

Le compilateur GCC : avancées récentes, greffons et outil MELT

Basile STARYNKÉVITCH

basile@starynkevitch.net

svn \$Revision: 103 \$



exposé à titre personnel

Plan

- 1 Introduction et généralités
 - Audience cible
 - compilateurs et interprètes.
 - déclaratif, procédural ; langages de haut et de bas niveau
 - complexité des ordinateurs donc des compilateurs
- 2 Évolution et survol interne de GCC
 - Évolutions de GCC
 - la communauté GCC
 - le travail de développement dans GCC
 - Survol interne de GCC
- 3 l'outil MELT
 - motivations, contexte, objectifs de MELT
 - aperçu du langage MELT
 - code chunks and primitives
 - MELT syntax overview
 - c-iterators
 - modules, environments, standard library, hooks
 - Using pattern matching
 - c-matchers and fun-matchers

Avertissement

Avertissement important

Les **opinions** exprimées ici sont **seulement miennes** et pas celles de mon employeur, de la communauté **GCC**, de *l'April* ou autre . . .

introduction et généralités

Public attendu :

- 1 (principalement) **“développeurs pssionnés”** de logiciels (libres) utilisant GCC.
- 2 (secondairement) **“libristes”** intéressés par un **gros logiciel libre patrimonial** *“legacy free software”* = GCC

Un compilateur

Compilateur =

“compiler”

- *programme* informatique
- *traduisant*
- un langage informatique *source*¹
- en un autre langage *cible*²
- en *préservant ± sémantique*³ du code source.

[± wikipedia]

¹Dans GCC, c'est C, C++, Java, Fortran, Ada, Objective-C, ...

²Dans GCC, assembleurs x86, AMD64, ARM, PowerPC, Sparc, ...

³sémantique = signification

Le langage source est souvent de plus haut niveau que le langage cible (sinon *dé-compileur*) \Rightarrow la sémantique n'est donc *pas totalement* préservée.

Le langage cible n'est pas toujours du code binaire ou assembleur. Ce peut être du C⁴, du code octet JVM⁵ ou CLR⁶ ou Parrot (Perl 6), du Ocaml (Coq) De même, le code source peut être une représentation intermédiaire (code octet).

On ne définit pas ce qu'est un langage informatique.

Beaucoup de logiciels sont à considérer comme **des compilateurs** : \LaTeX , transformateurs XML (en XSLT) . . .

\Rightarrow comprendre un peu la compilation est important pour tous les développeurs :

- 1 car ils utilisent quotidiennement un compilateur
- 2 car les concepts de la compilation sont réutilisables

⁴MELT, Chicken, Bigloo, ... (Scheme), Mercury, Malice/Caia ... génèrent du C.

⁵Java, Scala ... génèrent du code octet JVM

⁶C#, Nemerle, GCC-CLI génèrent du code CLR pour .NET ou Mono

Un interpréteur

Interpréteur =

(ou interprète) "*interpreter*" est un *outil informatique qui* analyse, traduit⁷ et *exécute un programme* écrit dans un langage informatique.

[± wikipedia]

Un interprète effectue le calcul ou traitement indiqué dans son entrée (le code source interprété).

Un programme qui compile un source puis exécute le code cible sans le conserver est donc en principe un interprète !

Beaucoup de logiciels sont à considérer comme **des interprètes** : navigateur ou serveur Web, jeux, ...

⁷Beaucoup d'interpréteurs traduisent en une représentation interne particulière le code du programme, donc sont un peu des "compilateurs".

Pourquoi compiler

- *Développer un logiciel* = un **acte social**. Le code source s'adresse :
 - 1 à soi-même, pour retenir ses idées (6 mois plus tard !)
 - 2 aux (futurs) collègues développeurs
 - 3 à la machine (souvent au compilateur)
 - 4 attention aux inévitables bogues !
 - 5 le code source doit être relisible et améliorable par des humains.
- Exécution du logiciel = par la machine (qui ne comprend que le langage machine)
- Utilisation du logiciel = par des utilisateurs humains.

Interpréteurs et compilateurs

Il y a une continuité entre interprètes et compilateurs.

- peu d'interpréteurs actuels ⁸ parcourent itérativement le texte du code source d'une boucle.
- souvent, *les interpréteurs traduisent* d'abord *le code source en une représentation plus efficace* : arborescences, graphes, code octet "*byte-code*", puis "exécutent" cette représentation.
- les compilateurs ont aussi un rôle d'interprétation : simplification des expressions constantes "*constant folding*" : $2 + 3 \rightarrow 5$
- "*Just In Time*" = JIT = traduction à la volée (souvent d'un langage source code octet) en du langage machine, et re-traduction optimisée du code chaud (fréquemment exécuté).
- aspects interprétés d'un langage compilé : format d'impression
`printf`

⁸Au contraire des premiers Basic des années 1980, qui re-parcouraient la chaîne d'une boucle en Basic !

Un langage informatique peut être implanté comme un interprète ou comme un compilateur (ou les deux). Il peut y avoir plusieurs compilateurs (parfois incompatibles) du même langage.

Sous Linux par exemple :

- interprètes : les shells, Lua, Tcl, Ocaml, Ruby, Python, Perl, CLisp, Ch [pour C] ...
- compilateurs : Ocaml, GHC/Haskell, SBCL, **GCC**, icc, open64, tinycc, nwcc, llvm/clang ... [JIT :] JVM, Mono ...

De nos jours : presque tous les compilateurs (Linux) sont gratuits. La plupart sont libres.

Déclaratif ou procédural.

connaissance déclarative ou procédurale

Une connaissance est *déclarative* si on peut l'utiliser de plusieurs façons. Une connaissance *procédurale* est un mode d'emploi (à utiliser tel quel, d'une seule manière). [d'après Jacques Pitrat]

Exemples de déclaratif

(lois physiques) gaz parfait : $PV = nRT$; Ohm : $U = RI$.

"L'adjectif s'accorde en genre et en nombre avec le nom qu'il qualifie".

Procédural : du code source C (ou même Haskell).

Deux livres intéressants de J.Pitrat :

- *méta-connaissances (futur de l'intelligence artificielle)* (Hermès 1990)
- *artificial beings (the conscience of a conscious machine)* (Wiley/ISTE 2009)

Le *"declarative programming"* n'est pas assez déclaratif pour J.Pitrat.

approches et points de vue complémentaires :

■ **déclaratif :**

- *privilégier les données* au code.
- décrire des méta-données, des règles, des méta-règles.
- faciliter l'expressivité (pour le développeur humain), parfois au détriment des performances.
- définir et utiliser des heuristiques.
- viser la *généralité*, la réutilisation, la reflexivité.
- *formaliser des connaissances pour plus tard*
(même si le système ne sait pas encore les utiliser)

■ **procédural :**

- mettre l'*accent sur le code*, pas les données.
- *programmer* finement *des algorithmes*.
- traiter tous les cas (même improbables).
- *viser la performance*, parfois au détriment de l'expressivité.
- toutes les données sont utiles maintenant.
- *difficulté d'évolution* du système.

le déclaratif : ambition historique des compilateurs

Les premiers langages doivent être efficaces, mais veulent être déclaratifs.

- (1955, USA) **FOR**mula **TRAN**slation
- (1958, France) **P**rogrammation **A**utomatique des **F**ormules

1960 : le programmeur (bon marché) énonce des formules ; la machine (onéreuse, 10 ans de salaire) les transcrit en instructions.

Années 1980-90 : Systèmes experts (“intelligence artificielle”)

Main d'œuvre chère, machine moins coûteuse (1 an de salaire)

“small is beautiful”

Actuellement (2010) : machine presque “gratuite” (2 semaines de salaire). *Coûts de développement*. Importance du logiciel existant, de sa *maintenabilité*. Rester compatible avec le vieux code source

“legacy code”

langages de haut niveau


Les langages de haut niveau sont plus concis que les autres :

Un mini-interprète en Ocaml

```
type expr = Ent of int | Var of string
          | Som of expr * expr | Prod of expr * expr ;;
let rec interp e (*environment*) f (*formule*) =
  match f with Ent i → i | Var n → Hashtbl.find e n
  | Som (x,y) → (interp e x) + (interp e y)
  | Prod (x,y) → (interp e x) * (interp e y) ;;
```

Moins procédural et plus concis⁹ que son équivalent en C++ ou Java.

Importance des *données arborescentes*, de l'*inférence de types* et du **filtrage** "*pattern matching*".

⁹Ocaml (compilé sur Amd64) est aussi efficace que C++. 

réticence (peu justifiée) des industriels pour les langages de haut niveau.

- mythe (facilité) du développeur kleenex interchangeable
- apprendre un langage est assez facile, quand la spécification du langage est simple.
- difficulté réelle : apprendre une bibliothèque (son API). Ça ne dépend pas du langage.

Mais : foules de langages spécifiques *“domain specific languages”*.
Mais : croissance des langages dynamiques interprétés (parfois peu efficaces) - PHP, ...

Question provocatrice/amusante *“food for thought”*

Combien de centrales nucléaires¹⁰ seraient économisées en passant de PHP à autre chose (Ocsigen, Hal) ?

¹⁰Internet consommerait 9 à 10% de l'électricité aux USA 

Conseils pour utiliser, faire ou améliorer un DSL

- piloter une grosse application par un langage de script (DSL) est structurant pour son architecture.
- se documenter sur la compilation/les interprètes et utiliser la bonne terminologie et methodologie
- se brancher sur une implémentation existante
- à défaut, s'inspirer de langages existants
- peut-être : générer du code (C, LLVM, libjit, GNU lightning).
- soigner l'interface avec des routines en C.
- documenter.

annotations de haut niveau (FRAMA-C)

FRAMA-C <http://frama-c.cea.fr/> est un **analyseur statique** (en LGPL, codé en Ocaml) de code C critique.

```
/** annotation de spécification en ACSL */  
/*@ requires \valid(p) && \valid(q) ;  
    ensures *p <= *q;  
    ensures (*p == \old(*p) && *q == \old(*q))  
           || (*p == \old(*q) && *q == \old(*p)) ; */  
/** code en C */  
void orderptr(int *p, int *q) {  
    if (*p > *q) {  
        int tmp = *p; *p = *q; *q = tmp;  
    }  
}
```

L'outil FRAMA-C est capable de **vérifier automatiquement** que le code de `orderptr` *suit sa spécification en ACSL*.

C et les processeurs actuels : mythes et réalités

- *Le langage C serait proche de la machine.*

C'était vrai en 1985 ; c'est faux en 2010.

*Les processeurs actuels **multi-cœurs** (Core 2, Phenom 2, i7) ne sont pas des i386 (année 1986) deux cents fois plus rapides.*

- *L'assembleur est toujours plus efficace que le C*

Les processeurs sont tellement complexes qu'il est trop difficile de coder à la main du code assembleur efficace.

- *Le langage C est le plus efficace des langages* (pour la performance du code généré).

Certains JIT (JVM, CLR ?) font mieux que le C pour certains programmes. Pour certains programmes, Ocaml est plus efficace que l'équivalent en C.

- *La gestion manuelle de la mémoire est la plus efficace.*
(`malloc` et `free` en C).

Les ramasse-miettes (copieurs générationnels, adaptatifs) peuvent être plus efficaces que des `malloc` et `free`

- *On peut tout coder en C.*

L'introspection de la pile, la gestion du cache, la synchronisation entre filaments (`mutex`, ...), les exceptions (le mécanisme de `set jmp`), les appels récursifs terminaux "tail-recursive calls", ... ne sont pas codables en C.

- *Le langage C est très portable.*

*non : difficulté du passage de 32 à 64 bits.
la portabilité entre architectures différentes a moins d'importance qu'autrefois : le x86-64 est devenu le "code octet" des processeurs. Il faut optimiser différemment pour AMD Phenom II ou pour Intel Core i7.*

Complexité des ordinateurs actuels.

Tous les processeurs se ressemblent (hélàs) : même architecture (imposée par Microsoft) x86-64 ¹¹.

L'architecture x86-64 est incommode : hétérogène, encore peu de (16) registres, pas d'opérations à 3 opérands ⇒ importance de l'allocateur de registre "*register allocator*" et de la sélection des instructions dans le compilateur.

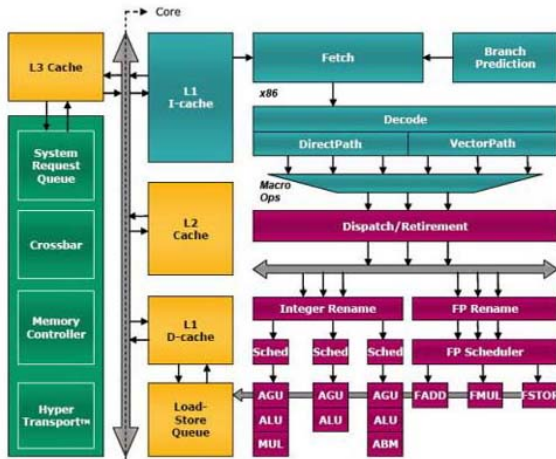
Un processeur actuel (Intel Core2 ou i7, AMD Phenom II) traduit¹² le code machine x86-64 en des micro-opérations (fortement propriétaires) spécifiques. Un même programme doit être optimisé différemment pour i7 ou pour Phenom II (qui ont pourtant le même jeu d'instructions x86-64). Certaines instructions sont plus coûteuses que d'autres (dépend du modèle).

¹¹Sauf dans l'embarqué : ARM, PowerPC, ... ; les super-calculateurs ; les GPGPU (processeurs graphiques). Le ia-64 (Itanium) est quasi-mort.

¹²"Compilation" par le matériel sur la puce.

micro-architecture AMD Phenom II - 1 cœur (parmi 4)

<http://www.xcpus.com/Images/Docs/doc117/AMD-slide2.jpg>



cache et mémoire

Les processeurs ont tous des caches mémoires :
L1 (64ko I + 64ko D), L2 (512ko), L3 (6Mo partagé entre cœurs).
L'accès à des octets sur une barrette RAM est ≈ 500 fois plus long
que l'accès au cache L1 ($< 1ns$)

\Rightarrow *importance* de la *localité des données* et du code et de leur
*alignement*¹³

Les disciplines de cache, et donc les performances du cache varient
d'un processeur à l'autre.

Les *défauts de cache* L1 "*cache miss*" sont *très pénalisants* : on
aurait pu exécuter plusieurs centaines d'instruction machine¹⁴.

Le compilateur doit gérer le cache, en générant à bon escient son
préchargement "*prefetch*". GCC offre aussi à l'utilisateur
`__builtin_prefetch` et `__builtin_clear_cache`.

¹³ GCC réordonne les variables sur la pile, ou même les champs des `struct` !

¹⁴ D'où les cœurs "*multi-thread*" chez Intel.

parallélisme interne à un cœur

Les processeurs sont *pipelinés* (travail à la chaîne) et *super-scalaires* (plusieurs unités d'exécution). ⇒ **Impossible de prévoir le temps de calcul** d'une petite boucle !

Des instructions machines consécutives gagnent à être indépendantes pour être exécutées en parallèle (sur plusieurs unités de traitement du cœur) ⇒ importance de l'*ordonnancement* "*scheduling*" des instructions par le compilateurs.

Le processeur a des mécanismes de prédiction de branchement (mais "*pipeline stall on branch misprediction*"). Le compilateur devrait prévoir la fréquence des branchements, ou éviter les sauts ou appels indirects¹⁵. GCC offre aussi `goto *x ;` et `__builtin_expect`.

Pour éviter des branchements trop fréquents, des instructions conditionnelles ont été ajoutées dans le jeu d'instruction x86.

¹⁵Sous Linux, les appels à des fonctions (telles que `printf` ou `select`) dans une librairie partagée externe comme `libc.6.so` sont indirects.

exemple de code source et cible

La fonction suivante contient des sauts et des tests.

```
/* fichier som4.c */  
long som4 (long t[]) {  
    long s = 0;  
    for (int i = 0; i < 4; i++)  
        if (t[i] > 0)  
            s += t[i];  
    return s;  
}
```

On peut la compiler avec

```
gcc -S -O3 -std=gnu99 -fverbose-asm som4.c  
pour obtenir le fichier assembleur som4.s
```

```
.type som4, @function
som4 : xorl %edx, %edx      # s
      cmpq $0, (%rdi)     #, * t
      movq 8(%rdi), %rcx  #, D.2725
      cmovns (%rdi), %rdx #* t,, s
      testq %rcx, %rcx   # D.2725
      leaq (%rdx,%rcx), %rax #, s
      movq 16(%rdi), %rcx #, D.2725
      cmovle %rdx, %rax  # s,, s
      leaq (%rax,%rcx), %rdx #, s
      testq %rcx, %rcx   # D.2725
      movq 24(%rdi), %rcx #, D.2725
      cmovle %rax, %rdx  # s,, s
      leaq (%rcx,%rdx), %rax #, s
      testq %rcx, %rcx   # D.2725
      cmovle %rdx, %rax  # s,, s
      ret
```

code *sans branchements*, avec chargements conditionnels `cmov...`

parallélisme externe entre cœurs ou processeurs

Les processeurs (ou cartes mères) actuels ne sont pas des PRAM "*Parallel Random Access Machine*". *L'accès à une mémoire n'est pas uniforme*, et coûte d'autant plus cher que la mémoire est loin du cœur : "*Non-Uniform Memory Access*".

Deux programmes (sur des processeurs ou cœurs distincts) *accédant* simultanément sans précautions *à la même case* mémoire *ne liront pas forcément la même valeur* : la **cohérence des caches**¹⁶ est assurée en partie par matériel, mais requiert des *instructions spécifiques coûteuses*.

GCC offre le support d'*OpenMP* par pragmas. Les opérations atomiques sont fournies : `--sync-fetch-and-add ...`. Les données filaires *"thread-local storage"* se déclarent avec `--thread...`

¹⁶Les mécanismes de cohérence de caches, donc de synchronisation, et donc leurs performances, sont **très** différents d'un modèle de processeur à un autre.

En pratique, la programmation parallèle à mémoire partagée s'appuie sur des bibliothèques comme `pthread`, mais ne passe pas à l'échelle *“scalability issues”*. Elle est délicate¹⁷.

Les instructions machines de synchronisation sont très coûteuses, car elles “vident” les caches. Leurs coûts et performances varient d'un processeur à l'autre et même d'un cœur à l'autre.

Le goulot d'étranglement est de moins en moins la vitesse brute des cœur, mais plutôt *la bande passante* entre cœurs et/ou barrettes de RAM.

¹⁷Le parallélisme par échange de message est plus simple à coder et passe mieux à l'échelle.

complexité des compilateurs

Écrire un compilateur naïf¹⁸ est assez simple : `tinycc` [37 KLOC environ] traduit *rapidement* “instruction par instruction” le C en x86.

Un compilateur se charge généralement de :

- 1 d'abord lire et analyser le texte du code source, pour en construire une représentation arborescente¹⁹, l'arbre de syntaxe abstrait (AST = “*abstract syntax tree*”).
- 2 transformer l'AST en d'autres représentations internes
- 3 malaxer ces représentations internes
- 4 finalement émettre le langage cible (assembleur ou code machine).

¹⁸c.à.d un compilateur générant trivialement du code “médiocre” pour du C.

¹⁹L'analyse lexicale et syntaxique est simple car bien codifiée (grammaires LALR ou LL, outils à la Yacc/Bison/Flex/Antr) sauf pour des syntaxes pathologiques comme C++ ☰

Souvent, un compilateur doit aussi (accessoirement) lancer d'autres outils (préprocesseurs, assembleurs, éditeurs de lien) pour le confort de l'utilisateur.

Travail des compilateurs

Un compilateur ne manipule pas essentiellement du texte, mais *passé son temps à malaxer des représentations internes* arborescentes ou en graphe (parfois cyclique).

Plus qu'autrefois, il est essentiel pour les performances de bien optimiser le code source. Mais l'optimisation est un art (comprendre l'intention du programmeur) difficile, et nécessite de trouver de judicieux compromis.

Faudrait-il optimiser

```
#include <stdlib.h>
#include <time.h>
/* estimation du temps d'un calloc+free */
int tempscallocfree(int ln) {
    clock_t tdeb,tfin; void* p;
    tdeb=clock();
    p = calloc(1, ln); free(p);
    tfin=clock(); return tfin-tdeb; }
```

en l'équivalent de

```
int tempscallocfree(int ln) { return 0; }
(car p n'est pas utilisé donc ne doit pas être alloué...)?
```

Nota : *l'optimisation est un problème indécidable* en toute généralité !

Auto-amorçage des compilateurs

Beaucoup de compilateurs sont auto-amorcés *“bootstrapped”* : le compilateur d'un langage L est lui-même écrit en L^{20}

- techniquement ce compilateur est souvent complexe, donc constitue un bon cas de test.
- humainement les auteurs d'un compilateur en sont fiers et l'utilisent ; les utilisateurs peuvent aussi être ainsi rassurés.
- mais il faut pouvoir compiler, ou avoir déjà compilé, ce compilateur.

Pourtant, C n'est pas le meilleur langage pour écrire un compilateur C. Et les compilateurs Fortran sont rarement codés en Fortran.

²⁰C'est le cas de C - dans GCC-, Ocaml ou MELT !

Complexité et imperfection des compilateurs

Les compilateurs sont de plus en plus complexes, à cause de :

- la complexité grandissante des machines (ou des langages cibles)
- le fossé croissant entre langage source et cible
- l'évolution des spécifications de langages
- les exigences des utilisateurs
- le besoin croissant d'optimisations ou de diagnostics
- la loi de Moore (performance $\times 2$ tous les 18 mois) devient fausse

Donc **tout compilateur est imparfait**. Mais voir le projet *CompCert*
<http://compcert.inria.fr/> de compilateur C certifié (prouvé par Coq).

Besoin de certification²¹ des outils pour les logiciels critiques
(avionique DOI178B ...). Mais n'*incriminez pas GCC pour vos bogues* !

²¹Certaines versions de GCC dans des modes d'utilisation restreints sont certifiées, validées par leur popularité et des tests extensifs.

Évolution et survol interne de GCC

Évolution de GCC en complexité

$\text{GCC}^{22} = \text{un énorme logiciel patrimonial toujours croissant}$

version	4.2.1	4.4.1	<i>trunk</i> r.152437	<i>trunk</i> r.155790
date	juillet 2007	juillet 2009	octobre 2009	janvier 2010
sloccount	2956KLOC	3844KLOC	3962KLOC	4007KLOC
Δ_{sloc}	0%	+30%	+34%	+35.5%
.tar.bz2	44.1Mb	62.9Mb	60Mb ?	
Δ_{tbz2}	0%	+42%	+36% ?	
# files	36.8K	66.0K	68.1K ?	
$\Delta_{\# \text{ files}}$	0%	+79%	+85% ?	
binaire <code>cc1</code> ²³	5.87Mb	8.97Mb	?	

Avec une *grosse communauté de développeurs*²⁴ obéissant à des règles sociales strictes.

²² **Gnu Compiler Collection** : gcc.gnu.org

²³ GNU/Linux Ubuntu Karmic AMD64

²⁴ \approx 400 *"maintainers"* sans aucun dictateur bénévole.

Historique de GCC

■ Autrefois (1987 !) Gnu C Compiler

- 1 **Langage dominant** sous Unix™ : **C** (Kernigan&Ritchie)
- 2 Les *compilateurs* étaient *propriétaires*
(mais gratuits sur SunOS)
- 3 *Unix™* était *écrit en C*
- 4 **Le logiciel libre exige un compilateur libre**
motivation initiale de GCC : libre et *simple*
- 5 architecture *homogène* des machines (68020 → Sparc)
 - *mémoire* RAM *petite* (quelques Mo)
 - d'accès *uniforme* et rapide (quelques cycles CPU $\equiv \mu s$)
 - disque limité (50 Mo)
 - performance *prévisible* des instructions machine
 - processeur séquentiel et déterministe
- 6 un compilateur *simple* convient.

GCC 1 : simple compilateur C (1987)

- mars 1987 : première annonce de GCC 0.9 (R.M.Stallman)
- cible 68020 + vax
- capable de compiler Emacs
- GCC 1.8 : août 1987
- `gcc-1.42.tar.gz` septembre 1992, 1818Ko

Langage C proche des machines de l'époque :

- mot-clé **register**
- traduction “syntaxique” bloc par bloc - en C { ... }
- programmes séparés :
 - 1 préprocesseur `cpp`
 - 2 traducteur `cc1`

(puis [BinUtils] assembleur `as` et édition de liens `ld`)

GCC 2 : Cygnus, C++ & RISC (1999) ; crise EGCS

- 1 ajout d'un compilateur *natif* C++ (Michael Tiemann, Cugnus)
 - C++ est encore experimental
 - C++ original (ATT) = un traducteur `cfront` C++ → C
 - `g++` compile nativement (donc plus vite que `cfront`)
- 2 ajout de nombreuses cibles RISC. Âge d'or des stations de travail (RISC = Reduced Instruction Set Computer). Sparc, PA RISC, PowerPC
- 3 crise EGCS (Experimental Gnu Compiler System) - "fork" de GCC (août 1997)
- 4 réunification : EGCS rejoint GCC : `gcc-2.95` (juillet 1999)
compile C,C++,Fortran,Chill,Java

GCC 3 et 4 : la maturité

GCC 3.0 (juin 2001)

- compile et optimise une fonction toute entière
- représentation[s] interne[s] intermédiaire[s] du code

GCC 4.0 (avril 2005)

- plus de 2.2MLOC
- multi-source ; C, C++, ObjectiveC, Java, Fortan 77&95, Ada, ...
- multi-cible multi-plateforme
- représentations internes Gimple Tree et Tree SSA
(independantes des langage source et cible)
- compile et optimise une unité de compilation

GCCs récents

GCC 4.3 (mars 2008) - près de 4MLOC

- compétitif par rapport à la concurrence
- plusieurs équipes de compilation migrent vers GCC
- représentations internes Generic + Gimple Tuple + SSA

GCC 4.5 (à paraître au 1^{er} trimestre 2010)

- toujours compétitif, même vis à vis de LLVM qui progresse beaucoup
- optimisations majeures (LTO)
- greffons
- monumental

Structure de la communauté GCC

GCC a toujours été libre, licence GPL, sous **copyright FSF**

- © FSF spécifique aux logiciels FSF (Gcc, BinUtils, Bash. . .)
- ≠ noyau Linux (GPLv2+, copyright par contributeurs)
- ⇒ *transfert de copyright à la FSF* avant toute contribution
<http://gcc.gnu.org/contribute.html>
- exigence contraignante (Δ *lobbying* dans les grosses structures)
- La FSF peut changer la licence : GPLv2 → GPLv3
- forte barrière légale d'entrée (co-responsabilité, solidarité)
- *pré-requis pour participer* au développement de GCC
- méfiance justifiée (mais excessive) de la FSF
- **code toujours revu par les pairs**

Les membres de la communauté GCC

> 400 personnes [MAINTAINERS], souvent **professionnels qualifiés** à temps plein, dont

- 12 “global reviewers” (approuvent tout sauf leur propre code)
- > 50 “CPU port maintainers”
- > 15 “OS port maintainers”
- > 12 “language front-end maintainers”
- > 100 “various” & “non-algorithmic” maintainers
- > 15 “various” reviewers
- 200 “write after approval” maintainers

Communauté anglophone

goulot d'étranglement : *revue de code*

Les employeurs des contributeurs

Les contributeurs sont souvent payés à temps plein par :

- 1 Industriels des processeurs : Intel, AMD, STMI, AXIS, ...
- 2 SSSL : CoreSourcery, Adacore, SuSE, Redhat. . .
- 3 Industrie du logiciel : IBM, Oracle, Apple, HP, ...
- 4 Service : Google
(binaire de 750Mo ! 0.1% d'amélioration paie douzaine de développeurs) . . .
- 5 Académique : labo de recherches, université, ...

Faible présence européenne

Actuellement : **GCC est un compilateur omniprésent.**

de moins en moins de compilateurs concurrents propriétaires

Plusieurs centaines de milliers d'utilisateurs de GCC !

“GCC Steering Committee”

Comité de Pilotage

12 personnes expertes cooptées [\pm global reviewers]

- nomment les “reviewers” et “maintainers”
(tout contributeur est “write after approval”)
- acceptent et décident les **orientations majeures** de GCC
(nouveaux : port, langage, trait)
- décident, avec la FSF, sur les licences (runtime, plug-in)
- *rôle politique*
- aucun rôle technique important

Principes de développement

- 1 **revue du code par les pairs** “peer-reviewed code”
chaque modification est lue et approuvée par autrui
- 2 **règles de codage** bien définies et formalisées
- 3 important **jeu de test**
- 4 *compilateur auto-amorcé* `make bootstrap`
- 5 soumettre plutôt des petits patches (pour approbation)

Tronc et branches

Le **tronc** (“trunk”) concentre le *développement des futures versions* (“releases”) en 3 étapes (“stage”)

- 1 toutes modifications même majeures ou discontinues
- 2 modifications et améliorations mineures
- 3 corrections de bogues

Les **branches** servent à l'*expérimentation* (code pouvant être incorporé au tronc plus tard)

Outils, méthodes et coutumes de développement

- les mailing-lists (messageries *publiques et archivées*)
 - `gcc@gcc.gnu.org` *débats techniques*
[517 messages en septembre 2008]
 - `gcc-patches@gcc.gnu.org` *proposition de code*
[2000 messages en septembre 2008]
 - etc...
- le chat `irc://irc.oftc.net/#gcc`
- documentation interne en Texinfo `*.texi`
- wiki `http://gcc.gnu.org/wiki/`
- base de bogues `http://gcc.gnu.org/bugzilla/`
- GCC Summit (Canada, 2 jours chaque été)
- workshops, conférences, ...

Traits principaux de GCC

quelques éléments copiés de l'exposé de Laurent Guerby à Toulibre

- **logiciel libre, sous licence GPLv3**, ©FSF, en C (C++, Ada)
- multi-langage source : *C, C++, Java, Objective C[++]*, *Fortran, Ada* (et aussi Modula 3, Pascal, PL/1, D, Mercury, VHDL en dehors)
- *plus de 30 machines/systèmes cibles* dont x86/AMD64, ARM, Alpha, IA-64, MIPS, Sparc, CRIS, PowerPC, M68K, Xtensa, RX
...

- *compilateur croisé* (cible \neq machine hôte) si besoin
- assez facile à construire puis installer

```
apt-get build-dep gcc #installer les dépendances : GMP...
tar xfj gcc-4.5.tar.bz2; mkdir Build; cd Build
../gcc-4.5/configure #avec les options, voir --help
make bootstrap; sudo make install
```

- **compilateur amorcé** : *make stage1* avec le compilateur système, puis *stage2* avec ce gcc, puis *stage3* pour comparer.

Options habituelles de GCC

Quelques options habituelles²⁵ (parmi plus d'une centaine) :

- `-v -time` affiche les programmes lancés (dont `cc1`) et leur temps
- `-O0` sans optimisation (compile vite, code médiocre), par défaut
- `-O1` ou `-O2` avec optimisations simples ou coûteuses
- `-O3` avec optimisations très coûteuses *"auto-inlining, vectorization"*
- `-Os` avec optimisation de la taille
- `-g` avec information de débogage
- `--help` pour l'aide (ou même `gcc -Q --help=optimizers -O1`)
- `-m32` (x86 32bits) ou `-m64` (x86-64) (ou même `-march=` ou `-mtune=`)
- `-Wall` : tous les avertissements - *très conseillé*
- beaucoup d'options `-fΩ`, y compris des optimisations supplémentaires ;
`-fno-inline -fprofile-generate -fprofile-use`

²⁵options communes à plusieurs versions de GCC

Options nouvelles de GCC 4.5

La prochaine version 4.5 apporte plusieurs nouveautés importantes.

- **optimisation à l'éditions de liens** "*link-time optimization*".
Il faut passer par exemple `-flto -O2` à la compilation et à l'édition de liens.

Il est préférable d'avoir le linker GOLD (paquet `binutils-gold`).

- **mécanisme de greffons** "*plugins*".

Le compilateur GCC peut être étendu `-fplugin=./foo.so` par des greffons extérieurs, qui pourraient par exemple :

- organiser différemment les phases de compilation
- étendre légèrement le langage, par des pragmas ou attributs
- optimiser pour une bibliothèque "*library*" particulière
- fournir un diagnostic spécifique à une application
- tout ce que vous pouvez imaginer "*everything but the kitchen sink*" !

Les greffons doivent être libres (compatibles GPLv3) !

Ecrivez vos propres greffons (par exemple avec MELT)

Autres nouveautés de GCC 4.5

<http://gcc.gnu.org/gcc-4.5/changes.html> :

- support du format de débogage Dwarf3 pour Gdb 7.0
- diagnostics affinés, notamment avec numéro de colonne
- support des dialectes expérimentaux de C, C++0x, Fortran, ...
- encore meilleur respect des standards
- optimisations avancées, dont Graphite (optimisation polyédrique) avec `-floop-parallelize-all`
- mode de profilage pour conseils sur conteneurs standards C++
- plusieurs cibles ajoutées (MeP, RX) ou ôtées (Itanium1)
- `-fwhopr` *“whole program analysis & optimization”* *expérimental*
- etc.

Conseils de travail dans GCC

Contribuer à **GCC** ou *coder son greffon*, c'est travailler dans **GCC**.
GCC est trop énorme pour être totalement compris :

- **ne pas chercher à tout comprendre**
- *lire la documentation interne*
- **demander de l'aide sur la liste** `gcc@gcc.gnu.org`
après avoir cherché un petit peu tout seul
- *s'inspirer de code existant*
mais lire seulement le code proche de ce qu'on veut faire.
- programmer défensivement
- **tester très fréquemment**
- *utiliser les bons outils* (par exemple **MELT**)
- ne pas trop bien faire ; publier son code très vite
- n'ayez pas peur !

quelques représentations internes de GCC

dump des représentations internes

```
gcc -S -O3 -fdump-tree-all t.c26
```

Le dump est une représentation *textuelle partielle* d'une représentation interne dans GCC.

En **très gros** :

- représentations *Generic* dans les frontaux (propres à chaque langage source).
- représentations *Tree* des déclarations (et du code avant "*gimplification*")
- représentations **Gimple** pour représenter le code dans le "*middle-end*"; dont *Gimple/Ssa* "*Static Single Assignment*"
- représentations *Rtl* "*register transfer language*" dans le "*back-end*" spécifique à une cible.
- une foule d'autres structures de données et variables globales.

²⁶A utiliser sur un petit fichier `t.c` seul dans son répertoire, car ça en génère une

Gestion mémoire dans GCC

Plusieurs données sont gérées manuellement (`xmalloc/xfree`), mais :

Un **ramasse-miettes** Ggc “*Gcc Garbage Collector*” gère la plupart des données importantes (dont Gimple et Tree) :

- ramasse-miettes marqueur précis
- sert aussi aux “*Pre-Compiled Headers*”
- annotations GTY sur les `struct`-ures de données et les variables (statiques ou globales)
- utilitaire `gengtype` pour générer le code de marquage
- *ne traite pas les variables locales* : il faut copier les pointeurs de données à conserver
- Ggc est lancé entre les passes de compilation²⁷, pas implicitement par l’allocateur `gcc_alloc`

²⁷Ou plus rarement explicitement par `gcc_collect`

Pratiquement :

- seules des données transmises entre passes sont traitées par Ggc
- Ggc est de mon point de vue insuffisant, donc peu investi et trop peu utilisé
- les données internes à une passe sont gérées à la main
- certaines données sont allouées à la main, ou parfois invalides.
- (presque) pas d'organisation objet avec héritage
⇒ on ne peut pratiquement pas définir des champs supplémentaires dans les données de GCC (il faut les associer ailleurs : hash-tables).
- quelques conteneurs génériques : vecteurs, hash-tables.

Les *Tree-s*

Représentation du code proche de *Generic*, et utilisée aussi pour les déclarations. Fichiers `gcc/tree.def` `gcc/tree.h` `gcc/tree.c`
Par exemple, pour les littéraux entiers

```
///// extrait de gcc/tree.def
/* Contents are in TREE_INT_CST_LOW and TREE_INT_CST_HIGH fields,
   32 bits each, giving us a 64 bit constant capability.      INTEGER_CST
   nodes can be shared, and therefore should be considered read only.
   They should be copied, before setting a flag such as TREE_OVERFLOW.
   If an INTEGER_CST has TREE_OVERFLOW already set, it is known to be unique.
   INTEGER_CST nodes are created for the integral types, for pointer
   types and for vector and float types in some circumstances. */
DEFTREECODE (INTEGER_CST, "integer_cst", tcc_constant, 0)
```

Tree littéral entier

//// extrait de tree.h

/ In an INTEGER_CST node. These two together make a 2-word integer.*

*If the data type is signed, the value is sign-extended to 2 words
even though not all of them may really be in use.*

*In an unsigned constant shorter than 2 words, the extra bits are 0. */*

```
#define TREE_INT_CST(NODE) (INTEGER_CST_CHECK (NODE)→int_cst.int_cst)
```

```
#define TREE_INT_CST_LOW(NODE) (TREE_INT_CST (NODE).low)
```

```
#define TREE_INT_CST_HIGH(NODE) (TREE_INT_CST (NODE).high)
```

```
#define INT_CST_LT(A, B)
```

```
(TREE_INT_CST_HIGH (A) < TREE_INT_CST_HIGH (B) \  
 || (TREE_INT_CST_HIGH (A) == TREE_INT_CST_HIGH (B) \  
    && TREE_INT_CST_LOW (A) < TREE_INT_CST_LOW (B)))
```

10

```
struct GTY(()) tree_int_cst {  
  struct tree_common common;  
  double_int int_cst;  
};
```

Tree 2 + 3 → 5

```
/// petit extrait de gcc/fold-const.c
```

```
/* Combine two integer constants ARG1 and ARG2 under operation CODE  
to produce a new constant. Return NULL_TREE if we don't know how  
to evaluate CODE at compile-time.
```

```
If NOTRUNC is nonzero, do not truncate the result to fit the data type. */
```

```
tree
```

```
int_const_binop (enum tree_code code, const_tree arg1, const_tree arg2, int notrunc)
```

```
{  
  unsigned HOST_WIDE_INT int1l, int2l; HOST_WIDE_INT int1h, int2h;  
  unsigned HOST_WIDE_INT low; HOST_WIDE_INT hi;  
  tree t; tree type = TREE_TYPE (arg1);  
  int uns = TYPE_UNSIGNED (type); int overflow = 0;  
  int is_sizetype = (TREE_CODE (type) == INTEGER_TYPE && TYPE_IS_SIZETYPE (type));  
  int1l = TREE_INT_CST_LOW (arg1); int1h = TREE_INT_CST_HIGH (arg1);  
  int2l = TREE_INT_CST_LOW (arg2); int2h = TREE_INT_CST_HIGH (arg2);  
  switch (code)
```

```
/// .....
```

```
  case PLUS_EXPR :
```

```
    overflow = add_double (int1l, int1h, int2l, int2h, &low, &hi);
```

```
    break;
```

Les *Gimple*-s - exemple de transformation

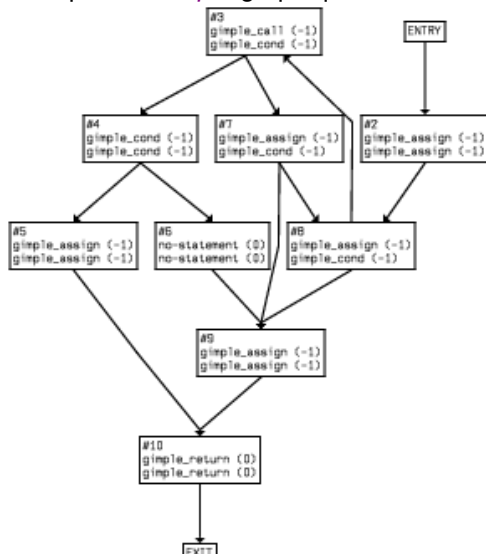
Représentations *normalisées* du code. Par exemple pour

```
/// extrait de dash-0.5.4/src/input.c
char *pfgets (char *line, int len) {
    char *p = line;
    int nleft = len;    int c;
    while (--nleft > 0) {
        c = pgetc2 ();
        if (c == PEOF) {
            if (p == line)    return NULL;
            break;
        }
        *p++ = c;
        if (c == '\n')    break;
    }
    *p = '\0';
    return line;
}
```

10

exemple de *Gimple* graphiquement

(gcc-4.4)



dump *Gimple* fichier pfgetc.c.004t.gimple

```
pfgets (char * line, int len)
{
  int c.0;
  char * D.1605;
  char D.1606;
  char * p;
  int nleft;
  int c;

  p = line;
  nleft = len;
  goto <D.1599>;
<D.1598>:
  c.0 = pgetc2 ();
  c = c.0;
  if (c == -1) goto <D.1601>;
    else goto <D.1602>;
<D.1601>:
  if (p == line) goto <D.1603>;
    else goto <D.1604>;
<D.1603>:
  D.1605 = 0B;
  return D.1605;
<D.1604>:
  goto <D.1597>;
<D.1602>:
  D.1606 = (char) c;
  *p = D.1606;
  p = p + 1;
  if (c == 10) goto <D.1597>;
    else goto <D.1607>;
<D.1607>:
<D.1599>:
  nleft = nleft - 1;
  if (nleft > 0) goto <D.1598>;
    else goto <D.1597>;
<D.1597>:
  *p = 0;
  D.1605 = line;
  return D.1605;
}
```

Gimple/SSA

static single assignment : chaque variable est affectée une seule fois \Rightarrow nœuds Φ en tête de bloc pour joindre des valeurs - dump *Gimple* pfgetc.c.119t.phiopt3

```
pfgets (char* line, int len) {
  int c; int nleft;
  char * p; char D.1606;
<bb 2>:
  goto <bb 6>;
<bb 3>:
  c_10 = pgetc2 ();
  if (c_10 == -1)
    goto <bb 4>;
  else goto <bb 5>;
<bb 4>:
  # p_22 =  $\Phi$  <p_1(3)>
  if (p_22 == line_5(D))
    goto <bb 8>;
  else goto <bb 7>;
<bb 5>:
  D.1606_13= (char)c_10;
  MEM[base: p_1] = D.1606_13;
```

```
p_14 = p_1 + 1;
if (c_10 == 10)
  goto <bb 7>;
else goto <bb 6>;
<bb 6>:
# p_1 =  $\Phi$  <line_5(D)(2), p_14(5)>
# nleft_3 =  $\Phi$  <len_7(D)(2), nleft_9(5)>
nleft_9 = nleft_3 + -1;
if (nleft_9 > 0)
  goto <bb 3>;
else goto <bb 7>;
<bb 7>:
# p_2 =  $\Phi$  <p_1(4), p_14(5), p_1(6)>
*p_2 = 0;
<bb 8>:
# line_4 =  $\Phi$  <0B(4), line_5(D)(7)>
return line_4; }
```

types de données *gimple*

Il y a un seul (gros)²⁸ type de donnée *gimple* dans les fichiers
gcc/gimple.def gcc/gimple.h gcc/gimple.c

Les différentes formes (SSA ou non, ...) de *gimple* sont représentées
dans le même type de données en C, plus ou moins contraintes.

Un nœud *gimple* contient ses arguments dans un tableau de pointeurs
tree, par exemple les Φ -nœuds dans gcc/gimple.h :

```
struct GTY(()) gimple_statement_phi {  
  /* [ WORD 1-4 ] */  
  struct gimple_statement_base gsbased;  
  /* [ WORD 5 ] */  
  unsigned capacity;  
  unsigned nargs;  
  /* [ WORD 6 ] */  
  tree result;  
  /* [ WORD 7 ] */  
  struct phi_arg_d GTY ((length ("%h.nargs"))) args[];  
};
```

²⁸L'union des `struct`-ures déclarées dans gcc/gimple.h 

le graphe de flot de contrôle

Il est représenté par plusieurs structures de données :

- 1 les séquences de gimple `gimple_seq`
- 2 les blocs élémentaires `basic_block` dans `gcc/basic-block.h`
- 3 les arêtes entre blocs `edge`
- 4 etc...

Chaque bloc connaît les arêtes en partant et y arrivant.

Chaque arête connaît ses extrémités.

Chaque instruction Gimple connaît le bloc la contenant.

Chaque fonction connaît ses blocs initial et final.

La “variable” `cfun` de type `struct function` dans `gcc/function.h` est la fonction compilée courante.
`gcc/cgraph.h` décrit les liens d'appels entre fonctions...

les traitements internes de GCC

La compilation est structurée en **passes** de compilation.

- Il y a *beaucoup* (≈ 200) de passes dans GCC.
- Leur lancement dépend des optimisations demandées.
- Chacune est déclarée dans `gcc/tree-passes.h` et lancée par `gcc/passes.c`.
- Chaque passe (ou jeu de passes) a son propre fichier source.
- Une passe peut être lancée 0 ou plusieurs fois.
- l'ordre des passes est (plus ou moins) dynamique.
- un greffon peut ajouter (ou ôter) des passes.
- une passe est organisée en arborescence (avec des sous-passes).

les passes internes de GCC

Il s'agit des passes après analyse syntaxique.

- passe simple GIMPLE `GIMPLE_PASS` sur une fonction
- passe interprocédurale simple *“Inter Procedural Analysis”*
`SIMPLE_IPA_PASS`
- passe interprocédurale complexe `IPA_PASS`
- passe de génération *“Register Transfer Language”* `RTL_PASS`

Chaque passe a une fonction de porte *“gate”* (décidant si on la lance ou non) et une fonction d'exécution *“execute”*.

```
//descripteur de passe dans tree-pass.h
struct opt_pass {
    enum opt_pass_type type; //le type, GIMPLE-PASS etc..
    const char *name; //le nom de la passe
    bool (*gate) (void); //la fonction porte
    unsigned int (*execute) (void); //la fonction d'exécution
    /* A list of sub-passes to run, dependent on gate predicate. */
    struct opt_pass *sub;
    /* Next in the list of passes to run, independent of gate predicate.
    struct opt_pass *next;
    /* Static pass number, used as a fragment of the dump file name. */
    int static_pass_number;
    /* The timevar id associated with this pass. */
    timevar_id_t tv_id;
    /* Sets of properties input and output from this pass. */
    unsigned int properties_required;
    unsigned int properties_provided;
    unsigned int properties_destroyed;
    /* Flags indicating common sets things to do before and after. */
    unsigned int todo_flags_start;
    unsigned int todo_flags_finish;
};
```

l'outil MELT

“Middle End Lisp Translator”

motivations et contexte de MELT

- Un outil pour faciliter le développement de passes d'analyse statique.
- Les outils d'analyse statique (Frama-C...) sont algorithmiquement difficiles.
- Les analyseurs statiques sont généralement codés en Ocaml (ou Lisp). Les coder en C est suicidaire.
- **impossibilité pratique de brancher un langage** de haut niveau (OCaml) dans GCC :

 - GCC est énorme
 - GCC évolue trop vite
 - GCC n'a pas d'API figée
 - GCC est codé en C avec ses conventions et styles particuliers

- Financé par le projet ITEA GlobalGCC

Idées directrices et objectifs de MELT

- traits linguistiques de haut niveau :
 - 1 valeurs fonctionnelles
 - 2 langage dynamique à objets
 - 3 filtrage *“pattern matching”*
 - 4 meta-programmation
- **génération de code C** conforme aux styles et usages internes de GCC
- langage amorcé (le traducteur MELT est codé en MELT).
- traite aussi bien des valeurs *“values”* (fermetures, objets, avec ramasse-miettes performant) que des trucs de GCC *“stuff”*.
- **s'adapte facilement aux évolutions de GCC**
- constructions de haut niveau pour générer du C
- syntaxe lispienne (une syntaxe alternative infix apparaît)

installation et utilisation de MELT

- MELT est une branche de GCC très souvent fusionnée avec le tronc.
- MELT est aussi un greffon de GCC

MELT s'installe donc facilement comme greffon

MELT s'utilise aussi facilement que

```
gcc  
-fplugin=.../melt.so  
-fplugin-arg-melt-mode=green  
foo.c
```

Il faut de la RAM (4Go) pour compiler MELT, car il y a 540KLOC de code C généré pour 34KLOC de source MELT.

Aperçu du langage MELT

Les valeurs sont typées dynamiquement

Les trucs (`:gimple`, `:edge`, `:tree`, `:long`, `:cstring`...) sont typés statiquement et déclarés

Chaque valeur a son discriminant (sa classe si c'est un objet) (comme `DISCR_INTEGER`, ou `CLASS_CLASS`) - c'est un objet MELT

Nil (= faux) représenté par `(void*)0` a conventionnellement `DISCR_NULL_RECEIVER` pour discriminant.

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 \neq `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` \neq `:gimple`]

Values are dynamically typed : closures are tested when applied

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

In MELT function applications²⁹ : first (if any) argument is a value, secondary arguments are things. Result is a value, secondary results (if any) are things. [unusual calling protocol]

²⁹So also method calls

Code chunks

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

```
(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 translated³⁰ to C code :

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

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

³⁰supposing `i → 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

<code>ctype :=</code>	<code>:long :gimple :tree ... :value :void</code>	c-types
<code>formals :=</code>	<code>∅ ctype[?] var⁺ formals</code>	formal arguments
<code>consexpr :=</code>	constructible expressions <code>(instance name_{class} { :name_{field} expr }[*]) (list expr[*]) (tuple expr[*]) (lambda (formals) expr⁺)</code>	object creation list creation tuple creation abstraction
<code>expr :=</code>	expressions <code>var literal consexpr (expr expr[*]) (quote literal) (quote name) (get_field name; ... expr)</code>	variables integer, string, nil constructions function application quotation field access in object

Syntax (2) – expressions...

<code>expr</code>	<code>:= ...</code>	expressions (continued)
	<code>(name_{primitive} expr*)</code>	primitive invocation
	<code>(name_{selector} expr_{receiver} expr*)</code>	message sending
	<code>(return expr*)</code>	function return
	<code>(setq var expr)</code>	assignment
	<code>(code_chunk name_{state} macrostring)</code>	C chunk
	<code>(progn expr+)</code>	sequentiality
	<code>(put_fields expr { :name_{field} expr }*)</code>	object update
	<code>(exit name_{label} expr+)</code>	local exit from loop
	<code>(if expr_{test} expr_{then} expr_{else}?)</code>	if-then-else
	<code>(and expr+)</code>	and then
	<code>(or expr+)</code>	or else
	<code>(cond (expr_{cond} expr_{then}+)*</code>	multiple conditional
	<code> (:else expr_{else}+)?)</code>	
	<code>(match expr (pattern expr+)+)</code>	pattern matching

Syntax (3) – binding expressions & definitions...

<code>expr</code>	<code>:= ...</code>	expressions (cont.)
	<code>(forever <i>name</i>_{label} <i>expr</i>⁺)</code>	loop
	<code>(<i>name</i>_{iterator} (<i>expr</i>[*]) (<i>formals</i>) <i>expr</i>⁺)</code>	iteration
	<code>(let ((<i>ctype</i>[?] <i>name</i> <i>expr</i>)[*]) <i>expr</i>⁺)</code>	sequential let
	<code>(letrec ((<i>name</i> <i>consexpr</i>)[*]) <i>expr</i>⁺)</code>	recursive let

<code>def :=</code>	definitions	
	<code>(defun <i>name</i> (<i>formals</i>) <i>expr</i>⁺)</code>	function def
	<code>(defprimitive <i>name</i> (<i>formals</i>) <i>ctype</i> <i>macrostring</i>)</code>	primitive def
	<code>(defclass <i>name</i> { :super <i>name</i>_{super} }[?] { :fields (<i>name</i>[*]) }[?])</code>	class def
	<code>(defselector <i>name</i> <i>name</i>_{class} { :formals (<i>formals</i>) }[?])</code>	selector def
	<code>(definstance <i>name</i> <i>name</i>_{class} { :<i>name</i>_{field} <i>expr</i> }[*])</code>	instance def
	<code>(defciterator <i>name</i> (<i>formals</i>_{start}) <i>name</i>_{state} (<i>formals</i>_{locals}) <i>macrostring</i>_{start} <i>macrostring</i>_{end})[*]</code>	c-iterator def

Syntax (4) – directives

dir :=

```
(export_values name*)  
| (export_class name*)  
| (export_macro namemacro exprexpander)  
| (export_patmacro namemacro  
    exprexpr-expander exprpatt-expander)  
| (export_synonym namenew nameold)  
| (load stringfile-path)
```

directives

export defined values
export defined classes³¹
export macro
[for pattern macros]
export patmacro
export synonym
include file

³¹ Also export the fields of classes !

Syntax (5) – miscellaneous

<code>expr :=</code>	<code>...</code>	<code>expr (cont.)</code>
	<code>(current_module_environment_container)</code>	for my env.
	<code>(parent_module_environment)</code>	parent env.
	<code>(debug_msg expr string)</code>	debug print ³²
	<code>(assert_msg string expr)</code>	assertion ³³
	<code>(compile_warning string expr)</code>	warning ³⁴
	<code>(cpp_if name expr_{then} expr_{else})</code>	cpp #if
	<code>...</code>	etc

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

NB : Syntax for pattern matching is given below.

³²when asked by option, debug printing with MELT source location.

³³shows the MELT stack on assertion failure !

³⁴Similar to #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

```
{ 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                    ;state
  (:gimple g)                      ;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 ϕ .melt file translated (via ϕ .c file) to ϕ .so module [single name-space]

The module's `start_module_melt` routine

- 1 takes a parent environment η ³⁵,
- 2 build various initial data (closures, ...),
- 3 executes top-level forms,
- 4 returns the module's environment η' containing the *exported definitions* of module ϕ (both η and η' are instances of `CLASS_ENVIRONMENT`).

so `start_module_melt` is mostly sequential and quite big.

Bindings are instances of sub-classes of `CLASS_ANY_BINDING`

³⁵For the very first module `warmelt-first` - translated specially - η is nil. 

exported definitions

- functions, selectors, c-iterators, ... (`export_value ...`)
- classes and their fields with (`export_class ...`). Classes and fields names are *globally* unique ³⁶.
- 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.

³⁶So (`get_field :named_name n`) translated to test that `n` is a `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 selectors classes ...).

Library : collecting types (map/reduce/...), iterators, primitives, functions, selectors, macros, ... debug printing ; run-time asserts³⁷ ; 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.

³⁷printing 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 evolution³⁸ [*Tree* ⇒ *Gimple* transition]

³⁸because `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**.

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` \equiv (`question` π) exactly like `' ϵ` \equiv (`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

<code>def := ...</code>		definitions [cont.]
	<code>(defcmatcher name (formals_{match&inputs}) (formals_{outputs}) name_{state} macrostring_{test} macrostring_{fill})</code>	c-matcher
	<code>(defunmatcher name (formals_{match&inputs}) (formals_{outputs}) expr_{matching} expr_{applying})</code>	fun-matcher

A matcher appears in pattern construct³⁹. A C-matcher defines its behavior by C expansion ; a fun-matcher defines it by a multivalued function.

³⁹Matchers 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

constant patterns

joker

pattern variable

matcher pattern

instance pattern

object pattern

tuple pattern

list pattern

as pattern

conjunctive pattern

disjunctive pattern

conditionned pattern

Patterns appear only in **match** expressions.

Pattern example

```
(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 ?callfn decl
        ?(tree_function_decl ?(cstring_same "fprintf") ?_)
        ?argfile ?argfmt ?nbargs)
      [handle the fprintf case to file argfile with format argfmt])
    (?_ [otherwise...]))
```

A bit naive to look for `fprintf` with string compare

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 variables⁴⁰. 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

⁴⁰i.e. `lhs not ?lhs`

pattern taxonomy

Patterns (when matching thing τ) can be

- 1 joker `?_` - always match
- 2 constants⁴¹ ξ - matches if ξ *identical* [in C ==] to τ
- 3 pattern variables like `?x`
if unset, set it to τ . Otherwise test for identity with τ
Non-linear patterns : `?(gimple_assign_simple ?var ?var)` useful to find assignments of a C variable var to itself.
- 4 control patterns :
 - `?(as ?var π)` : set or test `?var` to τ , then match π against τ
 - `?(and π_1 ... π_n)` matches iff π_1 matches τ and then π_2 ...
 - `?(or π_1 ... π_n)` matches iff π_1 matches τ or else π_2 ...
- 5 elementary *matchers* like `gimple_assign_cast` or `string_same`
- 6 object patterns `?(instance class :field1 π_1 ... :fieldk π_k)`
(some fields can be missing) and others object patterns

⁴¹Constants 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 $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)) }# )
```

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) 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* (with no other inputs), and `lhs` `rhs` are the output trees. `gimpascs` 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 `?(isbiggereven μ π)`⁴² 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 .

```
(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.

⁴²`isbiggereven` could be a c-matcher !

Coder des passes en MELT

Montrer le fichier ana-simple.melt

conclusion : codez vos extensions en MELT

Les greffons de GCC serviront à des **traitements spécifiques à un logiciel libres**.

Codez vos greffons pour GCC (modules) **en MELT** , par exemple

- diagnostic spécifique (par exemple vérifier que chaque fopen est testé)
- optimisations spécifiques
- aide au refactoring, navigation, métriques

Contribuez vos greffons en les développant en **MELT** pour des gros logiciels libres (Gnome, Kde, Kernel, Qt, ...)

Je vous aiderais