MELT, a Translated Domain Specific Language Embedded in the *GCC* Compiler

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

energie atomique • energies alternatives

September 6th 2011 – *DSL 2011 conference*
(LABRI - Talence [near Bordeaux], France)

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

---

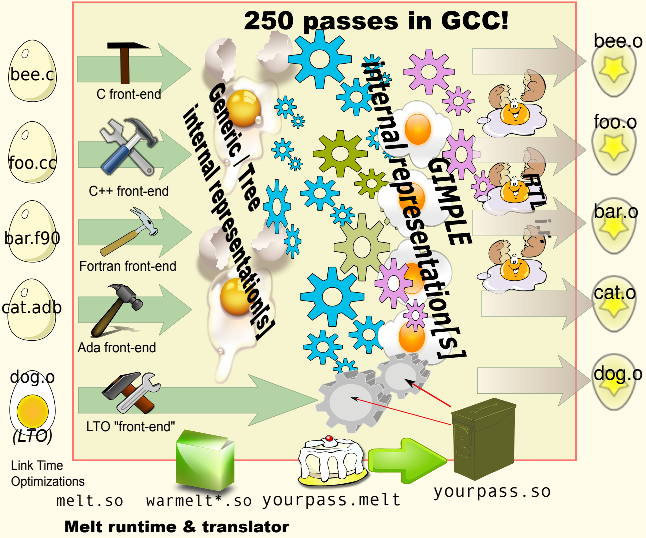
[1] Work on Melt have been possible thru the GlobalGCC ITEA and OpenGPU FUI collaborative research projects, with funding from DGCIS

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** (≈ **400** "maintainers", mostly full-time professionals)
- **peer-reviewed** development process, but **no main architect** ⇒ (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...)

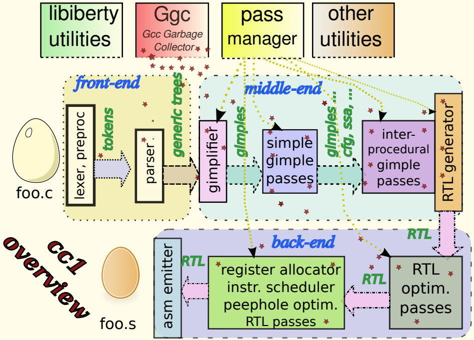
Current version (july 2011) is `gcc-4.6.1`

Gcc & Melt



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 [40 kloc] (of the translator) and its *C* translation [1200 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

[20 kloc of *C*, including utilities]

- 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):

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

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

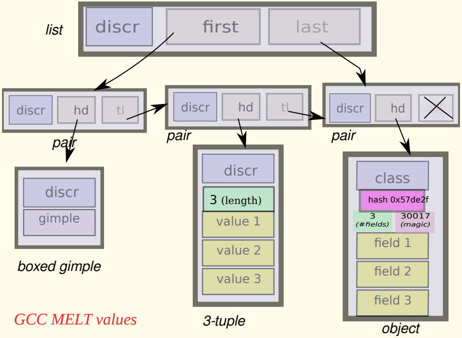
Melt copying garbage collection for values

- 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 → 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



GCC MELT values

- 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[7] 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**

---

[7] 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** $(\mu'$ **`==`** $\mu)$ in *C* parlance
- a **pattern variable** like **`?x`** matches if
  - `x` was unbound; then it is **bound** (locally to the clause) to μ
  - or else `x` was already bound to some μ' and $(\mu'$ **`==`** $\mu)$ ***[non-linear patterns]***
  - otherwise (`x` was bound to a different thing), the pattern variable `?x` match fails
- a **matcher pattern** $?(m\ \ \eta_1 \ldots \eta_n\ \ \pi'_1 \ldots \pi'_p)$ 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 :ϕ_1 π'_1 ... :ϕ_n π'_n) matches iff μ is an object of class κ (or a sub-class) with each field ϕ_i matching its sub-pattern π'_i

control patterns

We have controlling patterns

- **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

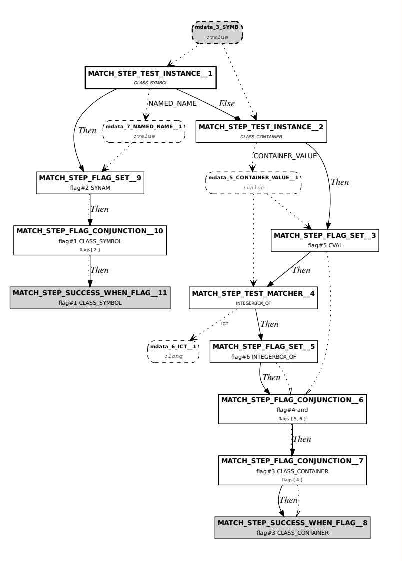
Two kinds of matchers:

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.

Contents

Translating to *C* is a pragmatic approach

- To add a DSL into a huge legacy program, embedding existing DSL may be not practical.
- Generating suitable *C* code suited to the target program is more flexible.
- Defining proper language constructs for *C* code generation
- Fitting into the legacy of the target program (adapting your runtime)
- providing high level constructs

Melt approach might be re-used for other big mature software

(because embedding a DSL is a major architectural issue)

MELT can be useful for your DSL

DSL implementations (in C or C++) require some coding styles and rules.

Melt extensions to Gcc can check these.

Thanks

Thanks (alphabetically) to:

- Romain Geissler
- Marie Krumpe
- Alexandre Lissy
- Jérémie Salvucci
- Pierre Vittet

for using and improving Melt
Thanks to Albert Cohen, Jan Midtgaard, Nic Volanschi and to the anonymous reviewers for help for the paper.
Thanks to you for your attention.

Questions are welcome.