MELT- a Lisp dialect matching GCC internals*

Basile STARYNKÉVITCH

CEA LIST (Software Reliability Lab)**
Saclay, France
basile@starynkevitch.net

Abstract. Since the GCC compiler is extensible thru plugins, many innovative applications can be considered, and would better be coded in a high-level language. But using an existing one is impractical. We demonstrate that GCC (a huge mature free software) can be flexibly extended thru MELT *(Middle End Lisp Translator)*, which translates a Lisp dialect into C code suitable inside GCC. We describe the infrastructure and the linguistic devices useful to easily extend GCC and adapt to its evolution. These are constructs defining how to generate C code fitted to GCC internals. Notably, a versatile pattern-matching facility has been implemented in our bootstrapped MELT translator, and is useful to work on GCC middle-end (`tree` & `gimple`, ...) internal representations.

1 Introduction

The current GCC[1] free compiler is an important production [straight|cross] compiler for many source languages (C, C++, Ada, Java, Fortran, ...), about 30 different machine architectures, and many systems. Its source code size is huge (3844KLOC)[2] and growing (42% increase in 2 years)[3]. It has hundreds of contributors (but no "benevolent dictator" contrarily to the Linux kernel) and strict social rules[4].

GCC [10] has several front-ends (parsing C, C++, Ada, Fortran, ... source code) producing common internal representations called *Tree* and *Generic*. These are later transformed into middle-end internal representations, called *Gimple* - thru a transformation called *gimplification*. The bulk of the compiler work is to operate repeatedly on these *Gimple* representations, and its middle-end contains nearly 200 passes moulding these (in different forms). Finally, target-machine specific back-ends generate, from a low-level Gimple, using the *Register Transfer Language* representations, the generated assembly code. Besides that, many other data structures exist within GCC (and a lot of global variables).

---

* svn `$Revision: 112 $` - unpublished draft of submission to CC2010.

** CEA DRT/LIST/DTSI/SOL; bat 528 pt courrier 94; 91191 GIF/YVETTE CEDEX; France.

[1] Gnu Compiler Collection (next is 4.5, at end 2009, from current GCC trunk, on `gcc.gnu.org`)

[2] Kilo-lines of code, measured by D.Wheeler's `sloccount` on GCC 4.4.1; On GCC trunk rev.152437 of October 4th 2009, `sloccount` gives 3962KLOC.

[3] `gcc-4.2.1.tar.bz2` (July 2007), is 44Mb; `gcc-4.4.1.tar.bz2`, (July 2009), almost 63Mb.

[4] Every submitted code patch should be accepted by a reviewer.

It should be stressed that *most* internal GCC *representations* are constantly *evolving*, and *there is no stability*[5] of the internal GCC API[6].

The forthcoming GCC 4.5 will provide several disruptive new features, including *Link Time Optimization* and *Plugins*[7]. This will enable using and extending GCC for non code-generation activities like static analysis [11, 3, 19, 31], threats detection (like in Two[12] or Astrée[5, 6]), code refactoring, coding rules validation[18], etc... or ad-hoc library optimizations[8] while taking advantage of the many powerful existing GCC passes and internal representations. In this perspective, **GCC should have "everything but the kitchen-sink" for source programs manipulation** (thru many plugins) [9], much like Emacs provides "everything but the kitchen-sink" for interactive text handing and editing. For such non-mainstream developments of GCC plugin extensions, and for prototyping, *better programmer's productivity is essential*. Higher level languages like Ocaml, Haskell, Lisp[23, 30] or scripting implementations like Python, Ruby, Guile, ... increase that productivity (perhaps at the expense of runtime performance). While several static analyzers (Frama-C, Astrée, ...) are coded in Ocaml [7], having similar tools available as GCC plugins would benefit the user community (adding extra options to `gcc` is much easier than using another tool) and perhaps even the GCC compiler (in the future, advanced but costly static analysis could improve code generation); most importantly, working on internal representations takes advantage of existing features in GCC (optimization, multiple front-ends, ...).

But **embedding an existing language implementation inside GCC is impractical**, because GCC is evolving very quickly, and has no stable interface. Also, most of GCC API are simple inlined functions, or short C macros, or even raw C `struct` access. Hand coding the necessary C glue code (for e.g. Ocaml, Python, or Guile runtime) is a huge boring effort (to be updated at every GCC release) or should be confined to a small subset of the API in a specific version of GCC (like [2]), and glue generators (Swig or Camlidl cannot really help since they expect a well formalized interface, with some peculiar restrictions. Furthermore, such glue code usually makes a C glue function out of every trait of the API, adding a performance burden. There is an *impedance mismatch* between GCC data and functions and some higher level language implementation. Also, GCC has its own garbage collector Ggc[10], and mixing different garbage collectors is

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

[6] GCC has no well defined and documented application programming interface for compiler extensions; its API is just a big set of internal header files, so is a bit messy for outsiders.

[7] The design and implementation of plugins within GCC has been influenced by MELT. Plugins, in particular MELT, are only available in GCC on hosts (GNU/Linux, Solaris, ...) having `dlopen`, which is a function to dynamically load a shared object [i.e. a plugin binary file] within a process.

[8] Optimizations like replacing `fprintf(stdout,...)` with `printf(...)` after function inlining - we call that our *make-green* optimization -, or more complex transformations caring about specific library semantics - like `stdc++` containers or Qt widgets, etc ....

[9] Coding a "psychotherapist" GCC plugin in MELT is left ☺ as an exercise to the courageous reader! Emacs already has `M-x doctor` but please make it better in MELT, without expecting it ☺ to surpass human professionals!

[10] The Ggc is implemented in `gcc/ggc*.[ch]` and in the `gengtype` code generator, which parses `GTY`-annotated C structures to generate their marking routines.

notoriously error-prone. Given the growing size of GCC and of its community it would be unrealistic to alter its coding habits.

Therefore, **the reasonable way to provide a higher level language for GCC plugins is to dynamically generate suitable C code** adapted to GCC style and legacy and similar in form to existing hand-coded C routines inside GCC. This is the driving idea of our MELT (*Middle End Lisp Translator*) language and plugin implementation [28, 29, 27]. 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 [24] whose architecture was designed and centered on an embedded interpreter (E-Lisp for Emacs, Lua^{ocaml} for C--).

This paper demonstrates that *a huge mature compiler can still be flexibly extended with a higher level language*, by generating C code suitable for/as plugins. It shows these extensions can be used thru an appropriate runtime (§2), describes some of the required linguistic devices (§3), notably a flexible pattern matching facility (§4). It concludes (§5) with future work and possible applications.

The reader is expected to understand the complexity of current compiler technology, and to know a bit some existing Lisp-like language, be it Common Lisp, Scheme[26], Emacs-Lisp etc ...

2 Using MELT and its runtime.

From the user's perspective, the GCC compiler enabled with MELT (GCC^{melt}) can be run with a command as: `gcc -fplugin=melt.so -fplugin-arg-melt-mode=makegreen -O -c foo.c`. This instructs `GCC` 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=makegreen` 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` ...`ana-simple.so`. These modules initialize MELT data, e.g. classes and handlers. The MELT handler associated to mode `makegreen` registers a new GCC pass (coded in `ana-simple.melt`) which is executed by GCC pass manager when compiling the file `foo.c`. This pass finds calls like `fprintf(stdout, ...)` and replaces them with `printf(...)` after GCC has inlined functions [thus perhaps creating such calls]! The `melt.so` plugin is hand-coded in C (in our `melt-runtime.[hc]` files[11] - 14KLOC). The modules `warmelt*.so` ...`ana*.so` are coded in MELT (as source files `warmelt*.melt` ...`ana*.melt` which have been translated by MELT into generated C files `warmelt*.c` ...`ana*.c`, themselves compiled into modules `warmelt*.so` etc ...).

The MELT translator (able to generate `*.c` from `*.melt`) is *bootstrapped* so exercises most of its features and its runtime : the translator's source code is coded in MELT, precisely the `warmelt*.melt` files (25.7KLOC), and the MELT *source* SubVersion repository also contains the *generated* `warmelt*.c` (504KLOC). Other MELT files, like `ana*.melt` [4KLOC, incomplete, implementing simple analysis GCC passes in MELT]

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

don't need that. The MELT translator[12] 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^{melt} compiler invocation, some temporary `*.c` code, run another `gcc` to compile that into a temporary `*.so`, and load (i.e. `dlopen`) and execute that - all this in a single `cc1` process; this can be useful for sophisticated static analysis specialized using [28] partial evaluation techniques within the analyzer, or just to "run" a MELT file.

Translation from MELT code to C code is fast: for instance, on a x86-64 Core2 GNU/Linux desktop system[13], the 5.1KLOC file `warmelt-normal.melt` file is translated into `warmelt-normal.c` in 5.2 seconds (wall time). But the `warmelt-normal.c` generated file has 164KLOC, needing 275 sec. to be compiled with `gcc-4.4 -O1 -g -fPIC`. So most of the time is spent in compiling the generated C code, not in generating it. All the module's data is built in the module starting routine[14].

The MELT runtime `melt-runtime.c` is built above the GCC infrastructure, notably its Ggc mark & sweep precise garbage collector (GC) [14], which is explicitly started, and provides hooks for plugins but does not handle any local variables by itself (in contrast with most other garbage collectors) : usually Ggc collection happens in the GCC pass manager between passes, not inside them. But as in most applicative or functional languages, MELT code tends to allocate a lot of temporary values (which often die quickly). These values are handled by a *generational copying* MELT GC, triggered by the MELT allocator when its birth region is full, and backed up by the existing Ggc (so the old generation of MELT GC is the Ggc heap). Generational copying GCs handle quickly dead young temporary values by discarding them at once, but require a scan of all local variables, a write barrier, and *normalization* of explicit intermediate values inside calls[15]. This is awkward in hand-written C code but easy to generate. A minor MELT GC is triggered after each GCC pass coded in MELT to ensure that all live young MELT values have migrated to the old Ggc heap, etc.

Explicit availability of local variables (and also of the current closure), required by the MELT GC, facilitates *introspective runtime reflection* [20, 21] at the MELT level; this might be useful for some future sophisticated analysis, e.g. in abstract interpretation [4, 3] of recursive functions, as a widening strategy.

The MELT runtime depends deeply upon Ggc, but does not depend much on the details of GCC main data structures like e.g. `tree`[16] or `gimple` : our `melt-runtime.c`

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

[13] A quad-core Intel Q9550 @ 2.83GHz, 6Mb cache, 8Gb RAM, fast 10KRPM Sata 150Gb disk, Debian/Sid/AMD64.

[14] In `warmelt-normal.c` this initializing `start_module_melt` routine has 27KLOC- including `#line` directives and empty lines; its compiled size is 684kbytes (about 43% of the module's binary text size).

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

[16] Actually GCC `coretypes.h` file has `typedef union tree_node *tree;` so when we speak of the details of `tree` we really mean details in `tree_node` or some other `struct tree_XXX` (with various *XXX*) in file `tree.h`, etc.

can usually be recompiled without changes when GCC's file `gimple.h` changes a little, or when passes are changed or added in GCC 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* [17] in 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 `ana-*.melt` or user MELT code), that is code implementing new passes, have to change only when the GCC API changed - 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 handle 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 data managed in C (either generated or hand-written C code within GCC^{melt}). So raw `long`-s or `tree`-s are *stuff*. Variables and [sub-]expressions in MELT code, hence locals in MELT call frames, can be of either kind (values or stuff).

Since Ggc requires each its pointer to be of a `gengtype`- known type, values are really different from stuff. There is unfortunately *no way to implement a full polymorphism* in MELT: we cannot have MELT tuples containing a mix of raw `tree`-s and MELT objects (even if both are Ggc managed pointers). This Ggc limitation has deep consequences in the MELT language (stuff sadly cannot be first-class!). And MELT cannot realistically be [22] a statically typed language like Ocaml, because designing a type system[18] for it would require some kind of type theory of the entire GCC code base!

3 MELT linguistic devices to fit into GCC.

In this section, we explain some of the various MELT values and give, thru explained code examples, an overview of some of the linguistic devices (idioms) provided in MELT to match GCC internals. Matching is covered in the next section (§4).

3.1 MELT values and stuff

Every MELT value has a *discriminant* (at the start of the memory zone containing that value). The discriminant of a value is used by the MELT runtime, by Ggc 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, etc ..., and MELT *objects*. Several *predefined objects* - e.g. `CLASS_CLASS`, `DISCR_NULLRECV` etc... - are required to be known

[17] In the old days of 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 the performance of GCC but required patching many files in it.

[18] An hypothetical MELT type system should be quite complex because of the stuff vs value dichotomy.

by the MELT runtime. As an exception, nil [19], represented by the C null pointer has conventionally a specific discriminant `DISCR_NULLRECV`. Discriminants are objects (of `CLASS_DISCR`).

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`. 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 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 short. MELT and Ggc discriminate quickly values' data-type (for marking, scanning and other purposes) thru the `obj_num` of their discriminant. Safely testing in C if a value `p` is a MELT closure is as fast as `p != NULL && p->discr->obj_num == OBMAG_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 (an hash table associating selectors to method bodies) and a parent discriminant (or super-class). Method bodies can be dynamically installed with `(install_method` *discriminant selector function*`)` and removed at any time in any discriminant or class. Method invocations go thru the method hash maps.

The MELT reader produces mostly objects: S-expressions are parsed as instances of `CLASS_SEXPR`; 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, not constant values. To have them considered as MELT values, we quote them, so (contrarily to other lisps) in MELT `2` $\not\equiv$ `'2` : the plain `2` denotes a stuff of c-type **`:long`**, but the quoted expression `'2` denotes the boxed integer 2 constant value of `DISCR_CONSTINTEGER`! As usual, a quoted symbol like `'j` denotes a constant value of `CLASS_SYMBOL`.

To associate some MELT value to some thing, hash-maps are extensively used: so hash tables keyed by objects, raw strings, or raw stuff like `tree`-s or `gimple`-s ... are values (of discriminant `DISCR_MAPOBJECTS` ... `DISCR_MAPTREES`). 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 C files 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`.

Adding a new important GCC C type like `gimple` [20] for some new stuff is fairly simple: extend the `melt-runtime.[ch]` files appropriately and add (in MELT code) a

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

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

new predefined C-type descriptor (like `CTYPE_GIMPLE` referring to keyword **`:gimple`**) and additional discriminants.

The **`:void`** keyword (and so `CTYPE_VOID`) is used for side-effecting code without results. C-type keywords (like **`:void`**, **`:long`**, **`:tree`**, **`:value`**, **`:gimple`**, **`:gimpleseq`**, 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`** stuffs and giving a **`:long`** result) is given a value or a **`:tree`** argument. And **`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.

Functions coded in MELT (with **`defun`** for named functions or **`lambda`** for anonymous ones) always return a value as their primary result. The first formal argument (if any) and the primary result of MELT functions should be values. Secondary arguments and results can be any things. The `(`**`multicall`** ...`)` syntax binds secondary results like Common Lisp's **`multiple-value-bind`**.

3.2 code chunks and primitives

These are simple constructs for C code generation.

Code chunks: They 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 very seldom used (like **asm** 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 $sta#_lab; }# )` would be translated to `{sta_1_lab: printf("i=%ld\n", curfnum[3]++); goto sta_1_lab;}` the first time it translated (`i` becoming `curfnum[3]` in C), but would use `sta_2_lab` the second time, etc. The first argument of **`code_chunk`** - `sta` here - is a *state* symbol, expanded to a C identifier unique to the code chunk's translation. The second argument here, starting with `#{` and ending with `}#` is a *macro-string*[21] and is parsed by MELT as an s-expression containing symbols (when preceded by a `$`) and strings (all the rest, which have been read verbatim).

Primitives: They 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, and at last the macro-string for the C expansion.

[21] This macro-string is exactly the s-expression `(sta "_lab:printf(\"i=%ld\\n\", " i "++); goto " sta "_lab; ")` but is much simpler to type - don't bother about intricate-d string encodings, ...

Primitives don't have state variables but are subject to normalization[22] and stuff type checking. During expansion, the formals appearing in the primitive definition are replaced appropriately.

3.3 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

```
{  gimple_simple_iterator it;
   for (it = gsi_start(s); !gsi_end_p(it);  gsi_next(&it)) {
    gimple g = gsi_stmt(it); /* do something with g */ } }
```

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 things.

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). As MELT expressions, c-iterator uses are considered **`:void`** since they are used only for their side effects.

Collecting higher-order functionals can easily be defined, using such c-iterators, by incrementally constructing their results.

There are no (Clu-like) iterators definable in pure MELT; higher-order functionals can play a similar role in practice (and MELT anonymous **`lambda`** functions are very useful).

[22] 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).

3.4 modules, environments, standard library and hooks

A single `*.melt` source file[23] is translated into a single module loaded by the MELT run-time. The module's `start_module_melt` generated 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 fields using `(`**`export_class`** ...`)` construct. Macros and pattern macros in MELT are expanded into an abstract syntax tree, not into s-expressions.

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[23]. 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[29] is implemented thru many exported definitions which can be used by clever MELT users to strongly extend the MELT language to suite even more their own needs. Language constructs[24] give total access to environments (instances of `CLASS_ENVIRONMENT`).

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

The *Parma Polyhedra Library* [1] is already used in GCC, and has been interfaced to MELT, so can easily be used by static analyzers using numerical abstractions and coded in MELT.

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`; GDBM index file interface; ...) 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.

---

[23] 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.

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

4 pattern matching in MELT

4.1 using patterns in MELT

Pattern matching [15, 16, 32] 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). And several tasks inside GCC are mostly pattern matching (like simplification and folding of constant expressions)[25].

Pattern matching: Patterns are major syntactic constructs (like are expressions and let-bindings in Scheme or MELT). In MELT, a pattern starts with a question mark, which is parsed similarly to the quote: `?x` is the same as `(question x)` [it is the pattern variable *x*], much like `'y` is the same as `(quote y)`. `?_` is the *joker pattern* (matching anything). An expression occurring in pattern context is a *constant* pattern. Patterns may be nested (thru composite patterns).

An example of pattern usage in GCC^{melt}: Many tasks depend on the form of [some intermediate internal representation of] user source code, and require extracting some of its sub-components.

For instance, our *make-green* optimization needs to track casting assignments, assignments of `stdout`, calls to `fprintf` etc.... This is easily coded with code like:

```
(let ( (:gimple g ⌈some code to get a gimple⌋) )
  ⌈display the gimple g for debugging⌋
  (match g
    (?(gimple_assign_cast ?lhs ?rhs)
      ⌈process lhs and rhs for a casting assignment⌋  )
    (?(gimple_assign_single
        ?lhs
        ?(as ?rhs
             ?(tree_var_decl ?(cstring_same "stdout"))))
      ⌈process lhs and rhs as an assignment from stdout.⌋ )
    (?(gimple_call_2_more ?lhs
       ?(as ?callfndcl
            ?(tree_function_decl ?(cstring_same "fprintf") ?_))
       ?argfile ?argfmt ?nbargs)
      ⌈handle the fprintf case to file argfile with format argfmt⌋)
    (?_ ⌈otherwise...⌋)))
```

Of course the above code is quite naive: checking that a particular call is indeed calling the standard `fprintf` by comparing the name of the called function is approximate (the user could have redefined `fprintf`) and inefficient. We should really test that the `fprintf` was declared in `<stdio.h>` etc and remember its `tree` once we parsed its standard declaration.

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

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

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 `gimple_assign_single` 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 joker `?_` pattern. Patterns can be a conjunction `?(`**`and`** $\pi_1 \ldots \pi_n$`)` (matched iff π_1 and then π_2 ... are matched) or a disjunction `?(`**`or`** $\pi_1 \ldots \pi_n$`)` (matched iff π_1 or else π_2 ... is matched) of sub-patterns π_k.

The `?(`**`as`** ...`)` pattern syntax is built-in[26] : when matching `?(`**`as`** `?`v π`)` to some thing τ, the pattern variable v is set to τ and then τ is matched against the pattern π.

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 things 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*[27] with == in generated C code on all next occurrences. This is useful in patterns like `?(gimple_assign_single ?var ?var)` to find assignments of a variable *var* to itself.

4.2 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), most composite patterns involve c-matchers: `cstring_same` ... `gimple_assign_cast` 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 c-matcher is our `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 match a string
$STR iff it equals to the constant string $CSTR.
The match fails if $STR is null or different from $CSTR.}#
#{ /*$strsam test*/ ($str && $cstr && !strcmp($str, $cstr)) }# )
```

[26] Much like expressions like `(`**`lambda`** ...`)` or `(`**`and`** ...`)` or `(`**`let`** ...`)` or `(`**`match`** ...`)` or `(`**`instance`** ...`)` are expanded thru a classical macro machinery (into some appropriate MELT abstract syntax), patterns with special pattern operators, including `?(`**`as`** ...`)`, `?(`**`and`** ...`)`, `?(`**`or`** ...`)`, or `?(`**`instance`** ...`)` are expanded thru pattern-macros or "*patmacros*" into their abstract syntax. The MELT language can be significantly extended by intrepid users defining their own macros or patmacros.

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

The first formal `str` is the matched stuff, then `cstr` is an input argument. `strsam` is a state symbol. The empty `()` indicates lack of sub-patterns. The macro-string is expanded into the test: we ensure both `str` & `cstr` are non-null to avoid crashing inside `strcmp`. There is no "fill" part, because there are no sub-patterns involved.

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

```
(defcmatcher gimple_assign_cast
   (:gimple ga)  (:tree lhs rhs)  gimpascs
   #{ /*$gimpascs test*/($ga && gimple_assign_cast_p ($ga)) }#
   #{ /*$gimpascs 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 GCC `tree`-s to sub-patterns!

C-matchers are a bit like Wadler's views[32], 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`[28] such as the pattern `?(isbiggereven` μ π`)` is matching 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 thru 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.

4.3 Matching MELT objects

In addition of the c-matcher based view-like patterns, MELT also have matching on objects (using the `?(instance ...)` syntax, ...) in a way similar to [8, 13]. For instance, the documentation generator has patterns like:

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

```
?(instance class_src_definstance :sdef_name ?dnam
     :sdef_doc ?(instance class_sexpr :loca_location ?docloc
                                :sexp_contents ?docont)
     :sobj_predef ?predef :sinst_class ?icla)
```

In such an `?(instance ...)` pattern, the [super-]class of objects to be matched is given, and some of the fields to be extracted are listed with their appropriate sub-patterns.

There is also a `?(object ...)` pattern, which is similar to the **`instance`** construct above, except that the specified class should be the exact discriminant of the matched value (so instances of sub-classes) should not be allowed.

4.4 Pattern matching implementation in MELT

While the previous sub-sections (§4.1,4.2) explained matching for a MELT user, we give here a short overview of the implementation of MELT matching facility, and how **`match`** expressions are translated into C.

The design of MELT patterns and the implementation[29] of their translation was painful to get right! But pattern matching optimization is still young [9, 15, 17, 16, 13]. The pattern language of MELT is designed semi-empirically with practical concerns in mind, that is the ability to filter easily complex GCC internal structures (i.e. stuff) and MELT values. Match-cases should be carefully ordered by the user, in particular because some c-matchers could be more general than others (and MELT has no way to know that). For instance, every `gimple` casting assign is an assign.

Some features are still missing in the MELT pattern sub-language, notably the ability to filter discriminant-s (or classes) of filtered values (or objects), when-patterns (i.e. patterns with predicative conditions on pattern variables of the match-case), regular expressions on strings, etc. We intend to add such features when needed.

To translate a `match` expression, we first compute the set of pattern variables inside each match-case's pattern [using the `scan_pattern` selector]. Then, a control tree is progressively constructed; its nodes are normalized tests (usually having both then and else successors), and the normalized sub-expression parts [inside match-cases] of the entire `match` expression are leafs. To avoid repeating the same test on the same thing twice, each matched intermediate thing (either the topmost **`match`**-ed thing, or any thing extracted in the fill step of elementary matching operators) is memoized with the sequence of tests on it. When we consider adding a new elementary test into the graph (that is, a c-matcher or fun-matcher test, an "instance-of" test, etc ...), we check that the tested thing does not have already that test, and if it does we avoid duplicating it. The expression parts in match-case are considered as terminating "always-succeed" tests.

The match translator heavily uses higher-order functionals: the `normal_pattern` selector gets a closure which is positioning the newly built tester appropriately in the control graph.

[29] In part of the file `warmelt-normal.melt` - which handles normalization of expressions, and all of file `warmelt-normatch.melt` which normalize matchers.

Once the control tree is fully constructed it can be easily translated into a graph of elementary C control blocks with many `if`-s and `goto`-s. Perhaps generating `switch`-s C statements [25] should sometimes be considered, notably using when available the fixed hash-code of constant object appearing in patterns.

5 Conclusions and future work

This work demonstrates that a huge compiler software like GCC can nevertheless be extend with a higher-level applicative language (providing anonymous and higher-order functions, objects with message passing, pattern-matching, reflection) by trying to design linguistic devices fitted to the software's coding style and practice and generating suitable C code. It is even surprising that no code generator has been widely available inside GCC middle-end before MELT. We believe this approach, provided that a garbage collector can be used or added, could be useful on other big mature software (at least when they permit plugins) when embedding a scripting language (or interfacing with another higher-level language implementation) is not easy, as is often the case with huge legacy software.

We have thus shown that the considerable existing assets of GCC can be used in extensions coded in a higher-level language, while following quite easily its evolution. This should increase the productivity of people interested in developing plugins for GCC. Of course, understanding the internals and architecture of GCC is still needed (when designing a pass - even coded in MELT- choosing its position in the jungle of all existing passes is still challenging).

The various linguistic devices (matchers, iterators, chunks, primitives) described in this paper should be general enough to be ported to other projects, when considering extending huge legacy software thru a more expressive language translated into C.

The pattern matching facilities are especially important in compilers, because many "filtering" operations on some GCC internal representation can be perceived as a pattern matching process.

Our MELT implementation is freely available (under GPLv3 licence) [27].

Future work will mostly include the use of MELT technology for compiler related issues, but MELT translator will still be extended and adapted when appropriate (possible missing language features include exceptions and persistence -perhaps thru LTO-; but GCC never uses them - i.e. `longjmp` in C - today). We should work in 2010-2011, within the OpenGPU French project, on improving GCC, thru modules to be coded in MELT, by detecting those [vectorial] routines in user code (input to GCC) which could take advantage of GPUs, and translating parts of it to OpenCL.

acknowledgments: This work has been partly funded, up to match 2009, by the French Ministry of Industry (MINEFE/DGCIS) thru the European ITEA project (IP05012) *GlobalGCC* `http://ggcc.info/`.

References

1. R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.
2. P. Collingbourne and P. Kelly. A compile-time infrastructure for GCC using Haskell. In *GROW09 workshop, within HIPEAC09,* `http://www.doc.ic.ac.uk/~phjk/GROW09/`, Paphos, Cyprus, january 2009.
3. P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Logic and Computation*, 2(4):511–547, Aug. 1992.
4. P. Cousot and R. Cousot. *Basic Concepts of Abstract Interpretation*, pages 359–366. Kluwer Academic Publishers, 2004.
5. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In *Proc. ESOP'05*, volume LNCS 3444, pages 21–30, Edinburgh, Scotland, April 2005.
6. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the ASTRÉE static analyzer. In M. Okada and I. Satoh, editors, *Eleventh Annual Asian Computing Science Conference (ASIAN'06)*, pages 1–24, Tokyo, Japan, LNCS, Dec. 6–8 2006. Springer, Berlin.
7. P. Cuoq, J. Signoles, P. Baudin, R. Bonichon, G. Canet, L. Correnson, B. Monate, V. Prevosto, and A. Puccetti. Experience report: Ocaml for an industrial-strength static analysis framework. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 281–286, New York, NY, USA, 2009. ACM.
8. B. Emir, M. Odersky, and J. Williams. Matching Objects with Patterns. In *ECOOP*, pages 273–298, 2007.
9. C. L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.
10. GCC community. GCC internals doc. `http://gcc.gnu.org/onlinedocs/gccint/`, september 2009.
11. T. Glek and D. Mandelin. Using GCC instead of Grep and Sed. In *GCC Summit*, pages 21–32, Ottawa, june 2008.
12. D. Guilbaud, E. Goubault, A. Pacalet, B. Starynkévitch, and F. Védrine. A simple abstract interpreter for threat detection and test case generation. In *WAPATV'01, with ICSE'01*, Toronto, 2001.
13. M. Hirzel, N. Nystrom, B. Bloom, and J. Vitek. Matchete: Paths through the pattern matching jungle. In *PADL*, pages 150–166, 2008.
14. R. Jones and R. Lins. *Garbage Collection (algorithms for automatic dynamic memory management)*. Wiley, 1996.
15. F. Le Fessant and L. Maranget. Optimizing pattern-matching. In *Proc. 2001 ICFP*. ACM Press, 2001.
16. L. Maranget. Warnings for pattern matching. *J. Functional Programming*, 17, May 2007.
17. L. Maranget. Compiling pattern matching to good decision trees. September 2008.
18. G. Marpons-Ucero, J. Mariño-Carballo, M. Carro, Á. Herranz-Nieva, J. J. Moreno-Navarro, and L.-Å. Fredlund. Automatic coding rule conformance checking using logic programming. In P. Hudak and D. S. Warren, editors, *PADL*, volume 4902 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2008.
19. B. Monate and J. Signoles. Slicing for security of code. In *TRUST*, pages 133–142, 2008.
20. J. Pitrat. Implementation of a reflective system. *Future Gener. Comput. Syst.*, 12(2-3):235–242, 1996.
21. J. Pitrat. *Artificial Beings (the conscience of a conscious machine)*. Wiley / ISTE, march 2009.

22. F. Pottier. private communication, september 2008.
23. C. Queinnec. *Lisp in Small Pieces*. Cambridge Univ. Pr., New York, NY, USA, 1996.
24. N. Ramsey and S. P. Jones. A single intermediate language that supports multiple implementations of exceptions. In *Proc. PLDI '00*, pages 285–298, New York, NY, USA, 2000. ACM.
25. R. A. Sayle. A superoptimizer analysis of multiway branch code generation. In *GCC Summit*, pages 103–116, Ottawa, june 2008.
26. M. Sperber, R. K. Dybvig, M. Flatt, A. Van Straaten, R. Findler, and J. Matthews. Revised^6 report on the algorithmic language Scheme. *J. Functional Programming*, 19(S.1):1–301, 2009.
27. B. Starynkevitch. MELT code [GPLv3] within GCC. `http://gcc.gnu.org/wiki/MiddleEndLispTranlator` and `svn://gcc.gnu.org/svn/gcc/branches/melt-branch`, 2006-2009.
28. B. Starynkevitch. Multi-stage construction of a global static analyzer. In *GCC Summit*, pages 143–156, Ottawa, july 2007.
29. B. Starynkevitch. Middle End Lisp Translator for GCC, achievements and issues. In *GROW09 workshop, within HIPEAC09,* `http://www.doc.ic.ac.uk/~phjk/GROW09/`, Paphos, Cyprus, january 2009.
30. G. L. Steele. *COMMON LISP: the language*. Digital Press, 1984. With contributions by Scott E. Fahlman and Richard P. Gabriel and David A. Moon and Daniel L. Weinreb.
31. A. Venet and G. Brat. 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*, pages 231–242, New York, NY, USA, 2004. ACM Press.
32. P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Proc. POPL '87*, pages 307–313, New York, NY, USA, 1987. ACM.

`$Id: MELT-Basile-Starynkevitch-CC2010.tex 112 2010-01-23 10:10:17Z basile018 $`