```cpp
//===- PassManager.h - Pass management infrastructure -----------*- C++ -*-===//
//
// Part of the LLVM Project, under the Apache License v2.0 with LLVM Exceptions.
// See https://llvm.org/LICENSE.txt for license information.
// SPDX-License-Identifier: Apache-2.0 WITH LLVM-exception
//
//===----------------------------------------------------------------------===//
/// \file
///
/// This header defines various interfaces for pass management in LLVM. There
/// is no "pass" interface in LLVM per se. Instead, an instance of any class
/// which supports a method to 'run' it over a unit of IR can be used as
/// a pass. A pass manager is generally a tool to collect a sequence of passes
/// which run over a particular IR construct, and run each of them in sequence
/// over each such construct in the containing IR construct. As there is no
/// containing IR construct for a Module, a manager for passes over modules
/// forms the base case which runs its managed passes in sequence over the
/// single module provided.
///
/// The core IR library provides managers for running passes over
/// modules and functions.
///
/// * FunctionPassManager can run over a Module, runs each pass over
///   a Function.
/// * ModulePassManager must be directly run, runs each pass over the Module.
///
/// Note that the implementations of the pass managers use concept-based
/// polymorphism as outlined in the "Value Semantics and Concept-based
/// Polymorphism" talk (or its abbreviated sibling "Inheritance Is The Base
/// Class of Evil") by Sean Parent:
/// * http://github.com/sean-parent/sean-parent.github.com/wiki/Papers-and-Presentations
/// * http://www.youtube.com/watch?v=_BpMYeUFXv8
/// *
http://channel9.msdn.com/Events/GoingNative/2013/Inheritance-Is-The-Base-Class-of-Evil
///
//===----------------------------------------------------------------------===//

#ifndef LLVM_IR_PASSMANAGER_H
#define LLVM_IR_PASSMANAGER_H

#include "llvm/ADT/DenseMap.h"
#include "llvm/ADT/SmallPtrSet.h"
#include "llvm/ADT/StringRef.h"
#include "llvm/ADT/TinyPtrVector.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/PassInstrumentation.h"
#include "llvm/IR/PassManagerInternal.h"
#include "llvm/Support/Debug.h"
#include "llvm/Support/TypeName.h"
#include "llvm/Support/raw_ostream.h"
#include <algorithm>
#include <cassert>
#include <cstring>
#include <iterator>
#include <list>
#include <memory>
#include <tuple>
#include <type_traits>
#include <utility>
#include <vector>

namespace llvm {

/// A special type used by analysis passes to provide an address that
/// identifies that particular analysis pass type.
///
/// Analysis passes should have a static data member of this type and derive
/// from the \c AnalysisInfoMixin to get a static ID method used to identify
```

```
/// the analysis in the pass management infrastructure.
struct alignas(8) AnalysisKey {};

/// A special type used to provide an address that identifies a set of related
/// analyses.  These sets are primarily used below to mark sets of analyses as
/// preserved.
///
/// For example, a transformation can indicate that it preserves the CFG of a
/// function by preserving the appropriate AnalysisSetKey.  An analysis that
/// depends only on the CFG can then check if that AnalysisSetKey is preserved;
/// if it is, the analysis knows that it itself is preserved.
struct alignas(8) AnalysisSetKey {};

/// This templated class represents "all analyses that operate over \<a
/// particular IR unit\>" (e.g. a Function or a Module) in instances of
/// PreservedAnalysis.
///
/// This lets a transformation say e.g. "I preserved all function analyses".
///
/// Note that you must provide an explicit instantiation declaration and
/// definition for this template in order to get the correct behavior on
/// Windows. Otherwise, the address of SetKey will not be stable.
template <typename IRUnitT> class AllAnalysesOn {
public:
  static AnalysisSetKey *ID() { return &SetKey; }

private:
  static AnalysisSetKey SetKey;
};

template <typename IRUnitT> AnalysisSetKey AllAnalysesOn<IRUnitT>::SetKey;

extern template class AllAnalysesOn<Module>;
extern template class AllAnalysesOn<Function>;

/// Represents analyses that only rely on functions' control flow.
///
/// This can be used with \c PreservedAnalyses to mark the CFG as preserved and
/// to query whether it has been preserved.
///
/// The CFG of a function is defined as the set of basic blocks and the edges
/// between them. Changing the set of basic blocks in a function is enough to
/// mutate the CFG. Mutating the condition of a branch or argument of an
/// invoked function does not mutate the CFG, but changing the successor labels
/// of those instructions does.
class CFGAnalyses {
public:
  static AnalysisSetKey *ID() { return &SetKey; }

private:
  static AnalysisSetKey SetKey;
};

/// A set of analyses that are preserved following a run of a transformation
/// pass.
///
/// Transformation passes build and return these objects to communicate which
/// analyses are still valid after the transformation. For most passes this is
/// fairly simple: if they don't change anything all analyses are preserved,
/// otherwise only a short list of analyses that have been explicitly updated
/// are preserved.
///
/// This class also lets transformation passes mark abstract *sets* of analyses
/// as preserved. A transformation that (say) does not alter the CFG can
/// indicate such by marking a particular AnalysisSetKey as preserved, and
/// then analyses can query whether that AnalysisSetKey is preserved.
///
/// Finally, this class can represent an "abandoned" analysis, which is
/// not preserved even if it would be covered by some abstract set of analyses.
```

```
///
/// Given a `PreservedAnalyses` object, an analysis will typically want to
/// figure out whether it is preserved. In the example below, MyAnalysisType is
/// preserved if it's not abandoned, and (a) it's explicitly marked as
/// preserved, (b), the set AllAnalysesOn<MyIRUnit> is preserved, or (c) both
/// AnalysisSetA and AnalysisSetB are preserved.
///
/// ```
///   auto PAC = PA.getChecker<MyAnalysisType>();
///   if (PAC.preserved() || PAC.preservedSet<AllAnalysesOn<MyIRUnit>>() ||
///       (PAC.preservedSet<AnalysisSetA>() &&
///        PAC.preservedSet<AnalysisSetB>())) {
///     // The analysis has been successfully preserved ...
///   }
/// ```
class PreservedAnalyses {
public:
  /// Convenience factory function for the empty preserved set.
  static PreservedAnalyses none() { return PreservedAnalyses(); }

  /// Construct a special preserved set that preserves all passes.
  static PreservedAnalyses all() {
    PreservedAnalyses PA;
    PA.PreservedIDs.insert(&AllAnalysesKey);
    return PA;
  }

  /// Construct a preserved analyses object with a single preserved set.
  template <typename AnalysisSetT>
  static PreservedAnalyses allInSet() {
    PreservedAnalyses PA;
    PA.preserveSet<AnalysisSetT>();
    return PA;
  }

  /// Mark an analysis as preserved.
  template <typename AnalysisT> void preserve() { preserve(AnalysisT::ID()); }

  /// Given an analysis's ID, mark the analysis as preserved, adding it
  /// to the set.
  void preserve(AnalysisKey *ID) {
    // Clear this ID from the explicit not-preserved set if present.
    NotPreservedAnalysisIDs.erase(ID);

    // If we're not already preserving all analyses (other than those in
    // NotPreservedAnalysisIDs).
    if (!areAllPreserved())
      PreservedIDs.insert(ID);
  }

  /// Mark an analysis set as preserved.
  template <typename AnalysisSetT> void preserveSet() {
    preserveSet(AnalysisSetT::ID());
  }

  /// Mark an analysis set as preserved using its ID.
  void preserveSet(AnalysisSetKey *ID) {
    // If we're not already in the saturated 'all' state, add this set.
    if (!areAllPreserved())
      PreservedIDs.insert(ID);
  }

  /// Mark an analysis as abandoned.
  ///
  /// An abandoned analysis is not preserved, even if it is nominally covered
  /// by some other set or was previously explicitly marked as preserved.
  ///
  /// Note that you can only abandon a specific analysis, not a *set* of
  /// analyses.
```

```cpp
  template <typename AnalysisT> void abandon() { abandon(AnalysisT::ID()); }

  /// Mark an analysis as abandoned using its ID.
  ///
  /// An abandoned analysis is not preserved, even if it is nominally covered
  /// by some other set or was previously explicitly marked as preserved.
  ///
  /// Note that you can only abandon a specific analysis, not a *set* of
  /// analyses.
  void abandon(AnalysisKey *ID) {
    PreservedIDs.erase(ID);
    NotPreservedAnalysisIDs.insert(ID);
  }

  /// Intersect this set with another in place.
  ///
  /// This is a mutating operation on this preserved set, removing all
  /// preserved passes which are not also preserved in the argument.
  void intersect(const PreservedAnalyses &Arg) {
    if (Arg.areAllPreserved())
      return;
    if (areAllPreserved()) {
      *this = Arg;
      return;
    }
    // The intersection requires the *union* of the explicitly not-preserved
    // IDs and the *intersection* of the preserved IDs.
    for (auto ID : Arg.NotPreservedAnalysisIDs) {
      PreservedIDs.erase(ID);
      NotPreservedAnalysisIDs.insert(ID);
    }
    for (auto ID : PreservedIDs)
      if (!Arg.PreservedIDs.count(ID))
        PreservedIDs.erase(ID);
  }

  /// Intersect this set with a temporary other set in place.
  ///
  /// This is a mutating operation on this preserved set, removing all
  /// preserved passes which are not also preserved in the argument.
  void intersect(PreservedAnalyses &&Arg) {
    if (Arg.areAllPreserved())
      return;
    if (areAllPreserved()) {
      *this = std::move(Arg);
      return;
    }
    // The intersection requires the *union* of the explicitly not-preserved
    // IDs and the *intersection* of the preserved IDs.
    for (auto ID : Arg.NotPreservedAnalysisIDs) {
      PreservedIDs.erase(ID);
      NotPreservedAnalysisIDs.insert(ID);
    }
    for (auto ID : PreservedIDs)
      if (!Arg.PreservedIDs.count(ID))
        PreservedIDs.erase(ID);
  }

  /// A checker object that makes it easy to query for whether an analysis or
  /// some set covering it is preserved.
  class PreservedAnalysisChecker {
    friend class PreservedAnalyses;

    const PreservedAnalyses &PA;
    AnalysisKey *const ID;
    const bool IsAbandoned;

    /// A PreservedAnalysisChecker is tied to a particular Analysis because
    /// `preserved()` and `preservedSet()` both return false if the Analysis
```

```cpp
    /// was abandoned.
    PreservedAnalysisChecker(const PreservedAnalyses &PA, AnalysisKey *ID)
        : PA(PA), ID(ID), IsAbandoned(PA.NotPreservedAnalysisIDs.count(ID)) {}

  public:
    /// Returns true if the checker's analysis was not abandoned and either
    ///  - the analysis is explicitly preserved or
    ///  - all analyses are preserved.
    bool preserved() {
      return !IsAbandoned && (PA.PreservedIDs.count(&AllAnalysesKey) ||
                              PA.PreservedIDs.count(ID));
    }

    /// Return true if the checker's analysis was not abandoned, i.e. it was not
    /// explicitly invalidated. Even if the analysis is not explicitly
    /// preserved, if the analysis is known stateless, then it is preserved.
    bool preservedWhenStateless() {
      return !IsAbandoned;
    }

    /// Returns true if the checker's analysis was not abandoned and either
    ///  - \p AnalysisSetT is explicitly preserved or
    ///  - all analyses are preserved.
    template <typename AnalysisSetT> bool preservedSet() {
      AnalysisSetKey *SetID = AnalysisSetT::ID();
      return !IsAbandoned && (PA.PreservedIDs.count(&AllAnalysesKey) ||
                              PA.PreservedIDs.count(SetID));
    }
  };

  /// Build a checker for this `PreservedAnalyses` and the specified analysis
  /// type.
  ///
  /// You can use the returned object to query whether an analysis was
  /// preserved. See the example in the comment on `PreservedAnalysis`.
  template <typename AnalysisT> PreservedAnalysisChecker getChecker() const {
    return PreservedAnalysisChecker(*this, AnalysisT::ID());
  }

  /// Build a checker for this `PreservedAnalyses` and the specified analysis
  /// ID.
  ///
  /// You can use the returned object to query whether an analysis was
  /// preserved. See the example in the comment on `PreservedAnalysis`.
  PreservedAnalysisChecker getChecker(AnalysisKey *ID) const {
    return PreservedAnalysisChecker(*this, ID);
  }

  /// Test whether all analyses are preserved (and none are abandoned).
  ///
  /// This is used primarily to optimize for the common case of a transformation
  /// which makes no changes to the IR.
  bool areAllPreserved() const {
    return NotPreservedAnalysisIDs.empty() &&
           PreservedIDs.count(&AllAnalysesKey);
  }

  /// Directly test whether a set of analyses is preserved.
  ///
  /// This is only true when no analyses have been explicitly abandoned.
  template <typename AnalysisSetT> bool allAnalysesInSetPreserved() const {
    return allAnalysesInSetPreserved(AnalysisSetT::ID());
  }

  /// Directly test whether a set of analyses is preserved.
  ///
  /// This is only true when no analyses have been explicitly abandoned.
  bool allAnalysesInSetPreserved(AnalysisSetKey *SetID) const {
    return NotPreservedAnalysisIDs.empty() &&
```

```
          (PreservedIDs.count(&AllAnalysesKey) || PreservedIDs.count(SetID));
  }

private:
  /// A special key used to indicate all analyses.
  static AnalysisSetKey AllAnalysesKey;

  /// The IDs of analyses and analysis sets that are preserved.
  SmallPtrSet<void *, 2> PreservedIDs;

  /// The IDs of explicitly not-preserved analyses.
  ///
  /// If an analysis in this set is covered by a set in `PreservedIDs`, we
  /// consider it not-preserved. That is, `NotPreservedAnalysisIDs` always
  /// "wins" over analysis sets in `PreservedIDs`.
  ///
  /// Also, a given ID should never occur both here and in `PreservedIDs`.
  SmallPtrSet<AnalysisKey *, 2> NotPreservedAnalysisIDs;
};

// Forward declare the analysis manager template.
template <typename IRUnitT, typename... ExtraArgTs> class AnalysisManager;

/// A CRTP mix-in to automatically provide informational APIs needed for
/// passes.
///
/// This provides some boilerplate for types that are passes.
template <typename DerivedT> struct PassInfoMixin {
  /// Gets the name of the pass we are mixed into.
  static StringRef name() {
    static_assert(std::is_base_of<PassInfoMixin, DerivedT>::value,
                  "Must pass the derived type as the template argument!");
    StringRef Name = getTypeName<DerivedT>();
    if (Name.startswith("llvm::"))
      Name = Name.drop_front(strlen("llvm::"));
    return Name;
  }
};

/// A CRTP mix-in that provides informational APIs needed for analysis passes.
///
/// This provides some boilerplate for types that are analysis passes. It
/// automatically mixes in \c PassInfoMixin.
template <typename DerivedT>
struct AnalysisInfoMixin : PassInfoMixin<DerivedT> {
  /// Returns an opaque, unique ID for this analysis type.
  ///
  /// This ID is a pointer type that is guaranteed to be 8-byte aligned and thus
  /// suitable for use in sets, maps, and other data structures that use the low
  /// bits of pointers.
  ///
  /// Note that this requires the derived type provide a static \c AnalysisKey
  /// member called \c Key.
  ///
  /// FIXME: The only reason the mixin type itself can't declare the Key value
  /// is that some compilers cannot correctly unique a templated static variable
  /// so it has the same addresses in each instantiation. The only currently
  /// known platform with this limitation is Windows DLL builds, specifically
  /// building each part of LLVM as a DLL. If we ever remove that build
  /// configuration, this mixin can provide the static key as well.
  static AnalysisKey *ID() {
    static_assert(std::is_base_of<AnalysisInfoMixin, DerivedT>::value,
                  "Must pass the derived type as the template argument!");
    return &DerivedT::Key;
  }
};

namespace detail {

```

```
/// Actual unpacker of extra arguments in getAnalysisResult,
/// passes only those tuple arguments that are mentioned in index_sequence.
template <typename PassT, typename IRUnitT, typename AnalysisManagerT,
          typename... ArgTs, size_t... Ns>
typename PassT::Result
getAnalysisResultUnpackTuple(AnalysisManagerT &AM, IRUnitT &IR,
                             std::tuple<ArgTs...> Args,
                             llvm::index_sequence<Ns...>) {
  (void)Args;
  return AM.template getResult<PassT>(IR, std::get<Ns>(Args)...);
}

/// Helper for *partial* unpacking of extra arguments in getAnalysisResult.
///
/// Arguments passed in tuple come from PassManager, so they might have extra
/// arguments after those AnalysisManager's ExtraArgTs ones that we need to
/// pass to getResult.
template <typename PassT, typename IRUnitT, typename... AnalysisArgTs,
          typename... MainArgTs>
typename PassT::Result
getAnalysisResult(AnalysisManager<IRUnitT, AnalysisArgTs...> &AM, IRUnitT &IR,
                  std::tuple<MainArgTs...> Args) {
  return (getAnalysisResultUnpackTuple<
          PassT, IRUnitT>)(AM, IR, Args,
                           llvm::index_sequence_for<AnalysisArgTs...>{});
}

} // namespace detail

// Forward declare the pass instrumentation analysis explicitly queried in
// generic PassManager code.
// FIXME: figure out a way to move PassInstrumentationAnalysis into its own
// header.
class PassInstrumentationAnalysis;

/// Manages a sequence of passes over a particular unit of IR.
///
/// A pass manager contains a sequence of passes to run over a particular unit
/// of IR (e.g. Functions, Modules). It is itself a valid pass over that unit of
/// IR, and when run over some given IR will run each of its contained passes in
/// sequence. Pass managers are the primary and most basic building block of a
/// pass pipeline.
///
/// When you run a pass manager, you provide an \c AnalysisManager<IRUnitT>
/// argument. The pass manager will propagate that analysis manager to each
/// pass it runs, and will call the analysis manager's invalidation routine with
/// the PreservedAnalyses of each pass it runs.
template <typename IRUnitT,
          typename AnalysisManagerT = AnalysisManager<IRUnitT>,
          typename... ExtraArgTs>
class PassManager : public PassInfoMixin<
                        PassManager<IRUnitT, AnalysisManagerT, ExtraArgTs...>> {
public:
  /// Construct a pass manager.
  ///
  /// If \p DebugLogging is true, we'll log our progress to llvm::dbgs().
  explicit PassManager(bool DebugLogging = false) : DebugLogging(DebugLogging) {}

  // FIXME: These are equivalent to the default move constructor/move
  // assignment. However, using = default triggers linker errors due to the
  // explicit instantiations below. Find away to use the default and remove the
  // duplicated code here.
  PassManager(PassManager &&Arg)
      : Passes(std::move(Arg.Passes)),
        DebugLogging(std::move(Arg.DebugLogging)) {}

  PassManager &operator=(PassManager &&RHS) {
    Passes = std::move(RHS.Passes);
    DebugLogging = std::move(RHS.DebugLogging);
```

```
    return *this;
  }

  /// Run all of the passes in this manager over the given unit of IR.
  /// ExtraArgs are passed to each pass.
  PreservedAnalyses run(IRUnitT &IR, AnalysisManagerT &AM,
                        ExtraArgTs... ExtraArgs) {
    PreservedAnalyses PA = PreservedAnalyses::all();

    // Request PassInstrumentation from analysis manager, will use it to run
    // instrumenting callbacks for the passes later.
    // Here we use std::tuple wrapper over getResult which helps to extract
    // AnalysisManager's arguments out of the whole ExtraArgs set.
    PassInstrumentation PI =
        detail::getAnalysisResult<PassInstrumentationAnalysis>(
            AM, IR, std::tuple<ExtraArgTs...>(ExtraArgs...));

    if (DebugLogging)
      dbgs() << "Starting " << getTypeName<IRUnitT>() << " pass manager run.\n";

    for (unsigned Idx = 0, Size = Passes.size(); Idx != Size; ++Idx) {
      auto *P = Passes[Idx].get();
      if (DebugLogging)
        dbgs() << "Running pass: " << P->name() << " on " << IR.getName()
               << "\n";

      // Check the PassInstrumentation's BeforePass callbacks before running the
      // pass, skip its execution completely if asked to (callback returns
      // false).
      if (!PI.runBeforePass<IRUnitT>(*P, IR))
        continue;

      PreservedAnalyses PassPA = P->run(IR, AM, ExtraArgs...);

      // Call onto PassInstrumentation's AfterPass callbacks immediately after
      // running the pass.
      PI.runAfterPass<IRUnitT>(*P, IR);

      // Update the analysis manager as each pass runs and potentially
      // invalidates analyses.
      AM.invalidate(IR, PassPA);

      // Finally, intersect the preserved analyses to compute the aggregate
      // preserved set for this pass manager.
      PA.intersect(std::move(PassPA));

      // FIXME: Historically, the pass managers all called the LLVM context's
      // yield function here. We don't have a generic way to acquire the
      // context and it isn't yet clear what the right pattern is for yielding
      // in the new pass manager so it is currently omitted.
      //IR.getContext().yield();
    }

    // Invalidation was handled after each pass in the above loop for the
    // current unit of IR. Therefore, the remaining analysis results in the
    // AnalysisManager are preserved. We mark this with a set so that we don't
    // need to inspect each one individually.
    PA.preserveSet<AllAnalysesOn<IRUnitT>>();

    if (DebugLogging)
      dbgs() << "Finished " << getTypeName<IRUnitT>() << " pass manager run.\n";

    return PA;
  }

  template <typename PassT> void addPass(PassT Pass) {
    using PassModelT =
        detail::PassModel<IRUnitT, PassT, PreservedAnalyses, AnalysisManagerT,
                          ExtraArgTs...>;
```

```cpp

    Passes.emplace_back(new PassModelT(std::move(Pass)));
  }

private:
  using PassConceptT =
      detail::PassConcept<IRUnitT, AnalysisManagerT, ExtraArgTs...>;

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

  /// Flag indicating whether we should do debug logging.
  bool DebugLogging;
};

extern template class PassManager<Module>;

/// Convenience typedef for a pass manager over modules.
using ModulePassManager = PassManager<Module>;

extern template class PassManager<Function>;

/// Convenience typedef for a pass manager over functions.
using FunctionPassManager = PassManager<Function>;

/// Pseudo-analysis pass that exposes the \c PassInstrumentation to pass
/// managers. Goes before AnalysisManager definition to provide its
/// internals (e.g PassInstrumentationAnalysis::ID) for use there if needed.
/// FIXME: figure out a way to move PassInstrumentationAnalysis into its own
/// header.
class PassInstrumentationAnalysis
    : public AnalysisInfoMixin<PassInstrumentationAnalysis> {
  friend AnalysisInfoMixin<PassInstrumentationAnalysis>;
  static AnalysisKey Key;

  PassInstrumentationCallbacks *Callbacks;

public:
  /// PassInstrumentationCallbacks object is shared, owned by something else,
  /// not this analysis.
  PassInstrumentationAnalysis(PassInstrumentationCallbacks *Callbacks = nullptr)
      : Callbacks(Callbacks) {}

  using Result = PassInstrumentation;

  template <typename IRUnitT, typename AnalysisManagerT, typename... ExtraArgTs>
  Result run(IRUnitT &, AnalysisManagerT &, ExtraArgTs &&...) {
    return PassInstrumentation(Callbacks);
  }
};

/// A container for analyses that lazily runs them and caches their
/// results.
///
/// This class can manage analyses for any IR unit where the address of the IR
/// unit suffices as its identity.
template <typename IRUnitT, typename... ExtraArgTs> class AnalysisManager {
public:
  class Invalidator;

private:
  // Now that we've defined our invalidator, we can define the concept types.
  using ResultConceptT =
      detail::AnalysisResultConcept<IRUnitT, PreservedAnalyses, Invalidator>;
  using PassConceptT =
      detail::AnalysisPassConcept<IRUnitT, PreservedAnalyses, Invalidator,
                                  ExtraArgTs...>;

  /// List of analysis pass IDs and associated concept pointers.
  ///
```

```
    /// Requires iterators to be valid across appending new entries and arbitrary
    /// erases. Provides the analysis ID to enable finding iterators to a given
    /// entry in maps below, and provides the storage for the actual result
    /// concept.
    using AnalysisResultListT =
        std::list<std::pair<AnalysisKey *, std::unique_ptr<ResultConceptT>>>;

    /// Map type from IRUnitT pointer to our custom list type.
    using AnalysisResultListMapT = DenseMap<IRUnitT *, AnalysisResultListT>;

    /// Map type from a pair of analysis ID and IRUnitT pointer to an
    /// iterator into a particular result list (which is where the actual analysis
    /// result is stored).
    using AnalysisResultMapT =
        DenseMap<std::pair<AnalysisKey *, IRUnitT *>,
                 typename AnalysisResultListT::iterator>;

  public:
    /// API to communicate dependencies between analyses during invalidation.
    ///
    /// When an analysis result embeds handles to other analysis results, it
    /// needs to be invalidated both when its own information isn't preserved and
    /// when any of its embedded analysis results end up invalidated. We pass an
    /// \c Invalidator object as an argument to \c invalidate() in order to let
    /// the analysis results themselves define the dependency graph on the fly.
    /// This lets us avoid building building an explicit representation of the
    /// dependencies between analysis results.
    class Invalidator {
    public:
      /// Trigger the invalidation of some other analysis pass if not already
      /// handled and return whether it was in fact invalidated.
      ///
      /// This is expected to be called from within a given analysis result's \c
      /// invalidate method to trigger a depth-first walk of all inter-analysis
      /// dependencies. The same \p IR unit and \p PA passed to that result's \c
      /// invalidate method should in turn be provided to this routine.
      ///
      /// The first time this is called for a given analysis pass, it will call
      /// the corresponding result's \c invalidate method.  Subsequent calls will
      /// use a cache of the results of that initial call.  It is an error to form
      /// cyclic dependencies between analysis results.
      ///
      /// This returns true if the given analysis's result is invalid. Any
      /// dependecies on it will become invalid as a result.
      template <typename PassT>
      bool invalidate(IRUnitT &IR, const PreservedAnalyses &PA) {
        using ResultModelT =
            detail::AnalysisResultModel<IRUnitT, PassT, typename PassT::Result,
                                        PreservedAnalyses, Invalidator>;

        return invalidateImpl<ResultModelT>(PassT::ID(), IR, PA);
      }

      /// A type-erased variant of the above invalidate method with the same core
      /// API other than passing an analysis ID rather than an analysis type
      /// parameter.
      ///
      /// This is sadly less efficient than the above routine, which leverages
      /// the type parameter to avoid the type erasure overhead.
      bool invalidate(AnalysisKey *ID, IRUnitT &IR, const PreservedAnalyses &PA) {
        return invalidateImpl<>(ID, IR, PA);
      }

    private:
      friend class AnalysisManager;

      template <typename ResultT = ResultConceptT>
      bool invalidateImpl(AnalysisKey *ID, IRUnitT &IR,
                          const PreservedAnalyses &PA) {
```

```
      // If we've already visited this pass, return true if it was invalidated
      // and false otherwise.
      auto IMapI = IsResultInvalidated.find(ID);
      if (IMapI != IsResultInvalidated.end())
        return IMapI->second;

      // Otherwise look up the result object.
      auto RI = Results.find({ID, &IR});
      assert(RI != Results.end() &&
             "Trying to invalidate a dependent result that isn't in the "
             "manager's cache is always an error, likely due to a stale result "
             "handle!");

      auto &Result = static_cast<ResultT &>(*RI->second->second);

      // Insert into the map whether the result should be invalidated and return
      // that. Note that we cannot reuse IMapI and must do a fresh insert here,
      // as calling invalidate could (recursively) insert things into the map,
      // making any iterator or reference invalid.
      bool Inserted;
      std::tie(IMapI, Inserted) =
          IsResultInvalidated.insert({ID, Result.invalidate(IR, PA, *this)});
      (void)Inserted;
      assert(Inserted && "Should not have already inserted this ID, likely "
                         "indicates a dependency cycle!");
      return IMapI->second;
    }

    Invalidator(SmallDenseMap<AnalysisKey *, bool, 8> &IsResultInvalidated,
                const AnalysisResultMapT &Results)
        : IsResultInvalidated(IsResultInvalidated), Results(Results) {}

    SmallDenseMap<AnalysisKey *, bool, 8> &IsResultInvalidated;
    const AnalysisResultMapT &Results;
  };

  /// Construct an empty analysis manager.
  ///
  /// If \p DebugLogging is true, we'll log our progress to llvm::dbgs().
  AnalysisManager(bool DebugLogging = false) : DebugLogging(DebugLogging) {}
  AnalysisManager(AnalysisManager &&) = default;
  AnalysisManager &operator=(AnalysisManager &&) = default;

  /// Returns true if the analysis manager has an empty results cache.
  bool empty() const {
    assert(AnalysisResults.empty() == AnalysisResultLists.empty() &&
           "The storage and index of analysis results disagree on how many "
           "there are!");
    return AnalysisResults.empty();
  }

  /// Clear any cached analysis results for a single unit of IR.
  ///
  /// This doesn't invalidate, but instead simply deletes, the relevant results.
  /// It is useful when the IR is being removed and we want to clear out all the
  /// memory pinned for it.
  void clear(IRUnitT &IR, llvm::StringRef Name) {
    if (DebugLogging)
      dbgs() << "Clearing all analysis results for: " << Name << "\n";

    auto ResultsListI = AnalysisResultLists.find(&IR);
    if (ResultsListI == AnalysisResultLists.end())
      return;
    // Delete the map entries that point into the results list.
    for (auto &IDAndResult : ResultsListI->second)
      AnalysisResults.erase({IDAndResult.first, &IR});

    // And actually destroy and erase the results associated with this IR.
    AnalysisResultLists.erase(ResultsListI);
```

```
  }

  /// Clear all analysis results cached by this AnalysisManager.
  ///
  /// Like \c clear(IRUnitT&), this doesn't invalidate the results; it simply
  /// deletes them.  This lets you clean up the AnalysisManager when the set of
  /// IR units itself has potentially changed, and thus we can't even look up a
  /// a result and invalidate/clear it directly.
  void clear() {
    AnalysisResults.clear();
    AnalysisResultLists.clear();
  }

  /// Get the result of an analysis pass for a given IR unit.
  ///
  /// Runs the analysis if a cached result is not available.
  template <typename PassT>
  typename PassT::Result &getResult(IRUnitT &IR, ExtraArgTs... ExtraArgs) {
    assert(AnalysisPasses.count(PassT::ID()) &&
           "This analysis pass was not registered prior to being queried");
    ResultConceptT &ResultConcept =
        getResultImpl(PassT::ID(), IR, ExtraArgs...);

    using ResultModelT =
        detail::AnalysisResultModel<IRUnitT, PassT, typename PassT::Result,
                                    PreservedAnalyses, Invalidator>;

    return static_cast<ResultModelT &>(ResultConcept).Result;
  }

  /// Get the cached result of an analysis pass for a given IR unit.
  ///
  /// This method never runs the analysis.
  ///
  /// \returns null if there is no cached result.
  template <typename PassT>
  typename PassT::Result *getCachedResult(IRUnitT &IR) const {
    assert(AnalysisPasses.count(PassT::ID()) &&
           "This analysis pass was not registered prior to being queried");

    ResultConceptT *ResultConcept = getCachedResultImpl(PassT::ID(), IR);
    if (!ResultConcept)
      return nullptr;

    using ResultModelT =
        detail::AnalysisResultModel<IRUnitT, PassT, typename PassT::Result,
                                    PreservedAnalyses, Invalidator>;

    return &static_cast<ResultModelT *>(ResultConcept)->Result;
  }

  /// Register an analysis pass with the manager.
  ///
  /// The parameter is a callable whose result is an analysis pass. This allows
  /// passing in a lambda to construct the analysis.
  ///
  /// The analysis type to register is the type returned by calling the \c
  /// PassBuilder argument. If that type has already been registered, then the
  /// argument will not be called and this function will return false.
  /// Otherwise, we register the analysis returned by calling \c PassBuilder(),
  /// and this function returns true.
  ///
  /// (Note: Although the return value of this function indicates whether or not
  /// an analysis was previously registered, there intentionally isn't a way to
  /// query this directly.  Instead, you should just register all the analyses
  /// you might want and let this class run them lazily.  This idiom lets us
  /// minimize the number of times we have to look up analyses in our
  /// hashtable.)
  template <typename PassBuilderT>
```

```cpp
  bool registerPass(PassBuilderT &&PassBuilder) {
    using PassT = decltype(PassBuilder());
    using PassModelT =
        detail::AnalysisPassModel<IRUnitT, PassT, PreservedAnalyses,
                                  Invalidator, ExtraArgTs...>;

    auto &PassPtr = AnalysisPasses[PassT::ID()];
    if (PassPtr)
      // Already registered this pass type!
      return false;

    // Construct a new model around the instance returned by the builder.
    PassPtr.reset(new PassModelT(PassBuilder()));
    return true;
  }

  /// Invalidate a specific analysis pass for an IR module.
  ///
  /// Note that the analysis result can disregard invalidation, if it determines
  /// it is in fact still valid.
  template <typename PassT> void invalidate(IRUnitT &IR) {
    assert(AnalysisPasses.count(PassT::ID()) &&
           "This analysis pass was not registered prior to being invalidated");
    invalidateImpl(PassT::ID(), IR);
  }

  /// Invalidate cached analyses for an IR unit.
  ///
  /// Walk through all of the analyses pertaining to this unit of IR and
  /// invalidate them, unless they are preserved by the PreservedAnalyses set.
  void invalidate(IRUnitT &IR, const PreservedAnalyses &PA) {
    // We're done if all analyses on this IR unit are preserved.
    if (PA.allAnalysesInSetPreserved<AllAnalysesOn<IRUnitT>>())
      return;

    if (DebugLogging)
      dbgs() << "Invalidating all non-preserved analyses for: " << IR.getName()
             << "\n";

    // Track whether each analysis's result is invalidated in
    // IsResultInvalidated.
    SmallDenseMap<AnalysisKey *, bool, 8> IsResultInvalidated;
    Invalidator Inv(IsResultInvalidated, AnalysisResults);
    AnalysisResultListT &ResultsList = AnalysisResultLists[&IR];
    for (auto &AnalysisResultPair : ResultsList) {
      // This is basically the same thing as Invalidator::invalidate, but we
      // can't call it here because we're operating on the type-erased result.
      // Moreover if we instead called invalidate() directly, it would do an
      // unnecessary look up in ResultsList.
      AnalysisKey *ID = AnalysisResultPair.first;
      auto &Result = *AnalysisResultPair.second;

      auto IMapI = IsResultInvalidated.find(ID);
      if (IMapI != IsResultInvalidated.end())
        // This result was already handled via the Invalidator.
        continue;

      // Try to invalidate the result, giving it the Invalidator so it can
      // recursively query for any dependencies it has and record the result.
      // Note that we cannot reuse 'IMapI' here or pre-insert the ID, as
      // Result.invalidate may insert things into the map, invalidating our
      // iterator.
      bool Inserted =
          IsResultInvalidated.insert({ID, Result.invalidate(IR, PA, Inv)})
              .second;
      (void)Inserted;
      assert(Inserted && "Should never have already inserted this ID, likely "
                        "indicates a cycle!");
    }
```

```
    // Now erase the results that were marked above as invalidated.
    if (!IsResultInvalidated.empty()) {
      for (auto I = ResultsList.begin(), E = ResultsList.end(); I != E;) {
        AnalysisKey *ID = I->first;
        if (!IsResultInvalidated.lookup(ID)) {
          ++I;
          continue;
        }

        if (DebugLogging)
          dbgs() << "Invalidating analysis: " << this->lookUpPass(ID).name()
                 << " on " << IR.getName() << "\n";

        I = ResultsList.erase(I);
        AnalysisResults.erase({ID, &IR});
      }
    }

    if (ResultsList.empty())
      AnalysisResultLists.erase(&IR);
  }

private:
  /// Look up a registered analysis pass.
  PassConceptT &lookUpPass(AnalysisKey *ID) {
    typename AnalysisPassMapT::iterator PI = AnalysisPasses.find(ID);
    assert(PI != AnalysisPasses.end() &&
           "Analysis passes must be registered prior to being queried!");
    return *PI->second;
  }

  /// Look up a registered analysis pass.
  const PassConceptT &lookUpPass(AnalysisKey *ID) const {
    typename AnalysisPassMapT::const_iterator PI = AnalysisPasses.find(ID);
    assert(PI != AnalysisPasses.end() &&
           "Analysis passes must be registered prior to being queried!");
    return *PI->second;
  }

  /// Get an analysis result, running the pass if necessary.
  ResultConceptT &getResultImpl(AnalysisKey *ID, IRUnitT &IR,
                                ExtraArgTs... ExtraArgs) {
    typename AnalysisResultMapT::iterator RI;
    bool Inserted;
    std::tie(RI, Inserted) = AnalysisResults.insert(std::make_pair(
        std::make_pair(ID, &IR), typename AnalysisResultListT::iterator()));

    // If we don't have a cached result for this function, look up the pass and
    // run it to produce a result, which we then add to the cache.
    if (Inserted) {
      auto &P = this->lookUpPass(ID);
      if (DebugLogging)
        dbgs() << "Running analysis: " << P.name() << " on " << IR.getName()
               << "\n";

      PassInstrumentation PI;
      if (ID != PassInstrumentationAnalysis::ID()) {
        PI = getResult<PassInstrumentationAnalysis>(IR, ExtraArgs...);
        PI.runBeforeAnalysis(P, IR);
      }

      AnalysisResultListT &ResultList = AnalysisResultLists[&IR];
      ResultList.emplace_back(ID, P.run(IR, *this, ExtraArgs...));

      PI.runAfterAnalysis(P, IR);

      // P.run may have inserted elements into AnalysisResults and invalidated
      // RI.
```

```
      RI = AnalysisResults.find({ID, &IR});
      assert(RI != AnalysisResults.end() && "we just inserted it!");

      RI->second = std::prev(ResultList.end());
    }

    return *RI->second->second;
  }

  /// Get a cached analysis result or return null.
  ResultConceptT *getCachedResultImpl(AnalysisKey *ID, IRUnitT &IR) const {
    typename AnalysisResultMapT::const_iterator RI =
        AnalysisResults.find({ID, &IR});
    return RI == AnalysisResults.end() ? nullptr : &*RI->second->second;
  }

  /// Invalidate a function pass result.
  void invalidateImpl(AnalysisKey *ID, IRUnitT &IR) {
    typename AnalysisResultMapT::iterator RI =
        AnalysisResults.find({ID, &IR});
    if (RI == AnalysisResults.end())
      return;

    if (DebugLogging)
      dbgs() << "Invalidating analysis: " << this->lookUpPass(ID).name()
             << " on " << IR.getName() << "\n";
    AnalysisResultLists[&IR].erase(RI->second);
    AnalysisResults.erase(RI);
  }

  /// Map type from module analysis pass ID to pass concept pointer.
  using AnalysisPassMapT =
      DenseMap<AnalysisKey *, std::unique_ptr<PassConceptT>>;

  /// Collection of module analysis passes, indexed by ID.
  AnalysisPassMapT AnalysisPasses;

  /// Map from function to a list of function analysis results.
  ///
  /// Provides linear time removal of all analysis results for a function and
  /// the ultimate storage for a particular cached analysis result.
  AnalysisResultListMapT AnalysisResultLists;

  /// Map from an analysis ID and function to a particular cached
  /// analysis result.
  AnalysisResultMapT AnalysisResults;

  /// Indicates whether we log to \c llvm::dbgs().
  bool DebugLogging;
};

extern template class AnalysisManager<Module>;

/// Convenience typedef for the Module analysis manager.
using ModuleAnalysisManager = AnalysisManager<Module>;

extern template class AnalysisManager<Function>;

/// Convenience typedef for the Function analysis manager.
using FunctionAnalysisManager = AnalysisManager<Function>;

/// An analysis over an "outer" IR unit that provides access to an
/// analysis manager over an "inner" IR unit.  The inner unit must be contained
/// in the outer unit.
///
/// For example, InnerAnalysisManagerProxy<FunctionAnalysisManager, Module> is
/// an analysis over Modules (the "outer" unit) that provides access to a
/// Function analysis manager.  The FunctionAnalysisManager is the "inner"
/// manager being proxied, and Functions are the "inner" unit.  The inner/outer
```

```
/// relationship is valid because each Function is contained in one Module.
///
/// If you're (transitively) within a pass manager for an IR unit U that
/// contains IR unit V, you should never use an analysis manager over V, except
/// via one of these proxies.
///
/// Note that the proxy's result is a move-only RAII object.  The validity of
/// the analyses in the inner analysis manager is tied to its lifetime.
template <typename AnalysisManagerT, typename IRUnitT, typename... ExtraArgTs>
class InnerAnalysisManagerProxy
    : public AnalysisInfoMixin<
          InnerAnalysisManagerProxy<AnalysisManagerT, IRUnitT>> {
public:
  class Result {
  public:
    explicit Result(AnalysisManagerT &InnerAM) : InnerAM(&InnerAM) {}

    Result(Result &&Arg) : InnerAM(std::move(Arg.InnerAM)) {
      // We have to null out the analysis manager in the moved-from state
      // because we are taking ownership of the responsibilty to clear the
      // analysis state.
      Arg.InnerAM = nullptr;
    }

    ~Result() {
      // InnerAM is cleared in a moved from state where there is nothing to do.
      if (!InnerAM)
        return;

      // Clear out the analysis manager if we're being destroyed -- it means we
      // didn't even see an invalidate call when we got invalidated.
      InnerAM->clear();
    }

    Result &operator=(Result &&RHS) {
      InnerAM = RHS.InnerAM;
      // We have to null out the analysis manager in the moved-from state
      // because we are taking ownership of the responsibilty to clear the
      // analysis state.
      RHS.InnerAM = nullptr;
      return *this;
    }

    /// Accessor for the analysis manager.
    AnalysisManagerT &getManager() { return *InnerAM; }

    /// Handler for invalidation of the outer IR unit, \c IRUnitT.
    ///
    /// If the proxy analysis itself is not preserved, we assume that the set of
    /// inner IR objects contained in IRUnit may have changed.  In this case,
    /// we have to call \c clear() on the inner analysis manager, as it may now
    /// have stale pointers to its inner IR objects.
    ///
    /// Regardless of whether the proxy analysis is marked as preserved, all of
    /// the analyses in the inner analysis manager are potentially invalidated
    /// based on the set of preserved analyses.
    bool invalidate(
        IRUnitT &IR, const PreservedAnalyses &PA,
        typename AnalysisManager<IRUnitT, ExtraArgTs...>::Invalidator &Inv);

  private:
    AnalysisManagerT *InnerAM;
  };

  explicit InnerAnalysisManagerProxy(AnalysisManagerT &InnerAM)
      : InnerAM(&InnerAM) {}

  /// Run the analysis pass and create our proxy result object.
  ///
```

```
  /// This doesn't do any interesting work; it is primarily used to insert our
  /// proxy result object into the outer analysis cache so that we can proxy
  /// invalidation to the inner analysis manager.
  Result run(IRUnitT &IR, AnalysisManager<IRUnitT, ExtraArgTs...> &AM,
             ExtraArgTs...) {
    return Result(*InnerAM);
  }

private:
  friend AnalysisInfoMixin<
      InnerAnalysisManagerProxy<AnalysisManagerT, IRUnitT>>;

  static AnalysisKey Key;

  AnalysisManagerT *InnerAM;
};

template <typename AnalysisManagerT, typename IRUnitT, typename... ExtraArgTs>
AnalysisKey
    InnerAnalysisManagerProxy<AnalysisManagerT, IRUnitT, ExtraArgTs...>::Key;

/// Provide the \c FunctionAnalysisManager to \c Module proxy.
using FunctionAnalysisManagerModuleProxy =
    InnerAnalysisManagerProxy<FunctionAnalysisManager, Module>;

/// Specialization of the invalidate method for the \c
/// FunctionAnalysisManagerModuleProxy's result.
template <>
bool FunctionAnalysisManagerModuleProxy::Result::invalidate(
    Module &M, const PreservedAnalyses &PA,
    ModuleAnalysisManager::Invalidator &Inv);

// Ensure the \c FunctionAnalysisManagerModuleProxy is provided as an extern
// template.
extern template class InnerAnalysisManagerProxy<FunctionAnalysisManager,
                                                Module>;

/// An analysis over an "inner" IR unit that provides access to an
/// analysis manager over a "outer" IR unit.  The inner unit must be contained
/// in the outer unit.
///
/// For example OuterAnalysisManagerProxy<ModuleAnalysisManager, Function> is an
/// analysis over Functions (the "inner" unit) which provides access to a Module
/// analysis manager.  The ModuleAnalysisManager is the "outer" manager being
/// proxied, and Modules are the "outer" IR unit.  The inner/outer relationship
/// is valid because each Function is contained in one Module.
///
/// This proxy only exposes the const interface of the outer analysis manager,
/// to indicate that you cannot cause an outer analysis to run from within an
/// inner pass.  Instead, you must rely on the \c getCachedResult API.
///
/// This proxy doesn't manage invalidation in any way -- that is handled by the
/// recursive return path of each layer of the pass manager.  A consequence of
/// this is the outer analyses may be stale.  We invalidate the outer analyses
/// only when we're done running passes over the inner IR units.
template <typename AnalysisManagerT, typename IRUnitT, typename... ExtraArgTs>
class OuterAnalysisManagerProxy
    : public AnalysisInfoMixin<
          OuterAnalysisManagerProxy<AnalysisManagerT, IRUnitT, ExtraArgTs...>> {
public:
  /// Result proxy object for \c OuterAnalysisManagerProxy.
  class Result {
  public:
    explicit Result(const AnalysisManagerT &AM) : AM(&AM) {}

    const AnalysisManagerT &getManager() const { return *AM; }

    /// When invalidation occurs, remove any registered invalidation events.
    bool invalidate(
```

```
        IRUnitT &IRUnit, const PreservedAnalyses &PA,
        typename AnalysisManager<IRUnitT, ExtraArgTs...>::Invalidator &Inv) {
      // Loop over the set of registered outer invalidation mappings and if any
      // of them map to an analysis that is now invalid, clear it out.
      SmallVector<AnalysisKey *, 4> DeadKeys;
      for (auto &KeyValuePair : OuterAnalysisInvalidationMap) {
        AnalysisKey *OuterID = KeyValuePair.first;
        auto &InnerIDs = KeyValuePair.second;
        InnerIDs.erase(llvm::remove_if(InnerIDs, [&](AnalysisKey *InnerID) {
          return Inv.invalidate(InnerID, IRUnit, PA); }),
                       InnerIDs.end());
        if (InnerIDs.empty())
          DeadKeys.push_back(OuterID);
      }

      for (auto OuterID : DeadKeys)
        OuterAnalysisInvalidationMap.erase(OuterID);

      // The proxy itself remains valid regardless of anything else.
      return false;
    }

    /// Register a deferred invalidation event for when the outer analysis
    /// manager processes its invalidations.
    template <typename OuterAnalysisT, typename InvalidatedAnalysisT>
    void registerOuterAnalysisInvalidation() {
      AnalysisKey *OuterID = OuterAnalysisT::ID();
      AnalysisKey *InvalidatedID = InvalidatedAnalysisT::ID();

      auto &InvalidatedIDList = OuterAnalysisInvalidationMap[OuterID];
      // Note, this is a linear scan. If we end up with large numbers of
      // analyses that all trigger invalidation on the same outer analysis,
      // this entire system should be changed to some other deterministic
      // data structure such as a `SetVector` of a pair of pointers.
      auto InvalidatedIt = std::find(InvalidatedIDList.begin(),
                                     InvalidatedIDList.end(), InvalidatedID);
      if (InvalidatedIt == InvalidatedIDList.end())
        InvalidatedIDList.push_back(InvalidatedID);
    }

    /// Access the map from outer analyses to deferred invalidation requiring
    /// analyses.
    const SmallDenseMap<AnalysisKey *, TinyPtrVector<AnalysisKey *>, 2> &
    getOuterInvalidations() const {
      return OuterAnalysisInvalidationMap;
    }

  private:
    const AnalysisManagerT *AM;

    /// A map from an outer analysis ID to the set of this IR-unit's analyses
    /// which need to be invalidated.
    SmallDenseMap<AnalysisKey *, TinyPtrVector<AnalysisKey *>, 2>
        OuterAnalysisInvalidationMap;
  };

  OuterAnalysisManagerProxy(const AnalysisManagerT &AM) : AM(&AM) {}

  /// Run the analysis pass and create our proxy result object.
  /// Nothing to see here, it just forwards the \c AM reference into the
  /// result.
  Result run(IRUnitT &, AnalysisManager<IRUnitT, ExtraArgTs...> &,
             ExtraArgTs...) {
    return Result(*AM);
  }

private:
  friend AnalysisInfoMixin<
      OuterAnalysisManagerProxy<AnalysisManagerT, IRUnitT, ExtraArgTs...>>;
```

```
  static AnalysisKey Key;

  const AnalysisManagerT *AM;
};

template <typename AnalysisManagerT, typename IRUnitT, typename... ExtraArgTs>
AnalysisKey
    OuterAnalysisManagerProxy<AnalysisManagerT, IRUnitT, ExtraArgTs...>::Key;

extern template class OuterAnalysisManagerProxy<ModuleAnalysisManager,
                                                Function>;
/// Provide the \c ModuleAnalysisManager to \c Function proxy.
using ModuleAnalysisManagerFunctionProxy =
    OuterAnalysisManagerProxy<ModuleAnalysisManager, Function>;

/// Trivial adaptor that maps from a module to its functions.
///
/// Designed to allow composition of a FunctionPass(Manager) and
/// a ModulePassManager, by running the FunctionPass(Manager) over every
/// function in the module.
///
/// Function passes run within this adaptor can rely on having exclusive access
/// to the function they are run over. They should not read or modify any other
/// functions! Other threads or systems may be manipulating other functions in
/// the module, and so their state should never be relied on.
/// FIXME: Make the above true for all of LLVM's actual passes, some still
/// violate this principle.
///
/// Function passes can also read the module containing the function, but they
/// should not modify that module outside of the use lists of various globals.
/// For example, a function pass is not permitted to add functions to the
/// module.
/// FIXME: Make the above true for all of LLVM's actual passes, some still
/// violate this principle.
///
/// Note that although function passes can access module analyses, module
/// analyses are not invalidated while the function passes are running, so they
/// may be stale.  Function analyses will not be stale.
template <typename FunctionPassT>
class ModuleToFunctionPassAdaptor
    : public PassInfoMixin<ModuleToFunctionPassAdaptor<FunctionPassT>> {
public:
  explicit ModuleToFunctionPassAdaptor(FunctionPassT Pass)
      : Pass(std::move(Pass)) {}

  /// Runs the function pass across every function in the module.
  PreservedAnalyses run(Module &M, ModuleAnalysisManager &AM) {
    FunctionAnalysisManager &FAM =
        AM.getResult<FunctionAnalysisManagerModuleProxy>(M).getManager();

    // Request PassInstrumentation from analysis manager, will use it to run
    // instrumenting callbacks for the passes later.
    PassInstrumentation PI = AM.getResult<PassInstrumentationAnalysis>(M);

    PreservedAnalyses PA = PreservedAnalyses::all();
    for (Function &F : M) {
      if (F.isDeclaration())
        continue;

      // Check the PassInstrumentation's BeforePass callbacks before running the
      // pass, skip its execution completely if asked to (callback returns
      // false).
      if (!PI.runBeforePass<Function>(Pass, F))
        continue;
      PreservedAnalyses PassPA = Pass.run(F, FAM);

      PI.runAfterPass(Pass, F);

```

```
      // We know that the function pass couldn't have invalidated any other
      // function's analyses (that's the contract of a function pass), so
      // directly handle the function analysis manager's invalidation here.
      FAM.invalidate(F, PassPA);

      // Then intersect the preserved set so that invalidation of module
      // analyses will eventually occur when the module pass completes.
      PA.intersect(std::move(PassPA));
    }

    // The FunctionAnalysisManagerModuleProxy is preserved because (we assume)
    // the function passes we ran didn't add or remove any functions.
    //
    // We also preserve all analyses on Functions, because we did all the
    // invalidation we needed to do above.
    PA.preserveSet<AllAnalysesOn<Function>>();
    PA.preserve<FunctionAnalysisManagerModuleProxy>();
    return PA;
  }

private:
  FunctionPassT Pass;
};

/// A function to deduce a function pass type and wrap it in the
/// templated adaptor.
template <typename FunctionPassT>
ModuleToFunctionPassAdaptor<FunctionPassT>
createModuleToFunctionPassAdaptor(FunctionPassT Pass) {
  return ModuleToFunctionPassAdaptor<FunctionPassT>(std::move(Pass));
}

/// A utility pass template to force an analysis result to be available.
///
/// If there are extra arguments at the pass's run level there may also be
/// extra arguments to the analysis manager's \c getResult routine. We can't
/// guess how to effectively map the arguments from one to the other, and so
/// this specialization just ignores them.
///
/// Specific patterns of run-method extra arguments and analysis manager extra
/// arguments will have to be defined as appropriate specializations.
template <typename AnalysisT, typename IRUnitT,
          typename AnalysisManagerT = AnalysisManager<IRUnitT>,
          typename... ExtraArgTs>
struct RequireAnalysisPass
    : PassInfoMixin<RequireAnalysisPass<AnalysisT, IRUnitT, AnalysisManagerT,
                                        ExtraArgTs...>> {
  /// Run this pass over some unit of IR.
  ///
  /// This pass can be run over any unit of IR and use any analysis manager
  /// provided they satisfy the basic API requirements. When this pass is
  /// created, these methods can be instantiated to satisfy whatever the
  /// context requires.
  PreservedAnalyses run(IRUnitT &Arg, AnalysisManagerT &AM,
                        ExtraArgTs &&... Args) {
    (void)AM.template getResult<AnalysisT>(Arg,
                                           std::forward<ExtraArgTs>(Args)...);

    return PreservedAnalyses::all();
  }
};

/// A no-op pass template which simply forces a specific analysis result
/// to be invalidated.
template <typename AnalysisT>
struct InvalidateAnalysisPass
    : PassInfoMixin<InvalidateAnalysisPass<AnalysisT>> {
  /// Run this pass over some unit of IR.
  ///
```

```
  /// This pass can be run over any unit of IR and use any analysis manager,
  /// provided they satisfy the basic API requirements. When this pass is
  /// created, these methods can be instantiated to satisfy whatever the
  /// context requires.
  template <typename IRUnitT, typename AnalysisManagerT, typename... ExtraArgTs>
  PreservedAnalyses run(IRUnitT &Arg, AnalysisManagerT &AM, ExtraArgTs &&...) {
    auto PA = PreservedAnalyses::all();
    PA.abandon<AnalysisT>();
    return PA;
  }
};

/// A utility pass that does nothing, but preserves no analyses.
///
/// Because this preserves no analyses, any analysis passes queried after this
/// pass runs will recompute fresh results.
struct InvalidateAllAnalysesPass : PassInfoMixin<InvalidateAllAnalysesPass> {
  /// Run this pass over some unit of IR.
  template <typename IRUnitT, typename AnalysisManagerT, typename... ExtraArgTs>
  PreservedAnalyses run(IRUnitT &, AnalysisManagerT &, ExtraArgTs &&...) {
    return PreservedAnalyses::none();
  }
};

/// A utility pass template that simply runs another pass multiple times.
///
/// This can be useful when debugging or testing passes. It also serves as an
/// example of how to extend the pass manager in ways beyond composition.
template <typename PassT>
class RepeatedPass : public PassInfoMixin<RepeatedPass<PassT>> {
public:
  RepeatedPass(int Count, PassT P) : Count(Count), P(std::move(P)) {}

  template <typename IRUnitT, typename AnalysisManagerT, typename... Ts>
  PreservedAnalyses run(IRUnitT &IR, AnalysisManagerT &AM, Ts &&... Args) {

    // Request PassInstrumentation from analysis manager, will use it to run
    // instrumenting callbacks for the passes later.
    // Here we use std::tuple wrapper over getResult which helps to extract
    // AnalysisManager's arguments out of the whole Args set.
    PassInstrumentation PI =
        detail::getAnalysisResult<PassInstrumentationAnalysis>(
            AM, IR, std::tuple<Ts...>(Args...));

    auto PA = PreservedAnalyses::all();
    for (int i = 0; i < Count; ++i) {
      // Check the PassInstrumentation's BeforePass callbacks before running the
      // pass, skip its execution completely if asked to (callback returns
      // false).
      if (!PI.runBeforePass<IRUnitT>(P, IR))
        continue;
      PA.intersect(P.run(IR, AM, std::forward<Ts>(Args)...));
      PI.runAfterPass(P, IR);
    }
    return PA;
  }

private:
  int Count;
  PassT P;
};

template <typename PassT>
RepeatedPass<PassT> createRepeatedPass(int Count, PassT P) {
  return RepeatedPass<PassT>(Count, std::move(P));
}

} // end namespace llvm
```

```
#endif // LLVM_IR_PASSMANAGER_H
```