Toward Null-Safety

… for Many Millions of Lines of C++ Code

Jan Voung, presenting work by the Google C++ Nullability Team

Proprietary + Confidential

Agenda

01 Intro

02 Checker (function-scope)

03 Inference (cross-function, cross-TU)

04 Experience

Lots of pointers and developers + unclear nullability

```
std::unique_ptr<P> create(const Q &q);

create(arg)->use(); // safe?
```

Lots of pointers and developers + unclear nullability

```
std::unique_ptr<P> create(const Q &q);

create(arg)->use(); // safe?
```

allocator? (nonnull)

or factory function that could fail? (nullable)

Lots of pointers and developers + unclear nullability

```
std::unique_ptr<P> create(const Q &q);

create(arg)->use();
```

allocator? (nonnull)

or factory function that could fail? (nullable)

XY% of C++ crashes at Google

Lots of pointers and developers + unclear nullability

```
std::unique_ptr<P> create(const Q &q);
crea
alloca
or fac
```

- **Testing** not enough.
- May be safe now... but will it remain safe?

Want: statically check for null safety

XY% of C++ crashes at Google

Annotations Clarify and Make **Checkable**

```
// Returns null if q is invalid…
_Nullable std::unique_ptr<P> create(const Q &q);

create(arg)->use();  // ❌
```

Build on Apple's proposal from 2015
https://discourse.llvm.org/t/rfc-nullability-qualifiers/35672

Three states for gradual adoption

`_Nonnull`
`_Nullable`
`_Null_unspecified` (default for unannotated)

Unannotated: Gradual, but uncaught issues

```
void Foo() {
  // Unspecified might be Nullable, so warn?
  // Or might be Nonnull (more common!)... avoid FPs
  UnspecReturn()->...; // no warning
}
```

Unannotated: Gradual, but uncaught issues

```
void Foo() {
  // Unspecified might be Nullable, so warn?
  // Or might be Nonnull (more common!)... avoid FPs
  UnspecReturn()->...; // no warning
}
```

Goal: annotate/classify all the pointers!

Unannotated: Gradual, but uncaught issues

```
void Foo() {
}
```

- Millions of "legacy" pointers.
- X mins each to manually annotate and review...
- Automatically infer?

Goal: annotate/classify all the pointers!

~1 annotation when no more `_Unspecified`

~~`T* _Nonnull` vs `T* _Nullable`~~

`DEFAULT_NONNULL`

`T*` vs `T* _Nullable`

(or `T&`)

Proprietary + Confidential

Agenda

01 Intro

02 Checker (function-scope)

03 Inference (cross-function, cross-TU)

04 Experience

Check beyond Clang's -Wnonnull

```
void DoWithDefault(T *_Nullable p, T *def) {
  UseNonnull(nullptr);  // -Wnonnull warns
  UseNonnull(p);        // -Wnonnull does not warn

  if (p == nullptr) {
    p = def;
  }
  UseNonnull(p); // ✅ (then: p is overwritten with
def
                        else: p != nullptr)
}
```

Legacy code with correlated branches

```
void Foo(T *_Nullable p, bool feature) {
  if (feature && p == nullptr)
    p = new …;

  // much later
  if (feature) // not checking p again here
    *p;
}
```

Brief overview: track complex branch conds + SAT solver

```
void Foo(T *_Nullable p, T *u) {



  if (p == nullptr) {

    p = u;
  }
  *p;  // ✅
}
```

Pointer State

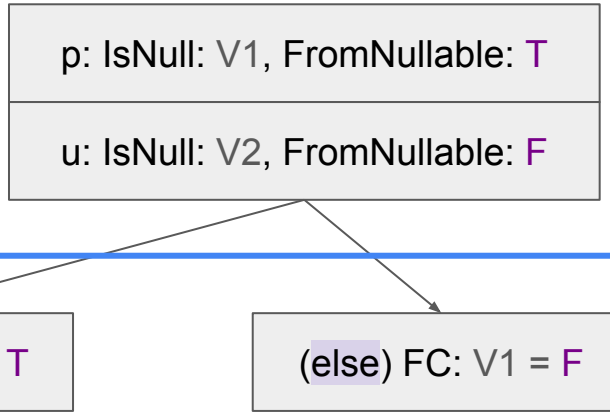
```
void Foo(T *_Nullable p, T *u) {


  if (p == nullptr) {

    p = u;
  }
  *p;  // ✅
}
```

p: IsNull: V1, FromNullable: T
u: IsNull: V2, FromNullable: F

- two bools to separate: {nullptr, Nonnull, Nullable, Unspec}
- bools compatible with SAT solver
- some literals, some variables (e.g., for Nullable)
- Clang Dataflow also tracks pointees

Constraints at branches ("Flow Condition")

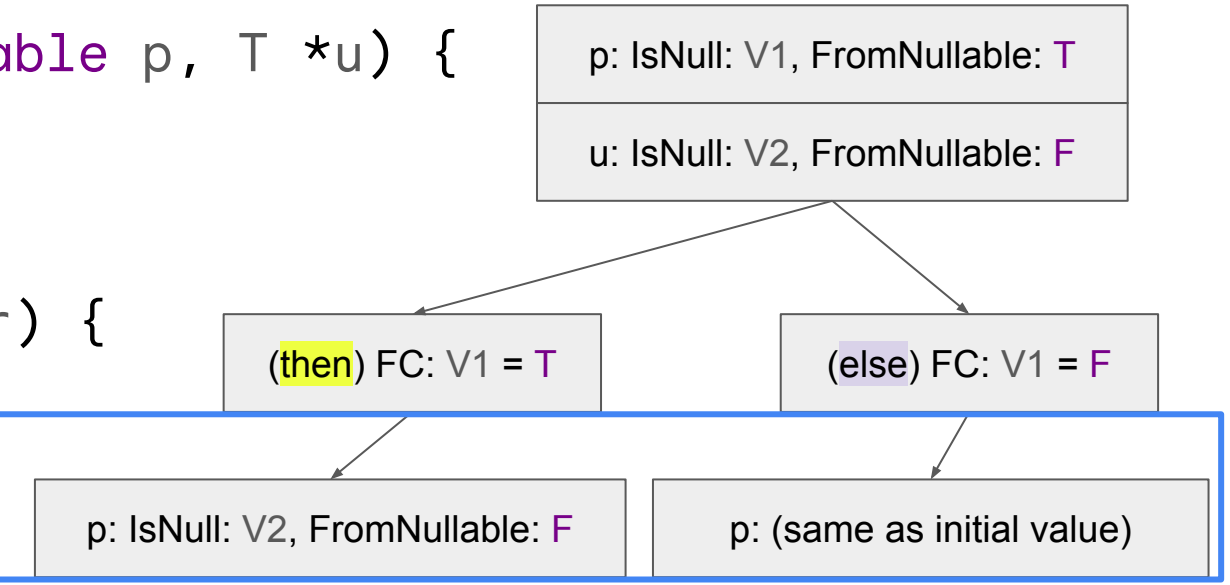
```
void Foo(T *_Nullable p, T *u) {

  if (p == nullptr) {
    p = u;
  }
  *p;  // ✅
}
```



- ... some literals, some variables ...
- p's `IsNull == V1`
- so... `if (p == nullptr)` leads to
- then: V1 = True, vs else: ...

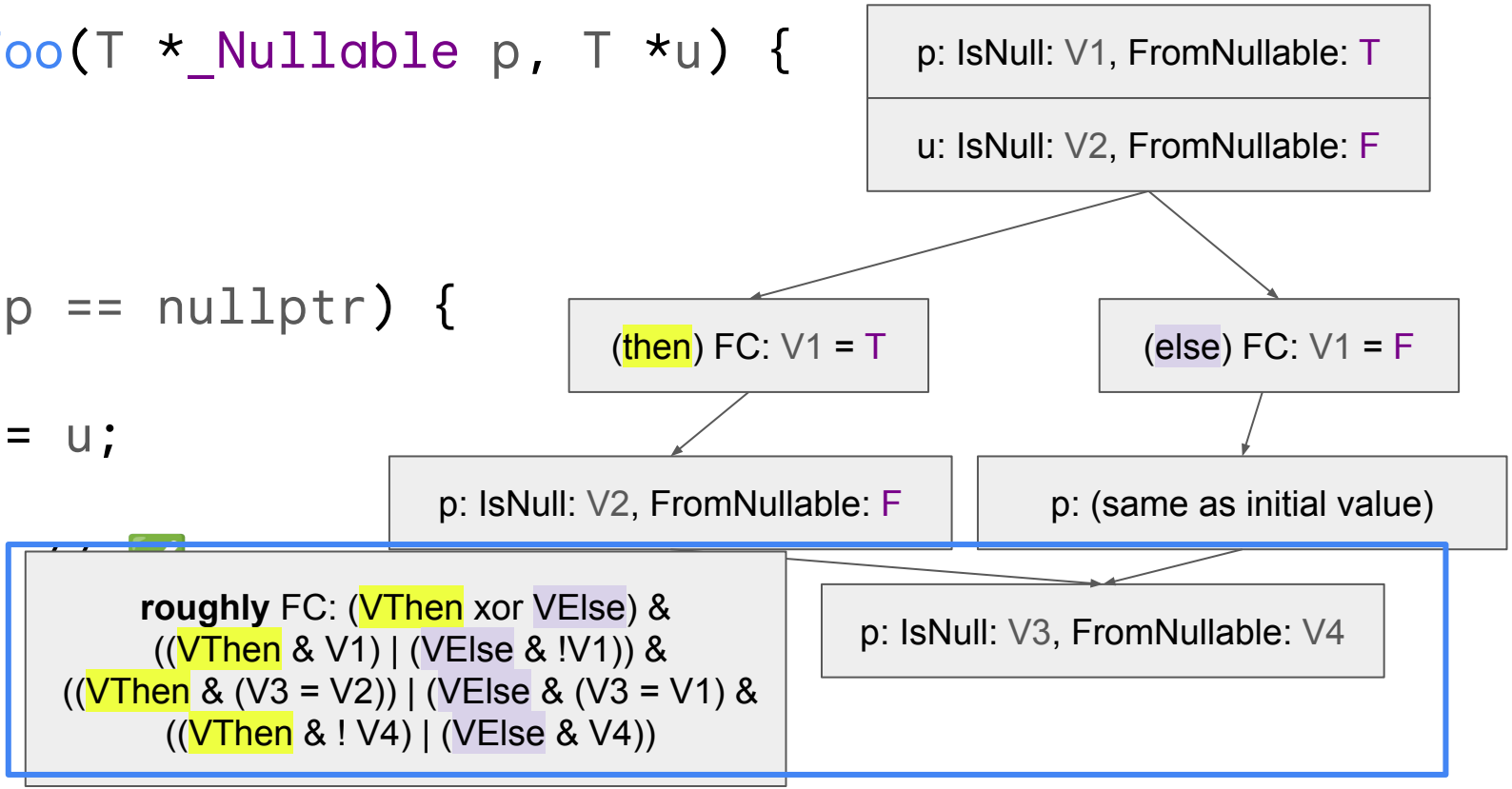
Assignments

```
void Foo(T *_Nullable p, T *u) {

  if (p == nullptr) {
    p = u;
  }
  *p;  // ✅
}
```



Join

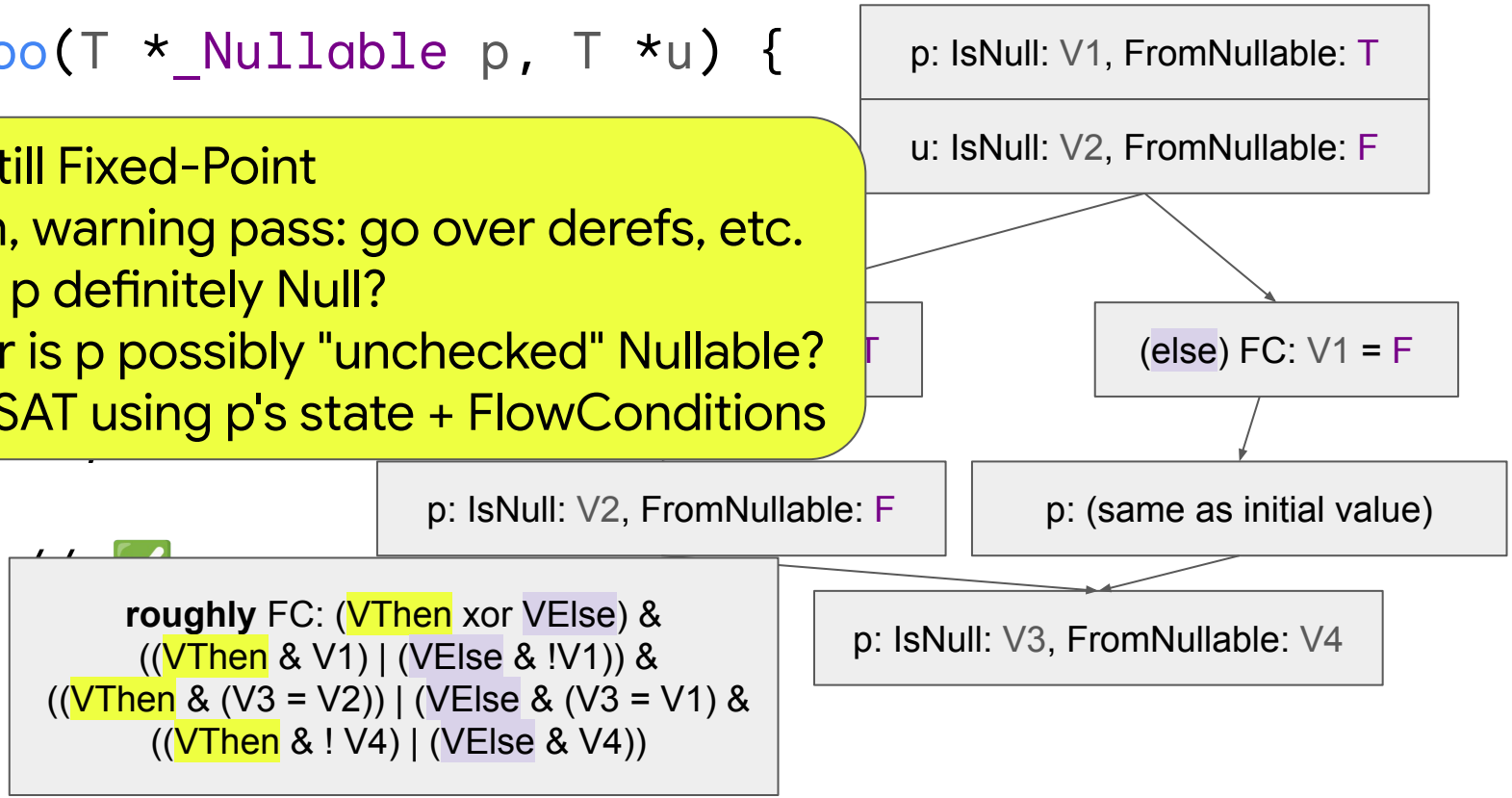
```
void Foo(T *_Nullable p, T *u) {


  if (p == nullptr) {

    p = u;
  }
  *p;
}
```



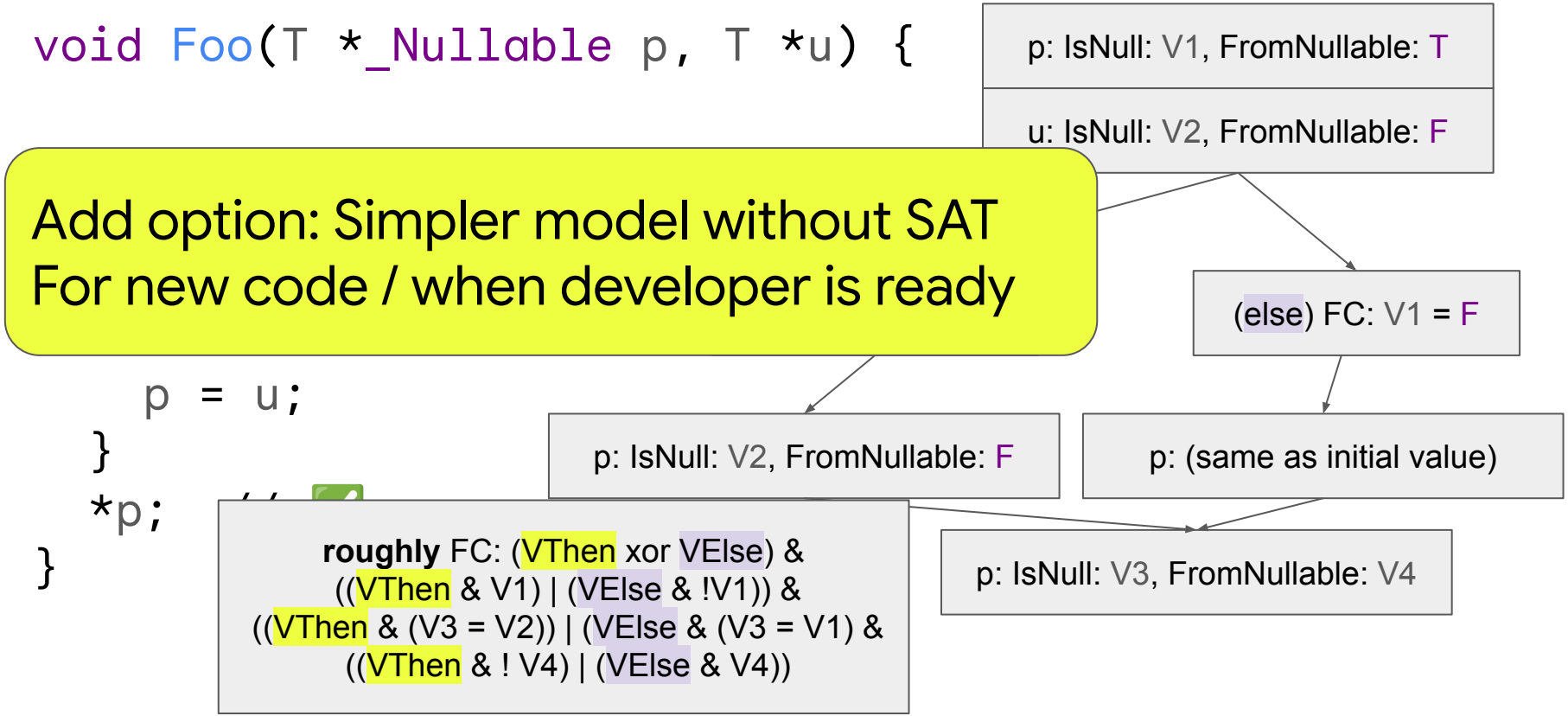
Warn or not? Two conditions, check with SAT solver

```
void Foo(T *_Nullable p, T *u) {
  }
  *p;
}
```

- Run till Fixed-Point
- Then, warning pass: go over derefs, etc.
- (1) Is p definitely Null?
- (2) or is p possibly "unchecked" Nullable?
- Ask SAT using p's state + FlowConditions



```
void Foo(T *_Nullable p, T *u) {
    p = u;
  }
  *p;
}
```



Other checker features

- **templates**
  - annotations are type sugar: only one version of vector<int*>
  - so "**resugar**" to recover vector<int* _Nonnull> vs _Nullable
  - `vector<int* _Nonnull> v; v.push_back(nullptr); //` ❌
- types with multiple pointers (vector of nullabilities)
- **common slightly inter-procedural**
  - macros: CHECK_NE(x, nullptr)
  - accessors: if (X.getFoo() != nullptr) X.getFoo()-> (vs Y.getFoo())
- smart pointers (some relaxation for Nonnull to allow `std::move`)
- more complete default nonnull (covering vector<int*>, etc.)
- lambdas (basic support)
- ... more

Proprietary + Confidential

Agenda

01 Intro

02 Checker (function-scope)

03 Inference (cross-function, cross-TU)

04 Experience

Alternatives to inference

1. **manual annotation**: takes too long

   (introduced annotations along with checker in 2024, not much progress)

2. **Pull from comments with AI:** only a few percent of APIs have documentation

```
// Must check return
// TODO: clarify…
int *Foo(int *P, int *Q);
```

Inference intuition: learn from return, assign, call, deref, ...

```
int* ? Foo(int *_Nullable p) {
  return p;
}
```

Checked vs unchecked

```
int* ? Foo(int *_Nullable p) {
  return p;
}

int* ? Bar(int *_Nullable p) {
  if (p != nullptr)
    return p;
  return &kDefault;
}
```

Connection to Checker!

```
int* ? Foo(int *_Nullable p) {
  return p;
}

int* ? Bar(int *_Nullable p) {
  if (p != nullptr)
    return p;
  return &kDefault;
}
```

Hey Checker:
what is p's state at the return?

Alternatively: would we get a warning if Nonnull vs Nullable?

Are derefs safe if Nonnull vs Nullable (for each param)

```
int Foo(int * ? p, int * ? q) {
  if (p != nullptr)
    return *p;
  return *q;
}
```

Can P be Nullable or must be Nonnull?

```
int Foo(int * ? p, int * ? q) {
  if (p != nullptr)
    return *p;
  return *q;
}
```

Try hypotheses: p Nullable vs p Nonnull (and q ...)

Result: doesn't matter according to Checker (already checked)

Can Q be Nullable or must be Nonnull?

```
int Foo(int * ? p, int * ? q) {
  if (p != nullptr)
    return *p;
  return *q;
}
```

Try hypotheses: q Nullable vs q Nonnull (and p...)

Result: q must be Nonnull according to Checker (unchecked)

Rerun analysis N times vs Symbolic state for "sources"

```
int Foo(int * ? p, int * ? q) {
  if (p != nullptr)
    return *p;
  return *q;
}
```

q: IsNull: V3, FromNullable: T

q: IsNull: F, FromNullable: F

… vs
"symbolic"

q: IsNull: V3, FromNullable: V4

More general Flow Condition in terms of variables

```
int Foo(int * ? p, int * ? q) {
  if (p != nullptr)
    return *p;
  return *q;
}
```

p: IsNull: V1, FromNullable: V2

q: IsNull: V3, FromNullable: V4

FC: V1 == F

FC: no constraints on V3 or V4

Try hypotheses with Additional Constraints

```
int Foo(int * ? p, int * ? q) {
  if (p != nullptr)
    return *p;
  return *q;
}
```

p: IsNull: V1, FromNullable: V2

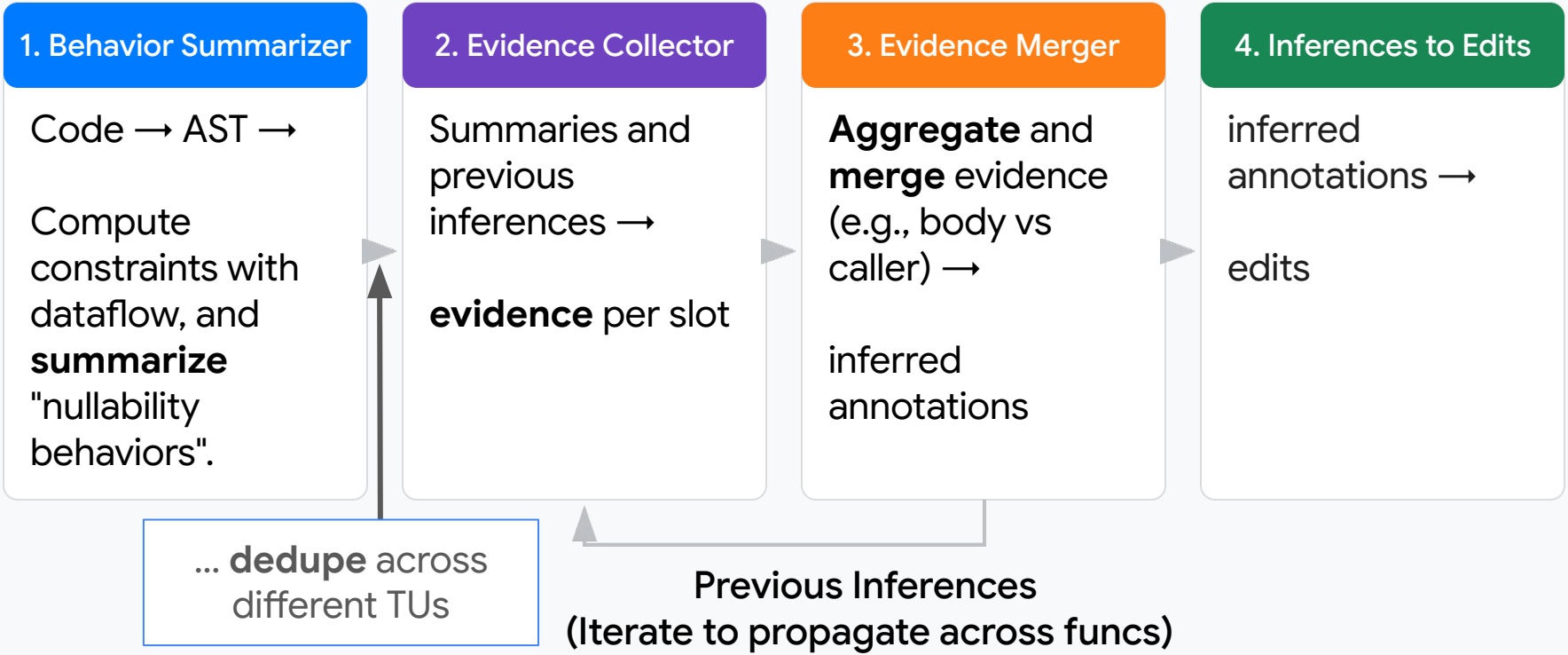
q: IsNull: V3, FromNullable: V4

FC: V1 == F

FC: no constraints on V3 or V4

Try Q Nonnull with Additional Constraint:
V3 = F & ... => ...

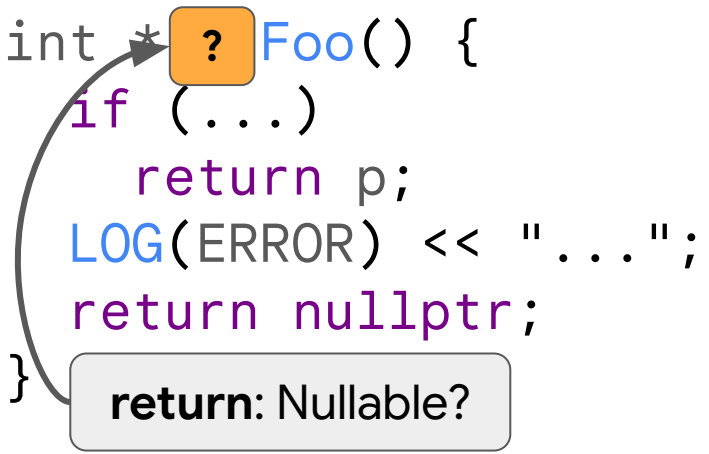
Per TU, Merge, iterate for Cross TU



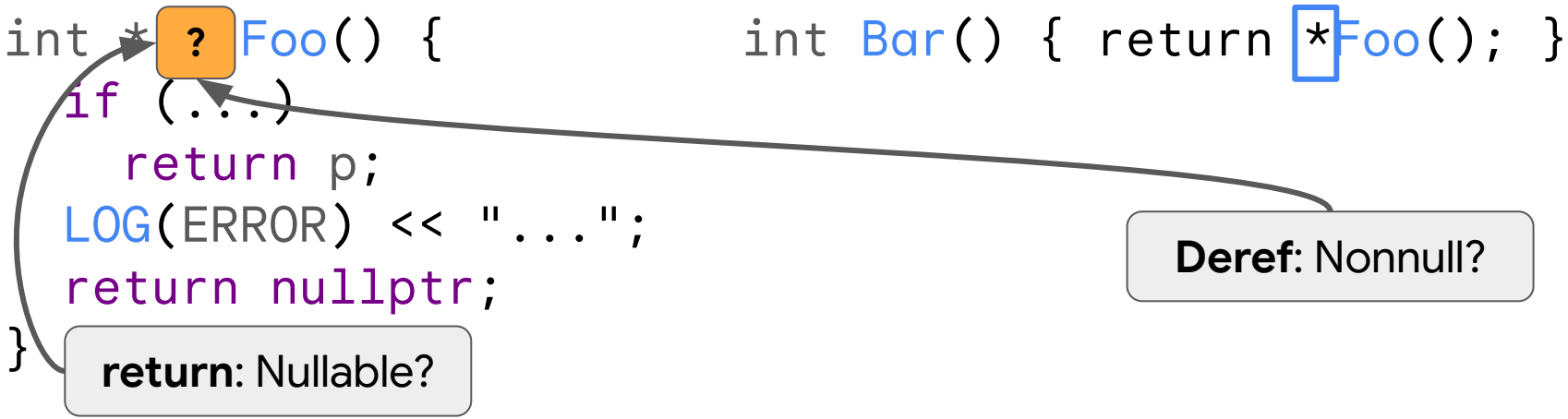
Body vs Caller: Merging Evidence

```
int *[?]Foo() {
  if (...)
    return p;
  LOG(ERROR) << "...";
  return nullptr;
}
```

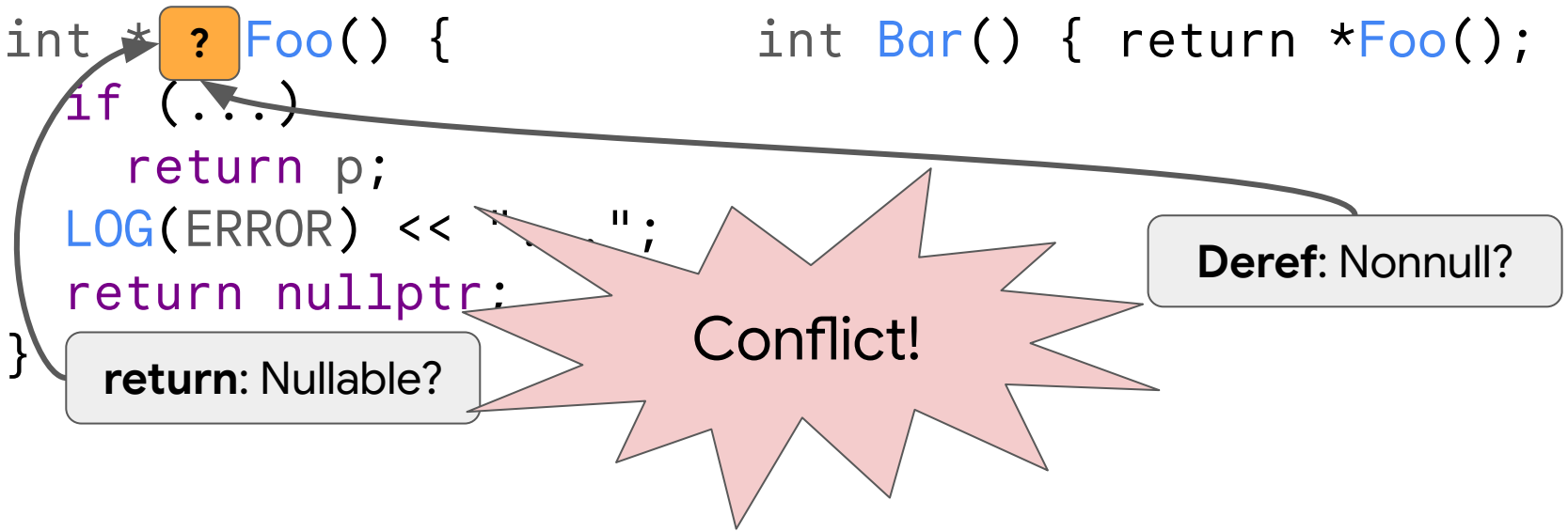
Merging Evidence (body)



Merging Evidence (caller)



Conflicts


int * ? Foo() {
if (...)
return p;
LOG(ERROR) << "...";
return nullptr;
}
int Bar() { return *Foo(); }
Deref: Nonnull?
Conflict!
return: Nullable?

Complex contracts

```
C *?ServerContext() {
  if (...)
    return c_;
  LOG(ERROR) << "...";
  return nullptr;
}
```

```
int HandleRequest() {
  Use(*ServerContext());
}
```

make Nonnull and LOG(FATAL) instead?

Proprietary + Confidential

Agenda

01 Intro

02 Checker (function-scope)

03 Inference (cross-function, cross-TU)

04 Experience

Results

So far:

- Checker deployed for code-review since 2024
- Inference scales to most of monorepo (w/ parallel processing)
- Non-conflicting Nullable: >20% of the way
- Some organic adoption of Default Nonnull

one major project sees 35% reduction in crashes, by also doing:

- Fixing existing violations (at scale)
- Make code-review check blocking

Test vs non-test evidence

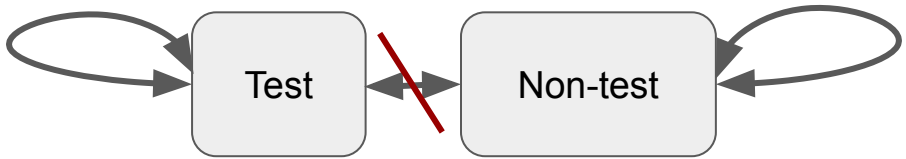
Code: outparam ptr "must" be Nonull. Null checks and otherwise returns Error

Prod: always Nonnull args (need it to work, easy to &var)

Test: wants to test Error

Annotation: Nonnull or Nullable? (what if ref instead of pointer?)

Resolution:



Code that doesn't conform, yet (helpers)

```
// helper to null check, vs check directly
if (hasFoo()) { getFoo()->… }
```

vs

```
if (auto *foo = getFoo(); foo != nullptr) { foo->… }
```

- rewriter?
- or summary attribute?
- similar to "slightly inter-procedural" macros
- **resolution**: conflict avoids annotating when unsure

Code that doesn't conform, yet (late init)

```
struct S {
  T* _Nonnull t = nullptr;  // ??? avoid use of uninit mem?
};
S s;
s.t = Make();
```

Or, 2-stage init (Ctor + Init()), vs Factory

Too lax annotation? Maintaining annotations

```
int *_Nullable Foo() {
  return new int(10);  // Normally Nullable = Nonnull is ok
}
```

- Or _Nullable **param** where **all callers** now only pass in _Nonnull

(inference merging has some information).

Closing thoughts

- Make checker more broadly available
  - currently ClangTidy
  - https://github.com/google/crubit/tree/main/nullability
- Expand features, evolve to be a Clang warning
- Checker is only the beginning
  - need annotations for cross-function cross-TU requirements
  - update developer (and AI?) habits
  - fixing existing bugs (or updating existing code)

Proprietary + Confidential

Thanks and Acknowledgements

Toward null safety:
Annotations → Checker → Inference → Deployment Experience

Samira Bazuzi Bakon

Martin Brænne

Dmytro Hyrbenko

Yitzhak Mandelbaum

Sam McCall

Rohan Jacob-Rao

Venkatesh Srinivasan

Kinuko Yasuda

Wei Yi Tee

Dani Moura

Sam Estep

(and more!)