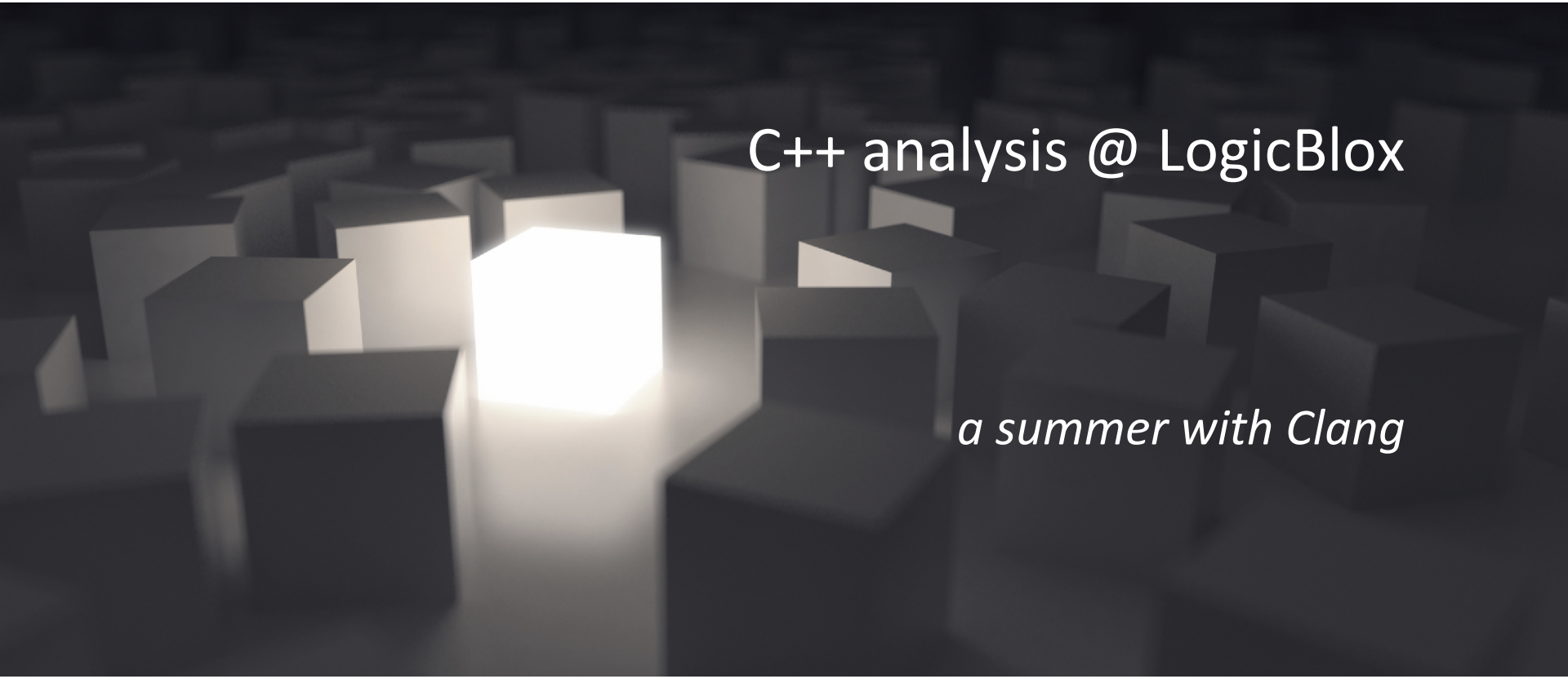


Presentation Overview

- Summer goals
- What is clang and LLVM
- Clang and LogiQL
- LB use case
- Implementation and technical details
- Evaluation - Conclusion

Summer Goals

- **Implement a C++ analysis framework**
- Evaluate available tools, and/or build our own
- **In-house usage on LB's codebase**
- Express LB-specific checks (code guidelines)
- Provide it as an open-source tool

Applications

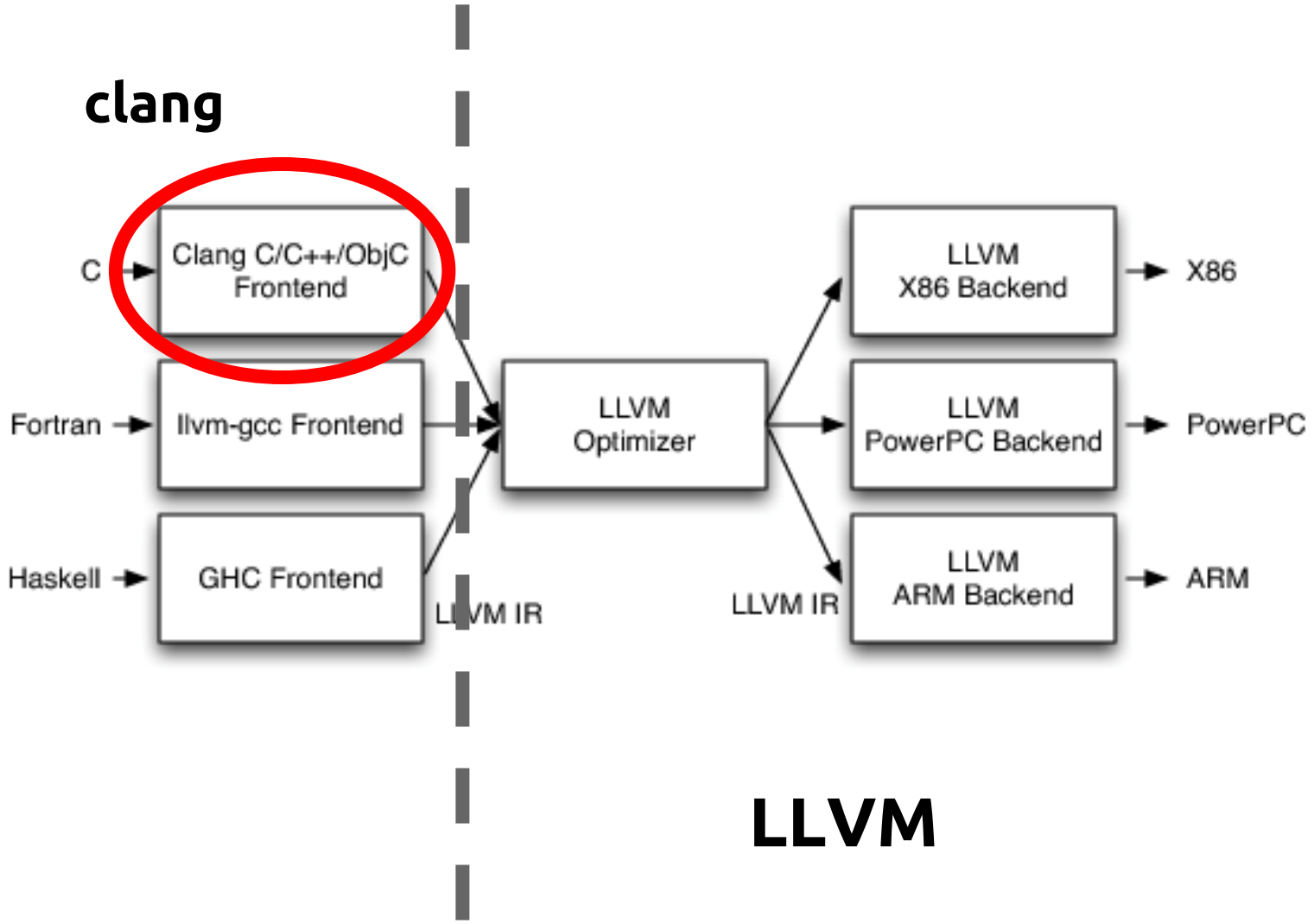
A framework to express...

- code constraints (**guidelines**)
- **bug finding** logic
- sophisticated source code **analyses**
- **code visualization**
- developer **assistance**

What is clang and LLVM

Wait, are they not the same thing?

Clang/LLVM Architecture



Clang 101

What?

A compiler **front end**...

from **source** code to an **intermediate** representation

for...

C, **C++**, Objective-C, Objective-C++

from...

University of Illinois, Apple, Google and others

Why?

- Generates a **detailed AST** from source
- Allows for **plugins** with full AST access
- Supports **every feature** of C/C++
- … even C++0x/C++11
- **Template** declarations & instantiations
- Not many good alternatives

How?

We use **Clang Plugins**...

- Run **extra user defined** actions during compilation
- Can **navigate** every part of the AST
- **Dynamically** linked with clang during compilation

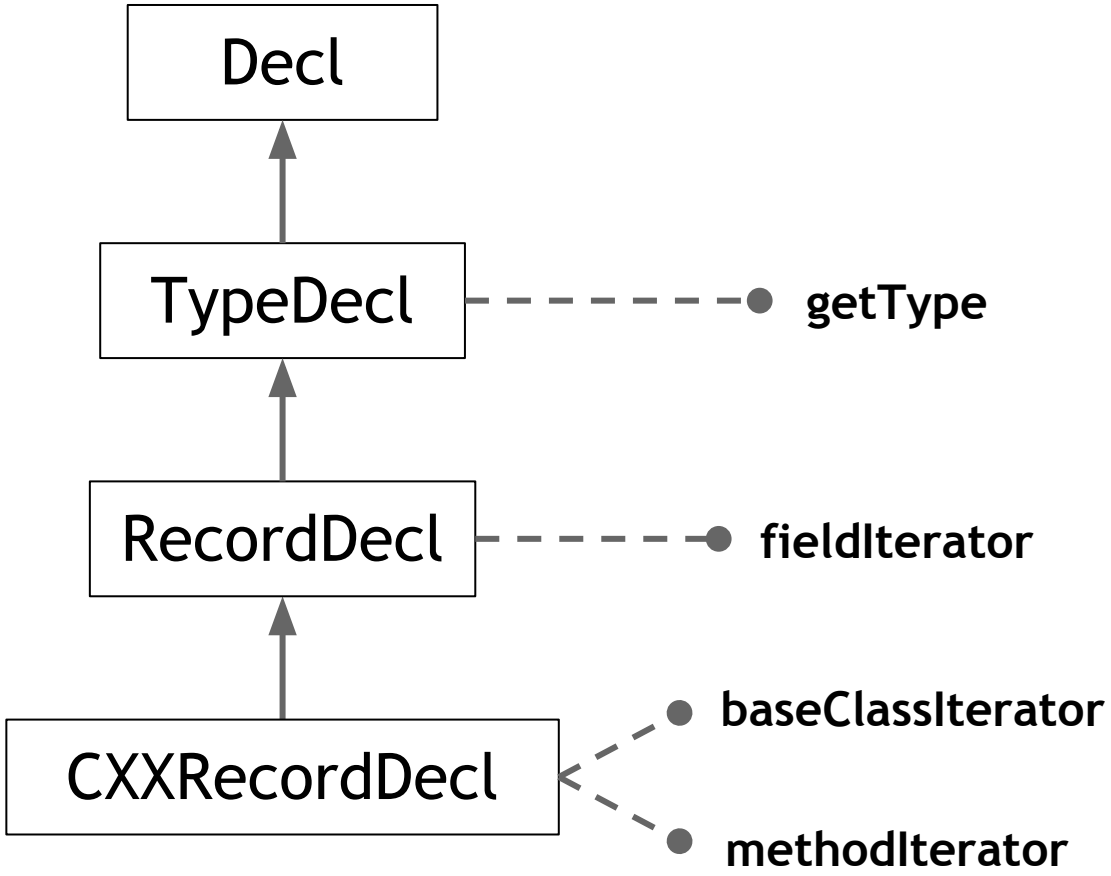
Clang AST

- **Rich** AST representation of source code
- Fully **type resolved** (when possible)
- > 100k LoC
- Three **core classes**
  - Declarations, Types and Statements/Expressions
  - with many many **specializations**
- **Glue** classes & methods

Clang AST

- **Declarations**
  - CXXRecord, Function, Var, …
- **Types**
  - Pointer, Array, TemplateSpecialization, …
- **Statements / Expressions**
  - ForLoop, BinaryOperator, Cast, …

Clang AST


Decl
TypeDecl
getType
RecordDecl
fieldIterator
CXXRecordDecl
baseClassIterator
methodIterator

Clang Plugins Revisited

- **clang::RecursiveASTVisitor** class
- **Visit method** for every kind of AST node
- Plugins **extend** the visitor and **override** the visit methods for the nodes of interest
- Visit the **whole hierarchy** of a node
  - e.g., for a CXXRecordDecl also visit RecordDecl, TypeDecl and Decl

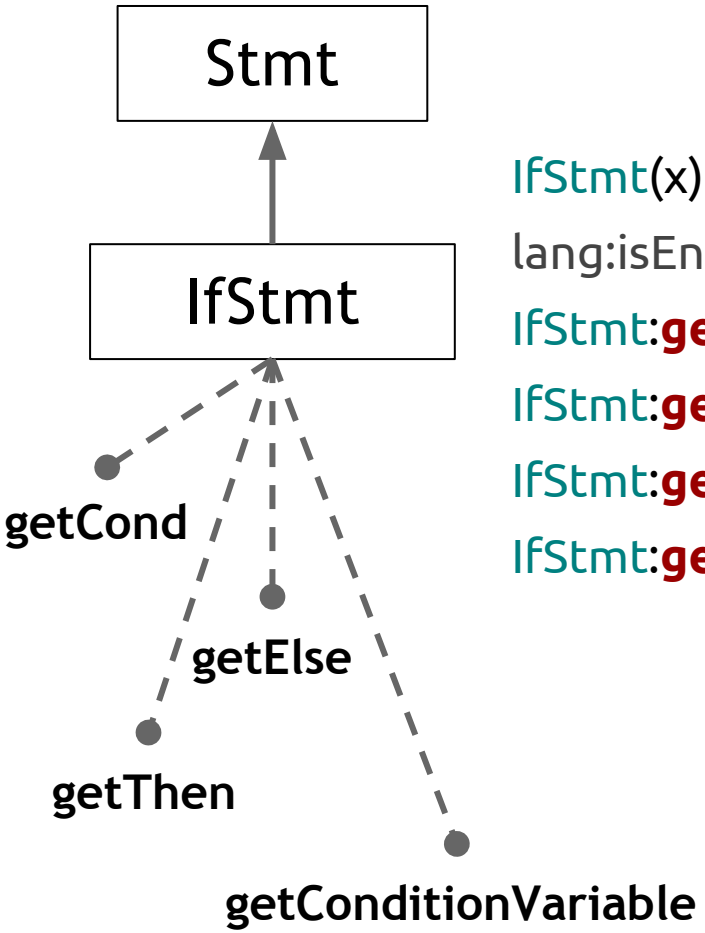
Clang and LogiQL

- Write a **specification** describing the AST
- **Automatically** generate plugin code
  - with **visit methods** for each AST node kind
  - **gather information** from each node
  - and export it in **CSV files**
- **Automatically** generate LogiQL schema
- Import data and **issue queries** in LogiQL

The Specification

- Written in our **own DSL!** (more later)
- Describes…
  - **AST nodes** and their relations (& glue classes)
  - **Properties** for each node and their cardinality
  - **C++ code** for retrieving those properties
- Everything else is **automated!**
- Might be applicable in other setups as well

The Schema




```
IfStmt(x) -> Stmt(x).
lang:isEntity[`IfStmt] = true.
IfStmt:getCondition[n] = v -> IfStmt(n), Expr(v).
IfStmt:getElse[n] = v -> IfStmt(n), Stmt(v).
IfStmt:getThen[n] = v -> IfStmt(n), Stmt(v).
IfStmt:getConditionVariable[n] = v -> IfStmt(n), VarDecl(v).
```

An in-depth look

LB use case

- We have a quite **large** C++ codebase
- The runtime team has some **code conventions**
- **Manually** validating that all code conforms to these conventions is impossible
- Our tool can **automate** this process

LB use case

Simple example

- Find all the **call sites**

CallExpr(call),

- that call **std::vector::operator[]**

CallExpr:**getCalleeDecl**[call] = callee, CXXMethodDecl(callee),
NamedDecl:**qName**(callee, "std::vector::operator[]"),

- for vectors of **blox::Ptr<blox::Box>**

Expr:**hasType**[call] = callType, QualType:**toString**[callType] =
"blox::Ptr<blox::Box>".

LB use case

Clang's alternative

Matchers are very limited, a lot of checks happen in the callback

```
StatementMatcher BracketCallMatcher = operatorCallExpr
(hasOverloadedOperatorName("[]")).bind("bracketCall");
…
class DeclPrinter : public MatchFinder::MatchCallback {
public :
 void run(const MatchFinder::MatchResult &Result) {
   ASTContext *Context = Result.Context;
   const CXXOperatorCallExpr *CallSite =
         Result.Nodes.getNodeAs<CXXOperatorCallExpr>("bracketCall");
   QualType callType = CallSite->getType();
   if(callType.getAsString(Context->getPrintingPolicy()) ==
      "blox::Ptr<blox::Box>"){ … }
}
```

LB use case

Code Convention Example

class **blox::oodb::Object**

- Data members of its subclasses must be
  - POD (**Plain Old Data**): e.g., int/float/..., structs/unions/fixed-size arrays of these
  - blox::oodb::**Pointer<T>**, where T must also derive from blox::oodb::Object

LB use case

How this translates to LogiQL

- Data members of classes that derive from Object class (and their types)

```
LB:oodbObjectFieldType(f,qType) -> FieldDecl(f),QualType(qType).
LB:oodbObjectFieldType(f,qType) <- FieldDecl(f),
                    FieldDecl:declaringRecord(f,rec),
                    CXXRecordDecl:nonTemplateDefinitionRec(rec),
                    CXXRecordDecl:derivesFrom(rec,"blox::oodb::Object"),
                    ValueDecl:hasType(f,qType).
```

LB use case

How this translates to LogiQL

- Plain Old Data data members

```
LB:primitiveType(t) -> Type(t).
LB:primitiveType(t) <- BuiltinType(t);
                       LB:validArray(t);
                       EnumType(t);
                       (RecordType(t),
                        RecordType:hasDecl(t,d),
                        !LB:invalidRecord(d)).

LB:validArray(t) <- ConstantArrayType(t),
  ArrayType:hasElementType(t,qualType),
  QualType:equivelantUnqualifiedType(qualType,unQualType),
  LB:primitiveType(unQualType).
```

LB use case

How this translates to LogiQL

- Structs that are **not** POD

```
LB:invalidRecord(rec) -> RecordDecl(rec).
LB:invalidRecord(rec) <- RecordDecl:hasField(rec,fld),
        ValueDecl:hasType(fld,qType),
        QualType:equivelantUnqualifiedType(qType,unQualTp),
        !LB:primitiveType(unQualTp).
```

LB use case

How this translates to LogiQL

- Fields that are “primitive”

```
LB:isValidField(f) -> FieldDecl(f).
LB:isValidField(f) <- LB:oodbObjectFieldType(f,qType),
                      QualType:equivelantUnqualifiedType(qType,unQualTp),
                      LB:primitiveType(unQualTp).
```

LB use case

How this translates to LogiQL

- Fields of type blox::oodb::Pointer<T>

```
LB:isValidField(f) <- LB:oodbObjectFieldType(f,qType),
        QualType:equivelantUnqualifiedType(qType,unQualTp),
        TemplateSpecializationType(unQualType),
        TemplateSpecializationType:hasDeclN(unQualType,
                                            "blox::oodb::Pointer"),
        TemplateSpecializationType:hasArgument(unQualType,0,arg),
        TemplateArgument:hasType(arg,argQTp),
        QualType:equivelantUnqualifiedType(argQTp,argsType),
        RecordType(argsType),
        RecordType:hasDeclDerives(argsType,"blox::oodb::Object").
```

LB use case

And the query

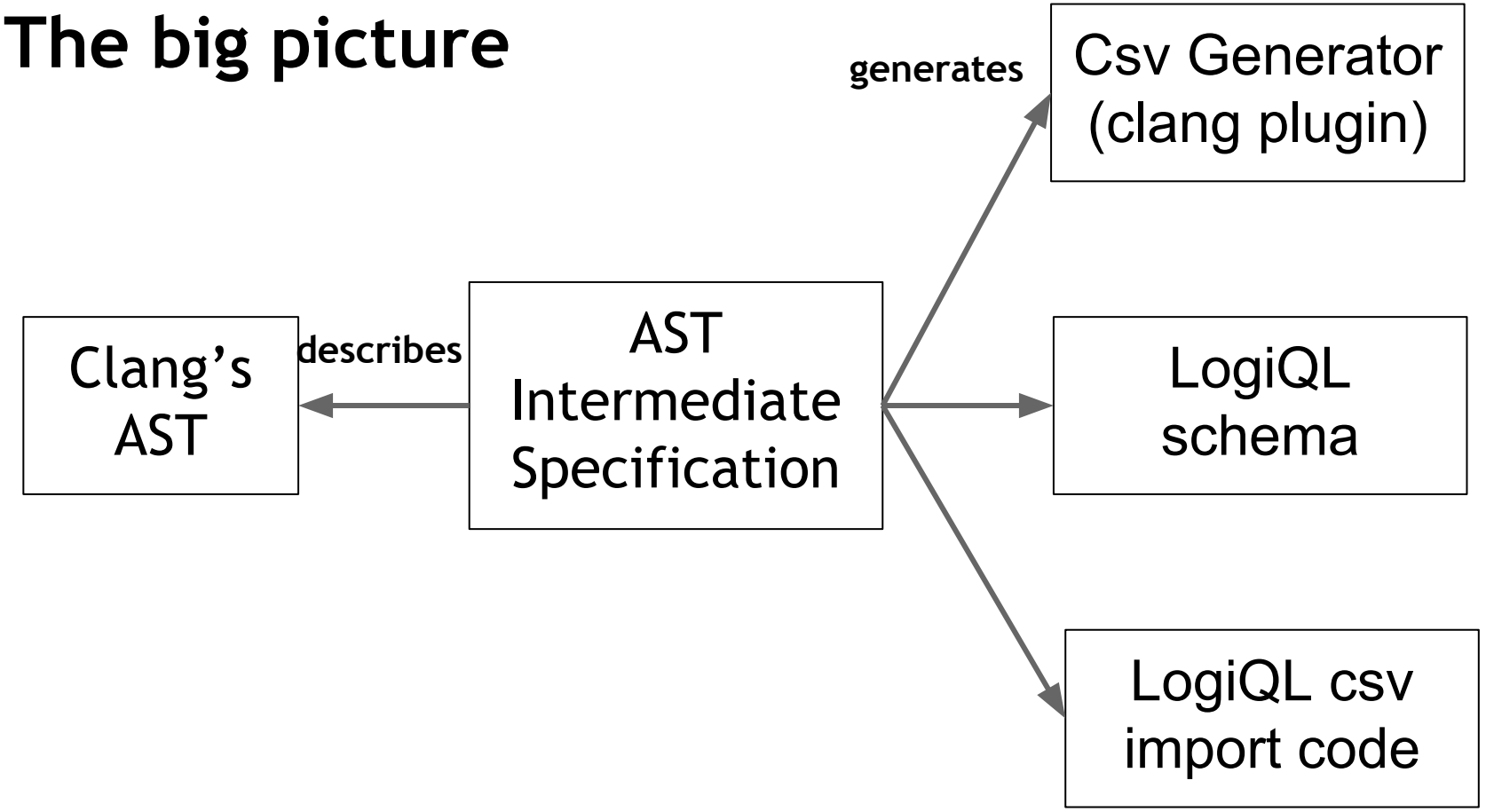
lb exec rt ‘ _(f) <- LB:**oodbObjectFieldType**(f,_),!LB:**isValidField**(f).’

Implementation

- As we saw earlier clang's AST is enormous
- Also clang is still under development. So, its API changes a lot
  - This affects both the CSV generation and the LogicQL schema
- For the above reasons, we decided to create an intermediate language that automates things

Implementation

The big picture



Implementation

Intermediate language code snippet

something like 'this'

inheritance relation between the AST nodes

```
@node(name = S, ...)
IfStmt extends Stmt [

    @property(type = many-to-one)
    getCondition(Expr e) [
       $e = "S->getCond()"
    ]

    @property(type = many-to-one)
    getThen(Stmt s) [
       $s = "S->getThen()"
    ]
    . . .
]
```

C++ expression to obtain the property

Implementation

Our intermediate specification in a nutshell

- Has a very simple type system to describe
  - **primitive types** (e.g., int, string, etc.)
  - **AST nodes** (e.g., Decl, Stmt, etc.)
  - **Other entities** used by the AST to describe complex concepts (e.g., FileEntry,BaseClass, etc.)
- We are also able to **annotate** various elements of the specification to make the generation easier
  - e.g., we might need the C++ type of an entity in order to produce valid C++ code

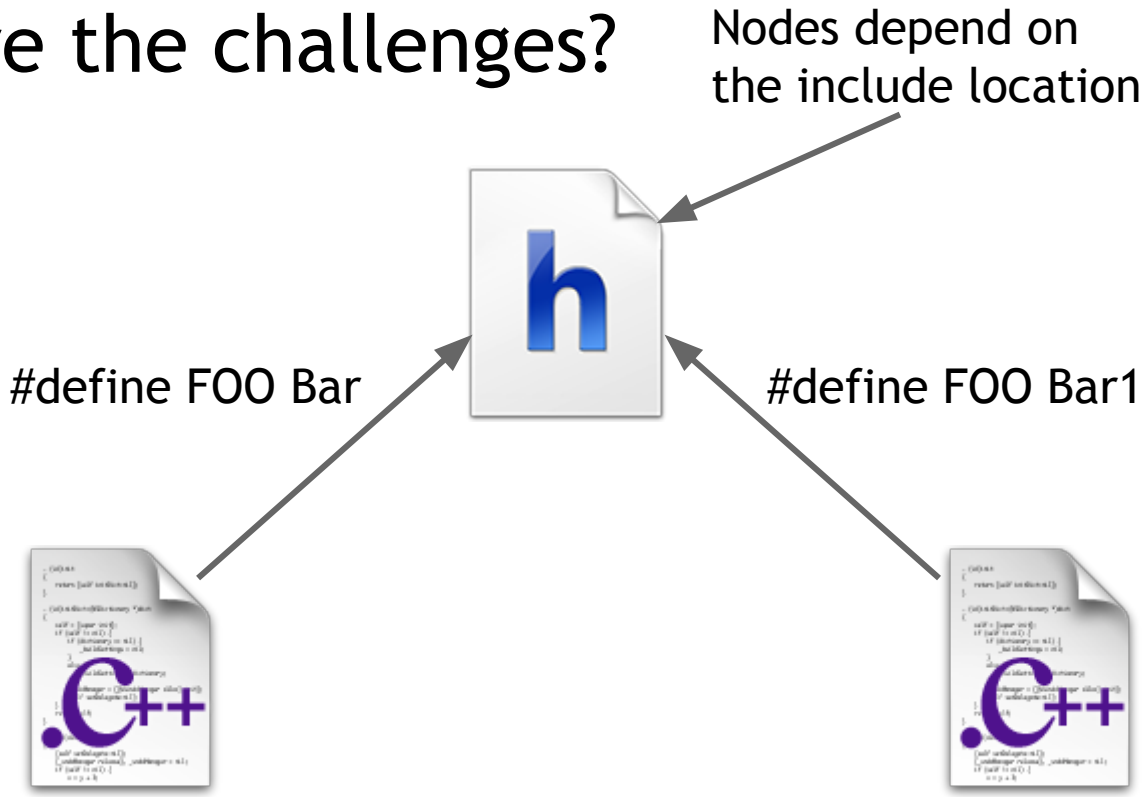
Some technical details

Analyzing C++ source is challenging

- We are interested in **whole program** analysis
  - For example, if a node is the same in two different translation units, we want to have one entity in LogiQL
- C++ has a huge variety of **features**
  - And more to come (C++14)
- There are also some features **inherited by C** that complicate the analysis (e.g., macros)

Some technical details

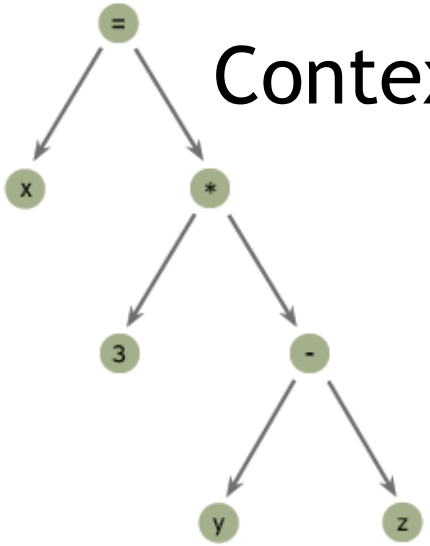
What are the challenges?



Some technical details

What are the challenges?

- C++ needs a lot of semantic context to be parsed
  - Comparing just the structure of the AST is not sufficient.



Context?

Evaluation and Future work

- For now our tool is in a pretty good state
  - CSV generation does not add a lot of overhead in the building process
  - We can analyze a large C++ codebase in a reasonable amount of time
- **But** we are not done yet
  - Bug fixes (CSV generation, identifying nodes, ...)
  - Provide useful **abstractions** (e.g., extra IDB rules) in order to make our framework usable and extensible (our schema is very “clang” oriented)

Conclusion

- Goals for our program analysis framework
- What is clang and LLVM
- From clang to LogiQL
- Runtime use case
- Implementation and challenges
- Our framework publicly available on bitbucket

Last but not least

Thanks

Thanks everybody for a great summer :D

Hope you enjoyed :)

Questions?