The Magic Behind Optimizing Compilers: Static Program Analysis

...

17th – 19th November, 2022

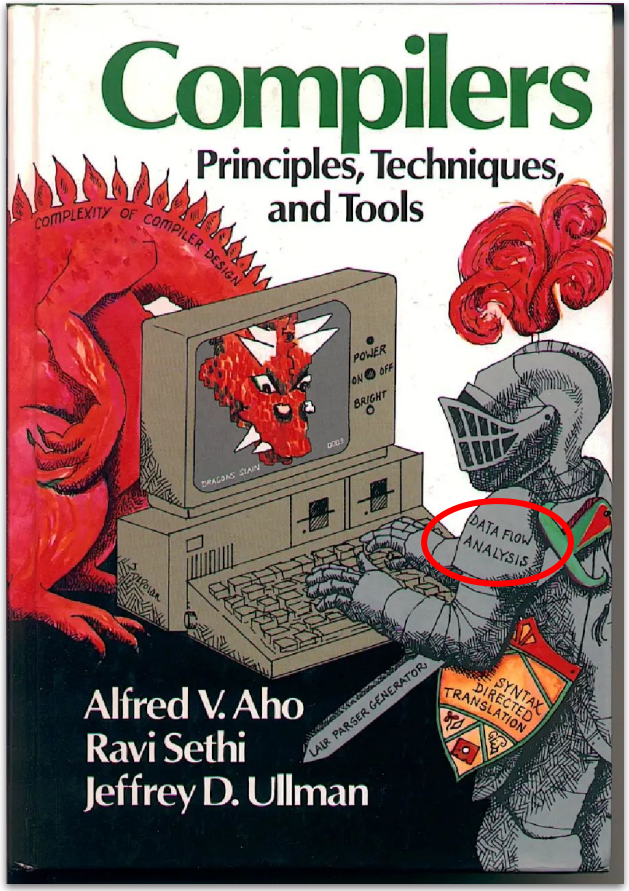
Philipp Dominik Schubert
philipp@gazar.eu

@pd_schubert

PADERBORN UNIVERSITY

Why Am I Here?

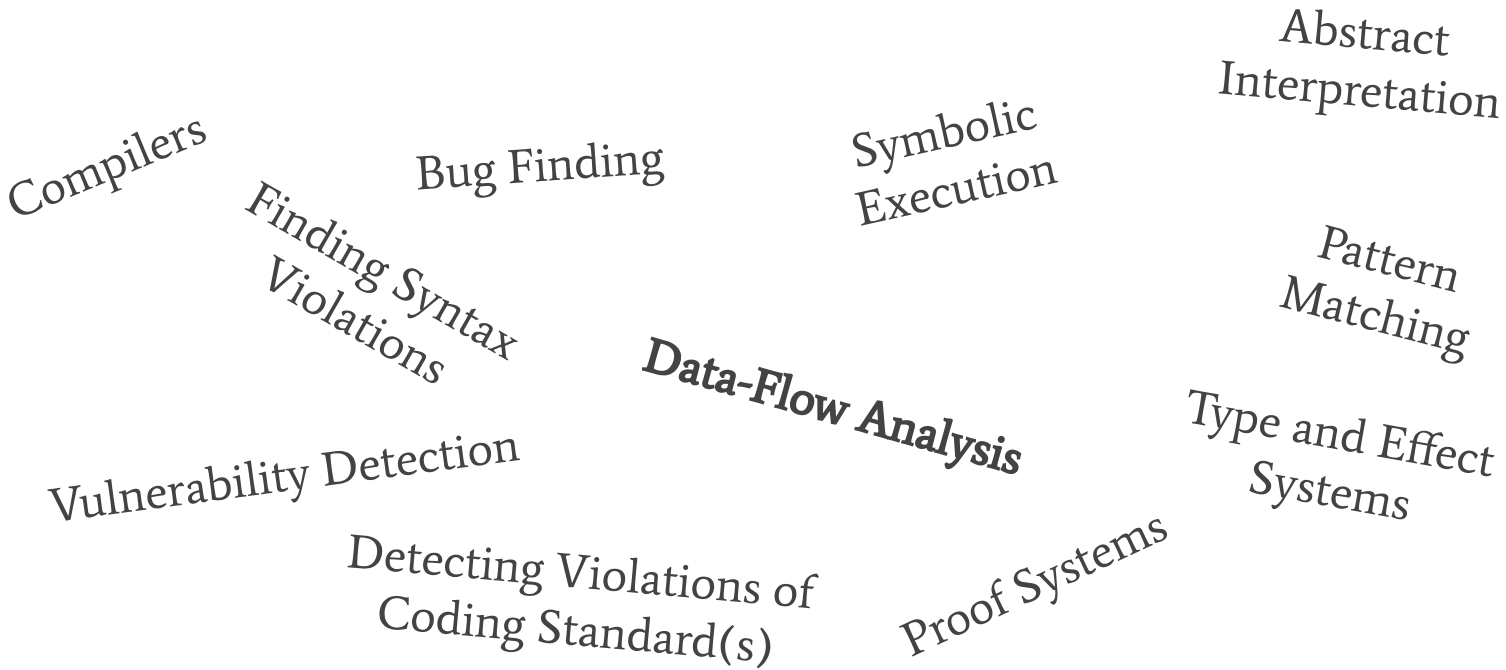
Compilers and Static Analysis



Static Data-Flow Analysis Is *Only a Small Part of One Chapter*

Why Static Analysis?

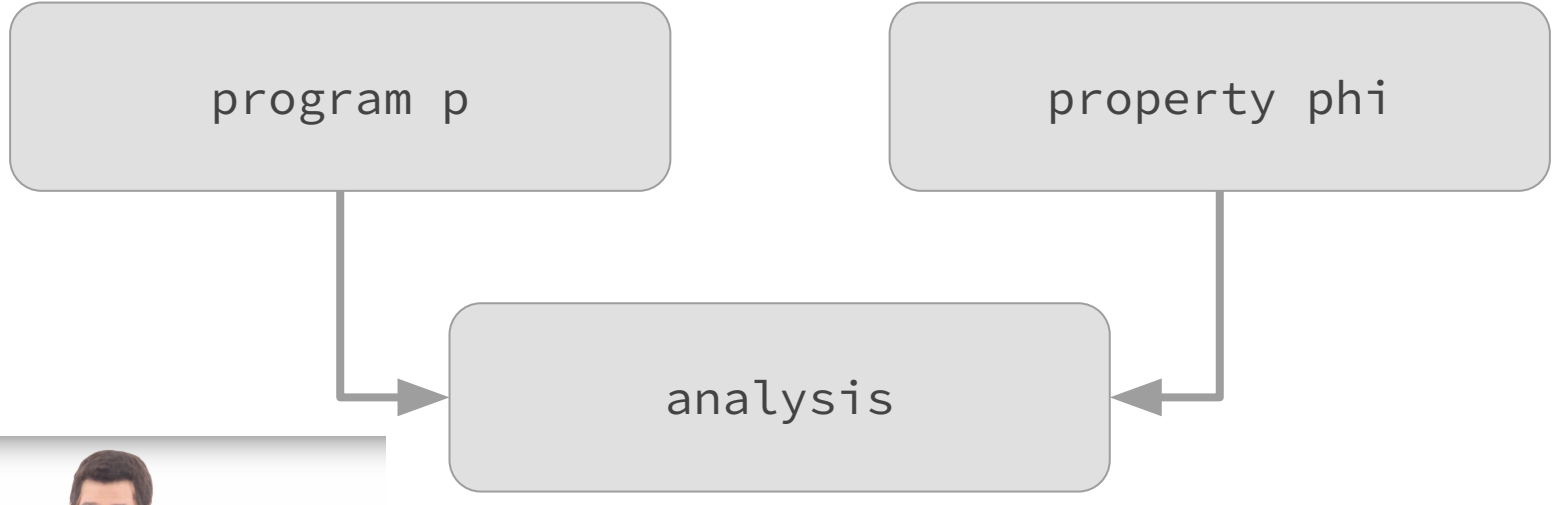
Applications and Techniques



"~~*Water's*~~ *Static analysis is useful for a lot of things.*", ~~Richard Hammond~~

But What Does It Do?

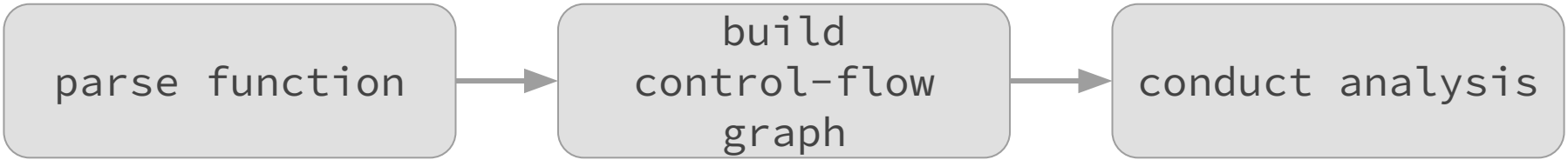
Static Analysis and Program Properties



Does property phi hold at statement s?

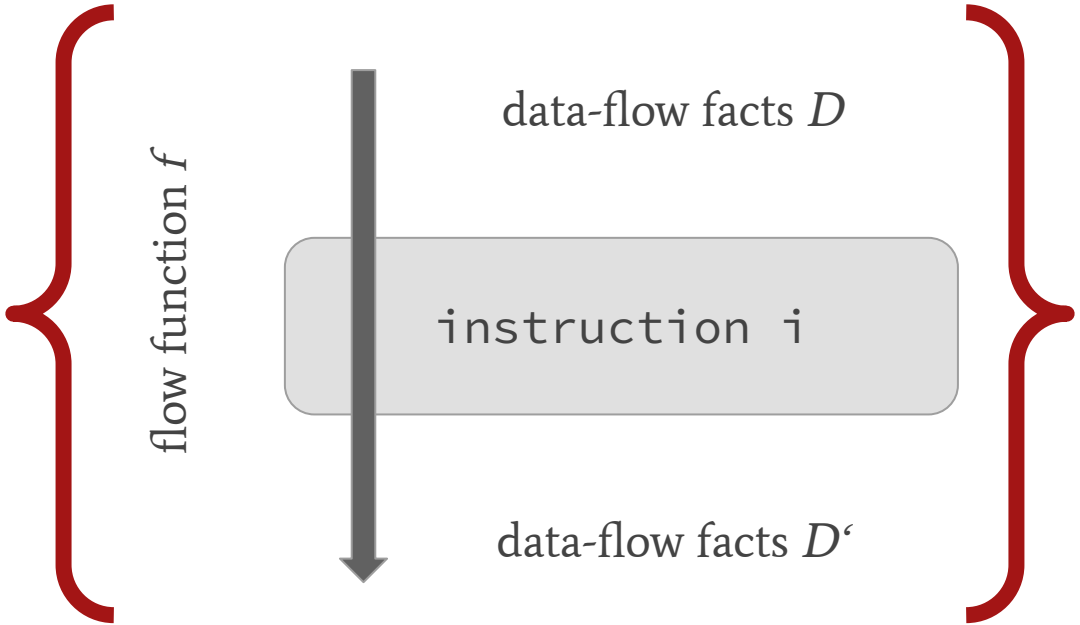
Static Data-Flow Analysis: The Recipe (for Bug Detection)

- Find and understand a bug
- Write analysis that finds bug
- Run analysis
- Check findings
  - If it's bug/vuln, then fix!
  - Otherwise, improve analysis



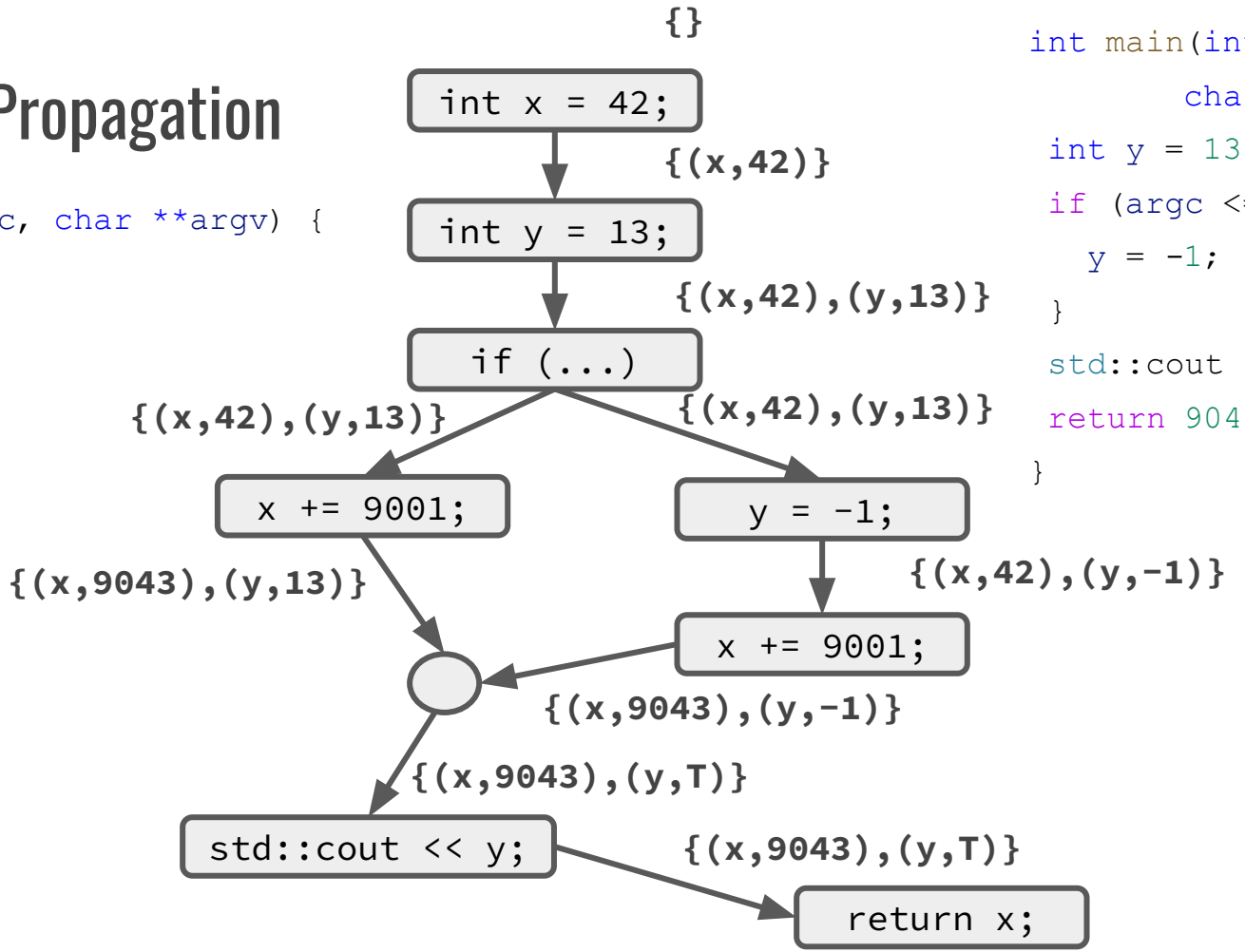
Data-Flow Analysis 101: Monotone Framework

Static Data-Flow Analysis



Constant Propagation

```
int main(int argc, char **argv) {
 int x = 42;
 int y = 13;
 if (argc > 1) {
   x += 9001;
 } else {
   y = -1;
   x += 9001;
 }
 std::cout << y;
 return x;
}
```




```
int main(int argc,
         char **argv) {
 int y = 13;
 if (argc <= 1) {
   y = -1;
 }
 std::cout << y;
 return 9043;
}
```

How Does It Look In LLVM?

LLVM's Intermediate Representation

```
define i32 @main(i32 %argc, i8** %argv) #4 {
entry:
 %retval = alloca i32, align 4
 %argc.addr = alloca i32, align 4
 %argv.addr = alloca i8**, align 8
 %x = alloca i32, align 4
 %y = alloca i32, align 4
 store i32 0, i32* %retval, align 4
 store i32 %argc, i32* %argc.addr, align 4
 store i8** %argv, i8*** %argv.addr, align 8
 store i32 42, i32* %x, align 4
 store i32 13, i32* %y, align 4
 %0 = load i32, i32* %argc.addr, align 4
 %cmp = icmp sgt i32 %0, 1
 br i1 %cmp, label %if.then, label %if.else
```

```
$ clang++ -emit-llvm -S -fno-discard-value-names main.cpp
```

Or use Compiler Explorer at https://godbolt.org/

```
if.then:
; preds = %entry
 %1 = load i32, i32* %x, align 4
 %add = add nsw i32 %1, 9001
 store i32 %add, i32* %x, align 4
 br label %if.end

if.else:
; preds = %entry
 store i32 -1, i32* %y, align 4
 %2 = load i32, i32* %x, align 4
 %add1 = add nsw i32 %2, 9001
 store i32 %add1, i32* %x, align 4
 br label %if.end

if.end:
; preds = %if.else, %if.then
 %3 = load i32, i32* %y, align 4
 %call = call [...]
@_ZNSolsEi(%"class.std::basic_ostream"* [...]
          @_ZSt4cout, i32 [...] %3)
 %4 = load i32, i32* %x, align 4
 ret i32 %4
```

LLVM's API for IR Inspection and Its Many Hierarchies

- `llvm::LLVMContext`
- `llvm::Module`
- `llvm::Function`
- `llvm::BasicBlock`
- `llvm::Instruction`
- `llvm::Operator`
- `llvm::Value`

`[...]`

- Custom (closed) RTTI system

```
llvm::dyn_cast<TargetTy>(Ptr)
llvm::isa<TargetTy>(Ptr)
```

- Allows for high-level interfaces

```
template <typename D>
D flowThrough(llvm::Instruction *I, D InSet);
```

- Details where necessary
- Eases code inspection, transformation and generation!

Nothing Stops You From (Automatically) Inspecting Your Code

```
int main(int Argc, char **Argv) {
 if (Argc != 2) { return 1; }
 llvm::SMDiagnostic Diag;
 llvm::LLVMContext Ctx;
 std::unique_ptr<llvm::Module> M =
   llvm::parseIRFile(Argv[1], Diag, Ctx);
 bool BrokenDbgInfo = false;
 if (llvm::verifyModule(*M,
                        &llvm::errs(),
                        &BrokenDbgInfo)) {
   llvm::errs() << "error!\n";
   return 1;
 }
 if (BrokenDbgInfo) {
   llvm::errs() << "broken dbg info!\n";
 }
```

```
 for (const auto &F : *M) {
   llvm::outs() << "found function " << F.getName() << '\n';
   for (const auto &BB : F) {
     for (const auto &I : BB) {
       if (const auto *Alloca = llvm::dyn_cast<llvm::AllocaInst>(&I)) {
         llvm::outs() << "found alloca inst: " << *Alloca << '\n';
       }
       if (const auto *CallSite = llvm::dyn_cast<llvm::CallBase>(&I)) {
         if (const auto *Callee = CallSite->getCalledFunction()) {
           llvm::outs() << "found callee: " << Callee->getName() << '\n';
         } else {
           llvm::outs() << "found indirect callsite!\n";
         }
       }
     }
   }
 }
 return 0; }
```

Too Good to Be True?

Sadly Yes, See Magical Initialization

```
int main() {
 int i;
 int j = i + 13;
 return j;
}
```

```
void magicInit(int *) {}

int main() {
 int i;
 magicInit(&i);
 int j = i + 13;
 return j;
}
```

A Realistic Analysis Requires More Information

- Data-flow analysis requires
  - Type hierarchy analysis
  - Points-to analysis
  - Callgraph analysis
  - Other data-flow analyses

```
struct Super {
 virtual ~Super() = default;
 virtual void f() {}
};

struct Duper : Super {
 ~Duper() override = default;
 void f() override {}
};

int main() {
 Super *S = new Duper;
 S->f();
 delete S;
 return 0;
}
```

State-Of-The-Art Data-Flow Analysis

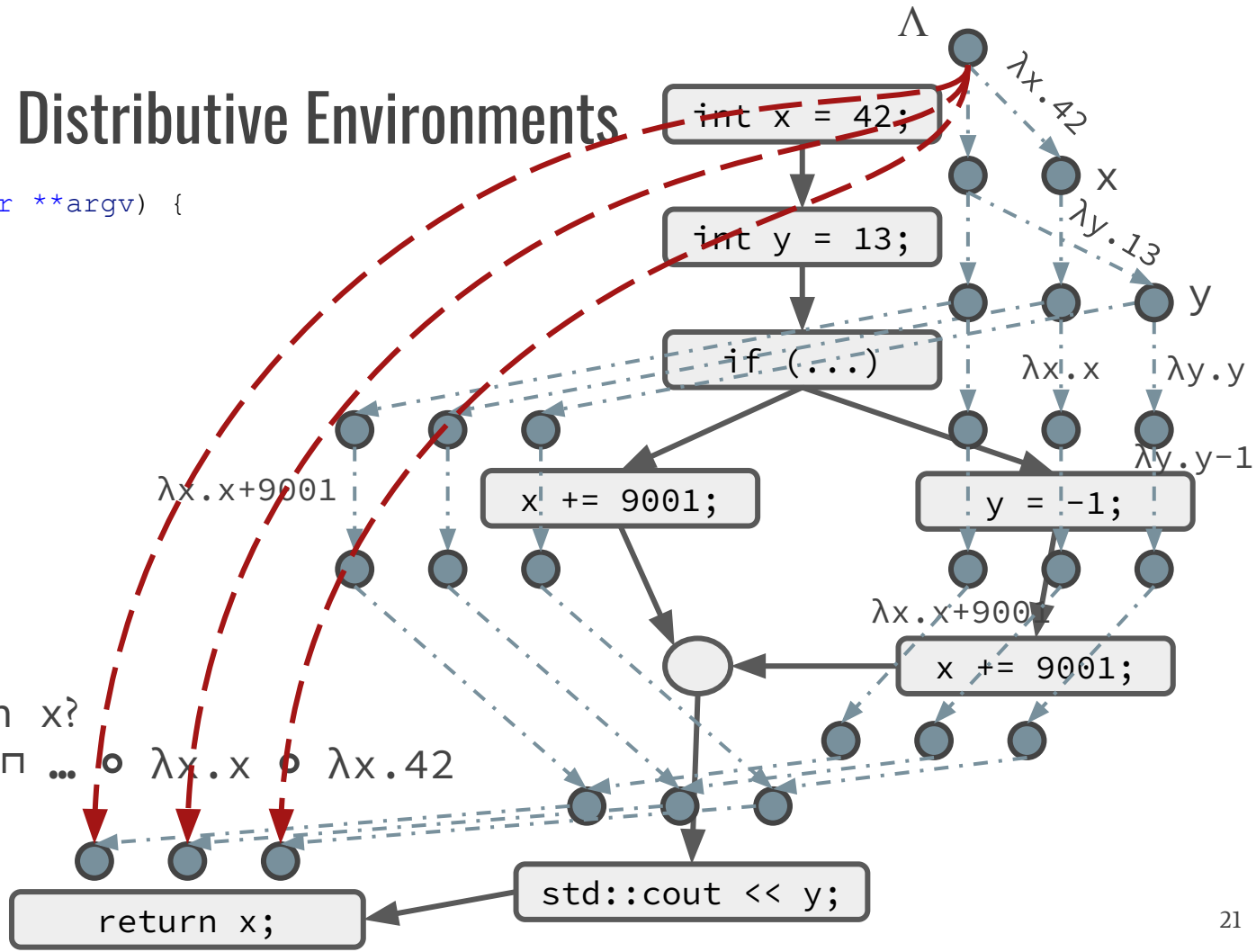
Inter-Procedural Distributive Environments

```
int main(int argc, char **argv) {
 int x = 42;
 int y = 13;
 if (argc > 1) {
   x += 9001;
 } else {
   y = -1;
   x += 9001;
 }
 std::cout << y;
 return x;
}
```

X's value at `return x`?

$\lambda x.x+9001 \circ \ldots \sqcap \ldots \circ \lambda x.x \circ \lambda x.42$

$= 9043$



Pros and Cons of the IDE Algorithm

- Flow sensitive
- Inter-procedural
- Infinite context sensitivity by design
- Fast

- Complex
- Only works for distributive analysis problems
  - Pointer analysis is not distributive
- Exploded super-graph gets very large

Compiler People vs. Static Analysis People

—

Optimization vs. Bug Finding

Different Objectives

- Compiler optimization
  - Speed
  - Soundness
  - Correctness

- Bug/vuln finding

Analysis Challenges

The Eternal Fight: Precise Points-to Information

- Points-to analysis is not distributive
- Will be hugely expensive in context-sensitive, inter-procedural setup
- Important sensitivities for pointer analysis
  - context sensitivity
  - flow sensitivity
- Analysis developers are constantly working around undecidable problems!
- `restrict` doesn't look so bad all of a sudden

```
int foo(int *a, int *b) {
   *a = 5;
   *b = 6;
   return *a + *b;
}

int rfoo(int *restrict a,
         int *restrict b) {
   *a = 5;
   *b = 6;
   return *a + *b;
}
```

https://en.cppreference.com/w/c/language/restrict

Further Challenges

- Scalability of deep, semantic whole-program analysis
- Analysis precision – *noise* to *useful information* ratio
- Complexity
  - This is hard in theory
  - It's even harder in practice
- We started a static analysis project
  - It started when I was still learning C++

Writing an LLVM-based Analysis

Analysis Passes in LLVM

```
class CallSiteFinderAnalysis
    : public
llvm::AnalysisInfoMixin<CallSiteFinderAnalysis> {
public:
 explicit CallSiteFinderAnalysis() = default;
 ~CallSiteFinderAnalysis() = default;
 // Provide a unique key, i.e., memory address to
 // be used by the LLVM's pass infrastructure.
 static inline llvm::AnalysisKey Key; // NOLINT
 friend
llvm::AnalysisInfoMixin<CallSiteFinderAnalysis>;

 // Specify the result type of this analysis pass.
 using Result = llvm::SetVector<llvm::CallBase *>;
```

```
// Analyze the bitcode/IR in the given LLVM module.
static Result run(llvm::Module &M,
                  llvm::ModuleAnalysisManager &MAM) {
  const static llvm::StringRef TargetFunName = "foo()";
  Result TargetCallSites;
  for (auto &F : M) {
    for (auto &BB : F) {
      for (auto &I : BB) {
        if (auto *CB = llvm::dyn_cast<llvm::CallBase>(&I)) {
          // Only find direct function calls.
          if (!CB->isIndirectCall() && CB->getCalledFunction()
              && llvm::demangle(
                 CB->getCalledFunction()->getName().str()) ==
                       TargetFunName) {
            llvm::outs() << "found a direct call!\n";
            TargetCallSites.insert(CB);
          }
        }
      }
    }
  }
  return TargetCallSites;
 }
};
```

Writing an LLVM-based Optimization

Transformation Passes in LLVM

```
class CallSiteReplacer : public
llvm::PassInfoMixin<CallSiteReplacer> {
public:
 explicit CallSiteReplacer() = default;
 ~CallSiteReplacer() = default;
 // Transform the IR in the given LLVM module.
 static llvm::PreservedAnalyses run(llvm::Module &M,
                        llvm::ModuleAnalysisManager &MAM) {
   // Request results of CallSiteFinderAnalysis analysis.
   auto TargetCallSites =
     MAM.getResult<CallSiteFinderAnalysis>(M);
   // Name of the replacement function.
   const static llvm::StringRef ReplacementFunName = "_Z3bari";
   auto *ReplacementFun = M.getFunction(ReplacementFunName);
   static unsigned ReplacementCounter = 1;
   llvm::outs() << "applying code transformation..\n";
```

```
for (auto *TargetCallSite : TargetCallSites) {
   // Create an int from our replacement counter.
   auto *ConstInt = llvm::ConstantInt::get(
     llvm::IntegerType::get(M.getContext(),
                            32 /* bits */),
                             ReplacementCounter);
   // Construct the new call site.
   auto *NewCallSite = llvm::CallInst::Create(
     llvm::FunctionCallee(ReplacementFun),
                           {ConstInt});
   // Replace target call site.
   llvm::ReplaceInstWithInst(TargetCallSite,
                             NewCallSite);
    ++ReplacementCounter;
  }
  // Sorry, we are just lazy.
  return llvm::PreservedAnalyses::none();
 }
};
```

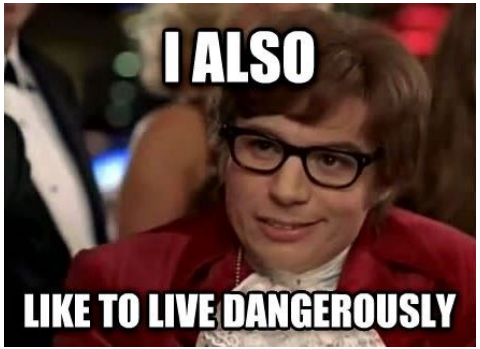
Putting the Pieces Together

```
int main(int Argc, char **Argv) {
 if (Argc != 2) { return 1; }
 // Parse an LLVM IR file.
 llvm::SMDiagnostic Diag;
 llvm::LLVMContext CTX;
 std::unique_ptr<llvm::Module> M =
     llvm::parseIRFile(Argv[1], Diag, CTX);
 bool BrokenDbgInfo = false;
 if (llvm::verifyModule(*M, &llvm::errs(),
                        &BrokenDbgInfo)) {
   llvm::errs() << "error: invalid module\n";
   return 1;
 }
 if (BrokenDbgInfo) {
   llvm::errs() << "broken dbg info\n";
 }
 llvm::PassBuilder PB;
 llvm::ModuleAnalysisManager MAM;
 llvm::ModulePassManager MPM;
 CallSiteFinderAnalysis CSF;
 // Register our analysis pass.
 MAM.registerPass([&]() { return CSF; });
 PB.registerModuleAnalyses(MAM);
 // Add our transformation pass.
 MPM.addPass(CallSiteReplacer());
 // Just to be sure.
 MPM.addPass(llvm::VerifierPass());
 // Run our transformation pass.
 MPM.run(*M, MAM);
 llvm::outs() << "The transformed program:\n"
              << "-----------------------\n";
 llvm::outs() << *M;
 return 0;
}
```

Do Developers Know Better Than the Compiler

Clang 15 with `-std=c++17 -O2`

restrict/__restrict__



- Lack of precise points-to/alias information

```
int foo(int *a, int *b) {
 *a = 5;
 *b = 6;
 return *a + *b;
}
int rfoo(int *__restrict__ a,
         int *__restrict__ b) {
 *a = 5;
 *b = 6;
 return *a + *b;
}
```

```
foo(int*, int*): # @foo(int*, int*)
 mov dword ptr [rdi], 5
 mov dword ptr [rsi], 6
 mov eax, dword ptr [rdi]
 add eax, 6
 ret
rfoo(int*, int*): # @rfoo(int*, int*)
 mov dword ptr [rdi], 5
 mov dword ptr [rsi], 6
 mov eax, 11
 ret
```

https://en.cppreference.com/w/c/language/restrict

Clang 15 with `-std=c++17 -O2`

constexpr

```
int factorial(int N) {
 int Ans = 1;
 for (int Val = 1; Val <= N; ++Val) {
   Ans *= Val;
 }
 return Ans;
}
int main(int Argc, char **Argv) { return factorial(5); }
```

```
define dso_local i32 @_Z9factorialiI(i32 %N) #0 {
[...]
}
define dso_local i32 @main(i32 %Argc,
                           i8** %Argv) #0 {
entry:
 ret i32 120
}
```

```
constexpr int factorial(int N) {
 int Ans = 1;
 for (int Val = 1; Val <= N; ++Val) {
   Ans *= Val;
 }
 return Ans;
}
int main(int Argc, char **Argv) { return factorial(5); }
```

```
define dso_local i32 @main(i32 %Argc,
                           i8** %Argv) #0 {
entry:
 ret i32 120
}
```

- Removes factorial's definition

Clang 15 with `-std=c++17 -O2`

noexcept

```
#include "MyFunctions.h"
int main() {
 try {
   foo();
   bar();
 } catch (...) {/* ¯\_(ツ)_/¯ */}
 return 0; }
```

```
#ifndef MYFUNCTIONS_H
#define MYFUNCTIONS_H
void foo();
void bar();
#endif
```

```
define i32 @main() personality i8* bitcast
  (i32 (...)* @__gxx_personality_v0 to i8*) {
entry:
 invoke void @_Z3foov()
         to label %invoke.cont unwind label %lpad

invoke.cont:                                      ;
preds = %entry
 invoke void @_Z3barv()
         to label %try.cont unwind label %lpad
lpad: [...]
```

```
#include "MyFunctions.h"
int main() {
 try {
   foo();
   bar();
 } catch (...) {/* ¯\_(ツ)_/¯ */}
 return 0;
}
```

```
#ifndef MYFUNCTIONS_H
#define MYFUNCTIONS_H
void foo();
void bar() noexcept;
#endif
```

- When termination is an acceptable response …

```
define i32 @main() personality i8* bitcast
  (i32 (...)* @__gxx_personality_v0 to i8*) {
entry:
 invoke void @_Z3foov()
         to label %invoke.cont unwind label %lpad

invoke.cont:                                      ;
preds = %entry
 tail call void @_Z3barv() #5
 br label %try.cont
lpad: [...]
```

Static Analysis Is Pretty Cool and Compilers Are Awesome

- Compilers rely on static analysis information
- Some tasks are statically undecidable

“Any interesting, general program analysis is undecidable.”

- But, we can still compute useful information!
- Know how static analysis works and write code accordingly
  - Procedure boundaries, pointers, global variables, etc.
- Help the compiler to figure it out and

Express Intent!

Clang 15 with `-std=c++17 -O2`

Student Code to Find the Maximum Value

```
int maxNaive(int A, int B, int C, int D)
{
 if (A >= B && A >= C && A >= D) {
   return A;
 } else if (B >= A && B >= C && B >= D) {
   return B;
 } else if (C >= A && C >= B && C >= D) {
   return C;
 } else {
   return D;
 }
}
```

```
maxNaive(int, int, int, int):
 mov eax, edi      ; 28 lines
 cmp edi, esi      ; of assembly
 jl .LBB0_3
 cmp eax, edx
 jl .LBB0_3
 cmp eax, ecx
 jl .LBB0_3
 ret
.LBB0_3:
 cmp esi, eax
 jl .LBB0_7
```

```
int maxSmart(int A, int B, int C, int D) {
 int Max = A;
 if (B > Max) {
   Max = B;
 }
 if (C > Max) {
   Max = C;
 }
 if (D > Max) {
   Max = D;
 }
 return Max;
}
```

```
int maxLazy(int A, int B,
            int C, int D) {
 return std::max({A, B, C,
D});
}
```

```
maxSmart/maxLazy(int, int,
                 int, int):
 mov eax, edi
 cmp esi, edi
 cmovg eax, esi
 cmp eax, edx
 cmovle eax, edx
 cmp eax, ecx
 cmovle eax, ecx
 ret
```

Conclusions

Interested?
Talk to us.
We need your help!

philipp@gazar.eu

- Keep In Mind How Static Analysis Works
- Know Its Limits
- Write Code That Aids Static Analysis and the Compiler
- Don’t Be Naive, but Also Don’t Be Smart

Code Examples

- “Hello, LLVM!”
  - https://github.com/GaZAR-UG/hello-llvm
- LLVM Analysis and Transformation Passes
  - https://github.com/GaZAR-UG/llvm-opt-pass
- PhASAR — A LLVM-based Static Analysis Framework
  - https://github.com/secure-software-engineering/phasar

Inception: Topics Covered in This Talk

- Why static analysis?
- Static analysis as a resource of information
- The basics of static analysis
- The monotone framework
- Implementing a framework on a rainy weekend
- How does it look in practice (in LLVM)?
- Static helper analyses
- Tension between compiler community and program analysis community
- State-of-the-art data-flow analysis
- Optimizations versus bug finding
- Challenges
- Analysis and optimization in LLVM
- Writing your own analysis and/or optimization

Abstract

Optimizing compilers are awesome for sure. But how do they actually get all the information required to perform those mind-boggling optimizations? And how useful are constexpr and noexcept--do developers know better than the compiler?

This talk presents the basics of static program analysis and shows what goes on behind the scenes of optimizing compilers. Starting from well-known static analysis frameworks--that any skilled C++ developer can implement on a rainy weekend, this talk walks you through more and more sophisticated analysis techniques and presents the current challenges that the program analysis and the compiler community face.

While presenting the necessary theory, this talk will put emphasis on presenting hands-on examples. The examples and compiler internals will be drawn from Clang/LLVM. At the end of this talk, you will (i) have a better understanding of what the compiler can and cannot know, allowing you to write more effective code, and (ii) be able to implement your own compiler analysis and optimization passes in LLVM.