\quad (\mathbf{ret}\ xs) \\
\quad (\mathbf{let}\ x = \mathbf{proj}_1\ xs\,;\ \mathbf{inc}\ x\,;\ \mathbf{let}\ s = \mathbf{proj}_2\ xs\,;\ \mathbf{inc}\ s\,; \\
\quad\ \ \mathbf{let}\ y = f\ x\,;\ \mathbf{let}\ ys = map\ f\ s\,; \\
\quad\ \ \mathbf{let}\ r = (\mathbf{reuse}\ xs\ \mathbf{in}\ \mathbf{ctor}_2\ y\ ys)\,;\ \mathbf{ret}\ r)
\end{array}
$$

# Reusing small objects

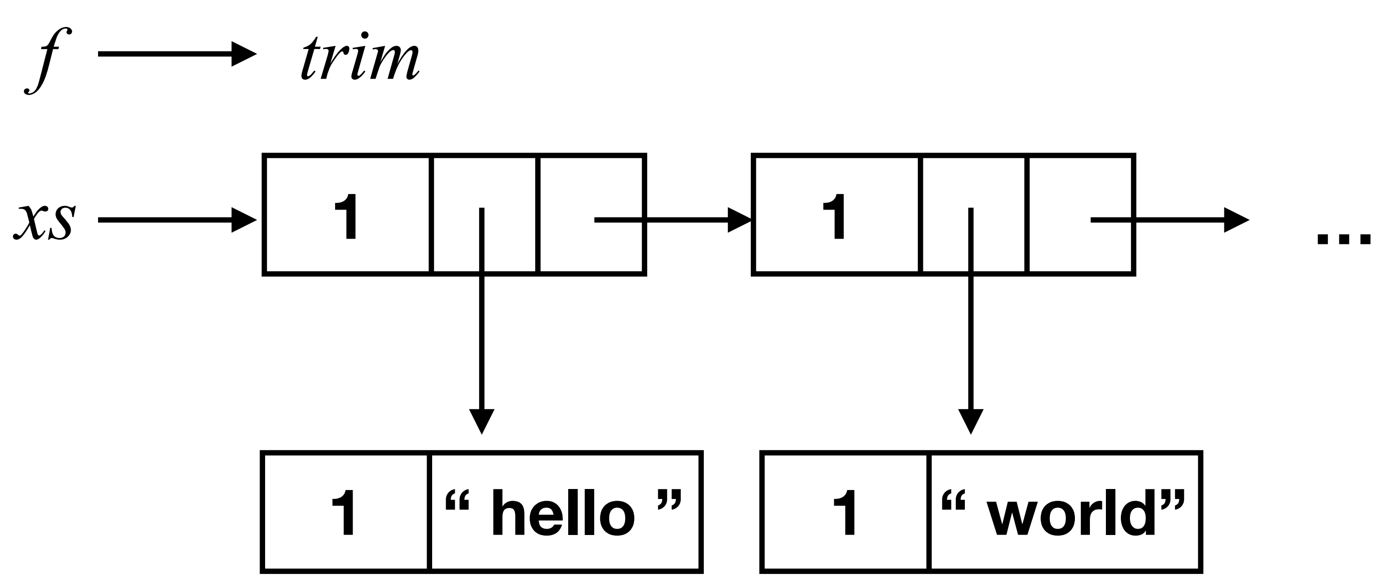
$map\ f\ xs = \textbf{case}\ xs\ \textbf{of}$

$\quad(\textbf{ret}\ xs)$

$\quad(\textbf{let}\ x = \textbf{proj}_1\ xs\,;\ \textbf{inc}\ x\,;\ \textbf{let}\ s = \textbf{proj}_2\ xs\,;\ \textbf{inc}\ s\,;$

$\quad\ \textbf{let}\ y = f\ x\,;\ \textbf{let}\ ys = map\ f\ s\,;$

$\quad\ \textbf{let}\ r = (\textbf{reuse}\ xs\ \textbf{in}\ \textbf{ctor}_2\ y\ ys)\,;\ \textbf{ret}\ r)$



# Reusing small objects

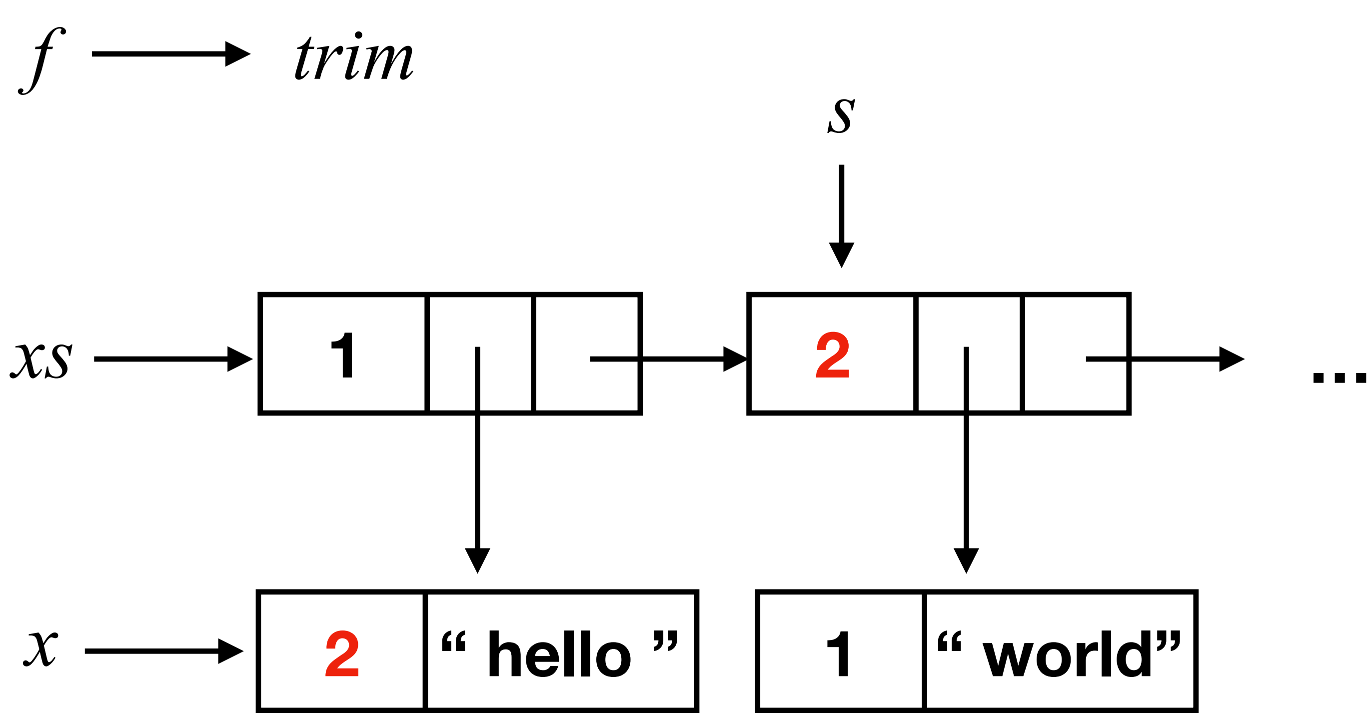
$map\ f\ xs = \mathbf{case}\ xs\ \mathbf{of}$

$\quad(\mathbf{ret}\ xs)$

$\quad(\mathbf{let}\ x = \mathbf{proj}_1\ xs\,;\ \mathbf{inc}\ x\,;\ \mathbf{let}\ s = \mathbf{proj}_2\ xs\,;\ \mathbf{inc}\ s\,;$

$\quad\ \mathbf{let}\ y = f\ x\,;\ \mathbf{let}\ ys = map\ f\ s\,;$

$\quad\ \mathbf{let}\ r = (\mathbf{reuse}\ xs\ \mathbf{in}\ \mathbf{ctor}_2\ y\ ys)\,;\ \mathbf{ret}\ r)$



# Reusing small objects

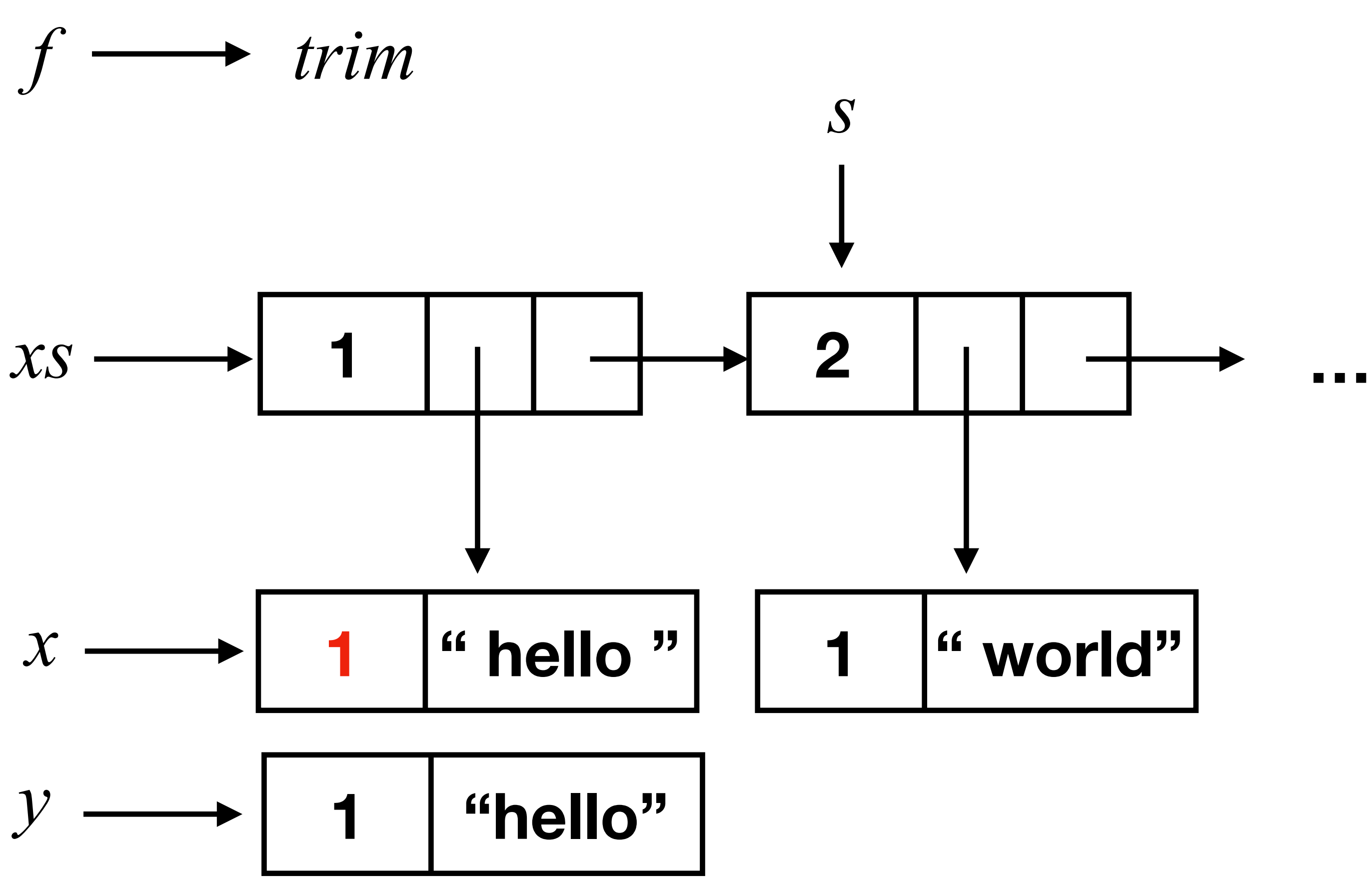
$map\ f\ xs = \mathbf{case}\ xs\ \mathbf{of}$

$(\mathbf{ret}\ xs)$

$(\mathbf{let}\ x = \mathbf{proj}_1\ xs\,;\ \mathbf{inc}\ x\,;\ \mathbf{let}\ s = \mathbf{proj}_2\ xs\,;\ \mathbf{inc}\ s\,;$

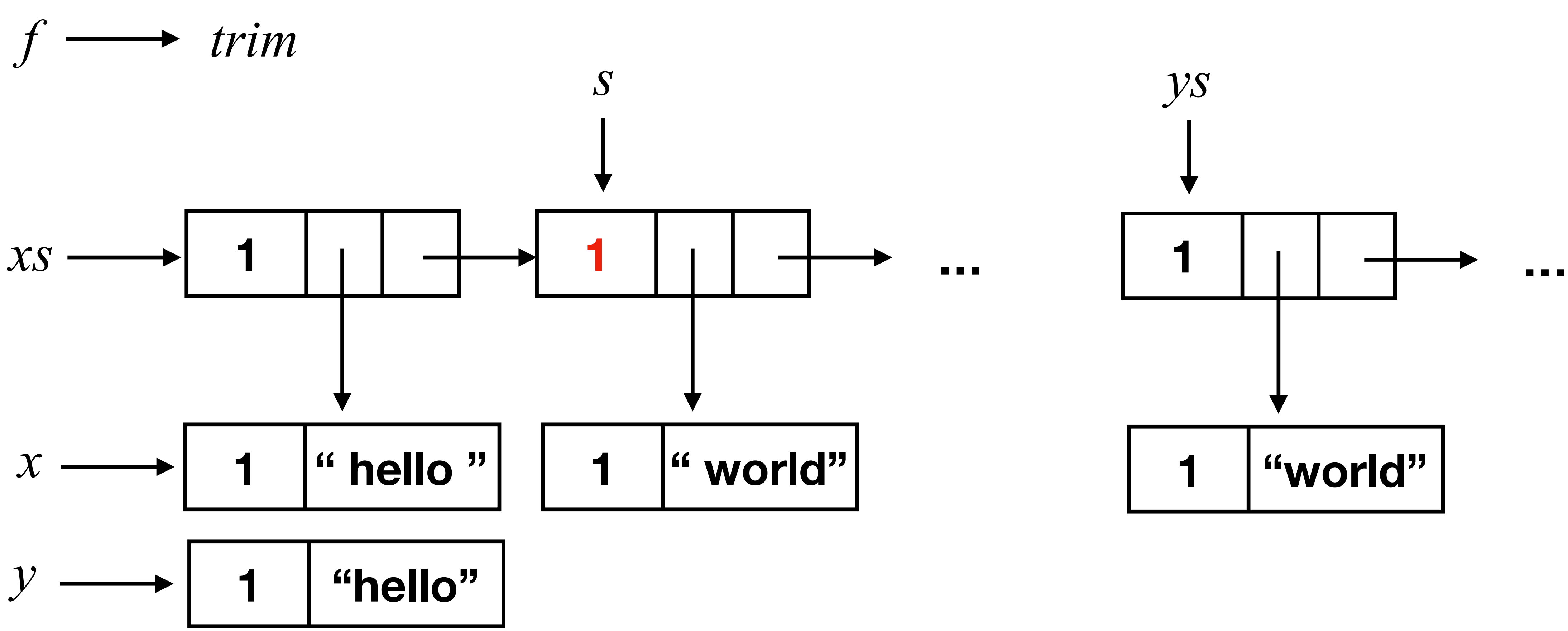
$\mathbf{let}\ y = f\ x\,;\ \mathbf{let}\ ys = map\ f\ s\,;$

$\mathbf{let}\ r = (\mathbf{reuse}\ xs\ \mathbf{in}\ \mathbf{ctor}_2\ y\ ys)\,;\ \mathbf{ret}\ r)$



# Reusing small objects

$$
\begin{array}{l}
map\ f\ xs = \mathbf{case}\ xs\ \mathbf{of} \\
\quad (\mathbf{ret}\ xs) \\
\quad (\mathbf{let}\ x = \mathbf{proj}_1\ xs\,;\ \mathbf{inc}\ x\,;\ \mathbf{let}\ s = \mathbf{proj}_2\ xs\,;\ \mathbf{inc}\ s\,; \\
\quad\ \ \mathbf{let}\ y = f\ x\,;\ \boxed{\mathbf{let}\ ys = map\ f\ s\,;} \\
\quad\ \ \mathbf{let}\ r = (\mathbf{reuse}\ xs\ \mathbf{in}\ \mathbf{ctor}_2\ y\ ys)\,;\ \mathbf{ret}\ r)
\end{array}
$$



# Reusing small objects

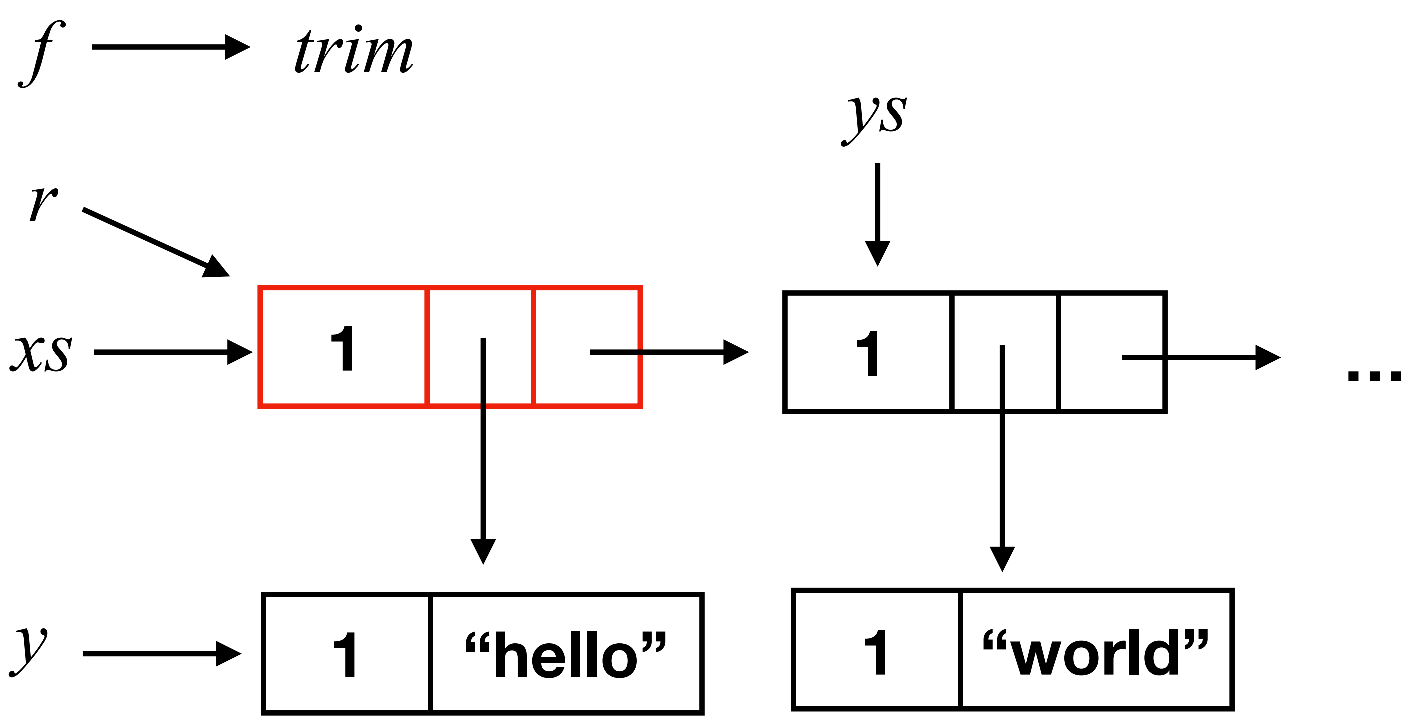
$map\ f\ xs = \textbf{case}\ xs\ \textbf{of}$

$\quad(\textbf{ret}\ xs)$

$\quad(\textbf{let}\ x = \textbf{proj}_1\ xs;\ \textbf{inc}\ x;\ \textbf{let}\ s = \textbf{proj}_2\ xs;\ \textbf{inc}\ s;$

$\quad\ \textbf{let}\ y = f\ x;\ \textbf{let}\ ys = map\ f\ s;$

$\quad\ \textbf{let}\ r = (\textbf{reuse}\ xs\ \textbf{in}\ \textbf{ctor}_2\ y\ ys);\ \textbf{ret}\ r)$




BAD. We only reused the one memory cell. We can do better!

# Reusing small objects

$$map\ f\ []\ =\ []$$

$$map\ f\ (x : xs) = (f\ x) : (map\ f\ xs)$$

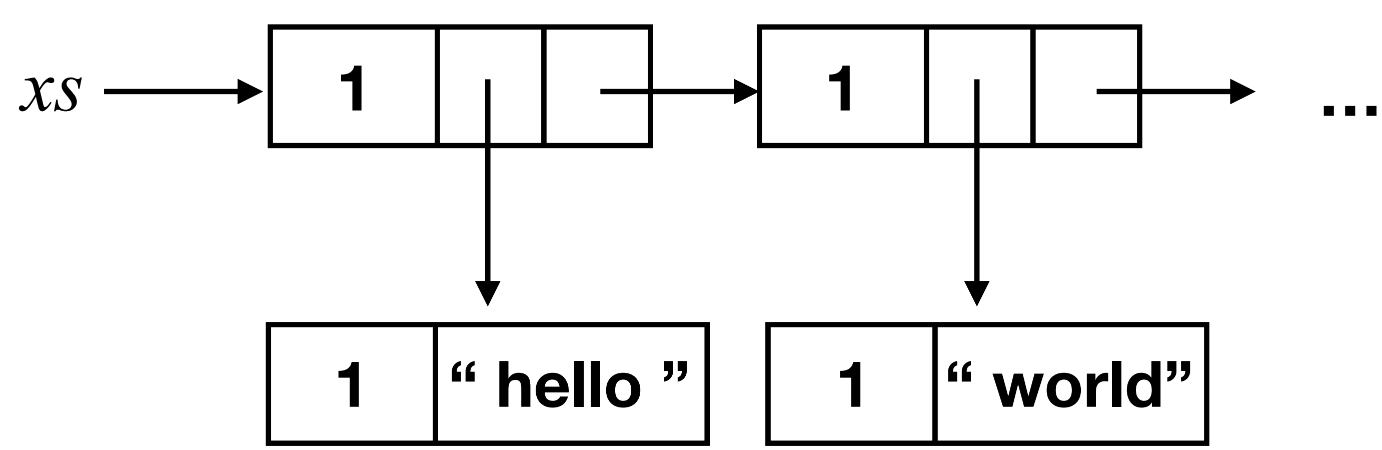
**Second attempt**

$$
\begin{aligned}
& map\ f\ xs = \mathbf{case}\ xs\ \mathbf{of} \\
& \quad (\mathbf{ret}\ xs) \\
& \quad (\mathbf{let}\ x = \mathbf{proj}_1\ xs\,;\ \mathbf{inc}\ x\,;\ \mathbf{let}\ s = \mathbf{proj}_2\ xs\,;\ \mathbf{inc}\ s\,; \\
& \quad\ \ \mathbf{let}\ w = \boxed{\mathbf{reset}\ xs\,;} \\
& \quad\ \ \mathbf{let}\ y = f\ x\,;\ \mathbf{let}\ ys = map\ f\ s\,; \\
& \quad\ \ \mathbf{let}\ r = (\mathbf{reuse}\ w\ \mathbf{in}\ \mathbf{ctor}_2\ y\ ys)\,;\ \mathbf{ret}\ r)
\end{aligned}
$$

# Reusing small objects

$$
\begin{array}{l}
map\ f\ xs = \mathbf{case}\ xs\ \mathbf{of} \\
\quad (\mathbf{ret}\ xs) \\
\quad (\mathbf{let}\ x = \mathbf{proj}_1\ xs;\ \mathbf{inc}\ x;\ \mathbf{let}\ s = \mathbf{proj}_2\ xs;\ \mathbf{inc}\ s; \\
\quad\ \ \mathbf{let}\ w = \mathbf{reset}\ xs; \\
\quad\ \ \mathbf{let}\ y = f\ x;\ \mathbf{let}\ ys = map\ f\ s; \\
\quad\ \ \mathbf{let}\ r = (\mathbf{reuse}\ w\ \mathbf{in}\ \mathbf{ctor}_2\ y\ ys);\ \mathbf{ret}\ r)
\end{array}
$$

$f \longrightarrow trim$



# Reusing small objects

$map\ f\ xs = \mathbf{case}\ xs\ \mathbf{of}$

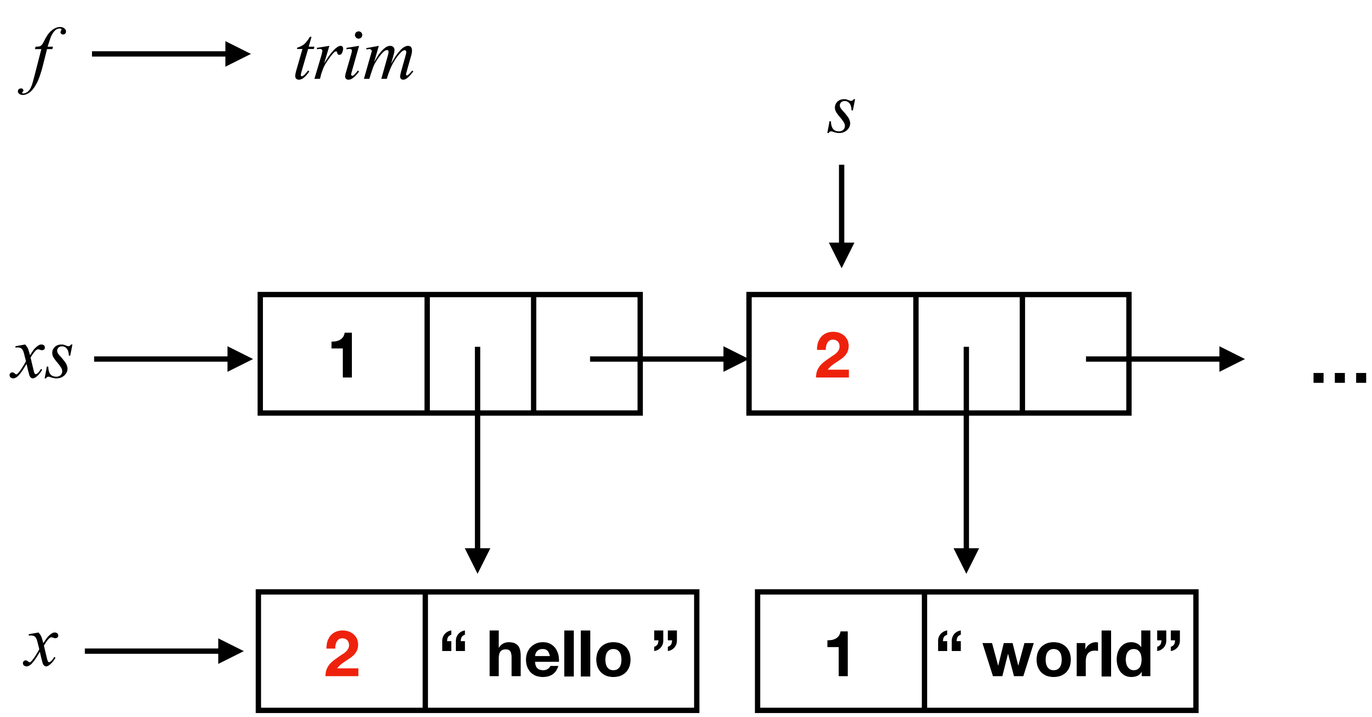
$\quad (\mathbf{ret}\ xs)$

$\quad (\mathbf{let}\ x = \mathbf{proj}_1\ xs\,;\ \mathbf{inc}\ x\,;\ \mathbf{let}\ s = \mathbf{proj}_2\ xs\,;\ \mathbf{inc}\ s\,;$

$\quad\ \mathbf{let}\ w = \mathbf{reset}\ xs\,;$

$\quad\ \mathbf{let}\ y = f\ x\,;\ \mathbf{let}\ ys = map\ f\ s\,;$

$\quad\ \mathbf{let}\ r = (\mathbf{reuse}\ w\ \mathbf{in}\ \mathbf{ctor}_2\ y\ ys)\,;\ \mathbf{ret}\ r)$



# Reusing small objects

$map\ f\ xs = \mathbf{case}\ xs\ \mathbf{of}$

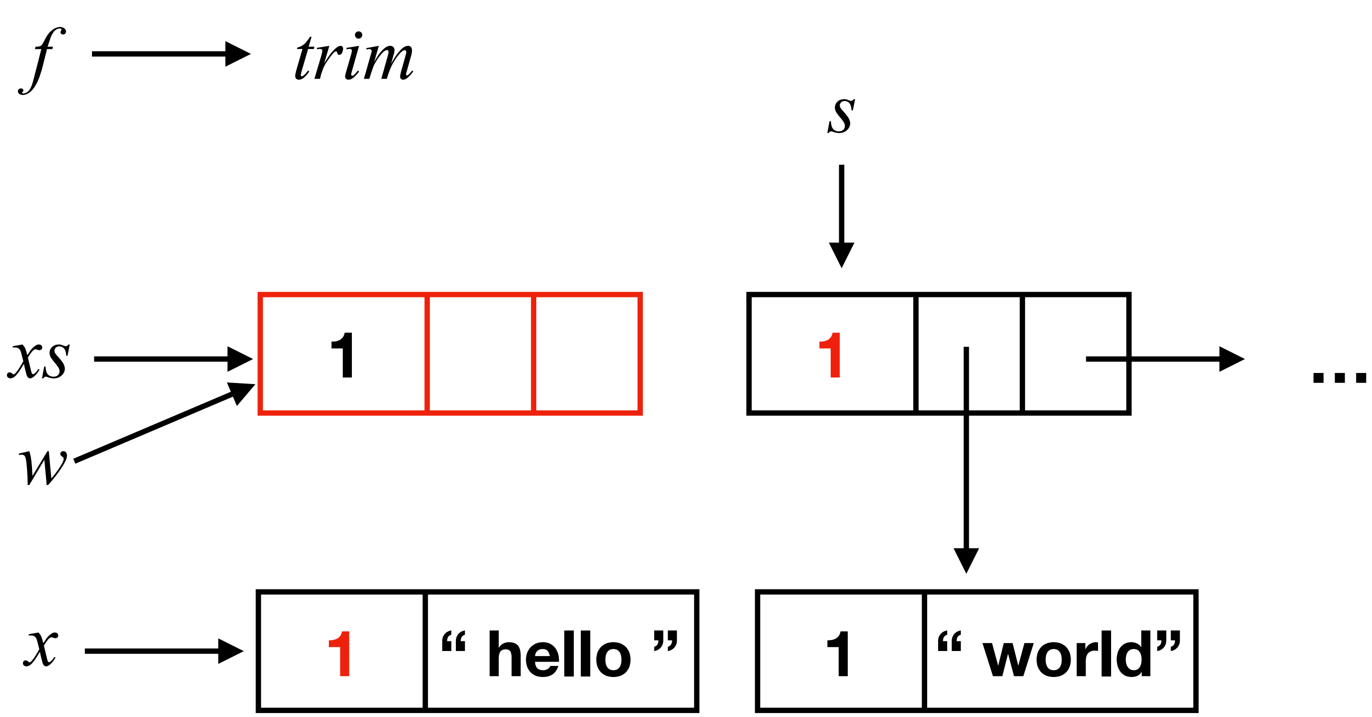
$\quad(\mathbf{ret}\ xs)$

$\quad(\mathbf{let}\ x = \mathbf{proj}_1\ xs\,;\ \mathbf{inc}\ x\,;\ \mathbf{let}\ s = \mathbf{proj}_2\ xs\,;\ \mathbf{inc}\ s\,;$

$\quad\ \mathbf{let}\ w = \mathbf{reset}\ xs\,;$

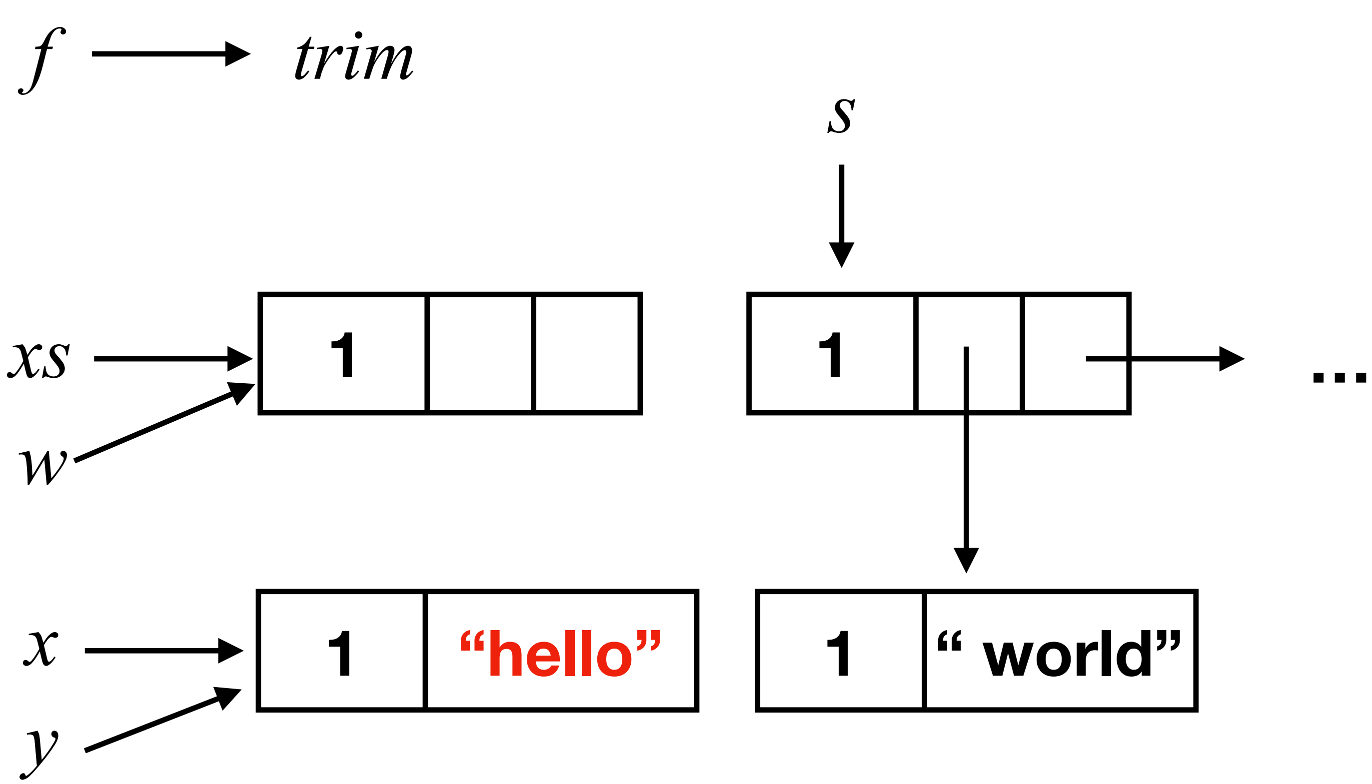
$\quad\ \mathbf{let}\ y = f\ x\,;\ \mathbf{let}\ ys = map\ f\ s\,;$

$\quad\ \mathbf{let}\ r = (\mathbf{reuse}\ w\ \mathbf{in}\ \mathbf{ctor}_2\ y\ ys)\,;\ \mathbf{ret}\ r)$



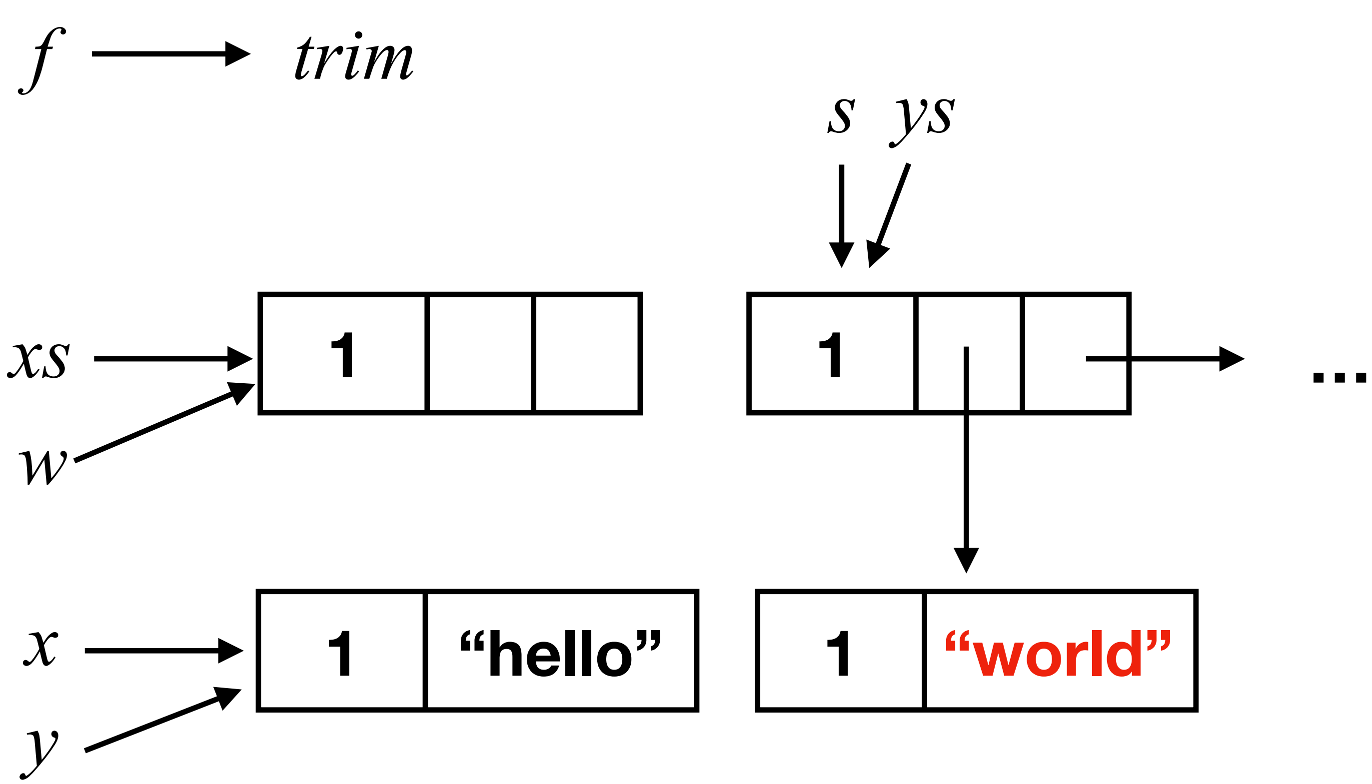
# Reusing small objects

$map\ f\ xs = \textbf{case}\ xs\ \textbf{of}$
$\quad (\textbf{ret}\ xs)$
$\quad (\textbf{let}\ x = \textbf{proj}_1\ xs\,;\ \textbf{inc}\ x\,;\ \textbf{let}\ s = \textbf{proj}_2\ xs\,;\ \textbf{inc}\ s\,;$
$\quad\ \ \textbf{let}\ w = \textbf{reset}\ xs\,;$
$\quad\ \ \boxed{\textbf{let}\ y = f\ x\,;}\ \textbf{let}\ ys = map\ f\ s\,;$
$\quad\ \ \textbf{let}\ r = (\textbf{reuse}\ w\ \textbf{in}\ \textbf{ctor}_2\ y\ ys)\,;\ \textbf{ret}\ r)$



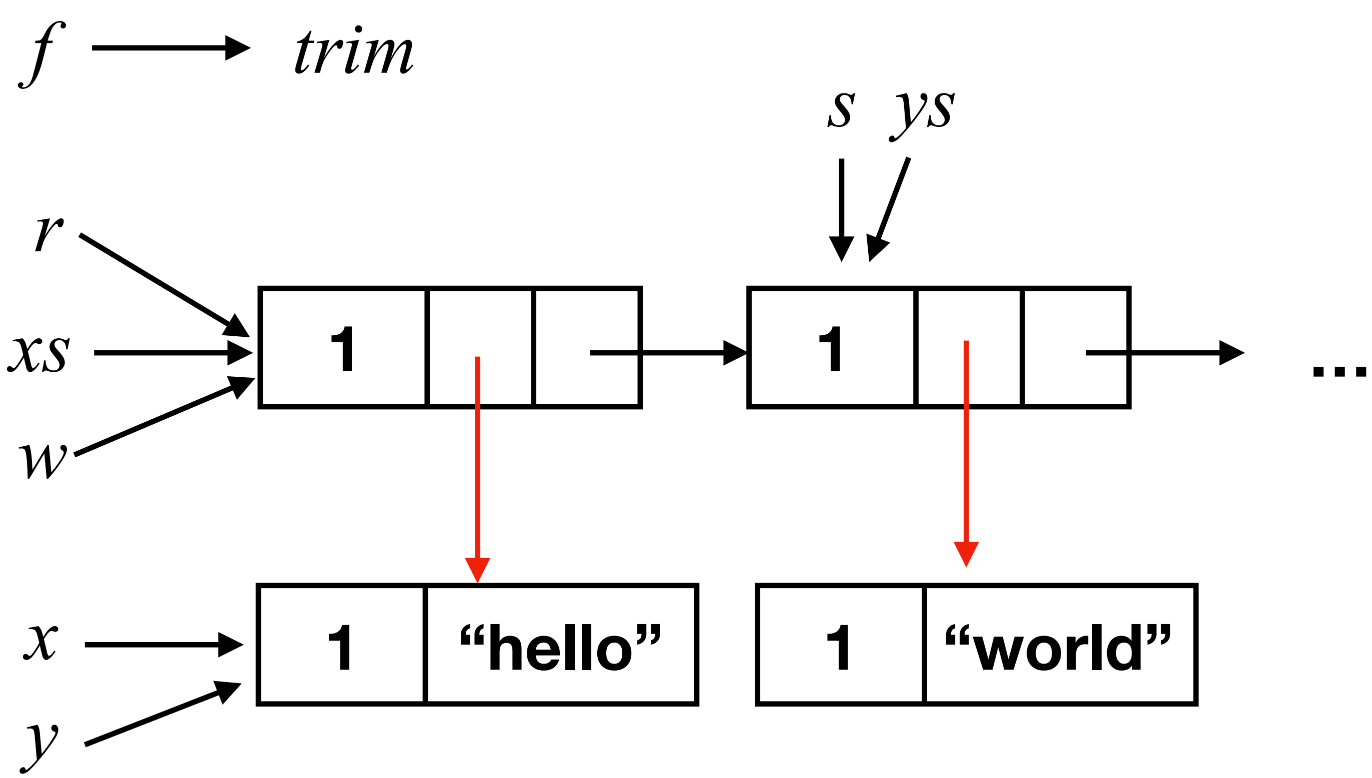
# Reusing small objects

$$\begin{array}{l} map\ f\ xs = \textbf{case}\ xs\ \textbf{of} \\ \quad (\textbf{ret}\ xs) \\ \quad (\textbf{let}\ x = \textbf{proj}_1\ xs\,;\ \textbf{inc}\ x\,;\ \textbf{let}\ s = \textbf{proj}_2\ xs\,;\ \textbf{inc}\ s\,; \\ \quad\ \ \textbf{let}\ w = \textbf{reset}\ xs\,; \\ \quad\ \ \textbf{let}\ y = f\ x\,;\ \boxed{\textbf{let}\ ys = map\ f\ s\,;} \\ \quad\ \ \textbf{let}\ r = (\textbf{reuse}\ w\ \textbf{in}\ \textbf{ctor}_2\ y\ ys)\,;\ \textbf{ret}\ r) \end{array}$$



# Reusing small objects

$$
\begin{aligned}
&\mathit{map}\ f\ \mathit{xs} = \mathbf{case}\ \mathit{xs}\ \mathbf{of}\\
&\quad (\mathbf{ret}\ \mathit{xs})\\
&\quad (\mathbf{let}\ x = \mathbf{proj}_1\ \mathit{xs};\ \mathbf{inc}\ x;\ \mathbf{let}\ s = \mathbf{proj}_2\ \mathit{xs};\ \mathbf{inc}\ s;\\
&\quad\ \ \mathbf{let}\ w = \mathbf{reset}\ \mathit{xs};\\
&\quad\ \ \mathbf{let}\ y = f\ x;\ \mathbf{let}\ \mathit{ys} = \mathit{map}\ f\ s;\\
&\quad\ \ \mathbf{let}\ \boxed{r = (\mathbf{reuse}\ w\ \mathbf{in}\ \mathbf{ctor}_2\ y\ \mathit{ys})};\ \mathbf{ret}\ r)
\end{aligned}
$$




**The whole list was destructively updated!**

# The compiler

- Lean => Lambda Pure
- Insert **reset/reuse** instructions
- Infer borrowed annotations
- Insert **inc/dec** instructions
- Additional optimizations

$$
\begin{aligned}
w, x, y, z &\in \mathit{Var} \\
c &\in \mathit{Const} \\
e &\in \mathit{Expr} &&::= c\ \overline{y} \mid \mathbf{pap}\ c\ \overline{y} \mid x\ y \mid \mathbf{ctor}_i\ \overline{y} \mid \mathbf{proj}_i\ x \\
F &\in \mathit{FnBody} &&::= \mathbf{ret}\ x \mid \mathbf{let}\ x = e;\ F \mid \mathbf{case}\ x\ \mathbf{of}\ \overline{F} \\
f &\in \mathit{Fn} &&::= \lambda\ \overline{y}.\ F \\
\delta &\in \mathit{Program} &&= \mathit{Const} \rightharpoonup \mathit{Fn}
\end{aligned}
$$

# Inserting reset/reuse

For each (**case** $x$ **of** $F_1 \ldots F_n$), for each branch $F_i$, if $F_i$ is of form ($P$; $S$; **let** $y$ := **ctor**$_i$ $zs$; $K$) where

1. $\#zs$ is equal to the number of fields of $x$ at branch $F_i$
2. $x$ is dead at ($S$; **let** $y$ := **ctor**$_i$ $zs$; $K$)

then replace with

$P$; **let** $w$ := **reset** $x$; $S$; **let** $y$ := **reuse** $w$ **in ctor**$_i$ $zs$; $K$

$$
\begin{aligned}
&swap\ [\,] = [\,] \\
&swap\ [x] = [x] \\
&swap\ (x : y : zs) = y : x : zs
\end{aligned}
$$

$$
\begin{aligned}
&swap\ xs = \textbf{case}\ xs\ \textbf{of} \\
&\quad (\textbf{ret}\ xs) \\
&\quad (\textbf{let}\ t_1 = \textbf{proj}_2\ xs\,;\ \textbf{case}\ t_1\ \textbf{of} \\
&\qquad (\textbf{ret}\ xs) \\
&\qquad (\textbf{let}\ h_1 = \textbf{proj}_1\ xs\,; \\
&\qquad\ \textbf{let}\ h_2 = \textbf{proj}_1\ t_1\,;\ \textbf{let}\ t_2 = \textbf{proj}_2\ t_1\,; \\
&\qquad\ \textbf{let}\ r_1 = \textbf{ctor}_2\ h_1\ t_2\,;\ \textbf{let}\ r_2 = \textbf{ctor}_2\ h_2\ r_1\,;\ \textbf{ret}\ r_2))
\end{aligned}
$$

# Inserting reset/reuse

For each (**case** $x$ **of** $F_1 \ldots F_n$), for each branch $F_i$, if $F_i$ is of form ($P$; $S$; **let** $y$ := **ctor**$_i$ $zs$; $K$) where

1. $\#zs$ is equal to the number of fields of $x$ at branch $F_i$
2. $x$ is dead at ($S$; **let** $y$ := **ctor**$_i$ $zs$; $K$)

then replace with

$P$; **let** $w$ := **reset** $x$; $S$; **let** $y$ := **reuse** $w$ **in** ctor$_i$ $zs$; $K$

```
swap xs = case xs of
  (ret xs)
  (let t1 = proj2 xs; case t1 of
    (ret xs)
    (let h1 = proj1 xs;
     let h2 = proj1 t1; let t2 = proj2 t1;
     let r1 = ctor2 h1 t2; let r2 = ctor2 h2 r1; ret r2))
```

```
swap xs = case xs of
  (ret xs)
  (let t1 = proj2 xs; case t1 of
    (ret xs)
    (let h1 = proj1 xs; let w1 = reset xs;
     let h2 = proj1 t1; let t2 = proj2 t1;
     let w2 = reset t1; let r1 = reuse w2 in ctor2 h1 t2;
     let r2 = reuse w1 in ctor2 h2 r1; ret r2))
```

# Inferring borrowed annotations

- Heuristic based on the fact that when we mark a parameter as borrowed
    - We reduce the number of RC operations needed, but we prevent reset/reuse and primitive operations from reusing memory cells.

- We also want to preserve tail calls.

- Our approach: collect variables that must be owned.
    - $x$ or one of its projections is used in a **reset.**
    - $x$ is passed to a function that takes an owned reference.
    - By marking $x$ as borrowed we destroy a tail call.

# Tail call preservation

$$f\,x = \textbf{case}\ x\ \textbf{of}$$
$$(\textbf{let}\ r = \textbf{proj}_1\ x;\ \textbf{ret}\ r)$$
$$(\textbf{let}\ y_1 = \textbf{ctor}_1\ ;\ \textbf{let}\ y_2 = \textbf{ctor}_1\ y_1\ ;\ \boxed{\textbf{let}\ r = f\,y_2\ ;\ \textbf{ret}\ r)}$$

**If we mark x as borrowed, we do not preserve tail calls**

$$f\,x = \textbf{case}\ x\ \textbf{of}$$
$$(\textbf{let}\ r = \textbf{proj}_1\ x;\ \textbf{inc}\ r;\ \textbf{ret}\ r)$$
$$(\textbf{let}\ y_1 = \textbf{ctor}_1\ ;\ \textbf{let}\ y_2\ \textbf{ctor}_1\ y_1\ ;$$
$$\textbf{let}\ r = f\,y_2\ ;\ \textbf{dec}\ y_2\ ;\ \textbf{ret}\ r)$$

# Multi-threading support

$$Task.mk : (Unit \to \alpha) \to Task\ \alpha$$
$$Task.bind : Task\ \alpha \to (\alpha \to Task\ \beta) \to Task\ \beta$$
$$Task.get : Task\ \alpha \to \alpha$$

- We store in the object header whether an object is multi-thread or not.
- New objects are not multi-threaded.
- We don't need memory fences for updating RC if an object is not multi-thread.
- The runtime has a *markMT(o)* primitive.

*(Task.mk f) => markMT(f)*

*(Task.bind x f) => markMT(x)* and *markMT(f)*

# Simple Optimizations

- Our compiler expands **reset** and **reuse** using lower level instructions: **isShared** $x$, **set** $x[i]$ $v$, …

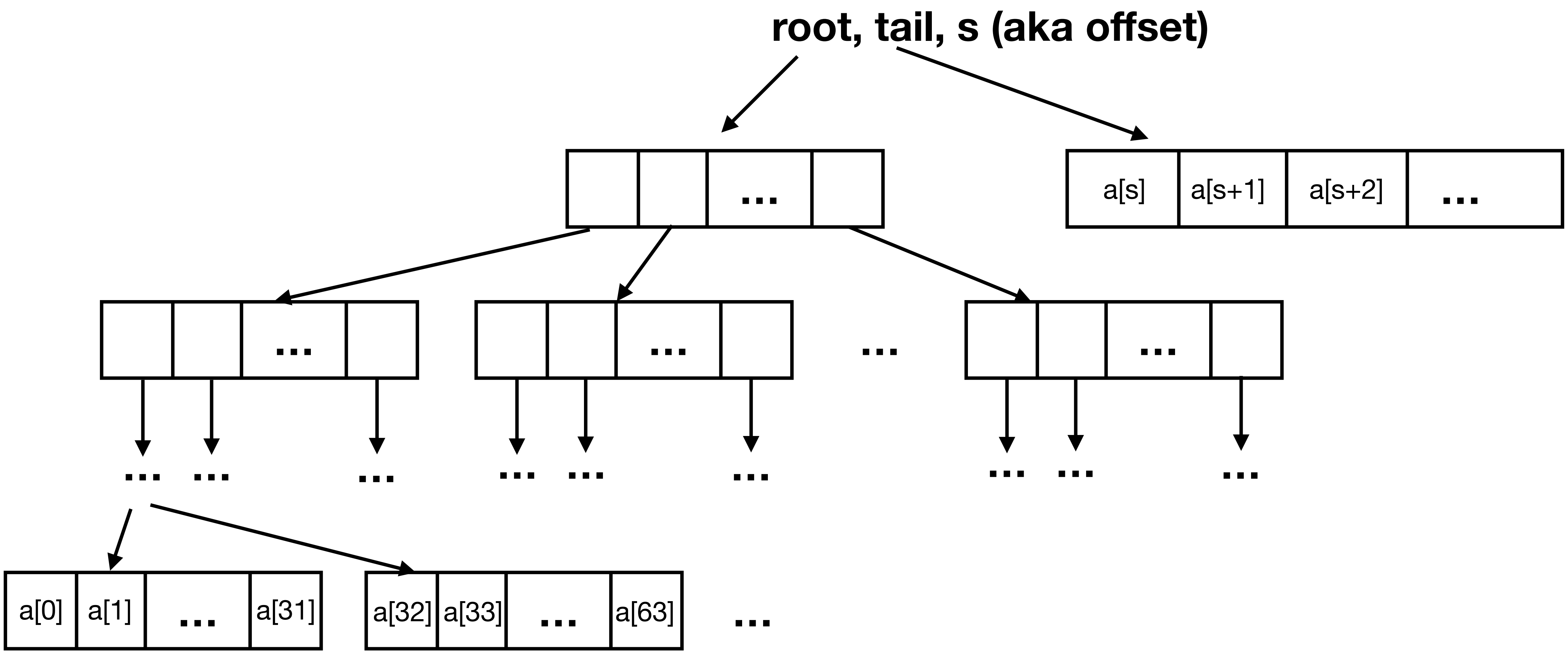
- The lower level instructions generate new optimization opportunities for many common IR sequences. Example: **reset** immediately followed by **reuse**.

- Minimizes the amount of copying and RC operations.

# Comparison with Linear/Uniqueness Types

- Values of types marked as linear/unique can be destructively updated.
- Compiler statically checks whether values are being used linearly or not.
- Pros: no runtime checks; compatible with tracing GCs.
- Cons: awkward to use; complicates a dependent type system even more.
- Big cons: all or nothing. A function *f* that takes non-shared values most of the time cannot perform destructive updates.

# Persistent Arrays




Reusing big and small objects.
Persistent arrays will often be shared.

# What about cycles?

- Inductive datatypes in Lean are acyclic.
- We can implement co-inductive datatypes without creating cycles.
- Only unsafe code in Lean can create cycles.
- **Cycles are overrated**.
- What about graphs? How do you represent them in Lean?
  - Use arrays like in Rust.
  - We have destructive updates in Lean.
  - Persistent arrays are also quite fast.

# Experimental evaluation

| Benchmark | Lean | del[%] | Cache Misses [1M/s] | GHC | GC | CM | OCaml | GC | CM |
|---|---|---|---|---|---|---|---|---|---|
| binarytrees | 1.0 | 40 | 37 | 3.09 | 72 | 120 | 1.20 | N/A | 186 |
| deriv | 1.0 | 24 | 32 | 1.89 | 51 | 31 | 1.42 | 76 | 59 |
| const_fold | 1.0 | 11 | 83 | 2.23 | 64 | 51 | 4.66 | 91 | 107 |
| qsort | 1.0 | 9 | 0 | 1.63 | 1 | 0 | 1.37 | 13 | 1 |
| rbmap | 1.0 | 2 | 6 | 2.41 | 39 | 23 | 1.00 | 31 | 27 |
| rbmap_10 | 1.48 | 15 | 34 | 16.37 | 88 | 47 | 1.93 | 60 | 59 |
| rbmap_1 | 5.1 | 27 | 55 | 16.41 | 88 | 48 | 9.83 | 88 | 89 |

# Conclusion

- It is feasible to implement functional languages using RC.
- We barely scratched the surface of the design space.
- We are implementing Lean4 in Lean.
- Compiler generates C code.
- Compiler source code and all experiments are available online. http://github.com/leanprover/lean4
- We are working on new optimizations.