MELT = a Lisp matching GCC internals

Basile STARYNKÉVITCH

CEA LIST (Saclay, France)

basile@starynkevitch.net

Rocquencourt [Gallium], October 12th 2009,
Paris Chevaleret [PPS], November 5th 2009

Basile STARYNKÉVITCH

MELT = a Lisp matching GCC internals
Outline

Introduction and motivations
- GCC today
- GCC features and internal properties
- Future GCC plugins and MELT motivations

Using MELT and its runtime
- Using MELT
- MELT runtime

MELT linguistic devices to fit into GCC
- MELT values, objects, and stuff
- Code chunks and primitives
- MELT syntax overview
- C-iterators
- Modules, environments, standard library, hooks

Pattern matching in MELT
- Using pattern matching
- C-matchers and fun-matchers
- Overview of pattern matching translation

Conclusions and future work
Introduction and motivations

Using MELT and its runtime

MELT linguistic devices to fit into GCC

Pattern matching in MELT

Conclusions and future work

GCC today

GCC

2

GCC = a huge, growing, legacy, free software

version

date

gcc1 binary

GCC today

4.2.1

july 2007

44.1Mb

2956KLOC

4.4.1

july 2009

62.9Mb

3844KLOC

trunk.r.152437
	october 2009

60Mb

3962KLOC

GCC2 = a huge, growing, legacy, free software

with a large developer community and strict social rules

4

400? maintainers but no "benevolent dictator"

Basile STARINKEVITCH

MELT = a Lisp matching GCC internals
GCC features

Next release **4.5** ≡ trunk (end 2009?):

- GPLv3 licence, FSF copyright
- *several source languages* (C, C++, Ada, Fortran, Objective-C, ...) and dialects (C++0x, OpenMP, ...)
- more than **30 target architectures** (x86/amd64, ARM, Sparc, Itanium, ...), on many systems (Linux, Solaris, ...)
- straight or *cross* (or canadian) compiler
- *many optimizations*\(^5\) [polyhedric loops, field reordering, ...]
- *infrastructure* for [GPL licensed] *plugins*\(^6\)
- *link-time optimization*\(^6\)

\(^5\)Some should be explicitly enabled even in \(-O3\). Tuning them with machine learning techniques in MILEPOST
\(^6\)New in 4.5
GCC internal properties

Internals:

- several *evolving* internal representations (IR):
  1. *Generic* inside front-ends
  2. *Gimple* and *Tree* inside middle-end
  3. *RTL* inside back-ends

- **unstable** and *messy* internal *API* (for extensions or plugins)

- **Ggc** = GCC Garbage Collector\(^7\), *don’t manage* local pointers! Marking generated by `gengtype` on GTY-ed C structs

- many global variables, C data structures, header files

- huge *legacy*

- more than 200 passes moulding and transforming the IRs

\(^7\)precise, mark & sweep, explicit GC.
Possible plugins uses:

1. prototyping future GCC extensions and optimizations
2. activities outside of code generation:
   - library specific optimization, e.g.:
     `fprintf(stdout, ...) → printf(...)` after inlining
   - library specific warnings, e.g.:
     checking Ocaml runtime GC conventions?
   - coding rules (MISRA-C, ...) validation
     G.Marpons et al. UPM - Madrid
   - code refactoring (Mozilla TreeHydra, T.Glek et al.)
   - static analyzers, threat detectors, code metrics, smart code browsing, ...
3. *everything but the kitchen-sink* for source program handling
4. GCC plugins [≡ motorcycle] complementary (not competitor) of high-tech
   [≡ Rolls-Royce] static analyzers (Frama-C, Astrée, ...)

Basile STARYNKÉVITCH

MELT = a Lisp matching GCC internals
Pros and conses of GCC plugins

Advantages:

- **ease of use** (just extend `CFLAGS` in users’ `Makefile`)
  wide availability and familiarity about GCC

- numerous **GCC IRs and passes**; at some point:
  GCC has already parsed, inlined, const-fold-ed, normalized, SSA, user code

- **GCC legacy**: front-ends, back-ends, middle-end

- **GCC infrastructure** for your plugin: language extensions (attributes, pragmas, ...), internal hooks (inserting your pass)

- **GCC community** support

---

8 E.g. a *Gimple*-based analyzer handles C, C++, Ada, Fortran, ...

Basile STARYNKÉVITCH  MELT = a Lisp matching GCC internals
Shortcomings of GCC plugins:

- *complexity and instability* of GCC internals API
  - What IR to work on? Where to insert your pass?
- low-level C programming language inside GCC
- coding plugins in *high-level languages* (Ocaml, Haskell) is *not realistic* [glue to small part of API or huge heavy glue]
MELT insight and motivations

Insight:

- design a *Lisp-like dialect translated into C, following usage* of GCC internals
- **MELT** = Middle End Lisp Translator
- *generate C code suitable* for GCC thru *linguistic devices*

Motivations and experience:

- *provide higher level language features* (lambdas, pattern-matching, objects, ...) to improve GCC plugins’ programmer’s productivity
- past experience: Two (static analyzer in C) and OCaml (Frama-C)

But: *understanding the internal representations and passes of GCC is still needed* to develop plugins inside!
Using MELT

gcc -fplugin=melt.so -fplugin-arg-melt-mode=makegreen -O -c foo.c

1. gcc started, so runs cc1
2. cc1 starts, dlopen-s GCC plugin melt.so = MELT runtime
3. the MELT runtime initializes, loads [wired] MELT modules\(^9\), so dlopen-s warmelt*.so ana*.so, and runs handler for mode makegreen
4. the handler installs a green pass in GCC after its phiopt pass. The green pass is coded in MELT.
5. cc1 parses foo.c and run many passes, including our green one [detecting fprintf(stderr, ...)]

---

\(^9\)module in the binary sense [Linux], not the abstract sense [Ocaml]
MELT runtime

melt-runtime.[ch] 14 KLOC $\rightarrow$ melt.so GCC plugin

- a *copying generational* garbage collector backed up by Ggc
- MELT can handle *things*:
  1. *stuff* = long, gimple, tree, and any other GCC datatype const char*, etc. (static typing)
  2. *values* are specific to MELT: closures, tuples, lists, objects, boxed strings, gimple, tree, ..., hash-maps keyed by string, trees, gimples, ... of values (dynamic typing)
- MELT reader, and all MELT builtins; hooks between GCC plugins and MELT; dlopen-s MELT module-s.
- could easily be extended to new GCC data types.
MELT is not polymorphic, because of Ggc limitations!

MELT’s GC & runtime:

- efficient for many young temporary values
- need explicit scan of local variables and current closure\(^1\)
- triggered when young allocation region is full
- old region is Ggc heap
- awkward coding style, but easy to generate C
- depend upon Ggc, not much on GCC datatypes
- normalization required:
  In C syntax, \( f(g(x), y) \rightarrow \tau = g(x); f(\tau, y) \)\(^{11}\)
  translation MELT \( \rightarrow \) C is more than templatized expansion.

\(^{10}\)Facilitate runtime introspective reflection
\(^{11}\)Where \( \tau \) is a fresh local \texttt{melt_ptr_t} variable
Predefined MELT [object] values are known to the runtime. Every MELT value has a fixed discriminant\(^{12}\) (like DISCR_INTEGER for most boxed integers, or CLASS_CLASS) - it is a MELT object (their class inside objects)

As an exception MELT nil (i.e. false, represented by (void*)0) has conventionally DISCR_NULLREC as its discriminant

MELT objects are exactly in C

```c
struct {
    melt_objectptr_t obj_class;
    unsigned obj_hash;
    ushort obj_num, obj_len;
    melt_ptr_t obj_vartab[obj_len];
} 
```

where obj_num is set at most once, and obj_len is fixed.

So testing in C iff p is a closure is fast:

\[
p \neq \text{NULL} \land \text{p} \rightarrow \text{discr} \rightarrow \text{obj_num} = \text{OBMAG_CLOSURE}
\]

\(^{12}\)at start of its memory zone
MELT objects

- The discriminant of an object is its class [à la ObjVlisp]
- Single-inheritance class hierarchy rooted at `CLASS_ROOT`
- Objects have a “magic number” `obj_num` & an hash-code.
- Classes are objects of `CLASS_CLASS`
- Field descriptors are objects of `CLASS_FIELD`
- Symbols like `>i` or `if` are instances of `CLASS_SYMBOL`
- “Keywords” like `:else` or `:gimple` are of `CLASS_KEYWORD`
- the reader produces s-exprs of `CLASS_SEXPR` with boxed integers of `DISCR_INTEGER` etc ...
- selectors are instances of `CLASS_SELECTOR`
- messages (method invocation) can be sent to any value. Each discriminant `≠ DISCR_ANYRECV` (or class) has a parent, a method table... Method installation can be dynamic.
MELT things and stuff

Each thing has a type in C like `long`, `gimple [Ggc-ed stuff]`, `melt_ptr_t [MELT values]`, ... This type is reified thru an descriptive `c-type` object (of `CLASS_CTYPE`) like `CTYPE_LONG`, `CTYPE_GIMPLE`, `CTYPE_VALUE` ...

In MELT source code, “keywords” like `:long`, `:gimple`, `:value` annotate the c-type of things.

*Stuff is statically typed* [e.g. `:long` ≠ `:gimple`]

*Values are dynamically typed* : closures are tested when applied

In MELT code `2 ≠ '2` : plain `2` denotes a raw `:long` stuff, quoted `'2` denotes a boxed `DISCR_CONSTANT_INTEGER` value!

In MELT function applications¹³: first (if any) argument is a value, secondary arguments are things. Result is a value, secondary results (if any) are things. [unusual calling protocol]

---

¹³So also method calls
In MELT source code, *chunks* (rarely used, like `asm` in C) are “trivially” translated into C. E.g. if in MELT $i$ is bound to some `long` stuff, the `void` expression

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

with *macro-string* `#{$sta...}` read as s-expr starting with `sta`, here a *state symbol*, is translated\(^{14}\) to C code:

```c
{STA_1_lab:printf("i=%ld\n", curfnum[3]++);
  goto STA_1_lab;}
```

the first time, and with `sta_2_lab` the second time.

\(^{14}\) *supposing* $i \rightarrow \text{curfnum}[3]$
Primitives

They define a MELT operator by its C expansion, e.g. unary negation:

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

Primitives are statically typed. (negi 2) is ok, but (negi ’x) is bad
defprimitive introduces a definition (hence a binding).
Syntax (1)

approximate and incomplete syntax

\[ \begin{align*}
\text{ctype} & := \text{:long} | \text{:gimple} | \text{:tree} \\
& \quad | \ldots | \text{:value} | \text{:void} \\
\text{formals} & := \emptyset \\
& \quad | \text{ctype}^{?} \text{ var}^{+} \text{ formals} \\
\text{consexpr} & := \text{( instance name_{class} \{ :name_{field} expr \}^{*} )} \\
& \quad | \text{( list expr}^{*} \text{ )} \\
& \quad | \text{( tuple expr}^{*} \text{ )} \\
& \quad | \text{( lambda (formals) expr}^{+} \text{ )} \\
\text{expr} & := \text{var} \\
& \quad | \text{literal} \\
& \quad | \text{consexpr} \\
& \quad | \text{( expr expr}^{*} \text{ )} \\
& \quad | \text{( quote literal )} \quad | \text{( quote name )} \\
& \quad | \text{( get\_field :name_{field} expr )}
\end{align*} \]

\[ \begin{align*}
\text{c-types} & \quad \text{formal arguments} \\
\text{constructible expressions} & \quad \text{object creation} \\
& \quad \text{list creation} \\
& \quad \text{tuple creation} \\
& \quad \text{abstraction} \\
\text{expressions} & \quad \text{variables} \\
& \quad \text{integer, string, nil} \\
& \quad \text{constructions} \\
& \quad \text{function application} \\
& \quad \text{quotation} \\
& \quad \text{field access in object}
\end{align*} \]
Syntax (2) – expressions...

\[
\text{expr} \quad := \ ...
\]

- \((\text{name}_{\text{primitive}} \quad \text{expr}^* )\)
- \((\text{name}_{\text{selector}} \quad \text{expr}_{\text{receiver}} \quad \text{expr}^* )\)
- \((\text{return} \quad \text{expr}^* )\)
- \((\text{setq} \quad \text{var} \quad \text{expr} )\)
- \((\text{code\_chunk} \quad \text{name}_{\text{state}} \quad \text{macrostring} )\)
- \((\text{progn} \quad \text{expr}^+ )\)
- \((\text{put\_fields} \quad \text{expr} \quad \{ \quad :\text{name}_{\text{field}} \quad \text{expr} \quad \}^* )\)
- \((\text{exit} \quad \text{name}_{\text{label}} \quad \text{expr}^+ )\)
- \((\text{if} \quad \text{expr} \quad \text{test} \quad \text{expr} \quad \text{then} \quad \text{expr}^? \quad \text{else} )\)
- \((\text{and} \quad \text{expr}^+ )\)
- \((\text{or} \quad \text{expr}^+ )\)
- \((\text{cond} \quad ( \quad \text{expr}_{\text{cond}} \quad \text{expr}_{\text{then}}^+ \quad )^* \quad \) \((:\text{else} \quad \text{expr}_{\text{else}}^? \quad )\)
- \((\text{match} \quad \text{expr} \quad ( \quad \text{pattern} \quad \text{expr}^+ \quad )^+ )\)

**expressions** (continued)

- Primitive invocation
- Message sending
- Function return
- Assignment
- C chunk
- Sequentiality
- Object update
- Local exit from loop
- If-then-else
- And then
- Or else
- Multiple conditional
- Pattern matching
Syntax (3) – binding expressions & definitions...

\[ expr \ := \ ... \]

- (forever \texttt{name}_label \ expr^+ )
- ( \texttt{name}_iterator ( \ expr^* ) ( \texttt{formals} ) \ expr^+ )
- (let ( ( \texttt{ctype} \ ? \ name \ expr )^* ) \ expr^+ )
- (letrec ( ( \texttt{name} \ consexpr )^* ) \ expr^+ )

\[ def \ := \ ]

- (defun \texttt{name} ( \texttt{formals} ) \ expr^+ )
- (defprimitive \texttt{name} ( \texttt{formals} ) \texttt{ctype} \texttt{macrostring})
- (defclass \texttt{name} \{ :super \texttt{name}_super \}?)
  \{ :fields ( \texttt{name}^* ) ? \}
- (defselector \texttt{name} \texttt{name}_class
  \{ :formals ( \texttt{formals} ) ? \}
- (definstance \texttt{name} \texttt{name}_class \{ :name_field \ expr \}^*)
- (defciterator \texttt{name} ( \texttt{formals}_{start} ) \texttt{name}_{state}
  ( \texttt{formals}_{locals} ) \texttt{macrostring}_{start} \texttt{macrostring}_{end} )
- ...
Syntax (4) – directives

\[
\text{dir ::= } \begin{array}{l}
(\text{export\_values } \text{name}^*) \\
(\text{export\_class } \text{name}^*) \\
(\text{export\_macro } \text{name}_{macro} \text{ expr}_{expander}) \\
(\text{export\_patmacro } \text{name}_{macro} \text{ expr}_{expr} - \text{expander} \text{ expr}_{patt} - \text{expander}) \\
(\text{export\_synonym } \text{name}_{new} \text{name}_{old}) \\
(\text{load } \text{string}_{file} - \text{path}) \\
\end{array}
\]

- export defined values
- export defined classes\(^{15}\)
- export macro
- export patmacro
- export synonym
- include file
- Also export the fields of classes!

\(^{15}\text{Also export the fields of classes!}\)
Syntax (5) – miscellaneous

\[ expr := \ldots \]
\[ \quad (current\_module\_environment\_container) \]
\[ \quad (parent\_module\_environment) \]
\[ \quad \text{(debug msg expr string)} \]
\[ \quad \text{(assert msg string expr)} \]
\[ \quad \text{(compile \_warning string expr)} \]
\[ \quad \text{(cpp \_if name expr \_then expr \_else)} \]
\[ \quad \ldots \]
\[ \text{expr (cont.) for my env.} \]
\[ \text{parent env.} \]
\[ \text{debug print}\textsuperscript{16} \]
\[ \text{assertion}\textsuperscript{17} \]
\[ \text{warning}\textsuperscript{18} \]
\[ \text{cpp \#if \ etc} \]

A MELT module source is a mix of expressions, directives, definitions.

NB: Syntax for pattern matching is given below.

\textsuperscript{16} when asked by option, debug printing with MELT source location.
\textsuperscript{17} shows the MELT stack on assertion failure!
\textsuperscript{18} Similar to \texttt{\#warning} for MELT!
C-iterators

**GCC** has many iterative constructs, e.g. to iterate on every `gimple g` inside a given `gimple_seq s GCC mandates`

```c
{    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 */
    }
}
```

described by a `c-iterator` in **MELT** (defining how to generate such constructs)!
c-iterator example

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

used as :void expression like - where :gimpleseq s -

(each_in_gimpleseq (s) (:gimple g)
  [do something with g...])
modules and environments

One \( \phi.m\text{elt} \) file translated (via \( \phi.c \) file) to \( \phi.so \) module [single name-space]
The module’s \texttt{start\_module\_melt} routine

1. takes a parent environment \( \eta^{19} \),
2. build various initial data (closures, ...),
3. executes top-level forms,
4. returns the module’s environment \( \eta' \) containing the exported definitions of module \( \phi \) (both \( \eta \) and \( \eta' \) are instances of \texttt{CLASS\_ENVIRONMENT}).

so \texttt{start\_module\_melt} is mostly sequential and quite big.
Bindings are instances of sub-classes of \texttt{CLASS\_ANY\_BINDING}.

\[^{19}\text{For the very first module warmelt\texttt{-}first - translated specially - } \eta \text{ is nil.}\]
exported definitions

➤ functions, selectors, c-iterators, ... (export_value ...)

➤ classes and their fields with (export_class ...). Classes and fields names are *globally* unique\(^{20}\).

➤ macros (expanded to some MELT AST for expression) with

(export_macro name expander)

➤ pattern-macros (expanded to some MELT AST for pattern) with

(export_patmacro name expander ...)

Also guru language constructs like (current.module.environment.container) and (parent.module.environment) give full access to environments.

---

\(^{20}\) So (get_field :named name n) translated to test that \(n\) is a \(\text{CLASS\_NAMED}\)...
MELT standard library

The library is used/useful for the translator and for applications. The entire translation process is “transparent” (i.e. extensible by power users thru many selecttors classes ...).

Library: collecting types (map/reduce/...), iterators, primitives, functions, selecttors, macros, ... debug printing; run-time asserts\(^{21}\); translate-time conditionals emitted as `#ifdef`; GDBM; interface to major GCC datatypes and patterns...

Also tools like run-time evaluation of MELT expression (thru translation to temporary modules) & .texi documentation generator.

\(^{21}\) printing the call stack on failure
MELT translator

- implemented in 29KLOC `warmelt-*.*.melt`
- translated to 520KLOC
- fast: 5.1 sec for 5KLOC of MELT code giving 164KLOC of C code
- bottleneck: compilation of generated C code (250s!)
- extensible by power user
- bootstrapped: `warmelt-*.*.c` in source svn so exercises most of itself and the runtime
- robust w.r.t. GCC evolution\(^\text{22}\) \(\rightarrow\) Gimple transition

\(^{22}\) because Ggc is stable and hidden by MELT runtime!
Hooks to GCC

The MELT runtime uses and provides several hooks to existing GCC hooks for plugins. Can be extended when needed to future GCC hooks. MELT hooks uses MELT objects i.e. a GCC pass in MELT is reified as an instance of `CLASS_GCC_PASS`. 

Basile STARYNKÉVITCH

MELT = a Lisp matching GCC internals
Using patterns

Fact

**Pattern matching** is an **important** operation in source program handling and compilation.

GCC has no code generator for patterns in middle end. gcc/fold-const.c is hand-written.

*Patterns* are major syntactic constructs (like expressions and bindings in Scheme or MELT).

?π ≡ \texttt{(question π)} exactly like ’ε ≡ \texttt{(quote ε)}.

?x is the **pattern variable** x.

?_ is the **joker pattern**.

In pattern context, expression are **constant patterns**.
Pattern related syntax (1) – definitions for matching

\[
def := \ldots
\]

\[
\begin{align*}
\text{def} & := \text{...} & \text{definitions [cont.]} \\
| & \quad (\text{defcmatcher } \text{name} (\text{formals}_\text{match} & & \text{& inputs}) \\
& & \quad \text{(formals}_\text{outputs} \text{)} \text{name}_\text{state} \\
& & \quad \text{macrostring}_\text{test} \text{ macrostring}_\text{fill}) & \text{c-matcher} \\
| & \quad (\text{defunmatcher } \text{name} (\text{formals}_\text{match} & & \text{& inputs}) \\
& & \quad \text{(formals}_\text{outputs} \text{)} \text{expr}_\text{matching} \text{ expr}_\text{applying}) & \text{fun-matcher}
\end{align*}
\]

A matcher appears in pattern construct\textsuperscript{23}. A C-matcher defines its behavior by C expansion; a fun-matcher defines it by a multivalued function.

\textsuperscript{23}Matchers can also appear in expression context.
Pattern related syntax (2) – patterns for matching

```
pattern ::= 
  expr
  | ?_  
  | ? var  
  | ? ( name matcher expr* patt* )  
  | ? ( instance name class { :name field patt }* )  
  | ? ( object name class { :name field patt }* )  
  | ? ( tuple patt* )  
  | ? ( list patt* )  
  | ? ( as ? var patt )  
  | ? ( and patt+ )  
  | ? ( or patt+ )  
  | ? ( when patt expr )

Patterns appear only in match expressions.
```
Pattern example

```lisp
(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...]))

A bit naive to seek for fprintf with string compare!
```

Basile STARYNKÉVITCH
MELT = a Lisp matching GCC internals
A **match** expression is a “conditional” containing a sequence of **match-cases**. Like in conditional, the tested [i.e. matched] expression and the result of the match can be any thing (value or stuff).

Each match-case has a pattern (possibly complex i.e. nested with sub-patterns) and sub-expressions possibly referring to the pattern variables\(^\text{24}\). All pattern variables are “cleared” before testing the match case’s pattern.

The translator tries to avoid duplicating tests.

Each pattern has generally two roles

1. **testing** if the matched thing fits
2. if it fits, **filling** data transmitted to sub-patterns

\(^\text{24}\)i.e. \(\text{lhs not ?lhs}\)
pattern taxinomy

Patterns (when matching thing $\tau$) can be

1. joker $\_\_ -$ always match
2. constants $^{25} \xi -$ matches if $\xi$ identical [in C $==$] to $\tau$
3. pattern variables like $?x$
   if unset, set it to $\tau$. Otherwise test for identity with $\tau$
   Non-linear patterns: $(\text{gimple\_assign\_simple} \ ?x \ ?x)$ useful to
   find assignments of a C variable $\text{var}$ to itself.
4. control patterns:
   ▶ $(\text{as} \ ?x \ ?x)$: set or test $?x$ to $\tau$, then match $\pi$ against $\tau$
   ▶ $(\text{and} \ \pi_1 \ldots \ \pi_n)$ matches iff $\pi_1$ matches $\tau$ and then $\pi_2 \ldots$
   ▶ $(\text{or} \ \pi_1 \ldots \ \pi_n)$ matches iff $\pi_1$ matches $\tau$ or else $\pi_2 \ldots$
5. elementary *matcher* s like $\text{gimple\_assign\_cast}$ or $\text{string\_same}$
6. object patterns $(\text{instance class} :\text{field}_1 \ \pi_1 \ldots :\text{field}_k \ \pi_k)$
   (some fields can be missing) and others object patterns

$^{25}$Constants includes expressions in pattern context
c-matchers

First example:

the :cstring stuff σ matches ?(cstring_same "fprintf") iff the const char* σ has the same char-s as "fprintf" = the c-matcher’s input.

(defcmatcher cstring_same (:cstring str cstr) () strsam
  :doc #{The $CSTRINGSAME 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)) }# )

The cstring_same c-matcher has a test part but no fill part.
c-matchers

Second example

Filtering casting assignments in Gimple (see Wadler’s views)

```
(defcmatcher gimple_assign_cast
  (:gimple ga) (:tree lhs rhs) gimpascscs
  #{ /*$gimpascscs test*/
    ($ga && gimple_assign_cast_p ($ga)) }#
  #{ /*$gimpascscs fill*/ $lhs = gimple_assign_lhs($ga);
    $rhs = gimple_assign_rhs1($ga); }# )
```

Here ga is the matched gimple (with no other inputs), and lhs rhs are the output trees. gimpascscs is a state variable.

c-matchers can be “intersecting” (i.e. both gimple_assign_cast and gimple_assign_single are gimple_assigns). Order is important when using them!
fun-matchers

**fun-matchers** are view defined thru a MELT function. The primary result gives the test, and secondary results if any the filling.

The pattern \(?{(\text{isbiggereven } \mu \ \pi)}^{26}\) is matching a :long stuff \(\sigma\) iff \(\sigma\) is a even number, greater than the number \(\mu\), and \(\sigma/2\) matches the sub-pattern \(\pi\).

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

Matchers can also define what they mean in expression context.

\(^{26}\text{isbiggereven could be a c-matcher!}\)
pattern matching translation

The **design of MELT patterns and the implementation** of their translation was **painful to get right!**

**Insights in implementation**

- compute the set of pattern variables in a match case and clear them.
- build incrementally a control graph of testers
- testers have both then and else siblings. Success test (for the sub-expressions of a match case) is a special case.
- track intermediate sub-matched things. Factorize similar matchers!
- each matched thing knows it sub-graph of testers.
- before adding a tester for matchers and instances, check [merge?] if it does not exist already.
- use high-level functionals in the implementation. The `normal_pattern` selector expects a closure to handle the tester it might build.
- generate C if-s and goto-s (not yet switch using hashcode...)

Basile STARYNKÉVITCH

MELT = a Lisp matching GCC internals
Conclusions and future work

- for legacy huge systems, generating “matching” C code is better than pushing an existing implementation.
- the language should provide linguistic devices describing how to generate
- the idioms should match the project’s culture and style
- why MELT was not done before?

On future GCC plugins: OpenGPU: MELT passes generating OpenCL code while cloning functions (in close cooperation with Albert Cohen)