

GCC Re-architecture II

This is a follow up document to the GCC modularization plan I produced last year. I've changed the original prototype into a preliminary working proposal and this will briefly describe the interface. There are also a multitude of decisions to be made.

The goals of the original plan are unchanged, this is now concerned mostly with practical implementation.

Note that gimple classes and methods are added only as they are encountered during conversion. Once conversion is complete, we have an exact list of methods required by the backend, and anything else in a tree node is front end only. As the interface evolved, adding new classes and methods for node types became more trivial, and no changes to the underlying implementation code should be required now. I believe it has reached the point where the base is flexible enough to proceed.

The GIMPLE interface

All gimple classes are defined in a gimple namespace. Originally I was hoping to use 'namespace gimple', but there were issues with gimple statements already using that name. I am currently using 'namespace G' since it is short to use and unique. I prefer 'namespace g' keeping the lowercase lettering, but that is currently being used for the global context variable. We should decide on a standard. I personally like 'namespace g' for gimple and 'namespace r' for rtl...

Tree-like implementation classes (tree-value-core.h, tree-value.[ch])

The overall base class (class tree_desc) effectively mirrors a tree_base node structure, and provides a c++ class implementing struct tree_base. Protected methods are provided which can be used by derived classes to do the runtime checks that are currently performed by the various TREE_CHECK macros and function calls. Class tree_desc also provides public methods to access the fields that are in every tree node base, such as used(), static() and side_effects().

The core gimple bases inherit from this base class. These form the basic objects that are not interchangeable with each other: type, value, and block. Others may be added as conversion progresses (perhaps binfo... just haven't encountered any other yet).

Derived classes are defined which provide the implementation of the more specific kinds of nodes, each containing methods to access that kind's information. The end result is a hierarchy of classes which are structured and provide all the same access that the current TREE_ macros provide.

All classes use a _desc suffix to identify that it is the descriptive version, and the pointer types are just the namet. So 'G::value_desc' is the class description, and 'G::value' is the pointer that replaces 'tree'

Smart pointer wrappers (gimple-wrapper.h)

In order for gimple classes to interact with trees, a simple pointer to a gimple object wont work. A `G::value` pointer is incompatible with a struct `tree *`. Furthermore, using a pointer locks us into a pointer interface, and it would be nice to be able to explore implementing the gimple IL as an index into a table. This would make streaming the IL at any point a simpler option.

In any case, there are 3 template classes.

- `_ptr<>` represents pointers to a base gimple object.. ie `G::value`, `G::type` and `G::block`
- `_dptr<>` represents pointers to classes which are derived from a base class. These classes have helpers for performing the `is_a`, `as_a`, and `dyn_cast` functionality as well.
- `_addr<>` for pointers to pointers. (ie `tree *`)

`_ptr<>` and `_dptr<>` have a set of methods which allow them to interact with trees. When passed to a function or expression which expects a tree, the gimple object is automatically reinterpreted as a tree. When constructed from a function or expression returning a tree, an automatic `TREE_CHECK` is performed to ensure the value is of the correct form. I tried and failed to make this a single class originally, but enough has changed that its worth another try again, but for the moment, they are classes still.

These are implemented by containing a pointer to a basic `tree_desc` object and re-interpreting that to the correct pointer when required. These objects are a single pointer in size and passed by value (like a pointer would be) this allows them to be constructed as required to interact with trees.

When the tree interface is no longer required, those methods can be removed, and the classes much simplified. In fact one could simply change the typedefs to be pointers to the gimple class description and everything should still work.

If we wish to experiment with table and indexes, one needs only modify the implementation of the class to keep a table index instead of a pointer, and supply the same routine names

`_addr<>` is provided for a pointer to pointer (ie, a `tree *`). Again its to allow a gimple object to interact seamlessly with a `tree *`. Since `_ptr<>` and `dptr<>` are passed by value, they cant really have their address taken. `addr<>` takes care of this by basically storing `typed tree_desc **`. methods are provided to convert back and forth to a `tree *` and assign/dereference a gimple object..

Again, once the tree interface is no longer required, this class can be eliminated and simply replaced with a typedef.

NULL_GIMPLE

A special class 'null_gimple' is introduced to replace NULL_TREE. This class can auto-convert to and from a tree, so it is compatible with all the tree routines. The smart pointer classes have methods with a null_gimple reference parameter for anywhere one might expect a NULL. I tried using an integer parameter for this so we could do something like NULL_TREE does, but ran into situations where the compiler found ambiguity between using the tree method or the integer method. In the end this seemed the cleanest

A static version of the class called NULL_GIMPLE is declared, and that is used throughout.

Unfortunately, this is one of the things that ripple into other files when converting. When a function returns a tree is converted to returns a G::value, there are places which won't implicitly convert.

Specifically the conditional operator. Both sides of the ':' have to explicitly be the same type, so

```
(cond) ? func_now_returning_gimple_value () : NULL_TREE
```

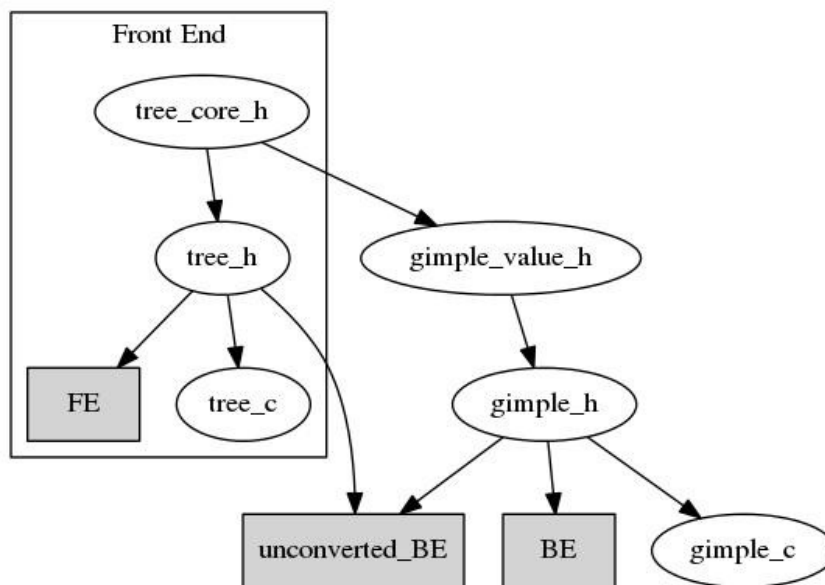
is a compile error. During builds, any files with this sort of paradigm need to have these expressions changed to use NULL_GIMPLE instead, ie

```
(cond) ? func_now_returning_gimple_value () : NULL_GIMPLE
```

It's a simple change, and the compiler will tell you everywhere you need to do it, but it trickles into other files I had hoped to avoid.

Header file separation

The header files are separated now such that any BE files converted to the new gimple interface do not see any definitions from tree.h or tree.c. Unconverted _BE files see both, so they are able to call any functions that have been converted, and convert to/from the required types.



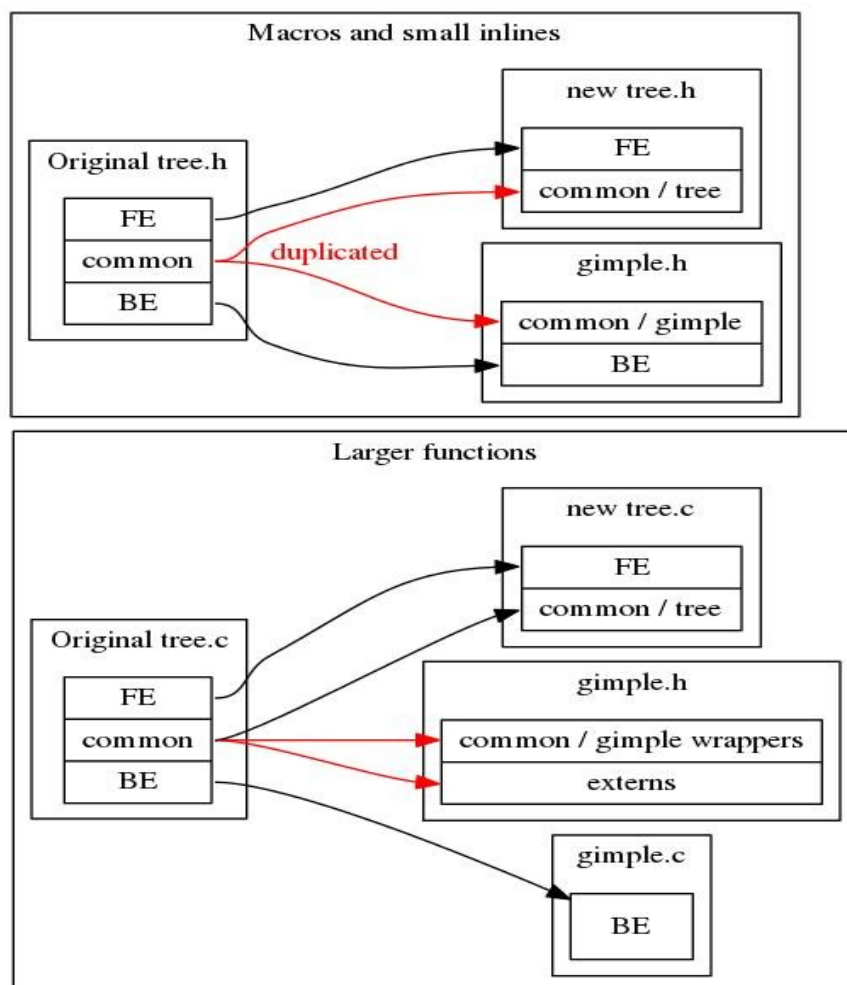
Gimple utility vs tree utility

Another large set of issues are the utilities and routines in tree.[ch] (and elsewhere) which the gimple interface also needs.

For the moment, I've created the files gimple-tree.[ch] in which I either re-implement the function as a gimple version, or simply create a wrapper which takes gimple parameters, and under the covers just calls the routine in tree.c

This is a placeholder approach until we sort out the right place to put these things, as well as how to deal with shared code for the duration of the conversion process.

I think we need to identify which of these routines are used in the front end exclusively, which are used in the back-end exclusively, and which are shared. I propose splitting these headers files up like so:



In the end, tree.c will have FE and common routines, gimple.c (or whatever name) would have all the BE only routines. This diagram doesn't show it, but I think maybe the common stuff should go into a 3rd file which the front and and back end can share, and provides both interfaces (or the gimple one provides wrappers). il_utils.[ch]?

Gimplification

Another sore spot which makes this process difficult is the re-gimplification of things from the back end. In theory, gimplify should be a gimple generator that takes trees as input and spits out gimple IL... and then be done..

Once things are converted to gimple, we **ought** to create gimple directly from within the backend. Unfortunately, that ability was never provided to do that easily, and with trees everywhere the common approach was to build the tree desired, and re-gimplify that.

I think we need to go through the middle/backend and replace all re-gimplifications with new gimple-building facilities as part of the conversion.. As long as we re-gimplify things, the middle/back end will need trees...

Future Direction

I'd like to arrive at some sort of consensus on whether this approach is something reasonable that we want to pursue and what kinds of changes are needed to make it acceptable.

Clearly performance, debuggability are issues that need to be addressed.

Currently, performance of the compiler is comparable on same code, although it tends to be just a slight bit slower (<1%) . I would expect it to be relatively neutral most of the time. The c++ aspects of it seem to introduce extra compile time, overall stage 3 compile time on my machine is about 10% slower. Presumably due to the new c++ templates and such. I have not investigated any of this in depth yet. (A few files do actually compile faster).

I still haven't added any debugging hook or utilities yet, so there isn't much there to discuss just yet

The regimplification removal work is something I would propose as an ongoing conversion using the new interface, creating the build utilities along the way using native gimple objects.

The splitting of tree routines into front-end, common, and backend should be better defined, and could be either a separate parallel project or could also be contained within the ongoing conversion. As functions are needed, determine if they are used by the front end or not and place them in an appropriate place.

I'd also propose that the current implementation be cleaned up to whatever requirements get set, and checked into mainline along with a carefully chosen couple of converted files. This will enable some day-to-day encountering of the code on a minimal basis, and any remaining issues with debugging or performance can be tackled without having committed large hunks of the compiler to it. I want to make sure this is usable before even thinking of throwing large hunks of the compiler into it.