Writing Your Own IR Transformation

LLVM Social Berlin: Feb 2023

Yoshitomo Nakanishi

Who am I?

- Software Engineer @Ethereum Foundation
- Working on Fe-lang (https://github.com/ethereum/fe) and Sonatina(https://github.com/fe-lang/sonatina)
- Github: https://github.com/Y-Nak

The Goal Of The Talk

- Learn To Create Dynamic Plugins
- Master Testing Framework
- Get Used To The New Pass Manager
- Learn Basic Concepts Of Data Structures On IR
- Feel Easy To Develop Custom Passes

Install LLVM

```
# For macOs
brew install LLVM@15

# For Ubuntu
curl -fsSL https://apt.llvm.org/llvm-snapshot.gpg.key | gpg --dearmor -o /etc/apt/keyrings/llvm.gpg
echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/llvm.gpg] http://apt.llvm.org/jammy/ llvm-toolchain-jammy-15 main" |
tee /etc/apt/sources.list.d/llvm.list > /dev/nul
apt update
aup install
```

LLVM Pass Manager

> A pass manager schedules transformation passes and analyses to be run on IR in a specific order. Passes can run on an entire module, a single function ...
> A pass manager is also responsible for managing analysis results.
> ...
> The pass manager must cache results and recompute them when they are invalidated by transforms.
>
> https://blog.llvm.org/posts/2021-03-26-the-new-pass-manager/

First Custom Transformation Pass

https://github.com/Y-Nak/write-your-own-pass/commit/8e501a52be31f502455e38ae3d2d29cef9f3e3ba

First Custom Transformation Pass

```
// include/FuncInfoAnalysis.h
#ifndef CUSTOM_IR_FUNC_INFO_ANALYSIS_H
#define CUSTOM_IR_FUNC_INFO_ANALYSIS_H
#include "llvm/IR/PassManager.h"
using namespace llvm;
/// This pass analyze the basic information of the function.
/// Then emit the result to stderr.
struct FuncInfoAnalysis : PassInfoMixin<FuncInfoAnalysis>
{
    PreservedAnalyses run(Function &F, FunctionAnalysisManager &);
    /// `isRequired` should be true in the dev process to
    /// ignore `optnone` attribute.
    static bool isRequired() { return true; }
};

#endif
```

First Custom Transformation Pass

```cpp
// lib/FuncInfoAnalysis.cpp
#include "FuncInfoAnalysis.h"
#include "llvm/Passes/PassBuilder.h"
#include "llvm/Passes/PassPlugin.h"

using namespace llvm;
PreservedAnalyses FuncInfoAnalysis::run(Function &F, FunctionAnalysisManager &)
{
    errs() << "Name: " << F.getName() << "\n";
    errs() << "NArgs: " << F.arg_size() << "\n";
    errs() << "NBlocks: " << F.size() << "\n";
    errs() << "NInsts: " << F.getInstructionCount() << "\n\n";
    // We don't transform the function actually,
    // so all analyses on the function should be preserved.
    return PreservedAnalyses::all();
}
```

First Custom Transformation Pass

```
// lib/FuncInfoAnalysis.cpp
llvm::PassPluginLibraryInfo
getFuncInfoAnalysisPass()
{
    // Define a callback to register the pass to Function Pass Manager.
    return {LLVM_PLUGIN_API_VERSION, "func-info-analysis", LLVM_VERSION_STRING,
            [](PassBuilder &PB)
            {
                PB.registerPipelineParsingCallback(
                    [](StringRef Name, FunctionPassManager &FPM,
                       ArrayRef<PassBuilder::PipelineElement>)
                    {
                        if (Name == "func-info-analysis")
                        {
                            // Register the pass to the Function Pass Manager .
                            FPM.addPass(FuncInfoAnalysis());
                            return true;
                        }
                        else
                        {
                            return false;
                        }
                    });
            }};
}
```

First Custom Transformation Pass

```
// lib/FuncInfoAnalysis.cpp
// This function is the entry point for `opt` tool to load the pass plugin.
extern "C" LLVM_ATTRIBUTE_WEAK ::llvm::PassPluginLibraryInfo llvmGetPassPluginInfo()
{
    return getFuncInfoAnalysisPass();
}
```

Let's Run!

```
cd build
cmake -DCUSTOM_PASS_LLVM15_DIR=$LLVM15_INSTALL_DIR ..
make
$LLVM15_INSTALL_DIR/bin/opt -load-pass-plugin lib/libFuncInfoAnalysis.dylib --passes=func-info-analysis -disable-output ../test/FuncInfoAnalysis.cpp
```

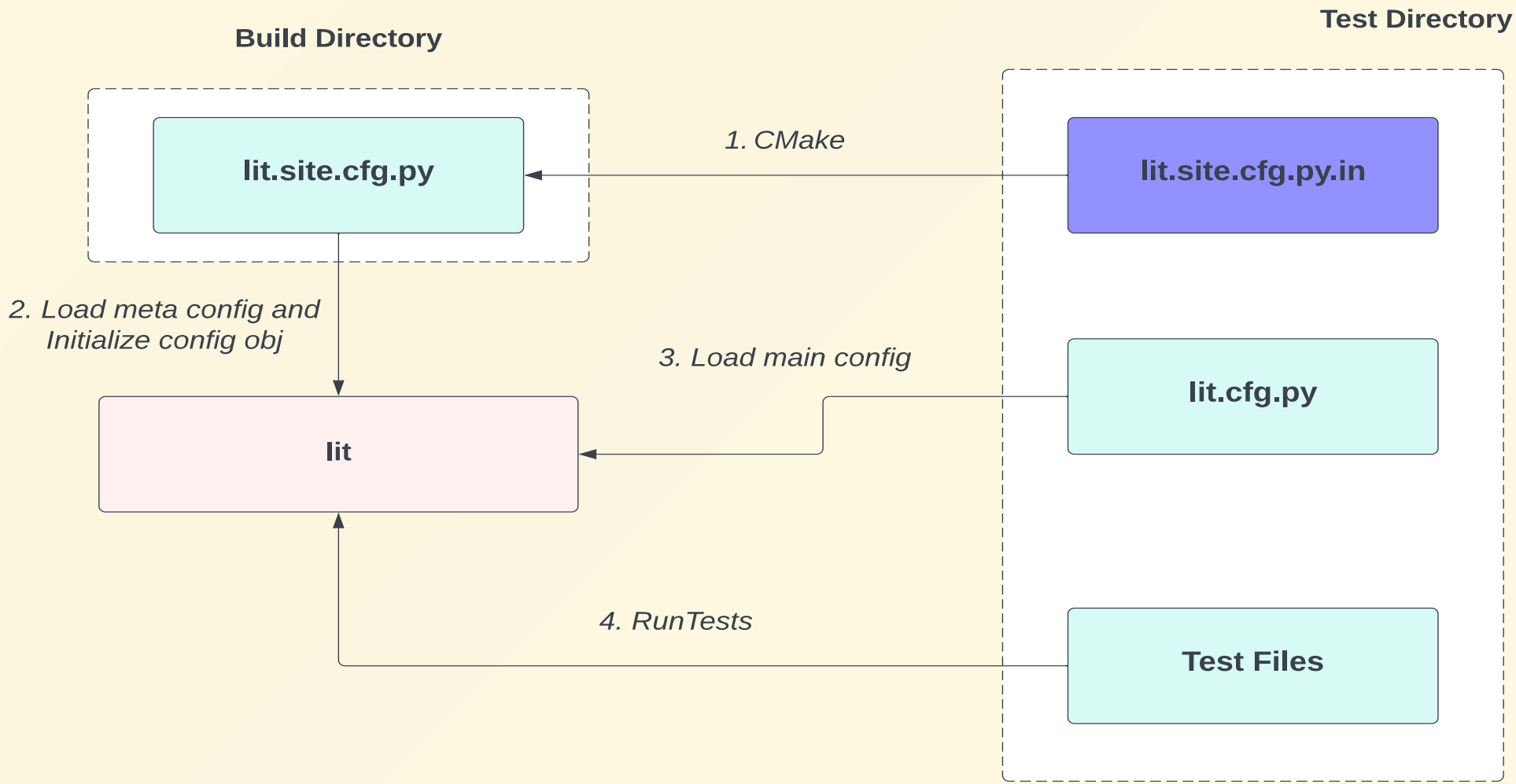
Testing

https://github.com/Y-Nak/write-your-own-pass/commit/03c0af816627fdb2d89a5f3a4229fabd8c0a7dd7

lit Tester

> lit is a portable tool for executing LLVM and Clang style test suites, summarizing their results, and providing indication of failures. lit is designed to be a lightweight testing tool with as simple a user interface as possible.
>
> https://llvm.org/docs/CommandGuide/lit.html

Configuration



Testing/Configuration

```
# test/lit.site.cfg.py.in
import lit.llvm
config.llvm_tools_dir = "@CUSTOM_PASS_LLVM15_DIR@/bin"
config.llvm_plugin_suffix = "@CMAKE_SHARED_LIBRARY_SUFFIX@"
config.llvm_plugin_dir = "@PROJECT_BINARY_DIR@/lib"

lit.llvm.initialize(lit_config, config)
lit_config.load_config(config, "@CMAKE_CURRENT_SOURCE_DIR@/lit.cfg.py")
```

```
# test/CMakeLists.txt
set(CUSTOM_PSSS_TEST_SRC_DIR "${CMAKE_CURRENT_SOURCE_DIR}")
set(CUSTOM_PASS_TEST_SITE_CFG_INPUT "${CMAKE_CURRENT_SOURCE_DIR}/lit.site.cfg.py.in")
configure_file("${CUSTOM_PASS_TEST_SITE_CFG_INPUT}"
  "${CMAKE_CURRENT_BINARY_DIR}/lit.site.cfg.py" @ONLY
)
```

Testing/Configuration

```
# test/lit.cfg.py
import os
import lit
from lit.llvm import llvm_config
from lit.llvm.subst import ToolSubst

config.name = "Custom Pass Test"
config.suffixes = ['.ll']
config.test_format = lit.formats.ShTest(True)
config.test_source_root = os.path.dirname(__file__)
config.test_exec_root = os.path.join("@CMAKE_CURRENT_BINARY_DIR@")

tools = ["opt", "FileCheck"]
llvm_config.add_tool_substitutions(tools, config.llvm_tools_dir)

config.substitutions.append(('%plugin_suffix', config.llvm_plugin_suffix))
config.substitutions.append(('%plugin_dir', config.llvm_plugin_dir))
```

FileCheck

> FileCheck reads two files (one from standard input, and one specified on the command line) and uses one to verify the other.
>
> https://llvm.org/docs/CommandGuide/FileCheck.html

Testing/FileCheck

```
; COM: test/FuncInfoAnalysis
; RUN: opt -load-pass-plugin %plugin_dir/libPrintFuncInfo%plugin_suffix -passes=print-func-info -disable-output 2>&1 %s | FileCheck %s
; CHECK: Name: add
; CHECK-NEXT: NArgs: 2
; CHECK-NEXT: NBlocks: 1
; CHECK-NEXT: NInsts: 8
;
; CHECK: Name: sub1
; CHECK-NEXT: NArgs: 1
; CHECK-NEXT: NBlocks: 1
; CHECK-NEXT: NInsts: 5
;
; CHECK: Name: abs
; CHECK-NEXT: NArgs: 1
; CHECK-NEXT: NBlocks: 4
; CHECK-NEXT: NInsts: 15
; ...
; ...
```

Let's Run Tests!

```
cd build
cmake -DCUSTOM_PASS_LLVM15_DIR=$LLVM15_INSTALL_DIR ..
make
lit test
```

Analysis Pass

https://github.com/Y-Nak/write-your-own-pass/commit/d3c06617681fbdae85f12936458393cd255910d2

> The new PM takes a different approach of completely separating analyses and normal passes. Rather than having the pass manager take care of analyses, a separate analysis manager is in charge of computing, caching, and invalidating analyses. Passes can simply request an analysis from the analysis manager, allowing for lazily computing analyses.
>
> https://blog.llvm.org/posts/2021-03-26-the-new-pass-manager/

Separate Our First Pass into two Passes

Analysis Pass

```
// include/AnalysisType.h
#ifndef CUSTOM_IR_ANALYSIS_TYPE_H
#define CUSTOM_IR_ANALYSIS_TYPE_H

using namespace llvm;

/// This struct is the result type of `FuncInfoAnalysis`
struct FuncInfo
{
    StringRef name;
    size_t nArgs;
    size_t nBlocks;
    size_t nInsts;

    FuncInfo(StringRef name, size_t nArgs, size_t nBlocks, size_t nInsts)
        : name(name), nArgs(nArgs), nBlocks(nBlocks), nInsts(nInsts) {}
};

#endif
```

Analysis Pass

```cpp
///
#ifndef CUSTOM_IR_FUNC_INFO_ANALYSIS_H
#define CUSTOM_IR_FUNC_INFO_ANALYSIS_H

#include "llvm/IR/PassManager.h"

#include "AnalysisType.h"

using namespace llvm;

struct FuncInfoAnalysis : AnalysisInfoMixin<FuncInfoAnalysis>
{
public:
    /// The result of
    using Result = FuncInfo;

    Result run(Function &F, FunctionAnalysisManager &);
    static bool isRequired() { return true; }

private:
    // This should be declared so that PassManager can identify the analysis pass.
    static AnalysisKey Key;
    friend struct AnalysisInfoMixin<FuncInfoAnalysis>;
};

#endif
```

Analysis Pass

```
#include "FuncInfoAnalysis.h"
#include "llvm/Passes/PassBuilder.h"
#include "llvm/Passes/PassPlugin.h"

using namespace llvm;

AnalysisKey FuncInfoAnalysis::Key;

FuncInfoAnalysis::Result FuncInfoAnalysis::run(Function &F, FunctionAnalysisManager &)
{
    auto name = F.getName();
    auto nArgs = F.arg_size();
    auto nBlocks = F.size();
    auto nInsts = F.getInstructionCount();

    return FuncInfoAnalysis::Result(name, nArgs, nBlocks, nInsts);
}

llvm::PassPluginLibraryInfo
getFuncInfoAnalysisPass()
{
    return {LLVM_PLUGIN_API_VERSION, "func-info-analysis", LLVM_VERSION_STRING,
            [](PassBuilder &PB)
            {
                PB.registerAnalysisRegistrationCallback(
                    [](FunctionAnalysisManager &FAM)
                    {
                        FAM.registerPass([&]
                                         { return FuncInfoAnalysis(); });
                    });
            }};
}

extern "C" LLVM_ATTRIBUTE_WEAK ::llvm::PassPluginLibraryInfo llvmGetPassPluginInfo()
{
    return getFuncInfoAnalysisPass();
}
```

Key Features of Analysis Pass

Preserved Analysis

```
// We've made no transformations that can affect any analyses.
return PreservedAnalyses::all();

// We've made transformations and don't want to bother to update any analyses.
return PreservedAnalyses::none();

// We've specifically updated the dominator tree alongside any transformations, but other analysis results may be invalid.
PreservedAnalyses PA;
PA.preserve<DominatorAnalysis>();
return PA;

// We haven't made any control flow changes, any analyses that only care about the control flow are still valid.
PreservedAnalyses PA;
PA.preserveSet<CFGAnalyses>();
return PA;
```

Analysis Invalidation

```
bool FooAnalysisResult::invalidate(Function &F, const PreservedAnalyses &PA,
                                   FunctionAnalysisManager::Invalidator &) {
  auto PAC = PA.getChecker<FooAnalysis>();
  // the default would be:
  // return !(PAC.preserved() || PAC.preservedSet<AllAnalysesOn<Function>>());
  return !(PAC.preserved() || PAC.preservedSet<AllAnalysesOn<Function>>()
      || PAC.preservedSet<CFGAnalyses>());
}
```

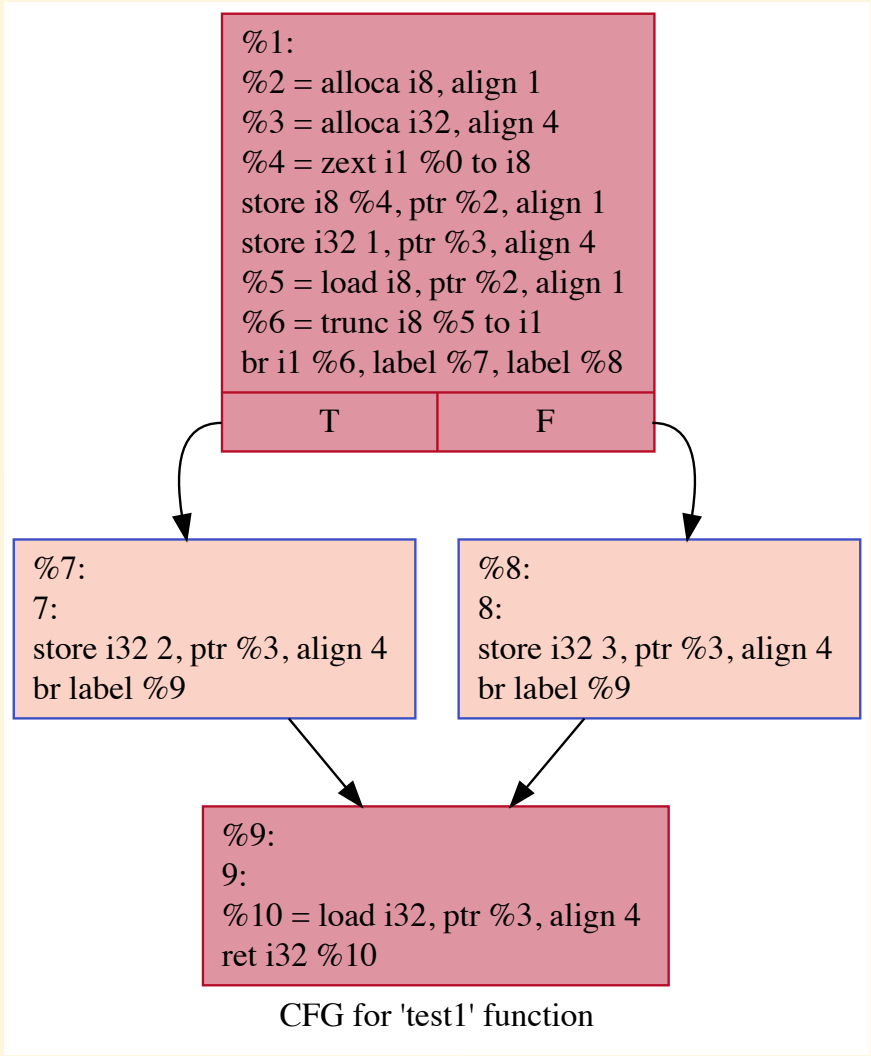
Write Custom Mem2Reg

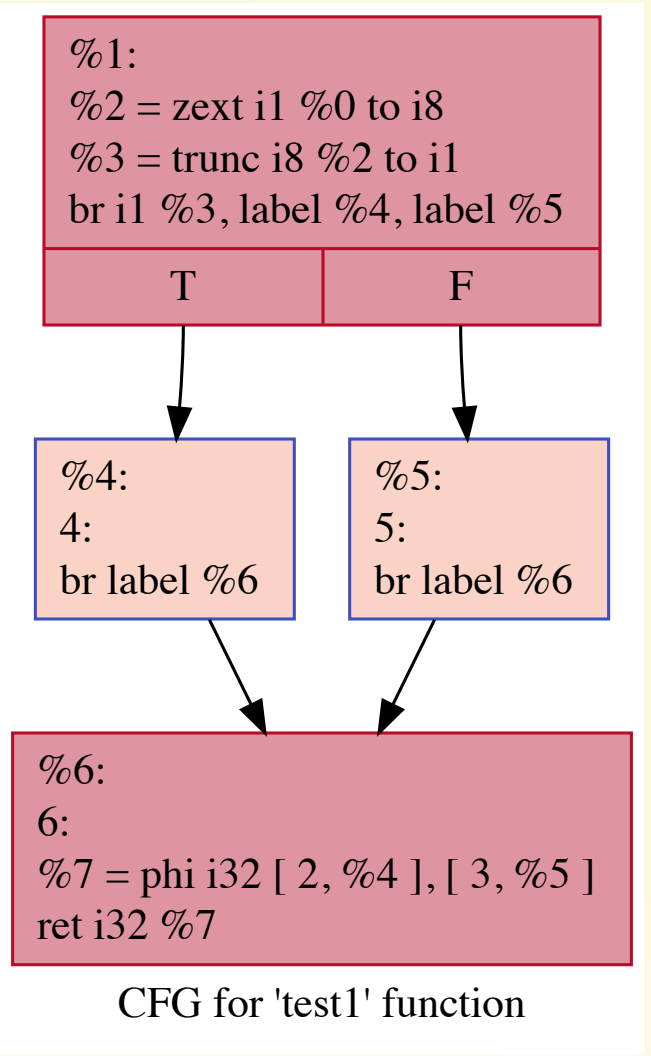
Handle single store instruction:
https://github.com/Y-Nak/write-your-own-pass/commit/39d5f1ae4b631fced9edfa67a38d01935cb0668b
Handle single block store/load instructions:
https://github.com/Y-Nak/write-your-own-pass/commit/d1b8342aba0bbc2ed94506c67efd92d7b20f5ae3
Handle multiple block store/load instructions:
https://github.com/Y-Nak/write-your-own-pass/commit/d1b8342aba0bbc2ed94506c67efd92d7b20f5ae3

Mem2Reg

> “ “ “ This file promotes memory references to be register references. It promotes alloca instructions which only have loads and stores as uses. An alloca is transformed by using dominator frontiers to place phi nodes, then traversing the function in depth-first order to rewrite loads and stores as appropriate. This is just the standard SSA construction algorithm to construct “pruned” SSA form.
>
> https://llvm.org/docs/Passes.html#mem2reg-promote-memory-to-register ” ” ”

Example


%1:
%2 = alloca i8, align 1
%3 = alloca i32, align 4
%4 = zext i1 %0 to i8
store i8 %4, ptr %2, align 1
store i32 1, ptr %3, align 4
%5 = load i8, ptr %2, align 1
%6 = trunc i8 %5 to i1
br i1 %6, label %7, label %8
T
F
%7:
7:
store i32 2, ptr %3, align 4
br label %9
%8:
8:
store i32 3, ptr %3, align 4
br label %9
%9:
9:
%10 = load i32, ptr %3, align 4
ret i32 %10
CFG for 'test1' function


%1:
%2 = zext i1 %0 to i8
%3 = trunc i8 %2 to i1
br i1 %3, label %4, label %5
T
F
%4:
4:
br label %6
%5:
5:
br label %6
%6:
6:
%7 = phi i32 [2, %4], [3, %5]
ret i32 %7
CFG for 'test1' function

Dominator Tree

> Dominator tree is a tree representing a dominance relationship.

Dominator

Dominator:

$$D \in \mathrm{dom}(N) \iff \forall p \in \mathrm{path}(E, N), D \in p$$

Strictly Dominator:

$$D \in \mathrm{sdom}(N) \iff D \in \mathrm{dom}(N) \wedge D \neq N$$

Immediate Dominator:

$$D \in \mathrm{idom}(N) \iff D \in \mathrm{sdom}(N) \wedge \forall p \in \mathrm{path}(D, N), \forall n \neq D \in p, n \notin \mathrm{sdom}(N)$$

Dominance Frontier

Dominance Frontier:

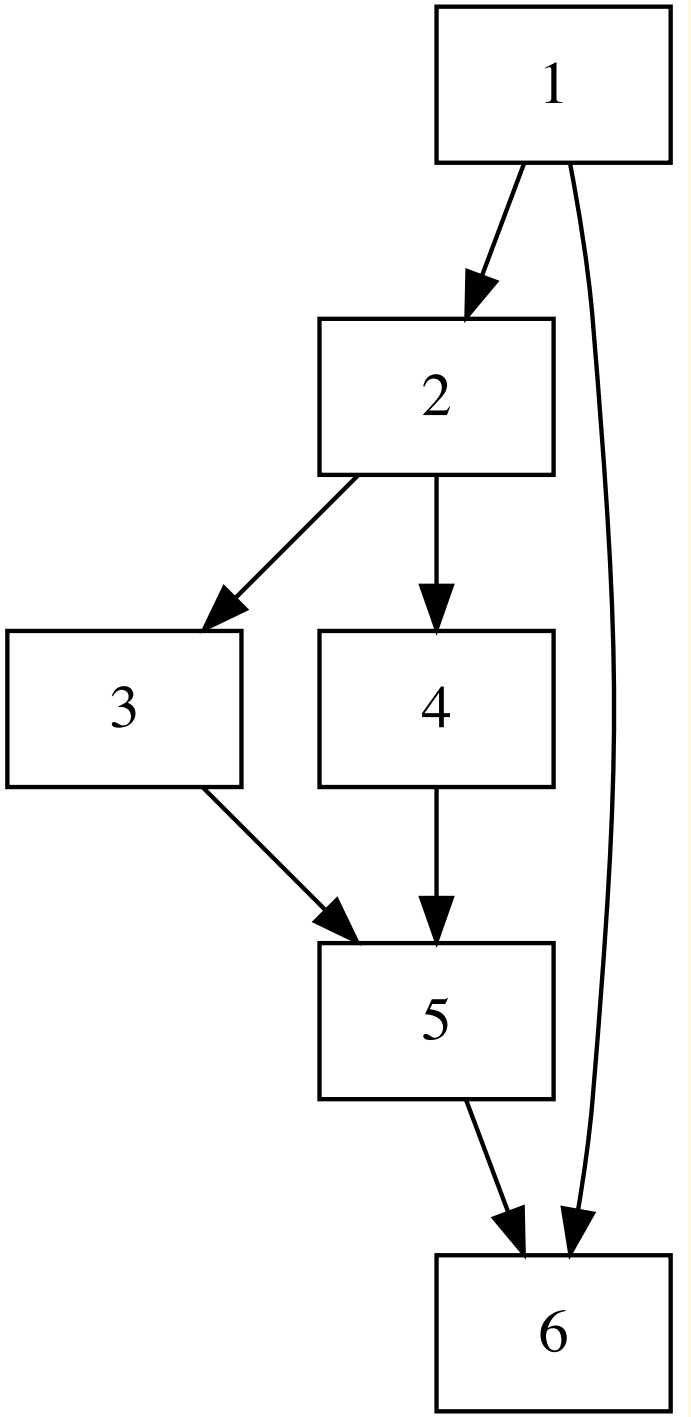
$$DF(N) = \{n \mid N \notin \mathrm{sdom}(n) \wedge \exists m \in \mathrm{pred}(n), N \in \mathrm{sdom}(m)\}$$

Iterated Dominance Frontier:

$$DF^0(N) = DF(N)$$
$$DF^i(N) = DF(DF^{i-1}(N) \cup N)$$
$$IDF(N) = \lim_{i \to \infty} DF^i(N)$$

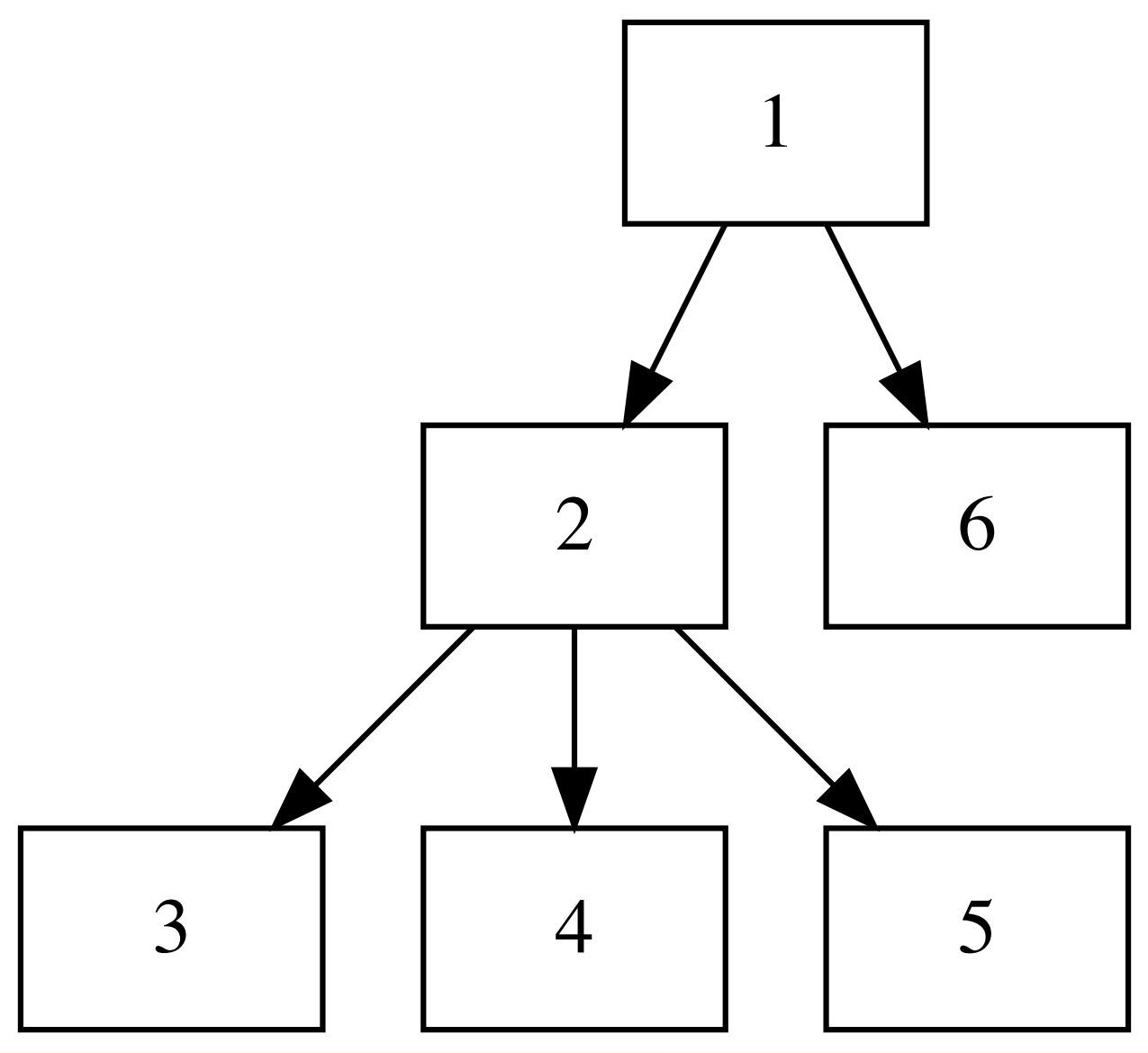
Example

$$dom(3) = \{1, 2, 3\}$$
$$\text{sdom}(3) = \{1, 2\}$$
$$\text{idom}(3) = \{2\}$$
$$\text{DF}(3) = \{5\}$$
$$\text{IDF}(3) = \{5, 6\}$$



Dominator Tree

From the definition of the dominator, we can build a tree structure to represent it because it has reflexive and transitive properties.



Dominator Tree Updater in LLVM

```
class DomTreeUpdater {
public:
  enum class UpdateStrategy : unsigned char { Eager = 0, Lazy = 1 };
    void applyUpdates(ArrayRef<DominatorTree::UpdateType> Updates);
    ...
}
```

```
SmallVector<DominatorTree::UpdateType> DTUpdates;
DTUpdates.emplace_back(DominatorTree::Delete, Src, DeadSucc);
DTU.applyUpdates(DTUpdates);
```

Reference for Incremental Updates of DominatorTree

Trees and incremental updates that transcend time@2017 LLVM Developers' Meeting:
https://www.youtube.com/watch?v=bNV18Wy-J0U

An Experimental Study of Dynamic Dominators:
https://arxiv.org/pdf/1604.02711.pdf

Simplified Mem2Reg Algorithm

1. Handle single store instruction
2. Handle single block store/load instructions
3. Handle multiple block store/load instructions by using Iterated Dominance Frontier.