

GCC plugins and MELT extensions

Basile STARYNKEVITCH

basile@starynkevitch.net (or basile.starynkevitch@cea.fr)



energie atomique • energies alternatives

list



June 16th 2011 – *ARCHI'11 summer school (Mont Louis, France)*



These slides are under a Creative Commons Attribution-ShareAlike 3.0 Unported License

creativecommons.org/licenses/by-sa/3.0 and downloadable from gcc-melt.org

Table of Contents

1 Introduction

- about you and me
- about GCC and MELT
- building GCC

2 GCC Internals

- complexity of GCC
- overview inside GCC (cc1)
- memory management inside GCC
- optimization passes
- plugins

3 MELT

- why MELT?
- handling GCC internal data with MELT
- matching GCC data with MELT
- future work on MELT

Contents

- 1 Introduction
 - about you and me
 - about GCC and MELT
 - building GCC
- 2 GCC Internals
 - complexity of GCC
 - overview inside GCC (cc1)
 - memory management inside GCC
 - optimization passes
 - plugins
- 3 MELT
 - why MELT?
 - handling GCC internal data with MELT
 - matching GCC data with MELT
 - future work on MELT

opinions are mine only

Opinions expressed here are only mine!

- not of my employer (CEA, LIST)
- not of the Gcc community
- not of funding agencies (e.g. DGCIS)¹

I don't understand or know all of Gcc ;
there are many parts of Gcc I know nothing about.

Beware that **I have some strong technical opinions** which are not the view of the majority of contributors to Gcc.

I am not a lawyer ⇒ don't trust me on licensing issues

¹Work on Melt have been possible thru the GlobalGCC ITEA and OpenGPU FUI collaborative research projects, with funding from DGCIS

expected audience

The audience is expected to:

- **have**, use, and administrate **a GNU/Linux system** (e.g. a **laptop with $\geq 3Gb$ RAM and $\geq 12Gb$ free disk space**) thru the command-line
- be able to install new packages (thru a working Internet connection)
- be **fluent in C** and **curious about compilers and languages** (knowledge of Scheme or Lisp is helpful but not mandatory)
- have already built some significant free software from source code
- be familiar with use of `gcc`, `make`, `sh` (shell scripts), `awk`, editors (e.g. `emacs`), **Posix** system calls...
- be sympathetic to free software ideals, communities and habits (GPL license, peer review, version control, mailing lists, wiki, ...)
- not be allergic to parenthesis 😊 [Lisp = *Lots of Insipid Stupid Parenthesis*]

Questions to the audience

- 1 Qui parle français? Qui ne lit pas l'anglais 😞 ?
- 2 Who understands [my bad] English ? Who don't understand French?
- 3 Who did build a Gnu software from its source code last year? Who built Gcc once?
- 4 What is your usual Gcc version? your latest Gcc?
- 5 Who did build a big (> 1MLOC) free software? When? Which one?
- 6 Who contributed to a Gnu (or GPL) software? Which one? To Gcc 😊?
- 7 Who knows and codes in
 - C language is **mandatory** since **you use gcc!**
 - C++ or Objective C or D or Go or OpenCL or Cuda (C "improvements")?
 - **Scheme**, Common Lisp, Clojure, or Emacs Lisp (lisp languages)?
 - **Ocaml** or Haskell or Scala (pattern matching, functional)?
 - Java or C# [.Net] (major VMs, GC-ed)?
 - Python, Ruby, Lua, PHP, Perl, Awk, Scilab, R (dynamic scripting languages)?
 - Fortran, Ada, Pascal, Modula3 (legacy, perhaps targeted by Gcc)?
- 8 Who wrote (or contributed to) a compiler? An interpreter? A C or JIT code generator?

Contributing to GCC

Bug reports are always welcome. <http://gcc.gnu.org/bugzilla/>
give carefully all needed information

For **code** (or documentation) **contributions**:

- read <http://gcc.gnu.org/contribute.html>
- legalese: **copyright assignment** of your work to **FSF**
 - need legal signature by important people: boss, dean, “President d’Université” (takes a lot of burocratic time)
 - never submit code which you did not write yourself
- coding rules and standards
<http://gcc.gnu.org/codingconventions.html>
- **peer-review of submitted code patches** on gcc-patches@gcc.gnu.org
every contribution to **Gcc** has been reviewed

GCC at a glance

You are expected to know about it, and to have used it!

<http://gcc.gnu.org/>
GNU COMPILER COLLECTION
(long time ago, started as **Gnu C Compiler**)

- **Gcc** is a [set of] **compiler[s]** for several languages and architectures with the necessary language and support libraries (e.g. `libstdc++`, etc.)
- **Gcc** is **free** -as in speech- software (mostly under GPLv3+ license)
- **Gcc** is **central to the GNU** movement, so ...
- **Gcc** is a **GNU** software
- **Gcc** is **used** to compile a lot of free (e.g. GNU) software, notably most of GNU/Linux distributions, Linux kernel, ...
- the <http://www.gnu.org/licenses/gcc-exception.html> permit you to use **Gcc** to compile **proprietary** software with **conditions**.
So, it probably forbids to distribute only binaries built with a proprietary enhancement of **Gcc**.

Everyone is using code compiled with Gcc

(e.g. in your smartphone, car, plane, ADSL box, laptop, TV set, Web servers ...).

A short history of GCC

- started in 1985-87 by RMS (Richard M. Stallman, father of GNU and FSF)
- may 1987: `gcc-1.0` released (a “statement at a time” compiler for C)
- december 1987: `gcc-1.15.3` with `g++`
- 1990s: the Cygnus company (M. Tiemann)
- february 1992: `gcc-2.0`
- 1997: The **EGCS** crisis (an Experimental Gnu Compiler System), a fork
- `ecgs 1.1.2` released in march 1999
- april 1999: **EGCS** reunited with FSF, becomes `gcc-2.95`
- march 2001: `gcc-2.95.3`
- june 2001: `gcc-3.0`
- november 2004: `gcc-3.4.3`
- april 2005: `gcc-4.0`
- april 2010: `gcc-4.5` enable plugins and LTO
- march 2011: `gcc-4.6` released

See also <http://www.h-online.com/open/features/>

[GCC-We-make-free-software-affordable-1066831.html](http://www.h-online.com/open/features/GCC-We-make-free-software-affordable-1066831.html)

GCC community

The community:

- more than 400 contributors (file `MAINTAINERS`), mostly nearly full-time corporate professionals (AMD, AdaCore, CodeSourcery, Google, IBM, Intel, Oracle, SuSE, and many others)
- copyright assigned to FSF (sine qua non for write `svn` access)
- peer-reviewed contributions, but no single leader
- several levels:
 - 1 Global Reviewers (able to Ok anything)
 - 2 Specialized Reviewers (port, language, or features maintainers)
 - 3 Write After Approval maintainers
(formally cannot approve patches, but may comment about them)

These levels are implemented socially, not technically

(e.g. every maintainer could `svn commit` any file but shouldn't.).

Public exchanges thru archived mailing-lists `gcc@gcc.gnu.org` & `gcc-patches@gcc.gnu.org`, IRC, meetings, GCC Summit-s.

GCC Steering Committee

<http://gcc.gnu.org/steering.html> and
<http://gcc.gnu.org/gccmission.html>

The SC is made of major **Gcc** experts (mostly global reviewers, not representing their employers). It takes major “political” decisions

- relation with FSF
- license update (e.g. GPLv2 → GPLv3) and exceptions
- approve (and advocate) major evolutions: plugins feature², new languages, new targets
- nominate reviewers

NB. Major technical improvements (e.g. LTO or Gimple) is not the role of the SC.

²The introduction of plugins required improvement of the GCC runtime exception licensing.

GCC major features

- **large free software** project, essential to GNU ideals and goals
 - 1 old mature software: **started in 1984**, lots of legacy
 - 2 essential to free software: **corner stone of GNU** and Linux systems
 - 3 big software, nearly **5 million lines** of source code
 - 4 **large community** (≈ 400) of full-time developers
 - 5 **no single leader** or benevolent dictator
 - ⇒ the `Gcc` code base or architectural design is sometimes messy!
- compiles **many source languages**: C, C++, Ada, Fortran, Objective C, Java, Go ... (supports several standards, provides significant extensions)
- makes non-trivial **optimizations** to generate **efficient binaries**
- **targets many processors** & variants : x86, ARM, Sparc, PowerPC, MIPS, ...
- can be used as a **cross-compiler**:
compile on your Linux PC for your ARM smartphone
- can **run on many systems** (Linux, FreeBSD, Windows, Hurd ...) and generate various binaries

Extending GCC

Recent `Gcc` can be extended by **plugins**.

This enables **extra-ordinary features**:

- **additional optimizations** (e.g. research or prototyping on optimizations)
- **domain-, project-, corporation-, software- ... specific extensions**:
 - 1 specific warnings , e.g. for untested calls to `fopen` or `fork`
 - 2 specific type checks, e.g. type arguments of variadic `g_object_set` in `Gtk`.
 - 3 coding rules validation, e.g. ensure that `pthread_mutex_lock` is matched with `pthread_mutex_unlock`
 - 4 specific optimizations, e.g. `fprintf(stdout, ...) ⇒ printf(...)`
- take advantage of `Gcc` power for your **“source-code” processing** (metrics, navigations, refactoring...)

Some people dream of enhancing GCC thru plugins to get a free competitor to `Coverity™`

<http://www.coverity.com/> source code analyzer

See also a C-only free static analyzer `Frama-C` <http://frama-c.com/> coded in Ocaml

alternatives to GCC

You can use other languages:

- 1 high-level functional statically-typed languages like e.g. **Ocaml**
`http://caml.inria.fr` or Haskell
- 2 more academic languages (SML, Mercury, Prolog, Scheme, ...) ³, or niche languages (Erlang ...)
- 3 dynamic scripting languages: Python, PHP, Perl, Lua ...
- 4 dynamic compiled languages: Smalltalk (Squeak), CommonLisp (SBCL)
(some implementations are even generating machine code on the fly!)
- 5 Java⁴ and JVM based languages (Scala, Clojure, ...)
- 6 etc ...
- 7 **assembly code is obsolete: compilers do better than humans**⁵
(you could use **Gcc** powerful `asm` statement)

³Some compilers -e.g. Chicken Scheme- are generating C code for **Gcc**

⁴**Gcc** accepts Java as `gcj` but that is rarely used.

⁵For a hundred lines of code.

Generating code yourself

You can generate code, either e.g. C or C++⁶, or machine code with JIT-ing libraries like GNU `lightning` or `libjit` or **LLVM**, a free BSD-licensed⁷ library <http://llvm.org> for [machine] code generation (e.g. Just In Time)

Melt is implemented by generating C code

meta-knowledge⁸ **meta-programming** and *multi-staged programming* are interesting subjects.

Advice: never generate by naïve text expansion (e.g. `printf...`). Always represent your generated code in some abstract syntax tree.

⁶You could even generate C code, compile it (by forking a `gcc` or a `make`), then `dlopen` it, all from the same process. On Linux you can call `dlopen` many times.

⁷IMHO, the BSD license of LLVM do not encourage enough a free community. LLVM is rumored to have many proprietary enhancements.

⁸J.Pitrat: *Méta-connaissances, futur de l'intelligence artificielle* (Hermès 1990)
Artificial beings - the conscience of a conscious machine (Wiley 2009)

Competitors to GCC

You can use other compilers, even for C or C++:

- proprietary compilers, e.g. Intel's `icc`
- **Compcert** <http://compcert.inria.fr/> - a C compiler formally proven in Coq (restrictive license, usable & readable by academia)
- **Clang**, a C and C++ front-end above **LLVM**.
- “toy” one-person compilers⁹, usually only on x86:
 - **tinycc** by Fabrice Bellard <http://tinycc.org/> and <http://savannah.nongnu.org/projects/tinycc>; compiles very quickly to slow machine code
 - **nwcc** by Nils Weller <http://nwcc.sourceforge.net/> - possibly stalled

NB: There is almost no market for costly proprietary compilers, competitors to GCC

Competition is IMHO good within free software

⁹Often quite buggy in practice

Why GCC matters?

Gcc matters to you and to me because:

- you are interested in computer architecture or performance or diagnostics¹⁰, so compilers matter to you
- you want to experiment some new compilation ideas
- you want to profit of Gcc to do some “extra-compilatory” activities
- so you need to understand (partly) Gcc internals
- it is fun to understand such a big free software!
- you want to contribute to Gcc itself (or to Melt)

¹⁰Execution speed, code size, fancy compiler warnings!

Link time or whole program optimization

Recent `Gcc` has **link-time optimizations**:
use `gcc -O2 -flto` for compile **and** for linking¹¹.

Then optimization between compilation units (e.g. inlining) can happen.

LTO can be costly.

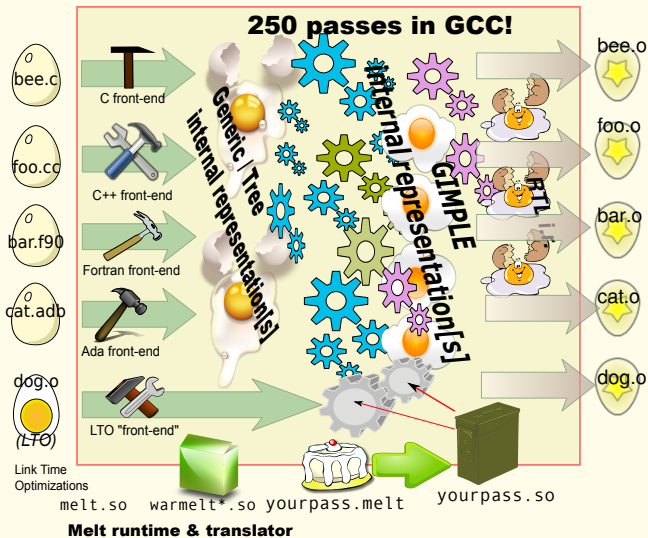
LTO is implemented by encoding GCC internal representations (Gimple, ...) in object files.

LTO can be extended for large whole program optimization (WHOPR)

LTO can be used by extensions to provide **program-wide** features

¹¹e.g. with `CC=gcc -O2 -flto` in your *Makefile*

Gcc and Melt big picture



GCC MELT

What is MELT?

Coding **Gcc** extensions (or prototyping new **Gcc** passes) is quite difficult in C:

- compiler technology is mostly symbolic processing (while C can be efficient, it is not easy to process complex data with it).
- specific **Gcc** extensions need to be developed quickly (so development productivity matters more than raw performance)
- an important part of the work is to detect or filter patterns in **Gcc** internal representations

Melt is a lisp **Domain Specific Language** for developing **Gcc** extensions

- **Melt** is designed to **fit very well into Gcc** internals; it is **translated to C** (in the style required by **Gcc**).
- **Melt** has powerful features: **pattern-matching**, applicative & object programming, ...
- **Melt** is itself a [meta-] plugin¹² for **Gcc**

¹²There is also an **experimental Gcc branch** for **Melt**!

Building GCC from its source code

You should build GCC from its source code¹³

because you need to study its source code to understand its internals

- building Gcc requires¹⁴ efforts, methods, tenacity, perseverance, time.
Don't give up (you may need to try more than once), and do it now!
- getting Gcc sources - see <http://gcc.gnu.org>. You can get
 - 1 a **released Gcc tar-ball** - see <http://gcc.gnu.org/mirrors.html> (get at least `gcc-4.6.0.tar.gz` of 91Mb!)
 - 2 the current Gcc **trunk** (future 4.7 release) by SubVersion:
`svn co svn://gcc.gnu.org/svn/gcc/trunk gcc-trunk`
 - 3 An **experimental branch**, e.g. `svn co \`
`svn://gcc.gnu.org/svn/gcc/branches/melt-branch gcc-melt`

If you use `svn`, **run** also (and often) `./contrib/gcc_update` which fires an `svn update` and generate REVISION source file

¹³Even if you have the latest Gcc installed by your distribution, you should build it.

¹⁴Gcc is no more difficult to build than other big free software (Linux kernel, Firefox, Qt, ...)

Gcc dependencies

The **Gcc** compiler needs several external libraries and tools. See

<http://gcc.gnu.org/install/>

easy use your distribution package manager, e.g. on Debian/Sid or Ubuntu

aptitude build-dep gcc-4.6

(beware that prerequisites for **Gcc** 4.5 are not the same as for **Gcc** 4.6)

hard build the dependencies yourself

- 1 probably needed on not-so new distributions
- 2 **versions** of the prerequisites **matter!** (e.g. PPL 0.11, not PPL 0.10.2!)
- 3 **beware of conflicts** with older versions already on your system
(you might need some `$PATH` tricks)
- 4 some prerequisites depends upon others, so building **order matters!**
- 5 some prerequisites need **special configuration**, e.g.
`--enable-interfaces=c` for PPL

Be very careful on the GCC prerequisites

downloadable from <ftp://gcc.gnu.org/pub/gcc/infrastructure/>

Important GCC prerequisites

As usual, **versions matter** (non-exclusive list!)

- 1 GNU **make** **awk** **svn** **emacs** **autoconf** **automake** **gzip** **tar** **bash**
binutils¹⁵ **texinfo**
- 2 a C compiler (e.g. an older **gcc**, or **clang**)
- 3 for the Ada front-end, an older **Gnat**
- 4 for the Go front-end, some C++, e.g. an older **g++**
- 5 GNU Multiple Precision library **GMP**
- 6 GNU Multiple Precision Floating point Rounding library **MPFR**
- 7 Multi Precision Complex library **MPC**
- 8 Parma Polyhedra Library **PPL**
- 9 Chunky LOOp Generatator **CLOOG**

Without CLOOG, or PPL you won't get the **Graphite** loop optimizations

Melt don't require more prerequisites than **Gcc** (but needs `unifdef` and `indent`).

It needs good **dlopen** and **dlsym** from your **-ldl** system library

¹⁵The Gold variant of binutils is preferred

The build tree

Big file tree (> 1Gb) containing generated files:

- configure generated Makefile
- various code generators
- various generated sources *.c *.h
- object files and executable binaries
- other files, etc ...

When a build fails¹⁶, **restart it in an empty build tree!**

Gcc build tree \neq **Gcc source tree**

For **Gcc**, the build tree should not be the source tree.

I recommend that the build tree and the source tree be inside the same containing directory, e.g.

GCCSOURCE=/usr/src/Lang/gcc-trunk source tree of **Gcc** trunk

GCCBUILD=/usr/src/Lang/_Build-gcc-trunk build tree of **Gcc** trunk

¹⁶e.g. after a contrib/gcc_update

Steps for building Gcc

Several building steps are required (like for most GNU software)

- 1 configuration
- 2 (multi-staged for a straight Gcc compiler) compilation
Gcc compiles itself (⇒ takes some time, e.g. more than an hour)
- 3 optional testing (may take a very long time)
- 4 installation

The build may fail after update with `./contrib/gcc_update` in source tree; in such case, run `make clean` in build tree, or even destroy and rebuild it.

Parallel build with `make -j` can fail¹⁷, but might partly work.

NB: Gcc is designed to be buildable together with many other GNU software (e.g. binutils and gdb). I never did that!

¹⁷In particular, **my Melt branch cannot be built with a parallel make!**

Caveat on configuring and building GCC

Gcc is quite a **complex** software, **with a baroque configuration** and building procedure.

If you never built any GNU software, please exercise¹⁸ on some simpler GNU software, e.g. one of

- a shell like BASH <http://www.gnu.org/software/bash/>
- a library like GMP <http://gmplib.org/>
- an editor like EMACS <http://www.gnu.org/software/emacs/>
- a graphical toolkit like GTK <http://www.gtk.org/>
- etc ...

You should be familiar with the famous `configure`; `make`; `make install` which is not always as simple as that.

You should know what `autoconf` & `automake` are for!

¹⁸The point is to be fluent with GNU configuration and building procedures, so please compile one if you never did, even if that software is already packaged in your GNU/Linux distribution.

Configuring GCC

As in every GNU software, run the source tree `configure` script¹⁹ inside the build tree. But for `Gcc` configuration:

- the option `--help to configure` don't work well
(because the top `Gcc configure` fires `sub-configure` in sub-directories)
- **default options are usually not enough** nor appropriate
- **your options are not your neighbor's options!**
(because you don't have the same prerequisites at the same places and versions)
- **Gcc configuration is tricky** (a black art)
- you often should destroy your build tree after mis-configuration

Know your own system and needs, and read the documentation, e.g <http://gcc.gnu.org/install/configure.html> etc. Remember and note your configuration options: `grep configure config.status /usr/bin/gcc -v` also gives the configuration of your system's `Gcc`

¹⁹That script is generated by `autoconf` from `configure.ac`

Some important GCC configuration options

The `--enable-...` options can be `--disable-...` and the `--with-...` options give prerequisites' directories

- 1 `--prefix /some/dir` (default is `/usr/local`, places for installation)
- 2 `--program-suffix -suffix` to install `gcc-suffix` program (important to use to have several `Gcc` eg trunk and 4.6 and `Melt` branch)
- 3 `--enable-plugin` you always want plugins
- 4 `--enable-languages=c, c++, go` specify the front-end languages
- 5 `--disable-multilib` to disable 32 bits support on 64 bits machine
- 6 `--enable-maintainer-mode` useful (force regeneration by `autoconf`)
- 7 `--with-ppl=/usr/local` (to use yours, not system, PPL)
- 8 `--disable-bootstrap` to avoid compiling `Gcc` with itself (so build time is smaller; but needed when contributing to `Gcc`)
- 9 `--enable-checking=all` add runtime checks but slow down your `Gcc`
- 10 `--target armv5-android-eabi` to build a `Gcc` cross-compiler (but I never did that)

configuration example: Debian/Sid

My system gcc 4.6 as given by `gcc-4.6 -v` on Debian/Sid:
Configured with:

```
../src/configure -v --with-pkgversion='Debian 4.6.0-10' \  
--with-bugurl=file:///usr/share/doc/gcc-4.6/README.Bugs \  
--enable-languages=c,c++,fortran,objc,obj-c++,go \  
--prefix=/usr --program-suffix=-4.6 --enable-shared \  
--enable-multiarch --with-multiarch-defaults=x86_64-linux-gnu \  
--enable-linker-build-id --with-system-zlib \  
--libexecdir=/usr/lib --without-included-gettext \  
--enable-threads=posix --enable-plugin \  
--with-gxx-include-dir=/usr/include/c++/4.6 \  
--libdir=/usr/lib --enable-nls --enable-clocale=gnu \  
--enable-libstdcxx-debug --enable-libstdcxx-time=yes \  
--enable-objc-gc --with-arch-32=i586 --with-tune=generic \  
--enable-checking=release --build=x86_64-linux-gnu \  
--host=x86_64-linux-gnu --target=x86_64-linux-gnu
```

configuration of my Melt branch

Of course, source and build trees are different (they should always be \neq)

```
cd /usr/src/Lang/_Build-gcc-melt
```

```
/usr/src/Lang/basile-melt-gcc/configure20 \  
  --program-suffix=melt  --libdir=/usr/local/lib/gcc-melt \  
  --libexecdir=/usr/local/libexec/gcc-melt \  
  --with-gxx-include-dir=/usr/local/lib/gcc-melt/include/c++/ \  
  --with-ppl=/usr/local --with-mpc-include=/usr/local/include \  
  --with-mpc-lib=/usr/local/lib \  
  --enable-maintainer-mode --enable-checks=tree,gc \  
  --disable-bootstrap --disable-multilib \  
  --enable-version-specific-runtime-libs --enable-plugin \  
  --enable-languages=c,c++,go,lto \  
  CFLAGS='-O -g'
```

You should find **your** configuration **flags** appropriate **for your system** !

²⁰Use an absolute path!

please **configure and start building your Gcc** now!

You should know well your system.

You might need to build some prerequisites.

(I could help you while I am here with you)

Building and installing GCC

To build your `Gcc` after configuration, run **make in your build tree**

- if something goes wrong, try **make** again after corrections.
- if you re-configure, better start from a scratch build tree (sometimes you can just run a `make`)
- parallel `make -j` usually work²¹ if you are lucky
- using `ccache` might help too

To **install after the build**, run **make install DESTDIR=/tmp/gccinst**, and copy as root the `/tmp/gccinst`, e.g:

```
sudo cp -va /tmp/gccinst/usr/local/. /usr/local/.
```

NB: if hacking or debugging `Gcc`, you don't need to install it, but use its `-B` option or run directly `./cc1`

²¹ But the `Melt` branch cannot be built with a `make -j`, it needs a sequential `make`.

Using your newly built Gcc

Take care of your `$PATH` and use the `gcc-suffix` (suffix that you passed as `.../configure --program-suffix = -suffix`)

This `gcc-suffix` is just **a driving program**. (I will just write `gcc` for it). It fires:

- a `cc1` process²² (preprocessing, parsing, optimization passes, emission of assembly code) to make a `.s` assembly temporary file
- the assembler `as` to translate that `.s` to an object file `.o`
- a `ld` or `collect2`²³ link-editor taking some object files
- perhaps a `lto1` for link time optimizations

The behavior of `gcc` is governed by its `specs` file.

²²That would be `cc1plus` for C++, etc.

²³The `collect2` takes care of constructors for C++ to be called before `main`

some useful options to your gcc

- `-v` to `Gcc` shows the fired sub-processes.
- `-time` shows their time.
- you know `-Wall`, `-O2` or `-g` or `-I` & `-Dprepro` or `-lsomelib` etc.
- `--help` give lot of help.
- Temporary files are deleted unless `-save-temps` is given.
- When `-pipe` is given, `cc1` and `as` run in parallel using a pipe.
- **many many more** obscure **options**:
 - `-std=gnu99` for language standards
 - `-Werror` to make all warnings become errors
 - `-Wextra`, `-Wuninitialized` etc. for more warnings
 - many additional optimizations²⁴ like `-floop-block`
 - target machine `-mtune=native` or `-mtune=core7`
 - plugins `-fplugin=melt` with arguments to them.

The `cc1` (or `cc1plus ... lto1`) **is doing most of the work.**

²⁴A lot of optimizations are not performed even with `-O3`, so you should explicitly require them!

driven by gcc

gcc -v -O hello.c -o hello

1 C-compile

```
/usr/lib/gcc/x86_64-linux-gnu/4.6.1/cc1 -quiet -v hello.c -quiet \
  -dumpbase hello.c -mtune=generic -march=x86-64 -auxbase hello \
  -O -version -o /tmp/ccTBI9E6.s
```

2 assemble **as** -64 -o /tmp/ccOMVPbN.o /tmp/ccTBI9E6.s

3 link

```
/usr/lib/gcc/x86_64-linux-gnu/4.6.1/collect2 --build-id
  --no-add-needed--eh-frame-hdr -m elf_x86_64 --hash-style=both
  -dynamic-linker /lib64/ld-linux-x86-64.so.2 -o hello
  /usr/lib/gcc/x86_64-linux-gnu/4.6.1/../../../../lib/crt1.o
  /usr/lib/gcc/x86_64-linux-gnu/4.6.1/../../../../lib/crti.o
  /usr/lib/gcc/x86_64-linux-gnu/4.6.1/crtbegin.o
  -L/usr/lib/gcc/x86_64-linux-gnu/4.6.1
  -L/usr/lib/gcc/x86_64-linux-gnu/4.6.1/../../../../lib
  -L/lib/../../lib -L/usr/lib/../../lib
  -L/usr/lib/gcc/x86_64-linux-gnu/4.6.1/../../../../
  -L/usr/lib/x86_64-linux-gnu /tmp/ccOMVPbN.o -lgcc --as-needed
  -lgcc_s --no-as-needed -lc -lgcc
  --as-needed -lgcc_s --no-as-needed
  /usr/lib/gcc/x86_64-linux-gnu/4.6.1/crtend.o
  /usr/lib/gcc/x86_64-linux-gnu/4.6.1/../../../../lib/crtn.o
```

driven by gcc [with C++ and LTO]

```
gcc -v -O -flto sayit.cc hello1.c -o hello1 -lstdc++
```

1 C++ compile

```
.../cclplus -quiet -v -v -D_GNU_SOURCE sayit.cc -quiet -dumpbase sayit.cc -mtune=generic -march=x86-64
```

2 assemble `as` -64 -o /tmp/ccMj0zRK.o /tmp/ccEErWK.s

3 C compile

```
.../ccl -quiet -quiet -v -v hello1.c -quiet -dumpbase hello1.c -mtune=generic -march=x86-64
```

4 assemble `as` -64 -o /tmp/ccocvgSK.o /tmp/ccEErWK.s

5 link

```
.../collect2 -plugin .../liblto_plugin.so -plugin-opt=.../lto-wrapper -plugin-opt=-fr
```

6 re-invoke `gcc` @/tmp/ccZF9We9.args SO

1 link-time optimize

```
.../lto1 -quiet -dumpdir ./ -dumpbase hello1.wpa -mtune=generic -march=x86-64
```

2 link-time optimize

```
.../lto1 -quiet -dumpdir ./ -dumpbase hello1.ltrans0 -mtune=generic -march=x86-64
```

3 assemble `as` -64 -o /tmp/ccBc8K58.ltrans0.ltrans.o /tmp/ccghYRN.a.s

what is happening inside `cc1` ?

make inside a **fresh directory** a small `hello.c` with a simple loop, and using `#include <stdio.h>`

- compile with `gcc -v -Wall -O hello.c -o hello` and run `./hello`
- preprocessing: `gcc -C -E hello.c > hello.i`. Look into `hello.i`
- generated assembly: `gcc -O -fverbose-asm -S hello.c`. See `hello.s`. Try again with `-g` or `-O2 -mtune=native`
- detailed timing report with `-ftime-report`. Try it with `-O1` and with `-O3`.
- internal dump files, to debug or understand **Gcc** itself.
`gcc -O2 -c -fdump-tree-all hello.c` produces hundreds of files. List them **chronologically** with `ls -lt hello.c.*` and look inside some.

Contents

- 1 Introduction
 - about you and me
 - about GCC and MELT
 - building GCC
- 2 GCC Internals
 - complexity of GCC
 - overview inside GCC (cc1)
 - memory management inside GCC
 - optimization passes
 - plugins
- 3 MELT
 - why MELT?
 - handling GCC internal data with MELT
 - matching GCC data with MELT
 - future work on MELT

Code size of GCC

Released `gcc-4.6.0.tar.gz` (on march 25th, 2011) is 92206220 bytes (90Mb).

The `gunzip-ed tar-ball gcc-4.6.0.tar` is 405Mb.

Previous `gcc-4.5.0.tar.gz` (released on april 14th, 2010)²⁵ was 82Mb.

`gcc-4.6.0/` measured with D.Wheeler's SLOCcount:

4,296,480 Physical Source Lines of Code

Measured with `ohcount -s`, in total:

- 57360 source files
- 5477333 source code lines
- 1689316 source comment lines
- 1204008 source blank lines
- 8370657 source total lines

²⁵There have been minor releases up to `gcc-4.5.3` in april 29th, 2011.

Why is GCC so complex?

- it **accepts many source languages** (C, C++, Ada, Fortran, Go, Objective-C, Java, ...), so has many front-ends
- it **targets several dozens of processors** thru many back-ends
 - common processors like x86 (ia-32), x86-64 (AMD64), ARM, PowerPC (32 & 64 bits), Sparc (32 & 64 bits) ...
 - less common processors: ia-64 (Itanium), IBM Z/390 mainframes, PA-RISC, ETRAX CRIS, MC68000 & DragonBall & ColdFire, ...
 - extinct or virtual processors: PDP-11, VAX, MMIX, ...
 - processors supported by external variants: M6809, PIC, Z8000 ...
- it **runs on many operating systems**, perhaps with **cross-compilation**
- it performs **many optimizations** (mostly target **neutral!**)
- because **today's processors are complex**, and **far from C**
- so **Gcc** has an **extensive test-suite**

Why GCC needs to be complex?

See the **Essential Abstractions in GCC** tutorial at CGO2011

<http://www.cse.iitb.ac.in/grc/index.php?page=gcc-tut> by Uday Khedker (India Institute of Technology, Bombay)

Because **Gcc** is not only the **Gnu Compiler Collection**, but is now a **compilation framework** so becomes the **Great Compiler Challenge**

Since current processors are big chips (10^9 transistors), their micro-architecture is complex (and GCC has to work a lot for them):

- GHz clock rate
- many functional units working in parallel
- massive L1, L2, L3 caches (access to RAM is very slow, ≈ 1000 cycles)
- out-of-order execution
- branch prediction

Today's x86 processors (AMD Bulldozer, Intel Sandy Bridge) are not like i486 (1990, at 50MHz) running much faster, even if they nearly share the same ia-32 instruction set (in 32 bits mode). **Gcc** needs to optimize differently for AMD than for Intel!

Why is understanding GCC difficult?

- “**Gcc is** not a compiler but **a compiler generation framework**”: (U.Khedker)
 - **a lot of C code** inside **Gcc is generated** at building time.
 - **Gcc** has many **ad-hoc code generators**
(some are simple `awk` scripts, others are big tools coded in many KLOC-s of C)
 - **Gcc** has **several** ad-hoc **formalisms** (perhaps call them *domain specific languages*)
- **Gcc** is growing gradually and does have some legacy (but powerful) code
- **Gcc** has no single architect (“benevolent dictator”):
(no “Linus Torvalds” equivalent for **Gcc**)
- **Gcc source code is heterogenous**:
 - coded in various programming languages (C, C++, Ada ...)
 - coded at very different times, by many people (with various levels of expertise).
 - no unified naming conventions
 - (*my opinion only*;) still weak infrastructure (but powerful)
 - not enough common habits or rules about: memory management, pass roles, debug help, comments, dump files ...
- **Gcc** code is sometimes quite messy (e.g. compared to Gtk).

What you should read on GCC

You should (find lots of resources on the Web, then) read:

- the **Gcc** user documentation

<http://gcc.gnu.org/onlinedocs/gcc/>, giving:

- how to invoke `gcc` (all the obscure optimization flags)
- various language (C, C++) extensions, including attributes and builtins.
- how to contribute to **Gcc** and to report bugs

- the **Gcc internal documentation**

<http://gcc.gnu.org/onlinedocs/gccint/>, explaining:

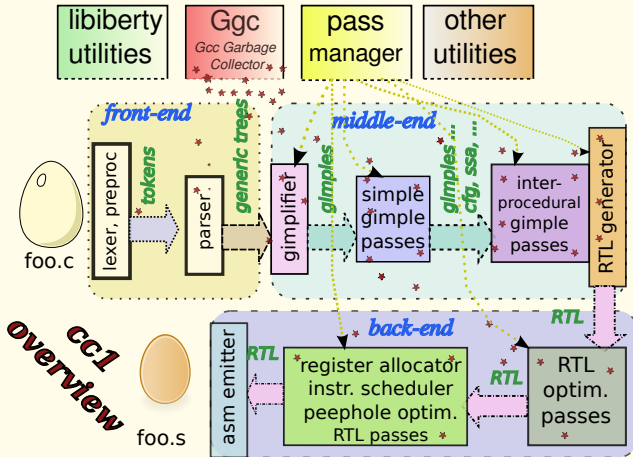
- the overall structure of **Gcc** and its pass management
- major (but not all) internal representations (notably Tree, Gimple, RTL ...).
- memory management, `GTy` annotations, `gengtype` generator
- interface available to plugins
- machine and target descriptions
- LTO internals

- the source code, mostly **header files** `*.h`, **definition files** `*.def`, option files `*.opt`. Don't be lost in **Gcc** monster source code.²⁶

²⁶You probably should avoid reading many `*.c` code files at first.

simplified cc1 internal overview [when running]

figure is simplified and not to scale



utilities and infrastructure

gcc is only a driver (file `gcc/gcc.c`). Most things happen in `cc1`. See file `gcc/toplev.c` for the `toplev_main` function starting `cc1` and others.

There are **many infrastructures and utilities** in `Gcc`

- 1 `libiberty/` to abstract system dependencies
- 2 the **Gcc Garbage Collector** i.e. `Ggc`:
 - a naive precise mark-and sweep garbage collector
 - sadly, not always used (many routines handle data manually, with explicit `free`)
 - runs only between passes, so used **for data shared between passes**
 - **don't handle any local variables** 😞
 - about 1800 `struct` inside `Gcc` are annotated with **GTY annotations**.
 - the **genctype** generator produces marking routines in C out of `GTY`

I love the idea of a garbage collector (but others don't).

I think `Ggc` should be better, and be more used.

- 3 diagnostic utilities
- 4 preprocessor library `libcpp/`
- 5 many hooks (e.g. language hooks to factorize code between C, C++, ObjectiveC)

cc1 front-end

The front-end (see function `compile_file` in `gcc/toplev.c`) is reading the input files of a translation unit (e.g. a `foo.c` file and all `#include-d *.h` files).

- **language specific hooks** are given thru `lang_hooks` global variable, in `$GCCSOURCE/gcc/langhooks.h`
- `$GCCSOURCE/libcpp/` is a common **library** (for C, C++, Objective C...) **for** lexing and **preprocessing**.
- C-like front-end processing happens under `$GCCSOURCE/gcc/c-family/`
- **parsing** happens in `$GCCSOURCE/gcc/c-parser.c` and `$GCCSOURCE/gcc/c-decl.c`, **using manual recursive descent parsing techniques**²⁷ to help syntax error diagnostics.
- abstract syntax **Tree-s** [AST] (and **Generic** to several front-ends)

In `gcc-4.6` **plugins cannot enhance the parsed language**

(except thru events for `#pragma-s` or `__attribute__` etc ...)

²⁷ Gcc don't use LALR parser generators like `yacc` or `bison` for C. 

GCC middle-end

The middle-end is the most important²⁸ (and bigger) **part** of **Gcc**

- it is mostly **independent of** both the **source language** and of the **target machine** (of course, `sizeof(int)` matters in it)
- it **factorizes all the optimizations** reusable for various sources languages or target systems
- it processes (i.e. transforms and enhances) several **middle-end internal** (and interleaved) **representations**, notably
 - 1 declarations and operands represented by **Tree-s**
 - 2 **Gimple** representations (“3 address-like” instructions)
 - 3 Control Flow Graph informations (**Edges**, **Basic Blocks**, ...)
 - 4 Data dependencies
 - 5 **Static Single Assignment** (SSA) variant of **Gimple**
 - 6 many others

I [Basile] am more familiar with the middle-end than with front-ends or back-ends.

²⁸Important to me, since I am a middle-end guy!

Middle End and Link Time Optimization

With LTO, the middle-end representations are both input and output.

- LTO enables optimization across several compilation units, e.g. inlining of a function defined in `foo.cc` and called in `bar.c`
(LTO existed in old proprietary compilers, and in LLVM)
- when compiling source translation units in LTO mode, the generated object `*.o` file contains both:
 - (as always) binary code, relocation directives (to the linker), debug information (for `gdb`)
 - (for LTO) **summaries**, a simplified serialized form of middle-end representations
- when “linking” these object files in LTO mode, `lt01` is a “front-end” to this middle-end data contained in `*.o` files. The program `lt01` is started by the `gcc` driver (like `cc1plus ...`)
- in **WHOPR** mode (whole program optimization), LTO is split in three stages (LGEN = local generation, in parallel; sequential WPA = whole program analysis; LTRANS = local transformation, in parallel).

GCC back-ends

The **back-end**²⁹ is the last layer of `Gcc` (specific to the target machine):

- it contains all **optimizations** (etc ...) **particular to its target system** (notably peephole target-specific optimizations).
- it **schedules** (machine) **instructions**
- it **allocates registers**³⁰
- it emits assembler code (and follows target system conventions)
- it transforms *gimple* (given by middle-end) into back-end representations, notably **RTL** (register transfer language)
- it optimizes the RTL representations
- some of the back-end C code is **generated** by **machine descriptions** * `.md` files.

☹ I [Basile] **don't know much about back-ends**

²⁹A given `cc1` or `ltol` has usually one back-end (except multilib ie `-m32` vs `-m64` on `x86-64`). But `Gcc` source release has many back-ends!

³⁰Register allocation is a very hard art. It has been rewritten many times in `Gcc`.

“meta-programming” C code generators in GCC

Gcc has several internal C code generators (built in `$GCCBUILD/gcc/build/`):

- **gengtype** for Ggc, generating marking code from GTY annotations
- **genhooks** for target hooks, generating `target-hooks-def.h` from `target.def`
- **genattrtab**, **genattr**, **gencodes**, **genconditions**, **gencondmmd**, **genconstants**, **genemit**, **genenums**, **genextract**, **genflags**, **genopinit**, **genoutput**, **genreds**, to generate machine attributes and code from machine description `*.md` files.
- **genautomata** to generate pipeline hazard automaton for instruction scheduling from `*.md`
- **genpeep** to generate peephole optimizations from `*.md`
- **genrecog** to generate code recognizing RTL from `*.md`
- etc ...

(genautomata, gengtype, genattrtab are quite big generators)

GCC pass manager and passes

The **pass manager** is coded in `$GCCSOURCE/gcc/passes.c` and `tree-optimize.c` with `tree-pass.h`

There are many (≈ 250) passes in `Gcc`:

The set of executed passes depend upon optimization flags (`-O1` vs `-O3` ...) and of the translation unit.

- middle-end passes process *Gimple* (and other representations)
 - **simple *Gimple* passes** handle *Gimple* code one function at a time.
 - simple and full **IPA *Gimple* passes** do **Inter-Procedural Analysis** optimizations.
- back-end passes handle *RTL* etc ...

Passes are organized in a tree. A pass may have sub-passes, and could be run several times.

Both middle-end and back-end passes go into `libbackend.a`

Plugins can add (or remove, or monitor) passes.

Garbage Collection inside GCC

`Ggc` is implemented in `$GCCSOURCE/gcc/ggc*. [ch]`³¹ and thru the **gengtype** generator `$GCCSOURCE/gcc/gengtype*. [ch]`.

- the **GTy** annotation (on `struct` and **global or static data**) is used to “declare” `Ggc` handled data and types.
- `gengtype` generates marking and allocating routines in `gt-*.h` and `gtyp*. [ch]` files (in `$GCCBUILD/gcc/`)
- `ggc_collect ()`; calls `Ggc`; it is mostly called by the pass manager.
- 😞 **local pointers** (variables inside `Gcc` functions) are **not preserved** by `Ggc` so `ggc_collect` can't be called³² everywhere!
- ⇒ passes have to copy (pointers to their data) to static `GTy`-ed variables
- so `Ggc` is unfortunately not systematically used (often data local to a pass is manually managed & explicitly freed)

³¹`ggc-zone.c` is often unused.

³²Be very careful if you need to call `ggc_collect` yourself *inside* your pass!

Why real compilers need garbage collection?

- compilers have complex internal representations (≈ 1800 GTY-ed types!)
- compilers are become very big and complex programs
- it is difficult to decide when a compiler data can be (manually) freed
- **circular data structures** (e.g. back-pointers from Gimple to containing Basic Blocks) are common inside compilers; compiler data are not (only) tree-like.
- **liveness** of a data is a **global** (non-modular) property!
- garbage collection techniques are mature
(garbage collection is a global trait in a program)
- memory is quite cheap

In my (strong) opinion, **Ggc** is not very good³³ -but cannot and shouldn't be avoided-, and **should systematically be used**, so improved.
Even today, some people manually sadly manage their data in their pass.

³³Chicken & egg issue here: **Ggc** not good enough \Rightarrow not very used \Rightarrow not improved!

using Ggc in your C code for Gcc

Annotate your `struct` declarations with **GTy** in your C code:

```
// from $GCCSOURCE/gcc/tree.h
struct GTy ((chain_next ("%h.next"), chain_prev ("%h.prev")))
    tree_statement_list_node {
    struct tree_statement_list_node *prev;
    struct tree_statement_list_node *next;
    tree stmt;          // The tree-s are GTy-ed pointers
};

struct GTy(()) tree_statement_list {
    struct tree_typed typed;
    struct tree_statement_list_node *head;
    struct tree_statement_list_node *tail;
};
```

Likewise for global or static variables:

```
extern GTy(()) VEC(alias_pair,gc) * alias_pairs;
```

Notice the poor man's vector "template" thru the **VEC** "mega"-macro (from `$GCCSOURCE/gcc/vec.h`) known by `gengtype`

GTY annotations

<http://gcc.gnu.org/onlinedocs/gccint/Type-Information.html>

Often empty, these annotations help to generate good marking routines:

- skip to ignore a field
- list chaining with `chain_next` and `chain_previous`
- [variable-] array length with `length` and `variable_size`
- discriminated unions with `descr` and `tag` ...
- poor man's genericity with `param2_is` or `use_params` etc ...
- marking hook routine with `mark_hook`
- etc ...

From `tree.h` **gengtype** is generating `gt-tree.h` which is `#include-d` from `tree.c`

Pre Compiled Headers (PCH)³⁴ also use **gengtype** & **GTY**.

³⁴PCH is a feature which might be replaced by “pre-parsed headers” in the future.

Example of **gengtype** generated code

Marking routine:

```
// in $GCCBUILD/gcc/gtype-desc.c
```

```
void gt_ggc_mx_tree_statement_list_node (void *x_p) {
  struct tree_statement_list_node * x = (struct tree_statement_list_node *)x_p;
  struct tree_statement_list_node * xlimit = x;
  while (ggc_test_and_set_mark (xlimit))
    xlimit = ((*xlimit).next);
  if (x != xlimit)
    for (;;) {
      struct tree_statement_list_node * const xprev = ((*x).prev);
      if (xprev == NULL) break;
      x = xprev;
      (void) ggc_test_and_set_mark (xprev);
    }
  while (x != xlimit) {
    gt_ggc_m_24tree_statement_list_node ((*x).prev);
    gt_ggc_m_24tree_statement_list_node ((*x).next);
    gt_ggc_m_9tree_node ((*x).stmt);
    x = ((*x).next);
  }
}
```

Allocators:

```
// in $GCCBUILD/gcc/gtype-desc.h
```

```
#define ggc_alloc_tree_statement_list() \
  ((struct tree_statement_list *) (ggc_internal_alloc_stat (sizeof (struct tree_statement_list) ME
#define ggc_alloc_cleared_tree_statement_list() \
  ((struct tree_statement_list *) (ggc_internal_cleared_alloc_stat (sizeof (struct tree_statement
#define ggc_alloc_vec_tree_statement_list(n) \
  ((struct tree_statement_list *) (ggc_internal_vec_alloc_stat (sizeof (struct tree_statement_lis
```


Ggc work

The **Ggc** garbage collector is a mark and sweep precise collector, so:

- each **Ggc**-aware memory zone has some kind of mark
- first **Ggc** clears all the marks
- then **Ggc** marks all the [global or static] roots³⁵, and “recursively” marks all the (still unmarked) data accessible from them, using routines generated by **gengtype**
- at last **Ggc** frees all the unmarked memory zones

Complexity of **Ggc** is $\approx O(m)$ where m is the **total memory size**.

When not much memory has been allocated, `ggc_collect` returns immediately and don't really run **Ggc**³⁶

Similar trick for pre-compiled headers: compiling a `*.h` file means parsing it and persisting all the roots (& data accessible from them) into a compiled header.

³⁵That is, `extern` or `static` **GTU**-ed variables.

³⁶Thanks to `ggc_force_collect` internal flag.

allocating **GTy**-ed data in your C code

gengtype also generates allocating macros named `ggc_alloc*`. Use them like you would use `malloc ...`.

```
// from function tsi_link_before in $GCCSOURCE/gcc/tree-iterator.c
struct tree_statement_list_node *head, *tail;
// ...
{
    head = ggc_alloc_tree_statement_list_node ();
    head->prev = NULL;  head->next = NULL;  head->stmt = t;
    tail = head;
}
```

Of course, 😊 you **don't** need to **free that memory**: `Ggc` will do it for you. **GTy**-ed allocation never starts automatically a `Ggc` collection³⁷, and has some little cost. Big data can be `GTy`-allocated. Variable-sized data allocation macros get as argument the total size (in bytes) to be allocated.

Often we wrap the allocation inside small inlined “constructor”-like functions.

³⁷Like almost every other garbage collector would do; `Ggc` can't behave like that because it ignores local pointers, but most other GCs handle them!

Pass descriptors

Middle-end and back-end passes are described in structures defined in `gcc/tree-pass.h`. They all are `opt_pass-es` with:

- some **type**, either `GIMPLE_PASS`, `SIMPLE_IPA_PASS`, `IPA_PASS`, or `RTL_PASS`
- some human readable **name**. If it starts with `*` no dump can happen.
- an optional **gate** function “hook”, deciding if the pass (and its optional sub-passes) should run.
- an **execute** function “hook”, doing the actual work of the pass.
- required, provided, or destroyed **properties** of the pass.
- **“to do” flags**
- other fields used by the pass manager to organize them.
- timing identifier `tv_id` (for `-freport-time` program option).

Full IPA passes have more descriptive fields (related to LTO serialization).

Most of file `tree-pass.h` declare pass descriptors, e.g.:

```
extern struct gimple_opt_pass pass_early_ipa_sra;
extern struct gimple_opt_pass pass_tail_recursion;
extern struct gimple_opt_pass pass_tail_calls;
```

A pass descriptor [control flow graph building]

In file `$GCCSOURCE/gcc/tree-cfg.c`

```

struct gimple_opt_pass pass_build_cfg = { {
  GIMPLE_PASS,
  "cfg",           /* name */
  NULL,           /* gate */
  execute_build_cfg, /* execute */
  NULL,           /* sub */
  NULL,           /* next */
  0,              /* static_pass_number */
  TV_TREE_CFG,   /* tv_id */
  PROP_gimple_leh, /* properties_required */
  PROP_cfg,      /* properties_provided */
  0,             /* properties_destroyed */
  0,             /* todo_flags_start */
  TODO_verify_stmts | TODO_cleanup_cfg
  | TODO_dump_func /* todo_flags_finish */
} };

```

Another pass descriptor [tail calls processing]

```

struct gimple_opt_pass pass_tail_calls = { {
  GIMPLE_PASS,
  "tailc",           /* name */
  gate_tail_calls, /* gate */
  execute_tail_calls, /* execute */
  NULL,             /* sub */
  NULL,             /* next */
  0,                /* static_pass_number */
  TV_NONE,          /* tv_id */
  PROP_cfg | PROP_ssa, /* properties_required */
  0,                /* properties_provided */
  0,                /* properties_destroyed */
  0,                /* todo_flags_start */
  TODO_dump_func | TODO_verify_ssa /* todo_flags_finish */ } };

```

This file `$GCCSOURCES/gcc/tree-tailcall.c` contains two related passes, for tail recursion elimination.

Notice that the human name (here "tailc") is unfortunately unlike the C identifier `pass_tail_calls` (so finding a pass by its name can be boring).

IPA pass descriptor: interprocedural constant propagation

```

struct ipa_opt_pass_d pass_ipa_cp = { { // in file $GCCSOURCE/gcc/ipa-cp.c
  IPA_PASS,
  "cp",                /* name */
  cgraph_gate_cp,    /* gate */
  ipcp_driver,        /* execute */
  NULL,                /* sub */
  NULL,                /* next */
  0,                   /* static_pass_number */
  TV_IPA_CONSTANT_PROP, /* tv_id */
  0,                   /* properties_required */
  0,                   /* properties_provided */
  0,                   /* properties_destroyed */
  0,                   /* todo_flags_start */
  TODO_dump_cgraph | TODO_dump_func |
  TODO_remove_functions | TODO_ggc_collect /* todo_flags_finish */
},
  ipcp_generate_summary, /* generate_summary routine for LTO */
  ipcp_write_summary,   /* write_summary routine for LTO */
  ipcp_read_summary,    /* read_summary routine for LTO */
  NULL,                  /* write_optimization_summary */
  NULL,                  /* read_optimization_summary */
  NULL,                  /* stmt_fixup */
  0,                     /* TODOs */
  NULL,                  /* function_transform */
  NULL,                  /* variable_transform */
};

```

RTL pass descriptor: dead-store elimination

```

struct rtl_opt_pass pass_rtl_dse1 = { { // in file $GCCSOURCE/gcc/dse.c
  RTL_PASS,
  "dse1", /* name */
  gate_dse1, /* gate */
  rest_of_handle_dse, /* execute */
  NULL, /* sub */
  NULL, /* next */
  0, /* static_pass_number */
  TV_DSE1, /* tv_id */
  0, /* properties_required */
  0, /* properties_provided */
  0, /* properties_destroyed */
  0, /* todo_flags_start */
  TODO_dump_func |
  TODO_df_finish | TODO_verify_rtl_sharing |
  TODO_ggc_collect /* todo_flags_finish */
} };

```

There is a similar `pass_rtl_dse2` in the same file.

How the pass manager is activated?

Language specific `lang_hooks.parse_file` (e.g. `c_parse_file` in `$GCCSOURCES/gcc/c-parser.c` for **cc1**) is called from `compile_file` in `$GCCSOURCES/gcc/toplev.c`.

When a C function has been entirely parsed by the front-end, `finish_function` (from `$GCCSOURCE/gcc/c-decl.c`) is called. Then

- 1 `c_genericize` in `$GCCSOURCE/gcc/c-family/c-gimplify.c` is called. The C-specific abstract syntax tree (AST) is transformed in **Generic** representations (common to several languages);
- 2 several functions from `$GCCSOURCE/gcc/gimplify.c` are called:
`gimplify_function_tree` → `gimplify_body` → `gimplify_stmt`
→ `gimplify_expr`
- 3 some language-specific gimplification happens thru `lang_hooks.gimplify_expr`, e.g. `c_gimplify_expr` for **cc1**.
- 4 etc ...

Then `tree_rest_of_compilation` (in file `$GCCSOURCE/gcc/tree-optimize.c`) is called.

Pass registration

Passes are **registered** within the pass manager. Plugins indirectly call `register_pass` thru the **PLUGIN_PASS_MANAGER_SETUP** event.

Most **Gcc** core passes are often statically registered, thru lot of code in **init_optimization_passes** like

```

struct opt_pass **p;
#define NEXT_PASS(PASS) (p = next_pass_1 (p, &((PASS).pass)))
p = &all_lowering_passes;
NEXT_PASS (pass_warn_unused_result);
NEXT_PASS (pass_diagnose_omp_blocks); NEXT_PASS (pass_mudflap_1);
NEXT_PASS (pass_lower_omp); NEXT_PASS (pass_lower_cf);
NEXT_PASS (pass_refactor_eh); NEXT_PASS (pass_lower_eh);
NEXT_PASS (pass_build_cfg); NEXT_PASS (pass_warn_function_return);
// etc ...

```

`next_pass_1` calls **make_pass_instance** which clones a pass. Passes may be dynamically duplicated.

Passes are organized in a **hierarchical tree of passes**. Some passes have sub-passes (which run only if the super-pass `gate` function succeeded).

Running the pass manager

Function `tree_rest_of_compilation` calls `execute_all_ipa_transforms` and most importantly **`execute_pass_list`** (`all_passes`) (file `$GCCSOURCE/gcc/passes.c`)
The role of the pass manager is to run passes using **`execute_pass_list`** thru **`execute_one_pass`**.

Some passes have sub-passes \Rightarrow `execute_pass_list` is recursive.

It has specific variants:

(e.g. `execute_ipa_pass_list` or `execute_all_ipa_transforms`, etc...)

Each pass has an **`execute`** function, returning a set of **`to do flags`**, merged with the `todo_finish` flags in the pass.

`To Do actions` are processed by **`execute_todo`**, with code like

```
if (flags & TODO_ggc_collect)
    ggc_collect ();
```

Issues when defining your pass

😊 The **easy** parts:

- **define what your pass should do**
- specify your **gate** function, if relevant
- specify your **exec** function
- define the **properties** and **to-do** flags

☹️ The **difficult** items:

- **position your new pass** within the existing passes
⇒ understand after which pass should you add yours!
- understand **what internal representations are really available**
- understand **what next passes expect!**
- ⇒ understand **which passes are running?**

I [Basile] also have these difficulties !!

pass dump

Usage: pass **-fdump-**-*** program flags³⁸ to gcc

- Each pass can **dump** information **into textual files**.

⇒ your new passes should provide dumps.³⁹

- ⇒ So you could get **hundreds of dump files**:

hello.c → hello.c.000i.cgraph.....hello.c.224t.statistics
(but the **numbering** don't means much 😞, they are **not chronological!**)

- try **-fdump-tree-all -fdump-ipa-all -fdump-rtl-all**

- you can choose your dumps:

- **-fdump-tree- π** to dump the **t**ree or GIMPLE_PASS named π

- **-fdump-ipa- π** to dump the **i.p.a.** SIMPLE_IPA_PASS or IPA_PASS named π

- **-fdump-rtl- π** to dump the **r.t.l.** RTL_PASS named π

- **dump files don't contain all the information**

(and there is no way to parse them)⁴⁰.

³⁸Next gcc-4.7 will have improved [before/after] flags

³⁹Unless the pass name starts with *.

⁴⁰Some Gcc gurus dream of a fully accurate and reparseable textual representation of

Dump example: input source `example1.c`

(using `gcc-melt`⁴¹ svn rev. 174968 \equiv `gcc-trunk` svn rev. 174941, of june 11th 2011)

```
1  /* example1.c */
   extern int gex(int);
3
   int foo(int x, int y) {
5     if (x>y)
       return gex(x-y) * gex(x+y);
7     else
       return foo(y,x);
9  }

11 void bar(int n, int *t) {
    int i;
13    for (i=0; i<n; i++)
        t[i] = foo(t[i], i) + i;
15 }
```

⁴¹The **Melt branch** (not the plugin) is dumping into *chronologically named* files, e.g. `example1.c.%0026.017t.ssa!`

Dump gimplification example1.c.004t.gimple

```

bar (int n, int * t) {
  long unsigned int D.2698;
  long unsigned int D.2699;
  int * D.2700;
  int D.2701; int D.2702; int D.2703;
  int i;
  i = 0;
  goto <D.1597>;
<D.1596>:
D.2698 = (long unsigned int) i;
D.2699 = D.2698 * 4;
D.2700 = t + D.2699;
D.2698 = (long unsigned int) i;
D.2699 = D.2698 * 4;
D.2700 = t + D.2699;
D.2701 = *D.2700;
D.2702 = foo (D.2701, i);
D.2703 = D.2702 + i;
*D.2700 = D.2703;
i = i + 1;

  <D.1597>:
  if (i < n) goto <D.1596>;
  else goto <D.1598>;
  <D.1598>: }

foo (int x, int y) {
  int D.2706; int D.2707; int D.2708;
  int D.2709; int D.2710;
  if (x > y) goto <D.2704>;
  else goto <D.2705>;
  <D.2704>:
  D.2707 = x - y;
  D.2708 = gex (D.2707);
  D.2709 = x + y;
  D.2710 = gex (D.2709);
  D.2706 = D.2708 * D.2710;
  return D.2706;
  <D.2705>:
  D.2706 = foo (y, x);
  return D.2706; }

```

functions in reverse order; 3 operands instructions; generated temporaries; generated **goto-s**

Dump SSA - [part of] example1.c.017t.ssa

only the `foo` function of that dump file, in **Static Single Assignment SSA** form

```
;; Function foo
(foo, funcdef_no=0, decl_uid=1589,
  cgraph_uid=0)
Symbols to be put in SSA form { .MEM }
Incremental SSA update started at block: 0
Number of blocks in CFG: 6
Number of blocks to update: 5 ( 83%)

foo (int x, int y) {
  int D.2710; int D.2709;
  int D.2708; int D.2707; int D.2706;

<bb 2>:
  if (x2(D) > y3(D))
    goto <bb 3>;
  else goto <bb 4>;

  <bb 3>:
    D.27074 = x2(D) - y3(D);
    D.27085 = gex (D.27074);
    D.27096 = x2(D) + y3(D);
    D.27107 = gex (D.27096);
    D.27068 = D.27085 * D.27107;
    goto <bb 5>;

  <bb 4>:
    D.27069 = foo (y3(D), x2(D));

  <bb 5>:
    # D.27061 =  $\Phi$  <D.27068(3), D.27069(4)>
    return D.27061; }
```

SSA \Leftrightarrow each variable is assigned once; suffix (D) for default definitions of SSA names
 e.g `D.27074` [appearing as `D.2707_4` in dump files]

Basic blocks: only entered at their start

ϕ -nodes; “union” of values coming from two edges

IPA dump - [tail of] example1.c.049i.inline

```

;; Function bar (bar, funcdef_no=1,
    decl_uid=1593, cgraph_uid=1)
bar (int n, int * t) {
  int i;
  int D.2703; int D.2702; int D.2701;
  int * D.2700;
  long unsigned int D.2699;
  long unsigned int D.2698;

  # BLOCK 2 freq:900
  # PRED: ENTRY [100.0%] (fallthru,exec)
  goto <bb 4>;
  # SUCC: 4 [100.0%] (fallthru,exec)

  # BLOCK 3 freq:9100
  # PRED: 4 [91.0%] (true,exec)
  D.2698_8 = (long unsigned int) i_1;
  D.2699_9 = D.2698_8 * 4; /// 4 ≡ sizeof(int)
  D.2700_10 = t_6(D) + D.2699_9;
  D.2701_11 = *D.2700_10;
  D.2702_12 = foo (D.2701_11, i_1);
}

D.2703_13 = D.2702_12 + i_1;
*D.2700_10 = D.2703_13;
i_14 = i_1 + 1;
# SUCC: 4 [100.0%]
    (fallthru,dfs_back,exec)

# BLOCK 4 freq:10000
# PRED: 2 [100.0%]
    (fallthru,exec) 3 [100.0%]
    (fallthru,dfs_back,exec)
# i_1 = PHI <0(2), i_14(3)>
if (i_1 < n_3(D))
  goto <bb 3>;
else goto <bb 5>;
# SUCC: 3 [91.0%] (true,exec) 5 [9.0%]

# BLOCK 5 freq:900
# PRED: 4 [9.0%] (false,exec)
return;
# SUCC: EXIT [100.0%]

```

The call to `foo` has been inlined; edges of CFG have frequencies

RTL dump [small part of] example1.c.162r.reginfo

```

;; Function bar (bar, funcdef_no=1, decl_uid=1593,
    cgraph_uid=1)
verify found no changes in insn with uid = 31.
(note 21 0 17 2 [bb 2] NOTE_INSN_BASIC_BLOCK)
(insn 17 21 18 2 (set (reg/v:SI 84 [ n ])
    (reg:SI 5 di [ n ]))
    example1.c:11 64 {*movsi_internal}
    (expr_list:REG_DEAD (reg:SI 5 di [ n ])
    (nil)))
(insn 18 17 19 2 (set (reg/v/f:DI 85 [ t ])
    (reg:DI 4 si [ t ]))
    example1.c:11 62 {*movdi_internal_rex64}
    (expr_list:REG_DEAD (reg:DI 4 si [ t ])
    (nil)))
(note 19 18 23 2 NOTE_INSN_FUNCTION_BEG)
(insn 23 19 24 2 (set (reg:CCNO 17 flags)
    (compare:CCNO (reg/v:SI 84 [ n ])
    (const_int 0 [0])))
    example1.c:13 2 {*cmpsi_ccno_1}
    (nil))
(jump_insn 24 23 25 2 (set (pc)
    (if_then_else (le (reg:CCNO 17 flags)
    (const_int 0 [0]))
    (label_ref:DI 42)
    (pc))) example1.c:13 594 *jcc_1
    (expr_list:REG_DEAD (reg:CCNO 17 flags)
    (expr_list:REG_BR_PROB (const_int 900 [0]
    (nil))))
-> 42)
(note 25 24 26 3 [bb 3] NOTE_INSN_BASIC_BLOCK)
(insn 26 25 20 3 (set (reg:DI 82 [ ivtmp.14 ])
    (reg/v/f:DI 85 [ t ])) 62 {*movdi_internal_rex64}
    (expr_list:REG_DEAD (reg/v/f:DI 85 [ t ])
    (nil)))
(insn 20 26 37 3 (set (reg/v:SI 78 [ i ])
    (const_int 0 [0])) example1.c:13 64
    {*movsi_internal}
    (nil))
(code_label 37 20 27 4 9 "" [1 uses])
(note 27 37 29 4 [bb 4] NOTE_INSN_BASIC_BLOCK)
(insn 29 27 30 4 (set (reg:SI 4 si)
    (reg/v:SI 78 [ i ])) example1.c:14 64 {*movsi_internal_rex64}
    (nil))
(insn 30 29 31 4 (set (reg:SI 5 di)
    (mem:SI (reg:DI 82 [ ivtmp.14 ])
    [2 MEM[base: D.2731_28, offset: 0B]+0 S
    example1.c:14 64 {*movsi_internal_rex64}
    (nil)))
    (nil))
/// etc...

```

I [Basile] can't explain it ☺; but notice x86 specific code

generated assembly [part of] example1.s

```

.file "example1.c"

# options enabled: -fasynchronous-unwind-tables
# -fauto-inc-dec
## etc etc etc ...
# -fverbose-asm -fzee -fzero-initialized-in-bss .L9:
# -m128bit-long-double -m64 -m80387
# -maccumulate-outgoing-args -malign-stringops
# -mfancy-math-387 mfp-ret-in-387 -mglibc
# -mieee-fp -mmmx -mno-sse4 -mpush-args
# -mred-zone msse -msse2 -mtls-direct-seg-refs

.globl bar
.type bar, @function

bar:
.LFB1:
.cfi_startproc
pushq %r12 #
.cfi_def_cfa_offset 16
.cfi_offset 12, -16
testl %edi, %edi # n
movl %edi, %r12d # n, n
pushq %rbp #
.cfi_def_cfa_offset 24
.cfi_offset 6, -24
pushq %rbx #
.cfi_def_cfa_offset 32
.cfi_offset 3, -32

jle .L7 #,
movq %rsi, %rbp # t, ivtmp.14
xorl %ebx, %ebx # i
.p2align 4,,10
.p2align 3

movl 0(%rbp), %edi # MEM[base: D.273
movl %ebx, %esi # i,
call foo #
addl %ebx, %eax # i, tmp86
addl $1, %ebx #, i
movl %eax, 0(%rbp) # tmp86, MEM[base
addq $4, %rbp #, ivtmp.14
cmpl %r12d, %ebx # n, i
jne .L9 #,

popq %rbx #
.cfi_def_cfa_offset 24
popq %rbp #
.cfi_def_cfa_offset 16
popq %r12 #
.cfi_def_cfa_offset 8
ret .cfi_endproc

.LFE1:
.size bar, .-bar
.ident "GCC: (GNU) 4.7.0 20110611 (exper
[trunk revision 174943]"

.section .note.GNU-stack,"",@progb

```

Order of executed passes; running gimple passes

- When **cc1 don't get** the **-quiet** program argument, names of executed **IPA** passes are printed.
- Plugins know about executed passes thru **PLUGIN_PASS_EXECUTION** events.
- global variable **current_pass**
- understanding all the executed passes is not very simple

Simple **GIMPLE_PASS**-es are executed one (compiled) function at a time.

- global **cfun** points to the **current function** as a **struct function** from `$GCCSOURCE/gcc/function.h`
- global **current_function_decl** is a `tree`
- `cfun` is `NULL` for non-gimple passes (i.e. `IPA_PASS`-es)

running inter-procedural passes

They obviously work on the whole compilation unit, so run “once”⁴².

Using the `cgraph_nodes` global from `$GCCSOURCE/gcc/cgraph.h`, they often do

```
struct cgraph_node *node;
for (node = cgraph_nodes; node; node = node->next) {
    if (!gimple_has_body_p (node->decl)
        || node->clone_of)
        continue;
    // do something useful with node
}
```

If `node->decl` is a `FUNCTION_DECL` tree, we can retrieve its body (a sequence of *Gimple*-s) using `gimple_body` (from `$GCCSOURCE/gcc/gimple.h`).

However, often that body is not available, because only the control flow graph exist at that point. We can use `DECL_STRUCT_FUNCTION` to retrieve a `struct function`, then `ENTRY_BLOCK_PTR_FOR_FUNCTION` to get a `basic_block`, etc...

⁴²But the pass manager could run again such a pass.

Plugins

- I [Basile] think that: **plugins are a very important feature of Gcc** , but
 - most Gcc **developers don't care**
 - some Gcc hackers are against them
 - Gcc has no stable API [yet?], no binary compatibility
Gcc internals are under-documented
 - plugins are dependent upon the version of Gcc
 - FSF was hard to convince (plugins required changes in licensing)
 - attracting outside developers to make plugins is hard

please code Gcc plugins or extensions (using Melt)
- There are still [too] **few plugins**:
TreeHydra (Mozilla), DragonEgg (LLVM), Milepost/Ctuning??, MELT, etc ...
- **plugins should be GPL compatible free software**
(GCC licence probably forbids to use proprietary Gcc plugins).
- some distributed Gcc compilers have disabled plugins ☹
- plugins might not work
(e.g. a plugin started from `lt_o1` can't do front-end things like registering pragmas)

Why code [plugins in C or] Gcc extensions [in MELT]

IMHO:

- Don't code plugins for features which should go in core Gcc
- You can't do everything thru plugins, e.g. a new front-end for a new language.

Gcc extensions (plugins in C, or extensions in MELT) are useful for:

- **research** and prototyping (of new compilation techniques)
- **specific processing of source code** (which don't have its place inside Gcc core):
 - coding rules validation (e.g. Misra-C, Embedded C++, DOI178?, ...), including library or software specific rules
(e.g. every `pthread_mutex_lock` should have its matching `pthread_mutex_unlock` in the same function or block)
 - improved type checking
(e.g. typing of variadic functions like `g_object_set` in Gtk)
 - specific optimizations - (e.g. `fprintf(stdout, ...)` → `printf(...)`)

Such specific processing don't have its place inside Gcc itself, because it is tied to a particular { domain, corporation, community, software ... }

dreams of Gcc extensions [in MELT]

You could dare coding these as Gcc plugins in plain C, but even as Melt extensions it is not easy!

- **Hyper-optimization** extensions i.e. $-O\infty$ optimization level ☺
Gcc guidelines require that passes execute in linear time; but some clever optimizations are provided by cubic or exponential algorithms; some particular users could afford them.
- **Clever warnings** and **static analysis**
 - a free competitor to Coverity™
idea explored in a Google Summer of Code 2011 project by Pierre Vittet, e.g. <https://github.com/Piervit/GMWarn>
 - application specific analysis
Alexandre Lissy, *Model Checking the Linux Kernel*
- tools support for large free software (Kde?, Gnome?, ...)

Free Software wants⁴³ you to code Gcc extensions!

⁴³Or is it just me ☺?

Running plugins

- Users can run plugins with program options to **gcc** like
 - fplugin=/path/to/name.so
 - fplugin-arg-name-key[=value]
- With a short option **-fplugin=name** plugins are loaded from a predefined plugin directory⁴⁴ as
 - fplugin='gcc -print-file-name=plugin`/name.so
- Several plugins can be loaded in sequence.
- **Gcc** accept plugins only on ELF systems (e.g. **Gnu/Linux**) with `dlopen`, provided plugins have been enabled at configuration time.
- the plugin is **dlopen**-ed by **cc1** or `cc1plus` or even `ltol1` (caveat: front-end functions are not in `ltol1`)

⁴⁴This could be enhanced in next `gcc-4.7` with language-specific subdirectories.

Plugin as used from Gcc core

Details on gcc.gnu.org/onlinedocs/gccint/Plugins.html; see also file `$GCCSOURCE/gcc/gcc-plugin.h` (which gets installed under the plugin directory)

`cc1` (or `lto1`, ...) is initializing plugins quite early (before parsing the compilation unit or running passes). It checks that `plugin_is_GPL_compatible` then run the plugin's `plugin_init` function (which gets version info, and arguments, etc...)

Inside `Gcc`, plugins are invoked from several places, e.g. `execute_one_pass` calls

```
invoke_plugin_callbacks (PLUGIN_PASS_EXECUTION, pass);
```

The `PLUGIN_PASS_EXECUTION` is a **plugin event**. Here, the `pass` is the event-specific **gcc data** (for many events, it is `NULL`). There are ≈ 20 events (and more could be dynamically added, e.g. for one plugin to hook other plugins.).

Event registration from plugins

Plugins should register the events they are interested in, usually from their `plugin_init` function, with a callback of type

```
/* The prototype for a plugin callback function.
   gcc_data - event-specific data provided by GCC
   user_data - plugin-specific data provided by the plug-in. */
typedef void (*plugin_callback_func)
             (void *gcc_data, void *user_data);
```

Plugins register their callback function `callback` of above type `plugin_callback_func` using **register_callback** (from file `$GCCSOURCE/gcc/gcc-plugin.h`), e.g. from `melt-runtime.c`

```
register_callback (/*name:*/ melt_plugin_name,
                  /*event:*/ PLUGIN_PASS_EXECUTION,
                  /*callback:*/ melt_passexec_callback,
                  /*no user_data:*/ NULL);
```

Adding or replacing passes in a plugin

(you should know where to add your new pass!)

Use `register_callback` with a struct `register_pass_info` data but no callback, e.g. to register `yourpass` *after* the pass named "cfg":

```
struct register_pass_info passinfo;
memset (&passinfo, 0, sizeof (passinfo));
passinfo.pass = (struct opt_pass*) yourpass;
passinfo.reference_pass_name = "cfg";
passinfo.ref_pass_instance_number = -1;
passinfo.pos_op = PASS_POS_INSERT_AFTER;
register_callback (plugin_info->base_name, PLUGIN_PASS_MANAGER_SETUP,
                 /*no callback routine*/ NULL,
                 &passinfo);
```

The `pos_op` could also be `PASS_POS_INSERT_BEFORE` or `PASS_POS_REPLACE`.

Main plugin events

A **non-exhaustive list** (extracted from `$GCCSOURCE/gcc/plugin.def`), with the role of the optional *gcc data*:

- 1 **PLUGIN_START** (called from `toptev.c`) called before `compile_file`
- 2 **PLUGIN_FINISH_TYPE**, called from `c-parser.c` with the new type `tree`
- 3 **PLUGIN_PRE_GENERICIZE** (from `c-parser.c`) to see the low level AST in C or C++ front-end, with the new function `tree`
- 4 **PLUGIN_GGC_START** or **PLUGIN_GGC_END** called by `Ggc`
- 5 **PLUGIN_ATTRIBUTES** (from `attribs.c`) or **PLUGIN_PRAGMAS** (from `c-family/c-pragma.c`) to register additional attributes or pragmas from front-end.
- 6 **PLUGIN_FINISH_UNIT** (called from `toptev.c`) can be used for LTO summaries
- 7 **PLUGIN_FINISH** (called from `toptev.c`) to signal the end of compilation
- 8 **PLUGIN_ALL_PASSES_{START,END}**, **PLUGIN_ALL_IPA_PASSES_{START,END}**, **PLUGIN_EARLY_GIMPLE_PASSES_{START,END}** are related to passes
- 9 **PLUGIN_PASS_EXECUTION** identify the given `pass`, and **PLUGIN_OVERRIDE_GATE** (with `&gate_status`) may override gate decisions

Contents

- 1 Introduction
 - about you and me
 - about GCC and MELT
 - building GCC
- 2 GCC Internals
 - complexity of GCC
 - overview inside GCC (cc1)
 - memory management inside GCC
 - optimization passes
 - plugins
- 3 MELT
 - why MELT?
 - handling GCC internal data with MELT
 - matching GCC data with MELT
 - future work on MELT

Motivations for MELT

Gcc extensions address a limited number of users⁴⁵, so their development should be facilitated (cost-effectiveness issues)

- extensions should be [meta-] plugins, not **Gcc** variants [branches, forks]⁴⁶ which are never used
⇒ **extensions** delivered for and **compatible with Gcc releases**
- when understanding **Gcc** internals, coding plugins in plain **C** is very hard (because **C** is a system-programming low-level language, not a high-level symbolic processing language)
⇒ a **higher-level language** is useful
- garbage collection - even inside passes - eases development for (complex and circular) compiler data structures
⇒ **Ggc** is not enough : a **G-C working inside passes** is needed
- Extensions filter or search existing **Gcc** internal representations
⇒ **powerful pattern matching** (e.g. on *Gimple*, *Tree-s*, ...) is needed

⁴⁵Any development useful to all **Gcc** users should better go inside **Gcc** core!

⁴⁶Most **Gnu/Linux** distributions don't even package **Gcc** branches or forks.

Embedding a scripting language is impossible

Many scripting or high-level languages ⁴⁷ can be embedded in some other software:
Lua, Ocaml, Python, Ruby, Perl, many Scheme-s, etc ...

But in practice **this is not doable** for Gcc (we tried one month for Ocaml) :

- mixing two garbage collectors (the one in the language & Ggc) is error-prone
- Gcc has many existing **GT**-ed types
- the Gcc API is huge, and still evolving
(glue code for some scripting implementation would be obsolete before finished)
- since some of the API is low level (accessing fields in `struct-s`), glue code would have big overhead \Rightarrow performance issues
- Gcc has an ill-defined, non “functional” [e.g. with only true functions] OR “object-oriented” API; e.g. iterating is not always thru functions and callbacks:

```
/* iterating on every gimple stmt inside a basic block bb */  
for (gimple_stmt_iterator gsi = gsi_start_bb (bb);  
     !gsi_end_p (gsi); gsi_next (&gsi)) {  
    gimple stmt = gsi_stmt (gsi); /* handle stmt ...*/ }  
}
```

⁴⁷Pedantically, languages' *implementations* can be embedded!

Melt, a Domain Specific Language translated to C

Melt is a **DSL** translated to C in the **style required** by Gcc

- C code generators are usual inside Gcc
- the Melt-generated C code is designed to fit well into Gcc (and Ggc)
- mixing small chunks of C code with Melt is easy
- Melt contains linguistic devices to help Gcc-friendly C code generation
- generating C code eases integration into the evolving Gcc API

The Melt language itself is tuned to fit into Gcc

In particular, it handles both its own Melt values and existing Gcc stuff

The Melt translator is bootstrapped, and Melt extensions are loaded by the `melt.so` plugin

With Melt, Gcc **may generate C code** while running, compiles it⁴⁸ into a Melt binary `.so` module and `dlopen-s` that module.

⁴⁸By invoking `make` from `melt.so` loaded by `cc1`; often that `make` will run another `gcc -fPIC`

Melt values vs Gcc stuff

Melt handles **first-citizen Melt values**:

- values **like many scripting languages have** (Scheme, Python, Ruby, Perl, even Ocaml ...)
- **Melt values are dynamically typed**⁴⁹, organized in a lattice; **each Melt value has its discriminant** (e.g. its class if it is an object)
- you should prefer dealing with Melt values in your Melt code
- values have their **own garbage-collector** (above Gcc), invoked implicitly

But Melt can also handle ordinary Gcc **stuff**:

- stuff is usually any **GTY-ed Gcc raw data**, e.g. **tree, gimple, edge, basic_block** or even **long**
- stuff is **explicitly typed** in Melt code thru **c-type annotations** like **:tree, :gimple** etc.
- adding new ctypes is possible (some of the Melt runtime is generated)

⁴⁹Because designing a type-system friendly with Gcc internals mean making a type theory of Gcc internals!

Things = (Melt Values) \cup (Gcc Stuff)

things	Melt values	Gcc stuff
memory manager	Melt GC (implicit, as needed, even inside passes)	Ggc (explicit, between passes)
allocation	quick , in the birth zone	ggc_alloc, by various zones
GC technique	copying generational (old \rightarrow ggc)	mark and sweep
GC time	$O(\lambda)$ λ = size of young live objects	$O(\sigma)$ σ = total memory size
typing	dynamic, with discriminant	static, GTy annotation
GC roots	local and global variables	only global data
GC suited for	many short-lived temporary values	quasi-permanent data
GC usage	in generated C code	in hand-written code
examples	lists, closures, hash-maps, boxed tree-s, objects ...	raw tree stuff, raw gimple ...

Melt garbage collection

- co-designed with the **Melt** language
- co-implemented with the **Melt** translator
- manage only **Melt** values
all **Gcc** raw stuff is still handled by **Ggc**
- **copying generational Melt garbage collector** (for **Melt** values only):
 - 1 **values quickly allocated** in birth region
(just by incrementing a pointer; a **Melt GC** is triggered when the birth region is full.)
 - 2 **handle** well very **temporary values** and **local variables**
 - 3 **minor Melt GC**: scan local values (in **Melt** call frames), copy and move them out of birth region into **Ggc** heap
 - 4 **full Melt GC** = minor GC + `ggc_collect ()`; ⁵⁰
 - 5 all local pointers (local variables) are in **Melt** frames
 - 6 needs a write barrier (to handle old \rightarrow young pointers)
 - 7 requires tedious C coding: call frames, barriers, **normalizing nested expressions** ($z = f(g(x), y) \rightarrow \text{temporary } \tau = g(x); z = f(\tau, y);$)
 - 8 **well suited for generated C code**

⁵⁰So **Melt** code can trigger **Ggc** collection even **inside** **Gcc** passes!

a first silly example of Melt code

Nothing meaningful, to give a **first taste of Melt language**:

```
;; -*- lisp -*- MELT code in firstfun.melt
(defun foo (x :tree t)
  (tuple x
    (make_tree discr_tree t)))
```

- comments start with `;` up to EOL; case is not meaningful: `defun` \equiv `deFUN`
- **Lisp-like syntax**: `(operator operands ...)` so
parenthesis are always significant in Melt `(f)` \neq `f`, but in `C` `f()` \neq `f` \equiv `(f)`
- `defun` is a “macro” for *defining functions* in Melt
- Melt is an **expression based language**: everything is an expression giving a result
- `foo` is here the name of the defined function
- `(x :tree t)` is a formal arguments list (of *two* formals `x` and `t`); the “ctype keyword” `:tree` qualifies next formals (here `t`) as raw `Gcc tree-s stuff`
- `tuple` is a “macro” to **construct a tuple** value - here made of 2 component values
- `make_tree` is a “primitive” operation, to **box** the raw tree stuff `t` **into a value**
- `discr_tree` is a “predefined value”, a discriminant object for boxed tree values

generated C code from previous example

The [low level] C code, has **more than 680 lines** in generated `firstfun.c`, including

```

melt_ptr_t MELT_MODULE_VISIBILITY
meltrout_1_firstfun_FOO
(meltclosure_ptr_t cloop_,
 melt_ptr_t firstargp_,
 const melt_argdescr_cell_t xargdescr[],
 union meltparam_un *xargtab_,
 const melt_argdescr_cell_t xresdescr[],
 union meltparam_un *xrestab_)
{
  struct frame_meltrout_1_firstfun_FOO_st {
    int m CFR_nbvar;
  #if ENABLE_CHECKING
    const char *m CFR_flocs;
  #endif
    struct meltclosure_st *m CFR_clos;
    struct exceptmelt_st *m CFR_exh;
    struct callframe_melt_st *m CFR_prev;
    void *m CFR_varptr[5];
    tree loc_TREE_o0;
  } *framptr_ = 0, meltfram_;
  memset (&meltfram_, 0, sizeof (meltfram_));
  meltfram_.m CFR_nbvar = 5;
  meltfram_.m CFR_clos = cloop_;
  meltfram_.m CFR_prev
    = (struct callframe_melt_st *) melt_topframe;
  melt_topframe
    = (struct callframe_melt_st *) &meltfram_;
  MELT_LOCATION ("firstfun.melt:2:/ getarg");
  #ifndef MELTGCC_NOLINENUMBERING
  #line 2 "firstfun.melt" /*:::getarg:::*/
  #endif /*MELTGCC_NOLINENUMBERING */
  /*_X_V2*/ meltfptr[1] = (melt_ptr_t) firstargp_;
  if (xargdescr[0] != MELTBPARG_TREE)
    goto lab_endgetargs;
  /*?*/ meltfram_.loc_TREE_o0 = xargtab[0].meltbparg;
  lab_endgetargs;
  /*_MAKE_TREE_V3*/ meltfptr[2] =
  #ifndef MELTGCC_NOLINENUMBERING
  #line 4 "firstfun.melt" /*:::expr:::*/
  #endif /*MELTGCC_NOLINENUMBERING */
  (meltgc_new_tree
   ((meltobject_ptr_t) (( /*!DISCR_TREE */ meltfptr[1]
    ( /*?*/ meltfram_.loc_TREE_o0))));
  {
    struct meltletrec_1_st {
      struct MELT_MULTIPLE_STRUCT (2) rtup_0_TUPLREC_
        long meltletrec_1_endgap;
    } *meltletrec_1_ptr = 0;
    meltletrec_1_ptr = (struct meltletrec_1_st *)
      meltgc_allocate (sizeof (struct meltletrec_1_st)
/*_TUPLREC_V5*/ meltfptr[4] =
  (void *) &meltletrec_1_ptr->rtup_0_TUPLREC_x1;
meltletrec_1_ptr->rtup_0_TUPLREC_x1.discr =
  (meltobject_ptr_t) (((void *)
    (MELT_PREDEF (DISCR_MULTIPLE)))));
    meltletrec_1_ptr->rtup_0_TUPLREC_x1.nbval = 2;
    ((meltmultiple_ptr_t) ( /*_TUPLREC_V5*/ meltfptr[1]
      (melt_ptr_t) ( /*_X_V2*/ meltfptr[1]));
    ((meltmultiple_ptr_t) ( /*_TUPLREC_V5*/ meltfptr[2]
      (melt_ptr_t) ( /*_MAKE_TREE_V3*/ meltfptr[2]));
    meltgc_touch ( /*_TUPLREC_V5*/ meltfptr[4]);
    /*_RETVAL_V1*/ meltfptr[0] = /*_TUPLREC_V4*/ meltfptr[0];

```

“hello world” in Melt, a mix of Melt and C code

```
;; file helloworld.melt
(code_chunk helloworldchunk
  #{ /* our $HELLOWORLDCHUNK */ int i=0;
  $HELLOWORLDCHUNK#_label:
  printf("hello world from MELT %d\n", i);
  if (i++ < 3) goto $HELLOWORLDCHUNK#_label; }# )
```

- **code_chunk** is to Melt what **asm** is to C: for **inclusion** of chunks in the **generated** code (C for Melt, assembly for C or gcc); rarely useful, but we can't live without!
- **helloworldchunk** is the **state symbol**; it gets **uniquely expanded**⁵¹ in the generated code (as a C identifier unique to the C file)
- **#{** and **}#** delimit **macro-strings**, lexed by Melt as a list of symbols (when prefixed by \$) and strings: `#{A"$B#C"\n"}#` \equiv `("A\" \" \" b \"C\"\\\"\\n")` [a 3-elements list, the 2nd is symbol **b**, others are strings]

⁵¹ Like Gcc predefined macro `__COUNTER__` or Lisp's gensym

running our helloworld.melt program

Notice that it has no `defun` so don't define any `Melt` function.

It has one single expression, useful for its side-effects!

With the `Melt branch`:

```
gcc-melt -fmelt-mode=runfile \  
-fmelt-arg=helloworld.melt -c example1.c
```

With the `Melt plugin`:


```
gcc-4.6 -fplugin=melt -fplugin-arg-melt-mode=runfile \  
-fplugin-arg-melt-arg=helloworld.melt -c example1.c
```

Run as

```
ccl: note: MELT generated new file  
/tmp/GCCMeltTmpdir-1c5b3a95/helloworld.c  
ccl: note: MELT has built module  
/tmp/GCCMeltTmpdir-1c5b3a95/helloworld.so in 0.416 sec.  
hello world from MELT  
hello world from MELT  
hello world from MELT  
hello world from MELT  
ccl: note: MELT removed 3 temporary files  
from /tmp/GCCMeltTmpdir-1c5b3a95
```

How Melt is running

- Using **Melt** as plugin is the same as using the **Melt** branch: $\forall\alpha\forall\sigma$
`-fmelt- α = σ` in the **Melt** branch
 \equiv `-fplugin-arg-melt- α = σ` with the **melt.so** plugin
- for **development, the Melt branch**⁵² could be preferable
 (more checks and debugging features)
- **Melt** don't do anything more than **Gcc** without a **mode**
 - so without any mode, `gcc-melt` \equiv `gcc-trunk`
 - use `-fmelt-mode=help` to get the list of modes
 - your **Melt** extension usually registers additional mode[s]
- **Melt is not a Gcc front-end**
 so you need to pass a *C* (or *C++*, ...) input file to `gcc-melt` or `gcc`
 often with `-c empty.c` or `-x c /dev/null`
 when asking **Melt** to translate your **Melt** file
- **some Melt modes run a make** to compile thru `gcc -fPIC` the
 generated *C* code; **most of the time is spent in** that `make` **compiling**
 the generated *C* code

⁵²The trunk is often merged (weekly at least) into the **Melt** branch. 

Melt modes for translating `*.melt` files

(usually run on `empty.c`)

The name of the `*.melt` file is passed with `-fmelt-arg=filename.melt`

The **mode** μ passed with `-fmelt-mode= μ`

- **runfile** to **translate** into a `C` file, make the `filename.so` Melt module, load it, **then discard everything**.
- **translatedebug** to translate into a `.so` Melt module built with `gcc -fPIC -g`
- **translatefile** to translate into a `.c` generated `C` file
- **translatetomodule** to translate into a `.so` Melt module (keeping the `.c` file).

Sometimes, **several** `C` files `filename.c`, `filename+01.c`, `filename+02.c`, ... are generated from your `filename.melt`

A single Melt module `filename.so` is generated, to be `dlopen`-ed by Melt you can pass `-fmelt-extra= $\mu_1:\mu_2$` to also load your μ_1 & μ_2 modules

expansion of the `code_chunk` in generated C

389 lines of generated C, including comments, `#line`, empty lines, with:

```
{
#ifdef MELTGCC_NOLINENUMBERING
#line 3
#endif
    int i=0; /* our HELLOWORLDCHUNK__1 */
        HELLOWORLDCHUNK__1_label: printf("hello world from MELT\n");
        if (i++ < 3) goto HELLOWORLDCHUNK__1_label; ;
    ;
}
```

Notice the **unique expansion** `HELLOWORLDCHUNK__1` of the **state symbol** `helloworldchunk`

Expansion of code with holes given thru *macro-strings* is central in **Melt**

Why Melt generates so many C lines?

- **normalization** requires lots of temporaries
- **translation** to C is “straightforward” ☺
- the **generated C code is very low-level!**
- code for **forwarding local pointers** (for Melt copying GC) is generated
- most of the code is in the **initialization**:
 - the generated `start_module_melt` takes a parent environment and produces a new environment
 - uses hooks in the `INITIAL_SYSTEM_DATA` predefined value
 - creates a new environment (binding **exported** variables)
 - Melt don't generate any “data” : all the data is built by (sequential, boring, huge) code in `start_module_melt`
- the Melt language is higher-level than C
- ratio of 10-35 lines of generated C code for one line of Melt is not uncommon
- ⇒ the **bottleneck** is the **compilation by gcc -fPIC** thru `make` **of the generated C code**

Gcc internal representations

Gcc has several “inter-linked” representations:

- **Generic** and **Tree**-s in the front-ends
(with language specific variants or extensions)
- **Gimple** and others in the middle-end
 - **Gimple** operands are **Tree**-s
 - Control Flow Graph **Edge**-s, **Basic Block**-s, **Gimple Seq**-ences
 - use-def chains
 - **Gimple/SSA** is a **Gimple** variant
- **RTL** and others in the back-end

A given representation is defined by many **GTy**-ed *C* types
(discriminated unions, “inheritance”, ...)

tree, **gimple**, **basic_block**, **gimple_seq**, **edge** ... **are typedef-ed pointers**

Some representations have various roles

Tree both for declarations and for **Gimple** arguments

in gcc-4.3 or before *Gimples* were *Trees*


Why a Lisp-y syntax for Melt

True reason: I [Basile] **am lazy** 😊, also

- **Melt** is bootstrapped
 - now **Melt** translator⁵³ is written in **Melt**
`$GCCMELTSOURCE/gcc/melt/warmelt-*.melt`
 \Rightarrow the **C translation** of **Melt** translator is **in its source repository**⁵⁴
`$GCCMELTSOURCE/gcc/melt/generated/warmelt-*.c`
 - parts of the **Melt** runtime (G-C) are generated
`$GCCMELTSOURCE/gcc/melt/generated/meltrunsup*. [ch]`
 - major dependency of **Melt** translator is **Ggc**⁵⁵
- reading in `melt-runtime.c` **Melt** syntax is nearly trivial
- as in many Lisp-s or Scheme-s, most of the parsing work is done by **macro-expansion** \Rightarrow modular syntax (extensible by advanced users)
- **existing support for Lisp** (Emacs mode) works for **Melt**
- **familiar look** if you know **Emacs Lisp**, **Scheme**, **Common Lisp**, or **Gcc .md**

⁵³ **Melt** started as a Lisp program

⁵⁴ This is unlike other C generators inside **Gcc**

⁵⁵ The **Melt** translator almost don't care of `tree-s` or `gimple-s` 

Why and how Melt is bootstrapped

- Melt delivered in both original `.melt` & translated `.c` forms
gurus could `make upgrade-warmelt` to regenerate all generated code in source tree.
- at installation, Melt translates itself several times
(most of installation time is spent in those [re]translations and in compiling them)
- \Rightarrow the Melt translator is a good test case for Melt;
it exercises its runtime and itself (and Gcc do likewise)
- historically, Melt translator written using less features than those newly implemented (e.g. pattern matching rarely used in translator)

main Melt traits [inspired by Lisp]

- **let** : define *sequential local bindings* (like **let*** in Scheme) and evaluate sub-expressions with them
letrec : define co-**recursive** local constructive bindings
- **if** : simple **conditional expression** (like **?:** in C)
cond : complex **conditional expression** (with several conditions)
- **instance** : build dynamically a new Melt object
definstance : define a static instance of some class
- **defun** : define a named function
lambda : build dynamically an anonymous function closure
- **match** : for **pattern-matching**⁵⁶
- **setq** : assignment
- **forever** : infinite loop, exited with **exit**
- **return** : return from a function
may return several things at once (primary result should be a value)
- **multicall** : call with several results

⁵⁶a huge generalization of **switch** in C

non Lisp-y features of MELT

Many linguistic devices to **describe how to generate C** code

- **code_chunk** to include bits of C
- **defprimitive** to define primitive operations
- **defciterator** to define iterative constructs
- **defcmatcher** to define matching constructs

Values vs stuff :

- **c-type** like **:tree**, **:long** to annotate stuff (in formals, bindings, ...) and **:value** to annotate values
- **quote**, with lexical convention $'\alpha \equiv (\text{quote } \alpha)$
 - **(quote 2)** $\equiv '2$ is a boxed constant integer (but 2 is a constant long thing)
 - **(quote "ab")** $\equiv '"ab"$ is a boxed constant string
 - **(quote x)** $\equiv 'x$ is a constant symbol (instance of `class_symbol`)

quote in MELT is different than **quote** in Lisp or Scheme.

In MELT it makes constant boxed values, so $'2 \neq 2$

defining your mode and pass in Melt

code by Pierre Vittet in his `GMWarn` extension

```
(defun test_fopen_docmd (cmd moduldata)
  (let ( (test_fopen           ;a local binding!
        (instance class_gcc_gimple_pass
                   :named_name ' "melt_test_fopen"
                   :gccpass_gate test_fopen_gate
                   :gccpass_exec test_fopen_exec
                   :gccpass_data (make_maptree discr_map_trees 1000)
                   :gccpass_properties_required ()
        ))) ;body of the let follows:
    (install_melt_gcc_pass test_fopen "after" "ssa" 0)
    (debug_msg test_fopen "test_fopen_mode installed test_fopen")
    ;; return the pass to accept the mode
    (return test_fopen)))

(definstance test_fopen class_melt_mode
  :named_name ' "test_fopen"
  :meltmode_help ' "monitor that after each call to fopen, there is a tes
  :meltmode_fun test_fopen_docmd
)

(install_melt_mode test_fopen)
```

Gcc *Tree-s*

A central front-end and middle-end representation in Gcc:
in C the type `tree` (a pointer)

See files `$GCCSOURCE/gcc/tree.{def,h,c}`, and also
`$GCCSOURCE/gcc/c-family/c-common.def` and other front-end
dependent files `#include-d` from `$GCCBUILD/gcc/all-tree.def`

`tree.def` contains \approx **190** definitions like

```
/* 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)
```

Or

```
/* C's float and double.  Different floating types are distinguished
   by machine mode and by the TYPE_SIZE and the TYPE_PRECISION.  */
DEFTREECODE (REAL_TYPE, "real_type", tcc_type, 0)
```

Tree representation in C

`tree.h` contains

```

struct GTY(()) tree_base {
    ENUM_BITFIELD(tree_code) code : 16;
    unsigned side_effects_flag : 1;
    unsigned constant_flag : 1;
    // many other flags
};
struct GTY(()) tree_typed {
    struct tree_base base;
    tree type;
};
// etc
union GTY ((ptr_alias (union lang_tree_node),
            desc ("tree_node_structure (&%h)", variable_size))) tree_node {
    struct tree_base GTY ((tag ("TS_BASE"))) base;
    struct tree_typed GTY ((tag ("TS_TYPED"))) typed;
    // many other cases
    struct tree_target_option GTY ((tag ("TS_TARGET_OPTION"))) target_option;
};

```

But `$GCCSOURCE/gcc/coretypes.h` has

```
typedef union tree_node *tree;
```

Gcc *Gimple-s*

Gimple-s represents instructions in Gcc

in C the type **gimple** (a pointer)

See files `$GCCSOURCE/gcc/gimple.{def,h,c}`

gimple.def contains **36** definitions (**14 are for OpenMP !**) like

```
/* 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)
```

Or

```
/* 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 assigns

```

/* 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. */
DEFGSSCODE(GIMPLE_ASSIGN, "gimple_assign", GSS_WITH_MEM_OPS)

```

Gimple operands are *Tree*-s. For *Gimple/SSA*, the *Tree* is often a **SSA_NAME**

Gimple data in C

in \$GCCSOURCE/gcc/**gimple.h**:

```

/* Data structure definitions for GIMPLE tuples. NOTE: word markers
   are for 64 bit hosts. */
struct GTY(()) gimple_statement_base {
  /* [ WORD 1 ] Main identifying code for a tuple. */
  ENUM_BITFIELD(gimple_code) code : 8;
  // etc...
  /* Number of operands in this tuple. */
  unsigned num_ops;
  /* [ WORD 3 ] Basic block holding this statement. */
  struct basic_block_def *bb;
  /* [ WORD 4 ] Lexical block holding this statement. */
  tree block; };

/* Base structure for tuples with operands. */
struct GTY(()) gimple_statement_with_ops_base {
  /* [ WORD 1-4 ] */
  struct gimple_statement_base gsbase;
  /* [ WORD 5-6 ] SSA operand vectors. NOTE: It should be possible to
     amalgamate these vectors with the operand vector OP. However,
     the SSA operand vectors are organized differently and contain
     more information (like immediate use chaining). */
  struct def_optype_d GTY((skip (""))) *def_ops;
  struct use_optype_d GTY((skip (""))) *use_ops;

```

inline accessors to *Gimple*

`gimple.h` also have many **inline functions**, like e.g.

```

/* Return the code for GIMPLE statement G. crash if G is null */
static inline enum gimple_code gimple_code (const_gimple g) {...}
/* Set the UID of statement. data for inside passes */
static inline void gimple_set_uid (gimple g, unsigned uid) {...}
/* Return the UID of statement. */
static inline unsigned gimple_uid (const_gimple g) {...}
/* Return true if GIMPLE statement G has register or memory operands. */
static inline bool gimple_has_ops (const_gimple g) {...}
/* Return the set of DEF operands for statement G. */
static inline struct def_optype_d *gimple_def_ops (const_gimple g) {...}
/* Return operand I for statement GS. */
static inline tree gimple_op (const_gimple gs, unsigned i) {...}
/* If a given GIMPLE_CALL's callee is a FUNCTION_DECL, return it.
   Otherwise return NULL. This function is analogous to get_callee_fndecl in tree.h */
static inline tree gimple_call_fndecl (const_gimple gs) {...}
/* Return the LHS of call statement GS. */
static inline tree gimple_call_lhs (const_gimple gs) {...}

```

There are also functions to **build or modify gimple**

control-flow related representations inside Gcc

- **gimple** are simple instructions
- **gimple_seq** are sequence of `gimple-s`
- **basic_block** are elementary blocks, containing a `gimple_seq` and connected to other basic blocks thru `edge-s`
- **edge-s** connect basic blocks (i.e. are jumps!)
- **loop-s** are for dealing with loops, knowing their basic block **headers** and **latches**
- the struct **control_flow_graph** packs entry and exit blocks and a vector of basic blocks for a function
- the struct **function** packs the `control_flow_graph` and the `gimple_seq` of the function body, etc ...
- `loop-s` are hierachically organized inside the struct **loops** (e.g. the **current_loops** global) for the current function.

NB: **not every representation** is **available** in every pass!

Basic Blocks in Gcc

`coretypes.h` has `typedef struct basic_block_def *basic_block;`

In `$GCCSOURCE/gcc/basic-block.h`

```

/* Basic block information indexed by block number. */
struct GTY((chain_next ("%h.next_bb"), chain_prev ("%h.prev_bb"))) basic_block_def
  /* The edges into and out of the block. */
  VEC(edge,gc) *preds;
  VEC(edge,gc) *succs;  //etc ...
  /* Innermost loop containing the block. */
  struct loop *loop_father;
  /* The dominance and postdominance information node. */
  struct et_node * GTY ((skip (""))) dom[2];
  /* Previous and next blocks in the chain. */
  struct basic_block_def *prev_bb;
  struct basic_block_def *next_bb;
  union basic_block_il_dependent {
    struct gimple_bb_info * GTY ((tag ("0"))) gimple;
    struct rtl_bb_info * GTY ((tag ("1"))) rtl;
  } GTY ((desc ("(%1.flags & BB_RTL) != 0"))) il;
  // etc ....
  /* Various flags. See BB_* below. */
  int flags;
};

```

gimple_bb_info & control_flow_graph

Also in `basic-block.h`

```
struct GTY(()) gimple_bb_info {
  /* Sequence of statements in this block.  */
  gimple_seq seq;
  /* PHI nodes for this block.  */
  gimple_seq phi_nodes;
};

/* A structure to group all the per-function control flow graph data. */
struct GTY(()) control_flow_graph {
  /* Block pointers for the exit and entry of a function.
     These are always the head and tail of the basic block list.  */
  basic_block x_entry_block_ptr;
  basic_block x_exit_block_ptr;
  /* Index by basic block number, get basic block struct info.  */
  VEC(basic_block,gc) *x_basic_block_info;
  /* Number of basic blocks in this flow graph.  */
  int x_n_basic_blocks;
  /* Number of edges in this flow graph.  */
  int x_n_edges;
  // etc ...
};
```

Control Flow Graph and loop-s in Gcc

In \$GCCSOURCE/gcc/cfgloop.h

```

/* Description of the loop exit. */
struct GTY (()) loop_exit {
  /* The exit edge. */
  struct edge_def *e;
  /* Previous and next exit in the list of the exits of the loop. */
  struct loop_exit *prev;      struct loop_exit *next;
  /* Next element in the list of loops from that E exits. */
  struct loop_exit *next_e; };

typedef struct loop *loop_p;
/* Structure to hold information for each natural loop. */
struct GTY ((chain_next ("%h.next"))) loop {
  /* Index into loops array. */
  int num;
  /* Number of loop insns. */
  unsigned ninsns;
  /* Basic block of loop header. */
  struct basic_block_def *header;
  /* Basic block of loop latch. */
  struct basic_block_def *latch;
  // etc ...
  /* True if the loop can be parallel. */
  bool can_be_parallel;
  /* Head of the cyclic list of the exits of the loop. */
  struct loop_exit *exits;
};

```

Caveats on Gcc internal representations

- in principle, **they are not stable** (could change in 4.7 or next)
- in practice, **changing central representations** (like `gimple` or `tree`) is very **difficult** :
 - Gcc gurus (and users?) care about compilation time
 - Gcc people could “fight” for some bits
 - changing them is very costly: \Rightarrow need to patch every pass
 - you need to convince the whole Gcc community to enhance them
 - some Gcc heroes could change them
- **extensions or plugins cannot add extra data fields** (into `tree-s`, `gimple-s`⁵⁷ or `basic_block-s`, ...)
 \Rightarrow use other data (e.g. associative hash tables) to link your data to them

⁵⁷ *Gimple-s* have *uid-s* but they are only for inside passes!

Handling GCC stuff with MELT

Gcc raw stuff is handled by Melt c-types like `:gimple_seq` or `:edge`

- raw stuff can be passed as formal arguments or given as secondary results
- Melt functions
 - **first argument⁵⁸ should be a value**
 - **first result is a value**
- raw stuff have boxed values counterpart
- raw stuff have hash-maps values (to associate a non-nil Melt value to a tree, a gimple etc)
- **primitive** operations can handle stuff or values
- **c-iterators** can iterate inside stuff or values

⁵⁸i.e. the receiver, when sending a message in Melt

Primitives in Melt

Primitive operations have arbitrary (but fixed) signature, and give one result (which could be `:void`).

used e.g. in `Melt` where `body` is some `:basic_block` stuff
(code by Jérémie Salvucci from `xtramelt-c-generator.melt`)

```
(let ( (:gimple_seq instructions (gimple_seq_of_basic_block body)) )
  ;; do something with instructions
)
```

(`gimple_seq_of_basic_block` takes a `:basic_block` stuff & gives a `:gimple_seq` stuff)

Primitives are defined thru `defprimitive` by macro-strings, e.g. in

`$GCCMELTSOURCE/gcc/melt/xtramelt-ana-base.melt`

```
(defprimitive gimple_seq_of_basic_block (:basic_block bb) :gimple_seq
  #{{{ ($BB) ?bb_seq ($BB) :NULL }#})
```

(always test for 0 or null, since `Melt` data is cleared initially)

Likewise, arithmetic on raw `:long` stuff is defined (in `warmelt-first.melt`):

```
(defprimitive +i (:long a b) :long
  :doc #{Integer binary addition of $a and $b.}#
  #{{{ ($A) + ($B) }#})
```

(no boxed arithmetic primitive yet in `Melt`)

c-iterators in Melt

C-iterators describe how to iterate, by generation of `for`-like constructs, with

- **input** arguments - for parameterizing the iteration
- **local** formals - giving locals changing on each iteration

So if `bb` is some **Melt** `:basic_block` stuff, we can iterate on its contained `:gimple-s` using

```
(eachgimple_in_basicblock
  (bb)      ;; input arguments
  (:gimple g)  ;; local formals
  (debuggimple "our g" g) ;; do something with g
)
```

The definition of a **c-iterator**, in a **defciterator**, uses a **state symbol** (like in `code_chunk-s`) and two “before” and “after” macro-strings, expanded in the head and the tail of the generated `C` loop.

Example of defciterator

in xtramel-t-ana-base.melt

```
(defciterator eachgimple_in_basicblock
  (:basic_block bb)          ;start formals
  eachgimpbb                 ;state symbol
  (:gimple g)                ;local formals
  ;; before expansion
  #{ /* start $EACHGIMPBB */
    gimple_stmt_iterator gsi_$EACHGIMPBB;
    if ($BB)
      for (gsi_$eachgimpbb = gsi_start_bb ($BB);
          !gsi_end_p (gsi_$EACHGIMPBB);
          gsi_next (&gsi_$EACHGIMPBB)) {
        $G = gsi_stmt (gsi_$EACHGIMPBB);
      }#
  ;; after expansion
  #{ } /* end $EACHGIMPBB */ }#
)
```

(most iterations in Gcc fit into *c-iterators*; because few are callbacks based)

values taxonomy

- classical almost **Scheme-like** (or **Python-like**) values:
 - 1 the **nil** value `()` - it is the only **false** value (unlike **Scheme**)
 - 2 **boxed integers**, e.g. `'2`; or **boxed strings**, e.g. `'"ab"`
 - 3 **symbols** (objects of `class_symbol`), e.g. `'x`
 - 4 **closures**, i.e. functions [only **values** can be **closed** by `lambda` or `defun`]
 - (also [internal to closures] **routines** containing constants)
 - e.g. `(lambda (f :tree t) (f y t))` has closed `y`
 - 5 **pairs** (rarely used alone)
- **boxed stuff**, e.g. **boxed gimples** or **boxed basic blocks**, etc ...
- **lists** of pairs (unlike **Scheme**, they know their first and last pairs)
- **tuples** \equiv fixed array of immutable components
- **associative homogenous hash-maps**, keyed by either
 - non-nil **Gcc** raw stuff like `:tree-s`, `:gimple-s` ... (**all keys of same type**), or
 - **Melt** objects

with each such key associated to a non-nil **Melt** value
- **objects** - (their discriminant is their class)

lattice of discriminants

- Each value has its immutable discriminant.
- Every discriminant is an object of **class_discriminant** (or a subclass)
- Classes are objects of **class_class**
Their fields are reified as instances of **class_field**
- The nil value (represented by the **NULL** pointer in generated C code) has **discr_null_receiever** as its discriminant.
- each discriminant has a parent discriminant (the super-class for classes)
- the top-most discriminant is **discr_any_receiever**
(usable for catch-all methods)
- discriminants are used by garbage collectors (both **Melt** and **Ggc!**)
- discriminants are used for **Melt message sending**:
 - each message send has a selector σ & a receiver ρ , i.e. $(\sigma \rho \dots)$
 - selectors are objects of **class_selector** defined with **defselector**
 - receivers can be any **Melt** value (even nil)
 - discriminants have a **:disc_methodict** field - an object-map associating selectors to methods (closures); and their **:disc_super**

C-type example: `ctype_tree`

Our **c-types** are described by **Melt** [predefined] objects, e.g.

```
;; the C type for gcc trees
(definstance ctype_tree class_ctype_gty
  :doc #{"The $CTYPE_TREE is the c-type
of raw GCC tree stuff. See also
$DISCR_TREE. Keyword is :tree.}#
  :predef CTYPE_TREE
  :named_name ' "CTYPE_TREE"
  :ctype_keyword ' :tree
  :ctype_cname ' "tree"
  :ctype_parchar ' "MELTBPAR_TREE"
  :ctype_parstring ' "MELTBPARSTR_TREE"
  :ctype_argfield ' "meltbp_tree"
  :ctype_resfield ' "meltbp_treeptr"
  :ctype_marker ' "gt_ggc_mx_tree_node"
;; GTY ctype
  :ctypg_boxedmagic ' "MELTOBMAG_TREE"
  :ctypg_mapmagic ' "MELTOBMAG_MAPTREES"
  :ctypg_boxedstruct ' "melttree_st"
  :ctypg_boxedunimemb ' "u_tree"
  :ctypg_entrystruct ' "entrytreemelt_st"
  :ctypg_mapstruct ' "meltmaptrees_st"
  :ctypg_boxdiscr ' discr_tree
  :ctypg_mapdiscr ' discr_map_trees
  :ctypg_mapunimemb ' "u_maptrees"
  :ctypg_boxfun ' "meltgc_new_tree"
  :ctypg_unboxfun ' "melt_tree_content"
  :ctypg_updateboxfun ' "meltgc_tree_update"
  :ctypg_newmapfun ' "meltgc_new_maptree"
  :ctypg_mapgetfun ' "melt_get_maptrees"
  :ctypg_mapputfun ' "melt_put_maptrees"
  :ctypg_mapremovefun ' "melt_remove_maptree"
  :ctypg_mapcountfun ' "melt_count_maptree"
  :ctypg_mapsizefun ' "melt_size_maptree"
  :ctypg_mapnattfun ' "melt_nthattr_maptree"
  :ctypg_mapnvalfun ' "melt_nthval_maptree"
)
(install_ctype_descr
  ctype_tree "GCC tree pointer")
```

The strings are the names of **generated run-time support** routines (or types, enum-s, fields ...)

in `$GCCMELTSOURCE/gcc/melt/generated/meltrunsup*.[ch]` 

Melt objects and classes

Melt objects have a single class (class hierarchy rooted at `class_root`)

Example of class definition in `warmelt-debug.melt`:

```
;; class for debug information (used for debug_msg & dbgout* stuff)
(defclass class_debug_information
  :super class_root
  :fields (dbg_out dbg_occmmap dbg_maxdepth)
  :doc #{The $CLASS_DEBUG_INFORMATION is for debug information output,
e.g. $DEBUG_MSG macro. The produced output or buffer is $DBGI_OUT,
the occurrence map is $DBGI_OCCMAP, used to avoid outputting twice the
same object. The boxed maximal depth is $DBGI_MAXDEPTH.}#
)
```

We use it in code like

```
(let ( (dbg (instance class_debug_information
                    :dbg_out out
                    :dbg_occmmap occmap
                    :dbg_maxdepth boxedmaxdepth))
      (:long framdepth (the_framdepth))
    )
  (add2out_strconst out "!!!!***#####")
  ;; etc
)
```

Melt fields and objects

Melt field names are globally unique

- ⇒ (**get_field** :**dbgi_out dbgi**) is translated to **safe code**:
 - 1 testing that indeed **dbgi** is instance of `class_debug_information`, then
 - 2 extracting its `dbgi_out` field.
- (⇒ never use **unsafe_get_field**, or your code could crash)
- Likewise, **put_fields** is safe
- (⇒ never use **unsafe_put_fields**)
- **convention**: all proper field names of a class share a common prefix
- no visibility restriction on fields
(except module-wise, on “private” classes not passed to **export_class**)

Classes are conventionally named `class_*`

Methods are dynamically installable on any discriminant, using
(**install_method discriminant selector method**)

About pattern matching

You already used it, e.g.

- in regular expressions for substitution with `sed`
- in XSLT or Prolog (or expert systems rules with variables, or formal symbolic computing)
- in Ocaml, Haskell, Scala

A tiny calculator in Ocaml:

```
(*discriminated unions [sum type], with cartesian products*)
type expr_t = Num of int
             | Add of expr_t * expr_t
             | Mul of expr_t * expr_t ;;

(*recursively compute an expression thru pattern matching*)
let rec compute e = match e with
  Num x → x
  | Add (a,b) → a + b
  (*disjunctive pattern with joker _ and constant sub-patterns::*)
  | Mul (_,Num 0) | Mul (Num 0,_) → 0
  | Mul (a,b) → a * b ;;

(*inferred type: compute : expr_t → int *)
```

Then `compute (Add (Num 1, Mul (Num 2, Num 3))) ⇒ 7`

Using pattern matching in your Melt code

code by Pierre Vittet

```
(defun detect_cond_with_null (grdata :gimple g)
  (match g ;; the matched thing
    ( ?(gimple_cond_notequal ?lhs
                                     ?(tree_integer_cst 0))
      (make_tree discr_tree lhs))
    ( ?(gimple_cond_equal ?lhs
                           ?(tree_integer_cst 0))
      (make_tree discr_tree lhs))
    ( ?_
      (make_tree discr_tree (null_tree))))))
```

- lexical shortcut: $?π \equiv (\text{question } π)$, much like $'ε \equiv (\text{quote } ε)$
- **patterns are major syntactic constructs** (like expressions or bindings are; parsed with *pattern macros* or “patmacros”), first in matching clauses
- $?_$ is the **joker pattern**, and $?lhs$ is a **pattern variable** (local to its clause)
- most **patterns are nested**, made with **matchers**, e.g. `gimple_cond_notequal` or `tree_integer_cst`

What `match` does?

- syntax is (`match` ϵ $\kappa_1 \dots \kappa_n$) with ϵ an expression giving μ and κ_j are matching clauses considered in sequence
- the `match` expression returns a result (some thing, perhaps `:void`)
- it is made of matching clauses (π_i $\epsilon_{i,1} \dots \epsilon_{i,n_i}$ η_i), each starting with a pattern⁵⁹ π_i followed by sub-expressions $\epsilon_{i,j}$ ending with η_i
- it matches (or filters) some thing μ
- **pattern variables** are **local** to their clause, and **initially cleared**
- when pattern π_i matches μ the expressions $\epsilon_{i,j}$ of clause i are executed in sequence, with the pattern variables inside π_i locally bound. The last sub-expression η_i of the match clause gives the result of the entire `match` (and all η_i should have a common c-type, or else `:void`)
- if no clause matches -this is bad taste, usually last clause has the `?_` joker pattern-, the result is cleared
- a pattern π_i can **match** the thing μ or **fail**

⁵⁹expressions, e.g. constant literals, are degenerate patterns!

pattern matching rules

rules for matching of pattern π against thing μ :

- the **joker pattern** `?_` **always match**
- an **expression** (e.g. a constant) \in (giving μ') matches μ **iff** $(\mu' == \mu)$ in C parlance
- a **pattern variable** like `?x` matches if
 - x was unbound; then it is **bound** (locally to the clause) to μ
 - or else x was already bound to some μ' and $(\mu' == \mu)$ [**non-linear patterns**]
 - otherwise (x was bound to a different thing), the pattern variable `?x` match fails
- a **matcher pattern** `? (m $\eta_1 \dots \eta_n$ $\pi'_1 \dots \pi'_p$)` with $n \geq 0$ input argument sub-expressions η_i and $p \geq 0$ sub-patterns π'_j
 - the matcher m does a **test** using results ρ_i of η_i ;
 - if the test succeeds, data are extracted in the **fill** step and each should match its π'_j
 - otherwise (the test fails, so) the match fails
- an **instance pattern** `? (instance κ : ϕ_1 π'_1 ... : ϕ_n π'_n)` matches iff μ is an object of class κ (or a sub-class) with each field ϕ_i matching its sub-pattern π'_i

control patterns

We have controlling patterns

- **conjunctive pattern** ? (**and** $\pi_1 \dots \pi_n$) matches μ iff π_1 matches μ and then π_2 matches $\mu \dots$
- **disjunctive pattern** ? (**or** $\pi_1 \dots \pi_n$) matches μ iff π_1 matches μ or else π_2 matches $\mu \dots$

Pattern variables are initially cleared, so `(match 1 (? (or ?x ?y) y))` gives 0 (as a **:long** stuff)

(other control patterns would be nice, e.g. backtracking patterns)

matchers

Two kinds of matchers:

- 1 **c-matchers** giving the *test* and the *fill* code thru expanded macro-strings

```
(defcmatcher gimple_cond_equal
  (:gimple gc) ;; matched thing  $\mu$ 
  (:tree lhs :tree rhs) ;; subpatterns putput
  gce ;; state symbol
  ;; test expansion:
  #({($GC &&
      gimple_code ($GC) == GIMPLE_COND &&
      gimple_cond_code ($GC) == EQ_EXPR)
    }#
  ;; fill expansion:
  #({ $LHS = gimple_cond_lhs ($GC);
      $RHS = gimple_cond_rhs ($GC);
    }#)
```

- 2 **fun-matchers** give test and fill steps thru a **Melt** function returning secondary results

known MELT weaknesses [corrections are worked upon]

- 1 pattern matching translation is weak⁶⁰
(a new pattern translator is nearly completed)
- 2 **Melt** passes can be slow
 - better and faster **Melt** application
 - memoization in message sends
 - optimization of **Melt** G-C invocations and **Ggc** invocations
- 3 variadic functions (e.g. debug printing)
- 4 dump support
- 5 debug support
 - plugins want their `gcc` with `-enable-check=all`, not `-enable-check=release`
 - **Melt** `debug_msg` **wants** `-fmelt-debug` **and** `-enable-check=...`
 - a probing process?

⁶⁰Sometimes crashing the **Melt** translator ☺

Exercice

Code a **Melt** pass counting calls to a given function with null argument ☺