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

Basile STARYNKEVITCH

CEA, LIST

Software Safety Laboratory, boîte courrier 94, 91191 GIF/YVETTE CEDEX, France

`basile@starynkevitch.net` `basile.starynkevitch@cea.fr`

The GCC free compiler is a very large software, compiling source in several languages for many targets on various systems. It can be extended by plugins, which may take advantage of its power to provide extra specific functionality (warnings, optimizations, source refactoring or navigation) by processing various GCC internal representations (Gimple, Tree, ...). Writing plugins in C is a complex and time-consuming task, but customizing GCC by using an existing scripting language inside is impractical. We describe *MELT*, a specific Lisp-like DSL which fits well into existing GCC technology and offers high-level features (functional, object or reflexive programming, pattern matching). MELT is translated to C fitted for GCC internals and provides various features to facilitate this. This work shows that even huge, legacy, software can be a posteriori extended by specifically tailored and translated high-level DSLs.

1 Introduction

GCC[1] is an industrial-strength free compiler for many source languages (C, C++, Ada, Objective C, Fortran, Go, ...), targetting about 30 different machine architectures, and supported on many operating systems. Its source code size is huge (4.296MLOC[2] for GCC 4.6.0), heterogenous, and still increasing by 6% annually[3]. It has no single main architect and hundreds of (mostly full-time) contributors, who follow strict social rules[4].

1.1 The powerful GCC legacy

The several GCC [8] front-ends (parsing C, C++, Go ... source) produce common internal *AST* (abstract syntax tree) representations called *Tree* and *Generic*. These are later transformed into middle-end internal representations, the *Gimple* statements - through a transformation called *gimplification*. The bulk of the compiler is its *middle-end* which operates repeatedly on these *Gimple* representations[5]. It contains nearly 200 passes moulding these (in different forms). Finally, back-ends (specific to the target) work on *Register Transfer Language* (RTL) representations and emit assembly code. Besides that, many other

[1] Gnu Compiler Collection (`gcc 4.6.0` released on march 25th 2011) on `gcc.gnu.org`

[2] 4.296 Millions Lines Of source Code, measured with David Wheeler's SLOCCount. Most other tools give bigger code measures, e.g., `ohcount` gives 8.370MLOC of source, with 5.477MLOC of code and 1.689MLOC of comments.

[3] GCC 4.4.1, released July 22th, 2009, was 3.884MLOC, so a 0.412MLOC = 10.6% increase in 1.67 years

[4] Every submitted code patch should be accepted by a code reviewer who cannot be the author of the patch, but there is no project leader or head architect, like Linus Torvalds is for the Linux kernel. So GCC has not a clean, well-designed, architecture.

[5] The GCC middle-end does not depend upon the source language or the target processor (except with parameters giving `sizeof(int)` etc.).

To appear in EPTCS.

data structures exist within GCC (and a lot of global variables). Most of the compiler code and optimizations work by various transformations on middle-end internal representations. GCC source code is mostly written in *C* (with a few parts in C++, or Ada), but it also has several internal *C* code generators. GCC does not use parser generators (like `flex`, `bison`, etc).

It should be stressed that *most* internal GCC *representations* are constantly *evolving*, and *there is no stability*[6] of the internal GCC API[7]. This makes the embedding of existing scripting languages (like Python, Ocaml, ...) impractical (§1.2). Since `gcc 4.5` it is possible to enhance GCC through external *plugins*.

External *plugins* can enhance or modify the behavior of the GCC compiler through a defined interface, practically provided by a set of *C* file headers, and made of functions, many *C* macros, and coding conventions. Plugins are loaded as `dlopen`-ed dynamic shared objects at `gcc` run time. They can profit from all the variety and power of the many internal representations and processing of GCC. Plugins enhance GCC by inserting new passes and/or by responding to a set of plugin events (like `PLUGIN_FINISH_TYPE` when a type has been parsed, `PLUGIN_PRAGMAS` to register new pragmas, ...).

GCC plugins can add specific warnings (e.g., to a library), specific optimizations (e.g., transform `fprintf(stdout,...)` → `printf(...)` in user code with `#include <stdio.h>`), compute software metrics, help on source code navigation or code refactoring, etc. GCC extensions or plugins enable using and extending GCC for non code-generation activities like static analysis [9, 2, 17, 28], threats detection (like in Two[10], Coverity™ [8], or Astrée[4, 5]), code refactoring, coding rules validation[16], etc. They could provide any processing taking advantage of the many facilities already existing inside GCC. However, since coding GCC plugins in *C* is not easy, a higher-level DSL could help. Because GCC plugins are usually specific to a narrow user community, shortening their development time (through a higher-level language) makes sense.

```
/* A node in a gimple_seq_d.  */
struct GTY((chain_next ("%h.next"), chain_prev ("%h.prev"))) gimple_seq_node_d {
  gimple stmt;
  struct gimple_seq_node_d *prev;
  struct gimple_seq_node_d *next; };
```

(code from `gcc/gimple.h` in GCC)

Figure 1: example of `GTY` annotation for Gg-c

Since compilers handle many complex (perhaps circular) data structures for their internal representations, explicitly managing memory is cumbersome during compilation. So the GCC community has added a crude *garbage collector* [11] Gg-c (GCC Garbage Collector): many *C* `struct`-ures in GCC code are annotated with `GTY` (figure 1) to be handled by Gg-c; passes can allocate them, and a precise [9] mark and sweep garbage collection may be triggered by the pass manager *only between passes*. Gg-c does not know about *local* pointers, so garbage collected data is live and kept only if it is (indirectly) reachable from known global or static `GTY`-annotated variables (data reachable only from local variables would be lost). Data internal to a GCC pass is usually manually allocated and freed. `GTY` annotations on types and

[6] This is nearly a dogma of its community, to discourage proprietary software abuse of GCC.

[7] GCC has no well defined and documented Application Programming Interface for compiler extensions; its API is just a big set of header files, so is a bit messy for outsiders.

[8] See `www.coverity.com`

[9] Gg-c is a *precise* G-C knowing each pointer to handle; using Boehm's conservative garbage collector with ambigous roots inside GCC has been considered and rejected on performance grounds.

variables inside GCC source are processed by `gengtype`, a specialized generator (producing *C* code for Gg-c allocation and marking routines and roots registration). There are more than 1800 GTY-ed types known by Gg-c, such as: `gimple` (pointing to the representation of a *Gimple* statement), `tree` (pointer to a structure representing a *Tree*), `basic_block` (pointing the representation of a basic block of Gimple-s), `edge` (pointing to the data representing an edge between basic blocks in the control flow graph), etc. Sadly, not all GCC data is handled by Gg-c; a lot of data is still manually micro-managed. We call ***stuff*** all the GCC internal data, either garbage-collected and GTY-annotated like `gimple`, `tree`, ..., or outside the heap like raw `long` numbers, or even manually allocated like `struct opt_pass` (data describing GCC optimization passes).

GCC is a big legacy system, so its API is large and quite heterogenous in style. It is not only made of data declarations and functions operating on them, but also contains various *C* macros. In particular, iterations inside internal representations may be provided by various styles of constructs:

1. Iterator abstract types like (to iterate on every `stmt`, a gimple inside a given 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 ...*/ }
```
2. Iterative `for`-like macros, e.g., (to iterate for each basic block `bb` inside the current function `cfun`)
```
basic_block bb;   FOR_EACH_BB (bb) { /* process bb */ }
```
3. More rarely, passing a callback to an iterating "higher-order" *C* function, e.g., (to iterate inside every index tree from `ref` and call `idx_infer_loop_bounds` on that index tree)
```
for_each_index (&ref, idx_infer_loop_bounds, &data);
```
with a static function `bool idx_infer_loop_bounds (tree base, tree *idx, void *dta)` called on every index tree `base`.

1.2 Embedding an existing scripting language is impractical

Interfacing GCC to an existing language implementation like Ocaml, Python, Guile, Lua, Ruby or some other scripting language is not realistic [10], because of an *impedance mismatch*:

1. Most scripting languages are garbage collected, and *mixing several garbage collectors is difficult and error-prone*, in particular when both Gg-c and scripting language heaps are intermixed.
2. The GCC API is very big, ill-defined, heterogenous, and evolving significantly. So manually coding the glue code between GCC and a general-purpose scripting language is a big burden, and would be obsoleted by a new GCC version when achieved.
3. The GCC API is not only made of *C* functions, but also of macros which are not easy to call from a scripting language.
4. Part of the GCC API is very low-level (e.g., field accessors), and would be invoked very often, so may become a performance bottleneck if used through a glue routine.
5. GCC handles various internal data (notably using hundreds of global variables), some through GTY-ed Gg-c collected pointers (like `gimple_seq`, `edge`, ...), others with manually allocated data (e.g., `omp_region` for OpenMP parallel region information) or with numbers mapping some opaque information (e.g., `location_t` are integers encoding source file locations). GCC data has widely different types, usage conventions, or liveness.
6. There is no single root type (e.g., a root class like `GObject` [11] in Gtk) which would facilitate gluing

[10] The author spent more than a month of work trying in vain to plug Ocaml into GCC!

[11] See `http://developer.gnome.org/gobject/`

GCC into a dynamically typed language interpreter (à la Python, Guile, or Ruby).

7. Statically typing GCC data into a strongly typed language with type inference like Ocaml or Haskell is impractical, since it would require the formalization of a type theory compatible with all the actual GCC code.
8. Easily filtering complex nested data structures is very useful inside compilers, so most GCC extensions need to *pattern-match* on existing GCC *stuff* (notably on *Gimple* or *Tree*-s).

The MELT (originally meaning *"Middle End Lisp Translator"*) Domain Specific Language has been developped to increase, as any high-level DSL does, the programmer's productivity. MELT has its specific generational copying garbage collector above Gg-c to address point 1. Oddity of the GCC API (points 2, 3, 4) is handled by generating well fit *C* code, and by providing mechanisms to ease that *C* source code generation. Items 4, 5, 6, 7 are tackled by mixing MELT dynamically typed values with raw GCC *stuff*. MELT has a powerful pattern matching ability to handle last point 8, because scripting languages don't offer extensible or embeddable pattern matching (on data structures internal to the embedding application).

MELT is being used for various GCC extensions (work in progress):

- simple warning and optimization like `fprintf(stdout,...)` detection and transformation (handling it on *Gimple* representation is preferable to simple textual replacement, because it cooperates with the compiler inlining transformation);
- Jérémie Salvucci has coded a *Gimple* $\rightarrow$ *C* transformer (to feed some other tool);
- Pierre Vittet is coding various domain-specific warnings (e.g., detection of untested calls to `fopen`);
- the author is developing an extension to generate *OpenCL* code from some *Gimple*, to transport some highly parallel regular (e.g., matrix) code to GPUs;

1.3 MELT = a DSL translated to code friendly with GCC internals

The legacy constraints given by GCC on additional (e.g., plugins') code suggest that a DSL for extending it could be implemented by generating *C* code suitable for GCC internals, and by providing language constructs translatable into *C* code conforming to GCC coding style and conventions. Other attempts to embed a scripting language into GCC (Javascript [9] for coding rules in Firefox, Haskell for enhancing C++ template meta-programming [1], or Python[12]) have restricted themselves to a tiny part of the GCC API; Volanschi [29] describes a modified GCC compiler with specialized matching rules.

Therefore, **the reasonable way to provide a higher-level domain specific language for GCC extensions is to dynamically generate suitable *C* code** adapted to GCC's style and legacy and similar in form to existing hand-coded *C* routines inside GCC. This is the driving idea of our MELT domain specific language and plugin implementation [24, 25, 26]. By generating suitable *C* code for GCC internals, MELT fits well into existing GCC technology. This is in sharp contrast with the Emacs editor or the C-- compiler [23] whose architecture was designed and centered on an embedded interpreter (E-Lisp for Emacs, Lua$^{\text{ocaml}}$ for C--).

MELT is a Lisp-looking DSL designed to work on GCC internals. It handles both dynamically typed MELT values and raw GCC *stuff* (like `gimple`, `tree`, `edge` and many others). It supports applicative, object and reflective programming styles. It offers powerful pattern matching facilities to work on GCC internal representations, essential inside a compiler. It is translated into *C* code and offer linguistic devices to deal nicely with GCC legacy code.

[12] See David Malcom's GCC Python plugin announced in `http://gcc.gnu.org/ml/gcc/2011-06/msg00293.html`

2 Using MELT and its runtime.

2.1 MELT usage and organization overview

From the user's perspective, the GCC compiler enabled with MELT (GCC$^{\text{melt}}$) can be run with a command as: `gcc -fplugin=melt -fplugin-arg-melt-mode=opengpu -O -c foo.c`. This instructs `gcc` (the `gcc-4.6` packaged in Debian) to run the compiler proper `cc1`, asks it to load the `melt.so` plugin which provides the MELT specific runtime infrastructure, and passes to that plugin the argument `mode=opengpu` while `cc1` compiles the user's `foo.c`. The `melt.so` plugin initializes the MELT runtime, hence itself `dlopen`-s MELT modules like `warmelt*.so` & `xtramelt*.so`. These modules initialize MELT data, e.g., classes, instances, closures, and handlers. The MELT handler associated to the `opengpu` mode registers a new GCC pass (available in `xtramelt-opengpu.melt`) which is executed by the GCC pass manager when compiling the file `foo.c`. This `opengpu` pass uses Graphite [27] to find optimization opportunities in loops and should [13] generate OpenCL code to run these on GPUs, transforming the Gimple to call that generated OpenCL code. The `melt.so` plugin is mostly hand-coded in C (in our `melt-runtime.[hc]` files - 15KLOC, which `#include` generated files). The MELT modules `warmelt*.so` & `xtramelt*.so` [14] are coded in MELT (as source files `warmelt*.melt`, ..., `xtramelt*.melt` which have been translated by MELT into generated C files `warmelt*.c` & `xtramelt-*.c`, themselves compiled into modules `warmelt*.so` ...).

The MELT translator (able to generate `*.c` from `*.melt`) is *bootstrapped* so that it exercises most of its features and its runtime : the translator's source code is coded in MELT, precisely the `melt/warmelt*.melt` files (39KLOC), and the MELT *source* repository also contains the *generated* files `melt/generated/warmelt*.c` (769KLOC). Other MELT files, like `melt/xtramelt*.melt` (6KLOC) don't need to have their generated translation kept. The MELT translator[15] is *not* a GCC front-end (since it produces C code for the host system, not *Generic* or *Gimple* internal representations suited for the target machine); and it is even able to dynamically generate, during an GCC$^{\text{melt}}$ compiler invocation, some temporary `*.c` code, run `make` to compile that into a temporary `*.so`, and load (i.e. `dlopen`) and execute that - all this in a single `gcc` user invocation; this can be useful for sophisticated static analysis [25] specialized using partial evaluation techniques within the analyzer, or just to "run" a MELT file.

The MELT translator works in several steps: the reader builds s-expressions in MELT heap. Macro-expansion translates them into a MELT AST. Normalization introduces necessary temporaries and builds a normal form. Generation makes a representation very close to *C* code. At last that representation is emitted to output generated *C* code. There is no optimization done by the MELT translator (except for compilation of pattern matching, see §4.4).

Translation from MELT code to C code is fast: on a x86-64 GNU/Linux desktop system[16], the 6.5KLOC `warmelt-normal.melt` file is translated into five `warmelt-normal*.c` files with a total of 239KLOC in just one second (wall time). But 32 seconds are needed to build the `warmelt-normal.so` module (with `make`[17] running `gcc -O1 -fPIC`) from these generated C files. So most of the time is spent in compiling the generated C code, not in generating it. In contrast to several DSLs persisting their

[13] In April 2011, the `opengpu` pass, coded in MELT, is still incomplete in MELT 0.7 svn rev.173182.

[14] The module names `warmelt*.so` & `xtramelt*.so` are somehow indirectly hard-coded in `melt-runtime.c` but could be overloaded by many explicit `-fplugin-arg-melt-*` options.

[15] The translation from file `ana-simple.melt` to `ana-simple.c` is done by invoking `gcc -fplugin=melt -fplugin-arg-melt-mode=translatefile -fplugin-arg-melt-arg=ana-simple.melt` ...on an *empty* C file `empty.c`, only to have `cc1` launched by `gcc`!

[16] An Intel Q9550 @ 2.83GHz, 8Gb RAM, fast 10KRPM Sata 150Gb disk, Debian/Sid/AMD64.

[17] So it helps to run that in parallel using `make -j`; the 32 seconds timing is a sequential single-job `make`.

```
melt_ptr_t meltgc_new_int (meltobject_ptr_t discr_p, long num) {
  MELT_ENTERFRAME (2, NULL);
#define newintv meltfram__.mcfr_varptr[0]
#define discrv  meltfram__.mcfr_varptr[1]
  discrv = (void *) discr_p;
  if (melt_magic_discr ((melt_ptr_t) (discrv)) != MELTOBMAG_OBJECT)
    goto end;
  if (((meltobject_ptr_t)discrv)->obj_num != MELTOBMAG_INT)
    goto end;
  newintv = meltgc_allocate (sizeof (struct meltint_st), 0);
  ((struct meltint_st*)newintv)->discr = (meltobject_ptr_t)discrv;
  ((struct meltint_st*)newintv)->val = num;
end:
  MELT_EXITFRAME ();
  return (melt_ptr_t) newintv;
}
```

Figure 2: MELT runtime function boxing an integer

closures[18] by serializing a mixture of data and code, MELT starts with an empty heap, so MELT modules' initialization routines are mostly long and sequential C code initializing the MELT heap.

2.2 MELT runtime infrastructure

The MELT runtime `melt-runtime.c` is built above the GCC infrastructure, notably Gg-c. However, Gg-c is not a sufficient garbage collector for MELT values, like closures, lists, tuples, objects, ... As in most applicative or functional languages, MELT code tends to allocate a lot of temporary values (which often die quickly). So garbage collection (G-C) of MELT values may happen often, and does need to happen even inside GCC passes written in MELT, not only between passes. These values are handled by our *generational copying* MELT G-C, triggered by the MELT allocator when its birth region is full, and backed up by the existing Gg-c (so the old generation of MELT G-C is the Gg-c heap). Generational copying GCs [11] handle quickly dead young temporary values by discarding them at once after having copied each live young value out of the birth region, but require a scan of all local variables, need to forward pointers to moved values, a write barrier, and *normalization* (like the *administrative normal form* in [7]) of explicit intermediate values inside calls[19]. This is awkward in hand-written C code but easy to generate. Minor MELT G-Cs are triggered before each call to `gcc_collect` (i.e. to the full Gg-c) to ensure that all live young MELT values have migrated to the old Gg-c heap. Compatibility between our MELT GC and Gg-c is thus achieved. An array of more than a hundred *predefined* values contains the only "global" MELT values (which are global roots for both the MELT GC and Gg-c).

MELT call frames are aggregated as local `struct`-ures, containing local MELT values, the currently called MELT closure, and local stuff (like raw `tree` pointers, etc.). Values inside these call frames are known to the MELT garbage collector, which scans them and possibly moves them. Expliciting these call frames facilitates *introspective runtime reflection* [18, 19, 20] at the MELT level; this might be useful for some future sophisticated analysis, e.g., in abstract interpretation [2, 3] of recursive functions, as a widening strategy. Concretely, local MELT values (and stuff) are aggregated in MELT call frames (represented as generated *C* local `struct`-ures) organized in a single-linked list. This also enables the display of the MELT backtrace stack on errors.

[18] Ocaml bytecode contains both code and data; GNU { Emacs, CLisp, Smalltalk } persist their entire heap image. But MELT has no persistent data files, to avoid serializing GCC's stuff (ie GCC's native data).

[19] That is, $f(g(x),y)$ should be normalized as $\tau = g(x); f(\tau,y)$ with τ being a fresh temporary.

```
struct {
  int mcfr_nbvar;                          /* number of MELT local values*/
  const char *mcfr_flocs;                  /* location string for debugging*/
  struct meltclosure_st *mcfr_clos;        /* current closure*/
  struct melt_callframe_st *mcfr_prev;     /* link to previous MELT frame */
  void *mcfr_varptr[2];                    /* local MELT values */
} meltfram__;        /* MELT current call frame */
static char locbuf_1591[84];        /* location string */
if (!locbuf_1591[0])
   snprintf (locbuf_1591, sizeof (locbuf_1591) - 1, "%s:%d",  basename ("gcc/melt-runtime.c"), (int) 1591);
memset (&meltfram__, 0, sizeof (meltfram__));
meltfram__.mcfr_nbvar = (2);
meltfram__.mcfr_flocs = locbuf_1591;
meltfram__.mcfr_prev = (struct melt_callframe_st *) melt_topframe;
meltfram__.mcfr_clos = (((void *) 0));
melt_topframe = ((struct melt_callframe_st *) &meltfram__);
```

Figure 3: C preprocessor expansion of `MELT_ENTERFRAME(2, NULL)` at line 1591

The figure 2 gives an example of hand-written code following MELT conventions (a function `meltgc_new_int` boxing an integer into a value of given dicriminant and number to be boxed). It uses the `MELT_ENTERFRAME` macro[20], which is expanded by the *C* preprocessor into the code in figure 3, which declares and initialize the MELT call frame `meltfram__`. The `MELT_EXITFRAME ()` macro occurrence is expanded into `melt_topframe = (struct melt_callframe_st *) (meltfram__.mcfr_prev);` to pop the current MELT frame. MELT provides a GCC pass checking some of MELT coding conventions in the hand-written part of the MELT runtime.

The MELT runtime depends deeply upon Gg-c, but does not depend much on the details of GCC's main data structures like e.g., `tree` or `gimple` or `loop` : our `melt-runtime.c` can usually be recompiled without changes when GCC's file `gimple.h` or `tree.h` changes, or when passes are changed or added in GCC's core. The MELT translator files `warmelt*.melt` (and the generated `warmelt*.c` files) don't depend really on GCC data structures like `gimple`. As a case in point, *the major "gimple to tuple" transition* [21] in `gcc-4.4`, which impacted a lot of GCC files, *was smoothly handled* within the MELT translator.

The MELT files which are actually processing GCC internal representations (like our `xtramelt-*.melt` or user MELT code), that is MELT code implementing new GCC passes, have to change only when the GCC API changes - exactly like other GCC passes. Often, since the change is compatible with existing code, these MELT files don't have to be changed at all (but should be recompiled into modules).

MELT handles two kinds of ***thing***s: the first-class MELT *value*s (allocated and managed in MELT's GC-ed heap) and other *stuff*, which are any other GCC data managed in C (either generated or hand-written C code within GCC$^{\mathsf{melt}}$). Informally, $Things = Values \cup Stuff$. So raw `long`-s, `edge`-s or `tree`-s are *stuff*, and appear exactly in MELT memory like C-coded GCC passes handle them (without extra boxing). Variables and [sub-]expressions in MELT code, hence locals in MELT call frames, can be *thing*s of either kind (values or *stuff*).

Since Gg-c requires each pointer to be of a `gengtype`- known type, values are really different from

[20] The *Ocaml* runtime has similar macros.

[21] In the old days of GCC version 4.3 the *Gimple* representation was physically implemented in `tree`-s and the C data structure `gimple` did not exist yet; at that time, *Gimple* was sharing the same physical structures as *Tree*s and *Generic* [so *Gimple* was mostly a conventional restriction on *Tree*s] - that is using many linked lists. The 4.4 release added the `gimple` structure to represent them, using arrays, not lists, for sibling nodes; this improved significantly GCC's performance but required patching many files.

stuff. There is unfortunately *no way to implement full polymorphism* in MELT: we cannot have MELT tuples containing a mix of raw `tree`-s and MELT objects (even if both are Gg-c managed pointers). This Gg-c limitation has deep consequences in the MELT language (*stuff*, i.e. GCC native data, sadly cannot be first-class MELT values!).

Some parts of the MELT runtime are generated (by a special MELT mode). Various MELT values' and *stuff* implementation are described by MELT instances. So adding extra types of values, or interfacing additional GCC *stuff* to MELT, is fairly simple, but requires a complete re-building of MELT. Their `GTY((...)) struct`-ure declarations in *C* are generated. Lower parts of the MELT runtime (allocating, forwarding, scanning routines - see chapters 6 & 7 of [11] - for the copying MELT G-C, hash-tables implementation, . . .) are also generated. This generated *C* code is kept in the source repository.

Notice that the distinction between first-class MELT *value*s and plain *stuff* is essential in MELT, and is required by current GCC practices (notably its Gg-c collector). Therefore, the MELT language itself needs to denote them separately and explicitly, and the MELT runtime (and generated code) handles them differently. In that respect, MELT is *not* like Lisp, Scheme, Guile, Lua and Python. However, MELT coders should usually prefer handling values (the "first class citizens"), not raw *stuff*.

2.3 MELT debugging aids

When generating non-trivial *C* code, it is important to lower the risk of crashing the generated code [22]. This is achieved by systematically clearing all data (both values and raw *stuff*) to avoid uninitialized pointers (and MELT G-C also requires that), and by carefully coding low-level operations (primitives §3.4.2, c-matchers §4.3, code chunks §3.4.1) with tests against null pointers.

The generated *C* code produced by the MELT translator contains many `#line` directives (suitably wrapped with `#ifdef`). In the rare cases when the gdb debugger needs to be used on MELT code (e.g., to deal with crashes or infinite loops), it will refer correctly to the originating MELT source file location. These positions are also written into MELT call frames, to ease backtracing on error.

MELT uses debug printing and assertions quite extensively. If enabled by the `-fplugin-arg-melt-debug` program argument to `gcc`, a lot of debug printing happens : each use of the `debug_msg` operation displays the current MELT source location, a message, and a value [23]. For debugging *stuff* data, primitives `debugtree`, `debuggimple`, etc. are available. Assertions are provided by `assert_msg` which takes a message and a condition to check. When the check fails, the entire MELT call stack is printed (with positions referring to `*.melt` source files).

When variadic functions will be available in MELT, their first use will support polymorphic debug printing. A `debug` "macro" would be expanded into calls to a `debug_at` variading function, which would get the source location value as its first argument, and the values or *stuff* to be debug-printed as secondary variadic arguments.

An older version of MELT could be used with an external probe, which was a graphical program interacting with `cc1` through asynchronous textual protocols. This approach required a quite invasive patch of GCC's code itself. The current GCC pass manager and plugin machinery now provides enough hooks, and future versions of MELT might communicate asynchronously with a central monitor (to be developed).

[22] However, it is still possible to make some MELT code crash, for instance by adding bugs in the *C* form of our code chunks §3.4.1. In practice, MELT code crashes very rarely; most often it fails by breaking some assertions.

[23] Values are printed for debug use with MELT message passing through the `DBG_OUTPUT` & `DBG_OUTPUTAGAIN` selectors.

3 The MELT language and its pecularities

Some familiarity with a Lisp-like language (like Emacs Lisp, Scheme, Common Lisp, etc.) is welcome to understand this section. Acquaintance with a dynamically typed scripting language like Python, Guile or Ruby could also help. See the web site `gcc-melt.org` for more material (notably tutorials) on MELT.

MELT has a Lisp-like syntax because it was (at its very beginning) implemented with an initial "external" MELT to C translator prototyped in Common Lisp. Since then, a lot of newer features have been progressively added (using an older version of MELT to bootstrap its current version). The Emacs Lisp language (in the Emacs editor), Guile (the Gnu implementation of *Scheme*), and *machine description* files in GCC back-end are successful examples of other Lisp dialects within Gnu software. Finally, existing editing modes[24] for Lisp are sufficient for MELT.

An alternative infix syntax (code-named Milt) for MELT is in the works; the idea is to have an infix parser, coded in MELT, for future `*.milt` files, which is parsed into MELT internal s-expressions (i.e. into the same instances of CLASS_SEXPR as the MELT Lisp-like reader does): symbols starting with `+` or `-` are parsed as infix operators (like Ocaml does) with additive precedences, those starting with `*` or `/` have multiplicative precedence, etc.

MELT shares with existing Lisp languages many syntactic and lexical conventions for comments, indentation, symbols (which may be non alpha-numerical), case-insensitivity, and a lot of syntax (like `if`, `let`, `letrec`, `defun`, `cond` ...). As in all Lisp dialects, everything is parenthesized like `(` *operator operands* ... `)` so parenthesis are highly significant. The quote, back-quote, comma and question mark characters have special significance, so `'a` is parsed exactly as `(quote a)`, `?b` as `(question b)` etc. Like in Common Lisp, words prefixed with a colon like `:long` are considered as "keywords" and are not subject to evaluation. Symbols and keywords exist both in source files and in the running MELT heap.

3.1 MELT macro-strings

Since "mixing" C code chunks (§3.4.1) inside MELT code is very important, simple meta-programming is implemented by a lexical trick [25]: *macro-strings* are strings prefixed with `#{` and suffixed with `}#` and are parsed specially; these prefix and suffix strings have been chosen because they usually don't appear in *C* code. Within a macro-string, backslash does not escape characters, but `$` and sometimes `#` are scanned specially, to parse symbols inside macro-strings.

For example, MELT reads the macro-string `#{/*$P#A*/printf("a=%ld\n", $A);}#` exactly as a list `("/*" p "A*/printf(\"a=%ld\\n\", " a ");")` of 5 elements whose 1st, 3rd and 5th elements are strings[26] and 2nd and 4th elements are symbols `p` and `a`. This is useful when one wants to mix C code inside MELT code; some macro-strings are several dozens of lines long, but don't need any extra escapes (as would be required by using plain strings).

Another example of *macro-string* is given in the following "hello-world" (complete) MELT program:

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

[24] Emacs mode for Lisp is nearly enough for editing, highlighting and indenting MELT code.

[25] Inspired by handling of `$` in strings or "here-documents" by shells, Perl, Ruby, ...

[26] The first string has the two characters `/*` and the last has the two characters `);`

The macro-string spans on 3 lines, and contains some *C* code with the `helloworldchunk` MELT symbol. The above `helloworld.melt` file (of 4 lines) is translated into a `helloworld.c` file (of 389 lines[27] in *C*). It uses the `code_chunk` construct explained in §3.4.1 below (to emit translated *C* code).

3.2 MELT values and stuff

Every MELT *value* has a *discriminant* (at the start of the memory zone containing that value). As an exception, nil [28], represented by the C null pointer has conventionally a specific discriminant `DISCR_NULL_RECIEVER`. The discriminant of a value is used by the MELT runtime, by Gg-c and in MELT code to separate them. MELT values can be boxed *stuff* (e.g., boxed `long` or boxed `tree`), closures, lists, pairs, tuples, boxed strings, ..., and MELT *objects*. Several *predefined objects*, e.g., `CLASS_CLASS`, `DISCR_NULL_RECEIVER` ..., are required by the MELT runtime. The hierarchy of discriminants is rooted at `DISCR_ANY_RECEIVER` [29]. Discriminants are objects (of `CLASS_DISCRIMINANT`). Core classes and discriminants are predefined as MELT values (known by both Gg-c and MELT G-C).

Each MELT object has its class as its discriminant. Classes are themselves objects and are organized in a single-inheritance hierarchy rooted at `CLASS_ROOT` (whose parent discriminant is `DISCR_ANY_RECIEVER`). Objects are represented in C as exactly a structure with its class (i.e. discriminant) `obj_class`, its unsigned hash-code `obj_hash` (initialized once and for all), an unsigned "magic" short number `obj_num`, the unsigned short number of fields `obj_len`, and the `obj_vartab[obj_len]` array of fields, which are MELT values. The `obj_num` in objects can be set *at most once* to a *non-zero* unsigned short, and may be used as a *tag*: MELT and Gg-c discriminate quickly a value's data-type (for marking, scanning and other purposes) through the `obj_num` of their discriminant. So, safely testing in *C* if a value `p` is a MELT closure is as fast as `p != NULL && p->discr->obj_num == MELTOBMAG_CLOSURE`.

MELT field descriptors and method selectors are objects. Every MELT *value* (object or not, even nil) can be sent a message, since its discriminant (i.e., its class, if it is an object) has a method map (a hash table associating selectors to method bodies) and a parent discriminant (or super-class). Message passing in MELT is similar to those in Smalltalk and Ruby. Method bodies can be dynamically installed with `(install_method` *discriminant selector function*`)` and removed at any time in any discriminant or class. Method invocations use the method hash-maps (similar to methods' dictionnaries in Smalltalk) to find the actual method to run.

The MELT reader produces mostly objects and sometimes other values: S-expressions are parsed as instances of `CLASS_SEXPR` (containing the expression's source location and the list of its components); symbols (like == or `let` or `x`) as instances of `CLASS_SYMBOL`; keywords like `:long` or `:else` as instances of `CLASS_KEYWORD`; numbers like `-1` as values of `DISCR_INTEGER` etc.

Each *stuff* (that is, non-value *thing*s like `long` or `tree` ...) have its boxed value counterpart, so boxed gimple-s are values containing, in addition of their discriminant (like `DISCR_GIMPLE`), a raw `gimple` pointer.

In MELT expressions, literal integers like `23` or strings like `"hello\n"` refer to raw `:long` or `:cstring` *stuff* [30], not constant values. To be considered as MELT values they need to be quoted, so (contrarily to other Lisps) in MELT `2` $\not\equiv$ `'2` : the plain `2` denotes a raw stuff of c-type `:long` so is not a value, but the

[27] With 260 lines of code, including 111 preprocessor directives, mostly `#line`, and 129 comment or blank lines, and all the code doing "initialization".

[28] As in Common Lisp or Emacs Lisp (or C itself), but not as in Scheme, MELT nil value is considered as false, and every non-nil value is true.

[29] `DISCR_ANY_RECEIVER` is rarely used, e.g., to install catch-all method handlers.

[30] All `:cstring` are `(const char*)` C-strings in the text segment of the executable, so they are not `malloc`-ed.

quoted expression '2 denotes the boxed integer 2 constant value of DISCR_CONSTANT_INTEGER so they are not equivalent! As in Lisp, a quoted symbol like 'j denotes a constant value (of CLASS_SYMBOL).

To associate *thing*s (either MELT objects or GCC *stuff*, all of the same type) to MELT values, hash-maps are extensively used: so homogenous hash tables keyed by objects, raw strings, or raw *stuff* like tree-s or gimple-s ... are values (of discriminant DISCR_MAP_OBJECTS ..., DISCR_MAP_TREES). While hash-maps are more costly than direct fields in structures to associate some data to these structures, they have the important benefit of avoiding disturbing existing data structures of GCC. And even C plugins of GCC cannot add for their own convenience extra fields into the carefully tuned tree or gimple structures of GCC's tree.h or gimple.h.

Aggregate MELT values include not only objects, hash-tables and pairs, but also tuples (a value containing a fixed number of immutable component values), closures, lists, ... Lists know their first and last pairs. Aggregate values of the same kind may have various discriminants. For instance, within a MELT class (which is itself a MELT object of CLASS_CLASS) a field gives the tuple of all super-classes starting with CLASS_ROOT. That tuple has DISCR_CLASS_SEQUENCE as discriminant, while most other tuples have DISCR_MULTIPLE as discriminant.

Decaying values may help algorithms using memoization; they contain a value reference and a counter, decremented at each major garbage collection. When the counter reaches 0, the reference is cleared to nil.

Adding a new important GCC C type like gimple [31] for some new stuff is fairly simple: add (in MELT code) a new predefined C-type descriptor (like CTYPE_GIMPLE referring to keyword :gimple) and additional discriminants, and regenerate all of MELT. C-type descriptors (e.g., CTYPE_EDGE) and value type descriptors (like VALDESC_LIST) contains dozen[s] of fields (names or body chunk of generated *C* routines) used when generating the runtime support routines.

The :void keyword (and so CTYPE_VOID) is used for side-effecting code without results. C-type keywords (like :void, :long, :tree, :value, :gimple, :gimple_seq, etc.) qualify (in MELT source code) formal arguments, local variables (bound by let, ...), etc.

MELT is typed for *thing*s: e.g., the translator complains if the +i primitive addition operator (expecting two raw :long *stuff* and giving a :long result) is given a value or a :tree argument. Furthermore, let bindings can be explicitly typed (by default they bind a value). Within values, typing is dynamic; for instance, a value is checked at runtime to be a closure before being applied. When applying a MELT closure to arguments, the first argument, if any, needs to be a value (it would be the receiver if the closure is a method for message passing)[32], others can be *thing*s, i.e. values or *stuff*. In MELT applications, the types of secondary arguments and secondary results are described by constant byte strings, and the secondary arguments or results are passed (in generated *C* code) as an array of unions. The generated MELT function prologue (in *C*) checks that the formal and actual type of secondary arguments are the same (otherwise, argument passing stops, and all following actual arguments are cleared).

All MELT *thing*s (value or *stuff*), in particular local variables (or mismatched formals), are initially cleared (usually by zeroing the whole MELT call frame in the *C* prologue of each generated routine). So MELT values are initially () (i.e., nil in MELT syntax), a :tree *stuff* is initially the null tree (i.e. (tree)0 in *C* syntax), a :long *stuff* is initially 0L, a :cstring *stuff* is initialized to (const char*)0. Notice that cleared *stuff* is considered as false in conditional context.

[31] This kind of radical addition don't happen often in the GCC community because it usually impacts a lot of GCC files.

[32] The somehow arbitrary requirement of having the first argument of every MELT function be a value speeds up calls to functions with one single value argument, and permits using closures as methods without checks: sending a message to a raw *stuff* like e.g., a tree won't work.

Functions written in MELT (with `defun` for named functions or `lambda` for anonymous ones) always return a value as their primary result (which may be ignored by the caller, and defaults to nil). The first formal argument (if any) and the primary result of MELT functions should be values (so nested function calls deal mainly with values). Secondary arguments and results can be any *thing*s (each one is either a value or some *stuff*). The `(multicall ...)` syntax binds primary and secondary results like Common Lisp's `multiple-value-bind`.

3.3 Syntax overview

The following constructs should be familiar (except the last one, `match`, for pattern matching) since they look like in other Lisps. Notice that our `let` is always sequential[33]. Formals in abstractions [34] are restricted to start with a formal value; this speeds up the common case of functions with a single value argument, and facilitates installation of any function as method (without checking that the formal reciever is indeed a value).

List of formal arguments (in `lambda`, `defun` etc.) contains either symbols (which are names of formals bound by e.g., the `lambda`) like `x` or `discr`, or c-type keywords like `:value` or `:long` or `:gimple` .... A c-type keyword qualify all successing formals up to the next c-type keywords, and the default c-type is `:value`. For example, the formal arguments list `(x y :long n k :gimple g :value v)` have 6 formals : `x y v` are MELT values, `n k` are raw long *stuff*, `g` is a raw gimple *stuff*.

Local bindings (in `let` or `letrec`) has an optional c-type annotation, then the newly bound symbol, then the sub-expression bounding it. So `(:long x 2)` locally binds (in the body of the enclosing `let`) the symbol `x` to the raw long *stuff* 2, and in the `let` body `x` is a raw long variable.

Patterns and pattern matching are explained in §4.

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

application	(ϕ α_1 ... α_n)	apply function (or primitive) ϕ to arguments α_i
assignment	(`setq` ν ε)	set local variable ν to ε
message passing	(σ ρ α_1 ... α_n)	send selector σ to reciever ρ with arguments α_i
let expression	(`let` (β_1...β_n) ε_1...ε_p ε')	with local **sequential** bindings β_i evaluate side-effecting sub-expressions ε_j and give result of ε'
sequence	(`progn` ε_1...ε_n ε')	evaluate ε_i (for their side effects) and at last ε', giving its result (like the operator `,` in C)
abstraction	(`lambda` ϕ ε_1...ε_n ε')	anonymous function with formals ϕ and side-effecting expressions ε_i, return result of ε'
pattern matching	(`match` ε χ_1 ... χ_n)	match result of ε against match clauses χ_i, giving result of last expression of matched clause.

Conditional expressions alter control flow as usual. However, conditions can be *thing*s, e.g., the `0 :long` *stuff* is false, other long *stuff* are true, a `gimple` *stuff* is false iff it is the null gimple pointer, etc. The "else" part ε of an `if` test is optional. When missing, it is false, that is a cleared *thing*. Notice that tested conditions and the result of a conditional expression can be either values or raw stuff, but all the conditional sub-expressions of a condition should have consistent types, otherwise the entire expression has `:void` type.

---

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

[34]Notice that `lambda` abstractions are constructive expressions and may appear in `letrec` or `let` bindings.

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
conjunction	`(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' otherwise the cleared *thing* of same type
disjunction	`(or` δ_1 ... δ_n`)`	δ_1 or else δ_2 ... = the first of the δ_i which is "true" (non nil, or non zero, ...)

In a `cond` expression, every condition κ_i (except perhaps the last) is like `(`γ_i $\varepsilon_{i,1}$ ... ε_{i,p_i} ε'`)` with $p_i \geq 0$. The first such condition for which γ_i is "true" gets its sub-expressions $\varepsilon_{i,j}$ evaluated sequentially for their side-effects and gives the result of 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 the result of ε'

MELT has some more expressions.

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 $\varepsilon_1 ... \varepsilon_n$ 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 (`$\beta_1 ... \beta_n$`)` $\varepsilon_1 ... \varepsilon_p$`)`	with [mutually-] recursive *constructive* bindings β_i evaluate sub-expressions ε_j
field access	`(get_field :`Φ ε`)`	if ε gives an appropriate object retrieves its field Φ, otherwise nil
unsafe field access	`(unsafe_get_field :`Φ ε`)`	unsafe access without check like the above operation
object update	`(put_fields` ε `:`Φ_1 ε_1 ... `:`Φ_n ε_n`)`	safely update (if appropriate) in the object given by ε each field Φ_i with ε_i
unsafe object update	`(unsafe_put_fields` ε `:`Φ_1 ε_1 ...`)`	unsafely update the object given by ε

The unsafe field access `unsafe_get_field` is reserved to expert MELT programmers, since it may crash. The safer variant test that the expression ε evaluates[35] to a MELT object of appropriate class before accessing a field Φ in it. Field updates with `put_fields` are safe [36], with an unsafe but quicker variant `unsafe_put_fields` available for MELT experts.

Mutually recursive `letrec` bindings should have only constructive expressions.

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

[35] I.e. test if the value ω of ε is an object which is a direct or indirect instance of the class defining field Φ, otherwise a nil value is given.

[36] Update object ω, value of ε, only if it is an object which is a direct or indirect instance of the class defining each field Φ_i

Of course `lambda` expressions are also constructive and can appear inside `letrec`. Notice that since MELT is translated into C, and because of runtime constraints, MELT recursion is never handled tail-recursively so always consume stack space. This also motivates iterative constructions (like `forever` and our iterators).

Name defining expressions have a syntax starting with `def`. Most of them (except `defun`, `defclass`, `definstance`) have no equivalent in other languages, because they define bindings related to C code generation. For the MELT translator, bindings have various kinds; each binding kind is implemented as some subclass of `CLASS_ANY_BINDING`.

Name exporting expressions are essentially directives for the module system of MELT. Only exported names are visible outside a module. A module initialization expects a parent environment and produces a newer environment containing exported bindings. Both name defining and exporting expressions are supposed to appear only at the top-level (and should not be nested inside other MELT expressions).

	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 classes	(`defclass` ν `:super` σ `:fields` ($\phi_1 \ldots \phi_n$))	define class ν of super-class σ and own fields ϕ_i
for instances	(`definstance` ι κ `:`f_1 ε_1 ... `:`f_n ε_n)	define an instance ι of class κ with each field f_i initialized to the value of ε_i
for selectors	(`defselector` σ κ [`:formals` Ψ] `:`f_1 ε_1 ... `:`f_n ε_n)	define an selector ι of class κ (usually `CLASS_SELECTOR`) with each extra field f_i initialized to the value of ε_i (usually no extra fields are given so $n = 0$) and with optional formals Ψ
for primitives	(`defprimitive` ν ϕ `:`θ η)	define primitive ν with formal arguments ϕ, 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 values (e.g., of functions, objects, matcher, selector, ...)
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_i and all their own fields (as value bindings)
as synonym	(`export_synonym` ν ν')	export the new name ν as a synonym of the existing name ν'

Macro-expansion is internally the first step of MELT translation to C: parsed (or in-heap) S-exprs (of `CLASS_SEXPR`) are macro-expanded into a MELT “abstract syntax tree” (a subclass of `CLASS_SOURCE`). This macro machinery is extensively used, e.g., `let` and `if` constructs are macro-expanded (to instances of `CLASS_SOURCE_LET` or `CLASS_SOURCE_IF` respectively.

Field names and class names are supposed to be globally unique, to enable checking their access or update. Conventionally class names start with `CLASS_` and field names usually share a common unique prefix in their class. There is no protection (i.e. visibility restriction like `private` in C++) for accessing a field.

All definitions accept documentation annotation using `:doc`, and a documentation generator mode produces documentation with-cross references in Texinfo format.

Miscellanous constructs are available, to help in debugging or coding or to generate various C code depending on compile-time conditions.

expressions for debugging

debug message	`(debug_msg` ε μ`)`	debug printing message μ & value ε
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 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 $\boxed{\text{code for } \varepsilon}$ `#else` $\boxed{\text{code for } \varepsilon'}$ `#endif`
Version test	`(gccif` β ε_1 ...`)`	the ε_i are translated only if GCC has version prefix string β

Reflective access to the current and parent environment is possible (but useful in exceptional cases, since *`export_`*... directives are available to extend the current exported environment):

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

3.4 Linguistic constructs to fit MELT into GCC

Several language constructs are available to help fit MELT into GCC, taking advantage of MELT and GCC runtime infrastructure (notably Gg-c). They usually use macro-strings to provide C code with holes. Code chunks (§3.4.1) simply permit to insert C code in MELT code. Higher-level constructs describe how to translate other MELT expressions into C: primitives (§3.4.2) describe how to translate low-level operations into C; c-iterators (§3.4.3) define how iterative expressions are translated into `for`-like loops; c-matchers (§4.3) define how to generate simple patterns (for matching), etc.

3.4.1 Code chunks

Code chunks are simple MELT templates (of `:void` c-type) for generated *C* code. They are the lowest possible way of impacting MELT *C* code generation, so are seldom used in MELT (like **asm** is rarely used in *C*).

As a trivial example where `i` is a MELT`:long` variable bound in an enclosing `let`,

```
(code_chunk sta
  #{$sta#_lab: printf("i=%ld\n", $i++); goto $sta#_lab; }# )
```

would be translated to

```
{sta_1_lab: printf("i=%ld\n", curfnum[3]++); goto sta_1_lab;}
```

the first time it translated (`i` becoming `curfnum[3]` in C), but would use `sta_2_lab` the second time, etc. The first argument of `code_chunk` - `sta` here - is a *state* symbol, expanded to a C identifier unique to the code chunk's translation. The second argument is the macro-string serving as template to the generated C code. The state symbol is uniquely expanded, and other symbols should be MELT variables and are replaced by their translation. So the `code_chunk` of state symbol `helloworldchunk` in §3.1 is translated into the following *C* code:

```
  int i=0; /* our HELLOWORLDCHUNK__1 */
     HELLOWORLDCHUNK__1_label: printf("hello world from MELT\n");
     if (i++ < 3) goto HELLOWORLDCHUNK__1_label; ;
```

3.4.2 Primitives

Primitives define a MELT operator by its C expansion. The unary negation `negi` is defined exactly as :

```
(defprimitive negi (:long i)  :long
  :doc #{Integer unary negation of $i.}#
  #{(-($i))}#  )
```

Here we specify that the formal argument `i` is, like the result of `negi`, a `:long` stuff. We give an optional documentation, followed by the macro-string for the C expansion. Primitives don't have state variables but are subject to normalization[37] and type checking. During expansion, the formals appearing in the primitive definition are replaced appropriately.

3.4.3 C-iterators

A MELT *c-iterator* is an operator translated into a `for`-like C loop. The GCC compiler defines many constructs similar to C `for` loops, usually with a mixture of macros and/or trivial inlined functions. C-iterators are needed in MELT because the GCC API defines many iterative conventions. For example, to iterate on every `gimple` *g* inside a given `gimple_seq` *s* GCC mandates (see §1.1) the use of a `gimple_simple_iterator`.

In MELT, to iterate on the `:gimpleseq s` obtained by the expression σ and do something on every `:gimple g` inside `s`, we can simply code `(let ( (:gimpleseq s σ) ) (each_in_gimpleseq (s) (:gimple g)` ⌈do something with g...⌋`))` by invoking the *c-iterator* `each_in_gimpleseq`, with a list of inputs - here simply `(s)` - and a list of local formals - here `(:gimple g)` - as the iterated *thing*s.

This c-iterator (a template for such **for**-like loops) is defined exactly as:

```
(defciterator each_in_gimpleseq
  (:gimpleseq gseq)                    ;start formals
  eachgimplseq                         ;state
  (:gimple g)                          ;local formals
  #{/* start $eachgimplseq: */
   gimple_stmt_iterator gsi_$eachgimplseq;
   if ($gseq) for (gsi_$eachgimplseq = gsi_start ($gseq);
                   !gsi_end_p (gsi_$eachgimplseq);
                   gsi_next (&gsi_$eachgimplseq)) {
    $g  = gsi_stmt (gsi_$eachgimplseq);   }#
  #{ } /* end $eachgimplseq*/  }#)
```

We give the start formals, state symbol, local formals and the "before" and "after" expansion of the generated loop block. The expansion of the body of the invocation goes between the before and after expansions. C-iterator occurrences are also normalized (like primitive occurrences are). MELT expressions using c-iterators give a `:void` result, since they are used only for their side effects.

3.5 Modules, environments, standard library and hooks

A single `*.melt` source file[38] is translated into a single module loaded by the MELT run-time. The module's generated `start_module_melt` routine [often quite big] takes a parent environment, executes the top-level forms, and finally returns the newly created module's environment. Environments and their bindings are reified as objects.

[37] Assuming that `x` is a MELT variable for a `:long` stuff, then the expression `(+i (negi x) 1)` is normalized as `let` $\alpha = -x, \beta = \alpha + 1$ `in` β in pseudo-code - suitably represented inside MELT (where α, β are fresh gensym-ed variables).

[38] MELT can also translate into C a sequence of S-expressions from memory, and then dynamically load the corresponding temporary module after it has been C-compiled.

Only exported names add bindings in the module's environment. MELT code can explicitly export defined values (like instances, selectors, functions, c-matchers, ...) using the `(export_values ...)` construct; macros (or pat-macros [that is pattern-macros producing abstract syntax of patterns]) definitions are exported using the `(export_macro ...)` construct or `(export_patmacro ...)`; classes and their own fields are exported using the `(export_class ...)` construct. Macros and pattern macros in MELT are expanded into an abstract syntax tree (made of objects of sub-classes of `CLASS_SOURCE`, e.g., instances of `CLASS_SOURCE_LET` or of `CLASS_SOURCE_APPLY`, ...), not into s-expressions (i.e. objects of `CLASS_SEXPR`, as provided by the reader).

Field names should be *globally* unique: this enables `(get_field :named_name x)` to be safely translated into something like "if `x` is an instance of `CLASS_NAMED` fetch its `:named_name` field otherwise give nil", since MELT knows that `named_name` is a field of `CLASS_NAMED`.

As in C, there is only one name-space in MELT which is technically, like Scheme, a Lisp_1 dialect[39] (in Queinnec's terminology [22]). This prompts a few naming conventions: most exported names of a module share a common prefix; most field names of a given class share the same prefix unique to the class, etc.

The entire MELT translation process [26] is implemented through many exported definitions which can be used by expert MELT users to customize the MELT language to suit their needs. Language constructs [40] give total access to environments (instances of `CLASS_ENVIRONMENT`).

Hooks for changing GCC's behavior are provided on top of the existing GCC plugin hooks (for instance, as exported primitives like `install_melt_gcc_pass` which installs a MELT instance describing a GCC pass and registers it inside GCC).

A fairly extensive MELT standard library is available (and is used by the MELT translator), providing many common facilities (map-reduce operations; debug output methods; run-time asserts printing the MELT call stack on failure; translate-time conditionals emitted as `#ifdef`; ...) and interfaces to GCC internals. Its `.texi` documentation is produced by a generator inside the MELT translator.

When GCC will provide additional hooks for plugins, making them available to MELT code should hopefully be quite easy.

4 Pattern matching in MELT

Pattern matching [12, 14, 18, 30] is an essential operation in symbolic processing and formal handling of programs, and is one of the buying features of high-level programming languages (notably Ocaml and Haskell). Several tasks inside GCC are mostly pattern matching (like simplification and folding of constant expressions)[41]. Code using MELT pattern matching facilities is much more concise than its (generated or even hand-written) *C* equivalent.

4.1 Using patterns in MELT

Developers using MELT often need to filter complex GCC *stuff* (in particular `gimple` or `tree`-s) in their GCC passes coded in MELT. This is best achieved with pattern matching. The matching may fail (if the data failed to pass the filter) or may extract information from the matched data.

[39] Each bound name is bound only once, and there are no separate namespaces like in C or Common Lisp.

[40] Like `(current_module_environment_container)` and `(parent_module_environment)`, etc.

[41] Strangely, GCC has several specialized code generators, but none for pattern matching: so the file `gcc/fold-const.c` is hand-written (16KLOC).

4.1.1 About pattern matching

Patterns are major syntactic constructs (like expressions and let-bindings in Scheme or MELT). In MELT, a pattern starts with a question mark, which is parsed particularly: `?x` is the same as `(question x)` [it is the pattern variable *x*]. `?_` is [42] the *wildcard pattern* (matching anything). An expression occurring in pattern context is a *constant* pattern. Patterns may be nested (in composite patterns) and occur in `match` expressions.

Elementary patterns are ultimately translated into code that *tests* that the matched *thing* μ can be filtered by the pattern π followed by code which extracts appropriate data from μ and *fills* some locals with information extracted from μ. Composite patterns need to be translated and optimized to avoid, when possible, repetitive tests or fills.

4.1.2 An example of pattern usage in gcc$^{\text{melt}}$

Many tasks depend upon the form of [some intermediate internal representation of] user source code, and require extracting some of its sub-components. For instance, the author has written (in a single day) a GCC extension in MELT to check simple coding rules in `melt-runtime.c`, (e.g., in function of figure 2). When enabled with `-fplugin-melt-arg-mode=meltframe`, it adds a new pass (after the `"ssa"` pass[43]. of GCC [21]) `melt_frame_pass` to GCC. This pass first finds the declaration of the local `meltfram__` in the following pass execute function:

```
1  (defun meltframe_exec (pass)
2    (let (
3        (:tree tfundecl (cfun_decl))          (:long nbvarptr 0)
4        (:tree tmeltframdecl (null_tree))     (:tree tmeltframtype (null_tree)) )
5      (each_local_decl_cfun ()   (:tree tlocdecl :long ix)
6        (match tlocdecl
7               ( ?(tree_var_decl
8                   ?(and ?tvtyp ?(tree_record_type_with_fields ?tmeltframrecnam ?tmeltframfields))
9                   ?(cstring_same "meltfram__") ?_)
10                (setq tmeltframdecl tlocdecl)    (setq tmeltframtype tvtyp)
11                (foreach_field_in_record_type (tmeltframfields) (:tree tcurfield)
12                  (match tcurfield
13                         ( ?(tree_field_decl
14                             ?(tree_identifier ?(cstring_same "mcfr_varptr"))
15                             ?(tree_array_type ?telemtype
16                                               ?(tree_integer_type_bounded ?tindextype
17                                                                           ?(tree_integer_cst 0)
18                                                                           ?(tree_integer_cst ?lmax)
19                                                                           ?tsize)))
20                            (setq tmeltframvarptr tcurfield)  (setq nbvarptr lmax)))))
21              ( ?_ (void))))
```

The `let` line 2 spans the entire MELT function `meltframe_exec`, with bindings lines 3 & 4 for `tfundecl`, `nbvarptr`, `tmeltframdecl` & `tmeltframtype` locals. The `each_local_decl_cfun` is a c-iterator (iterating -lines 5 to 11- on the *Tree*-s representing the local declarations in the function). The `match` expression filters the current local declaration `tlocdecl` (lines 7-11). When it is a variable declaration (line 7) whose type matches the sub-pattern line 8 and whose name (line 9) is exactly `meltfram__`, we assign (line 10) appropriately `tmeltframdecl` & `tmeltframtype`, and we iterate (line 11) on its fields to find, by the `match` (lines 12-21), the declaration of field `mcfr_varptr` (in the

[42] `?_` can be pronounced as "joker"

[43] `ssa` means Static Single Assignment, so at that stage the code is represented in *Gimple/SSA* form, so each SSA variable is assigned once!

C code), and its array index upper bound `lmax`, assigning them (line 20) to locals `tmeltframvarptr` & `nbvarptr`. Otherwise, using the wildcard pattern `?_`, we give a `:void` result for the match of `tlocdecl` (line 21).

Once the declaration of `meltfram__` and of its `mcfr_varptr` field has been found[44] in the current function (given by `cfun` inside GCC), we iterate on each basic block `bb` of that function, and on each *gimple* statement `g` of that basic block, and we match that statement `g` to find assignments to or from `meltfram__.mcfr_varptr[`κ`]` where κ is some constant integer index:

```
22        (each_bb_cfun  ()   (:basic_block bb  :tree fundecl)
23         (eachgimple_in_basicblock (bb)
24          (:gimple g)
25          (match g
26                 ( ?(gimple_assign_single
27                     ?(tree_array_ref  ?(tree_component_ref tmeltframdecl tmeltframvarptr)
28                       ?(tree_integer_cst ?idst))
29                     ?(tree_array_ref  ?(tree_component_ref  tmeltframdecl  tmeltframvarptr)
30                       ?(tree_integer_cst ?isrc)))
31                   ⌈handle assign ‘‘meltfram__.mcfr_varptr[idst] = meltfram__.mcfr_varptr[isrc];’’⌋)
32                 ( ?(gimple_assign_single
33                     ?(tree_array_ref  ?(tree_component_ref tmeltframdecl tmeltframvarptr)
34                       ?(tree_integer_cst ?idst))
35                     ?rhs)
36                   ⌈handle assign ‘‘meltfram__.mcfr_varptr[idst] = rhs;’’⌋)
37                 ( ?(gimple_assign_single ?lhs
38                     ?(tree_array_ref  ?(tree_component_ref tmeltframdecl tmeltframvarptr)
39                       ?(tree_integer_cst ?isrc)))
40                   ⌈handle assign ‘‘lhs = meltfram__.mcfr_varptr[isrc];’’⌋)
```

The gimple `g` is matched against the most filtering pattern (lines 26-30, for assignments like "`meltfram__.mcfr_varptr[`*idst*`] = meltfram__.mcfr_varptr[`*isrc*`];`") first, then against the more general patterns -for "`meltfram__.mcfr_varptr[`*idst*`] =` *rhs* `;`" where *rhs* is any simple operand- lines 32-36, and for "*lhs* `= meltfram__.mcfr_varptr[`*isrc*`];`" lines 37-40. The MELT programmer should order his matching clauses from the more specific to the more general.

Other code (not shown here) in function `meltframe_exec` remembers all left-hand side and right-hand side occurences of `meltfram__.mcfr_varptr[`κ`]`, and issues a warning when such a slot is not used.

We see that a `match` is made of several match-cases, tested in sequence until a match is found. Each case starts with a pattern, followed by sub-expressions which are computed with the pattern variables of the case set appropriately by the matching of the pattern; the last such sub-expression is the result of the entire `match`. Like other conditional forms in MELT, `match` expressions can give any *thing* (*stuff*, e.g., `:long` ... or even `:void`, or *value*) as their result. Patterns may be nested like the `tree_var_decl` or `tree_record_type` above. All the locals for pattern variables in a given match-case are cleared (before testing the pattern). It is good style to end a `match` with a catch-all wildcard `?_` pattern.

A pattern is usually composite (with nested sub-patterns) and has a double role: first, it should *test* if the matched *thing* fits; second, when it does, it should extract *thing*s and transmit them to eventual sub-patterns; this is the *fill* of the pattern. The matching of a pattern should conventionally be without side-effects (other than the fill, i.e. the assignment of pattern variables).

Patterns may be *non-linear*: in a matching case, the same pattern variable can occur more than once; then it is set at its first occurrence, and tested for *identity*[45] with `==` in the generated *C* code on all

[44] A warning is issued if `meltfram__` or `mcfr_varptr` has not been found.

[45] We don't test for *equality* of values or other *thing*s, knowing that λ-term equality is undecidable, and acknowledging that deep equality compare of ASTs like `tree` or `gimple` is too expensive.

the following occurrences. This is useful in patterns like `?(gimple_assign_single ?var ?var)` to find assignments of a variable `var` to itself.

4.2 Pattern syntax overview

A pattern π may match some matched *thing* μ, or may fail. It the matching succeeds, sub-patterns may be matched, and pattern variables may become bound. The *thing* bound by some pattern variable is checked in following occurrences of the same pattern variables and is available inside the match-clause body.

Patterns may be one of:

- expressions ε (e.g., constant literals) are (degenerated) patterns. They match the matched data μ iff ε == μ (for the C sense of equality, which for pointers is their identity).
- The **wildcard** noted `?_` matches everything (every *value* or *stuff*) and never fails.
- a pattern variable `?`v matches μ if it was unset (by a previous [sub-]matching of the same `?`v). In addition, it is then bound to μ. If the pattern variable was previously set, it is tested for identity (with equality in the C sense).
- most patterns are **matcher** patterns `?(`m ε_1 ... ε_n π_1 ... π_p`)` where the $n \geq 0$ expressions ε_i are input parameters to the matcher m and the π_j sub-patterns are passed extracted data. The matcher is either a *c-matcher* (declaring how to translate that pattern to *C* code) or it is a *fun-matcher* (matching is done by a MELT function returning secondary *things*).
- instance patterns are like `?(instance` κ `:`Φ_1 π_1 ... `:`Φ_n π_n`)`; the matched μ is an object of [a sub-] class κ whose field Φ_i matches sub-pattern π_i.
- conjunctive patterns are `?(and` π_1 ... π_n`)` and they match μ iff every π_i in sequence matches μ; notice that when some π_i is a pattern variable `?`v that variable is matched and μ should match the further π_j (with $j > i$) with v appropriately bound to μ. (This generalizes the `as` keyword inside Ocaml patterns).
- disjunctive patterns are `?(or` π_1 ... π_n`)` and they match μ if one of the π_i matches μ.

4.3 C-matchers and fun-matchers

The *c-matchers* are one of the building blocks of patterns - much like primitives are one of the building blocks of expressions. Like primitives, c-matchers are defined as a specialized C code generation template. In the example above (§4.1.2), most composite patterns involve c-matchers: `tree_var_decl`, `tree_record_type` and `cstring_same` are C-matchers.

Like for every pattern, a C-matcher defines how the pattern using it should perform its test, and then how it should do its fill. A simple example of a C-matcher is `cstring_same`: some `:cstring` *stuff* σ matches the pattern `?(cstring_same "fprintf")` iff σ is the same as the `const char*` string `"fprintf"` given as input to our c-matcher. This c-matcher has a test part, but no fill part (because used without sub-patterns).

```
(defcmatcher cstring_same  (:cstring str cstr) () strsam
  :doc #{The $CSTRING_SAME c-matcher matches a string $STR iff it equals the constant string $CSTR.
        The match fails if $STR is null or different from $CSTR.}#
  #{ /*$STRSAM test*/ ($STR != (const char*)0 && $CSTR != (const char*)0 && !strcmp($STR, $CSTR)) }# )
```

Notice that the state symbol `strsam` is used inside a comment, to uniquely identify each occurrence in the generated C, and that we take care of testing against null `const char*` pointers to avoid crashes.

A more complex (and GCC specific) example is the `gimple_assign_single` c-matcher (to filter single assignments in compiled code). It defines both a testing and a filling expansion using two macro-strings:

```
(defcmatcher gimple_assign_single
   (:gimple ga)  (:tree lhs rhs)  gimpassi
   #{ /*$GIMPASSI test*/($GA && gimple_assign_single_p ($GA)) }#
   #{ /*$GIMPASSI fill*/ $LHS = gimple_assign_lhs ($GA);   $RHS = gimple_assign_rhs1($GA); }# )
```

Here `ga` is the matched gimple, and `lhs` & `rhs` are the output formals: they are assigned in the fill expansion to transmit `tree`-s to sub-patterns!

C-matchers are a bit like Wadler's notion of *Views* [30], but are expanded into C code. MELT also has *fun-matchers* which similarly are views defined by a MELT function returning a non-nil value if the test succeeded with several secondary results giving the extracted *thing*s to sub-patterns. For example the following code defines a fun-matcher `isbiggereven`[46] such that the pattern `?(isbiggereven` μ π`)` matches a `:long` stuff σ iff σ is a even number, greater than the number μ, and $\sigma/2$ matches the sub-pattern π. We define an auxiliary function `matchbiggereven` to do the matching [we could have used a `lambda`]. If the match succeeds, it returns a true (i.e. non nil) value (here `fmat`) and the integer to be matched with π. Its first actual argument is the fun-matcher `isbiggereven` itself. The testing behavior of the matching function is its first result (nil or not), and the fill behavior is through the secondary results.

```
(defun matchbiggereven (fmat :long s m)
; fmat is the funmatcher, s is the matched σ, m is the minimal μ
   (if (==i (%iraw s 2) 0)
      (if (>i s m) (return fmat (/iraw m 2)))))
(defunmatcher isbiggereven (:long s m) (:long o) matchbiggereven)
```

The fun-matcher definition has an input formals list and an output formal list, together defining the expected usage of the fun-matcher operator in patterns.

Both c-matchers and fun-matchers can also define what they mean in expression context (not in pattern one). So the same name can be used for constructing expressions and for destructuring patterns.

4.4 Implementing patterns in MELT

Designing and implementing patterns in MELT was quite difficult, because a good translation of pattern matching should :

- factorize, when possible, common sub-patterns, to avoid testing twice the same *thing*.
- share, when appropriate, data extracted from subpatterns.
- preferably re-use the many temporary locals used by the translation of the match, to lower the current MELT stack frame size.

Our first implementation of pattern translation to C is quite naive, and uses simple memoization techniques to factorize sub-patterns or share extracted data.

A better implementation of the pattern translator builds explicitly a directed graph (with shared nodes for tests and data), like figure 4. The graph has data nodes (for temporary variables for [sub-]matched *thing*s, or for boolean flags internal to the match) and elementary control steps. These steps are either tests (with both a “then” and an “else” jumps to other steps) or computations (usually with a single jump to a successor step). Some steps just set an internal boolean flag, or compute the conjunction of other flags. Other steps represent the testing or the filling parts of c-matchers or fun-matchers. Final success steps correspond to sub-expressions in the body of the matched clause and are executed if a flag is set.

[46] Our `isbiggereven` could also be defined as a c-matcher!

For instance a simple match (where v is the matched value) like below is translated into the complex internal graph [47] given in figure 4:

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

A more complex match like (match tcurfield ...) of §4.1.2 code line 12-20 produces about 20 match steps and 12 match data. This enhanced pattern matching is not entirely implemented at time of writing: the generation of the control graph for the match is implemented, but its translation into C is incomplete.

5 Conclusions and future work

Enhancing a legacy huge software with a domain specific language or scripting language is always a major challenge (§1), since incorporating a DSL inside a software is a major architectural design decision which should be taken early. Mature big software like GCC have their coding habits, memory management strategies and data organization which makes it very difficult to embed an existing scripting language (like Python, Ocaml, Ruby, ...).

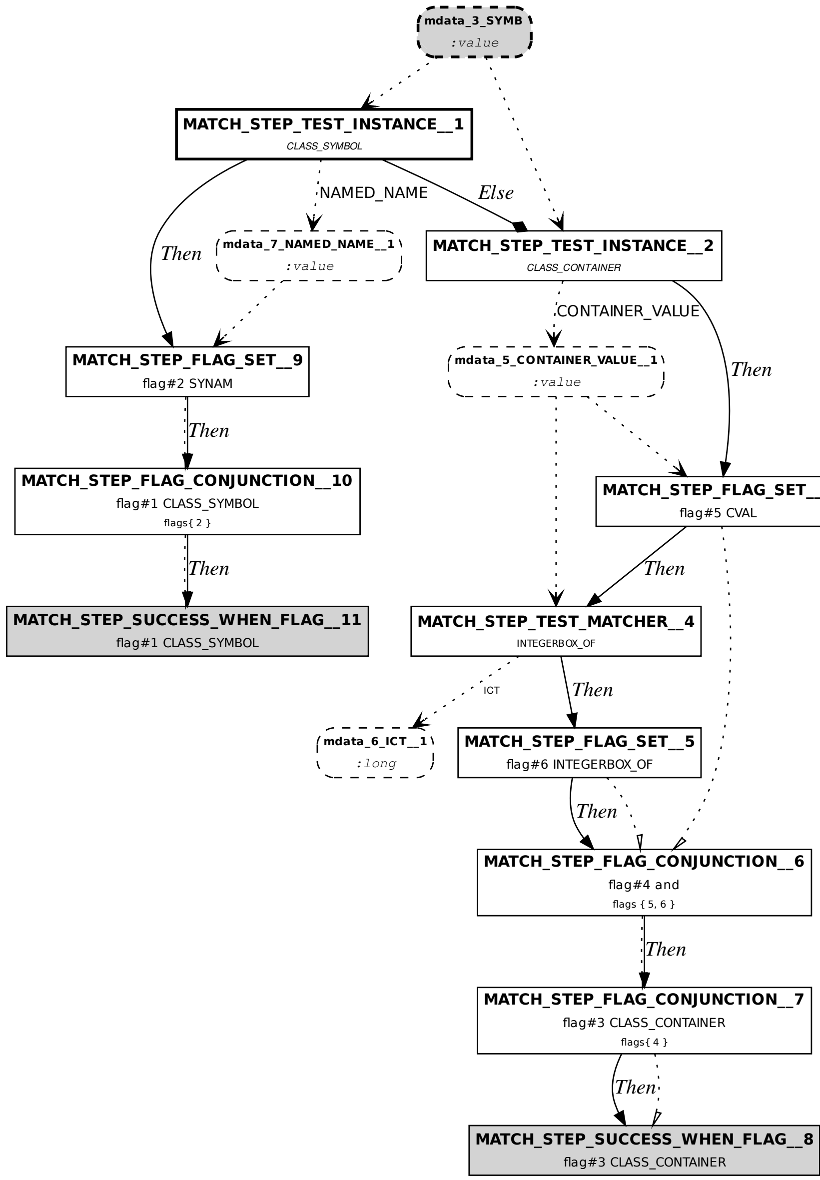
We have shown that adding a high-level DSL to a big software like GCC is still possible, by designing a run-time system §2 compatible with the existing infrastructure (notably Gg-c) and most importantly, by having the DSL deal both with boxed *values* and raw existing *stuff* in §3.2. Translating the DSL to the language (with its habits) used in that big software (*C* for GCC) enables high-level language constructs in our DSL. We have described a set of language constructs in §3.4 (c-matchers, primitives, c-iterators, ...) which give templates for *C* code generation.

Our empirical approach of designing and implementing a DSL like MELT to fit into a large software like GCC, could probably be re-used for adding DSLs inside other huge mature software projects: designing a runtime suitable for such a project, having several sorts of *thing*s (*value*s and *stuff*), generating code in the style of the existing legacy, and defining adequate language constructs giving code-generating templates.

Future work within MELT is mostly using this DSL to build interesting GCC extensions. P. Vittet has started in May 2011 a *Google Summer of Code* project to add specific warnings into GCC using MELT. A. Lissy considers using it for Linux kernel [13] code analysis. The opengpu mode should be completed. Also, some language features can be added or improved:

1. variadic functions, possibly provided by a `:rest` keyword similar to Common Lisp's `&rest`. These should be very useful for debugging and tracing messages.
2. adding backtracking or iterating pattern constructs; for instance to be able to have a pattern for any `:gimple_seq` *stuff* containing at least one gimple matching a given sub-pattern.
3. adding a nice usable and hygenic macro system, inspired by Scheme's `defsyntax`
4. performance improvements might be achieved by sometimes translating MELT function calls into a C function call whose signature mimicks the MELT function signature.

[47] To debug the pattern-match translator, MELT is generating a graph to be displayed with GraphViz. We have edited it (by removing details like source code location) for clarity.



Legend:

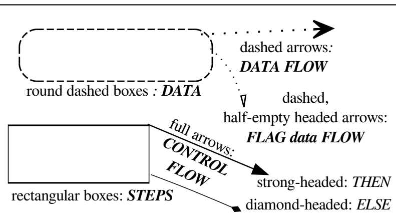


Figure 4: internal graph for match

5. a message caching machinery, where every MELT message passing occurrence would use a cache (keeping the last class of the sending).
6. a central monitor, which would communicate with parallel GCC$^{\mathsf{melt}}$ compilations through asynchronous textual protocols.

More generally, making MELT more high-level and more declarative (in J.Pitrat's [19, 20] sense) to be able to express GCC passes easily and concisely is an interesting challenge, and could be transposed to other legacy software.

Acknowledgments

Work on MELT has been funded by DGCIS thru ITEA GlobalGCC and FUI OpenGPU projects.

Thanks to Albert Cohen, Jan Midtgaard, Nic Volanschi and to the anonymous reviewers for their constructive suggestions and their proof-reading. Residual mistakes are mine.

References

[1] P. Collingbourne & P. Kelly (2009): *A Compile-Time Infrastructure for GCC Using Haskell*. In: *GROW09 workshop, within HIPEAC09*, `http://www.doc.ic.ac.uk/~phjk/GROW09/`, Paphos, Cyprus.

[2] P. Cousot & R. Cousot (1992): *Abstract Interpretation Frameworks*. *J. Logic and Computation* 2(4), pp. 511–547, doi:10.1093/logcom/2.4.511.

[3] P. Cousot & R. Cousot (2004): *Basic Concepts of Abstract Interpretation*, pp. 359–366. Kluwer Academic Publishers, doi:10.1007/978-1-4020-8157-6_27.

[4] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux & X. Rival (2005): *The ASTRÉE Analyser*. In: *Proc. ESOP'05*, LNCS 3444, Edinburgh, Scotland, pp. 21–30, doi:10.1007/978-3-540-31987-0_3.

[5] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux & X. Rival (2006): *Combination of Abstractions in the* ASTRÉE *Static Analyzer*. In M. Okada & I. Satoh, editors: *Eleventh Annual Asian Computing Science Conference (ASIAN'06)*, 4435, Springer, Berlin, Tokyo, Japan, LNCS, pp. 1–24, doi:10.1007/978-3-540-77505-8_23.

[6] D. Engler, B. Chelf, A. Chou & S. Hallem (2000): *Checking system rules using system-specific, programmer-written compiler extensions*. In: *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, USENIX Association, Berkeley, CA, USA, pp. 1–16.

[7] C. Flanagan, A. Sabry, B. F. Duba & M. Felleisen (2004): *The essence of compiling with continuations*. *SIGPLAN Not.* 39, pp. 502–514, doi:10.1145/989393.989443.

[8] GCC community (2011): *GCC internals doc*. Available at `http://gcc.gnu.org/onlinedocs/gccint/`.

[9] T. Glek & D. Mandelin (2008): *Using GCC instead of Grep and Sed*. In: *GCC Summit 2008*, Ottawa, pp. 21–32.

[10] D. Guilbaud, E. Goubault, A. Pacalet, B. Starynkévitch & F. Védrine (2001): *A Simple Abstract Interpreter for Threat Detection and Test Case Generation*. In: *WAPATV'01, with ICSE'01*, Toronto.

[11] R. Jones & R. D. Lins (1996): *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley.

[12] F. Le Fessant & L. Maranget (2001): *Optimizing Pattern-Matching*. In: *Proc. 2001 ICFP*, ACM Press.

[13] A. Lissy (2011): *Model Checking the Linux Kernel (FOSDEM2011)*.

[14] L. Maranget (2007): *Warnings for pattern matching*. *J. Functional Programming* 17.

[15] L. Maranget (2008): *Compiling Pattern Matching to Good Decision Trees* .

[16] G. Marpons-Ucero, J. Mariño-Carballo, M. Carro, Á. Herranz-Nieva, J. J. Moreno-Navarro & L. Fredlund (2008): *Automatic Coding Rule Conformance Checking Using Logic Programming*. In: *PADL*, pp. 18–34, doi:10.1007/978-3-540-77442-6_3.

[17] B. Monate & J. Signoles (2008): *Slicing for Security of Code*. In: *TRUST*, pp. 133–142, doi:10.1007/978-3-540-68979-9_10.

[18] J. Pitrat (1995): *Speaking about and Acting upon Oneself*. Technical Report 1995/29, LIP6/Laforia.

[19] J. Pitrat (1996): *Implementation of a reflective system*. *Future Gener. Comput. Syst.* 12(2-3), pp. 235–242, doi:10.1016/0167-739X(96)00011-8.

[20] J. Pitrat (2009): *Artificial Beings (the conscience of a conscious machine)*. Wiley / ISTE, doi:10.1002/9780470611791.

[21] S. Pop (2006): *The SSA Representation Framework: Semantics, Analyses and GCC Implementation.* Ph.D. thesis, Ecole des Mines de Paris. Available at `http://www.cri.ensmp.fr/classement/doc/A-381.pdf`.

[22] C. Queinnec (1996): *Lisp in Small Pieces*. Cambridge Univ. Pr., New York, NY, USA.

[23] N. Ramsey & S.P. Jones (2000): *A single intermediate language that supports multiple implementations of exceptions*. In: *Proc. PLDI '00*, ACM, New York, NY, USA, pp. 285–298, doi:10.1145/349299.349337.

[24] B. Starynkevitch (2006-2011): *MELT code [GPLv3] within GCC*. `http://gcc-melt.org/` and `svn://gcc.gnu.org/svn/gcc/branches/melt-branch`.

[25] B. Starynkevitch (2007): *Multi-Stage Construction of a Global Static Analyzer*. In: *GCC Summit 2007*, Ottawa, pp. 143–156.

[26] B. Starynkevitch (2009): *Middle End Lisp Translator for GCC, achievements and issues*. In: *GROW09 workshop, within HIPEAC09*, `http://www.doc.ic.ac.uk/~phjk/GROW09/`, Paphos, Cyprus.

[27] K. Trifunovic, A. Cohen, D. Edelsohn, F. Li, T. Grosser, H. Jagasia, R. Ladelsky, S. Pop, J. Sjödin & R. Upadrasta (2010): *GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation*. In: *GCC Research Opportunities Workshop (GROW'10)*, Pisa Italie. Available at `http://hal.inria.fr/inria-00551516/en/`.

[28] A. Venet & G. Brat (2004): *Precise and efficient static array bound checking for large embedded C programs*. In: *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, ACM Press, New York, NY, USA, pp. 231–242, doi:10.1145/996841.996869.

[29] N. Volanschi (2006): *A Portable Compiler-Integrated Approach to Permanent Checking*. In: *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, IEEE Computer Society, Washington, DC, USA, pp. 103–112, doi:10.1109/ASE.2006.8.

[30] P. Wadler (1987): *Views: a way for pattern matching to cohabit with data abstraction*. In: *Proc. POPL '87*, ACM, New York, NY, USA, pp. 307–313, doi:10.1145/41625.41653.