TTYPE

Overview

This document will provide a brief rundown of my proposal to replace types in GCC with a new class which is distinct from the ubiquitous tree node pointer.. A new type for trees, or ‘ttype’.

The work is too invasive to develop and maintain on a branch, so the approach developed will allow the work to be done on trunk with a minimal amount of impact to the typical developer.

The general approach is to leave trunk in a “compatibility” mode where the new ttype is simply defined to be a tree node. This allows code using types to be changed to utilize ttype, but still compile and function exactly as it does today. Developers will start to see ttype in the code base, but are under no requirements to make things work with it, nor deal with compilation issues during the transition After all it is still just a tree.

A singular patch (ttype.patch) is maintained which switches from compatibility mode to a functional ttype mode. This patch provides an implementation of ttype where it is distinct from a tree, but has some compatibility to facilitate conversion of the code base. When converting a file to use the new ttype class the ttype.h header file is included. The developer proceeds to fix the compilation errors and eventually achieve a full bootstrap/testsuite run with the patch applied. The ttype patch is then unapplied, and a bootstrap/testsuite run is confirmed, and those changes can then be checked into trunk.

The idea being that in this way all the files in the compiler can be converted a few at a time and checked into trunk. Once all the files have been converted, we can replace the ttype compatibility class from the patch with a truly distinct ttype class which is incompatible with a tree, and the switchover is complete.

During this development phase, there is no requirement for a typical trunk developer to ensure existing ttype modifications compile with any new changes. Anyone doing the conversion work would be responsible for fixing up any incompatible changes that have been introduced by others as they encounter them. This is typically not much effort. As enough files are converted we can revisit the situation and during stage 1 we may reach some point where we want to leave the patch applied during development.. We’ll see.

The conversion process itself is not complicated. The vast majority of changes involve changing ‘tree’ to ‘ttype *’. Much of this would be suitable for a basic gcc level knowledge.

Technical Approach

The fundamental approach can be viewed conceptually as identifying all the places in GCC that refer to the tree type structure, and replace them with a pointer directly to that type field rather than accessing it through the tree union. Ie, instead of

void set_type_quals (tree, int)

The converted file would have

*void set_type_quals (ttype *, int)*

Unfortunately it is not quite that simple. There are a number of technical hurdles and a few resulting warts that have to be addressed. The changes are vast, and cannot all be done at once. To that end there is a transition period during which the new type is compatible with a tree, allowing files that have not been converted to still work properly with files that have not been converted. When all files have been converted, the compatibility link can be broken.

Until then, the new ttype is a derived class from a tree_node. This allows a ttype to be used like it is a tree in all the accessor macros and functions.

The first wart shows up in parameter passing. GCC defines a tree as a pointer to a tree_node. We desire to expose this pointer for various reasons (proper constness being a big one), and as such you see ‘ttype *’ as the parameter. Unfortunately, C++ does not consider pointers to classes to have any inheritable properties, and so a tree and a ttype * are NOT compatible. To resolve this, we use a couple of macros for parameters when they are to be shared between converted and unconverted code. Ie:

#define ttype_p tree

*#define ttype_pp tree **

And the above function would look something like:

void set_type_quals (ttype_p, int)

During conversion of a file to the new ttype standard, ttype_p is instead defined as a compatibility class which looks something like:

```
class ttype_p {
  ttype *type;
public:
  inline ttype_p () { type = NULL; }
  inline ttype_p (tree t) { type = TTYPE (t); }
  inline ttype_p (const_tree t) { type = const_cast<ttype *> (TTYPE (t)); }
  inline ttype_p (ttype *t) { type = t; }
  inline ttype_p& operator= (ttype *t) { type = t; return *this; }
  inline ttype_p& operator= (const ttype_p &t) { type = t.type; return *this; }
```

```
  inline operator ttype *() const { return type; }
  ttype_p * operator &() const __attribute__((error("Don't take address of ttype_p ")));
  inline ttype * operator->() { return type; }
  inline ttype * operator->() const { return type; }
};
```

During conversion this will allow function to be called from places using either the new ttype * or the old tree pointer (a ttype_p can be created from either), but any uses of ttype_p within the function can only be treated as ttype * as there is only a conversion operator TO a ttype *. The TTYPE() function used in the conversion code simply does a TYPE_CHECK on the tree to ensure it it in fact a type.

Interestingly, the same effort is not needed by the return type of a function, it is only during the evaluation of parameters that this is an issue. All the various use cases for the return value simply work when it is a ttype pointer.

A second wart show up occasionally when a there is unplanned interaction between a converted function used by an unconverted one that can't be altered as yet, ie, vec_safe_push. A few macros are provided which allow explicit casts to be made and can be easily identified and fixed later during conversion. These are things like

```
#define TREE_CAST(NODE) ((tree)(NODE))
#define TREE_PTR_CAST(NODE) ((tree *)(NODE))
```

None of the warts are usually that big a deal, and will all go away eventually.

A new file ttype.h is provided which is included by any source file which is converted to the new approach. Its primary function is to undefine any of the TYPE_CHECK macros from tree.h and replaces them with inline functions that accept only ttype * parameters. Ie:

```
#undef FUNC_OR_METHOD_CHECK
inline ttype *FUNC_OR_METHOD_CHECK (ttype *t)
{  return TTYPE (TREE_CHECK2 (t, FUNCTION_TYPE, METHOD_TYPE)); }
inline const ttype *FUNC_OR_METHOD_CHECK (const ttype *t)
{  return TTYPE (TREE_CHECK2 (t, FUNCTION_TYPE, METHOD_TYPE)); }
```

This means that any attempts to access fields from a type node will trigger a compilation error if the type being accessed through hasn't been declared as ttype *. When conversion of all files is complete, this will be changed to a more appropriate and efficient implementation.

ttype.h also provides new overloaded functions for accessing common tree fields which are currently shared between the tree structure and the type structure, ie:

```
inline struct tree_base& TREE_BASE (ttype *t)
{  return t->u.type_common.common.typed.base;}

inline const struct tree_base& TREE_BASE (const ttype* t)
{ return t->u.type_common.common.typed.base; }
```

The conversion process consist of including ttype.h at the top your source file, then proceeding to compile the file and replace parameters and local variables with the new type and resolve whatever other compilation errors occur. There are time when a function deals with both types and trees, and those are usually the most work. Generally they are split into 2 functions, one taking a tree and one a ttype *. Other common things encountered are replacing NULL_TREE with NULL.

When the day arrives that ttype has proliferated into all the source files, there will be some additional work, but the primary effort would involve removing the compatibility layer completely so that ttype is its own type and is never converted to or from a tree node. The TYPE macros can be removed from tree.h and replaced with the new functions from ttype.h

Trunk Preparation

It is highly desirable to minimize the size and impact of the ttype compatibility patch. There are a variety of generic tweak to trunk that will help facilitate that without impacting code generation or performance. These changes are supplied as a series of independent patches which are applied before any sort of ttype work is done. A quick rundown of what each patch contains follows:

01-bugfixes

The first patch contains a couple of minor bugfixes which were uncovered during the ttype protoyping. No big deal, they just should be applied. Most have already been applied since they were brought up, but I think there is one remaining.

02-avoid-direct-tree-access

This patch removes any direct tree accesses within gcc source files. It adds a few new macros to tree.h to access fields which do not have access macros (code use to directly access the field only), or in some cases utilizes an existing macro which was not being used in the source file. After application of this patch, the code base only accesses tree fields through accessors. This

is important in order to ensure fields aren't being accessed in an uncontrolled way as the underlying structure eventually changes. ITs also a good gneeral cleanup.

03-unify-tree-code-chain-type

This patch alters where TREE_CHAIN and TREE_TYPE are defined in tree.h. Currently they have alternate definitions depending on whether ENABLE_TREE_CHECKING is defined or not. It turns out that this isn't really necessary and this simplifies it.

04-flatten-struct-tree-type

There are currently 3 possible type structures within a tree. They are type_common, type_non_common, and type_with_lang_specific. It probably seemed like a good idea once upon a time, but it turns out that we **always** allocate the largest one (type_non_common). Any type specific information is always checked via the various CHECK macros during access to the fields, so there is no real benefit to segregating the 3 structures. This patch consolidates the type structures down to just type_common, and changes the various places in the compiler that refer to the other ones.

05-tree-base-chain-reference-inlines

This patch is the first structural re-arrangement. The most basic accessors which are shared between all trees and the type structure are converted to simple inline functions rather than macros. This is the TREE_CHAIN macro, as well as the new TREE_BASE and CONTAINS_STRUCT_CHECK functions.

Accessor macros which use the base tree structure use to directly access it, ie

```
#define TREE_ADDRESSABLE(NODE) ((NODE)->base.addressable_flag)
```

Now they will use the new TREE_BASE function wrapper:

```
inline struct tree_base&  TREE_BASE (tree t)  {return t->base;  }
#define TREE_ADDRESSABLE(NODE) (TREE_BASE (NODE).addressable_flag)
```

The reason for this change is when compiling for ttype, the TREE_BASE accessors is overloaded to also accept a ttype * as a parameter, and will eventually allow different checking/processing when ttype is passed in. This is very important for when the layout of the structure is eventually different than that of a tree.

```
inline struct tree_base& TREE_BASE (ttype *t)
                { return t->u.type_common.common.typed.base; }
```

06-access-thru-tree-type

Eventually trees and types will not share anything, and this includes the tree_code field. This patch lays the groundwork for this by changing the TREE_CHECK macros generated by gencheck.c to generate a new TYPE_CODE_CHECK macro for any tree code which is a tcc_type.

All the accessor macros in tree.h are then changed to use these new CHECK macros, and in the case where we use to check multiple tree codes, replace them with new unified CHECKs.. *Ie:*

```
#define TYPE_SATURATING(NODE) \
!   (TREE_NOT_CHECK4 (NODE, RECORD_TYPE, UNION_TYPE, QUAL_UNION_TYPE,
ARRAY_TYPE)->base.u.bits.saturating_flag)
```

Is replaced with

```
#define SATURATING_TYPE_CHECK(T) \
+   TREE_NOT_CHECK4 (T, RECORD_TYPE, UNION_TYPE, QUAL_UNION_TYPE,
ARRAY_TYPE)
<...>
#define TYPE_SATURATING(NODE) \
!   (SATURATING_TYPE_CHECK (NODE) \
!     ->type_common.common.typed.base.u.bits.saturating_flag)
```

It doesn't seem like it does much, but the ttype.h conversion patch replaces all the various TYPE_CHECK macros with inline functions that accept only 'ttype * . This then triggers a compilation error whenever any type field is accessed by anything other than 'ttype *'.

07-struct-tree-node

The last of the general changes is core to being able to implement ttype with a class. A union cannot be used as a base class, meaning inheritance won't work to allow compatibility between ttype and trees. In order to facilitate this, the current union tree_node is renamed to union tree_node_u and a new struct tree_node is created which contains this union. Ie:

```
union GTY ((desc ("tree_node_structure (&%h)"), variable_size)) tree_node_u {
<... existing struct tree_node...>
};
struct GTY ((ptr_alias (union lang_tree_node))) tree_node {
  union tree_node_u u;
};
```

And all accessors in tree,h, and the other odd place union_node is referred to directly are changed to access the union through the struct, so

#define TREE_OPERAND_CHECK(T, I) ((T)->exp.operands[I])
Changes to
#define TREE_OPERAND_CHECK(T, I) ((T)->u.exp.operands[I])

This then allows for an initial definition of ttype:
class GTY(()) ttype : public tree_node { };

Which allows ttype to be compatible with trees in unconverted code, but allow us to differentiate between tree and ttype * in function overloading and such.

That ends the basic changes that are unrelated to ttype and which help facilitate making the ttype compatibility patch fairly small and subject to very little churn by mainline changes. From the original branch which was mothballed in may of 2016 to my update to trunk in Feb 2017, there was only one line in the patch which had to be changed.. And that was for a new field in tree.h.

The TTYPE patchsets

The initial ttype patch set changes a few core things in the compiler over to use the new type, and tis then followed by the more straightforward conversion patches. It is broken up into logical changes in an attempt to make reviewing a little easier. As such, some patches introduce a bit of “wart code” which follow on patches then undo. This also provides a good example of what the warts are, when they occur, and how they are eventually fixed.

08-tree-basetypes

This patch converts the basic builtin types for the backend and various front ends to utilize ttype * directly. That means things like integer_type_node is now defined as a ‘ttype *’

This requires splitting the global_trees[] vector into global_trees[] and global_types[] (in GCC and all the front ends), altering the enums used by those vectors to be distinct enums, and changing numerous functions which manipulate or set these vectors and types to return the new type instead of a tree.

In order to limit the scope of these patches (so they don't proliferate to too many unrelated files), compatibility macros are introduced in a few places, (notably langhook functionality) to prevent the patch growing tremendously.

At the same time, tree.c is given the “full conversion” to include ttype.h. There were so many manipulation and affected functions that the decision was made to avoid using any compatibility macros and just convert it up front.

The other thing of note in this patch is a bit of hackery in c-format.c. Many places in gcc print trees with %q or a few other codes. As pointers are not compatible, trying to print a ‘ttype *’ when a ‘tree’ is expected doesn’t work out of the box. I originally tried adding a new code for types, but it turned out that the amount of change that triggered was amazingly prolific and resulted in a lot of unrelated code base churn. Instead I punted and hacked up the format code to check for and allow ttype * when a tree was expected. This could easily be revisited later when time allows, or we are getting ready to segregate the types completely. Ideally it would be a separate phase/patchset when the time comes to convert all %codes for types to a new one. Better to find all the places first.

ttype

At the same time, we introduce the ttype.patch itself. This is the patch which toggles the code between ttype actually being a defined as a tree for 100% compatibility, and it being defined as a separate class so we can find all the places that need changing. This is triggered by altering coretypes.h:

```
  typedef const struct tree_node *const_tree;

  /* These declarations are part of the effort to split types from trees.  */
! typedef struct tree_node ttype;
! #define ttype_p ttype *
! #define ttype_pp ttype **

  struct gimple;
  typedef gimple *gimple_seq;
--- 84,92 ----
  typedef const struct tree_node *const_tree;

  /* These declarations are part of the effort to split types from trees.  */
! class ttype;
! class ttype_p;
! class ttype_pp;

  struct gimple;
  typedef gimple *gimple_seq;
```

Any structure which contains a type has a parallel structure created in the tree node union which actually refers to a ‘ttype *’ rather than a tree in the appropriate places. For instance, we have 2 versions of the decl_minimal structure:

```
struct GTY(()) tree_decl_minimal {
  struct tree_common common;
  location_t locus;
```

```
  unsigned int uid;
  tree name;
  tree context;
};
struct GTY(()) ttype_decl_minimal {
  struct tree_common common;
  location_t locus;
  unsigned int uid;
  tree name;
  ttype *context;
};
```

Both are elements of the tree_node union. The ttype accessors and macros will pick up the ttype_decl_minimal variant of the structure in order to access the context field, yielding the proper ttype * field. Note this definition is only used during conversion. The code sitting on trunk without the patch applied still access this as a tree, so we don't raise any spectres of type aliasing.

Tree.h is modified to have all the TYPE_CHECK macros redefined to be an inline function call accepting only 'ttype *'.. Meaning the compiler will generate a compilation errors if one tries to access a type field with tree variable… it won't be able to find a matching function call.

Those are the major aspects… A few other odds and ends are included (such as reworking walk_tree) , but applying this patch allows us to find all the places in a file which use a type, and then execute and test the code ,and bootstrap the compiler. By default, it would not be applied on the trunk, but maintained so it can be applied or unapplied. (it is actually checked into the working branch as ttype.patch)

09-langhooks

This patch changes the langhook table to utilize ttype as appropriate. This converts every langhook function which deals with types to pass in a ttype_p and/or return a ttype *. All the front ends and ports that provide overrides are also changed so their functions conform.

10-attributes

This patch is a pain in the butt. Attributes are basically set up by initializing a vector of structures, where each instance of the structure indicates

- what an attribute is,
- its characteristics,
- indicators for whether it was appropriate for decls, types, or both,
- and a handler to implement the attribute.

This was the most work of any conversion, simply because decls and types were used interchangeably throughout the code (and why not, they were just trees). I had to split each handler into 2 handlers... A handler for decls (which took a tree as a parameter) and one for types (which takes a ttype * as a parameter). When working on splitting each of the handlers, it became obvious that in many cases the port writers didn't actually understand the relationship between all the fields and how the handler was called.. There were cases which could only be called for types which contained code to handle cases for decls, and vice versa.

The end result is a much cleaner and clearer separation of the decl and type code, but as the patch adds a field to the attribute structure, it is super sensitive to any change anyone makes to the attributes. Ie, if someone changes a single entry in an attribute table, that entire piece of the patch doesn't apply. Maintaining this patch has always been the most unpleasant part of this work :-)

The attribs.c file was given the full ttype conversion since ttype was pretty significant to this file :-)

And with that, the primary underlying core of ttype has been implemented, and now it moves on to converting files. I chose to start with some of the front ends files since they tend to be consumer oriented and actually contained some “trickier” parts.

11-c-typeck-cfamily-c-common

This patch “ttypifies” c-family/c-common.c and c/c-typeck.c. These 2 files tended to have a lot of type related code in them and seemed like a good logical next step. I seem to recall they has the most “compatibility” macros being used from the initial conversion, so I chose these to eliminate those warts.

Its worth noting that although I indicated earlier that parameters to function used the ttype_p macro, this is only needed when the function is external and can be called by an unconverted file using a tree parameter If you don’t do this, the compiler will give you a warning when linking and you fix it by using ttype_p.

On the other hand, if a function is static and thus local to a file, then it can be immediately converted to using ttype * in parameters. By converting the entire file, all places that call it will be converted to using ttype and there is no need for the compatibility provided by ttype_p.
Ie from c-typeck:

```
********** addr_space_superset (addr_space_t as1, a
*** 315,322 ****
 /* Return a variant of TYPE which has all the type qualifiers of LIKE
    as well as those of TYPE.  */
```

```
! static tree
! qualify_type (tree type, tree like)
  {
    addr_space_t as_type = TYPE_ADDR_SPACE (type);
    addr_space_t as_like = TYPE_ADDR_SPACE (like);
--- 317,324 ----
  /* Return a variant of TYPE which has all the type qualifiers of LIKE
     as well as those of TYPE.  */

! static ttype *
! qualify_type (ttype *type, ttype *like)
  {
    addr_space_t as_type = TYPE_ADDR_SPACE (type);
    addr_space_t as_like = TYPE_ADDR_SPACE (like);
}
```

12-c-ada-spec.c,13-cilk, 14-cfamily, 15-c-pretty-print, 16-array-notation, 17-c-convert

These patches are simply conversions of a bunch of .c files to utilize ttype, demonstrating the straightforwardness of conversions. Many files are only a couple of changes, and some require none whatsoever. They demonstrates the generally low impact of the ttype conversion.

Moving forward

That is the high level view of the ttype proposal. There are clearly a number of small technical hurdles which are not brought to the forefront here, but most of those fall into the “compatibility warts” I mentioned earlier. I will provide a more detailed writeup of the individual issues in each of the patch writeups of the submission. I have avoided doing that until now as there has been so much flux in this project over the past two years that everything I wrote kept getting thrown out as it was replaced with something different.

This is simply meant to give an overview of the general approach so there is some context for the writeup in each patch.

Some of the advantages of moving forward with this work is:

- Provide compile time type safety for types. Converting only a few files found potential future bugs that compile time safety can detect which the existing runtime checks only find when the correct code path is followed.
- No runtime checks should produce better and smaller code.
- A single type structure will eliminate various aliasing issues that may happen with tree pointers.
- Extracting types from trees will allow us to replace the type structure in the compiler with something more efficient for gimple containing just the parts we actually need. We will know what fields are actually used, and how frequently. We can also use a different representation for RTL if desired during gimple->rtl conversion.
- This could simplify streaming types for LTO, and well as making the type merging simpler and more efficient. We may even be able to leverage the current LTO type representation into a new gimple representation.
- When completed, we can replace the macros with method calls, if desired. This would improve debuggability.
- When completed, the same model can be used to replace DECLs in the compiler… or other tree components. This would leave trees to represent mostly expressions, which is what they are well suited to.

Current State

Currently, the patches have been applied to the branch ttype-2017. The branch revisions are as follows:

245316 Initial codebase trunk Feb 9/2017, this revision includes a fix.
245317 01-bugfixes
245321 02-avoid-direct-tree-access
245322 03-unify-tree-code-chain-type
245323 04-flatten-struct-tree-type
245329 05-tree-base-chain-reference-inlines
245330 06-access-thru-tree-type
245333 07-struct-tree-node ** I think all config targets worked.

245402 08-tree-basetypes ** debug testsuite failures
245403 ttype.patch
245517 09-langhooks ** most failures went away.. Except self-test
245541 10-attributes ** lots of test failures.have to Run all targets too
245554 11-c-typeck-cfamily-c-common

The changes above the line are the ttype independent changes which should go into trunk

The rest are the patches ported from the orignal “ttype-trial” which was fullt bootstrapping with no failures. I did not spend the time to fix all the little bugs, it was more a quick-update-proof-of-concept for this document. I think most of the changes after 10-attributes are possibly better off simply re-converted than porting. A number of parser things changes and it would be easy to miss something. 10-attributes also need to be audited and testsuite failures fixed.

The original fully tested branch revisions for the patches in ttype-trial are:
234726 inital branch apr 4/2016
234733 01-bugfixes
234734 02-avoid-direct-tree-access
234735 03-unify-tree-code-chain-type
234736 04-flatten-struct-tree-type
234745 05-tree-base-chain-reference-inlines
234747 06-access-thru-tree-type
234751 07-struct-tree-node * config-list

234778 08-tree-basetypes
234785 09-langhooks * config-list successfully passed

234791 10-attributes * config-list successful
234793 fix up 2 TREE_TTYPE uses from 10-attributes. .patch file fixed
234795 11-c-typeck-cfamily-c-common
234796 12-c-ada-spec.c
234797 13-cilk
234798 14-cfamily
234799 15-c-pretty-print
234800 16-array-notation
234807 17-c-convert * config-list successful, with and without ttype patch applied

234878 ttype-patch add the conversion patch so it doesnt get lost.