Let’s actually create a log for this analysis manager stuff.

Sat Jul 23 21:05:12 PDT 2016

Did quite a bit of thinking today about different alternatives for how to design the analysis manager. The main thing is that adding the dependency tracking machinery leads to a ton of pointer chasing. For now, I think we’ll just have to live with that.

Another thing I’ve been thinking about is that inside getResult<AnalysisT> / getCachedResult<AnalysisT>, we actually already know the concrete analysis class (it is the template argument!).
So there’s no need for indirect calls along that path. AFAICT the only indirect calls that are needed are for destructing the analysis result objects: to invalidate “all” analysis results for an IRUnit we need to iterate a list of type-erased analysis results and destruct them.
Why not make `AnalysisT::run` just a static method? What state would the analysis usefully hold? Maybe some sort of configuration? E.g. an AAManager can hold information about which specific AA’s it should query and in what order.
Okay, I can buy that.
Oh, another thing is that having a class gives a clear “name” (the class name), which is extracted with getTypeName<T> in libSupport.
Okay, so I guess we *do* need to have an indirect call for getResult<AnalysisT>/getCachedResult<AnalysisT>. The indirection effectively represents a closure of the “configuration state” of the analysis.

Let’s do a quick systematic overview of the existing code.

I haven’t done a systematic “inhale” of the existing code yet. Let’s do that now.

It seems to reside primarily in:

- include/llvm/IR/PassManager.h
- include/llvm/IR/PassManagerInternal.h

Looking at r276542.

include/llvm/IR/PassManager.h

Okay, the beginning is the file header.

/// 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.

Looks like this is still trying to talk about analyses and transformations as both forming a common “pass” thing. The code says otherwise though (in PassManagerInternal.h you can clearly see the PassConcept, AnalysisResultConcept, AnalysisPassConcept base classes for the concept-based polymorphism stuff; PassConcept and AnalysisPassConcept are completely indepedent).

Yeah, nice links to the concept based polymorphism stuff. It’s a nice pattern.

Okay, #includes….

PreservedAnalyses class. Basically just a specialized set. It holds analysis ID’s with a special sentinel value representing “all” analysis ID’s.

PassInfoMixin class. Just a helper for transformations to define their name. Uses getTypeName<T>.

```
template <typename DerivedT>
struct AnalysisInfoMixin : PassInfoMixin<DerivedT> {
```

Interesting. AnalysisInfoMixing inherits from PassInfoMixin. Seems sort of weird, but I guess PassInfoMixin is pretty much just “ClassNameMixin” right now so whatever.

template <typename IRUnitT>

class PassManager : public PassInfoMixin<PassManager<IRUnitT>> {

Okay, so this is basically just a generic version of the “runOnFunction” method of “FPPassManager”. (there’s a separate “adaptor” for the “runOnModule” method fo FPPassManager).

It’s sort of weird how this DebugLogging is a member of PassManger and not a cl::opt. Otherwise every tool needs to have its own special logic for setting this (I’ve already been annoyed by this when testing out the new PM in LLD/ELF (--lto-newpm-passes=...)).

Interesting:

```
/// \brief A CRTP base used to implement analysis managers.
///
/// This class template serves as the boiler plate of an analysis manager. Any
/// analysis manager can be implemented on top of this base class. Any
/// implementation will be required to provide specific hooks:
///
/// - getResultImpl
/// - getCachedResultImpl
/// - invalidateImpl
///
/// The details of the call pattern are within.
///
/// Note that there is also a generic analysis manager template which implements
/// the above required functions along with common datastructures used for
/// managing analyses. This base class is factored so that if you need to
/// customize the handling of a specific IR unit, you can do so without
/// replicating *all* of the boilerplate.
template <typename DerivedT, typename IRUnitT> class AnalysisManagerBase {
```

Since the new analysis manager is unified for all IRUnitT’s (to centralize dependency tracking), I guess this CRTP base class isn’t needed? (edit: for simplicity, we probably want to keep it, but just remove the top-level IRUnitT templating)
Anyway, I guess this shows the main interface of an analysis manager. getResult, getCachedResult, and invalidate.
Oh, interesting. What is this `registerPass` stuff?

```
  /// \brief Register an analysis pass with the manager.
  ///
  /// The argument is a callable whose result is a pass. This allows passing in
  /// a lambda to construct the pass.
  ///
  /// The pass type registered is the result type of calling the argument. If
  /// that pass has already been registered, then the argument will not be
  /// called and this function will return false. Otherwise, the pass type
  /// becomes registered, with the instance provided by calling the argument
  /// once, and this function returns true.
  ///
  /// While this returns whether or not the pass type was already registered,
  /// there in't an independent way to query that as that would be prone to
  /// risky use when *querying* the analysis manager. Instead, the only
  /// supported use case is avoiding duplicate registry of an analysis. This
  /// interface also lends itself to minimizing the number of times we have to
  /// do lookups for analyses or construct complex passes only to throw them
  /// away.
  template <typename PassBuilderT> bool registerPass(PassBuilderT PassBuilder) {
    typedef decltype(PassBuilder()) PassT;
    typedef detail::AnalysisPassModel<IRUnitT, PassT> PassModelT;

    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;
  }
```

Oh, this explains that weird stuff that is needed when setting up to run a pass pipeline in the new PM. E.g. this stuff in opt (and there is corresponding code in lld for --lto-newpm-passes):

```
  // Register the AA manager first so that our version is the one used.
  FAM.registerPass([&] { return std::move(AA); });
```

Also I find it curious again that this uses the terminology “pass” to describe an analysis.

Seems sort of error-prone but whatever. I guess this registration is only needed for analyses that need “configuration” information passed to them (which right now seems to be just AAManager). Seems sort of weird because **this makes the AA pipeline effectively global to the analysis manager**. This also implies that **the AA pipeline is effectively global to the entire pass pipeline**.
It may make sense at some point to just make the AA pipeline a cl::opt or something set on the context or whatever. All this analysis registration and the indirection when calling getResult seems to be due entirely to the AA manager stuff. Other analyses don’t seem to need this (`run` can just be a static function on the analysis class).
(in fact, for AAManager I don’t think we even really need this.

One thing to notice is that AAManager looks sort of like a cross between PassManager and AnalysisManger (it keeps a linear sequence of things to run; but the things it runs are analyses; also, it doesn't do any caching itself).
I think that AAManager can just be something that transformations use as an on-stack helper class. E.g. a transformation pass just creates an object on the stack: `AAManager AAM(AM, Context->getAAPipeline());` which internally just calls `AM.getResult` as necessary to get handles to the analysis passes it needs to query.
)

As far as rewriting the analysis manager to handle dependencies (which is the issue at hand), we can just support this with this same somewhat strange behavior. It doesn't seem excessively difficult to support. We can change this later if it proves to be problematic.

Okay, now we're to the actual analysis manager. Let's pay special attention.

```
/// \brief A generic analysis pass manager with lazy running and caching of
/// results.
///
/// This analysis manager can be used for any IR unit where the address of the
/// IR unit sufficies as its identity. It manages the cache for a unit of IR via
/// the address of each unit of IR cached.
template <typename IRUnitT>
class AnalysisManager
    : public detail::AnalysisManagerBase<AnalysisManager<IRUnitT>, IRUnitT> {
```

Interesting mention of "address of the IR unit suffices as its identity". When wouldn't that be the case? Maybe this is what the "AnalysisManagerBase" thing was talking about customizing (when there is something other than address identity).
I guess in rewriting the analysis manager to be unified over the IRUnit's, we can just assume address identity. It would be annoying to support anything else (how would you have a generic map with type-erased IRUnit's as the key?)

Ugh, again this annoying DebugLogging thing. So many times I wish I could easily set this from LLD's -mllvm options…

Interesting:

```
  /// \brief Clear the analysis result cache.
  ///
  /// This routine allows cleaning up when the set of IR units itself has
  /// potentially changed, and thus we can't even look up a a result and
  /// invalidate it directly. Notably, this does *not* call invalidate functions
  /// as there is nothing to be done for them.
  void clear() {
    AnalysisResults.clear();
    AnalysisResultLists.clear();
  }
```

Looks like the only user of this is in InnerAnalysisManagerProxy, which is going away once we unify the IRUnit's as part of rewriting the analysis manager. It is probably useful to expose a method like this nonetheless?

So the core data structures are:

- AnalysisResultLists: A mapping IRUnit -> list of type erased analysis results for this IRUnit
- AnalysisResults: A mapping (IRUnit, AnalysisID) -> pointer into the list of analysis results for `IRUnit`. The element pointed to is the analysis result for AnalysisID. So this is basically just an “index” of the list entries in AnalysisResultLists.

When we cache an analysis result, we add it to the per-IRUnit list (inside AnalysisResultLists) and update the “index” mapping (AnalysisResults) to point at it.

Okay, so here are the 3 main methods: getResultImpl, getCachedResultImpl, invalidateImpl.

getCachedResultImpl just checks the “index” mapping and returns the result if it is present.

getResultImpl checks the “index” and returns the result if present. Otherwise, it looks up the AnalysisPassConcept (type-erased analysis) for the analysis ID being requested and calls its `run` method to compute an analysis result. It appends this analysis result to the corresponding list in AnalysisResultsLists and updates the “index” mapping to point at it.

Interesting, there are 2 overloads for invalidateImpl:

```
  void invalidateImpl(void *PassID, IRUnitT &IR) {

  PreservedAnalyses invalidateImpl(IRUnitT &IR, PreservedAnalyses PA) {
```

Also, one curious thing is that the PreservedAnalyses one is not implemented by calling the single-analysis one multiple times. I guess it makes sense. The single analysis one is a “blacklist” of analyses to invalidate (well, a trivial case of a blacklist: just one element). The PreservedAnalyses one iterates over the preserved analyses “whitelist” of analyses to preserve for this IRUnit; it just invalidates everything unless it is in the whitelist (and also subject to the “invalidate” callback; need to think more about that callback, it only seems to be used by the proxies (which are going away) and to indicate “don’t invalidate me”; is this callback actually useful?).

These all need to be modified to add dependency tracking.
First off, let’s consider invalidateImpl, which is the thing that actually needs the dependency information.
Also, let’s first consider the case of invalidating a single analysis on a single IRUnit.
The key issue is that we want to make sure to leave nothing potentially referencing this computed analysis result.
The most obvious situation are other analysis results holding pointers to it. To solve that, we need to keep, for each analysis result, a list of other analysis results that are (or might be) holding a pointer directly to it. Intuitively, we may keep a reference to the other analysis results (essentially, we need to track its dependents). I can think of a couple ways to do this:

1. An iterator to their element in the per-IRUnit list that owns them
    a. Problem: you can’t remove an element from a std::list without a handle to the actual std::list object. (i.e. just an iterator isn’t enough). The actual std::list object is problematic to hold a pointer to since it is held by value in the map so its address isn’t stable.
        i. Possible solution 1: add a layer of indirection to the map so that the std::list objects have stable addresses. Still, it seems annoying to keep a pointer to the std::list object (which is technically redundant; a doubly linked list element should be able to be removed without knowledge of the containing container).
        ii. Possible solution 2: use llvm::ilist
2. Some sort of pointer to the actual type-erased analysis result object
    a. Problem: you can’t find the actual element of the per-IRUnit list to be deleted.
3. A pair (IRUnit, AnalysisID).
    a. Problem: requires a lookup through the “index” map in order to access its element. Note however, that when we invalidate an analysis we need to update this mapping anyway (to delete the entry), so this might not actually require an extra lookup.

One thing to note is that this linking needs to be bidirectional. We can’t leave any of the dependency-tracking references dangling. For example (to quote myself from the thread `[PM] I think that the new PM needs to learn about inter-analysis dependencies…`):

```
One annoying problem is that I think that the dependency links will need to be bidirectional. To use the example analysis cache from my other post:
(AssumptionAnalysis, function @bar) -> (AssumptionCache for @bar, [(SomeModuleAnalysis, module TheModule)])
(AssumptionAnalysis, function @baz) -> (AssumptionCache for @baz, [(SomeModuleAnalysis, module TheModule)])
(SomeModuleAnalysis, module TheModule) -> (SomeModuleAnalysisResult for TheModule, [(SomeFunctionAnalysis, function @baz)])
(SomeFunctionAnalysis, function @baz) -> (SomeFunctionAnalysisResult for @baz, [])
```

```
if we delete function @baz, then the dependent list  [(SomeFunctionAnalysis, function @baz)] for `(SomeModuleAnalysis, module TheModule)` will now have a stale pointer to function @baz.
```

Even without considering deleting IRUnit's, another reason the bidirectional link is needed is simply to remove the entry in the "dependent" list of any dependencies to prevent that list from growing without bound. E.g. if we compute DominatorTree and LoopInfo (which depends on DominatorTree), then repeatedly invalidate LoopInfo followed by recomputing it (while keeping the same DominatorTree cached the whole time), then the "dependents" list of that DominatorTree analysis result will grow (we could use a set to mitigate this but that starts to seem like overkill).

If we go with using llvm::ilist for the per-IRUnit analysis result lists, then it seems like roughly speaking each element of the per-IRUnit lists need to contain a struct something like this (I'm not too familiar with llvm::ilist/iplist/etc (it seems fairly complex; I'll need to dig in and understand it), so bear with me):
```
struct PerIRUnitAnalysisResultListElement;
struct DependentTrackingNode : public ilist_node<DependentTrackingNode> {
  PerIRUnitAnalysisResultListElement *Dependent;
};
struct PerIRUnitAnalysisResultListElement : public ilist_node<Element> {
  std::unique_ptr<AnalysisResultConcept> AnalysisResult;
  ilist<DependentTrackingNode> Dependents;
  std::vector<DependentTrackingNode *> DependentTrackingNodesThatPointAtMe;
};
```

(
Just noticed: one other complexity here btw that I'll need to deal with is that AnalysisResultConcept is templated on IRUnitT. It needs to be type-erased for different IRUnitT. Ugh.
I'll think more about this when I look at PassManagerInternal.h
)

Notice that DependentTrackingNodesThatPointAtMe doesn't need to support insertion/deletion at arbitrary positions and so can be a vector. The reason is that all these pointers point into the `Dependents` list of a dependency. So if we ever are going to remove an element from DependentTrackingNodesThatPointAtMe, it means that a dependency was invalidated. Hence the "me" must have been invalidated first and so these pointers are gone (and so they can't dangle). On the other hand, the `Dependents` list may have elements removed from arbitrary positions since a dependent can be invalidated independently of its dependency.

All this linked list business is annoying. An alternative is to notice that all lists (even the per-IRUnit lists) are almost always pretty small. Each one can have at most one entry per analysis. And there aren't that many different analyses (but maybe enough that it isn't worth doing linear scans instead of just following a pointer).
(edit on the next day: this is totally wrong, we may have one dependency for each IRUnit; e.g. module analysis might have one function analysis result *for each function in the module*)

I think that's enough thinking about that problem for now. Tomorrow I can start sketching out the exact code for getResult/getCachedResult and invalidate. It should look mostly similar, but the dependency tracking needs to be added (as I described in `[PM] I think that the new PM needs to learn about inter-analysis dependencies…` I think all that is needed for this is to keep a stack of "analysis results we are currently inside getResult for" so that we can detect that the analysis manager has been re-entered and add a dependency)

Let's finish up PassManager.h

Okay, next are InnerAnalysisManagerProxy and OuterAnalysisManagerProxy. These aren't needed with a unified analysis manager for all IRUnitT's.

Next is ModuleToFunctionPassAdaptor. This is the most basic adaptor. It is just a module pass that iterates over functions, calling a single function transformation on them (use a Function PassManager to compose multiple transformations into a single transformation).

Finally are some helper/dummy/testing "transformations":

- RequireAnalysisPass<AnalysisT> which just calls getResult<AnalysisT>
- InvalidateAnalysisPass<AnalysisT> which just invalidates AnalysisT.
- InvalidateAllAnalysesPass which just invalidates all analyses

That's it for PassManager.h

include/llvm/IR/PassManagerInternal.h

Interesting that "None of these APIs should be used elsewhere". All the abstract base classes are in here. How are we going to do dynamic loading (e.g. from a DSO) of passes (i.e. transformations) without exposing this in some form?

Okay, so this file is basically the "concept-based polymorphism" internals (see the header comment in PassManager.h for links to info on this pattern).

There are 3 abstract base classes corresponding to 3 concepts:

- PassConcept: for transformations
- AnalysisPassConcept: for analyses
- AnalysisResultConcept: for analysis results

There are also the corresponding "model" classes which essentially are templates which wrap a type (which is "duck" typed to follow the concept) in an actual polymorphic object which inherits from the abstract base class corresponding to the concept. This is the core of the "concept-based polymorphism" pattern: the code you write is duck-typed and there is an external class which will polymorphically wrap it and expose that duck-typed code in a type-erased fashion.

There's also a ResultHasInvalidateMethod SFINAE helper to detect if an analysis class did or did not provide the optional "invalidate" callback.

Okay, so thinking about moving to a model where there is a single analysis manager for all IRUnitT's, the most obvious change is that AnalysisPassConcept and AnalysisResultConcept will need to have their IRUnitT type-erased. I think this will actually be fairly straightforward. I think that all we need to do is basically to change the IRUnitT argument to be `void *` or whatever. The corresponding change to AnalysisPassModel and AnalysisResultModel will require casting through the type-erased handle to call into the duck-typed object being wrapped; this isn't a problem because AnalysisPassModel and AnalysisResultModel are templated on IRUnitT and so know the right type to cast to.

Okay, that's it!

TODO for tomorrow:

[ ] Start sketching out the exact code for getResult/getCachedResult and invalidate
(one other random thing: as part of this analysis manager change, we probably want to add an AnalysisManager.h to keep the analysis manager code separate since it is actually unrelated to any particular "transformation running" scheme (at this point "pass manager" just seems like harmful terminology; with analyses split out, there is just "transformation running"; there is really

no “management” per se; the thing called “PassManager” in PassManager.h is really just “TransformationComposer”))

Sun Jul 24 12:23:11 PDT 2016

I was totally wrong about the thing that the dependency lists are inherently small. The thing I overlooked is that e.g. a module analysis result might depend on function analysis results *for each function in the module*.
The dependent lists can be large too in the reverse situation: function analysis results for each function all depend on a single module analysis.

(btw, thinking of porting more incrementally, here are approximately the analyses that need to be ported so that we can migrate the old PM to the new analysis manager. Looks like mostly backend stuff.

```
sean:~/pg/llvm % gg 'getAnalysis.*<[^>]*>' | grep -v -e WrapperPass -e 'rst' -e 'AssumptionCache' -e 'AnalysisT' -e 'FPass' | sed -e 's/^.*<\([^>]*\)>.*$/\1/' | sort | uniq -c | sort -n
   1
   1 DataLayout
   1 MachineFunctionAnalysis
   1 RegionInfoPass
   1 SpillPlacement
   2 LazyBlockFrequencyInfoPass
   2 LiveDebugVariables
   2 LiveRegMatrix
   2 PhysicalRegisterUsageInfo
   3 EdgeBundles
   3 MachineDominanceFrontier
   4 DivergenceAnalysis
   4 LiveStacks
   4 LiveVariables
   4 MachinePostDominatorTree
   4 MachineTraceMetrics
   4 StackProtector
```

```
 5 LoopAccessLegacyAnalysis
 5 TargetPassConfig
 5 VirtRegMap
 6 Pass
 7 SlotIndexes
 8 GCModuleInfo
11 MachineModuleInfo
12 MachineBranchProbabilityInfo
18 MachineBlockFrequencyInfo
25 LiveIntervals
32 MachineLoopInfo
38 MachineDominatorTree
```

Are any of these fundamentally not modelable in the new PM analysis management? Need to look into that.
)

Okay, so now that I think about it, adding the dependency management is a second step after making the analysis manager work for all IRUnit's. So let's look at what we need to do for that. Off the top of my head, I can think of:

1. Everywhere that uses AnalysisManager<T> needs to become just AnalysisManager (including typedefs like FunctionAnalysisManager).
2. Everywhere that uses proxies needs to be changed to not use proxies (and directly call getResult instead)
3. The concepts / models need to be made to take a `void*`.
4. AnalysisManagerBase needs:
    a. to not be templated on IRUnit
    b. the entry point methods do need to be.
5. AnalysisManager needs:
    a. to be made not templated on IRUnitT.
    b. The "impl" methods on AnalysisManager need to be made templated on IRUnit.
6. Is there an issue with registration? E.g. TargetLibraryInfo has a `run` method for both Function and Module. A simple solution is to define an integer ID for each different IRUnit and then incorporate that into the key used to look up the analysis pass. E.g. `static inline int getIRUnitID(Function &) { return 0; }`, `static inline int getIRUnitID(Module &) { return 1; }`, etc. (or it can return its address as a void * or whatever if that works on windows DLL builds (IIRC AnalysisInfoMixin was thwarted by this limitation))
7. Anything else?
8. Oh, we need to delete various things that are not needed (e.g. the proxies)

Ok, so what is the natural order for this? Are any of these steps separable?

- The Concept/Model changes can probably be done separately before everything else. (3.)
- Once that is done, we can probably make the existing analysis managers work for all IRUnit's, even though we have multiple of them and only use them for a specific IRUnit. (4., 5., 6.)
- Then we can move everything to canonically use just one of the analysis managers (e.g. just the "module analysis manager") as "the" single analysis manager
    - This will require making OuterAnalysisManagerProxy return a non-const manager temporarily, but that seems fine since the proxies are going away.
    - This will require touching the analysis registration.
    - This will require touching every new PM transformation and analysis.
- Then we can remove the IRUnit templating of AnalysisManager (and AnalysisManagerBase. (1., 2., 8.)
    - This will require changing the arguments of all the transformations and analyses.
    - This will require removing all the proxy boilerplate throughout the all the transformations and analyses.

Okay, that seems reasonable.

Mon Jul 25 16:34:23 PDT 2016

Hal had some good comments on the mailing list. Had a good back and forth.

Thinking a bit more, I really want to make O(all transformations) edits on only one step. So the last two steps from the plan I outlined yesterday should be merged.

Starting to implement patches for this.

Okay, switching the arguments of Analysis{Pass,Result}{Concept,Model} to void * went fine. Ugh, can't de-templatize AnalysisPassConcept because it takes an AnalysisManager<IRUnitT> as the argument to `invalidate`.

Lots of void* flying around. Need more typedefs.

Okay, so actually this has been slightly less painful than I anticipated. I currently have done 4.b 5.b and 6. And I hit this kind of error (all of the passes compile successfully! Just PassBuilder left.):

```
In file included from /home/sean/pg/llvm/include/llvm/Analysis/LazyCallGraph.h:49:0,
                 from /home/sean/pg/llvm/include/llvm/Analysis/CGSCCPassManager.h:24,
                 from /home/sean/pg/llvm/include/llvm/Passes/PassBuilder.h:19,
                 from /home/sean/pg/llvm/lib/Passes/PassBuilder.cpp:18:
/home/sean/pg/llvm/include/llvm/IR/PassManager.h:433:8: note: candidate:
template<class IRUnitT, class PassBuilderT> bool
llvm::detail::AnalysisManagerBase<DerivedT,
IRUnitTOnClass>::registerPass(PassBuilderT) [with IRUnitT = IRUnitT; PassBuilderT =
PassBuilderT; DerivedT = llvm::AnalysisManager<llvm::Loop>; IRUnitTOnClass =
llvm::Loop]
   bool registerPass(PassBuilderT PassBuilder) {
        ^
/home/sean/pg/llvm/include/llvm/IR/PassManager.h:433:8: note:   template argument
deduction/substitution failed:
/home/sean/pg/llvm/lib/Passes/PassBuilder.cpp:618:73: note:   couldn't deduce
template parameter ‘IRUnitT’
   LAM.registerPass([&] { return FunctionAnalysisManagerLoopProxy(FAM); });
```

The issue is that we can’t infer the IRUnit. I think we’ll have to pass it explicitly.
Note that we can’t even template metaprogram ourselves out of this one: the analysis class may have multiple “run” methods for different IRUnitT’s.

Okay, looks like this worked (and at least passed check-llvm)!

Tomorrow’s TODO:
[ ] eliminate the template argument to the AnalysisManager, the proxies, etc. which requires churning all the passes and analyses that have been ported so far.
(
Need to make sure to use sed for as much of this as possible.
Hopefully AnalysisManager<T>, FooAnalysisManager, and also AM/FAM/MAM/etc. can be handled by sed
Proxies will require special treatment though.
(Maybe there is a way to make source changes to the proxies so that they stay source-compatible as well for this next incremental step? Hmm..)

Just doing some numbers:

```
sean:~/pg/llvm % gg 'PreservedAnalyses.*::run' | wc -l
```

87
sean:~/pg/llvm % gg 'Result.*run(' | wc -l
44

So 131 total. As a ballpark estimate, assuming naively an 8 hour workday, that will require about 16 passes per hour, or about 1 pass every 4 minutes.

(edit: looks like my regex for analyses was missing some analyses)
)

Also, one thing I just thought about: currently, when a function PassManager (the new PM type) is running passes, the invalidation it does is completely constrained to the AnalysisManager<Function>. **So this means that, effectively, module passes can't be invalidated between running two function passes**. (the invalidation does eventually bubble up through the ModuleToFunctionPassAdaptor, but only after all function passes have been run)
This will may lead to some surprises when initially using a unified analysis manager (e.g. a quadratic behavior; now each function that a transformation pass visits might trigger an entire module analysis to re-run). **Keep an eye out for this**.
Hopefully by tomorrow you'll have things in a state where you can run test-suite with the LTO pipeline (will need to rebase my port-cgscc branch (or at least a copy of it… don't want to lose the original in this case) onto the analysis-manager branch which will be sort of annoying). That should hopefully flush out any O(functions in module) quadratic behaviors besides being an overall good integration test / sanity check.
We may be getting by simply because we seem to not have many module analyses:
```
sean:~/pg/llvm % gg 'run(Module.*AnalysisManager' | grep -v PreservedAnalyses
include/llvm/Analysis/CallGraph.h:  CallGraph run(Module &M, ModuleAnalysisManager &) { return
CallGraph(M); }
include/llvm/Analysis/GlobalsModRef.h:  GlobalsAAResult run(Module &M, AnalysisManager<Module> &AM);
include/llvm/Analysis/LazyCallGraph.h:  LazyCallGraph run(Module &M, ModuleAnalysisManager &) {
include/llvm/Analysis/ProfileSummaryInfo.h:  Result run(Module &M, ModuleAnalysisManager &);
include/llvm/Analysis/TargetLibraryInfo.h:  TargetLibraryInfo run(Module &M, ModuleAnalysisManager &);
include/llvm/IR/Verifier.h:  Result run(Module &M, ModuleAnalysisManager &);
lib/Analysis/GlobalsModRef.cpp:GlobalsAAResult GlobalsAA::run(Module &M, AnalysisManager<Module> &AM) {
lib/Passes/PassBuilder.cpp:  Result run(Module &, AnalysisManager<Module> &) { return Result(); }
lib/Transforms/IPO/ElimAvailExtern.cpp:EliminateAvailableExternallyPass::run(Module &M,
ModuleAnalysisManager &) {
lib/Transforms/IPO/FunctionAttrs.cpp:ReversePostOrderFunctionAttrsPass::run(Module &M,
AnalysisManager<Module> &AM) {
unittests/Analysis/CGSCCPassManagerTest.cpp:  Result run(Module &M, ModuleAnalysisManager &AM) {
unittests/IR/PassManagerTest.cpp:  Result run(Module &M, ModuleAnalysisManager &AM) {
```

GlobalsModRef is resistant to function updates IIRC. CallGraph is used by the inliner on my port-cgscc branch, but it is conservatively correct even as the functions change. LazyCallGraph is dead. TargetLibraryInfo is immutable. ProfileSummaryInfo is probably getting recomputed a lot (but it's not O(Module size)).

The others are just for testing purposes.
Okay, I can see how things haven't blown up.
If I rebase the port-cgscc branch though I will need to careful about the lifetime of the CallGraph used by the ModuleToPostOrderCGSCCAdaptor. (probably best is to just compute the call graph independently of the AnalysisManager). Chandler's http://reviews.llvm.org/D21464 will probably be updated as well.

My response in the mailing list thread has some more brainstorming on this topic. Hopefully I'll be able to get some empirical information about this problem soon (once the unified analysis manager is done).

Gah, just noticed a lot of the stuff above is totally wrong: when we call `AM.invalidate(IR, PA)` we currently only iterate a single IRUnit. So e.g. when we invalidate on a function, we won't invalidate on the module. Interesting. So we should still have the behavior of delaying invalidation until done with the adaptor even with the unified analysis manager.

Tue Jul 26 22:33:59 PDT 2016

Btw, one thing I noticed yesterday is that a couple unittests create analysis managers. Need to keep an eye on them when rebuilding.

Starting with:
```
sean:~/pg/llvm % gg -E
'(\bAnalysisManager<[^>]*>|\bFunctionAnalysisManager\b|\bModuleAnalysisManager\b|\bLoopA
nalysisManager\b|\bCGSCCAnalysisManager\b)'
```

Over 400 results :(

Will probably need to do another pass to clean up the proxy stuff.

Had to move the "PassManager" template down in the file since AnalysisManager is no longer dependent on its IRUnitT template argument and so we hit an error which would otherwise have been deferred.

opt links!
Let's see how `ninja check-llvm` goes...
Only llvm/unittests/IR/PassManagerTest.cpp fails!
It is this check:
```
  // Validate the analysis counters:
  //   first run over 3 functions, then module pass invalidates
  //   second run over 3 functions, nothing invalidates
  //   third run over 0 functions, but 1 function invalidated
  //   fourth run over 1 function
  EXPECT_EQ(7, FunctionAnalysisRuns);
```
We get 4 instead of 7
Interesting. It is because after the "first run", the module proxy was invalidating all the function analyses. So we need something analogous to that.
Okay, I'll implement that tomorrow.

Interesting… looking at InnerAnalysisManagerProxy:
```
    bool invalidate(IRUnitT &IR, const PreservedAnalyses &PA) {
      // If this proxy isn't marked as preserved, then we can't even invalidate
      // individual function analyses, there may be an invalid set of Function
      // objects in the cache making it impossible to incrementally preserve
      // them. Just clear the entire manager.
      if (!PA.preserved(InnerAnalysisManagerProxy::ID()))
        AM->clear();

      // Return false to indicate that this result is still a valid proxy.
      return false;
    }
```
Notice that we don't get fine-grained dependency tracking. It is just "clear everything". In fact, it seems that the current state of things is that a module transformation can't say that it preserves a function analysis.
This also means that a function transformation running on a single function will invalidate loop analyses for *all* loops.

I think this is what Chandler was talking about w.r.t. "downward invalidation" in https://reviews.llvm.org/D21921#inline-187608
I.e. that what is in trunk does not currently support it well.

I put some comments about a possible longer-term design on the mailing list. Essentially, the analysis manager needs to track parent-child relationships between IRUnit's (relationships that may not be present in the IR; e.g. after a module pass runs, we may still have analysis results cached on a function that has been deleted; so you can't just iterate the module to find all its child functions).
I described on the mailing list an approach based on the adaptors pushing/popping a stack representing the current "stack" of IRUnit's being visited.
E.g. the stack might contain at a given point in time something like `[Module foo, Function @bar]` or `[Module foo, SCC bar, Function @baz, Loop %qux]`

Thu Jul 28 00:05:52 PDT 2016

(really, the entry for Wed July 27).

Caught up with internal things today. No progress on the code.

One thing that I thought of earlier today was that the solution of keeping the stack of IRUnit's is not necessarily enough by itself. E.g. a module analysis might delete a function from the module, then create a new function. These might have the same address. So we would need the module pass to invalidate analyses cached on the function before deleting it.
We can maybe use a ValueHandle to get a notification. Otherwise, we just need to add this to the pile of restrictions on passes (can't delete inner IRUnit unless you notify the analysis manager).

Another thing that I was thinking of today was that a good way to indicate "categories" of analyses, which works using the existing "invalidate" callback.
Essentially, it would look something like this:
```
AnalysisCategories.h



inline bool invalidateMeIfIAmAModuleAnalysisThatCannotBeInvalidated(const PreservedAnalyses &PA) {
  return false;
}

extern char ModuleAnalysesThatCannotBeInvalidatedByCGSCCTransformations;
inline bool invalidateMeIfIAmAModuleAnalysisThatCannotBeInvalidatedByCGSCCTransformations(const
PreservedAnalyses &PA) {
  return PA.preserved((void*)ModuleAnalysesThatCannotBeInvalidatedByCGSCCTransformations) ||
invalidateMeIfIAmAModuleAnalysisThatCannotBeInvalidated(PA);
}


extern char ModuleAnalysesThatCannotBeInvalidatedByFunctionTransformations;
inline bool invalidateMeIfIAmAModuleAnalysisThatCannotBeInvalidatedByFunctionTransformations(const
PreservedAnalyses &PA) {
  return PA.preserved((void*)ModuleAnalysesThatCannotBeInvalidatedByFunctionTransformations) ||
invalidateMeIfIAmAModuleAnalysisThatCannotBeInvalidatedByCGSCCTransformations(PA);
}

extern char FunctionAnalysesThatOnlyDependOnTheCFG;
...


MyModuleAnalysis.h
...
// The way that MyAnalysis works means that it cannot be invalidated by function transformations.
struct MyAnalysis {
  ...
  bool invalidate(Module &, const PreservedAnalyses &PA) {
    return invalidateMeIfIAmAModuleAnalysisThatCannotBeInvalidatedByFunctionTransformations(PA);
  }
};
```
(The naming here is horrible but it is just for illustration purposes)
Essentially, if an analysis category is a “weaker” version of another one (i.e. less robust against invalidation), then it “inherits” from it by adding it into an || of preservation checking.

Tomorrow I will at least need to implement something to get the invalidation of inner IRUnit’s working again. One naive solution is to just walk *all* IRUnit’s that the analysis manager is tracking and call invalidate on them (if we make the IRUnitTKindID’s sequential from “largest” to “smallest” then we can avoid iterating “larger” ones, preserving the current behavior). The behavior with the proxies is extremely naive anyway and so implementing something slightly more naive probably isn’t a huge deal.
Once we have dependency tracking, we can properly evaluate solutions because we’ll be able to run real pipelines and measure stuff.
The beauty of cache invalidation is that you can always be conservative: it just costs you perf.

Thu Jul 28 14:18:48 PDT 2016

I think that one easy way to simulate the current proxy-based invalidation is to just define an enum:
```
enum IRUnitKind {
  IRK_Loop,
  IRK_Function,
  IRK_CGSCC,
  IRK_Module
};
```
Then we just walk the “index” mapping, whose keys are now AnalysisKey which contains an IRUnitKind, and invalidate any that have a IRUnitKindID smaller than the IRUnitKind of the IRUnit that is being invalidated.

I think this actually should have *identical* semantics to the current proxy-based invalidation. Actually to make it be identical we can’t just call “invalidate” on the inner IRUnit’s: we have to “hard invalidate” the entire result list (i.e. ignore the `invalidate` callback of the analysis results). This is analogous to how the proxies “clear” the inner analysis manager instead of just calling invalidate on it.

Another issue I just thought of: analyses that claim to be “immutable” cached on IRUnit’s that can be deleted.
E.g. TargetLibraryInfo which runs on Function.
Without using a CallbackVH or similar to notify the analysis manager of deletion, module transformations will be required to notify the analysis manager if they delete a function so that the analysis manager can “hard invalidate” any analysis results cached on it. I.e. this must override any signal from the analysis that it “shouldn’t be invalidated”.
Or, thought about in another way, does it even make sense to have “immutable” analyses that can’t be invalidated on any IRUnit other than the module itself? (the module obviously can’t be deleted)

Okay, I implemented the IRUnitKind enum. For now, I’m just doing functions, because it’s hard to cast back to SCC and Loop and reinvoke invalidateImpl because they’re defined in Analysis/ but the pass manager is in IR/.

So, here’s a neat issue. After fixing things to emulate the proxy invalidation, turns out the test was still failing. I get 9 instead of 7. Looking at the diff, it seems like the issue is that I accidentally just deleted:
PA.preserve<FunctionAnalysisManagerModuleProxy>();
When in reality I needed to replace it with:
PA.preserve<TestFunctionAnalysis>();
To get the same effect.

Great, now everything works with the unified analysis manager. I should probably add an indirect interface which allows handling Loop and SCC; one simple solution actually is to just store a functor in the getResult method alongside the analysis result object. Sort of wasteful of memory but should be pretty simple.

Fri Jul 29 13:31:19 PDT 2016

So, the “dumb” approach to downward invalidation I implemented yesterday has a fatal flaw, which is that if you have a function analysis that depends on a module analysis, it will trigger recursive invalidation.
I.e. you go to invalidate the function analysis, see the module analysis depends on it, then invalidate the module analysis, which then “downward” invalidates the function analysis recursively.

Once the dependency handling stuff is in place though (hopefully I’ll have a something working today), we can define for each IRUnitT a “getParentIRUnit” which gives the parent IRUnit. Then we can cache an analysis on that “parent” IRUnit whose invalidation (via the dependency management) invalidates the IRUnit.
This has a downside, which is that the parent is actually context-dependent.

The issue is that a function might be being visited as part of (just) a module visitation or as part of a CGSCC visitation. So there is no easy way to find the “right” parent given just the inner IRUnit.
This is equivalent to saying that there is only a single “adaptor” class with each IRUnit as its inner IRUnit. (E.g. the presence of the ModuleToFunction and CGSCCToFunction adapters means that there is no unique parent).
To implement this better, we need the adaptors to provide this information to the analysis manager.
This is conceptually what the proxies were sort of capturing already (notice that the adaptors were already calling the proxies).

Sat Jul 30 16:22:43 PDT 2016

Ugh, so even with dependency tracking and cooperation from the “outer to inner IRUnit” adaptors there is still not quite enough to support fine-grained downward invalidation of CGSCC analyses when the SCC’s themselves are lazily computed.
The fundamental problem is that you may have a CGSCC transformation that calls function analyses on functions in parent SCC’s. This means that if you are lazily computing the SCC’s like LazyCallGraph does (or any Tarjan-based algorithm) you end up calling a function analysis on a function whose “outer” SCC IRUnit has not even been created! Therefore there is no way to establish a dependency on it!

Note that this is precisely the situation that occurs in the inliner wanting to get BFI on callers of functions in the current SCC (done as part of its shouldDefer logic IIRC).

The only way I can think of to handle this case is for the actual SCC construction routine to call into the analysis manager to inform it “hey, we just formed an outer IRUnit that contains this inner IRUnit, so register a dependency”.

Nonetheless, for fine-grained downward invalidation, there is the problem of functions having both CGSCC and Module as the enclosing IRUnit in a context-dependent way.
I.e. `AM.getResult<FooAnalysis>(F)` might be being called by a function pass nested in a module to function adaptor or a CGSCC to function adaptor.
It is so much better if we can just statically find “the” parent IRUnit.

So one solution is to have a “traits” for each IRUnit that says how to get its “static” parent.
For functions, the traits would identify the module as the “static” parent IRUnit.
I think this is enough for dependency tracking to set up a dependency on a dummy analysis on the “static” parent.
For SCC’s, the SCC construction (and potentially modification, if we go with an approach like http://reviews.llvm.org/D21464) will be required to manually call into the analysis manager and add/edit dependencies between function analyses and a dummy analysis on the outer SCC.
This ends up with functions having a dependency on the module and their parent SCC, but that seems fine.
(presumably, the SCC IRUnit’s would also have a “static” parent link to the module so this is technically redundant).

Overall, I’m getting the feeling that we just want to step back, focus on a specific case that is currently blocked on the new PM (such as getting BFI in the inliner) and use an analysis manager in the old PM to solve this issue.
We can then see what steps are needed to use an analysis manager for all the analyses in the old PM.
If that is feasible, then we move to using an analysis manager for getting analyses in the old PM. We can probably do that transition incrementally (since the only cost would be duplicated analysis computation in the old PM’s analysis stuff and the analysis manager).
Eventually, we can eliminate the old PM’s analysis management. This will greatly simplify the old PM’s code and then we can evaluate more closely whether the “running transformation” parts of the old PM are actually bad enough that we should make a wholesale transition to the new PM, or whether we should just incrementally refactor the old PM.

Sun Jul 31 19:36:22 PDT 2016

Ugh, seems like ilist doesn’t have the ability to remove an element given just a pointer to the element.

Looks like we’re going to be stuck holding back pointers and need to heap-allocate the list headers.

In the end things are going to look something like:

```
struct PerIRUnitAnalysisResultListElement;
struct DependentTrackingNode {
  PerIRUnitAnalysisResultListElement *Dependent;
  std::list<DependentTrackingNode> &Backpointer; // So we can erase ourselves.
};
struct PerIRUnitAnalysisResultListElement {
  AnalysisKey AK;
  std::unique_ptr<AnalysisResultConcept> AnalysisResult;
  std::list<DependentTrackingNode> Dependents;
  std::vector<std::list<DependentTrackingNode>::iterator>
DependentTrackingNodesThatPointAtMe;
};
```

And the AnalysisResultLists lists will be std::list<PerIRUnitAnalysisResultListElement>.

Okay, check-llvm passes with the dependency tracking data structures in place.
Hopefully tomorrow I’ll be able to write the dependency walking logic.

Mon Aug 1 19:59:11 PDT 2016

One issue is that PreservedAnalyses is just a set of Analysis ID’s. Since it doesn’t provide information about the IRUnit, it doesn’t identify a specific analysis result.

The issue of “downward invalidation” can be handled fairly cleanly by making all analyses for an inner IRUnit (e.g. a function) depend on a dummy analysis on the parent IRUnit (e.g. the module).
The one place this falls over is with CGSCC. The first and simpler problem is that there is no way to go from a function to its containing SCC. No big deal, we can have the CGSCC to function adapter register this manually on the analysis manager.
The second and harder problem is that the usage pattern in the inliner (getting function analyses on callers) means that we are getting function analyses on functions that have likely not been found by the SCC walk yet. (actually, they provably haven’t been found yet since Tarjan’s algorithm discovers SCC’s bottom-up)
I see two solutions:

- Calculate all the SCC’s up front and cache them.
- Require the analysis manager to keep some sort of “pending functions that have not had their parent SCC assigned” list and lazily add the dependency links as SCC’s are discovered (or analyses are cached on them). That seems really annoying though.

Okay, I have the dependency tracking in place I think.
Tomorrow:

- Tests for the dependency tracking
- Implement the “get static parent” thing for IRUnit’s and get rid of the ugly hack.

(maybe rebase onto master too)

(also, what about “upward” invalidation?)

Just a note on correctness of downward invalidation via dependencies.
When you get a loop analysis, it registers a dependency on a dummy analysis on the function.
That dummy analysis on the function results in a dependency on a dummy analysis on the module.
The net result is that the downward invalidation will propagate transitively: if you invalidate the module dummy analysis, it will invalidate the function dummy analysis, which will invalidate the loop analysis.

Tue Aug 2 16:39:43 PDT 2016

Ugh, can't run realistic pipelines yet. Need to rebase my port-cgscc branch onto this analysis manager stuff.

Okay, have that rebased.

Looks like PR28400 is raising its head again. I think the culprit is scalar evolution this time.

```
sean:~/pg/llvm % gg -l AssertingVH | grep Analysis
include/llvm/Analysis/AliasSetTracker.h
include/llvm/Analysis/CGSCCPassManager.h
include/llvm/Analysis/CallGraph.h
include/llvm/Analysis/ScalarEvolution.h
include/llvm/Analysis/ScalarEvolutionExpander.h
lib/Analysis/LazyValueInfo.cpp
```

Yep, it was scalar evolution.

(need to invalidate after the loop pass manager too, since many loop passes do 'getCachedResult' on it; and you can't (currently) invalidate there because it is that annoying getCachedResult for outer IRUnit analyses thing:

```
sean:~/pg/llvm % gg -l 'getCachedResult<ScalarEvolution'
lib/Analysis/IVUsers.cpp
lib/Analysis/LoopAccessAnalysis.cpp
lib/Transforms/Scalar/IndVarSimplify.cpp
lib/Transforms/Scalar/LICM.cpp
lib/Transforms/Scalar/LoopDeletion.cpp
lib/Transforms/Scalar/LoopIdiomRecognize.cpp
lib/Transforms/Scalar/LoopRotation.cpp
lib/Transforms/Scalar/LoopStrengthReduce.cpp
lib/Transforms/Scalar/LoopUnrollPass.cpp
lib/Transforms/Utils/LCSSA.cpp
lib/Transforms/Utils/LoopSimplify.cpp
```

)

Needed to change loop-simplify,lcssa to lcssa,loop-simplify, because loop-simplify preseves LCSSA in the new PM, and this causes it to require LCSSA on the input.

Gah, the way the analysis manager code is structured, it is really difficult to define a “ParentIRUnitTrackingAnalysis”
Okay, managed to do it with a hack in lookupPass. (basically, we don’t want to have to register them all up front for all IRUnitT, since we might not have all of them available)
This should be correct.

With my changes, we are down to 3 crashed out of test-suite (building with the LTO pipeline in the new PM).
And they are two major types, which seem to be unrelated to the stuff I’m doing.

```
ld.lld: /home/sean/pg/llvm/lib/Analysis/BlockFrequencyInfoImpl.cpp:666: void
findIrreducibleHeaders(const llvm::BlockFrequencyInfoImplBase&, const
llvm::bfi_detail::IrreducibleGraph&, const std::vector<const
llvm::bfi_detail::IrreducibleGraph::IrrNode*>&,
llvm::BlockFrequencyInfoImplBase::LoopData::NodeList&,
llvm::BlockFrequencyInfoImplBase::LoopData::NodeList&): Assertion `Headers.size() >= 2 &&
"Expected irreducible CFG; -loop-info is likely invalid"' failed.
```

(this happens inside LoopVectorizePass
The stack trace for this is weird though. It looks like BFI is somehow getting called from getCachedResult<TargetLibraryInfo> which is bizarre.
)

```
ld.lld: /home/sean/pg/llvm/lib/Transforms/Utils/LoopSimplify.cpp:383: llvm::Loop*
separateNestedLoop(llvm::Loop*, llvm::BasicBlock*, llvm::DominatorTree*, llvm::LoopInfo*,
llvm::ScalarEvolution*, bool, llvm::AssumptionCache*): Assertion
`NewOuter->isRecursivelyLCSSAForm(*DT) && "LCSSA is broken after separating nested loops!"'
failed.
```

(this happens inside LoopSimplify, puzzlingly; and we ran lcssa just before it...)

For reference, this is with:

```
//Extra flags appended to CMAKE_C_FLAGS + CMAKE_CXX_FLAGS
TEST_SUITE_DIAGNOSE_FLAGS:STRING=-flto -O3

//Extra flags appended to CMAKE_EXE_LINKER_FLAGS
TEST_SUITE_DIAGNOSE_LINKER_FLAGS:STRING=-fuse-ld=/home/sean/pg/release-asan/bin/ld.lld -Xlinker
"--lto-newpm-passes=module(verify,globaldce,forceattrs,inferattrs,pgo-icall-prom,ipsccp,cgscc(function-att
rs),rpo-functionattrs,globalopt,function(mem2reg),constmerge,deadargelim,function(instcombine),require<pro
file-summary>,cgscc(inline,prune-eh),globalopt,globaldce,cgscc(argpromotion,function(instcombine,jump-thre
ading,sroa)),cgscc(function-attrs),function(lcssa,loop-simplify,require<scalar-evolution>,require<targetli
binfo>,require<aa>,loop(licm),invalidate<scalar-evolution>,mldst-motion,gvn,memcpyopt,dse,lcssa,loop-simpl
ify,require<scalar-evolution>,loop(indvars,loop-deletion),invalidate<scalar-evolution>,require<targetir>,r
equire<assumptions>,require<scalar-evolution>,loop(unroll),invalidate<scalar-evolution>,loop-vectorize,lcs
sa,loop-simplify,require<targetir>,require<assumptions>,require<scalar-evolution>,loop(unroll),invalidate<
scalar-evolution>,instcombine,simplify-cfg,sccp,instcombine,bdce,slp-vectorizer,instcombine,jump-threading
),cross-dso-cfi,lowertypetests,function(simplify-cfg),elim-avail-extern,globaldce,verify)" -Xlinker
"--lto-aa-pipeline=basic-aa,scoped-noalias-aa,type-based-aa,globals-aa"
```

(this is test-suite + SPEC cpu2006)

One thing I now notice: our pipeline contains:

```
loop(unroll),invalidate<scalar-evolution>,loop-vectorize,lcssa,loop-simplify
```

(and that is the only instance of loop-vectorize)
I have a feeling that this issue is due to `loop(unroll)` somehow screwing up loop-info.
Putting verify<loops> right before loop-vectorize doesn't signal any issues.
But putting invalidate<loops> there avoids the two BFI for LoopVectorize assertion.

I tried putting in `invalidate<loops>` immediately before every call to loop-simplify but it still can't get rid of that assertion.
I'll need to reduce that tomorrow. (and the BFI for LoopVectorize one too, hopefully).

(just did a bit of reduction; bugpoint does a really poor job on this; need to rebase onto master to get DannyB's improved bugpoint cfg reduction logic)
Filed: https://llvm.org/bugs/show_bug.cgi?id=28825

Wed Aug 3 13:13:03 PDT 2016

Commented on https://reviews.llvm.org/D21462

Commented on https://reviews.llvm.org/D23114

Spent most of the day helping Davide with bootloader debugging (w.r.t. Trying to debug why the LLD-linked FreeBSD kernel won't boot).

Rebased stuff onto master at least.

Really long discussion on IRC with Manuel about the new PM. Hopefully it is useful for visibility to the larger community.

Thu Aug 4 17:20:45 PDT 2016

Thinking a bit about the main deficiencies of the current analysis manager's design (well, besides the lack of dependency tracking), I think one of the biggest complexities comes from the fact that it does not have a "key" data structure that uniquely identifies an analysis result.
This leads to various issues, mainly related to **not being able to cleanly separate the operations on the type-erased side of the AM with the parts that are templated** (which conceptually are just thin veneers over the type-erased part).
(not to mention that AnalysisManagerBase CRTP base class which just adds an extra layer of complexity to the code without adding any functionality)

Another issue is that it has that weird analysis registration thing that is really only used by AAManager (and has dubious semantics; e.g. if you register twice, it just ignores the second one. Also, the construction of the analysis is global to the pipeline, so it isn't as flexible as it may seem at first).

Fri Aug 5 19:58:50 PDT 2016

Wtf, seems like there is a network outage at work 20:00-24:00? (writing this from home…)

Main thing I *wanted* to resolve today was to reduce the assertion failure:

```
ld.lld: /home/sean/pg/llvm/lib/Analysis/BlockFrequencyInfoImpl.cpp:666: void
findIrreducibleHeaders(const llvm::BlockFrequencyInfoImplBase&, const
llvm::bfi_detail::IrreducibleGraph&, const std::vector<const
llvm::bfi_detail::IrreducibleGraph::IrrNode*>&,
llvm::BlockFrequencyInfoImplBase::LoopData::NodeList&,
llvm::BlockFrequencyInfoImplBase::LoopData::NodeList&): Assertion `Headers.size() >= 2 &&
"Expected irreducible CFG; -loop-info is likely invalid"' failed.
```

Just need to remove some of the `invalidate<loops>` calls to bring it back.

Sat Aug 6 20:28:21 PDT 2016

Ugh, had to reopen https://llvm.org/bugs/show_bug.cgi?id=28825 as it seems the original test case still fails even after Michael Z's fix.

Okay, filed https://llvm.org/bugs/show_bug.cgi?id=28888 for the "Assertion `Headers.size() >= 2 && "Expected irreducible CFG; -loop-info is likely invalid"' failed" assertion failure.

I guess while I'm waiting on those bugs, tomorrow I can clean up the analysis manager patches and post them to phab.

Sun Aug 7 19:01:13 PDT 2016

While I'm waiting on PR28825 and PR28888, I guess I'll go ahead and factor out a branch with just the analysis manager stuff.

I can think of a couple things to do to clean this up:

- Use a void* as the IRUnitT kind ID.
    - This avoids hardcoding an enum of them.
- Don't use that hacky stuff for ParentIRUnitTrackingAnalysis.
    - The main thing that avoids is the need to "register" the ParentIRUnitTrackingAnalysis for each IRUnit when creating the analysis manager.
        - This is actually nontrivial because different IRUnit's are visible in different places. E.g. in the unittest for the PassManager, which is in IR, the Loop and CGSCC IRUnit's aren't visible (they live in Analysis).
    - Or maybe we can just put it in the .def file.
        - That way only the unittests will actually be affected by needing to do things manually, which seems sane.
- Write test cases.

Protip: when iterating on PassManager.h, try `ninja unittests/IR/IRTests` instead of `ninja opt`. This doesn't hit as much (e.g. it doesn't touch any Loop or CGSCC stuff), but can be useful when trying to get template stuff right.

Okay, the ParentIRUnitTrackingAnalysis cleanup is done.

As a first-order test case, I'll just add a regular FileCheck test of `-aa-pipeline=basic-aa -passes='require<aa>,invalidate<basic-aa>,gvn'`.
A more in-depth test checking for downward invalidation would be good to add too (i.e. don't invalidate analyses for *all* loops when a single function has analyses invalidated on it), but is lower priority.

Opened https://reviews.llvm.org/D23256

One thing that just came to me is that we need to make sure that when LoopInfo is invalidated, then all loop analyses are invalidated.
This can be done via an extension point like the getStaticIRUnitParent.
It can be something like
```
void registerExtraDependencies(Loop &L, AnalysisManager &AM) {
  AM.getCachedResult<LoopAnalysis>(*L.getHeader()->getParent());
}
```

We then call into this “register extra dependencies” hook right next to where we call getStaticIRUnitParent (while we have the analysis pushed onto the InFlightAnalysesStack so that the dependency gets registered).
The same can be done for CGSCC passes.
This extension point can just be a no-op for Function and Module.

Mon Aug 8 15:52:38 PDT 2016

David had a good suggestion for pulling the mechanical changes out of
https://reviews.llvm.org/D23256#508756

Unfortunately this requires dealing with O(all transformations) patches many times :(
Let’s get busy.

Okay, that is done. Updated now. That took a while.

So, an idea I got last night that I should probably write down is that for analysis-based IRUnit’s we need a dependency on the analysis that generates them.
Oh, right now the loop analyses all actually get LoopAnalysis. So they are already being invalidated when it is invalidated.
Thinking a bit more about this, I guess it is somewhat similar to the situation of Chandler wanting to pass the LazyCallGraph object itself as an argument to CGSCC analyses/transformations.
But in the case of dependency tracking, we would automatically have a dependency on the analysis that generates the IRUnit, which provides the analogous guarantee.
Therefore, we can always use e.g.
`AM.getCachedResult<LoopAnalysis>(*L.getHeader()->getParent())` to get the “correct” LoopInfo object that own the loop that this analysis result is cached on.