gccjit Documentation

Release 0.4

David Malcolm

June 18, 2015

Contents

1 Tutorial **3**
- 1.1 Creating a trivial machine code function . . . 3
- 1.2 Loops and variables . . . 7
- 1.3 Implementing a “brainf” compiler . . . 14

2 Topic Reference **23**
- 2.1 Compilation contexts . . . 23
- 2.2 Types . . . 29
- 2.3 Expressions . . . 32
- 2.4 Functions . . . 35
- 2.5 Source Locations . . . 37
- 2.6 Compiling a context . . . 38

3 Indices and tables **41**

This document describes the Python bindings to libgccjit.

The bindings support both CPython 2 and CPython 3 (using Cython).

Note that both libgccjit and the bindings are of “Alpha” quality; the APIs are not yet set in stone, and they shouldn’t be used in production yet.

Contents:

CHAPTER 1

Tutorial

1.1 Creating a trivial machine code function

Consider this C function:

```
int square(int i)
{
  return i * i;
}
```

How can we construct this from within Python using libgccjit?

First we need to import the Python bindings to libgccjit:

```
>>> import gccjit
```

All state associated with compilation is associated with a `gccjit.Context`:

```
>>> ctxt = gccjit.Context()
```

The JIT library has a system of types. It is statically-typed: every expression is of a specific type, fixed at compile-time. In our example, all of the expressions are of the C *int* type, so let's obtain this from the context, as a `gccjit.Type`:

```
>>> int_type = ctxt.get_type(gccjit.TypeKind.INT)
```

The various objects in the API have reasonable __*str*__ methods:

```
>>> print(int_type)
int
```

Let's create the function. To do so, we first need to construct its single parameter, specifying its type and giving it a name:

```
>>> param_i = ctxt.new_param(int_type, b'i')
>>> print(param_i)
i
```

Now we can create the function:

```
>>> fn = ctxt.new_function(gccjit.FunctionKind.EXPORTED,
...                        int_type, # return type
...                        b"square", # name
...                        [param_i]) # params
>>> print(fn)
square
```

To define the code within the function, we must create basic blocks containing statements.

Every basic block contains a list of statements, eventually terminated by a statement that either returns, or jumps to another basic block.

Our function has no control-flow, so we just need one basic block:

```
>>> block = fn.new_block(b'entry')
>>> print(block)
entry
```

Our basic block is relatively simple: it immediately terminates by returning the value of an expression. We can build the expression:

```
>>> expr = ctxt.new_binary_op(gccjit.BinaryOp.MULT,
...                           int_type,
...                           param_i, param_i)
>>> print(expr)
i * i
```

This in itself doesn't do anything; we have to add this expression to a statement within the block. In this case, we use it to build a return statement, which terminates the basic block:

```
>>> block.end_with_return(expr)
```

OK, we've populated the context. We can now compile it:

```
>>> jit_result = ctxt.compile()
```

and get a `gccjit.Result`.

We can now look up a specific machine code routine within the result, in this case, the function we created above:

```
>>> void_ptr = jit_result.get_code(b"square")
```

We can now use ctypes.CFUNCTYPE to turn it into something we can call from Python:

```
>>> import ctypes
>>> int_int_func_type = ctypes.CFUNCTYPE(ctypes.c_int, ctypes.c_int)
>>> callable = int_int_func_type(void_ptr)
```

It should now be possible to run the code:

```
>>> callable(5)
25
```

1.1.1 Options

To get more information on what's going on, you can set debugging flags on the context using `gccjit.Context.set_bool_option()`.

Setting `gccjit.BoolOption.DUMP_INITIAL_GIMPLE` will dump a C-like representation to stderr when you compile (GCC's "GIMPLE" representation):

```
>>> ctxt.set_bool_option(gccjit.BoolOption.DUMP_INITIAL_GIMPLE, True)
>>> jit_result = ctxt.compile()
square (signed int i)
{
  signed int D.260;
  entry:
  D.260 = i * i;
  return D.260;
}
```

We can see the generated machine code in assembler form (on stderr) by setting *gccjit.BoolOption.DUMP_GENERATED_CODE* on the context before compiling:

```
>>> ctxt.set_bool_option(gccjit.BoolOption.DUMP_GENERATED_CODE, True)
>>> jit_result = ctxt.compile()
        .file   "fake.c"
        .text
        .globl  square
        .type   square, @function
square:
.LFB6:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    %edi, -4(%rbp)
.L14:
        movl    -4(%rbp), %eax
        imull   -4(%rbp), %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE6:
        .size   square, .-square
        .ident  "GCC: (GNU) 4.9.0 20131023 (Red Hat 0.2-0.5.1920c315ff984892399893b380305ab36e07b455
        .section       .note.GNU-stack,"",@progbits
```

By default, no optimizations are performed, the equivalent of GCC's *-O0* option. We can turn things up to e.g. *-O3* by calling *gccjit.Context.set_int_option()* with *gccjit.IntOption.OPTIMIZATION_LEVEL*:

```
>>> ctxt.set_int_option(gccjit.IntOption.OPTIMIZATION_LEVEL, 3)
>>> jit_result = ctxt.compile()
        .file   "fake.c"
        .text
        .p2align 4,,15
        .globl  square
        .type   square, @function
square:
.LFB7:
        .cfi_startproc
.L16:
        movl    %edi, %eax
        imull   %edi, %eax
        ret
        .cfi_endproc
.LFE7:
        .size   square, .-square
        .ident  "GCC: (GNU) 4.9.0 20131023 (Red Hat 0.2-0.5.1920c315ff984892399893b380305ab36e07b455
        .section        .note.GNU-stack,"",@progbits
```

Naturally this has only a small effect on such a trivial function.

1.1.2 Full example

Here's what the above looks like as a complete program:

```
import ctypes

import gccjit

def create_fn():
    # Create a compilation context:
    ctxt = gccjit.Context()

    # Turn these on to get various kinds of debugging:
    if 0:
        ctxt.set_bool_option(gccjit.BoolOption.DUMP_INITIAL_TREE, True)
        ctxt.set_bool_option(gccjit.BoolOption.DUMP_INITIAL_GIMPLE, True)
        ctxt.set_bool_option(gccjit.BoolOption.DUMP_GENERATED_CODE, True)

    # Adjust this to control optimization level of the generated code:
    if 0:
        ctxt.set_int_option(gccjit.IntOption.OPTIMIZATION_LEVEL, 3)

    int_type = ctxt.get_type(gccjit.TypeKind.INT)

    # Create parameter "i":
    param_i = ctxt.new_param(int_type, b'i')
    # Create the function:
    fn = ctxt.new_function(gccjit.FunctionKind.EXPORTED,
                           int_type,
                           b"square",
                           [param_i])

    # Create a basic block within the function:
    block = fn.new_block(b'entry')

    # This basic block is relatively simple:
    block.end_with_return(
        ctxt.new_binary_op(gccjit.BinaryOp.MULT,
                           int_type,
                           param_i, param_i))

    # Having populated the context, compile it.
    jit_result = ctxt.compile()

    # This is what you get back from ctxt.compile():
    assert isinstance(jit_result, gccjit.Result)

    return jit_result

def test_calling_fn(i):
    jit_result = create_fn()

    # Look up a specific machine code routine within the gccjit.Result,
    # in this case, the function we created above:
    void_ptr = jit_result.get_code(b"square")

    # Now use ctypes.CFUNCTYPE to turn it into something we can call
    # from Python:
    int_int_func_type = ctypes.CFUNCTYPE(ctypes.c_int, ctypes.c_int)
    code = int_int_func_type(void_ptr)

    # Now try running the code:
    return code(i)

if __name__ == '__main__':
    print(test_calling_fn(5))
```

1.2 Loops and variables

Consider this C function:

```
int loop_test (int n)
{
  int sum = 0;
  for (int i = 0; i < n; i++)
    sum += i * i;
  return sum;
}
```

This example demonstrates some more features of libgccjit, with local variables and a loop.

Let's construct this from Python. To break this down into libgccjit terms, it's usually easier to reword the *for* loop as a *while* loop, giving:

```
int loop_test (int n)
{
  int sum = 0;
  int i = 0;
  while (i < n)
  {
    sum += i * i;
    i++;
  }
  return sum;
}
```

Here's what the final control flow graph will look like:

As before, we import the libgccjit Python bindings and make a *gccjit.Context*:

```
>>> import gccjit
>>> ctxt = gccjit.Context()
```

The function works with the C *int* type:

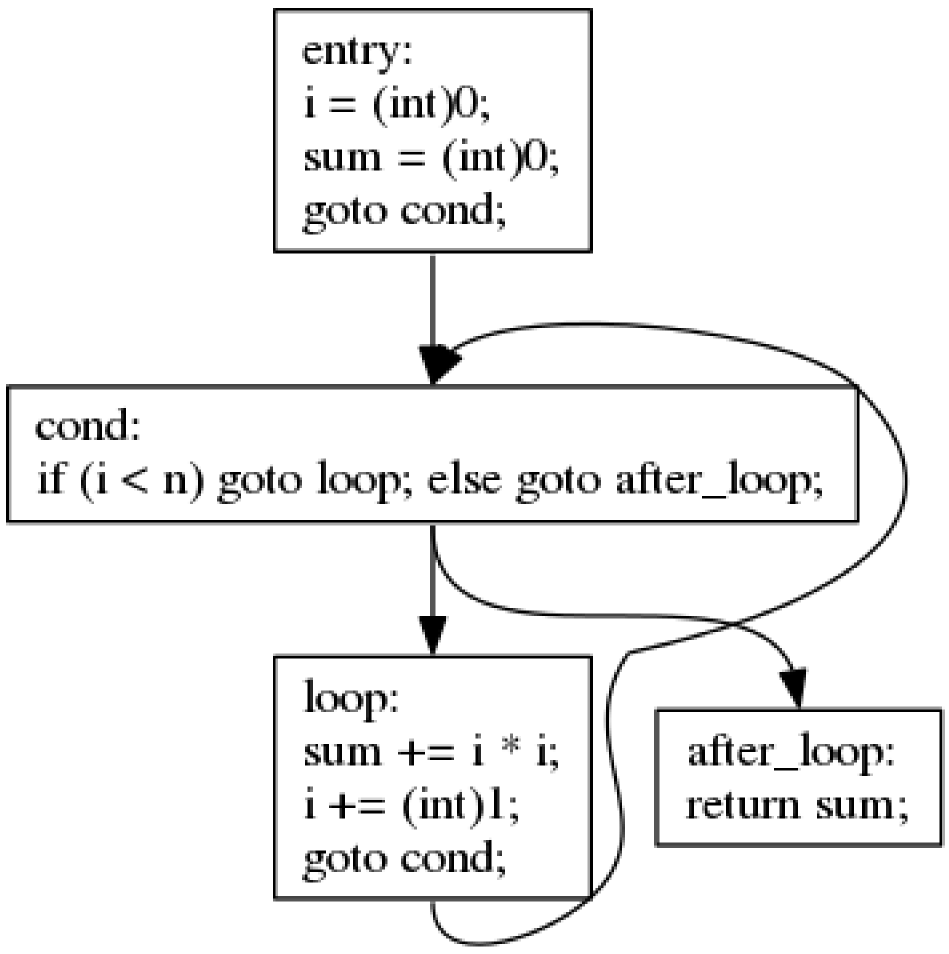
```
>>> the_type = ctxt.get_type(gccjit.TypeKind.INT)
```

though we could equally well make it work on, say, *double*:

```
>>> the_type = ctxt.get_type(gccjit.TypeKind.DOUBLE)
```

Let's build the function:

```
>>> return_type = the_type
>>> param_n = ctxt.new_param(the_type, b"n")
```


entry:
i = (int)0;
sum = (int)0;
goto cond;
cond:
if (i < n) goto loop; else goto after_loop;
loop:
sum += i * i;
i += (int)1;
goto cond;
after_loop:
return sum;

```
>>> fn = ctxt.new_function(gccjit.FunctionKind.EXPORTED,
...                        return_type,
...                        b"loop_test",
...                        [param_n])
>>> print(fn)
loop_test
```

The base class of expression is the `gccjit.RValue`, representing an expression that can be on the *right*-hand side of an assignment: a value that can be computed somehow, and assigned *to* a storage area (such as a variable). It has a specific `gccjit.Type`.

Anothe important class is `gccjit.LValue`. A `gccjit.LValue` is something that can of the *left*-hand side of an assignment: a storage area (such as a variable).

In other words, every assignment can be thought of as:

```
    LVALUE = RVALUE;
```

Note that `gccjit.LValue` is a subclass of `gccjit.RValue`, where in an assignment of the form:

```
    LVALUE_A = LVALUE_B;
```

the *LVALUE_B* implies reading the current value of that storage area, assigning it into the *LVALUE_A*.

So far the only expressions we've seen are *i * i*:

```
ctxt.new_binary_op(gccjit.BinaryOp.MULT,
                   int_type,
                   param_i, param_i)
```

which is a `gccjit.RValue`, and the various function parameters: *param_i* and *param_n*, instances of `gccjit.Param`, which is a subclass of `gccjit.LValue` (and, in turn, of `gccjit.RValue`): we can both read from and write to function parameters within the body of a function.

Our new example has a couple of local variables. We create them by calling `gccjit.Function.new_local()`, supplying a type and a name:

```
>>> local_i = fn.new_local(the_type, b"i")
>>> print(local_i)
i
>>> local_sum = fn.new_local(the_type, b"sum")
>>> print(local_sum)
sum
```

These are instances of `gccjit.LValue` - they can be read from and written to.

Note that there is no precanned way to create *and* initialize a variable like in C:

```
int i = 0;
```

Instead, having added the local to the function, we have to separately add an assignment of *0* to *local_i* at the beginning of the function.

This function has a loop, so we need to build some basic blocks to handle the control flow. In this case, we need 4 blocks:

1. before the loop (initializing the locals)
2. the conditional at the top of the loop (comparing *i* < *n*)
3. the body of the loop
4. after the loop terminates (*return sum*)

so we create these as *gccjit.Block* instances within the *gccjit.Function*:

```
>>> entry_block = fn.new_block(b'entry')
>>> cond_block = fn.new_block(b"cond")
>>> loop_block = fn.new_block(b"loop")
>>> after_loop_block = fn.new_block(b"after_loop")
```

We now populate each block with statements.

The entry block consists of initializations followed by a jump to the conditional. We assign *0* to *i* and to *sum*, using *gccjit.Block.add_assignment()* to add an assignment statement, and using *gccjit.Context.zero()* to get the constant value *0* for the relevant type for the right-hand side of the assignment:

```
>>> entry_block.add_assignment(local_i, ctxt.zero(the_type))
>>> entry_block.add_assignment(local_sum, ctxt.zero(the_type))
```

We can then terminate the entry block by jumping to the conditional:

```
>>> entry_block.end_with_jump(cond_block)
```

The conditional block is equivalent to the line *while (i < n)* from our C example. It contains a single statement: a conditional, which jumps to one of two destination blocks depending on a boolean *gccjit.RValue*, in this case the comparison of *i* and *n*. We build the comparison using *gccjit.Context.new_comparison()*:

```
>>> guard = ctxt.new_comparison(gccjit.Comparison.LT, local_i, param_n)
>>> print(guard)
i < n
```

and can then use this to add *cond_block*'s sole statement, via *gccjit.Block.end_with_conditional()*:

```
>>> cond_block.end_with_conditional(guard,
...                                 loop_block, # on true
...                                 after_loop_block) # on false
```

Next, we populate the body of the loop.

The C statement *sum += i * i;* is an assignment operation, where an lvalue is modified "in-place". We use *gccjit.Block.add_assignment_op()* to handle these operations:

```
>>> loop_block.add_assignment_op(local_sum,
...                              gccjit.BinaryOp.PLUS,
...                              ctxt.new_binary_op(gccjit.BinaryOp.MULT,
...                                                 the_type,
...                                                 local_i, local_i))
```

The *i++* can be thought of as *i += 1*, and can thus be handled in a similar way. We use *gccjit.Context.one()* to get the constant value *1* (for the relevant type) for the right-hand side of the assignment:

```
>>> loop_block.add_assignment_op(local_i,
...                              gccjit.BinaryOp.PLUS,
...                              ctxt.one(the_type))
```

The loop body completes by jumping back to the conditional:

```
>>> loop_block.end_with_jump(cond_block)
```

Finally, we populate the *after_loop* block, reached when the loop conditional is false. At the C level this is simply:

```
return sum;
```

so the block is just one statement:

```
>>> after_loop_block.end_with_return(local_sum)
```

Note: You can intermingle block creation with statement creation, but given that the terminator statements generally include references to other blocks, I find it's clearer to create all the blocks, *then* all the statements.

We've finished populating the function. As before, we can now compile it to machine code:

```
>>> jit_result = ctxt.compile()
>>> void_ptr = jit_result.get_code(b'loop_test')
```

and use *ctypes* to turn it into a Python callable:

```
>>> import ctypes
>>> int_int_func_type = ctypes.CFUNCTYPE(ctypes.c_int, ctypes.c_int)
>>> callable = int_int_func_type(void_ptr)
```

Now we can call it:

```
>>> callable(10)
285
```

1.2.1 Visualizing the control flow graph

You can see the control flow graph of a function using `gccjit.Function.dump_to_dot()`:

```
>>> fn.dump_to_dot('/tmp/sum-of-squares.dot')
```

giving a .dot file in GraphViz format.

You can convert this to an image using *dot*:

```
$ dot -Tpng /tmp/sum-of-squares.dot -o /tmp/sum-of-squares.png
```

or use a viewer (my preferred one is xdot.py; see https://github.com/jrfonseca/xdot.py; on Fedora you can install it with *yum install python-xdot*):

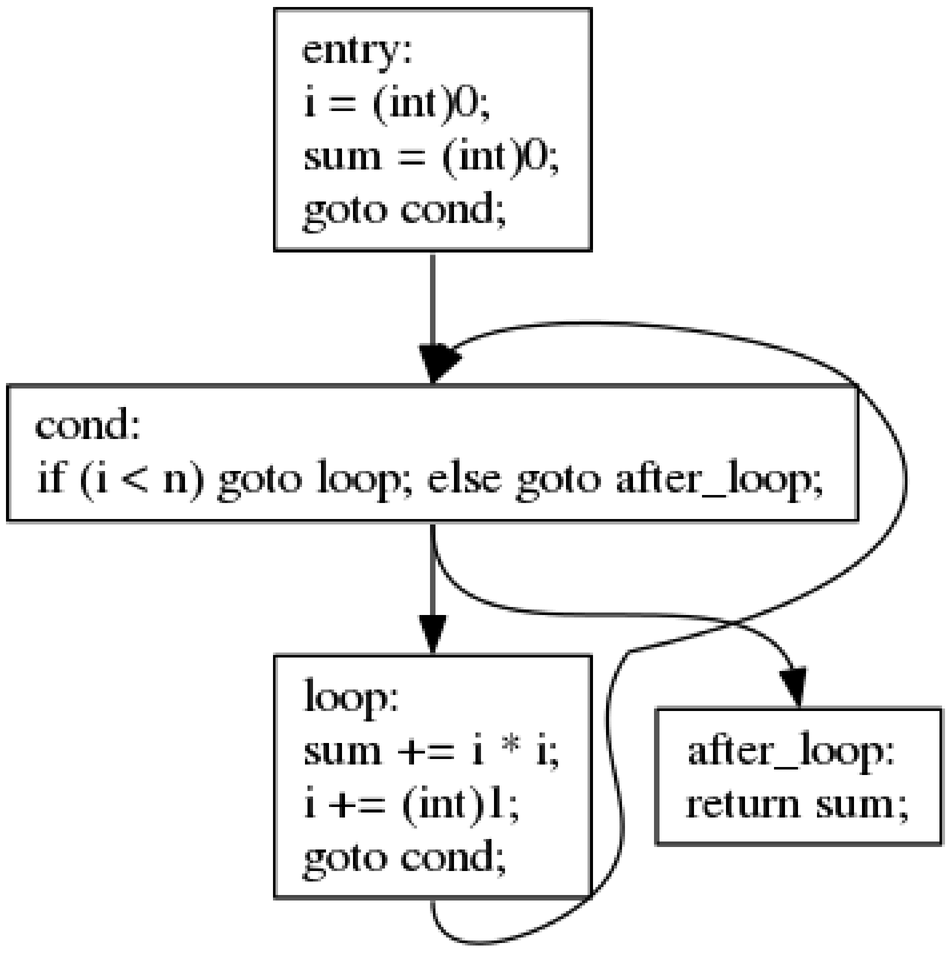
1.2.2 Full example

Here's what the above looks like as a complete program:

```
import ctypes

import gccjit

def populate_ctxt(ctxt):
    the_type = ctxt.get_type(gccjit.TypeKind.INT)
    return_type = the_type
    param_n = ctxt.new_param(the_type, b"n")
    fn = ctxt.new_function(gccjit.FunctionKind.EXPORTED,
                           return_type,
                           b"loop_test",
                           [param_n])
    # Build locals
    local_i = fn.new_local(the_type, b"i")
    local_sum = fn.new_local(the_type, b"sum")
```


entry:
i = (int)0;
sum = (int)0;
goto cond;
cond:
if (i < n) goto loop; else goto after_loop;
loop:
sum += i * i;
i += (int)1;
goto cond;
after_loop:
return sum;

```
        assert str(local_i) == 'i'

        # Build blocks
        entry_block = fn.new_block(b'entry')
        cond_block = fn.new_block(b"cond")
        loop_block = fn.new_block(b"loop")
        after_loop_block = fn.new_block(b"after_loop")

        # entry_block: #########################################

        # sum = 0
        entry_block.add_assignment(local_sum, ctxt.zero(the_type))

        # i = 0
        entry_block.add_assignment(local_i, ctxt.zero(the_type))

        entry_block.end_with_jump(cond_block)

        ### cond_block: ########################################

        # while (i < n)
        cond_block.end_with_conditional(ctxt.new_comparison(gccjit.Comparison.LT,
                                                            local_i, param_n),
                                        loop_block,
                                        after_loop_block)

        ### loop_block: ########################################

        # sum += i * i
        loop_block.add_assignment_op(local_sum,
                                     gccjit.BinaryOp.PLUS,
                                     ctxt.new_binary_op(gccjit.BinaryOp.MULT,
                                                        the_type,
                                                        local_i, local_i))

        # i++
        loop_block.add_assignment_op(local_i,
                                     gccjit.BinaryOp.PLUS,
                                     ctxt.one(the_type))

        # goto cond_block
        loop_block.end_with_jump(cond_block)

        ### after_loop_block: ##################################

        # return sum
        after_loop_block.end_with_return(local_sum)

    def create_fn():
        # Create a compilation context:
        ctxt = gccjit.Context()

        if 0:
            ctxt.set_bool_option(gccjit.BoolOption.DUMP_INITIAL_TREE, True)
            ctxt.set_bool_option(gccjit.BoolOption.DUMP_INITIAL_GIMPLE, True)
            ctxt.set_bool_option(gccjit.BoolOption.DUMP_EVERYTHING, True)
            ctxt.set_bool_option(gccjit.BoolOption.KEEP_INTERMEDIATES, True)
        if 0:
            ctxt.set_int_option(gccjit.IntOption.OPTIMIZATION_LEVEL, 3)

        populate_ctxt(ctxt)

        jit_result = ctxt.compile()
        return jit_result

    def test_calling_fn(i):
        jit_result = create_fn()

        int_int_func_type = ctypes.CFUNCTYPE(ctypes.c_int, ctypes.c_int)
        code = int_int_func_type(jit_result.get_code(b"loop_test"))

        return code(i)

    if __name__ == '__main__':
        print(test_calling_fn(10))
```

1.3 Implementing a "brainf" compiler

In this example we use libgccjit to construct a compiler for an esoteric programming language that we shall refer to as "brainf".

The compiler can run the generated code in-process (JIT compilation), or write the generated code as a machine code executable (classic ahead-of-time compilation).

1.3.1 The "brainf" language

brainf scripts operate on an array of bytes, with a notional data pointer within the array.

brainf is hard for humans to read, but it's trivial to write a parser for it, as there is no lexing; just a stream of bytes. The operations are:

Character	Meaning
>	`idx += 1`
<	`idx -= 1`
+	`data[idx] += 1`
-	`data[idx] -= 1`
.	`output (data[idx])`
,	`data[idx] = input ()`
[	loop until `data[idx] == 0`
]	end of loop
Anything else	ignored

Unlike the previous example, we'll implement an ahead-of-time compiler, which reads `.bf` scripts and outputs executables (though it would be trivial to have it run them JIT-compiled in-process).

Here's what a simple `.bf` script looks like:

```
    [
      Emit the uppercase alphabet
    ]

    cell 0 = 26
    ++++++++++++++++++++++++++
    cell 1 = 65
    >+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++<

    while cell#0 != 0
    [
     >
     .      emit cell#1
     +      increment cell@1
     <-     decrement cell@0
    ]
```

Note: This example makes use of whitespace and comments for legibility, but could have been written as:

```
++++++++++++++++++++++++++
>+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++<
[>.+<-]
```

It's not a particularly useful language, except for providing compiler-writers with a test case that's easy to parse.

1.3.2 Converting a brainf script to libgccjit IR

We write simple code to populate a `gccjit.Context`.

```
import datetime
from contextlib import contextmanager

@contextmanager
def timer(desc):
    start = datetime.datetime.now()
    yield
    stop = datetime.datetime.now()
    print('%s: %ss' % (desc, (stop - start).total_seconds()))

class Paren:
    def __init__(self, b_test, b_body, b_after):
        self.b_test = b_test
        self.b_body = b_body
        self.b_after = b_after

class CompileError(Exception):
    def __init__(self, compiler, msg):
        self.filename = compiler.filename
        self.line = compiler.line
        self.column = compiler.column
        self.msg = msg

    def __str__(self):
        return ("%s:%i:%i: %s"
                % (self.filename, self.line, self.column, self.msg))

class Compiler:
    def __init__(self):
        self.ctxt = gccjit.Context()
        if 1:
            self.ctxt.set_int_option(gccjit.IntOption.OPTIMIZATION_LEVEL,
                                     3);
            self.ctxt.set_bool_option(gccjit.BoolOption.DUMP_INITIAL_GIMPLE,
                                      0);
            self.ctxt.set_bool_option(gccjit.BoolOption.DUMP_GENERATED_CODE,
                                      0);
            self.ctxt.set_bool_option(gccjit.BoolOption.DEBUGINFO,
                                      1);
            self.ctxt.set_bool_option(gccjit.BoolOption.DUMP_EVERYTHING,
                                      0);
            self.ctxt.set_bool_option(gccjit.BoolOption.KEEP_INTERMEDIATES,
                                      0);
        self.void_type = self.ctxt.get_type(gccjit.TypeKind.VOID)
        self.int_type = self.ctxt.get_type(gccjit.TypeKind.INT)
        self.byte_type = self.ctxt.get_type(gccjit.TypeKind.UNSIGNED_CHAR)
        self.array_type = self.ctxt.new_array_type(self.byte_type,
                                                   30000)
        self.func_getchar = (
            self.ctxt.new_function(gccjit.FunctionKind.IMPORTED,
                                   self.int_type,
                                   b"getchar", []))
        self.func_putchar = (
            self.ctxt.new_function(gccjit.FunctionKind.IMPORTED,
                                   self.void_type,
                                   b"putchar",
                                   [self.ctxt.new_param(self.int_type,
                                                        b"c")]))
        self.func = self.ctxt.new_function(gccjit.FunctionKind.EXPORTED,
                                           self.void_type, b'func', [])
        self.curblock = self.func.new_block(b"initial")
        self.int_zero = self.ctxt.zero(self.int_type)
        self.int_one = self.ctxt.one(self.int_type)
        self.byte_zero = self.ctxt.zero(self.byte_type)
        self.byte_one = self.ctxt.one(self.byte_type)
        self.data_cells = self.ctxt.new_global(gccjit.GlobalKind.INTERNAL,
                                               self.array_type,
                                               b"data_cells")
        self.idx = self.func.new_local(self.int_type,
                                       b"idx")

        self.open_parens = []

        self.curblock.add_comment(b"idx = 0;")
        self.curblock.add_assignment(self.idx,
                                     self.int_zero)

    def get_current_data(self, loc):
        """Get 'data_cells[idx]' as an lvalue. """
        return self.ctxt.new_array_access(self.data_cells,
                                          self.idx,
                                          loc)


    def current_data_is_zero(self, loc):
        """Get 'data_cells[idx] == 0' as a boolean rvalue."""
        return self.ctxt.new_comparison(gccjit.Comparison.EQ,
                                        self.get_current_data(loc),
                                        self.byte_zero,
                                        loc)
    def compile_char(self, ch):
        """Compile one bf character."""
        loc = self.ctxt.new_location(self.filename,
                                     self.line,
                                     self.column)

        # Turn this on to trace execution, by injecting putchar()
        # of each source char.
        if 0:
            arg = self.ctxt.new_rvalue_from_int (self.int_type,
                                                 ch)
            call = self.ctxt.new_call (self.func_putchar,
                                       [arg],
                                       loc)
            self.curblock.add_eval (call, loc)

        if ch == '>':
            self.curblock.add_comment(b"'>': idx += 1;", loc)
            self.curblock.add_assignment_op(self.idx,
                                            gccjit.BinaryOp.PLUS,
                                            self.int_one,
                                            loc)
        elif ch == '<':
            self.curblock.add_comment(b"'<': idx -= 1;", loc)
            self.curblock.add_assignment_op(self.idx,
                                            gccjit.BinaryOp.MINUS,
                                            self.int_one,
                                            loc)
        elif ch == '+':
            self.curblock.add_comment(b"'+': data[idx] += 1;", loc)
            self.curblock.add_assignment_op(self.get_current_data (loc),
                                            gccjit.BinaryOp.PLUS,
                                            self.byte_one,
                                            loc)
        elif ch == '-':
            self.curblock.add_comment(b"'-': data[idx] -= 1;", loc)
            self.curblock.add_assignment_op(self.get_current_data(loc),
                                            gccjit.BinaryOp.MINUS,
                                            self.byte_one,
                                            loc)
        elif ch == '.':
            arg = self.ctxt.new_cast(self.get_current_data(loc),
                                     self.int_type,
                                     loc)
            call = self.ctxt.new_call(self.func_putchar,
                                      [arg],
                                      loc)
            self.curblock.add_comment(b"'.': putchar ((int)data[idx]);",
                                      loc)
            self.curblock.add_eval(call, loc)
        elif ch == ',':
            call = self.ctxt.new_call(self.func_getchar, [], loc)
            self.curblock.add_comment(b"',': data[idx] = (unsigned char)getchar ();",
                                      loc)
            self.curblock.add_assignment(self.get_current_data(loc),
                                         self.ctxt.new_cast(call,
                                                            self.byte_type,
                                                            loc),
                                         loc)
        elif ch == '[':
            loop_test = self.func.new_block()
            on_zero = self.func.new_block()
            on_non_zero = self.func.new_block()

            self.curblock.end_with_jump(loop_test, loc)

            loop_test.add_comment(b"'['", loc)
            loop_test.end_with_conditional(self.current_data_is_zero(loc),
                                           on_zero,
                                           on_non_zero,
                                           loc)
            self.open_parens.append(Paren(loop_test, on_non_zero, on_zero))
            self.curblock = on_non_zero;
        elif ch == ']':
            self.curblock.add_comment(b"']'", loc)
            if not self.open_parens:
                raise CompileError(self, "mismatching parens")
            paren = self.open_parens.pop()
            self.curblock.end_with_jump(paren.b_test)
            self.curblock = paren.b_after
        elif ch == '\n':
            self.line +=1;
            self.column = 0;

        if ch != '\n':
            self.column += 1


    def parse_into_ctxt(self, filename):
        """
        Parse the given .bf file into the gccjit.Context, containing a
        single function "func".
        """
        self.filename = filename;
        self.line = 1
        self.column = 0
        with open(filename) as f_in:
            for ch in f_in.read():
                self.compile_char(ch)
        self.curblock.end_with_void_return()

    # Compiling to an executable
```

1.3.3 Compiling a context to a file

In previous examples, we compiled and ran the generated machine code in-process. We can do that:

```
        def run(self):
            import ctypes
            with timer("compiling"):
                result = self.ctxt.compile()
            py_func_type = ctypes.CFUNCTYPE(None)
            py_func = py_func_type(result.get_code(b'func'))
            with timer("running"):
                py_func()
```

but this time we'll also provide a way to compile the context directly to an executable, using *gccjit.Context.compile_to_file()*.

To do so, we need to export a `main` function. A helper function for doing so is provided by the JIT API:

```
    def make_main(ctxt):
        """
        Make "main" function:
          int
          main (int argc, char **argv)
          {
          ...
          }
        Return (func, param_argc, param_argv)
        """
        int_type = ctxt.get_type(TypeKind.INT)
        param_argc = ctxt.new_param(int_type, b"argc")
        char_ptr_ptr_type = (
            ctxt.get_type(TypeKind.CHAR).get_pointer().get_pointer())
        param_argv = ctxt.new_param(char_ptr_ptr_type, b"argv")
        func_main = ctxt.new_function(FunctionKind.EXPORTED,
                                      int_type,
                                      b"main",
                                      [param_argc, param_argv])
        return (func_main, param_argc, param_argv)
```

which we can use (as `gccjit.make_main`) to compile the function to an executable:

```
        def compile_to_file(self, output_path):
            # Wrap "func" up in a "main" function
            mainfunc, argv, argv = gccjit.make_main(self.ctxt)
            block = mainfunc.new_block()
            block.add_eval(self.ctxt.new_call(self.func, []))
            block.end_with_return(self.int_zero)
            self.ctxt.compile_to_file(gccjit.OutputKind.EXECUTABLE,
                                      output_path)
```

Finally, here's the top-level of the program:

```
    def main(argv):
        from optparse import OptionParser
        parser = OptionParser()
        parser.add_option("-o", "--output", dest="outputfile",
                          help="compile to FILE", metavar="FILE")
        (options, args) = parser.parse_args()
        if len(args) != 1:
            raise ValueError('No input file')
        inputfile = args[0]
        c = Compiler()
        with timer("total"):
            with timer("parsing"):
                c.parse_into_ctxt(inputfile)
            if options.outputfile:
                c.compile_to_file(options.outputfile)
            else:
                c.run()

    if __name__ == '__main__':
        try:
            main(sys.argv)
        except Exception as exc:
            print(exc)
            sys.exit(1)
```

The overall script *examples/bf.py* is thus a bf-to-machine-code compiler, which we can use to compile .bf files, either to run in-process,

```
$ PYTHONPATH=. python examples/bf.py \
     emit-alphabet.bf
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

or to compile into machine code executables:

```
$ PYTHONPATH=. python examples/bf.py \
     emit-alphabet.bf \
     -o a.out
```

which we can run independently:

```
$ ./a.out
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Success!

We can also inspect the generated executable using standard tools:

```
$ objdump -d a.out |less
```

which shows that libgccjit has managed to optimize the function somewhat (for example, the runs of 26 and 65 increment operations have become integer constants 0x1a and 0x41):

```
0000000000400620 <func>:
  400620:       80 3d 39 0a 20 00 00    cmpb   $0x0,0x200a39(%rip)        # 601060 <data_cells>
  400627:       74 07                   je     400630 <func+0x10>
  400629:       eb fe                   jmp    400629 <func+0x9>
  40062b:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
  400630:       48 83 ec 08             sub    $0x8,%rsp
  400634:       0f b6 05 26 0a 20 00    movzbl 0x200a26(%rip),%eax        # 601061 <data_cells+0x1>
  40063b:       c6 05 1e 0a 20 00 1a    movb   $0x1a,0x200a1e(%rip)        # 601060 <data_cells>
  400642:       8d 78 41                lea    0x41(%rax),%edi
  400645:       40 88 3d 15 0a 20 00    mov    %dil,0x200a15(%rip)        # 601061 <data_cells+0x1>
  40064c:       0f 1f 40 00             nopl   0x0(%rax)
  400650:       40 0f b6 ff             movzbl %dil,%edi
  400654:       e8 87 fe ff ff          callq  4004e0 <putchar@plt>
  400659:       0f b6 05 01 0a 20 00    movzbl 0x200a01(%rip),%eax        # 601061 <data_cells+0x1>
  400660:       80 2d f9 09 20 00 01    subb   $0x1,0x2009f9(%rip)        # 601060 <data_cells>
  400667:       8d 78 01                lea    0x1(%rax),%edi
  40066a:       40 88 3d f0 09 20 00    mov    %dil,0x2009f0(%rip)        # 601061 <data_cells+0x1>
  400671:       75 dd                   jne    400650 <func+0x30>
  400673:       48 83 c4 08             add    $0x8,%rsp
  400677:       c3                      retq
  400678:       0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
  40067f:       00

0000000000400680 <main>:
  400680:       48 83 ec 08             sub    $0x8,%rsp
  400684:       e8 97 ff ff ff          callq  400620 <func>
  400689:       31 c0                   xor    %eax,%eax
  40068b:       48 83 c4 08             add    $0x8,%rsp
  40068f:       c3                      retq
```

We also set up debugging information (via *gccjit.Context.new_location()* and *gccjit.BoolOption.DEBUGINFO*), so it's possible to use `gdb` to singlestep through the generated binary and inspect the internal state `idx` and `data_cells`:

```
(gdb) break main
Breakpoint 1 at 0x400790
(gdb) run
Starting program: a.out

Breakpoint 1, 0x0000000000400790 in main (argc=1, argv=0x7fffffffe448)
(gdb) stepi
0x0000000000400797 in main (argc=1, argv=0x7fffffffe448)
(gdb) stepi
0x00000000004007a0 in main (argc=1, argv=0x7fffffffe448)
(gdb) stepi
9     >+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++<
(gdb) list
4
5     cell 0 = 26
6     ++++++++++++++++++++++++++
7
8     cell 1 = 65
9     >+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++<
10
11    while cell#0 != 0
12    [
13     >
(gdb) n
6     ++++++++++++++++++++++++++
(gdb) n
9     >+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++<
(gdb) p idx
$1 = 1
(gdb) p data_cells
$2 = "\032", '\000' <repeats 29998 times>
(gdb) p data_cells[0]
$3 = 26 '\032'
(gdb) p data_cells[1]
$4 = 0 '\000'
(gdb) list
4
5     cell 0 = 26
6     ++++++++++++++++++++++++++
7
8     cell 1 = 65
9     >+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++<
10
11    while cell#0 != 0
12    [
13     >
```

1.3.4 Other forms of ahead-of-time-compilation

The above demonstrates compiling a *gccjit.Context* directly to an executable. It's also possible to compile it to an object file, and to a dynamic library. See the documentation of *gccjit.Context.compile_to_file()* for more information.

CHAPTER 2

Topic Reference

2.1 Compilation contexts

class `gccjit.`**`Context`**

The top-level of the API is the *gccjit.Context* class.

A *gccjit.Context* instance encapsulates the state of a compilation.

You can set up options on it, and add types, functions and code. Invoking `gccjit.Context.compile()` on it gives you a `gccjit.Result`.

`dump_to_file`(*path*, *update_locations*)

`get_first_error`()

`new_location`(*filename*, *line*, *column*)

Make a `gccjit.Location` representing a source location, for use by the debugger:

```
loc = ctxt.new_location('web.js', 5, 0)
```

Note: You need to enable `gccjit.BoolOption.DEBUGINFO` on the context for these locations to actually be usable by the debugger:

```
ctxt.set_bool_option(gccjit.BoolOption.DEBUGINFO, True)
```

Return type `gccjit.Location`

`new_global`(*Type type_*, *name*, *Location loc=None*)

Return type `gccjit.LValue`

`new_array_type`(*Type element_type*, *int num_elements*, *Location loc=None*)

Return type `gccjit.Type`

`new_field`(*Type type_*, *name*, *Location loc=None*)

Return type `gccjit.Field`

`new_struct`(*name*, *fields=None*, *Location loc=None*)

Return type `gccjit.Struct`

`new_union`(*name*, *fields=None*, *Location loc=None*)

Construct a new “union” type.

Return type `gccjit.Type`

Parameters

- **field** – The fields that make up the union
- **loc** (*gccjit.Location*) – The source location, if any, or None

For example, to create the equivalent of:

```
union u
{
  int as_int;
  float as_float;
};
```

you can use:

```
ctxt = gccjit.Context()
int_type = ctxt.get_type(gccjit.TypeKind.INT)
float_type = ctxt.get_type(gccjit.TypeKind.FLOAT)
as_int = ctxt.new_field(int_type, b'as_int')
as_float = ctxt.new_field(float_type, b'as_float')
u = ctxt.new_union(b'u', [as_int, as_float])
```

new_function_ptr_type(*return_type*, *param_types*, *loc=None*, *is_variadic=False*)

Parameters

- **return_type** (*gccjit.Type*) – The return type of the function
- **param_types** (A sequence of *gccjit.Type*) – The types of the parameters
- **loc** (*gccjit.Location*) – The source location, if any, or None
- **is_variadic** (bool) – Is the function variadic (i.e. accepts a variable number of arguments)

Return type *gccjit.Type*

For example, to create the equivalent of:

```
typedef void (*fn_ptr_type) (int, int int);
```

you can use:

```
>>> ctxt = gccjit.Context()
>>> void_type = ctxt.get_type(gccjit.TypeKind.VOID)
>>> int_type = ctxt.get_type(gccjit.TypeKind.INT)
>>> fn_ptr_type = ctxt.new_function_ptr_type (void_type,
                                              [int_type,
                                               int_type,
                                               int_type])
>>> print(fn_ptr_type)
void (*) (int, int, int)
```

new_param(*Type type_*, *name*, *Location loc=None*)

Return type *gccjit.Param*

new_function(*kind*, *Type return_type*, *name*, *params*, *Location loc=None*, *is_variadic=False*)

Return type *gccjit.Function*

get_builtin_function(*name*)

Return type *gccjit.Function*

zero(*type_*)

Given a *gccjit.Type*, which must be a numeric type, get the constant 0 as a *gccjit.RValue* of that type.

Return type *gccjit.RValue*

one (*type_*)

Given a *gccjit.Type*, which must be a numeric type, get the constant 1 as a *gccjit.RValue* of that type.

Return type *gccjit.RValue*

new_rvalue_from_double (*numeric_type*, *value*)

Given a *gccjit.Type*, which must be a numeric type, get a floating-point constant as a *gccjit.RValue* of that type.

Return type *gccjit.RValue*

new_rvalue_from_int (*type_*, *value*)

Given a *gccjit.Type*, which must be a numeric type, get an integer constant as a *gccjit.RValue* of that type.

Return type *gccjit.RValue*

new_rvalue_from_ptr (*pointer_type*, *value*)

Given a *gccjit.Type*, which must be a pointer type, and an address, get a *gccjit.RValue* representing that address as a pointer of that type:

```
ptr = ctxt.new_rvalue_from_ptr(int_star, 0xDEADBEEF)
```

Return type *gccjit.RValue*

null (*pointer_type*)

Given a *gccjit.Type*, which must be a pointer type, get a *gccjit.RValue* representing the *NULL* pointer of that type:

```
ptr = ctxt.null(int_star)
```

Return type *gccjit.RValue*

new_string_literal (*value*)

Make a *gccjit.RValue* for the given string literal value (actually bytes):

```
msg = ctxt.new_string_literal(b'hello world\n')
```

Parameters **value** (*bytes*) – the bytes of the string literal

Return type *gccjit.RValue*

new_unary_op (*op*, *result_type*, *rvalue*, *loc=None*)

Make a *gccjit.RValue* for the given unary operation.

Parameters

- **op** (*gccjit.UnaryOp*) – Which unary operation
- **result_type** (*gccjit.Type*) – The type of the result
- **rvalue** (*gccjit.RValue*) – The input expression
- **loc** (*gccjit.Location*) – The source location, if any, or None

Return type *gccjit.RValue*

new_binary_op (*op*, *result_type*, *a*, *b*, *loc=None*)

Make a *gccjit.RValue* for the given binary operation.

Parameters

- **op** (*gccjit.BinaryOp*) – Which binary operation
- **result_type** (*gccjit.Type*) – The type of the result
- **a** (*gccjit.RValue*) – The first input expression
- **b** (*gccjit.RValue*) – The second input expression
- **loc** (*gccjit.Location*) – The source location, if any, or None

Return type *gccjit.RValue*

new_comparison (*op*, *a*, *b*, *loc=None*)
Make a `gccjit.RValue` of boolean type for the given comparison.

Parameters

- **op** (`gccjit.Comparison`) – Which comparison
- **a** (`gccjit.RValue`) – The first input expression
- **b** (`gccjit.RValue`) – The second input expression
- **loc** (`gccjit.Location`) – The source location, if any, or None

Return type `gccjit.RValue`

new_child_context (*self*)

Return type `gccjit.Context`

new_cast (*RValue rvalue*, *Type type_*, *Location loc=None*)

Return type `gccjit.RValue`

new_array_access (*ptr*, *index*, *loc=None*)

Parameters

- **ptr** (`gccjit.RValue`) – The pointer or array
- **index** (`gccjit.RValue`) – The index within the array
- **loc** (`gccjit.Location`) – The source location, if any, or None

Return type `gccjit.LValue`

new_call (*Function func*, *args*, *Location loc=None*)

Return type `gccjit.RValue`

new_call_through_ptr (*fn_ptr*, *args*, *loc=None*)

Parameters

- **fn_ptr** (`gccjit.RValue`) – A function pointer
- **args** (A sequence of `gccjit.RValue`) – The arguments to the function call
- **loc** (`gccjit.Location`) – The source location, if any, or None

Return type `gccjit.RValue`

For example, to create the equivalent of:

```
typedef void (*fn_ptr_type) (int, int, int);
fn_ptr_type fn_ptr;

fn_ptr (a, b, c);
```

you can use:

```
block.add_eval (ctxt.new_call_through_ptr(fn_ptr, [a, b, c]))
```

2.1.1 Debugging

`gccjit.Context.`**dump_reproducer_to_file** (*self*, *path*)
Write C source code into *path* that can be compiled into a self-contained executable (i.e. with libgccjit as the only dependency). The generated code will attempt to replay the API calls that have been made into the given context, at the C level, eliminating any dependency on Python or on client code or data.

This may be useful when debugging the library or client code, for reducing a complicated recipe for reproducing a bug into a simpler form.

Typically you need to supply `-Wno-unused-variable` when compiling the generated file (since the result of each API call is assigned to a unique variable within the generated C source, and not all are necessarily then used).

`gccjit.Context.`**`set_logfile`**(*self*, *f*)

To help with debugging; enable ongoing logging of the context's activity to the given file object.

For example, the following will enable logging to stderr:

```
ctxt.set_logfile(sys.stderr)
```

Examples of information logged include:

- API calls
- the various steps involved within compilation
- activity on any *`gccjit.Result`* instances created by the context
- activity within any child contexts

The precise format and kinds of information logged is subject to change.

Unfortunately, doing so creates a leak of an underlying `FILE *` object.

There may a performance cost for logging.

2.1.2 Options

String options

`gccjit.Context.`**`set_str_option`**(*self*, *opt*, *val*)

Set a string option of the context; see *`gccjit.StrOption`* for notes on the options and their meanings.

Parameters

- **`opt`** (*`gccjit.StrOption`*) – Which option to set
- **`val`** (*str*) – The new value

class `gccjit.`**`StrOption`**

`PROGNAME`

The name of the program, for use as a prefix when printing error messages to stderr. If *None*, or default, "libgccjit.so" is used.

Boolean options

`gccjit.Context.`**`set_bool_option`**(*self*, *opt*, *val*)

Set a boolean option of the context; see *`gccjit.BoolOption`* for notes on the options and their meanings.

Parameters

- **`opt`** (*`gccjit.BoolOption`*) – Which option to set
- **`val`** (*str*) – The new value

class `gccjit.`**`BoolOption`**

`DEBUGINFO`

If true, *`gccjit.Context.compile()`* will attempt to do the right thing so that if you attach a debugger to the process, it will be able to inspect variables and step through your code.

Note that you can't step through code unless you set up source location information for the code (by creating and passing in *gccjit.Location* instances).

DUMP_INITIAL_TREE

If true, `gccjit.Context.compile()` will dump its initial "tree" representation of your code to stderr (before any optimizations).

Here's some sample output (from the *square* example):

```
<statement_list 0x7f4875a62cc0
   type <void_type 0x7f4875a64bd0 VOID
       align 8 symtab 0 alias set -1 canonical type 0x7f4875a64bd0
       pointer_to_this <pointer_type 0x7f4875a64c78>>
   side-effects head 0x7f4875a761e0 tail 0x7f4875a761f8 stmts 0x7f4875a62d20 0x7f4875a62d00

   stmt <label_expr 0x7f4875a62d20 type <void_type 0x7f4875a64bd0>
       side-effects
       arg 0 <label_decl 0x7f4875a79080 entry type <void_type 0x7f4875a64bd0>
           VOID file (null) line 0 col 0
           align 1 context <function_decl 0x7f4875a77500 square>>>
   stmt <return_expr 0x7f4875a62d00
       type <integer_type 0x7f4875a645e8 public SI
           size <integer_cst 0x7f4875a623a0 constant 32>
           unit size <integer_cst 0x7f4875a623c0 constant 4>
           align 32 symtab 0 alias set -1 canonical type 0x7f4875a645e8 precision 32 min <in
           pointer_to_this <pointer_type 0x7f4875a6b348>>
       side-effects
       arg 0 <modify_expr 0x7f4875a72a78 type <integer_type 0x7f4875a645e8>
           side-effects arg 0 <result_decl 0x7f4875a7a000 D.54>
           arg 1 <mult_expr 0x7f4875a72a50 type <integer_type 0x7f4875a645e8>
               arg 0 <parm_decl 0x7f4875a79000 i> arg 1 <parm_decl 0x7f4875a79000 i>>>>>
```

DUMP_INITIAL_GIMPLE

If true, `gccjit.Context.compile()` will dump the "gimple" representation of your code to stderr, before any optimizations are performed. The dump resembles C code:

```
square (signed int i)
{
  signed int D.56;

  entry:
  D.56 = i * i;
  return D.56;
}
```

DUMP_GENERATED_CODE

If true, `gccjit.Context.compile()` will dump the final generated code to stderr, in the form of assembly language:

```
    .file    "fake.c"
    .text
    .globl    square
    .type    square, @function
square:
.LFB0:
    .cfi_startproc
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
```

```
            .cfi_def_cfa_register 6
            movl    %edi, -4(%rbp)
        .L2:
            movl    -4(%rbp), %eax
            imull    -4(%rbp), %eax
            popq    %rbp
            .cfi_def_cfa 7, 8
            ret
            .cfi_endproc
        .LFE0:
            .size    square, .-square
            .ident    "GCC: (GNU) 4.9.0 20131023 (Red Hat 0.1-%{gcc_release})"
            .section    .note.GNU-stack,"",@progbits
```

DUMP_SUMMARY
If true, *gccjit.Context.compile()* will print information to stderr on the actions it is performing, followed by a profile showing the time taken and memory usage of each phase.

DUMP_EVERYTHING
If true, *gccjit.Context.compile()* will dump copious amount of information on what it's doing to various files within a temporary directory. Use *gccjit.BoolOption.KEEP_INTERMEDIATES* (see below) to see the results. The files are intended to be human-readable, but the exact files and their formats are subject to change.

SELFCHECK_GC
If true, libgccjit will aggressively run its garbage collector, to shake out bugs (greatly slowing down the compile). This is likely to only be of interest to developers *of* the library. It is used when running the selftest suite.

KEEP_INTERMEDIATES
If true, the gccjit.Context will not clean up intermediate files written to the filesystem, and will display their location on stderr.

Integer options

gccjit.Context.**set_int_option**(*seld*, *opt*, *val*)
Set an integer option of the context; see *gccjit.IntOption* for notes on the options and their meanings.

Parameters

- **opt** (*gccjit.IntOption*) – Which option to set
- **val** (*str*) – The new value

class gccjit.**IntOption**

OPTIMIZATION_LEVEL
How much to optimize the code.

Valid values are 0-3, corresponding to GCC's command-line options -O0 through -O3.

The default value is 0 (unoptimized).

2.2 Types

Types can be created in several ways:

- fundamental types can be accessed using *gccjit.Context.get_type()*:

```
int_type = ctxt.get_type(gccjit.TypeKind.INT)
```

  See *gccjit.TypeKind* for the available types.

  You can get *int* types of specific sizes (in bytes) using *gccjit.Context.get_int_type()*:

```
int_type = ctxt.get_int_type(4, is_signed=True)
```

- derived types can be accessed by calling methods on an existing type:

```
const_int_star = int_type.get_const().get_pointer()
int_const_star = int_type.get_pointer().get_const()
```

- by creating structures (see below).

class gccjit.**Type**

get_pointer()

Given type *T* get type *T**.

Return type *gccjit.Type*

get_const()

Given type *T* get type *const T*.

Return type *gccjit.Type*

get_volatile()

Given type *T* get type *volatile T*.

Return type *gccjit.Type*

2.2.1 Standard types

gccjit.Context.**get_type**(*self*, *type_enum*)

Look up one of the standard types (see *gccjit.TypeKind*):

```
int_type = ctxt.get_type(gccjit.TypeKind.INT)
```

Parameters **type_enum** (*gccjit.TypeKind*) – Which type to lookup

class gccjit.**TypeKind**

VOID

C's "void" type.

VOID_PTR

C's "void *".

BOOL

C++'s bool type; also C99's "_Bool" type, aka "bool" if using stdbool.h.

CHAR

SIGNED_CHAR

UNSIGNED_CHAR

C's "char" (of some signedness) and the variants where the signedness is specified.

SHORT

UNSIGNED_SHORT
C's "short" (signed) and "unsigned short".

INT

UNSIGNED_INT
C's "int" (signed) and "unsigned int":

```
int_type = ctxt.get_type(gccjit.TypeKind.INT)
```

LONG

UNSIGNED_LONG
C's "long" (signed) and "unsigned long".

LONG_LONG

UNSIGNED_LONG_LONG
C99's "long long" (signed) and "unsigned long long".

FLOAT

DOUBLE

LONG_DOUBLE
Floating-point types

CONST_CHAR_PTR
C type: (const char *):

```
const_char_p = ctxt.get_type(gccjit.TypeKind.CONST_CHAR_PTR)
```

SIZE_T
The C "size_t" type.

FILE_PTR
C type: (FILE *)

gccjit.Context.**get_int_type**(*self*, *num_bytes*, *is_signed*)
Look up an integet type of the given size:

```
int_type = ctxt.get_int_type(4, is_signed=True)
```

2.2.2 Structures

You can model C *struct* types by creating *gccjit.Struct* and *gccjit.Field* instances, in either order:

- by creating the fields, then the structure. For example, to model:

```
struct coord {double x; double y; };
```

  you could call:

```
field_x = ctxt.new_field(double_type, b'x')
field_y = ctxt.new_field(double_type, b'y')
coord = ctxt.new_struct(b'coord', [field_x, field_y])
```

  (see *gccjit.Context.new_field()* and *gccjit.Context.new_struct()*), or

- by creating the structure, then populating it with fields, typically to allow modelling self-referential structs such as:

```
struct node { int m_hash; struct node *m_next; };
```

like this:

```
node = ctxt.new_struct(b'node')
node_ptr = node.get_pointer()
field_hash = ctxt.new_field(int_type, b'm_hash')
field_next = ctxt.new_field(node_ptr, b'm_next')
node.set_fields([field_hash, field_next])
```

(see `gccjit.Struct.set_fields()`)

class `gccjit.`**`Field`**

class `gccjit.`**`Struct`**

`set_fields`(*fields*, *loc=None*)
Populate the fields of a formerly-opaque struct type. This can only be called once on a given struct type.

2.3 Expressions

class `gccjit.`**`RValue`**

`dereference_field`(*Field field*, *Location loc=None*)

`dereference`(*loc=None*)

`get_type`()

class `gccjit.`**`LValue`**

`get_address`(*loc=None*)
Get the address of this lvalue, as a `gccjit.RValue` of type *T**.

2.3.1 Unary Operations

Unary operations are `gccjit.RValue` instances built using `gccjit.Context.new_unary_op()` with an operation from one of the following:

Unary Operation	C equivalent
`gccjit.UnaryOp.MINUS`	*-(EXPR)*
`gccjit.UnaryOp.BITWISE_NEGATE`	*~(EXPR)*
`gccjit.UnaryOp.LOGICAL_NEGATE`	*!(EXPR)*

class `gccjit.`**`UnaryOp`**

`MINUS`
Negate an arithmetic value; analogous to:

```
-(EXPR)
```

in C.

BITWISE_NEGATE

Bitwise negation of an integer value (one's complement); analogous to:

```
~(EXPR)
```

in C.

LOGICAL_NEGATE

Logical negation of an arithmetic or pointer value; analogous to:

```
!(EXPR)
```

in C.

2.3.2 Binary Operations

Unary operations are *gccjit.RValue* instances built using *gccjit.Context.new_binary_op()* with an operation from one of the following:

Binary Operation	C equivalent
gccjit.BinaryOp.PLUS	*x* + *y*
gccjit.BinaryOp.MINUS	*x* - *y*
gccjit.BinaryOp.MULT	*x* * *y*
gccjit.BinaryOp.DIVIDE	*x* / *y*
gccjit.BinaryOp.MODULO	*x* % *y*
gccjit.BinaryOp.BITWISE_AND	*x* & *y*
gccjit.BinaryOp.BITWISE_XOR	*x* ^ *y*
gccjit.BinaryOp.BITWISE_OR	*x* | *y*
gccjit.BinaryOp.LOGICAL_AND	*x* && *y*
gccjit.BinaryOp.LOGICAL_OR	*x* || *y*

class gccjit.**BinaryOp**

PLUS

Addition of arithmetic values; analogous to:

```
(EXPR_A) + (EXPR_B)
```

in C.

For pointer addition, use *gccjit.Context.new_array_access()*.

MINUS

Subtraction of arithmetic values; analogous to:

```
(EXPR_A) - (EXPR_B)
```

in C.

MULT

Multiplication of a pair of arithmetic values; analogous to:

```
(EXPR_A) * (EXPR_B)
```

in C.

DIVIDE

Quotient of division of arithmetic values; analogous to:

```
(EXPR_A) / (EXPR_B)
```

in C.

The result type affects the kind of division: if the result type is integer-based, then the result is truncated towards zero, whereas a floating-point result type indicates floating-point division.

MODULO
Remainder of division of arithmetic values; analogous to:

```
(EXPR_A) % (EXPR_B)
```

in C.

BITWISE_AND
Bitwise AND; analogous to:

```
(EXPR_A) & (EXPR_B)
```

in C.

BITWISE_XOR
Bitwise exclusive OR; analogous to:

```
(EXPR_A) ^ (EXPR_B)
```

in C.

BITWISE_OR
Bitwise inclusive OR; analogous to:

```
(EXPR_A) | (EXPR_B)
```

in C.

LOGICAL_AND
Logical AND; analogous to:

```
(EXPR_A) && (EXPR_B)
```

in C.

LOGICAL_OR
Logical OR; analogous to:

```
(EXPR_A) || (EXPR_B)
```

in C.

2.3.3 Comparisons

Comparisons are *gccjit.RValue* instances of boolean type built using *gccjit.Context.new_comparison()* with an operation from one of the following:

Comparison	C equivalent
gccjit.Comparison.EQ	$x == y$
gccjit.Comparison.NE	$x != y$
gccjit.Comparison.LT	$x < y$
gccjit.Comparison.LE	$x <= y$
gccjit.Comparison.GT	$x > y$
gccjit.Comparison.GE	$x >= y$

class gccjit.**Comparison**

EQ

NE

LT

LE

GT

GE

2.4 Functions

class gccjit.**Param**

class gccjit.**Function**

new_local(*type_*, *name*, *loc=None*)
Add a new local variable to the function:

```
i = fn.new_local(int_type, b'i')
```

Return type *gccjit.LValue*

new_block(*name*)
Create a *gccjit.Block*.

The name can be None, or you can give it a meaningful name, which may show up in dumps of the internal representation, and in error messages:

```
entry = fn.new_block('entry')
on_true = fn.new_block('on_true')
```

get_param(*index*)

dump_to_dot(*path*)
Write a dump in GraphViz format to the given path.

class gccjit.**Block**
A *gccjit.Block* is a basic block within a function, i.e. a sequence of statements with a single entry point and a single exit point.

The first basic block that you create within a function will be the entrypoint.

Each basic block that you create within a function must be terminated, either with a conditional, a jump, or a return.

It's legal to have multiple basic blocks that return within one function.

add_eval(*rvalue*, *loc=None*)
Add evaluation of an rvalue, discarding the result (e.g. a function call that "returns" void), for example:

```
call = ctxt.new_call(some_fn, args)
block.add_eval(call)
```

This is equivalent to this C code:

```
(void)expression;
```

add_assignment (*lvalue*, *rvalue*, *loc=None*)
Add evaluation of an rvalue, assigning the result to the given lvalue, for example:

```
# i = 0
entry_block.add_assignment(local_i, ctxt.zero(the_type))
```

This is roughly equivalent to this C code:

```
lvalue = rvalue;
```

add_assignment_op (*lvalue*, *op*, *rvalue*, *loc=None*)
Add evaluation of an rvalue, using the result to modify an lvalue via the given *gccjit.BinaryOp*. For example:

```
# i++
loop_block.add_assignment_op(local_i,
                             gccjit.BinaryOp.PLUS,
                             ctxt.one(the_type))
```

This is analogous to "+=" and friends:

```
lvalue += rvalue;
lvalue *= rvalue;
lvalue /= rvalue;
/* etc */
```

add_comment (*text*, *Location loc=None*)
Add a no-op textual comment to the internal representation of the code. It will be optimized away, but will be visible in the dumps seen via *gccjit.BoolOption.DUMP_INITIAL_TREE* and *gccjit.BoolOption.DUMP_INITIAL_GIMPLE* and thus may be of use when debugging how your project's internal representation gets converted to the libgccjit IR.

end_with_conditional (*boolval*, *on_true*, *on_false=None*, *loc=None*)
Terminate a block by adding evaluation of an rvalue, branching on the result to the appropriate successor block.

This is roughly equivalent to this C code:

```
if (boolval)
  goto on_true;
else
  goto on_false;
```

Example:

```
# while (i < n)
cond_block.end_with_conditional(
  ctxt.new_comparison(gccjit.Comparison.LT, local_i, param_n),
  loop_block,
  after_loop_block)
```

end_with_jump (*target*, *loc=None*)
Terminate a block by adding a jump to the given target block.

This is roughly equivalent to this C code:

```
goto target;
```

Example:

```
        loop_block.end_with_jump(cond_block)
```

end_with_return(*RValue rvalue*, *loc=None*)

Terminate a block by adding evaluation of an rvalue, returning the value.

This is roughly equivalent to this C code:

```
        return expression;
```

Example:

```
        # return sum
        after_loop_block.end_with_return(local_sum)
```

end_with_void_return(*loc=None*)

Terminate a block by adding a valueless return, for use within a function with "void" return type.

This is equivalent to this C code:

```
        return;
```

get_function()

Get the *gccjit.Function* that this block is within.

class gccjit.**FunctionKind**

EXPORTED

INTERNAL

IMPORTED

ALWAYS_INLINE

2.5 Source Locations

class gccjit.**Location**

A *gccjit.Location* encapsulates a source code location, so that you can (optionally) associate locations in your language with statements in the JIT-compiled code, allowing the debugger to single-step through your language.

You can construct them using *gccjit.Context.new_location()*.

You need to enable *gccjit.BoolOption.DEBUGINFO* on the *gccjit.Context* for these locations to actually be usable by the debugger:

```
    ctxt.set_bool_option(gccjit.BoolOption.DEBUGINFO, True)
```

gccjit.Location instances are optional; most API entrypoints accepting one default to *None*.

2.5.1 Faking it

If you don't have source code for your internal representation, but need to debug, you can generate a C-like representation of the functions in your context using *gccjit.Context.dump_to_file()*:

```
ctxt.dump_to_file(b'/tmp/something.c', True)
```

This will dump C-like code to the given path. If the *update_locations* argument is *True*, this will also set up *gccjit.Location* information throughout the context, pointing at the dump file as if it were a source file, giving you *something* you can step through in the debugger.

2.6 Compiling a context

Once populated, a `gccjit.Context` can be compiled to machine code, either in-memory via `gccjit.Context.compile()` or to disk via `gccjit.Context.compile_to_file()`.

You can compile a context multiple times (using either form of compilation), although any errors that occur on the context will prevent any future compilation of that context.

2.6.1 In-memory compilation

`gccjit.Context.`**`compile`**`(`*self*`)`

rtype `gccjit.Result`

This calls into GCC and builds the code, returning a `gccjit.Result`.

class `gccjit.`**`Result`**

A `gccjit.Result` encapsulates the result of compiling a `gccjit.Context` in-memory, and the lifetimes of any machine code functions or globals that are within the result.

`get_code``(`*funcname*`)`

Locate the given function within the built machine code.

Functions are looked up by name. For this to succeed, a function with a name matching *funcname* must have been created on *result*'s context (or a parent context) via a call to `gccjit.Context.new_function()` with *kind* `gccjit.FunctionKind.EXPORTED`.

The returned value is an *int*, actually a pointer to the machine code within the address space of the process. This will need to be wrapped up with *ctypes* to be callable:

```
import ctypes

# "[int] -> int" functype:
int_int_func_type = ctypes.CFUNCTYPE(ctypes.c_int, ctypes.c_int)
code = int_int_func_type(jit_result.get_code(b"square"))
assert code(5) == 25
```

The code has the same lifetime as the `gccjit.Result` instance; the pointer becomes invalid when the result instance is cleaned up.

2.6.2 Ahead-of-time compilation

Although libgccjit is primarily aimed at just-in-time compilation, it can also be used for implementing more traditional ahead-of-time compilers, via the `gccjit.Context.compile_to_file()` API entrypoint.

`gccjit.Context.`**`compile_to_file`**`(`*self*, *kind*, *path*`)`

Compile the context to a file of the given kind:

```
ctxt.compile_to_file(gccjit.OutputKind.EXECUTABLE,
                     'a.out')
```

gccjit.Context.compile_to_file() ignores the suffix of `path`, and insteads uses *kind* to decide what to do.

Note: This is different from the `gcc` program, which does make use of the suffix of the output file when determining what to do.

The available kinds of output are:

Output kind	Typical suffix
gccjit.OutputKind.ASSEMBLER	.s
gccjit.OutputKind.OBJECT_FILE	.o
gccjit.OutputKind.DYNAMIC_LIBRARY	.so or .dll
gccjit.OutputKind.EXECUTABLE	None, or .exe

class `gccjit.`**`OutputKind`**

`ASSEMBLER`

Compile the context to an assembler file.

`OBJECT_FILE`

Compile the context to an object file.

`DYNAMIC_LIBRARY`

Compile the context to a dynamic library.

There is currently no support for specifying other libraries to link against.

`EXECUTABLE`

Compile the context to an executable.

There is currently no support for specifying libraries to link against.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

A

add_assignment() (gccjit.Block method), 36
add_assignment_op() (gccjit.Block method), 36
add_comment() (gccjit.Block method), 36
add_eval() (gccjit.Block method), 35

C

compile() (gccjit.Context method), 38
compile_to_file() (gccjit.Context method), 38

D

dereference() (gccjit.RValue method), 32
dereference_field() (gccjit.RValue method), 32
dump_reproducer_to_file() (gccjit.Context method), 26
dump_to_dot() (gccjit.Function method), 35
dump_to_file() (gccjit.Context method), 23

E

end_with_conditional() (gccjit.Block method), 36
end_with_jump() (gccjit.Block method), 36
end_with_return() (gccjit.Block method), 37
end_with_void_return() (gccjit.Block method), 37

G

gccjit.BinaryOp (built-in class), 33
gccjit.BinaryOp.BITWISE_AND (built-in variable), 34
gccjit.BinaryOp.BITWISE_OR (built-in variable), 34
gccjit.BinaryOp.BITWISE_XOR (built-in variable), 34
gccjit.BinaryOp.DIVIDE (built-in variable), 33
gccjit.BinaryOp.LOGICAL_AND (built-in variable), 34
gccjit.BinaryOp.LOGICAL_OR (built-in variable), 34
gccjit.BinaryOp.MINUS (built-in variable), 33
gccjit.BinaryOp.MODULO (built-in variable), 34
gccjit.BinaryOp.MULT (built-in variable), 33
gccjit.BinaryOp.PLUS (built-in variable), 33
gccjit.Block (built-in class), 35
gccjit.BoolOption (built-in class), 27
gccjit.BoolOption.DEBUGINFO (built-in variable), 27
gccjit.BoolOption.DUMP_EVERYTHING (built-in variable), 29
gccjit.BoolOption.DUMP_GENERATED_CODE (built-in variable), 28
gccjit.BoolOption.DUMP_INITIAL_GIMPLE (built-in variable), 28
gccjit.BoolOption.DUMP_INITIAL_TREE (built-in variable), 28
gccjit.BoolOption.DUMP_SUMMARY (built-in variable), 29
gccjit.BoolOption.KEEP_INTERMEDIATES (built-in variable), 29
gccjit.BoolOption.SELFCHECK_GC (built-in variable), 29
gccjit.Comparison (built-in class), 34
gccjit.Comparison.EQ (built-in variable), 35
gccjit.Comparison.GE (built-in variable), 35
gccjit.Comparison.GT (built-in variable), 35
gccjit.Comparison.LE (built-in variable), 35
gccjit.Comparison.LT (built-in variable), 35
gccjit.Comparison.NE (built-in variable), 35
gccjit.Context (built-in class), 23
gccjit.Field (built-in class), 32
gccjit.Function (built-in class), 35
gccjit.FunctionKind (built-in class), 37
gccjit.FunctionKind.ALWAYS_INLINE (built-in variable), 37
gccjit.FunctionKind.EXPORTED (built-in variable), 37
gccjit.FunctionKind.IMPORTED (built-in variable), 37
gccjit.FunctionKind.INTERNAL (built-in variable), 37
gccjit.IntOption (built-in class), 29
gccjit.IntOption.OPTIMIZATION_LEVEL (built-in variable), 29
gccjit.Location (built-in class), 37
gccjit.LValue (built-in class), 32
gccjit.OutputKind (built-in class), 39
gccjit.OutputKind.ASSEMBLER (built-in variable), 39
gccjit.OutputKind.DYNAMIC_LIBRARY (built-in variable), 39
gccjit.OutputKind.EXECUTABLE (built-in variable), 39
gccjit.OutputKind.OBJECT_FILE (built-in variable), 39
gccjit.Param (built-in class), 35
gccjit.Result (built-in class), 38

gccjit.RValue (built-in class), 32
gccjit.StrOption (built-in class), 27
gccjit.StrOption.PROGNAME (built-in variable), 27
gccjit.Struct (built-in class), 32
gccjit.Type (built-in class), 30
gccjit.TypeKind (built-in class), 30
gccjit.TypeKind.BOOL (built-in variable), 30
gccjit.TypeKind.CHAR (built-in variable), 30
gccjit.TypeKind.CONST_CHAR_PTR (built-in variable), 31
gccjit.TypeKind.DOUBLE (built-in variable), 31
gccjit.TypeKind.FILE_PTR (built-in variable), 31
gccjit.TypeKind.FLOAT (built-in variable), 31
gccjit.TypeKind.INT (built-in variable), 31
gccjit.TypeKind.LONG (built-in variable), 31
gccjit.TypeKind.LONG_DOUBLE (built-in variable), 31
gccjit.TypeKind.LONG_LONG (built-in variable), 31
gccjit.TypeKind.SHORT (built-in variable), 30
gccjit.TypeKind.SIGNED_CHAR (built-in variable), 30
gccjit.TypeKind.SIZE_T (built-in variable), 31
gccjit.TypeKind.UNSIGNED_CHAR (built-in variable), 30
gccjit.TypeKind.UNSIGNED_INT (built-in variable), 31
gccjit.TypeKind.UNSIGNED_LONG (built-in variable), 31
gccjit.TypeKind.UNSIGNED_LONG_LONG (built-in variable), 31
gccjit.TypeKind.UNSIGNED_SHORT (built-in variable), 31
gccjit.TypeKind.VOID (built-in variable), 30
gccjit.TypeKind.VOID_PTR (built-in variable), 30
gccjit.UnaryOp (built-in class), 32
gccjit.UnaryOp.BITWISE_NEGATE (built-in variable), 32
gccjit.UnaryOp.LOGICAL_NEGATE (built-in variable), 33
gccjit.UnaryOp.MINUS (built-in variable), 32
get_address() (gccjit.LValue method), 32
get_builtin_function() (gccjit.Context method), 24
get_code() (gccjit.Result method), 38
get_const() (gccjit.Type method), 30
get_first_error() (gccjit.Context method), 23
get_function() (gccjit.Block method), 37
get_int_type() (gccjit.Context method), 31
get_param() (gccjit.Function method), 35
get_pointer() (gccjit.Type method), 30
get_type() (gccjit.Context method), 30
get_type() (gccjit.RValue method), 32
get_volatile() (gccjit.Type method), 30

N

new_array_access() (gccjit.Context method), 26
new_array_type() (gccjit.Context method), 23
new_binary_op() (gccjit.Context method), 25
new_block() (gccjit.Function method), 35
new_call() (gccjit.Context method), 26
new_call_through_ptr() (gccjit.Context method), 26
new_cast() (gccjit.Context method), 26
new_child_context() (gccjit.Context method), 26
new_comparison() (gccjit.Context method), 25
new_field() (gccjit.Context method), 23
new_function() (gccjit.Context method), 24
new_function_ptr_type() (gccjit.Context method), 24
new_global() (gccjit.Context method), 23
new_local() (gccjit.Function method), 35
new_location() (gccjit.Context method), 23
new_param() (gccjit.Context method), 24
new_rvalue_from_double() (gccjit.Context method), 25
new_rvalue_from_int() (gccjit.Context method), 25
new_rvalue_from_ptr() (gccjit.Context method), 25
new_string_literal() (gccjit.Context method), 25
new_struct() (gccjit.Context method), 23
new_unary_op() (gccjit.Context method), 25
new_union() (gccjit.Context method), 23
null() (gccjit.Context method), 25

O

one() (gccjit.Context method), 24

S

set_bool_option() (gccjit.Context method), 27
set_fields() (gccjit.Struct method), 32
set_int_option() (gccjit.Context method), 29
set_logfile() (gccjit.Context method), 26
set_str_option() (gccjit.Context method), 27

Z

zero() (gccjit.Context method), 24