GCC plugins and MELT extensions (e.g. Talpo)

Basile STARYNKEVITCH and Pierre VITTET
basile@starynkevitch.net (or basile.starynkevitch@cea.fr)
piervit@pvittet.com

energie atomique • energies alternatives

August 24th 2011 – *Gnu Hackers Meeting 2011 (Paris,IRILL France)*

Table of Contents

Contents

opinions are mine only

Opinions expressed here are only mine!

- not of my employer (CEA, LIST) or school (Polytech Tours)
- not of the Gcc community
- not of funding agencies (e.g. DGCIS or Google GSOC)[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

(slides presented before at Archi11 (Basile), and RMLL11 (Pierre))

[1] Work on Melt have been possible thru the GlobalGCC ITEA and OpenGPU FUI collaborative research projects, with funding from DGCIS, and GSOC (for Talpo)

Questions to the audience

1. What is your usual Gcc version? your latest Gcc?
2. Who contributed to Gcc ☺?
3. 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 (lispy 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)?
4. 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: ECGS 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`

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 Commitee

`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 $\rightarrow$ GPLv3) and exceptions
- approve (and advocate) major evolutions: plugins feature [2], new languages, new targets
- nominate reviewers

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

[2] The 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** ($\approx$ 400) of full-time developers
  5. **no single leader** or benevolent dictator

     $\Rightarrow$ 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 signficant 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. Java[4] 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)

---

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

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

[5] For 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-licensed[7] library `http://llvm.org` for [machine] code generation (e.g. Just In Time)

Melt is implemented by generating C code

meta-knowledge[8] **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.

---

[6] You 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.

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

[8] J.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 compilers[9], 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

[9] Often quite buggy in practice

Why GCC matters?

Gcc matters to you and to me because:

- you are interested in computer architecture or performance or diagnostics[10],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)

[10] Execution 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 linking[11].

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

LTO can be costly.

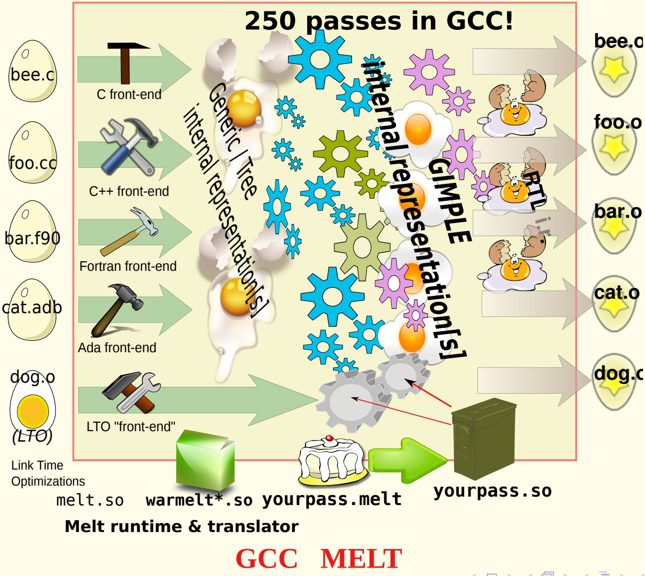
LTO is implemented by encoding GCC internal representations (Gimple, ...) in object files.

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

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

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

Gcc and Melt big picture



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 developped 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 lispy **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-] plugin[12] for Gcc

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

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

   `.../cc1plus -quiet -v -v -D_GNU_SOURCE sayit.cc -quiet -dumpbase sayit.cc -mtune=`

2. assemble **as** `-64 -o /tmp/ccMj0zRK.o /tmp/cccEErWK.s`
3. C compile

   `.../cc1 -quiet -quiet -v -v hello1.c -quiet -dumpbase hello1.c -mtune=generic -ma`

4. assemble **as** `-64 -o /tmp/ccocvgSK.o /tmp/cccEErWK.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=x8`

   3. assemble **as** `-64 -o /tmp/ccBc8K58.ltrans0.ltrans.o /tmp/ccghYRNa.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

Motivations for MELT

Gcc extensions address a limited number of users[13], so their development should be facilitated (cost-effectiveness issues)

- extensions should be [meta-] plugins, not Gcc variants [branches, forks] [14] 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

[13] Any development useful to all Gcc users should better go inside Gcc core!
[14] Most Gnu/Linux distributions don't even package Gcc branches or forks.

Embedding a scripting language is impossible

Many scripting or high-level languages [15] 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 **GTY**-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 ⇒ 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 ...*/ }
```

[15] Pedantically, languages' *implementations* can be embedded!

Melt, a **D**omain **S**pecific **L**anguage 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 it[16] into a Melt binary `.so` module and `dlopen`-s that module.

[16] By 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**[17], 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 Ggc), 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)

[17] Because 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	$O(\lambda)$ $\lambda =$ size of young live objects	$O(\sigma)$ $\sigma =$ 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 ();` [18]
  5. all local pointers (local variables) are in Melt frames
  6. needs a write barrier (to handle old → young pointers)
  7. requires tedious C coding: call frames, barriers, **normalizing nested expressions** (`z = f(g(x),y)` → temporary τ = `g(x); z=f(`τ`, y);`)
  8. **well suited for *generated* C code**

[18] So 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` ≡ `deFUn`
- **Lisp-like syntax**: **(** *operator operands ...* **)** so **parenthesis are always significant** in Melt `(f)` ≢ `f`, but in **C** `f()` ≢ `f` ≡ `(f)`
- `defun` is a "macro" for ***def***ining ***fun***ctions 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 closp_,
  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 mcfr_nbvar;
#if ENABLE_CHECKING
    const char *mcfr_flocs;
#endif
    struct meltclosure_st *mcfr_clos;
    struct excepth_melt_st *mcfr_exh;
    struct callframe_melt_st *mcfr_prev;
    void *mcfr_varptr[5];
    tree loc_TREE__o0;
  }  *framptr_ = 0, meltfram__;
  memset (&meltfram__, 0, sizeof (meltfram__));
  meltfram__.mcfr_nbvar = 5;
  meltfram__.mcfr_clos = closp_;
  meltfram__.mcfr_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] != MELTBPAR_TREE)
    goto lab_endgetargs;
 /*_?*/ meltfram__.loc_TREE__o0 = xargtab_[0].meltbp_tre
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 */ meltfrout
        ( /*_?*/ 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*/ meltfpt
        (melt_ptr_t) ( /*_.X__V2*/ meltfptr[1]);
      ((meltmultiple_ptr_t) ( /*_.TUPLREC___V5*/ meltfpt
        (melt_ptr_t) ( /*_.MAKE_TREE__V3*/ meltfptr[2]);
      meltgc_touch ( /*_.TUPLREC___V5*/ meltfptr[4]);
     /*_.RETVAL___V1*/ meltfptr[0] = /*_.TUPLE___V4*/ me
```

"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** [19] 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"}#` ≡ `("A\""  b  "C\"\\n")` [a 3-elements list, the 2nd is symbol **b**, others are strings]

[19] 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

```
cc1: note: MELT generated new file
      /tmp/GCCMeltTmpdir-1c5b3a95/helloworld.c
cc1: 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
cc1: 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**[20] 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

[20] The 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:

```
 {
#ifndef 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 *Gimple*s were *Tree*s

Why a Lisp-y syntax for Melt

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

- Melt is bootstrapped
  - now Melt translator[21] is written in Melt
    `$GCCMELTSOURCE/gcc/`**`melt/warmelt-*.melt`**
    ⇒ the ***C* translation** of Melt translator is **in its *source* repository**[22]
    `$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[23]
- 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** ⇒ 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`**

---

[21] Melt started as a Lisp program

[22] This is unlike other *C* generators inside Gcc

[23] 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)
- ⇒ the Melt translator is a good test case for Melt;
  it exercices 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-**rec**ursive 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**[24]
- `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

[24] a huge generalization of `switch` in *C*

non Lisp-y features of Melt

Many linguistic devices to **decribe 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 `'`$\alpha \equiv$ `(quote` α`)`
  - `(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` $\not\equiv$ `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 ≈ **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 uniq
   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 nod
   accepted by is_gimple_lvalue.
   RHS1 is the first operand on the RHS of the assignment.  It must always
   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
   node accepted by is_gimple_val.  This argument exists only if SUBCODE i
   of class GIMPLE_BINARY_RHS.  */
DEFGSCODE(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
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: ⇒ 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[25] or `basic_block`-s, ...)
  ⇒ use other data (e.g. associative hash tables) to link your data to them

[25] *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 argument[26] 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

[26] i.e. the reciever, 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

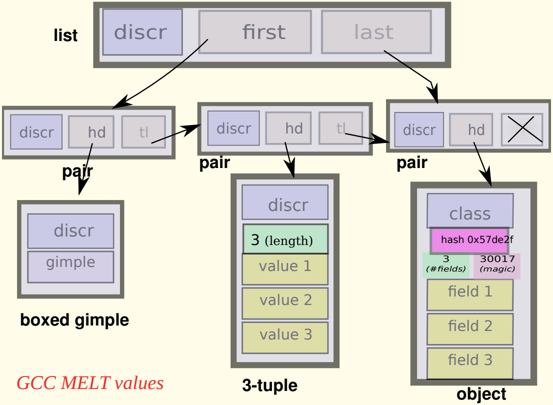
in xtramelt-ana-base.melt

```
(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 in Melt

Each value starts with an immutable [often predefined] **discriminant** (for a Melt object value, the discriminant is its class).



Melt copying generational garbage collector manages [only] values (it copies live Melt values into Ggc heap).

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** ≡ 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 discrimnant.
- 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_reciever` as its discriminant.
- each discriminant has a parent discriminant (the super-class for classes)
- the top-most discriminant is `discr_any_reciever`
  (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 reciever ρ, i.e. **(**σ ρ **...)**
  - selectors are objects of `class_selector` defined with `defselector`
  - recievers 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_updat
  :ctypg_newmapfun     '"meltgc_new_maptre
  :ctypg_mapgetfun     '"melt_get_maptrees
  :ctypg_mapputfun     '"melt_put_maptrees
  :ctypg_mapremovefun  '"melt_remove_maptr
  :ctypg_mapcountfun   '"melt_count_maptre
  :ctypg_mapsizefun    '"melt_size_maptree
  :ctypg_mapnattfun    '"melt_nthattr_mapt
  :ctypg_mapnvalfun    '"melt_nthval_maptr
)

(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 (dbgi_out dbgi_occmap dbgi_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 ( (dbgi (instance class_debug_information
                       :dbgi_out out
                       :dbgi_occmap occmap
                       :dbgi_maxdepth boxedmaxdepth))
       (:long framdepth (the_framedepth))
     )
  (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: `?`$\pi \equiv$ `(question` π`)`, much like `'`$\epsilon \equiv$ `(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_const`

What **match** does?

- syntax is **(match** $\epsilon \;\; \kappa_1 \ldots \kappa_n$ **)** 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 **(** $\pi_i \;\; \epsilon_{i,1} \ldots \epsilon_{i,n_i} \;\; \eta_i$ **)**, each starting with a pattern[27] 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**

[27] expressions, e.g. constant litterals, 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) ϵ (giving μ') matches μ **iff** `(`μ' **`==`** μ`)` 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 `(`μ' **`==`** μ`)` ***[non-linear patterns]***
  - otherwise (`x` was bound to a different thing), the pattern variable `?x` match fails
- a **matcher pattern** `?(`$m\ \ \eta_1 \ldots \eta_n\ \ \pi'_1 \ldots \pi'_p$`)` 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 `:`$\phi_1\ \ \pi'_1$ ... `:`$\phi_n\ \ \pi'_n$`)` 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

- **conjonctive pattern** `?(and` $\pi_1 \ldots \pi_n$`)` matches μ iff π_1 matches μ and then π_2 matches μ ...
- **disjonctive pattern** `?(or` $\pi_1 \ldots \pi_n$`)` matches μ iff π_1 matches μ or else π_2 matches μ ...

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 μ
  (: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 weak[28]
   (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?

[28] Sometimes crashing the Melt translator ☹

Contents

Foreword about Talpo

Talpo means mole in Esperanto!

The idea of the name is that it digs blindly into GCC (without knowing much where it goes :)) and calls still found useful informations you need.

In fact: a customizable **GCC** extension (written in **MELT**) to run simple analysis in your C/C++ programs.
Use case:

- You want to check that a call to **malloc** function is followed by a call to **free** in the same function.
- You want to check that a call to **fopen** is immediately followed by a test on his returned pointer.
- Checking that there is (or not) code after an ***execX**** (execl, execlp, execle, execv, execvp, execvpe) (to check for error for example).

Talpo

Talpo[29] started with the idea that a static analysis tool can use the powerful functionalities of **GCC** and must be customized for a project: A **Talpo** test can be easily parameterized by people ignoring (much of) **GCC** and **MELT**.
With Talpo you can check this:

- for each call to a foo function, result of the call is tested to be (not) NULL/negative/zero.
- each call to a foo function is immediately followed by a call to a bar function.
- each call to a foo function is followed by a call to a bar function in the same function body.

This is quite simple and limited but can already be useful in many cases!

[29] https://gitorious.org/talpo

Easy to use

There are different ways to pass argument to **Talpo**:

Using pragma in code file

```
#pragma MELT talpo testNull(fopen)
#pragma MELT talpo test_followed\by(chroot, chdir)
```

Using direct argument

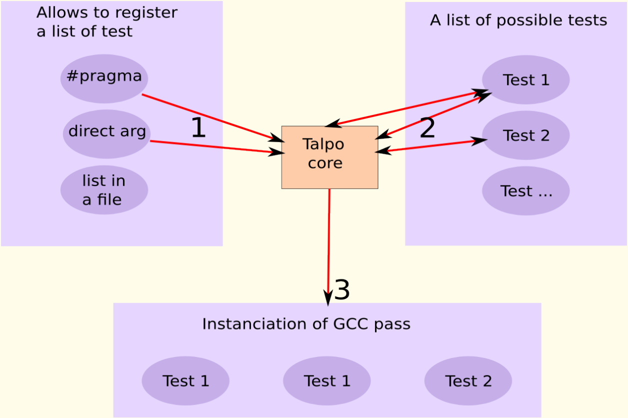
```
gcc ...
-fplugin-arg-melt-option=talpo-arg='(testNull "fopen")\
(test_followed_by "chroot" "chown")'
...
```

Using a file to list argument

```
-fplugin-arg-melt-option=talpo-arg-file='myfile'
```

Modularity

- You can easily implement a new way to read user input.
- You can easily implement a new test.



Try it!

C code

```
#include <stdio.h>
int main(void){
  FILE * test;
  test=fopen ("test","a");
  return 0;
}
```

Result

```
gcc -Wall -fplugin=melt -fplugin-arg-melt-mode=talpo \
-fplugin-arg-melt-module-path='$(talpoPath)' \
-fplugin-arg-melt-source-path=. \
-fplugin-arg-melt-extra=@$(talpoPath)/talpo \
-fplugin-arg-melt-option=talpo-arg='(testNull "fopen")' \
-O2 test.c -o test.o
```

```
test_simple_check_cfile.c:5:10: warning: Melt Warning[#221]: Function 'fopen' \
not followed by a test on his returned pointer.
```

Using post-dominating basicblock

C code

```
int main(void){
  int i=0;
  FILE * test;
  FILE * test2;
  if( i ==1)
    test=fopen("toto","a");
  else
    test2=fopen("tata", "a");
  if (test == NULL)
    return 1;
  return 0;
}
```

Result

```
test_simple_check_cfile.c:8:10: warning: Melt Warning[#216]: Function 'fopen' \
not followed by a test on his returned pointer.
```

Using a struct

C code

```
typedef struct _myStr
{
  FILE * ptrfile;
}myStr;

int
use_struct_no_warn (void)
{
  myStr * testStr = (myStr *) malloc(sizeof(myStr));
  testStr->ptrfile=fopen("toto", "a");
  if(!testStr->ptrfile){
    return 1;
  }
  return 0;
}
```

Result

(No warnings returned!)

On a different variable

C code

```
int
not_same_var_warn_once()
{
  char * curDir = (char *) malloc (sizeof(char) *3);
  char * notCurDir = (char *) malloc (sizeof(char) *3);
  curDir = ".";
  chroot(curDir);
  chdir(notCurDir);
  return 0;
}
```

Result

```
gcc -Wall -fplugin=melt -fplugin-arg-melt-mode=talpo \
-fplugin-arg-melt-module-path='$(talpoPath)' \
-fplugin-arg-melt-source-path=. \
-fplugin-arg-melt-extra=@$(talpoPath)/talpo \
-fplugin-arg-melt-option=talpo-arg='(testFollowedBy␣"chroot"␣1␣"chdir"␣1)' \
-O2 test.c -o test.o
```

```
test.c:8:4: attention : Melt Warning[#254]: Call to 'chroot' is not followed \
by a call to 'chdir'. [enabled by default]
```

What does not work

For some code samples, it still returns false positives.

Function pointer are not detected!

C code

```
int main()
{
  FILE * (*myPtr)(char *, char *);
  myPtr = fopen;

  FILE * res = myPtr("path","a");
  return 0;
}
```

Result

(No warnings returned!)

Function pointer are not detected!

C++ code

```
  The following case has been asked by Jonathan Wakely on the GCC mailing list:

struct Guard {
    Guard(void* p) : p(p) { if (!p) throw std::bad_alloc(); }
    ~Guard() { grub_free(p); }
    void* p;
};

void func(grub_size_t n)
{
    Guard g(grub_malloc(n));
    // do something with g.p
}
```

Result

(No warnings returned with -O0!)

Contents

Conclusion

- Free software is about adapting software to your needs: Plugins are a great way to customize Gcc.
- Melt[30] simplifies writing Gcc extensions (and is more fun than coding in C).
- We invite you to test Talpo, code in MELT, extend them!
- Manu projects could provide their specific MELT extensions to help their C coders: Hurd, MPC, MPFR, GTK, Qt...

[30] MELT also has a mailing list : `gcc-melt@googlegroup.com`