A Middle End Lisp Translator for GCC
achievements and issues

Basile STARYNKEVITCH

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

Abstract. Some extensions of GCC, like those for static analysis, cannot easily be coded in C. A Lisp-like dialect has been designed and implemented (as a translator to C) to fit well into the GCC middle-end. We give key language features, and improvements.We describe the design of its runtime system fitting into GGC. Idioms to interoperate with the evolving GCC internal API are explained.

1 Introduction and motivations

Program analysis and symbolic processing are a well known application domain of functional and lisp-like languages. Hence compiler and static analysis tools[1] benefit from being written in such languages, e.g. in Ocaml. However, industrial compilers (like GCC[2]) for usual procedural or object-oriented languages (like C, C++, Ada, Fortran, Java) are generally huge software (GCC is almost four million lines of source code, mostly C), and interfacing a real functional language (like Ocaml) to them is not practical because of an impedance mismatch : all their internal data of GCC's middle-end are C structures which are not compatible with Ocaml runtime, so to interface them to e.g. Ocaml would require boxing[3] data-structures and wrapping routines for each access to (or update of) such data. This is error-prone, inefficient, and requires a lot of efforts. There are also typing issues: designing a (statically typed) ML dialect compatible with GCC internals requires the formalisation of a type system suitable for (actual or future) GCC data representations. This is not easily achievable[3].

Extending a real compiler like GCC for various needs, outside of usual code generation, e.g. static analysis (as illustrated by ASTRÉE or FRAMA-C [1]), is very appealing, since it would benefit from most of the efforts put into GCC : various front-ends[4], many middle-end analysis, representations, and transformations already existing in GCC. But

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

[1] Abstract interpretation tools like FRAMA-C `http://frama-c.cea.fr/` [1] or ASTRÉE `http://www.astree.ens.fr/` [2], both coded in Ocaml.

[2] The Gnu Compiler Collection, at `http://gcc.gnu.org/` which is used on most Linux and many others systems.

[3] To box a data means to make a pointer to some container of the data.

[4] For instance, extending ASTRÉE or FRAMA-C to eat C++, Ada and Fortran source code would require an unrealistic effort.

coding ambitious analysis passes inside GCC as it is done today (in C) is not very practical (as learnt in TWO[4]), especially for “static analysis” goals which are not intimately related to the usual code generation aim of GCC, and which will have fewer developers.

Hence, it should be interesting to provide some high-level, functional[5], language to permit prototyping and development of special middle-end passes, and the peculiarities of GCC require such a language to be quite tailored to the existing GCC code base. This approach is complementary to other ongoing GCC plugin efforts.

So this paper describes a Lisp[5] dialect, MELT[6][6] which has been designed and implemented to fits well into the GCC compiler (MELT was originally called “basilys” hence many function names and files still start with `basilys`). Some familiarity with the GCC compiler is expected from the reader. The GCC compiler branch with MELT enabled is noted GCC$^{\text{melt}}$.

Adding a Lisp dialect into a big program is not new: consider GNU Emacs. And Greenspun’s Tenth rule is well known [7]. MELT is compiled to C (not interpreted), to fit very tightly into GCC internals and also for efficiency. MELT is used inside some specific GCC “optimisation” passes, which are only invoked when GCC$^{\text{melt}}$ is run with an explicit `-fbasilys` flag, and which may do “arbitrary” processing on GCC internals (e.g. some static analysis with user warnings). All of the MELT code should be explicitly enabled at GCC build time (thru arguments to `configure`)[8] and at GCC run time.

2 Overview

We assume that the host system (running the GCC$^{\text{melt}}$ compiler, e.g. Linux) supports plugins (with `dlopen` or `lt_dlopenext`). The figure shows MELT in action. A MELT source file `ana-base.bysl` defines some MELT functions and MELT passes. It is translated into C code `ana-base.c`. This C code is compiled[9] into a shared object `ana-base.so`. When GCC$^{\text{melt}}$ compiles or analyses some (e.g. C, C++, Ada) source file[s] it dynamically loads this `ana-base.so` (and others) plug-in[s], and may apply the functions defined there inside some MELT specific passes. The MELT translator is

[5] In this paper, a functional programming language is a language emphasising application of functions, in particular permitting higher-order and first-class functions [e.g. a λ construct]. We still do use object orientation and data mutability.

[6] Middle End Lisp Translator branch of GCC: see `http://gcc.gnu.org/wiki/MiddleEndLispTranslator` and `svn+ssh://gcc.gnu.org/svn/gcc/branches/melt-branch` to get the code.

[7] See `http://en.wikipedia.org/wiki/Greenspuns_tenth_rule`

[8] The MELT branch `configure`-d without enabling MELT is almost the same as the GCC trunk : unless enabled, MELT code is not even linked into `cc1`.

[9] Practically, compilation of MELT generated C code into a dynamically loadable plug-in is done thru a script `melt-cc-script` which may be invoked by GCC$^{\text{melt}}$.

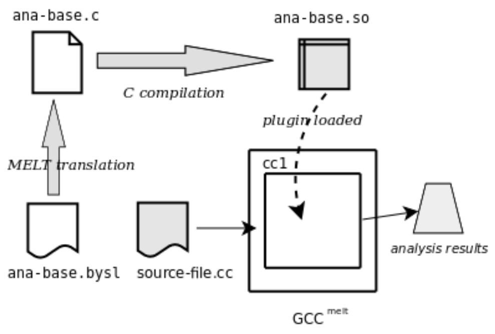

Fig. 1. GCC$^{\text{melt}}$ overview

bootstrapped[10], it is coded in MELT, and constitutes a real test of the GCC$^{\text{melt}}$ infrastructure.

A MELT function giving the execute function of a GCC pass can be something like the following code, which issues a warning for every occurrences of `fprintf(stdout,...)` in the analysed source code:

```
;; execution of IPA pass
(defun simpana_ipaexec (ipapass)
  (do_each_cfun_body
   (lambda (declv bodyv) ;;; anonymous function
     (debug_msg declv "simpana_ipaexec eachcfunbdy declv")
     (debug_msg bodyv "simpana_ipaexec eachcfunbdy bodyv")
     (walk_gimpleseqv◇ bodyv
       (lambda (gimpv)
         (match gimpv
             (?gimple_callv◇ fprintf_symb
                 (?treeseq_starting stdout_symb))
             (warning (gimplev_location gimpv)
               "fprintf could be replaced by printf")))))))
```

The `do_each_cfun_body` function called above takes a function as an argument, here the anonymous function given thru the first **lambda** and apply it to every cfun declaration and body. `debug_msg` is used to nicely print a MELT value for debugging, with a user message. The function `walk_gimpleseq_v`◇recursively walks thru a gimpleseq and apply a function (the second **lambda**) to every gimple. The **match** expression uses pattern matching to find `printf` calls with `stdout` and then issue a warning.

[10] Technically, MELT *is not a GCC front-end* (it generates C code, not Generic or Gimple internal representation), and the MELT translator is actually a special invocation of `cc1` or `gcc` without any input files (the MELT source file is passed as a special argument)! A very first implementation of MELT was coded in CLisp.

To use GCC$^{\text{melt}}$, run `gcc -fbasilys=simpana -O -c source-file.cc`. The `-fbasilys=simpana` program argument triggers the `simpana` MELT command[11]. This command runs the following MELT function, which installs into the `basilys_ipa_gccpass`[12] a MELT gate and execution function. A more flexible pass manager (like in MilePost[7]) would be welcome.

```
;;; our simple analysis command
(defun simpana_command (dispatcher arg secarg moduldata)
  ;; fill the ipa pass by setting fields in the gcc-pass object
  (put_fields basilys_ipa_gccpass
      :gccpass_gate simpana_ipagate   :gccpass_exec simpana_ipaexec)
  ;; debug printing of the updated pass
  (debug_msg basilys_ipa_gccpass "basilys_ipa_gccpass in simpana"))
;; install the above function as a command
(initial_command_install simpana_command "simpana")
```

Several MELT specific optimisation passes have been added inside GCC$^{\text{melt}}$. When GCC$^{\text{melt}}$ processes a C (or C++, or Ada...) source file, these additional passes may execute MELT code, if a MELT command function like above has enabled them. The `initial_command_install` invocation above illustrates that MELT handle a mix of MELT [boxed] *values* (like the `simpana_command` function, i.e. a MELT closure[13]) and [unboxed] non-value *stuff* like read-only constant strings (implemented as in C like `const char[]` strings), `gimple`[14] or `long`-s. Non-value stuff can also be GCC internal data (e.g. `gimple` or `tree`). Boxed values and unboxed stuffs are called *thing*s here.

3 Major MELT language features by examples

MELT is a Lisp dialect, so every language construct is written in parenthesis: `(`*`operator arguments...`*`)`. Assuming some familiarity with Scheme, Common Lisp or some other Lisp dialect, we give here salient MELT peculiarities: handing of values and stuffs §3.1, the object system §3.2, multiple results §3.3, values and discriminants §3.4, other features such as matching §3.5, bindings and modules §3.6.

3.1 boxed values and unboxed stuff

MELT handles both boxed values (allocated and handled by MELT's copying garbage collector) and unboxed stuff. Unboxed stuff is given a *c-type* thru keywords like e.g.

[11] Many other program arguments are available, e.g. `-fbasilys-dynlibdir` to explicit the MELT plug-in directory, `-fbasilys-init` to give the list of plug-ins to load, etc.... Read the internal GCC$^{\text{melt}}$ documentation for details.

[12] A MELT object, of class `class_gcc_pass`, reifing the MELT-enabled pass after IPA.

[13] A closure [5] is a value implementing a function by packing some code [e.g. a routine] with the closed variables occurring in the function.

[14] One of the main GCC middle-end internal representation `http://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html`

`:gimpleseq` or `:long` or even `:void` for side-effecting only constructs. For instance, the `(f :gimpleseq gseq)` formal argument list declares a MELT value formal `f` and a `gimple_seq` C stuff formal `gseq`. MELT function application and message sending give the type of arguments (or secondary results). Pattern matching$^{\diamond}$[15] is possible on both values and stuffs.

3.2 MELT object system : e.g. s-expressions and c-types

MELT provides a single-inheritance object system. MELT objects are values, and have value fields, and an object number or *objnum*[16], which is a `short` unboxed stuff "field". There is a single root class `class_root`. For instance, source s-exprs (as parsed by the MELT reader) are objects. Their class is defined as:

```
(defclass class_located ;; class of located stuff
  :super class_proped          :predef CLASS_LOCATED
  :fields (loca_location))     ;its source location
(defclass class_sexpr ;; class of source expressions
  :super class_located         :predef CLASS_SEXPR
  :fields (sexp_contents))     ;list of contents
```

The optional `:predef` keyword gives a symbolic name to some few predefined values known by the C runtime. The `class_proped` super-class provides property lists. MELT symbols and keywords are reified and interned (by the parser) as instances of `class_symbol` and `class_keyword`. Fields and classes are themselves objects (of `class_field` and `class_class`). Once read, MELT s-expressions, in particalar those using basic language constructs, are macro-expanded into abstract syntax trees (using the macro machinery).

MELT c-types are reified as instances of `class_ctype` but should also be supported by the MELT runtime (i.e. files `gcc/basilys.[ch]`) and GC. For example, GCC `gimple` have the following c-type[17]:

```
;; the C type for gcc gimples
(definstance ctype_gimple class_ctype
  :predef CTYPE_GIMPLE
  :named_name  (stringconst2val discr_string "CTYPE_GIMPLE")
  :ctype_keyword ':gimple ;; keyword for the type
  ;; the C name of the type
  :ctype_cname  (stringconst2val discr_string "gimple")
  ;; how to pass such stuff as arguments or secondary results
  :ctype_parchar (stringconst2val discr_string "BPAR_GIMPLE")
  :ctype_parstring
    (stringconst2val discr_string "BPARSTR_GIMPLE")
  :ctype_argfield (stringconst2val discr_string "bp_gimple")
  :ctype_resfield (stringconst2val discr_string "bp_gimpleptr")
```

[15] The $\diamond$ indicate features designed but not entirely implemented in october 2008.

[16] The objnum is available for any use, and usually mutable. For classes or discriminants, it is fixed and plays a particular role.

[17] The `stringconst2val` primitive boxes a string, i.e. makes a string value from of a constant C string stuff.

```
  ;; the GGC marking routine
  :ctype_marker
   (stringconst2val discr_string "gt_ggc_mx_gimple_statement_d"))
(install_ctype_descr ctype_gimple "GCC gimple pointer")
```

Adding new c-types is fairly simple (just patch files `gcc/basilys.[ch]` and `gcc/melt/warmelt-first.bysl` appropriately).

The `definstance` construct is used to define static instances (bound inside a MELT module). To build an instance dynamically (e.g. in a function), use the `instance` construct. For example, the `definstance` construct is handled by a MELT macro which builds

```
  (instance class_src_definstance  :src_loc loc   :sdef_name symb
     :sobj_predef predef  :sinst_class cla  :sinst_clabind clabind
     :sinst_objnum objnum  :sinst_fields fastup)
```

Fields of MELT objects are given by keywords like `:src_loc`; practically, field names and class names are *globally unique*.

3.3 Multiple results, e.g. for normalization

The translation of MELT code into C (see below §4) illustrates the need for returning multiple results in one of its important steps, normalisation. Normalisation of MELT abstract syntax trees is an example of multi-valued message sending. A message selector is declared with `(`**`defselector`**` normal_exp class_selector)` and then used e.g. for normalising a function application (of `class_src_apply`)

```
;; normalize an application
(defun normexp_apply (ap env ncx psloc)
  (let ( (sloc (get_field :src_loc ap)) (sfun (get_field :sapp_fun ap))
         (sargs (get_field :sapp_args ap)) )
    (multicall (nfun nbindfun) (normal_exp sfun env ncx sloc)
     (assert_msg "check nbindfun" (is_list_or_null nbindfun))
     (multicall (nargs nbindargs)
      (normalize_tuple sargs env ncx sloc)
     ;;; etc....
))))
;; install this as method for normal_exp in class_src_apply
(install_method class_src_apply normal_exp normexp_apply)
```

In the code above, **let** introduces sequential[18] scoped bindings (and could even bind unboxed stuff like `(:long i 1)`), `get_field` safely fetches a field from inside object[19], `normal_exp` is in a message send and `normalize_tuple` is a function application. Both return multiple values, captured and bound by **multicall** constructs[20].

[18] So our `let` is more like a `let*` in Scheme or Common Lisp.

[19] `get_field`, returns nil if accessing a non-object, or an object of irrelevant class. The `unsafe_get_field` variant is faster but don't check anything and may crash.

[20] MELT's `multicall` is inspired by Scheme's `call-with-values` or Common Lisp's `multiple-value-bind` constructs.

The first operand of **multicall** is, like a formal arguments list, binding variables to the primary and [possibly several] secondary results. So the `normal_exp` send gives `nfun` as its primary result, and `nbindfun` as its first secondary result; indeed normalisation returns an nrep and a binding list. Method implementations are installed by `install_method` in a method dictionary (an object hash map associating selectors to closures) and can be installed or removed very dynamically.

3.4 various values and their discriminant

MELT provides many other kinds of values, including closures, tuples (i.e. fixed sequence of values), boxed stuff (i.e. containers for C `gimple`, `gimple_seq` ...or `long` data), pairs (like cons in Lisp - the head is any value, the tail is a pair or nil), lists (containing the first and last pair, so appending or prepending is fast), various hash maps (hash-tables associating an object to a non-null value, or a `gimple` or `tree` or string to a non-null value.), etc.... Hash maps are used to unintrusively associate information; for example, associating MELT values to GCC `gimple` is done with gimple maps (hashing `gimple` to MELT values.), without adding any extra MELT specific C field into `struct gimple_statement_base`.

Each MELT value has a *discriminant* which is an object ([in-]directly an instance of `class_discr`). The discriminant of nil is `discr_nullrecv`, the discriminant of a MELT instance is its class. The objnum of the discriminant is used by the runtime (including GGC) to discriminate marking or tracing of pointers. Discriminant-s are also used by message sending (so a message can even be sent to a MELT list, etc...).

3.5 other lispy features : conditional and matching

MELT have many other features similar to other Lisps, such as conditionals (`cond`, `if`, `and`, `or`), the `quote` machinery[21] and notation (`'x` is the same as `(quote x)`), infinite loop `forever` and a local `exit`, the `return` operation, value and stuff assignment thru `setq`, sequential side-effecting evaluation thru `progn`, etc.... But *MELT does not provide tail-recursion* or a `letrec` construct for local [mutually] recursive functions[22]. Also, most MELT primitives or standard functions do not have a Lispy name: e.g. the head of a pair is `pair_head` not `car`. And *MELT is neither polymorphic nor polytypic* : no support of a `map` operation usable on every data - because of the essential difference between boxed values[23] and unboxed stuff, and no variable argument list (like `<stdarg.h>` and `...` in C or `&rest` in Common Lisp).

[21] Notice that a string e.g. `"hello"` in MELT source code is an unboxed `:cstring` stuff, and `15` is an unboxed `:long` stuff; its quotation `'"hello"` should$^\diamond$denote a constant boxed string value, and `'15` should$^\diamond$denote a constant boxed integer value..

[22] We are considering implementing a `localdef` construct, e.g. `(localdef (defun odd (x) ...) (defun even (y) ...) (odd '5))` but cross-recursion à la `letrec` seems rarely useful for our purposes.

[23] If restricted to MELT values, one could implement a quite "generic" `map` as a message selector and install suitable methods for it; this is possible because even non-object values like tuples or lists have a discriminant used in message sending.

MELT also has a *pattern matching*$^\diamond$ability. Patterns use the `(question π)` notation, same as `?`π[24], in the `match` construct. A toy example is:

```
(defun testmatch (v x)
  (match v
      ( ?(or
          ?(instance class_named :named_name ?n)
          ?(tuple_sized 1))
         (debug_msg n "testmatch firstcase n"))
      ( ?(tuple_nth 0 ?c)
         (debug_msg c "testmatch secondcase c"))
      ( x   (debug_msg x "testmatch got the x"))
      (?_   (debug_msg v "testmatch lastcase v"))))
```

A `match` construct has a sequence of match-cases. Each match cases has a pattern and a sequence of expressions. When the matched value (`v` in code above) match a pattern, the pattern variables (like `?n` or `?c` above) appearing in the case are locally bound to their matching values. Patterns may be abstract views [8] or non-linear (containing several occurrences of the same pattern variable). The joker `?_` matches anything. A non-pattern (like `x` above) is matched by testing for identity.

3.6 reified bindings, environments and modules

Like Scheme, and unlike Common Lisp, MELT has a single name space. MELT environments and bindings are reified (thru `class_environment` `class_any_binding` etc…). A name is bound to a value (imported from a previous module), a function [inside the current module], a primitive, a class, an instance, a selector, a local let or formal variable, a pattern variable, a loop label, a macro, a pattern macro, a matcher [or pattern view], a c-iterator, or a field. Any previous binding is hidden. A module (e.g. produced by translation of a single MELT source file) is importing values and macros and may export (using `export_values` `export_class` `export_macro`) some bindings. The very first module (file `gcc/melt/warmelt-first.bysl`) is particular and translated specially. The bootstrapped MELT translator is implemented in several files `gcc/melt/warmelt-*.bysl` or more than 22KLOC of MELT code. The generated[25] files `gcc/warmelt-*-0.c` (nearly 400KLOC) are also in the source repository. Most of the generated code is in big initialisation routines (since MELT does not generate any data file; all initial data is built by generated C code). Each module initialisation (generated C) routine is called only once (when the shared-object is loaded), it takes the previous environment as an argument, and returns the newly reified current environment (with all the definitions exported by the module). Generating C code from MELT goes fast, the bottleneck is the compilation of the generated C code into shared objects.

[24] So `question` and `?` notations are equivalent, like `'` and `quote` are.
[25] Run `make upgrade-warmelt` to regenerate them.

4 Fitting into GCC runtime and translating into C.

We need to play nice with the GCC runtime system, i.e. all the utility code of the GCC platform. In particular, GCC contains a garbage collector[26][9], called GGC[27] (for GCC Garbage Collector) , and we need to fit into it. GGC is a precise mark and sweep garbage collection mechanism[28], only triggered by *explicit* calls to `ggc_collect`: ordinary allocations inside GCC never trigger any GGC garbage collection. And GGC does not maintain any local roots (e.g. local variables on the machine stack frame when GCC is running.). Only global (and static) roots are considered when `ggc_collect` is called. This is acceptable because such calls happen mostly between GCC optimisation passes, not inside them.

But any functional language allocates a lot of temporary data, e.g. closures (most of which will die quickly), so calling the garbage collector only outside of MELT code would be impractical: MELT needs to collect data when it has allocated enough memory, explicitly tracing *local* variables. Also, fast allocation is important, since most MELT data is expected to die quickly. Therefore, a *generational copying garbage collector* [9] has been implemented above the native GGC. Such a collector requires explicit handling of local data (since local pointers are updated by the MELT collector), including intermediate temporary values. This means that each MELT call frame is an explicit C `struct`-ure containing all the local pointers; This is unconvenient in hand-written C code[29] but easy in generated C code.

MELT copying generational garbage collector is backed up by the GGC collector. When the allocation region (young region, or nursery) is full enough, live MELT values are copied outside, into the GGC (mark&sweep) heap, and then the nursery is freed. Allocation of MELT values is fast (in-lined `basilysgc_allocate` function); often incrementing the current allocation pointer, and comparing it to the limit of the nursery. A MELT garbage collection is often minor or more rarely full (the nursery is copied into GGC heap, and then GGC collection is explicitly done.). If only MELT values are allocated, this is enough, provided all local MELT values (inside MELT call frames) are copied into a `GTY`-ed global vector visible to GGC. This happens when no GCC stuff is allocated (e.g. in "code analysis" MELT passes). Should some GGC stuff (like `gimple`) be allocated in a MELT pass (e.g. some expensive optimisation pass using MELT static analysis), it should be explicitly known to the GGC collector, which is called by the MELT collector at arbitrary times. This could be done by copying it elsewhere (requiring a global vector for each type of stuff, e.g. a vector of `gimple`-s, another of `tree`-s, etc...). A simpler solution is to enhance `ggc_collect` as a function `ggc_collect_extra_marking` called with a walking function and its data. The walking function calls appropriate GGC marking functions on the data. For

[26] Garbage collection is essential in every high-level language implementation; if GCC didn't had one, we would have to code it from scratch.

[27] See `http://gcc.gnu.org/onlinedocs/gccint/Type-Information.html` and files `gcc/ggc.h` and `gcc/ggc*.c` in the GCC source tree.

[28] The marking routines are generated by `gengtype` using `GTY` annotations in C header files inside GCC.

[29] This is why macros `BASIYS_ENTERFRAME` and `BASILYS_EXITFRAME` are often used in hand-written code in `gcc/basilys.c` for MELT basics.

MELT, the walking function scans all MELT call frames (thru MELT generated code) and mark there all the MELT values and the other stuff. With such a simple enhancement to GGC (about 20 lines changed in `gcc/ggc*.[ch]`)[30] copying of MELT local values into a global vector is no longer needed. The requirements of MELT runtime are easily handled in the generated C code by MELT translation to C.

MELT translation to C is happening in a number of steps. It starts with macro expansion, which builds (from s-exprs) an AST [abstract syntax tree] represented by instances of sub-classes of `class_src` (e.g. `class_src_definstance` above §3.2). Then this AST is normalised, into *n-rep*s (normal representations, sub-classes of `class_nrep`) which add many additional normal let bindings for intermediate temporary values, transforming `(f (g x) y)` into some normal representation like **let** $\gamma =$ `(g x)` **in** `(f` γ `y)`[31] where γ is a fresh cloned symbol. Normalisation is required by the precise copying MELT collector. The abstract normal tree (of nrep-s) is still expression-like. It is further transformed into mostly a forest of statement-like representations, called *objcode*-s, instances of subclasses of `class_objcode`. At last the forest of objcode-s is pretty-printed as (unreadable) C code.

5 MELT idioms to fit into GGC code in C.

Several MELT idioms, quite specific to MELT and unavailable in other Lisp implementations, are provided to fit very well into the existing GCC code. These low-level constructs describe how to translate some elementary s-expressions (suitably normalised) into C code.

Every low-level operation is done by *primitives* which describe how to translate their use into C code. For instance,

```
(defprimitive is_gimple (v) :long
  "(basilys_magic_discr((basilys_ptr_t)("
  v ")) == OBMAG_GIMPLE)")
(defprimitive make_gimple (discr :gimple g) :value
  "(basilysgc_new_gimple((basilysobject_ptr_t)("
  discr "),(" g ")))")
```

defines a primitive `is_gimple` to test if a value is a boxed gimple, and a primitive `make_gimple` to build a boxed gimple value out of a discriminant and an unboxed `gimple` stuff.

Iterative C constructs like `for` are described by *c-iterator*s. For instance,

```
(defciterator each_in_gimpleseq
  (:gimpleseq gseq) ;start formals
  eachgimplseq                         ;state symbol
  (:gimple g) ;local formals
  ( ;;; before expansion
   "gimple_stmt_iterator gsi_" eachgimplseq ";"
```

[30] See `http://gcc.gnu.org/ml/gcc/2008-10/msg00231.html` and following messages.

[31] This is a textual simplification. The actual abstract normal tree (ANT) is made of instances of sub-classes of `class_nrep`.

```
   "if (" gseq ") for (gsi_"
       eachgimplseq " = gsi_start (" gseq
        "); !gsi_end_p (gsi_" eachgimplseq ");"
   " gsi_next (&gsi_" eachgimplseq ")) "
    g " = gsi_stmt (gsi_" eachgimplseq ");"  )
  ( ;;; after expansion
   "" ))
```

defines a c-iterator called `each_in_gimpleseq`. The description gives how to generate a C block (usually a `for` C statement). Using a c-iterator is even simpler:

```
;; apply a function to each boxed gimple in a gimple seq
(defun do_each_gimpleseq (f :gimpleseq gseq)
  (each_in_gimpleseq
   (gseq) ;input
   (:gimple g)                  ;local variables list
   (let ( (gplval (make_gimple discr_gimple g)) )
     (f gplval))))
```

As an expression, a c-iterator invocation has the `:void` c-type, because it don't give anything when evaluated. Such an invocation is given a list of input arguments [just `(gseq)` here], and a list of locally bound variables [just `(:gimple g)` here]. The c-iterator definition also uses a state symbol, which is translated into a uniquely generated C identifier (this enables arbitrary nesting of c-iterator invocations).

In patterns, atomic constructs are doing a test (does the matched thing indeed match) and some extractions or fillings (getting the matched things for sub-patterns). The corresponding items are *c-matcher*s.

```
(defcmatcher tuple_nth
  (tup :long rk) ;match & ins
  (comp) ;out
  tupnth ;statesymb
  ( ;test expansion
   "(basilys_is_multiple_at_least(((basilys_ptr_t)"
   tup "), (int)(" rk ")))" )
  ( ;fill expansion
   comp " = basilys_multiple_nth((basilys_ptr_t)("
    tup "),(int)(" rk "));" ))
```

which describes the code for testing and the code for filling.

Atomic pattern constructs implemented in MELT (not in C) are possible, using *funmatcher*s$^\diamond$. These are implementing the matching with a function returning primarily[32] a non-null value if the match succeeded, and secondarily the extracted/filled things. So funmatchers also gives the signature of secondary results of the MELT function implementing them.

Pattern macros or *patmacro*s generalise MELT macros: they describe how to expand an s-expr in pattern position and in expression position. Actually, `or` is a patmacro since

[32] In MELT the primary result of a function or message, the first [if any] argument of a function, the receiver of a message send, is always a MELT value and cannot be a stuff.

it is useful both in expressions and in patterns. Macros and patmacros are defined by `export_macro` and `export_patmacro` which export specific bindings.

In practice, the idioms above have been general enough to follow the evolution of the GCC trunk during 2008, even during major disruptive stage 1 enhancements (like tree $\rightarrow$ tuple, or graphite). Merging them into MELT was done in few days.

6 Conclusions.

The above MELT language features, including all the idioms (or dirty tricks) specific to MELT and describing translation into C, have been experimentally sufficient to follow major GCC evolutions, and are built above a fairly stable GCC low-level infrastructure (notably GGC), which is "abstracted" in the MELT runtime. So adding a new or evolving API inside GCC is fairly easy. MELT is usable not only for static analysis (using PPL[10]) but also for other tasks like [11] (code refactoring, stylistic checking, ...).

Acknowledgements

This work has been partly funded by the French Ministry of Industry (MINEFE) thru the ITEA project (ref.IP05012) GlobalGCC `http://ggcc.info/`. Thanks to Guillem Marpons, Roberto Bagnara, and to the reviewers for helpful comments.

References

1. Monate, B., Signoles, J.: Slicing for security of code. In: TRUST. (2008) 133–142
2. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE Analyser. In Sagiv, M., ed.: Proc. European Symposium on Programming (ESOP'05). Volume 3444 of LNCS., Edinburgh, Scotland (April 2–10 2005) 21–30
3. Pottier, F. private communication (september 2008)
4. Guilbaud, D., Goubault, E., Pacalet, A., Starynkévitch, B., Védrine, F.: A simple abstract interpreter for threat detection and test case generation. In: WAPATV'01, with ICSE'01, Toronto (2001)
5. Queinnec, C.: Lisp in Small Pieces. Cambridge Univ. Pr., New York, NY, USA (1996)
6. Starynkevitch, B.: Multi-stage construction of a global static analyzer. In: GCC Summit, Ottawa (july 2007) 143–156
7. Fursin, G., et al.: MILEPOST GCC: machine learning based research compiler. In: Proceedings of the GCC Developers' Summit. (June 2008)
8. Wadler, P.: Views: a way for pattern matching to cohabit with data abstraction. In: POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, New York, NY, USA, ACM (1987) 307–313
9. Jones, R., Lins, R.: Garbage Collection (algorithms for automatic dynamic memory management). Number ISBN 0-471-94148-4. Wiley (1996)
10. Bagnara, R., Hill, P.M., Zaffanella, E.: 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) (2008) 3–21
11. Glek, T., Mandelin, D.: Using GCC instead of grep and sed. In: GCC Summit, Ottawa (june 2008) 21–32