Reprinted from the

Proceedings of the GCC Developers' Summit

June 2nd–4th, 2004
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Eric Christopher, *Red Hat, Inc.*
Janis Johnson, *IBM*
Toshi Morita, *Renesas Technologies*
Zack Weinberg, *CodeSourcery*
Al Stone, *Hewlett-Packard*
Richard Henderson, *Red Hat, Inc.*
Andrew Hutton, *Steamballoon, Inc.*
Gerald Pfeifer, *SuSE, GmbH*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.

Statically Typed Trees in GCC

Nathan Sidwell
CodeSourcery, LLC
nathan@codesourcery.com

Zachary Weinberg
CodeSourcery, LLC
zack@codesourcery.com

Abstract

The current abstract syntax tree of GCC uses a dynamically typed über-union for nearly all nodes. The desire for a statically typed tree design has been raised several times over recent years, but there has been no concerted effort to implement such a design. We describe the impacts of the current design, both in implementation and performance degradation. We present a design for statically typed trees, along with case studies of part of the conversion. We outline a plan for full conversion and discuss further improvements that this would enable.

1 Current architecture

GCC uses a data structure called a **tree** for its high-level intermediate representation. The parser and semantic analyzer for a given programming language construct an initial tree representation of the program to be compiled. The high-level optimizers work directly on this tree. After they are done, the "expander" converts the optimized tree to a lower-level representation called **RTL** for further optimization and assembly output. We will not be discussing RTL in this paper, but it is worth mentioning that many of the same issues also apply.

A tree structure is a directed graph of **nodes**. Each node is a block of memory (a C `struct`) on the heap; the graph edges are pointers between these blocks. Tree nodes are dynamically typed. All variables and structure fields pointing to tree nodes have the type `tree`, which can address any node no matter what its internal structure is. To access the data carried in a node, one must use the macros defined in `tree.h`. These hide the exact representation and can be configured to carry out consistency checks at runtime (of GCC). We discuss the in-memory representation and the accessor macros in more detail below.

The **code** of a tree node determines its dynamic type. The generic (language independent) portion of the compiler defines approximately 150 codes. Front ends can define additional codes if necessary. There are ten **classes** (conceptual categories) of tree codes; each has a tag character to identify it. Front ends cannot define new classes. Presently, the classes are

- **`'c'`**, constants
- **`'1'`**, unary arithmetic operators
- **`'2'`**, binary arithmetic operators
- **`'<'`**, comparison operators
- **`'r'`**, references (e.g. array indexing)
- **`'e'`**, other expressions (e.g. `?:`)
- **`'s'`**, statements
- **`'d'`**, declarations
- **`'t'`**, types
- **`'x'`**, miscellaneous

Here are some example tree nodes, with the information they carry:

`STRING_CST` (class "constant")
A string constant. The node holds a

pointer to a separately-allocated byte array, and the length of this array.

PLUS_EXPR ("binary expression")
An addition operation. The node holds pointers to tree nodes representing the two addends.

IF_STMT ("statement")
An `if` statement. The node holds pointers to tree nodes representing the controlling expression, the "then" clause, and the "else" clause.

VAR_DECL ("declaration")
A declaration of a variable. The node is the root of a directed graph of nodes which collectively describe the properties of the variable.

INTEGER_TYPE ("type")
A description of an integer data type, either intrinsic to the programming language or defined on the fly by the program being compiled. Again, the node is the root of a directed graph describing the properties of the type.

TREE_LIST ("miscellaneous")
A linked list of other trees. Each node of the list can point to up to three different trees (known as the *type*, *purpose*, and *value*); however, usually only one of these slots is used.

ERROR_MARK ("miscellaneous")
A placeholder used when an error is encountered during compilation. This node carries no information. The compiler allocates only one `ERROR_MARK` node per invocation.

Trees exhibit three levels of polymorphism, which we will refer to as **substructure**, **multipurposing**, and **overloading**.

1.1 Substructure

The `tree` type is a pointer to a `union` of `struct`s. We will call these structs "substructures."

```
union tree_node
{
  struct tree_common common;
  struct tree_type type;
  struct tree_decl decl;
  struct tree_list list;
  ...
};
typedef union tree_node *tree;
```

All tree nodes include the fields of `struct tree_common`.[1] Most nodes also carry additional information stored in one of the other substructures. The tree code, which is a field of the common substructure, determines which substructure is active.

We can therefore categorize tree structures according to which substructure is valid. This categorization is similar, but not identical, to the categorization into classes. Front ends can also define new substructures, if necessary. Unfortunately the mechanism for this is somewhat awkward, since there is no way in C to augment the contents of a union.

Naturally, accessing the wrong substructure of a node can have grave consequences. To prevent this, GCC can be configured so that the accessor macros inspect the tree code and verify that they have been applied to the proper kind of tree. These checks are partially ad-hoc and partially machine-generated. The code is only known when the compiler is running, so the checks perforce must occur then. If one fails, GCC halts translation with the infamous "internal compiler error" (ICE) message.[2]

[1] because all the other substructures include `struct tree_common` as their first member.

[2] Jeff Law added the checking mechanism in 1998.

Accessor	Used with	Content
TYPE_VALUES	ENUMERAL_TYPE	A list of CONST_DECLs, one for each enumeration constant.
TYPE_DOMAIN	SET_TYPE, ARRAY_TYPE	An integer type whose range determines the set of all valid indexes of this type.
TYPE_FIELDS	RECORD_TYPE, UNION_TYPE	A list of FIELD_DECLs, one for each data member of the type.
TYPE_ARG_TYPES	FUNCTION_TYPE, METHOD_TYPE	A list giving the type of each parameter, in order, to the function or method.
TYPE_DEBUG_ REPRESENTATION_ TYPE	VECTOR_TYPE	The type to use when describing this type to the debugger. (Most debuggers do not understand vectors.)

Table 1: Multipurposing of the `values` field of `tree_type`

1.2 Multipurposing

Some fields of a substructure have different meanings for different tree codes. When there is more than one possible meaning, we say that that field is multipurposed. For instance, a `tree_type` structure represents a data type in the program being compiled. There are twenty tree codes that use this substructure. Eight of them assign one of five possible meanings to the `values` field. Table 1 enumerates the possibilities. The field goes unused in type nodes with one of the other twelve codes.

A relatively common special case of multipurposing is when a field has only one possible meaning, but only a subset of the tree codes for that substructure need to use that field. The others leave it as `NULL`.

1.3 Overloading

Many of the fields of a tree node are pointers to other nodes. These, like all pointers to tree nodes, have the type `tree`; as far as the C type system is concerned, they can point to any tree node. The operands of an `PLUS_EXPR` need not be expressions; they can be declarations, constants, types, or anything else.

Of course, not all possibilities can occur within a valid tree structure. The accessor macros partially validate the targets of pointer fields, and hand-coded assertions finish the job. When a field can legitimately point to more than one kind of node, we say that the field is overloaded.

The distinction between overloading and multipurposing is whether the code of the node containing the field determines what the field points to. The `values` field discussed above is multipurposed. An `PLUS_EXPR`'s operand fields are overloaded—we do not know, upon encountering an `PLUS_EXPR`, whether its operands are expressions, declarations, or constants. (We *do* know that they are in one of those three categories.)

2 Issues of the status quo

The present architecture has a number of design issues, which manifest either as runtime overhead (both space and time) or as increased burden on the maintainers of the program. For an obvious example of both, the runtime checking done by the accessor macros slows the compiler down 5–15% (depending on in-

put). This is substantial enough that checking is disabled in release builds, which can mean that bugs go undetected. It is on by default in development builds, which means GCC developers all put up with a slower compiler for the sake of dynamic type safety. A slow compiler, hence a slow edit-compile-link-debug cycle, is a maintenance burden in itself; also, the checking mechanism is complicated and easy to misprogram (see section 2.2 for an example).

Each of the above varieties of polymorphism has its own set of issues, which we will discuss in turn. We will also discuss a number of related issues that we intend to address at the same time.

2.1 Substructure overhead

The dynamic type system has a certain level of intrinsic overhead. In many cases, GCC's own source code, not the content of the program being compiled, completely determines the code of a tree node. However, we must still maintain the node header, which is a full word (the code plus 24 flags). For smaller nodes, this can be a considerable amount of memory overhead.

In the larger substructures, many of the fields are only applicable to a few of the tree codes that use those substructures. This obviously wastes memory. It is a particularly severe problem for type and declaration nodes; the content of a `CONST_DECL` could fit into 16 bytes or so on a 32-bit host, but it occupies 116 bytes anyway. The other side of this problem is that adding a new field to a substructure consumes memory proportional to the total number of nodes using that substructure, not just the number of nodes it's relevant to. People therefore avoid adding fields to substructures. Instead they add new purposes to existing fields, which adds to maintenance burden instead. We could solve this within the existing framework by defining new substructures, at the cost of additional complexity in the accessor macros.

While many nodes have fields that are never used, some nodes do not have enough, which leads to ancillary data being maintained outside the tree structure. This may consume more memory than would have been required otherwise, and it also makes the program harder to maintain, since all the necessary information is not in one place. Ironically, the declaration structure is also an example of this, with substantial ancillary data being carried in the `cgraph_node` structures.

2.2 Multipurposing and generic accessors

In the past, the accessor macros and the debugging pretty-printer (`debug_tree`) did not know anything about multipurposing. One would use the same accessor macro (`TYPE_VALUES`) for all five purposes listed in Table 1. This led to confusion about which tree codes might use a given field. While considerable work has gone into introducing more specific accessors, some generic accessors still exist. Furthermore, the set of valid codes for each accessor may be incorrect. As we were writing this paper, we discovered that two of the accessor macros for the `values` field allowed a `VECTOR_TYPE`. Obviously the same field cannot serve two purposes simultaneously. Tightening up the checks exposed a harmless bug in `expr.c` and a more serious bug in `cp/decl.c`.

Accessors for fields with only one use are still likely to check only that the substructure is correct, not that the field is relevant to the specific code. They thus fail to document or enforce which codes the fields *are* meaningful for. Generic routines that inspect trees (such as the debug-info generators) won't bother to check for an appropriate code; they'll rely on the fields being `NULL` when they are irrelevant. This situation can persist unnoticed until some-

one decides to introduce a second purpose for one of these fields. In the process that person will tighten the checking macros, which will probably cause the generic routines to fail.

2.3 Abusive overloading

Tree overloading sometimes happens naturally. For instance, the tree the parser builds for a complex arithmetic expression will consist of `EXPR` nodes which may point to other `EXPR`s, to `DECL`s, or to constants. This is a straightforward way to represent an abstract syntax tree, and it rarely causes trouble.

However, since all pointers to trees have the generic type `tree`, overloading can potentially happen anywhere. Since this flexibility is available, it has been used whenever it was locally convenient, without thought for global consequences. Indeed, usually there are none—at the time. Once overloading has been added to a tree, every routine that examines it must be prepared for whatever it might find in the overloaded field. The only way to prove that a given tree field is not overloaded is to do a global data flow analysis, which can be very difficult. Thus, global consequences creep into the compiler over time, as new routines are added that inspect trees that might be overloaded.

An example of these creeping consequences is the `name` field of `struct tree_type`. This usually points to a `TYPE_DECL` node, but sometimes it points to an `IDENTIFIER_NODE` instead. When you get which, and what that means, is not documented anywhere. Routines that just want to know the printable name of a type have to use locutions like the following:

```
name = TYPE_NAME (t);

if (TREE_CODE (name)
    == TYPE_DECL)
  name = DECL_NAME (name);

if (TREE_CODE (name)
    != IDENTIFIER_NODE)
  abort ();
```

A less troublesome, but still unwise, case of overloading is the C and C++ parsers' reuse of expression nodes while parsing declarations. Normally a `CALL_EXPR` represents a call to a function; its operands are the function to call, and a list of actual arguments. But the C and C++ front ends also use this expression to represent a function declaration; then its operands are the function's name, and a list of formal parameter declarations. This is convenient for the parser, but necessitates a complicated conversion routine (`grokdeclarator`) to generate the type and declaration structures expected by the rest of the compiler. These peculiar expressions are intended never to escape the C front end, so they have not had creeping global consequences. However, from time to time one does escape and cause an ICE elsewhere in the compiler.

We can generate a crude estimate of the number of places that have to take care when inspecting overloaded trees by counting uses of the `TYPE_P` and `DECL_P` macros. As of March 15, there were 41 and 80 uses, respectively, of these macros in the main `gcc` directory, or about one use every 4000 lines. The C++ front end had more, 143 and 67 uses respectively, or about one use every 500 lines. This is due to heavy overloading in the trees used to represent templates; see section 5.3 for further discussion.

2.4 Lists of trees

Linked lists are very common within trees. This data structure is convenient when the size of the list is not known in advance. However,

```
struct tree_list {
  struct tree_common {
    tree chain;
    tree type;
    enum tree_code code :8;
    /* 24 flag bits */
  };
  tree purpose;
  tree value;
};
```

Substructure of `TREE_LIST`

linked lists have notably more overhead than vectors on several different grounds.

Singly linked lists can be constructed using reserved fields in the nodes carrying the data, or using separate "cons cells." Ignoring malloc overhead, a linked list using reserved fields in the data nodes consumes exactly the same amount of memory as a vector of pointers to those nodes. Either way, there is one extra pointer for each node. Linked lists built out of separate cons cells, on the other hand, use twice as much memory as a vector; two extra pointers per node. In exchange, a data node can be on more than one list if separate cons cells are used. Either way, traversing a linked list is more likely to cause memory-cache thrashing than traversing the vector.

All tree nodes have a `chain` field, reserved for chaining the node into a linked list. However, this field goes unused in approximately two-thirds of all nodes (not counting `TREE_LIST`; see more detailed analysis below, in Section 3.1). Instead, separate lists are built out of `TREE_LIST` nodes. This is the "cons cell" technique, but with far more overhead, because each node in the list has the ability to point to three data nodes instead of just one.

In practice, slightly more than half of all lists use only one data pointer per node, and almost all the rest use only two. Also, the node header (as always) consumes a full word; it is fair to consider that entirely wasted, since lists are always known from context and the flag bits go unused. (See section 3.2 for details.) For a list with only one data pointer per node, this structure is 60% wasted space; compared to a vector or an internally chained list, 80%.

Because all the pointers are generic, a `TREE_LIST` does not reveal any information about its contents. Code that processes lists must know from context what the list contains, or else be prepared to encounter anything. Context determines the content in most cases; again, this will be discussed in detail in section 3.2.

2.5 Language-specific trees

As we mentioned above, language front ends have the ability to define new tree codes. Often these codes do not need their own substructures. For instance, all of the language-specific codes defined by the C front end are for C-specific operators, which use the generic "expression" substructure. However, some languages need their own substructures. The C++ front end defines five such. Since the definition of the basic `tree` type is in a language-independent header file, there is no way to include these substructures in the tree union. Thus, the accessor macros for those substructures must include casts to the appropriate type, which is a minor hassle. Also, the garbage collector must assume that language-specific substructures can be encountered anywhere, which adds both runtime overhead (determining which substructure is active costs two function calls per node visited) and source complexity (special annotations to indicate that the tree union is not exhaustive).

The `type` and `decl` substructures include an opaque pointer field that front ends can use to attach their own special data to type and declaration nodes. This mechanism provides a

clear separation between generic and language-specific data. It requires no casting, since the opaque pointer refers to a forward-declared `struct` type. Front ends simply provide a complete declaration. However, it does require a second memory allocation, which adds overhead.

Also, the front end might need to multipurpose this field—storing different information depending on what sort of type or declaration it is—but this is inconvenient, since these structures are *not* trees and cannot use the machinery that exists for tree polymorphism. The C++ and Java front ends solve this problem by duplicating much of that machinery. The Ada front end, instead, pretends that the field points to a tree, which can then be multipurposed in the normal fashion. Neither is an ideal solution.

The substructure for a bare identifier (code `IDENTIFIER_NODE`) also provides for front ends to attach their own data. Because identifiers are so frequent, this data is appended to the generic substructure instead of being separately allocated. This is efficient, but requires front ends to define complex macros to access their own data, just as they would for entirely language-specific substructures. Also, `IDENTIFIER_NODE`s are used in contexts where the language-specific data will never be used (notably `DECL_ASSEMBLER_NAME`) but space is allocated for it anyway.

The `tree_common` structure carries seven flag bits specifically for use by front ends, and several more that have generic names but are only relevant to front ends. The `type` substructure carries another seven, the `decl` substructure eight. These are not overhead as they occupy space that would otherwise be padding. However, they are a maintenance burden, because they are heavily multipurposed. It is often unclear which front ends use which bits for what, and `debug_tree` prints them with generic names.

Languages sometimes invent their own multipurposings for fields that would otherwise go unused. The C front end has recycled the `TYPE_VFIELD` field of incomplete `RECORD_TYPE` nodes to carry a list of `VAR_DECL`s with the incomplete type, so that it can adjust them later if the type is completed. This is much more efficient than the previous approach of carrying around a list of all variables with incomplete types in the translation unit. However, it directly violates the language-independent compiler's assumptions about what can appear in `TYPE_VFIELD`. Several bugs have been traced to this list escaping the C front end.

`TYPE_VFIELD` is available for use in the C front end because `RECORD_TYPE`s in C never have vtables. The `RECORD_TYPE` code is used for object classes as well as "plain old data" structs, so it has all the fields necessary to handle both, even though classes never occur in C. More generally, language-independent trees carry fields needed to represent the constructs of *all* the languages that GCC supports, even if they are being used to represent a language that doesn't have those constructs. This is memory overhead, no more. . . unless, as with `TYPE_VFIELD`, someone gets clever.

2.6 Memory allocation, precompiled headers

GCC uses a garbage-collecting allocator for all trees. This is convenient, because no one ever has to worry about the lifetime of these data structures.[3] It also facilitates precompiled headers (PCH). The current implementation, to first order, simply serializes to disk all live data in garbage-collected memory.

[3] Before the garbage collector was introduced, in 1999, use-after-free bugs appeared about once every two weeks; now they are unheard of.

When the garbage collector was first introduced, the marking routine for each data structure had to be written by hand. Now instead we use special "`GTY`" annotations in the source code, and a program called `gengtype` which understands a subset of C's type grammar. It scans the source code and generates marking routines, directed by the annotations. It also generates slightly different walking routines which are used for PCH save and restore.

Both these things are great achievements from a software maintenance standpoint. In the normal course of affairs, programmers need never worry about memory lifetime. PCH requires slightly more attention as one must ensure that everything that needs serialization is properly annotated. The `gengtype` program is a powerful tool for doing introspection on GCC's data structures. We used it for this paper, to gather statistics on how fields of tree nodes are used. We discuss below some other ways it could be helpful.

On the other hand, the garbage collector is not at all efficient. It allocates memory out of fixed-size buckets, with pages reserved for allocations of a given size, which causes considerable memory fragmentation. The collector uses a naïve mark-and-sweep algorithm, which has to scan the entire active memory set on each collection. This is so slow that GCC contains throttling heuristics that effectively disable all memory reuse for average-size translation units. The auto-generated marking routines require that type tags be in the same block of memory as the unions they disambiguate; in some places (notably the C++ front end's `struct lang_decl`) this forces the creation of a redundant tag.

This paper does not directly address any of the problems with the garbage collector. However, we expect our changes will cause trees to use substantially less memory and have somewhat more predictable lifetimes. In conjunction with the "zone collector" project, which is working towards a generational collection algorithm, this should offer substantial performance improvements.

3 Measurements

In order to make sensible plans to solve the problems we have discussed, we need hard data on how severe they are. Code inspection can reveal potential problems, but does not tell us what the actual allocation patterns are, and there is no way to get a sense of the "big picture." Overloading in particular is very hard to discover by code inspection.

We therefore modified the `gengtype` program to generate instrumentation which would measure how much overloading appeared in the trees produced by compilation of a test program. We classified each node twice, once by its tree code and once by its substructure.

For each field that pointed to another tree node, we recorded what kinds of tree it could point to, including nothing. When substructures contained arrays, such as `struct tree_exp`, we considered each element a separate field. This reveals for instance that the first operand of a `CALL_EXPR` is usually an `ADDR_EXPR`, and the second is always a `TREE_LIST`. We instrumented lists specially, recording their average length and the value distribution of the entire list, instead of treating each node as a separate entity.

Using CVS HEAD as of 15 March 2004, we measured allocations for the compilation of GCC's own C and C++ front ends (this exercises only the C compiler) and for a small STL-based C++ program. Each of these was compiled in a single pass, using GCC's intermodule mode. All inlining was disabled, and all function bodies retained, so that each function body

would be counted exactly once. Measurements were taken once at the end of compilation, so transitory tree nodes were not inspected. Unfortunately this means we missed some of the more bizarre things done with trees, such as the declaration expressions discussed in Section 2.3.

The C compiler generated a similar distribution of tree nodes during compilation of both front ends, so we present here only data for the C++ front end. Compiling this C program generated about 1 million instrumented nodes, occupying 75MB of storage. The C++ program was smaller. Compiling it generated about 150,000 nodes, occupying 9MB.[4]

3.1 Fields of `tree_common`

The `tree_common` substructure contains two tree-pointer fields, `chain` and `type`, which are present in every node whether it needs them or not. The utilization of these fields is laid out in Tables 2 and 3. (The "proportion" column is proportion of total GC memory allocation; not all of this is trees.) It is immediately clear that memory could be saved just by excluding these fields from substructures that never use them.

For our C++ test case, removing the `chain` pointer from nodes where it isn't used saves 134KB, or 1.5% of the total memory allocation. Removing the `type` pointer saves 58KB, or 0.6% of total memory. The numbers are more impressive for C: removing `chain` saves 2.3MB, or 3.1% of memory; removing `type` saves 780KB, or 1.0% of memory. If internal memory fragmentation is reduced by this change, which is likely as many of the affected nodes are one word bigger than a power of two, memory savings could be even bigger.

With more code changes, all of the uses of the `chain` field could be eliminated, saving even more memory. `DECL`s and `BLOCK`s are chained together to indicate the lexical scope of declarations and these lists could easily be replaced with vectors. Furthermore, in the GIMPLE representation (which had not yet been merged when these measurements were taken) statements are held in sequence with an external doubly-linked list, so they do not need internal chaining either.

3.2 List distribution

`TREE_LIST` nodes are used for all external singly-linked lists. If we looked at these nodes in isolation, all their fields would appear to be heavily overloaded. However, our instrumentation captured the context of each list, revealing that most lists have predictable dynamic types.

The C front end allocated roughly 300,000 list nodes while compiling the C++ front end. There were seven major contexts, which are enumerated in Table 4. Of these, only two have nontrivial amounts of overloading, and one of those is because `CONSTRUCTOR` nodes are used to initialize both arrays and structures. It is also apparent that the `type` field of these lists is completely unused, and the `purpose` field is unused in half of the cases. We could save roughly 5MB (7% of the total allocation) by converting them all to specialized vectors.

The C++ front end uses a wider variety of lists. Our C++ test case produced 70,000 tree nodes in about 30 different uses, which are enumerated in Table 5. Like the C front end, the `type` field is unused in nearly all contexts, and the `purpose` field is unused in about half of the cases. There is quite a bit of overloading, but in most cases there is one primary usage and a few outliers. The structures used to represent templates, however, will require special attention and is discussed in Section 5.3. If all of these uses were converted to specialized vec-

[4] All statistics are for a host architecture with 32-bit pointers.

tors, we might be able to save about $2/3$MB of memory (8% of the total).

We did not instrument `TREE_VEC` as carefully as `TREE_LIST`, but it shows similar properties. It does not carry three data pointers per entry, but it does have the full overhead of a `tree_common` header, whose `chain` and `type` fields go unused. The entries are, as usual, declared as `tree`s rather than anything more specific, but in most cases the entries are homogeneous within a given class.

3.3 Overloaded fields

Tables 6 and 7 show the distribution of overloaded and/or multipurposed fields for the C and C++ test programs respectively. Multipurposed fields are in *italics*. We only show cross-class overloading, as we are not proposing to get rid of within-class overloading. Most overloading occurs among one primary class and a few outliers. Where there are "secondary" uses, appearing in more than 5% of measured nodes, that is usually a case of multipurposing.

The primary class is not always what one expects—in C, both BLOCK.supercontext and EXPR.operands are 99% `DECL`s, where one might expect to find more `BLOCK`s and `EXPR`s respectively. This reflects the form of the typical C program. Inner scopes tend not to have variable declarations, and therefore not to need `BLOCK` nodes. Expressions tend to be simple, hence most `EXPR` nodes point directly to variable `DECL`s rather than to subexpressions. The C++ front end does more overloading than C, but we still observe the same pattern of primary uses and outliers, except where there is multipurposing. Expressions appear to be more complicated in C++ than in C, but still 94% of `EXPR`s point directly to `DECL`s.

TYPE.context and DECL.context are anomalous in having substantial secondary targets without multipurposing being involved. These fields point "upward" in the abstract syntax tree, toward larger lexical structures. Since `TYPE`s and `DECL`s can nest inside each other (especially in C++), the context fields need to be able to point to both `TYPE`s and `DECL`s.

4 Redesign

Our primary goal in redesigning trees is to reduce runtime overhead and maintenance burdens. As we have discussed, overhead comes first from wasted memory. The primary causes of wasted memory are unused fields in various tree substructures, and overuse of linked lists.

We could address unused fields without introducing any new static types. We could simply promote all instances of multipurposed fields to substructures. Constants are already like this. Each code in the "constant" class (integer, real, complex, string, vector) has its own substructure. Structure initializers are exceptional in that they are not treated as constants, but as expressions—this should probably be changed. It would not be hard to extend this to other structures. We would also want to break up `tree_common`, moving its pointers into the substructures where they are actually used.

Furthermore, we already have a `TREE_VEC` node that could replace `TREE_LIST` whenever the list length is known in advance and only one pointer per element is needed. For instance, it would be feasible to do this for `BLOCK_VARS`. Where this will not work, we could invent new lists with only one or two data pointers per node.

These changes would reduce maintenance burdens only because accessor macros would have more specific names, and the documentation would be improved. They would do nothing at all for the overhead entailed by runtime type checking. In fact, they might make it worse,

since many checking macros would become more specific. For instance, TREE_CHAIN and TREE_TYPE currently do no checking at all; in the above regime they would be replaced by several new macros, which would check for specific substructures.

In order to go any further, we need to make the static types of trees more specific. That is, we need to stop using tree as the type declaration of every pointer to a tree. If we are to do this, we must decide how specific to be in our static declarations. Where possible we will use pointers to specific structures. However, some degree of overloading is necessary. We propose to introduce four new types, each of which covers a subset of the present tree classes. A pointer with one of these types can be overloaded freely within that subset, but not outside. We discuss techniques for removing cross-class overloading in section 4.3. The replacement types are:

TYPE Type nodes: the present 't' class. For instance, INTEGER_TYPE, POINTER_TYPE, and RECORD_TYPE.

DECL Declaration nodes: the present 'd' class. For instance, FUNCTION_DECL, VAR_DECL, and TYPE_DECL.

EXPR Expression nodes: the present '1', '2', 'r', '<', and 'e' classes. For instance, PLUS_EXPR, LE_EXPR, and ADDR_REF.

CONST Constant nodes: the present 'c' class. For instance, INTEGER_CST and STRING_CST.

The 's' class is not included in this mapping because, with the introduction of GENERIC and GIMPLE, the language-independent compiler no longer makes a strong distinction between statements and expressions. For instance, COND_EXPR can be either a ?: operator or an if statement. This does not preclude a front end from making a strong distinction in its own data structures, if that is appropriate to the language it recognizes.

Each of the miscellaneous trees (class 'x') requires individual attention. Some of them can be replaced with plain C structs that never participate in overloading. The BLOCK node for instance will get this treatment. Other nodes will be be recategorized into one or more of the above classes. For instance, we need equivalents of ERROR_MARK for each of the above categories; these should *not* be unique, so that they can carry information (such as the location of the error).

Obviously it will not be possible to continue using one structure, carrying no static type information, for all linked lists. However, as we detail in Section 3.2, most lists point to data items whose dynamic types are both predictable and homogeneous. Therefore, with a moderate amount of effort we can replace TREE_LIST with specialized list nodes for each of the classes.

4.1 Type safety

Under the old design, all pointers had the same static type, so there was never any need to convert them. Under the new design, we would like to make the static types of pointers as specific as possible. The four classes above are base types in the C++ sense, and each substructure is a derived type. We will need a type-safe and terse way to convert between base and derived type pointers. Unfortunately the C language does not provide convenient facilities for this sort of operation. Pointers to different structs are not assignment-compatible. There is only one cast operator, (*type*), which does not validate the incoming type at all.

We can simulate the C++ derived-type compatibility rule and `dynamic_cast<>` operator in C, with a small amount of extra verbosity and some GNU extensions. In Figure 1 we illustrate one way to implement the conversion operations, and the associated structure layout. Code written to this convention should look almost the same as code written to the old convention, but with specific variable types and occasional explicit conversions. It might be possible to use `gengtype` to generate all of the accessor macros and checking logic from the substructure definitions, thus eliminating that source of bugs and tedium.

There would be a `_common` structure for each of the four major static types. Any fields that truly are common to all substructures of that type can be placed there. In the example, we included two boolean fields which are documented as relevant to all constants. We have not yet decided what naming convention use for the new types; the mixture of struct tags and all-caps typedefs in figure 1 is only one possibility.

The GNU extensions are only necessary for type checking. When GCC is built with a compiler that does not support them, the macros can expand to unchecked casts; the compiler will still work. The compile-time error message produced by these macros is suboptimal; it could be improved with a `__builtin_error` primitive. Also, in real life the runtime checks would call a more specific ICE-reporting routine than `abort`. These details were omitted from the example for brevity.

Some checking does still occur at runtime. We expect that the overhead will be substantially lower in this scheme, but we can still disable runtime checking in release builds for efficiency.

4.2 Language augmentations

The coding convention shown in Figure 1 deliberately does not use unions, unlike the current convention. This is because the union cannot include any language-specific substructures, and we want to put them on an equal footing with language-independent substructures. The checked-cast approach is similar to what is done now for language-specific substructures, but safer. If the macros are automatically generated, it will also be much less tedious. Front ends are also free to declare new polymorphic classes; for instance, a language that wants a strong distinction between statements and expressions can invent a **STMT** class.

We also want to make it easier to add language-specific data to generic substructures. It is straightforward for a language to declare an augmented substructure and accessors, as they do now for `IDENTIFIER_NODE`. However, the garbage collector must be advised to allocate more memory for the augmented structure, and to walk the complete structure for pointers when marking live data. This is done for `IDENTIFIER_NODE` with special `GTY` markers and language hooks, which do not scale. We have not yet decided on a tactic for this problem.

Finally, we intend to make tree codes more specific so that languages do not have to incur overhead for functionality they do not use. For instance, the `RECORD_TYPE` code will apply only to "plain old data;" we will introduce a new `CLASS_TYPE` node for object classes.

4.3 Adaptor nodes

Section 3.3 outlined instances of cross-class overloading, that is, cases where `tree` pointers can refer to more than one of the four static classes discussed in Section 4. We can elimi-

nate many of these, but some are legitimate.

We do not want to combine the DECL, EXPR, and CONST classes, but we could introduce **adaptor** nodes, which fit into one class and carry a pointer to another class. They might or might not carry other information. We already have the notion of a TYPE_DECL; we could reuse it as an adaptor for context fields pointing to a TYPE. Context fields can also point to BLOCKs; for that, we would need a new BLOCK_DECL adaptor.

The statistics in tables 6 and 7 show that 94–99% of expression operands are DECLs, so it would be most efficient to make that the unmarked case. We would add an EXPR_DECL adaptor for subexpressions, and use the existing CONST_DECL as an adaptor for literal constants. This could facilitate conversion to GIMPLE form, where all subexpressions are separated from their contexts.

5 Conversion plan

Converting to statically typed trees is a considerable amount of work. It will have to be done either piecemeal on the mainline, or on its own dedicated branch. If the work is done on a branch, it will rapidly become very hard to merge in changes from the mainline. However, if the work is done on the mainline, it is likely to be disruptive to other projects. The conversion may not be monotonic, and there are several issues as yet unresolved, for which experimentation will be necessary. Also, this project is more work than one person can do alone. Collaboration by emailing patches back and forth is tedious, compared to collaboration by working on the same branch.

On balance, we believe that most of the work should be done on a branch. However, in order to avoid severe divergence, the project should be broken into steps which can be merged back to mainline when complete. We will partition these steps into three stages.

The first stage of the process is to promote all multipurposed fields to substructures. It may be feasible to do this stage before branching. It is very simple and low-impact for fields whose accessor macros are already as specific as they can get. Fields that have non-specific accessor macros require more thought, and the change may be quite large, but still mostly mechanical. The chain and type fields of tree_common will migrate into the substructures that actually use them. It would be nice to do the same for the common flag bits, but that may not be feasible without introducing unwanted padding.

The tree-ssa branch has introduced a number of new 'x' nodes that are used in expressions, such as SSA_NAME. These are not in class 'e' mainly to avoid wasting memory on useless fields attached to all expressions. If the substructure conversion is done properly it will be possible to put them in class 'e' or possibly a new expression subclass.

The second stage is to eliminate as much overloading as possible, particularly what we might describe as "abusive" overloading. We discuss approaches to some of these in sections 5.1–5.3. The branch will be merged after each abuse has been rectified. This stage will have to occur semi-concurrently with the next one, because we do not know where all of the problems are.

The third stage is to peel off the major tree classes from the über-union, one at a time. The branch will be merged after each step. Except where we encounter unexpected abuses, the substantial changes in this stage affect only the implementation of the accessor macros. However, this is the stage where we change variable declarations, introduce explicit conversions, and rename accessor macros to con-

form to a naming scheme that facilitates automatic generation. This will entail mechanical changes all over the compiler. We propose to do this stage in the following order:

Identifiers With the exception of C++ template bodies, there are only a few places where a tree node might or might not be an identifier, and they are all arguably bugs. The new C++ parser should make it feasible to use custom data structures for C++ template bodies, so that `IDENTIFIER_NODE` need not be an overloading candidate at all. In some places, identifiers are used where unboxed strings would suffice; we will remove all such identifiers in this step.

`ERROR_MARK` There is one error mark node, which can appear in any context where the tree is incomplete because the input program was incorrect. It carries no information. We mean to replace it with separate `INVALID_TYPE`, `INVALID_DECL`, `INVALID_EXPR`, and possibly `INVALID_CST` codes. These nodes will not be unique, and will carry enough information that later stages of compilation do not need to be aware of them.

Lists and vectors `TREE_LIST` must be replaced with specialized list nodes that carry static type information. It is also desirable to use vectors where possible, instead of lists. In this step we will design a macro API for synthesizing vector and list types, and the associated runtime API for building lists, converting lists to vectors, etc. This will allow us to save memory immediately, by removing the unused pointers from most lists. In further steps we will use it to define specialized list and vector types as needed.

Blocks The lexical binding node, `BLOCK`, can only appear within certain nodes and contexts, and therefore can be separated out relatively easily. It contains a list of `DECL`s, which will be the first use of specialized vector types.

Types Of the remaining tree nodes, types are the most distinct; there is rarely cross-class overloading between types and other things. However, we will need to create specialized lists of types, and we expect to find abuses in their relationship to declarations.

Constants In this step we will replace overloading between declarations and strings with anonymous `CONST_DECL` adaptors. Also, trees which are always `INTEGER_` or `STRING_CST` nodes will be replaced with unboxed integers or strings.

Expressions Next, we give expressions a distinct type, and make their operands always be `DECL` nodes. Subexpressions will be wrapped in `EXPR_DECL` adaptor nodes. This is one of the most invasive changes to be made; however, a suitably clever definition of `TREE_OPERAND` should make it possible to do it piecemeal.

Declarations At this point the only things left in the tree union are declarations. We can replace all remaining `tree` variables with `DECL` variables, and delete the union entirely.

We will now discuss a few conversion steps in more detail.

5.1 C declaration parsing

The C and C++ parsers reuse expression nodes for temporary structures while parsing declarations, as described in section 2.3. This is incompatible with static typing. Also, it is inefficient; the temporary structure is far larger

```
struct binfo {
  unsigned int flags;
  tree type;
  struct binfo *next;
  struct binfo *inheritance;
  tree offset;
  tree vtable;
  tree virtuals;
  tree vptr_field;
  unsigned int num_bases;
  struct base {
    tree access;
    struct binfo *base;
  } bases[];
};
```

Custom BINFO structure

than it needs to be (for instance, lists of identifiers are used in places where flag words would suffice) and the entire thing is discarded after processing by grokdeclarator, producing lots of garbage.

We plan to replace these expressions with a custom data structure. It need only contain fields for the information added at its level (cv-qualifiers, attributes, array or function parameters), an enumeration of what is being declared (array, pointer, etc), and a pointer to the structure for the next level. It would use the polymorphism techniques described in Section 4.1, but static type constraints would ensure that it never escaped the front end.

We expect this project to have the pleasant side effect of replacing grokdeclarator with a set of simpler functions, none of which is 1200 lines long.

5.2 BINFOs

The RECORD_TYPE for each class declared in a C++ program has a set of BINFO structures to represent its base class organization. There is one BINFO for each base class, arranged in a directed acyclic graph which mirrors the class hierarchy. They carry data such as the location of the base sub-object, the class type of the base, etc.

A BINFO is a TREE_VEC with indexes defined for each piece of information. Information about a BINFO's base BINFOs is held in two additional TREE_VECs, which is unnecessary fragmentation. There is a comment in tree.h suggesting that this be changed:

> ??? This could probably be done by just allocating the base types at the end of this TREE_VEC (instead of using another TREE_VEC). This would simplify the calculation of how many basetypes a given type had.

As with declarator expressions, we mean to replace BINFO with a custom structure. The fields that point to BINFOs are never overloaded, so we do not need to make it a tree substructure. An example structure is shown above, as it would appear before conversion to specific static types. Further memory savings are possible: we can store less information in the BINFO and more in the RECORD_TYPE of the base class, where it is not copied for every derived class. The virtuals field is a long list, with one entry for every virtual function in that class. If it can't be moved to the RECORD_TYPE, we can at least convert it to a specialized vector.

5.3 Template arguments and levels

C++ template parameters may be types, expressions, or nested templates. Presently, the C++ front end takes advantage of overloading to put all these things in a single parameter vector. Many of the uses of TYPE_P and DECL_P within the C++ front end are due to this overloading. In this context, types are the most

```
struct inner_vec {
  unsigned int num_args;
  tree args[];
};
struct outer_vec {
  unsigned int num_levels;
  struct inner_vec *levels[];
};
```

Two-dimensional template parameter array

common sort of parameter. We could use C++-specific `EXPR_TYPE` and `DECL_TYPE` adaptor nodes. Another option is to use a tagged array of unions, but then we would have to find somewhere to put the tags.

It is possible for a template to have more than one level of template parameters. Such templates have a vector of parameter vectors, one for each level. To avoid overhead, templates with only one level of parameters omit the outer vector. This is another kind of overloading, and it costs quite a bit of complexity (mostly in `cp-tree.h`'s macros for manipulating template trees). A specialized two-dimensional array would have substantially less overhead. One possible structure layout is shown here.

6 Closing remarks

This paper concentrates mostly on the common code, and the C and C++ front ends. The Java, Ada, Fortran and Objective C front ends will no doubt have specific issues during conversion. With the possible exception of Ada, we expect that these will be no more trouble than the C++ front end. We will need support from front end maintainers to complete the conversion for all front ends.

We have glossed over the process of defining specialized list and vector types. By the time that is necessary, we will have already converted some list usages, giving experience in the features that are necessary. We expect that at that time a good approach will be obvious.

7 Acknowledgments

We would like to thank Diego Novillo and Christian Lavoie for commenting on drafts of the paper, and Sumana Harihareswara, Michael Ellsworth, and Julia Bernd for copyeditorial assistance above and beyond the call of duty.

Tables and figures

In Tables 4–7, upper case indicates nodes with a particular tree structure; lower case indicates nodes with a particular tree code. An entry with just a dash (—) indicates a field that was never used.

		Utilization	
Class	Proportion	chain	type
BLOCK	1.61%	47.78%	0.00%
DECL	26.46%	89.81%	99.30%
EXPR	35.72%	0.00%	100.00%
STMT	14.85%	60.21%	0.00%
IDENTIFIER	1.72%	0.00%	0.00%
CONSTANT	14.75%	0.00%	100.00%
TYPE	4.89%	0.00%	71.42%

Table 2: `tree_common` utilization by class in C program

		Utilization	
Class	Proportion	chain	type
BLOCK	3.85%	2.35%	0.00%
DECL	33.60%	60.80%	99.68%
EXPR	19.23%	0.00%	43.45%
STMT	14.46%	38.93%	0.00%
IDENTIFIER	7.26%	0.00%	7.40%
CONSTANT	3.18%	0.00%	100.00%
TYPE	12.80%	0.00%	65.98%

Table 3: `tree_common` utilization by class in C++ program

Field	Null	Len	Type	Purpose	Value
call_expr.op[1]	2%	3.5	—	—	EXPR
record_type.minval[a]	99%	3.0	—	—	DECL
function_type.values	0%	3.7	—	—	TYPE
enumeral_type.values	0%	23.1	—	identifier	integer_cst
DECL.attributes	91%	1.4	—	identifier	—[b]
TYPE.attributes	98%	1.9	—	identifier	list
constructor.op[0]	0%	9.6	—	field_decl 65%	EXPR
				integer_cst 35%	
TYPE.attributes.value	0%	2.1	—	—	identifier 26%
					integer_cst 74%

[a] `C_TYPE_INCOMPLETE_VARS`; the C front end has invented its own multipurposing for this field (see section 2.5).

[b] This field is non-NULL for some attributes, none of which are used in the program we measured.

Table 4: Lists in C program

Field	Null	Len	Type	Purpose	Value
record_t.pure_virtuals	99%	8.7	—	—	method_t
record_t.befriending_classes	96%	1.3	—	—	record_t
record_t.vfields	85%	1.0	—	—	record_t
record_t.friend_classes	97%	2.3	—	—	record_t
type_d.initial.value	0%	2.0	—	—	DECL
var_d.initial	17%	2.3	—	—	EXPR
nw_expr.operands[0]	77%	1.0	—	—	EXPR
call_expr.operands[1]	32%	2.4	—	—	EXPR >99%
					identifier <1%
TYPE.attributes.value	0%	1.6	—	—	integer 82%
					identifier 18%
function_t.binfo	73%	1.0	—	—	null 99%
					record_t 1%
method_t.binfo	82%	1.0	—	—	null >99%
					record_t <1%
cast_expr.operands[0]	32%	1.1	—	—	DECL 55%
					EXPR 38%
					CONST 7%
namespace_d.initial	57%	1.0	—	namespace	—
namespace_d.saved_tree	71%	1.0	—	namespace	—
DECL.attributes	96%	1.4	—	identifier	—
TYPE.attributes	99%	1.7	—	identifier	list
type_d.initial	99%	1.3	—	identifier	list
enumeral_t.values	0%	16.9	—	identifier	integer
record_t.vcall_indices	85%	5.6	—	function_d	integer
constructor.operands[0]	0%	8.6	—	integer	EXPR
record_t.template_info	24%	1.0	—	DECL	vec
record_t.vbases	98%	1.0	—	record_t	vec
template_d.arguments	0%	1.0	—	int_cst	vec
DECL.template_info	63%	1.0	—	DECL >99%	vec
				overload <1%	
ctor_initializer.operands[0]	10%	2.1	—	DECL 95%	list
				record_t 5%	
record_t.decl_list	50%	19.4	—	record_t 99%	DECL
				null 1%	
function_t.values	<1%	3.3	—	null >99%	TYPE
				EXPR <1%	
method_t.values	0%	3.3	—	null 97%	TYPE
				EXPR 3%	
TEMPLATE_PARMS	0%	1.0	—	null 74%	DECL
				TYPE 25%	
				EXPR 1%	
template_d.vindex	96%	3.4	—	vec	record_t 97%
					null 3%
template_d.size	56%	2.0	null 99%	vec	DECL 99%
			record_t 1%		vec 1%
namespace_d.vindex	57%	1.0	—	null 67%	null 67%
				namespace 33%	namespace 33%

Note: **_t** is short for **_type**, **_d** for **_decl**.

Table 5: Lists in C++ program

In Tables 6 and 7, *italics* indicate a multipurposed field; roman font indicates an overloaded field.

Field	Primary		Secondary		Outlier	
BLOCK.supercontext	DECL	99%			BLOCK	1%
DECL.context	DECL	100%			TYPE	<1%
DECL.initial	DECL	79%	EXPR	19%	TYPE	2%
					BLOCK	<1%
DECL.result	TYPE	86%	DECL	14%		
EXPR.operands	DECL	99%			EXPR	<1%
					IDENTIFIER	<1%
					LIST	<1%
					BLOCK	<1%
TYPE.context	DECL	87%	BLOCK	13%		
TYPE.name	DECL	100%			IDENTIFIER	<1%
TYPE.values	LIST	76%	DECL	24%	TYPE	<1%

Table 6: Multipurposing and overloading in C program

Field	Primary		Secondary		Outlier	
BLOCK.supercontext	DECL	98%			BLOCK	2%
DECL.arguments	DECL	79%	LIST	14%		
			INT_CST	7%		
DECL.context	DECL	98%			TYPE	2%
DECL.initial	TYPE	54%	DECL	16%	LIST	1%
			BLOCK	12%	STRING	<1%
			INT_CST	11%		
			EXPR	5%		
DECL.befriending_classes	LIST	60%				
	DECL	40%				
DECL.result	DECL	98%			TYPE	2%
DECL.saved_tree	EXPR	100%			LIST	<1%
DECL.size	INT_CST	88%	LIST	12%		
DECL.vindex	DECL	54%	INT_CST	22%	TYPE	4%
			LIST	19%		
EXPR.operands	DECL	94%	EXPR	5%	LIST	<1%
					INT_CST	<1%
					BLOCK	<1%
					STRING	<1%
					TYPE	<1%
TYPE.context	DECL	62%	TYPE	38%		
TYPE.values	LIST	67%	DECL	22%	IDENTIFIER	1%
			TPI	9%	EXPR	<1%
					TYPE	<1%

Table 7: Multipurposing and overloading in C++ program

```
/* If V has type T, return V, else issue an error.  */
#define verify_type(T,V)                            \
  (__builtin_choose_expr                            \
   (__builtin_types_compatible_p (typeof(V), T),    \
    (V), (void) 0))

/* If V has type T or F, return (T)V, else issue an error.  */
#define validated_cast(T,F,V)                       \
 (__builtin_choose_expr                             \
  (__builtin_types_compatible_p (typeof(V), T)      \
   || __builtin_types_compatible_p (typeof(V), F), \
  (T) (V), (void) 0))

/* If V has static type F or T and dynamic type K, return (T)V, else
   issue an error.  F and T are checked at compile time, K at runtime.  */
#define with_dynamic_type(K,T,F,V)                  \
  ({ T _v = validated_cast(T,F,V);                  \
     if (_v->common.kind != K)                      \
       abort ();                                    \
     _v; })

enum cst_kind { INTEGER_CST, ... };

struct cst_common
{
  enum cst_kind kind : 8;
  bool warned_overflow : 1;
  bool overflow : 1;
  /* possibly other flag bits */
};
typedef struct cst_common *CONST;
#define CONST(C) verify_type(CONST, &C->common)

#define CONST_OVERFLOW(C) CONST(C)->overflow
#define CONST_WARNED_OVERFLOW(C) CONST(C)->warned_overflow

struct cst_int
{
  struct cst_common common;
  unsigned HOST_WIDE_INT low;
  HOST_WIDE_INT high;
};
#define CST_INT(C) \
 with_dynamic_type(INTEGER_CST, struct cst_int *, CONST, C)

#define CST_INT_LOW(C) CST_INT(C)->low
#define CST_INT_HIGH(C) CST_INT(C)->high
```

Figure 1: Structure and macro conventions for type safety