# 走进 GCC 插件时代

HelloGCC 2011

邢明杰

# 纲要

- 插件的由来
- 好处与问题
- 目前的实现
- 现有插件简介

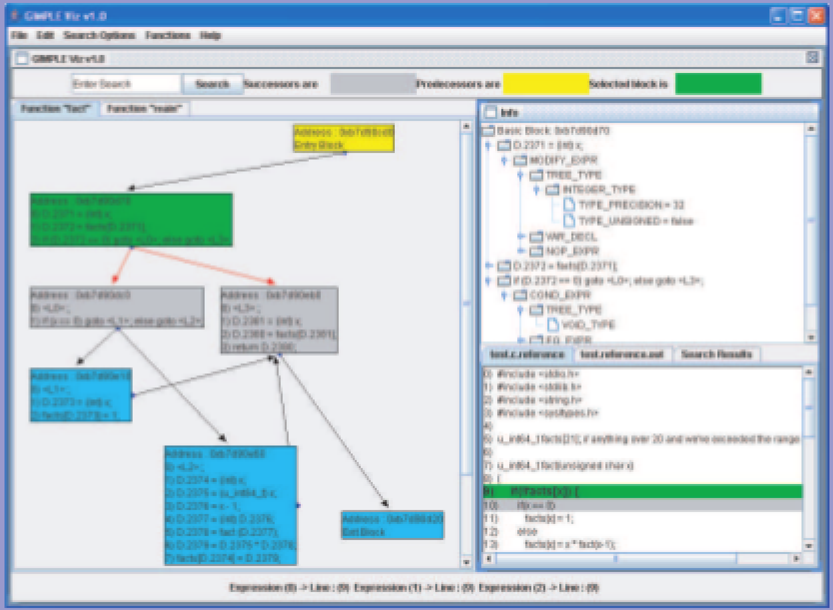
# 插件的由来

**Extending GCC with Modular GIMPLE Optimizations**

Sean Callanan, Daniel J. Dean, and Erez Zadok

Stony Brook University

GCC Summit 2007



```
Basic Block 0xb7d98d70
  D.2371 = (int) x;
    MODIFY_EXPR
      TREE_TYPE
        INTEGER_TYPE
          TYPE_PRECISION = 32
          TYPE_UNSIGNED = false
      VAR_DECL
      NOP_EXPR
  D.2372 = facts[D.2371];
  if (D.2372 == 0) goto <L0>; else goto <L3>;
    COND_EXPR
      TREE_TYPE
        VOID_TYPE
      EQ_EXPR
```

```
0)  #include <stdio.h>
1)  #include <stdlib.h>
2)  #include <string.h>
3)  #include <sys/types.h>
4)
5)  u_int64_t facts[21]; // anything over 20 and we've exceeded the range
6)
7)  u_int64_t fact(unsigned char x)
8)  {
9)      if(!facts[x]) {
10)         if(x == 0)
11)             facts[x] = 1;
12)         else
13)             facts[x] = x * fact(x-1);
```

- 实现了类似 Eclipse 的插件系统
- 实现了多个插件和可视化工具
  - verbose dump plugin
  - call trace plugin
  - malloc checking plugin
  - bounds checking plugin
  - Gimple Viz

# 插件的好处

- 独立发布，方便维护

  不再使用 patch 方式，gcc-xml， EDoc++

- 单独编译，节省时间

  更新插件时，无需重新编译 GCC 本身

- 动态加载，使用方便

  不需要安装多个工具链

- 易于扩展，方便实现额外的功能，特性

  静态分析，代码重构，可视化，代码导航等等

- 快速构建，方便做实验，研究

  快速搭建原型系统，新的优化，性能调优

I think it's quite important for gcc's long-term health to permit and even encourage academic researchers and students to use it.

# GCC XML

```
struct EmptyClass {};
int a_function(float f, EmptyClass e)
{
}
int main(void)
{
  return 0;
}
```

```
$ gccxml example1.cxx -fxml=example1.xml
```

```
<?xml version="1.0"?>
<GCC_XML>
  <Namespace id="_1" name="::" members="_2 _3 _4 "/>
  <Function id="_2" name="main" returns="_5" context="_1" location="f0:8"/>
  <Function id="_3" name="a_function" returns="_5" context="_1" location="f0:4">
    <Argument name="f" type="_6"/>
    <Argument name="e" type="_4"/>
  </Function>
  <Struct id="_4" name="EmptyClass" context="_1" location="f0:1" members="_7 _8 " bases=""/>
  <FundamentalType id="_5" name="int"/>
  <FundamentalType id="_6" name="float"/>
  <Constructor id="_7" name="EmptyClass" context="_4" location="f0:1">
    <Argument name="_ctor_arg" type="_9"/>
  </Constructor>
  <Constructor id="_8" name="EmptyClass" context="_4" location="f0:1"/>
  <ReferenceType id="_9" type="_4c"/>
  <File id="f0" name="example1.cxx"/>
</GCC_XML>
```

# 问题与争执

- GPL 问题
  - 会使得私有代码很容易被集成到 GCC 中
  - 私有（proprietary）插件
  - 过渡插件（marshalling plugin, shim layer）

    GCC 前端 + 私有编译器
- 开源问题
  - 鼓励参与社区，贡献代码
- 基本共识
  - 支持插件，同时防止私有插件
  - 控制在 GPL 范围内

RMS is worried that this will make it too easy to integrate proprietary code directly with GCC.

# 观点，建议

- 强制 GPL 许可证

  包含带有 GPL 许可证的头文件；进行许可证检查

- 不保证稳定的插件 API

  不设计标准的 API，不同版本之间会有变动

- 把结构设计的复杂些

  比如 tree，使其不容易被导出，导入

  但是，复杂的结构不利于新人参与！

- 我们过于担心了

  没有插件，问题依然会存在

  即使有私有插件，我们也可以仿照实现一个 GPL 的

a major argument against this risk is that there is no stability in the API offered by GCC to plugins. So any proprietary plugin would be a nightmare to maintain

# 目前的实现

- 加载方式

```
-fplugin=/path/to/name.so -fplugin-arg-name-key[=value]
```

- 构建方法

```
GCC=gcc
PLUGIN_SOURCE_FILES= plugin1.c plugin2.c
PLUGIN_OBJECT_FILES= $(patsubst %.c,%.o,$
(PLUGIN_SOURCE_FILES))
GCCPLUGINS_DIR:= $(shell $(GCC) -print-file-name=plugin)
CFLAGS+= -I$(GCCPLUGINS_DIR)/include -fPIC -O2

plugin.so: $(PLUGIN_OBJECT_FILES)
	$(GCC) -shared $^ -o $@
```

# 目前的实现

- 检查许可证

```
int plugin_is_GPL_compatible;
```

- 初始化

```
#include "plugin-version.h"
int
plugin_init (struct plugin_name_args *plugin_info,
             struct plugin_gcc_version *version)
{
  if (!plugin_default_version_check (version, &gcc_version))
    return 1;
  ...
}
```

# 目前的实现

- 事件
  - type
  - declare
  - ggc
  - attribute
  - pragma
  - pass

```
enum plugin_event
{
  PLUGIN_PASS_MANAGER_SETUP,
  PLUGIN_FINISH_TYPE,
  PLUGIN_FINISH_DECL,
  PLUGIN_FINISH_UNIT,
  PLUGIN_PRE_GENERICIZE,
  PLUGIN_FINISH,
  PLUGIN_INFO,
  PLUGIN_GGC_START,
  PLUGIN_GGC_MARKING,
  PLUGIN_GGC_END,
  PLUGIN_REGISTER_GGC_ROOTS,
  PLUGIN_REGISTER_GGC_CACHES,
  PLUGIN_ATTRIBUTES,
  PLUGIN_START_UNIT,
  PLUGIN_PRAGMAS,
  PLUGIN_ALL_PASSES_START,
  PLUGIN_ALL_PASSES_END,
  PLUGIN_ALL_IPA_PASSES_START,
  PLUGIN_ALL_IPA_PASSES_END,
  PLUGIN_OVERRIDE_GATE,
  PLUGIN_PASS_EXECUTION,
  PLUGIN_EARLY_GIMPLE_PASSES_START,
  PLUGIN_EARLY_GIMPLE_PASSES_END,
  PLUGIN_NEW_PASS,
  PLUGIN_EVENT_FIRST_DYNAMIC
};
```

# 目前的实现

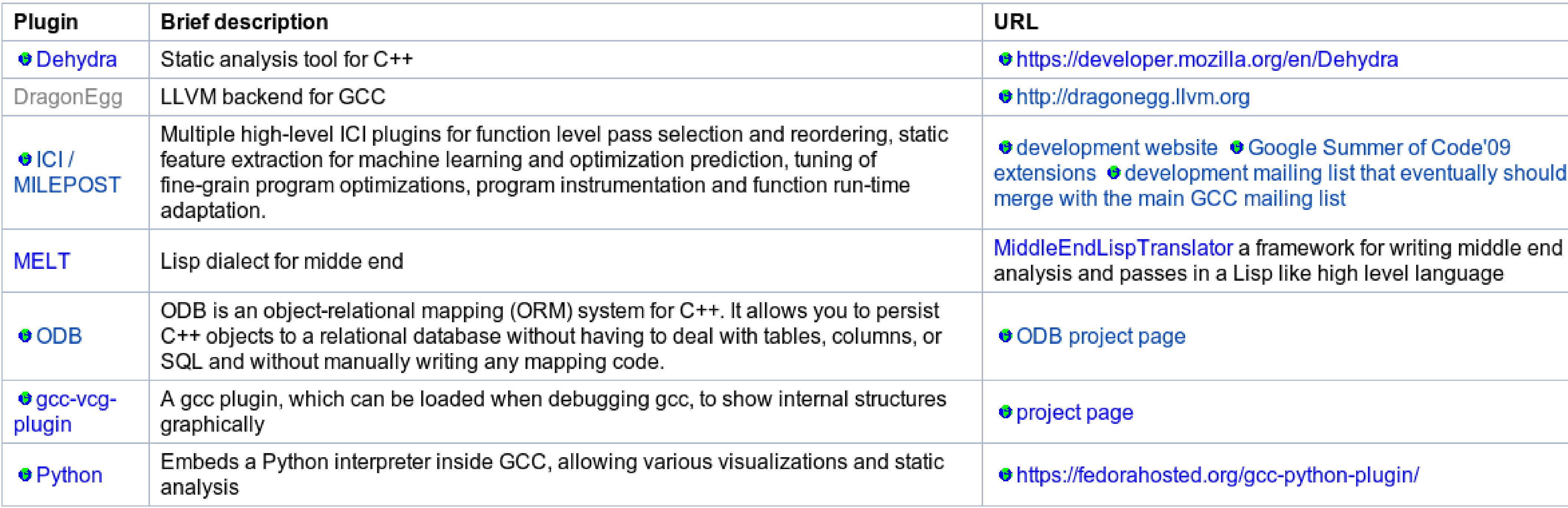
- 回调函数

```
/* The prototype for a plugin callback function.
   gcc_data  - event-specific data provided by GCC
   user_data - plugin-specific data provided by the plug-in.  */
typedef void (*plugin_callback_func)(void *gcc_data, void *user_data);
```

- 注册

```
extern void register_callback (const char *plugin_name,
                               int event,
                               plugin_callback_func callback,
                               void *user_data);
```

# 现有插件简介



| Plugin | Brief description | URL |
|---|---|---|
| Dehydra | Static analysis tool for C++ | https://developer.mozilla.org/en/Dehydra |
| DragonEgg | LLVM backend for GCC | http://dragonegg.llvm.org |
| ICI / MILEPOST | Multiple high-level ICI plugins for function level pass selection and reordering, static feature extraction for machine learning and optimization prediction, tuning of fine-grain program optimizations, program instrumentation and function run-time adaptation. | development website Google Summer of Code'09 extensions development mailing list that eventually should merge with the main GCC mailing list |
| MELT | Lisp dialect for midde end | MiddleEndLispTranslator a framework for writing middle end analysis and passes in a Lisp like high level language |
| ODB | ODB is an object-relational mapping (ORM) system for C++. It allows you to persist C++ objects to a relational database without having to deal with tables, columns, or SQL and without manually writing any mapping code. | ODB project page |
| gcc-vcg-plugin | A gcc plugin, which can be loaded when debugging gcc, to show internal structures graphically | project page |
| Python | Embeds a Python interpreter inside GCC, allowing various visualizations and static analysis | https://fedorahosted.org/gcc-python-plugin/ |

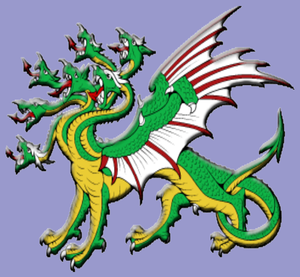
摘自 http://gcc.gnu.org/wiki/plugins

# Dehydra & Treehydra

Dehydra： C++ 静态分析工具，将 C++ 类型和变量表示为 JavaScript 对象，并提供相应的处理回调函数，用户使用 JavaScripts 来编写分析脚本。

Treehydra： 类似 Dehydra，提供了 C++ 抽象语法树， GIMPLE 抽象语法树以及控制流图的处理回调函数。

这些插件被用于分析 Mozilla 源代码。



```
dumptypes.cc:

typedef int MyInt;

struct Foo { int i; char *c; };
```

```
dumptypes.js:

function process_type(t)
{
  print("Type found: " + t.name + " location: " + t.loc);
}
function input_end()
{
  print("Hello, world!");
}
```

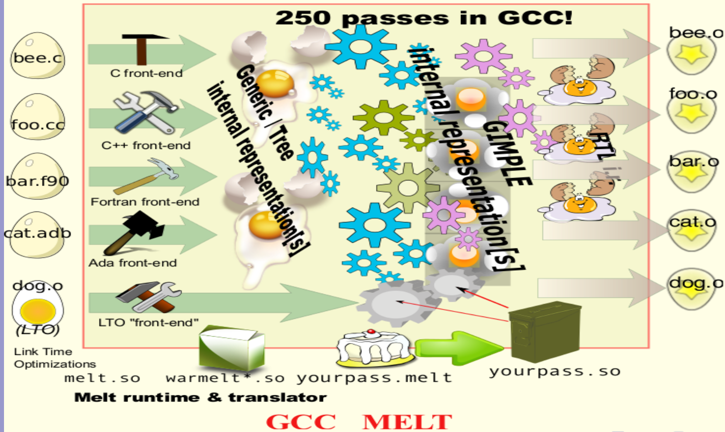
```
$ g++ -fplugin=~/dehydra/gcc_dehydra.so -fplugin-arg-gcc_dehydra-script=~/dumptypes.js -o/dev/null -c dumptypes.cc

Type found: Foo location: test.cc:2:12
Type found: MyInt location: test.cc:1:13
Hello, world!
```

# MELT

MELT： Middle End Lisp Translator

用来辅助（加速）开发 GCC 扩展功能（插件），使用面向 GCC 定制的 Lisp 语言进行开发，将其转换为 C 语言代码，并生成相应的插件。

作者 Basile STARYNKEVITCH，工作经验包括 Ocaml 语言开发， JIT 开发，垃圾收集器 Qish 等。



```
;;;;;;; file hello.melt                                  -*- lisp -*-
;;; a comment for the generated C code
(comment "hello world is public domain")
;;; a code chunk containing C
(code_chunk say-hello-chunk
                    #{printf("hello from MELT %s:%d\n",
                                 __FILE__, __LINE__);}#)
;;;;;;; eof hello.melt
```

```
$ gcc-melt -fmelt-mode=runfile -fmelt-arg=hello.melt -c empty.c

hello from MELT hello.melt:8
```

```
$ gcc-melt -fmelt-mode=translatetomodule -fmelt-arg=hello.melt
-c empty.c

$ ls hello.so
```

# GCC VCG Plugin

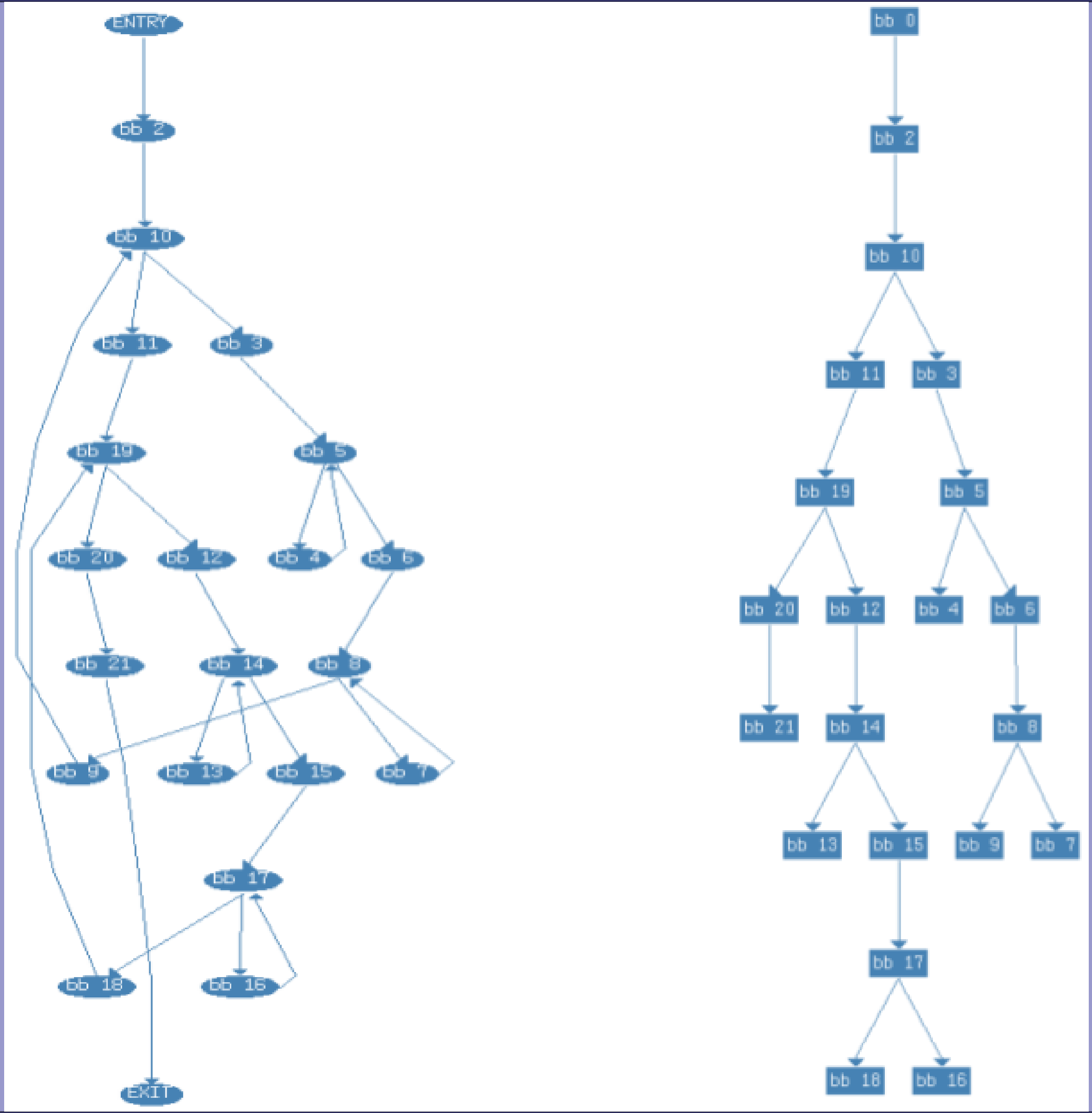
- 通过 gcc 命令行使用 ( -fplugin-arg-vcg_plugin-option )
  - cgraph ---- dump the call graph before IPA passes.
  - cgraph-callee ---- dump the callee graph for each function.
  - cgraph-caller ---- dump the caller graph for each function.
  - gimple-hierarchy ---- dump the gimple hierarchy graph.
  - help ---- show this help.
  - passes ---- dump the passes graph.
  - pass-lists ---- dump the pass lists graph.
  - tree-hierarchy ---- dump the tree hierarchy graph.
  - tree-hierarchy-4-6 ---- dump the tree hierarchy graph for gcc 4.6.
  - tree-hierarchy-4-7 ---- dump the tree hierarchy graph for gcc 4.7.
  - viewer=name ---- set the vcg viewer, default is vcgview.

# GCC VCG Plugin

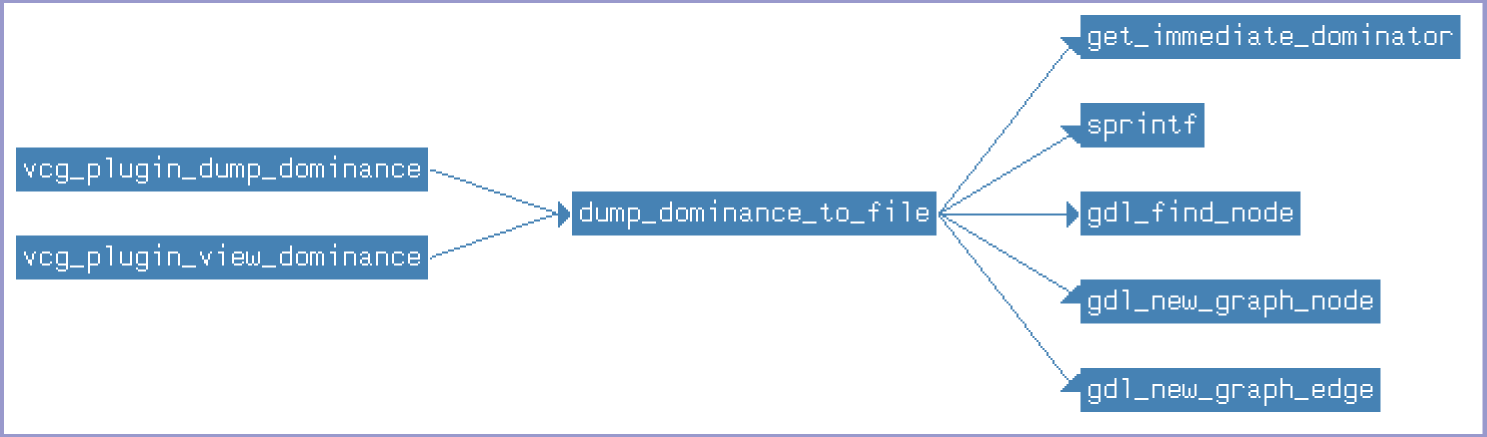
- 通过 gdb 自定义命令使用 ( view-option 或者 dump-option )
  - view-bb ---- show the basic block
  - view-bbs ---- show the basic blocks
  - view-cfg ---- show the current control flow graph in tree-level
  - view-cgraph ---- show the current call graph
  - view-dominance ---- show the current dominance graph
  - view-gimple-hierarchy ---- show gimple statement structure hierarchy
  - view-loop ---- show the loop
  - view-pass-lists ---- show pass lists
  - view-rtx ---- show a specified rtx
  - view-tree ---- show a specified tree
  - view-tree-hierarchy ---- show tree structure hierarchy
  - view-tree-hierarchy-4-6 ---- show tree structure hierarchy for gcc 4.6
  - view-tree-hierarchy-4-7 ---- show tree structure hierarchy for gcc 4.7

# GCC VCG Plugin

```
$ gdb -q -args /path/to/cc1 -O3
-fplugin=/path/to/vcg-plugin.so foo.c
(gdb) so /path/to/vcg-plugin.gdbinit
(gdb) b execute_build_cfg
(gdb) r
(gdb) finish
(gdb) view-cfg
(gdb) view-dominance
```



# GCC VCG Plugin



```
$ export VCGPLUGIN=/path/to/vcg-plugin.so
$ gcc -fplugin=$VCGPLUGIN -fplugin-arg-vcg_plugin-cgraph -c foo.c
$ gcc -fplugin=$VCGPLUGIN -fplugin-arg-vcg_plugin-cgraph-callee -c foo.c
$ gcc -fplugin=$VCGPLUGIN -fplugin-arg-vcg_plugin-cgraph-caller -c foo.c
```

# 问题?