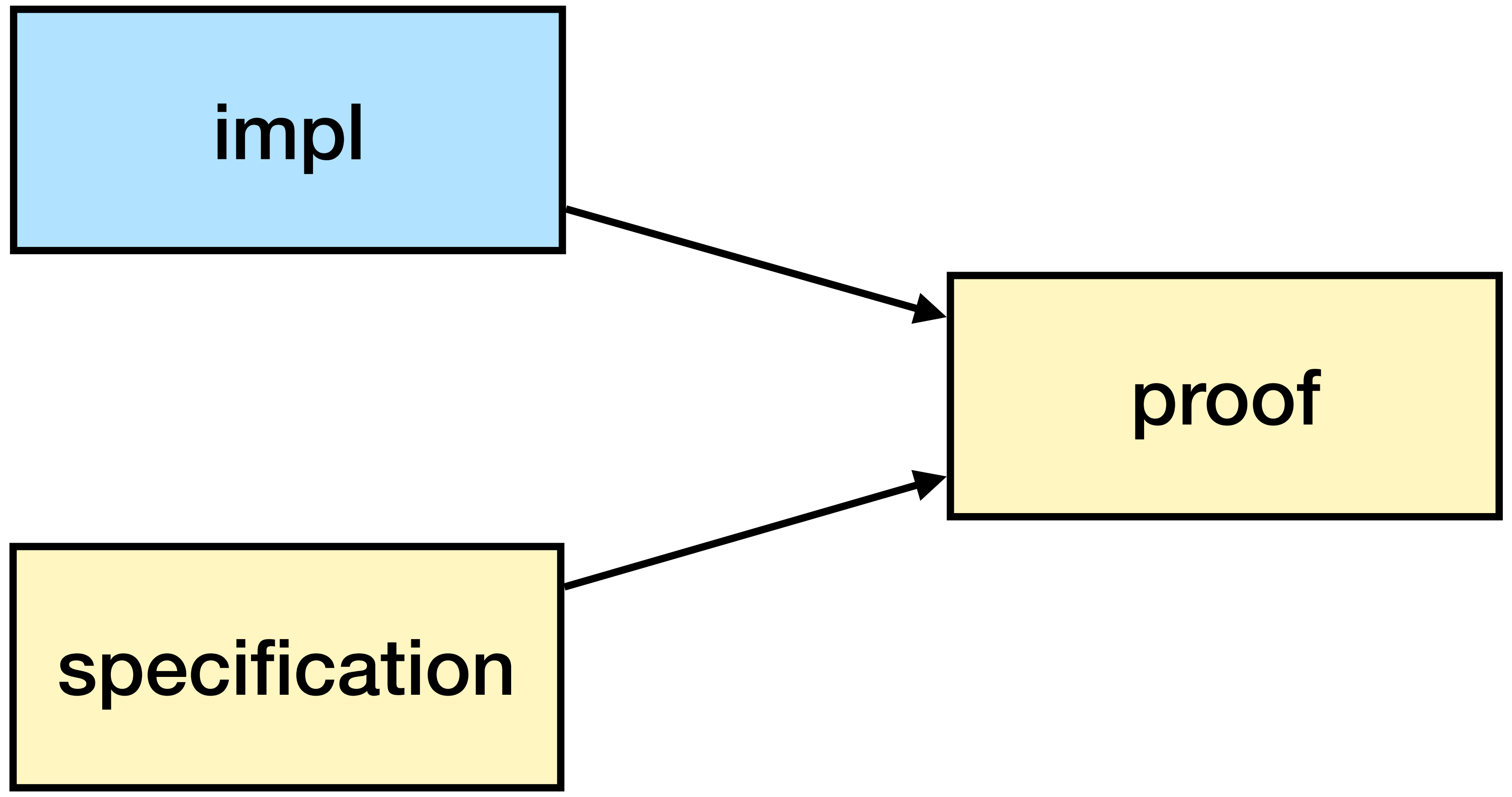
Verifying concurrent Go code in Coq with **Goose**

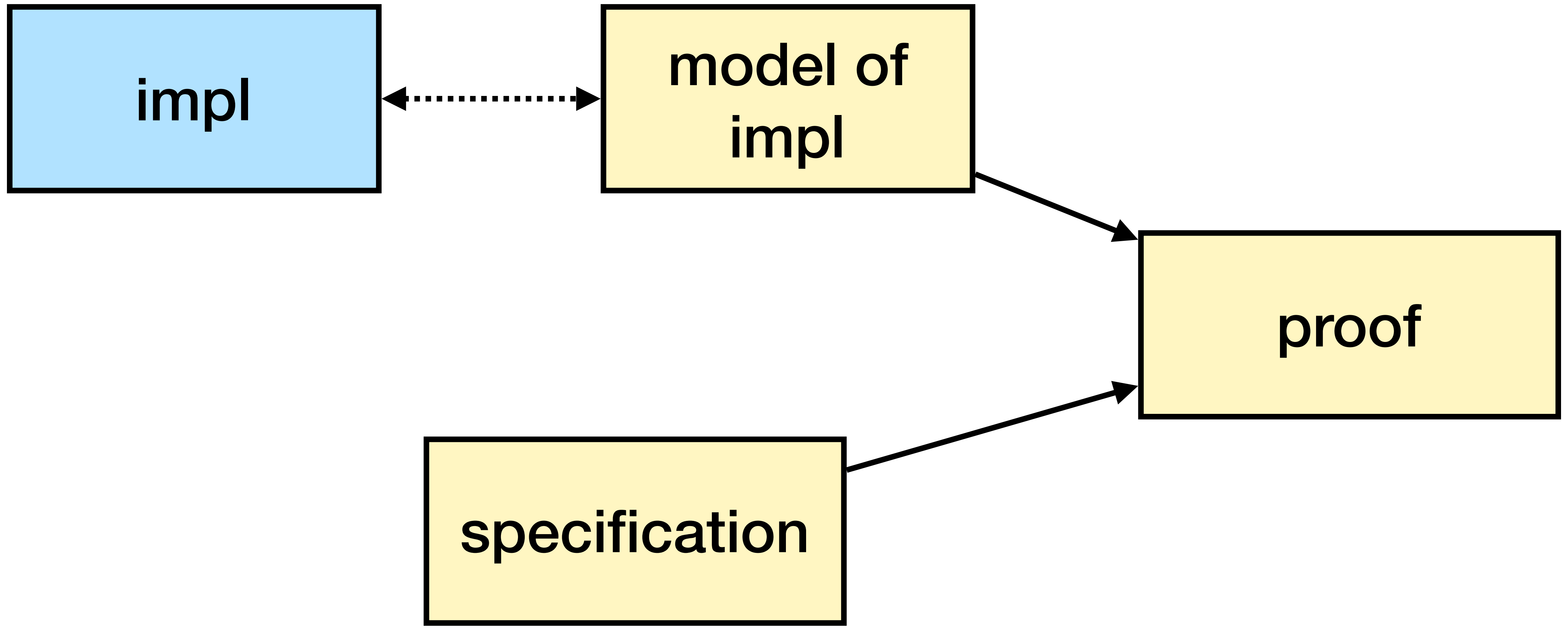
Tej Chajed, Joseph Tassarotti*, Frans Kaashoek, Nickolai Zeldovich

MIT and *Boston College

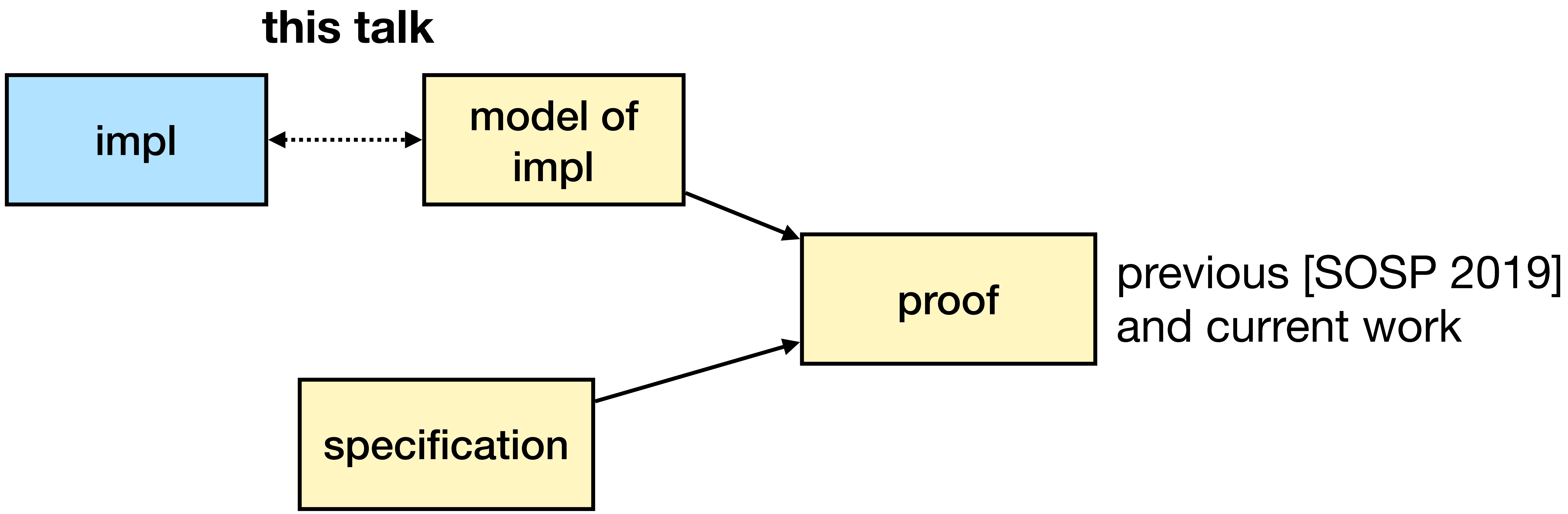
Systems verification, broadly



Systems verification requires connecting implementation to proof



Systems verification requires connecting implementation to proof



We aim to verify realistic systems

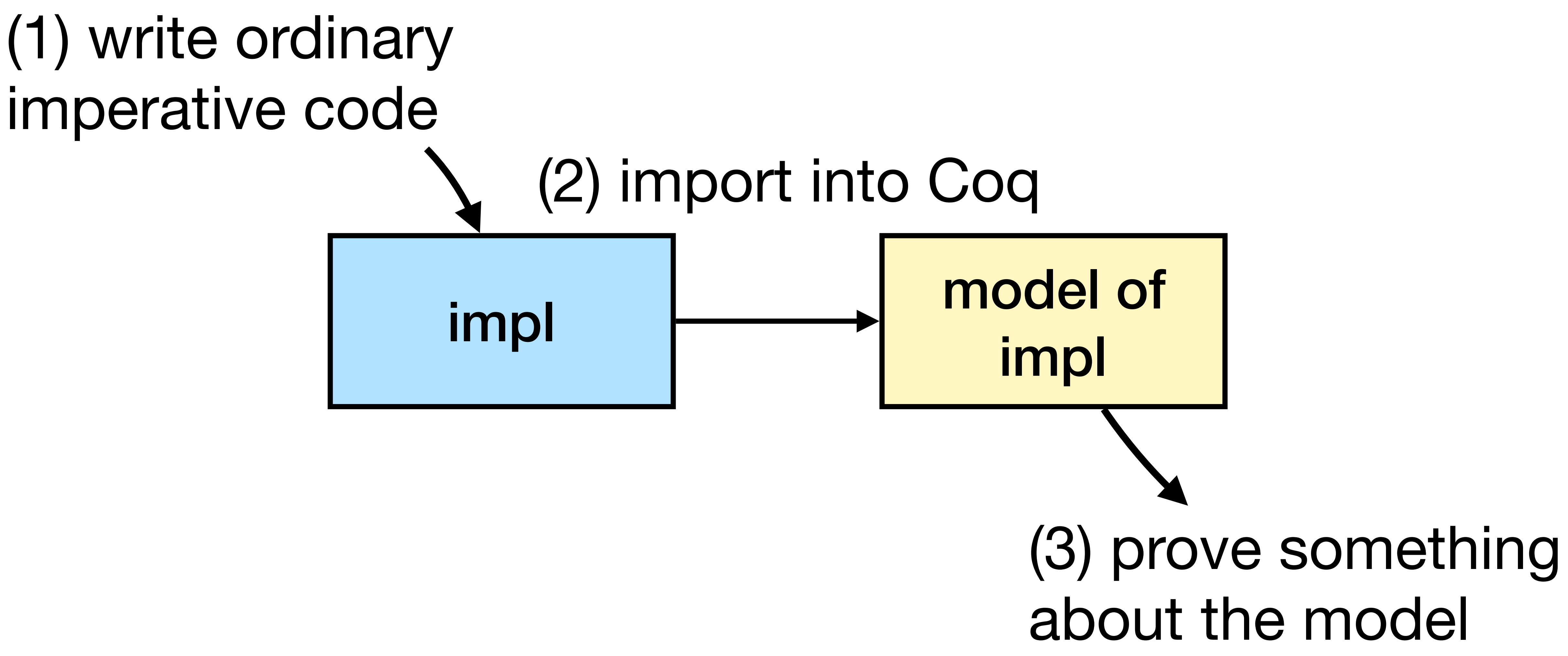
PDOS (the part that does verification)

Systems: running code, interacts with outside world

Realistic: reasonably efficient, concurrency

Verification: functional correctness, focus on crash safety

Goal: implement in a systems language

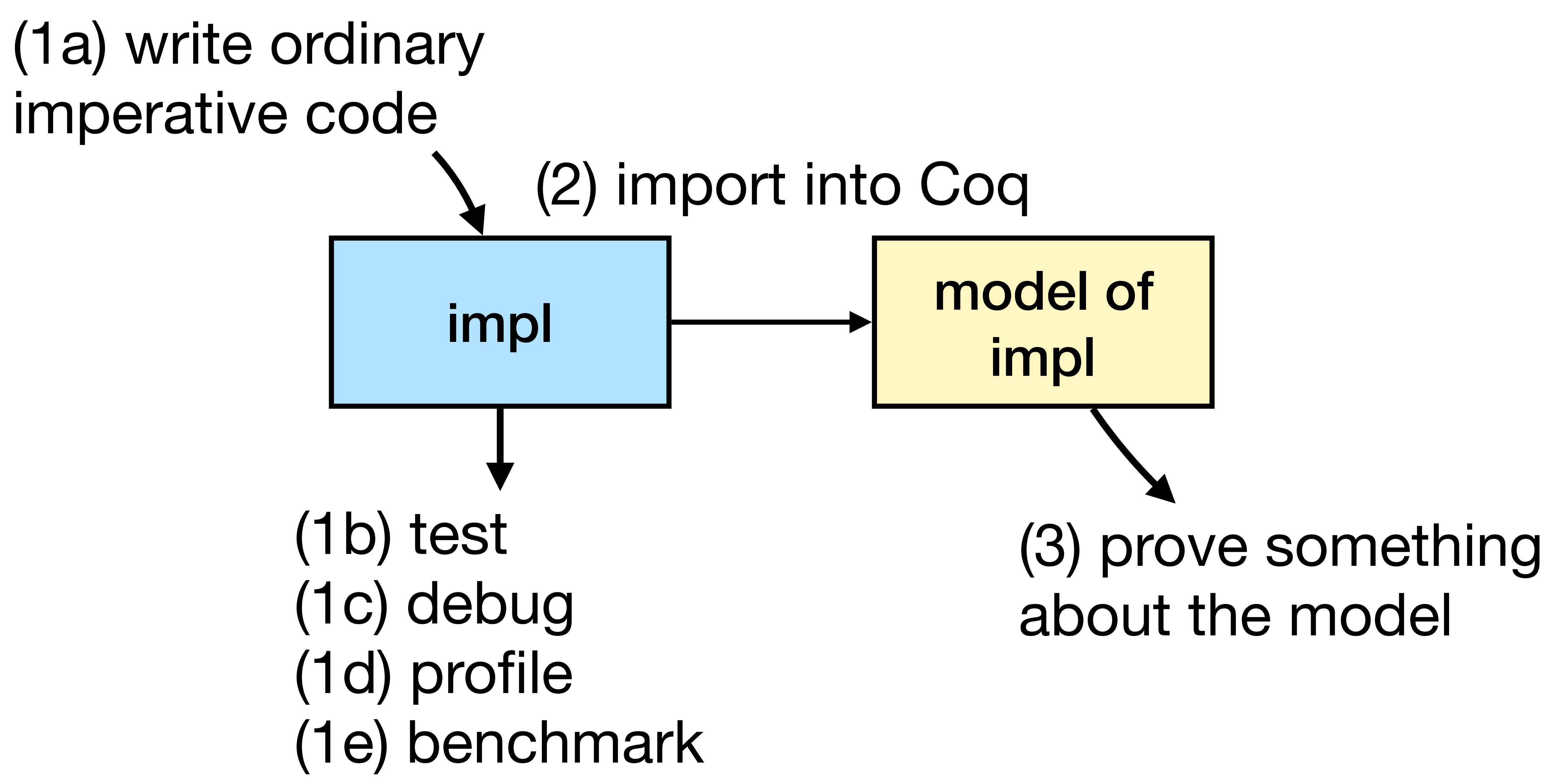


Goose: write code in Go and prove with Iris

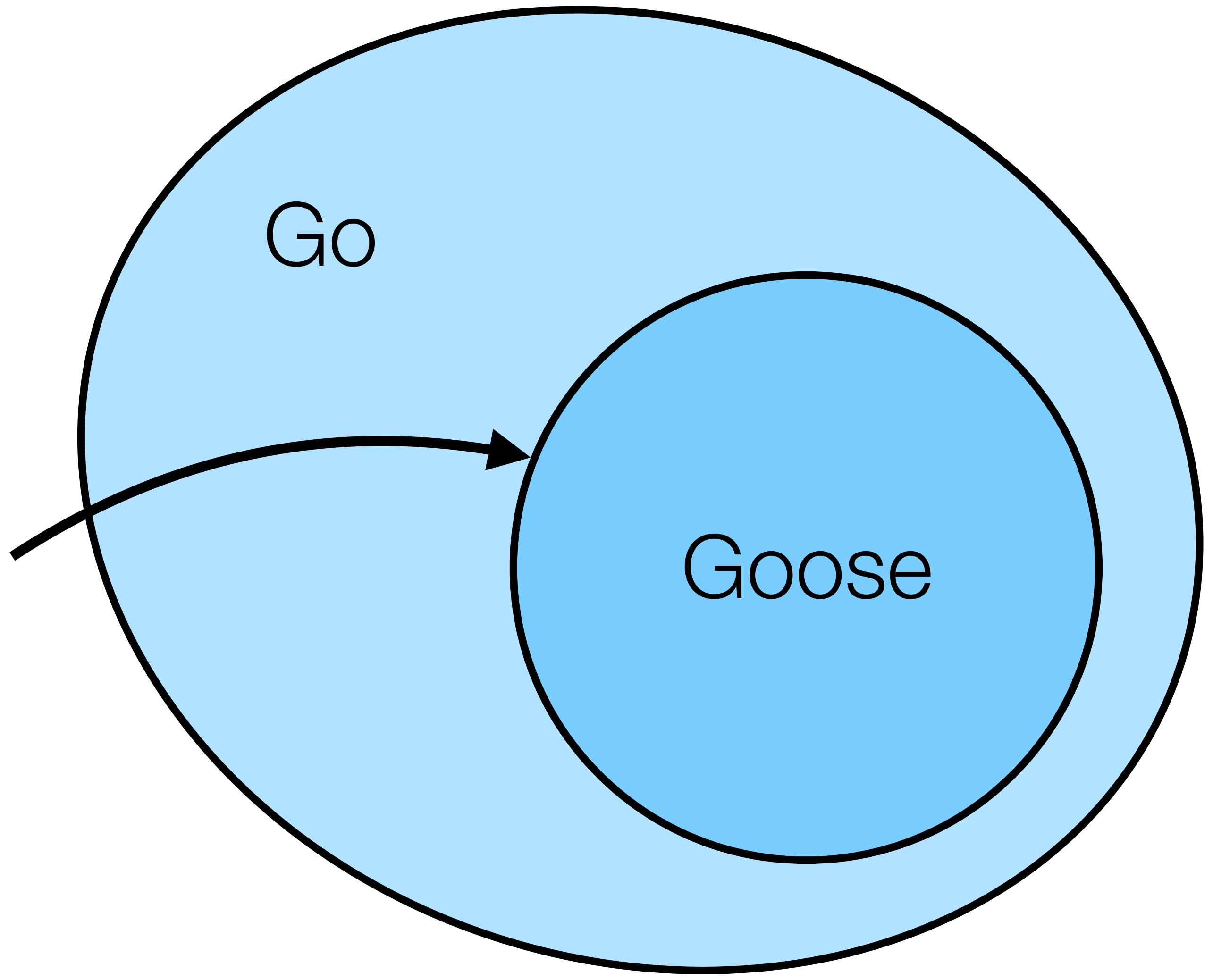
Why Go (vs. C or Rust)**? Simple, good tooling**

Why Iris (vs. VST)**? Concurrency, extensibility**

Implementing in Go helps build the software

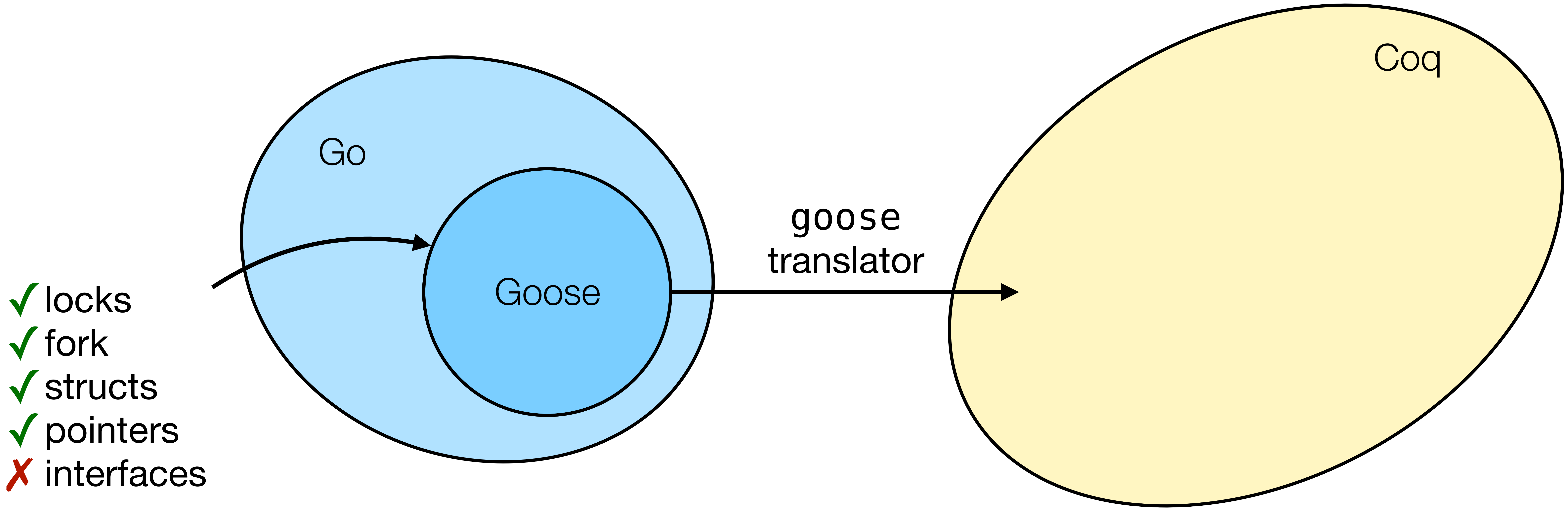


Goose: import subset of Go into a Coq model

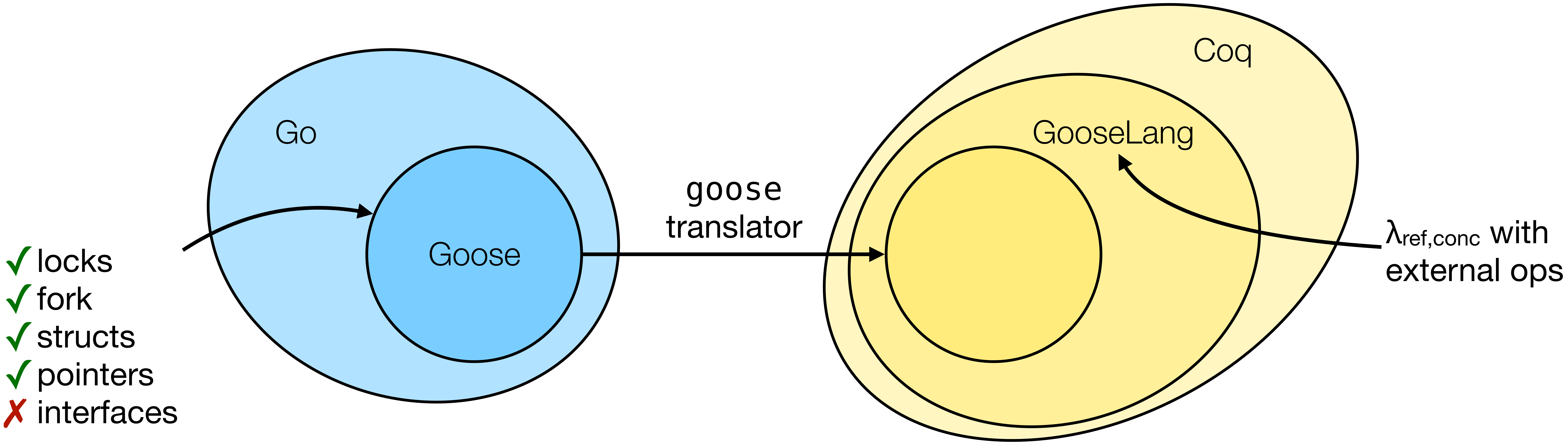


✓ locks
✓ fork
✓ structs
✓ pointers
✗ interfaces

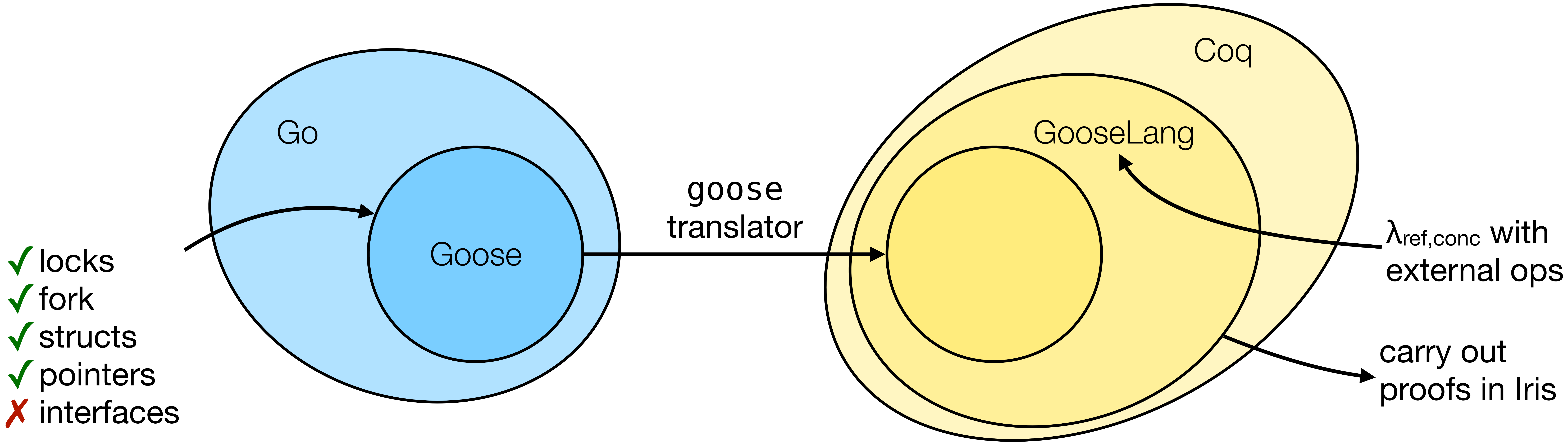
Goose: import subset of Go into a Coq model



Goose: import subset of Go into a Coq model



Goose: import subset of Go into a Coq model



Our systems verification research using Goose

Persistent key-value store using file system (unverified)

Mail server using file system (appeared in SOSP ’19)

Concurrent file system using disk (in progress)

Go is a systems language

C-like: functions, structs, pointers

Exposes system calls

Efficient runtime (garbage collection, threads)

Goose code

Looks like standard Go, but avoids most of the standard library

Use narrow interfaces for file system or disk

More of Go is supported frequently

Challenges in implementing Goose

Defining GooseLang, a semantic model of Go

Translating Go to GooseLang

GooseLang, a semantic model of Go

```
e ::= x | λx. e | e₁ e₂          // λ-calculus
      | ref e | !e | e₁ ← e₂  // heap operations
      | fork e | cmpxchg       // concurrency
```

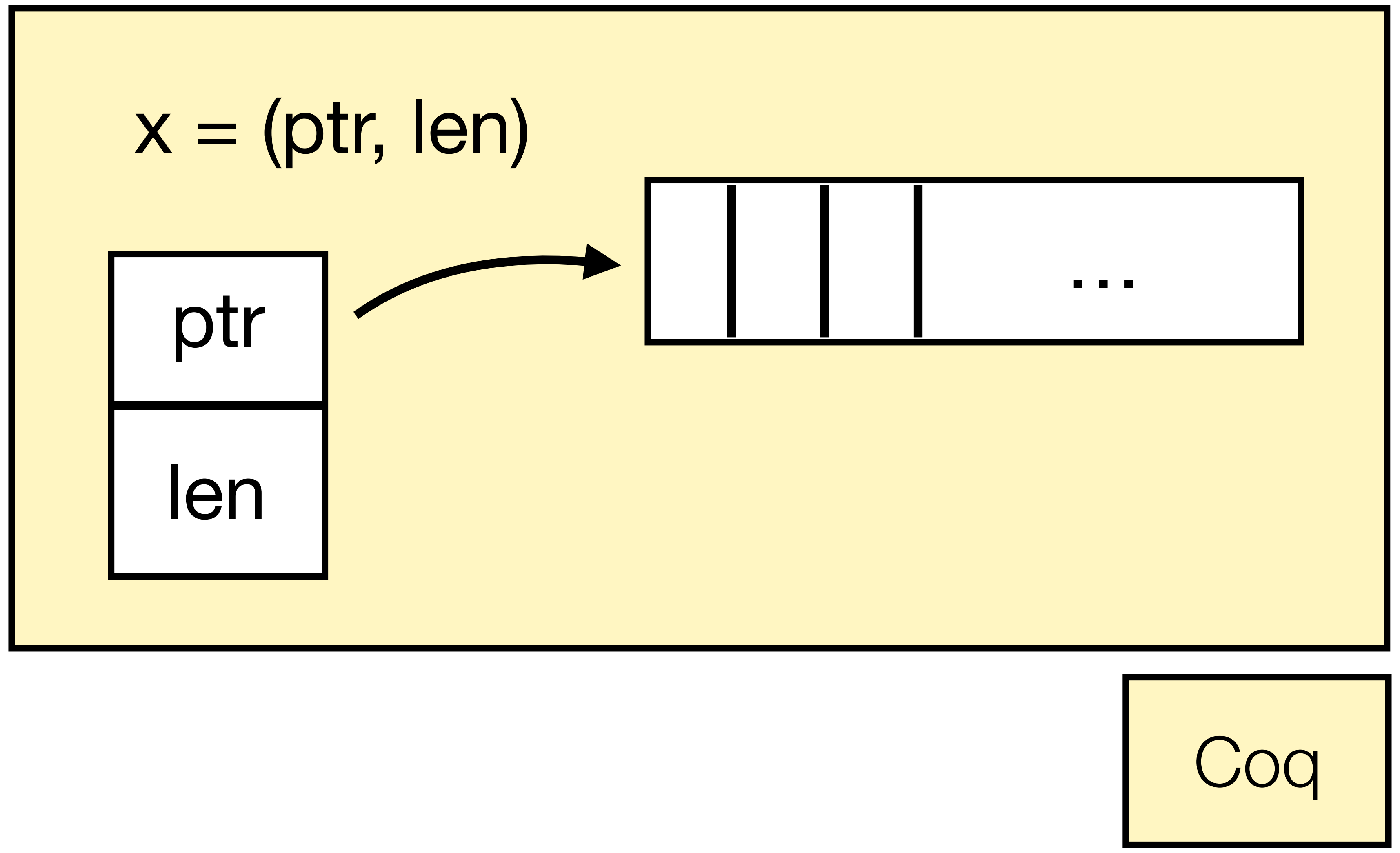
GooseLang, a semantic model of Go

```
e ::= x | λx. e | e₁ e₂           // λ-calculus
      | ref e | !e | e₁ ← e₂  // heap operations
      | fork e | cmpxchg      // concurrency
      | call op e             // external operations
```

GooseLang, a semantic model of Go

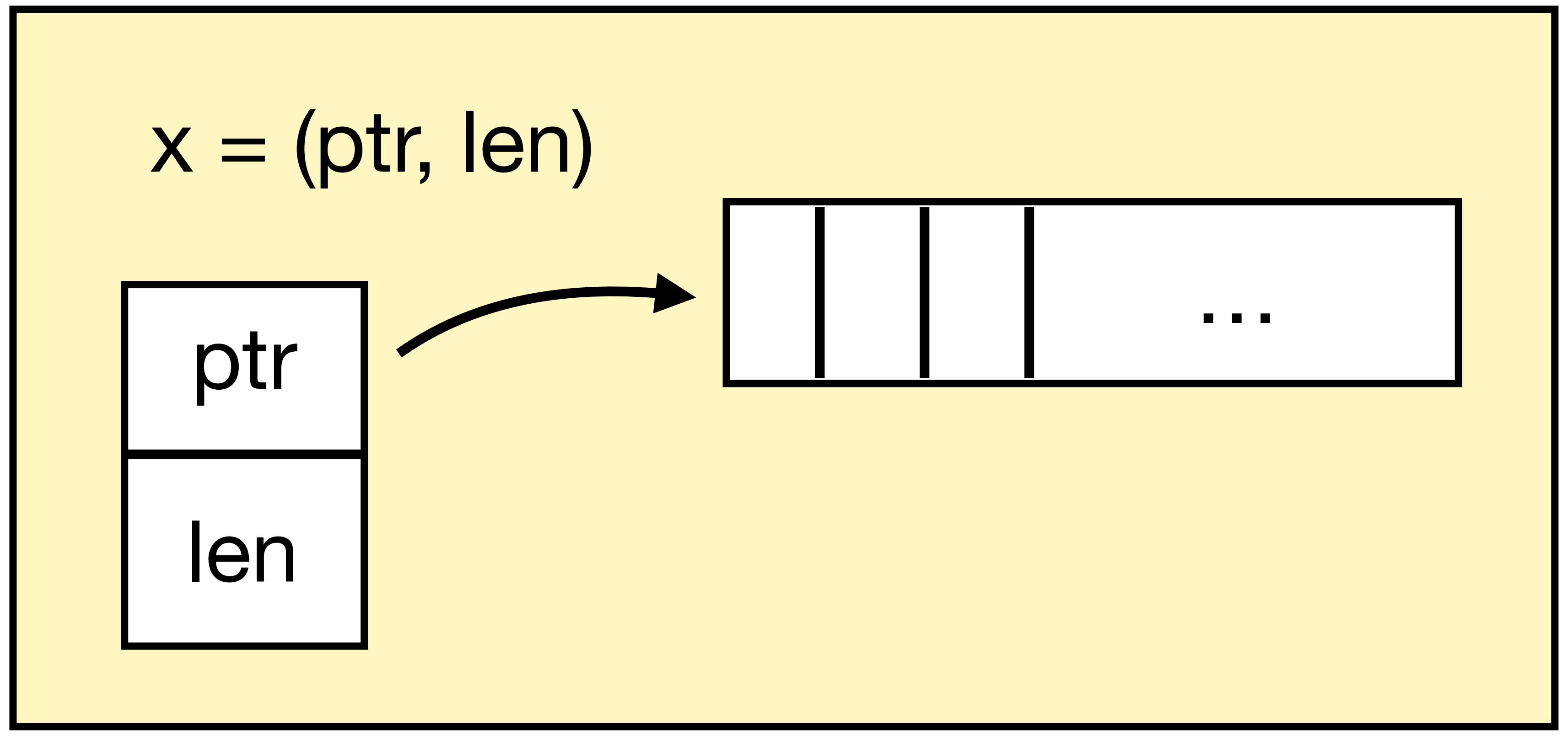
```
e ::= x | λx. e | e₁ e₂          // λ-calculus
      | ref e | !e | e₁ ← e₂  // heap operations
      | fork e | cmpxchg       // concurrency
      | call op e              // external operations

v ::= U64 x | Loc z | …       // literals
      | Pair | InjL | InjR    // sums, products
```

Excerpt from GooseLang: slices




```
Definition sliceAppend :=
   λ s, x.
     let s’ := alloc (s.len + #1) () in
     … (* fill s’ *)
     (s’, s.len + #1).
```

Excerpt from GooseLang: slices



Coq

```
Definition sliceAppend :=
   λ s, x.
     let s' := alloc (s.len + #1) () in
     … (* fill s' *)
     (s', s.len + #1).
```

Go

```
func example(x []uint64) {
  x1 := x[1]
  append(x, 5)
}
```

goose →

```
Definition example :=
  λ x.
    let x1 := !(x.ptr +l #1) in
    sliceAppend x #5;;
    #().
```

Excerpt from GooseLang: modeling concurrency and locking

```
func coin() bool {
  m := new(sync.Mutex)
  x := new(bool)
  go func() {
    m.Lock()
    *x = true
    m.Unlock()
  }()
  m.Lock()
  v := *x
  m.Unlock()
  return v
```

Excerpt from GooseLang: modeling concurrency and locking

```go
func coin() bool {
  m := new(sync.Mutex)
  x := new(bool)
  go func() {
    m.Lock()
    *x = true
    m.Unlock()
  }()
  m.Lock()
  v := *x
  m.Unlock()
  return v
```

goose →

```
Definition coin: val :=
  λ <>.
    let: “m” := lock.new #() in
    let: “x” := ref #(zero_val boolT) in
    fork (lock.acquire “m”;;
          “x” ← #true;;
          lock.release “m”);;
    lock.acquire “m”;;
    let: “v” := !”x” in
    lock.release “m”;;
    “v”.
```

Excerpt from GooseLang: modeling concurrency and locking

```go
func coin() bool {
  m := new(sync.Mutex)
  x := new(bool)
  go func() {
    m.Lock()
    *x = true
    m.Unlock()
  }()
  m.Lock()
  v := *x
  m.Unlock()
  return v
```

goose →

```
Definition coin: val :=
  λ <>.
    let: “m” := lock.new #() in
    let: “x” := ref #(zero_val boolT) in
    fork (lock.acquire “m”;;
          “x” ← #true;;
          lock.release “m”);;
    lock.acquire “m”;;
    let: “v” := !”x” in
    lock.release “m”;;
    “v”.
```

Excerpt from GooseLang: modeling concurrency and locking

```
func coin() bool {
  m := new(sync.Mutex)
  x := new(bool)
  go func() {
    m.Lock()
    *x = true
    m.Unlock()
  }()
  m.Lock()
  v := *x
  m.Unlock()
  return v
```

goose →

```
Definition coin: val :=
  λ <>.
    let: “m” := lock.new #() in
    let: “x” := ref #(zero_val boolT) in
    fork (lock.acquire “m”;;
          “x” ← #true;;
          lock.release “m”);;
    lock.acquire “m”;;
    let: “v” := !”x” in
    lock.release “m”;;
    “v”.
```

Excerpt from GooseLang: modeling concurrency and locking

```
func coin() bool {
  m := new(sync.Mutex)
  x := new(bool)
  go func() {
    m.Lock()
    *x = true
    m.Unlock()
  }()
  m.Lock()
  v := *x
  m.Unlock()
  return v
```

goose →

```
Definition coin: val :=
  λ <>.
    let: "m" := lock.new #() in
    let: "x" := ref #(zero_val boolT) in
    fork (lock.acquire "m";;
          "x" ← #true;;
          lock.release "m");;
    lock.acquire "m";;
    let: "v" := !"x" in
    lock.release "m";;
    "v".
```

Challenge in modeling Go: weak memory

```
func uhOh(x *uint64) {
  go func() {
    *x = 1
    print(“set x”)
  }()
  print(“x=”, *x)
}
```

goose →

```
Definition uhOh: val :=
  λ x.
    fork (x ← #1
            print “set x” !x);;
    print “x=” !x.
```

x86-TSO

Sequential consistency

If we first see “set x”, then

Challenge in modeling Go: weak memory

```
func uhOh(x *uint64) {
  go func() {
    *x = 1
    print("set x")
  }()
  print("x=", *x)
}
```

x86-TSO

goose →

```
Definition uhOh: val :=
  λ x.
    fork (x ← #1
          print "set x" !x);;
 ✓  print "x=" !x.
```

Sequential consistency

If we first see "set x", then
sequential consistency means x=1

Challenge in modeling Go: weak memory

```
func uhOh(x *uint64) {
  go func() {
    *x = 1
    print("set x")
  }()
  print("x=", *x)
}
```

x86-TSO

goose →

```
Definition uhOh: val :=
  λ x.
    fork (x ← #1
          print "set x" !x);;
    print "x=" !x.
```

Sequential consistency

If we first see "set x", then
sequential consistency means x=1
but TSO allows x=0

Challenge in modeling Go: weak memory

```
func uhOh(x *uint64) {
  go func() {
✗   *x = 1
    print("set x")
  }()
  print("x=", *x)
}
```

goose →

```
Definition uhOh: val :=
  λ x.
    fork (x ← #1
✓         print "set x" !x);;
    print "x=" !x.
```

~~Sequential consistency~~

x86-TSO

If we first see "set x", then
sequential consistency means x=1
but TSO allows x=0

Disallow racy loads and stores

```
Definition Store: val :=
  λ p, v. BeginStore p;;
          FinishStore p v.
Notation “p ← v” := (Store p v).
```

Track in-progress stores

Concurrent store/store and load/store are undefined

Compatibility with Iris gives us amazing verification technology

Concurrent separation logic with higher-order ghost state

Iris Proof Mode (IPM) for interactive proofs

Compatibility with Iris gives us amazing verification technology

Concurrent separation logic with higher-order ghost state

Iris Proof Mode (IPM) for interactive proofs

Connect to our unwritten POPL 2021 paper for crash safety

Proofs using non-atomic memory

Load

$\{p \mapsto v\}$

$!p$

$\{\lambda v\,.\, p \mapsto v\}$

(non-atomic) Store

$\{p \mapsto v_0\}$

$p \leftarrow v$

$\{p \mapsto v\}$

These triples are sound because

$p \mapsto v$ is *exclusive* access to p

Proofs using non-atomic memory

Load

$\{p \mapsto v\}$

$!p$

$\{\lambda v\,.\,p \mapsto v\}$

(non-atomic) Store

$\{p \mapsto v_0\}$

$p \leftarrow v$

$\{p \mapsto v\}$

These triples are sound because

$p \mapsto v$ is *exclusive* access to p

→ exclude using locks

→ exclude by using local variables

GooseLang programs can make system calls

```
import “github.com/tchajed/goose/
machine/disk"

func Copy() {
  b := disk.Read(0)
  disk.Write(1, b)
}
```

goose →

```
Import disk.


Definition Copy: val :=
  λ_.
    let b := call ReadOp #0 in
    call WriteOp (#1, b).
```

Language is parameterized by external calls

Currently implementing GooseLang + file-system ops in terms of GooseLang + disk ops

Semantics of GooseLang

Small-step operational semantics, mostly standard and following design of HeapLang

For testing, have executable semantics (interpreter + soundness proof)

Previous approach: shallow embedding as semantic model

GooseLang was a free monad instead of a λ-calculus

Go code had to explicitly sequence effectful operations

Pure operations were expressed directly in Gallina

GooseLang is a mix of shallow and deep embedding

Heap operations, concurrency are deeply represented

Data structures are shallowly built out of sums

Goose translator

2.5k lines of Go

Implemented using `go/ast` and `go/types`

Single pass, per function

Goose translator supports enough Go

multiple return values

early return

for loops

slice and map iteration

panic

struct field pointers

struct literals

slice element pointers

sub-slicing

pointers to local variables

mutexes and cond vars

goroutines

++ and +=

uint64, uint32, bytes

bitwise ops

Goose supports more of Go whenever Frans and Nickolai need something

my advisors

✓Multiple packages
✓First-class functions
✓Interfaces and type casts

Goose supports more of Go whenever Frans and Nickolai need something

my advisors

✓ Multiple packages
✓ First-class functions
✓ Interfaces and type casts
✗ Channels

Goose supports more of Go whenever Frans and Nickolai need something

my advisors

- ✓ Multiple packages
- ✓ First-class functions
- ✓ Interfaces and type casts
- ✗ Channels
- ✗ Control flow like `return` from loop, `defer`

Making the goose translator sound

Simple and syntactic translation

Make mistakes result in **undefined behavior**

Basic **type checking** catches many mistakes

Hand-audited integration **tests**

Related work

Extraction

VST and CompCert

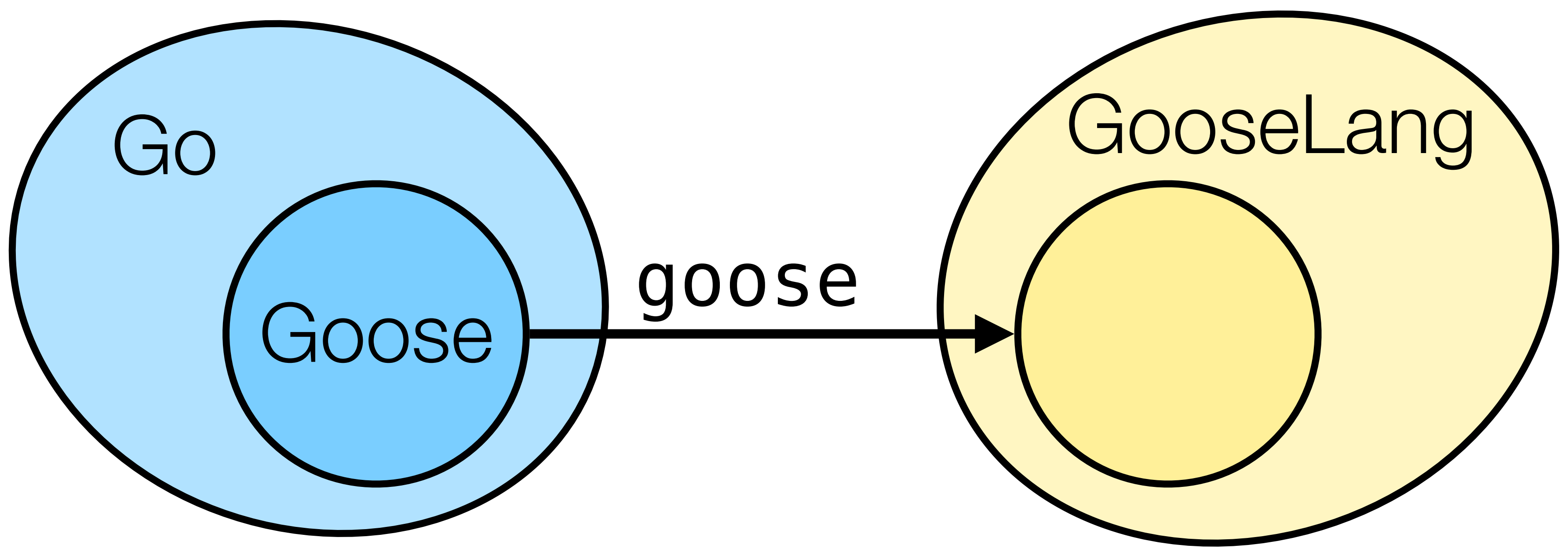
RustBelt

Ongoing work

Scaling Goose: handling a large, efficient program

Testing: using executable semantics to test translator

Conclusion



Goose is a new approach to concurrent systems verification: imports Go into Coq

Actively using it for current research

Come talk to us!

Tej and Joe are at CoqPL

https://github.com/tchajed/goose