

Major work items

flatten header files.

The include file lack of structure is a major obstacle to being able to separate the FE from the rest of the compiler. In the existing environment, including cgraph.h for a few data structures results in bringing in a huge web of header files, including most of the back end.. And we're trying to not expose these to the front ends.

The idea here is to flatten all the header files such that no .h file includes other .h files. That way every source file includes every header file it needs. at the same time, any given .h file should only contain prototypes and data structures that are relevant to the appropriate .c file.

At this point, we can analyze what things are naturally needed by the different components, reorganize the locations of things that don't make sense, and rebuild natural component header files which include what is needed for that component. Its such a mess now that we don't actually know what dependencies we have and why things are needed.

Generally this involves replicating any given .h file's list of include files to all source files which include it, and manually going through the .h and .c file to match up prototypes. Any prototypes which are in the header which don't belong there are moved to an appropriate header file where it does belongs.

examples:

basic-block.h:

- <https://gcc.gnu.org/ml/gcc-patches/2014-10/msg01765.html>
- <https://gcc.gnu.org/ml/gcc-patches/2014-10/msg01954.html>
- <https://gcc.gnu.org/ml/gcc-patches/2014-10/msg02141.html>
- and finally <https://gcc.gnu.org/ml/gcc-patches/2014-10/msg02141.html>

cgraph.h: <https://gcc.gnu.org/ml/gcc-patches/2014-10/msg02570.html>

There are still numerous files which need this treatment, at least they are getting a little easier now.

Include file reduction..

With all the include files from each header copied into source files, each source file has acquired a growing list of includes, many of which are not actually needed to compile the file.

I currently remove any duplicates, and also trim out any newly created header files which contain prototypes which aren't used (as in the basic-block.h flattening series). Many of the system files are much trickier.

The original scheme was to automate include file trimming by trying to remove each header file, and see if the file compiles. If it compiles, remove the header. If it doesn't it's required.

The problem is situations with conditional compilation based on the existence of a macro:

header.h

```
#define FOO
```

file.c:

```
int func(int x)
{
    #ifdef FOO
        x = 2
    #endif
}
return x;
```

The existing tools will try to remove header.h from the include list of file.c, and since it still compiles, decide header.h isn't needed. There is a run time change that could go undetected for a long time... and we have a lot of such things with HAVE_insn macros and such. In particular tm.h causes lots of this.

The change will be to only do this when it's known to be safe. We need to generate a list for each file of what is imported and exported as macros.

- if the source file or other included files don't consume any of these, and still compiles, then it's OK to remove.
- if the source file DOES consume it at some point, but compiles anyway, flag it for a manual inspection.
- if the source file does consume it, and doesn't compile without it, then it's clearly a required header. (this is what the tools do right now)

The idea is to run this on the source base after all the files are flattened, but certainly serves a purpose to do it at any point.

This tool needs to be worked on.

Targethookify tm.h

The rtl backend has traditionally just supplied a list for macros for anything that the middle or front ends might care about. These are mostly exported through the file tm.h. We **really** need to formalize this.

<https://gcc.gnu.org/ml/gcc-patches/2014-09/msg01849.html> ,
<https://gcc.gnu.org/ml/gcc-patches/2014-09/msg01856.html> ,
<https://gcc.gnu.org/ml/gcc-patches/2014-09/msg02031.html> , and finally
<https://gcc.gnu.org/ml/gcc-patches/2014-09/msg02046.html>

basically we want to turn all these target macros into target hooks. each target macro is a piece of the backend which is exposed to other parts of the compiler.. and its currently uncontrolled and not well documented. Joseph Myers has been selectively converting a few, but we need to do them all.

The process is to take a macro, determine its inputs and outputs, document that, and turn it into a function call in targhooks.[ch]. And then do it for the rest. Main priority would be to convert the macros used by the front end so that we can break the requirement link between the FE source files and tm.h, and instead use a well documented target hook interface.

split tree.[ch]

in 2013, we split tree.h into tree.h which had all the tree accessing macros and such, and tree-core.h had all the basic data structures. The intent was to isolate the method of accessing the tree structure from the structure itself. Then during rearchitecture the new gimple wrappers can include tree-core.h and access the same data structure under the covers without exposing the original method of access.

we need to further expand this now by identifying and separating the components of tree.h which are used by

- a) the front end exclusively (tree-fe.[ch])
- b) the back end exclusively (tree-be.[ch])
- c) shared between FE and BE (tree.[ch])

When we replace all tree TYPE nodes in the backend, we won't need to touch anything in tree-fe.[ch].

We can change all occurrences in tree-be.[ch] with the new gimple type.

Any routines in tree.[ch] are common to both, so we copy the routine as it is into tree-fe.[ch] (since it will now be a front-end only routine). A new version which works on the gimple type would then be added into tree-be.[ch] for back end exclusive use.

This is how I imagine it at the moment, but this has not been brought forward yet. names may change, or some details in the exact approach.

Since we're only doing types first, we may decide it is easier to simply create tree-type.[ch] and gimple-type.[ch] and move a copy of each relevant thing out of tree.[ch] into both.. then convert tree-type.[ch] to the new gimple type. In fact, as I write this, this seems better already :-)

* FE interface

With all the include file flattening, there are still little bits and pieces of files which are required by the front end. For example, function.h and cgraph.h are required for compilation... but in both cases its a very small subset of each structure which is required... often just a half dozen fields.

Here my thoughts are to identify exactly what is needed by the front end by removing the includes and compiling. Then we create a front end interface version which contains just the bits the front end need. so for instance, we'd create function-fe.h which contains the small subset of struct function the front end uses in a class , and accessors/methods as required, and change the front end files.

Then in the backend version of function.h we'll have the full structure we have now, and it will include the function-fe.h structure as a component (or inherits if we c++ify it at the same time) , and adds to it the rest of the functionality.

This way we can control exactly what the front end has access to and this begins to form part of the "official" interface to the backend.

Gimple type wrapper - find and replace type uses in ME/BE

This is the post stage 1 work that will be a bit easier once the header files aren't bringing in the world.

Basically, using the gimple wrapper mechanism developed on the gimple-wrapper branch, we'll hunt down everywhere in the BE that tree TYPE nodes are used, and replace them with a gimple wrapper. This is basically everything from gimplification on.

Starting with the easy cases, replace all places the TYPE_ macros are used with a gimple wrapper access in functions. This will require changing the parameters to some of these functions, and tracing back to their callers modify the call locations. I plan to set up the wrappers at some point after stage 1 ends, and then start converting files one at a time on a branch. There will be significant ripple effects that will have to be dealt with here, especially on bits that are also used by the FE (since those still need to work on tree TYPEs.)

Identify and split other things that are shared between FE/ME

there are a lot of other routines and structures that are currently shared between the front and back ends. We need to identify these and in the current case of tree TYPE nodes, turn them into front end and back-end versions. This can be done iteratively as we push the gimple TYPE wrappers through the backend.

Remove re-gimplification

We also need to remove new tree creation from the backend. In particular, we didn't create any gimple build routines. Instead when we need a new gimple statement, we usually build the tree representation of what we want, and then call the gimplify routines to turn those trees into gimple statements.

Richard Biener has been working on merging tree and gimple folding, and as a part of this has been creating a set of gimple builders. We need to go through the compiler and in all the places where we create trees to build gimple, replace that sequence with calls to these build routines. I do not believe the full suite of builders has not been fully fleshed out, so there will be enhancements required to the gimple builders.

He would also like to see this fully fleshed out by converting one of the front ends to build gimple directly. There was a project a few years ago to so a gimple generating C (?) front-end, but at that time the facilities were insufficient and the project stopped. I believe he thinks the builders are good enough now, and has expressed an interest in supporting someone willing to tackle this.