# Extending the GCC compiler with MELT

Basile Starynkevitch
basile@starynkevitch.net or basile.starynkevitch@cea.fr

may, 22nd, 2013, EPITA LRDE seminar

## Caveat

All **opinions are mines only** (not of CEA or of GCC etc...)

- I (Basile) don't speak for my employer, CEA (or my institute LIST)
- I don't speak for GCC community
- I don't speak for anyone else (e.g. funding agencies)
- some of my opinions here are highly controversial
- (my opinions may change)

# Contents

# Programming languages

- programming languages are used by **human programmers**
- they are the *preferred* form to *communicate* between *human* programmers, and also between programmers and computers.
- programming languages are not understood by computers
- balance between
  - more expressive, more powerful, languages
  - established code legacy
- **free software** is about *source code*:
  - freedom to **use** the program and run it for any puprose
  - freedom to **study** the program (its source code), and change it
  - freedom to **redistribute** copies (in source form usually)
  - freedom to **improve** the program (its source)
- source code is *the* preferred form to work on programs (for human developers)

# the declarative ideal

## declarative knowledge

> *"Declarative knowledge is given without directions for use. [...] It is much easier to define, understand, and modify declarative knowledge"*

J.Pitrat [a french pionner in artificial intelligence]

*Artificial Beings (the conscious of a conscious machine)* [Wiley 2009]

Because of the growing gap between (much more) complex hardware systems and (even low-level) programming languages, programs need to be somehow "declaratively" understood by the system.

Programmers need more and more declarative languages to improve their productivity.

# C is becoming "silently" more "declarative"

While *C* is a low-level [system] programming language, it evolves to be less "procedural" (= giving code with usage instructions):

- **`register`** is obsolete and useless. The compiler will use machine registers better than a human programmer can.
- functions may be *inlined* (even without **`inline`**!) or [partially] cloned.
- some **`#pragma`**-s (notably for OpenMP) are useful hints to the compiler.

Notice that *C* recent code is quite different in style from 199x-s era. The programmer expects the *C* compiler to be smarter, and the *C* code is increasingly farther from the hardware[1].

So *C* (and *C++*, etc...) is becoming more expressive.

[1] Because current processors [e.g. Intel i7] are much more complex than 1990-s era ones [eg i486], even if they understand nearly the same instruction set.

# languages vs libraries

Languages, notably **domain specific languages**, are:

- usually easy to learn
- often difficult to implement
- making sense when more expressive (or “declarative”)

Libraries are:

- generally tied to a language (e.g. *C* as an “esperanto”)
- usually very complex (so are also hard to implement and to use)
- providing ad hoc abstractions (e.g. C++ “iterators”)
- difficult to learn

Unfortunately, people (i.e. decision makers) prefer new libraries to new languages (even if learning a library is much more difficult than learning a new programming language).

# About compilers

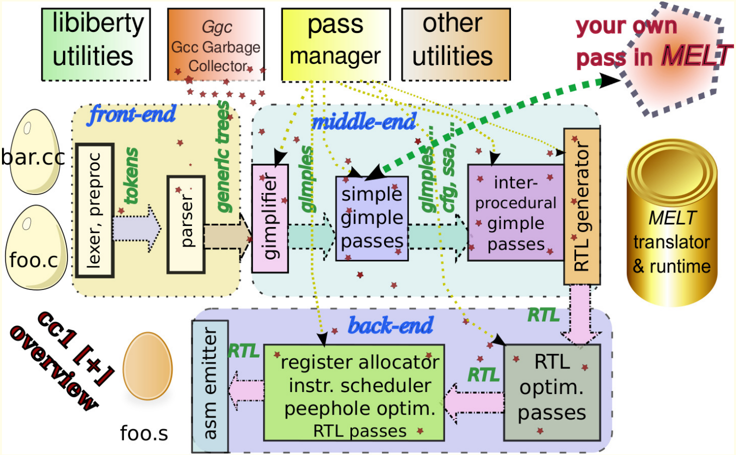
Roles of an “industrial strength” compiler :

- accept legacy source code base;
  huge source code bases exist (Firefox, Linux kernel, ... dozens of MLOC each)
- provide feedback to programmer: good diagnostics (warnings, errors) are increasingly important.
- ability to generate (when optimizing) good machine code, even for source programs increasingly far from machine constraints (out-of-order execution on parallel processing units [ → instruction scheduling], caching [→ prefetching], ...)

A good optimizing compiler needs to transform non-trivially its internal representations of the compiled program.

See A.Cohen et G.Fursin’s MILEPOST experiment: dozens of thousands of machine instructions generated from a trivial C code (matrix multiplication in a few lines of *C*), twice as efficient as `gcc -O2`.

# Internal complexity of *GCC*



# About GCC

The **GCC** compiler:

- coded in C and C++ (officially in C++ since 4.7, but most code is C like)
- current release 4.8 (march 2013) see **`gcc.gnu.org`** 108Mbyte `.tgz`
- community of $\approx$ 400 developers (mostly full time, paid by major corporations: Google, Intel, Suse, Redhat, ....)
- see also **`www.cse.iitb.ac.in/grc/`** and **`gcc-melt.org`**
- nearly 10MLOC: D.Wheeler *SLOCcount* 4,781,343;
  **`wc:`** `13978379 52386984 488154761 total`
- 25+ years old software
- peer reviewed software code
- use its own several specialized C code generators
- quite messy code: hundreds of global variables, ....
- some community members may be harsh
- several thousands of monthly messages: `gcc@gcc.gnu.org` (development) `gcc-patches@gcc.gnu.org` (patches and review)

# Timing `gcc -O2 -ftime-report -c melt-runtime.c`

Only lines with $\geq$ 2% wall time (**most of the work is "optimizing"**, not "parsing")

```
phase parsing            :   0.45 (10%) usr   0.23 (53%) sys   0.69 (14%) wall   75943 kB (36%) ggc
phase opt and generate   :   3.89 (89%) usr   0.20 (47%) sys   4.11 (85%) wall  135216 kB (63%) ggc
|name lookup             :   0.07 ( 2%) usr   0.02 ( 5%) sys   0.11 ( 2%) wall    2132 kB ( 1%) ggc
cfg cleanup              :   0.08 ( 2%) usr   0.00 ( 0%) sys   0.11 ( 2%) wall    1299 kB ( 1%) ggc
df live regs             :   0.20 ( 5%) usr   0.00 ( 0%) sys   0.22 ( 5%) wall       0 kB ( 0%) ggc
df live&initialized regs:   0.05 ( 1%) usr   0.00 ( 0%) sys   0.11 ( 2%) wall       0 kB ( 0%) ggc
df reg dead/unused notes:   0.09 ( 2%) usr   0.00 ( 0%) sys   0.15 ( 3%) wall    1481 kB ( 1%) ggc
preprocessing            :   0.08 ( 2%) usr   0.10 (23%) sys   0.20 ( 4%) wall   12572 kB ( 6%) ggc
parser (global)          :   0.10 ( 2%) usr   0.05 (12%) sys   0.16 ( 3%) wall   46233 kB (22%) ggc
parser function body     :   0.17 ( 4%) usr   0.06 (14%) sys   0.23 ( 5%) wall    9063 kB ( 4%) ggc
tree CFG cleanup         :   0.04 ( 1%) usr   0.00 ( 0%) sys   0.12 ( 2%) wall     252 kB ( 0%) ggc
tree VRP                 :   0.14 ( 3%) usr   0.00 ( 0%) sys   0.10 ( 2%) wall    4899 kB ( 2%) ggc
tree PRE                 :   0.13 ( 3%) usr   0.00 ( 0%) sys   0.09 ( 2%) wall    4101 kB ( 2%) ggc
tree FRE                 :   0.08 ( 2%) usr   0.02 ( 5%) sys   0.10 ( 2%) wall    4150 kB ( 2%) ggc
CSE                      :   0.14 ( 3%) usr   0.01 ( 2%) sys   0.12 ( 2%) wall     560 kB ( 0%) ggc
CPROP                    :   0.09 ( 2%) usr   0.00 ( 0%) sys   0.17 ( 4%) wall    3874 kB ( 2%) ggc
combiner                 :   0.15 ( 3%) usr   0.00 ( 0%) sys   0.23 ( 5%) wall    3575 kB ( 2%) ggc
integrated RA            :   0.25 ( 6%) usr   0.02 ( 5%) sys   0.26 ( 5%) wall   10322 kB ( 5%) ggc
reload CSE regs          :   0.16 ( 4%) usr   0.00 ( 0%) sys   0.13 ( 3%) wall    2788 kB ( 1%) ggc
scheduling 2             :   0.21 ( 5%) usr   0.00 ( 0%) sys   0.13 ( 3%) wall     466 kB ( 0%) ggc
rest of compilation      :   0.05 ( 1%) usr   0.01 ( 2%) sys   0.11 ( 2%) wall    1426 kB ( 1%) ggc
```

*... etc ... (85 other lines) ....*

```
TOTAL                 :   4.35             0.43             4.81             213018 kB
```

(preprocessed 103751 lines, 448560 word tokens; source: 15KLOC + 10 KLOC of MELT headers)

# Features of GCC

- free software mostly GPLv3+ licensed and FSF copyrighted `http://www.gnu.org/licenses/gcc-exception-3.1.en.html` permit compilation of proprietary programs
- several accepted source languages :
  C, C++, Objective C, Ada, Fortran, Go, (Java, D, ...)
- many host and target operating systems (Linux, Hurd, AIX, Solaris, MacOSX, Windows, ...)
- many target processors and systems, ABIs (x86, Sparc, ARM, PowerPC, ... both 32 and 64 bits, and many others)
- can be a cross-compiler (even Canadian Cross compiler)
- accepts (free software) **plugins**
- many program options (e.g. `-O2 -flto -g` etc etc...)
- competitive and complex optimizations
- $>$ 200 optimization passes (tree organized pass manager)
  most passes are in the middle-end (source and target “independent”)

# Bootstrapping

Using a compiler to compile itself.

Usual practice:

- Ocaml compiler is coded in Ocaml. The primordial compiler is distributed as bytecode with the source.
- Rust (Mozilla language) is coded in Rust. The installation procedure fetches old binaries on the Web.
- GCC: the compiler (including a lot of generated C code) is compiling itself several times `stage1`, `stage2`, `stage3`. Its *Ada* front-end is in *Ada*.
- MELT: the MELT to C translator is bootstrapped. The source code repository also contains its translated form in `melt/generated/*.[ch]` (2MLOC). But some code (e.g. `melt-runtime.c`) is still mostly hand written.
- J.Pitrat's *CAIA* declarative system is entirely bootstrapped: generates all of its 500KLOC of low-level C (but still requires an *optimizing* C compiler)

# Why bootstrap a compiler?

- even a trivial compiler (`tinycc` 30KLOC) is complex. Even a simple translator (*MELT* 63KLOC of MELT code) is complex. A real compiler (GCC, LLVM) is huge: **bootstrapping is a good test**
- social issue: self confidence of the compiler coder
- for evolving high-level languages, progessively **improve the expressivity** of the language; replace old parts of the system with better new parts : trivial example `(if` *test* `(begin` *exprs* ...`))`
  $\rightarrow$ `(when` *test* *exprs* ... `)`
  bootstrapping as a ladder for more declarativity
  See J.Pitrat's work for more.
- ideally requires an IDE-like[2] tool (within the translator) to help refactoring

NB: some compilers are not bootstrapped (Fortran front-end)

[2] Integrated Development Environment; clever editor; emacs mode; ....

# MELT

MELT **`gcc-melt.org`** is a [meta-]plugin for GCC providing a high-level domain specific language to extend GCC.

- plugging Ocaml into GCC is not humanly feasible (I tried)
  GCC has more than 2000 types and $\approx 10 MLOC$ [3]
- MELT is a free (GPLv3 licensed, FSF copyrighted) plugin for GCC 4.6 or 4.7 or 4.8
- MELT is a DSL fitting into GCC internals
- MELT provide some features of Ocaml (or Scheme)
  1. garbage collection of values
  2. pattern matching
  3. high-order programming (closures)
  4. (but not static typing or type inference) unlike Ocaml, MELT is a mostly dynamicly typed language (à la Scheme)

[3] See David Malcom's `gcc-python-plugin`

# GCC internal representations

GCC has many rich internal representations
(thousands of C data types, i.e. `struct`)

- *Tree*-*s*[4] for the AST of declarations, source [or SSA] variables, operands
- *Gimple*-*s*[5] for the simple instructions (e.g. 3 operands instructions à la $x \leftarrow y + z$)
- `basicblock`-s made of `gimple`-s (thru `gimpleseq`-s)
- `edge`-s for the control flow graph, between `basicblock`-s
- etc

The `GTY(())` annotation is for garbage collection in Gcc source code

[4] 200 different variants of `tree`-s, see file `gcc/tree.def` of Gcc
[5] 38 different variants of `gimple`-s, see file `gcc/gimple.def`, half for OpenMP

# Looking into some of the GCC internals

- dumping facilities, e.g. `gcc -fdump-tree-all -O -c foo.c` gives hundreds of files like[6] `foo.c.073t.phiopt1` ...
- with MELT's probe facility:
  `gcc -fplugin=melt -fplugin-arg-melt-mode=probe -O -c foo.c`
  - `-fplugin=melt` loads the MELT plugin[7]
  - `-fplugin-arg-melt-mode=probe` gives the *mode* for the MELT plugin[8]
  - MELT has many other options `-fplugin-arg-melt-debug` shows a lot of debugging output (to debug MELT or your MELT extensions).

[6] the number `073t` is absolutely meaningless

[7] You could load several plugins, but you usually load one at most

[8] without any mode, MELT does nothing. Use the `help` mode to get help about existing modes.

# Contents

# Motivations for MELT

Gcc extensions address a limited number of users[9], so their development should be facilitated (cost-effectiveness issues)

- extensions should be [meta-] plugins, not Gcc variants [branches, forks] [10] 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

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

# Embedding a scripting language is impossible

Many scripting or high-level languages [11] 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 ...*/ }
```

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

# Melt, a Domain Specific Language translated to C

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

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

The Melt language itself is tuned to fit into Gcc
In particular, it handles both its own Melt values and existing Gcc stuff

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

With Melt, Gcc **may generate *C* code** while running, compiles it[12] into a Melt binary `.so` module and `dlopen`-s that module.

[12] 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**[13], 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)

[13] 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 ();` [14]
  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 $\tau$ = `g(x); z=f(`$\tau$`, y);` )
  8. **well suited for *generated* C code**

[14] 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

# “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** [15] 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]

[15] 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 **plugin**:

```
gcc-4.7 -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

- Melt don't do anything more than Gcc without a **mode**
  - so without any mode, `gcc -fplugin=melt` ≡ `gcc`
  - use **`-fplugin-arg-melt-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

# Melt modes for translating `*.melt` files

(usually run on `empty.c`)

The name of the `*.melt` file is passed with
**`-fplugin-arg-melt-arg=`*`filename`*`.melt`**
The **mode** $\mu$ passed with **`-fplugin-arg-melt-mode=`**$\mu$

- **`translatedebug`** to translate into a `.so` Melt module built with `gcc -fPIC -g`
- **`translatequickly`** to translate into a `.so` Melt module built with `gcc -fPIC -O0`
- **`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 **`-fplugin-arg-melt-extra=`**$\mu_1$**`:`**$\mu_2$ to also load your $\mu_1$ & $\mu_2$ modules

# Melt modes for running `*.melt` files

The **`-fplugin-arg-melt-workdir=`*`directory`*** is very useful: the work directory help “caching” C and `.so` generated file.

- the **`runfile`** mode to **translate** into a *C* file, `make` the *`filename`*`.so` Melt module, load it, **then discard everything**.
- the **`repl`** mode to run an interactive read eval print loop (reading several expressions at once, ended by two newlines).
- the **`eval`** mode to evaluate expressions from argument
- the **`evalfile`** mode to evaluate expressions from a file

Evaluation prints the last evaluated expressions

# 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*); `when`, `unless` sugar
  `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**[16]
- `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

[16] 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
- **new in 0.9.9** `defhook` to define hooks, i.e. routines (called by C code) with a C calling convention coded in MELT.

**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` $\alpha$`)`
  - `(quote 2)` $\equiv$ `'2` is a boxed constant integer (but `2` is a constant long thing)
  - `(quote "ab")` $\equiv$ `'"ab"` is a boxed constant string
  - `(quote x)` $\equiv$ `'x` is a constant symbol (instance of `class_symbol`)

`quote` in Melt is different than `quote` in Lisp or Scheme.
In Melt it makes constant boxed values, so `'2` $\not\equiv$ `2`

# 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

# 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

# *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[17] or `basic_block`-s, ...)
  ⇒ use other data (e.g. associative hash tables) to link your data to them

[17] *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[18] 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
- (new in 0.9.8) :auto implicit annotation inside **let**

[18] 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
        (debug "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**

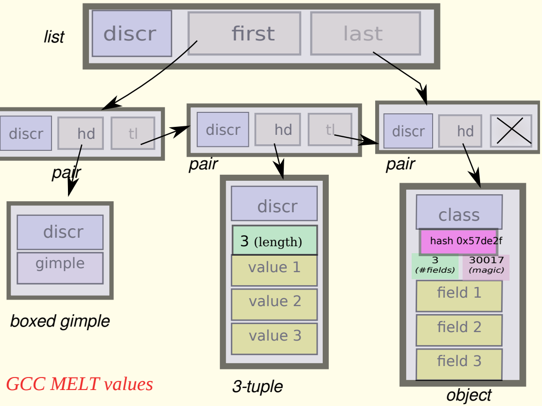
in `xtramelt-ana-base.melt`

```
(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** $\equiv$ fixed array of immutable components
- **associative homogenous hash-maps**, keyed by either
  - non-nil Gcc raw stuff like `:tree`-s, `:gimple`-s ... **(all keys of same type)**, or
  - Melt objects

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

# lattice of discriminants

- Each value has its immutable 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 $\sigma$ & a reciever $\rho$, i.e. **(**$\sigma$ $\rho$ **...)**
  - 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` $\pi$`)`, much like `'`$\epsilon \equiv$ `(quote` $\epsilon$`)`
- **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 $\epsilon$ an expression giving $\mu$ and $\kappa_j$ 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[19] $\pi_i$ followed by sub-expressions $\epsilon_{i,j}$ ending with $\eta_i$
- it matches (or filters) some thing $\mu$
- **pattern variables** are **local** to their clause, and **initially cleared**
- when pattern $\pi_i$ matches $\mu$ the expressions $\epsilon_{i,j}$ of clause *i* are executed in sequence, with the pattern variables inside $\pi_i$ locally bound. The last sub-expression $\eta_i$ of the match clause gives the result of the entire `match` (and all $\eta_i$ 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 $\pi_i$ can **match** the thing $\mu$ or **fail**

[19] expressions, e.g. constant litterals, are degenerate patterns!

# pattern matching rules

rules for matching of pattern $\pi$ against thing $\mu$:

- the **joker pattern `?_` always match**
- an **expression** (e.g. a constant) $\epsilon$ (giving $\mu'$) matches $\mu$ **iff** `(`$\mu'$ `==` $\mu$`)` in *C* parlance
- a **pattern variable** like **`?x`** matches if
  - `x` was unbound; then it is **bound** (locally to the clause) to $\mu$
  - or else `x` was already bound to some $\mu'$ and `(`$\mu'$ `==` $\mu$`)` ***[non-linear patterns]***
  - otherwise (`x` was bound to a different thing), the pattern variable `?x` match fails
- a **matcher pattern** `?(`$m$ $\eta_1 \ldots \eta_n$ $\pi'_1 \ldots \pi'_p$`)` with $n \geq 0$ input argument sub-expressions $\eta_i$ and $p \geq 0$ sub-patterns $\pi'_j$
  - the matcher $m$ does a **test** using results $\rho_i$ of $\eta_i$;
  - if the test succeeds, data are extracted in the **fill** step and each should match its $\pi'_j$
  - otherwise (the test fails, so) the match fails
- an **instance pattern** `?(`**`instance`** $\kappa$ `:`$\phi_1$ $\pi'_1$ ... `:`$\phi_n$ $\pi'_n$`)` matches iff $\mu$ is an object of class $\kappa$ (or a sub-class) with each field $\phi_i$ matching its sub-pattern $\pi'_i$

# control patterns

We have controlling patterns

- **conjonctive pattern** `?(and` $\pi_1 \ldots \pi_n$`)` matches $\mu$ iff $\pi_1$ matches $\mu$ and then $\pi_2$ matches $\mu$ ...
- **disjonctive pattern** `?(or` $\pi_1 \ldots \pi_n$`)` matches $\mu$ iff $\pi_1$ matches $\mu$ or else $\pi_2$ matches $\mu$ ...

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

# Contents

# work to be done on MELT (language and implementation)

- even more powerful matcher (perhaps backtracking)
- C++ generation:
  - friendly call frames, enabling introspection
  - C++ friendly MELT values
- LTO support (technically difficult)
  persitency
- Web interface and project persistency machinery
  (value related)
- code real multi-translation unit static analyzers
  (coding rules validation, ...)
- pass real sized applications, perhaps GCC itself
- getting more users

# compilation dreams - low level languages

Both GCC and LLVM suck. We ideally need new compilers (for low level languages like C, C++, Rust, Go, ...)

- incremental [re]compilation
- modularity (see LLVM module proposal for C and C++)
- multi-threaded compiler
- silent JIT techniques for C or C++
- heterogeneous architectures
- mixing static analysis, compilation, development environment (refactoring)
- generating C code inside a compiler is a good idea

# compilation dreams - new low level languages

Like Rust, Go, ....

Something in which the successor of Linux (or of Firefox, or of Apache) could be coded in

Something in which GC could be coded

# compilation dreams - high level declarative languages

Compilers are a typical example of why they are needed!

We need even more declarative languages to code even more complex compilers