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 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 for GCC 4.6.0), heterogenous, and still increasing by 6% annually. It has no single main architect and hundreds of (mostly full-time) contributors, who follow strict social rules.

1.1 The powerful GCC legacy

The several GCC 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. 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

---

1Gnu Compiler Collection (gcc 4.6.0 released on march 25th 2011) on gcc.gnu.org
24.296 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.
3GCC 4.4.1, released July 22th, 2009, was 3.884MLOC, so a 0.412MLOC = 10.6% increase in 1.67 years
4Every 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.
5The GCC middle-end does not depend upon the source language or the target processor (except with parameters giving sizeof(int) etc.).
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 (\(\S 1.2\)). 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,...) \(\rightarrow\) 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.

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 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

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

\[(\text{code from gcc/gimple.h in GCC})\]

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

---

\(^6\)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](http://www.coverity.com)
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)
   ```c
   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)
   ```c
   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)
   ```c
   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 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 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 developed 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 → 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 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.

\[\text{See David Malcom’s GCC Python plugin announced in } \text{http://gcc.gnu.org/ml/gcc/2011-06/msg00293.html}\]
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 have 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 G 2.83GHz, 8Gb RAM, fast 10KRFM 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;
}
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 discriminant and number to be boxed). It uses the MELT_ENTERFRAME macro 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 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 things: the first-class MELT values (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’s heap). Informally, Things = Values ∪ 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 things of either kind (values or stuff).

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

---

20The Ocaml runtime has similar macros.
21In 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 Trees and Generic [so Gimple was mostly a conventional restriction on Trees] - 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.
There is unfortunately no way to implement full polymorphism in MELT: we cannot have MELT tuples containing a mix of raw trees 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 values 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. 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. 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. 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 peculiarities

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](http://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 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 SEXP 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: 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  

```melt
#{/*$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 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:

```melt
;; 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; 
    }#
```

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

25Inspired by handling of $ in strings or “here-documents” by shells, Perl, Ruby, ...

26The 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_RECEIVER`. 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_RECEIVER`). 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[0..]` 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 == MELTBOMAG_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 things 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 `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.
To associate things (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 trees or gimples...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 C type like gimple 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 dozens 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 things: 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 others can be things, 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 things (value or stuff), in particular local variables (or mismatched forms), 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.

---

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

32The 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 `things` (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 receiver 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 succeeding 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 `(setq ν ε)` locally binds (in the body of the enclosing `let`) the symbol `ν` to the raw long `stuff` 2, and in the `let` body `ν` is a raw long variable.

Patterns and pattern matching are explained in §4.

<table>
<thead>
<tr>
<th>expressions where ( n \geq 0 ) and ( p \geq 0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>application ( (Φ α_1 \ldots α_n) )</td>
</tr>
<tr>
<td>assignment ( (setq v ε) )</td>
</tr>
<tr>
<td>message passing ( (σ ρ α_1 \ldots α_n) )</td>
</tr>
<tr>
<td>let expression ( (let (β_1 \ldots β_n) ; ε_1 \ldots ε_p) ; ε′) )</td>
</tr>
<tr>
<td>sequence ( (progn ε_1 \ldots ε_n) ; ε′) )</td>
</tr>
<tr>
<td>abstraction ( (lambda Φ ; ε_1 \ldots ε_n) ; ε′) )</td>
</tr>
<tr>
<td>pattern matching ( (match ε χ_1 \ldots χ_n) )</td>
</tr>
</tbody>
</table>

Conditional expressions alter control flow as usual. However, conditions can be `things`, 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


<table>
<thead>
<tr>
<th>Test</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(if (\tau\ \theta\ \varepsilon))</td>
<td>if (\tau) then (\theta) else (\varepsilon) (like ?: in C)</td>
</tr>
</tbody>
</table>

### Conditional Expressions

- **Conditional:** `(cond \(k_1\) ... \(k_n\))`
  - evaluate conditions \(k_i\) until one is satisfied

- **Conjunction:** `(and \(k_1\) ... \(k_n\) \(k'\))`
  - if \(k_1\) and then \(k_2\) and ... and \(k_n\) is true (non-nil or non-zero) then \(k'\) otherwise the cleared thing of same type

- **Disjunction:** `(or \(\delta_1\) ... \(\delta_n\))`
  - \(\delta_1\) or else \(\delta_2\) ... = the first of the \(\delta_i\) which is true (non-nil or non-zero, ...)

In a `cond` expression, every condition \(k_i\) (except perhaps the last) is like `(\(\gamma_i\) \(e_{i,1}\) ... \(e_{i,p_i}\) \(\varepsilon'\))` with \(p_i \geq 0\). The first such condition for which \(\gamma_i\) is true gets its sub-expressions \(e_{i,j}\) evaluated sequentially for their side-effects and gives the result of \(\varepsilon'\). The last condition can be `(else \(e_1\) ... \(e_n\) \(\varepsilon'\))`, is triggered if all previous conditions failed, and (with the sub-expressions \(e_i\) evaluated sequentially for their side-effects) gives the result of \(\varepsilon'\)

**MELT** has some more expressions.

### More Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop</td>
<td>(forever (\lambda\ \alpha_1) ... (\alpha_n))</td>
</tr>
<tr>
<td>Exit</td>
<td>(exit (\lambda\ \varepsilon_1) ... (\varepsilon_n) (\varepsilon'))</td>
</tr>
<tr>
<td>Return</td>
<td>(return (\varepsilon\ \varepsilon_1) ... (\varepsilon_n))</td>
</tr>
<tr>
<td>Multiple Call</td>
<td>(multicall (\phi\ \kappa\ \varepsilon_1) ... (\varepsilon_n) (\varepsilon'))</td>
</tr>
<tr>
<td>Recursive Let</td>
<td>(letrec (\beta_1) ... (\beta_n) (\varepsilon_1) ... (\varepsilon_p) )</td>
</tr>
<tr>
<td>Field Access</td>
<td>(get_field (\Phi\ \varepsilon) )</td>
</tr>
<tr>
<td>Unsafe Field Access</td>
<td>(unsafe_get_field (\Phi\ \varepsilon) )</td>
</tr>
<tr>
<td>Object Update</td>
<td>(put_fields (\varepsilon\ :\Phi_1\ \varepsilon_1) ... (:\Phi_n\ \varepsilon_n) )</td>
</tr>
<tr>
<td>Unsafe Object Update</td>
<td>(unsafe_put_fields (\varepsilon\ :\Phi_1\ \varepsilon_1) ... )</td>
</tr>
</tbody>
</table>

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

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

### Constructive Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>(list (\alpha_1) ... (\alpha_n) )</td>
</tr>
<tr>
<td>Tuple</td>
<td>(tuple (\alpha_1) ... (\alpha_n) )</td>
</tr>
<tr>
<td>Instance</td>
<td>(instance (\kappa\ :\Phi_1\ \varepsilon_1) ... (:\Phi_n\ \varepsilon_n) )</td>
</tr>
</tbody>
</table>

\(^{35}\)I.e. test if the value \(\alpha\) of \(\varepsilon\) is an object which is a direct or indirect instance of the class defining field \(\Phi\), otherwise a nil value is given.

\(^{36}\)Update object \(\alpha\), value of \(\varepsilon\), only if it is an object which is a direct or indirect instance of the class defining each field \(\Phi_i\)
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).

<table>
<thead>
<tr>
<th>expressions defining names</th>
<th>expressions exporting names</th>
</tr>
</thead>
</table>

| for functions | (defun \( V \phi \varepsilon_1 \ldots \varepsilon_n \varepsilon' \)) | define function \( V \) with formal arguments \( \phi \) and body \( \varepsilon_1 \ldots \varepsilon_n \varepsilon' \) |
|----------------|-------------------------------------------------|
| for classes    | (defclass \( V :\text{super} \sigma :\text{fields} (\phi_1 \ldots \phi_n) \)) | define class \( V \) of super-class \( \sigma \) and own fields \( \phi_1 \ldots \phi_n \) |
| for instances  | (definstance \( I \kappa :f_1 \varepsilon_1 \ldots :f_n \varepsilon_n \)) | define an instance \( I \) of class \( \kappa \) with each field \( f_i \) initialized to the value of \( \varepsilon_i \) |
| for selectors  | (defselector \( \sigma \kappa [ :\text{formals} \Psi ] :f_1 \varepsilon_1 \ldots :f_n \varepsilon_n \)) | define an selector \( \sigma \) of class \( \kappa \) (usually class\_selector) with each extra field \( f_i \) initialized to the value of \( \varepsilon_i \) (usually no extra fields are given so \( n = 0 \)) and with optional formals \( \Psi \) |
| for primitives | (defprimitive \( V \phi :\theta \eta \)) | define primitive \( V \) with formal arguments \( \phi \), result c-type \( \theta \) by macro-string expansion \( \eta \) |
| for c-iterators | (defciterator \( V \Phi \Psi \eta \eta' \)) | define c-iterator \( V \) with input formals \( \Phi \), state symbol \( \sigma \), local formals \( \Psi \), start expansion \( \eta \), end expansion \( \eta' \) |
| for c-matchers  | (defcmatcher \( V \Phi \Psi \sigma \eta \eta' \)) | define c-matcher \( V \) with input formals \( \Phi \) [the matched thing, then other inputs], output formals \( \Psi \), state symbol \( \sigma \), test expansion \( \eta \), fill expansion \( \eta' \) |
| for fun-matchers | (defunmatcher \( V \Phi \Psi \varepsilon \)) | define funmatcher \( V \) with input formals \( \Phi \), output formals \( \Psi \), with function \( \varepsilon \) |

| of values     | (export\_value \( V_1 \ldots \)) | export the names \( V_i \) as bindings of values (e.g., of functions, objects, matcher, selector, ...) |
| of macros     | (export\_macro \( V \varepsilon \)) | export name \( V \) as a binding of a macro (expanded by the \( \varepsilon \) function) |
| of classes    | (export\_class \( V_1 \ldots \)) | export every class name \( V_i \) and all their own fields (as value bindings) |
| as synonym    | (export\_synonym \( V \varepsilon' \)) | export the new name \( V \) as a synonym of the existing name \( V' \) |

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.

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

<table>
<thead>
<tr>
<th>expressions for debugging</th>
<th>debug printing message $\mu$ &amp; value $\varepsilon$</th>
<th>nice “halt” showing message $\mu$ when asserted test $\tau$ is false</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug message $\varepsilon$</td>
<td>(debug_msg $\varepsilon$ $\mu$)</td>
<td>like #warning in C: emit warning $\mu$ at MELT translation time and gives $\varepsilon$</td>
</tr>
<tr>
<td>assert check $\tau$</td>
<td>(assert_msg $\mu$ $\tau$)</td>
<td>conditional on a preprocessor symbol: emitted C code is #if $\sigma$ code for $\varepsilon$ #else code for $\varepsilon'$ #endif</td>
</tr>
<tr>
<td>warning $\varepsilon$</td>
<td>(compile_warning $\mu$ $\varepsilon$)</td>
<td>conditional on a preprocessor symbol: emitted C code is #if $\sigma$ code for $\varepsilon$ #else code for $\varepsilon'$ #endif</td>
</tr>
</tbody>
</table>
| Cpp test $\varepsilon'$ | (cppif $\sigma$ $\varepsilon$ $\varepsilon'$) | the $\varepsilon_i$ are translated only if GCC has version prefix string $\beta$

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,

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

would be translated to

```c
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 \texttt{negi} is defined exactly as:

\begin{verbatim}
(defprimitive negi (:long i) :long
  :doc #\textquote{Integer unary negation of \$i.}#
  #(-($i))#)
\end{verbatim}

Here we specify that the formal argument \(i\) is, like the result of \texttt{negi}, a :\texttt{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\footnote{Assuming that \(x\) is a MELT variable for a :\texttt{long} stuff, then the expression \((+1 \texttt{negi} x) 1\) is normalized as let \(\alpha = -x, \beta = \alpha + 1\) in \(\beta\) in pseudo-code - suitably represented inside MELT (where \(\alpha, \beta\) are fresh gensym-ed variables).} and type checking. During expansion, the formals appearing in the primitive definition are replaced appropriately.

3.4.3  C-iterators

A MELT \texttt{c-iterator} is an operator translated into a \texttt{for}-like C loop. The GCC compiler defines many constructs similar to \texttt{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 \texttt{gimple} \(g\) inside a given \texttt{gimple_seq} \(s\) GCC mandates (see \cite{1.1}) the use of a \texttt{gimple_simple_iterator}.

In MELT, to iterate on the \texttt{:gimpleseq} \(s\) obtained by the expression \(\sigma\) and do something on every \texttt{:gimple} \(g\) inside \(s\), we can simply code (let ( (:\texttt{gimpleseq} s \sigma) ) (each\_in\_\texttt{gimpleseq} (s) (:\texttt{gimple} g) (do something with \(g\)...)]) by invoking the \texttt{c-iterator} \texttt{each\_in\_\texttt{gimpleseq}}, with a list of inputs - here simply \(s\) - and a list of local formals - here (:\texttt{gimple} \(g\)) - as the iterated things.

This \texttt{c-iterator} (a template for such \texttt{for}-like loops) is defined exactly as:

\begin{verbatim}
(defciterator each\_in\_\texttt{gimpleseq}
  (:\texttt{gimpleseq} gseq) ;start formals
each\texttt{gimplseq} ;state
  (:\texttt{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*/
  )#)
\end{verbatim}

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 :\texttt{void} result, since they are used only for their side effects.

3.5  Modules, environments, standard library and hooks

A single *\texttt{.melt}* source file\footnote{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.} is translated into a single module loaded by the MELT run-time. The module’s generated \texttt{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.
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_sexp, 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 dialect (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 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). 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.

---

39Each bound name is bound only once, and there are no separate namespaces like in C or Common Lisp.
40Like (current_module_environment_container) and (parent_module_environment), etc.
41Strangely, 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 \( \text{question } x \) [it is the pattern variable \( x \)]. \( ?_\) is \(^{42}\) the \textit{wildcard pattern} (matching anything). An expression occurring in pattern context is a \textit{constant} pattern. Patterns may be nested (in composite patterns) and occur in \textit{match} expressions.

Elementary patterns are ultimately translated into code that \textit{tests} that the matched \textit{thing} \( \mu \) can be filtered by the pattern \( \pi \) followed by code which extracts appropriate data from \( \mu \) and \textit{fills} some locals with information extracted from \( \mu \). 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\textsuperscript{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 \texttt{melt-runtime.c}, (e.g., in function of figure \( \text{2} \)). When enabled with \texttt{-fplugin=melt-arg-mode=meltframe}, it adds a new pass (after the "ssa" pass\(^{43}\) of GCC \texttt{(21)} \texttt{melt\_frame\_pass} to GCC. This pass first finds the declaration of the local \texttt{meltfram\_} in the following pass execute function:

```lisp
(defun meltframe_exec (pass)
  (let (:
        (:tree tfundecl (cfun_decl)) (:long nbvarptr 0)
        (:tree tmeltframdecl (null_tree)) (:tree tmeltframtype (null_tree))
        (each_local_decl_cfun () (:tree tlocdecl :long ix)
          (match tlocdecl
            (? (tree_var_decl
                (? (and ?tvtyp
                    (? (tree_record_type_with_fields ?tmeltframrecnam ?tmeltframfields))
                  (? (cstring_same "meltfram\_") ?)_)
                (setq tmeltframdecl tlocdecl) (setq tmeltframtype tvtyp)
                (foreach_field_in_record_type (tmeltframfields) (:tree tcurfield)
                  (match tcurfield
                    (? (tree_field_decl
                        (? (tree_identifier (? (cstring_same "mcfr_varptr"))
                          (? (tree_array_type ?telemtype
                            (? (tree_integer_type_bounded ?tindextype
                              (? (tree_integer_cst ?idxmax)
                                (? (tree_integer_cst ?lmax)
                                  ?tsize))))
                            (setq tmeltframvarptr tcurfield) (setq nbvarptr lmax))))))
                    (_ (void)))))
          ))
          (match tlocdecl
            (? (tree_var_decl
                (? (and ?tvtyp
                    (? (tree_record_type_with_fields ?tmeltframrecnam ?tmeltframfields))
                  (? (cstring_same "meltfram\_") ?)_)
                (setq tmeltframdecl tlocdecl) (setq tmeltframtype tvtyp)
                (foreach_field_in_record_type (tmeltframfields) (:tree tcurfield)
                  (match tcurfield
                    (? (tree_field_decl
                        (? (tree_identifier (? (cstring_same "mcfr_varptr"))
                          (? (tree_array_type ?telemtype
                            (? (tree_integer_type_bounded ?tindextype
                              (? (tree_integer_cst ?idxmax)
                                (? (tree_integer_cst ?lmax)
                                  ?tsize))))
                            (setq tmeltframvarptr tcurfield) (setq nbvarptr lmax)))))))))
            ))
    ))
)

The \texttt{let} line 2 spans the entire MELT function \texttt{meltframe\_exec}, with bindings lines 3 & 4 for \texttt{tfundecl}, \texttt{nbvarptr}, \texttt{tmeltframdecl} & \texttt{tmeltframtype} locals. The \texttt{each\_local\_decl\_cfun} is a \texttt{c}\texttt{-}iterator (iterating -lines 5 to 11- on the \textit{Tree-s} representing the local declarations in the function). The \textit{match} expression filters the current local declaration \texttt{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 \texttt{meltfram\_}, we assign (line 10) appropriately \texttt{tmeltframdecl} & \texttt{tmeltframtype}, and we iterate (line 11) on its fields to find, by the \textit{match} (lines 12-21), the declaration of field \texttt{mcfr\_varptr} (in the

\(^{42}\)\_\_ can be pronounced as “joker”

\(^{43}\)\textit{ssa} means Static Single Assignment, so at that stage the code is represented in \textit{Gimple/SSA} form, so each SSA variable is assigned once!
C code), and its array index upper bound \( l_{\text{max}} \), assigning them (line 20) to locals \texttt{tmeltframvarptr} \& \texttt{nbvarptr}. Otherwise, using the wildcard pattern \(?_\ldots\) we give a \texttt{void} result for the match of \texttt{tlocdecl} (line 21).

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

\begin{verbatim}
(each_bb_cfun () (:basic_block bb :tree fundecl))
(eachgimple_in_basicblock (bb))
 (:gimple g)
 (match g)
 (each_bb_cfun () (:basic_block bb :tree fundecl))

(eachgimple_in_basicblock (bb))
 (:gimple g)
 (match g)

( ?(gimple_assign_single
  ?(tree_array_ref ?(tree_component_ref tmeltframdecl tmeltframvarptr)
   ?(tree_integer_cst ?idst))
  ?(tree_array_ref ?(tree_component_ref tmeltframdecl tmeltframvarptr)
   ?(tree_integer_cst ?isrc)))
  [handle assign "meltfram\_.mcfr_varptr[\idst] = meltfram\_.mcfr_varptr[\isrc];"]
  ( ?(gimple_assign_single
    ?(tree_array_ref ?(tree_component_ref tmeltframdecl tmeltframvarptr)
     ?(tree_integer_cst ?idst))
    ?rhs)
  [handle assign "meltfram\_.mcfr_varptr[\idst] = rhs;"]
  ( ?(gimple_assign_single ?lhs
    ?(tree_array_ref ?(tree_component_ref tmeltframdecl tmeltframvarptr)
     ?(tree_integer_cst ?isrc)))
    [handle assign "lhs = meltfram\_.mcfr_varptr[\isrc];"]

The \texttt{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 \( \texttt{rhs} \) 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 \texttt{meltframe\_exec} remembers all left-hand side and right-hand side occurrences of \texttt{meltfram\_.mcfr_varptr[\kappa]}, 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 \texttt{thing} (\texttt{stuff}, e.g., :long \ldots or even :void, or \texttt{value}) as their result. Patterns may be nested like the \texttt{tree_var_decl} or \texttt{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 \(?_\ldots\) pattern.

A pattern is usually composite (with nested sub-patterns) and has a double role: first, it should test if the matched \texttt{thing} fits; second, when it does, it should extract \texttt{things} and transmit them to eventual sub-patterns; this is the \textit{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 \textit{identity} \(^{45}\) with \texttt{==} in the generated C code on all

\(^{44}\) A warning is issued if \texttt{meltfram\_} or \texttt{mcfr_varptr} has not been found.

\(^{45}\) We don’t test for \textit{equality} of values or other \texttt{things}, knowing that \( \lambda \)-term equality is undecidable, and acknowledging that deep equality compare of ASTs like \texttt{tree} or \texttt{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. If 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 `?ν` matches `µ` if it was unset (by a previous [sub-[sub-]matching of the same ?ν). 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 ε₁ ... εₙ π₁ ... πₚ)` where the `n ≥ 0` expressions `εᵢ` are input parameters to the matcher `m` and the `πⱼ` 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 :Φ₁ π₁ ... :Φₙ πₙ)`; the matched `µ` is an object of [a sub-] class `κ` whose field `Φᵢ` matches sub-pattern `πᵢ`.
- conjunctive patterns are `?(and π₁ ... πₙ)` and they match `µ` iff every `πᵢ` in sequence matches `µ`; notice that when some `πᵢ` is a pattern variable `?ν` that variable is matched and `µ` should match the further `πⱼ` (with `j > i`) with `ν` appropriately bound to `µ`. (This generalizes the `as` keyword inside Ocaml patterns).
- disjunctive patterns are `?(or π₁ ... πₙ)` and they match `µ` if one of the `πᵢ` 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).

```c
(defcmatcher cstring_same (:cstring str cstr) () strsam
  :doc #![The $CSTRINGSAME 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 things 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 σ/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 sub-patterns.
- 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 things, 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:

\[
\begin{align*}
\text{(match } v & \\
\text{ ( ?(instance class_symbol :named_name } & \text{ ?synam)} \\
\text{ (f synam))} \\
\text{( ?(instance class_container :container_value } & \text{ ?(and } & \text{ ?cval } & \text{ ?(integerbox_of } & \text{ ?.)))} \\
\text{(g cval))}
\end{align*}
\]

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 things (values 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.
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\textsuperscript{melt} compilations through asynchronous textual protocols.

More generally, making MELT more high-level and more declarative (in J.Pitrat’s \cite{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

\begin{thebibliography}{12}
\bibitem{1} P. Collingbourne & P. Kelly (2009): \textit{A Compile-Time Infrastructure for GCC Using Haskell}. In: GROW09 workshop, within HIPEAC09, \url{http://www.doc.ic.ac.uk/~phjk/GROW09/}, Paphos, Cyprus.

\bibitem{2} P. Cousot & R. Cousot (1992): \textit{Abstract Interpretation Frameworks}. \textit{J. Logic and Computation} 2(4), pp. 511–547, doi\cite{10.1093/logcom/2.4.511}


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


\bibitem{7} C. Flanagan, A. Sabry, B. F. Duba & M. Felleisen (2004): \textit{The essence of compiling with continuations}. \textit{SIGPLAN Not.} 39, pp. 502–514, doi\cite{10.1145/989393.989443}

\bibitem{8} GCC community (2011): \textit{GCC internals doc}. Available at \url{http://gcc.gnu.org/onlinedocs/gccint/}


\bibitem{13} A. Lissy (2011): \textit{Model Checking the Linux Kernel (FOSDEM2011)}.

\end{thebibliography}
B. Starynkevitch


