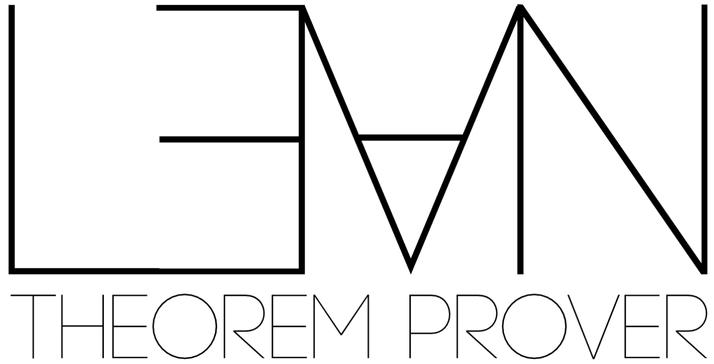


Programming Language

http://leanprover.github.io

Lean in Lean

Leonardo de Moura - MSR - USA

Workshop

Lean

LEAN
THEOREM PROVER
Programming Language

- Goals
  - Extensibility, Expressivity, Scalability, Proof stability
  - Functional Programming (efficiency)
- Platform for
  - Developing custom automation and domain specific languages (DSLs)
  - Software verification
  - Formalized Mathematics
- Dependent Type Theory
- de Bruijn's principle: small trusted kernel, external proof/type checkers

Lean Timeline

- Lean 1 (2013) Leo and Soonho Kong
  - Almost useless
  - Brave (crazy?) users in 2014: Jeremy Avigad, Cody Roux and Floris van Doorn
- Lean 2 (2015) Leo and Soonho Kong
  - First official release
  - Emacs interface
  - Floris van Doorn develops the HoTT library for Lean
  - First Math library (Jeremy Avigad, Rob Lewis, and many others)
- Lean 3 (2016) Leo, Daniel Selsam, Gabriel Ebner, Jared Roesch, Sebastian Ullrich
  - Lean is now a programming language (interpreter)
  - Metaprogramming and White box automation
  - VS Code interface
- Lean 4 (202x) Leo and Sebastian Ullrich
  - Lean In Lean
  - Compiler

Metaprogramming

- Extend Lean using Lean
- Proof/Program synthesis
- Access Lean internals using Lean
  - Type inference
  - Unifier
  - Simplifier
  - Decision procedures
  - Type class resolution
  - ...

White box automation

APIs (in Lean) for accessing data-structures and procedures found in SMT solvers and ATPs.

Dependent Type Theory

- Before we started Lean, we have studied different theorem provers: ACL2, Agda, Automath, Coq, HOL, HOL Light, Isabelle, Mizar, PVS.
- Dependent Type Theory is really beautiful.
- Some advantages:
  - Builtin computational interpretation.
  - Same data structure for representing proofs and terms.
  - Reduce code duplication:
    - Compiler for Haskell-like recursive equations, we can use it to write proofs.
  - Mathematical structures (e.g., Groups and Rings) are first-class citizens.
- Some references:
  - In praise of dependent types (Mike Shulman)
  - Type inference in mathematics (Jeremy Avigad)

Applications

Certigrad

Bug-free machine learning on stochastic computation graphs

Daniel Selsam (Stanford, now MSR)

Source code: https://github.com/dselsam/certigrad

ICML paper: https://arxiv.org/abs/1706.08605

Video: https://www.youtube.com/watch?v=-A1tVNTHUFw

Certigrad at Hacker news: https://news.ycombinator.com/item?id=14739491

Protocol Verification

Joe Hendrix, Joey Dodds, Ben Sherman, Ledah Casburn, Simon Hudon

Galois Inc

“We defined a hash-chained based distributed time stamping service down to the byte-level message wire format, and specified the system correctness as an LTL liveness property over an effectively infinite number of states, and then verified the property using Lean. We used some custom tactics for proving the correctness of the byte-level serialization/ deserialization routines, defined an abstraction approach for reducing reasoning about the behavior of the overall network transition system to the behavior of individual components, and then verified those components primarily using existing Lean tactics.”

https://github.com/GaloisInc/lean-protocol-support

SQL Query Equivalence Checker

Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries

Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, Dan Suciu

University of Washington

https://arxiv.org/pdf/1802.02229.pdf

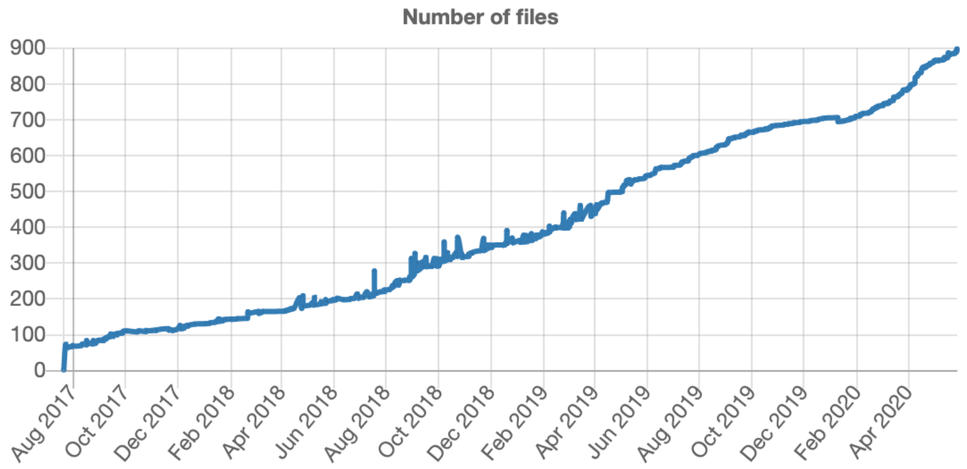
Mathlib

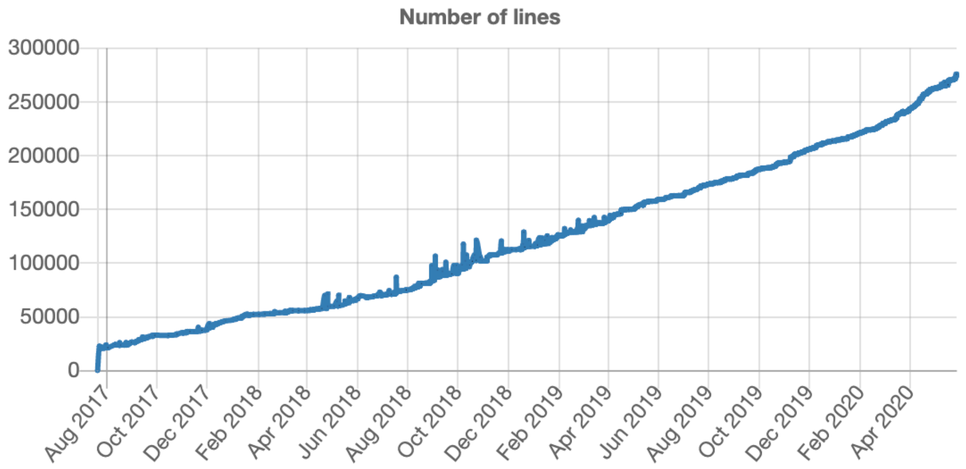
The Lean mathematical library, mathlib, is a community-driven effort to build a unified library of mathematics formalized in the Lean prover.

Jeremy Avigad, Reid Barton, Mario Carneiro, …

https://leanprover-community.github.io/meet.html

Paper: https://arxiv.org/abs/1910.09336





Lean perfectoid spaces

by Kevin Buzzard, Johan Commelin, and Patrick Massot

https://leanprover-community.github.io/lean-perfectoid-spaces/

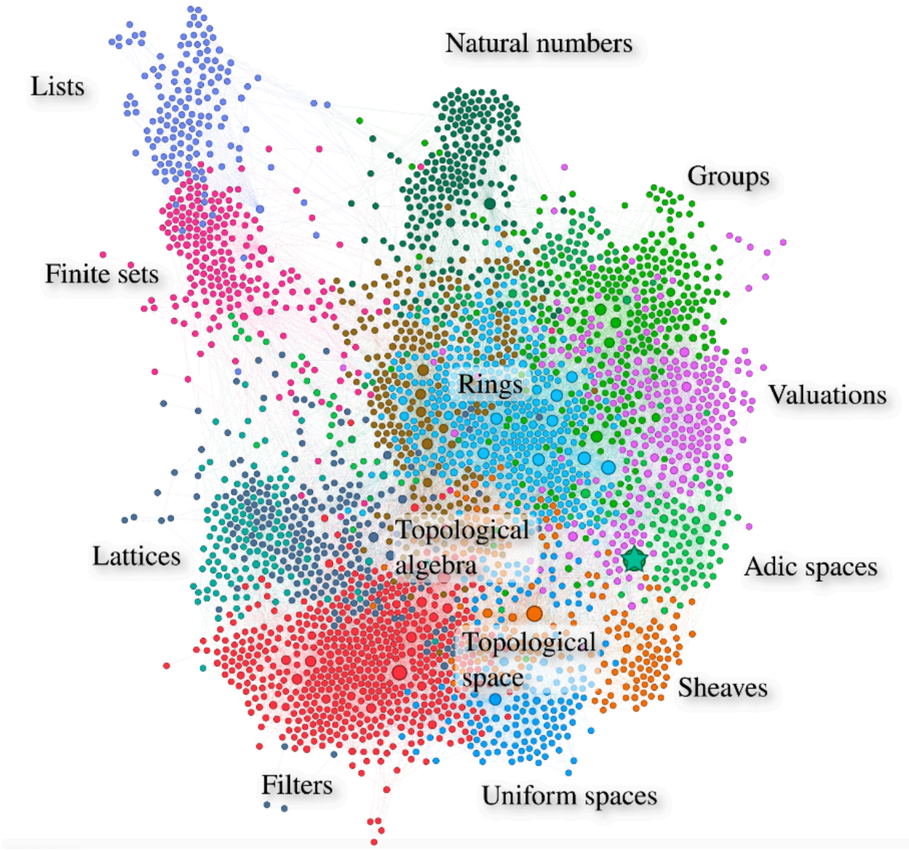
The Future of Mathematics?

```
/-- A perfectoid ring is a Huber ring that is complete, uniform,
that has a pseudo-uniformizer whose p-th power divides p in the power bounded subring,
and such that Frobenius is a surjection on the reduction modulo p.-/
structure perfectoid_ring (R : Type) [Huber_ring R] extends Tate_ring R : Prop :=
(complete  : is_complete_hausdorff R)
(uniform   : is_uniform R)
(ramified  : ∃ ϖ : pseudo_uniformizer R, ϖ^p | p in R°)
(Frobenius : surjective (Frob R°/p))

/-
CLVRS ("complete locally valued ringed space") is a category
whose objects are topological spaces with a sheaf of complete topological rings
and an equivalence class of valuation on each stalk, whose support is the unique
maximal ideal of the stalk; in Wedhorn's notes this category is called 𝒱.
A perfectoid space is an object of CLVRS which is locally isomorphic to Spa(A) with
A a perfectoid ring. Note however that CLVRS is a full subcategory of the category
`PreValuedRingedSpace` of topological spaces equipped with a presheaf of topological
rings and a valuation on each stalk, so the isomorphism can be checked in
PreValuedRingedSpace instead, which is what we do.
-/

/-- Condition for an object of CLVRS to be perfectoid: every point should have an open
neighbourhood isomorphic to Spa(A) for some perfectoid ring A.-/
def is_perfectoid (X : CLVRS) : Prop :=
∀ x : X, ∃ (U : opens X) (A : Huber_pair) [perfectoid_ring A],
  (x ∈ U) ∧ (Spa A ≅ U)

/-- The category of perfectoid spaces.-/
def PerfectoidSpace := {X : CLVRS // is_perfectoid X}
```



Tom Hales (University of Pittsburgh)

“To develop software and services for transforming mathematical results as they appear in journal article abstracts into formally structured data that machines can read, process, search, check, compute with, and learn from as logical statements.”

https://sloan.org/grant-detail/8439

https://hanoifabs.wordpress.com/2018/05/31/tentative-schedule/

https://github.com/formalabstracts/formalabstracts

Usable Computer-Checked Proofs and Computations for Number Theorists.

https://lean-forward.github.io/

"The ultimate aim is to develop a proof assistant that actually helps mathematicians, by making them more productive and more confident in their results."

VU Amsterdam

IMO Grand Challenge

The challenge: build an AI that can win a gold medal in the competition.

https://imo-grand-challenge.github.io/

Daniel Selsam (MSR)

Other applications

- IVy metatheory, Ken McMillan, MSR Redmond
- AliveInLean, Nuno Lopes, MSR Cambridge
- Education
  - Introduction to Logic (CMU)
  - Type theory (CMU)
  - Software verification and Logic (VU Amsterdam)
  - Programming Languages (UW)
  - Introduction to Proof (Imperial College)
- 6 papers at ITP 2019

Extensibility

Lean 3 users extend Lean using Lean

Examples:

- Ring Solver
- Coinductive predicates
- Transfer tactic
- Superposition prover
- Linters
- Fourier-Motzkin & Omega
- Many more

Lean 3.x limitations

- Lean programs are compiled into byte code and then interpreted (slow).
- Lean expressions are foreign objects reflected in Lean.
- Very limited ways to extend the parser.

```
infix >=        := ge
infix ≥         := ge
infix >         := gt
```

```
notation `∃` binders `, ` r:(scoped P, Exists P) := r
```

```
notation `[` l:(foldr `, ` (h t, list.cons h t) list.nil `]`) := l
```

- Users cannot implement their own elaboration strategies.
- Trace messages are just strings.

Lean 4

- Implement Lean in Lean
  - Parser, elaborator, compiler, tactics and formatter.
  - Hygienic macro system.
  - Structured trace messages.
  - Only the runtime and basic primitives are implemented in C/C++.
- Foreign function interface.
- Runtime has support for boxed and unboxed data.
- Runtime uses reference counting for GC and performs destructive updates when RC = 1
- (Safe) support for low-level tricks such as pointer equality.
- A better value proposition: use proofs for obtaining more efficient code.

Lean 4 is being implemented in Lean

```
inductive Expr
| bvar    : Nat → Data → Expr                          -- bound variables
| fvar    : FVarId → Data → Expr                       -- free variables
| mvar    : MVarId → Data → Expr                       -- meta variables
| sort    : Level → Data → Expr                        -- Sort
| const   : Name → List Level → Data → Expr            -- constants
| app     : Expr → Expr → Data → Expr                  -- application
| lam     : Name → Expr → Expr → Data → Expr           -- lambda abstraction
| forallE : Name → Expr → Expr → Data → Expr           -- (dependent) arrow
| letE    : Name → Expr → Expr → Expr → Data → Expr    -- let expressions
| lit     : Literal → Data → Expr                      -- literals
| mdata   : MData → Expr → Data → Expr                 -- metadata
| proj    : Name → Nat → Expr → Data → Expr           -- projection
```

Lean 4 is being implemented in Lean

```
def mkBinding (isLambda : Bool) (lctx : LocalContext) (xs : Array Expr) (b : Expr) : Expr :=
let b := b.abstract xs;
xs.size.foldRev (fun i b =>
  let x := xs.get! i;
  match lctx.findFVar? x with
  | some (LocalDecl.cdecl _ _ n ty bi)  =>
    let ty := ty.abstractRange i xs;
    if isLambda then
      Lean.mkLambda n bi ty b
    else
      Lean.mkForall n bi ty b
  | some (LocalDecl.ldecl _ _ n ty val) =>
    if b.hasLooseBVar 0 then
      let ty  := ty.abstractRange i xs;
      let val := val.abstractRange i xs;
      mkLet n ty val b
    else
      b.lowerLooseBVars 1 1
  | none => panic! "unknown free variable") b
```

Beyond CIC

- In CIC, all functions are total, but to implement Lean in Lean, we want
  - General recursion.
  - Foreign functions.
  - Unsafe features (e.g., pointer equality).

The unsafe keyword

- Unsafe functions may not terminate.
- Unsafe functions may use (unsafe) type casting.
- Regular (non unsafe) functions cannot call unsafe functions.
- Theorems are regular (non unsafe) functions.

A Compromise

- Make sure we cannot prove False in Lean.
  - Theorems proved in Lean 4 may still be checked by reference checkers.
  - Unsafe functions are ignored by reference checkers.
- Allow developers to provide an unsafe version for any (opaque) function whose type is inhabited.
- Examples:
  - Primitives implemented in C

```
@[extern "lean_uint64_mix_hash"]
constant mixHash64 (u₁ u₂ : UInt64) : UInt64 := 0
```

  - Sealing unsafe features

```
unsafe def setStateUnsafe {σ : Type} (ext : EnvExtension σ) (env : Environment) (s : σ) : Environment :=
{ env with extensions := env.extensions.set! ext.idx (unsafeCast s) }

@[implementedBy setStateUnsafe]
constant setState {σ : Type} (ext : EnvExtension σ) (env : Environment) (s : σ) : Environment := env
```

The partial keyword

- General recursion is a major convenience.
  - Some functions in our implementation may not terminate or cannot be shown to terminate in Lean, and we want to avoid an artificial “fuel" argument.
  - In many cases, the function terminates, but we don’t want to “waste" time proving it.

```
partial def whnfImpl : Expr → MetaM Expr
```

- A partial definition is just syntax sugar for the unsafe + implementedBy idiom.
- Future work: allow users to provide termination later, and use meta programming to generate a safe and non-opaque version of a partial function.

Proofs for performance and profit

- A better value proposition: use proofs for obtaining more efficient code.
- Example: skip runtime array bounds checks

```
def get (a : Array α) (i : Nat) (h : i < a.size) : α
```

- Example: pointer equality

```
def withPtrEq (x y : α) (k : Unit → Bool)
  (h : x = y → k () = true) : Bool := k ()
```

The definition is called a reference implementation

The compiler generates:

```
def withPtrEq (x y : α) (k : Unit → Bool)
  (h : x = y → k () = true) : Bool :=
if ptrAddr x = ptrAddr y
  then true
  else k ()
```

Proofs for performance and profit

- Example: theorems as compiler rewriting rules.
  - map f (map g xs) = map (f . g) xs
  - (h : assoc f) -> foldl f a xs = foldr f a xs
    xs = #[x1, x2, x3]
    f (f (f a x1) x2) x3 = f a (f x1 (f x2 x3)))

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

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

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

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

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

$$fst\ x\ y = \texttt{dec}\ y\,;\ \texttt{ret}\ x$$

Borrowed references

- If *xs* is an owned reference

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

- If *xs* is a borrowed reference

$$\mathit{isNil}\ \mathit{xs} = \mathbf{case}\ \mathit{xs}\ \mathbf{of}\ (\mathit{Nil} \to \mathtt{ret}\ \mathit{true})\ (\mathit{Cons} \to \mathtt{ret}\ \mathit{false})$$

Borrowed references

$$
\begin{aligned}
&\mathit{hasNone}\ [] = \mathit{false}\\
&\mathit{hasNone}\ (\mathit{None} : \mathit{xs}) = \mathit{true}\\
&\mathit{hasNone}\ (\mathit{Some}\ x : \mathit{xs}) = \mathit{hasNone}\ \mathit{xs}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{hasNone}\ \mathit{xs} = \mathbf{case}\ \mathit{xs}\ \mathbf{of}\\
&\quad (\mathit{Nil} \to \mathbf{ret}\ \mathit{false})\\
&\quad (\mathit{Cons} \to \mathbf{let}\ h = \mathbf{proj}_{\mathit{head}}\ \mathit{xs}\,;\ \mathbf{case}\ h\ \mathbf{of}\\
&\qquad (\mathit{None} \to \mathbf{ret}\ \mathit{true})\\
&\qquad (\mathit{Some} \to \mathbf{let}\ t = \mathbf{proj}_{\mathit{tail}}\ \mathit{xs}\,;\ \mathbf{let}\ r = \mathit{hasNone}\ t\,;\ \mathbf{ret}\ r))
\end{aligned}
$$

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

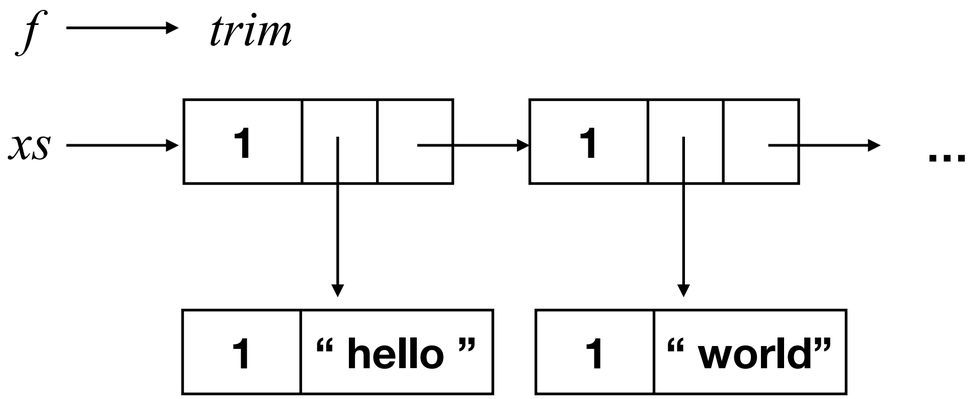
$$map\ f\ [] = []$$
$$map\ f\ (x : xs) = (f\ x) : (map\ f\ xs)$$

First attempt

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

Reusing small objects

```
map f xs = case xs of
  (ret xs)
  (let x = proj1 xs; inc x; let s = proj2 xs; inc s;
   let y = f x; let ys = map f s;
   let r = (reuse xs in ctor2 y ys); ret r)
```



Reusing small objects

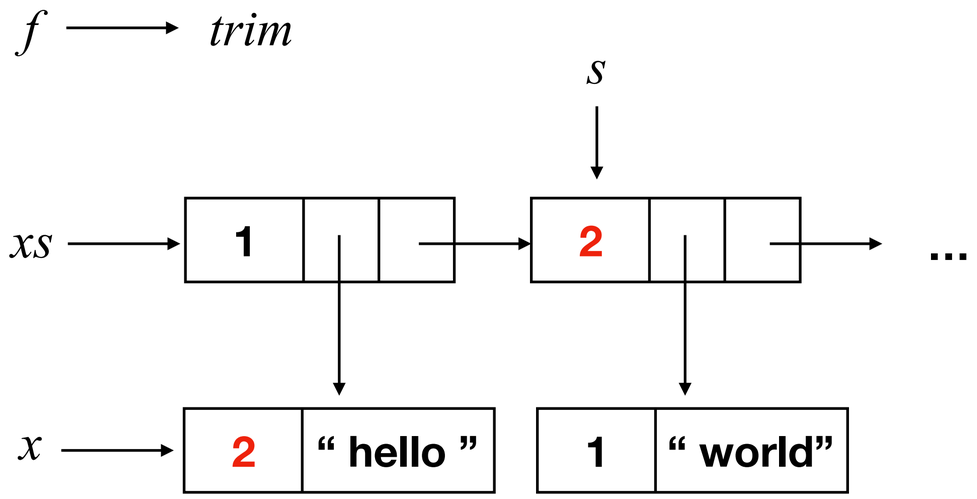
$map\ f\ xs = \mathbf{case}\ xs\ \mathbf{of}$

$(\mathbf{ret}\ xs)$

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

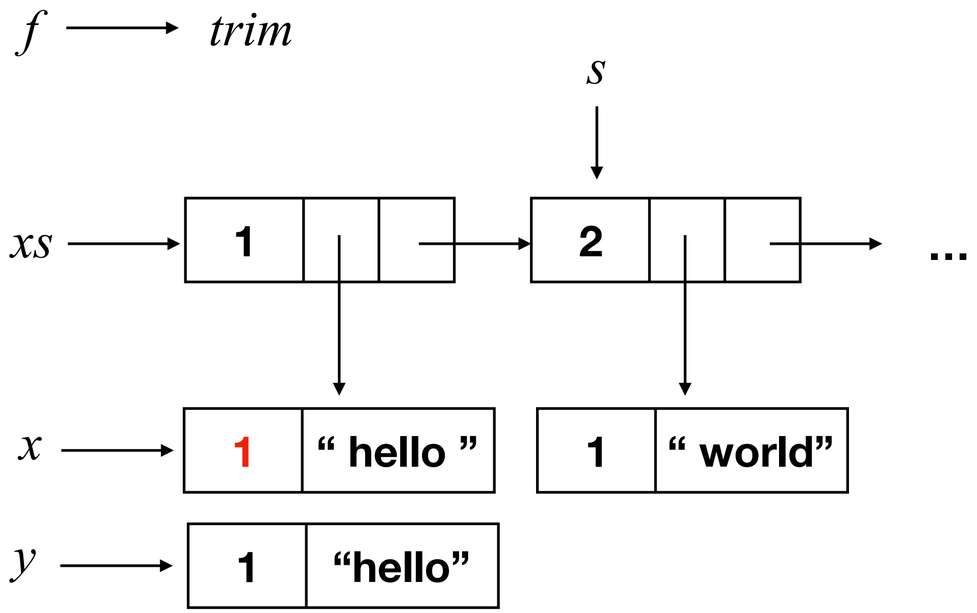
$\mathbf{let}\ y = f\ x\,;\ \mathbf{let}\ ys = map\ f\ s\,;$

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



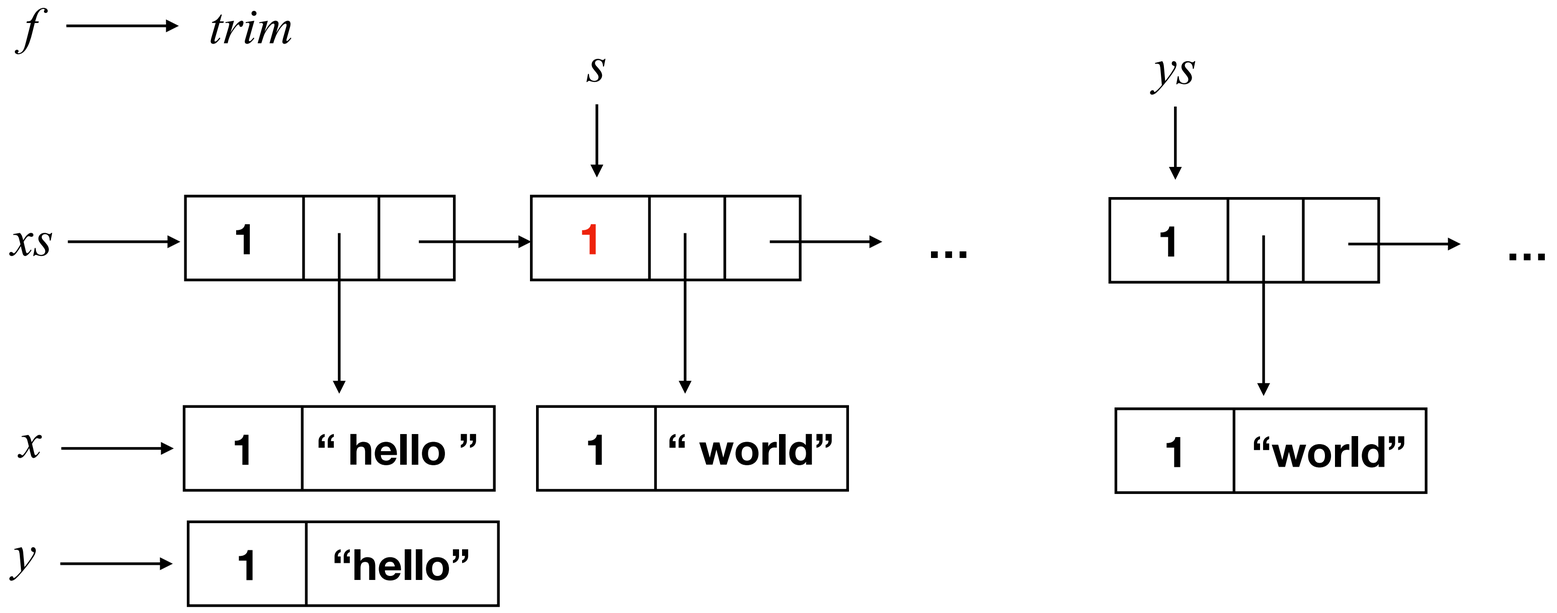
Reusing small objects

```
map f xs = case xs of
  (ret xs)
  (let x = proj1 xs; inc x; let s = proj2 xs; inc s;
   let y = f x; let ys = map f s;
   let r = (reuse xs in ctor2 y ys); ret r)
```



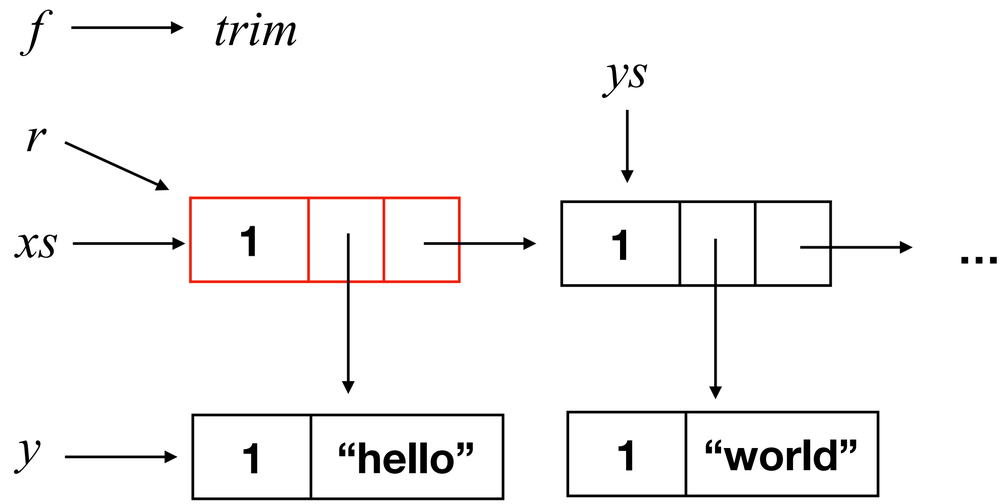
Reusing small objects

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



Reusing small objects

```
map f xs = case xs of
  (ret xs)
  (let x = proj1 xs; inc x; let s = proj2 xs; inc s;
   let y = f x; let ys = map f s;
   let r = (reuse xs in ctor2 y ys); ret r)
```



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

Reusing small objects

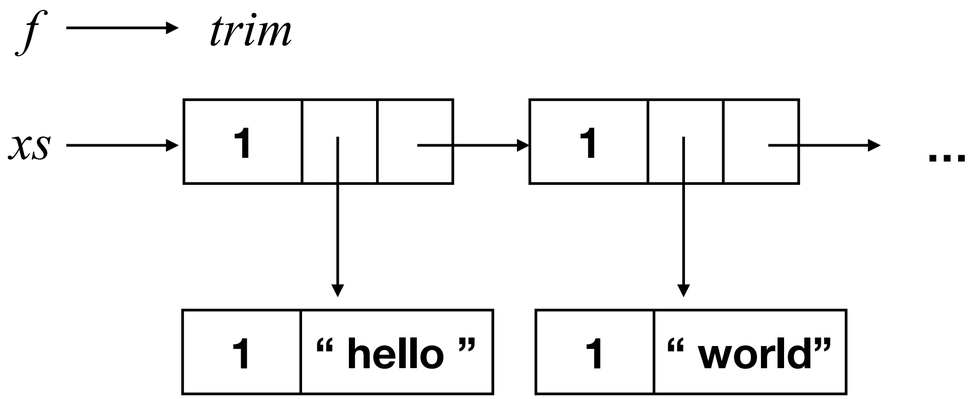
$$map\ f\ [] = []$$
$$map\ f\ (x : xs) = (f\ x) : (map\ f\ xs)$$

Second attempt

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

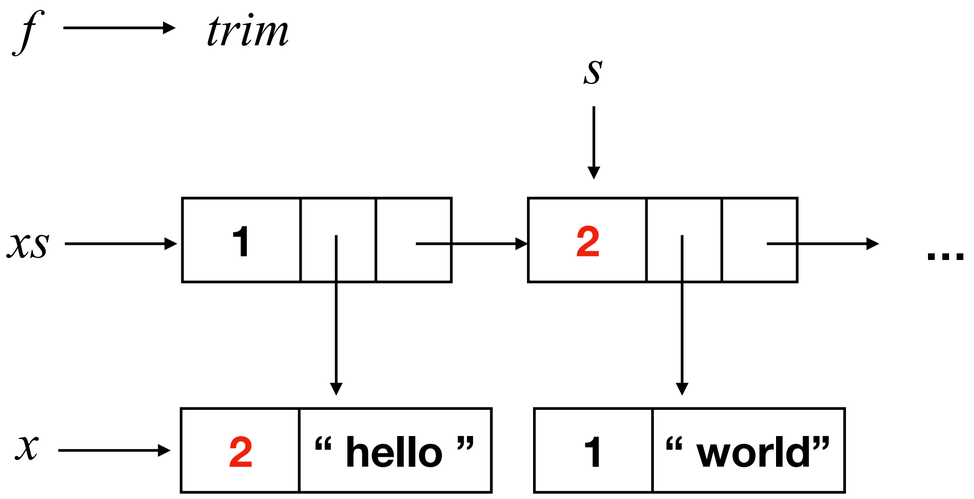
Reusing small objects

```
map f xs = case xs of
  (ret xs)
  (let x = proj1 xs; inc x; let s = proj2 xs; inc s;
   let w = reset xs;
   let y = f x; let ys = map f s;
   let r = (reuse w in ctor2 y ys); ret r)
```



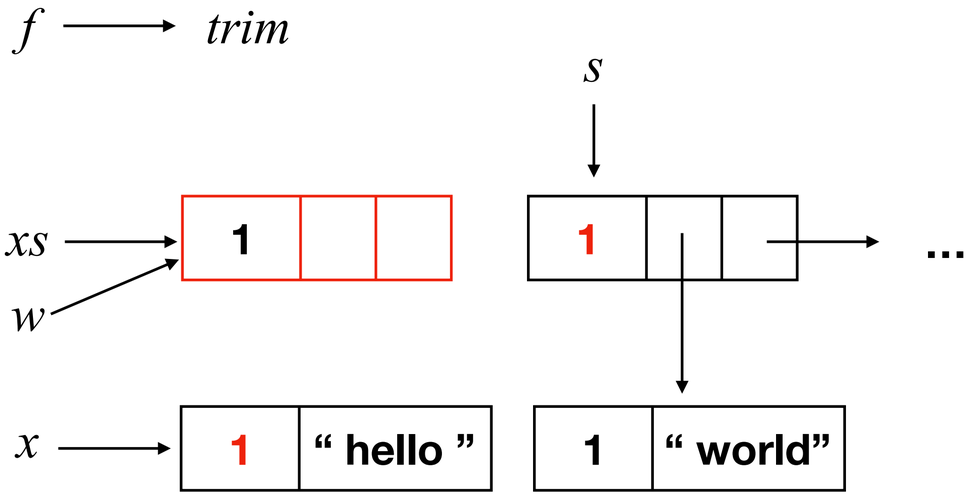
Reusing small objects

```
map f xs = case xs of
  (ret xs)
  (let x = proj1 xs; inc x; let s = proj2 xs; inc s;
   let w = reset xs;
   let y = f x; let ys = map f s;
   let r = (reuse w in ctor2 y ys); ret r)
```



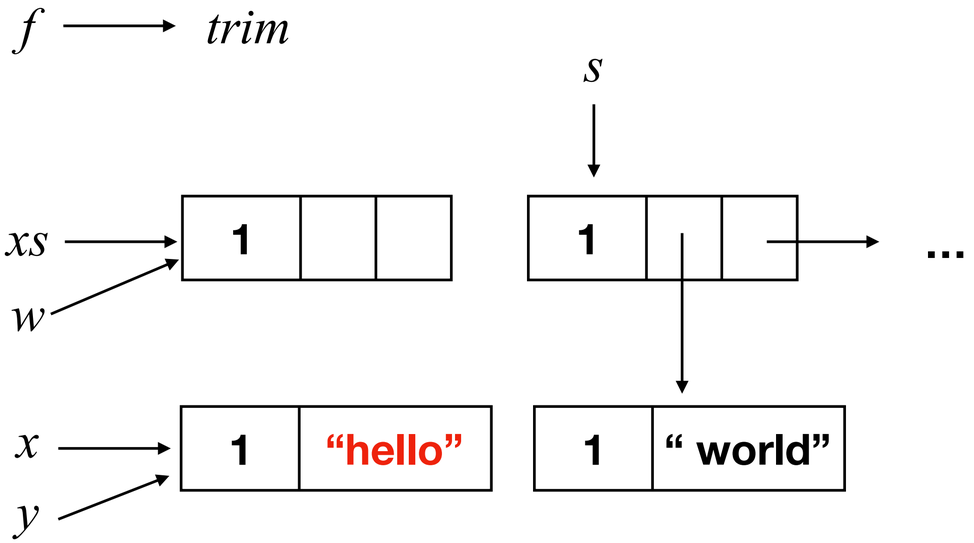
Reusing small objects

```
map f xs = case xs of
  (ret xs)
  (let x = proj1 xs; inc x; let s = proj2 xs; inc s;
   let w = reset xs;
   let y = f x; let ys = map f s;
   let r = (reuse w in ctor2 y ys); ret r)
```



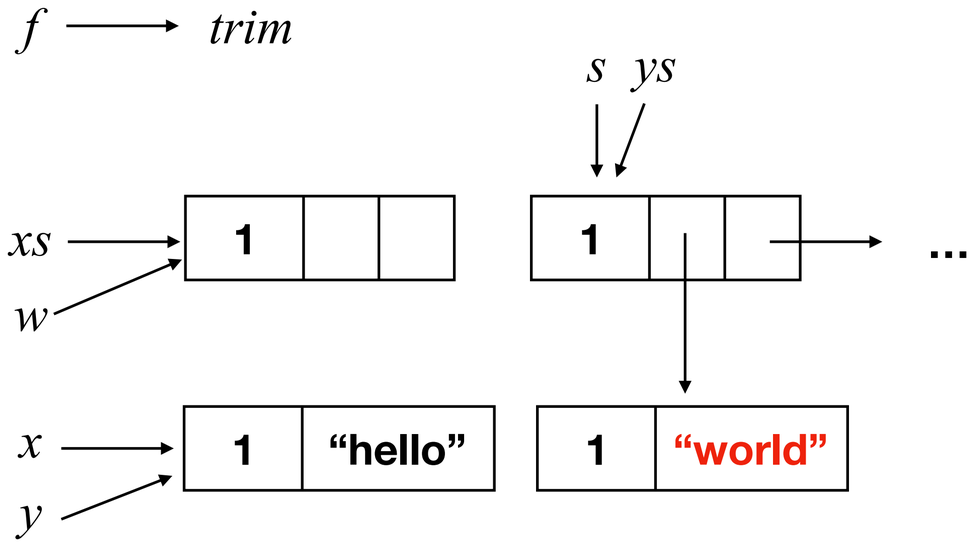
Reusing small objects

```
map f xs = case xs of
  (ret xs)
  (let x = proj1 xs; inc x; let s = proj2 xs; inc s;
   let w = reset xs;
   let y = f x; let ys = map f s;
   let r = (reuse w in ctor2 y ys); ret r)
```



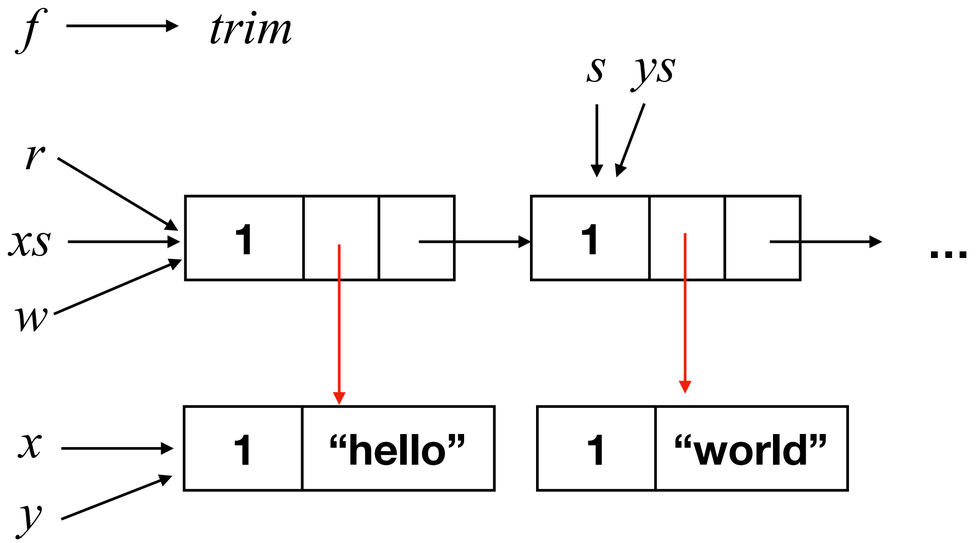
Reusing small objects

```
map f xs = case xs of
  (ret xs)
  (let x = proj1 xs; inc x; let s = proj2 xs; inc s;
   let w = reset xs;
   let y = f x; let ys = map f s;
   let r = (reuse w in ctor2 y ys); ret r)
```



Reusing small objects

```
map f xs = case xs of
  (ret xs)
  (let x = proj1 xs; inc x; let s = proj2 xs; inc s;
   let w = reset xs;
   let y = f x; let ys = map f s;
   let r = (reuse w in ctor2 y ys); ret r)
```



The whole list was destructively updated!

The compiler

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

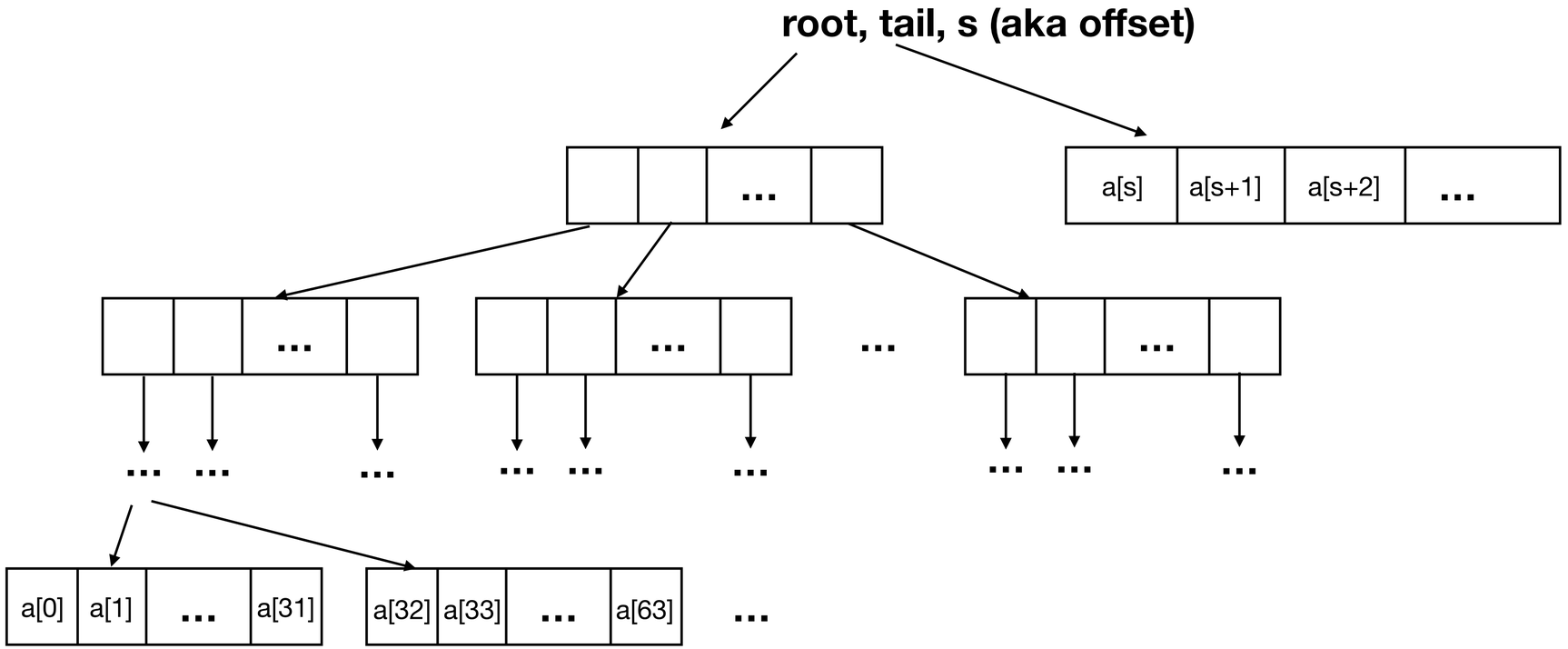
$$
\begin{aligned}
w, x, y, z &\in \mathit{Var} \\
c &\in \mathit{Const} \\
e &\in \mathit{Expr} && ::= c\ \overline{y} \mid \mathbf{pap}\ c\ \overline{y} \mid x\ y \mid \mathbf{ctor}_i\ \overline{y} \mid \mathbf{proj}_i\ x \\
F &\in \mathit{FnBody} && ::= \mathbf{ret}\ x \mid \mathbf{let}\ x = e;\ F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

Paper: "Counting Immutable Beans: Reference Counting Optimized for Purely Functional Programming", IFL 2019

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.

New idioms

```
structure ParserState :=
(stxStack  : Array Syntax)
(pos       : String.Pos)
(cache     : ParserCache)
(errorMsg : Option Error)

def pushSyntax (s : ParserState) (n : Syntax) : ParserState :=
{ stxStack := s.stxStack.push n, .. s }

def mkNode (s : ParserState) (k : SyntaxNodeKind) (iniStackSz : Nat) : ParserState :=
match s with
| ⟨stack, pos, cache, err⟩ =>
  let newNode := Syntax.node k (stack.extract iniStackSz stack.size);
  let stack   := stack.shrink iniStackSz;
  let stack   := stack.push newNode;
  ⟨stack, pos, cache, err⟩
```

Object layout

- In Haskell and OCaml, object header is 1 word only.
- We need space for the RC, can we be as compact? YES!
- In 64-bit machine, 1 word = 8 bytes = 64 bits
  - 8 bits for tag
  - 8 bits for number of fields
  - 3 bits for memory kind (single-threaded, multi-threaded, persistent, stack, …)
  - 45 bits for RC. Modern hardware can address only 2^{48}
  - **8 + 8 + 3 + 45 = 64**

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.

Conclusion

- We are implementing Lean4 in Lean.
- Users will be able and customize all modules of the system.
- Sealing unsafe features. Logical consistency is preserved.
- Compiler generates C code. Allows users to mix compiled and interpreted code.
- It is feasible to implement functional languages using RC.
- We barely scratched the surface of the design space.
- Source code available online. http://github.com/leanprover/lean4