| 21.1  | Interprocedural analysis (IPA)                      | 109 |
| 21.2  | Whole-program Analysis via Link-Time Optimization (LTO) | 110 |
| 21.3  | Inspecting GCC’s command-line options               | 111 |
| 21.4  | Working with GCC’s tunable parameters               | 113 |
| 21.5  | Working with the preprocessor                       | 114 |
| 21.6  | Version handling                                    | 115 |
| 21.7  | Register Transfer Language (RTL)                    | 116 |
| 22    | Release Notes                                       | 117 |
| 22.1  | 0.16                                               | 117 |
| 22.2  | 0.15                                               | 117 |
| 22.3  | 0.14                                               | 118 |
| 22.4  | 0.13                                               | 118 |
| 22.5  | 0.12                                               | 120 |
| 22.6  | 0.11                                               | 122 |
| 22.7  | 0.10                                               | 123 |
| 22.8  | 0.9                                                | 125 |
| 22.9  | 0.8                                                | 127 |
| 22.10 | 0.7                                                | 130 |
| 23    | Appendices                                          | 137 |
| 23.1  | All of GCC’s passes                                 | 137 |
| 23.2  | gcc.Tree operators by symbol                        | 143 |
| 24    | Indices and tables                                  | 147 |
Contents:
The plugin has the following requirements:

- GCC: 4.6 or later (it uses APIs that weren’t exposed to plugins in 4.5)
- Python: requires 2.7 or 3.2 or later
- “six”: The libcpychecker code uses the “six” Python compatibility library to smooth over Python 2 vs Python 3 differences, both at build-time and run-time:
  
  http://pypi.python.org/pypi/six/
- “pygments”: The libcpychecker code uses the “pygments” Python syntax-highlighting library when writing out error reports:
  
  http://pygments.org/
- “lxml”: The libcpychecker code uses the “lxml” internally when writing out error reports.
- graphviz: many of the interesting examples use “dot” to draw diagrams (e.g. control-flow graphs), so it’s worth having graphviz installed.
Various distributions ship with pre-built copies of the plugin. If you’re using Fedora, you can install the plugin via RPM on Fedora 16 onwards using:

```bash
yum install gcc-python2-plugin
```

as root for the Python 2 build of the plugin, or:

```bash
yum install gcc-python3-plugin
```

for the Python 3 build of the plugin.

On Gentoo, use `layman` to add the `dMaggot` overlay and `emerge` the `gcc-python-plugin` package. This will build the plugin for Python 2 and Python 3 should you have both of them installed in your system. A live ebuild is also provided to install the plugin from git sources.
3.1 Build-time dependencies

If you plan to build the plugin from scratch, you’ll need the build-time dependencies. On a Fedora box you can install them by running the following as root:

```
yum install gcc-plugin-devel python-devel python-six python-pygments graphviz
```
for building against Python 2, or:

```
yum install gcc-plugin-devel python3-devel python3-six python3-pygments graphviz
```
when building for Python 3.

3.2 Building the code

You can obtain the source code from git by using:

```
$ git clone git@github.com:davidmalcolm/gcc-python-plugin.git
```

To build the plugin, run:

```
make plugin
```

To build the plugin and run the selftests, run:

```
make
```

You can also use:

```
make demo
```
to demonstrate the new compiler errors.

By default, the Makefile builds the plugin using the first `python-config` tool found in $PATH (e.g. /usr/bin/python-config), which is typically the system copy of Python 2. You can override this (e.g. to build against Python 3) by overriding the `PYTHON` and `PYTHON_CONFIG` Makefile variables with:

```
make PYTHON=python3 PYTHON_CONFIG=python3-config
```

There isn’t a well-defined process yet for installing the plugin (though the rpm specfile in the source tree contains some work-in-progress towards this).


**Note:** Unfortunately, the layout of the header files for GCC plugin development has changed somewhat between different GCC releases. In particular, older builds of GCC flattened the “c-family” directory in the installed plugin headers.

This was fixed in this GCC commit:

```
http://gcc.gnu.org/viewcvs?view=revision&revision=176741
```

So if you’re using an earlier build of GCC using the old layout you’ll need to apply the following patch (reversed with “-R”) to the plugin’s source tree to get it to compile:

```
$ git show 215730cbec40a6fe482fabb7f1ecc3d747f1b5d2 | patch -p1 -R
```

If you have a way to make the plugin’s source work with either layout, please email the plugin’s [mailing list](mailto:mailing list)
Once you’ve built the plugin, you can invoke a Python script like this:

```bash
gcc -fplugin=./python.so -fplugin-arg-python-script=PATH_TO_SCRIPT.py OTHER_ARGS
```

and have it run your script as the plugin starts up.

Alternatively, you can run a one-shot Python command like this:

```bash
gcc -fplugin=./python.so -fplugin-arg-python-command="python code" OTHER_ARGS
```

such as:

```bash
gcc -fplugin=./python.so -fplugin-arg-python-command="import sys; print(sys.path)" OTHER_ARGS
```

The plugin automatically adds the absolute path to its own directory to the end of its `sys.path`, so that it can find support modules, such as `gccutils.py` and `libcpychecker`.

There is also a helper script, `gcc-with-python`, which expects a python script as its first argument, then regular gcc arguments:

```bash
./gcc-with-python PATH_TO_SCRIPT.py other args follow
```

For example, this command will use graphviz to draw how GCC “sees” the internals of each function in `test.c` (within its SSA representation):

```bash
./gcc-with-python examples/show-ssa.py test.c
```

Most of the rest of this document describes the Python API visible for scripting.

The plugin GCC’s various types as Python objects, within a “gcc” module. You can see the API by running the following within a script:

```python
import gcc
help(gcc)
```
To make this easier, there’s a script to do this for you:

```
./gcc-python-docs
```

from where you can review the built-in documentation strings (this document may be easier to follow though). The exact API is still in flux: and may well change (this is an early version of the code; we may have to change things as GCC changes in future releases also).

## 4.1 Debugging your script

You can place a forced breakpoint in your script using this standard Python one-liner:

```
import pdb; pdb.set_trace()
```

If Python reaches this location it will interrupt the compile and put you within the `pdb` interactive debugger, from where you can investigate.

See [http://docs.python.org/library/pdb.html#debugger-commands](http://docs.python.org/library/pdb.html#debugger-commands) for more information.

If an exception occurs during Python code, and isn’t handled by a try/except before returning into the plugin, the plugin prints the traceback to stderr and treats it as an error:

```
/home/david/test.c: In function 'main':
/home/david/test.c:28:1: error: Unhandled Python exception raised within callback
Traceback (most recent call last):
  File "test.py", line 38, in my_pass_execution_callback
    dot = gccutils.tree_to_dot(fun)
NameError: global name 'gccutils' is not defined
```

(In this case, it was a missing `import` statement in the script)

GCC reports errors at a particular location within the source code. For an unhandled exception such as the one above, by default, the plugin reports the error as occurring at the top of the current source function (or the last location within the current source file for passes and callbacks that aren’t associated with a function).

You can override this using `gcc.set_location`:

```
gcc.set_location(loc)
```

Temporarily overrides the error-reporting location, so that if an exception occurs, it will use this `gcc.Location`, rather than the default. This may be of use when debugging tracebacks from scripts. The location is reset each time after returning from Python back to the plugin, after printing any traceback.

## 4.2 Accessing parameters

```
gcc.argument_dict
```

Exposes the arguments passed to the plugin as a dictionary.

For example, running:

```
gcc -fplugin=python.so \
    -fplugin-arg-python-script=test.py \
    -fplugin-arg-python-foo=bar
```

with `test.py` containing:
import gcc
print(gcc.argument_dict)

has output:

{'script': 'test.py', 'foo': 'bar'}

gcc.argument_tuple
Exposes the arguments passed to the plugin as a tuple of (key, value) pairs, so you have ordering. (Probably worth removing, and replacing argument_dict with an OrderedDict instead; what about duplicate args though?)

4.3 Adding new passes to the compiler

You can create new compiler passes by subclassing the appropriate gcc.Pass subclasses. For example, here’s how to wire up a new pass that displays the control flow graph of each function:

```python
# Show the GIMPLE form of each function, using GraphViz
import gcc
from gccutils import get_src_for_loc, cfg_to_dot, invoke_dot

# We'll implement this as a custom pass, to be called directly after the
# builtin "cfg" pass, which generates the CFG:

class ShowGimple(gcc.GimplePass):
    def execute(self, fun):
        # (the CFG should be set up by this point, and the GIMPLE is not yet
        # in SSA form)
        if fun and fun.cfg:
            dot = cfg_to_dot(fun.cfg, fun.decl.name)
            # print dot
            invoke_dot(dot, name=fun.decl.name)

ps = ShowGimple(name='show-gimple')
ps.register_after('cfg')
```

For more information, see Creating new optimization passes

4.4 Wiring up callbacks

The other way to write scripts is to register callback functions to be called when various events happen during compilation, such as using gcc.PLUGIN_PASS_EXECUTION to piggyback off of an existing GCC pass.

```python
# Show all the passes that get executed
import gcc

def my_pass_execution_callback(*args, **kwargs):
    (optpass, fun) = args
    print(args)

gcc.register_callback(gcc.PLUGIN_PASS_EXECUTION, my_pass_execution_callback)
```
For more information, see *Working with callbacks*
CHAPTER 5

Global data access

gcc.get_variables()
Get all variables in this compilation unit as a list of gcc.Variable
class gcc.Variable
Wrapper around GCC’s struct varpool_node, representing a variable in the code being compiled.
    decl
    The declaration of this variable, as a gcc.Tree
gccutils.get_variables_as_dict()
Get a dictionary of all variables, where the keys are the variable names (as strings), and the values are instances of gcc.Variable
gcc.maybe_get_identifier(str)
Get the gcc.IdentifierNode with this name, if it exists, otherwise None. (However, after the front-end has run, the identifier node may no longer point at anything useful to you; see gccutils.get_global_typedef() for an example of working around this)
gcc.get_translation_units()
Get a list of all gcc.TranslationUnitDecl for the compilation units within this invocation of GCC (that’s “source code files” for the layperson).
class TranslationUnitDecl
    Subclass of gcc.Tree representing a compilation unit
        block
        The gcc.Block representing global scope within this source file.
        language
        The source language of this translation unit, as a string (e.g. “GNU C”)
gcc.get_global_namespace()
C++ only: locate the gcc.NamespaceDecl for the global namespace (a.k.a. “::”)
gccutils.get_global_typedef(name)
Given a string name, look for a C/C++ typedef in global scope with that name, returning it as a gcc.TypeDecl, or None if it wasn’t found
**gccutils.get_global_vardecl_by_name***(name)***

Given a string `name`, look for a C/C++ variable in global scope with that name, returning it as a `gcc.VarDecl`, or None if it wasn’t found.

**gccutils.get_field_by_name**(decl, name)

Given one of a `gcc.RecordType`, `gcc.UnionType`, or `gcc.QualUnionType`, along with a string `name`, look for a field with that name within the given struct or union, returning it as a `gcc.FieldDecl`, or None if it wasn’t found.
To add a new compiler warning to GCC, it’s helpful to have a high-level understanding of how GCC works, so here's the 10,000 foot view of how GCC turns source code into machine code.

The short version is that GCC applies a series of optimization passes to your code, gradually converting it from a high-level representation into machine code, via several different internal representations.

Each programming language supported by GCC has a “frontend”, which parses the source files.

For the case of C and C++, the preprocessor manipulates the code first before the frontend sees it. You can see the preprocessor output with the `-E` option.

Exactly what happens in each frontend varies by language: some language frontends emit language-specific trees, and some convert to a language-independent tree representation known as GENERIC. In any case, we eventually reach a representation known as GIMPLE. The GIMPLE representation contains simplified operations, with temporary variables added as necessary to avoid nested sub-expressions.

For example, given this C code:

```c
int main(int argc, char **argv)
{
    int i;
    printf("argc: %i\n", argc);
    for (i = 0; i < argc; i++) {
        printf("argv[%i]: %s\n", argv[i]);
    }
    helper_function();
    return 0;
}
```
we can see a dump of a C-like representation of the GIMPLE form by passing `-fdump-tree-gimple` to the command-line:

```bash
$ gcc -fdump-tree-gimple test.c
$ cat test.c.004t.gimple
```

giving something like this:

```c
main (int argc, char ** argv)
{
    const char * restrict D.3258;
    long unsigned int D.3259;
    long unsigned int D.3260;
    char * * D.3261;
    char * D.3262;
    const char * restrict D.3263;
    int D.3264;
    int i;

    D.3258 = (const char * restrict) &'argc: %i
    printf (D.3258, argc);
    i = 0;
    goto <D.2050>;
  <D.2049>:
    D.3259 = (long unsigned int) i;
    D.3260 = D.3259 * 8;
    D.3261 = argv + D.3260;
    D.3262 = *D.3261;
    D.3263 = (const char * restrict) &'argv[%i]: %s
    printf (D.3263, D.3262);
    i = i + 1;
  <D.2050>:
    if (i < argc) goto <D.2049>; else goto <D.2051>;
  <D.2051>:
    helper_function ();
    D.3264 = 0;
  return D.3264;
}
```

It’s far easier to see the GIMPLE using:

```bash
./gcc-with-python examples/show-gimple.py test.c
```

which generates bitmaps showing the “control flow graph” of the functions in the file, with source on the left-hand side, and GIMPLE on the right-hand side:

Each function is divided into “basic blocks”. Each basic block consists of a straight-line sequence of code with a single entrypoint and exit: all branching happens between basic blocks, not within them. The basic blocks form a “control flow graph” of basic blocks, linked together by edges. Each block can contain a list of `gcc.Gimple` statements.

You can work with this representation from Python using `gcc.Cfg`

Once the code is in GIMPLE form, GCC then attempts a series of optimizations on it.

Some of these optimizations are listed here: [http://gcc.gnu.org/onlinedocs/gccint/Tree-SSA-passes.html](http://gcc.gnu.org/onlinedocs/gccint/Tree-SSA-passes.html)

If you’re looking to add new compiler warnings, it’s probably best to hook your code into these early passes.

The GIMPLE representation actually has several forms:
• an initial “high gimple” form, potentially containing certain high-level operations (e.g. control flow, exception handling)

• the lower level gimple forms, as each of these operations are rewritten in lower-level terms (turning control flow from jumps into a CFG etc)

• the SSA form of GIMPLE. In Static Single Assignment form, every variable is assigned to at most once, with additional versions of variables added to help track the impact of assignments on the data flowing through a function. See http://gcc.gnu.org/onlinedocs/gccint/SSA.html

You can tell what form a function is in by looking at the flags of the current pass. For example:

```python
if ps.properties_provided & gcc.PROP_cfg:
    # ...then this gcc.Function ought to have a gcc.Cfg:
    do_something_with_cfg(fn.cfg)

if ps.properties_provided & gcc.PROP_ssa:
    # ...then we have SSA data
    do_something_with_ssa(fn)
```

Here’s our example function, after conversion to GIMPLE SSA:

```bash
./gcc-with-python examples/show-ssa.py test.c
```

You can see that the local variable `i` has been split into three versions:

• `i_4`, assigned to in block 2

• `i_11`, assigned to at the end of block 3

• `i_1`, assigned to at the top of block 4.
As is normal with SSA, GCC inserts fake functions known as “PHI” at the start of basic blocks where needed in order to merge the multiple possible values of a variable. You can see one in our example at the top of the loop in block 4:

\[ i_1 = \text{PHI} <i_4(2), i_{11}(3)> \]

where \( i_1 \) either gets the value of \( i_4 \), or of \( i_{11} \), depending on whether we reach here via block 2 (at the start of the iteration) or block 3 (continuing the “for” loop).

After these optimizations passes are done, GCC converts the GIMPLE SSA representation into a lower-level representation known as Register Transfer Language (RTL). This is probably too low-level to be of interest to those seeking to add new compiler warnings: at this point it’s attempting to work with the available opcodes and registers on the target CPU with the aim of generating efficient machine code.


The RTL form uses the same Control Flow Graph machinery as the GIMPLE representation, but with RTL expressions within the basic blocks.

Once in RTL, GCC applies a series of further optimizations, before finally generating assembly language (which it submits to as, the GNU assembler): [http://gcc.gnu.org/onlinedocs/gccint/RTL-passes.html](http://gcc.gnu.org/onlinedocs/gccint/RTL-passes.html) You can see the assembly language using the \(-S\) command line option.

\`
$ ./gcc -S test.c
$ cat test.s
\`
There are various sample scripts located in the *examples* subdirectory. Once you’ve built the plugin (with *make*), you can run them via:

```
$ ./gcc-with-python examples/NAME-OF-SCRIPT.py test.c
```

### 7.1 *show-docs.py*

A trivial script to make it easy to read the builtin documentation for the gcc API:

```
$ ./gcc-with-python examples/show-docs.py test.c
```

with this source:

```python
import gcc
help(gcc)
```

giving output:

```
Help on built-in module gcc:

NAME
gcc

FILE
(built-in)

CLASSES
__builtin__.object
BasicBlock
Cfg
```

(continues on next page)
7.2 show-passes.py

You can see the passes being executed via:

```
$ ./gcc-with-python examples/show-passes.py test.c
```

This is a simple script that registers a trivial callback:

```python
# Sample python script, to be run by our gcc plugin
# Show all the passes that get executed
import gcc

def my_pass_execution_callback(*args, **kwargs):
    (optpass, fun) = args
    print(args)

gcc.register_callback(gcc.PLUGIN_PASS_EXECUTION,
                     my_pass_execution_callback)
```

Sample output, showing passes being called on two different functions (main and helper_function):

```
(gcc.GimplePass(name='*warn_unused_result'), gcc.Function('main'))
(gcc.GimplePass(name='omplower'), gcc.Function('main'))
(gcc.GimplePass(name='lower'), gcc.Function('main'))
(gcc.GimplePass(name='eh'), gcc.Function('main'))
(gcc.GimplePass(name='cfg'), gcc.Function('main'))
(gcc.GimplePass(name='*warn_function_return'), gcc.Function('main'))
(gcc.GimplePass(name='*build_cgraph_edges'), gcc.Function('main'))
(gcc.GimplePass(name='*warn_unused_result'), gcc.Function('helper_function'))
(gcc.GimplePass(name='omplower'), gcc.Function('helper_function'))
(gcc.GimplePass(name='lower'), gcc.Function('helper_function'))
(gcc.GimplePass(name='eh'), gcc.Function('helper_function'))
(gcc.GimplePass(name='cfg'), gcc.Function('helper_function'))
[...truncated...]
```

7.3 show-gimple.py

A simple script for viewing each function in the source file after it’s been converted to “GIMPLE” form, using GraphViz to visualize the control flow graph:

```
$ ./gcc-with-python examples/show-gimple.py test.c
```

It will generate a file test.png for each function, and opens it in an image viewer.

The Python code for this is:
# Show the GIMPLE form of each function, using GraphViz

```python
import gcc
from gccutils import get_src_for_loc, cfg_to_dot, invoke_dot

# We'll implement this as a custom pass, to be called directly after the
# builtin "cfg" pass, which generates the CFG:

class ShowGimple(gcc.GimplePass):
    def execute(self, fun):
        # (the CFG should be set up by this point, and the GIMPLE is not yet
        # in SSA form)
        if fun and fun.cfg:
            dot = cfg_to_dot(fun.cfg, fun.decl.name)
            print dot
            invoke_dot(dot, name=fun.decl.name)

ps = ShowGimple(name='show-gimple')
ps.register_after('cfg')
```

## 7.4 show-ssa.py

This is similar to `show-gimple.py`, but shows each function after the GIMPLE has been converted to Static Single Assignment form (“SSA”):

```
$ ./gcc-with-python examples/show-ssa.py test.c
```

As before, it generates an image file for each function and opens it in a viewer.
The Python code for this is:

```python
# Sample python script, to be run by our gcc plugin
# Show the SSA form of each function, using GraphViz
import gcc
from gccutils import get_src_for_loc, cfg_to_dot, invoke_dot

# A custom GCC pass, to be called directly after the builtin "ssa" pass;
# which
# generates the Static Single Assignment form of the GIMPLE within the CFG:
class ShowSsa(gcc.GimplePass):
    def execute(self, fun):
        # (the SSA form of each function should have just been set up)
        if fun and fun.cfg:
            dot = cfg_to_dot(fun.cfg, fun.decl.name)
            print(dot)
            invoke_dot(dot, name=fun.decl.name)

ps = ShowSsa(name='show-ssa')
ps.register_after('ssa')
```

### 7.5 show-callgraph.py

This simple script sends GCC’s interprocedural analysis data through GraphViz.
$ ./gcc-with-python examples/show-callgraph.py test.c

It generates an image file showing which functions call which other functions, and opens it in a viewer.

The Python code for this is:

```python
# Sample python script, to be run by our gcc plugin
# Show the call graph (interprocedural analysis), using GraphViz
import gcc
from gccutils import callgraph_to_dot, invoke_dot

# In theory we could have done this with a custom gcc.Pass registered
# directly after "*build_cgraph_edges". However, we can only register
# relative to passes of the same kind, and that pass is a
# gcc.GimplePass, which is called per-function, and we want a one-time
# pass instead.
# So we instead register a callback on the one-time pass that follows it

def on_pass_execution(p, fn):
    if p.name == '*free_lang_data':
        # The '*free_lang_data' pass is called once, rather than per-
        # function,
        # and occurs immediately after "*build_cgraph_edges", which is the
        # pass that initially builds the callgraph
        # So at this point we're likely to get a good view of the callgraph
        # before further optimization passes manipulate it
        dot = callgraph_to_dot()
        invoke_dot(dot)

gcc.register_callback(gcc.PLUGIN_PASS_EXECUTION,
                     on_pass_execution)
```

7.5. `show-callgraph.py`
CHAPTER 8

Working with C code

8.1 “Hello world”

Here’s a simple “hello world” C program:

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hello, python\n");
}
```

Here’s a Python script that locates the function at one pass within the compile and prints various interesting things about it:

```python
import gcc

def on_pass_execution(p, fn):
    # This pass is called fairly early on, per-function, after the
    # CFG has been built:
    if p.name == '*warn_function_return':
        # For this pass, "fn" will be an instance of gcc.Function:
        print('fn: %r' % fn)
        print('fn.decl.name: %r' % fn.decl.name)

        # fn.decl is an instance of gcc.FunctionDecl:
        print('return type: %r' % str(fn.decl.type.type))
        print('argument types: %r' % [str(t) for t in fn.decl.type.argument_types])

        assert isinstance(fn.cfg, gcc.Cfg)  # None for some early passes
```

(continues on next page)
assert len(fn.cfg.basic_blocks) == 3
assert fn.cfg.basic_blocks[0] == fn.cfg.entry
assert fn.cfg.basic_blocks[1] == fn.cfg.exit
bb = fn.cfg.basic_blocks[2]
for i, stmt in enumerate(bb.gimple):
    print('gimple[%i]:' % i)
    print(' str(stmt): %r' % str(stmt))
    print(' repr(stmt): %r' % repr(stmt))
    if isinstance(stmt, gcc.GimpleCall):
        from gccutils import pprint
        print(' type(stmt.fn): %r' % type(stmt.fn))
        print(' str(stmt.fn): %r' % str(stmt.fn))
        for i, arg in enumerate(stmt.args):
            print(' str(stmt.args[%i]): %s' % (i, str(stmt.args[i])))
        print(' str(stmt.lhs): %s' % str(stmt.lhs))

# Wire up our callback:
gcc.register_callback(gcc.PLUGIN_PASS_EXECUTION, on_pass_execution)

We can run the script during the compile like this:

    ./gcc-with-python script.py test.c

Here’s the expected output:

```
fn: gcc.Function('main')
fn.decl.name: 'main'
return type: 'int'
argument types: ['int', 'char * *']
gimple[0]:
    str(stmt): '__builtin_puts (&"Hello, python"[0]);'
    repr(stmt): 'gcc.GimpleCall()
    type(stmt.fn): <type 'gcc.AddrExpr'>
    str(stmt.fn): '__builtin_puts'
    str(stmt.args[0]): '"Hello, python"[0]
    str(stmt.lhs): None

   gimple[1]:
    str(stmt): 'return;
    repr(stmt): 'gcc.GimpleReturn()'
```

Notice how the call to `printf` has already been optimized into a call to `__builtin_puts`.

### 8.2 Spell-checking string constants within source code

This example add a spell-checker pass to GCC: all string constants are run through the “enchant” spelling-checker:

```
$ ./gcc-with-python tests/examples/spelling-checker/script.py input.c
```

The Python code for this is:
import gcc

# Use the Python bindings to the "enchant" spellchecker:
import enchant
spellingdict = enchant.Dict("en_US")

class SpellcheckingPass(gcc.GimplePass):
    def execute(self, fun):
        # This is called per-function during compilation:
        for bb in fun.cfg.basic_blocks:
            if bb.gimple:
                for stmt in bb.gimple:
                    stmt.walk_tree(self.spellcheck_node, stmt.loc)

    def spellcheck_node(self, node, loc):
        # Spellcheck any textual constants found within the node:
        if isinstance(node, gcc.StringCst):
            words = node.constant.split()
            for word in words:
                if not spellingdict.check(word):
                    # Warn about the spelling error (controlling the warning
                    # with the -Wall command-line option):
                    if gcc.warning(loc,
                        'Possibly misspelt word in string
                        constant: %r
                        ' % word,
                        gcc.Option('-Wall')):
                        # and, if the warning was not suppressed at the
                        command line, emit
                        # suggested respellings:
                        suggestions = spellingdict.suggest(word)
                        if suggestions:
                            gcc.inform(loc, 'Suggested respellings: %r
                            ' % ',
                            .join(suggestions))

ps = SpellcheckingPass(name='spellchecker')
ps.register_after('cfg')

Given this sample C source file:

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    const char *p = argc ? "correctly spelled" : "not so korectly speled";
    printf("The quick brown fox jumps over the lazy dog\n");
    printf("Ths s n xmpl f splng mstke\n");
}
```

these warnings are emitted on stderr:

```
tests/examples/spelling-checker/input.c: In function 'main':
tests/examples/spelling-checker/input.c:24:48: warning: Possibly misspelt word in string constant: 'korectly' [-Wall]
```
8.3 Finding global variables

This example adds a pass that warns about uses of global variables:

```bash
$ ./gcc-with-python \
  tests/examples/find-global-state/script.py \
  -c \
  tests/examples/find-global-state/input.c
```

The Python code for this is:

```python
import gcc
from gccutils import get_src_for_loc
DEBUG=0
def is_const(type_):
    if DEBUG:
        type_.debug()
    if hasattr(type_, 'const'):
        if type_.const:
            return True

    # Don't bother warning about an array of const e.g.
    # const char []
    if isinstance(type_, gccArrayType):
        item_type = type_.dereference
        if is_const(item_type):
            return True
```
class StateFinder:
  def __init__(self):
    # Locate all declarations of variables holding "global" state:
    self.global_decls = set()

    for var in gcc.get_variables():
      type_ = var.decl.type

      if DEBUG:
        print('var.decl: %r' % var.decl)
        print(type_)

      # Don't bother warning about const data:
      if is_const(type_):
        continue

      self.global_decls.add(var.decl)

      if DEBUG:
        print('self.global_decls: %r' % self.global_decls)

    self.state_users = set()

  def find_state_users(self, node, loc):
    if isinstance(node, gcc.VarDecl):
      if node in self.global_decls:
        # store the state users for later replay, so that
        # we can eliminate duplicates
        # e.g. two references to "q" in "q += p"
        # and replay in source-location order:
        self.state_users.add((loc, node))

  def flush(self):
    # Emit warnings, sorted by source location:
    for loc, node in sorted(self.state_users,
                             key=lambda pair:pair[0]):
      gcc.inform(loc,
                 'use of global state "$s $s" here'
                 % (node.type, node))

  def on_pass_execution(p, fn):
    if p.name == '*free_lang_data':
      sf = StateFinder()

      # Locate uses of such variables:
      for node in gcc.get_callgraph_nodes():
        fun = node.decl.function
        if fun:
          cfg = fun.cfg
          if cfg:
            for bb in cfg.basic_blocks:
              stmts = bb.gimple
              if stmts:
                for stmt in stmts:
                  stmt.walk_tree(sf.find_state_users,
                                  stmt.loc)
Given this sample C source file:

```c
#include <stdio.h>

static int a_global;

struct {
    int f;
} bar;

extern int foo;

int test(int j)
{
    /* A local variable, which should *not* be reported: */
    int i;
    i = j * 4;
    return i + 1;
}

int test2(int p)
{
    static int q = 0;
    q += p;
    return p * q;
}

int test3(int k)
{
    /* We should *not* report about __FUNCTION__ here: */
    printf("%s:%i:%s\n", __FILE__, __LINE__, __FUNCTION__);  
}

int test4()
{
    return foo;
}

int test6()
{
    return bar.f;
}

struct banana {
    int f;
};

const struct banana a_banana;
```

(continues on next page)
```c
int test7()
{
    return a_banana.f;
}
```

These warnings are emitted on stderr:

```
tests/examples/find-global-state/input.c:41:nn: note: use of global state
    → “int q” here
```
```
tests/examples/find-global-state/input.c:41:nn: note: use of global state
    → “int q” here
```
```
tests/examples/find-global-state/input.c:42:nn: note: use of global state
    → “int q” here
```
```
tests/examples/find-global-state/input.c:53:nn: note: use of global state
    → “int foo” here
```
```
tests/examples/find-global-state/input.c:58:nn: note: use of global state
    → “struct
    {
        int f;
    } bar” here
```
Locations

`gccutils.get_src_for_loc(loc)`  
Given a `gcc.Location`, get the source line as a string (without trailing whitespace or newlines)

```python
class gcc.Location

Wrapper around GCC's `location_t`, representing a location within the source code. Use `gccutils.get_src_for_loc()` to get at the line of actual source code.

The output from `__repr__` looks like this:

```
gcc.Location(file='./src/test.c', line=42)
```

The output from `__str__` looks like this:

```
./src/test.c:42
```

- **file**  
  (string) Name of the source file (or header file)

- **line**  
  (int) Line number within source file (starting at 1, not 0)

- **column**  
  (int) Column number within source file (starting at 1, not 0)

- **in_system_header**  
  (bool) This attribute flags locations that are within a system header file. It may be of use when writing custom warnings, so that you can filter out issues in system headers, leaving just those within the user's code:

  ```python
  # Don't report on issues found in system headers:
  if decl.location.in_system_header:
      return
  ```

- **offset_column** *(self, offset)*  
  Generate a new `gcc.Location` based on the caret location of this location, offsetting the column by the given amount.
From GCC 6 onwards, these values can represent both a caret and a range, e.g.:

```python
a = (foo && bar)
    ~~~~~^~~~~~~
```

```python
__init__(self, caret, start, finish)

Construct a location, using the caret location of caret as the caret, and the start/finish of start and finish respectively:

```python
compound_loc = gcc.Location(caret, start, finish)
```

caret

`(gcc.Location)` The caret location within this location. In the above example, the caret is on the first ‘&’ character.

start

`(gcc.Location)` The start location of this range. In the above example, the start is on the opening parenthesis.

finish

`(gcc.Location)` The finish location of this range. In the above example, the finish is on the closing parenthesis.

class gcc.RichLocation

Wrapper around GCC’s `rich_location`, representing one or more locations within the source code, and zero or more fix-it hints.

Note: gcc.RichLocation is only available from GCC 6 onwards

```python
add_fixit_replace(self, new_content)

Add a fix-it hint, suggesting replacement of the content covered by range 0 of the rich location with new_content.
Generating custom errors and warnings

`gcc.warning(location, message, option=None)`

Emits a compiler warning at the given `gcc.Location`, potentially controlled by a `gcc.Option`.

If no option is supplied (or `None` is supplied), then the warning is an unconditional one, always issued:

```python
gcc.warning(func.start, 'this is an unconditional warning')
```

```
$ ./gcc-with-python script.py input.c
input.c:25:1: warning: this is an unconditional warning [enabled by default]
```

and will be an error if `-Werror` is supplied as a command-line argument to GCC:

```
$ ./gcc-with-python script.py -Werror input.c
input.c:25:1: error: this is an unconditional warning [-Werror]
```

It’s possible to associate the warning with a command-line option, so that it is controlled by that option.

For example, given this Python code:

```python
gcc.warning(func.start, 'Incorrect formatting', gcc.Option('-Wformat'))
```

if the given warning is enabled, a warning will be printed to stderr:

```
$ ./gcc-with-python script.py input.c
input.c:25:1: warning: incorrect formatting [-Wformat]
```

If the given warning is being treated as an error (through the usage of `-Werror`), then an error will be printed:

```
$ ./gcc-with-python script.py -Werror input.c
input.c:25:1: error: incorrect formatting [-Werror=format]
ccl: all warnings being treated as errors
```

```
$ ./gcc-with-python script.py -Werror=format input.c
input.c:25:1: error: incorrect formatting [-Werror=format]
ccl: some warnings being treated as errors
```
If the given warning is disabled, the warning will not be printed:

```
$ ./gcc-with-python script.py -Wno-format input.c
```

**Note:** Due to the way GCC implements some options, it's not always possible for the plugin to fully disable some warnings. See `gcc.Option.is_enabled` for more information.

The function returns a boolean, indicating whether or not anything was actually printed.

```python
gcc.error(location, message)
```

Emits a compiler error at the given `gcc.Location`.

For example:

```
gcc.error(func.start, 'something bad was detected')
```

would lead to this error being printed to stderr:

```
$ ./gcc-with-python script.py input.c
input.c:25:1: error: something bad was detected
```

```python
gcc.permerror(loc, str)
```

This is a wrapper around GCC’s `permerror` function.

Expects an instance of `gcc.Location` (not None) and a string

Emit a “permissive” error at that location, intended for things that really ought to be errors, but might be present in legacy code.

In theory it’s suppressable using “-fpermissive” at the GCC command line (which turns it into a warning), but this only seems to be legal for C++ source files.

Returns True if the warning was actually printed, False otherwise

```python
gcc.inform(location, message)
```

This is a wrapper around GCC’s `inform` function.

Expects an instance of `gcc.Location` or `gcc.RichLocation`, (not None) and a string

Emit an informational message at that location.

For example:

```
gcc.inform(stmt.loc, 'this is where X was defined')
```

would lead to this informational message being printed to stderr:

```
$ ./gcc-with-python script.py input.c
input.c:23:3: note: this is where X was defined
```
Many of the plugin events are called for each function within the source code being compiled. Each time, the plugin passes a `gcc.Function` instance as a parameter to your callback, so that you can work on it.

You can get at the control flow graph of a `gcc.Function` via its `cfg` attribute. This is an instance of `gcc.Cfg`.

```python
class gcc.Function
    Wrapper around one of GCC's `struct function *`
    
    cfg
        An instance of `gcc.Cfg` for this function (or None during early passes)
    
    decl
        The declaration of this function, as a `gcc.FunctionDecl`
    
    local_decls
        List of `gcc.VarDecl` for the function’s local variables. It does not contain arguments; for those see the `arguments` property of the function’s `decl`.

        Note that for locals with initializers, `initial` only seems to get set on those `local_decls` that are static variables. For other locals, it appears that you have to go into the gimple representation to locate assignments.
    
    start
        The `gcc.Location` of the beginning of the function
    
    end
        The `gcc.Location` of the end of the function
    
    funcdef_no
        Integer: a sequence number for profiling, debugging, etc.
```

```python
class gcc.Cfg
    A `gcc.Cfg` is a wrapper around GCC’s `struct control_flow_graph`.
    
    basic_blocks
        List of `gcc.BasicBlock`, giving all of the basic blocks within this CFG
    
    entry
        Instance of `gcc.BasicBlock`: the entrypoint for this CFG
```
exit

Instance of `gcc.BasicBlock`: the final one within this CFG

**get_block_for_label** *(labeldecl)*

Given a `gcc.LabelDecl`, get the corresponding `gcc.BasicBlock`

You can use `gccutils.cfg_to_dot` to render a gcc.Cfg as a graphviz diagram. It will render the diagram, showing each basic block, with source code on the left-hand side, interleaved with the “gimple” representation on the right-hand side. Each block is labelled with its index, and edges are labelled with appropriate flags.

For example, given this sample C code:

```c
int main(int argc, char **argv)
{
  int i;
  printf("argc: %i\n", argc);
  for (i = 0; i < argc; i++) {
    printf("argv[%i]: %s\n", argv[i]);
  }
  helper_function();
  return 0;
}
```

then the following Python code:

```python
dot = gccutils.cfg_to_dot(fun.cfg)
gccutils.invoke_dot(dot)
```

will render a CFG bitmap like this:

**class gcc.BasicBlock**

A `gcc.BasicBlock` is a wrapper around GCC’s `basic_block` type.

- **index**
  The index of the block (an int), as seen in the `cfg_to_dot` rendering.

- **preds**
  The list of predecessor `gcc.Edge` instances leading into this block

- **succs**
  The list of successor `gcc.Edge` instances leading out of this block

- **phi_nodes**
  The list of `gcc.GimplePhi` phoney functions at the top of this block, if appropriate for this pass, or None

- **gimple**
  The list of `gcc.Gimple` instructions, if appropriate for this pass, or None

- **rtl**
  The list of `gcc.Rtl` expressions, if appropriate for this pass, or None

**class gcc.Edge**

A wrapper around GCC’s `edge` type.
**src**

The source `gcc.BasicBlock` of this edge

**dest**

The destination `gcc.BasicBlock` of this edge

**true_value**

Boolean: *True* if this edge is taken when a `gcc.GimpleCond` conditional is true, *False* otherwise

**false_value**

Boolean: *True* if this edge is taken when a `gcc.GimpleCond` conditional is false, *False* otherwise

**complex**

Boolean: *True* if this edge is “special” e.g. due to exception-handling, or some other kind of “strange” control flow transfer, *False* otherwise
The various language front-ends for GCC emit “tree” structures (which I believe are actually graphs), used throughout the rest of the internal representation of the code passing through GCC.

```python
class gcc.Tree
    A gcc.Tree is a wrapper around GCC’s tree type

dump()
    Dump the tree to stderr, using GCC’s own diagnostic routines

type
    Instance of gcc.Tree giving the type of the node

addr
    (long) The address of the underlying GCC object in memory
```

The `__str__` method is implemented using GCC’s own pretty-printer for trees, so e.g.: 

```python
str(t)
```

might return:

```
'int <T531> (int, char * *')
```

for a `FunctionDecl`

```python
str_no_uid
```

A string representation of this object, like `str()`, but without including any internal UIDs.

This is intended for use in selftests that compare output against some expected value, to avoid embedding values that change into the expected output.

For example, given the type declaration above, where `str(t)` might return:

```
'int <T531> (int, char * *')
```

where the UID “531” is liable to change from compile to compile, whereas `t.str_no_uid` has value:
which won’t arbitrarily change each time.

There are numerous subclasses of \texttt{gcc.Tree}, some with numerous subclasses of their own. Some important parts of the class hierarchy include:

<table>
<thead>
<tr>
<th>Subclass</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{gcc.Binary}</td>
<td>A binary arithmetic expression, with numerous subclasses</td>
</tr>
<tr>
<td>\texttt{gcc.Block}</td>
<td>A symbol-binding block</td>
</tr>
<tr>
<td>\texttt{gcc.Comparison}</td>
<td>A relational operators (with various subclasses)</td>
</tr>
<tr>
<td>\texttt{gcc.Constant}</td>
<td>Subclasses for constants</td>
</tr>
<tr>
<td>\texttt{gcc.Constructor}</td>
<td>An aggregate value (e.g. in C, a structure or array initializer)</td>
</tr>
<tr>
<td>\texttt{gcc.Declaration}</td>
<td>Subclasses relating to declarations (variables, functions, etc)</td>
</tr>
<tr>
<td>\texttt{gcc.Expression}</td>
<td>Subclasses relating to expressions</td>
</tr>
<tr>
<td>\texttt{gcc.IdentifierNode}</td>
<td>A name</td>
</tr>
<tr>
<td>\texttt{gcc.Reference}</td>
<td>Subclasses for relating to reference to storage (e.g. pointer values)</td>
</tr>
<tr>
<td>\texttt{gcc.SsaName}</td>
<td>A variable reference for SSA analysis</td>
</tr>
<tr>
<td>\texttt{gcc.Statement}</td>
<td>Subclasses for statement expressions, which have side-effects</td>
</tr>
<tr>
<td>\texttt{gcc.Type}</td>
<td>Subclasses for describing the types of variables</td>
</tr>
<tr>
<td>\texttt{gcc.Unary}</td>
<td>Subclasses for unary arithmetic expressions</td>
</tr>
</tbody>
</table>

\textbf{Note:} Each subclass of \texttt{gcc.Tree} is typically named after either one of the \texttt{enum tree_code_class} or \texttt{enum tree_code} values, with the names converted to Camel Case:

For example a \texttt{gcc.Binary} is a wrapper around a \texttt{tree} of type \texttt{tcc_binary}, and a \texttt{gcc.PlusExpr} is a wrapper around a \texttt{tree} of type \texttt{PLUS_EXPR}.

As of this writing, only a small subset of the various fields of the different subclasses have been wrapped yet, but it’s generally easy to add new ones. To add new fields, I’ve found it easiest to look at \texttt{gcc/tree.h} and \texttt{gcc/print-tree.c} within the GCC source tree and use the \texttt{print_node} function to figure out what the valid fields are. With that information, you should then look at \texttt{generate-tree-c.py}, which is the code that generates the Python wrapper classes (it’s used when building the plugin to create \texttt{autogenerated-tree.c}). Ideally when exposing a field to Python you should also add it to the API documentation, and add a test case.

\texttt{gccutils.pformat(tree)}

This function attempts to generate a debug dump of a \texttt{gcc.Tree} and all of its “interesting” attributes, recursively. It’s loosely modelled on Python’s \texttt{pprint} module and GCC’s own \texttt{debug_tree} diagnostic routine using indentation to try to show the structure.

It returns a string.

It differs from \texttt{gcc.Tree.debug()} in that it shows the Python wrapper objects, rather than the underlying GCC data structures themselves. For example, it can’t show attributes that haven’t been wrapped yet.

Objects that have already been reported within this call are abbreviated to “…” to try to keep the output readable.

Example output:

```python
<FunctionDecl
def repr():
def superclasses = (<type 'gcc.Declaration'>, <type 'gcc.Tree'>)
def function = gcc.Function('main')
def location = /home/david/coding/gcc-python/test.c:15
```
.name = 'main'
.type = <FunctionType

repr() = <gcc.FunctionType object at 0x2f62a60>
str() = 'int <T531> (int, char * *)'
superclasses = ({type 'gcc.Type'}, {type 'gcc.Tree'})
.name = None
.type = <IntegerType

repr() = <gcc.IntegerType object at 0x2f629d0>
str() = 'int'
superclasses = ({type 'gcc.Type'}, {type 'gcc.Tree'})
.const = False
.name = <TypeDecl

repr() = gcc.TypeDecl('int')
superclasses = ({type 'gcc.Declaration'}, {type 'gcc.Tree'})
.location = None
.name = 'int'
.pointer = <PointerType

repr() = <gcc.PointerType object at 0x2f62b80>
str() = ' *'
superclasses = ({type 'gcc.Type'}, {type 'gcc.Tree'})
.dereference = ... ({type 'gcc.TypeDecl('int')})

repr() = gcc.TypeDecl('int')
.type = ... ({type 'gcc.IntegerType object at 0x2f629d0'})

precision = 32
.restrict = False
.type = None
.unsigned = False
.volatile = False

> gccutils.pprint(tree)

Similar to gccutils.pformat(), but prints the output to stdout.

(should this be stderr instead? probably should take a stream as an arg, but what should the default be?)

12.1 Blocks

class gcc.Block

A symbol binding block, such as the global symbols within a compilation unit.

vars

The list of gcc.Tree for the declarations and labels in this block
12.2 Declarations

class gcc.Declaration
A subclass of gcc.Tree indicating a declaration

Corresponds to the tcc_declaration value of enum tree_code_class within GCC’s own C sources.

name
(string) the name of this declaration

location
The gcc.Location for this declaration

is_artificial
(bool) Is this declaration a compiler-generated entity, rather than one provided by the user?

An example of such an “artificial” declaration occurs within the arguments of C++ methods: the initial this argument is a compiler-generated gcc.ParmDecl.

is_builtin
(boo) Is this declaration a compiler-built-in?

class gcc.FieldDecl
A subclass of gcc.Declaration indicating the declaration of a field within a structure.

name
(string) The name of this field

class gcc.FunctionDecl
A subclass of gcc.Declaration indicating the declaration of a function. Internally, this wraps a (struct tree_function_decl *)

function
The gcc.Function for this declaration

arguments
List of gcc.ParmDecl representing the arguments of this function

result
The gcc.ResultDecl representing the return value of this function

fullname

Note: This attribute is only usable with C++ code. Attempting to use it from another language will lead to a RuntimeError exception.

(string) The “full name” of this function, including the scope, return type and default arguments.

For example, given this code:

```c++
namespace Example {
    struct Coord {
        int x;
        int y;
    };

    class Widget {
        public:
```
```c
void set_location(const struct Coord& coord);
```

`set_location`'s fullname is:

```quote
'verd Example::Widget::set_location(const Example::Coord &)'
```

**callgraph_node**

The `gcc.CallgraphNode` for this function declaration, or None

**is_public**

(BOOL) For C++: is this declaration “public”

**is_private**

(BOOL) For C++: is this declaration “private”

**is_protected**

(BOOL) For C++: is this declaration “protected”

**is_static**

(BOOL) For C++: is this declaration “static”

**class gcc.ParmDecl**

A subclass of `gcc.Declaration` indicating the declaration of a parameter to a function or method.

**class gcc.ResultDecl**

A subclass of `gcc.Declaration` declaring a dummy variable that will hold the return value from a function.

**class gcc.VarDecl**

A subclass of `gcc.Declaration` indicating the declaration of a variable (e.g. a global or a local).

**initial**

The initial value for this variable as a `gcc.Constructor`, or None

**static**

(boolean) Is this variable to be allocated with static storage?

**class gcc.NamespaceDecl**

A subclass of `gcc.Declaration` representing a C++ namespace

**alias_of**

The `gcc.NamespaceDecl` which this namespace is an alias of or None if this namespace is not an alias.

**declarations**

**Note:** This attribute is only usable with non-alias namespaces. Accessing it on an alias will lead to a `RuntimeError` exception.

List of `gcc.Declaration` objects in this namespace. This attribute is only valid for non-alias namespaces.

### 12.2. Declarations
Note: This attribute is only usable with non-alias namespaces. Accessing it on an alias will lead to a RuntimeError exception.

List of `gcc.NamespaceDecl` objects nested in this namespace. This attribute is only valid for non-aliases

`lookup(name)`
Locate the given name within the namespace, returning a `gcc.Tree` or `None`

`unalias()`
Always returns a `gcc.NamespaceDecl` object which is not an alias. Returns self if this namespace is not an alias.

12.3 Types

class `gcc.Type`
A subclass of `gcc.Tree` indicating a type

Corresponds to the `tcc_type` value of `enum tree_code_class` within GCC’s own C sources.

`name` The `gcc.IdentifierNode` for the name of the type, or `None`.

`pointer` The `gccPointerType` representing the `(this_type *)` type

`attributes` The user-defined attributes on this type (using GCC’s `__attribute` syntax), as a dictionary (mapping from attribute names to list of values). Typically this will be the empty dictionary.

`sizeof()` this type, as an `int`, or raising `TypeError` for those types which don’t have a well-defined size

Note: This attribute is not usable from within `lto1`; attempting to use it there will lead to a `RuntimeError` exception.

Additional attributes for various `gcc.Type` subclasses:

`const`
(Boostrap) Does this type have the `const` modifier?

`constEquivalent`
The `gcc.Type` for the `const` version of this type

`volatile`
(Boostrap) Does this type have the `volatile` modifier?

`volatileEquivalent`
The `gcc.Type` for the `volatile` version of this type

`restrict`
(Boostrap) Does this type have the `restrict` modifier?

`restrictEquivalent`
The `gcc.Type` for the `restrict` version of this type
unqualified_equivalent

The \texttt{gcc.Type} for the version of this type that does not have any qualifiers.

The standard C types are accessible via class methods of \texttt{gcc.Type}. They are only created by GCC after plugins are loaded, and so they’re only visible during callbacks, not during the initial run of the code. (yes, having them as class methods is slightly clumsy).

Each of the following returns a \texttt{gcc.Type} instance representing the given type (or None at startup before any passes, when the types don’t yet exist)

<table>
<thead>
<tr>
<th>Class method</th>
<th>C Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc.Type.void()</td>
<td>void</td>
</tr>
<tr>
<td>gcc.Type.size_t()</td>
<td>size_t</td>
</tr>
<tr>
<td>gcc.Type.char()</td>
<td>char</td>
</tr>
<tr>
<td>gcc.Type.signed_char()</td>
<td>signed char</td>
</tr>
<tr>
<td>gcc.Type.unsigned_char()</td>
<td>unsigned char</td>
</tr>
<tr>
<td>gcc.Type.double()</td>
<td>double</td>
</tr>
<tr>
<td>gcc.Type.float()</td>
<td>float</td>
</tr>
<tr>
<td>gcc.Type.short()</td>
<td>short</td>
</tr>
<tr>
<td>gcc.Type.unsigned_short()</td>
<td>unsigned short</td>
</tr>
<tr>
<td>gcc.Type.int()</td>
<td>int</td>
</tr>
<tr>
<td>gcc.Type.unsigned_int()</td>
<td>unsigned int</td>
</tr>
<tr>
<td>gcc.Type.long()</td>
<td>long</td>
</tr>
<tr>
<td>gcc.Type.unsigned_long()</td>
<td>unsigned long</td>
</tr>
<tr>
<td>gcc.Type.long_double()</td>
<td>long double</td>
</tr>
<tr>
<td>gcc.Type.long_long()</td>
<td>long long</td>
</tr>
<tr>
<td>gcc.Type.unsigned_long_long()</td>
<td>unsigned long long</td>
</tr>
<tr>
<td>gcc.Type.int128()</td>
<td>int128</td>
</tr>
<tr>
<td>gcc.Type.unsigned_int128()</td>
<td>unsigned int128</td>
</tr>
<tr>
<td>gcc.Type.uint32()</td>
<td>uint32</td>
</tr>
<tr>
<td>gcc.Type.uint64()</td>
<td>uint64</td>
</tr>
</tbody>
</table>

\texttt{class gcc.IntegerType}

Subclass of \texttt{gcc.Type}, adding a few properties:

\texttt{unsigned}

(Boolean) True for ‘unsigned’, False for ‘signed’

\texttt{precision}

(int) The precision of this type in bits, as an int (e.g. 32)

\texttt{signed_equivalent}

The gcc.IntegerType for the signed version of this type

\texttt{Note:} This attribute is not usable from within \texttt{lto1}; attempting to use it there will lead to a \texttt{RuntimeError} exception.

\texttt{unsigned_equivalent}

The gcc.IntegerType for the unsigned version of this type

\texttt{Note:} This attribute is not usable from within \texttt{lto1}; attempting to use it there will lead to a \texttt{RuntimeError} exception.
**max_value**
The maximum possible value for this type, as a `gcc.IntegerCst`

**min_value**
The minimum possible value for this type, as a `gcc.IntegerCst`

class **gcc.FloatType**
Subclass of `gcc.Type` representing C's `float` and `double` types

**precision**
(int) The precision of this type in bits (32 for `float`; 64 for `double`)

class **gcc.PointerType**
Subclass of `gcc.Type` representing a pointer type, such as an `int *`

**dereference**
The `gcc.Type` that this type points to. In the above example (`int *`), this would be the `int` type.

class **gcc.EnumerateType**
Subclass of `gcc.Type` representing an enumeral type.

**values**
A list of tuple representing the constants defined in this enumeration. Each tuple consists of two elements; the first being the name of the constant, a `gcc.IdentifierNode`; and the second being the value, a `gcc.Constant`.

class **gccArrayType**
Subclass of `gcc.Type` representing an array type. For example, in a C declaration such as:

```c
char buf[16]
```

we have a `gcc.VarDecl` for `buf`, and its type is an instance of `gccArrayType`, representing `char[16]`.

**dereference**
The `gcc.Type` that this type points to. In the above example, this would be the `char` type.

**range**
The `gcc.Type` that represents the range of the array’s indices. If the array has a known range, then this will ordinarily be an `gcc.IntegerType` whose `min_value` and `max_value` are the (inclusive) bounds of the array. If the array does not have a known range, then this attribute will be `None`.

That is, in the example above, `range.min_val` is 0, and `range.max_val` is 15.

But, for a C declaration like:

```c
extern char array[];
```

the type’s `range` would be `None`.

class **gcc.VectorType**

**dereference**
The `gcc.Type` that this type points to

class **gcc.FunctionType**
Subclass of `gcc.Type` representing the type of a given function (or or a typedef to a function type, e.g. for callbacks).

See also `gcc.FunctionType`
The `type` attribute holds the return type.
**is_variadic**
True if this type represents a variadic function. Note that for a variadic function, the final ... argument is not explicitly represented in *argument_types*.

**argument_types**
A tuple of *gcc.Type* instances, representing the function’s argument types

**gccutils.getnonnull_arguments**(funtype)
This is a utility function for working with the “nonnull” custom attribute on function types:

http://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html

Return a *frozenset* of 0-based integers, giving the arguments for which we can assume “nonnull-ness”, handling the various cases of:
- the attribute isn’t present (returning the empty frozenset)
- the attribute is present, without args (all pointer args are non-NULL)
- the attribute is present, with a list of 1-based argument indices (Note that the result is still 0-based)

**class gcc.MethodType**
Subclass of *gcc.Type* representing the type of a given method. Similar to *gcc.FunctionType*

The *type* attribute holds the return type.

**argument_types**
A tuple of *gcc.Type* instances, representing the function’s argument types

**class gcc.RecordType**
A compound type, such as a C *struct*

**fields**
The fields of this type, as a list of *gcc.FieldDecl* instances

**methods**
The methods of this type, as a list of *gcc.MethodType* instances

You can look up C structures by looking within the top-level *gcc.Block* within the current translation unit. For example, given this sample C code:

```c
/* Example of a struct: */
struct test_struct {
    int a;
    char b;
    float c;
};

void foo()
{
}
```

then the following Python code:

```python
import gcc

class TestPass(gcc.GimplePass):
    def execute(self, fn):
        print('fn: %r' % fn)
        for u in gcc.get_translation_units():
```

(continues on next page)
for decl in u.block.vars:
    if isinstance(decl, gcc.TypeDecl):
        # "decl" is a gcc.TypeDecl
        # "decl.type" is a gcc.RecordType:
        print(' type(decl): %s' % type(decl))
        print(' type(decl.type): %s' % type(decl.type))
        print(' decl.type.name: %r' % decl.type.name)
        for f in decl.type.fields:
            print(' type(f): %s' % type(f))
            print(' f.name: %r' % f.name)
            print(' f.type: %s' % f.type)
            print(' type(f.type): %s' % type(f.type))

test_pass = TestPass(name='test-pass')

fn: gcc.Function('foo')

type(decl): <type 'gcc.TypeDecl'>
type(decl.type): <type 'gcc.RecordType'>
decl.type.name: gcc.IdentifierNode(name='test_struct')
type(f): <type 'gcc.FieldDecl'>
f.name: 'a'
f.type: int
type(f.type): <type 'gcc.IntegerType'>
type(f): <type 'gcc.FieldDecl'>
f.name: 'b'
f.type: char
type(f.type): <type 'gcc.IntegerType'>
type(f): <type 'gcc.FieldDecl'>
f.name: 'c'
f.type: float
type(f.type): <type 'gcc.RealType'>

12.4 Constants

class gcc.Constant
    Subclass of gcc.Tree indicating a constant value.

    Corresponds to the tcc_constant value of enum tree_code_class within GCC’s own C sources.

    constant
        The actual value of this constant, as the appropriate Python type:
Subclass | Python type
--- | ---
class ComplexCst |  

class FixedCst |  

class IntegerCst | int or long  

class PtrmemCst |  

class RealCst | float  

class StringCst | str  

class VectorCst |  

### 12.5 Binary Expressions

class gcc.Binary

Subclass of gcc.Tree indicating a binary expression.

Corresponds to the tcc_binary value of enum tree_code_class within GCC’s own C sources.

**location**

The gcc.Location for this binary expression

```
classmethod get_symbol()
```

Get the symbol used in debug dumps for this gcc.Binary subclass, if any, as a str. A table showing these strings can be seen here.

Has subclasses for the various kinds of binary expression. These include:

Simple arithmetic:

<table>
<thead>
<tr>
<th>Subclass</th>
<th>C/C++ operators</th>
<th>enum tree_code</th>
</tr>
</thead>
<tbody>
<tr>
<td>class gcc.PlusExpr</td>
<td>+</td>
<td>PLUS_EXPR</td>
</tr>
<tr>
<td>class gcc.MinusExpr</td>
<td>-</td>
<td>MINUS_EXPR</td>
</tr>
<tr>
<td>class gcc.MultExpr</td>
<td>*</td>
<td>MULT_EXPR</td>
</tr>
</tbody>
</table>

Pointer addition:
Various division operations:

<table>
<thead>
<tr>
<th>Subclass</th>
<th>C/C++ operators</th>
<th>enum tree_code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>class gcc.PointerPlusExpr</strong></td>
<td></td>
<td><strong>POINTER_PLUS_EXPR</strong></td>
</tr>
</tbody>
</table>

The remainder counterparts of the above division operators:

<table>
<thead>
<tr>
<th>Subclass</th>
<th>C/C++ operators</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>class gcc.TruncModExpr</strong></td>
<td></td>
</tr>
<tr>
<td><strong>class gcc.CeilModExpr</strong></td>
<td></td>
</tr>
<tr>
<td><strong>class gcc.FloorModExpr</strong></td>
<td></td>
</tr>
<tr>
<td><strong>class gcc.RoundModExpr</strong></td>
<td></td>
</tr>
</tbody>
</table>

Division for reals:

<table>
<thead>
<tr>
<th>Subclass</th>
<th>C/C++ operators</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>class gcc.RdivExpr</strong></td>
<td></td>
</tr>
</tbody>
</table>

Division that does not need rounding (e.g. for pointer subtraction in C):

<table>
<thead>
<tr>
<th>Subclass</th>
<th>C/C++ operators</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>class gcc.ExactDivExpr</strong></td>
<td></td>
</tr>
</tbody>
</table>

Max and min:
### Shift and rotate operations:

<table>
<thead>
<tr>
<th>Subclass</th>
<th>C/C++ operators</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class gcc.MaxExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.MinExpr</code></td>
<td></td>
</tr>
</tbody>
</table>

### Bitwise binary expressions:

<table>
<thead>
<tr>
<th>Subclass</th>
<th>C/C++ operators</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class gcc.LrotateExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.LshiftExpr</code></td>
<td><code>&lt;&lt;, &lt;&lt;=</code></td>
</tr>
<tr>
<td><code>class gcc.RrotateExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.RshiftExpr</code></td>
<td><code>&gt;&gt;, &gt;&gt;=</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subclass</th>
<th>C/C++ operators</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class gcc.BitAndExpr</code></td>
<td><code>&amp;</code>, <code>&amp;=</code> (bitwise “and”)</td>
</tr>
<tr>
<td><code>class gcc.BitIorExpr</code></td>
<td>`</td>
</tr>
<tr>
<td><code>class gcc.BitXorExpr</code></td>
<td><code>^</code>, <code>^=</code> (bitwise “xor”)</td>
</tr>
</tbody>
</table>

### Other gcc.Binary subclasses:

12.5. Binary Expressions 53
<table>
<thead>
<tr>
<th>Subclass</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>class gcc.CompareExpr</td>
<td></td>
</tr>
<tr>
<td>class gcc.CompareGExpr</td>
<td></td>
</tr>
<tr>
<td>class gcc.CompareLEexpr</td>
<td></td>
</tr>
<tr>
<td>class gcc.ComplexExpr</td>
<td></td>
</tr>
<tr>
<td>class gcc.MinusNomodExpr</td>
<td></td>
</tr>
<tr>
<td>class gcc.PlusNomodExpr</td>
<td></td>
</tr>
<tr>
<td>class gcc.RangeExpr</td>
<td></td>
</tr>
<tr>
<td>class gcc.UrshiftExpr</td>
<td></td>
</tr>
<tr>
<td>class gcc.VecExtractEvenExpr</td>
<td></td>
</tr>
<tr>
<td>class gcc.VecExtractOddExpr</td>
<td></td>
</tr>
<tr>
<td>class gcc.VecInterleaveHighExpr</td>
<td></td>
</tr>
<tr>
<td>class gcc.VecInterleavelowExpr</td>
<td></td>
</tr>
<tr>
<td>class gcc.VecLshiftExpr</td>
<td></td>
</tr>
<tr>
<td>class gcc.VecPackFixTruncExpr</td>
<td></td>
</tr>
<tr>
<td>class gcc.VecPackSatExpr</td>
<td></td>
</tr>
<tr>
<td>class gcc.VecPackTruncExpr</td>
<td></td>
</tr>
<tr>
<td>class gcc.VecRshiftExpr</td>
<td></td>
</tr>
<tr>
<td>class gcc.WidenMultExpr</td>
<td></td>
</tr>
<tr>
<td>class gcc.WidenMultHiExpr</td>
<td></td>
</tr>
<tr>
<td>class gcc.WidenMultLoExpr</td>
<td></td>
</tr>
<tr>
<td>class gcc.WidenSumExpr</td>
<td></td>
</tr>
</tbody>
</table>
12.6 Unary Expressions

class gcc.Unary

Subclass of gcc.Tree indicating a unary expression (i.e. taking a single argument).

Corresponds to the tcc_unary value of enum tree_code_class within GCC’s own C sources.

operand

The operand of this operator, as a gcc.Tree.

location

The gcc.Location for this unary expression

classmethod get_symbol()

Get the symbol used in debug dumps for this gcc.Unary subclass, if any, as a str. A table showing these strings can be seen here.

Subclasses include:
<table>
<thead>
<tr>
<th>Subclass</th>
<th>Meaning; C/C++ operators</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class gcc.AbsExpr</code></td>
<td>Absolute value</td>
</tr>
<tr>
<td><code>class gcc.AddrSpaceConvertExpr</code></td>
<td>Conversion of pointers between address spaces</td>
</tr>
<tr>
<td><code>class gcc.BitNotExpr</code></td>
<td>~ (bitwise “not”)</td>
</tr>
<tr>
<td><code>class gcc.CastExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.ConjExpr</code></td>
<td>For complex types: complex conjugate</td>
</tr>
<tr>
<td><code>class gcc.ConstCastExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.ConvertExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.DynamicCastExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.FixTruncExpr</code></td>
<td>Convert real to fixed-point, via truncation</td>
</tr>
<tr>
<td><code>class gcc.FixedConvertExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.FloatExpr</code></td>
<td>Convert integer to real</td>
</tr>
<tr>
<td><code>class gcc.NegateExpr</code></td>
<td>Unary negation</td>
</tr>
<tr>
<td><code>class gcc.NoexceptExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.NonLvalueExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.NopExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.ParenExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.ReducMaxExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.ReducMinExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.ReducPlusExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.ReinterpretCastExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.StaticCastExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.UnaryPlusExpr</code></td>
<td></td>
</tr>
</tbody>
</table>
12.7 Comparisons

class gcc.Comparison
   Subclass of gcc.Tree for comparison expressions
   Corresponds to the tcc_comparison value of enum tree_code_class within GCC’s own C sources.

location
   The gcc.Location for this comparison

classmethod get_symbol()
   Get the symbol used in debug dumps for this gcc.Comparison subclass, if any, as a str. A table showing these strings can be seen here.

Subclasses include:

<table>
<thead>
<tr>
<th>Subclass</th>
<th>C/C++ operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>class EqExpr</td>
<td>==</td>
</tr>
<tr>
<td>class GeExpr</td>
<td>&gt;=</td>
</tr>
<tr>
<td>class GtExpr</td>
<td>&gt;</td>
</tr>
<tr>
<td>class LeExpr</td>
<td>&lt;=</td>
</tr>
<tr>
<td>class LtExpr</td>
<td>&lt;</td>
</tr>
<tr>
<td>class LtgtExpr</td>
<td></td>
</tr>
<tr>
<td>class NeExpr</td>
<td>!=</td>
</tr>
<tr>
<td>class OrderedExpr</td>
<td></td>
</tr>
<tr>
<td>class UneqExpr</td>
<td></td>
</tr>
<tr>
<td>class UngeExpr</td>
<td></td>
</tr>
<tr>
<td>class UngtExpr</td>
<td></td>
</tr>
<tr>
<td>class UnleExpr</td>
<td></td>
</tr>
<tr>
<td>class UnltExpr</td>
<td></td>
</tr>
<tr>
<td>class UnorderedExpr</td>
<td></td>
</tr>
</tbody>
</table>
12.8 References to storage

class gcc.Reference
Subclass of gcc.Tree for expressions involving a reference to storage.

Corresponds to the tcc_reference value of enum tree_code_class within GCC’s own C sources.

location
The gcc.Location for this storage reference

class method get_symbol()
Get the symbol used in debug dumps for this gcc.Reference subclass, if any, as a str. A table showing these strings can be seen here.

class gcc.ArrayRef
A subclass of gcc.Reference for expressions involving an array reference:

```python
unsigned char buffer[4096];
...
/* The left-hand side of this gcc.GimpleAssign is a gcc.ArrayRef: */
buffer[42] = 0xff;
```

array
The gcc.Tree for the array within the reference (gcc.VarDecl('buffer') in the example above)

index
The gcc.Tree for the index within the reference (gcc.IntegerCst(42) in the example above)

class gcc.ComponentRef
A subclass of gcc.Reference for expressions involving a field lookup.

This can mean either a direct field lookup, as in:

```python
struct mystruct s;
...
s.idx = 42;
```

or dereferenced field lookup:

```python
struct mystruct *p;
...
p->idx = 42;
```

target
The gcc.Tree for the container of the field (either s or *p in the examples above)

field
The gcc.FieldDecl for the field within the target.

class gcc.MemRef
A subclass of gcc.Reference for expressions involving dereferencing a pointer:

```python
int p, *q;
...
p = *q;
```

operand
The gcc.Tree for the expression describing the target of the pointer

Other subclasses of gcc.Reference include:
### 12.9 Other expression subclasses

```python
class gcc.Expression
    Subclass of gcc.Tree indicating an expression that doesn’t fit into the other categories.
    Corresponds to the tcc_expression value of enum tree_code_class within GCC’s own C sources.

    location
        The gcc.Location for this expression

    classmethod get_symbol()
        Get the symbol used in debug dumps for this gcc.Expression subclass, if any, as a str. A table
        showing these strings can be seen [here](#).
```

Subclasses include:

<table>
<thead>
<tr>
<th>Subclass</th>
<th>C/C++ operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>class ArrayRangeRef</td>
<td></td>
</tr>
<tr>
<td>class AttrAddrExpr</td>
<td></td>
</tr>
<tr>
<td>class BitFieldRef</td>
<td></td>
</tr>
<tr>
<td>class ImagpartExpr</td>
<td></td>
</tr>
<tr>
<td>class IndirectRef</td>
<td></td>
</tr>
<tr>
<td>class MemberRef</td>
<td></td>
</tr>
<tr>
<td>class OffsetRef</td>
<td></td>
</tr>
<tr>
<td>class RealpartExpr</td>
<td></td>
</tr>
<tr>
<td>class ScopeRef</td>
<td></td>
</tr>
<tr>
<td>class TargetMemRef</td>
<td></td>
</tr>
<tr>
<td>class UnconstrainedArrayRef</td>
<td></td>
</tr>
<tr>
<td>class ViewConvertExpr</td>
<td></td>
</tr>
</tbody>
</table>
### C/C++ operators

<table>
<thead>
<tr>
<th>Subclass</th>
<th>C/C++ operators</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class gcc.AddrExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.AlignofExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.ArrowExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.AssertExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.AtEncodeExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.BindExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.CMaybeConstExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.ClassReferenceExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.CleanupPointExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.CompoundExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.CompoundLiteralExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.CondExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.CtorInitializer</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.DlExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.DotProdExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.DotstarExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.EmptyClassExpr</code></td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Subclass</th>
<th>C/C++ operators</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class gcc.ExcessPrecisionExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.ExprPackExpansion</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.ExprStmt</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.FdescExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.FmaExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.InitExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.MessageSendExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.ModifyExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.ModopExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.MustNotThrowExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.NonDependentExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.NontypeArgumentPack</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.NullExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.NwExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.ObjTypeRef</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.OffsetofExpr</code></td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Subclass</th>
<th>C/C++ operators</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class gcc.PolynomialChrec</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.PostdecrementExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.PostincrementExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.PredecrementExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.PredictExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.PreincrementExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.PropertyRef</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.PseudoDtorExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.RealignLoad</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.SaveExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.ScevKnown</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.ScevNotKnown</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.SizeofExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.StmtExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.TagDefn</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.TargetExpr</code></td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
### Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Subclass</th>
<th>C/C++ operators</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class gcc.TemplateIdExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.ThrowExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.TruthAndExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.TruthAndifExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.TruthNotExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.TruthOrExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.TruthOrifExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.TruthXorExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.TypeString</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.TypeIdExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.VaArgExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.VecCondExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.VecDlExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.VecInitExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.VecNwExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.WidenMultMinusExpr</code></td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Subclass Class</th>
<th>C/C++ operators</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class gcc.WidenMultPlusExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.WithCleanupExpr</code></td>
<td></td>
</tr>
<tr>
<td><code>class gcc.WithSizeExpr</code></td>
<td></td>
</tr>
</tbody>
</table>

TODO

12.10 Statements

class gcc.Statement
A subclass of `gcc.Tree` for statements
Corresponds to the `tcc_statement` value of `enum tree_code_class` within GCC’s own C sources.

class gcc.CaseLabelExpr
A subclass of `gcc.Statement` for the `case` and `default` labels within a `switch` statement.

low
- for single-valued case labels, the value, as a `gcc.Tree`
- for range-valued case labels, the lower bound, as a `gcc.Tree`
- `None` for the default label

high
For range-valued case labels, the upper bound, as a `gcc.Tree`. `None` for single-valued case labels, and for the default label

target
The target of the case label, as a `gcc.LabelDecl`

12.11 SSA Names

class gcc.SsaName
A subclass of `gcc.Tree` representing a variable references during SSA analysis. New SSA names are created every time a variable is assigned a new value.

var
The variable being referenced, as a `gcc.VarDecl` or `gcc.ParmDecl`

def_stmt
The `gcc.Gimple` statement which defines this SSA name

version
An `int` value giving the version number of this SSA name
class gcc.Gimple
   A statement, in GCC’s Gimple representation.

   The __str__ method is implemented using GCC’s own pretty-printer for gimple, so e.g.:

   str(stmt)

   might return:

   'D.3259 = (long unsigned int) i;'

loc
   Source code location of this statement, as a gcc.Location (or None)

block
   The lexical block holding this statement, as a gcc.Tree

exprtype
   The type of the main expression computed by this statement, as a gcc.Tree (which might be gcc.VoidType)

str_no_uid
   A string representation of this statement, like str(), but without including any internal UIDs.

   This is intended for use in selftests that compare output against some expected value, to avoid embedding values that change into the expected output.

   For example, given an assignment to a temporary, the str(stmt) for the gcc.GimpleAssign might be:

   'D.3259 = (long unsigned int) i;'

   where the UID “3259” is liable to change from compile to compile, whereas the stmt.str_no_uid has value:

   'D.xxxx = (long unsigned int) i;'

   which won’t arbitrarily change each time.
**walk_tree** *(callback, *args, **kwargs)*

Visit all **gcc.Tree** nodes associated with this statement, potentially more than once each. This will visit both the left-hand-side and right-hand-side operands of the statement (if any), and recursively visit any of their child nodes.

For each node, the callback is invoked, supplying the node, and any extra positional and keyword arguments passed to **walk_tree**:

```
callback(node, *args, **kwargs)
```

If the callback returns a true value, the traversal stops, and that **gcc.Tree** is the result of the call to **walk_tree**. Otherwise, the traversal continues, and **walk_tree** eventually returns **None**.

**gcc.Gimple** has various subclasses, each corresponding to one of the kinds of statement within GCC’s internal representation.

The following subclasses have been wrapped for use from Python scripts:

<table>
<thead>
<tr>
<th>Subclass</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc.GimpleAsm</td>
<td>One or more inline assembly statements</td>
</tr>
<tr>
<td>gcc.GimpleAssign</td>
<td>An assignment of an expression to an l-value:</td>
</tr>
<tr>
<td></td>
<td>LHS = RHS1 EXPRCODE RHS2;</td>
</tr>
<tr>
<td>gcc.GimpleCall</td>
<td>A function call:</td>
</tr>
<tr>
<td></td>
<td>[ LHS = ] FN(ARG1, ..., ARGN);</td>
</tr>
<tr>
<td>gcc.GimpleCond</td>
<td>A conditional jump, of the form:</td>
</tr>
<tr>
<td></td>
<td>if (LHS EXPRCODE RHS) goto TRUE_LABEL;</td>
</tr>
<tr>
<td></td>
<td>→ else goto FALSE_LABEL;</td>
</tr>
<tr>
<td>gcc.GimpleLabel</td>
<td>A label statement (jump target):</td>
</tr>
<tr>
<td></td>
<td>LABEL:</td>
</tr>
<tr>
<td>gcc.GimpleNop</td>
<td>The “do nothing” statement</td>
</tr>
<tr>
<td>gcc.GimplePhi</td>
<td>Used in the SSA passes:</td>
</tr>
<tr>
<td></td>
<td>LHS = PHI &lt;ARG1, ..., ARGN&gt;;</td>
</tr>
<tr>
<td>gcc.GimpleReturn</td>
<td>A “return” statement:</td>
</tr>
<tr>
<td></td>
<td>RETURN [RETVAL];</td>
</tr>
<tr>
<td>gcc.GimpleSwitch</td>
<td>A switch statement:</td>
</tr>
<tr>
<td></td>
<td>switch (INDEXVAR) {</td>
</tr>
<tr>
<td></td>
<td>case LAB1: ...; break;</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>case LABN: ...; break;</td>
</tr>
<tr>
<td></td>
<td>default: ...</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

There are some additional subclasses that have not yet been fully wrapped by the Python plugin (email the gcc-python-plugin’s mailing list if you’re interested in working with these):
### Subclass

<table>
<thead>
<tr>
<th>Subclass</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc.GimpleBind</td>
<td>A lexical scope</td>
</tr>
<tr>
<td>gcc.GimpleCatch</td>
<td>An exception handler</td>
</tr>
<tr>
<td>gcc.GimpleDebug</td>
<td>A debug statement</td>
</tr>
<tr>
<td>gcc.GimpleEhDispatch</td>
<td>Used in exception-handling</td>
</tr>
<tr>
<td>gcc.GimpleEhFilter</td>
<td>Used in exception-handling</td>
</tr>
<tr>
<td>gcc.GimpleEhMustNotThrow</td>
<td>Used in exception-handling</td>
</tr>
<tr>
<td>gcc.GimpleErrorMark</td>
<td>A dummy statement used for handling internal errors</td>
</tr>
<tr>
<td>gcc.GimpleGoto</td>
<td>An unconditional jump</td>
</tr>
<tr>
<td>gcc.GimpleOmpAtomicLoad</td>
<td>Used for implementing OpenMP</td>
</tr>
<tr>
<td>gcc.GimpleOmpAtomicStore</td>
<td>(ditto)</td>
</tr>
<tr>
<td>gcc.GimpleOmpContinue</td>
<td>(ditto)</td>
</tr>
<tr>
<td>gcc.GimpleOmpCritical</td>
<td>(ditto)</td>
</tr>
<tr>
<td>gcc.GimpleOmpFor</td>
<td>(ditto)</td>
</tr>
<tr>
<td>gcc.GimpleOmpMaster</td>
<td>(ditto)</td>
</tr>
<tr>
<td>gcc.GimpleOmpOrdered</td>
<td>(ditto)</td>
</tr>
<tr>
<td>gcc.GimpleOmpParallel</td>
<td>(ditto)</td>
</tr>
<tr>
<td>gcc.GimpleOmpReturn</td>
<td>(ditto)</td>
</tr>
<tr>
<td>gcc.GimpleOmpSection</td>
<td>(ditto)</td>
</tr>
<tr>
<td>gcc.GimpleOmpSections</td>
<td>(ditto)</td>
</tr>
<tr>
<td>gcc.GimpleOmpSectionsSwitch</td>
<td>(ditto)</td>
</tr>
<tr>
<td>gcc.GimpleOmpSingle</td>
<td>(ditto)</td>
</tr>
<tr>
<td>gcc.GimpleOmpTask</td>
<td>(ditto)</td>
</tr>
<tr>
<td>gcc.GimplePredict</td>
<td>A hint for branch prediction</td>
</tr>
<tr>
<td>gcc.GimpleResx</td>
<td>Resumes execution after an exception</td>
</tr>
<tr>
<td>gcc.GimpleTry</td>
<td>A try/catch or try/finally statement</td>
</tr>
<tr>
<td>gcc.GimpleWithCleanupExpr</td>
<td>Internally used when generating GIMPLE</td>
</tr>
</tbody>
</table>

**class gcc.GimpleAsm**

Subclass of *gcc.Gimple*: a fragment of inline assembler code.

**string**

The inline assembler code, as a *str*.

**class gcc.GimpleAssign**

Subclass of *gcc.Gimple*: an assignment of an expression to an l-value:

\[ \text{LHS} = \text{RHS1} \text{ EXPRCODE } \text{RHS2} \; \]

**lhs**

Left-hand-side of the assignment, as a *gcc.Tree*

**rhs**

The operands on the right-hand-side of the expression, as a list of *gcc.Tree* instances (either of length 1 or length 2, depending on the expression).

**expcode**

The kind of the expression, as an *gcc.Tree* subclass (the type itself, not an instance)

**class gcc.GimpleCall**

Subclass of *gcc.Gimple*: an invocation of a function, potentially assigning the result to an l-value:

\[ [ \text{LHS} = ] \text{FN} (\text{ARG1}, \ldots, \text{ARGN}) ; \]
lhs
Left-hand-side of the assignment, as a `gcc.Tree`, or `None`

rhs
The operands on the right-hand-side of the expression, as a list of `gcc.Tree` instances

fn
The function being called, as a `gcc.Tree`

fndecl
The declaration of the function being called (if any), as a `gcc.Tree`

args
The arguments for the call, as a list of `gcc.Tree`

noreturn
(boolean) Has this call been marked as not returning? (e.g. a call to `exit`)

class gcc.GimpleReturn
Subclass of `gcc.Gimple`: a “return” statement, signifying the end of a `gcc.BasicBlock`:

```
RETURN [RETVAL];
```

retval
The return value, as a `gcc.Tree`, or `None`.

class gcc.GimpleCond
Subclass of `gcc.Gimple`: a conditional jump, of the form:

```
if (LHS EXPRCODE RHS) goto TRUE_LABEL else goto FALSE_LABEL
```

lhs
Left-hand-side of the comparison, as a `gcc.Tree`

exprcode
The comparison predicate, as a `gcc.Comparison` subclass (the type itself, not an instance). For example, the gcc.GimpleCond statement for this fragment of C code:

```
if (a == b)
```

would have stmt.exprcode == gcc.EqExpr

rhs
The right-hand-side of the comparison, as a `gcc.Tree`

true_label
The `gcc.LabelDecl` node used as the jump target for when the comparison is true

false_label
The `gcc.LabelDecl` node used as the jump target for when the comparison is false

Note that a C conditional of the form:

```
if (some_int) {suiteA} else {suiteB}
```

is implicitly expanded to:

```
if (some_int != 0) {suiteA} else {suiteB}
```

and this becomes a gcc.GimpleCond with `lhs` as the integer, `exprcode` as `<type 'gcc.NeExpr'>`, and `rhs` as `gcc.IntegerCst(0)`.
class gcc.GimplePhi

Subclass of gcc.Gimple used in the SSA passes: a “PHI” or “phoney” function, for merging the various possible values a variable can have based on the edge that we entered this gcc.BasicBlock on:

```
LHS = PHI <ARG1, ..., ARGN>;
```

lhs
Left-hand-side of the assignment, as a gcc.SsaName

args
A list of (gcc.Tree, gcc.Edge) pairs representing the possible (expr, edge) inputs. Each expr is either a gcc.SsaName or a gcc.Constant

class gcc.GimpleSwitch

Subclass of gcc.Gimple: a switch statement, signifying the end of a gcc.BasicBlock:

```
switch (INDEXVAR) {
  case LAB1: ...; break;
  ...
  case LABN: ...; break;
  default: ...
}
```

indexvar
The index variable used by the switch statement, as a gcc.Tree

labels
The labels of the switch statement, as a list of gcc.CaseLabelExpr.

  The initial label in the list is always the default.

class gcc.GimpleLabel

Subclass of gcc.Gimple, representing a “label” statement:

```
.. py:attribute:: labels
```

  The underlying gcc.LabelDecl node representing this jump target

class gcc.GimpleAssign

Subclass of gcc.Gimple: an assignment of an expression to an l-value:

```
LHS = RHS1 EXPRCODE RHS2;
```

lhs
Left-hand-side of the assignment, as a gcc.Tree

rhs
The operands on the right-hand-side of the expression, as a list of gcc.Tree instances (either of length 1 or length 2, depending on the expression).

exprcode
The kind of the expression, as an gcc.Tree subclass (the type itself, not an instance)

class gcc.GimpleNop

14.1 Working with existing passes

GCC organizes the optimization work it does as “passes”, and these form trees: passes can have both successors and child passes.

There are actually five “roots” to this tree:

- The `gcc.Pass` holding all “lowering” passes, invoked per function within the callgraph, to turn high-level GIMPLE into lower-level forms (this wraps `all_lowering_passes` within `gcc/passes.c`).
- The `gcc.Pass` holding all “small IPA” passes, working on the whole callgraph (IPA is “Interprocedural Analysis”; `all_small_ipa_passes` within `gcc/passes.c`)
- The `gcc.Pass` holding all regular IPA passes (`all_regular_ipa_passes` within `gcc/passes.c`)
- The `gcc.Pass` holding those passes relating to link-time-optimization (`all_lto_gen_passes` within `gcc/passes.c`)
- The “all other passes” `gcc.Pass` catchall, holding the majority of the passes. These are called on each function within the call graph (`all_passes` within `gcc/passes.c`)

**classmethod gcc.Pass.get_roots()**

Returns a 5-tuple of `gcc.Pass` instances, giving the 5 top-level passes within GCC’s tree of passes, in the order described above.

**classmethod gcc.Pass.get_by_name(name)**

Get the `gcc.Pass` instance for the pass with the given name, raising `ValueError` if it isn’t found

**class gcc.Pass**

This wraps one of GCC’s `struct opt_pass *` instances.

Beware: “pass” is a reserved word in Python, so use e.g. `ps` as a variable name for an instance of `gcc.Pass`

**name**

The name of the pass, as a string
sub
   The first child pass of this pass (if any)

next
   The next sibling pass of this pass (if any)

properties_required

properties_provided

properties_destroyed
   Currently these are int bitfields, expressing the flow of data between the various passes.

   They can be accessed using bitwise arithmetic:

   

   ```python
   if ps.properties_provided & gcc.PROP_cfg:
      print(fn.cfg)
   ```

   Here are the bitfield flags:

<table>
<thead>
<tr>
<th>Mask</th>
<th>Meaning</th>
<th>Which pass sets this up?</th>
<th>Which pass clears this?</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc.PROP_gimple_any</td>
<td>Is the full GIMPLE grammar allowed?</td>
<td>(the frontend)</td>
<td>“expand”</td>
</tr>
<tr>
<td>gcc.PROP_gimple_lcf</td>
<td>Has control flow been lowered?</td>
<td>“lower”</td>
<td>“expand”</td>
</tr>
<tr>
<td>gcc.PROP_gimple_eh</td>
<td>Has exception-handling been lowered?</td>
<td>“eh”</td>
<td>“expand”</td>
</tr>
<tr>
<td>gcc.PROP_cfg</td>
<td>Does the gcc.Function have a non-None “cfg”?</td>
<td>“cfg”</td>
<td>“*free_cfg”</td>
</tr>
<tr>
<td>gcc.PROP_referenced_vars</td>
<td>Data on which functions reference which variables? (Dataflow analysis, aka DFA). This flag was removed in GCC 4.8</td>
<td>“*referenced_vars”</td>
<td>(none)</td>
</tr>
<tr>
<td>gcc.PROP_ssa</td>
<td>Is the GIMPLE in SSA form?</td>
<td>“ssa”</td>
<td>“expand”</td>
</tr>
<tr>
<td>gcc.PROP_no_crit_edges</td>
<td>Have all critical edges within the CFG been split?</td>
<td>“crited”</td>
<td>(none)</td>
</tr>
<tr>
<td>gcc.PROP_rtl</td>
<td>Is the function now in RTL form? (rather than GIMPLE-SSA)</td>
<td>“rtl”</td>
<td>“*clean_state”</td>
</tr>
<tr>
<td>gcc.PROP_openmp</td>
<td>Has OpenMP directives been lowered into explicit calls to the runtime library (libgomp)</td>
<td>“omp_lower”</td>
<td>“expand”</td>
</tr>
<tr>
<td>gcc.PROP_cfglayout</td>
<td>Are we reorganizing the CFG into a more efficient order?</td>
<td>“into_cfglayout”</td>
<td>“outof_cfglayout”</td>
</tr>
<tr>
<td>gcc.PROP_cplxlower</td>
<td>Operations on complex numbers been lowered to scalar operations?</td>
<td>“cplxlower”</td>
<td>“cplxlower0”</td>
</tr>
</tbody>
</table>

**static_pass_number**
   (int) The number of this pass, used as a fragment of the dump file name. This is assigned automatically for custom passes.

**dump_enabled**
   (boolean) Is dumping enabled for this pass? Set this attribute to True to enable dumping. Not available from GCC 4.8 onwards

There are four subclasses of **gcc.Pass**:

**class gcc.GimplePass**
   Subclass of **gcc.Pass**, signifying a pass called per-function on the GIMPLE representation of that function.
class gcc.RtlPass
    Subclass of gcc.Pass, signifying a pass called per-function on the RTL representation of that function.

class gcc.SimpleIpaPass
    Subclass of gcc.Pass, signifying a pass called once (not per-function)

class gcc.IpaPass
    Subclass of gcc.Pass, signifying a pass called once (not per-function)

14.2 Creating new optimization passes

You can create new optimization passes. This involves three steps:

• subclassing the appropriate gcc.Pass subclass (e.g. gcc.GimplePass)
• creating an instance of your subclass
• registering the instance within the pass tree, relative to another pass

Here’s an example:

```python
# Here's the (trivial) implementation of our new pass:
class MyPass(gcc.GimplePass):
    # This is optional.
    # If present, it should return a bool, specifying whether or not
    # to execute this pass (and any child passes)
    def gate(self, fun):
        print('gate() called for %r' % fun)
        return True

    def execute(self, fun):
        print('execute() called for %r' % fun)

# We now create an instance of the class:
my_pass = MyPass(name='my-pass')

# ...and wire it up, after the "cfg" pass:
my_pass.register_after('cfg')
```

For gcc.GimplePass and gcc.RtlPass, the signatures of gate and execute are:

```python
gate(self, fun)
execute(self, fun)
```

where fun is a gcc.Function.

For gcc.SimpleIpaPass and gcc.IpaPass, the signature of gate and execute are:

```python
gate(self)
execute(self)
```

**Warning:** Unfortunately it doesn’t appear to be possible to implement gate() for gcc.IpaPass yet; for now, the gate() method on such passes will not be called. See [http://gcc.gnu.org/bugzilla/show_bug.cgi?id=54959](http://gcc.gnu.org/bugzilla/show_bug.cgi?id=54959)

If an unhandled exception is raised within gate or execute, it will lead to a GCC error:
gcc.Pass.register_after(name[, instance_number=0])

Given the name of another pass, register this gcc.Pass to occur immediately after that other pass.

If the other pass occurs multiple times, the pass will be inserted at the specified instance number, or at every instance, if supplied 0.

**Note:** The other pass must be of the same kind as this pass. For example, if it is a subclass of gcc.GimplePass, then this pass must also be a subclass of gcc.GimplePass.

If they don’t match, GCC won’t be able to find the other pass, giving an error like this:

ccl: fatal error: pass 'ssa' not found but is referenced by new pass 'my-ipa-pass'

where we attempted to register a gcc.IpaPass subclass relative to ‘ssa’, which is a gcc.GimplePass

gcc.Pass.register_before(name[, instance_number=0])

As above, but this pass is registered immediately before the referenced pass.

gcc.Pass.replace(name[, instance_number=0])

As above, but replace the given pass. This method is included for completeness; the result is unlikely to work well.

14.3 Dumping per-pass information

GCC has a logging framework which supports per-pass logging (“dump files”).

By default, no logging is done; dumping must be explicitly enabled.

Dumping of passes can be enabled from the command-line in groups:

- `-fdump-tree-all` enables dumping for all gcc.GimplePass (both builtin, and custom ones from plugins)
- `-fdump-rtl-all` is similar, but for all gcc.RtlPass
- `-fdump-ipa-all` as above, but for all gcc.IpaPass and gcc.SimpleIpaPass

For more information, see [http://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html](http://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html)

It’s not possible to directly enable dumping for a custom pass from the command-line (it would require adding new GCC command-line options). However, your script can directly enable dumping for a custom pass by writing to the dump_enabled attribute (perhaps in response to the arguments passed to plugin, or a driver script).

If enabled for a pass, then a file is written to the same directory as the output file, with a name based on the input file and the pass number.

For example, given a custom gcc.Pass with name ‘test-pass’, then when input.c is compiled to build/output.o:

```
$ gcc -fdump-tree-all -o build/output.o src/input.c
```

then a dump file input.c.225t.test-pass will be written to the directory build. In this case, 225 is the static_pass_number field, “t” signifies a tree pass, with the pass name appearing as the suffix.
**gcc.dump** *(obj)*

Write `str()` of the argument to the current dump file. No newlines or other whitespace are added.

Note that dumping is disabled by default; in this case, the call will do nothing.

**gcc.get_dump_file_name()**

Get the name of the current dump file.

If called from within a pass for which dumping is enabled, it will return the filename in string form.

If dumping is disabled for this pass, it will return *None*.

The typical output of a dump file will contain:

```plaintext
;; Function bar (bar)
(dumped information when handling function bar goes here)

;; Function foo (foo)
(dumped information when handling function foo goes here)
```

For example:

```python
class TestPass(gcc.GimplePass):
    def execute(self, fun):
        # Dumping of strings:
        gcc.dump('hello world')

        # Dumping of other objects:
        gcc.dump(42)

ps = TestPass(name='test-pass')
pw.register_after('cfg')
ps.dump_enabled = True
```

would have a dump file like this:

```plaintext
;; Function bar (bar)
hello world42

;; Function foo (foo)
hello world42
```

Alternatively, it can be simpler to create your own logging system, given that one can simply open a file and write to it.

**gcc.get_dump_base_name()**

Get the base file path and name prefix for GCC’s dump files.

You can use this when creating non-standard logfiles and other output.

For example, the libcpychecker code can write HTML reports on reference-counting errors within a function, writing the output to a file named:

```plaintext
filename = '%s.%s-refcount-errors.html' % (gcc.get_dump_base_name(), fun.decl.name)
```

given *fun*, a *gcc.Function*. 

---

**14.3. Dumping per-pass information**
By default, this is the name of the input file, but within the output file’s directory. (It can be overridden using the `-dumpbase` command-line option).
CHAPTER 15

Working with callbacks

One way to work with GCC from the Python plugin is via callbacks. It’s possible to register callback functions, which will be called when various events happen during compilation.

For example, it’s possible to piggyback off of an existing GCC pass by using `gcc.PLUGIN_PASS_EXECUTION` to piggyback off of an existing GCC pass.

```python
import gcc

def my_pass_execution_callback(*args, **kwargs):
    print('my_pass_execution_callback was called: args=%r  kwargs=%r'
          % (args, kwargs))

gcc.register_callback(gcc.PLUGIN_PASS_EXECUTION,
                      my_pass_execution_callback)
```

The exact arguments passed to your callback vary: consult the documentation for the particular event you are wiring up to (see below).

You can pass additional arguments when registering the callback - they will be passed to the callback after any normal arguments. This is denoted in the descriptions of events below by `*extraargs`.

You can also supply keyword arguments: they will be passed on as keyword arguments to the callback. This is denoted in the description of events below by `**kwargs`.

The various events are exposed as constants within the `gcc` module and directly wrap GCC’s plugin mechanism.

The following GCC events are currently usable from the Python plugin via `gcc.register_callback()`:

```python
```
### ID | Meaning
---|---
gcc.PLUGIN_ATTRIBUTES | For creating custom GCC attributes

gcc.PLUGIN_PRE_GENERICIZE | For working with the AST in the C and C++ frontends

gcc.PLUGIN_PASS_EXECUTION | Called before each pass is executed

gcc.PLUGIN_FINISH_UNIT | At the end of working with a translation unit (aka source file)

gcc.PLUGIN_FINISH_TYPE | After a type has been parsed

gcc.PLUGIN_FINISH_DECL | After a declaration has been parsed (GCC 4.7 or later)

gcc.PLUGIN_FINISH | Called before GCC exits

**gcc.PLUGIN_ATTRIBUTES**

Called when GCC is creating attributes for use with its non-standard `__attribute__(())` syntax.

If you want to create custom GCC attributes, you should register a callback on this event and call `gcc.register_attribute()` from within that callback, so that they are created at the same time as the GCC’s built-in attributes.

No arguments are passed to your callback other than those that you supply yourself when registering it:

```python
(*extraargs, **kwargs)
```

See *creating custom GCC attributes* for examples and more information.

**gcc.PLUGIN_PASS_EXECUTION**

Called when GCC is about to run one of its passes.

Arguments passed to the callback are:

```python
(ps, fun, *extraargs, **kwargs)
```

where `ps` is a `gcc.Pass` and `fun` is a `gcc.Function`. Your callback will typically be called many times: there are many passes, and each can be invoked zero or more times per function (in the code being compiled).

More precisely, some passes have a “gate check”: the pass first checks a condition, and only executes if the condition is true.

Any callback registered with `gcc.PLUGIN_PASS_EXECUTION` will get called if this condition succeeds.

The actual work of the pass is done after the callbacks return.

In pseudocode:

```python
if pass.has_gate_condition:
    if !pass.test_gate_condition():
        return
    invoke_all_callbacks()
    actually_do_the_pass()
```

For passes working on individual functions, all of the above is done per-function.

To connect to a specific pass, you can simply add a conditional based on the name of the pass:

```python
import gcc

def my_callback(ps, fun):
    if ps.name != '*warn_function_return':
        # Not the pass we want
        return
    # Do something here
    print(fun.decl.name)
```

(continues on next page)
gcc.register_callback(gcc.PLUGIN_PASS_EXECUTION, my_callback)

gcc.PLUGIN_PRE_GENERICIZE  
Arguments passed to the callback are:  
  (fndecl, *extraargs, **kwargs)  
where fndecl is a gcc.Tree representing a function declaration within the source code being compiled.

gcc.PLUGIN_FINISH_UNIT  
Called when GCC has finished compiling a particular translation unit.  
Arguments passed to the callback are:  
  (*extraargs, **kwargs)

gcc.PLUGIN_FINISH_DECL  
Note: Only available in GCC 4.7 onwards.  
Called when GCC has finished compiling a declaration (variables, functions, parameters to functions, types, etc)  
Arguments passed to the callback are:  
  (decl, *extraargs, **kwargs)  
where decl is a gcc.Declaration.

gcc.PLUGIN_FINISH_TYPE  
Called when GCC has finished parsing a type. Arguments to the callback are:  
  (type, *extraargs, **kwargs)  
where type is a gcc.Type.

gcc.PLUGIN_FINISH  
Called before GCC exits.  
Arguments passed to the callback are:  
  (*extraargs, **kwargs)

The remaining GCC events aren’t yet usable from the plugin; an attempt to register a callback on them will lead to an exception being raised. Email the gcc-python-plugin’s mailing list if you’re interested in working with these):
<table>
<thead>
<tr>
<th>ID</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc.PLUGIN_PASS_MANAGER_SETUP</td>
<td>To hook into pass manager</td>
</tr>
<tr>
<td>gcc.PLUGIN_INFO</td>
<td>Information about the plugin</td>
</tr>
<tr>
<td>gcc.PLUGIN_GGC_START</td>
<td>For interacting with GCC’s garbage collector</td>
</tr>
<tr>
<td>gcc.PLUGIN_GGC_MARKING</td>
<td>(ditto)</td>
</tr>
<tr>
<td>gcc.PLUGIN_GGC_END</td>
<td>(ditto)</td>
</tr>
<tr>
<td>gcc.PLUGIN_REGISTER_GGC_ROOTS</td>
<td>(ditto)</td>
</tr>
<tr>
<td>gcc.PLUGIN_REGISTER_GGC_CACHES</td>
<td>(ditto)</td>
</tr>
<tr>
<td>gcc.PLUGIN_START_UNIT</td>
<td>Called before processing a translation unit (aka source file)</td>
</tr>
<tr>
<td>gcc.PLUGIN_PRAGMAS</td>
<td>For registering pragmas</td>
</tr>
<tr>
<td>gcc.PLUGIN_ALL_PASSES_START</td>
<td>Called before the first pass of the “all other passes” gcc.Pass catchall</td>
</tr>
<tr>
<td>gcc.PLUGIN_ALL_PASSES_END</td>
<td>Called after last pass of the “all other passes” gcc.Pass catchall</td>
</tr>
<tr>
<td>gcc.PLUGIN_ALL_IPA_PASSES_START</td>
<td>Called before the first IPA pass</td>
</tr>
<tr>
<td>gcc.PLUGIN_ALL_IPA_PASSES_END</td>
<td>Called after last IPA pass</td>
</tr>
<tr>
<td>gcc.PLUGIN_OVERRIDE_GATE</td>
<td>Provides a way to disable a built-in pass</td>
</tr>
<tr>
<td>gcc.PLUGIN_EARLY_GIMPLE_PASSES_START</td>
<td></td>
</tr>
<tr>
<td>gcc.PLUGIN_EARLY_GIMPLE_PASSES_END</td>
<td></td>
</tr>
<tr>
<td>gcc.PLUGIN_NEW_PASS</td>
<td></td>
</tr>
</tbody>
</table>
Creating custom GCC attributes

GNU C supports a non-standard \_
\_attribute__\((())\) syntax for marking declarations with additional information that may be of interest to the optimizer, and for checking the correctness of the code.

The GCC Python plugin allows you to create custom attributes, which may be of use to your scripts: you can use this to annotate C code with additional information. For example, you could create a custom attribute for functions describing the interaction of a function on mutex objects:

```c
extern void some_function(void)
__attribute__((claims_mutex("io")));

extern void some_other_function(void)
__attribute__((releases_mutex("io")));
```

and use this in a custom code-checker.

Custom attributes can take string and integer parameters. For example, the above custom attributes take a single string parameter. A custom attribute can take more than one parameter, or none at all.

To create custom attributes from Python, you need to wire up a callback response to the `gcc.PLUGIN_ATTRIBUTES` event:

```python
gcc.register_callback(gcc.PLUGIN_ATTRIBUTES,
    register_our_attributes)
```

This callback should then call `gcc.register_attribute()` to associate the name of the attribute with a Python callback to be called when the attribute is encountered in C code.

```python
gcc.register_attribute(name, min_length, max_length, decl_required, type_required, function_type_required, callable)
```

Registers a new GCC attribute with the given `name`, usable in C source code via \_
\_attribute__\((())\).

**Parameters**

- `name (str)` – the name of the new attribute
- `min_length (int)` – the minimum number of arguments expected when the attribute is used
In this example, we can simply print when the attribute is seen, to verify that the callback mechanism is working:

```python
def attribute_callback_for_claims_mutex(*args):
    print('attribute_callback_for_claims_mutex called: args: %s' % (args, ))
def attribute_callback_for_releases_mutex(*args):
    print('attribute_callback_for_releases_mutex called: args: %s' % (args, ))
def register_our_attributes():
    gcc.register_attribute('claims_mutex',
        1, 1,
        False, False, False,
        attribute_callback_for_claims_mutex)
    gcc.register_attribute('releases_mutex',
        1, 1,
        False, False, False,
        attribute_callback_for_releases_mutex)
```

Putting it all together, here is an example Python script for the plugin:

```python
import gcc

# Verify that we can register custom attributes:
def attribute_callback_for_claims_mutex(*args):
    print('attribute_callback_for_claims_mutex called: args: %s' % (args, ))
def attribute_callback_for_releases_mutex(*args):
    print('attribute_callback_for_releases_mutex called: args: %s' % (args, ))
def register_our_attributes():
    gcc.register_attribute('claims_mutex',
        1, 1,
        False, False, False,
        attribute_callback_for_claims_mutex)
    gcc.register_attribute('releases_mutex',
        1, 1,
        False, False, False,
        attribute_callback_for_releases_mutex)

def wire_up_callback():
    gcc.register_callback(gcc.PLUGIN_ATTRIBUTES,
        register_our_attributes)
```

Compiling this test C source file:
/* Function declarations with custom attributes: */
extern some_function(void) __attribute__((claims_mutex("io")));
extern some_other_function(void) __attribute__((releases_mutex("io")));
extern yet_another_function(void) __attribute__((claims_mutex("db"),
claims_mutex("io"),
releases_mutex("io"));)

leads to this output from the script:

attribute_callback_for_claims_mutex called: args: (gcc.FunctionDecl('some_function'), gcc.StringCst('io'))
attribute_callback_for_releases_mutex called: args: (gcc.FunctionDecl('some_other_function'), gcc.StringCst('io'))
attribute_callback_for_claims_mutex called: args: (gcc.FunctionDecl('yet_another_function'), gcc.StringCst('db'))
attribute_callback_for_claims_mutex called: args: (gcc.FunctionDecl('yet_another_function'), gcc.StringCst('io'))
attribute_callback_for_releases_mutex called: args: (gcc.FunctionDecl('yet_another_function'), gcc.StringCst('io'))

16.1 Using the preprocessor to guard attribute usage

Unfortunately, the above C code will only work when it is compiled with the Python script that adds the custom attributes.

You can avoid this by using `gcc.define_macro()` to pre-define a preprocessor name (e.g. “WITH_ATTRIBUTE_CLAIMS_MUTEX”) at the same time as when you define the attribute:

```python
import gcc

def attribute_callback_for_claims_mutex(*args):
    print('attribute_callback_for_claims_mutex called: args: $s' % (args, ))

def attribute_callback_for_releases_mutex(*args):
    print('attribute_callback_for_releases_mutex called: args: $s' % (args, ))

def register_our_attributes():
    gcc.register_attribute('claims_mutex',
                          1, 1,
                          False, False, False,
                          attribute_callback_for_claims_mutex)
    gcc.define_macro('WITH_ATTRIBUTE_CLAIMS_MUTEX')

    gcc.register_attribute('releases_mutex',
                          1, 1,
                          False, False, False,
                          attribute_callback_for_releases_mutex)
    gcc.define_macro('WITH_ATTRIBUTERELEASES_MUTEX')

    # Wire up our callback:
    gcc.register_callback(gcc.PLUGIN_ATTRIBUTES,
```
This way the user can write this C code instead, and have it work both with and without the Python script:

```c
#if defined(WITH_ATTRIBUTE_CLAIMS_MUTEX)
  #define CLAIMS_MUTEX(x) __attribute__((claims_mutex(x)))
#else
  #define CLAIMS_MUTEX(x)
#endif

#if defined(WITH_ATTRIBUTE_RELEASES_MUTEX)
  #define RELEASES_MUTEX(x) __attribute__((releases_mutex(x)))
#else
  #define RELEASES_MUTEX(x)
#endif

/* Function declarations with custom attributes: */
extern void some_function(void) CLAIMS_MUTEX("io");
extern void some_other_function(void) RELEASES_MUTEX("io");
extern void yet_another_function(void) CLAIMS_MUTEX("db")
                                        CLAIMS_MUTEX("io")
                                        RELEASES_MUTEX("io");
```

giving this output from the script:

```c
attribute_callback_for_claims_mutex called: args: (gcc.FunctionDecl('some_˓
                                        →function'), gcc.StringCst('io'))
attribute_callback_for_releases_mutex called: args: (gcc.FunctionDecl('some_˓
                                        →other_function'), gcc.StringCst('io'))
attribute_callback_for_claims_mutex called: args: (gcc.FunctionDecl('yet_˓
                                        →another_function'), gcc.StringCst('db'))
attribute_callback_for_claims_mutex called: args: (gcc.FunctionDecl('yet_˓
                                        →another_function'), gcc.StringCst('io'))
attribute_callback_for_releases_mutex called: args: (gcc.FunctionDecl('yet_˓
                                        →another_function'), gcc.StringCst('io'))
```
Usage example: a static analysis tool for CPython extension code

Note: This code is under heavy development, and still contains bugs. It is not unusual to see Python tracebacks when running the checker. You should verify what the checker reports before acting on it: it could be wrong.

An example of using the plugin is a static analysis tool I’m working on which checks the C source of CPython extension modules for common coding errors.

This was one of my main motivations for writing the GCC plugin, and I often need to extend the plugin to support this use case.

For this reason, the checker is embedded within the gcc-python source tree itself for now:

- `gcc-with-cpychecker` is a harness script, which invokes GCC, adding the arguments necessary to use the Python plugin, using the `libcpychecker` Python code
- the `libcpychecker` subdirectory contains the code that does the actual work
- various test cases (in the source tree, below `tests/cpychecker`)

### 17.1 gcc-with-cpychecker

`gcc-with-cpychecker` is a harness script, which invokes GCC, adding the arguments necessary to use the Python plugin, using the `libcpychecker` Python code

You should be able to use the checker on arbitrary CPython extension code by replacing “gcc” with “gcc-with-cpychecker” in your build with something like:

```
made CC=/path/to/built/plugin/gcc-with-cpychecker
```

to override the Makefile variable `CC`.

You may need to supply an absolute path, especially if the “make” recursively invokes “make” within subdirectories (thus having a different working directory).
Similarly, for projects that use `distutils`, the code is typically built with an invocation like this:

```bash
python setup.py build
```

This respects the environment variable `CC`, so typically you can replace the above with something like this in order to add the additional checks:

```bash
CC=/path/to/built/plugin/gcc-with-cpychecker python setup.py build
```

### 17.1.1 Additional arguments for `gcc-with-cpychecker`

**`--maxtrans <int>`**

Set the maximum number of transitions to consider within each function before pruning the analysis tree. You may need to increase this limit for complicated functions.

**`--dump-json`**

Dumps a JSON representation of any problems. For example, given a function `foo.c`, if any warnings or errors are found in function `bar`, a file `foo.c.bar.json` will be written out in JSON form.

### 17.2 Reference-count checking

The checker attempts to analyze all possible paths through each function, tracking the various `PyObject*` objects encountered.

For each path through the function and `PyObject*`, it determines what the reference count ought to be at the end of the function, issuing warnings for any that are incorrect.

The warnings are in two forms: the classic textual output to GCC’s standard error stream, together with an HTML report indicating the flow through the function, in graphical form.

For example, given this buggy C code:

```c
PyObject *
test(PyObject *self, PyObject *args)
{
    PyObject *list;
    PyObject *item;
    list = PyList_New(1);
    if (!list)
        return NULL;
    item = PyLong_FromLong(42);
    /* This error handling is incorrect: it's missing an invocation of Py_DECREF(list): */
    if (!item)
        return NULL;
    /* This steals a reference to item; item is not leaked when we get here: */
    PyList_SetItem(list, 0, item);
    return list;
}
```

the checker emits these messages to stderr:

```plaintext
input.c: In function 'test':
input.c:38:1: warning: ob_refcnt of '*list' is 1 too high [enabled by default]
input.c:38:1: note: was expecting final ob_refcnt to be N + 0 (for some unknown N)
(continues on next page)
```
along with this HTML report (as referred to by the final line on stderr):

File: input.c
Function test
Error: ob_refcnt of 'list' is 1 too high

22 PyObject ^
23 test(PyObject *self, PyObject *args)
24 {
25     PyObject *list;
26     PyObject *item;
27     list = PyList_New(1);
28     if (!list)
29         return NULL;
30     item = PyLong_FromLong(42);
31     /* This error handling is incorrect: it's missing an
32        invocation of Py_DECREF(list): */
33     if (!item)
34         return NULL;
35     /* This steals a reference to item; item is not leaked when we get here: */
36     PyList_SetItem(list, 0, item);
37     return list;
38 }

ob_refcnt of '*list' is 1 too high
was expecting final ob_refcnt to be N + 0 (for some unknown N)
but final ob_refcnt is N + 1

The HTML report is intended to be relatively self-contained, and thus easy to attach to bug tracking systems (it embeds its own CSS inline, and references the JavaScript it uses via URLs to the web).

Note: The arrow graphics in the HTML form of the report are added by using the JSPlumb JavaScript library to generate HTML 5 <canvas> elements. You may need a relatively modern browser to see them.
Note: The checker tracks reference counts in an abstract way, in two parts: a part of the reference count that it knows about within the context of the function, along with a second part: all of the other references held by the rest of the program.

For example, in a call to PyInt_FromLong(0), it is assumed that if the call succeeds, the object has a reference count of 1 + N, where N is some unknown amount of other references held by the rest of the program. The checker knows that N \geq 0.

If the object is then stored in an opaque container which is known to increment the reference count, the checker can say that the reference count is then 1 + (N+1).

If the function then decrements the reference count (to finish transferring the reference to the opaque container), the checker now treats the object as having a reference count of 0 + (N+1): it no longer owns any references on the object, but the reference count is actually unchanged relative to the original 1 + N amount. It also knows, given that N \geq 0 that the actual reference count is \geq 1, and thus the object won’t (yet) be deallocated.

17.2.1 Assumptions and configuration

For any function returning a \texttt{PyObject*}, it assumes that the \texttt{PyObject*} should be either a new reference to an object, or NULL (with an exception set) - the function’s caller should “own” a reference to that object. For all other \texttt{PyObject*}, it assumes that there should be no references owned by the function when the function terminates.

It will assume this behavior for any function (or call through a function pointer) that returns a \texttt{PyObject*}.

It is possible to override this behavior using custom compiler attributes as follows:

Marking functions that return borrowed references

The checker provides a custom GCC attribute:

\begin{verbatim}
__attribute__((cpychecker_returns_borrowed_ref))
\end{verbatim}

which can be used to mark function declarations:

\begin{verbatim}
/* The checker automatically defines this preprocessor name when creating
 the custom attribute: */
#if defined(WITH_CPYCHECKER_RETURNS_BORROWED_REF_ATTRIBUTE)
 #define CPYCHECKER_RETURNS_BORROWED_REF __attribute__((cpychecker_returns_borrowed_ref))
#else
 #define CPYCHECKER_RETURNS_BORROWED_REF
#endif

PyObject *foo(void)
CPYCHECKER_RETURNS_BORROWED_REF;
\end{verbatim}

Given the above, the checker will assume that invocations of \texttt{foo()} are returning a borrowed reference (or NULL), rather than a new reference. It will also check that this is that case when verifying the implementation of \texttt{foo()} itself.

Marking functions that steal references to their arguments

The checker provides a custom GCC attribute:
which can be used to mark function declarations:

```c
#define CPYCHECKER_STEALS_REFERENCE_TO_ARG(n) _attribute__((cpychecker_steals_reference_to_arg(n)))
```

Given the above, the checker will assume that invocations of `foo()` steal a reference to the first argument (`obj`). It will also verify that this is the case when analyzing the implementation of `foo()` itself.

More then one argument can be marked:

```c
extern void foo(PyObject *obj) CPYCHECKER_STEALS_REFERENCE_TO_ARG(1);
```

The argument indices are 1-based (the above example is thus referring to `obj` and to `other`). All such arguments to the attribute should be `PyObject*` (or a pointer to a derived structure type).

It is assumed that such references are stolen for all possible outcomes of the function - if a function can either succeed or fail, the reference is stolen in both possible worlds.

### 17.3 Error-handling checking

The checker has knowledge of much of the CPython C API, and will generate a trace tree containing many of the possible error paths. It will issue warnings for code that appears to not gracefully handle an error.

(TODO: show example)

As noted above, it assumes that any function that returns a `PyObject*` can return can either NULL (setting an exception), or a new reference. It knows about much of the other parts of the CPython C API, including many other functions that can fail.

The checker will emit warnings for various events:

- if it detects a dereferencing of a NULL value
- if a NULL value is erroneously passed to various CPython API entrypoints which are known to implicitly dereference those arguments (which would lead to a segmentation fault if that code path were executed):
• if it detects that an uninitialized local variable has been used

• if it detects access to an object that has been deallocated, or such an object being returned:

```c
input.c: In function 'test':
  input.c:43:1: warning: returning pointer to deallocated memory
  input.c:29:15: note: when PyLong_FromLong() succeeds at:
  → PyLong_FromLong(0x1000);
  input.c:31:8: note: taking False path at: if (!tmp) {
  input.c:39:5: note: calling tp_dealloc on PyLongObject allocated at input.c:29
  → at: Py_DECREF(tmp);
  input.c:42:5: note: reaching: return tmp;
  input.c:39:5: note: memory deallocated here
  input.c:27:1: note: graphical error report for function 'returning_dead_object'
  → written out to 'input.c.test.html'
```

17.4 Errors in exception-handling

The checker keeps track of the per-thread exception state. It will issue a warning about any paths through functions returning a `PyObject*` that return NULL for which the per-thread exception state has not been set:

```c
input.c: In function 'test':
  input.c:32:5: warning: returning (PyObject*)NULL without setting an exception
```

The checker does not emit the warning for cases where it is known that such behavior is acceptable. Currently this covers functions used as `tp_iternext` callbacks of a `PyTypeObject`.

If you have a helper function that always sets an exception, you can mark this property using a custom GCC attribute:

```c
__attribute__((cpychecker_sets_exception))
```

which can be used to mark function declarations.

```c
extern void raise_error(const char *msg)
  __attribute__((cpychecker_sets_exception));
```

Given the above, the checker will know that an exception is set whenever a call to `raise_error()` occurs. It will also verify that `raise_error()` actually behaves this way when compiling the implementation of `raise_error`.

There is an analogous attribute for the case where a function returns a negative value to signify an error, where the exception state is set whenever a `negative` value is returned:
which can be used to mark function declarations.

```c
,__attribute__((cpychecker_negative_result_sets_exception))
```

Given the above, the checker will know that an exception is raised whenever a call to `foo` returns a negative value. It will also verify that `foo` actually behaves this way when compiling the implementation of `foo`.

The checker already knows about many of the functions within the CPython API which behave this way.

## 17.5 Format string checking

The checker will analyze some Python APIs that take format strings and detect mismatches between the number and types of arguments that are passed in, as compared with those described by the format string.

It currently verifies the arguments to the following API entrypoints:

- `PyArg_ParseTuple`
- `PyArg_ParseTupleAndKeywords`
- `PyArg_Parse`
- `Py_BuildValue`
- `PyObject_CallFunction`
- `PyObject_CallMethod`

along with the variants that occur if you define `PY_SSIZE_T_CLEAN` before `#include <Python.h>`.

For example, type mismatches between `int` vs `long` can lead to flaws when the code is compiled on big-endian 64-bit architectures, where `sizeof(int) != sizeof(long)` and the in-memory layout of those types differs from what you might expect.

The checker will also issue a warning if the list of keyword arguments in a call to `PyArg_ParseTupleAndKeywords` is not NULL-terminated.

**Note:** All of the various “#” codes in these format strings are affected by the presence of the macro `PY_SSIZE_T_CLEAN`. If the macro was defined before including Python.h, the various lengths for these format codes are of C type `Py_ssize_t` rather than `int`.

This behavior was clarified in the Python 3 version of the C API documentation, though the Python 2 version of the API docs leave the matter of which codes are affected somewhat ambiguous.

Nevertheless, the API does work this way in Python 2: all format codes with a “#” do work this way.

Internally, the C preprocessor converts such function calls into invocations of:
The checker handles this behavior correctly, by checking “#” codes in the regular functions against `int` and those in the modified functions against `Py_ssize_t`.

### 17.5.1 Associating PyTypeObject instances with compile-time types

The “O!” format code to `PyArg_ParseTuple` takes a `PyTypeObject` followed by the address of an object. This second argument can point to a `PyObject*`, but it can also point to a pointer to a derived class.

For example, CPython’s own implementation contains code like this:

```c
static PyObject *
unicodedata_decomposition(PyObject *self, PyObject *args)
{
    PyUnicodeObject *v;
    /* ...snip... */
    if (!PyArg_ParseTuple(args, "O!:decomposition",
                         &PyUnicode_Type, &v))
    /* ...etc... */
```

in which the input argument is written out into the `PyUnicodeObject*`, provided that it is indeed a unicode instance.

When the `cpychecker` verifies the types in this format string it verifies that the run-time type of the `PyTypeObject` matches the compile-time type (`PyUnicodeObject*`). It is able to do this since it contains hard-coded associations between these worlds for all of Python’s built-in types: for the above case, it “knows” that `PyUnicode_Type` is associated with `PyUnicodeObject`.

If you need to provide a similar association for an extension type, the checker provides a custom GCC attribute:

```c
__attribute__((cpychecker_type_object_for_typedef(typename)))
```

which can be used to mark `PyTypeObject` instance, giving the name of the typedef that `PyObject` instances of that type can be safely cast to.

```c
/* The checker automatically defines this preprocessor name when creating
   the custom attribute: */
#if defined(WITH_CPYCHECKER_TYPE_OBJECT_FOR_TYPEDEF_ATTRIBUTE)
    #define CPYCHECKER_TYPE_OBJECT_FOR_TYPEDEF(typename) __attribute__((cpychecker_type_object_for_typedef(typename)))
#else
    /* This handles the case where we're compiling with a "vanilla"
       compiler that doesn't supply this attribute: */
    #define CPYCHECKER_TYPE_OBJECT_FOR_TYPEDEF(typename)
#endif

/* Define some PyObject subclass, as both a struct and a typedef */
struct OurObjectStruct {
    PyObject_HEAD
    /* other fields */
};
```

(continues on next page)
typedef struct OurObjectStruct OurExtensionObject;

/*
  Declare the PyTypeObject, using the custom attribute to associate it with
  the typedef above:
*/
extern PyTypeObject UserDefinedExtension_Type
CPYCHECKER_TYPE_OBJECT_FOR_TYPEDEF("OurExtensionObject");

Given the above, the checker will associate the given PyTypeObject with the given typedef.

17.6 Verification of PyMethodDef tables

The checker will verify the types within tables of PyMethodDef initializers: the callbacks are typically cast to PyCFunction, but the exact type needs to correspond to the flags given. For example (METH_VARARGS | METH_KEYWORDS) implies a different function signature to the default, which the vanilla C compiler has no way of verifying.

```c
/*
  BUG: there's a mismatch between the signature of the callback and
  that implied by ml_flags below.
*/
static PyObject *widget_display(PyObject *self, PyObject *args);

static PyMethodDef widget_methods[] = {
    {"display",}
    {(PyCFunction)widget_display,
     (METH_VARARGS | METH_KEYWORDS), /* ml_flags */
     NULL},
    {NULL, NULL, 0, NULL} /* terminator */
};
```

Given the above, the checker will emit an error like this:

```
input.c:59:6: warning: flags do not match callback signature for 'widget_display'
   within PyMethodDef table
input.c:59:6: note: expected ml_meth callback of type "PyObject (fn)(someobject *,
   ->PyObject *args, PyObject *kwargs)" due to METH_KEYWORDS flag (3 arguments)
input.c:59:6: note: actual type of underlying callback: struct PyObject * <Tc53>->
   (struct PyObject *, struct PyObject *) (2 arguments)
input.c:59:6: note: see http://docs.python.org/c-api/structures.html#PyMethodDef
```

It will also warn about tables of PyMethodDef initializers that are lacking a NULL sentinel value to terminate the iteration:

```c
static PyMethodDef widget_methods[] = {
    {"display",}
    {(PyCFunction)widget_display,
     0, /* ml_flags */
     NULL},
    /* BUG: this array is missing a NULL value to terminate
       the list of methods, leading to a possible segfault */
};
```

(continues on next page)
at run-time */
}

Given the above, the checker will emit this warning:

```
input.c:39:6: warning: missing NULL sentinel value at end of PyMethodDef table
```

### 17.7 Additional tests

- the checker will verify the argument lists of invocations of `PyObject_CallFunctionObjArgs` and `PyObject_CallMethodObjArgs`, checking that all of the arguments are of the correct type (`PyObject*` or subclasses), and that the list is NULL-terminated:

```
input.c: In function 'test':
input.c:33:5: warning: argument 2 had type char[12] * but was expecting a
PyObject* (or subclass)
input.c:33:5: warning: arguments to PyObject_CallFunctionObjArgs were not NULL-terminated
```

### 17.8 Limitations and caveats

Compiling with the checker is significantly slower than with “vanilla” gcc. I have been focussing on correctness and features, rather than optimization. I hope that it will be possible to greatly speed up the checker via ahead-of-time compilation of the Python code (e.g. using Cython).

The checker does not yet fully implement all of C: expect to see Python tracebacks when it encounters less common parts of the language. (We’ll fix those bugs as we come to them)

The checker has a rather simplistic way of tracking the flow through a function: it builds a tree of all possible traces of execution through a function. This brings with it some shortcomings:

- In order to guarantee that the analysis terminates, the checker will only track the first time through any loop, and stop analysing that trace for subsequent iterations. This appears to be good enough for detecting many kinds of reference leaks, especially in simple wrapper code, but is clearly suboptimal.

- In order to avoid combinatorial explosion, the checker will stop analyzing a function once the trace tree gets sufficiently large. When it reaches this cutoff, a warning is issued:

```
input.c: In function 'add_module_objects':
input.c:31:1: note: this function is too complicated for the reference-count
checker to analyze
```

To increase this limit, see the `--maxtrans` option.

- The checker doesn’t yet match up similar traces, and so a single bug that affects multiple traces in the trace tree can lead to duplicate error reports.

Only a subset of the CPython API has been modelled so far. The functions known to the checker are:

- `PyArg_Parse` and `_PyArg_Parse_SizeT`,
- `PyArg_ParseTuple` and `_PyArg_ParseTuple_SizeT`,
- `PyArg_ParseTupleAndKeywords` and `_PyArg_ParseTupleAndKeywords_SizeT`,
- `PyArg_UnpackTuple`, `Py_AtExit`,
- `PyBool_FromLong`, `Py_BuildValue` and `_Py_BuildValue_SizeT`,
- `PyObject_Call`, `PyObject_CallList`, `PyObject_CallMethod`,
- `Py_Delmethod`, `PyDict_GetItem`,
- `PyCObject_AsVoidPtr`, `PyCObject_FromVoidPtr`, `PyCObject_FromVoidPtrAndDesc`,
- `PyCode_New`, `PyDict_GetItem`,
- `PyGLenum_to_char`, `PyGLenum_to_int`, `PyGLenum_to_long`, `PyGLenum_to_ptr`,
- `PyCObject_AsVoidPtr`, `PyCObject_FromVoidPtr`, `PyCObject_FromVoidPtrAndDesc`,
- `PyCode_New`, `PyDict_GetItem`,
17.9 Ideas for future tests

Here's a list of some other C coding bugs I intend for the tool to detect:

- `tp_traverse` errors (which can mess up the garbage collector); missing it altogether, or omitting fields
- Errors in GIL-handling
  - lock/release mismatches
  - missed opportunities to release the GIL (e.g. compute-intensive functions; functions that wait on IO/syscalls)

Ideas for other tests are most welcome (patches even more so!)

We will probably need various fallbacks and suppression modes for turning off individual tests (perhaps pragmas, perhaps compile-line flags, etc)

17.10 Reusing this code for other projects

It may be possible to reuse the analysis engine from cpychecker for other kinds of analysis - hopefully the python-specific parts are relatively self-contained. Email the gcc-python-plugin’s mailing list if you’re interested in adding verifiers for other kinds of code.
### 17.11 Common mistakes

Here are some common mistakes made using the CPython extension API, along with the fixes.

#### 17.11.1 Missing `Py_INCREF()` on `Py_None`

The following is typically incorrect: a method implementation is required to return a new reference, but this code isn’t incrementing the reference count on `Py_None`.

```c
PyObject*
some_method(PyObject *self, PyObject *args)
{
    [...snip...]

    /* BUG: loses a reference to Py_None */
    return Py_None;
}
```

If called enough, this could cause `Py_None` to be deallocated, crashing the interpreter:

```
Fatal error: deallocating None
```

The `Py_RETURN_NONE` macro takes care of incrementing the reference count for you:

```c
PyObject*
some_method(PyObject *self, PyObject *args)
{
    [...snip...]

    /* Fixed version of the above: */
    Py_RETURN_NONE;
}
```

#### 17.11.2 Reference leak in `Py_BuildValue`

`Py_BuildValue` with “O” adds a new reference on the object for use by the new tuple, hence the following code leaks the reference already owned on the object:

```c
/* BUG: reference leak: */
return Py_BuildValue("O", some_object_we_own_a_ref_on);
```

`Py_BuildValue` with “N” steals the reference (and copes with it being NULL by propagating the exception):

```c
/* Fixed version of the above: */
return Py_BuildValue("N", some_object_we_own_a_ref_on);
```
If you use the gcc python plugin to improve your code, we’d love to hear about it. If you want to share a success story here, please email the plugin’s mailing list.

18.1 The GNU Debugger

Bugs found in gdb by compiling it with the plugin’s gcc-with-cpychecker script:

- http://sourceware.org/bugzilla/show_bug.cgi?id=13308
- http://sourceware.org/bugzilla/show_bug.cgi?id=13309
- http://sourceware.org/bugzilla/show_bug.cgi?id=13310
- http://sourceware.org/bugzilla/show_bug.cgi?id=13316
- http://sourceware.org/bugzilla/show_bug.cgi?id=13331

Tom Tromey also wrote specialized Python scripts to use the GCC plugin to locate bugs within GDB. One of his scripts analyzes gdb’s resource-management code, which found some resource leaks and a possible crasher:


The other generates a whole-program call-graph, annotated with information on gdb’s own exception-handling mechanism. A script then finds places where these exceptions were not properly integrated with gdb’s embedded Python support:

- http://sourceware.org/bugzilla/show_bug.cgi?id=13369
18.2 LibreOffice

Stephan Bergmann wrote a script to analyze LibreOffice’s source code, detecting a particular usage pattern of C++ method calls:

- https://bugs.freedesktop.org/show_bug.cgi?id=43460

18.3 psycopg

Daniele Varrazzo used the plugin’s gcc-with-cpychecker script on psycopg, the popular Python interface to PostgreSQL, and was able to find and fix numerous subtle errors:

- http://initd.org/psycopg/articles/2012/03/29/psycopg-245-released/

18.4 pycups

Bugs found in the Python bindings for the CUPS API by compiling it with the plugin’s gcc-with-cpychecker script:

- https://fedorahosted.org/pycups/ticket/17

18.5 python-krbV

Bug found in the Python bindings for the Kerberos 5 API by compiling it with the plugin’s gcc-with-cpychecker script:

- https://fedorahosted.org/python-krbV/ticket/1

18.6 Bugs found in itself

Bugs found and fixed in the gcc Python plugin itself, by running the the plugin’s gcc-with-cpychecker script when compiling another copy:

- various reference counting errors:
  - http://git.fedorahosted.org/git/?p=gcc-python-plugin.git;a=commitdiff;h=a9f48fac24a66c77007d99bf23f2eab188eb909e
  - http://git.fedorahosted.org/git/?p=gcc-python-plugin.git;a=commitdiff;h=2922ad81c8e0ea954d462433ecc83d86d9ebab68
  - http://git.fedorahosted.org/git/?p=gcc-python-plugin.git;a=commitdiff;h=4642a564e03c9e2c8114bca206205ad9c8fbc308>


- minor const-correctness error: http://git.fedorahosted.org/git/?p=gcc-python-plugin.git;a=commitdiff;h=4fe4a83288e04be35a96d0bfec332197fb32c358
The plugin’s web site is this GitHub repository:

https://github.com/davidmalcolm/gcc-python-plugin

The primary place for discussion of the plugin is the mailing list: https://fedorahosted.org/mailman/listinfo/gcc-python-plugin

A pre-built version of the HTML documentation can be seen at:


The project’s mailing list is here: https://fedorahosted.org/mailman/listinfo/gcc-python-plugin

19.1 Ideas for using the plugin

Here are some ideas for possible uses of the plugin. Please email the plugin’s mailing list if you get any of these working (or if you have other ideas!). Some guesses as to the usefulness and difficulty level are given in parentheses after some of the ideas. Some of them might require new attributes, methods and/or classes to be added to the plugin (to expose more of GCC internals), but you can always ask on the mailing list if you need help.

- extend the libcpychcker code to add checking for the standard C library. For example, given this buggy C code:

```c
int foo() {
    FILE *src, *dst;
    src = fopen("source.txt", "r");
    if (!src) return -1;

    dst = fopen("dest.txt", "w");
    if (!dst) return -1;  /* <<< BUG: this error-handling leaks "src" */
    /* etc, copy src to dst (or whatever) */
}
```

it would be great if the checker could emit a compile-time warning about the buggy error-handling path above (or indeed any paths through functions that leak FILE*, file descriptors, or other resources). The way to do this
(I think) is to add a new Facet subclass to libcpychecker, analogous to the CPython facet subclass that already exists (though the facet handling is probably rather messy right now). (useful but difficult, and a lot of work)

- extend the libcpychecker code to add checking for other libraries. For example:
  - reference-count checking within glib and gobject
    (useful for commonly-used C libraries but difficult, and a lot of work)
- detection of C++ variables with non-trivial constructors that will need to be run before main - globals and static locals (useful, ought to be fairly easy)
- finding unused parameters in definitions of non-virtual functions, so that they can be removed - possibly removing further dead code. Some care would be needed for function pointers. (useful, ought to be fairly easy)
- detection of bad format strings (see e.g. https://lwn.net/Articles/478139/)
- compile gcc’s own test suite with the cpychecker code, to reuse their coverage of C and thus shake out more bugs in the checker (useful and easy)
- a new PyPy gc root finder, running inside GCC (useful for PyPy, but difficult)
- reimplement GCC-XML in Python (probably fairly easy, but does anyone still use GCC-XML now that GCC supports plugins?)
- .gir generation for GObject Introspection (unknown if the GNOME developers are actually interested in this though)
- create an interface that lets you view the changing internal representation of each function as it’s modified by the various optimization passes: lets you see which passes change a given function, and what the changes are (might be useful as a teaching tool, and for understanding GCC)
- add array bounds checking to C (to what extent can GCC already do this?)
- taint mode for GCC! e.g. detect usage of data from network/from disk/etc; identify certain data as untrusted, and track how it gets used; issue a warning (very useful, but very difficult: how does untainting work? what about pointers and memory regions? is it just too low-level?)
- implement something akin to PyPy’s pygame-based viewer, for viewing control flow graphs and tree structures: an OpenGL-based GUI giving a fast, responsive UI for navigating the data - zooming, panning, search, etc. (very useful, and fairly easy)
- generation of pxd files for Cython (useful for Cython, ought to be fairly easy)
- reverse-engineering a .py or .pyx file from a .c file: turning legacy C Python extension modules back into Python or Cython sources (useful but difficult)

### 19.2 Tour of the C code

The plugin’s C code heavily uses Python’s extension API, and so it’s worth knowing this API if you’re going to hack on this part of the project. A good tutorial for this can be seen here:

http://docs.python.org/extending/index.html

and detailed notes on it are here:

http://docs.python.org/c-api/index.html

Most of the C “glue” for creating classes and registering their methods and attributes is autogenerated. Simple C one-liners tend to appear in the autogenerated C files, whereas longer implementations are broken out into a hand-written C file.
Adding new methods and attributes to the classes requires editing the appropriate generate-* .py script to wire up the new entrypoint. For very simple attributes you can embed the C code directly there, but anything that’s more than a one-liner should have its implementation in the relevant C file.

For example, to add new methods to a gcc.Cfg you’d edit:

- \texttt{generate-cfg-c.py} to add the new methods and attributes to the relevant tables of callbacks
- \texttt{gcc-python-wrappers.h} to add declarations of the new C functions
- \texttt{gcc-python-cfg.c} to add the implementations of the new C functions

Please try to make the API “Pythonic”.

My preference with getters is that if the implementation is a simple field lookup, it should be an attribute (the “getter” is only implicit, existing at the C level):

\begin{verbatim}
    print(bb.loopcount)
\end{verbatim}

whereas if getting the result involves some work, it should be an explicit method of the class (where the “getter” is explicit at the Python level):

\begin{verbatim}
    print(bb.get_loop_count())
\end{verbatim}

### 19.3 Using the plugin to check itself

Given that the \texttt{cpychecker} code implements new error-checking for Python C code, and that the underlying plugin is itself an example of such code, it’s possible to build the plugin once, then compile it with itself (using \texttt{CC=gcc-with-cpychecker} as a Makefile variable:

\begin{verbatim}
$ make CC=/path/to/a/clean/build/of/the/plugin/gcc-with-cpychecker
\end{verbatim}

Unfortunately it doesn’t quite compile itself cleanly right now.

### 19.4 Test suite

There are three test suites:

- \texttt{testcpybuilder.py}: a minimal test suite which is used before the plugin itself is built. This verifies that the \texttt{cpybuilder} code works.
- \texttt{make test-suite} (aka \texttt{run-test-suite.py}): a test harness and suite which was written for this project. See the notes below on patches.
- \texttt{make testcpychecker} and \texttt{testcpychecker.py}: a suite based on Python’s \texttt{unittest} module

### 19.5 Debugging the plugin’s C code

The \texttt{gcc} binary is a harness that launches subprocesses, so it can be fiddly to debug. Exactly what it launches depend on the inputs and options. Typically, the subprocesses it launches are (in order):

- \texttt{cc1} or \texttt{cc1plus}: The C or C++ compiler, generating a .s assembler file.
- \texttt{as}: The assembler, converting a .s assembler file to a .o object file.
• *collect2*: The linker, turning one or more .o files into an executable (if you’re going all the way to building an a.out-style executable).

The easiest way to debug the plugin is to add these parameters to the gcc command line (e.g. to the end):

```
-wraper gdb,--args
```

Note the lack of space between the comma and the --args.

e.g.:

```
./gcc-with-python examples/show-docs.py test.c -wraper gdb,--args
```

This will invoke each of the subprocesses in turn under gdb: e.g. cc1, as and collect2; the plugin runs with cc1 (cc1plus for C++ code).

For example:

```
$ ./gcc-with-cpychecker -c -I/usr/include/python2.7 demo.c -wraper gdb,--args
```

Another way to do it is to add “-v” to the gcc command line (verbose), so that it outputs the commands that it’s running. You can then use this to launch:

```
$ gdb --args ACTUAL PROGRAM WITH ACTUAL ARGS
```

to debug the subprocess that actually loads the Python plugin.

For example:

```
$ gcc -v -fplugin=$(pwd)/python.so -fplugin-arg-python-script=test.py test.c
```

on my machine emits this:

```
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/libexec/gcc/x86_64-redhat-linux/4.6.1/lto-wrapper
Target: x86_64-redhat-linux
Thread model: posix
```

```
```

(continues on next page)
This allows us to see the line in which `cc1` is invoked: in the above example, it’s the final line before the output from the script:

```
```

We can then take this line and rerun this subprocess under gdb by adding `gdb --args` to the front like this:

```
```

This approach to obtaining a debuggable process doesn’t seem to work in the presence of `ccache`, in that it writes to a temporary directory with a name that embeds the process ID each time, which then gets deleted. I’ve worked around this by uninstalling ccache, but apparently setting:

```
CCACHE_DISABLE=1
```

before invoking `gcc -v` ought to also work around this.

I’ve also been running into this error from gdb:

```
[Thread debugging using libthread_db enabled] Cannot find new threads: generic error
```

Apparently this happens when debugging a process that uses dlopen to load a library that pulls in libpthread (as does gcc when loading in my plugin), and a workaround is to link cc1 with -lpthread

The workaround I’ve been using (to avoid the need to build my own gcc) is to use `LD_PRELOAD`, either like this:

```
LD_PRELOAD=libpthread.so.0 gdb --args ARGs GO HERE...
```

or this:

```
(gdb) set environment LD_PRELOAD libpthread.so.0
```

### 19.5.1 Handy tricks

Given a (PyGccTree*) named “self”:

```
(gdb) call debug_tree(self->t)
```

will use GCC’s prettyprinter to dump the embedded (tree*) and its descendants to stderr; it can help to put a breakpoint on that function too, to explore the insides of that type.

### 19.5. Debugging the plugin’s C code

`gcc-python-plugin Documentation, Release 0.16`
19.6 Patches

The project doesn’t have any copyright assignment requirement: you get to keep copyright in any contributions you make, though AIUI there’s an implicit licensing of such contributions under the GPLv3 or later, given that any contribution is a derived work of the plugin, which is itself licensed under the GPLv3 or later. I’m not a lawyer, though.

The Python code within the project is intended to be usable with both Python 2 and Python 3 without running 2to3: please stick to the common subset of the two languages. For example, please write print statements using parentheses:

```
print(42)
```

Under Python 2 this is a *print* statement with a parenthesized number: (42) whereas under Python 3 this is an invocation of the *print* function.

Please try to stick PEP-8 for Python code, and to PEP-7 for C code (rather than the GNU coding conventions).

In C code, I strongly prefer to use multiline blocks throughout, even where single statements are allowed (e.g. in an “if” statement):

```
if (foo()) {
    bar();
}
```

as opposed to:

```
if (foo())
    bar();
```

since this practice prevents introducing bugs when modifying such code, and the resulting “diff” is much cleaner.

A good patch ought to add test cases for the new code that you write, and documentation.

The test cases should be grouped in appropriate subdirectories of “tests”. Each new test case is a directory with an:

- *input.c* (or *input.cc* for C++)
- *script.py* exercising the relevant Python code
- *stdout.txt* containing the expected output from the script.

For more realistic examples of test code, put them below *tests/examples*; these can be included by reference from the docs, so that we have documentation that’s automatically verified by *run-test-suite.py*, and users can use this to see the relationship between source-code constructs and the corresponding Python objects.

More information can be seen in *run-test-suite.py*

By default, *run-test-suite.py* will invoke all the tests. You can pass it a list of paths and it run all tests found in those paths and below.

You can generate the “gold” stdout.txt by hacking up this line in *run-test-suite.py*:

```
out.check_for_diff(out.actual, err.actual, p, args, 'stdout', 0)
```

so that the final 0 is a 1 (the “writeback” argument to *check_for_diff*). There may need to be a non-empty stdout.txt file in the directory for this to take effect though.

Unfortunately, this approach over-specifies the selftests, making them rather “brittle”. Improvements to this approach would be welcome.

To directly see the GCC command line being invoked for each test, and to see the resulting stdout and stderr, add --show to the arguments of *run-test-suite.py*. 

104 Chapter 19. Getting Involved
For example:

```bash
$ python run-test-suite.py tests/plugin/diagnostics --show

tests/plugin/diagnostics: gcc -c -o tests/plugin/diagnostics/output.o -fplugin=/home/david/coding/gcc-python-plugin/python.so -fplugin-arg-python-script=tests/plugin/diagnostics/script.py -Wno-format tests/plugin/diagnostics/input.c
tests/plugin/diagnostics/input.c: In function 'main':
tests/plugin/diagnostics/input.c:23:1: error: this is an error (with positional args)
tests/plugin/diagnostics/input.c:23:1: error: this is an error (with keyword args)
tests/plugin/diagnostics/input.c:25:1: warning: this is a warning (with positional args) [-Wdiv-by-zero]
tests/plugin/diagnostics/input.c:25:1: warning: this is a warning (with keyword args) [-Wdiv-by-zero]
tests/plugin/diagnostics/input.c:23:1: error: a warning with some embedded format strings %s and %i

tests/plugin/diagnostics/input.c:25:1: warning: this is an unconditional warning, [-Wdiv-by-zero]
tests/plugin/diagnostics/input.c:25:1: warning: this is another unconditional warning, [-Wdiv-by-zero]
expected error was found: option must be either None, or of type gcc.Option

tests/plugin/diagnostics/input.c:23:1: note: This is the start of the function
tests/plugin/diagnostics/input.c:23:1: note: This is the end of the function
OK
1 success; 0 failures; 0 skipped
```
We use Sphinx for documentation, which makes it easy to keep the documentation up-to-date. For notes on how to document Python in the .rst form accepted by Sphinx, see e.g.:

http://sphinx.pocoo.org/domains.html#the-python-domain
The following odds and ends cover the more esoteric aspects of GCC, and are documented here for completeness. They may or may not be useful when writing scripts.

## 21.1 Interprocedural analysis (IPA)

GCC builds a “call graph”, recording which functions call which other functions, and it uses this for various optimizations.

It is constructed by the “*build_cgraph_edges*” pass.

In case it’s of interest, it is available via the following Python API:

```python
gcc.get_callgraph_nodes()
    Get a list of all `gcc.CallgraphNode` instances

gcutils.callgraph_to_dot()
    Return the GraphViz source for a rendering of the current callgraph, as a string.

    Here’s an example of such a rendering:

```python
class gcc.CallgraphNode

  decl
      The `gcc.FunctionDecl` for this node within the callgraph

  callees
      The function calls made by this function, as a list of `gcc.CallgraphEdge` instances

  callers
      The places that call this function, as a list of `gcc.CallgraphEdge` instances

    Internally, this wraps a `struct cgraph_node`

```python
class gcc.CallgraphEdge

```
The function that makes this call, as a \texttt{gcc.CallgraphNode}

- \texttt{callee}
  The function that is called here, as a \texttt{gcc.CallgraphNode}

- \texttt{call_stmt}
  The \texttt{gcc.GimpleCall} statement for the function call

Internally, this wraps a \texttt{struct cgraph_edge *}

\section*{21.2 Whole-program Analysis via Link-Time Optimization (LTO)}

You can enable GCC’s “link time optimization” feature by passing \texttt{-flto}.

When this is enabled, gcc adds extra sections to the compiled .o file containing the SSA-Gimple internal representation of every function, so that this SSA representation is available at link-time. This allows gcc to inline functions defined in one source file into functions defined in another source file at link time.

Although the feature is intended for optimization, we can also use it for code analysis, and it’s possible to run the Python plugin at link time.

This means we can do interprocedural analysis across multiple source files.

\textbf{Warning:} Running a gcc plugin from inside link-time optimization is rather novel, and you’re more likely to run into bugs. See e.g. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=54962

An invocation might look like this:

\begin{verbatim}
gcc \ -flto \\
\end{verbatim}

(continues on next page)
Looking at the above options in turn:

- `-flto` enables link-time optimization
- `-flto-partition=none` : by default, gcc with LTO partitions the code and generates summary information for each partition, then combines the results of the summaries (known as “WPA” and “LTRANS” respectively). This appears to be of use for optimization, but to get at the function bodies, for static analysis, you should pass this option, which instead gathers all the code into one process.
- `-v` means “verbose” and is useful for seeing all of the subprograms that gcc invokes, along with their command line options. Given the above options, you should see invocations of `cc1` (the C compiler), `collect2` (the linker) and `lto1` (the link-time optimizer).

For example,

```
$ ./gcc-with-python
  examples/show-lto-supergraph.py
  -flto
  -flto-partition=none
  tests/examples/lto/input-*.c
```

will render a bitmap of the supergraph like this:

```
gcc.is_lto()

    Return type  bool
    Determine whether or not we’re being invoked during link-time optimization (i.e. from within the lto1 program)

    Warning: The underlying boolean is not set up until passes are being invoked: it is always False during the initial invocation of the Python script.
```

### 21.3 Inspecting GCC’s command-line options

GCC’s command-line options are visible from Python scripts as instances of `gcc.Option`.

```
class gcc.Option
    Wrapper around one of GCC’s command-line options.
    You can locate a specific option using its text attribute:
    
    option = gcc.Option('-Wdiv-by-zero')
```

The plugin will raise a `ValueError` if the option is not recognized.

It does not appear to be possible to create new options from the plugin.
**text**
(string) The text used at the command-line to affect this option e.g. `-Werror`.

**help**
(string) The help text for this option (e.g. “Warn about uninitialized automatic variables”)

**is_enabled**
(bool) Is this option enabled?

**is_driver**
(bool) Is this a driver option?

**is_optimization**
(bool) Does this option control an optimization?

**is_target**
(bool) Is this a target-specific option?

**is_warning**
(bool) Does this option control a warning message?

Internally, the class wraps GCC’s enum opt_code (and thus a struct cl_option)

**gcc.get_option_list()**
Returns a list of all gcc.Option instances.

**gcc.get_option_dict()**
Returns a dictionary, mapping from the option names to gcc.Option instances.

### 21.4 Working with GCC’s tunable parameters

GCC has numerous tunable parameters, which are integer values, tweakable at the command-line by:

```bash
--param <name>=<value>
```

A detailed description of the current parameters (in GCC 4.6.0) can be seen at http://gcc.gnu.org/onlinedocs/gcc-4.6.0/gcc/Optimize-Options.html#Optimize-Options (search for “–param” on that page; there doesn’t seem to be an anchor to the list)

The parameters are visible from Python scripts using the following API:

**gcc.get_parameters()**
Returns a dictionary, mapping from the option names to gcc.Parameter instances

**class gcc.Parameter**

**option**
(string) The name used with the command-line --param switch to set this value
current_value
    (int/long)
default_value
    (int/long)
min_value
    (int/long) The minimum acceptable value
max_value
    (int/long) The maximum acceptable value, if greater than min_value
help
    (string) A short description of the option.

21.5 Working with the preprocessor

For languages that support a preprocessor, it’s possible to inject new “built-in” macros into the compilation from a Python script.

The motivation for this is to better support the creation of custom attributes, by creating preprocessor names that can be tested against.

```
gcc.define_macro(\"argument\")
```

Defines a preprocessor macro with the given argument, which may be of use for code that needs to test for the presence of your script. The argument can either be a simple name, or a name with a definition:

```
gcc.define_macro(\"SOMETHING\")   # define as the empty string
gcc.define_macro(\"SOMETHING=72\")
```

This function can only be called from within specific event callbacks, since it manipulates the state of the preprocessor for a given source file.

For now, only call it in a handler for the event `gcc.PLUGIN_ATTRIBUTES`:

```
import gcc

def attribute_callback_for_claims_mutex(*args):
    print('attribute_callback_for_claims_mutex called: args: %s' % (args, ))

def attribute_callback_for_releases_mutex(*args):
    print('attribute_callback_for_releases_mutex called: args: %s' % (args, ))

def register_our_attributes():
    gcc.register_attribute('claims_mutex',
                           1, 1,
                           False, False, False,
                           attribute_callback_for_claims_mutex)
    gcc.define_macro('WITH_ATTRIBUTE_CLAIMS_MUTEX')

gcc.register_attribute('releases_mutex',
                       1, 1,
                       False, False, False,
                       attribute_callback_for_releases_mutex)

gcc.define_macro('WITH_ATTRIBUTE_RELEASES_MUTEX')

# Wire up our callback:
```

(continues on next page)
21.6 Version handling

gcc.get_gcc_version()
Get the gcc.Version for this version of GCC

gcc.get_plugin_gcc_version()
Get the gcc.Version that this plugin was compiled with

Typically the above will be equal (the plugin-loading mechanism currently checks for this, and won’t load the plugin otherwise).

On my machine, running this currently gives:

gcc.Version(basever='4.6.0', datestamp='20110321', devphase='Red Hat 4.6.0-0.15',
→revision='', ...)

class gcc.Version
Information on the version of GCC being run. The various fields are accessible by name and by index.

  basever
    (string) On my machine, this has value:

    '4.6.0'

  datestamp
    (string) On my machine, this has value:

    '20110321'

  devphase
    (string) On my machine, this has value:

    'Red Hat 4.6.0-0.15'

  revision
    (string) On my machine, this is the empty string

  configuration_arguments
    (string) On my machine, this has value:

    '/../configure --prefix=/usr --mandir=/usr/share/man --infodir=/usr/share/info_
      →--with-bugurl=http://bugzilla.redhat.com/bugzilla --enable-bootstrap --
      →enable-shared --enable-threads=posix --enable-checking=release --with-
      →system-zlib --enable-__cxa_atexit --disable-libunwind-exceptions --enable-
      →gnu-unique-object --enable-linker-build-id --enable-languages=c,c++,objc,
      →obj-c++, java, fortran, ada, go, lto --enable-plugin --enable-java-awt=gtk --
      →disable-dssi --with-java-home=/usr/lib/jvm/java-1.5.0-gcj-1.5.0.0/jre --
      →enable-libgcj-multifile --enable-java-maintainer-mode --with-ecj-jar=/usr/
      →share/java/eclipse-ecj.jar --disable-libjava-multilib --with-ppl --with-
      →cloog --with-tune=generic --with-arch_32=i686 --build=x86_64-redhat-linux'

Internally, this is a wrapper around a struct plugin_gcc_version
gcc\_GCC\_VERSION
(int) This corresponds to the value of GCC\_VERSION within GCC’s internal code: (MAJOR * 1000) + MINOR:

<table>
<thead>
<tr>
<th>GCC version</th>
<th>Value of gcc.GCC_VERSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.6</td>
<td>4006</td>
</tr>
<tr>
<td>4.7</td>
<td>4007</td>
</tr>
<tr>
<td>4.8</td>
<td>4008</td>
</tr>
<tr>
<td>4.9</td>
<td>4009</td>
</tr>
</tbody>
</table>

21.7 Register Transfer Language (RTL)

class gcc.Rtl

A wrapper around GCC’s struct rtx\_def type: an expression within GCC’s Register Transfer Language

loc

The gcc.Location of this expression, or None

operands

The operands of this expression, as a tuple. The precise type of the operands will vary by subclass.

There are numerous subclasses. However, this part of the API is much less polished than the rest of the plugin.
22.1 0.16

This releases adds support for gcc 7 and gcc 8 (along with continued support for gcc 4.6, 4.7, 4.8, 4.9, 5 and 6).

The upstream location for the plugin has moved from fedorahosted.org to https://github.com/davidmalcolm/gcc-python-plugin

Additionally, this release contains the following improvements:

- add `gcc.RichLocation` for GCC 6 onwards
- `gcc.Location`
  - add `caret, start, finish` attributes for GCC 7 onwards
  - add `gcc.Location.offset_column()` method

22.2 0.15

This releases adds support for gcc 6 (along with continued support for gcc 4.6, 4.7, 4.8, 4.9 and 5).

Additionally, this release contains the following improvements (contributed by Tom Tromey; thanks Tom):

- document `gcc.PLUGIN_FINISH_TYPE`
- document `gcc.EnumerateType`; add ‘values’ attribute
- add unqualified_xpath equivalent to gcc.Type subclasses
- preserve qualifiers when adding more qualifiers
- fix include for gcc 4.9.2
- handle variadic function types
This releases adds support for gcc 5 (along with continued support for gcc 4.6, 4.7, 4.8 and 4.9).

The major features in this release are:

- gcc 4.9 compatibility
- a major revamping to the HTML output from gcc-with-cpychecker

New dependency: lxml. The new HTML output format uses lxml internally.

### 22.4.1 Changes to the GCC Python Plugin

#### GCC 4.9 compatibility

This release of the plugin adds support for gcc 4.9 (along with continued support for gcc 4.6, 4.7 and gcc 4.8). Building against 4.9 requires a GCC 4.9 with the fix for GCC bug 63410 applied.

#### Other fixes

- fixed a build-time incompatibility with Python 3.3.0
- various internal bug fixes:
  - bug in garbage-collector integration ([https://bugzilla.redhat.com/show_bug.cgi?id=864314](https://bugzilla.redhat.com/show_bug.cgi?id=864314))
  - the test suite is now parallelized (using multiprocess)
- improvements to Makefile
- improvements to documentation
- add gcc.Location.in_system_header attribute

### 22.4.2 Improvements to gcc-with-cpychecker

The major improvement to gcc-with-cpychecker is a big revamp of the output.

A new “v2” HTML report is available, written to `SOURCE_NAME.v2.html` e.g. `demo.c.v2.html`:

The new HTML report is easier to read in the presence of complicated control flow. It also include links to the API documentation for calls made to the CPython API.

For both old-style and new-style reports, the wording of the messages has been clarified:

- Reference-count tracking messages now largely eliminate the `0 + N` where `N >= gobbledegook`, since this was confusing to everyone (including me). Instead, error reports talk about references as owned vs borrowed references e.g.
  - “refs: 1 owned”
  - “refs: 0 owned 1 borrowed”
  resorting to ranges:
only where necessary.

- Reports now add memory leak: and future use-after-free: prefixes where appropriate, to better indicate the issue.
- Objects are referred to more in terms the user is likely to understand e.g. *dictA rather than PyDictObject. The checker also reports better source locations in its messages e.g. in the presence of multiple return statements (https://fedorahosted.org/gcc-python-plugin/ticket/58).

Other improvements

- Add a new test script: tests/examples/find-global-state, showing examples of finding global state in the code being compiled.
- handle PySequence_DelItem()
- fix bug in handling of PyRun_SimpleStringFlags()
- fix issue with handling of PyArg_ParseTuple() (https://fedorahosted.org/gcc-python-plugin/ticket/50)
- although we don’t model the internals of C++ exceptions, fix things so we don’t crash with a traceback in the absence of -fno-exceptions (https://fedorahosted.org/gcc-python-plugin/ticket/51)

22.4.3 Contributors

Thanks to Buck Golemon, Denis Efremov, Philip Herron, and Tom Tromey for their contributions to this release.
22.5 0.12

22.5.1 Changes to the GCC Python Plugin

GCC 4.8 compatibility

This release of the plugin adds support for gcc 4.8 (along with continued support for gcc 4.7 and gcc 4.6).

gcc-c-api

The source tree contains a new component: gcc-c-api. This provides a wrapper library libgcc-c-api.so that hides much of the details of GCC’s internals (such as the binary layout of structures, and the differences between GCC 4.6 through 4.8).

I plan for this to eventually be its own project, aiming at providing a stable API and ABI for working with GCC, once it has proven itself in the context of the python plugin.

The API provides an XML description of itself, which should greatly simplify the job of generating bindings for accessing GCC internals from other languages.

Link-Time Optimization support

The plugin can now be used with GCC’s Link-Time Optimization feature (LTO), allowing whole-program visualizations and analysis.

For example, you can rendering a whole-program “supergraph” of control flow graphs using this invocation:

```
$ ./gcc-with-python \\
  examples/show-lto-supergraph.py \\
  -flto \\
  -flto-partition=none \\
  tests/examples/lto/input-*.c
```

which will render a bitmap of the supergraph like this:

API improvements

Sane repr() implementations have been added to the following classes: gcc.CaseLabelExpr gcc.GimpleLabel gcc.BasicBlock gcc.SsaName gcc.ArrayRef gcc.ComponentRef gcc.PointerType gcc.IntegerType gcc.Location

gcc.Location instances can now be compared and sorted. They are ordered alphabetically by file, then by line number, then by column)

Other fixes

- the Makefile has a “make install” target (at last)
- prevent forkbomb when running with CC=gcc-with-cpychecker
- fixed memory leak within gcc.Gimple.walk_tree()
• ensure that the result of \texttt{gcc.Cfg.basic_blocks} can’t contain any \texttt{None} items (which used to sometimes happen when certain optimizations had occurred).
• run-test-suite.py now has a \texttt{--show} option, giving more verbose information on what the test suite is doing
• fix hashing and equality for \texttt{gcc.Function} and \texttt{gcc.Gimple}
• fix \texttt{gcc.IntegerCst.__hash__()} and ensure it compares sanely against \texttt{int}
• ensure that equivalent \texttt{gcc.ComponentRef} objects have the same hash and are equal
• ensure there is a unique \texttt{gcc.CallgraphEdge} for each underlying edge, and a unique \texttt{gcc.Cfg} for each underlying control flow graph
• add a “label” attribute to \texttt{gcc.GimpleLabel}
• add \texttt{gcc.GCC_VERSION}

22.5.2 Internal improvements to gcc-with-cpychecker

• fix exception on pointer comparisons
• fix exception on int-to-float casts
• fix traceback when analyzing a callsite that discards the LHS when an \texttt{Outcome.return()} a value
• fix two different exceptions when casting an integer value to a pointer
• add example of refcounting bugs to “make demo”
• fix a traceback seen on bogus uses of \texttt{Py_XDECREF()}

22.6 0.11

22.6.1 Changes to the GCC Python Plugin

The main change in this release is support for compiling the plugin with a C++ compiler. Recent versions of GCC 4.7 are now built with C++ rather than C, meaning that plugins must also be built with C++ (since all of GCC’s internal symbols are name-mangled). This release fixes the plugin’s Makefile so that it autodetects whether the plugin needs to be built with a C or C++ compiler and (I hope) does the right thing automatically. I’ve also made the necessary changes to the C source code of the plugin so that it is compilable as either language.

This should enable the plugin to now be usable with recent builds of gcc 4.7.* (along with gcc 4.6).

The plugin doesn’t yet support gcc 4.8 prereleases.

Other fixes:
• there is now a unique \texttt{gcc.Edge} wrapper for each underlying edge in GCC’s control flow graphs, rather than the old erroneous behavior of having multiple identical duplicate wrappers.
• fixed missing documentation for \texttt{gcc.SsaName}, and \texttt{gcc.Edge}’s \texttt{true_value} and \texttt{false_value} flags

22.6.2 Internal improvements to gcc-with-cpychecker

The CPython static analysis code shipped with the plugin contains a detailed description of the behavior of the CPython API (e.g. which arguments will lead to a segfault if NULL, and why; the possible outcomes of a call and their impact on reference-counts; etc).
However, these descriptions were tightly bound to implementation details of the checker.

This release introduces a new internal API to the analyzer for describing the possible behaviors of CPython API entrypoints, in an attempt to decouple these descriptions from the checker, and ports many of the descriptions to using it.

These changes shouldn’t be visible to users of the checker, but should make future maintenance much easier.

22.7 0.10

Thanks to Buck Golemon, Daniele Varrazzo, David Narvaez, Eevee, Jason Mueller, Kevin Pyle, Matt Rice and Tom Tromey for their contributions to this release.

22.7.1 Changes to the GCC Python Plugin

- The plugin can now be used with Python 3.3 (fixing Unicode issues and dict-ordering assumptions).
- The plugin now exposes inline assembler to Python scripts via `gcc.GimpleAsm`.
- There is a new `gccutils.sorted_callgraph()` function to get the callgraph in topologically-sorted order.
- The test suite has been reworked to fix issues with checkouts on OS X case-insensitive filesystems.
- C++ support: support for locating the global namespace (aka “::”), for locating declarations and child namespaces within a namespace, and aliases.
- `gcc.Declaration` now has an `isbuiltin` attribute
- Numerous improvements to the plugin’s Makefile

22.7.2 Improvements to gcc-with-cpychecker

- By default, the refcount checker is now only run on code that includes `<Python.h>` (implemented by checking if the “PyObject” typedef exists).
  This greatly speeds up compilation of large projects for which the Python extension modules are only a small subset of the source tree.
- Added some custom attributes for marking functions that set an exception, either always, or when returning a negative value:

```c
__attribute__((cpychecker_negative_result_sets_exception))
__attribute__((cpychecker_sets_exception))
```

- Improve descriptions of ranges: rather than emitting descriptions with the rather vague “value”, such as:

```plaintext
when considering range: 1 <= value <= 0x7fffffff
```

instead try to embed a descriptive name for the value, such as:

```plaintext
when considering range: 1 <= n <= 0x7fffffff
```
Mass recompile of Fedora 17’s Python extension code

I ran the reference-count checker on all of the C/C++ Python extension modules in Fedora 17 and reported hundreds of genuine problems, many of which have been fixed.

In the process of doing this I found and fixed many problems in the checker itself. For example:

- the checker now understands GCC’s __builtin_expect, fixing various false reports about dereferencing NULL pointers when running the checker on Cython-generated code in python-lxml-2.3
- added descriptions of part of SWIG and Cython’s internal APIs to suppress some false positives seen with SWIG and Cython-generated code.
- tweak the refcount rules to fix some false positives where the checker erroneously considered the case of a deallocation by:

```c
Py_DECREF(obj);
```

where “obj” provably had other references not owned by the function being analyzed, and thus for the case where obj->ob_refcnt > 1 the deallocation could not happen.

The plugin also now has a triaging script which can examine all of the errors within a build and provide a report, showing all of them in prioritized categories.

The source tree now contains helper scripts for conducting such a mass recompile.

Pyscopg support

Daniele Varrazzo used the checker extensively on psycopg, the popular Python interface to PostgreSQL, and was able to find and fix numerous subtle errors:

- http://initd.org/psycopg/articles/2012/03/29/psycopg-245-released/

Experimental new error visualization

The checker can now dump its internal representation in JSON form, via a new --dump-json option, and an experimental new renderer can generate HTML from this. An example can be seen here:

http://fedorapeople.org/~dmalcolm/gcc-python-plugin/2012-03-19/example/example.html

This is still a work-in-progress

C++ support

The checker is now able to run on C++ code: support has been added for methods, references, “this”, destructors, the gcc.GimpleNop operation.

Coverage of the CPython API

The format code handling for Py_BeildValue was missing support for the following codes:

- ‘u’ and ‘u#’
- ‘f’ and ‘d’
- ‘D’
• ‘c’

In addition, the handling for ‘s#' and ‘z#' had a bug in which it erroneously expected an int* or Py_ssize_t*, rather than just a int or Py_ssize_t.

This release fixes these issues, and gives full coverage of all valid format codes for Py_BuildValue in Python 2.

This release adds heuristics for the behavior of the following CPython API entrypoints:

• PyCode_New
• PyCode_NewDictAndDesc
• PyDict_Size
• PyErr_Clear
• PyEval_CallMethod
• Py_FatalError
• PyFile_SoftSpace, PyFile_WriteObject, and PyFile_WriteString
• PyFloat_AsDouble and PyFloat_FromDouble
• PyFrame_New
• Py_GetVersion
• PyImport_AddModule
• PyInterp_GetNative
• PyInterp_GetNative
• PyNumber_Int, PyNumber_Remainder
• PyObject_CallObject, PyObject_GetAttr, PyObject_GetAttrString, PyObject_GetItem, PyObject_SetAttr, and PyObject_SetAttrString
• PyOS_snprintf
• PyString_InternFromString
• PySequence_Concat, PySequence_GetSlice, PySequence_SetItem, PySequence_Size
• PySys_GetObject
• PyTraceBack_Here
• PyTuple_GetItem
• PyUnicodeUCS4_DecodeUTF8
• PyWeakref_GetObject

along with various other bugfixes.

22.8 0.9

22.8.1 Changes to the GCC Python Plugin

The plugin now works with GCC 4.7 prereleases (ticket #21).

The plugin is now integrated with GCC’s garbage collector: Python wrapper objects keep their underlying GCC objects alive when GCC's garbage collector runs, preventing segfaults that could occur if the underlying objects were swept away from under us (ticket #1).
It’s now possible to attach Python callbacks to more GCC events: `gcc.PLUGIN_FINISH`, `gcc.PLUGIN_GGC_START`, `gcc.PLUGIN_GGC_MARKING`, `gcc.PLUGIN_GGC_FINISH`, `gcc.PLUGIN_FINISH_DECL` (gcc 4.7)

`gcc.ArrayType` has gained a “range” attribute, allowing scripts to detect out-of-bounds conditions in array-handling.

A number of memory leaks were fixed: these were found by running the plugin on itself.

Various documentation improvements (ticket #6, ticket #31).

### 22.8.2 Improvements to gcc-with-cpychecker

The `gcc-with-cpychecker` tool has received some deep internal improvements in this release.

The logic for analyzing the outcome of comparisons has been rewritten for this release, fixing some significant bugs that could lead to the analyzer incorrectly deciding whether or not a block of code was reachable.

Similarly, the logic for detecting loops has been rewritten, eliminating a bug in which the checker would prematurely stop analyzing loops with complicated termination conditions, and not analyze the body of the loop.

Doing so extended the reach of the checker, and enabled it to find the memory leaks referred to above.

In addition, the checker now emits more detailed information on the ranges of possible values it’s considering when a comparison occurs against an unknown value:

```c
input.c: In function 'test':
input.c:41:5: warning: comparison against uninitialized data (item) at input.c:41,
---[enabled by default]
input.c:34:12: note: when PyList_New() succeeds at: result = PyList_New(len);
input.c:35:8: note: taking False path at: if (!result) {
input.c:39:12: note: reaching: for (i = 0; i < len; i++) {
input.c:39:5: note: when considering range: 1 <= value <= 0x7fffffff at: for (i = 0;
---[0; i < len; i++) {
input.c:39:5: note: reaching: if (!item) {
input.c:41:5: note: reaching: if (!item) {
```

The checker should do a better job of identifying PyObject subclasses. Previously it was treating any struct beginning with “ob_refcnt” and “ob_type” as a Python object (with some tweaks for python 3 and debug builds). It now also covers structs that begin with a field that’s a PyObject (or subclass), since these are likely to also be PyObject subclasses.

### Usage of deallocated memory

Previously, the checker would warn about paths through a function that could return a pointer to deallocated memory, or which tried to read through such a pointer. With this release, the checker will now also warn about paths through a function in which a pointer to deallocated memory is passed to a function.

For example, given this buggy code:

```c
test (PyObject *self, PyObject *args)
{
    /* Create an object: */
    PyObject *tmp = PyLong_FromLong (0x1000);  
}
```

(continues on next page)
```c
if (!tmp) {
    return;
}

/*
   Now decref the object. Depending on what other references are owned
   on the object, it can reach a refcount of zero, and thus be deallocated:
*/
Py_DECREF(tmp);

/* BUG: the object being returned may have been deallocated */
some_function(tmp);
```

the checker will emit this warning:

```
input.c: In function 'test':
input.c:45: warning: passing pointer to deallocated memory as argument 1 of function
    at input.c:45: memory deallocated at input.c:42 [enabled by default]
input.c:32: note: when PyLong_FromLong() succeeds at: PyObject *tmp = PyLong__FromLong(0x1000);
input.c:34: note: taking False path at: if (!tmp) {
input.c:42: note: reaching: Py_DECREF(tmp);
input.c:42: note: calling tp_dealloc on PyLongObject allocated at input.c:32 at:
    Py_DECREF(tmp);
input.c:45: note: reaching: foo(tmp);
input.c:30: note: graphical error report for function 'passing_dead_object_to_function'
    written out to 'input.c.test-refcount-errors.html'
```

**Coverage of the CPython API**

This release adds heuristics for the behavior of the following CPython API entrypoints:

- PyString_Concat
- PyString_ConcatAndDel

along with various other bugfixes and documentation improvements.

## 22.9 0.8

Thanks to David Narvaez and Tom Tromey for their code contributions to this release.

### 22.9.1 Changes to the GCC Python Plugin

**Initial C++ support**

This release adds the beginnings of C++ support: `gcc.FunctionDecl` instances now have a “fullname” attribute, along with “is_public”, “is_private”, “is_protected”, “is_static” booleans.
For example, given this code:

```cpp
namespace Example {
    struct Coord {
        int x;
        int y;
    };

    class Widget {
        public:
            void set_location(const struct Coord& coord);
    };
};
```

`set_location`'s fullname is:

'void Example::Widget::set_location(const Example::Coord&)'

This is only present when the plugin is invoked from the C++ frontend (`cc1plus`), gracefully handling the case when we’re invoked from other language frontends.

Similarly, `gcc.MethodType` has gained an “argument_types” attribute.

### Unconditional warnings

The `gcc.warning()` function in previous versions of the plugin required an “option” argument, such as `gcc.Option('-Wformat')`

It’s now possible to emit an unconditional warning, by supplying `None` for this argument, which is now the default value:

```
gcc.warning(func.start, 'this is an unconditional warning')
```

which will be an error if `-Werror` is supplied as a command-line argument to gcc:

```
$ ./gcc-with-python script.py input.c
input.c:25:1: warning: this is an unconditional warning [enabled by default]
```

```
$ ./gcc-with-python script.py -Werror input.c
input.c:25:1: error: this is an unconditional warning [-Werror]
```

### 22.9.2 Improvements to gcc-with-cpychecker

The “libcpychecker” Python code is a large example of using the plugin: it extends GCC with code that tries to detect various bugs in CPython extension modules.

As of this release, all of the errors emitted by the tool have been converted to warnings. This should make `gcc-with-cpychecker` more usable as a drop-in replacement for `gcc`: the first source file with a refcounting error should no longer terminate the build (unless the program uses `-Werror`, of course).

### Verification of PyMethodDef tables

This release adds checking of tables of PyMethodDef initialization values, used by Python extension modules for binding C functions to Python methods.
The checker will verify that the signatures of the callbacks match the flags, and that the such tables are NULL terminated:

```c
input.c:48:22: warning: flags do not match callback signature for 'test' within PyMethodDef table
        input.c:48:22: note: actual type of underlying callback: struct PyObject * <Tc58> (...)
        input.c:48:22: note: see http://docs.python.org/c-api/structures.html#PyMethodDef
```

**Coverage of the CPython API**

When the checker warns about code that can erroneously pass NULL to various CPython API entrypoints which are known to implicitly dereference those arguments, the checker will now add an explanatory note about why it is complaining.

For example:

```c
input.c: In function 'test':
input.c:38:33: warning: calling PyString_AsString with NULL as argument 1 at input.c:38
        input.c:38:33: note: when PyObject_Repr() fails at: repr_args = PyObject_Repr(args);
        input.c:38:33: note: PyString_AsString() invokes Py_TYPE() on the pointer via the PyString_Check() macro, thus accessing (NULL)->ob_type
        input.c:27:1: note: graphical error report for function 'test' written out to 'input.c.test-refcount-errors.html'
```

The checker will now verify the argument lists of invocations of PyObject_CallFunctionObjArgs and PyObject_CallMethodObjArgs, checking that all of the arguments are of the correct type (PyObject* or subclasses), and that the list is NULL-terminated:

```c
input.c: In function 'test':
input.c:33:5: warning: argument 2 had type char[12] * but was expecting a PyObject* (or subclass)
        input.c:33:5: warning: arguments to PyObject_CallFunctionObjArgs were not NULL-terminated
```

This release also adds heuristics for the behavior of the following CPython API entrypoints:

- PyArg_Parse
- PyCObject_{As,From}VoidPtr
- PyCallable_Check
- PyCapsule_GetPointer
- PyErr_{NewException,SetNone,WarnEx}
- PyEval_CallObjectWithKeywords
- PyEval_{Save,Restore}Thread (and thus the Py_{BEGIN,END}_ALLOW_THREADS macros)
- PyList_{GetItem,Size}
- PyLong_{From,LongLong}
- PyMapping_Size
Bug fixes

- gcc-with-cpychecker will now try harder on functions that are too complicated to fully handle. Previously, when a function was too complicated for the reference-count tracker to fully analyze, it would give up, performing no analysis. The checker will now try to obtain at least some subset of the list of all traces through the function, and analyze those. It will still note that the function was too complicated to fully analyze.

  Given that we do a depth-first traversal of the tree, and that “success” transitions are typically visited before “failure” transitions, this means that it should at least analyze the trace in which all functions calls succeed, together with traces in which some of the later calls fail.

- the reference-count checker now correctly handles “static” PyObject* local variables: a static PyObject * local preserves its value from call to call, and can thus permanently own a reference.

  Fixes a false-positive seen in psycopg2-2.4.2 (psycopg/psycopgmodule.c:psyco_GetDecimalType) where the refcount checker erroneously reported that a reference was leaked.

- the checker for Py_BuildValue("O") (and “S” and “N”) was being too strict, requiring a (PyObject*). Although it’s not explicitly documented, it’s clear that these can also accept pointers to any PyObject subclass.

  Fixes a false positive seen when running gcc-with-cpychecker on coverage-3.5.1b1.tar.gz, in which coverage/tracer.c:Tracer_trace passes a PyFrameObject* as an argument to such a call.

- the reference-count checker now correctly suppresses reports about “leaks” for traces that call a function that never return (such as abort()).

  Fixes a false positive seen in rpm-4.9.1.2 in a handler for fatal errors: (in python/rpmts-py.c:die) where the checker erroneously reported that a reference was leaked.

- tp_iternext callbacks are allowed to return NULL without setting an exception. The reference-count checker will now notice if a function is used in such a role, and suppress warnings about such behavior.

- fixed various Python tracebacks (tickets #14, #19, #20, #22, #23, #24, #25)

- various other fixes

22.10 0.7

This is a major update to the GCC Python plugin.

The main example script, cpychecker, has seen numerous improvements, and has now detected many reference-counting bugs in real-world CPython extension code. The usability and signal:noise ratio is greatly improved over previous releases.
22.10.1 Changes to the GCC Python Plugin

It’s now possible to create custom GCC attributes from Python, allowing you to add custom high-level annotation to a C API, and to write scripts that will verify these properties. It’s also possible to inject preprocessor macros from Python. Taken together, this allows code like this:

```c
#ifdef WITH_ATTRIBUTE_CLAIMS_MUTEX
#define CLAIMS_MUTEX(x) __attribute__((claims_mutex(x)))
#else
#define CLAIMS_MUTEX(x)
#endif

#ifdef WITH_ATTRIBUTE_RELEASES_MUTEX
#define RELEASES_MUTEX(x) __attribute__((releases_mutex(x)))
#else
#define RELEASES_MUTEX(x)
#endif

/* Function declarations with custom attributes: */
extern void some_function(void) CLAIMS_MUTEX("io");
extern void some_other_function(void) RELEASES_MUTEX("io");
extern void yet_another_function(void) CLAIMS_MUTEX("db")
                                           CLAIMS_MUTEX("io")
                                           RELEASES_MUTEX("io");
```

Other improvements:

- GCC’s debug dump facilities are now exposed via a Python API
- It’s now possible to run Python commands in GCC (rather than scripts) using `-fplugin-arg-python-command`
- Improvements to the “source location” when reporting on an unhandled Python exception. Amongst other tweaks, it’s now possible for a script to override this, which the cpychecker uses, so that if it can’t handle a particular line of C code, the GCC error report gives that location before reporting the Python traceback (making debugging much easier).
- “switch” statements are now properly wrapped at the Python level (gcc.GimpleSwitch)
- C bitfields are now wrapped at the Python level
- gcc.Type instances now have a “sizeof” attribute, and an “attributes” attribute.
- Added a gcc.Gimple.walk_tree method, to make it easy to visit all nodes relating to a statement
- Added a new example: spell-checking all string literals in code

22.10.2 Improvements to “cpychecker”

The “libcpychecker” Python code is a large example of using the plugin: it extends GCC with code that tries to detect various bugs in CPython extension modules.

The cpychecker analyzes the paths that can be followed through a C function, and verifies various properties, including reference-count handling.
As of this release, the pass has found many reference-counting bugs in real-world code. You can see a list of the bugs that it has detected at:


The checker is now almost capable of fully handling the C code within the gcc python plugin itself.

The checker has also been reorganized to (I hope) make it easy to add checking for other libraries and APIs.

**Major rewrite of reference-count tracking**

I’ve rewritten the internals of how reference counts are tracked: the code now makes a distinction between all of the reference that can be analysed within a single function, versus all of the other references that may exist in the rest of the program.

This allows us to know for an object e.g. that the function doesn’t directly own any references, but that the reference count is still > 0 (a “borrowed reference”), as compared to the case where the function owns a reference, but we don’t know of any in the rest of the program (this is typical when receiving a “new reference” e.g. from a function call to a constructor).

Within the reference-count checker, we now look for memory locations that store references to objects. If those locations not on the stack, then the references they store are now assumed to legally count towards the ob_refcnt that the function “owns”. This is needed in order to correctly handle e.g. the PyList_SET_ITEM() macro, which directly writes to the list’s ob_item field, “stealing” a reference: we can detect these references, and count them towards the ob_refcnt value.

The checker can now detect types that look like PyObject subclasses at the C level (by looking at the top-most fields), and uses this information in various places.

The checker now exposes custom GCC attributes allowing you to mark APIs that have non-standard reference-handling behavior:

```c
PyObject *foo(void)
    CPYCHECKER_RETURNS_BORROWED_REF;

extern void bar(int i, PyObject *obj, int j, PyObject *other)
    CPYCHECKER_STEALS_REFERENCE_TO_ARG(2)
    CPYCHECKER_STEALS_REFERENCE_TO_ARG(4);
```

It also exposes an attribute allowing you to the run-time and compile-time type information for a Python extension class:

```c
/* Define some PyObject subclass, as both a struct and a typedef */
struct OurObjectStruct {
    PyObject_HEAD
    /* other fields */
};
typedef struct OurObjectStruct OurExtensionObject;

/*
 * Declare the PyTypeObject, using the custom attribute to associate it with
 * the typedef above:
 */
extern PyTypeObject UserDefinedExtension_Type
    CPYCHECKER_TYPE_OBJECT_FOR_TYPEDEF("OurExtensionObject");
```
Function calls with NULL-pointer arguments

The checker knows about various CPython API hooks that will crash on NULL pointer arguments, and will emit warnings when it can determine a path through the code that will lead to a definite call with a NULL value.

Dereferences of uninitialized pointers

The checker will now complain about paths through a function for which it can prove that an uninitialized pointer will be dereferenced.

Error-reporting improvements

The error-reporting machinery can generate HTML reports: see e.g.: http://readthedocs.org/docs/gcc-python-plugin/en/latest/cpychecker.html#reference-count-checking and http://dmalcolm.livejournal.com/6560.html

The checker can now annotate its HTML (and textual) reports with information showing how some pertinent aspect of the program’s state changes during a particular path through a function.

For example, when reporting on reference-counting errors, the HTML report showing the flow through the function will now display all changes to an object’s ob_refcnt, together with all changes to what the value ought to be (e.g. due to pointers being stored to persistent memory locations):

File: input.c
Function test
Error: ob_refcnt of *list is 1 too high

22 PyObject *
23 test(PyObject *self, PyObject *args)
24 {
25     PyObject *list;
26     PyObject *item;
27     list = PyList_New(1);
28         when PyList_New() succeeds
29         PyListObject allocated at: list = PyList_New(1);
30         ob_refcnt is now refs: 1 + N where N >= 0
31     if (!list)
32         taking False path
33         return NULL;
34     item = PyLong_FromLong(42);
35         when PyLong_FromLong() fails
36         /* This error handling is incorrect: it's missing an
37            invocation of Py_DECREF(list): */
38     if (!item)
39         taking True path
40         return NULL;
41         /* This steals a reference to item; item is not leaked when we get here: */
42     PyList_SetItem(list, 0, item);
43     return list;
44 }

ob_refcnt of *list is 1 too high
was expecting final ob_refcnt to be N + 0 (for some unknown N)
but final ob_refcnt is N + 1
Similarly, when reporting on exception-handling errors, it now displays the “history” of changes to the thread-local exception state.

There’s also a debug mode which dumps _everything_ that changes within the report, which is helpful for debugging the checker itself.

The error report will attempt to use the most representative name for a leaked object, using a variable name or a C expression fragment as appropriate.

The checker will attempt to combine duplicate error reports, so that it will only emit one error for all of the various traces of execution that exhibit a particular reference-counting bug.

Finally, when writing out an HTML report, the path to the HTML is now noted within gcc’s regular stderr messages.

**Signal:noise ratio improvements**

To suppress various false-positives that I commonly ran into on real code, the checker now makes certain assumptions:

- When encountering an unknown function that returns a PyObject*, the checker assumes that it will either return a new reference to a sane object (with a sane ob_type), or return NULL and set the thread-local exception state.

- The checker assumes that a PyObject* argument to a function is non-NULL and has a >0 refcount, and has a sane ob_type (e.g. with a sane refcount and tp_dealloc)

- When dereferencing a pointer that it has no knowledge about (e.g. a pointer field in a structure), the checker now assumes that it’s non-NULL, unless it knows that NULL is a definite possibility i.e. it optimistically assumes that you know what you’re doing (this could be turned into a command-line option). Note that for the cases where we know that the pointer can _definitely_ be NULL, an error will still be reported (e.g. when considering the various possible return values for a function known to be able to return NULL).

**Coverage of the CPython API**

I’ve gone through much of the CPython API, “teaching” the checker about the reference-count semantics of each API call (and which calls will crash if fed a NULL pointer). This involves writing a simple fragment of Python code for each function, which describes the various different affects that the call can have on the internal state within the callee.

This release adds support for calls to the following:

- _PyObject_New
- Py_{Initialize,Finalize}
- Py_InitModule4
- PyArg_ParseTuple[AndKeywords], and the PY_SSIZE_T_CLEAN variants (only partial coverage so far: “O”, “O!” should work though)
- PyArg_UnpackTuple
- PyBool_FromLong
- Py_BuildValue and the PY_SSIZE_T_CLEAN variant (only partial coverage so far)
- PyDict_{GetItem,GetItemString,New,SetItem,SetItemString}
- PyErr_{Format,NoMemory,Occurred,Print,PrintEx,SetFromErrno[WithFilename], SetObject,SetString}
- PyEval_InitThreads
- PyGILState_{Ensure,Release}
- PyImport_{AppendInittab,ImportModule}
I’ve been targeting those API entrypoints that I use myself in the plugin; this is one area which is particularly amenable to patching, for anyone who wants to get involved. I’ve also added a (disabled) hook that complains about Python API entrypoints that weren’t explicitly handled, to make it easy to find gaps in our coverage of the CPython API.

Other user-visible improvements

- There’s now a “gcc-with-cpychecker” harness, to make it easier to invoke GCC with the cpychecker code from e.g. Makefiles
- The checker now respects __attribute__((nonnull)) on function arguments when detecting NULL pointers
- Handle functions that don’t return (e.g. “exit”)
- Number the unknown heap regions, to clarify things when there’s more than one

Internal improvements

- The cpychecker now has some protection against combinatorial explosion for functions that have very large numbers of possible routes through them. For such functions, the checker will emit a note on stderr and not attempt to find reference-counting bugs in the function.
- The cpychecker is now done as a custom pass (rather than by wiring up a callback associated with every pass)
- I’ve tuned the logging within the checker, eliminating some CPU/memory consumption issues seen when analysing complicated C code. In particular, the log message arguments are now only expanded when logging is enabled (previously this was happening all the time).
- Lots of other internal improvements and bug fixes (e.g. handling of arrays vs pointers, static vs auto local variables, add missing handlers for various kinds of C expression, lots of work on improving the readability of error messages)
Appendices

The following contain tables of reference material that may be useful when writing scripts.

23.1 All of GCC’s passes

This diagram shows the various GCC optimization passes, arranged vertically, showing child passes via indentation. The lifetime of the various properties that they maintain is shown, giving the pass that initially creates the data (if any), the pass that destroys it (if any), and each pass that requires a particular property (based on the PROP_* flags).

These tables contain the same information. The diagram and tables were autogenerated, using GCC 4.6.0

23.1.1 The lowering passes

<table>
<thead>
<tr>
<th>Pass Name</th>
<th>Required properties</th>
<th>Provided properties</th>
<th>Destroyed properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>*warn_unused_result</td>
<td>gimple_any</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*diagnose_omp_blocks</td>
<td>gimple_any</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mudflap1</td>
<td>gimple_any</td>
<td></td>
<td></td>
</tr>
<tr>
<td>omplower</td>
<td>gimple_any</td>
<td>gimple_lomp</td>
<td></td>
</tr>
<tr>
<td>lower</td>
<td>gimple_any</td>
<td>gimple_lcf</td>
<td></td>
</tr>
<tr>
<td>ehopt</td>
<td>gimple_lcf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eh</td>
<td>gimple_lcf</td>
<td>gimple_leh</td>
<td></td>
</tr>
<tr>
<td>cfg</td>
<td>gimple_leh</td>
<td>cfg</td>
<td></td>
</tr>
<tr>
<td>*warn_function_return</td>
<td>cfg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*build_cgraph_edges</td>
<td>cfg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## 23.1.2 The “small IPA” passes

<table>
<thead>
<tr>
<th>Pass Name</th>
<th>Required properties</th>
<th>Provided properties</th>
<th>Destroyed properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>*free_lang_data</td>
<td>gimple_any, gimple_lcf, gimple_leh, cfg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>visibility</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>early_local_cleanups</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; *free_cfg_annotations</td>
<td>cfg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; *init_datastructures</td>
<td>cfg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; ompxexp</td>
<td>gimple_any</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; *=referenced_vars</td>
<td>gimple_leh, cfg</td>
<td></td>
<td>referenced_vars</td>
</tr>
<tr>
<td>&gt; ssa</td>
<td>cfg, referenced_vars</td>
<td></td>
<td>ssa</td>
</tr>
<tr>
<td>&gt; veclower</td>
<td>cfg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; *=early_warn_uninitialized</td>
<td>ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; *=rebuild_cgraph_edges</td>
<td>cfg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; inline_param</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; einline</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; early_optimizations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; *=remove_cgraph_callee_edges</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; copyrename</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; cgp</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; forwprop</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; ealias</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; esra</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; copyprop</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; mergephi</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; cddce</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; eipa_sra</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; tailr</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; switchconv</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; ehcleanup</td>
<td>gimple_lcf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; profile</td>
<td>cfg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; local-pure-const</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; fnsplit</td>
<td>cfg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; release_ssa</td>
<td>ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; *=rebuild_cgraph_edges</td>
<td>cfg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; inline_param</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tree_profile_ipa</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; feedback_fnsplit</td>
<td>cfg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>increase_alignment</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>matrix-reorg</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>emutls</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## 23.1.3 The “regular IPA” passes
### 23.1.4 Passes generating Link-Time Optimization data

<table>
<thead>
<tr>
<th>Pass Name</th>
<th>Required properties</th>
<th>Provided properties</th>
<th>Destroyed properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>lto_gimple_out</td>
<td>gimple_any, gimple_lcf, gimple_leh, cfg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lto_decls_out</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 23.1.5 The “all other passes” catch-all

<table>
<thead>
<tr>
<th>Pass Name</th>
<th>Required properties</th>
<th>Provided properties</th>
<th>Destroyed properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>ehdisp</td>
<td>gimple_any, gimple_lcf, gimple_leh, cfg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*all_optimizations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; *remove_cgraph callee_edges</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; *strip_predict_hints</td>
<td>cfg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; copyrename</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; cunrolli</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; ccp</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; forwprop</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; cdce</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; alias</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; retslot</td>
<td>ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; phiprop</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; fre</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; copyprop</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; mergephi</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; vrp</td>
<td>ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; dce</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; cselim</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; ifcombine</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; phiopt</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pass Name</td>
<td>Required properties</td>
<td>Provided properties</td>
<td>Destroyed properties</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------</td>
<td>---------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>tailr</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ch</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stdarg</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cplxlower</td>
<td>ssa</td>
<td></td>
<td>gimple_lcx</td>
</tr>
<tr>
<td>sra</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>copyrename</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dom</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>phicprop</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dse</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reassoc</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dce</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>forwprop</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>phiopt</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>objsz</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ccp</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>copyprop</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sincos</td>
<td>ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bswap</td>
<td>ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>crited</td>
<td>cfg</td>
<td></td>
<td>no_crit_edges</td>
</tr>
<tr>
<td>pre</td>
<td>cfg, ssa, no_crit_edges</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sink</td>
<td>cfg, ssa, no_crit_edges</td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td>cfg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; loopinit</td>
<td>cfg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; lim</td>
<td>cfg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; copyprop</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; dceloop</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; unswitch</td>
<td>cfg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; sccp</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; *record_bounds</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; ckdd</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; ldist</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; copyprop</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; graphite0</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; graphite</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; lim</td>
<td>cfg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; copyprop</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; dceloop</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; ivcanon</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; ifcvt</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; vect</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; veclower2</td>
<td>cfg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; dceloop</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; pcom</td>
<td>cfg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; cunroll</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; slp</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; parloops</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; aprefetch</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; i opts</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; loopdone</td>
<td>cfg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2 – continued from previous page

<table>
<thead>
<tr>
<th>Pass Name</th>
<th>Required properties</th>
<th>Provided properties</th>
<th>Destroyed properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; recip</td>
<td>ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; reassoc</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; vrp</td>
<td>ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; dom</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; phicprop</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; cdce</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; tracer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; uninit</td>
<td>ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; dse</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; forwprop</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; phiopt</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; fab</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; widening_mul</td>
<td>ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; tailc</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; copyrename</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; uncprop</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; local-pure-const</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cplxlower0</td>
<td>cfg</td>
<td>gimple_lcx</td>
<td></td>
</tr>
<tr>
<td>ehcleanup</td>
<td>gimple_lcf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>resx</td>
<td>gimple_lcf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nrv</td>
<td>cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mudflap2</td>
<td>gimple_leh, cfg, ssa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>optimized</td>
<td>cfg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*warn_function_noreturn</td>
<td>cfg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>expand</td>
<td>gimple_leh, cfg, ssa, gimple_lcx</td>
<td>rtl</td>
<td>gimple_any, gimple_lcf, gimple_lcx</td>
</tr>
<tr>
<td>*rest_of_compilation</td>
<td>rtl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; *init_function</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; sibling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; rtl eh</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; initvals</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; unshare</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; vregs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; into_cfglayout</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; jump</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; subreg1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; dfinit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; cse1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; fwprop1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; cprop</td>
<td>cfglayout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; rtl pre</td>
<td>cfglayout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; hoist</td>
<td>cfglayout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; cprop</td>
<td>cfglayout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; store_motion</td>
<td>cfglayout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; cse_local</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; ce1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; reginfo</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; loop2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; loop2_init</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; &gt; loop2_invariant</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

23.1. All of GCC’s passes
Table 2 – continued from previous page

<table>
<thead>
<tr>
<th>Pass Name</th>
<th>Required properties</th>
<th>Provided properties</th>
<th>Destroyed properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop2_unswitch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop2_unroll</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop2_doloop</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop2_done</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>web</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cprop</td>
<td>cfglayout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cse2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dse1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lwpred2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>auto_inc_dec</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>init-reg</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ud dce</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>combine</td>
<td>cfglayout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ce2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bbpart</td>
<td>cfglayout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>regmove</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>outof_cfglayout</td>
<td></td>
<td>cfglayout</td>
<td></td>
</tr>
<tr>
<td>split1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>subreg2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>no-opt dfinit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*stack_ptr_mod</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mode_sw</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>asmcons</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sms</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sched1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ica</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*all-postreload</td>
<td>rtl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>postreload</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gcse2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>split2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>zee</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmpelim</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bt1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pro_and_epilogue</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dse2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>csa</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>peephole2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ce3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nrreg</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cprop_hardreg</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rtl dce</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bbro</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bt1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*leaf_regs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>split4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sched2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*stack_regs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*stack_regs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>split3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stack</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2 – continued from previous page

<table>
<thead>
<tr>
<th>Pass Name</th>
<th>Required properties</th>
<th>Provided properties</th>
<th>Destroyed properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;&gt; alignments</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; compgotos</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; vartrack</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; *free_cfg</td>
<td></td>
<td></td>
<td>cfg</td>
</tr>
<tr>
<td>&gt;&gt; mach</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; barriers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; dbr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; split5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; eh_ranges</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; shorten</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; nothrow</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; final</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;&gt; dfinish</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*clean_state</td>
<td></td>
<td></td>
<td>rtl</td>
</tr>
</tbody>
</table>

23.2 \texttt{gcc.Tree} operators by symbol

The following shows the symbol used for each expression subclass in debug dumps, as returned by the various \texttt{get\_symbol()} class methods.

There are some duplicates (e.g. `\texttt{-}` is used for both \texttt{gcc.MinusExpr} as an infix binary operator, and by \texttt{gcc.NegateExpr} as a prefixed unary operator).

<table>
<thead>
<tr>
<th>Class</th>
<th>\texttt{get_symbol()}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{gcc.AddrExpr}</td>
<td><code>&amp;</code></td>
</tr>
<tr>
<td>\texttt{gcc.BitAndExpr}</td>
<td><code>&amp;</code></td>
</tr>
<tr>
<td>\texttt{gcc.BitIorExpr}</td>
<td>`</td>
</tr>
<tr>
<td>\texttt{gcc.BitNotExpr}</td>
<td><code>~</code></td>
</tr>
<tr>
<td>\texttt{gcc.BitXorExpr}</td>
<td><code>^</code></td>
</tr>
<tr>
<td>\texttt{gcc.CeilDivExpr}</td>
<td><code>/\text{cl}</code></td>
</tr>
<tr>
<td>\texttt{gcc.CeilModExpr}</td>
<td><code>%\text{cl}</code></td>
</tr>
<tr>
<td>\texttt{gcc.EqExpr}</td>
<td><code>==</code></td>
</tr>
<tr>
<td>\texttt{gcc.ExactDivExpr}</td>
<td><code>/\text{ex}</code></td>
</tr>
<tr>
<td>\texttt{gcc.FloorDivExpr}</td>
<td><code>/\text{fl}</code></td>
</tr>
<tr>
<td>\texttt{gcc.FloorModExpr}</td>
<td><code>%\text{fl}</code></td>
</tr>
<tr>
<td>\texttt{gcc.GeExpr}</td>
<td><code>&gt;=</code></td>
</tr>
<tr>
<td>\texttt{gcc.GtExpr}</td>
<td><code>&gt;</code></td>
</tr>
<tr>
<td>\texttt{gcc.IndirectRef}</td>
<td><code>*</code></td>
</tr>
<tr>
<td>\texttt{gcc.LeExpr}</td>
<td><code>&lt;=</code></td>
</tr>
<tr>
<td>\texttt{gcc.LrotateExpr}</td>
<td><code>r&lt;&lt;</code></td>
</tr>
<tr>
<td>\texttt{gcc.LshiftExpr}</td>
<td><code>&lt;&lt;</code></td>
</tr>
<tr>
<td>\texttt{gcc.LtExpr}</td>
<td><code>&lt;</code></td>
</tr>
<tr>
<td>\texttt{gcc.LtgtExpr}</td>
<td><code>&lt;&gt;</code></td>
</tr>
<tr>
<td>\texttt{gcc.MaxExpr}</td>
<td><code>max</code></td>
</tr>
<tr>
<td>\texttt{gcc.MinExpr}</td>
<td><code>min</code></td>
</tr>
<tr>
<td>\texttt{gcc.MinusExpr}</td>
<td><code>-</code></td>
</tr>
</tbody>
</table>

Continued on next page
This document describes the Python plugin I’ve written for GCC. In theory the plugin allows you to write Python scripts that can run inside GCC as it compiles code, exposing GCC’s internal data structures as a collection of Python classes and functions. The bulk of the document describes the Python API it exposes.

Hopefully this will be of use for writing domain-specific warnings, static analysers, and the like, and for rapid prototyping of new GCC features.

I’ve tried to stay close to GCC’s internal representation, but using classes. I hope that the resulting API is pleasant to work with.

The plugin is a work-in-progress; the API may well change.

Bear in mind that writing this plugin has been the first time I have worked with the insides of GCC. I have only wrapped the types I have needed, and within them, I’ve only wrapped properties that seemed useful to me. There may well be plenty of interesting class and properties for instances that can be added (patches most welcome!). I may also

Table 3 – continued from previous page

<table>
<thead>
<tr>
<th>Class</th>
<th>get_symbol()</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc.ModifyExpr</td>
<td>=</td>
</tr>
<tr>
<td>gcc.MultExpr</td>
<td>*</td>
</tr>
<tr>
<td>gcc.NeExpr</td>
<td>!=</td>
</tr>
<tr>
<td>gcc.NegateExpr</td>
<td>-</td>
</tr>
<tr>
<td>gcc.OrderedExpr</td>
<td>ord</td>
</tr>
<tr>
<td>gcc.PlusExpr</td>
<td>+</td>
</tr>
<tr>
<td>gcc.PointerPlusExpr</td>
<td>+</td>
</tr>
<tr>
<td>gcc.PostdecrementExpr</td>
<td>-</td>
</tr>
<tr>
<td>gcc.PostincrementExpr</td>
<td>++</td>
</tr>
<tr>
<td>gcc.PredecrementExpr</td>
<td>-</td>
</tr>
<tr>
<td>gcc.PreincrementExpr</td>
<td>++</td>
</tr>
<tr>
<td>gcc.RedivExpr</td>
<td>/</td>
</tr>
<tr>
<td>gcc.ReducPlusExpr</td>
<td>r+</td>
</tr>
<tr>
<td>gcc.RoundDivExpr</td>
<td>/ [rd]</td>
</tr>
<tr>
<td>gcc.RoundModExpr</td>
<td>% [rd]</td>
</tr>
<tr>
<td>gcc.RrotateExpr</td>
<td>r&gt;&gt;</td>
</tr>
<tr>
<td>gcc.RshiftExpr</td>
<td>&gt;&gt;</td>
</tr>
<tr>
<td>gcc.TruncDivExpr</td>
<td>/</td>
</tr>
<tr>
<td>gcc.TruncModExpr</td>
<td>%</td>
</tr>
<tr>
<td>gcc.TruthAndExpr</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>gcc.TruthAndifExpr</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>gcc.TruthNotExpr</td>
<td>!</td>
</tr>
<tr>
<td>gcc.TruthOrExpr</td>
<td></td>
</tr>
<tr>
<td>gcc.TruthOrifExpr</td>
<td></td>
</tr>
<tr>
<td>gcc.TruthXorExpr</td>
<td>^</td>
</tr>
<tr>
<td>gcc.UneqExpr</td>
<td>u==</td>
</tr>
<tr>
<td>gcc.UngeExpr</td>
<td>u&gt;=</td>
</tr>
<tr>
<td>gcc.UngtExpr</td>
<td>u&gt;</td>
</tr>
<tr>
<td>gcc.UnleExpr</td>
<td>u&lt;=</td>
</tr>
<tr>
<td>gcc.UnltExpr</td>
<td>u&lt;</td>
</tr>
<tr>
<td>gcc.UnorderedExpr</td>
<td>unord</td>
</tr>
<tr>
<td>gcc.VecLshiftExpr</td>
<td>v&lt;&lt;</td>
</tr>
<tr>
<td>gcc.VecRshiftExpr</td>
<td>v&gt;&gt;</td>
</tr>
<tr>
<td>gcc.WidenMultExpr</td>
<td>w*</td>
</tr>
<tr>
<td>gcc.WidenSumExpr</td>
<td>w+</td>
</tr>
</tbody>
</table>
have misunderstood how things work.

Most of my development has been against Python 2 (2.7, actually), but I’ve tried to make the source code of the plugin buildable against both Python 2 and Python 3 (3.2), giving separate python2.so and python3.so plugins. (I suspect that it’s only possible to use one or the other within a particular invocation of “gcc”, due to awkward dynamic-linker symbol collisions between the two versions of Python).

The plugin is Free Software, licensed under the GPLv3 (or later).
CHAPTER 24

Indices and tables

- genindex
- modindex
- search
Symbols

--dump-json
   gcc-with-cpychecker command line option, 86
--maxtrans <int>
   gcc-with-cpychecker command line option, 86
_init__() (gcc.Location method), 34

A

add_fixit_replace() (gcc.RichLocation method), 34
addr (gcc.Tree attribute), 41
alias_of (gcc.NamespaceDecl attribute), 45
args (gcc.GimpleCall attribute), 68
args (gcc.GimplePhi attribute), 69
argument_types (gcc.FunctionType attribute), 49
argument_types (gcc.MethodType attribute), 49
arguments (gcc.FunctionDecl attribute), 44
array (gcc.ArrayRef attribute), 58
ArrayRangeRef (built-in class), 59
AttrAddrExpr (built-in class), 59
attributes (gcc.Type attribute), 46

B

basever (gcc.Version attribute), 115
basic_blocks (gcc.Cfg attribute), 37
BitFieldRef (built-in class), 59
block (gcc.Gimple attribute), 65
block (gcc.TranslationUnitDecl attribute), 13

call_stmt (gcc.CallgraphEdge attribute), 110
callee (gcc.CallgraphEdge attribute), 110
callees (gcc.CallgraphNode attribute), 109
caller (gcc.CallgraphEdge attribute), 109
callers (gcc.CallgraphNode attribute), 109
callgraph_node (gcc.FunctionDecl attribute), 45
caret (gcc.Location attribute), 34
cfg (gcc.Function attribute), 37
column (gcc.Location attribute), 33
complex (gcc.Edgettribute), 39
configuration_arguments (gcc.Version attribute), 115
const, 46
const_equivalent, 46
constant (gcc.Constant attribute), 50
current_value (gcc.Parameter attribute), 113

d

datestamp (gcc.Version attribute), 115
debug() (gcc.Tree method), 41
decl (gcc.CallgraphNode attribute), 109
decl (gcc.Function attribute), 37
decl (gcc.Variable attribute), 13
declarations (gcc.NamespaceDecl attribute), 45
def_stmt (gcc.SsaName attribute), 64
default_value (gcc.Parameter attribute), 114
dereference (gcc.ArrayType attribute), 48
dereference (gcc.PointerType attribute), 48
dereference (gcc.VectorType attribute), 48
dest (gcc.Edge attribute), 39
devphase (gcc.Version attribute), 115
dump_enabled (gcc.Pass attribute), 72

e

end (gcc.Function attribute), 37
entry (gcc.Cfg attribute), 37
execute(), 73
exit (gcc.Cfg attribute), 37
exprcode (gcc.GimpleAssign attribute), 67, 69
exprcode (gcc.GimpleCond attribute), 68
exprtype (gcc.Gimple attribute), 65

F

false_label (gcc.GimpleCond attribute), 68
false_value (gcc.Edge attribute), 39
field (gcc.ComponentRef attribute), 58
fields (gcc.RecordType attribute), 49
file (gcc.Location attribute), 33
finish (gcc.Location attribute), 34
fn (gcc.GimpleCall attribute), 68
gcc.Parameter (built-in class), 113
gcc.ParmDecl (built-in class), 45
gcc.Pass (built-in class), 71
gcc.permerror() (built-in function), 36
gcc.PLUGIN_ATTRIBUTES (built-in variable), 78
gcc.PLUGIN_FINISH (built-in variable), 79
gcc.PLUGIN_FINISHDECL (built-in variable), 79
gcc.PLUGIN_FINISH_TYPE (built-in variable), 79
gcc.PLUGIN_FINISH_UNIT (built-in variable), 79
gcc.PLUGIN_PASS_EXECUTION (built-in variable), 78
gcc.PLUGIN_PRE_GENERICIZE (built-in variable), 79
gcc.PointerType (built-in class), 48
gcc.RecordType (built-in class), 49
gcc.Register (built-in class), 58
gcc.register_attribute() (built-in function), 81
gcc.register_callback() (built-in function), 77
gcc.ResultDecl (built-in class), 45
gcc.RichLocation (built-in class), 34
gcc.Rtl (built-in class), 116
gcc.RtlPass (built-in class), 72
gcc.set_location() (built-in function), 10
gcc.SimpleIpaPass (built-in class), 73
gcc.SsaName (built-in class), 64
gcc.Statement (built-in class), 64
gcc.TranslationUnitDecl (built-in class), 13
gcc.Tree (built-in class), 41
gcc.Type (built-in class), 46
gcc.Unary (built-in class), 55
gcc.Unary.gcc.AbsExpr (built-in class), 56
gcc.Unary.gcc.AddrSpaceConvertExpr (built-in class), 56
gcc.Unary.gcc.BitNotExpr (built-in class), 56
gcc.Unary.gcc.CastExpr (built-in class), 56
gcc.Unary.gcc.ConjExpr (built-in class), 56
gcc.Unary.gcc.ConvertExpr (built-in class), 56
gcc.Unary.gcc.DynamicCastExpr (built-in class), 56
gcc.Unary.gcc.FixedConvertExpr (built-in class), 56
gcc.Unary.gcc.FixTruncExpr (built-in class), 56
gcc.Unary.gcc.FloatExpr (built-in class), 56
gcc.Unary.gcc.NegateExpr (built-in class), 56
gcc.Unary.gcc.NoexceptExpr (built-in class), 56
gcc.Unary.gcc.NormalExpr (built-in class), 56
gcc.Unary.gcc.NopExpr (built-in class), 56
gcc.Unary.gcc.ParenExpr (built-in class), 56
gcc.Unary.gcc.ReducMaxExpr (built-in class), 56
gcc.Unary.gcc.ReducMinExpr (built-in class), 56
gcc.Unary.gcc.ReducPlusExpr (built-in class), 56
gcc.Unary.gcc.ReinterpretCastExpr (built-in class), 56
gcc.Unary.gcc.StaticCastExpr (built-in class), 56
gcc.Unary.gcc.UnaryPlusExpr (built-in class), 56
gcc.VarDecl (built-in class), 45
gcc.Variable (built-in class), 13
gcc.VectorType (built-in class), 48
gcc.Version (built-in class), 115
gcc.warning() (built-in function), 35
gccutils.callgraph_to_dot() (built-in function), 109
gccutils.get_field_by_name() (built-in function), 14
gccutils.get_global_typedef() (built-in function), 13
gccutils.get_global_vardecl_by_name() (built-in function), 13
gccutils.get_src_for_loc() (built-in function), 33
gccutils.get_variables_as_dict() (built-in function), 13
get_block_for_label() (gcc.Cfg method), 38
get_by_name() (gcc.Pass class method), 71
get_roots() (gcc.Pass class method), 71
get_symbol() (gcc.Binary class method), 51
get_symbol() (gcc.Comparison class method), 57
get_symbol() (gcc.Expression class method), 59
get_symbol() (gcc.Reference class method), 58
gccutils.pformat() (built-in function), 42
gccutils.pprint() (built-in function), 43
help (gcc.Option attribute), 113
help (gcc.Parameter attribute), 114
high (gcc.CaseLabelExpr attribute), 64
ImagpartExpr (built-in class), 59
in_system_header (gcc.Location attribute), 33
index (gcc.ArrayRef attribute), 58
index (gcc.BasicBlock attribute), 38
indexvar (gcc.BasicBlock attribute), 69
IndirectRef (built-in class), 59
initial (gcc.VarDecl attribute), 45
is_artificial (gcc.Declaration attribute), 44
is_builtin (gcc.Declaration attribute), 44
is_driver (gcc.Option attribute), 113
is_enabled (gcc.Option attribute), 113
is_optimization (gcc.Option attribute), 113
is_private (gcc.FunctionDecl attribute), 45
is_protected (gcc.FunctionDecl attribute), 45
is_public (gcc.FunctionDecl attribute), 45
is_static (gcc.FunctionDecl attribute), 45
is_target (gcc.Option attribute), 113
is_variadic (gcc.FunctionType attribute), 48
is_warning (gcc.Option attribute), 113
labels (gcc.GimpleSwitch attribute), 69
language (gcc.TranslationUnitDecl attribute), 13
lhs (gcc.GimpleAssign attribute), 67, 69
lhs (gcc.GimpleCall attribute), 67
lhs (gcc.GimpleCond attribute), 68
lhs (gcc.GimplePhi attribute), 69
line (gcc.Location attribute), 33
loc (gcc.Gimple attribute), 65
loc (gcc.Rtl attribute), 116
local_decls (gcc.Function attribute), 37
location (gcc.Binary attribute), 51
location (gcc.Comparison attribute), 57
location (gcc.Declaration attribute), 44
location (gcc.Expression attribute), 59
location (gccREFERENCE attribute), 58
location (gcc.Unary attribute), 55
lookup() (gcc.NamespaceDecl method), 46
low (gcc CASELABEL_EXPR attribute), 64
max_value (gcc.IntegerType attribute), 47
max_value (gcc.Parameter attribute), 114
MemberRef (built-in class), 59
methods (gcc.RecordType attribute), 49
min_value (gcc.IntegerType attribute), 48
min_value (gcc.Parameter attribute), 114
name (gcc.Declaration attribute), 44
name (gcc.FieldDecl attribute), 44
name (gcc.Pass attribute), 71
name (gcc.Type attribute), 46
namespaces (gcc.NamespaceDecl attribute), 45
next (gcc.Pass attribute), 72
noreturn (gcc.GimpleCall attribute), 68
offset_column() (gcc.Location method), 33
OffsetRef (built-in class), 59
operand (gcc.MemRef attribute), 58
operand (gcc.Unary attribute), 55
operands (gcc.Rtl attribute), 116
option (gcc.Parameter attribute), 113
precision (gcc.FloatType attribute), 48
precision (gcc.IntegerType attribute), 47
preds (gcc.BasicBlock attribute), 38
properties_destroyed (gcc.Pass attribute), 72
properties_provided (gcc.Pass attribute), 72
properties_required (gcc.Pass attribute), 72
range (gccArrayType attribute), 48
RealpartExpr (built-in class), 59
register_after() (gcc.Pass method), 74
register_before() (gcc.Pass method), 74
replace() (gcc.Pass method), 74
restrict, 46
restrict_equivalent, 46
result (gcc.FunctionDecl attribute), 44
retval (gcc.GimpleReturn attribute), 68
revision (gcc.Version attribute), 115
rhs (gcc.GimpleAssign attribute), 67, 69
rhs (gcc.GimpleCall attribute), 68
rhs (gcc.GimpleCond attribute), 68
rtl (gcc.BasicBlock attribute), 38
ScopeRef (built-in class), 59
signed_equivalent (gcc.IntegerType attribute), 47
sizeof (gcc.Type attribute), 46
src (gcc.Edge attribute), 38
start (gcc.Function attribute), 37
start (gcc.Location attribute), 34
static (gcc.VarDecl attribute), 45
static_pass_number (gcc.Pass attribute), 72
str_no_uid (gcc.Gimple attribute), 65
str_no_uid (gcc.Tree attribute), 41
string (gcc.GimpleAsm attribute), 67
sub (gcc.Pass attribute), 71
succ (gcc.BasicBlock attribute), 38
target (gcc сайетLabelExpr attribute), 64
target (gcc.ComponentRef attribute), 58
TargetMemRef (built-in class), 59
text (gcc.Option attribute), 111
ture_label (gcc.GimpleCond attribute), 68
ture_value (gcc.Edge attribute), 39
type (gcc.Tree attribute), 41
unalias() (gcc.NamespaceDecl method), 46
UnconstrainedArrayRef (built-in class), 59
unqualified_equivalent, 46
unsigned (gcc.IntegerType attribute), 47
unsigned_equivalent (gcc.IntegerType attribute), 47
values (gcc.EnumerалType attribute), 48
var (gcc.SsaName attribute), 64
vars (gcc.Block attribute), 43
version (gcc.SsaName attribute), 64
ViewConvertExpr (built-in class), 59
volatile, 46
volatile_equivalent, 46
walk_tree() (gcc.Gimple method), 65

Index 153