MELT, a Domain Specific Language to extend the *GCC* compiler

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

energie atomique • energies alternatives

December 9th 2011 - Grenoble - INRIA/LIG/MOAIS seminar

Table of Contents

Contents

disclaimer: 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

(many slides copied from previous talks)

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

Why extend a compiler?

Extending a compiler is worthwhile:

- to **add** some **specific behavior** to the compiler
  notably, behavior particular to specific needs, which won't be added inside the compiler
- while **taking advantage of the** existing **compiler's infrastructure**
  internal representations, framework, optimization passes...

Extensible compilers:

1. **LLVM/Clang**; a young C++ *library* (BSD license) providing a common internal representation and code generators; evolved into a full C and C++ compiler `clang`; see `llvm.org` [v3.0 in december 2011]
   The BSD license don't require a fully free development community; Apple is rumored to have its specific LLVM
2. **GCC** (the **Gnu Compiler Collection**) `gcc.gnu.org`: a set of legacy compilers (GPLv3 license) for many languages and systems. [v4.6.2: october 2011]
   organized as a bunch of self-sufficient programs; the GPL license entails a living community.

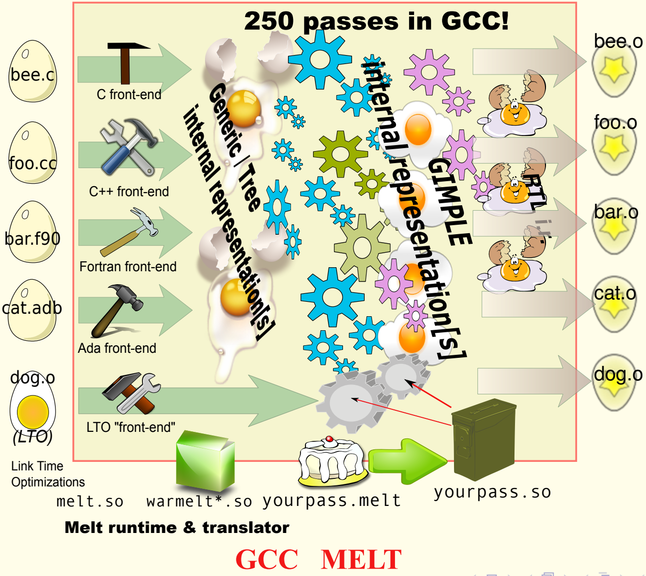
NB: **nobody knows well both** GCC & LLVM compilers

GCC (Gnu Compiler Collection) `gcc.gnu.org`

- perhaps **the most used compiler** : your phone, camera, dish washer, printer, car, house, train, airplane, web server, data center, Internet have Gcc compiled code
- [cross-] compiles **many languages** (C, C++, Ada, Fortran, Go, Objective C, Java, ...) **on many systems** (GNU/Linux, Hurd, Windows, AIX, ...) for **dozens of target processors** (x86, ARM, Sparc, PowerPC, MIPS, C6, SH, VAX, MMIX, ...)
- **free** software (GPLv3+ licensed, FSF copyrighted)
- **huge** (**5** or 8? **MLOC**), **legacy** (started in **1985**) software
- still **alive** and **growing** (+6% in 2 years)
- **big** contributing **community** ($\approx$ **400** "maintainers", mostly full-time professionals)
- **peer-reviewed** development process, but **no main architect** $\Rightarrow$ (IMHO) "sloppy" software architecture, not fully modular yet
- **various coding styles** (mostly *C* & *C++* code, with some **generated *C* code**)
- **industrial-quality compiler** with **powerful optimizations** and **diagnostics** (lots of tuning parameters and options...)

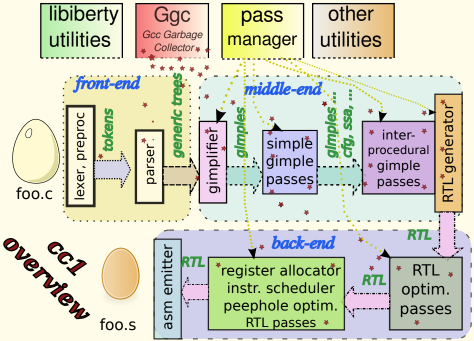
Current version (october 2011) is `gcc-4.6.2`

Gcc & Melt



250 passes in GCC!
bee.c
foo.cc
bar.f90
cat.adb
dog.o
(LTO)
Link Time Optimizations
C front-end
C++ front-end
Fortran front-end
Ada front-end
LTO "front-end"
Generic / Tree
internal representation[s]
internal representation[s]
GIMPLE
RTL
bee.o
foo.o
bar.o
cat.o
dog.o
melt.so
warmelt*.so
yourpass.melt
yourpass.so
Melt runtime & translator
GCC MELT

cc1 organization



Gcc is really `cc1`

- **3 layers :** front-ends → a **common middle-end** → back-ends
- accepting **plugins**
- utilities & (meta-programming) *C* **code generators**
- **internal representations** (Generic/Tree, Gimple[/SSA], CFG ...)
- **pass manager**
- Ggc (= Gcc garbage collection)

Ggc (= Gcc garbage collection)

- compilers handle **complex circular** data-structures
  ⇒ they **need** a **G**arbage **C**ollector
- Ggc is a **simple mark** & sweep **precise garbage collector**
- explicitly invoked **between** passes (by pass manager)
- Ggc **don't handle local** pointers (while other G-Cs often do)
- not run inside passes (even with memory pressure by lots of allocation)
- started as a quick hack to manage long-living Gcc **typed** data (common to several passes); most Gcc representations are handled by Ggc.
- using **GTY annotations** on [≈ 1800] **data structures** & **global variables** :

```
/* Mapping from indices to trees.  */ // from lto-streamer.h
struct GTY(()) lto_tree_ref_table {
  /* Array of referenced trees . */
  tree * GTY((length ("%h.size"))) trees;
  /* Size of array. */
  unsigned int size; };
```

- **gengtype** code generator produces marking routines from **GTY** annotations

plugins and extensibility

- infrastructure for plugins started in **gcc-4.5** (april 2010)
- `cc1` can `dlopen` user plugins[2]
- plugin **hooks** provided:
  1. a plugin can **add** its own **new passes** (or remove some passes)
  2. a plugin can handle **events** (e.g. Ggc start, pass start, type declaration)
  3. a plugin can accept its own **#pragma**-s or **__attribute__** etc...
  4. . . .
- plugin writers need to **understand Gcc internals**
- plugin may provide **customization** and application- or **project- specific** features:
  1. specific warnings (e.g. for untested `fopen` ...)
  2. specific optimizations (e.g. `fprintf(stdout, ...)` → `printf(...)`)
  3. code refactoring, navigation help, metrics
  4. etc etc . . .
- coding plugins in ***C*** may be **not cost-effective**
  higher-level languages are welcome!

[2] Gcc plugins should be free software, GPLv3 compatible

extending GCC with an existing scripting language

A **nearly impossible task**, because of **impedance mismatch**:

- rapid evolution of Gcc
- using a a scripting language like Ocaml, Python[3] or Javascript[4] is difficult, unless focusing on a tiny part of Gcc
- **mixing several unrelated G-Cs** (Ggc and the language one) is **error-prone**
- the Gcc internal API is ill-defined, and has non "functional" sides:
  1. extensive use of *C* macros
  2. ad-hoc iterative constructs
  3. lots of low-level data structures (possible performance cost to access them)
- the Gcc API is huge, and not well defined (a bunch of header files)
- needed **glue code** is big and would change often
- Gcc extensions need **pattern-matching** (on existing Gcc internal representations like *Gimple* or *Tree*-s) and high-level programming (functional/applicative, object-orientation, reflection).

[3] See Dave Malcom's Python plugin

[4] See TreeHydra in Mozilla

Contents

Why MELT?

- embedding an existing DSL [implementation] is inpractical.
- re-implementing a dynamic language (e.g. Python, Lua, or Scheme-like) don't fit well into Gcc practice
- designing a statically typed language [with type inference] would require type formalization of Gcc (intractable).
- Melt[5] is an ad-hoc Lisp-like **domain specific language translated to *C*** code (suitable with Gcc), to develop Gcc extensions
- Melt can **handle existing native** Gcc **stuff** (without boxing) **and** [boxed] Melt **values**
- Melt provides **linguistic devices** describing how ***C* is generated**
- Melt has **high-level programming traits** for functional/applicative, object oriented, reflective programming styles
- Melt has extensible **pattern-matching** compatible with Gcc internal representations
- Melt [Ggc compatible] **runtime** and implementation was **incrementally co-designed** with the **language** (bootstrapped translator)

[5] originally for "Middle End Lisp Translator"

MELT implementation : translator

Melt translator (Melt → *C*)

- implemented in Melt (so exercises well most of Melt)
  (initially, a sub-set was translated by a Lisp program)
- `svn` source code repository contains both Melt source `melt/warmelt*.melt` [43 kloc] (of the translator) and its *C* translation `melt/generated/warmelt*.c` [1440 kloc]
- translation (Melt → *C*) is quick: the **bottleneck is** the **compilation of the generated *C*** code
- can translate in-memory Melt expressions (inside Melt heap) -or a `*.melt` file- to *C*
- co-designed with Melt runtime: generated *C* code respects runtime requirements

MELT implementation : runtime and utilities

Melt runtime [21 kloc of *C*]

- Melt **copying** garbage collector for Melt values
  copy into Ggc heap - partly Melt generated
- runs `make` to compile generated *C* into `*.so`
- `dlopen`-s Melt modules
- provides Gcc plugin hooks
- boxing [mostly Melt generated] of stuff into Melt values

Melt utilities

- “standard” library (in Melt)
- glue (in Melt), e.g. for pattern matching Gcc trees or gimples
- small Gcc passes in Melt, e.g. pass checking Melt runtime
- more to come (OpenCL generation)

MELT values and GCC stuff

Melt deals with two kinds of **things**:

1. Melt first-class (dynamically typed) **values**
   objects, tuples, lists, closures, boxed strings, boxed gimples, boxed trees, homogenous hash-tables…
2. existing Gcc **stuff** (statically and explicitly typed)
   raw `long`-s, `tree`-s, `gimple`-s as already known by Gcc …

Essential distinction (mandated by lack of polymorphism of Ggc):

$$\textit{Things} = \textit{Values} \cup \textit{Stuff}$$

Melt code explicitly annotates stuff with **c-types** like **`:long`**, **`:tree`** …(and **`:value`** for values, when needed).

handling Melt values is preferred (and easier) in Melt code.

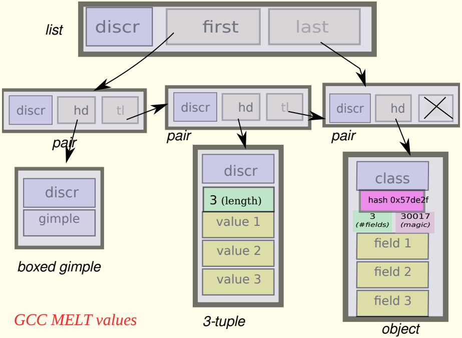
Melt argument passing is typed

Melt copying garbage collection for values

- copying Melt GC well suited for **fast allocation**[6] and many **temporary** (quickly dying) values
- live young values copied into Ggc heap (but needs write barrier)
- Melt GC requires **normalization** $z := \phi(\psi(x), y) \rightarrow \tau := \psi(x); z := \phi(\tau, y)$
- Melt GC handles **locals** and may trigger Ggc at any time
- well suited for **generated** *C* code
  hand-written code for Melt value is cumbersome
- old generation of values is the Ggc heap $\rightarrow$ built-in compatibility of Melt GC with Ggc
- Melt call frames are known to both Melt GC & Ggc
  call frames are singly-linked `struct`-ures.

[6] Melt values are allocated in a birth region by a pointer increment; when the birth region is full, live values are copied out, into Ggc heap, then the birth region is de-allocated.

Melt value taxonomy



- values boxing some stuff
- objects (single-inheritance; classes are also objects)
- tuples, lists and pairs
- closures and routines
- homogenous hash-tables (e.g. all keys are `tree` stuff, associated to a non-null value)
- etc . . .

Each value has a **discriminant** (which for an object is its class).

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**[7], 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)

[7]Because designing a type-system friendly with Gcc internals mean making a type theory of Gcc internals!

Things = (Melt Values) ∪ (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 → 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 ();` [8]
  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**

[8] So Melt code can trigger Ggc collection even **inside** Gcc passes!

Melt Lisp-like look

Melt has a **lisp-like syntax**[9], so almost **every operator is in parenthesis**:

(operator operands ...)

So in Melt **`(f)`** is the call of function `f` without arguments like **`f()`** is in *C*
in Melt *function call* `(f)` $\not\equiv$ `f` *function value*, like in *C function call* `f()` $\not\equiv$ `f` *function address*

Melt is **expression-based**. Expressions are **evaluated** and produce a **result**:
$2 \times 3 + 5$ is **`(+i (*i 2 3) 5)`** $\Rightarrow$ `11`
`*i` and **`+i`** are names of primitive arithmetic operations handling *raw* `long` stuff.

Control operations usally have **names inspired by existing Lisp dialects**
`if cond lambda let`[10] **`letrec defun define definstance setq`**

Primitives and **standard functions** usually have **names different of Lisp** habits
(no `car`, `cons`, `string?`, > in Melt; but **`pair_head`**, **`list`**, **`>i`**, **`make_integerbox`**)

---

9 Because it is simple to parse, and because *Emacs* supports it.
10 Melt's **`let`** is **sequential**, like **Scheme's `let*`**

primitives and macro-strings

Definition of (stuff) addition:

```
(defprimitive +i (:long a b) :long
#{($A) + ($B)}#)
```

Macro-strings `#{...}#` mix *C* code with Melt symbols `$A`, used as “templates”

Primitives have a typed result and arguments.

Since locals are initially cleared, many Gcc related primitives test for null (e.g. `tree` or `gimple`) pointers, e.g.

```
(defprimitive gimple_seq_first_stmt (:gimple_seq gs) :gimple
  #{(($GS)?gimple_seq_first_stmt(($GS)):NULL) }#)
```

`:void` primitives translate to *C* statement blocks; other primitives are translated to *C* expressions

“hello world” in Melt with a code chunk

```
;; -*- lisp -*-       file   helloworld.melt
(code_chunk hello ;;state symbol
 #{ int $HELLO#_cnt =0;
 $HELLO#_lab:printf("hello world %d\n",$HELLO#_cnt++);
 if ($HELLO#_cnt <2) goto $HELLO#_lab; }#)
```

The “state symbol” `hello` is expanded to a unique *C* identifier (e.g. `HELLO_1` the first time, `HELLO_2` the second one, etc...), e.g. generates in *C*

```
int HELLO_1_cnt =0;
 HELLO_1_lab:printf("hello world %d\n", HELLO_1_cnt++);
 if (HELLO_1_cnt <2) goto HELLO_1__lab;
```

State symbols are really useful to generate unique identifiers in nested constructions like iterations.

`code_chunk` is for Melt → *C*, like **`asm`** is for *C* → *assembler*

c-iterators to generate iterative statements

Using an c-iterator

```
;; apply a function f to each boxed gimple in a  gimple seq gseq
(defun do_each_gimpleseq (f :gimple_seq gseq)
  (each_in_gimpleseq
   (gseq)    ;; the input of the iteration
   (:gimple g) ;; the local formals
   (let ( (gplval (make_gimple discr_gimple g)) ) ;; boxing a raw Gimple
     (f gplval))))
```

Defining the c-iterator

```
(defciterator each_in_gimpleseq
  (:gimple_seq gseq)                        ;start formals
  eachgimplseq                              ;state symbol
  (:gimple g)                               ;local formals
  ;;; before expansion
  #{/*$EACHGIMPLSEQ*/ gimple_stmt_iterator gsi_$EACHGIMPLSEQ;
   if ($GSEQ) for (gsi_$EACHGIMPLSEQ = gsi_start ($GSEQ);
          !gsi_end_p (gsi_$EACHGIMPLSEQ);
          gsi_next (&gsi_$EACHGIMPLSEQ)) {
    $G  = gsi_stmt (gsi_$EACHGIMPLSEQ); }#
  ;;; after expansion
  #{ } }# )
```

building MELT - requirements

(the experimental MELT branch is built like the GCC trunk)

The **MELT plugin** (version **0.9.2.b** for GCC 4.6) **requires** [6Gb RAM, 0.5Gb disk]

- a **GCC 4.6 compiler** [on Linux] with **plugins enabled**
  on Debian `aptitude install gcc-4.6 g++-4.6`
- GCC 4.6 **dependencies** (e.g. Parma Polyhedra Library, `gawk`, `texi2html`, ...)
  on Debian `aptitude build-dep gcc-4.6 g++-4.6`
- GCC 4.6 **plugin development files**
  on Debian `aptitude install gcc-4.6-plugin-dev`

These are needed **when building** `melt.so` and **when running** it
because Melt may fork a compilation of generated *C* code when running!

your Melt **extensions** (or GCC plugins) [nearly] should be **GPLv3 compatible**
`http://www.gnu.org/licenses/gcc-exception.html`

Legal prerequisites `gcc.gnu.org/contribute.html` (take time!!)
(**copyright transfer to FSF** needed before submitting even small patches to MELT or to GCC)

compiling the **melt.so** [meta-] plugin

1. retrieve & untar the latest MELT plugin source

```
wget http://gcc-melt.org/melt-0.9.2-plugin-for-gcc-4.6.tgz
tar xzvf melt-0.9.2-plugin-for-gcc-4.6.tgz
```

2. if you want, edit the `Makefile` (a symlink to `MELT-Plugin-Makefile`):

```
emacs melt-0.9.2-plugin-for-gcc-4.6/MELT-Plugin-Makefile
```

(you probably *don't need* to edit it)

3. run a **sequential** make (lasting about 8 minutes) :

```
cd melt-0.9.2-plugin-for-gcc-4.6; make
```

- the `melt.so` plugin for GCC is built (from `melt-runtime.c` ...)
- it is used to regenerate the Melt translator from the `warmelt*.melt` source
- the generated `warmelt*.c` are compiled into `warmelt*.so` modules
- the translation `warmelt*.melt` → `warmelt*.c` → `warmelt*.so` is repeated several times (bootstrapping)
- the extra standard modules `xtramelt*.melt` are also translated
- the Melt runtime is re-compiled with a Melt extension checking its coding style.

Melt should be **re-built** for even a tiny GCC change (i.e. `4.6.1` → `4.6.2`)

installing the **melt.so** [meta-] plugin

after successful compilation, in the same `melt-0.9.2-plugin-for-gcc-4.6/` directory:

1. run the installer with a temporary `DESTDIR`
   `make install DESTDIR=/tmp/meltinst`
2. copy that directory as root:
   `sudo cp -v -d -R /tmp/meltinst/ /`

On my Debian system it will populate `/usr/lib/gcc/x86_64-linux-gnu/`**`4.6/plugin/`** with ≈ 670 files (total 0.5Gb) like **`include/`**`melt-run.h` or **`melt-modules/`**`xtramelt-ana-base.e1807af85330ba5b5359e8208236c7c5.quicklybuilt`**`.so`** or **`melt-sources/`**`xtramelt-ana-base+02`**`.c`** or **`melt-sources/`**`xtramelt-ana-base`**`.melt`** or **`melt-module.mk`** and the Gcc plugin for Melt itself **`melt.so`**

NB. Melt makefiles could be better. Help and patches are welcome!

Running Melt - program arguments

As for every Gcc plugin, you need to ask for it with

```
gcc-4.6 -fplugin=melt
```

The **melt.so** plugin is actually `dlopen`-ed by the **cc1** or **cc1plus** compiler program, not its `gcc-4.6` driver. You usually need a `*.c` file to get `cc1` started.

Melt won't do anything useful without several additional plugin arguments, named **-fplugin-arg-melt-**α, e.g.

- **-fplugin-arg-melt-mode=** to specify the (mandatory) **mode** in which Melt should run. Melt don't do anything without a mode. Try `-fplugin-arg-melt-mode=help`
- **-fplugin-arg-melt-workdir=** to give a **work directory** (containing generated `.c` and `.so` files).

A `Makefile` using some Melt extension probably wants

```
CFLAGS +=  -fplugin=melt \
       -fplugin-arg-melt-workdir=my-melt-work-dir/
```

Running **helloworld.melt** directly

```
% gcc-4.6 -fplugin=melt -fplugin-arg-melt-mode=runfile \
    -fplugin-arg-melt-arg=helloworld.melt -c empty.c
cc1: note: MELT generating C code of module ↯
   /tmp/fileRZNNjT-GccMeltTmp-110f7f5b/helloworld
cc1: note: MELT generated new file  ↯
   /tmp/fileRZNNjT-GccMeltTmp-110f7f5b/helloworld.c
cc1: note: MELT generated C code of module  ↯
   /tmp/fileRZNNjT-GccMeltTmp-110f7f5b/helloworld  ↯
   with 0 secondary files in 0 CPU millisec.
MELT is building binary helloworld from source   ↯
   /tmp/fileRZNNjT-GccMeltTmp-110f7f5b/helloworld with  ↯
   flavor quicklybuilt
cc1: note: MELT plugin has built module helloworld flavor quicklybuilt ↯
   in /home/basile/MELT-InriaGrenoble
hello world 0
hello world 1
cc1: note: MELT removed 4 temporary files from  ↯
    /tmp/fileRZNNjT-GccMeltTmp-110f7f5b
```

Some ***C*** files are **generated and compiled** and **dlopen-ed** by Melt
(inside a *temporary* directory, cleaned up before cc1 exits)

Making a **helloworld.optimized.so** module

```
% gcc-4.6 -fplugin=melt \
    -fplugin-arg-melt-workdir=my-melt-work-dir/ \
    -fplugin-arg-melt-mode=translateoptimized  \
    -fplugin-arg-melt-arg=helloworld.melt -c empty.c
cc1: note: MELT generating C code of module helloworld
cc1: note: MELT generated new file helloworld.c in ↯
   /home/basile/MELT-InriaGrenoble
cc1: note: MELT generated C code of module helloworld  ↯
   with 0 secondary files in 0 CPU millisec.
MELT is building binary helloworld from source helloworld with ↯
   flavor optimized
cc1: note: MELT plugin has built module helloworld flavor optimized in ↯
   /home/basile/MELT-InriaGrenoble
```

```
% ls -l helloworld*
-rw-r--r-- 1 basile basile 11748 Dec  7 16:28 helloworld.c
-rw-r--r-- 1 basile basile 11748 Dec  7 16:28 helloworld.c%
-rw-r--r-- 1 basile basile   187 Dec  7 10:46 helloworld.melt
-rw-r--r-- 1 basile basile  1429 Dec  7 16:28 helloworld+meltdesc.c
lrwxrwxrwx 1 basile basile   149 Dec  7 16:28 helloworld.optimized.so ->  ↯
/home/basile/MELT-InriaGrenoble/my-melt-work-dir/ ↯
helloworld.d8216f8d73349ea62ba76a0c0f5a128f.optimized.so
```

Using the **helloworld.optimized.so** module

```
% gcc-4.6 -fplugin=melt \
    -fplugin-arg-melt-workdir=my-melt-work-dir/ \
    -fplugin-arg-melt-mode=nop \
    -fplugin-arg-melt-extra=./helloworld -c empty.c
hello world 0
hello world 1
```

- A **mode** is still needed (e.g. `nop`). Often, your Melt modules will install their own modes.
- one or several colon-separated **extra modules**[11] can be specified
- no compilation of generated *C* code happens (so faster)
- the generated *C* code is needed: conceptually, it is loaded as the modules, and the `*.so` are "cached"
- the file **`helloworld+meltdesc.c`** is mostly parsed **meta-data** about the generated *C* files (but also compiled as *C*)

[11] In addition of the standard ones!

MELT modules flavors

A given Melt module (the μ.so shared object `dlopen`-ed by the `melt.so` meta-plugin) comes with different **flavors** (different ways to build the μ.so from $\mu\star$.c, see **`melt-module.mk`** file)

- **`quicklybuilt`** flavor (for **development**): generated *C* code quickly compiled without (= `-O0`) optimization, but **with `#line`** directives **and Melt debugging** support.
  Use **`-fplugin-arg-melt-mode=translatequickly`**
- **`optimized`** flavor (for **production** use): compiled with (= `-O1`) optimization, but with `#line` directives and **without** Melt **debugging** support **`-fplugin-arg-melt-mode=translateoptimized`**
- **`debugnoline`** flavor (for low level debugging): compiled with (= `-g`) debugging information, without `#line` directives, and **with** Melt **debugging** support. **Rarely useful** to debug a Melt module with `gdb` **`-fplugin-arg-melt-mode=translatetodebug`**

Debugging hints

Two useful **debug-related** program arguments:

1. **-fplugin-arg-melt-debug** : if given, a lot of debugging output appear (except with **optimized** flavor of modules)
   Hint: run your Melt extension inside an *Emacs* shell buffer
2. **-fplugin-arg-melt-debugskip=**1234 to skip the first 1234 debugging messages.

Several **debugging constructs** in Melt (enabled with flavors, and at run time) :

- (**debug** *any arguments ....*) ; use it often!
- (**assert_msg** *"message-string" (assertion-test)*) ; when the *assertion-test* fails, a backtrace stack is printed with the *"message-string"*
- (**shortbacktrace_dbg** *"message-string" max-depth*) to print the backtrace stack

Using **gdb** is **rarely needed** (only for SIGSEGV) and painful (**debugnoline** flavor)

The helloworld+meltdesc.c "meta-data"

```
/** GENERATED MELT DESCRIPTOR FILE helloworld+meltdesc.c - ↯
  ** NEVER EDIT OR MOVE THIS, IT IS GENERATED & PARSED! **/
/* version of the GCC compiler & MELT runtime generating this */
const char melt_genversionstr[]="4.6 20111121 () [MELT plugin] MELT_0.9.2";
const char melt_versionmeltstr[]="0.9.2 [melt-branch_revision_182101]";
/* source name & real path of the module */
/*MELTMODULENAME helloworld */
const char melt_modulename[]="helloworld";
const char melt_modulerealpath[]="/home/basile/MELT-InriaGrenoble/helloworld";
/* MELT generation timestamp */
const char melt_gen_timestamp[]="Thu Dec  8 14:24:36 2011 CET";
const long long melt_gen_timenum=1323350676;
const char melt_build_timestamp[]= __DATE__ "@" __TIME__;
/* hash of preprocessed melt-run.h generating this */
const char melt_prepromd5meltrun[]="d41d8cd98f00b204e9800998ecf8427e";
/* hexmd5checksum of primary C file */
const char melt_primaryhexmd5[]="725c130e6c7eb8780c2e7f76c58eae0e";
/* hexmd5checksum of secondary C files */
const char* const melt_secondaryhexmd5tab[]= (const char*)0 ;
/* last index of secondary files */
const int melt_lastsecfileindex=0;
/* cumulated checksum of primary & secondary files */
const char melt_cumulated_hexmd5[]="725c130e6c7eb8780c2e7f76c58eae0e";
/* end of melt descriptor file */
```

NB: Melt parses & (conceptually) loads such files (the `*.so` modules are cached)

Some examples

Look at:

- the simple "high-order" *iterator* function **multiple_every** file ***melt/warmelt-base.melt*** near line 1435
- its *C* translation in **meltrout_18_warmelt_base_MULTIPLE_EVERY** file ***melt/generated/warmelt-base+01.c*** near line 4835; notice the Melt frame and normalization
- the Gcc pass **meltframe** (to check the `melt-runtime.c` file) coded in Melt file ***melt/xtramelt-ana-simple.melt*** lines 1090-1368:
  1. pass gate and execute functions
  2. *Gimple* and *Tree* pattern matching
  3. inserting the pass inside existing passes

Contents

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.[12]

[12] You probably should avoid reading many `*.c` code files at first.

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 **gengtype** 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**[13] 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 ...)

[13] Gcc don't use LALR parser generators like yacc or bison for C.

GCC middle-end

The middle-end is the most important[14] (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.

[14] Important 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, `lto1` is a "front-end" to this middle-end data contained in `*.o` files. The program `lto1` 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**[15] is the last layer of Gcc (specific to the target machine):

- it contains all **optimizations** (etc ...) **particular to its target system** (notably peepwhole target-specific optimizations).
- it **schedules** (machine) **instructions**
- it **allocates registers**[16]
- 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**

[15] A given `cc1` or `lto1` has usually one back-end (except multilib ie `-m32` vs `-m64` on `x86-64`). But Gcc source release has many back-ends!

[16] Register 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**, **gencondmd**, **genconstants**, **genemit**, **genenums**, **genextract**, **genflags**, **genopinit**, **genoutput**, **genpreds**, 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 `$GCCSOURCE/gcc/ggc*.[ch]`[17] and thru the **gengtype** generator `$GCCSOURCE/gcc/gengtype*.[chl]`.

- 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 called[18] 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)

---

[17] `ggc-zone.c` is often unused.

[18] Be 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 good[19] -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.

[19] Chicken & egg issue here: Ggc not good enough ⇒ not very used ⇒ 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)[20] also use **gengtype** & **GTY**.

[20] PCH 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] roots[21], 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[22]

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.

[21] That is, `extern` or `static` ***GTY***-ed variables.
[22] Thanks 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 collection[23], 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.

[23] Like 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 **$GCCSOURCE/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 ⇒ `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 flags[24] to gcc

- Each pass can **dump** information **into textual files**.
  ⇒ your new passes should provide dumps.[25]
- ⇒ 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 tree 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) [26].

[24] Next gcc-4.7 will have improved [before/after] flags
[25] Unless the pass name starts with *.
[26] Some Gcc gurus dream of a fully accurate and reparsable textual representation of *Gimple*

Dump example: input source `example1.c`

(using `gcc-melt`[27] svn rev. 174968 ≡ `gcc-trunk` svn rev. 174941, of june 11th 2011)

```c
/* example1.c */
extern int gex(int);

int foo(int x, int y) {
  if (x>y)
    return gex(x-y) * gex(x+y);
  else
    return foo(y,x);
}

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

[27] The 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 (x_2(D) > y_3(D))
    goto <bb 3>;
  else goto <bb 4>;

<bb 3>:
  D.2707_4 = x_2(D) - y_3(D);
  D.2708_5 = gex (D.2707_4);
  D.2709_6 = x_2(D) + y_3(D);
  D.2710_7 = gex (D.2709_6);
  D.2706_8 = D.2708_5 * D.2710_7;
  goto <bb 5>;

<bb 4>:
  D.2706_9 = foo (y_3(D), x_2(D));

<bb 5>:
  # D.2706_1 = Φ <D.2706_8(3), D.2706_9(4)>
  return D.2706_1;    }
```

SSA ⇔ each variable is assigned once; suffix `(D)` for default definitions of SSA names

e.g $D.2707_4$ [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.162**r**.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 [0x
            (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_interna
     (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 {*m
     (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_interna
     (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
# -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
.L9:
        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     #,
.L7:
        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"[28].

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...

[28] But 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 `lto1` 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∞** 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 wants[29] you to code Gcc extensions!

[29] Or 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 directory[30] 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 lto1 (caveat: front-end functions are not in lto1)

[30] This 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 `toplev.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 `toplev.c`) can be used for LTO summaries
7. **`PLUGIN_FINISH`** (called from `toplev.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

MELT values and GCC stuff

Melt deals with two kinds of **things**:

1. Melt first-class (dynamically typed) **values**
   objects, tuples, lists, closures, boxed strings, boxed gimples, boxed trees, homogenous hash-tables. . .
2. existing Gcc **stuff** (statically and explicitly typed)
   raw `long`-s, `tree`-s, `gimple`-s as already known by Gcc . . .

Essential distinction (mandated by lack of polymorphism of Ggc):

$$\textit{Things} = \textit{Values} \cup \textit{Stuff}$$

Melt code explicitly annotates stuff with **c-types** like **`:long`**, **`:tree`** . . . (and **`:value`** for values, when needed).

handling Melt values is preferred (and easier) in Melt code.

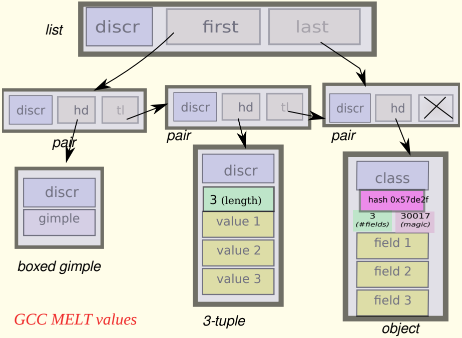
Melt argument passing is typed

Melt copying garbage collection for values

- copying Melt GC well suited for **fast allocation**[31] and many **temporary** (quickly dying) values
- live young values copied into Ggc heap (but needs write barrier)
- Melt GC requires **normalization** $z := \phi(\psi(x), y) \rightarrow \tau := \psi(x); z := \phi(\tau, y)$
- Melt GC handles **locals** and may trigger Ggc at any time
- well suited for **generated** *C* code
  hand-written code for Melt value is cumbersome
- old generation of values is the Ggc heap → built-in compatibility of Melt GC with Ggc
- Melt call frames are known to both Melt GC & Ggc
  call frames are singly-linked `struct`-ures.

[31] Melt values are allocated in a birth region by a pointer increment; when the birth region is full, live values are copied out, into Ggc heap, then the birth region is de-allocated.

Melt value taxonomy



- values boxing some stuff
- objects (single-inheritance; classes are also objects)
- tuples, lists and pairs
- closures and routines
- homogenous hash-tables (e.g. all keys are `tree` stuff, associated to a non-null value)
- etc ...

Each value has a **discriminant** (which for an object is its class).

primitives and macro-strings

Definition of (stuff) addition:

```
(defprimitive +i (:long a b) :long
#{($A) + ($B)}#)
```

Macro-strings `#{...}#` mix *C* code with Melt symbols `$A`, used as "templates"

Primitives have a typed result and arguments.

Since locals are initially cleared, many Gcc related primitives test for null (e.g. `tree` or `gimple`) pointers, e.g.

```
(defprimitive gimple_seq_first_stmt (:gimple_seq gs) :gimple
  #{(($GS)?gimple_seq_first_stmt(($GS)):NULL)}#)
```

`:void` primitives translate to *C* statement blocks; other primitives are translated to *C* expressions

"hello world" in Melt with a code chunk

```
(code_chunk hello ;;state symbol
 #{int $HELLO#_cnt =0;
 $HELLO#_lab:printf("hello world %d\n",$HELLO#_cnt++);
 if ($HELLO#_cnt <2) goto $HELLO#_lab;}#)
```

The "state symbol" is expanded to a unique *C* identifier (e.g. `HELLO_1` the first time, `HELLO_2` the second one, etc...), e.g. generates in *C*

```
int HELLO_1_cnt =0;
 HELLO_1_lab:printf("hello world %d\n", HELLO_1_cnt++);
 if (HELLO_1_cnt <2) goto HELLO_1__lab;
```

State symbols are really useful to generate unique identifiers in nested constructions like iterations.

c-iterators to generate iterative statements

Using an c-iterator

```
;; apply a function f to each boxed gimple in a  gimple seq gseq
(defun do_each_gimpleseq (f :gimple_seq gseq)
  (each_in_gimpleseq
   (gseq)    ;; the input of the iteration
   (:gimple g) ;; the local formals
   (let ( (gplval (make_gimple discr_gimple g)) )
     (f gplval))))
```

Defining the c-iterator

```
(defciterator each_in_gimpleseq
  (:gimple_seq gseq)                         ;start formals
  eachgimplseq                               ;state symbol
  (:gimple g)                                ;local formals
  ;;; before expansion
  #{/*$EACHGIMPLSEQ*/ gimple_stmt_iterator gsi_$EACHGIMPLSEQ;
   if ($GSEQ) for (gsi_$EACHGIMPLSEQ = gsi_start ($GSEQ);
          !gsi_end_p (gsi_$EACHGIMPLSEQ);
          gsi_next (&gsi_$EACHGIMPLSEQ)) {
    $G  = gsi_stmt (gsi_$EACHGIMPLSEQ); }#
  ;;; after expansion
  #{ } }# )
```

Contents

Pattern matching example: Talpo by Pierre Vittet

```
;;detect a gimple cond with the null pointer
;;the cond can be of type == or !=
;;returns the lhs part of the cond (or boxed null tree if no match)
(defun test_detect_cond_with_null (useless :gimple g )
    (match g
        (  ?(gimple_cond_notequal ?lhs
                                  ?(tree_integer_cst 0))
            (return (make_tree discr_tree lhs))
          )
        ( ?(gimple_cond_equal ?lhs
                              ?(tree_integer_cst  0))
            (return (make_tree discr_tree lhs))
          )
        (
          ?_
            (return (make_tree discr_tree (null_tree))))))
```

Patterns start with `?`, so **`?_`** is the wildcard (joker). **`?lhs`** is a pattern variable.

What match does?

- syntax is `(match` ϵ $\kappa_1 \ldots \kappa_n$ `)` 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 `(` π_i $\epsilon_{i,1} \ldots \epsilon_{i,n_i}$ η_i `)`, each starting with a pattern[32] 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**

[32] 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 `:`$\phi_1 \;\; \pi'_1 \quad \ldots \quad$ `:`$\phi_n \;\; \pi'_n$`)` 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

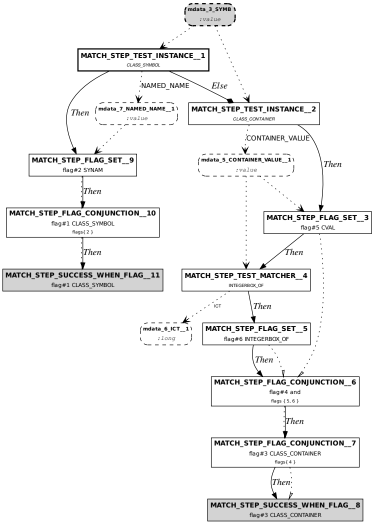
Two kinds of matchers:

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

translating pattern matching



Naive approach might be not very efficient: tests are done more than needed.

translate

```
(match v
   ( ?(instance class_symbol
            :named_name  ?synam)
      (f synam))
   ( ?(instance class_container
            :container_value
               ?(and ?cval
                  ?(integerbox_of ?_)))
      (g cval)))
```

into a graph of matching steps, with tests. Share steps when possible.

main Melt syntactic constructs

expressions where $n \geq 0$ and $p \geq 0$

application	$(\phi\ \alpha_1\ \ldots\ \alpha_n)$	apply function (or primitive) ϕ to arguments α_i
assignment	(**setq** $\nu\ \epsilon$)	set local variable ν to ϵ
message send	$(\sigma\ \rho\ \alpha_1\ \ldots\ \alpha_n)$	send selector σ to reciever ρ with arguments α_i
let expression	(**let** $(\beta_1 \ldots \beta_n)\ \epsilon_1 \ldots \epsilon_p\ \epsilon'$)	with local **sequential**[33] bindings β_i evaluate side-effecting sub-expressions ϵ_j and give result of ϵ'
sequence	(**progn** $\epsilon_1 \ldots \epsilon_n\ \epsilon'$)	evaluate ϵ_i (for their side effects) and at last ϵ', giving its result (like operator `,` in C)
abstraction[34]	(**lambda** $\phi\ \epsilon_1 \ldots \epsilon_n\ \epsilon'$)	anonymous function with formals ϕ and side-effecting expressions ϵ_i, return result of ϵ'
pattern matching	(**match** $\epsilon\ \chi_1\ \ldots\ \chi_n$)	match result of ϵ against match clauses χ_i, giving result of last expression of matched clause.

[33] So the `let` of Melt is like the `let*` of Scheme!

[34] abstractions are constructive expressions and may appear in letrec bindings

A cleared thing[35] (represented by all zero bits) is nil, or the `long` 0 stuff, or the null `gimple` or `tree` ... stuff. It is false.

conditional expressions where $n \geq 0$ and $p \geq 0$

test	`(if` τ θ ϵ`)`	if τ then θ else ϵ (like `?:` in C)
conditional	`(cond` κ_1 ... κ_n`)`	evaluate conditions κ_i until one is satisfied
conjonction	`(and` κ_1 ... κ_n κ'`)`	if κ_1 and then κ_2 ... and then κ_n is "true" (non nil or non zero) then κ' else the cleared thing of same type
disjunction	`(or` δ_1 ... δ_n`)`	the first of the δ_i which is "true" (non nil, or zero, ...)

In a `cond` conditional expression, every condition κ_i -except perhaps the last- is like `(`γ_i $\epsilon_{i,1}$ ... ϵ_{i,p_i} ϵ'`)` with $p_i \geq 0$. The first such condition for which γ_i is "true" gets its sub-expressions $\epsilon_{i,j}$ evaluated sequentially for their side-effects and gives its ϵ'. The last condition can be `(:else` ϵ_1 ... ϵ_n ϵ'`)`, is triggered if all previous conditions failed, and (with the sub-expressions ϵ_i evaluated sequentially for their side-effects) gives its ϵ'

[35] Every local thing (value, stuff ...) is cleared at start of its containing Melt function.

more expressions

loop	(**forever** λ α_1 ... α_n)	loop indefinitely on the α_i which may exit
exit	(**exit** λ ϵ_1 ... ϵ_n ϵ')	exit enclosing loop λ after side-effects of ϵ_i and result of ϵ'
return	(**return** ϵ ϵ_1 ... ϵ_n)	return ϵ as the main result, and the ϵ_i as secondary results
multiple call	(**multicall** ϕ κ $\epsilon_1 ... \epsilon_n$ ϵ')	locally bind formals ϕ to main and secondary result[s] of application or send κ and evaluate the ϵ_i for side-effects and ϵ' for result
recursive let	(**letrec** ($\beta_1 ... \beta_n$) $\epsilon_1 ... \epsilon_p$)	with co-recursive *constructive* bindings β_i evaluate sub-expressions ϵ_j
field access	(**get_field** :Φ ϵ)	if ϵ gives an appropriate object[36] retrieves its field Φ, otherwise nil
unsafe field access	(**unsafe_get_field** :Φ ϵ)	unsafe[37] access without check like above
object update	(**put_fields** ϵ :Φ_1 ϵ_1 ... :Φ_n ϵ_n)	safely update [38] (if appropriate) in object given by ϵ each field Φ_i by value of ϵ_i

[36] i.e. if the value ω of ϵ is an object which is a direct or indirect instance of the class defining field Φ.

[37] Only for Melt gurus, since it may crash!

[38] i.e. update object ω only if the value ω of ϵ is an object which is a direct or indirect instance of the class defining each field Φ_i

constructive expressions

list	(**list** α_1 ... α_n)	make a list of n values α_i
tuple	(**tuple** α_1 ... α_n)	make a tuple of n values α_i
instance	(**instance** κ :Φ_1 ϵ_1 ... :Φ_n ϵ_n)	make an instance of class κ and n fields Φ_i set to value ϵ_i

Abstractions (**lambda** expressions) are also constructive.

Constructive expressions may be recursively bound in **letrec**:

```
(letrec (
     (a (list b c))
     (b (tuple a b))
     (c (lambda (x y) (if (== x a) b y)))
     (d (instance class_container :container_value a))
  )
  (c d bar))
```

Note: contrarily to Scheme, **Melt has no tail recursive calls**.

Every [recursive] Melt call grows the stack (because it is translated to a C call).

expressions about names

expressions defining names

for functions	(**defun** ν ϕ ϵ_1 ... ϵ_n ϵ')	define function ν with formal arguments ϕ and body ϵ_1 ... ϵ_n ϵ'
for primitives	(**defprimitive** ν ϕ :θ η)	define primitive ν with formal arguments *phi*, result c-type θ by macro-string expansion η
for c-iterators	(**defciterator** ν Φ σ Ψ η η')	define c-iterator ν with input formals Φ, state symbol σ, local formals Ψ, start expansion η, end expansion η'
for c-matchers	(**defcmatcher** ν Φ Ψ σ η η')	define c-matcher ν with input formals Φ *[the matched thing, then other inputs]*, output formals Ψ, state symbol σ, test expansion η, fill expansion η'
for fun-matchers	(**defunmatcher** ν Φ Ψ ϵ)	define funmatcher ν with input formals Φ, output formals Ψ, with function ϵ

expressions exporting names

of values	(**export_value** ν_1 ...)	export the names ν_i as bindings of value (e.g. of functions, objects, matcher)
of macros	(**export_macro** ν ϵ)	export name ν as a binding of a macro (expanded by the ϵ function)
of classes	(**export_class** ν_1 ...)	export every class name ν and all their fields (as value bindings)
as synonym	(**export_synonym** ν ν')	export the new name ν as a synonym of the existing name ν'

miscellanous expressions

For all:

expressions for debugging

debug message	`(debug` ϵ ...)	debug printing message
assert check	`(assert_msg` μ τ)	nice "halt" showing message μ when asserted test τ is false
warning	`(compile_warning` μ ϵ)	like `#warning` in Gcc C: emit warning μ at Melt translation time and gives ϵ

meta-conditionals

Cpp test	`(cppif` σ ϵ ϵ')	conditional on a preprocessor symbol: emitted C code is `#if` σ code for ϵ `#else` code for ϵ' `#endif`
Version test	`(gccif` β ϵ_1 ...)	the ϵ_i are translated only if the Gcc translating them has version prefix string β

introspective expressions

Parent environment	`(parent_module_environment)`	gives the previous module environment
Current environment	`(current_module_environment_container)`	gives the container of the current module's environment

variadic functions

- `:rest` as last formal argument (like . . . in *C*)
- (**variadic** *variadic-cases*) construct to consume variadic arguments:

```
(defun varidbg (x y :rest)
   (forever argloop
     (variadic
      ( () ;; no more variadic argument
        (return))
      ( (:value v) ;; consume a value
         (debug "varidbg v=" v))
      ( (:tree t) ;; consume a raw tree
         (debug "varidbg t=" t))
      ( :else ;; unexpected kind
         (assert_msg "varidbg bad variadic")))))
```