LEAN
THEOREM PROVER
Programming Language
http://leanprover.github.io

Lean 4

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

Microsoft® Research

PLDI
LONDON
2020

KIT
Karlsruhe Institute of Technology

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

THEOREM PROVER
Programming Language

Resources

- Website: http://leanprover.github.io
- Online tutorial: https://leanprover.github.io/theorem_proving_in_lean/
- Zulip channel: https://leanprover.zulipchat.com/
- Community website: https://leanprover-community.github.io/
  - Maintainers of the official release (Lean 3)
- Mathlib: https://leanprover-community.github.io/mathlib-overview.html
- Lean 4 repository: https://github.com/leanprover/lean4

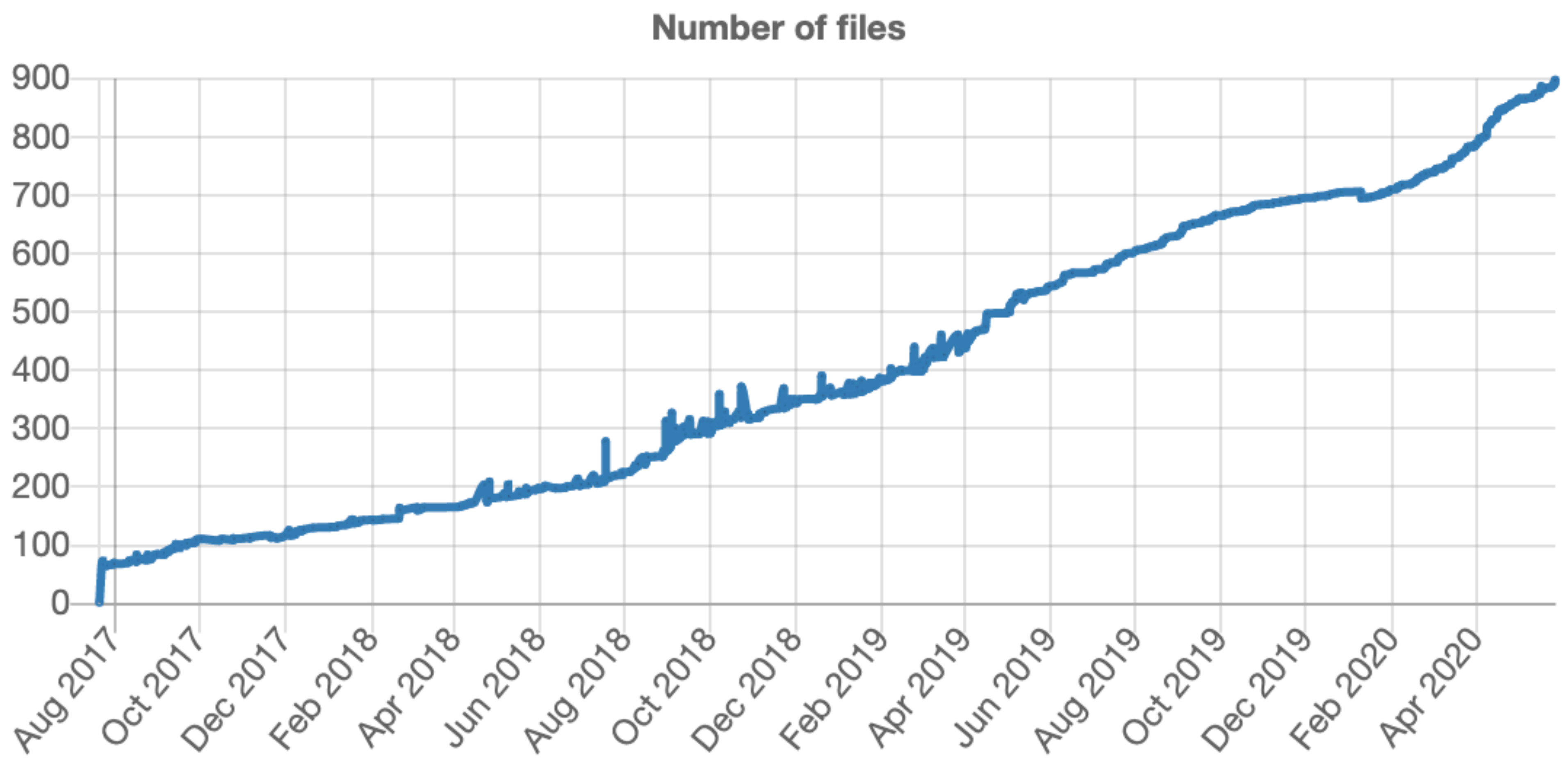
Mathlib

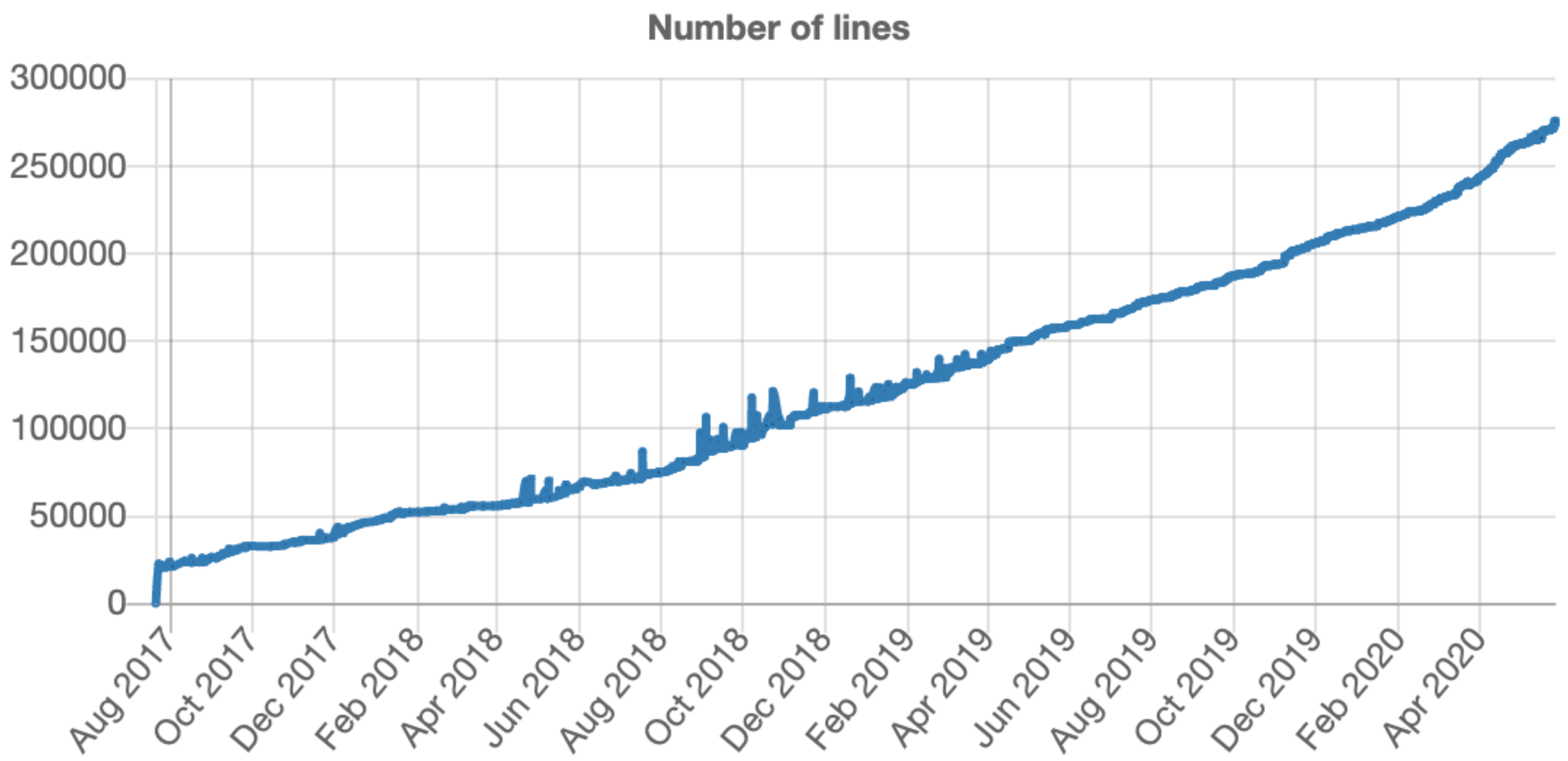
The Lean mathematical library, mathlib, is a community-driven effort to build a unified library of mathematics formalized in the Lean prover.

Jeremy Avigad, Reid Barton, Mario Carneiro, …

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

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





Extensibility

Lean 3 users extend Lean using Lean

Examples:

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

- Access Lean internals using Lean
  - Type inference, Unifier, Simplifier, Decision procedures,
  - Type class resolution, …

Applications

FAB STRACTS	Tom Hales, University of Pittsburgh
FLYPITCH formally proving the independence of the continuum hypothesis	Jesse Michael Han, Floris Van Doorn
Lean perfectoid spaces by Kevin Buzzard, Johan Commelin, and Patrick Massot	
Lean Forward	VU Amsterdam
IMO Grand Challenge	Daniel Selsam, MSR

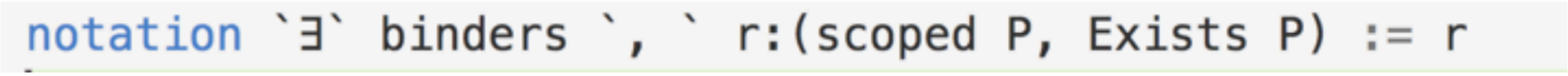
Other applications

- Certigrad, Daniel Selsam, Stanford
- IVy metatheory, Ken McMillan, MSR Redmond
- AliveInLean, Nuno Lopes, MSR Cambridge
- Protocol Verification, Galois Inc
- SQL Query Verification, Univ. Washington
- Education
  - Introduction to Logic (CMU), Type theory (CMU), Introduction to Proof (Imperial College),
  - Software verification and Logic (VU Amsterdam)
  - Programming Languages (UW)
- 6 papers at ITP 2019

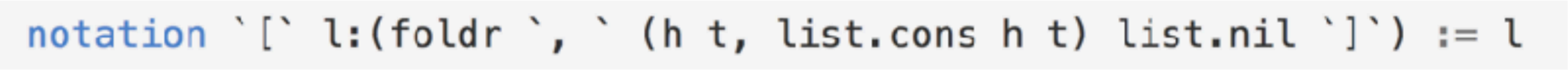
Lean 3.x limitations

- 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.
- Compiler generates C code. We can mix byte code and compiled code.
- (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

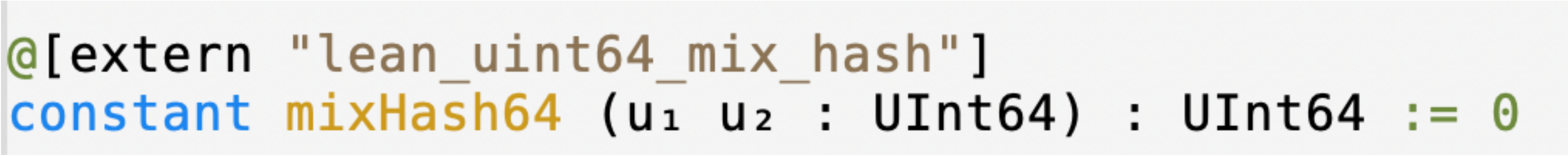
- In CIC, all functions are total, but to implement Lean in Lean, we want
  - General recursion.
  - Foreign functions.
  - Unsafe features (e.g., pointer equality).

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.

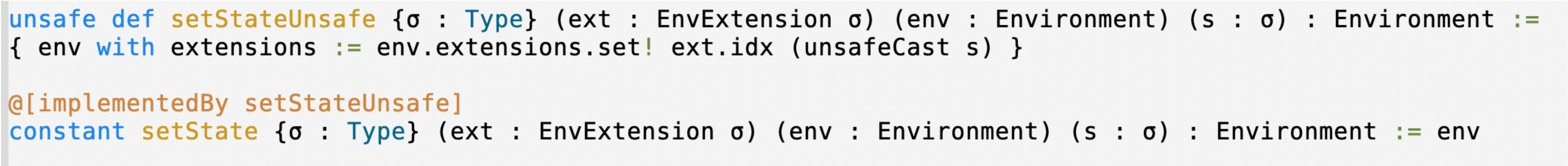
A Compromise

- 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
```

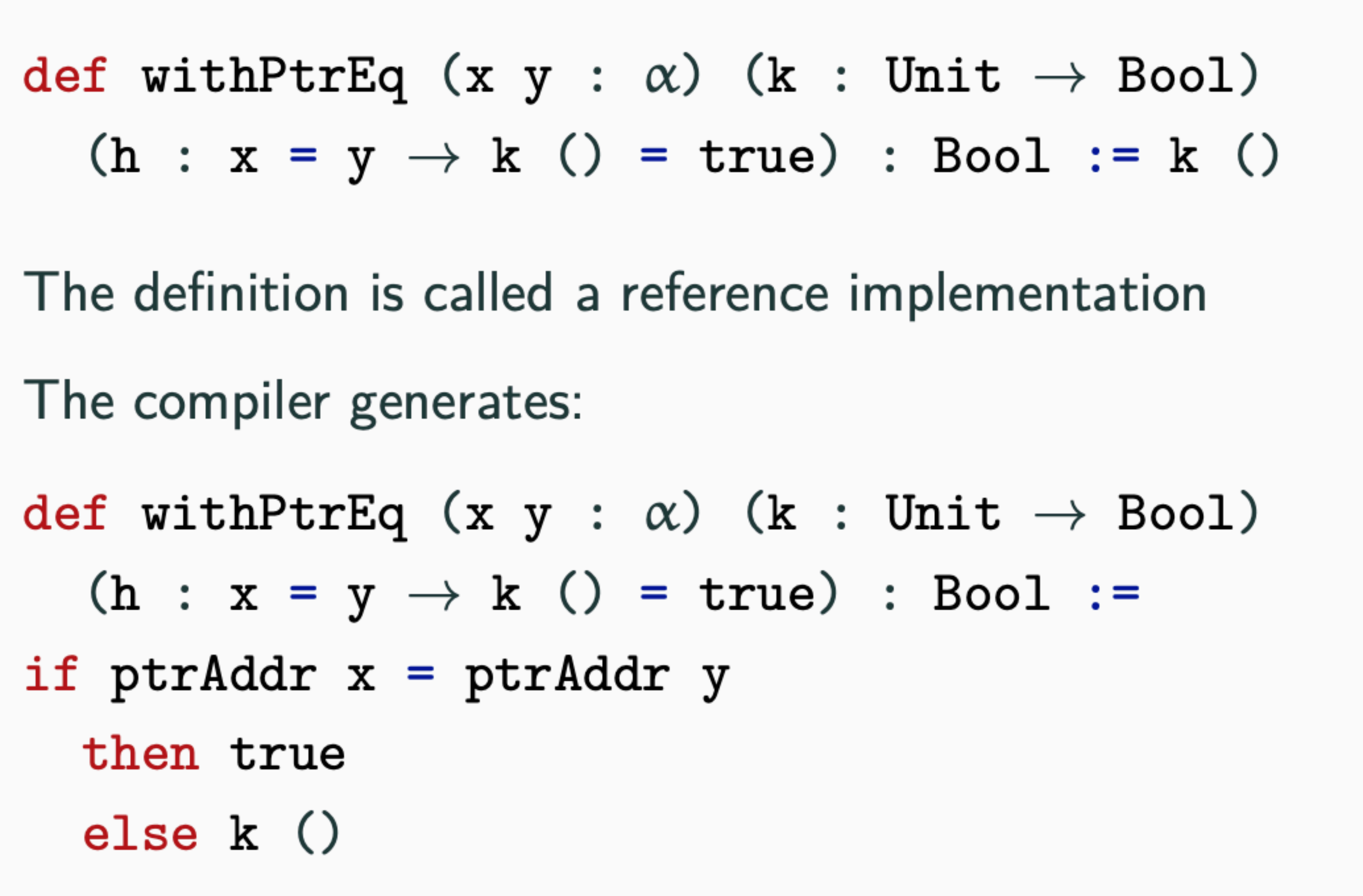
- A partial definition is just syntax sugar for the unsafe + implementedBy idiom.
- Future work: allow users to provide termination later, and use meta programming to generate a safe and non-opaque version of a partial function.

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 ()
```

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).
  - 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.

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

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

$$\mathit{goForward}\,([\,], bs) = ([\,], bs)$$

$$\mathit{goForward}\,(x : xs, bs) = (xs, x : bs)$$

New idioms

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

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

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⟩
```

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