GCC plugins through the MELT example

Basile STARYNKEVITCH
gcc-melt.org
basile@starynkevitch.net or basile.starynkevitch@cea.fr

CEA, LIST (Software Reliability Lab.), France
[soon within Université Paris Saclay]

March, 28th, 2014,
Linux Foundation Collaboration Summit (Napa Valley, California, USA)

Plan

1. Introduction
2. Writing simple plugins [in C++]
   - overview and hints
   - Gimple internal representation[s]
   - Tree internal representation[s]
   - Optimization passes
   - Gcc hooks for plugins
3. Extending GCC with MELT
   - MELT overview
   - Example pass in MELT
4. Advices and Conclusions
   - advices
   - why free software need GCC customization?

Caveat

All **opinions are mine only**

- I (Basile) don't speak for my employer, CEA (or my institute LIST)
- I don't speak for the GCC community
- I don't speak for anyone else
- My opinions may be highly controversial
- My opinions may change

Many slides, but some may be skipped...

1 Introduction

2 Writing simple plugins [in C++]
- overview and hints
- Gimple internal representation[s]
- Tree internal representation[s]
- Optimization passes
- Gcc hooks for plugins

3 Extending GCC with MELT
- MELT overview
- Example pass in MELT

4 Advices and Conclusions
- advices
- why free software need GCC customization?

What really is Gcc

Gnu Compiler Collection `gcc.gnu.org` (FSF copyrighted, GPLv3 licensed)

Gcc is mostly **working on** [various] **internal representations** of the *user code* it is currently compiling, much like a baker is kneading dough or pastry.

gcc & g++ drivers, `cc1` etc...

The **gcc** or **g++** [1] are driver programs. They are starting

- **cc1** (for *C*) or **cc1plus** ... for the compiler proper (includes preprocessing), emitting assembly code.
- **as** for the assembler
- **lto1** for **Link Time Optimization**
- the **ld** or **gold** linker [2]
- the **collect2** specific linker (creating a table of C++ constructors to be called for static data)

Run **g++ -v** instead of **g++** to understand what is going on.

GCC plugins are `dlopen`-ed by `cc1`, `cc1plus`, `lto1` ... So GCC "is mostly" `cc1plus`, or `cc1`, or `g951`, or `gnat1`, etc...

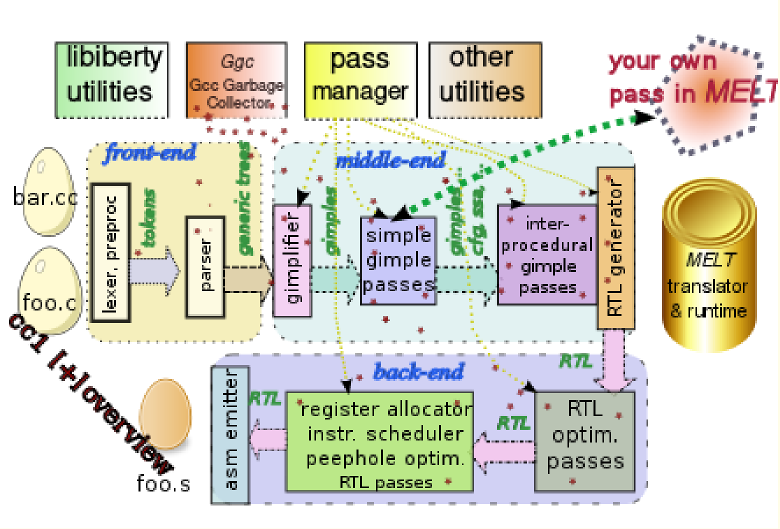
[1] And also **gccgo** for Go, **gfortran** for Fortran, **gnat** for Ada, **gdc** for D, etc...
[2] LTO may use linker plugins.

terminology: *Gimple*, *Trees*, ...

- **Gimple** = elementary (generally "3 addresses") virtual instruction, like `x := y + 3` or `x := f(z,t,u[3])` or `if (x > y) goto lab;`
- **SSA** = static single assignment (e.g. *Gimple/SSA*)
- **Basic Block** = elementary sequence of *Gimple*-s with one starting point
- **Tree** = abstract syntax tree (AST)[3]; operands of *Gimple* instructions are *Tree*-s
- **Generic** = those trees common to many source languages
- **RTL** = register transfer language (in backend)
- **CFG** = control flow graph
- **Edge** between two basic blocks
- **Loop** representation
- **pass** = optimization pass (often transforming *Gimple*)
  (Gcc has hundreds of passes)

[3]Before gimplification the entire compiled source code is represented in *Tree*-s.

internal picture of **cc1** & **cc1plus**



Using a plugin in Gcc

Legalese: **Gcc runtime library exception**
`http://www.gnu.org/licenses/gcc-exception`

from FAQ:

> *Lay the groundwork for a plugin infrastructure in GCC. For a while now, the GCC developers have considered adding a plugin framework to the compiler. This would make it easier for others to contribute to the project, and accelerate the development of new compilation techniques for GCC. However, there have also been concerns that **unscrupulous developers could write plugins that called out to proprietary software** to transform the compiled code—effectively creating proprietary extensions to GCC and defeating the purpose of the GPL. The updated exception prevents such **abuse**, enabling the GCC team to look forward to plugin developments.*

GCC plugins are expected [4] to be GPL compatible free software.
You want to develop your plugin in the open, e.g. to get help from the GCC community.

[4] You are probably not allowed to compile then distribute proprietary programs with a GCC augmented with proprietary plugins. But **I (Basile) am not a lawyer**; make your own opinion!

Using the MELT plugin in Gcc

MELT (see **gcc-melt.org**) is ...

- a high-level **domain specific language to extend or customize Gcc**
- implemented as a **Gcc plugin** (GPLv3 licensed, FSF copyrighted)
- also useful as a **grep** or **awk** like utility on Gcc internals
- an experimental Gcc *branch*
- a bootstrapped (Lisp-like) language translated to Gcc-friendly C++ code

Example "grep-like" compilation command using MELT:

```
gcc -fplugin=melt -fplugin-arg-melt-mode=findgimple \
   -fplugin-arg-melt-gimple-pattern='?(gimple_call_1 ?_
   ?(tree_function_decl_of_name "xmalloc" ?_ ?_)
    ?(tree_integer_cst ?(some_integer_greater_than 30)))'   \
   -Isomedir -DYOUR_CONST_PARAM=10 -O2 -c foo.c
```

will compile your `foo.c` and notice all calls to `xmalloc` with a size larger than 30 (**impossible** to do ***outside*** the compiler, e.g. with `grep` because of `sizeof` and constant expressions).

Why and when use Gcc plugins?

Take advantage of the power of the Gcc compiler and its internals representations (I.R.):

- some understanding of Gcc internals is needed...
- explore or **inspect the internal representations** of your program code as seen by Gcc (like `findgimple` example with MELT before)
  - source code navigation
  - specific coding rules validation (result of `open` should be checked)
  - API or library-specific warnings (check calls to variadic functions in Gtk?)
  - more sophisticated static code analysis

  can ignore most I.R.
- modify or **enhance the internal representations**
  - API or library-specific optimizations: `fprintf(stdout,...);` → `printf(...);`
  - "aspect oriented" programming
  - precise GC for C (glue code for local pointers handling)?

  it is usually harder, since all of I.R. should usually be handled

Plugins are useful for many *specific* things which should *not* go *inside* Gcc!

When Gcc plugins are useless?

- if you wished stability of Gcc internals:

> *the [Gcc] plugin interface simply exposes GCC internals, and as such is not stable across releases. [...] The most effective way for people to write plugins for GCC today is to use something like MELT (`http://gcc-melt.org`) or the GCC Python plugin (`https://fedorahosted.org/gcc-python-plugin/`). These provide a somewhat more standard interface across releases.*
>
> *Ian Taylor, on `gcc@gcc.gnu.org` list, January, 21st, 2014*

  So **your plugin is dependent**[5] **of the version of Gcc** (e.g. `gcc-4.8` vs `gcc-4.9`)

- to add a new front-end (like Gdc) to Gcc
- to add a new back-end (or target processor) to your Gcc
- to add your magical preprocessor macros (à la `__COUNTER__`)
- when your `gcc` does not enable plugins (try `gcc -v` to find out; look for `--enable-plugin`; report a distribution bug if missing)

[5] In practice, the dependency on the version number of `gcc` is not that huge.

a crashing plugin

Symbols **plugin_init** and **plugin_is_GPL_compatible** are required (and `dlsym`-ed by `cc1`, or `cc1plus` etc ...) in plugins.

```
// file crashplugin.cc in public domain
#include <cstdio>
#include <cstdlib>
#include "gcc-plugin.h"
#include "plugin-version.h"
#include "diagnostic.h"
int plugin_is_GPL_compatible=1;
extern "C" int plugin_init (struct plugin_name_args* pluginfo,
                            struct plugin_gcc_version* compvers) {
  fprintf(stderr, "plugin info full name %s\n", pluginfo->full_name);
  fprintf(stderr, "crashing GCC compiler version %s date %s\n",
          compvers->basever, compvers->datestamp);
  if (!pluginfo->version)
    pluginfo->version = "0.01-crashing";
  if (!pluginfo->help)
    pluginfo->help = "a plugin to crash your GCC";
  if (!plugin_default_version_check (compvers, &gcc_version))
    return 1;
  fatal_error("crashing plugin");
  return 0; /* not reached, 0 means ok */ }
```

compiling a plugin

On Debian (or Ubuntu, ...) first install appropriate packages `g++-4.8 gcc-4.8-plugin-dev`
Use `gcc -print-file-name=plugin` giving */usr/lib/gcc/x86_64-linux-gnu/4.8/plugin*

So compile the `crashplugin.cc` with

```
g++ -g -Wall -shared -fPIC \
      -I $(gcc -print-file-name=plugin)/include \
      crashplugin.cc -o crashplugin.so
```

Test it on some `helloworld.c` with

```
gcc -fplugin=./crashplugin.so -Wall -O -c helloworld.c
```

```
crashing GCC compiler version 4.8 date 20140217
cc1: fatal error: crashing plugin
compilation terminated.
```

Many **distribution specific `gcc`** compilers sometimes **give** incorrectly **a strange message** “***The bug is not reproducible, so it is likely a hardware or OS problem.***” **for plugin errors**. To distribution makers: please improve the wording by mentioning *plugins*!

About Gcc

- Gcc is **complex** (≈ 10 millions lines of source code) and old.
  - Gcc can be a cross-compiler
  - complexity of source languages (C, C++, Ada, Fortran) [multi-threading]
  - complexity of target processors [multi-core]
  - complexity and variety of compiled software
  - the **increasing gap between languages and processsors** requires powerful optimization[6] techniques

  Gcc is constantly **evolving** (≈ 400 maintainers, +3.96% more code from 4.7 to 4.8)
- Gcc is mostly written[7] in ***C++2003*** but most files are still named `*.c`
  Some of the code is generated by **internal C++ code generators**.
- read *Gcc internals* on `http://gcc.gnu.org/onlinedocs/gccint/`
- many tutorial resources on ***GCC Resource Center***, *IIT Bombay* `http://www.cse.iitb.ac.in/grc/`
- many other resources on the web, and the wiki `http://gcc.gnu.org/wiki`

[6] Recall that optimization is an undecidable (or at least untractable) problem!

[7] C++ is possible in Gcc since May 2010, effective since `gcc-4.8` on March 2013.

practical hints

- use a very recent[8] version of Gcc
- subscribe and read `gcc@gcc.gnu.org` archived on `http://gcc.gnu.org/ml/gcc/` (also, use IRC)
- for Melt subscribe to **`gcc-melt@googlegroups.com`**
- perhaps build [9] your debug version of Gcc from the FSF source tarball

```
../gcc-4.8.2/configure --enable-plugins --enable-checks \
        --disable-bootstrap --enable-languages=c,c++,lto \
        --program-suffix=-4.8-dbg 'CFLAGS=-g -O' 'CXXFLAGS=-O -g'
make -j 4
make install DESTDIR=/tmp/gccinst/
sudo cp -v -a /tmp/gccinst/usr/local/. /usr/local/.
```

- keep and study your Gcc source tree
- **publish *early* your plugin** (e.g. to easily get help), even in α stage.

[8] Plugin support is improving in Gcc, so is better in GCC 4.9 [to be released in spring 2014] than in 4.8 or 4.7!

[9] in a build tree *outside* of the `gcc-4.8.2/` source tree!

Gimple internal representation

Defined in commented file **`gcc/gimple.def`** of the Gcc source tree:

- *Gimple* are **internal data** in `cc1` memory and type **`gimple`** is a **pointer**: `typedef struct gimple_statement_base *gimple;` in file `gcc/coretypes.h`
- also defined by corresponding `struct` and the API of functions handling them in **`gcc/gimple.h`** etc ... (`gcc/gimple-*.h`)
- **40 cases**, i.e. invocation of macro `DEFGSCODE`, with 17 dedicated to OpenMP (e.g. `GIMPLE_OMP_FOR`)
- often a “3 address” virtual instruction with important exceptions: `GIMPLE_CALL`, `GIMPLE_PHI` for *SSA*, `GIMPLE_SWITCH`, ...
- *Gimple* operands are often *Tree*-s
- *Gimple*-s are linked together, so `typedef gimple gimple_seq;` and `gimple_seq` are `gimple`-s.

Gimple internal representation - conditional

(code from trunk on march 2014, i.e. future **GCC 4.9**)

```
/* GIMPLE_COND <COND_CODE, OP1, OP2, TRUE_LABEL, FALSE_LABEL>
   represents the conditional jump:

   if (OP1 COND_CODE OP2) goto TRUE_LABEL else goto FALSE_LABEL

   COND_CODE is the tree code used as the comparison predicate.  It
   must be of class tcc_comparison.

   OP1 and OP2 are the operands used in the comparison.  They must be
   accepted by is_gimple_operand.

   TRUE_LABEL and FALSE_LABEL are the LABEL_DECL nodes used as the
   jump target for the comparison.  */
DEFGSCODE(GIMPLE_COND, "gimple_cond", GSS_WITH_OPS)
```

Gimple I.R. - assignment & arithmetic

For C code like `x = y * z;` or `x = p->f;` or `x = y;` or `x = (int)y;`

```
/* GIMPLE_ASSIGN <SUBCODE, LHS, RHS1[, RHS2]> represents the assignment
   statement

   LHS = RHS1 SUBCODE RHS2.

   SUBCODE is the tree code for the expression computed by the RHS of the
   assignment.  It must be one of the tree codes accepted by
   get_gimple_rhs_class.  If LHS is not a gimple register according to
   is_gimple_reg, SUBCODE must be of class GIMPLE_SINGLE_RHS.

   LHS is the operand on the LHS of the assignment.  It must be a tree node
   accepted by is_gimple_lvalue.

   RHS1 is the first operand on the RHS of the assignment.  It must always be
   present.  It must be a tree node accepted by is_gimple_val.

   RHS2 is the second operand on the RHS of the assignment.  It must be a tree
   node accepted by is_gimple_val.  This argument exists only if SUBCODE is
   of class GIMPLE_BINARY_RHS.  */
DEFGSCODE(GIMPLE_ASSIGN, "gimple_assign", GSS_WITH_MEM_OPS)
```

Gimple I.R. - call & return

```
/* GIMPLE_CALL <FN, LHS, ARG1, ..., ARGN[, CHAIN]> represents function
   calls.

   FN is the callee.  It must be accepted by is_gimple_call_addr.

   LHS is the operand where the return value from FN is stored.  It may
   be NULL.

   ARG1 ... ARGN are the arguments.  They must all be accepted by
   is_gimple_operand.

    CHAIN is the optional static chain link for nested functions.  */
DEFGSCODE(GIMPLE_CALL, "gimple_call", GSS_CALL)
```

```
/* GIMPLE_RETURN <RETVAL> represents return statements.

   RETVAL is the value to return or NULL.  If a value is returned it
   must be accepted by is_gimple_operand.  */
DEFGSCODE(GIMPLE_RETURN, "gimple_return", GSS_WITH_MEM_OPS)
```

Gimple I.R. - goto, label, switch

```
/* GIMPLE_GOTO <TARGET> represents unconditional jumps.
   TARGET is a LABEL_DECL or an expression node for computed GOTOs.  */
DEFGSCODE(GIMPLE_GOTO, "gimple_goto", GSS_WITH_OPS)

/* GIMPLE_LABEL <LABEL> represents label statements.  LABEL is a
   LABEL_DECL representing a jump target.  */
DEFGSCODE(GIMPLE_LABEL, "gimple_label", GSS_WITH_OPS)
```

```
/* GIMPLE_SWITCH <INDEX, DEFAULT_LAB, LAB1, ..., LABN> represents the
   multiway branch:

   switch (INDEX)
   {
     case LAB1: ...; break;
     ...
     case LABN: ...; break;
     default: ...
   }

   INDEX is the variable evaluated to decide which label to jump to.

   DEFAULT_LAB, LAB1 ... LABN are the tree nodes representing case labels.
   They must be CASE_LABEL_EXPR nodes.  */
DEFGSCODE(GIMPLE_SWITCH, "gimple_switch", GSS_WITH_OPS)
```

Gimple I.R. - ϕ nodes in *SSA*

In **SSA** (static single assigment) form, each *SSA variable* is assigned once[10]
So "merging" values (coming from different control paths) within *phi nodes* is needed:

```
/* GIMPLE_PHI <RESULT, ARG1, ..., ARGN> represents the PHI node

   RESULT = PHI <ARG1, ..., ARGN>

   RESULT is the SSA name created by this PHI node.

   ARG1 ... ARGN are the arguments to the PHI node.  N must be
   exactly the same as the number of incoming edges to the basic block
   holding the PHI node.  Every argument is either an SSA name or a
   tree node of class tcc_constant.  */
DEFGSCODE(GIMPLE_PHI, "gimple_phi", GSS_PHI)
```

NB: a ϕ node might generate no code!

[10] So the compiler knows where an *SSA variable* is defined, and where it is used: **use-def chains**.

Dumping the **Gimple** I.R. - code example

Use the **`-fdump-tree-gimple`** or **`-fdump-tree-all`** program argument to `gcc` or `g++`, ... Our C source code example:

```
// file mean-stat.c
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdint.h>
// return the mean modification time of paths after a given time
time_t
mean_mtime_after (const char*pathtab[], unsigned nbpath, time_t aftertim) {
  int64_t sumtime=0;
  unsigned count=0;
  for (unsigned ix=0; ix<nbpath; ix++) {
    const char*curpath = pathtab[ix];
    if (!curpath) break;
    struct stat st= {0};
    if (stat (curpath, &st) || st.st_mtime < aftertim) continue;
    sumtime += (int64_t) st.st_mtime; // i.e. st.st_mtim.tv_sec
    count++;
  }
  return (count>0)?(time_t)(sumtime/count):-1; }
```

Compile with **`gcc -std=c99 -Wall -O -fdump-tree-all -c mean-stat.c`**

Dumping the **Gimple** I.R. - many dump files

Getting about **85 dump files** like [incomplete list] :

```
mean-stat.c.033t.profile_estimate   mean-stat.c.088t.copyprop4
mean-stat.c.249t.statistics         mean-stat.c.089t.sincos
mean-stat.c.134t.phiopt3            mean-stat.c.091t.crited1
mean-stat.c.135t.fab1               mean-stat.c.093t.sink
mean-stat.c.138t.copyrename4        mean-stat.c.096t.loop
mean-stat.c.139t.crited2            mean-stat.c.097t.loopinit
mean-stat.c.140t.uninit1            mean-stat.c.098t.lim1
mean-stat.c.058t.phiprop            mean-stat.c.011t.cfg
mean-stat.c.059t.forwprop2          mean-stat.c.015t.ssa
mean-stat.c.071t.phiopt1            mean-stat.c.061t.alias
```

Anatomy of dump file name **`mean-stat.c.134t.phiopt3`** :

- `mean-stat.c` prefix: the same base name as the compiled source
- `.134t` unique infix : a **meaningless** **unique** number[11] + dump type
- `.phiopt` suffix : print name of some internal Gcc pass producing that dump
- `3` optional pass numbering : if the same pass is invoked several times

[11] That dump file number is **not** *chronological* and not logical!

Gimple I.R. dumps - caveat

Important things to know:

- dump files show a **partial textual view** of internal representations (the in memory representation is richer - it is a complex **graph of data structures** -, e.g. knows about source location, containing basic blocks, etc . . .)
- Gcc optimization passes are **enriching** & **modifying** the I.R.
- in some [early] passes the **I.R. may be incomplete** (and some early I.R. might become lost in later passes)
- some passes are **transforming** and **replacing** the I.R.
- textual dumps are useful to give a glimpse, but what really matters to plugins (and to Gcc middle-end) is the **in memory data internal representations**

(dump files are **verbose**; we are reformatting and/or editing them.)

upper Gimple I.R. `mean-stat.c.004t.gimple`

```
mean_mtime_after (const char ** pathtab,
      unsigned int nbpath, time_t aftertim) {
 long unsigned int D.2231, D.2232;
 const char ** D.2233; int D.2237; long int D.2239;
 time_t D.2240, iftmp.0; long int D.2244;
 int64_t sumtime; unsigned int count;
 sumtime = 0;
 count = 0;
 { unsigned int ix;
   ix = 0;
   goto <D.2229>;
   <D.2228>:
   { const char * curpath; struct stat st;
     try {
       D.2231 = (long unsigned int) ix;
       D.2232 = D.2231 * 8;
       D.2233 = pathtab + D.2232;
       curpath = *D.2233;
       if (curpath == 0B) goto <D.2225>;
       else goto <D.2234>;
      <D.2234>:
       st = {};
       D.2237 = stat (curpath, &st);
       if (D.2237 != 0) goto <D.2235>;
       else goto <D.2238>;
       <D.2238>:
       D.2239 = st.st_mtime;
       if (D.2239 < aftertim) goto <D.2235>;
       else goto <D.2236>;
       <D.2235>:
       // predicted unlikely by continue predictor
       goto <D.2227>;
       <D.2236>:
       D.2239 = st.st_mtime;
       sumtime = D.2239 + sumtime;
       count = count + 1;
     } finally {
         st = {CLOBBER}; } }
   <D.2227>:
   ix = ix + 1;
   <D.2229>:
   if (ix < nbpath) goto <D.2228>;
   else goto <D.2225>;
   <D.2225>: }
 if (count != 0) goto <D.2242>;
 else goto <D.2243>;
 <D.2242>:
 D.2244 = (long int) count;
 iftmp.0 = sumtime / D.2244;
 goto <D.2245>;
 <D.2243>:
 iftmp.0 = -1;
 <D.2245>:
 D.2240 = iftmp.0;
 return D.2240; }
// missing stat function below
```

SSA Gimple I.R. `mean-stat.c.015t.ssa` partly

```
mean_mtime_after (const char ** pathtab,
        unsigned int nbpath, time_t aftertim)
{ struct stat st; const char * curpath;
  unsigned int ix, count;
  int64_t sumtime;
  time_t iftmp.0_6;
  long unsigned int _15, _16;
  const char * * _18;
  int _22;
  long int _23, _25, _32;
  time_t iftmp.0_33, iftmp.0_34, _35;

  <bb 2>:
  sumtime_10 = 0;
  count_11 = 0;
  ix_12 = 0;
  goto <bb 10>;

  <bb 3>:
  _15 = (long unsigned int) ix_5;
  _16 = _15 * 8;
  _18 = pathtab_17(D) + _16;
  curpath_19 = *_18;
  if (curpath_19 == 0B)
    goto <bb 8>;
  else
    goto <bb 4>;
```

```
<bb 4>:
st = {};
_22 = stat (curpath_19, &st);
if (_22 != 0)
  goto <bb 6>;
else
  goto <bb 5>;

<bb 5>:
_23 = st.st_mtime;
if (_23 < aftertim_24(D))
  goto <bb 6>;
else
  goto <bb 7>;

<bb 6>:
// predicted unlikely by continue predictor.
st ={v} {CLOBBER};
goto <bb 9>;

<bb 7>:
_25 = st.st_mtime;
sumtime_26 = _25 + sumtime_2;
count_27 = count_4 + 1;
st ={v} {CLOBBER};
goto <bb 9>;
```

ϕ-optimized **Gimple** I.R. `mean-stat.c.134t.phiopt3` partly

```
mean_mtime_after (const char ** pathtab,
    unsigned int nbpath, time_t aftertim){
  unsigned long ivtmp.7; struct stat st;
  const char * curpath; unsigned int ix, count;
  int64_t sumtime; time_t iftmp.0_6;
  const char * * _8; long int _19, _27;
  time_t iftmp.0_28;  int _29; void * _30;
  unsigned int _31; sizetype _32, _49, _50;
  const char ** _51; unsigned long _52;

  <bb 2>:
  if (nbpath_11(D) != 0)
    goto <bb 3>;
  else goto <bb 14>;

  <bb 3>:
  curpath_37 = *pathtab_14(D);
  if (curpath_37 == 0B)
    goto <bb 10>;
  else goto <bb 4>;

  <bb 4>:
  _8 = pathtab_14(D) + 8;
  ivtmp.7_5 = (unsigned long) _8;
  _31 = nbpath_11(D) + 4294967295;
  _32 = (sizetype) _31;
  _49 = _32 + 1;
  _50 = _49 * 8;
  _51 = pathtab_14(D) + _50;
  _52 = (unsigned long) _51;
  goto <bb 6>;

  <bb 5>:
  ivtmp.7_4 = ivtmp.7_2 + 8;
  _30 = (void *) ivtmp.7_4;
  curpath_16 = MEM[base: _30, offset: -8B];
  if (curpath_16 == 0B)
    goto <bb 10>;
  else goto <bb 6>;

  <bb 6>:
  # sumtime_38 = PHI <0(4), sumtime_1(5)>
  # count_41 = PHI <0(4), count_3(5)>
  # curpath_47 = PHI <curpath_37(4), curpath_16(5)>
  # ivtmp.7_2 = PHI <ivtmp.7_5(4), ivtmp.7_4(5)>
  st = {};
  _29 = __xstat (1, curpath_47, &st);
  if (_29 != 0)
    goto <bb 8>;
  else goto <bb 7>;

  <bb 7>:
  _19 = st.st_mtime;
  if (_19 < aftertim_20(D))
    goto <bb 8>;
  else goto <bb 9>;
  // etc
```

Tree internal representations

Look first inside **`gcc/tree.def`**: **211** common **tree codes** defined with `DEFTREECODE`, notably the **operators inside *Gimple*** instructions[12]. Called **Generic** since language independent.

Then look inside **`gcc/c-family/c-common.def`**: 5 more tree codes [13] for C family (C, C++, Objective C . . .) languages, like `SIZEOF_EXPR`.

Also (for C++ AST) in **`gcc/cp/cp-tree.def`**, 74 more tree codes for C++ front-end, e.g. `NEW_EXPR` [14] or `IF_STMT`.

etc . . .

---

macro usage:

`DEFTREECODE` **(*tree_code* , *print_string* , *tree_class* , *arity*)**

[12]Before gimplification the source code abstract syntax tree (AST) is represented by *Tree*-s.
[13]near the front-end
[14]for `operator new`...

Tree I.R. - simple nodes

```
/* Any erroneous construct is parsed into a node of this type.
   This type of node is accepted without complaint in all contexts
   by later parsing activities, to avoid multiple error messages
   for one error.
   No fields in these nodes are used except the TREE_CODE.  */
DEFTREECODE (ERROR_MARK, "error_mark", tcc_exceptional, 0)

/* Used to represent a name (such as, in the DECL_NAME of a decl node).
   Internally it looks like a STRING_CST node.
   There is only one IDENTIFIER_NODE ever made for any particular name.
   Use `get_identifier' to get it (or create it, the first time).  */
DEFTREECODE (IDENTIFIER_NODE, "identifier_node", tcc_exceptional, 0)
/* First, the constants.  */

/* Contents are in TREE_INT_CST_LOW and TREE_INT_CST_HIGH fields,
   32 bits each, giving us a 64 bit constant capability.  INTEGER_CST
   nodes can be shared, and therefore should be considered read only.
   They should be copied, before setting a flag such as TREE_OVERFLOW.
   If an INTEGER_CST has TREE_OVERFLOW already set, it is known to be unique.
   INTEGER_CST nodes are created for the integral types, for pointer
   types and for vector and float types in some circumstances.  */
DEFTREECODE (INTEGER_CST, "integer_cst", tcc_constant, 0)

/* Contents are in TREE_REAL_CST field.  */
DEFTREECODE (REAL_CST, "real_cst", tcc_constant, 0)
```

Tree I.R. - function declarations

```
/* Declarations.  All references to names are represented as ..._DECL
   nodes.  The decls in one binding context are chained through the
   TREE_CHAIN field.  Each DECL has a DECL_NAME field which contains
   an IDENTIFIER_NODE.  (Some decls, most often labels, may have zero
   as the DECL_NAME).  DECL_CONTEXT points to the node representing
   the context in which this declaration has its scope.  For
   FIELD_DECLs, this is the RECORD_TYPE, UNION_TYPE, or
   QUAL_UNION_TYPE node that the field is a member of.  For VAR_DECL,
   PARM_DECL, FUNCTION_DECL, LABEL_DECL, and CONST_DECL nodes, this
   points to either the FUNCTION_DECL for the containing function, the
   RECORD_TYPE or UNION_TYPE for the containing type, or NULL_TREE or
   a TRANSLATION_UNIT_DECL if the given decl has "file scope".
   /// ....
   FUNCTION_DECLs use four special fields:
   DECL_ARGUMENTS holds a chain of PARM_DECL nodes for the arguments.
   DECL_RESULT holds a RESULT_DECL node for the value of a function.
    The DECL_RTL field is 0 for a function that returns no value.
    (C functions returning void have zero here.)
    The TREE_TYPE field is the type in which the result is actually
    returned.  This is usually the same as the return type of the
    FUNCTION_DECL, but it may be a wider integer type because of
    promotion.
   DECL_FUNCTION_CODE is a code number that is nonzero for
    built-in functions.  Its value is an enum built_in_function
    that says which built-in function it is.
   DECL_SOURCE_FILE holds a filename string and DECL_SOURCE_LINE
   holds a line number.  In some cases these can be the location of
   a reference, if no definition has been seen.  */
DEFTREECODE (FUNCTION_DECL, "function_decl", tcc_declaration, 0)
```

Tree I.R. - reference to storage

```
/* Value is structure or union component.
   Operand 0 is the structure or union (an expression).
   Operand 1 is the field (a node of type FIELD_DECL).
   Operand 2, if present, is the value of DECL_FIELD_OFFSET, measured
   in units of DECL_OFFSET_ALIGN / BITS_PER_UNIT.  */
DEFTREECODE (COMPONENT_REF, "component_ref", tcc_reference, 3)

/* Reference to a group of bits within an object.  Similar to COMPONENT_REF
   except the position is given explicitly rather than via a FIELD_DECL.
   Operand 0 is the structure or union expression;
   operand 1 is a tree giving the constant number of bits being referenced;
   operand 2 is a tree giving the constant position of the first referenced bit.
   The result type width has to match the number of bits referenced.
   If the result type is integral, its signedness specifies how it is extended
   to its mode width.  */
   DEFTREECODE (BIT_FIELD_REF, "bit_field_ref", tcc_reference, 3)

/* Array indexing.
   Operand 0 is the array; operand 1 is a (single) array index.
   Operand 2, if present, is a copy of TYPE_MIN_VALUE of the index.
   Operand 3, if present, is the element size, measured in units of
   the alignment of the element type.  */
   DEFTREECODE (ARRAY_REF, "array_ref", tcc_reference, 4)

/* C unary '*' or Pascal '^'.  One operand, an expression for a pointer.  */
DEFTREECODE (INDIRECT_REF, "indirect_ref", tcc_reference, 1)
```

Tree I.R. - some binary arithmetic operations

```
/* Simple arithmetic.  */
DEFTREECODE (PLUS_EXPR, "plus_expr", tcc_binary, 2)
DEFTREECODE (MINUS_EXPR, "minus_expr", tcc_binary, 2)
DEFTREECODE (MULT_EXPR, "mult_expr", tcc_binary, 2)
/* Pointer addition.  The first operand is always a pointer and the
   second operand is an integer of type sizetype.  */
DEFTREECODE (POINTER_PLUS_EXPR, "pointer_plus_expr", tcc_binary, 2)

/* Highpart multiplication.  For an integral type with precision B,
   returns bits [2B-1, B] of the full 2*B product.  */
DEFTREECODE (MULT_HIGHPART_EXPR, "mult_highpart_expr", tcc_binary, 2)

/* Division for integer result that rounds the quotient toward zero.  */
DEFTREECODE (TRUNC_DIV_EXPR, "trunc_div_expr", tcc_binary, 2)

/* Division for integer result that rounds the quotient toward infinity.  */
DEFTREECODE (CEIL_DIV_EXPR, "ceil_div_expr", tcc_binary, 2)
/// ...
/* Bitwise operations.  Operands have same mode as result.  */
DEFTREECODE (BIT_IOR_EXPR, "bit_ior_expr", tcc_binary, 2)
DEFTREECODE (BIT_XOR_EXPR, "bit_xor_expr", tcc_binary, 2)
DEFTREECODE (BIT_AND_EXPR, "bit_and_expr", tcc_binary, 2)
/// ...
/* Minimum and maximum values.  When used with floating point, if both
   operands are zeros, or if either operand is NaN, then it is unspecified
   which of the two operands is returned as the result.  */
DEFTREECODE (MIN_EXPR, "min_expr", tcc_binary, 2)
DEFTREECODE (MAX_EXPR, "max_expr", tcc_binary, 2)
```

Tree I.R. - some unary arithmetic operations

```
/* Conversion of real to fixed point by truncation.  */
DEFTREECODE (FIX_TRUNC_EXPR, "fix_trunc_expr", tcc_unary, 1)
/* Conversion of an integer to a real.  */
DEFTREECODE (FLOAT_EXPR, "float_expr", tcc_unary, 1)
/* Unary negation.  */
DEFTREECODE (NEGATE_EXPR, "negate_expr", tcc_unary, 1)
/* Represents the absolute value of the operand.
   An ABS_EXPR must have either an INTEGER_TYPE or a REAL_TYPE.  The
   operand of the ABS_EXPR must have the same type.  */
DEFTREECODE (ABS_EXPR, "abs_expr", tcc_unary, 1)
/// bitwise not
DEFTREECODE (BIT_NOT_EXPR, "bit_not_expr", tcc_unary, 1)
/* Represents a re-association barrier for floating point expressions
   like explicit parenthesis in fortran.  */
DEFTREECODE (PAREN_EXPR, "paren_expr", tcc_unary, 1)
/* Represents a conversion of type of a value.
   All conversions, including implicit ones, must be
   represented by CONVERT_EXPR or NOP_EXPR nodes.  */
DEFTREECODE (CONVERT_EXPR, "convert_expr", tcc_unary, 1)
/* Represents a conversion expected to require no code to be generated.  */
DEFTREECODE (NOP_EXPR, "nop_expr", tcc_unary, 1)
/// ....
/* Unpack (extract) the high/low elements of the input vector, convert
   fixed point values to floating point and widen elements into the
   output vector.  The input vector has twice as many elements as the output
   vector, that are half the size of the elements of the output vector.  */
DEFTREECODE (VEC_UNPACK_FLOAT_HI_EXPR, "vec_unpack_float_hi_expr", tcc_unary, 1)
DEFTREECODE (VEC_UNPACK_FLOAT_LO_EXPR, "vec_unpack_float_lo_expr", tcc_unary, 1)
```

Tree I.R. - tree code classes

In `gcc/tree-core.h`:

```
/* Tree code classes.  Each tree_code has an associated code class
   represented by a TREE_CODE_CLASS.  */
enum tree_code_class {
  tcc_exceptional, /* An exceptional code (fits no category).  */
  tcc_constant,    /* A constant.  */
  /* Order of tcc_type and tcc_declaration is important.  */
  tcc_type,        /* A type object code.  */
  tcc_declaration, /* A declaration (also serving as variable refs).  */
  tcc_reference,   /* A reference to storage.  */
  tcc_comparison,  /* A comparison expression.  */
  tcc_unary,       /* A unary arithmetic expression.  */
  tcc_binary,      /* A binary arithmetic expression.  */
  tcc_statement,   /* A statement expression, which have side effects
                      but usually no interesting value.  */
  tcc_vl_exp,      /* A function call or other expression with a
                      variable-length operand vector.  */
  tcc_expression   /* Any other expression.  */
};
```

NB: ***Tree*-s and *Gimple*-s are carefully crafted** (changing rarely) **data structures** !

Gcc optimization passes

- Gcc has **many** ($\approx$ 275) **optimization passes** (organized in a tree, with some passes running sub-passes)
- **Every compilation**[15] **runs a lot of them**, try flag **`-fdump-passes`**
- the actually running passes **depend** upon the **optimization flags** (`-O2` or `-O0` etc . . .) and **dynamically** upon the compiled source code.
- Four types of passes (in **`gcc/tree-pass.h`**):
  1. **`GIMPLE_PASS`** : simple (intra-procedural) **Gimple pass** (on a single function, pointed by **`cfun`**)
  2. **`SIMPLE_IPA_PASS`** : **simple inter-procedural analysis** (IPA) **pass**
  3. **`IPA_PASS`** : **full IPA pass** (may be LTO related)
  4. **`RTL_PASS`** : **RTL** (back-end) **pass**
- **Plugins can add**, remove, or reorganize **passes**
- Some passes are highly specialized (e.g. `sincos` pass for *sin* and *cos* and power), others are very general (e.g. `phiopt` running several times).
- See files **`gcc/passes.def`**, **`gcc/tree-passes.h`**, **`gcc/passes.c`**, **`gcc/pass_manager.h`** and **`gcc/tree-optimize.c`**

[15] Even without optimization

Passes definition - `gcc/passes.def` partly

```
/*
 Macros that should be defined when using this file:
   INSERT_PASSES_AFTER (PASS)
   PUSH_INSERT_PASSES_WITHIN (PASS)
   POP_INSERT_PASSES ()
   NEXT_PASS (PASS)
   TERMINATE_PASS_LIST ()
 */
 /* All passes needed to lower the function into shape optimizers can
    operate on.  These passes are always run first on the function, but
    backend might produce already lowered functions that are not processed
    by these passes.  */
  INSERT_PASSES_AFTER (all_lowering_passes)
  NEXT_PASS (pass_warn_unused_result);
  NEXT_PASS (pass_diagnose_omp_blocks);
  NEXT_PASS (pass_diagnose_tm_blocks);
  NEXT_PASS (pass_lower_omp);
  /// ... etc
  NEXT_PASS (pass_build_cfg);
  NEXT_PASS (pass_warn_function_return);
  NEXT_PASS (pass_expand_omp);
  NEXT_PASS (pass_build_cgraph_edges);
  TERMINATE_PASS_LIST ()
  /* Interprocedural optimization passes.  */
  INSERT_PASSES_AFTER (all_small_ipa_passes)
  NEXT_PASS (pass_ipa_free_lang_data);
  NEXT_PASS (pass_ipa_function_and_variable_visibility);
  NEXT_PASS (pass_early_local_passes);
  PUSH_INSERT_PASSES_WITHIN (pass_early_local_passes)
      NEXT_PASS (pass_fixup_cfg);
      NEXT_PASS (pass_init_datastructures);
```

pass instances

- indirect sub-classes of **`gcc::opt_pass`** from `gcc/tree-pass.h`
- **`name`** field: terse name[16] of the pass used as a fragment of the dump file name (unless starting with ⋆)
- **`gate`** function with `has_gate` flag: decide if the pass should be executed.
- **`execute`** function with `has_execute` flag: do the real work (changing the I.R.)
- **properties**: bit flags for required, provided, destroyed properties, e.g. `PROP_cfg` or `PROP_ssa`, etc...
- **todo**: bit flags for things to do before start or after finishing the pass e.g. `TODO_do_not_ggc_collect`, or `TODO_update_ssa`, or `TODO_verify_flow`, etc...
- **full IPA passes** have much more LTO related hooks: e.g. `generate_summary`, `write_summary`, `read_summary`, `function_transform` etc...

[16] The pass name is not always immediately related to identifiers inside the Gcc source code!

Gcc plugin **events** in **gcc/plugin.def** (part 1 of 2)

```
/* To hook into pass manager.  */
DEFEVENT (PLUGIN_PASS_MANAGER_SETUP)
/* After finishing parsing a type.  */
DEFEVENT (PLUGIN_FINISH_TYPE)
/* After finishing parsing a declaration. */
DEFEVENT (PLUGIN_FINISH_DECL)
/* Useful for summary processing.  */
DEFEVENT (PLUGIN_FINISH_UNIT)
/* Allows to see low level AST in C and C++ frontends. */
DEFEVENT (PLUGIN_PRE_GENERICIZE)
/* Called before GCC exits.  */
DEFEVENT (PLUGIN_FINISH)
/* Information about the plugin. */
DEFEVENT (PLUGIN_INFO)
/* Called at start of GCC Garbage Collection. */
DEFEVENT (PLUGIN_GGC_START)
/* Extend the GGC marking. */
DEFEVENT (PLUGIN_GGC_MARKING)
/* Called at end of GGC. */
DEFEVENT (PLUGIN_GGC_END)
/* Register an extra GGC root table. */
DEFEVENT (PLUGIN_REGISTER_GGC_ROOTS)
/* Register an extra GGC cache table. */
DEFEVENT (PLUGIN_REGISTER_GGC_CACHES)
/* Called during attribute registration.  */
DEFEVENT (PLUGIN_ATTRIBUTES)
```

Gcc plugin **events** in `gcc/plugin.def` (part 2 of 2)

```
/* Called before processing a translation unit.  */
DEFEVENT (PLUGIN_START_UNIT)
/* Called during pragma registration.  */
DEFEVENT (PLUGIN_PRAGMAS)
/* Called before first pass from all_passes.  */
DEFEVENT (PLUGIN_ALL_PASSES_START)
/* Called after last pass from all_passes.  */
DEFEVENT (PLUGIN_ALL_PASSES_END)
/* Called before first ipa pass.  */
DEFEVENT (PLUGIN_ALL_IPA_PASSES_START)
/* Called after last ipa pass.  */
DEFEVENT (PLUGIN_ALL_IPA_PASSES_END)
/* Allows to override pass gate decision for current_pass.  */
DEFEVENT (PLUGIN_OVERRIDE_GATE)
/* Called before executing a pass.  */
DEFEVENT (PLUGIN_PASS_EXECUTION)
/* Called before executing subpasses of a GIMPLE_PASS in
   execute_ipa_pass_list.  */
DEFEVENT (PLUGIN_EARLY_GIMPLE_PASSES_START)
/* Called after executing subpasses of a GIMPLE_PASS in
   execute_ipa_pass_list.  */
DEFEVENT (PLUGIN_EARLY_GIMPLE_PASSES_END)
/* Called when a pass is first instantiated.  */
DEFEVENT (PLUGIN_NEW_PASS)
/* Called when a file is #include-d or given thru #line directive.
   Could happen many times.  The event data is the included file path,
   as a const char* pointer.  */
DEFEVENT (PLUGIN_INCLUDE_FILE)
```

your hook for an event

Most events[17] are for **your hooks** or **callbacks**, called **from inside Gcc** (actually from `cc1` etc...) thru `invoke_plugin_callbacks`

For some events[18], Gcc is passing a pointer[19] to your hook (otherwise `NULL`).

Advanced [meta-] plugins might register new events with `get_named_event_id` from `gcc/gcc-plugin.h`.

Signature of your plugin hooks:

```
/* Function type for a plugin callback routine.

   GCC_DATA  - event-specific data provided by GCC
   USER_DATA - plugin-specific data provided by the plugin  */

typedef void (*plugin_callback_func) (void *gcc_data, void *user_data);
```

[17] Except `PLUGIN_PASS_MANAGER_SETUP`, `PLUGIN_INFO`, `PLUGIN_REGISTER_GGC_ROOTS`, `PLUGIN_REGISTER_GGC_CACHES`

[18] `PLUGIN_NEW_PASS`, `PLUGIN_OVERRIDE_GATE`, `PLUGIN_PASS_EXECUTION`, `PLUGIN_FINISH_DECL`, `PLUGIN_FINISH_TYPE`, `PLUGIN_PRE_GENERICIZE`

[19] The type and role of the pointed Gcc data is specific to the event.

Registering (& unregistering) your callback for an event

Often, your **plugin_init** will register a callback by calling:

```
/* This is also called without a callback routine for the
   PLUGIN_PASS_MANAGER_SETUP, PLUGIN_INFO, PLUGIN_REGISTER_GGC_ROOTS and
   PLUGIN_REGISTER_GGC_CACHES pseudo-events, with a specific user_data.
  */

extern void register_callback (const char *plugin_name,
                               int event,
                               plugin_callback_func callback,
                               void *user_data);
```

Notice that you need to register a callback -called once- even to **add your pragmas or attributes**; you could not call **register_attribute** or **c_register_pragma** from your `plugin_init`; that would be too early!

Later you might [rarely] want to remove your callback using

```
extern int unregister_callback (const char *plugin_name, int event);
```

Adding a new pass from your plugin

To add your pass after the first pass named **ssa** :

1. define and fill your static pass meta-data **const pass_data your_pass_data = ...** (with pass name, todo, properties, gate, execute . . .)
2. define **YourPassClass**, sub-class of `gimple_opt_pass`, using the shared `your_pass_data` etc ...
3. define a factory function e.g.

```
static gimple_opt_pass *makeyourpass (gcc::context *ctxt)
{ return new YourPassClass(ctxt); }
```

4. declare a **struct register_pass_info yourpassinfo;** and fill it:

```
yourpassinfo.pass = makeyourpass (gcc::g);
yourpassinfo.reference_pass_name = "ssa";
yourpassinfo.ref_pass_instance_number = 1;
yourpassinfo.pos_op = PASS_POS_INSERT_AFTER;
```

   Notice that `gcc::g` is a global pointer from **gcc/context.h**

5. register this new pass (near the end of your `plugin_init`)

```
register_callback (plugin_name, PLUGIN_PASS_MANAGER_SETUP, NULL,
                   &yourpassinfo);
```

Adding your attributes in your plugin

(function or variable attributes are a Gcc extension, or in *C++11*, e.g. `format` or `noreturn`)

To add an attribute **`foo`**:

1. define your attribute handler callback

```
static tree handle_foo_attribute (tree *node, tree name, tree args,
                                  int flags, bool *no_add_attrs);
```

2. define your attribute specification (see **`gcc/tree-core.h`**)

```
static struct attribute_spec foo_attr =
  { "foo", 1, 1, false,  false, false, handle_foo_attribute, false };
```

3. define & register (in your `plugin_init`) a callback for `PLUGIN_ATTRIBUTES`

```
static void register_your_attributes (void *, void *userdata)
{  register_attribute (&foo_attr); }
```

4. do something useful with your attribute, perhaps at `PLUGIN_PRE_GENERICIZE` or in some pass . . .

Adding your builtins or pragmas in your plugin

(builtin "functions" are compiled specially, e.g. `__builtin_bswap16`)

To add your builtins, at `PLUGIN_START_UNIT` time, call **`add_builtin_function`** from **`gcc/langhook.h`**

(**`#pragma`** preprocessor directives or **`_Pragma`** operators convey specific hints to the compiler)

To add your pragmas, at `PLUGIN_PRAGMAS` time, call appropriately some of the functions from **`gcc/c-family/c-pragma.h`** e.g.

```
/* Front-end wrappers for pragma registration.  */
typedef void (*pragma_handler_1arg)(struct cpp_reader *);
/* A second pragma handler, which adds a void * argument allowing to pass extra
   data to the handler.  */
typedef void (*pragma_handler_2arg)(struct cpp_reader *, void *);
extern void c_register_pragma (const char *space, const char *name,
                               pragma_handler_1arg handler);
extern void c_register_pragma_with_data (const char *space, const char *name,
                                         pragma_handler_2arg handler,
                                         void *data);
extern void c_register_pragma_with_expansion (const char *space,
                                              const char *name,
                                              pragma_handler_1arg handler);
extern void c_register_pragma_with_expansion_and_data (const char *space,
                                                       const char *name,
                                                  pragma_handler_2arg handler,
                                                       void *data);
```

Front-end functions unavailable from `lto1`

Your plugin cannot be `dlopen`-ed by **`lto1`** if it references (as hard symbols) front-end functions (pragma, attribute, builtin related) because `lto1` don't contain any front-end.

Workaround (plugin working with both `cc1plus` and `lto1`): use **weak symbols**!

```
// Function pragma_lex is declared in c-family/c-pragma.h
extern enum cpp_ttype pragma_lex (tree *) __attribute__((weak));
// Function c_register_pragma_with_expansion_and_data from c-family/c-pragma.h
extern void
c_register_pragma_with_expansion_and_data (const char *space,
                                           const char *name,
                                           pragma_handler_2arg handler,
                                           void *data)  __attribute__((weak));
```

Then later in your plugin hook for `PLUGIN_PRAGMAS`

```
      if (c_register_pragma_with_expansion_and_data) {
        //// really register your pragmas
        c_register_pragmas_with_expansion_and_data
            ("yourspace", "foobar", yourpragmahandler, yourdata);
      }
```

Memory management : Ggc (*Gcc* garbage collector)

Gcc has a (IMHO somehow poor) precise mark-&-sweep **garbage collector**:

- many types[20] or globals are annotated with **GTY**, e.g. (in gcc/ipa-ref.h)

```
struct GTY(()) ipa_ref {
  symtab_node *referring;
  symtab_node *referred;
  gimple stmt;
  unsigned int lto_stmt_uid;
  unsigned int referred_index;
  ENUM_BITFIELD (ipa_ref_use) use:2;
  unsigned int speculative:1;
};
```

- the **gengtype** *C++* code generator emits allocating & marking routines
- A garbage collection is **explicitly** called (thru **ggc_collect** from **gcc/ggc.h**), usually *between passes*; some passes call ggc_free **!!**
- but **locals are not handled** (so could be lost by GC)
- so Ggc is not very used (at least usually not for “temporary” data *inside* passes)

Plugins could use Ggc: PLUGIN_REGISTER_GGC_ROOTS, PLUGIN_GGC_MARKING, ...

[20] Including tree-s, basic_block-s, edge-s, gimple-s etc ...; ≈ 2000 types are GTY-ed!

Why would you use MELT?

Coding your plugins in *C++* may be painful, because

- Gcc I.R. is complex
- plugins often want to **find patterns** in it (e.g. on **Gimple** or **Tree**-s)
- manually managing memory can be error-prone (and Ggc is not funny)
- meta-programming facilities can be needed

MELT is a domain specific language to extend Gcc:

- with **strong pattern matching** facilities
- simple Lisp-like look & feel : closures, **powerful macros**[21], **homoiconic**
- "dynamically" typed (for its values) and **garbage collected**
- internally **translated to *C++*** with ability to **mix *C++* with Melt code**
  (Sometimes, the **generated *C++* code is compiled and `dlopen`-ed on the fly**!)

But MELT has a Lisp-like syntax : **(** *operator* *operands* ... **)**
(which should look familiar if you know Lisp or Scheme)

[21] In the Lisp sense: **arbitrary meta-programming** by runtime s-expr generation!

Hello, world in **MELT**

(for MELT 1.1 to be released soon, when GCC 4.9 is)

```
;; file helloworld.melt in the public domain
(module_is_gpl_compatible "public domain")
(let ( (:cstring who (or (melt_argument "i-am") "world")) )
       (code_chunk sayhello_chk #{ /* in $SAYHELLO_CHK */
            printf("MELT hello to %s\n", $WHO); }#))
```

Run it with the following command:

```
gcc -fplugin=melt -fplugin-arg-melt-mode=runfile \
    -fplugin-arg-melt-arg=helloworld.melt \
    -fplugin-arg-melt-i-am=Basile -c empty.c
```

getting (partly) with generation of *C++* code with `/* in SAYHELLO_CHK01 */` ... then compilation and `dlopen`-ing

```
cc1: note: MELT generated new file /tmp/filerHkNQj-GccMeltTmp-50823f/helloworld.cc
cc1: note: MELT plugin has built module helloworld flavor quicklybuilt in /home/basil
MELT hello to Basile
cc1: note: MELT removed 7 temporary files from /tmp/filerHkNQj-GccMeltTmp-50823f
```

MELT values vs stuff !

MELT brings you **dynamically typed values** (à la Python, Scheme, Javascript):

- nil, or **boxed** { strings, integers, ***Tree***-s, ***Gimples***, ...}, closures, tuples, lists, pairs, objects, homogeneous hash-tables . . .
- garbage collected by MELT using copying generational techniques (old generation is **GTY**-ed Ggc heap)
- quick allocation, favoring very temporary values
- **first class** citizens (every value has its discriminant - for objects their Melt class)

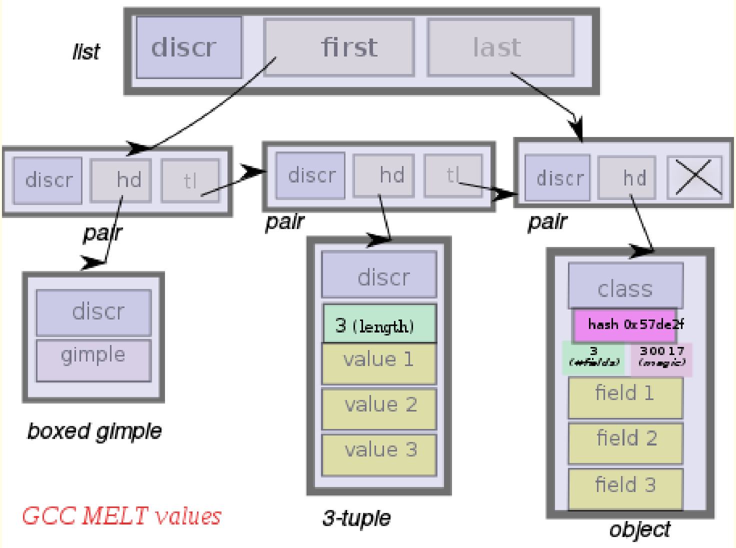
But Gcc **stuff** can be handled by MELT: ***raw*** **Gcc** **`tree`-s, `gimple`-s, `long`-s,** **`const char*`** strings, etc . . .

Local data is garbage-collected[22] (values by MELT GC, stuff only by Ggc)

Type annotations like `:long`, `:cstring` , `:edge` or `:gimple` . . . or `:value` may be needed in MELT code (but also `:auto` à la C++11)

[22] Forwarding or marking routines for locals are generated!

MELT values


list
discr
first
last
discr
hd
tl
pair
discr
hd
tl
pair
discr
hd
pair
discr
gimple
boxed gimple
discr
3 (length)
value 1
value 2
value 3
3-tuple
class
hash 0x57de2f
3 (#fields)
30017 (magic)
field 1
field 2
field 3
object
GCC MELT values

Some MELT language features

- **expression**-based language
- **local variable bindings** with `let` or `letrec`
- named `defun` and **anonymous** with `lambda` **functions** closures
- Smalltalk-like object system `defclass`, `defselector`, `instance` w. dynamic method dictionnary (inside classes or discrimants)
- **iterative constructs** `forever`, `exit`, `again`, … (but no tail-recursion)
- **pattern matching** with `match` (patterns with `?`, so `?_` is wildcard catch-all pattern)
- **dynamic evaluation** w. `eval`, **quasi-quotation** `backquote` ≡ `` ` `` & `comma` ≡ `,`
- **macros** with `defmacro` or local `:macro` binding in `let`
- **conditionals** with `if`, `cond`, `when`, `or`, `and`, `gccif` (testing version of Gcc), …
- multiple data results in function `return` and `multicall`
- many ways to mix *C++* code with Melt code: `code_chunk`, `expr_chunk` and defining C++ generations `defprimitive`, `defcmatcher`, `defciterator`
- environment introspection `parent_module_environment` and `current_module_environment_reference`

A pass in MELT - verifying `melt-runtime.cc`

The MELT runtime contains code like (for boxing integers)

```
melt_ptr_t meltgc_new_int (meltobject_ptr_t discr_p, long num) {
  MELT_ENTERFRAME (2, NULL); // defines and initialize meltfram__
#define newintv meltfram__.mcfr_varptr[0]
#define discrv  meltfram__.mcfr_varptr[1]
#define object_discrv ((meltobject_ptr_t)(discrv))
#define int_newintv ((struct meltint_st*)(newintv))
  discrv = (melt_ptr_t) discr_p;
  if (!discrv)
    discrv = (melt_ptr_t) MELT_PREDEF (DISCR_INTEGER);
  if (melt_magic_discr ((melt_ptr_t) (discrv)) != MELTOBMAG_OBJECT)
    goto end;
  if (object_discrv->meltobj_magic != MELTOBMAG_INT)
    goto end;
  newintv = (melt_ptr_t) meltgc_allocate (sizeof (struct meltint_st), 0);
  int_newintv->discr = object_discrv;
  int_newintv->val = num;
end:
  MELT_EXITFRAME ();
  return (melt_ptr_t) newintv; }
#undef newintv
#undef discrv
#undef int_newintv
#undef object_discrv
```

etc ...

A pass in **MELT** - checks to be done

On functions whose name contains `meltgc_` or starts with `meltrout_` :

- notice all assignments and uses of some local pointer slot `meltfram__.mcfr_varptr[`*i*`]`
- warn about unused such slots
- warn about bad index slots (negative or too big)

the gate function in **MELT**

```
(defun meltframe_gate (pass)
  (with_cfun_decl ()
   (:tree cfundecl)
   (match cfundecl
      ( ?(tree_function_decl_named
                   ?(cstring_containing "meltgc_") ?_)
             (return :true)
         )
      ( ?(tree_function_decl_named
                   ?(cstring_prefixed "meltrout_") ?_)
             (return :true)
         )
      ( ?_ (return ())))))
```

Using the *C-matcher* defined as

```
;; cmatcher for a cstring starting with a given prefix
(defcmatcher cstring_prefixed
  (:cstring str cstr)    ()
  strprefixed
  #{/* cstring_prefixed $STRPREFIXED test*/ ($STR && $CSTR
          && !strncmp($STR, $CSTR, strlen ($CSTR))) }#)
```

the execute function in **MELT** part 1 : finding `meltfram__`

```
(defun meltframe_exec (pass)
  (let ( (:long declcnt 0)
         (:long bbcnt 0)
         (:long gimplecnt 0)
         (:tree tmeltframdecl (null_tree))
         (:tree tmeltframtype (null_tree))
         (:tree tmeltframvarptr (null_tree))
         (:tree tfundecl (cfun_decl))
         (:long nbvarptr 0)
         )
    (each_local_decl_cfun ()
     (:tree tlocdecl :long ix)
     (match tlocdecl
            (?(tree_var_decl_named
               ?(and ?tvtyp
                     ?(tree_record_type_with_fields ?tmeltframrecnam
                                                    ?tmeltframfields))
               ?(cstring_same "meltfram__") ?_)
              (setq tmeltframdecl tlocdecl)
              (setq tmeltframtype tvtyp)
```

the execute function in **MELT** part 2 : find its `mcfr_varptr`

```
          (foreach_field_in_record_type
           (tmeltframfields)
           (:tree tcurfield)
           (match tcurfield
                  ( ?(tree_field_decl
                       ?(tree_identifier ?(cstring_same "mcfr_varptr"))
                       ?(tree_array_type ?telemtype
                                         ?(tree_integer_type_bounded ?t
                                              ?(tree_integer_cst 0)
                                              ?(tree_integer_cst ?lmax)
                                              ?tsize)))
                     (setq tmeltframvarptr tcurfield)
                     ((setq nbvarptr lmax)
                     )
                  (?_  (void)))))))
    ((setq declcnt (+i declcnt 1)))
```

execute function in MELT 3: find mcfr_varptr[*i*] = ...

```
(each_bb_cfun  ()
 (:basic_block bb :tree fundecl)
 (setq bbcnt (+i bbcnt 1))
 (eachgimple_in_basicblock
  (bb)  (:gimple g)
  (setq gimplecnt (+i gimplecnt 1))
  (match g
       ?(gimple_assign_single
         ?(tree_array_ref
           ?(tree_component_ref
             tmeltframdecl
             tmeltframvarptr)
           ?(tree_integer_cst ?ixkdst))
         ?rhs)
        (cond
         (  (<i ixkdst 0)
           (warning_at_gimple g "negative meltvarptr destination pointer index"
           )
         (  (>i ixkdst nbvarptr)
           (warning_at_gimple g "too big meltvarptr destination pointer index")
           )
         (:else
          (meltframe_update_tuple_ptr tupdefptr ixkdst g)
          ))
        )
```

mode in **MELT** installing the pass - mode processing

```
(defun meltframe_docmd (cmd moduldata)
  (let ( (meltframedata
           (instance class_melt_frame_data
                     :meltfram_funcount (box 0)
                     ))
         (meltframepass
           (instance class_gcc_gimple_pass
                     :named_name '"melt_frame_pass"
                     :gccpass_gate meltframe_gate
                     :gccpass_exec meltframe_exec
                     :gccpass_data meltframedata
                     :gccpass_properties_required ()
                     ))
         )
    (install_melt_pass_in_gcc meltframepass :after '"ssa" 0)
    (at_exit_first
     (lambda (x)
       (let ( (:long nbmeltrout
                     (get_int (get_field :meltfram_funcount meltframedata))) )
         (code_chunk informnbmelt
          #{/* $INFORMNBMELT */
          inform(UNKNOWN_LOCATION, "melt_frame_pass found %ld MELT routines",
                 $NBMELTROUT);
          }#))))
    (return :true)))
```

defining and installing the mode

```
(definstance meltframe_mode
  class_melt_mode
  :named_name '"meltframe"
  :meltmode_help
      '"install a pass checking MELT frame accesses"
  :meltmode_fun meltframe_docmd
)
(install_melt_mode meltframe_mode)
```

The `tree_integer_cst` *C-matcher* from `melt/xtramelt-ana-tree.melt`

```
(defcmatcher tree_integer_cst
    (:tree tr) (:long n) treeintk
  ;; test expander
  #{ /*tree_integer_cst $TREEINTK ?*/
#if  MELT_GCC_VERSION >= 4009 /* GCC 4.9 or later */
  (($TR) && TREE_CODE($TR) == INTEGER_CST
         && tree_fits_shwi_p ($TR))
#else /* GCC 4.8*/
  (($TR) && TREE_CODE($TR) == INTEGER_CST
         && host_integerp($TR, 0))
#endif /* GCC 4.8 */
  }#
  ;; fill expander
  #{ /*tree_integer_cst $TREEINTK !*/
#if  MELT_GCC_VERSION >= 4009 /* GCC 4.9 or later */
    $N = tree_to_shwi(($TR));
#else  /* GCC 4.8 */
    $N  = tree_low_cst(($TR), 0);
#endif /* GCC 4.9 */
  }#)
```

General advices when customizing Gcc

- non-trivial effort (weeks, not hours, of work)
- make some toy (or test) cases (of compiled source code)
- look carefully into their various *Gimple* representations before designing your plugin
- two kinds of customization:
  1. inspection of existing I.R. (e.g. check coding rules, metrics, source navigation)
     need to process the relevant cases only
  2. changing the existing I.R. (e.g. library specific optimization)
     probably harder (be sure to handle all the cases!)
- study the source code of Gcc
- interact with the Gcc community
- **consider using MELT**

  (learning Melt is much easier than studying Gcc)

Where to insert your extra optimization pass?

A difficult issue for everyone (at least even for me, Basile).

- at some **early passes**, the I.R. is **incomplete**
- at some **late passes**, some of the I.R. is **“rotten”**
- passes depend upon optimization levels and compiled source code
- High-Gimple vs Gimple/Ssa?
- interesting points: after `cfg`, `ssa`, `phiopt`
- trial and error approach!
- pass management (and set of available passes) vary slightly with Gcc versions

See `https://gcc-python-plugin.readthedocs.org/en/latest/tables-of-passes.html`

Gcc plugins - version sensitivity

Gcc plugins are sensible to Gcc versions

- plugins should be recompiled for a patch level change (e.g. `4.8.1` → `4.8.2`); generally a simple recompilation is enough (but some API might slightly change[23] ...)
- plugins source code should be significantly changed for a Gcc version change (e.g. `4.8.2` → `4.9.0`); Some of the internal representations is changing[24]
- MELT demonstrates that with some work a plugin[25] can be made to work for two consecutive Gcc versions.

  (mostly because of "social" or "workforce" reasons: Gcc is so big and complex that it is *slowly* changing) So MELT abstracts a tiny bit such changes (but is not a silver bullet), making your life slightly easier

---

[23] e.g. `check_default_argument` got a third argument in `gcc/cp/cp-tree.h`

[24] `GIMPLE_OMP_TASKGROUP` is new in 4.9!

[25] In its source form; the shared objects are specific to each version and configuration of Gcc!

Useful potential *Gcc* customizations

- in Gtk, typing of variadic functions e.g. `g_object_set`
- in Posix applications, coding rules check (e.g. every `fork` is checked for failure in its calling function)
- for Sql client libraries, *simple* checks to avoid some Sql injections
- navigation aid to large free software (e.g. Linux kernel)
- simple coding rules related to locking in `pthread`
- supporting a precise garbage collection in C or C++
- D.Malcom's Python plugin checking Python $\leftrightarrow$ C glue code
- (but *TreeHydra* abandonned in Firefox)
- *some* simple symbolic simplification in numerical libraries? $0\vec{v} = \vec{0}$
- etc ...

Free software communities should **know better what *Gcc* customizations are beneficial** to them

Why free software needs *Gcc* customization (with Melt) ?

Free software is increasingly important, and generally compiled by Gcc (e.g. in Linux distributions)
Large free software communities should consider developping their own tools as *Gcc* extensions (e.g. with Melt)

The Gcc compiler will become more "plugin friendly" when real *Gcc* plugins (or extension Melt) will be developped - outside of, but in collaboration with, the Gcc community

Consider customizing[26] Gcc for your free software
⇒ Then MELT is an appropriate tool for that customization!

[26] or asking someone to customize

Thanks

Questions are welcome