CHANDLER CARRUTH
LLVM DEVMTG 2014

THE LLVM PASS MANAGER PART 2

FROM THE PREVIOUS TALK:

- A pass operates on some “unit” of IR (Function, Module, …)
- Generally to transform it from one form to another
- Alternatively to analyze its properties and expose higher-level information

LET THE CODE MODEL THIS

A PASS FORMS A SIMPLE CONCEPT:

```cpp
class MyPass {
  // ...

  MyPass(Module *M, bool EnableFoo,
         unsigned BarThreshold) {
    // Set things up...
  }

  void run(Function *F) {
    // Do stuff to F...
  }
};
```

HOW SIMPLE CAN A PASS MANAGER GET?

```cpp
class FunctionPassManager {
  typedef detail::PassConcept<Function *> FunctionPassConcept;

  template <typename PassT>
  struct FunctionPassModel : detail::PassModel<Function *, PassT> {
    FunctionPassModel(PassT Pass)
        : detail::PassModel<Function *, PassT>(std::move(Pass)) {}
  };

  std::vector<std::unique_ptr<FunctionPassConcept>> Passes;

public:
  template <typename FunctionPassT>
  void addPass(FunctionPassT Pass) {
    Passes.emplace_back(
        new FunctionPassModel<FunctionPassT>(std::move(Pass)));
  }

  void run(Function *F) {
    for (const auto &P : Passes)
      P->run(F);
  }
};
```

WHAT IS THIS CONCEPT/MODEL THING?

```
template <typename IRUnitT> struct PassConcept {
  virtual ~PassConcept() {}
  virtual void run(IRUnitT IR) = 0;
};

template <typename IRUnitT, typename PassT>
struct PassModel : PassConcept<IRUnitT> {
  explicit PassModel(PassT Pass) : Pass(std::move(Pass)) {}

  void run(IRUnitT IR) override {
    Pass.run(IR);
  }
  PassT Pass;
};
```

ADAPTING PASS MAPS ACROSS IR UNITS:

```cpp
template <typename FunctionPassT> class ModuleToFunctionPassAdaptor {
  FunctionPassT Pass;
public:
  explicit ModuleToFunctionPassAdaptor(FunctionPassT Pass)
      : Pass(std::move(Pass)) {}

  /// \brief Runs the function pass across every function in the module.
  void run(Module *M) {
    for (Function *F : *M)
      Pass.run(F);
  }
};
```

SO WE’RE DONE! ;]

WHAT ABOUT ANALYSES?
THOSE ARE WHAT MAKE THIS COMPLEX

AN ANALYSIS PASS IS "JUST" A SPECIAL KIND OF PASS…

ANATOMY OF AN ANALYSIS PASS:

- Immutable view of IR
- Produces some result which can be queried
- Result may actually be a lazy-computing interface

LET'S LOOK AT A CONCRETE EXAMPLE…

```
class DominatorTree : public DominatorTreeBase<BasicBlock> {
public:
  typedef DominatorTreeBase<BasicBlock> Base;

  DominatorTree() : DominatorTreeBase<BasicBlock>(false) {}

  inline bool compare(const DominatorTree &Other) const;

  using Base::dominates;
  bool dominates(const Instruction *Def, const Use &U) const;
  bool dominates(const Instruction *Def, const Instruction *User) const;
  bool dominates(const Instruction *Def, const BasicBlock *BB) const;
  bool dominates(const BasicBlockEdge &BBE, const Use &U) const;
  bool dominates(const BasicBlockEdge &BBE, const BasicBlock *BB) const;

  using Base::isReachableFromEntry;
  bool isReachableFromEntry(const Use &U) const;

  void verifyDomTree() const;
};
```

```cpp
class DominatorTreeAnalysis {
public:
  typedef DominatorTree Result;

  /// \brief Returns an opaque, unique ID for this pass type.
  static void *ID() { return (void *)&PassID; }

  DominatorTreeAnalysis() {}

  DominatorTree run(Function *F) {
    DominatorTree DT;
    DT.recalculate(F);
    return std::move(DT);
  }

private:
  /// \brief Private static data to provide unique ID.
  static char PassID;
};
```

THE QUESTION IS, WHEN DO WE RUN THE ANALYSIS PASS?

HISTORICALLY: UP-FRONT DEPENDENCY-BASED SCHEDULING

- Slow to compute schedule due to multitude of abstractions
- Schedule is wasteful as it must conservatively assume each pass trashes the IR and thus invalidates the analysis
- Hard to lazily run analyses for sub-units of IR

INSTEAD, THINK OF THIS AS A CACHING PROBLEM

ANALYSIS "SCHEDULE" BY CACHING RESULTS OF LAZY RUNS

- Because analysis passes cannot mutate IR, we can never hit a cycle
- Can cache each result of an analysis pass on the unit of IR it pertains to, allowing easy access across IR units
- Need some interface for querying (and populating) the cache

```cpp
class FunctionAnalysisManager {
  typedef detail::AnalysisResultConcept<Function *> ResultConceptT;
  typedef detail::AnalysisPassConcept<Function *, FunctionAnalysisManager>
      PassConceptT;

public:
  template <typename PassT>
  typename PassT::Result &getResult(Function *F);
  template <typename PassT>
  typename PassT::Result *getCachedResult(Function *F) const;

  template <typename PassT> void registerPass(PassT Pass);

  template <typename PassT> void invalidate(Module *M);
  void invalidate(Function *F);

private:
  // ... details ...
};
```

NATURALLY, THE PROBLEM BECOMES CACHE INVALIDATION

```
class PreservedAnalyses {
public:
  static PreservedAnalyses none() { return PreservedAnalyses(); }
  static PreservedAnalyses all() {
    PreservedAnalyses PA;
    PA.PreservedPassIDs.insert((void *)AllPassesID);
    return PA;
  }

  template <typename PassT> void preserve() {
    if (!areAllPreserved())
      PreservedPassIDs.insert(PassT::ID());
  }

  template <typename PassT> bool preserved() const {
    return areAllPreserved() || PreservedPassIDs.count(Pass::ID());
  }

private:
  static const uintptr_t AllPassesID = (intptr_t)(-3);
  bool areAllPreserved() const {
    return PreservedPassIDs.count((void *)AllPassesID);
  }

  SmallPtrSet<void *, 2> PreservedPassIDs;
};
```

```
class FunctionPassManager {
public:
  // ...

  PreservedAnalyses run(Function *F, FunctionAnalysisManager *AM = nullptr) {
    PreservedAnalyses PA = PreservedAnalyses::all();

    for (auto &P : Passes) {
      PreservedAnalyses PassPA = P->run(F, AM);
      if (AM)
        AM->invalidate(F, PassPA);
      PA.intersect(std::move(PassPA));
    }

    return PA;
  }

private:
  // ...
};
```

```
class FunctionPassManager {
public:
  // ...

  PreservedAnalyses run(Function *F, FunctionAnalysisManager *AM = nullptr) {
    PreservedAnalyses PA = PreservedAnalyses::all();

    for (auto &P : Passes) {
      PreservedAnalyses PassPA = P->run(F, AM);
      if (AM)
        AM->invalidate(F, PassPA);
      PA.intersect(std::move(PassPA));
    }

    return PA;
  }

private:
  // ...
};
```

```cpp
class FunctionPassManager {
public:
  // ...

  PreservedAnalyses run(Function *F, FunctionAnalysisManager *AM = nullptr) {
    PreservedAnalyses PA = PreservedAnalyses::all();

    for (auto &P : Passes) {
      PreservedAnalyses PassPA = P->run(F, AM);
      if (AM)
        AM->invalidate(F, PassPA);
      PA.intersect(std::move(PassPA));
    }

    return PA;
  }

private:
  // ...
};
```

ANALYSIS MANAGERS USE UNITS OF IR

- Unlike normal passes, these can be bi-directional
- Lower level IR analyses can always be run on demand
- Higher level IR analysis cannot be run on demand, it could conflict with some sibling transformation
- Invalidation must also be propagated bi-directionally!

```cpp
class FunctionAnalysisManagerModuleProxy {
public:
  class FunctionAnalysisManagerModuleProxy::Result {
  public:
    explicit Result(FunctionAnalysisManager &FAM) : FAM(&FAM) {}
    ~Result();
    FunctionAnalysisManager &getManager() { return *FAM; }

    bool invalidate(Module *M, const PreservedAnalyses &PA) {
      if (!PA.preserved(ID()))
        FAM->clear();
      return false;
    }

  private:
    FunctionAnalysisManager *FAM;
  };

  static void *ID() { return (void *)&PassID; }
  explicit FunctionAnalysisManagerModuleProxy(FunctionAnalysisManager &FAM)
      : FAM(&FAM) {}
  Result run(Module *M) { return Result(*FAM); }

private:
  static char PassID;
  FunctionAnalysisManager *FAM;
};
```

```
class FunctionAnalysisManagerModuleProxy {
public:
  class FunctionAnalysisManagerModuleProxy::Result {
  public:
    explicit Result(FunctionAnalysisManager &FAM) : FAM(&FAM) {}
    ~Result();
    FunctionAnalysisManager &getManager() { return *FAM; }

    bool invalidate(Module *M, const PreservedAnalyses &PA) {
      if (!PA.preserved(ID()))
        FAM->clear();
      return false;
    }

  private:
    FunctionAnalysisManager *FAM;
  };

  static void *ID() { return (void *)&PassID; }
  explicit FunctionAnalysisManagerModuleProxy(FunctionAnalysisManager &FAM)
      : FAM(&FAM) {}
  Result run(Module *M) { return Result(*FAM); }

private:
  static char PassID;
  FunctionAnalysisManager *FAM;
};
```

```
class FunctionAnalysisManagerModuleProxy {
public:
  class FunctionAnalysisManagerModuleProxy::Result {
  public:
    explicit Result(FunctionAnalysisManager &FAM) : FAM(&FAM) {}
    ~Result();
    FunctionAnalysisManager &getManager() { return *FAM; }

    bool invalidate(Module *M, const PreservedAnalyses &PA) {
      if (!PA.preserved(ID()))
        FAM->clear();
      return false;
    }

  private:
    FunctionAnalysisManager *FAM;
  };

  static void *ID() { return (void *)&PassID; }
  explicit FunctionAnalysisManagerModuleProxy(FunctionAnalysisManager &FAM)
      : FAM(&FAM) {}
  Result run(Module *M) { return Result(*FAM); }

private:
  static char PassID;
  FunctionAnalysisManager *FAM;
};
```

```cpp
class FunctionAnalysisManagerModuleProxy {
public:
  class FunctionAnalysisManagerModuleProxy::Result {
  public:
    explicit Result(FunctionAnalysisManager &FAM) : FAM(&FAM) {}
    ~Result();
    FunctionAnalysisManager &getManager() { return *FAM; }

    bool invalidate(Module *M, const PreservedAnalyses &PA) {
      if (!PA.preserved(ID()))
        FAM->clear();
      return false;
    }

  private:
    FunctionAnalysisManager *FAM;
  };

  static void *ID() { return (void *)&PassID; }
  explicit FunctionAnalysisManagerModuleProxy(FunctionAnalysisManager &FAM)
      : FAM(&FAM) {}
  Result run(Module *M) { return Result(*FAM); }

private:
  static char PassID;
  FunctionAnalysisManager *FAM;
};
```

```cpp
class ModuleAnalysisManagerFunctionProxy {
public:
  class Result {
  public:
    explicit Result(const ModuleAnalysisManager &MAM) : MAM(&MAM) {}

    const ModuleAnalysisManager &getManager() const { return *MAM; }

    bool invalidate(Function *) { return false; }

  private:
    const ModuleAnalysisManager *MAM;
  };

  static void *ID() { return (void *)&PassID; }
  ModuleAnalysisManagerFunctionProxy(const ModuleAnalysisManager &MAM)
      : MAM(&MAM) {}
  Result run(Function *) { return Result(*MAM); }

private:
  static char PassID;
  const ModuleAnalysisManager *MAM;
};
```

```
class ModuleAnalysisManagerFunctionProxy {
public:
  class Result {
  public:
    explicit Result(const ModuleAnalysisManager &MAM) : MAM(&MAM) {}

    const ModuleAnalysisManager &getManager() const { return *MAM; }

    bool invalidate(Function *) { return false; }

  private:
    const ModuleAnalysisManager *MAM;
  };

  static void *ID() { return (void *)&PassID; }
  ModuleAnalysisManagerFunctionProxy(const ModuleAnalysisManager &MAM)
      : MAM(&MAM) {}
  Result run(Function *) { return Result(*MAM); }

private:
  static char PassID;
  const ModuleAnalysisManager *MAM;
};
```

```
class ModuleAnalysisManagerFunctionProxy {
public:
  class Result {
  public:
    explicit Result(const ModuleAnalysisManager &MAM) : MAM(&MAM) {}

    const ModuleAnalysisManager &getManager() const { return *MAM; }

    bool invalidate(Function *) { return false; }

  private:
    const ModuleAnalysisManager *MAM;
  };

  static void *ID() { return (void *)&PassID; }
  ModuleAnalysisManagerFunctionProxy(const ModuleAnalysisManager &MAM)
      : MAM(&MAM) {}
  Result run(Function *) { return Result(*MAM); }

private:
  static char PassID;
  const ModuleAnalysisManager *MAM;
};
```

```cpp
template <typename FunctionPassT> class ModuleToFunctionPassAdaptor {
public:
  explicit ModuleToFunctionPassAdaptor(FunctionPassT Pass)
      : Pass(std::move(Pass)) {}

  PreservedAnalyses run(Module *M, ModuleAnalysisManager *AM) {
    FunctionAnalysisManager *FAM = nullptr;
    if (AM)
      FAM = &AM->getResult<FunctionAnalysisManagerModuleProxy>(M).getManager();

    PreservedAnalyses PA = PreservedAnalyses::all();
    for (Function *F : *M) {
      PreservedAnalyses PassPA = Pass.run(F, FAM);

      if (FAM)
        FAM->invalidate(I, PassPA);

      PA.intersect(std::move(PassPA));
    }

    PA.preserve<FunctionAnalysisManagerModuleProxy>();
    return PA;
  }

private:
  FunctionPassT Pass;
};
```

```cpp
template <typename FunctionPassT> class ModuleToFunctionPassAdaptor {
public:
  explicit ModuleToFunctionPassAdaptor(FunctionPassT Pass)
      : Pass(std::move(Pass)) {}

  PreservedAnalyses run(Module *M, ModuleAnalysisManager *AM) {
    FunctionAnalysisManager *FAM = nullptr;
    if (AM)
      FAM = &AM->getResult<FunctionAnalysisManagerModuleProxy>(M).getManager();

    PreservedAnalyses PA = PreservedAnalyses::all();
    for (Function *F : *M) {
      PreservedAnalyses PassPA = Pass.run(F, FAM);

      if (FAM)
        FAM->invalidate(F, PassPA);

      PA.intersect(std::move(PassPA));
    }

    PA.preserve<FunctionAnalysisManagerModuleProxy>();
    return PA;
  }

private:
  FunctionPassT Pass;
};
```

```
template <typename FunctionPassT> class ModuleToFunctionPassAdaptor {
public:
  explicit ModuleToFunctionPassAdaptor(FunctionPassT Pass)
      : Pass(std::move(Pass)) {}

  PreservedAnalyses run(Module *M, ModuleAnalysisManager *AM) {
    FunctionAnalysisManager *FAM = nullptr;
    if (AM)
      FAM = &AM->getResult<FunctionAnalysisManagerModuleProxy>(M).getManager();

    PreservedAnalyses PA = PreservedAnalyses::all();
    for (Function *F : *M) {
      PreservedAnalyses PassPA = Pass.run(F, FAM);

      if (FAM)
        FAM->invalidate(I, PassPA);

      PA.intersect(std::move(PassPA));
    }

    PA.preserve<FunctionAnalysisManagerModuleProxy>();
    return PA;
  }

private:
  FunctionPassT Pass;
};
```

HOW DO WE *USE* THESE APIS?

```cpp
class TestFunctionAnalysis {
public:
  struct Result { /* ... */ };
  static void *ID() { return (void *)&PassID; }
  TestFunctionAnalysis() {}

  Result run(Function *F, FunctionAnalysisManager *AM) { return Result(); }

private:
  static char PassID;
};

class TestModuleAnalysis {
public:
  struct Result { /* ... */ };
  static void *ID() { return (void *)&PassID; }
  TestModuleAnalysis() {}

  Result run(Module *M, ModuleAnalysisManager *AM) { return Result(); }

private:
  static char PassID;
};
```

```cpp
struct TestModulePass {
  TestModulePass() {}
  PreservedAnalyses run(Module *M) {
    return PreservedAnalyses::none();
  }
};

struct TestFunctionPass {
  TestFunctionPass() {}

  PreservedAnalyses run(Function *F, FunctionAnalysisManager *AM) {
    const ModuleAnalysisManager &MAM =
        AM->getResult<ModuleAnalysisManagerFunctionProxy>(F).getManager();
    if (TestModuleAnalysis::Result *TMA =
            MAM.getCachedResult<TestModuleAnalysis>(F->getParent()))
      // Do stuff...

    TestFunctionAnalysis::Result &AR = AM->getResult<TestFunctionAnalysis>(F);

    return PreservedAnalyses::all();
  }
};
```

```
void optimize(Module *M) {
  FunctionAnalysisManager FAM;
  FAM.registerPass(TestFunctionAnalysis());

  ModuleAnalysisManager MAM;
  MAM.registerPass(TestModuleAnalysis());
  MAM.registerPass(FunctionAnalysisManagerModuleProxy(FAM));
  FAM.registerPass(ModuleAnalysisManagerFunctionProxy(MAM));

  ModulePassManager MPM;

  {
    ModulePassManager NestedMPM;
    FunctionPassManager FPM;
    {
      FunctionPassManager NestedFPM;
      NestedFPM.addPass(TestFunctionPass());
      FPM = std::move(NestedFPM);
    }
    NestedMPM.addPass(createModuleToFunctionPassAdaptor(std::move(FPM)));
    MPM = std::move(NestedMPM);
  }

  MPM.addPass(TestModulePass());

  MPM.run(M.get(), &MAM);
  // ...
```

LET’S TIE UP SOME LOOSE ENDS

AUTOMATIC REGISTRATION: NOPE.

- Cumbersome without global constructors
- Easily replaced by explicit registration and simple tools to reduce boiler plate
- Plan is to have in-tree passes register with a line in a .def file
- Eventually, need a reasonably rich API to allow plugins access

COMMAND LINE NEEDS MORE STRUCTURE

- Can’t use a flat list of flags
- Instead, parse a very simple textual string
- Expose parsing library for use in other tools
- (future) Expose plugin hooks to parsing code

WHAT ABOUT RE-USING PASSES?

- Compile time hit of re-building pass pipelines largely obviated by avoiding expensive scheduling
- If necessary, passes can expose a pass context that can be leveraged to share data structure allocations
- Unlikely to be the common case

WHERE DO THINGS STAND?

INFRASTRUCTURE IS IN-TREE, FUNCTIONING

- A few passes are even ported and usable with it
- Most analysis passes aren't available
- No wiring for machine level passes, need separate llc-style pipeline
- Still is enough to play around with if interested =]

SUPPORT FOR IR CONSTRUCTS

- Currently, supports Modules, Functions, and SCCs in the call graph.
- No support for Loops yet.
- No support planned for basic blocks, instructions, or regions.

WHAT'S NEXT?

FIRST: PORT THE EXISTING PASS PIPELINE(S)

- Involves porting each pass within todays pipelines to new infrastructure
- Requires them to have really accurate invalidation information
- Requires using some new base analyses, ex. the call graph
- Needs generic analysis API shared between the two systems
- Lots of plumbing of flags, frontend options, etc.

SECOND: SOLVE LONG-STANDING PROBLEMS

- Use loop nest and profile info in the inliner
- Teach the inline cost analysis to forward stores to loads across the call site
- Add a cold region outliner based on dominator trees and profile information
- Lazy-loading-friendly LTO DCE

THIRD: PARALLELIZE THE OPTIMIZER

- Allow adaptors to spin up passes over smaller units of IR in parallel where the IR data structures allow
- Needs rich parallelization APIs in LLVM
- Needs low-synchronization thread-safe use lists for globals
- Many constraints of the new pass manager are designed to enable and facilitate this

THANK YOU!

QUESTIONS?