GENERIC and GIMPLE: A New Tree Representation for Entire Functions

Jason Merrill
Red Hat, Inc.
`jason@redhat.com`

1 Abstract

The tree SSA project requires a tree representation of functions for the optimizers to operate on. There was an existing functions-as-trees representation shared by the C and C++ front ends, and another used by the Java front end, but neither was adequate for use in optimization. In this paper, we will discuss the design of GENERIC, the new language-independent tree representation, and GIMPLE, the reduced subset used during optimization.

2 Introduction

For most of its history, GCC has compiled functions directly to RTL (Register Transfer Language) on a statement-by-statement basis. RTL has been a very useful intermediate language (IL) for low-level optimizations, but has significant limitations that keep it from being very useful for higher level optimizations:

- Its notion of data types is limited to machine words; it has no ability to deal with structures and arrays as a whole.
- It introduces the stack too soon; taking the address of an object forces it into the stack, even if later optimization removes the need for the object to be addressible.

GCC also has another IL: its abstract syntax tree representation. In the past, the compiler would only build up trees for a single statement, and then lower them to RTL before moving on to the next statement. This began to change in GCC 3.0: CodeSourcery, LLC modified the C++ compiler to store entire functions as trees and only lower them to RTL as part of compiling to assembly. As part of the same work, they introduced the first tree-level optimization pass, the inliner. Inlining at the tree level partially addressed the second limitation of RTL mentioned above, since C++ objects passed as arguments to a function are usually passed by address.

The tree SSA project is intented to expand on this by performing a full set of optimizations at the tree level. But to do this, we needed to refine how we use trees to represent whole functions. The result is GIMPLE, and its superset GENERIC.

3 Existing Tree ILs

The C++ compiler work was later extended to work with the C compiler as well, but never became a language-independent tree IL. Initial work on tree-ssa was based on the C front end trees, but they were unsuited for use in optimization.

The main shortcoming of C trees, from an optimization standpoint, is that they are highly context-dependent. Many `_STMT` codes just serve as placeholders for calls to `expand_` functions and rely on the RTL layer to keep track of scoping. For a tree IL to be useful for optimization, things such as the target of a `break` or `continue` statement, or the scope of a C++ cleanup, must be made explicit.

The other preexisting tree IL is the one in the Java front end. Java made an effort to use backend tree codes whenever possible, added a few new tree codes to the backend, and retained a few in the front end. GENERIC is largely based on Java front end trees, adjusted to be entirely language independent.

4 GENERIC

The purpose of GENERIC is simply to provide a language-independent way of representing an entire function in trees. To this end, it was necessary to add a few new tree codes to the backend, but most everything was already there. If you can say it with the codes in `gcc/tree.def`, it's GENERIC.

Early on, there was a great deal of debate about how to think about statements in a tree IL. In GENERIC, a statement is any expression whose value, if any, is ignored. A statement will always have `TREE_SIDE_EFFECTS` set (or it will be discarded), but a non-statement expression may also have side effects. A `CALL_EXPR`, for instance.

It would be possible for some local optimizations to work on the GENERIC form of a function; indeed, the adapted tree inliner works fine on GENERIC, but the current compiler performs inlining after lowering to GIMPLE.

If necessary, a front end can use some language-dependent tree codes in its GENERIC representation, so long as it provides a hook for converting them to GIMPLE and doesn't expect them to work with any (hypothetical) optimizers that run before the conversion to GIMPLE.

5 GIMPLE

GIMPLE is a simplified subset of GENERIC for use in optimization. The particular subset chosen (and the name) was heavily influenced by the SIMPLE IL used by the McCAT compiler project at McGill University [SIMPLE], though we have made some different choices. For one thing, SIMPLE doesn't support `goto`; a production compiler can't afford that kind of restriction.

GIMPLE retains much of the structure of the parse trees: lexical scopes and control constructs such as loops are represented as containers, rather than markers. However, expressions are broken down into a 3-address form, using temporary variables to hold intermediate values.

Similarly, in GIMPLE no container node is ever used for its value; if a `COND_EXPR` or `BIND_EXPR` has a value, it is stored into a temporary within the controlled blocks, and that temporary is used in place of the container.

The compiler pass which lowers GENERIC to GIMPLE is referred to as the "gimplifier." The gimplifier works recursively, replacing complex statements with sequences of simple statements. Currently, the only way to tell whether or not an expression is in GIMPLE form is by recursively examining it; in the future there will probably be a flag to help avoid redundant work.

6 Interfaces

The tree representation of a function is stored in `DECL_SAVED_TREE`. It is lowered to GIMPLE by a call to `simplify_function_tree`.

If a front end wants to include language-specific tree codes in the tree representation which it provides to the back-end, it must provide a definition of `LANG_HOOKS_SIMPLIFY_EXPR` which knows how to convert the front end trees to GIMPLE. Usually such a hook will involve much of the same code for expanding front end trees to RTL. This function can return fully lowered GIMPLE, or it can return GENERIC trees and let the main gimplifier lower them the rest of the way; this is often simpler.

The C and C++ front ends currently convert directly from front end trees to GIMPLE, and hand that off to the backend rather than first converting to GENERIC. Their gimplifier hooks know about all the `_STMT` nodes and how to convert them to GENERIC forms. I worked for a while on a genericization pass which would run first, but the existence of `STMT_EXPR` meant that in order to convert all of the C statements into GENERIC equivalents would involve walking the entire tree anyway, so it was simpler to reduce all the way. This may change in the future if someone writes an optimization pass which would work better with higher-level trees, but currently the optimizers all expect GIMPLE.

A frontend which wants to use the tree optimizers (and already has some sort of whole-function tree representation) only needs to provide a definition of `LANG_HOOKS_SIMPLIFY_EXPR` and call `simplify_function_tree` and `optimize_function_tree` before they start expanding to RTL. Note that there actually is no real handoff to the tree backend at the moment; in the future there will be a `tree_rest_of_compilation` which will take over, but it hasn't been written yet.

Note that there are still a large number of functions and even files in the gimplifier which use “simplify” instead of “gimplify.” This will be corrected before the project is merged into the GCC trunk.

You can tell the compiler to dump a C-like representation of the GIMPLE form with the flag `-fdump-tree-simple`.

7 GIMPLE reference

7.1 Temporaries

When gimplification encounters a subexpression which is too complex, it creates a new temporary variable to hold the value of the subexpression, and adds a new statement to initialize it before the current statement. These special temporaries are known as “expression temporaries,” and are allocated using `get_formal_tmp_var`. The compiler tries to always evaluate identical expressions into the same temporary, to simplify elimination of redundant calculations.

We can only use expression temporaries when we know that it will not be reevaluated before its value is used, and that it will not be otherwise modified (these restrictions are derived from those in [Morgan]

4.8). Other temporaries can be allocated using `get_initialized_tmp_var` or `create_tmp_var`.

Currently, an expression like `a = b + 5` is not reduced any further, though in future this may be converted to

```
T1 = b + 5;
a = T1;
```

to avoid problems with optimizers trying to refer to variables after they've gone out of scope.

7.2 Expressions

In general, expressions in GIMPLE consist of an operation and the appropriate number of simple operands; these operands must either be a constant or a variable. More complex operands are factored out into temporaries, so that

```
a = b + c + d
```

becomes

```
T1 = b + c;
a = T1 + d;
```

The same rule holds for arguments to a `CALL_EXPR`.

The target of an assignment is usually a variable, but can also be an `INDIRECT_REF` or a compound lvalue as described below.

7.2.1 Compound Expressions

The left-hand side of a C comma expression is simply moved into a separate statement.

7.2.2 Compound Lvalues

Currently compound lvalues involving array and structure field references are not broken down; an expression like `a.b[2] = 42` is not reduced any further (though complex array subscripts are). This restriction is a workaround for limitations in later optimizers; if we were to convert this to

```
T1 = &a.b;
T1[2] = 42;
```

alias analysis would not remember that the reference to `T1[2]` came by way of `a.b`, so it would think that the assignment could alias another member of `a`; this broke `struct-alias-1.c`. Future optimizer improvements may make this limitation unnecessary.

7.2.3 Conditional Expressions

A C `?:` expression is converted into an `if` statement with each branch assigning to the same temporary. So,

```
a = b ? c : d;
```

becomes

```
if (b)
  T1 = c;
else
  T1 = d;
a = T1;
```

Note that in GIMPLE, `if` statements are also represented using `COND_EXPR`, as described below.

7.2.4 Logical Operators

Except when they appear in the condition operand of a `COND_EXPR`, logical ‘and’ and ‘or’ operators are simplified as follows: `a = b && c` becomes

```
  T1 = (bool)b;
  if (T1)
    T1 = (bool)c;
  a = T1;
```

Note that `T1` in this example cannot be an expression temporary, because it has two different assignments.

7.3 Statements

Most statements will be assignment statements, represented by `MODIFY_EXPR`. A `CALL_EXPR` whose value is ignored can also be a statement. No other C expressions can appear at statement level; a reference to a volatile object is converted into a `MODIFY_EXPR`.

There are also several varieties of complex statements.

7.3.1 Blocks

Block scopes and the variables they declare in GENERIC and GIMPLE are expressed using the `BIND_EXPR` code, which in previous versions of GCC was primarily used for the C statement-expression extension.

Variables in a block are collected into `BIND_EXPR_VARS` in declaration order. Any runtime initialization is moved out of `DECL_INITIAL` and into a statement in the controlled block. When gimplifying from C or C++, this initialization replaces the `DECL_STMT`.

Variable-length arrays (VLAs) complicate this process, as their size often refers to variables initialized earlier in the block. To handle this, we currently split the block at that point, and move the VLA into a new, inner `BIND_EXPR`. This strategy may change in the future.

`DECL_SAVED_TREE` for a GIMPLE function will always be a `BIND_EXPR` which contains declarations for the temporary variables used in the function.

A C++ program will usually contain more `BIND_EXPR`s than there are syntactic blocks in the source code, since several C++ constructs have implicit scopes associated with them. On the other hand, although the C++ front end uses pseudo-scopes to handle cleanups for objects with destructors, these don’t translate into the GIMPLE form; multiple declarations at the same level use the same BIND_EXPR.

7.3.2 Statement Sequences

Currently, multiple statements at the same nesting level are connected via `COMPOUND_EXPR`s. This representation was chosen both because of precedent and because it simplified the implementation of the gimplifier. However, it makes transformations during optimization more complicated, and there is some concern about the memory overhead involved.

The complication is mostly encapsulated by the use of iterators declared in `tree-iterator.h`. The representation may be extended in the future, perhaps to use statement vectors or a double-chained list, but the iterators should also avoid the need for any changes in the optimizers.

7.3.3 Empty Statements

Whenever possible, statements with no effect are discarded. But if they are nested within another construct which cannot be discarded for some reason, they are instead replaced with an empty statement, generated by `build_empty_stmt`. Initially, all empty statements were shared, after the pattern of the Java front end, but this caused a lot of trouble in practice, and they were recently unshared.

An empty statement is represented as `(void)0`.

7.3.4 Loops

All loops are currently expressed in GIMPLE using `LOOP_EXPR`, which represents an infinite loop. Loop conditions, `break` and `continue` are converted into explicit gotos.

A future loop optimization pass may represent canonicalized loops using another tree code, perhaps `DO_LOOP_EXPR`, but this has not been implemented yet.

7.3.5 Selection Statements

A simple selection statement, such as the C `if` statement, is expressed in GIMPLE using a void `COND_EXPR`. If only one branch is used, the other is filled with an empty statement.

Normally, the condition expression is reduced to a simple comparison. If it is a shortcut (&& or `||`) expression, however, we try to break up the `if` into multiple `if`s so that the implied shortcut is taken directly, much like the transformation done by `do_jump` in the RTL expander. Currently, this is only done when it can be done simply by adding more `if`s; in the future, this transformation will handle more cases and use `goto` if necessary.

The representation of a `switch` is still unsettled. Currently, a `SWITCH_EXPR` contains the condition, the body, and a `TREE_VEC` of the `LABEL_DECL`s which the `switch` can jump to, and `case` labels are represented in the body by `CASE_LABEL_EXPR`s. In future, we may want to move even more information about the cases into the `SWITCH_EXPR` itself, and reduce the `CASE_LABEL_EXPR`s to plain `LABEL_EXPR`s.

7.3.6 Jumps

Other jumps are expressed by either `GOTO_EXPR` or `RETURN_EXPR`.

The operand of a `GOTO_EXPR` must be either a label or a variable containing the address to jump to.

The operand of a `RETURN_EXPR` is either `NULL_TREE` or a `MODIFY_EXPR` which sets the return value. I wanted to move the `MODIFY_EXPR` into a separate statement, but the special return semantics in `expand_return` make that difficult. It may still happen in the future.

7.3.7 Cleanups

Destructors for local C++ objects and similar dynamic cleanups are represented in GIMPLE by a `TRY_FINALLY_EXPR`. When the controlled block exits, the cleanup is run.

`TRY_FINALLY_EXPR` complicates the flow graph, since the cleanup needs to appear on every edge out of the controlled block; this reduces our freedom to move code across these edges. In the future, we will want to lower TRY_FINALLY_EXPR to simpler forms at some point in optimization, probably by changing it into a TRY_CATCH_EXPR and inserting an additional copy of the cleanup along each normal edge out of the block.

7.3.8 Exception Handling

Other exception handling constructs are represented using TRY_CATCH_EXPR. The handler operand of a TRY_CATCH_EXPR can be a normal statement to be executed if the controlled block throws an exception, or it can have one of two special forms:

- A CATCH_EXPR executes its handler if the thrown exception matches one of the allowed types. Multiple handlers can be expressed by a sequence of CATCH_EXPR statements.
- An EH_FILTER_EXPR executes its handler if the thrown exception does not match one of the allowed types.

Currently throwing an exception is not directly represented in GIMPLE, since it is implemented by calling a function. At some point in the future we will want to add some way to express that the call will throw an exception of a known type.

8 Example

```
struct A { A(); ~A(); };

int i;
int g();
void f ()
{
  A a;
  int j = (--i, i ? 0 : 1);

  for (int x = 42; x > 0; --x)
    {
      i += g()*4 + 32;
    }
}
```

becomes

```
void f() ()
{
  struct A * a.1;
  int iftmp.2;
  int T.3;
  int T.4;
  int T.5;
  struct A * a.6;

  {
    struct A a;
    int j;

    a.1 = &a;
    __comp_ctor (a.1);
    try
      {
        i = i - 1;
        if (i == 0)
          iftmp.2 = 1;
        else
          iftmp.2 = 0;
        j = iftmp.2;

        {
          int x;

          x = 42;
          while (1)
            {
              if (x <= 0)
                goto break_label;

              T.3 = g ();
              T.4 = T.3 * 4;
              T.5 = i + T.4;
              i = T.5 + 32;

              x = x - 1;
            };
          break_label:;
        }
      }
    finally
      {
        a.6 = &a;
        __comp_dtor (a.6);
      }
  }
}
```

9 Rough GIMPLE Grammar

```
function:
  FUNCTION_DECL
    DECL_SAVED_TREE → block
block:
  BIND_EXPR
    BIND_EXPR_VARS → DECL chain
    BIND_EXPR_BLOCK → BLOCK
    BIND_EXPR_BODY
      → compound-stmt
compound-stmt:
  COMPOUND_EXPR
    op0 → non-compound-stmt
    op1 → stmt
stmt: compound-stmt
  | non-compound-stmt
non-compound-stmt:
  block
  | loop-stmt
  | if-stmt
  | switch-stmt
  | jump-stmt
  | label-stmt
  | try-stmt
  | modify-stmt
  | call-stmt
loop-stmt:
  LOOP_EXPR
    LOOP_EXPR_BODY
      → stmt | NULL_TREE
  | DO_LOOP_EXPR
    (to be defined later)
if-stmt:
  COND_EXPR
    op0 → condition
    op1 → stmt
    op2 → stmt
switch-stmt:
  SWITCH_EXPR
    op0 → val
    op1 → stmt
    op2 → TREE_VEC of LABEL_DECLs
jump-stmt:
    GOTO_EXPR
      op0 → LABEL_DECL | ,*, ID
  | RETURN_EXPR
      op0 → modify-stmt
            | NULL_TREE
label-stmt:
  LABEL_EXPR
      op0 → LABEL_DECL
  | CASE_LABEL_EXPR
      CASE_LOW → val | NULL_TREE
      CASE_HIGH → val | NULL_TREE
      CASE_LABEL → LABEL_DECL
try-stmt:
  TRY_CATCH_EXPR
    op0 → stmt
    op1 → handler
  | TRY_FINALLY_EXPR
    op0 → stmt
    op1 → stmt
handler:
  catch-seq
  | EH_FILTER_EXPR
  | stmt
catch-seq:
  CATCH_EXPR
  | COMPOUND_EXPR
      op0 → CATCH_EXPR
      op1 → catch-seq
modify-stmt:
  MODIFY_EXPR
```

```
    op0 → lhs
    op1 → rhs
call−stmt: CALL_EXPR
  op0 → _DECL | ,&, _DECL
  op1 → arglist
arglist:
  NULL_TREE
  | TREE_LIST
      op0 → val
      op1 → arglist

varname : compref | _DECL
lhs: varname | ,*, _DECL
pseudo−lval: _DECL | ,*, _DECL
compref :
  COMPONENT_REF
    op0 → compref | pseudo−lval
  | ARRAY_REF
    op0 → compref | pseudo−lval
    op1 → val

condition : val | val relop val
val : _DECL | CONST

rhs: varname | CONST
   | ,*, _DECL
   | ,&, varname
   | call_expr
   | unop val
   | val binop val
   | ,(, cast ,), varname

(cast here stands for all valid C
typecasts.  Use of varname here seems
odd; it may change to val.)

unop: ,+, | ,-, | ,!, | ,~,

binop: relop | ,-, | ,+, | ,/, | ,*,
  | ,%, | ,&, | ,|, | ,«, | ,», | ,^,

relop: All tree codes of class ,<,
```

References

[SIMPLE] L. Hendren and C. Donawa and M. Emami and G. Gao and Justiani and B. Sridharan, *Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations*, Lecture Notes in Computing Science no. 757 (1992) p. 406-420

[Morgan] Robert Morgan. *Building an Optimizing Compiler*, Digital Press (1998).