libgccjit

Release 13.0.0 (experimental 20221108)

David Malcolm

Nov 10, 2022

CONTENTS

This document describes libgccjit, an API for embedding GCC inside programs and libraries.

There are actually two APIs for the library:

- a pure C API: `libgccjit.h`
- a C++ wrapper API: `libgccjit++.h`. This is a collection of "thin" wrapper classes around the C API, to save typing.

Contents:

CHAPTER

ONE

TUTORIAL

1.1 Tutorial part 1: "Hello world"

Before we look at the details of the API, let's look at building and running programs that use the library.

Here's a toy "hello world" program that uses the library to synthesize a call to *printf* and uses it to write a message to stdout.

Don't worry about the content of the program for now; we'll cover the details in later parts of this tutorial.

```
/* Smoketest example for libgccjit.so
   Copyright (C) 2014-2022 Free Software Foundation, Inc.

This file is part of GCC.

GCC is free software; you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 3, or (at your option)
any later version.

GCC is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with GCC; see the file COPYING3.  If not see
<http://www.gnu.org/licenses/>.  */

#include <libgccjit.h>

#include <stdlib.h>
#include <stdio.h>

static void
create_code (gcc_jit_context *ctxt)
{
  /* Let's try to inject the equivalent of:
```

(continues on next page)

(continued from previous page)

```
    void
    greet (const char *name)
    {
       printf ("hello %s\n", name);
    }
  */
  gcc_jit_type *void_type =
    gcc_jit_context_get_type (ctxt, GCC_JIT_TYPE_VOID);
  gcc_jit_type *const_char_ptr_type =
    gcc_jit_context_get_type (ctxt, GCC_JIT_TYPE_CONST_CHAR_PTR);
  gcc_jit_param *param_name =
    gcc_jit_context_new_param (ctxt, NULL, const_char_ptr_type, "name");
  gcc_jit_function *func =
    gcc_jit_context_new_function (ctxt, NULL,
                                  GCC_JIT_FUNCTION_EXPORTED,
                                  void_type,
                                  "greet",
                                  1, &param_name,
                                  0);

  gcc_jit_param *param_format =
    gcc_jit_context_new_param (ctxt, NULL, const_char_ptr_type, "format");
  gcc_jit_function *printf_func =
    gcc_jit_context_new_function (ctxt, NULL,
                                  GCC_JIT_FUNCTION_IMPORTED,
                                  gcc_jit_context_get_type (
                                     ctxt, GCC_JIT_TYPE_INT),
                                  "printf",
                                  1, &param_format,
                                  1);
  gcc_jit_rvalue *args[2];
  args[0] = gcc_jit_context_new_string_literal (ctxt, "hello %s\n");
  args[1] = gcc_jit_param_as_rvalue (param_name);

  gcc_jit_block *block = gcc_jit_function_new_block (func, NULL);

  gcc_jit_block_add_eval (
    block, NULL,
    gcc_jit_context_new_call (ctxt,
                              NULL,
                              printf_func,
                              2, args));
  gcc_jit_block_end_with_void_return (block, NULL);
}

int
main (int argc, char **argv)
{
  gcc_jit_context *ctxt;
  gcc_jit_result *result;
```

(continues on next page)

(continued from previous page)

```
  /* Get a "context" object for working with the library.  */
  ctxt = gcc_jit_context_acquire ();
  if (!ctxt)
    {
      fprintf (stderr, "NULL ctxt");
      exit (1);
    }

  /* Set some options on the context.
     Let's see the code being generated, in assembler form.  */
  gcc_jit_context_set_bool_option (
    ctxt,
    GCC_JIT_BOOL_OPTION_DUMP_GENERATED_CODE,
    0);

  /* Populate the context.  */
  create_code (ctxt);

  /* Compile the code.  */
  result = gcc_jit_context_compile (ctxt);
  if (!result)
    {
      fprintf (stderr, "NULL result");
      exit (1);
    }

  /* Extract the generated code from "result".  */
  typedef void (*fn_type) (const char *);
  fn_type greet =
    (fn_type)gcc_jit_result_get_code (result, "greet");
  if (!greet)
    {
      fprintf (stderr, "NULL greet");
      exit (1);
    }

  /* Now call the generated function: */
  greet ("world");
  fflush (stdout);

  gcc_jit_context_release (ctxt);
  gcc_jit_result_release (result);
  return 0;
}
```

Copy the above to *tut01-hello-world.c*.

Assuming you have the jit library installed, build the test program using:

```
$ gcc \
    tut01-hello-world.c \
    -o tut01-hello-world \
```

(continues on next page)

(continued from previous page)

```
    -lgccjit
```

You should then be able to run the built program:

```
$ ./tut01-hello-world
hello world
```

1.2 Tutorial part 2: Creating a trivial machine code function

Consider this C function:

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

How can we construct this at run-time using libgccjit?

First we need to include the relevant header:

```
#include <libgccjit.h>
```

All state associated with compilation is associated with a `gcc_jit_context*`.

Create one using `gcc_jit_context_acquire()`:

```
gcc_jit_context *ctxt;
ctxt = gcc_jit_context_acquire ();
```

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 `gcc_jit_type*`, using `gcc_jit_context_get_type()`:

```
gcc_jit_type *int_type =
  gcc_jit_context_get_type (ctxt, GCC_JIT_TYPE_INT);
```

`gcc_jit_type*` is an example of a "contextual" object: every entity in the API is associated with a `gcc_jit_context*`.

Memory management is easy: all such "contextual" objects are automatically cleaned up for you when the context is released, using `gcc_jit_context_release()`:

```
gcc_jit_context_release (ctxt);
```

so you don't need to manually track and cleanup all objects, just the contexts.

Although the API is C-based, there is a form of class hierarchy, which looks like this:

```
+- gcc_jit_object
    +- gcc_jit_location
    +- gcc_jit_type
       +- gcc_jit_struct
    +- gcc_jit_field
    +- gcc_jit_function
    +- gcc_jit_block
    +- gcc_jit_rvalue
        +- gcc_jit_lvalue
           +- gcc_jit_param
```

There are casting methods for upcasting from subclasses to parent classes. For example, `gcc_jit_type_as_object()`:

```
gcc_jit_object *obj = gcc_jit_type_as_object (int_type);
```

One thing you can do with a `gcc_jit_object*` is to ask it for a human-readable description, using `gcc_jit_object_get_debug_string()`:

```
printf ("obj: %s\n", gcc_jit_object_get_debug_string (obj));
```

giving this text on stdout:

```
obj: int
```

This is invaluable when debugging.

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

```
gcc_jit_param *param_i =
  gcc_jit_context_new_param (ctxt, NULL, int_type, "i");
```

Now we can create the function, using `gcc_jit_context_new_function()`:

```
gcc_jit_function *func =
  gcc_jit_context_new_function (ctxt, NULL,
                                GCC_JIT_FUNCTION_EXPORTED,
                                int_type,
                                "square",
                                1, &param_i,
                                0);
```

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:

```
gcc_jit_block *block = gcc_jit_function_new_block (func, NULL);
```

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

We can build the expression using `gcc_jit_context_new_binary_op()`:

```
gcc_jit_rvalue *expr =
  gcc_jit_context_new_binary_op (
    ctxt, NULL,
    GCC_JIT_BINARY_OP_MULT, int_type,
    gcc_jit_param_as_rvalue (param_i),
    gcc_jit_param_as_rvalue (param_i));
```

A `gcc_jit_rvalue*` is another example of a `gcc_jit_object*` subclass. We can upcast it using `gcc_jit_rvalue_as_object()` and as before print it with `gcc_jit_object_get_debug_string()`.

```
printf ("expr: %s\n",
        gcc_jit_object_get_debug_string (
          gcc_jit_rvalue_as_object (expr)));
```

giving this output:

```
expr: i * i
```

Creating the expression 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:

```
gcc_jit_block_end_with_return (block, NULL, expr);
```

OK, we've populated the context. We can now compile it using `gcc_jit_context_compile()`:

```
gcc_jit_result *result;
result = gcc_jit_context_compile (ctxt);
```

and get a `gcc_jit_result*`.

At this point we're done with the context; we can release it:

```
gcc_jit_context_release (ctxt);
```

We can now use `gcc_jit_result_get_code()` to look up a specific machine code routine within the result, in this case, the function we created above.

```
void *fn_ptr = gcc_jit_result_get_code (result, "square");
if (!fn_ptr)
  {
    fprintf (stderr, "NULL fn_ptr");
    goto error;
  }
```

We can now cast the pointer to an appropriate function pointer type, and then call it:

```
typedef int (*fn_type) (int);
fn_type square = (fn_type)fn_ptr;
printf ("result: %d", square (5));
```

```
result: 25
```

Once we're done with the code, we can release the result:

```
gcc_jit_result_release (result);
```

We can't call `square` anymore once we've released `result`.

1.2.1 Error-handling

Various kinds of errors are possible when using the API, such as mismatched types in an assignment. You can only compile and get code from a context if no errors occur.

Errors are printed on stderr; they typically contain the name of the API entrypoint where the error occurred, and pertinent information on the problem:

```
./buggy-program: error: gcc_jit_block_add_assignment: mismatching types: assignment to i (type:␣
↪int) from "hello world" (type: const char *)
```

The API is designed to cope with errors without crashing, so you can get away with having a single error-handling check in your code:

```
void *fn_ptr = gcc_jit_result_get_code (result, "square");
if (!fn_ptr)
  {
    fprintf (stderr, "NULL fn_ptr");
    goto error;
  }
```

For more information, see the error-handling guide within the Topic eference.

1.2.2 Options

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

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

```
gcc_jit_context_set_bool_option (
  ctxt,
  GCC_JIT_BOOL_OPTION_DUMP_INITIAL_GIMPLE,
  1);
result = gcc_jit_context_compile (ctxt);
```

```
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 `GCC_JIT_BOOL_OPTION_DUMP_GENERATED_CODE` on the context before compiling:

```
gcc_jit_context_set_bool_option (
  ctxt,
  GCC_JIT_BOOL_OPTION_DUMP_GENERATED_CODE,
  1);
result = gcc_jit_context_compile (ctxt);
```

```
      .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)"
      .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 `gcc_jit_context_set_int_option()` with `GCC_JIT_INT_OPTION_OPTIMIZATION_LEVEL`:

```
gcc_jit_context_set_int_option (
  ctxt,
  GCC_JIT_INT_OPTION_OPTIMIZATION_LEVEL,
  3);
```

```
      .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)"
      .section        .note.GNU-stack,"",@progbits
```

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

1.2.3 Full example

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

```
/* Usage example for libgccjit.so
   Copyright (C) 2014-2022 Free Software Foundation, Inc.

This file is part of GCC.

GCC is free software; you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 3, or (at your option)
any later version.

GCC is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with GCC; see the file COPYING3.  If not see
<http://www.gnu.org/licenses/>.  */

#include <libgccjit.h>

#include <stdlib.h>
#include <stdio.h>

void
create_code (gcc_jit_context *ctxt)
{
```

(continues on next page)

(continued from previous page)

```
  /* Let's try to inject the equivalent of:

     int square (int i)
     {
       return i * i;
     }
  */
  gcc_jit_type *int_type =
    gcc_jit_context_get_type (ctxt, GCC_JIT_TYPE_INT);
  gcc_jit_param *param_i =
    gcc_jit_context_new_param (ctxt, NULL, int_type, "i");
  gcc_jit_function *func =
    gcc_jit_context_new_function (ctxt, NULL,
                                  GCC_JIT_FUNCTION_EXPORTED,
                                  int_type,
                                  "square",
                                  1, &param_i,
                                  0);

  gcc_jit_block *block = gcc_jit_function_new_block (func, NULL);

  gcc_jit_rvalue *expr =
    gcc_jit_context_new_binary_op (
      ctxt, NULL,
      GCC_JIT_BINARY_OP_MULT, int_type,
      gcc_jit_param_as_rvalue (param_i),
      gcc_jit_param_as_rvalue (param_i));

   gcc_jit_block_end_with_return (block, NULL, expr);
}

int
main (int argc, char **argv)
{
  gcc_jit_context *ctxt = NULL;
  gcc_jit_result *result = NULL;

  /* Get a "context" object for working with the library.  */
  ctxt = gcc_jit_context_acquire ();
  if (!ctxt)
    {
      fprintf (stderr, "NULL ctxt");
      goto error;
    }

  /* Set some options on the context.
     Let's see the code being generated, in assembler form.  */
  gcc_jit_context_set_bool_option (
    ctxt,
    GCC_JIT_BOOL_OPTION_DUMP_GENERATED_CODE,
    0);
```

(continues on next page)

(continued from previous page)

```
  /* Populate the context.  */
  create_code (ctxt);

  /* Compile the code.  */
  result = gcc_jit_context_compile (ctxt);
  if (!result)
    {
      fprintf (stderr, "NULL result");
      goto error;
    }

  /* We're done with the context; we can release it: */
  gcc_jit_context_release (ctxt);
  ctxt = NULL;

  /* Extract the generated code from "result".  */
  void *fn_ptr = gcc_jit_result_get_code (result, "square");
  if (!fn_ptr)
     {
       fprintf (stderr, "NULL fn_ptr");
       goto error;
     }

  typedef int (*fn_type) (int);
  fn_type square = (fn_type)fn_ptr;
  printf ("result: %d\n", square (5));

 error:
  if (ctxt)
    gcc_jit_context_release (ctxt);
  if (result)
    gcc_jit_result_release (result);
  return 0;
}
```

Building and running it:

```
$ gcc \
    tut02-square.c \
    -o tut02-square \
    -lgccjit

# Run the built program:
$ ./tut02-square
result: 25
```

1.3 Tutorial part 3: 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.

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 include the libgccjit header and make a `gcc_jit_context*`.

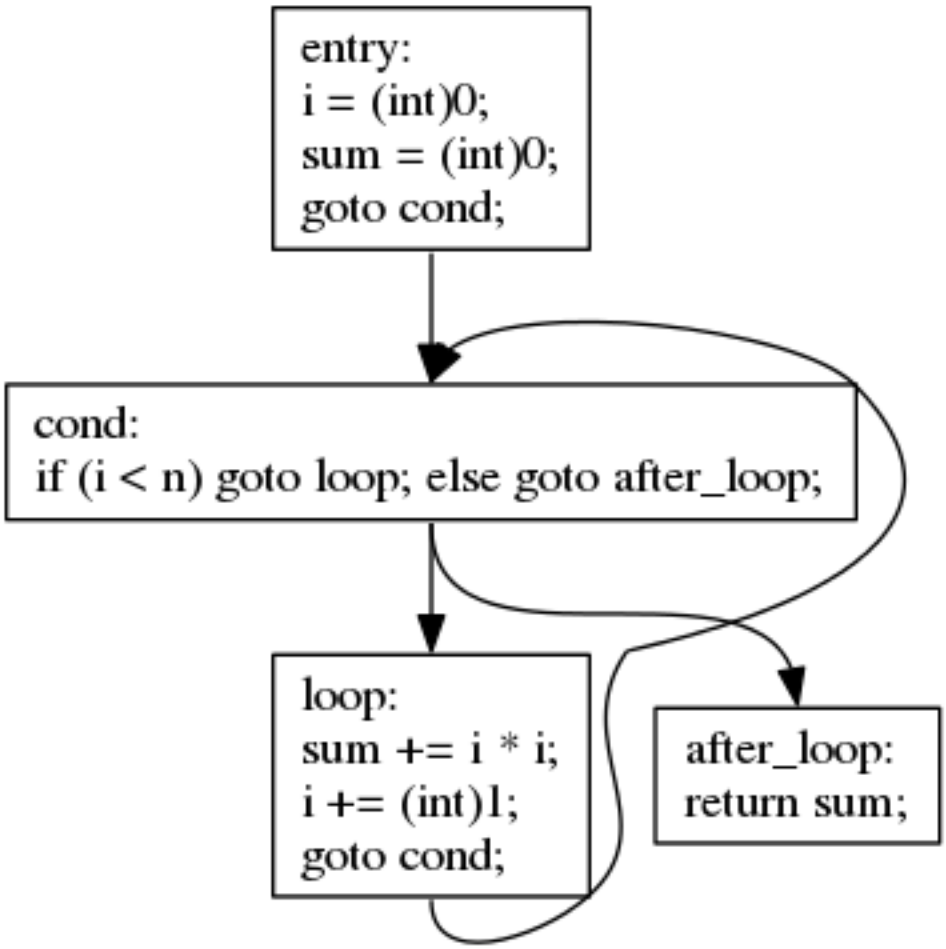
```
#include <libgccjit.h>

void test (void)
{
  gcc_jit_context *ctxt;
  ctxt = gcc_jit_context_acquire ();
```

The function works with the C *int* type:

```
gcc_jit_type *the_type =
  gcc_jit_context_get_type (ctxt, GCC_JIT_TYPE_INT);
gcc_jit_type *return_type = the_type;
```

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

```
gcc_jit_type *the_type =
  gcc_jit_context_get_type (ctxt, GCC_JIT_TYPE_DOUBLE);
```


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;

Let's build the function:

```
gcc_jit_param *n =
  gcc_jit_context_new_param (ctxt, NULL, the_type, "n");
gcc_jit_param *params[1] = {n};
gcc_jit_function *func =
  gcc_jit_context_new_function (ctxt, NULL,
                                GCC_JIT_FUNCTION_EXPORTED,
                                return_type,
                                "loop_test",
                                1, params, 0);
```

1.3.1 Expressions: lvalues and rvalues

The base class of expression is the `gcc_jit_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 `gcc_jit_type*`.

Another important class is `gcc_jit_lvalue*`. A `gcc_jit_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 `gcc_jit_lvalue*` is a subclass of `gcc_jit_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*:

```
gcc_jit_rvalue *expr =
  gcc_jit_context_new_binary_op (
    ctxt, NULL,
    GCC_JIT_BINARY_OP_MULT, int_type,
    gcc_jit_param_as_rvalue (param_i),
    gcc_jit_param_as_rvalue (param_i));
```

which is a `gcc_jit_rvalue*`, and the various function parameters: *param_i* and *param_n*, instances of `gcc_jit_param*`, which is a subclass of `gcc_jit_lvalue*` (and, in turn, of `gcc_jit_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 `gcc_jit_function_new_local()`, supplying a type and a name:

```
/* Build locals:  */
gcc_jit_lvalue *i =
  gcc_jit_function_new_local (func, NULL, the_type, "i");
```

(continues on next page)

(continued from previous page)

```
gcc_jit_lvalue *sum =
  gcc_jit_function_new_local (func, NULL, the_type, "sum");
```

These are instances of `gcc_jit_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.

1.3.2 Control flow

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 `gcc_jit_block*` instances within the `gcc_jit_function*`:

```
gcc_jit_block *b_initial =
  gcc_jit_function_new_block (func, "initial");
gcc_jit_block *b_loop_cond =
  gcc_jit_function_new_block (func, "loop_cond");
gcc_jit_block *b_loop_body =
  gcc_jit_function_new_block (func, "loop_body");
gcc_jit_block *b_after_loop =
  gcc_jit_function_new_block (func, "after_loop");
```

We now populate each block with statements.

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

```
/* sum = 0; */
gcc_jit_block_add_assignment (
  b_initial, NULL,
  sum,
  gcc_jit_context_zero (ctxt, the_type));

/* i = 0; */
gcc_jit_block_add_assignment (
```

(continues on next page)

(continued from previous page)

```
  b_initial, NULL,
  i,
  gcc_jit_context_zero (ctxt, the_type));
```

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

```
gcc_jit_block_end_with_jump (b_initial, NULL, b_loop_cond);
```

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 `gcc_jit_rvalue*`, in this case the comparison of *i* and *n*. We build the comparison using `gcc_jit_context_new_comparison()`:

```
/* (i >= n) */
 gcc_jit_rvalue *guard =
   gcc_jit_context_new_comparison (
     ctxt, NULL,
     GCC_JIT_COMPARISON_GE,
     gcc_jit_lvalue_as_rvalue (i),
     gcc_jit_param_as_rvalue (n));
```

and can then use this to add *b_loop_cond*'s sole statement, via `gcc_jit_block_end_with_conditional()`:

```
/* Equivalent to:
     if (guard)
       goto after_loop;
     else
       goto loop_body;  */
gcc_jit_block_end_with_conditional (
  b_loop_cond, NULL,
  guard,
  b_after_loop, /* on_true */
  b_loop_body); /* 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 `gcc_jit_block_add_assignment_op()` to handle these operations:

```
/* sum += i * i */
gcc_jit_block_add_assignment_op (
  b_loop_body, NULL,
  sum,
  GCC_JIT_BINARY_OP_PLUS,
  gcc_jit_context_new_binary_op (
    ctxt, NULL,
    GCC_JIT_BINARY_OP_MULT, the_type,
    gcc_jit_lvalue_as_rvalue (i),
    gcc_jit_lvalue_as_rvalue (i)));
```

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

```
/* i++ */
gcc_jit_block_add_assignment_op (
  b_loop_body, NULL,
  i,
  GCC_JIT_BINARY_OP_PLUS,
  gcc_jit_context_one (ctxt, the_type));
```

Note: For numeric constants other than 0 or 1, we could use `gcc_jit_context_new_rvalue_from_int()` and `gcc_jit_context_new_rvalue_from_double()`.

The loop body completes by jumping back to the conditional:

```
gcc_jit_block_end_with_jump (b_loop_body, NULL, b_loop_cond);
```

Finally, we populate the *b_after_loop* block, reached when the loop conditional is false. We want to generate the equivalent of:

```
return sum;
```

so the block is just one statement:

```
/* return sum */
gcc_jit_block_end_with_return (
  b_after_loop,
  NULL,
  gcc_jit_lvalue_as_rvalue (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:

```
gcc_jit_result *result;
result = gcc_jit_context_compile (ctxt);

typedef int (*loop_test_fn_type) (int);
loop_test_fn_type loop_test =
 (loop_test_fn_type)gcc_jit_result_get_code (result, "loop_test");
if (!loop_test)
  goto error;
printf ("result: %d", loop_test (10));
```

```
result: 285
```

1.3.3 Visualizing the control flow graph

You can see the control flow graph of a function using `gcc_jit_function_dump_to_dot()`:

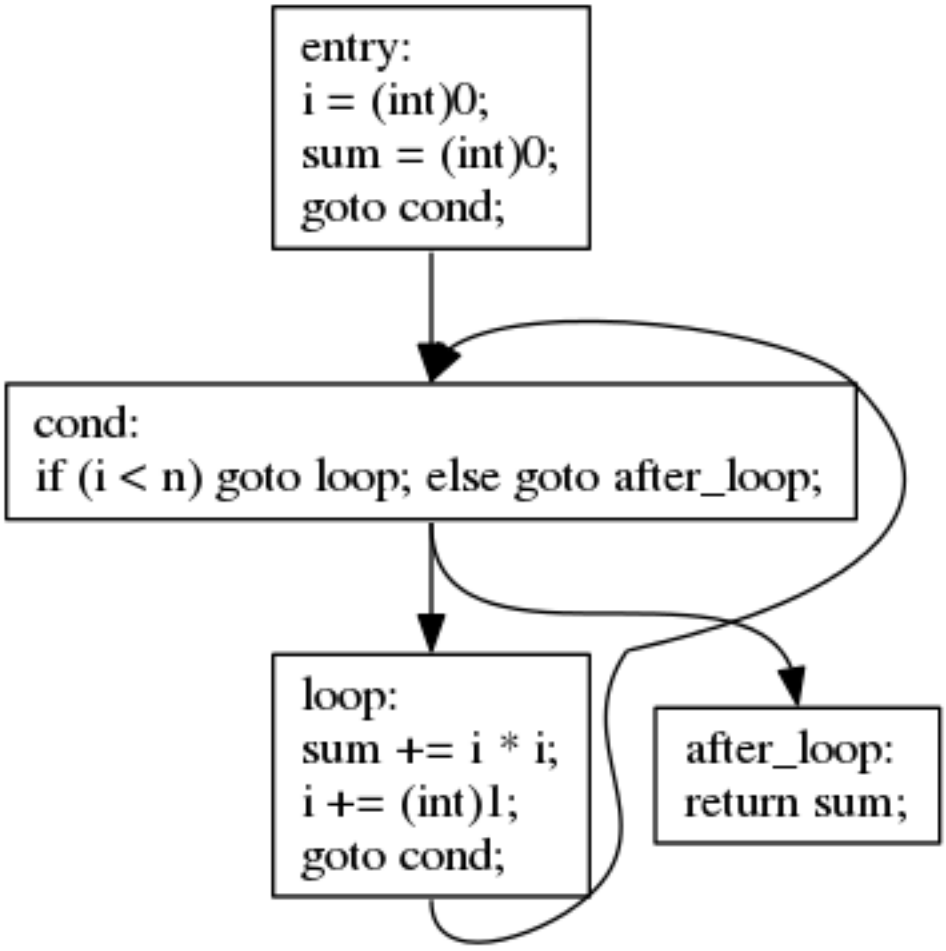
```
gcc_jit_function_dump_to_dot (func, "/tmp/sum-of-squares.dot");
```

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*):



1.3.4 Full example

```
/* Usage example for libgccjit.so
   Copyright (C) 2014-2022 Free Software Foundation, Inc.

This file is part of GCC.

GCC is free software; you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 3, or (at your option)
any later version.

GCC is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with GCC; see the file COPYING3.  If not see
<http://www.gnu.org/licenses/>.  */

#include <libgccjit.h>

#include <stdlib.h>
#include <stdio.h>

void
create_code (gcc_jit_context *ctxt)
{
  /*
    Simple sum-of-squares, to test conditionals and looping

    int loop_test (int n)
    {
      int i;
      int sum = 0;
      for (i = 0; i < n ; i ++)
      {
        sum += i * i;
      }
      return sum;
   */
  gcc_jit_type *the_type =
    gcc_jit_context_get_type (ctxt, GCC_JIT_TYPE_INT);
  gcc_jit_type *return_type = the_type;

  gcc_jit_param *n =
    gcc_jit_context_new_param (ctxt, NULL, the_type, "n");
  gcc_jit_param *params[1] = {n};
  gcc_jit_function *func =
    gcc_jit_context_new_function (ctxt, NULL,
                                  GCC_JIT_FUNCTION_EXPORTED,
```

(continues on next page)

(continued from previous page)

```
                                return_type,
                                "loop_test",
                                1, params, 0);

/* Build locals:  */
gcc_jit_lvalue *i =
  gcc_jit_function_new_local (func, NULL, the_type, "i");
gcc_jit_lvalue *sum =
  gcc_jit_function_new_local (func, NULL, the_type, "sum");

gcc_jit_block *b_initial =
  gcc_jit_function_new_block (func, "initial");
gcc_jit_block *b_loop_cond =
  gcc_jit_function_new_block (func, "loop_cond");
gcc_jit_block *b_loop_body =
  gcc_jit_function_new_block (func, "loop_body");
gcc_jit_block *b_after_loop =
  gcc_jit_function_new_block (func, "after_loop");

/* sum = 0; */
gcc_jit_block_add_assignment (
  b_initial, NULL,
  sum,
  gcc_jit_context_zero (ctxt, the_type));

/* i = 0; */
gcc_jit_block_add_assignment (
  b_initial, NULL,
  i,
  gcc_jit_context_zero (ctxt, the_type));

gcc_jit_block_end_with_jump (b_initial, NULL, b_loop_cond);

/* if (i >= n) */
gcc_jit_block_end_with_conditional (
  b_loop_cond, NULL,
  gcc_jit_context_new_comparison (
     ctxt, NULL,
     GCC_JIT_COMPARISON_GE,
     gcc_jit_lvalue_as_rvalue (i),
     gcc_jit_param_as_rvalue (n)),
  b_after_loop,
  b_loop_body);

/* sum += i * i */
gcc_jit_block_add_assignment_op (
  b_loop_body, NULL,
  sum,
  GCC_JIT_BINARY_OP_PLUS,
  gcc_jit_context_new_binary_op (
    ctxt, NULL,
```

(continues on next page)

(continued from previous page)

```
      GCC_JIT_BINARY_OP_MULT, the_type,
      gcc_jit_lvalue_as_rvalue (i),
      gcc_jit_lvalue_as_rvalue (i)));

  /* i++ */
  gcc_jit_block_add_assignment_op (
    b_loop_body, NULL,
    i,
    GCC_JIT_BINARY_OP_PLUS,
    gcc_jit_context_one (ctxt, the_type));

  gcc_jit_block_end_with_jump (b_loop_body, NULL, b_loop_cond);

  /* return sum */
  gcc_jit_block_end_with_return (
    b_after_loop,
    NULL,
    gcc_jit_lvalue_as_rvalue (sum));
}

int
main (int argc, char **argv)
{
  gcc_jit_context *ctxt = NULL;
  gcc_jit_result *result = NULL;

  /* Get a "context" object for working with the library.  */
  ctxt = gcc_jit_context_acquire ();
  if (!ctxt)
    {
      fprintf (stderr, "NULL ctxt");
      goto error;
    }

  /* Set some options on the context.
     Let's see the code being generated, in assembler form.  */
  gcc_jit_context_set_bool_option (
    ctxt,
    GCC_JIT_BOOL_OPTION_DUMP_GENERATED_CODE,
    0);

  /* Populate the context.  */
  create_code (ctxt);

  /* Compile the code.  */
  result = gcc_jit_context_compile (ctxt);
  if (!result)
    {
      fprintf (stderr, "NULL result");
      goto error;
    }
```

(continues on next page)

(continued from previous page)

```
  /* Extract the generated code from "result".  */
  typedef int (*loop_test_fn_type) (int);
  loop_test_fn_type loop_test =
    (loop_test_fn_type)gcc_jit_result_get_code (result, "loop_test");
  if (!loop_test)
    {
      fprintf (stderr, "NULL loop_test");
      goto error;
    }

  /* Run the generated code.  */
  int val = loop_test (10);
  printf("loop_test returned: %d\n", val);

 error:
  gcc_jit_context_release (ctxt);
  gcc_jit_result_release (result);
  return 0;
}
```

Building and running it:

```
$ gcc \
    tut03-sum-of-squares.c \
    -o tut03-sum-of-squares \
    -lgccjit

# Run the built program:
$ ./tut03-sum-of-squares
loop_test returned: 285
```

1.4 Tutorial part 4: Adding JIT-compilation to a toy interpreter

In this example we construct a "toy" interpreter, and add JIT-compilation to it.

1.4.1 Our toy interpreter

It's a stack-based interpreter, and is intended as a (very simple) example of the kind of bytecode interpreter seen in dynamic languages such as Python, Ruby etc.

For the sake of simplicity, our toy virtual machine is very limited:

- The only data type is *int*
- It can only work on one function at a time (so that the only function call that can be made is to recurse).
- Functions can only take one parameter.

- Functions have a stack of *int* values.
- We'll implement function call within the interpreter by calling a function in our implementation, rather than implementing our own frame stack.
- The parser is only good enough to get the examples to work.

Naturally, a real interpreter would be much more complicated that this.

The following operations are supported:

Operation	Meaning	Old Stack	New Stack
DUP	Duplicate top of stack.	`[..., x]`	`[..., x, x]`
ROT	Swap top two elements of stack.	`[..., x, y]`	`[..., y, x]`
BINARY_ADD	Add the top two elements on the stack.	`[..., x, y]`	`[..., (x+y)]`
BINARY_SUBTRACT	Likewise, but subtract.	`[..., x, y]`	`[..., (x-y)]`
BINARY_MULT	Likewise, but multiply.	`[..., x, y]`	`[..., (x*y)]`
BINARY_COMPARE_LT	Compare the top two elements on the stack and push a nonzero/zero if (x<y).	`[..., x, y]`	`[..., (x<y)]`
RECURSE	Recurse, passing the top of the stack, and popping the result.	`[..., x]`	`[..., fn(x)]`
RETURN	Return the top of the stack.	`[x]`	`[]`
PUSH_CONST *arg*	Push an int const.	`[...]`	`[..., arg]`
JUMP_ABS_IF_TRUE *arg*	Pop; if top of stack was nonzero, jump to `arg`.	`[..., x]`	`[...]`

Programs can be interpreted, disassembled, and compiled to machine code.

The interpreter reads `.toy` scripts. Here's what a simple recursive factorial program looks like, the script `factorial.toy`. The parser ignores lines beginning with a #.

```
# Simple recursive factorial implementation, roughly equivalent to:
#
#  int factorial (int arg)
#  {
#    if (arg < 2)
#      return arg
#    return arg * factorial (arg - 1)
#  }

# Initial state:
# stack: [arg]

# 0:
DUP
# stack: [arg, arg]

# 1:
```

(continues on next page)

(continued from previous page)

```
PUSH_CONST 2
# stack: [arg, arg, 2]

# 2:
BINARY_COMPARE_LT
# stack: [arg, (arg < 2)]

# 3:
JUMP_ABS_IF_TRUE 9
# stack: [arg]

# 4:
DUP
# stack: [arg, arg]

# 5:
PUSH_CONST 1
# stack: [arg, arg, 1]

# 6:
BINARY_SUBTRACT
# stack: [arg,  (arg - 1)

# 7:
RECURSE
# stack: [arg, factorial(arg - 1)]

# 8:
BINARY_MULT
# stack: [arg * factorial(arg - 1)]

# 9:
RETURN
```

The interpreter is a simple infinite loop with a big `switch` statement based on what the next opcode is:

```
static int
toyvm_function_interpret (toyvm_function *fn, int arg, FILE *trace)
{
  toyvm_frame frame;
#define PUSH(ARG) (toyvm_frame_push (&frame, (ARG)))
#define POP(ARG) (toyvm_frame_pop (&frame))

  frame.frm_function = fn;
  frame.frm_pc = 0;
  frame.frm_cur_depth = 0;

  PUSH (arg);
```

(continues on next page)

(continued from previous page)

```
while (1)
  {
    toyvm_op *op;
    int x, y;
    assert (frame.frm_pc < fn->fn_num_ops);
    op = &fn->fn_ops[frame.frm_pc++];

    if (trace)
      {
        toyvm_frame_dump_stack (&frame, trace);
        toyvm_function_disassemble_op (fn, op, frame.frm_pc, trace);
      }

    switch (op->op_opcode)
      {
        /* Ops taking no operand.  */
      case DUP:
        x = POP ();
        PUSH (x);
        PUSH (x);
        break;

      case ROT:
        y = POP ();
        x = POP ();
        PUSH (y);
        PUSH (x);
        break;

      case BINARY_ADD:
        y = POP ();
        x = POP ();
        PUSH (x + y);
        break;

      case BINARY_SUBTRACT:
        y = POP ();
        x = POP ();
        PUSH (x - y);
        break;

      case BINARY_MULT:
        y = POP ();
        x = POP ();
        PUSH (x * y);
        break;

      case BINARY_COMPARE_LT:
        y = POP ();
        x = POP ();
        PUSH (x < y);
```

(continues on next page)

(continued from previous page)

```
          break;

        case RECURSE:
          x = POP ();
          x = toyvm_function_interpret (fn, x, trace);
          PUSH (x);
          break;

        case RETURN:
          return POP ();

          /* Ops taking an operand.  */
        case PUSH_CONST:
          PUSH (op->op_operand);
          break;

        case JUMP_ABS_IF_TRUE:
          x = POP ();
          if (x)
            frame.frm_pc = op->op_operand;
          break;

        default:
          assert (0); /* unknown opcode */

        } /* end of switch on opcode */
    } /* end of while loop */

#undef PUSH
#undef POP
}
```

1.4.2 Compiling to machine code

We want to generate machine code that can be cast to this type and then directly executed in-process:

```
typedef int (*toyvm_compiled_code) (int);
```

The lifetime of the code is tied to that of a `gcc_jit_result*`. We'll handle this by bundling them up in a structure, so that we can clean them up together by calling `gcc_jit_result_release()`:

```

struct toyvm_compiled_function
{
  gcc_jit_result *cf_jit_result;
  toyvm_compiled_code cf_code;
```

(continues on next page)

(continued from previous page)

```
};
```

Our compiler isn't very sophisticated; it takes the implementation of each opcode above, and maps it directly to the operations supported by the libgccjit API.

How should we handle the stack? In theory we could calculate what the stack depth will be at each opcode, and optimize away the stack manipulation "by hand". We'll see below that libgccjit is able to do this for us, so we'll implement stack manipulation in a direct way, by creating a `stack` array and `stack_depth` variables, local within the generated function, equivalent to this C code:

```
int stack_depth;
int stack[MAX_STACK_DEPTH];
```

We'll also have local variables `x` and `y` for use when implementing the opcodes, equivalent to this:

```
int x;
int y;
```

This means our compiler has the following state:

```

struct compilation_state
{
  gcc_jit_context *ctxt;

  gcc_jit_type *int_type;
  gcc_jit_type *bool_type;
  gcc_jit_type *stack_type; /* int[MAX_STACK_DEPTH] */

  gcc_jit_rvalue *const_one;

  gcc_jit_function *fn;
  gcc_jit_param *param_arg;
  gcc_jit_lvalue *stack;
  gcc_jit_lvalue *stack_depth;
  gcc_jit_lvalue *x;
  gcc_jit_lvalue *y;

  gcc_jit_location *op_locs[MAX_OPS];
  gcc_jit_block *initial_block;
  gcc_jit_block *op_blocks[MAX_OPS];

};
```

1.4.3 Setting things up

First we create our types:

```
  state.int_type =
    gcc_jit_context_get_type (state.ctxt, GCC_JIT_TYPE_INT);
  state.bool_type =
    gcc_jit_context_get_type (state.ctxt, GCC_JIT_TYPE_BOOL);
  state.stack_type =
    gcc_jit_context_new_array_type (state.ctxt, NULL,
                                    state.int_type, MAX_STACK_DEPTH);
```

along with extracting a useful *int* constant:

```
  state.const_one = gcc_jit_context_one (state.ctxt, state.int_type);
```

We'll implement push and pop in terms of the `stack` array and `stack_depth`. Here are helper functions for adding statements to a block, implementing pushing and popping values:

```
static void
add_push (compilation_state *state,
          gcc_jit_block *block,
          gcc_jit_rvalue *rvalue,
          gcc_jit_location *loc)
{
  /* stack[stack_depth] = RVALUE */
  gcc_jit_block_add_assignment (
    block,
    loc,
    /* stack[stack_depth] */
    gcc_jit_context_new_array_access (
      state->ctxt,
      loc,
      gcc_jit_lvalue_as_rvalue (state->stack),
      gcc_jit_lvalue_as_rvalue (state->stack_depth)),
    rvalue);

  /* "stack_depth++;".  */
  gcc_jit_block_add_assignment_op (
    block,
    loc,
    state->stack_depth,
    GCC_JIT_BINARY_OP_PLUS,
    state->const_one);
}

static void
add_pop (compilation_state *state,
         gcc_jit_block *block,
```

(continues on next page)

(continued from previous page)

```
        gcc_jit_lvalue *lvalue,
        gcc_jit_location *loc)
{
  /* "--stack_depth;".  */
  gcc_jit_block_add_assignment_op (
    block,
    loc,
    state->stack_depth,
    GCC_JIT_BINARY_OP_MINUS,
    state->const_one);

  /* "LVALUE = stack[stack_depth];".  */
  gcc_jit_block_add_assignment (
    block,
    loc,
    lvalue,
    /* stack[stack_depth] */
    gcc_jit_lvalue_as_rvalue (
      gcc_jit_context_new_array_access (
        state->ctxt,
        loc,
        gcc_jit_lvalue_as_rvalue (state->stack),
        gcc_jit_lvalue_as_rvalue (state->stack_depth))));
}
```

We will support single-stepping through the generated code in the debugger, so we need to create `gcc_jit_location` instances, one per operation in the source code. These will reference the lines of e.g. `factorial.toy`.

```
  for (pc = 0; pc < fn->fn_num_ops; pc++)
    {
      toyvm_op *op = &fn->fn_ops[pc];

      state.op_locs[pc] = gcc_jit_context_new_location (state.ctxt,
                                                        fn->fn_filename,
                                                        op->op_linenum,
                                                        0); /* column */
    }
```

Let's create the function itself. As usual, we create its parameter first, then use the parameter to create the function:

```
  state.param_arg =
    gcc_jit_context_new_param (state.ctxt, state.op_locs[0],
                               state.int_type, "arg");
  state.fn =
    gcc_jit_context_new_function (state.ctxt,
                                  state.op_locs[0],
                                  GCC_JIT_FUNCTION_EXPORTED,
```

(continues on next page)

(continued from previous page)

```
                                state.int_type,
                                funcname,
                                1, &state.param_arg, 0);
```

We create the locals within the function.

```
state.stack =
  gcc_jit_function_new_local (state.fn, NULL,
                              state.stack_type, "stack");
state.stack_depth =
  gcc_jit_function_new_local (state.fn, NULL,
                              state.int_type, "stack_depth");
state.x =
  gcc_jit_function_new_local (state.fn, NULL,
                              state.int_type, "x");
state.y =
  gcc_jit_function_new_local (state.fn, NULL,
                              state.int_type, "y");
```

1.4.4 Populating the function

There's some one-time initialization, and the API treats the first block you create as the entrypoint of the function, so we need to create that block first:

```
state.initial_block = gcc_jit_function_new_block (state.fn, "initial");
```

We can now create blocks for each of the operations. Most of these will be consolidated into larger blocks when the optimizer runs.

```
for (pc = 0; pc < fn->fn_num_ops; pc++)
  {
    char buf[100];
    sprintf (buf, "instr%i", pc);
    state.op_blocks[pc] = gcc_jit_function_new_block (state.fn, buf);
  }
```

Now that we have a block it can jump to when it's done, we can populate the initial block:

```
/* "stack_depth = 0;".  */
gcc_jit_block_add_assignment (
  state.initial_block,
  state.op_locs[0],
  state.stack_depth,
  gcc_jit_context_zero (state.ctxt, state.int_type));
```

(continues on next page)

(continued from previous page)

```
/* "PUSH (arg);".  */
add_push (&state,
          state.initial_block,
          gcc_jit_param_as_rvalue (state.param_arg),
          state.op_locs[0]);

/* ...and jump to insn 0.  */
gcc_jit_block_end_with_jump (state.initial_block,
                             state.op_locs[0],
                             state.op_blocks[0]);
```

We can now populate the blocks for the individual operations. We loop through them, adding instructions to their blocks:

```
for (pc = 0; pc < fn->fn_num_ops; pc++)
  {
    gcc_jit_location *loc = state.op_locs[pc];

    gcc_jit_block *block = state.op_blocks[pc];
    gcc_jit_block *next_block = (pc < fn->fn_num_ops
                                 ? state.op_blocks[pc + 1]
                                 : NULL);

    toyvm_op *op;
    op = &fn->fn_ops[pc];
```

We're going to have another big `switch` statement for implementing the opcodes, this time for compiling them, rather than interpreting them. It's helpful to have macros for implementing push and pop, so that we can make the `switch` statement that's coming up look as much as possible like the one above within the interpreter:

```
#define X_EQUALS_POP()\
      add_pop (&state, block, state.x, loc)
#define Y_EQUALS_POP()\
      add_pop (&state, block, state.y, loc)
#define PUSH_RVALUE(RVALUE)\
      add_push (&state, block, (RVALUE), loc)
#define PUSH_X()\
      PUSH_RVALUE (gcc_jit_lvalue_as_rvalue (state.x))
#define PUSH_Y() \
      PUSH_RVALUE (gcc_jit_lvalue_as_rvalue (state.y))
```

Note: A particularly clever implementation would have an *identical* `switch` statement shared by the interpreter and the compiler, with some preprocessor "magic". We're not doing that here, for

the sake of simplicity.

When I first implemented this compiler, I accidentally missed an edit when copying and pasting the `Y_EQUALS_POP` macro, so that popping the stack into `y` instead erroneously assigned it to `x`, leaving `y` uninitialized.

To track this kind of thing down, we can use `gcc_jit_block_add_comment()` to add descriptive comments to the internal representation. This is invaluable when looking through the generated IR for, say `factorial`:

```
gcc_jit_block_add_comment (block, loc, opcode_names[op->op_opcode]);
```

We can now write the big `switch` statement that implements the individual opcodes, populating the relevant block with statements:

```
switch (op->op_opcode)
  {
  case DUP:
    X_EQUALS_POP ();
    PUSH_X ();
    PUSH_X ();
    break;

  case ROT:
    Y_EQUALS_POP ();
    X_EQUALS_POP ();
    PUSH_Y ();
    PUSH_X ();
    break;

  case BINARY_ADD:
    Y_EQUALS_POP ();
    X_EQUALS_POP ();
    PUSH_RVALUE (
     gcc_jit_context_new_binary_op (
       state.ctxt,
       loc,
       GCC_JIT_BINARY_OP_PLUS,
       state.int_type,
       gcc_jit_lvalue_as_rvalue (state.x),
       gcc_jit_lvalue_as_rvalue (state.y)));
    break;

  case BINARY_SUBTRACT:
    Y_EQUALS_POP ();
    X_EQUALS_POP ();
    PUSH_RVALUE (
     gcc_jit_context_new_binary_op (
       state.ctxt,
```

(continues on next page)

(continued from previous page)

```
            loc,
            GCC_JIT_BINARY_OP_MINUS,
            state.int_type,
            gcc_jit_lvalue_as_rvalue (state.x),
            gcc_jit_lvalue_as_rvalue (state.y)));
          break;

        case BINARY_MULT:
          Y_EQUALS_POP ();
          X_EQUALS_POP ();
          PUSH_RVALUE (
           gcc_jit_context_new_binary_op (
            state.ctxt,
            loc,
            GCC_JIT_BINARY_OP_MULT,
            state.int_type,
            gcc_jit_lvalue_as_rvalue (state.x),
            gcc_jit_lvalue_as_rvalue (state.y)));
          break;

        case BINARY_COMPARE_LT:
          Y_EQUALS_POP ();
          X_EQUALS_POP ();
          PUSH_RVALUE (
            /* cast of bool to int */
            gcc_jit_context_new_cast (
              state.ctxt,
              loc,
              /* (x < y) as a bool */
              gcc_jit_context_new_comparison (
                state.ctxt,
                loc,
                GCC_JIT_COMPARISON_LT,
                gcc_jit_lvalue_as_rvalue (state.x),
                gcc_jit_lvalue_as_rvalue (state.y)),
              state.int_type));
          break;

        case RECURSE:
          {
            X_EQUALS_POP ();
            gcc_jit_rvalue *arg = gcc_jit_lvalue_as_rvalue (state.x);
            PUSH_RVALUE (
              gcc_jit_context_new_call (
                state.ctxt,
                loc,
                state.fn,
                1, &arg));
            break;
          }
```

(continues on next page)

(continued from previous page)

```
      case RETURN:
        X_EQUALS_POP ();
        gcc_jit_block_end_with_return (
          block,
          loc,
          gcc_jit_lvalue_as_rvalue (state.x));
        break;

        /* Ops taking an operand.  */
      case PUSH_CONST:
        PUSH_RVALUE (
          gcc_jit_context_new_rvalue_from_int (
            state.ctxt,
            state.int_type,
            op->op_operand));
        break;

      case JUMP_ABS_IF_TRUE:
        X_EQUALS_POP ();
        gcc_jit_block_end_with_conditional (
          block,
          loc,
          /* "(bool)x".  */
          gcc_jit_context_new_cast (
            state.ctxt,
            loc,
            gcc_jit_lvalue_as_rvalue (state.x),
            state.bool_type),
          state.op_blocks[op->op_operand], /* on_true */
          next_block); /* on_false */
        break;

      default:
        assert(0);
      } /* end of switch on opcode */
```

Every block must be terminated, via a call to one of the `gcc_jit_block_end_with_` entrypoints. This has been done for two of the opcodes, but we need to do it for the other ones, by jumping to the next block.

```
    if (op->op_opcode != JUMP_ABS_IF_TRUE
        && op->op_opcode != RETURN)
      gcc_jit_block_end_with_jump (
        block,
        loc,
        next_block);
```

This is analogous to simply incrementing the program counter.

1.4.5 Verifying the control flow graph

Having finished looping over the blocks, the context is complete.

As before, we can verify that the control flow and statements are sane by using `gcc_jit_function_dump_to_dot()`:

```
gcc_jit_function_dump_to_dot (state.fn, "/tmp/factorial.dot");
```

and viewing the result. Note how the label names, comments, and variable names show up in the dump, to make it easier to spot errors in our compiler.

1.4.6 Compiling the context

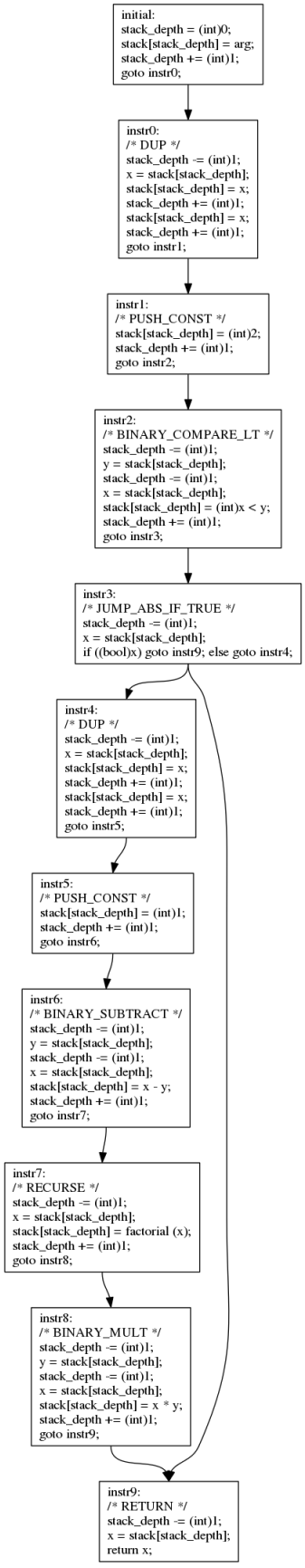
Having finished looping over the blocks and populating them with statements, the context is complete.

We can now compile it, and extract machine code from the result:

```
gcc_jit_result *jit_result = gcc_jit_context_compile (state.ctxt);
gcc_jit_context_release (state.ctxt);

toyvm_compiled_function *toyvm_result =
  (toyvm_compiled_function *)calloc (1, sizeof (toyvm_compiled_function));
if (!toyvm_result)
  {
    fprintf (stderr, "out of memory allocating toyvm_compiled_function\n");
    gcc_jit_result_release (jit_result);
    return NULL;
  }

toyvm_result->cf_jit_result = jit_result;
toyvm_result->cf_code =
  (toyvm_compiled_code)gcc_jit_result_get_code (jit_result,
                                                funcname);
```

We can now run the result:

```
toyvm_compiled_function *compiled_fn
  = toyvm_function_compile (fn);

toyvm_compiled_code code = compiled_fn->cf_code;
printf ("compiler result: %d\n",
        code (atoi (argv[2])));

gcc_jit_result_release (compiled_fn->cf_jit_result);
free (compiled_fn);
```


initial:
stack_depth = (int)0;
stack[stack_depth] = arg;
stack_depth += (int)1;
goto instr0;
instr0:
/* DUP */
stack_depth -= (int)1;
x = stack[stack_depth];
stack[stack_depth] = x;
stack_depth += (int)1;
stack[stack_depth] = x;
stack_depth += (int)1;
goto instr1;
instr1:
/* PUSH_CONST */
stack[stack_depth] = (int)2;
stack_depth += (int)1;
goto instr2;
instr2:
/* BINARY_COMPARE_LT */
stack_depth -= (int)1;
y = stack[stack_depth];
stack_depth -= (int)1;
x = stack[stack_depth];
stack[stack_depth] = (int)x < y;
stack_depth += (int)1;
goto instr3;
instr3:
/* JUMP_ABS_IF_TRUE */
stack_depth -= (int)1;
x = stack[stack_depth];
if ((bool)x) goto instr9; else goto instr4;
instr4:
/* DUP */
stack_depth -= (int)1;
x = stack[stack_depth];
stack[stack_depth] = x;
stack_depth += (int)1;
stack[stack_depth] = x;
stack_depth += (int)1;
goto instr5;
instr5:
/* PUSH_CONST */
stack[stack_depth] = (int)1;
stack_depth += (int)1;
goto instr6;
instr6:
/* BINARY_SUBTRACT */
stack_depth -= (int)1;
y = stack[stack_depth];
stack_depth -= (int)1;
x = stack[stack_depth];
stack[stack_depth] = x - y;
stack_depth += (int)1;
goto instr7;
instr7:
/* RECURSE */
stack_depth -= (int)1;
x = stack[stack_depth];
stack[stack_depth] = factorial (x);
stack_depth += (int)1;
goto instr8;
instr8:
/* BINARY_MULT */
stack_depth -= (int)1;
y = stack[stack_depth];
stack_depth -= (int)1;
x = stack[stack_depth];
stack[stack_depth] = x * y;
stack_depth += (int)1;
goto instr9;
instr9:
/* RETURN */
stack_depth -= (int)1;
x = stack[stack_depth];
return x;

1.4.7 Single-stepping through the generated code

It's possible to debug the generated code. To do this we need to both:

- Set up source code locations for our statements, so that we can meaningfully step through the code. We did this above by calling `gcc_jit_context_new_location()` and using the results.
- Enable the generation of debugging information, by setting `GCC_JIT_BOOL_OPTION_DEBUGINFO` on the `gcc_jit_context` via `gcc_jit_context_set_bool_option()`:

```
gcc_jit_context_set_bool_option (
  ctxt,
  GCC_JIT_BOOL_OPTION_DEBUGINFO,
  1);
```

Having done this, we can put a breakpoint on the generated function:

```
$ gdb --args ./toyvm factorial.toy 10
(gdb) break factorial
Function "factorial" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (factorial) pending.
(gdb) run
Breakpoint 1, factorial (arg=10) at factorial.toy:14
14    DUP
```

We've set up location information, which references `factorial.toy`. This allows us to use e.g. `list` to see where we are in the script:

```
(gdb) list
9
10    # Initial state:
11    # stack: [arg]
12
13    # 0:
14    DUP
15    # stack: [arg, arg]
16
17    # 1:
18    PUSH_CONST 2
```

and to step through the function, examining the data:

```
(gdb) n
18    PUSH_CONST 2
(gdb) n
22    BINARY_COMPARE_LT
(gdb) print stack
$5 = {10, 10, 2, 0, -7152, 32767, 0, 0}
(gdb) print stack_depth
$6 = 3
```

You'll see that the parts of the `stack` array that haven't been touched yet are uninitialized.

Note: Turning on optimizations may lead to unpredictable results when stepping through the generated code: the execution may appear to "jump around" the source code. This is analogous to turning up the optimization level in a regular compiler.

1.4.8 Examining the generated code

How good is the optimized code?

We can turn up optimizations, by calling `gcc_jit_context_set_int_option()` with `GCC_JIT_INT_OPTION_OPTIMIZATION_LEVEL`:

```
gcc_jit_context_set_int_option (
  ctxt,
  GCC_JIT_INT_OPTION_OPTIMIZATION_LEVEL,
  3);
```

One of GCC's internal representations is called "gimple". A dump of the initial gimple representation of the code can be seen by setting:

```
gcc_jit_context_set_bool_option (ctxt,
                                 GCC_JIT_BOOL_OPTION_DUMP_INITIAL_GIMPLE,
                                 1);
```

With optimization on and source locations displayed, this gives:

```
factorial (signed int arg)
{
  <unnamed type> D.80;
  signed int D.81;
  signed int D.82;
  signed int D.83;
  signed int D.84;
  signed int D.85;
  signed int y;
  signed int x;
  signed int stack_depth;
  signed int stack[8];

  try
    {
      initial:
      stack_depth = 0;
      stack[stack_depth] = arg;
      stack_depth = stack_depth + 1;
      goto instr0;
      instr0:
      /* DUP */:
      stack_depth = stack_depth + -1;
      x = stack[stack_depth];
```

(continues on next page)

(continued from previous page)

```
      stack[stack_depth] = x;
      stack_depth = stack_depth + 1;
      stack[stack_depth] = x;
      stack_depth = stack_depth + 1;
      goto instr1;
      instr1:
      /* PUSH_CONST */:
      stack[stack_depth] = 2;
      stack_depth = stack_depth + 1;
      goto instr2;

      /* etc */
```

You can see the generated machine code in assembly form via:

```
gcc_jit_context_set_bool_option (
  ctxt,
  GCC_JIT_BOOL_OPTION_DUMP_GENERATED_CODE,
  1);
result = gcc_jit_context_compile (ctxt);
```

which shows that (on this x86_64 box) the compiler has unrolled the loop and is using MMX instructions to perform several multiplications simultaneously:

```
        .file   "fake.c"
        .text
.Ltext0:
        .p2align 4,,15
        .globl  factorial
        .type   factorial, @function
factorial:
.LFB0:
        .file 1 "factorial.toy"
        .loc 1 14 0
        .cfi_startproc
.LVL0:
.L2:
        .loc 1 26 0
        cmpl    $1, %edi
        jle     .L13
        leal    -1(%rdi), %edx
        movl    %edx, %ecx
        shrl    $2, %ecx
        leal    0(,%rcx,4), %esi
        testl   %esi, %esi
        je      .L14
        cmpl    $9, %edx
        jbe     .L14
        leal    -2(%rdi), %eax
        movl    %eax, -16(%rsp)
        leal    -3(%rdi), %eax
```

(continues on next page)

(continued from previous page)

```
        movd    -16(%rsp), %xmm0
        movl    %edi, -16(%rsp)
        movl    %eax, -12(%rsp)
        movd    -16(%rsp), %xmm1
        xorl    %eax, %eax
        movl    %edx, -16(%rsp)
        movd    -12(%rsp), %xmm4
        movd    -16(%rsp), %xmm6
        punpckldq       %xmm4, %xmm0
        movdqa  .LC1(%rip), %xmm4
        punpckldq       %xmm6, %xmm1
        punpcklqdq      %xmm0, %xmm1
        movdqa  .LC0(%rip), %xmm0
        jmp     .L5
        # etc - edited for brevity
```

This is clearly overkill for a function that will likely overflow the `int` type before the vectorization is worthwhile - but then again, this is a toy example.

Turning down the optimization level to 2:

```
gcc_jit_context_set_int_option (
  ctxt,
  GCC_JIT_INT_OPTION_OPTIMIZATION_LEVEL,
  3);
```

yields this code, which is simple enough to quote in its entirety:

```
        .file   "fake.c"
        .text
        .p2align 4,,15
        .globl  factorial
        .type   factorial, @function
factorial:
.LFB0:
        .cfi_startproc
.L2:
        cmpl    $1, %edi
        jle     .L8
        movl    $1, %edx
        jmp     .L4
        .p2align 4,,10
        .p2align 3
.L6:
        movl    %eax, %edi
.L4:
.L5:
        leal    -1(%rdi), %eax
        imull   %edi, %edx
        cmpl    $1, %eax
        jne     .L6
```

(continues on next page)

(continued from previous page)

```
.L3:
.L7:
        imull   %edx, %eax
        ret
.L8:
        movl    %edi, %eax
        movl    $1, %edx
        jmp     .L7
        .cfi_endproc
.LFE0:
        .size   factorial, .-factorial
        .ident  "GCC: (GNU) 4.9.0 20131023 (Red Hat 0.2)"
        .section        .note.GNU-stack,"",@progbits
```

Note that the stack pushing and popping have been eliminated, as has the recursive call (in favor of an iteration).

1.4.9 Putting it all together

The complete example can be seen in the source tree at `gcc/jit/docs/examples/tut04-toyvm/toyvm.c`

along with a Makefile and a couple of sample .toy scripts:

```
$ ls -al
drwxrwxr-x. 2 david david   4096 Sep 19 17:46 .
drwxrwxr-x. 3 david david   4096 Sep 19 15:26 ..
-rw-rw-r--. 1 david david    615 Sep 19 12:43 factorial.toy
-rw-rw-r--. 1 david david    834 Sep 19 13:08 fibonacci.toy
-rw-rw-r--. 1 david david    238 Sep 19 14:22 Makefile
-rw-rw-r--. 1 david david  16457 Sep 19 17:07 toyvm.c

$ make toyvm
g++ -Wall -g -o toyvm toyvm.c -lgccjit

$ ./toyvm factorial.toy 10
interpreter result: 3628800
compiler result: 3628800

$ ./toyvm fibonacci.toy 10
interpreter result: 55
compiler result: 55
```

1.4.10 Behind the curtain: How does our code get optimized?

Our example is done, but you may be wondering about exactly how the compiler turned what we gave it into the machine code seen above.

We can examine what the compiler is doing in detail by setting:

```
gcc_jit_context_set_bool_option (state.ctxt,
                                 GCC_JIT_BOOL_OPTION_DUMP_EVERYTHING,
                                 1);
gcc_jit_context_set_bool_option (state.ctxt,
                                 GCC_JIT_BOOL_OPTION_KEEP_INTERMEDIATES,
                                 1);
```

This will dump detailed information about the compiler's state to a directory under `/tmp`, and keep it from being cleaned up.

The precise names and their formats of these files is subject to change. Higher optimization levels lead to more files. Here's what I saw (edited for brevity; there were almost 200 files):

```
intermediate files written to /tmp/libgccjit-KPQbGw
$ ls /tmp/libgccjit-KPQbGw/
fake.c.000i.cgraph
fake.c.000i.type-inheritance
fake.c.004t.gimple
fake.c.007t.omplower
fake.c.008t.lower
fake.c.011t.eh
fake.c.012t.cfg
fake.c.014i.visibility
fake.c.015i.early_local_cleanups
fake.c.016t.ssa
# etc
```

The gimple code is converted into Static Single Assignment form, with annotations for use when generating the debuginfo:

```
$ less /tmp/libgccjit-KPQbGw/fake.c.016t.ssa
```

```
;; Function factorial (factorial, funcdef_no=0, decl_uid=53, symbol_order=0)

factorial (signed int arg)
{
  signed int stack[8];
  signed int stack_depth;
  signed int x;
  signed int y;
  <unnamed type> _20;
  signed int _21;
  signed int _38;
  signed int _44;
  signed int _51;
```

(continues on next page)

(continued from previous page)

```
  signed int _56;

initial:
  stack_depth_3 = 0;
  # DEBUG stack_depth => stack_depth_3
  stack[stack_depth_3] = arg_5(D);
  stack_depth_7 = stack_depth_3 + 1;
  # DEBUG stack_depth => stack_depth_7
  # DEBUG instr0 => NULL
  # DEBUG /* DUP */ => NULL
  stack_depth_8 = stack_depth_7 + -1;
  # DEBUG stack_depth => stack_depth_8
  x_9 = stack[stack_depth_8];
  # DEBUG x => x_9
  stack[stack_depth_8] = x_9;
  stack_depth_11 = stack_depth_8 + 1;
  # DEBUG stack_depth => stack_depth_11
  stack[stack_depth_11] = x_9;
  stack_depth_13 = stack_depth_11 + 1;
  # DEBUG stack_depth => stack_depth_13
  # DEBUG instr1 => NULL
  # DEBUG /* PUSH_CONST */ => NULL
  stack[stack_depth_13] = 2;

  /* etc; edited for brevity */
```

We can perhaps better see the code by turning off `GCC_JIT_BOOL_OPTION_DEBUGINFO` to suppress all those `DEBUG` statements, giving:

```
$ less /tmp/libgccjit-1Hywc0/fake.c.016t.ssa
```

```
;; Function factorial (factorial, funcdef_no=0, decl_uid=53, symbol_order=0)

factorial (signed int arg)
{
  signed int stack[8];
  signed int stack_depth;
  signed int x;
  signed int y;
  <unnamed type> _20;
  signed int _21;
  signed int _38;
  signed int _44;
  signed int _51;
  signed int _56;

initial:
  stack_depth_3 = 0;
  stack[stack_depth_3] = arg_5(D);
  stack_depth_7 = stack_depth_3 + 1;
  stack_depth_8 = stack_depth_7 + -1;
```

(continues on next page)

(continued from previous page)

```
  x_9 = stack[stack_depth_8];
  stack[stack_depth_8] = x_9;
  stack_depth_11 = stack_depth_8 + 1;
  stack[stack_depth_11] = x_9;
  stack_depth_13 = stack_depth_11 + 1;
  stack[stack_depth_13] = 2;
  stack_depth_15 = stack_depth_13 + 1;
  stack_depth_16 = stack_depth_15 + -1;
  y_17 = stack[stack_depth_16];
  stack_depth_18 = stack_depth_16 + -1;
  x_19 = stack[stack_depth_18];
  _20 = x_19 < y_17;
  _21 = (signed int) _20;
  stack[stack_depth_18] = _21;
  stack_depth_23 = stack_depth_18 + 1;
  stack_depth_24 = stack_depth_23 + -1;
  x_25 = stack[stack_depth_24];
  if (x_25 != 0)
    goto <bb 4> (instr9);
  else
    goto <bb 3> (instr4);

instr4:
/* DUP */:
  stack_depth_26 = stack_depth_24 + -1;
  x_27 = stack[stack_depth_26];
  stack[stack_depth_26] = x_27;
  stack_depth_29 = stack_depth_26 + 1;
  stack[stack_depth_29] = x_27;
  stack_depth_31 = stack_depth_29 + 1;
  stack[stack_depth_31] = 1;
  stack_depth_33 = stack_depth_31 + 1;
  stack_depth_34 = stack_depth_33 + -1;
  y_35 = stack[stack_depth_34];
  stack_depth_36 = stack_depth_34 + -1;
  x_37 = stack[stack_depth_36];
  _38 = x_37 - y_35;
  stack[stack_depth_36] = _38;
  stack_depth_40 = stack_depth_36 + 1;
  stack_depth_41 = stack_depth_40 + -1;
  x_42 = stack[stack_depth_41];
  _44 = factorial (x_42);
  stack[stack_depth_41] = _44;
  stack_depth_46 = stack_depth_41 + 1;
  stack_depth_47 = stack_depth_46 + -1;
  y_48 = stack[stack_depth_47];
  stack_depth_49 = stack_depth_47 + -1;
  x_50 = stack[stack_depth_49];
  _51 = x_50 * y_48;
  stack[stack_depth_49] = _51;
  stack_depth_53 = stack_depth_49 + 1;
```

(continues on next page)

(continued from previous page)

```
  # stack_depth_1 = PHI <stack_depth_24(2), stack_depth_53(3)>
instr9:
/* RETURN */:
  stack_depth_54 = stack_depth_1 + -1;
  x_55 = stack[stack_depth_54];
  _56 = x_55;
  stack ={v} {CLOBBER};
  return _56;

}
```

Note in the above how all the `gcc_jit_block` instances we created have been consolidated into just 3 blocks in GCC's internal representation: `initial`, `instr4` and `instr9`.

Optimizing away stack manipulation

Recall our simple implementation of stack operations. Let's examine how the stack operations are optimized away.

After a pass of constant-propagation, the depth of the stack at each opcode can be determined at compile-time:

```
$ less /tmp/libgccjit-1Hywc0/fake.c.021t.ccp1
```

```
;; Function factorial (factorial, funcdef_no=0, decl_uid=53, symbol_order=0)

factorial (signed int arg)
{
  signed int stack[8];
  signed int stack_depth;
  signed int x;
  signed int y;
  <unnamed type> _20;
  signed int _21;
  signed int _38;
  signed int _44;
  signed int _51;

initial:
  stack[0] = arg_5(D);
  x_9 = stack[0];
  stack[0] = x_9;
  stack[1] = x_9;
  stack[2] = 2;
  y_17 = stack[2];
  x_19 = stack[1];
  _20 = x_19 < y_17;
  _21 = (signed int) _20;
  stack[1] = _21;
```

(continues on next page)

(continued from previous page)

```
  x_25 = stack[1];
  if (x_25 != 0)
    goto <bb 4> (instr9);
  else
    goto <bb 3> (instr4);

instr4:
/* DUP */:
  x_27 = stack[0];
  stack[0] = x_27;
  stack[1] = x_27;
  stack[2] = 1;
  y_35 = stack[2];
  x_37 = stack[1];
  _38 = x_37 - y_35;
  stack[1] = _38;
  x_42 = stack[1];
  _44 = factorial (x_42);
  stack[1] = _44;
  y_48 = stack[1];
  x_50 = stack[0];
  _51 = x_50 * y_48;
  stack[0] = _51;

instr9:
/* RETURN */:
  x_55 = stack[0];
  x_56 = x_55;
  stack ={v} {CLOBBER};
  return x_56;

}
```

Note how, in the above, all those `stack_depth` values are now just constants: we're accessing specific stack locations at each opcode.

The "esra" pass ("Early Scalar Replacement of Aggregates") breaks out our "stack" array into individual elements:

```
$ less /tmp/libgccjit-1Hywc0/fake.c.024t.esra
```

```
;; Function factorial (factorial, funcdef_no=0, decl_uid=53, symbol_order=0)

Created a replacement for stack offset: 0, size: 32: stack$0
Created a replacement for stack offset: 32, size: 32: stack$1
Created a replacement for stack offset: 64, size: 32: stack$2

Symbols to be put in SSA form
{ D.89 D.90 D.91 }
Incremental SSA update started at block: 0
Number of blocks in CFG: 5
```

(continues on next page)

(continued from previous page)

```
Number of blocks to update: 4 ( 80%)


factorial (signed int arg)
{
  signed int stack$2;
  signed int stack$1;
  signed int stack$0;
  signed int stack[8];
  signed int stack_depth;
  signed int x;
  signed int y;
  <unnamed type> _20;
  signed int _21;
  signed int _38;
  signed int _44;
  signed int _51;

initial:
  stack$0_45 = arg_5(D);
  x_9 = stack$0_45;
  stack$0_39 = x_9;
  stack$1_32 = x_9;
  stack$2_30 = 2;
  y_17 = stack$2_30;
  x_19 = stack$1_32;
  _20 = x_19 < y_17;
  _21 = (signed int) _20;
  stack$1_28 = _21;
  x_25 = stack$1_28;
  if (x_25 != 0)
    goto <bb 4> (instr9);
  else
    goto <bb 3> (instr4);

instr4:
/* DUP */:
  x_27 = stack$0_39;
  stack$0_22 = x_27;
  stack$1_14 = x_27;
  stack$2_12 = 1;
  y_35 = stack$2_12;
  x_37 = stack$1_14;
  _38 = x_37 - y_35;
  stack$1_10 = _38;
  x_42 = stack$1_10;
  _44 = factorial (x_42);
  stack$1_6 = _44;
  y_48 = stack$1_6;
  x_50 = stack$0_22;
  _51 = x_50 * y_48;
```

(continues on next page)

(continued from previous page)

```
  stack$0_1 = _51;

  # stack$0_52 = PHI <stack$0_39(2), stack$0_1(3)>
instr9:
/* RETURN */:
  x_55 = stack$0_52;
  x_56 = x_55;
  stack ={v} {CLOBBER};
  return x_56;

}
```

Hence at this point, all those pushes and pops of the stack are now simply assignments to specific temporary variables.

After some copy propagation, the stack manipulation has been completely optimized away:

```
$ less /tmp/libgccjit-1Hywc0/fake.c.026t.copyprop1
```

```
;; Function factorial (factorial, funcdef_no=0, decl_uid=53, symbol_order=0)

factorial (signed int arg)
{
  signed int stack$2;
  signed int stack$1;
  signed int stack$0;
  signed int stack[8];
  signed int stack_depth;
  signed int x;
  signed int y;
  <unnamed type> _20;
  signed int _21;
  signed int _38;
  signed int _44;
  signed int _51;

initial:
  stack$0_39 = arg_5(D);
  _20 = arg_5(D) <= 1;
  _21 = (signed int) _20;
  if (_21 != 0)
    goto <bb 4> (instr9);
  else
    goto <bb 3> (instr4);

instr4:
/* DUP */:
  _38 = arg_5(D) + -1;
  _44 = factorial (_38);
  _51 = arg_5(D) * _44;
  stack$0_1 = _51;
```

(continues on next page)

(continued from previous page)

```
  # stack$0_52 = PHI <arg_5(D)(2), _51(3)>
instr9:
/* RETURN */:
  stack ={v} {CLOBBER};
  return stack$0_52;

}
```

Later on, another pass finally eliminated `stack_depth` local and the unused parts of the *stack'* array altogether:

```
$ less /tmp/libgccjit-1Hywc0/fake.c.036t.release_ssa
```

```
;; Function factorial (factorial, funcdef_no=0, decl_uid=53, symbol_order=0)

Released 44 names, 314.29%, removed 44 holes
factorial (signed int arg)
{
  signed int stack$0;
  signed int mult_acc_1;
  <unnamed type> _5;
  signed int _6;
  signed int _7;
  signed int mul_tmp_10;
  signed int mult_acc_11;
  signed int mult_acc_13;

  # arg_9 = PHI <arg_8(D)(0)>
  # mult_acc_13 = PHI <1(0)>
initial:

  <bb 5>:
  # arg_4 = PHI <arg_9(2), _7(3)>
  # mult_acc_1 = PHI <mult_acc_13(2), mult_acc_11(3)>
  _5 = arg_4 <= 1;
  _6 = (signed int) _5;
  if (_6 != 0)
    goto <bb 4> (instr9);
  else
    goto <bb 3> (instr4);

instr4:
/* DUP */:
  _7 = arg_4 + -1;
  mult_acc_11 = mult_acc_1 * arg_4;
  goto <bb 5>;

  # stack$0_12 = PHI <arg_4(5)>
instr9:
/* RETURN */:
```

(continues on next page)

(continued from previous page)

```
  mul_tmp_10 = mult_acc_1 * stack$0_12;
  return mul_tmp_10;

}
```

Elimination of tail recursion

Another significant optimization is the detection that the call to `factorial` is tail recursion, which can be eliminated in favor of an iteration:

```
$ less /tmp/libgccjit-1Hywc0/fake.c.030t.tailr1
```

```
;; Function factorial (factorial, funcdef_no=0, decl_uid=53, symbol_order=0)


Symbols to be put in SSA form
{ D.88 }
Incremental SSA update started at block: 0
Number of blocks in CFG: 5
Number of blocks to update: 4 ( 80%)


factorial (signed int arg)
{
  signed int stack$2;
  signed int stack$1;
  signed int stack$0;
  signed int stack[8];
  signed int stack_depth;
  signed int x;
  signed int y;
  signed int mult_acc_1;
  <unnamed type> _20;
  signed int _21;
  signed int _38;
  signed int mul_tmp_44;
  signed int mult_acc_51;

  # arg_5 = PHI <arg_39(D)(0), _38(3)>
  # mult_acc_1 = PHI <1(0), mult_acc_51(3)>
initial:
  _20 = arg_5 <= 1;
  _21 = (signed int) _20;
  if (_21 != 0)
    goto <bb 4> (instr9);
  else
    goto <bb 3> (instr4);

instr4:
```

(continues on next page)

(continued from previous page)

```
/* DUP */:
  _38 = arg_5 + -1;
  mult_acc_51 = mult_acc_1 * arg_5;
  goto <bb 2> (initial);

  # stack$0_52 = PHI <arg_5(2)>
instr9:
/* RETURN */:
  stack ={v} {CLOBBER};
  mul_tmp_44 = mult_acc_1 * stack$0_52;
  return mul_tmp_44;

}
```

1.5 Tutorial part 5: Implementing an Ahead-of-Time compiler

If you have a pre-existing language frontend that's compatible with libgccjit's license, it's possible to hook it up to libgccjit as a backend. In the previous example we showed how to do that for in-memory JIT-compilation, but libgccjit can also compile code directly to a file, allowing you to implement a more traditional ahead-of-time compiler ("JIT" is something of a misnomer for this use-case).

The essential difference is to compile the context using `gcc_jit_context_compile_to_file()` rather than `gcc_jit_context_compile()`.

1.5.1 The "brainf" language

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

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. The point is that you can use `gcc_jit_context_compile_to_file()` to use libgccjit as a backend for a pre-existing language frontend (provided that the pre-existing frontend is compatible with libgccjit's license).

1.5.2 Converting a brainf script to libgccjit IR

As before we write simple code to populate a `gcc_jit_context*`.

```
typedef struct bf_compiler
{
  const char *filename;
  int line;
  int column;

  gcc_jit_context *ctxt;

  gcc_jit_type *void_type;
```

(continues on next page)

(continued from previous page)

```
  gcc_jit_type *int_type;
  gcc_jit_type *byte_type;
  gcc_jit_type *array_type;

  gcc_jit_function *func_getchar;
  gcc_jit_function *func_putchar;

  gcc_jit_function *func;
  gcc_jit_block *curblock;

  gcc_jit_rvalue *int_zero;
  gcc_jit_rvalue *int_one;
  gcc_jit_rvalue *byte_zero;
  gcc_jit_rvalue *byte_one;
  gcc_jit_lvalue *data_cells;
  gcc_jit_lvalue *idx;

  int num_open_parens;
  gcc_jit_block *paren_test[MAX_OPEN_PARENS];
  gcc_jit_block *paren_body[MAX_OPEN_PARENS];
  gcc_jit_block *paren_after[MAX_OPEN_PARENS];

} bf_compiler;

/* Bail out, with a message on stderr.  */

static void
fatal_error (bf_compiler *bfc, const char *msg)
{
  fprintf (stderr,
           "%s:%i:%i: %s",
           bfc->filename, bfc->line, bfc->column, msg);
  abort ();
}

/* Get "data_cells[idx]" as an lvalue.  */

static gcc_jit_lvalue *
bf_get_current_data (bf_compiler *bfc, gcc_jit_location *loc)
{
  return gcc_jit_context_new_array_access (
    bfc->ctxt,
    loc,
    gcc_jit_lvalue_as_rvalue (bfc->data_cells),
    gcc_jit_lvalue_as_rvalue (bfc->idx));
}

/* Get "data_cells[idx] == 0" as a boolean rvalue.  */

static gcc_jit_rvalue *
bf_current_data_is_zero (bf_compiler *bfc, gcc_jit_location *loc)
```

(continues on next page)

(continued from previous page)

```
{
  return gcc_jit_context_new_comparison (
    bfc->ctxt,
    loc,
    GCC_JIT_COMPARISON_EQ,
    gcc_jit_lvalue_as_rvalue (bf_get_current_data (bfc, loc)),
    bfc->byte_zero);
}

/* Compile one bf character.  */

static void
bf_compile_char (bf_compiler *bfc,
                 unsigned char ch)
{
  gcc_jit_location *loc =
    gcc_jit_context_new_location (bfc->ctxt,
                                  bfc->filename,
                                  bfc->line,
                                  bfc->column);

  /* Turn this on to trace execution, by injecting putchar ()
     of each source char. */
  if (0)
    {
      gcc_jit_rvalue *arg =
        gcc_jit_context_new_rvalue_from_int (
                                             bfc->ctxt,
                                             bfc->int_type,
                                             ch);
      gcc_jit_rvalue *call =
        gcc_jit_context_new_call (bfc->ctxt,
                                  loc,
                                  bfc->func_putchar,
                                  1, &arg);
      gcc_jit_block_add_eval (bfc->curblock,
                              loc,
                              call);
    }

  switch (ch)
    {
      case '>':
        gcc_jit_block_add_comment (bfc->curblock,
                                   loc,
                                   "'>': idx += 1;");
        gcc_jit_block_add_assignment_op (bfc->curblock,
                                         loc,
                                         bfc->idx,
                                         GCC_JIT_BINARY_OP_PLUS,
                                         bfc->int_one);
```

(continues on next page)

(continued from previous page)

```
      break;

    case '<':
      gcc_jit_block_add_comment (bfc->curblock,
                                 loc,
                                 "'<': idx -= 1;");
      gcc_jit_block_add_assignment_op (bfc->curblock,
                                       loc,
                                       bfc->idx,
                                       GCC_JIT_BINARY_OP_MINUS,
                                       bfc->int_one);
      break;

    case '+':
      gcc_jit_block_add_comment (bfc->curblock,
                                 loc,
                                 "'+': data[idx] += 1;");
      gcc_jit_block_add_assignment_op (bfc->curblock,
                                       loc,
                                       bf_get_current_data (bfc, loc),
                                       GCC_JIT_BINARY_OP_PLUS,
                                       bfc->byte_one);
      break;

    case '-':
      gcc_jit_block_add_comment (bfc->curblock,
                                 loc,
                                 "'-': data[idx] -= 1;");
      gcc_jit_block_add_assignment_op (bfc->curblock,
                                       loc,
                                       bf_get_current_data (bfc, loc),
                                       GCC_JIT_BINARY_OP_MINUS,
                                       bfc->byte_one);
      break;

    case '.':
      {
        gcc_jit_rvalue *arg =
          gcc_jit_context_new_cast (
            bfc->ctxt,
            loc,
            gcc_jit_lvalue_as_rvalue (bf_get_current_data (bfc, loc)),
            bfc->int_type);
        gcc_jit_rvalue *call =
          gcc_jit_context_new_call (bfc->ctxt,
                                    loc,
                                    bfc->func_putchar,
                                    1, &arg);
        gcc_jit_block_add_comment (bfc->curblock,
                                   loc,
                                   "'.': putchar ((int)data[idx]);");
```

(continues on next page)

(continued from previous page)

```
      gcc_jit_block_add_eval (bfc->curblock,
                              loc,
                              call);
    }
    break;

  case ',':
    {
      gcc_jit_rvalue *call =
        gcc_jit_context_new_call (bfc->ctxt,
                                  loc,
                                  bfc->func_getchar,
                                  0, NULL);
      gcc_jit_block_add_comment (
        bfc->curblock,
        loc,
        "',': data[idx] = (unsigned char)getchar ();");
      gcc_jit_block_add_assignment (bfc->curblock,
                                    loc,
                                    bf_get_current_data (bfc, loc),
                                    gcc_jit_context_new_cast (
                                      bfc->ctxt,
                                      loc,
                                      call,
                                      bfc->byte_type));
    }
    break;

  case '[':
    {
      gcc_jit_block *loop_test =
        gcc_jit_function_new_block (bfc->func, NULL);
      gcc_jit_block *on_zero =
        gcc_jit_function_new_block (bfc->func, NULL);
      gcc_jit_block *on_non_zero =
        gcc_jit_function_new_block (bfc->func, NULL);

      if (bfc->num_open_parens == MAX_OPEN_PARENS)
        fatal_error (bfc, "too many open parens");

      gcc_jit_block_end_with_jump (
        bfc->curblock,
        loc,
        loop_test);

      gcc_jit_block_add_comment (
        loop_test,
        loc,
        "'['");
      gcc_jit_block_end_with_conditional (
        loop_test,
```

(continues on next page)

(continued from previous page)

```
            loc,
            bf_current_data_is_zero (bfc, loc),
            on_zero,
            on_non_zero);
          bfc->paren_test[bfc->num_open_parens] = loop_test;
          bfc->paren_body[bfc->num_open_parens] = on_non_zero;
          bfc->paren_after[bfc->num_open_parens] = on_zero;
          bfc->num_open_parens += 1;
          bfc->curblock = on_non_zero;
        }
        break;

      case ']':
        {
          gcc_jit_block_add_comment (
            bfc->curblock,
            loc,
            "']'");

          if (bfc->num_open_parens == 0)
            fatal_error (bfc, "mismatching parens");
          bfc->num_open_parens -= 1;
          gcc_jit_block_end_with_jump (
            bfc->curblock,
            loc,
            bfc->paren_test[bfc->num_open_parens]);
          bfc->curblock = bfc->paren_after[bfc->num_open_parens];
        }
        break;

    case '\n':
      bfc->line +=1;
      bfc->column = 0;
      break;
    }

  if (ch != '\n')
    bfc->column += 1;
}

/* Compile the given .bf file into a gcc_jit_context, containing a
   single "main" function suitable for compiling into an executable.  */

gcc_jit_context *
bf_compile (const char *filename)
{
  bf_compiler bfc;
  FILE *f_in;
  int ch;

  memset (&bfc, 0, sizeof (bfc));
```

(continues on next page)

(continued from previous page)

```
bfc.filename = filename;
f_in = fopen (filename, "r");
if (!f_in)
  fatal_error (&bfc, "unable to open file");
bfc.line = 1;

bfc.ctxt = gcc_jit_context_acquire ();

gcc_jit_context_set_int_option (
  bfc.ctxt,
  GCC_JIT_INT_OPTION_OPTIMIZATION_LEVEL,
  3);
gcc_jit_context_set_bool_option (
  bfc.ctxt,
  GCC_JIT_BOOL_OPTION_DUMP_INITIAL_GIMPLE,
  0);
gcc_jit_context_set_bool_option (
  bfc.ctxt,
  GCC_JIT_BOOL_OPTION_DEBUGINFO,
  1);
gcc_jit_context_set_bool_option (
  bfc.ctxt,
  GCC_JIT_BOOL_OPTION_DUMP_EVERYTHING,
  0);
gcc_jit_context_set_bool_option (
  bfc.ctxt,
  GCC_JIT_BOOL_OPTION_KEEP_INTERMEDIATES,
  0);

bfc.void_type =
  gcc_jit_context_get_type (bfc.ctxt, GCC_JIT_TYPE_VOID);
bfc.int_type =
  gcc_jit_context_get_type (bfc.ctxt, GCC_JIT_TYPE_INT);
bfc.byte_type =
  gcc_jit_context_get_type (bfc.ctxt, GCC_JIT_TYPE_UNSIGNED_CHAR);
bfc.array_type =
  gcc_jit_context_new_array_type (bfc.ctxt,
                                  NULL,
                                  bfc.byte_type,
                                  30000);

bfc.func_getchar =
  gcc_jit_context_new_function (bfc.ctxt, NULL,
                                GCC_JIT_FUNCTION_IMPORTED,
                                bfc.int_type,
                                "getchar",
                                0, NULL,
                                0);

gcc_jit_param *param_c =
```

(continues on next page)

(continued from previous page)

```
    gcc_jit_context_new_param (bfc.ctxt, NULL, bfc.int_type, "c");
  bfc.func_putchar =
    gcc_jit_context_new_function (bfc.ctxt, NULL,
                                  GCC_JIT_FUNCTION_IMPORTED,
                                  bfc.void_type,
                                  "putchar",
                                  1, &param_c,
                                  0);

  bfc.func = make_main (bfc.ctxt);
   bfc.curblock =
    gcc_jit_function_new_block (bfc.func, "initial");
  bfc.int_zero = gcc_jit_context_zero (bfc.ctxt, bfc.int_type);
  bfc.int_one = gcc_jit_context_one (bfc.ctxt, bfc.int_type);
  bfc.byte_zero = gcc_jit_context_zero (bfc.ctxt, bfc.byte_type);
  bfc.byte_one = gcc_jit_context_one (bfc.ctxt, bfc.byte_type);

  bfc.data_cells =
    gcc_jit_context_new_global (bfc.ctxt, NULL,
                                GCC_JIT_GLOBAL_INTERNAL,
                                bfc.array_type,
                                "data_cells");
  bfc.idx =
    gcc_jit_function_new_local (bfc.func, NULL,
                                bfc.int_type,
                                "idx");

  gcc_jit_block_add_comment (bfc.curblock,
                             NULL,
                             "idx = 0;");
  gcc_jit_block_add_assignment (bfc.curblock,
                                NULL,
                                bfc.idx,
                                bfc.int_zero);

  bfc.num_open_parens = 0;

  while ( EOF != (ch = fgetc (f_in)))
    bf_compile_char (&bfc, (unsigned char)ch);

  gcc_jit_block_end_with_return (bfc.curblock, NULL, bfc.int_zero);

  fclose (f_in);

  return bfc.ctxt;
}
```

1.5.3 Compiling a context to a file

Unlike the previous tutorial, this time we'll compile the context directly to an executable, using `gcc_jit_context_compile_to_file()`:

```
gcc_jit_context_compile_to_file (ctxt,
                                 GCC_JIT_OUTPUT_KIND_EXECUTABLE,
                                 output_file);
```

Here's the top-level of the compiler, which is what actually calls into `gcc_jit_context_compile_to_file()`:

```
int
main (int argc, char **argv)
{
  const char *input_file;
  const char *output_file;
  gcc_jit_context *ctxt;
  const char *err;

  if (argc != 3)
    {
      fprintf (stderr, "%s: INPUT_FILE OUTPUT_FILE\n", argv[0]);
      return 1;
    }

  input_file = argv[1];
  output_file = argv[2];
  ctxt = bf_compile (input_file);

  gcc_jit_context_compile_to_file (ctxt,
                                   GCC_JIT_OUTPUT_KIND_EXECUTABLE,
                                   output_file);

  err = gcc_jit_context_get_first_error (ctxt);

  if (err)
    {
      gcc_jit_context_release (ctxt);
      return 1;
    }

  gcc_jit_context_release (ctxt);
  return 0;
}
```

Note how once the context is populated you could trivially instead compile it to memory using `gcc_jit_context_compile()` and run it in-process as in the previous tutorial.

To create an executable, we need to export a `main` function. Here's how to create one from the JIT API:

```
/* Make "main" function:
     int
     main (int argc, char **argv)
     {
       ...
     }
*/
static gcc_jit_function *
make_main (gcc_jit_context *ctxt)
{
  gcc_jit_type *int_type =
    gcc_jit_context_get_type (ctxt, GCC_JIT_TYPE_INT);
  gcc_jit_param *param_argc =
    gcc_jit_context_new_param (ctxt, NULL, int_type, "argc");
  gcc_jit_type *char_ptr_ptr_type =
    gcc_jit_type_get_pointer (
      gcc_jit_type_get_pointer (
        gcc_jit_context_get_type (ctxt, GCC_JIT_TYPE_CHAR)));
  gcc_jit_param *param_argv =
    gcc_jit_context_new_param (ctxt, NULL, char_ptr_ptr_type, "argv");
  gcc_jit_param *params[2] = {param_argc, param_argv};
  gcc_jit_function *func_main =
    gcc_jit_context_new_function (ctxt, NULL,
                                  GCC_JIT_FUNCTION_EXPORTED,
                                  int_type,
                                  "main",
                                  2, params,
                                  0);
  return func_main;
}
```

Note: The above implementation ignores `argc` and `argv`, but you could make use of them by exposing `param_argc` and `param_argv` to the caller.

Upon compiling this C code, we obtain a bf-to-machine-code compiler; let's call it `bfc`:

```
$ gcc \
    tut05-bf.c \
    -o bfc \
    -lgccjit
```

We can now use `bfc` to compile .bf files into machine code executables:

```
$ ./bfc \
     emit-alphabet.bf \
     a.out
```

which we can run directly:

```
$ ./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 <main>:
  400620:     80 3d 39 0a 20 00 00    cmpb   $0x0,0x200a39(%rip)        # 601060 <data
  400627:     74 07                   je     400630 <main
  400629:     eb fe                   jmp    400629 <main+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 <main+0x30>
  400673:     31 c0                   xor    %eax,%eax
  400675:     48 83 c4 08             add    $0x8,%rsp
  400679:     c3                      retq
  40067a:     66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
```

We also set up debugging information (via `gcc_jit_context_new_location()` and `GCC_JIT_BOOL_OPTION_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
```

(continues on next page)

(continued from previous page)

```
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     >
```

1.5.4 Other forms of ahead-of-time-compilation

The above demonstrates compiling a `gcc_jit_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 `gcc_jit_context_compile_to_file()` for more information.

CHAPTER

TWO

TOPIC REFERENCE

2.1 Compilation contexts

type **gcc_jit_context**

The top-level of the API is the gcc_jit_context type.

A gcc_jit_context instance encapsulates the state of a compilation.

You can set up options on it, and add types, functions and code. Invoking gcc_jit_context_compile() on it gives you a gcc_jit_result.

2.1.1 Lifetime-management

Contexts are the unit of lifetime-management within the API: objects have their lifetime bounded by the context they are created within, and cleanup of such objects is done for you when the context is released.

gcc_jit_context *__**gcc_jit_context_acquire**__(void)

This function acquires a new gcc_jit_context* instance, which is independent of any others that may be present within this process.

void __**gcc_jit_context_release**__(gcc_jit_context *ctxt)

This function releases all resources associated with the given context. Both the context itself and all of its gcc_jit_object* instances are cleaned up. It should be called exactly once on a given context.

It is invalid to use the context or any of its "contextual" objects after calling this.

```
gcc_jit_context_release (ctxt);
```

gcc_jit_context *__**gcc_jit_context_new_child_context**__(gcc_jit_context *parent_ctxt)

Given an existing JIT context, create a child context.

The child inherits a copy of all option-settings from the parent.

The child can reference objects created within the parent, but not vice-versa.

The lifetime of the child context must be bounded by that of the parent: you should release a child context before releasing the parent context.

If you use a function from a parent context within a child context, you have to compile the parent context before you can compile the child context, and the gcc_jit_result of the parent context must outlive the gcc_jit_result of the child context.

This allows caching of shared initializations. For example, you could create types and declarations of global functions in a parent context once within a process, and then create child contexts whenever a function or loop becomes hot. Each such child context can be used for JIT-compiling just one function or loop, but can reference types and helper functions created within the parent context.

Contexts can be arbitrarily nested, provided the above rules are followed, but it's probably not worth going above 2 or 3 levels, and there will likely be a performance hit for such nesting.

2.1.2 Thread-safety

Instances of `gcc_jit_context*` created via `gcc_jit_context_acquire()` are independent from each other: only one thread may use a given context at once, but multiple threads could each have their own contexts without needing locks.

Contexts created via `gcc_jit_context_new_child_context()` are related to their parent context. They can be partitioned by their ultimate ancestor into independent "family trees". Only one thread within a process may use a given "family tree" of such contexts at once, and if you're using multiple threads you should provide your own locking around entire such context partitions.

2.1.3 Error-handling

Various kinds of errors are possible when using the API, such as mismatched types in an assignment. You can only compile and get code from a context if no errors occur.

Errors are printed on stderr and can be queried using `gcc_jit_context_get_first_error()`.

They typically contain the name of the API entrypoint where the error occurred, and pertinent information on the problem:

```
./buggy-program: error: gcc_jit_block_add_assignment: mismatching types: assignment to i (type:␣
↪int) from "hello world" (type: const char *)
```

In general, if an error occurs when using an API entrypoint, the entrypoint returns NULL. You don't have to check everywhere for NULL results, since the API handles a NULL being passed in for any argument by issuing another error. This typically leads to a cascade of followup error messages, but is safe (albeit verbose). The first error message is usually the one to pay attention to, since it is likely to be responsible for all of the rest:

const char *__gcc_jit_context_get_first_error__(gcc_jit_context *ctxt)

Returns the first error message that occurred on the context.

The returned string is valid for the rest of the lifetime of the context.

If no errors occurred, this will be NULL.

If you are wrapping the C API for a higher-level language that supports exception-handling, you may instead be interested in the last error that occurred on the context, so that you can embed this in an exception:

const char ***gcc_jit_context_get_last_error**(gcc_jit_context *ctxt)

Returns the last error message that occurred on the context.

If no errors occurred, this will be NULL.

If non-NULL, the returned string is only guaranteed to be valid until the next call to libgccjit relating to this context.

2.1.4 Debugging

void **gcc_jit_context_dump_to_file**(gcc_jit_context *ctxt, const char *path, int update_locations)

To help with debugging: dump a C-like representation to the given path, describing what's been set up on the context.

If "update_locations" is true, then also set up `gcc_jit_location` information throughout the context, pointing at the dump file as if it were a source file. This may be of use in conjunction with `GCC_JIT_BOOL_OPTION_DEBUGINFO` to allow stepping through the code in a debugger.

void **gcc_jit_context_set_logfile**(gcc_jit_context *ctxt, FILE *logfile, int flags, int verbosity)

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

For example, the following will enable logging to stderr.

```
gcc_jit_context_set_logfile (ctxt, stderr, 0, 0);
```

Examples of information logged include:

- API calls
- the various steps involved within compilation
- activity on any `gcc_jit_result` instances created by the context
- activity within any child contexts

An example of a log can be seen here, though the precise format and kinds of information logged is subject to change.

The caller remains responsible for closing *logfile*, and it must not be closed until all users are released. In particular, note that child contexts and `gcc_jit_result` instances created by the context will use the logfile.

There may a performance cost for logging.

You can turn off logging on *ctxt* by passing *NULL* for *logfile*. Doing so only affects the context; it does not affect child contexts or `gcc_jit_result` instances already created by the context.

The parameters "flags" and "verbosity" are reserved for future expansion, and must be zero for now.

To contrast the above: `gcc_jit_context_dump_to_file()` dumps the current state of a context to the given path, whereas `gcc_jit_context_set_logfile()` enables on-going logging of future activies on a context to the given *FILE **.

void **gcc_jit_context_dump_reproducer_to_file**(gcc_jit_context *ctxt, const char *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.

This may be useful when debugging the library or client code, for reducing a complicated recipe for reproducing a bug into a simpler form. For example, consider client code that parses some source file into some internal representation, and then walks this IR, calling into libgccjit. If this encounters a bug, a call to *gcc_jit_context_dump_reproducer_to_file* will write out C code for a much simpler executable that performs the equivalent calls into libgccjit, without needing the client code and its data.

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).

void **gcc_jit_context_enable_dump**(gcc_jit_context *ctxt, const char *dumpname, char **out_ptr)

Enable the dumping of a specific set of internal state from the compilation, capturing the result in-memory as a buffer.

Parameter "dumpname" corresponds to the equivalent gcc command-line option, without the "-fdump-" prefix. For example, to get the equivalent of `-fdump-tree-vrp`, supply `"tree-vrp1"`:

```
static char *dump_vrp1;

void
create_code (gcc_jit_context *ctxt)
{
   gcc_jit_context_enable_dump (ctxt, "tree-vrp1", &dump_vrp1);
   /* (other API calls omitted for brevity) */
}
```

The context directly stores the dumpname as a `(const char *)`, so the passed string must outlive the context.

`gcc_jit_context_compile()` will capture the dump as a dynamically-allocated buffer, writing it to `*out_ptr`.

The caller becomes responsible for calling:

```
free (*out_ptr)
```

each time that `gcc_jit_context_compile()` is called. `*out_ptr` will be written to, either with the address of a buffer, or with `NULL` if an error occurred.

> **Warning:** This API entrypoint is likely to be less stable than the others. In particular, both the precise dumpnames, and the format and content of the dumps are subject to change.
>
> It exists primarily for writing the library's own test suite.

2.1.5 Options

Options present in the initial release of libgccjit were handled using enums, whereas those added subsequently have their own per-option API entrypoints.

Adding entrypoints for each new option means that client code that use the new options can be identified directly from binary metadata, which would not be possible if we instead extended the various `enum gcc_jit_*_option`.

String Options

void **gcc_jit_context_set_str_option**(gcc_jit_context *ctxt, enum gcc_jit_str_option opt, const char *value)

Set a string option of the context.

enum **gcc_jit_str_option**

The parameter `value` can be NULL. If non-NULL, the call takes a copy of the underlying string, so it is valid to pass in a pointer to an on-stack buffer.

There is just one string option specified this way:

GCC_JIT_STR_OPTION_PROGNAME

The name of the program, for use as a prefix when printing error messages to stderr. If *NULL*, or default, "libgccjit.so" is used.

Boolean options

void **gcc_jit_context_set_bool_option**(gcc_jit_context *ctxt, enum gcc_jit_bool_option opt, int value)

Set a boolean option of the context. Zero is "false" (the default), non-zero is "true".

enum **gcc_jit_bool_option**

GCC_JIT_BOOL_OPTION_DEBUGINFO

If true, `gcc_jit_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 `gcc_jit_location` instances).

GCC_JIT_BOOL_OPTION_DUMP_INITIAL_TREE

If true, gcc_jit_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 <integer_cst 0x7f4875a62340 -2147483648> max <integer_cst 0x7f4875a62360␣
↪2147483647>
           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>>>
↪>>
```

GCC_JIT_BOOL_OPTION_DUMP_INITIAL_GIMPLE

If true, gcc_jit_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;
}
```

GCC_JIT_BOOL_OPTION_DUMP_GENERATED_CODE

If true, gcc_jit_context_compile() will dump the final generated code to stderr, in the form of assembly language:

```
    .file   "fake.c"
    .text
```

(continues on next page)

(continued from previous page)

```
    .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 0.2)"
    .section    .note.GNU-stack,"",@progbits
```

GCC_JIT_BOOL_OPTION_DUMP_SUMMARY

If true, `gcc_jit_context_compile()` will print information to stderr on the actions it is performing.

GCC_JIT_BOOL_OPTION_DUMP_EVERYTHING

If true, `gcc_jit_context_compile()` will dump copious amount of information on what it's doing to various files within a temporary directory. Use `GCC_JIT_BOOL_OPTION_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.

GCC_JIT_BOOL_OPTION_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.

GCC_JIT_BOOL_OPTION_KEEP_INTERMEDIATES

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

void **gcc_jit_context_set_bool_allow_unreachable_blocks**(gcc_jit_context *ctxt, int bool_value)

By default, libgccjit will issue an error about unreachable blocks within a function.

This entrypoint can be used to disable that error.

This entrypoint was added in LIBGCCJIT_ABI_2; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_gcc_jit_context_set_bool_allow_unreachable_blocks
```

void **gcc_jit_context_set_bool_use_external_driver**(gcc_jit_context *ctxt, int bool_value)

libgccjit internally generates assembler, and uses "driver" code for converting it to other formats (e.g. shared libraries).

By default, libgccjit will use an embedded copy of the driver code.

This option can be used to instead invoke an external driver executable as a subprocess.

This entrypoint was added in LIBGCCJIT_ABI_5; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_gcc_jit_context_set_bool_use_external_driver
```

void **gcc_jit_context_set_bool_print_errors_to_stderr**(gcc_jit_context *ctxt, int enabled)

By default, libgccjit will print errors to stderr.

This entrypoint can be used to disable the printing.

This entrypoint was added in LIBGCCJIT_ABI_23; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_gcc_jit_context_set_bool_print_errors_to_stderr
```

Integer options

void **gcc_jit_context_set_int_option**(gcc_jit_context *ctxt, enum gcc_jit_int_option opt, int value)

Set an integer option of the context.

enum **gcc_jit_int_option**

There is just one integer option specified this way:

GCC_JIT_INT_OPTION_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).

Additional command-line options

void **gcc_jit_context_add_command_line_option**(gcc_jit_context *ctxt, const char *optname)

Add an arbitrary gcc command-line option to the context, for use by `gcc_jit_context_compile()` and `gcc_jit_context_compile_to_file()`.

The parameter `optname` must be non-NULL. The underlying buffer is copied, so that it does not need to outlive the call.

Extra options added by *gcc_jit_context_add_command_line_option* are applied *after* the regular options above, potentially overriding them. Options from parent contexts are inherited by child contexts; options from the parent are applied *before* those from the child.

For example:

```
gcc_jit_context_add_command_line_option (ctxt, "-ffast-math");
gcc_jit_context_add_command_line_option (ctxt, "-fverbose-asm");
```

Note that only some options are likely to be meaningful; there is no "frontend" within libgccjit, so typically only those affecting optimization and code-generation are likely to be useful.

This entrypoint was added in LIBGCCJIT_ABI_1; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_gcc_jit_context_add_command_line_option
```

void **gcc_jit_context_add_driver_option**(gcc_jit_context *ctxt, const char *optname)

Add an arbitrary gcc driver option to the context, for use by `gcc_jit_context_compile()` and `gcc_jit_context_compile_to_file()`.

The parameter `optname` must be non-NULL. The underlying buffer is copied, so that it does not need to outlive the call.

Extra options added by *gcc_jit_context_add_driver_option* are applied *after* all other options potentially overriding them. Options from parent contexts are inherited by child contexts; options from the parent are applied *before* those from the child.

For example:

```
gcc_jit_context_add_driver_option (ctxt, "-lm");
gcc_jit_context_add_driver_option (ctxt, "-fuse-linker-plugin");

gcc_jit_context_add_driver_option (ctxt, "obj.o");

gcc_jit_context_add_driver_option (ctxt, "-L.");
gcc_jit_context_add_driver_option (ctxt, "-lwhatever");
```

Note that only some options are likely to be meaningful; there is no "frontend" within libgccjit, so typically only those affecting assembler and linker are likely to be useful.

This entrypoint was added in LIBGCCJIT_ABI_11; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_gcc_jit_context_add_driver_option
```

2.2 Objects

type **gcc_jit_object**

Almost every entity in the API (with the exception of gcc_jit_context* and gcc_jit_result*) is a "contextual" object, a gcc_jit_object*

A JIT object:

- is associated with a gcc_jit_context*.
- is automatically cleaned up for you when its context is released so you don't need to manually track and cleanup all objects, just the contexts.

Although the API is C-based, there is a form of class hierarchy, which looks like this:

```
+- gcc_jit_object
    +- gcc_jit_location
    +- gcc_jit_type
       +- gcc_jit_struct
    +- gcc_jit_field
    +- gcc_jit_function
    +- gcc_jit_block
    +- gcc_jit_rvalue
        +- gcc_jit_lvalue
           +- gcc_jit_param
    +- gcc_jit_case
    +- gcc_jit_extended_asm
```

There are casting methods for upcasting from subclasses to parent classes. For example, gcc_jit_type_as_object():

```
gcc_jit_object *obj = gcc_jit_type_as_object (int_type);
```

The object "base class" has the following operations:

gcc_jit_context ***gcc_jit_object_get_context**(gcc_jit_object *obj)

Which context is "obj" within?

const char ***gcc_jit_object_get_debug_string**(gcc_jit_object *obj)

Generate a human-readable description for the given object.

For example,

```
printf ("obj: %s\n", gcc_jit_object_get_debug_string (obj));
```

might give this text on stdout:

```
obj: 4.0 * (float)i
```

Note: If you call this on an object, the *const char ** buffer is allocated and generated on the first call for that object, and the buffer will have the same lifetime as the object i.e. it

will exist until the object's context is released.

2.3 Types

type **gcc_jit_type**

gcc_jit_type represents a type within the library.

gcc_jit_object *__gcc_jit_type_as_object__(gcc_jit_type *type)

Upcast a type to an object.

Types can be created in several ways:

- fundamental types can be accessed using `gcc_jit_context_get_type()`:

```
gcc_jit_type *int_type = gcc_jit_context_get_type (ctxt, GCC_JIT_TYPE_INT);
```

  See `gcc_jit_context_get_type()` for the available types.

- derived types can be accessed by using functions such as `gcc_jit_type_get_pointer()` and `gcc_jit_type_get_const()`:

```
gcc_jit_type *const_int_star = gcc_jit_type_get_pointer (gcc_jit_type_get_const (int_
↪type));
gcc_jit_type *int_const_star = gcc_jit_type_get_const (gcc_jit_type_get_pointer (int_
↪type));
```

- by creating structures (see below).

2.3.1 Standard types

gcc_jit_type *__gcc_jit_context_get_type__(gcc_jit_context *ctxt, enum gcc_jit_types type_)

Access a specific type. The available types are:

enum gcc_jit_types value	Meaning
GCC_JIT_TYPE_VOID	C's `void` type.
GCC_JIT_TYPE_VOID_PTR	C's `void *`.
GCC_JIT_TYPE_BOOL	C++'s `bool` type; also C99's `_Bool` type, aka `bool` if using stdbool.h.
GCC_JIT_TYPE_CHAR	C's `char` (of some signedness)
GCC_JIT_TYPE_SIGNED_CHAR	C's `signed char`
GCC_JIT_TYPE_UNSIGNED_CHAR	C's `unsigned char`
GCC_JIT_TYPE_SHORT	C's `short` (signed)
GCC_JIT_TYPE_UNSIGNED_SHORT	C's `unsigned short`
GCC_JIT_TYPE_INT	C's `int` (signed)
GCC_JIT_TYPE_UNSIGNED_INT	C's `unsigned int`
GCC_JIT_TYPE_LONG	C's `long` (signed)

continues on next page

Table 1 – continued from previous page

enum gcc_jit_types value	Meaning
GCC_JIT_TYPE_UNSIGNED_LONG	C's `unsigned long`
GCC_JIT_TYPE_LONG_LONG	C99's `long long` (signed)
GCC_JIT_TYPE_UNSIGNED_LONG_LONG	C99's `unsigned long long`
GCC_JIT_TYPE_UINT8_T	C99's `uint8_t`
GCC_JIT_TYPE_UINT16_T	C99's `uint16_t`
GCC_JIT_TYPE_UINT32_T	C99's `uint32_t`
GCC_JIT_TYPE_UINT64_T	C99's `uint64_t`
GCC_JIT_TYPE_UINT128_T	C99's `__uint128_t`
GCC_JIT_TYPE_INT8_T	C99's `int8_t`
GCC_JIT_TYPE_INT16_T	C99's `int16_t`
GCC_JIT_TYPE_INT32_T	C99's `int32_t`
GCC_JIT_TYPE_INT64_T	C99's `int64_t`
GCC_JIT_TYPE_INT128_T	C99's `__int128_t`
GCC_JIT_TYPE_FLOAT	
GCC_JIT_TYPE_DOUBLE	
GCC_JIT_TYPE_LONG_DOUBLE	
GCC_JIT_TYPE_CONST_CHAR_PTR	C type: `(const char *)`
GCC_JIT_TYPE_SIZE_T	C's `size_t` type
GCC_JIT_TYPE_FILE_PTR	C type: `(FILE *)`
GCC_JIT_TYPE_COMPLEX_FLOAT	C99's `_Complex float`
GCC_JIT_TYPE_COMPLEX_DOUBLE	C99's `_Complex double`
GCC_JIT_TYPE_COMPLEX_LONG_DOUBLE	C99's `_Complex long double`

gcc_jit_type *__gcc_jit_context_get_int_type__(gcc_jit_context *ctxt, int num_bytes, int is_signed)

Access the integer type of the given size.

2.3.2 Pointers, *const*, and *volatile*

gcc_jit_type *__gcc_jit_type_get_pointer__(gcc_jit_type *type)

Given type "T", get type "T*".

gcc_jit_type *__gcc_jit_type_get_const__(gcc_jit_type *type)

Given type "T", get type "const T".

gcc_jit_type *__gcc_jit_type_get_volatile__(gcc_jit_type *type)

Given type "T", get type "volatile T".

gcc_jit_type *__gcc_jit_context_new_array_type__(gcc_jit_context *ctxt, gcc_jit_location *loc, gcc_jit_type *element_type, int num_elements)

Given non-*void* type "T", get type "T[N]" (for a constant N).

gcc_jit_type *__gcc_jit_type_get_aligned__(gcc_jit_type *type, size_t alignment_in_bytes)

Given non-*void* type "T", get type:

```
T __attribute__ ((aligned (ALIGNMENT_IN_BYTES)))
```

The alignment must be a power of two.

This entrypoint was added in LIBGCCJIT_ABI_7; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_gcc_jit_type_get_aligned
```

2.3.3 Vector types

gcc_jit_type ***gcc_jit_type_get_vector**(gcc_jit_type *type, size_t num_units)

Given type "T", get type:

```
T  __attribute__ ((vector_size (sizeof(T) * num_units))
```

T must be integral or floating point; num_units must be a power of two.

This can be used to construct a vector type in which operations are applied element-wise. The compiler will automatically use SIMD instructions where possible. See: https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html

For example, assuming 4-byte `ints`, then:

```
typedef int v4si __attribute__ ((vector_size (16)));
```

can be obtained using:

```
gcc_jit_type *int_type = gcc_jit_context_get_type (ctxt,
                                                   GCC_JIT_TYPE_INT);
gcc_jit_type *v4si_type = gcc_jit_type_get_vector (int_type, 4);
```

This API entrypoint was added in LIBGCCJIT_ABI_8; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_gcc_jit_type_get_vector
```

Vector rvalues can be generated using `gcc_jit_context_new_rvalue_from_vector()`.

2.3.4 Structures and unions

type **gcc_jit_struct**

A compound type analagous to a C *struct*.

type **gcc_jit_field**

A field within a `gcc_jit_struct`.

You can model C *struct* types by creating `gcc_jit_struct` and `gcc_jit_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:

```
gcc_jit_field *field_x =
  gcc_jit_context_new_field (ctxt, NULL, double_type, "x");
gcc_jit_field *field_y =
  gcc_jit_context_new_field (ctxt, NULL, double_type, "y");
gcc_jit_field *fields[2] = {field_x, field_y};
gcc_jit_struct *coord =
  gcc_jit_context_new_struct_type (ctxt, NULL, "coord", 2, fields);
```

- 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:

```
gcc_jit_type *node =
  gcc_jit_context_new_opaque_struct (ctxt, NULL, "node");
gcc_jit_type *node_ptr =
  gcc_jit_type_get_pointer (node);
gcc_jit_field *field_hash =
  gcc_jit_context_new_field (ctxt, NULL, int_type, "m_hash");
gcc_jit_field *field_next =
  gcc_jit_context_new_field (ctxt, NULL, node_ptr, "m_next");
gcc_jit_field *fields[2] = {field_hash, field_next};
gcc_jit_struct_set_fields (node, NULL, 2, fields);
```

gcc_jit_field ***gcc_jit_context_new_field**(gcc_jit_context *ctxt, gcc_jit_location *loc, gcc_jit_type *type, const char *name)

Construct a new field, with the given type and name.

The parameter `type` must be non-*void*.

The parameter `name` must be non-NULL. The call takes a copy of the underlying string, so it is valid to pass in a pointer to an on-stack buffer.

gcc_jit_field ***gcc_jit_context_new_bitfield**(gcc_jit_context *ctxt, gcc_jit_location *loc, gcc_jit_type *type, int width, const char *name)

Construct a new bit field, with the given type width and name.

The parameter `name` must be non-NULL. The call takes a copy of the underlying string, so it is valid to pass in a pointer to an on-stack buffer.

The parameter `type` must be an integer type.

The parameter `width` must be a positive integer that does not exceed the size of `type`.

This API entrypoint was added in LIBGCCJIT_ABI_12; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_gcc_jit_context_new_bitfield
```

gcc_jit_object *__gcc_jit_field_as_object__(gcc_jit_field *field)

Upcast from field to object.

gcc_jit_struct *__gcc_jit_context_new_struct_type__(gcc_jit_context *ctxt, gcc_jit_location *loc, const char *name, int num_fields, gcc_jit_field **fields)

Construct a new struct type, with the given name and fields.

The parameter `name` must be non-NULL. The call takes a copy of the underlying string, so it is valid to pass in a pointer to an on-stack buffer.

gcc_jit_struct *__gcc_jit_context_new_opaque_struct__(gcc_jit_context *ctxt, gcc_jit_location *loc, const char *name)

Construct a new struct type, with the given name, but without specifying the fields. The fields can be omitted (in which case the size of the struct is not known), or later specified using `gcc_jit_struct_set_fields()`.

The parameter `name` must be non-NULL. The call takes a copy of the underlying string, so it is valid to pass in a pointer to an on-stack buffer.

gcc_jit_type *__gcc_jit_struct_as_type__(gcc_jit_struct *struct_type)

Upcast from struct to type.

void __gcc_jit_struct_set_fields__(gcc_jit_struct *struct_type, gcc_jit_location *loc, int num_fields, gcc_jit_field **fields)

Populate the fields of a formerly-opaque struct type.

This can only be called once on a given struct type.

gcc_jit_type *__gcc_jit_context_new_union_type__(gcc_jit_context *ctxt, gcc_jit_location *loc, const char *name, int num_fields, gcc_jit_field **fields)

Construct a new union type, with the given name and fields.

The parameter `name` must be non-NULL. It is copied, so the input buffer does not need to outlive the call.

Example of use:

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

void
create_code (gcc_jit_context *ctxt, void *user_data)
{
```

(continues on next page)

(continued from previous page)

```
/* Let's try to inject the equivalent of:
   float
   test_union (int i)
   {
      union int_or_float u;
      u.as_int = i;
      return u.as_float;
   }
*/
gcc_jit_type *int_type =
  gcc_jit_context_get_type (ctxt, GCC_JIT_TYPE_INT);
gcc_jit_type *float_type =
  gcc_jit_context_get_type (ctxt, GCC_JIT_TYPE_FLOAT);
gcc_jit_field *as_int =
  gcc_jit_context_new_field (ctxt,
                             NULL,
                             int_type,
                             "as_int");
gcc_jit_field *as_float =
  gcc_jit_context_new_field (ctxt,
                             NULL,
                             float_type,
                             "as_float");
gcc_jit_field *fields[] = {as_int, as_float};
gcc_jit_type *union_type =
  gcc_jit_context_new_union_type (ctxt, NULL,
                                  "int_or_float", 2, fields);

/* Build the test function.  */
gcc_jit_param *param_i =
  gcc_jit_context_new_param (ctxt, NULL, int_type, "i");
gcc_jit_function *test_fn =
  gcc_jit_context_new_function (ctxt, NULL,
                                GCC_JIT_FUNCTION_EXPORTED,
                                float_type,
                                "test_union",
                                1, &param_i,
                                0);

gcc_jit_lvalue *u =
  gcc_jit_function_new_local (test_fn, NULL,
                              union_type, "u");

gcc_jit_block *block = gcc_jit_function_new_block (test_fn, NULL);

/* u.as_int = i; */
gcc_jit_block_add_assignment (
  block,
  NULL,
  /* "u.as_int = ..." */
  gcc_jit_lvalue_access_field (u,
```

(continues on next page)

(continued from previous page)

```
                                NULL,
                                as_int),
    gcc_jit_param_as_rvalue (param_i));

  /* return u.as_float; */
  gcc_jit_block_end_with_return (
    block, NULL,
    gcc_jit_rvalue_access_field (gcc_jit_lvalue_as_rvalue (u),
                                NULL,
                                as_float));
}
```

2.3.5 Function pointer types

Function pointer types can be created using `gcc_jit_context_new_function_ptr_type()`.

2.3.6 Reflection API

gcc_jit_type *__gcc_jit_type_dyncast_array__(gcc_jit_type *type)

Get the element type of an array type or NULL if it's not an array.

int __gcc_jit_type_is_bool__(gcc_jit_type *type)

Return non-zero if the type is a bool.

gcc_jit_function_type *__gcc_jit_type_dyncast_function_ptr_type__(gcc_jit_type *type)

Return the function type if it is one or NULL.

gcc_jit_type *__gcc_jit_function_type_get_return_type__(gcc_jit_function_type *function_type)

Given a function type, return its return type.

size_t __gcc_jit_function_type_get_param_count__(gcc_jit_function_type *function_type)

Given a function type, return its number of parameters.

gcc_jit_type *__gcc_jit_function_type_get_param_type__(gcc_jit_function_type *function_type, size_t index)

Given a function type, return the type of the specified parameter.

int __gcc_jit_type_is_integral__(gcc_jit_type *type)

Return non-zero if the type is an integral.

gcc_jit_type *__gcc_jit_type_is_pointer__(gcc_jit_type *type)

Return the type pointed by the pointer type or NULL if it's not a pointer.

gcc_jit_vector_type *__gcc_jit_type_dyncast_vector__(gcc_jit_type *type)

Given a type, return a dynamic cast to a vector type or NULL.

gcc_jit_struct *__gcc_jit_type_is_struct__(gcc_jit_type *type)

Given a type, return a dynamic cast to a struct type or NULL.

size_t __gcc_jit_vector_type_get_num_units__(gcc_jit_vector_type *vector_type)

Given a vector type, return the number of units it contains.

gcc_jit_type *__gcc_jit_vector_type_get_element_type__(gcc_jit_vector_type *vector_type)

Given a vector type, return the type of its elements.

gcc_jit_type *__gcc_jit_type_unqualified__(gcc_jit_type *type)

Given a type, return the unqualified type, removing "const", "volatile" and alignment qualifiers.

gcc_jit_field *__gcc_jit_struct_get_field__(gcc_jit_struct *struct_type, size_t index)

Get a struct field by index.

size_t __gcc_jit_struct_get_field_count__(gcc_jit_struct *struct_type)

Get the number of fields in the struct.

The API entrypoints related to the reflection API:

- `gcc_jit_function_type_get_return_type()`
- `gcc_jit_function_type_get_param_count()`
- `gcc_jit_function_type_get_param_type()`
- `gcc_jit_type_unqualified()`
- `gcc_jit_type_dyncast_array()`
- `gcc_jit_type_is_bool()`
- `gcc_jit_type_dyncast_function_ptr_type()`
- `gcc_jit_type_is_integral()`
- `gcc_jit_type_is_pointer()`
- `gcc_jit_type_dyncast_vector()`
- `gcc_jit_vector_type_get_element_type()`
- `gcc_jit_vector_type_get_num_units()`
- `gcc_jit_struct_get_field()`
- `gcc_jit_type_is_struct()`
- `gcc_jit_struct_get_field_count()`

were added in LIBGCCJIT_ABI_16; you can test for their presence using

```
#ifdef LIBGCCJIT_HAVE_REFLECTION
```

type **gcc_jit_case**

int **gcc_jit_compatible_types**(gcc_jit_type *ltype, gcc_jit_type *rtype)

Return non-zero if the two types are compatible. For instance, if `GCC_JIT_TYPE_UINT64_T` and `GCC_JIT_TYPE_UNSIGNED_LONG` are the same size on the target, this will return non-zero. The parameters `ltype` and `rtype` must be non-NULL. Return 0 on errors.

This entrypoint was added in LIBGCCJIT_ABI_20; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_SIZED_INTEGERS
```

ssize_t **gcc_jit_type_get_size**(gcc_jit_type *type)

Return the size of a type, in bytes. It only works on integer types for now. The parameter `type` must be non-NULL. Return -1 on errors.

This entrypoint was added in LIBGCCJIT_ABI_20; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_SIZED_INTEGERS
```

2.4 Expressions

2.4.1 Rvalues

type **gcc_jit_rvalue**

A `gcc_jit_rvalue` is an expression that can be computed.

It can be simple, e.g.:

- an integer value e.g. *0* or *42*
- a string literal e.g. *"Hello world"*
- a variable e.g. *i*. These are also lvalues (see below).

or compound e.g.:

- a unary expression e.g. *!cond*
- a binary expression e.g. *(a + b)*
- a function call e.g. *get_distance (&player_ship, &target)*
- etc.

Every rvalue has an associated type, and the API will check to ensure that types match up correctly (otherwise the context will emit an error).

gcc_jit_type *__gcc_jit_rvalue_get_type__(gcc_jit_rvalue *rvalue)

Get the type of this rvalue.

gcc_jit_object *__gcc_jit_rvalue_as_object__(gcc_jit_rvalue *rvalue)

Upcast the given rvalue to be an object.

Simple expressions

gcc_jit_rvalue *__gcc_jit_context_new_rvalue_from_int__(gcc_jit_context *ctxt, gcc_jit_type *numeric_type, int value)

Given a numeric type (integer or floating point), build an rvalue for the given constant `int` value.

gcc_jit_rvalue *__gcc_jit_context_new_rvalue_from_long__(gcc_jit_context *ctxt, gcc_jit_type *numeric_type, long value)

Given a numeric type (integer or floating point), build an rvalue for the given constant `long` value.

gcc_jit_rvalue *__gcc_jit_context_zero__(gcc_jit_context *ctxt, gcc_jit_type *numeric_type)

Given a numeric type (integer or floating point), get the rvalue for zero. Essentially this is just a shortcut for:

```
gcc_jit_context_new_rvalue_from_int (ctxt, numeric_type, 0)
```

gcc_jit_rvalue *__gcc_jit_context_one__(gcc_jit_context *ctxt, gcc_jit_type *numeric_type)

Given a numeric type (integer or floating point), get the rvalue for one. Essentially this is just a shortcut for:

```
gcc_jit_context_new_rvalue_from_int (ctxt, numeric_type, 1)
```

gcc_jit_rvalue *__gcc_jit_context_new_rvalue_from_double__(gcc_jit_context *ctxt, gcc_jit_type *numeric_type, double value)

Given a numeric type (integer or floating point), build an rvalue for the given constant `double` value.

gcc_jit_rvalue *__gcc_jit_context_new_rvalue_from_ptr__(gcc_jit_context *ctxt, gcc_jit_type *pointer_type, void *value)

Given a pointer type, build an rvalue for the given address.

gcc_jit_rvalue *__gcc_jit_context_null__(gcc_jit_context *ctxt, gcc_jit_type *pointer_type)

Given a pointer type, build an rvalue for `NULL`. Essentially this is just a shortcut for:

```
gcc_jit_context_new_rvalue_from_ptr (ctxt, pointer_type, NULL)
```

gcc_jit_rvalue *__gcc_jit_context_new_string_literal__(gcc_jit_context *ctxt, const char *value)

Generate an rvalue for the given NIL-terminated string, of type `GCC_JIT_TYPE_CONST_CHAR_PTR`.

The parameter `value` must be non-NULL. The call takes a copy of the underlying string, so it is valid to pass in a pointer to an on-stack buffer.

Constructor expressions

The following functions make constructors for array, struct and union types.

The constructor rvalue can be used for assignment to locals. It can be used to initialize global variables with `gcc_jit_global_set_initializer_rvalue()`. It can also be used as a temporary value for function calls and return values, but its address can't be taken.

Note that arrays in libgccjit do not collapse to pointers like in C. I.e. if an array constructor is used as e.g. a return value, the whole array would be returned by value - array constructors can be assigned to array variables.

The constructor can contain nested constructors.

Note that a string literal rvalue can't be used to construct a char array; the latter needs one rvalue for each char.

These entrypoints were added in LIBGCCJIT_ABI_19; you can test for their presence using:

```
#ifdef LIBGCCJIT_HAVE_CTORS
```

gcc_jit_rvalue *__gcc_jit_context_new_array_constructor__(gcc_jit_context *ctxt, gcc_jit_location *loc, gcc_jit_type *type, size_t num_values, gcc_jit_rvalue **values)

Create a constructor for an array as an rvalue.

Returns NULL on error. `values` are copied and do not have to outlive the context.

`type` specifies what the constructor will build and has to be an array.

`num_values` specifies the number of elements in `values` and it can't have more elements than the array type.

Each value in `values` sets the corresponding value in the array. If the array type itself has more elements than `values`, the left-over elements will be zeroed.

Each value in `values` need to be the same unqualified type as the array type's element type.

If `num_values` is 0, the `values` parameter will be ignored and zero initialization will be used.

This entrypoint was added in LIBGCCJIT_ABI_19; you can test for its presence using:

```
#ifdef LIBGCCJIT_HAVE_CTORS
```

gcc_jit_rvalue *__gcc_jit_context_new_struct_constructor__(gcc_jit_context *ctxt, gcc_jit_location *loc, gcc_jit_type *type, size_t num_values, gcc_jit_field **fields, gcc_jit_rvalue **values)

Create a constructor for a struct as an rvalue.

Returns NULL on error. The two parameter arrays are copied and do not have to outlive the context.

`type` specifies what the constructor will build and has to be a struct.

`num_values` specifies the number of elements in `values`.

`fields` need to have the same length as `values`, or be NULL.

If `fields` is null, the values are applied in definition order.

Otherwise, each field in `fields` specifies which field in the struct to set to the corresponding value in `values`. `fields` and `values` are paired by index.

The fields in `fields` have to be in definition order, but there can be gaps. Any field in the struct that is not specified in `fields` will be zeroed.

The fields in `fields` need to be the same objects that were used to create the struct.

Each value has to have have the same unqualified type as the field it is applied to.

A NULL value element in `values` is a shorthand for zero initialization of the corresponding field.

If `num_values` is 0, the array parameters will be ignored and zero initialization will be used.

This entrypoint was added in LIBGCCJIT_ABI_19; you can test for its presence using:

```
#ifdef LIBGCCJIT_HAVE_CTORS
```

gcc_jit_rvalue *__**gcc_jit_context_new_union_constructor**__(gcc_jit_context *ctxt, gcc_jit_location *loc, gcc_jit_type *type, gcc_jit_field *field, gcc_jit_rvalue *value)

Create a constructor for a union as an rvalue.

Returns NULL on error.

`type` specifies what the constructor will build and has to be an union.

`field` specifies which field to set. If it is NULL, the first field in the union will be set.“field“ need to be the same object that were used to create the union.

`value` specifies what value to set the corresponding field to. If `value` is NULL, zero initialization will be used.

Each value has to have have the same unqualified type as the field it is applied to.

This entrypoint was added in LIBGCCJIT_ABI_19; you can test for its presence using:

```
#ifdef LIBGCCJIT_HAVE_CTORS
```

Vector expressions

gcc_jit_rvalue *__gcc_jit_context_new_rvalue_from_vector__(gcc_jit_context *ctxt, gcc_jit_location *loc, gcc_jit_type *vec_type, size_t num_elements, gcc_jit_rvalue **elements)

Build a vector rvalue from an array of elements.

"vec_type" should be a vector type, created using `gcc_jit_type_get_vector()`.

"num_elements" should match that of the vector type.

This entrypoint was added in LIBGCCJIT_ABI_10; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_gcc_jit_context_new_rvalue_from_vector
```

Unary Operations

gcc_jit_rvalue *__gcc_jit_context_new_unary_op__(gcc_jit_context *ctxt, gcc_jit_location *loc, enum gcc_jit_unary_op op, gcc_jit_type *result_type, gcc_jit_rvalue *rvalue)

Build a unary operation out of an input rvalue.

The parameter `result_type` must be a numeric type.

enum __gcc_jit_unary_op__

The available unary operations are:

Unary Operation	C equivalent
GCC_JIT_UNARY_OP_MINUS	*-(EXPR)*
GCC_JIT_UNARY_OP_BITWISE_NEGATE	*~(EXPR)*
GCC_JIT_UNARY_OP_LOGICAL_NEGATE	*!(EXPR)*
GCC_JIT_UNARY_OP_ABS	*abs (EXPR)*

GCC_JIT_UNARY_OP_MINUS

Negate an arithmetic value; analogous to:

```
-(EXPR)
```

in C.

GCC_JIT_UNARY_OP_BITWISE_NEGATE

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

```
~(EXPR)
```

in C.

GCC_JIT_UNARY_OP_LOGICAL_NEGATE

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

```
!(EXPR)
```

in C.

GCC_JIT_UNARY_OP_ABS

Absolute value of an arithmetic expression; analogous to:

```
abs (EXPR)
```

in C.

Binary Operations

gcc_jit_rvalue ***gcc_jit_context_new_binary_op**(gcc_jit_context *ctxt, gcc_jit_location *loc, enum gcc_jit_binary_op op, gcc_jit_type *result_type, gcc_jit_rvalue *a, gcc_jit_rvalue *b)*

Build a binary operation out of two constituent rvalues.

The parameter `result_type` must be a numeric type.

enum **gcc_jit_binary_op**

The available binary operations are:

Binary Operation	C equivalent
GCC_JIT_BINARY_OP_PLUS	$x + y$
GCC_JIT_BINARY_OP_MINUS	x - y
GCC_JIT_BINARY_OP_MULT	x * y
GCC_JIT_BINARY_OP_DIVIDE	x / y
GCC_JIT_BINARY_OP_MODULO	x % y
GCC_JIT_BINARY_OP_BITWISE_AND	x & y
GCC_JIT_BINARY_OP_BITWISE_XOR	$x \hat{} y$
GCC_JIT_BINARY_OP_BITWISE_OR	$x \mid y$
GCC_JIT_BINARY_OP_LOGICAL_AND	x && y
GCC_JIT_BINARY_OP_LOGICAL_OR	$x \mid\mid y$
GCC_JIT_BINARY_OP_LSHIFT	x « y
GCC_JIT_BINARY_OP_RSHIFT	x » y

GCC_JIT_BINARY_OP_PLUS

Addition of arithmetic values; analogous to:

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

in C.

For pointer addition, use `gcc_jit_context_new_array_access()`.

GCC_JIT_BINARY_OP_MINUS

Subtraction of arithmetic values; analogous to:

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

in C.

GCC_JIT_BINARY_OP_MULT

Multiplication of a pair of arithmetic values; analogous to:

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

in C.

GCC_JIT_BINARY_OP_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.

GCC_JIT_BINARY_OP_MODULO

Remainder of division of arithmetic values; analogous to:

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

in C.

GCC_JIT_BINARY_OP_BITWISE_AND

Bitwise AND; analogous to:

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

in C.

GCC_JIT_BINARY_OP_BITWISE_XOR

Bitwise exclusive OR; analogous to:

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

in C.

GCC_JIT_BINARY_OP_BITWISE_OR

Bitwise inclusive OR; analogous to:

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

in C.

GCC_JIT_BINARY_OP_LOGICAL_AND

Logical AND; analogous to:

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

in C.

GCC_JIT_BINARY_OP_LOGICAL_OR

Logical OR; analogous to:

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

in C.

GCC_JIT_BINARY_OP_LSHIFT

Left shift; analogous to:

```
(EXPR_A) << (EXPR_B)
```

in C.

GCC_JIT_BINARY_OP_RSHIFT

Right shift; analogous to:

```
(EXPR_A) >> (EXPR_B)
```

in C.

Comparisons

gcc_jit_rvalue ***gcc_jit_context_new_comparison**(gcc_jit_context *ctxt, gcc_jit_location *loc, enum gcc_jit_comparison op, gcc_jit_rvalue *a, gcc_jit_rvalue *b)

Build a boolean rvalue out of the comparison of two other rvalues.

enum **gcc_jit_comparison**

Comparison	C equivalent
GCC_JIT_COMPARISON_EQ	$x == y$
GCC_JIT_COMPARISON_NE	$x != y$
GCC_JIT_COMPARISON_LT	$x < y$
GCC_JIT_COMPARISON_LE	$x <= y$
GCC_JIT_COMPARISON_GT	$x > y$
GCC_JIT_COMPARISON_GE	$x >= y$

Function calls

gcc_jit_rvalue *__gcc_jit_context_new_call__(gcc_jit_context *ctxt, gcc_jit_location *loc, gcc_jit_function *func, int numargs, gcc_jit_rvalue **args)

Given a function and the given table of argument rvalues, construct a call to the function, with the result as an rvalue.

Note: `gcc_jit_context_new_call()` merely builds a `gcc_jit_rvalue` i.e. an expression that can be evaluated, perhaps as part of a more complicated expression. The call *won't* happen unless you add a statement to a function that evaluates the expression.

For example, if you want to call a function and discard the result (or to call a function with `void` return type), use `gcc_jit_block_add_eval()`:

```
/* Add "(void)printf (arg0, arg1);".  */
gcc_jit_block_add_eval (
  block, NULL,
  gcc_jit_context_new_call (
    ctxt,
    NULL,
    printf_func,
    2, args));
```

---

gcc_jit_rvalue *__gcc_jit_context_new_call_through_ptr__(gcc_jit_context *ctxt, gcc_jit_location *loc, gcc_jit_rvalue *fn_ptr, int numargs, gcc_jit_rvalue **args)

Given an rvalue of function pointer type (e.g. from `gcc_jit_context_new_function_ptr_type()`), and the given table of argument rvalues, construct a call to the function pointer, with the result as an rvalue.

Note: The same caveat as for `gcc_jit_context_new_call()` applies.

---

void __gcc_jit_rvalue_set_bool_require_tail_call__(gcc_jit_rvalue *call, int require_tail_call)

Given an `gcc_jit_rvalue` for a call created through `gcc_jit_context_new_call()` or `gcc_jit_context_new_call_through_ptr()`, mark/clear the call as needing tail-call optimization. The optimizer will attempt to optimize the call into a jump instruction; if it is unable to do do, an error will be emitted.

This may be useful when implementing functions that use the continuation-passing style (e.g. for functional programming languages), in which every function "returns" by calling a "continuation" function pointer. This call must be guaranteed to be implemented as a jump, otherwise the program could consume an arbitrary amount of stack space as it executed.

This entrypoint was added in LIBGCCJIT_ABI_6; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_gcc_jit_rvalue_set_bool_require_tail_call
```

Function pointers

Function pointers can be obtained:

- from a `gcc_jit_function` using `gcc_jit_function_get_address()`, or
- from an existing function using `gcc_jit_context_new_rvalue_from_ptr()`, using a function pointer type obtained using `gcc_jit_context_new_function_ptr_type()`.

Type-coercion

gcc_jit_rvalue ***gcc_jit_context_new_cast**(gcc_jit_context *ctxt, gcc_jit_location *loc, gcc_jit_rvalue *rvalue, gcc_jit_type *type)*

Given an rvalue of T, construct another rvalue of another type.

Currently only a limited set of conversions are possible:

- int <-> float
- int <-> bool
- P* <-> Q*, for pointer types P and Q

gcc_jit_rvalue ***gcc_jit_context_new_bitcast**(gcc_jit_context *ctxt, gcc_jit_location *loc, gcc_jit_rvalue *rvalue, gcc_jit_type *type)*

Given an rvalue of T, bitcast it to another type, meaning that this will generate a new rvalue by interpreting the bits of `rvalue` to the layout of `type`.

The type of rvalue must be the same size as the size of `type`.

This entrypoint was added in LIBGCCJIT_ABI_21; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_gcc_jit_context_new_bitcast
```

2.4.2 Lvalues

type **gcc_jit_lvalue**

An lvalue is something that can of the *left*-hand side of an assignment: a storage area (such as a variable). It is also usable as an rvalue, where the rvalue is computed by reading from the storage area.

gcc_jit_object ***gcc_jit_lvalue_as_object**(gcc_jit_lvalue *lvalue)*

Upcast an lvalue to be an object.

gcc_jit_rvalue ***gcc_jit_lvalue_as_rvalue**(gcc_jit_lvalue *lvalue)*

Upcast an lvalue to be an rvalue.

gcc_jit_rvalue *__gcc_jit_lvalue_get_address__(gcc_jit_lvalue *lvalue, gcc_jit_location *loc)

Take the address of an lvalue; analogous to:

```
&(EXPR)
```

in C.

void __gcc_jit_lvalue_set_tls_model__(gcc_jit_lvalue *lvalue, enum gcc_jit_tls_model model)

Make a variable a thread-local variable.

The "model" parameter determines the thread-local storage model of the "lvalue":

enum **gcc_jit_tls_model**

GCC_JIT_TLS_MODEL_NONE

Don't set the TLS model.

GCC_JIT_TLS_MODEL_GLOBAL_DYNAMIC

GCC_JIT_TLS_MODEL_LOCAL_DYNAMIC

GCC_JIT_TLS_MODEL_INITIAL_EXEC

GCC_JIT_TLS_MODEL_LOCAL_EXEC

This is analogous to:

```
_Thread_local int foo __attribute__ ((tls_model("MODEL")));
```

in C.

This entrypoint was added in LIBGCCJIT_ABI_17; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_gcc_jit_lvalue_set_tls_model
```

void __gcc_jit_lvalue_set_link_section__(gcc_jit_lvalue *lvalue, const char *section_name)

Set the link section of a variable. The parameter `section_name` must be non-NULL and must contain the leading dot. Analogous to:

```
int variable __attribute__((section(".section")));
```

in C.

This entrypoint was added in LIBGCCJIT_ABI_18; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_gcc_jit_lvalue_set_link_section
```

void __gcc_jit_lvalue_set_register_name__(gcc_jit_lvalue *lvalue, const char *reg_name);

Set the register name of a variable. The parameter `reg_name` must be non-NULL. Analogous to:

```
register int variable asm ("r12");
```

in C.

This entrypoint was added in LIBGCCJIT_ABI_22; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_gcc_jit_lvalue_set_register_name
```

void **gcc_jit_lvalue_set_alignment**(gcc_jit_lvalue *lvalue, unsigned bytes)

Set the alignment of a variable, in bytes. Analogous to:

```
int variable __attribute__((aligned (16)));
```

in C.

This entrypoint was added in LIBGCCJIT_ABI_24; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_ALIGNMENT
```

unsigned **gcc_jit_lvalue_get_alignment**(gcc_jit_lvalue *lvalue)

Return the alignment of a variable set by `gcc_jit_lvalue_set_alignment`. Return 0 if the alignment was not set. Analogous to:

```
_Alignof (variable)
```

in C.

This entrypoint was added in LIBGCCJIT_ABI_24; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_ALIGNMENT
```

Global variables

gcc_jit_lvalue *__gcc_jit_context_new_global__(gcc_jit_context *ctxt, gcc_jit_location *loc, enum gcc_jit_global_kind kind, gcc_jit_type *type, const char *name)

Add a new global variable of the given type and name to the context.

The parameter `type` must be non-*void*.

The parameter `name` must be non-NULL. The call takes a copy of the underlying string, so it is valid to pass in a pointer to an on-stack buffer.

The "kind" parameter determines the visibility of the "global" outside of the `gcc_jit_result`:

enum **gcc_jit_global_kind**

GCC_JIT_GLOBAL_EXPORTED

Global is defined by the client code and is visible by name outside of this JIT context via `gcc_jit_result_get_global()` (and this value is required for the global to be accessible via that entrypoint).

GCC_JIT_GLOBAL_INTERNAL

Global is defined by the client code, but is invisible outside of it. Analogous to a "static" global within a .c file. Specifically, the variable will only be visible within this context and within child contexts.

GCC_JIT_GLOBAL_IMPORTED

Global is not defined by the client code; we're merely referring to it. Analogous to using an "extern" global from a header file.

gcc_jit_lvalue *__gcc_jit_global_set_initializer__(gcc_jit_lvalue *global, const void *blob, size_t num_bytes)

Set an initializer for `global` using the memory content pointed by `blob` for `num_bytes`. `global` must be an array of an integral type. Return the global itself.

The parameter `blob` must be non-NULL. The call copies the memory pointed by `blob` for `num_bytes` bytes, so it is valid to pass in a pointer to an on-stack buffer. The content will be stored in the compilation unit and used as initialization value of the array.

This entrypoint was added in LIBGCCJIT_ABI_14; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_gcc_jit_global_set_initializer
```

gcc_jit_lvalue *__gcc_jit_global_set_initializer_rvalue__(gcc_jit_lvalue *global, gcc_jit_rvalue *init_value)

Set the initial value of a global with an rvalue.

The rvalue needs to be a constant expression, e.g. no function calls.

The global can't have the `kind` `GCC_JIT_GLOBAL_IMPORTED`.

As a non-comprehensive example it is OK to do the equivalent of:

```
int foo = 3 * 2; /* rvalue from gcc_jit_context_new_binary_op.  */
int arr[] = {1,2,3,4}; /* rvalue from gcc_jit_context_new_constructor.  */
int *bar = &arr[2] + 1; /* rvalue from nested "get address" of "array access".  */
const int baz = 3; /* rvalue from gcc_jit_context_rvalue_from_int.  */
int boz = baz; /* rvalue from gcc_jit_lvalue_as_rvalue.  */
```

Use together with `gcc_jit_context_new_struct_constructor()`, `gcc_jit_context_new_union_constructor()`, `gcc_jit_context_new_array_constructor()` to initialize structs, unions and arrays.

On success, returns the `global` parameter unchanged. Otherwise, `NULL`.

This entrypoint was added in LIBGCCJIT_ABI_19; you can test for its presence using:

```
#ifdef LIBGCCJIT_HAVE_CTORS
```

2.4.3 Working with pointers, structs and unions

gcc_jit_lvalue *__gcc_jit_rvalue_dereference__(gcc_jit_rvalue *rvalue, gcc_jit_location *loc)

Given an rvalue of pointer type `T *`, dereferencing the pointer, getting an lvalue of type `T`. Analogous to:

```
*(EXPR)
```

in C.

Field access is provided separately for both lvalues and rvalues.

gcc_jit_lvalue *__gcc_jit_lvalue_access_field__(gcc_jit_lvalue *struct_, gcc_jit_location *loc, gcc_jit_field *field)

Given an lvalue of struct or union type, access the given field, getting an lvalue of the field's type. Analogous to:

```
(EXPR).field = ...;
```

in C.

gcc_jit_rvalue *__gcc_jit_rvalue_access_field__(gcc_jit_rvalue *struct_, gcc_jit_location *loc, gcc_jit_field *field)

Given an rvalue of struct or union type, access the given field as an rvalue. Analogous to:

```
(EXPR).field
```

in C.

gcc_jit_lvalue *__gcc_jit_rvalue_dereference_field__(gcc_jit_rvalue *ptr, gcc_jit_location *loc, gcc_jit_field *field)

Given an rvalue of pointer type `T *` where T is of struct or union type, access the given field as an lvalue. Analogous to:

```
(EXPR)->field
```

in C, itself equivalent to `(*EXPR).FIELD`.

gcc_jit_lvalue *__gcc_jit_context_new_array_access__(gcc_jit_context *ctxt, gcc_jit_location *loc, gcc_jit_rvalue *ptr, gcc_jit_rvalue *index)

Given an rvalue of pointer type `T *`, get at the element T at the given index, using standard C array indexing rules i.e. each increment of `index` corresponds to `sizeof(T)` bytes. Analogous to:

```
PTR[INDEX]
```

in C (or, indeed, to `PTR + INDEX`).

2.5 Creating and using functions

2.5.1 Params

type **gcc_jit_param**

A *gcc_jit_param* represents a parameter to a function.

gcc_jit_param *__gcc_jit_context_new_param__(gcc_jit_context *ctxt, gcc_jit_location *loc, gcc_jit_type *type, const char *name)

In preparation for creating a function, create a new parameter of the given type and name.

The parameter `type` must be non-*void*.

The parameter `name` must be non-NULL. The call takes a copy of the underlying string, so it is valid to pass in a pointer to an on-stack buffer.

Parameters are lvalues, and thus are also rvalues (and objects), so the following upcasts are available:

gcc_jit_lvalue *__gcc_jit_param_as_lvalue__(gcc_jit_param *param)

Upcasting from param to lvalue.

gcc_jit_rvalue *__gcc_jit_param_as_rvalue__(gcc_jit_param *param)

Upcasting from param to rvalue.

gcc_jit_object *__gcc_jit_param_as_object__(gcc_jit_param *param)

Upcasting from param to object.

2.5.2 Functions

type **gcc_jit_function**

A *gcc_jit_function* represents a function - either one that we're creating ourselves, or one that we're referencing.

gcc_jit_function *__gcc_jit_context_new_function__(gcc_jit_context *ctxt, gcc_jit_location *loc, enum gcc_jit_function_kind kind, gcc_jit_type *return_type, const char *name, int num_params, gcc_jit_param **params, int is_variadic)

Create a gcc_jit_function with the given name and parameters.

enum **gcc_jit_function_kind**

This enum controls the kind of function created, and has the following values:

GCC_JIT_FUNCTION_EXPORTED

Function is defined by the client code and visible by name outside of the JIT.

This value is required if you want to extract machine code for this function from a `gcc_jit_result` via `gcc_jit_result_get_code()`.

GCC_JIT_FUNCTION_INTERNAL

Function is defined by the client code, but is invisible outside of the JIT. Analogous to a "static" function.

GCC_JIT_FUNCTION_IMPORTED

Function is not defined by the client code; we're merely referring to it. Analogous to using an "extern" function from a header file.

GCC_JIT_FUNCTION_ALWAYS_INLINE

Function is only ever inlined into other functions, and is invisible outside of the JIT.

Analogous to prefixing with `inline` and adding `__attribute__((always_inline))`

Inlining will only occur when the optimization level is above 0; when optimization is off, this is essentially the same as GCC_JIT_FUNCTION_INTERNAL.

The parameter `name` must be non-NULL. The call takes a copy of the underlying string, so it is valid to pass in a pointer to an on-stack buffer.

gcc_jit_function *__gcc_jit_context_get_builtin_function__(gcc_jit_context *ctxt, const char *name)

Get the `gcc_jit_function` for the built-in function with the given name. For example:

```
gcc_jit_function *fn
  = gcc_jit_context_get_builtin_function (ctxt, "__builtin_memcpy");
```

Note: Due to technical limitations with how libgccjit interacts with the insides of GCC, not all built-in functions are supported. More precisely, not all types are supported for parameters of built-in functions from libgccjit. Attempts to get a built-in function that uses such a parameter will lead to an error being emitted within the context.

---

gcc_jit_object *__gcc_jit_function_as_object__(gcc_jit_function *func)

Upcasting from function to object.

gcc_jit_param *__gcc_jit_function_get_param__(gcc_jit_function *func, int index)

Get the param of the given index (0-based).

void __gcc_jit_function_dump_to_dot__(gcc_jit_function *func, const char *path)

Emit the function in graphviz format to the given path.

gcc_jit_lvalue *__gcc_jit_function_new_local__(gcc_jit_function *func, gcc_jit_location *loc, gcc_jit_type *type, const char *name)

Create a new local variable within the function, of the given type and name.

The parameter `type` must be non-*void*.

The parameter `name` must be non-NULL. The call takes a copy of the underlying string, so it is valid to pass in a pointer to an on-stack buffer.

size_t **gcc_jit_function_get_param_count**(gcc_jit_function *func)

Get the number of parameters of the function.

gcc_jit_type ***gcc_jit_function_get_return_type**(gcc_jit_function *func)

Get the return type of the function.

The API entrypoints relating to getting info about parameters and return types:

- `gcc_jit_function_get_return_type()`
- `gcc_jit_function_get_param_count()`

were added in LIBGCCJIT_ABI_16; you can test for their presence using

```
#ifdef LIBGCCJIT_HAVE_REFLECTION
```

type **gcc_jit_case**

2.5.3 Blocks

type **gcc_jit_block**

A *gcc_jit_block* represents 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, a return, or a switch.

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

gcc_jit_block ***gcc_jit_function_new_block**(gcc_jit_function *func, const char *name)

Create a basic block of the given name. The name may be NULL, but providing meaningful names is often helpful when debugging: it may show up in dumps of the internal representation, and in error messages. It is copied, so the input buffer does not need to outlive the call; you can pass in a pointer to an on-stack buffer, e.g.:

```
for (pc = 0; pc < fn->fn_num_ops; pc++)
 {
   char buf[16];
   sprintf (buf, "instr%i", pc);
   state.op_blocks[pc] = gcc_jit_function_new_block (state.fn, buf);
 }
```

gcc_jit_object ***gcc_jit_block_as_object**(gcc_jit_block *block)

Upcast from block to object.

gcc_jit_function ***gcc_jit_block_get_function**(gcc_jit_block *block)

Which function is this block within?

2.5.4 Statements

void **gcc_jit_block_add_eval**(gcc_jit_block *block, gcc_jit_location *loc, gcc_jit_rvalue *rvalue)

Add evaluation of an rvalue, discarding the result (e.g. a function call that "returns" void).

This is equivalent to this C code:

```
(void)expression;
```

void **gcc_jit_block_add_assignment**(gcc_jit_block *block, gcc_jit_location *loc, gcc_jit_lvalue *lvalue, gcc_jit_rvalue *rvalue)

Add evaluation of an rvalue, assigning the result to the given lvalue.

This is roughly equivalent to this C code:

```
lvalue = rvalue;
```

void **gcc_jit_block_add_assignment_op**(gcc_jit_block *block, gcc_jit_location *loc, gcc_jit_lvalue *lvalue, enum gcc_jit_binary_op op, gcc_jit_rvalue *rvalue)

Add evaluation of an rvalue, using the result to modify an lvalue.

This is analogous to "+=" and friends:

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

etc. For example:

```
/* "i++" */
gcc_jit_block_add_assignment_op (
  loop_body, NULL,
  i,
  GCC_JIT_BINARY_OP_PLUS,
  gcc_jit_context_one (ctxt, int_type));
```

void **gcc_jit_block_add_comment**(gcc_jit_block *block, gcc_jit_location *loc, const char *text)

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 `GCC_JIT_BOOL_OPTION_DUMP_INITIAL_TREE` and `GCC_JIT_BOOL_OPTION_DUMP_INITIAL_GIMPLE`, and thus may be of use when debugging how your project's internal representation gets converted to the libgccjit IR.

The parameter `text` must be non-NULL. It is copied, so the input buffer does not need to outlive the call. For example:

```
char buf[100];
snprintf (buf, sizeof (buf),
          "op%i: %s",
```

(continues on next page)

(continued from previous page)

```
          pc, opcode_names[op->op_opcode]);
gcc_jit_block_add_comment (block, loc, buf);
```

void **gcc_jit_block_end_with_conditional**(gcc_jit_block *block, gcc_jit_location *loc, gcc_jit_rvalue *boolval, gcc_jit_block *on_true, gcc_jit_block *on_false)

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;
```

block, boolval, on_true, and on_false must be non-NULL.

void **gcc_jit_block_end_with_jump**(gcc_jit_block *block, gcc_jit_location *loc, gcc_jit_block *target)

Terminate a block by adding a jump to the given target block.

This is roughly equivalent to this C code:

```
goto target;
```

void **gcc_jit_block_end_with_return**(gcc_jit_block *block, gcc_jit_location *loc, gcc_jit_rvalue *rvalue)

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

This is roughly equivalent to this C code:

```
return expression;
```

void **gcc_jit_block_end_with_void_return**(gcc_jit_block *block, gcc_jit_location *loc)

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;
```

void **gcc_jit_block_end_with_switch**(gcc_jit_block *block, gcc_jit_location *loc, gcc_jit_rvalue *expr, gcc_jit_block *default_block, int num_cases, gcc_jit_case **cases)

Terminate a block by adding evalation of an rvalue, then performing a multiway branch.

This is roughly equivalent to this C code:

```
switch (expr)
  {
  default:
    goto default_block;

  case C0.min_value ... C0.max_value:
    goto C0.dest_block;

  case C1.min_value ... C1.max_value:
    goto C1.dest_block;

  ...etc...

  case C[N - 1].min_value ... C[N - 1].max_value:
    goto C[N - 1].dest_block;
}
```

`block`, `expr`, `default_block` and `cases` must all be non-NULL.

`expr` must be of the same integer type as all of the `min_value` and `max_value` within the cases.

`num_cases` must be >= 0.

The ranges of the cases must not overlap (or have duplicate values).

The API entrypoints relating to switch statements and cases:

- `gcc_jit_block_end_with_switch()`
- `gcc_jit_case_as_object()`
- `gcc_jit_context_new_case()`

were added in LIBGCCJIT_ABI_3; you can test for their presence using

```
#ifdef LIBGCCJIT_HAVE_SWITCH_STATEMENTS
```

type **gcc_jit_case**

A *gcc_jit_case* represents a case within a switch statement, and is created within a particular `gcc_jit_context` using `gcc_jit_context_new_case()`.

Each case expresses a multivalued range of integer values. You can express single-valued cases by passing in the same value for both *min_value* and *max_value*.

gcc_jit_case *__gcc_jit_context_new_case__(gcc_jit_context *ctxt, gcc_jit_rvalue *min_value, gcc_jit_rvalue *max_value, gcc_jit_block *dest_block)

Create a new gcc_jit_case instance for use in a switch statement. *min_value* and *max_value* must be constants of an integer type, which must match that of the expression of the switch statement.

dest_block must be within the same function as the switch statement.

gcc_jit_object *__gcc_jit_case_as_object__(gcc_jit_case *case_)

Upcast from a case to an object.

Here's an example of creating a switch statement:

```
void
create_code (gcc_jit_context *ctxt, void *user_data)
{
  /* Let's try to inject the equivalent of:
     int
     test_switch (int x)
     {
       switch (x)
         {
         case 0 ... 5:
            return 3;

         case 25 ... 27:
            return 4;

         case -42 ... -17:
            return 83;

         case 40:
            return 8;

         default:
            return 10;
         }
     }
   */
  gcc_jit_type *t_int =
    gcc_jit_context_get_type (ctxt, GCC_JIT_TYPE_INT);
  gcc_jit_type *return_type = t_int;
  gcc_jit_param *x =
    gcc_jit_context_new_param (ctxt, NULL, t_int, "x");
  gcc_jit_param *params[1] = {x};
  gcc_jit_function *func =
    gcc_jit_context_new_function (ctxt, NULL,
                                  GCC_JIT_FUNCTION_EXPORTED,
                                  return_type,
                                  "test_switch",
                                  1, params, 0);

  gcc_jit_block *b_initial =
    gcc_jit_function_new_block (func, "initial");

  gcc_jit_block *b_default =
    gcc_jit_function_new_block (func, "default");
  gcc_jit_block *b_case_0_5 =
    gcc_jit_function_new_block (func, "case_0_5");
  gcc_jit_block *b_case_25_27 =
```

(continues on next page)

(continued from previous page)

```
    gcc_jit_function_new_block (func, "case_25_27");
  gcc_jit_block *b_case_m42_m17 =
    gcc_jit_function_new_block (func, "case_m42_m17");
  gcc_jit_block *b_case_40 =
    gcc_jit_function_new_block (func, "case_40");

  gcc_jit_case *cases[4] = {
    gcc_jit_context_new_case (
      ctxt,
      gcc_jit_context_new_rvalue_from_int (ctxt, t_int, 0),
      gcc_jit_context_new_rvalue_from_int (ctxt, t_int, 5),
      b_case_0_5),
    gcc_jit_context_new_case (
      ctxt,
      gcc_jit_context_new_rvalue_from_int (ctxt, t_int, 25),
      gcc_jit_context_new_rvalue_from_int (ctxt, t_int, 27),
      b_case_25_27),
    gcc_jit_context_new_case (
      ctxt,
      gcc_jit_context_new_rvalue_from_int (ctxt, t_int, -42),
      gcc_jit_context_new_rvalue_from_int (ctxt, t_int, -17),
      b_case_m42_m17),
    gcc_jit_context_new_case (
      ctxt,
      gcc_jit_context_new_rvalue_from_int (ctxt, t_int, 40),
      gcc_jit_context_new_rvalue_from_int (ctxt, t_int, 40),
      b_case_40)
  };
  gcc_jit_block_end_with_switch (
    b_initial, NULL,
    gcc_jit_param_as_rvalue (x),
    b_default,
    4, cases);

  gcc_jit_block_end_with_return (
    b_case_0_5, NULL,
    gcc_jit_context_new_rvalue_from_int (ctxt, t_int, 3));
  gcc_jit_block_end_with_return (
    b_case_25_27, NULL,
    gcc_jit_context_new_rvalue_from_int (ctxt, t_int, 4));
  gcc_jit_block_end_with_return (
    b_case_m42_m17, NULL,
    gcc_jit_context_new_rvalue_from_int (ctxt, t_int, 83));
  gcc_jit_block_end_with_return (
    b_case_40, NULL,
    gcc_jit_context_new_rvalue_from_int (ctxt, t_int, 8));
  gcc_jit_block_end_with_return (
    b_default, NULL,
    gcc_jit_context_new_rvalue_from_int (ctxt, t_int, 10));
}
```

See also gcc_jit_extended_asm for entrypoints for adding inline assembler statements to a function.

2.6 Function pointers

You can generate calls that use a function pointer via gcc_jit_context_new_call_through_ptr().

To do requires a gcc_jit_rvalue of the correct function pointer type.

Function pointers for a gcc_jit_function can be obtained via gcc_jit_function_get_address().

gcc_jit_rvalue *gcc_jit_function_get_address(gcc_jit_function *fn, gcc_jit_location *loc)

Get the address of a function as an rvalue, of function pointer type.

This entrypoint was added in LIBGCCJIT_ABI_9; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_gcc_jit_function_get_address
```

Alternatively, given an existing function, you can obtain a pointer to it in gcc_jit_rvalue form using gcc_jit_context_new_rvalue_from_ptr(), using a function pointer type obtained using gcc_jit_context_new_function_ptr_type().

Here's an example of creating a function pointer type corresponding to C's `void (*)(int, int, int)`:

```
gcc_jit_type *void_type =
  gcc_jit_context_get_type (ctxt, GCC_JIT_TYPE_VOID);
gcc_jit_type *int_type =
  gcc_jit_context_get_type (ctxt, GCC_JIT_TYPE_INT);

/* Build the function ptr type.  */
gcc_jit_type *param_types[3];
param_types[0] = int_type;
param_types[1] = int_type;
param_types[2] = int_type;

gcc_jit_type *fn_ptr_type =
  gcc_jit_context_new_function_ptr_type (ctxt, NULL,
                                         void_type,
                                         3, param_types, 0);
```

gcc_jit_type *gcc_jit_context_new_function_ptr_type(gcc_jit_context *ctxt, gcc_jit_location *loc, gcc_jit_type *return_type, int num_params, gcc_jit_type **param_types, int is_variadic)

Generate a gcc_jit_type for a function pointer with the given return type and parameters.

Each of *param_types* must be non-*void*; *return_type* may be *void*.

2.7 Source Locations

type **gcc_jit_location**

A *gcc_jit_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.

gcc_jit_location instances are optional: you can always pass NULL to any API entrypoint accepting one.

You can construct them using `gcc_jit_context_new_location()`.

You need to enable `GCC_JIT_BOOL_OPTION_DEBUGINFO` on the `gcc_jit_context` for these locations to actually be usable by the debugger:

```
gcc_jit_context_set_bool_option (
  ctxt,
  GCC_JIT_BOOL_OPTION_DEBUGINFO,
  1);
```

gcc_jit_location *__gcc_jit_context_new_location__(gcc_jit_context *ctxt, const char *filename, int line, int column)

Create a *gcc_jit_location* instance representing the given source location.

The parameter `filename` must be non-NULL. The call takes a copy of the underlying string, so it is valid to pass in a pointer to an on-stack buffer.

2.7.1 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 `gcc_jit_context_dump_to_file()`:

```
gcc_jit_context_dump_to_file (ctxt, "/tmp/something.c",
                              1 /* update_locations */);
```

This will dump C-like code to the given path. If the *update_locations* argument is true, this will also set up *gcc_jit_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.

2.8 Compiling a context

Once populated, a `gcc_jit_context*` can be compiled to machine code, either in-memory via `gcc_jit_context_compile()` or to disk via `gcc_jit_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.

2.8.1 In-memory compilation

gcc_jit_result *__gcc_jit_context_compile__(gcc_jit_context *ctxt)

This calls into GCC and builds the code, returning a *gcc_jit_result **.

If the result is non-NULL, the caller becomes responsible for calling `gcc_jit_result_release()` on it once they're done with it.

type __gcc_jit_result__

A *gcc_jit_result* encapsulates the result of compiling a context in-memory, and the lifetimes of any machine code functions or globals that are within the result.

void *__gcc_jit_result_get_code__(gcc_jit_result *result, const char *funcname)

Locate a 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 `gcc_jit_context_new_function()` with *kind* `GCC_JIT_FUNCTION_EXPORTED`:

```
gcc_jit_context_new_function (ctxt,
                              any_location, /* or NULL */
                              /* Required for func to be visible to
                                 gcc_jit_result_get_code: */
                              GCC_JIT_FUNCTION_EXPORTED,
                              any_return_type,
                              /* Must string-compare equal: */
                              funcname,
                              /* etc */);
```

If such a function is not found (or *result* or *funcname* are `NULL`), an error message will be emitted on stderr and `NULL` will be returned.

If the function is found, the result will need to be cast to a function pointer of the correct type before it can be called.

Note that the resulting machine code becomes invalid after `gcc_jit_result_release()` is called on the `gcc_jit_result*`; attempting to call it after that may lead to a segmentation fault.

void *__gcc_jit_result_get_global__(gcc_jit_result *result, const char *name)

Locate a given global within the built machine code.

Globals are looked up by name. For this to succeed, a global with a name matching *name* must have been created on *result*'s context (or a parent context) via a call to `gcc_jit_context_new_global()` with *kind* `GCC_JIT_GLOBAL_EXPORTED`.

If the global is found, the result will need to be cast to a pointer of the correct type before it can be called.

This is a *pointer* to the global, so e.g. for an `int` this is an `int*`.

For example, given an `int foo;` created this way:

```
gcc_jit_lvalue *exported_global =
  gcc_jit_context_new_global (ctxt,
  any_location, /* or NULL */
  GCC_JIT_GLOBAL_EXPORTED,
  int_type,
  "foo");
```

we can access it like this:

```
int *ptr_to_foo =
  (int *)gcc_jit_result_get_global (result, "foo");
```

If such a global is not found (or *result* or *name* are `NULL`), an error message will be emitted on stderr and `NULL` will be returned.

Note that the resulting address becomes invalid after `gcc_jit_result_release()` is called on the `gcc_jit_result*`; attempting to use it after that may lead to a segmentation fault.

void **gcc_jit_result_release**(gcc_jit_result *result)

Once we're done with the code, this unloads the built .so file. This cleans up the result; after calling this, it's no longer valid to use the result, or any code or globals that were obtained by calling `gcc_jit_result_get_code()` or `gcc_jit_result_get_global()` on it.

2.8.2 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 `gcc_jit_context_compile_to_file()` API entrypoint.

For linking in object files, use `gcc_jit_context_add_driver_option()`.

void **gcc_jit_context_compile_to_file**(gcc_jit_context *ctxt, enum gcc_jit_output_kind output_kind, const char *output_path)

Compile the `gcc_jit_context*` to a file of the given kind.

`gcc_jit_context_compile_to_file()` ignores the suffix of `output_path`, and insteads uses the given `gcc_jit_output_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.

---

enum **gcc_jit_output_kind**

The available kinds of output are:

Output kind	Typical suffix
GCC_JIT_OUTPUT_KIND_ASSEMBLER	.s
GCC_JIT_OUTPUT_KIND_OBJECT_FILE	.o
GCC_JIT_OUTPUT_KIND_DYNAMIC_LIBRARY	.so or .dll
GCC_JIT_OUTPUT_KIND_EXECUTABLE	None, or .exe

GCC_JIT_OUTPUT_KIND_ASSEMBLER

Compile the context to an assembler file.

GCC_JIT_OUTPUT_KIND_OBJECT_FILE

Compile the context to an object file.

GCC_JIT_OUTPUT_KIND_DYNAMIC_LIBRARY

Compile the context to a dynamic library.

GCC_JIT_OUTPUT_KIND_EXECUTABLE

Compile the context to an executable.

2.9 ABI and API compatibility

The libgccjit developers strive for ABI and API backward-compatibility: programs built against libgccjit.so stand a good chance of running without recompilation against newer versions of libgccjit.so, and ought to recompile without modification against newer versions of libgccjit.h.

Note: The libgccjit++.h C++ API is more experimental, and less locked-down at this time.

API compatibility is achieved by extending the API rather than changing it. For ABI compatiblity, we avoid bumping the SONAME, and instead use symbol versioning to tag each symbol, so that a binary linked against libgccjit.so is tagged according to the symbols that it uses.

For example, `gcc_jit_context_add_command_line_option()` was added in `LIBGCCJIT_ABI_1`. If a client program uses it, this can be detected from metadata by using `objdump`:

```
$ objdump -p testsuite/jit/test-extra-options.c.exe | tail -n 8

Version References:
  required from libgccjit.so.0:
    0x00824161 0x00 04 LIBGCCJIT_ABI_1
    0x00824160 0x00 03 LIBGCCJIT_ABI_0
  required from libc.so.6:
```

You can see the symbol tags provided by libgccjit.so using `objdump`:

```
$ objdump -p libgccjit.so | less
[...snip...]
Version definitions:
```

(continues on next page)

(continued from previous page)

```
1 0x01 0x0ff81f20 libgccjit.so.0
2 0x00 0x00824160 LIBGCCJIT_ABI_0
3 0x00 0x00824161 LIBGCCJIT_ABI_1
       LIBGCCJIT_ABI_0
[...snip...]
```

2.9.1 Programmatically checking version

Client code can programmatically check libgccjit version using:

int **gcc_jit_version_major**(void)

Return libgccjit major version. This is analogous to ___GNUC___ in C code.

int **gcc_jit_version_minor**(void)

Return libgccjit minor version. This is analogous to ___GNUC_MINOR___ in C code.

int **gcc_jit_version_patchlevel**(void)

Return libgccjit patchlevel version. This is analogous to ___GNUC_PATCHLEVEL___ in C code.

Note: These entry points has been added with `LIBGCCJIT_ABI_13` (see below).

2.9.2 ABI symbol tags

The initial release of libgccjit (in gcc 5.1) did not use symbol versioning.

Newer releases use the following tags.

LIBGCCJIT_ABI_0

All entrypoints in the initial release of libgccjit are tagged with `LIBGCCJIT_ABI_0`, to signify the transition to symbol versioning.

Binaries built against older copies of `libgccjit.so` should continue to work, with this being handled transparently by the linker (see this post)

LIBGCCJIT_ABI_1

LIBGCCJIT_ABI_1 covers the addition of gcc_jit_context_add_command_line_option()

LIBGCCJIT_ABI_2

LIBGCCJIT_ABI_2 covers the addition of gcc_jit_context_set_bool_allow_unreachable_blocks()

LIBGCCJIT_ABI_3

LIBGCCJIT_ABI_3 covers the addition of switch statements via API entrypoints:

- gcc_jit_block_end_with_switch()
- gcc_jit_case_as_object()
- gcc_jit_context_new_case()

LIBGCCJIT_ABI_4

LIBGCCJIT_ABI_4 covers the addition of timers via API entrypoints:

- gcc_jit_context_get_timer()
- gcc_jit_context_set_timer()
- gcc_jit_timer_new()
- gcc_jit_timer_release()
- gcc_jit_timer_push()
- gcc_jit_timer_pop()
- gcc_jit_timer_print()

LIBGCCJIT_ABI_5

LIBGCCJIT_ABI_5 covers the addition of gcc_jit_context_set_bool_use_external_driver()

LIBGCCJIT_ABI_6

LIBGCCJIT_ABI_6 covers the addition of gcc_jit_rvalue_set_bool_require_tail_call()

LIBGCCJIT_ABI_7

LIBGCCJIT_ABI_7 covers the addition of gcc_jit_type_get_aligned()

LIBGCCJIT_ABI_8

LIBGCCJIT_ABI_8 covers the addition of gcc_jit_type_get_vector()

LIBGCCJIT_ABI_9

LIBGCCJIT_ABI_9 covers the addition of gcc_jit_function_get_address()

LIBGCCJIT_ABI_10

LIBGCCJIT_ABI_10 covers the addition of gcc_jit_context_new_rvalue_from_vector()

LIBGCCJIT_ABI_11

LIBGCCJIT_ABI_11 covers the addition of gcc_jit_context_add_driver_option()

LIBGCCJIT_ABI_12

LIBGCCJIT_ABI_12 covers the addition of gcc_jit_context_new_bitfield()

LIBGCCJIT_ABI_13

LIBGCCJIT_ABI_13 covers the addition of version functions via API entrypoints:

- gcc_jit_version_major()
- gcc_jit_version_minor()
- gcc_jit_version_patchlevel()

LIBGCCJIT_ABI_14

LIBGCCJIT_ABI_14 covers the addition of gcc_jit_global_set_initializer()

LIBGCCJIT_ABI_15

`LIBGCCJIT_ABI_15` covers the addition of API entrypoints for directly embedding assembler instructions:

- `gcc_jit_block_add_extended_asm()`
- `gcc_jit_block_end_with_extended_asm_goto()`
- `gcc_jit_extended_asm_as_object()`
- `gcc_jit_extended_asm_set_volatile_flag()`
- `gcc_jit_extended_asm_set_inline_flag()`
- `gcc_jit_extended_asm_add_output_operand()`
- `gcc_jit_extended_asm_add_input_operand()`
- `gcc_jit_extended_asm_add_clobber()`
- `gcc_jit_context_add_top_level_asm()`

LIBGCCJIT_ABI_16

`LIBGCCJIT_ABI_16` covers the addition of reflection functions via API entrypoints:

- `gcc_jit_function_get_return_type()`
- `gcc_jit_function_get_param_count()`
- `gcc_jit_type_dyncast_array()`
- `gcc_jit_type_is_bool()`
- `gcc_jit_type_is_integral()`
- `gcc_jit_type_is_pointer()`
- `gcc_jit_type_is_struct()`
- `gcc_jit_type_dyncast_vector()`
- `gcc_jit_type_unqualified()`
- `gcc_jit_type_dyncast_function_ptr_type()`
- `gcc_jit_function_type_get_return_type()`
- `gcc_jit_function_type_get_param_count()`
- `gcc_jit_function_type_get_param_type()`
- `gcc_jit_vector_type_get_num_units()`
- `gcc_jit_vector_type_get_element_type()`
- `gcc_jit_struct_get_field()`
- `gcc_jit_struct_get_field_count()`

LIBGCCJIT_ABI_17

LIBGCCJIT_ABI_17 covers the addition of an API entrypoint to set the thread-local storage model of a variable:

- gcc_jit_lvalue_set_tls_model()

LIBGCCJIT_ABI_18

LIBGCCJIT_ABI_18 covers the addition of an API entrypoint to set the link section of a variable:

- gcc_jit_lvalue_set_link_section()

LIBGCCJIT_ABI_19

LIBGCCJIT_ABI_19 covers the addition of API entrypoints to set the initial value of a global with an rvalue and to use constructors:

- gcc_jit_context_new_array_constructor()
- gcc_jit_context_new_struct_constructor()
- gcc_jit_context_new_union_constructor()
- gcc_jit_global_set_initializer_rvalue()

LIBGCCJIT_ABI_20

LIBGCCJIT_ABI_20 covers the addition of sized integer types, including 128-bit integers and helper functions for types:

- gcc_jit_compatible_types()
- gcc_jit_type_get_size()
- GCC_JIT_TYPE_UINT8_T
- GCC_JIT_TYPE_UINT16_T
- GCC_JIT_TYPE_UINT32_T
- GCC_JIT_TYPE_UINT64_T
- GCC_JIT_TYPE_UINT128_T
- GCC_JIT_TYPE_INT8_T
- GCC_JIT_TYPE_INT16_T
- GCC_JIT_TYPE_INT32_T
- GCC_JIT_TYPE_INT64_T
- GCC_JIT_TYPE_INT128_T

LIBGCCJIT_ABI_21

LIBGCCJIT_ABI_21 covers the addition of an API entrypoint to bitcast a value from one type to another:

- gcc_jit_context_new_bitcast()

LIBGCCJIT_ABI_22

LIBGCCJIT_ABI_22 covers the addition of an API entrypoint to set the register name of a variable:

- gcc_jit_lvalue_set_register_name()

LIBGCCJIT_ABI_23

LIBGCCJIT_ABI_23 covers the addition of an API entrypoint to hide stderr logs:

- gcc_jit_context_set_bool_print_errors_to_stderr()

LIBGCCJIT_ABI_24

LIBGCCJIT_ABI_24 covers the addition of functions to get and set the alignment of a variable:

- gcc_jit_lvalue_set_alignment()
- gcc_jit_lvalue_get_alignment()

2.10 Performance

2.10.1 The timing API

As of GCC 6, libgccjit exposes a timing API, for printing reports on how long was spent in different parts of code.

You can create a gcc_jit_timer instance, which will measure time spent since its creation. The timer maintains a stack of "timer items": as control flow moves through your code, you can push and pop named items relating to your code onto the stack, and the timer will account the time spent accordingly.

You can also asssociate a timer with a gcc_jit_context, in which case the time spent inside compilation will be subdivided.

For example, the following code uses a timer, recording client items "create_code", "compile", and "running code":

```
/* Create a timer.  */
gcc_jit_timer *timer = gcc_jit_timer_new ();
if (!timer)
  {
     error ("gcc_jit_timer_new failed");
     return -1;
  }

/* Let's repeatedly compile and run some code, accumulating it
   all into the timer.  */
for (int i = 0; i < num_iterations; i++)
  {
    /* Create a context and associate it with the timer.  */
    gcc_jit_context *ctxt = gcc_jit_context_acquire ();
    if (!ctxt)
      {
        error ("gcc_jit_context_acquire failed");
        return -1;
      }
    gcc_jit_context_set_timer (ctxt, timer);

    /* Populate the context, timing it as client item "create_code".  */
    gcc_jit_timer_push (timer, "create_code");
    create_code (ctxt);
    gcc_jit_timer_pop (timer, "create_code");

    /* Compile the context, timing it as client item "compile".  */
    gcc_jit_timer_push (timer, "compile");
    result = gcc_jit_context_compile (ctxt);
    gcc_jit_timer_pop (timer, "compile");

    /* Run the generated code, timing it as client item "running code".  */
    gcc_jit_timer_push (timer, "running code");
    run_the_code (ctxt, result);
    gcc_jit_timer_pop (timer, "running code");

    /* Clean up.  */
    gcc_jit_context_release (ctxt);
    gcc_jit_result_release (result);
}

/* Print the accumulated timings.  */
gcc_jit_timer_print (timer, stderr);
gcc_jit_timer_release (timer);
```

giving output like this, showing the internal GCC items at the top, then client items, then the total:

```
Execution times (seconds)
GCC items:
 phase setup             :   0.29 (14%) usr   0.00 ( 0%) sys   0.32 ( 5%) wall   10661 kB (50%)␣
↪ggc
```

(continues on next page)

(continued from previous page)

```
 phase parsing           :   0.02 ( 1%) usr   0.00 ( 0%) sys   0.00 ( 0%) wall     653 kB ( 3%)␣
↪ggc
 phase finalize          :   0.01 ( 1%) usr   0.00 ( 0%) sys   0.00 ( 0%) wall       0 kB ( 0%)␣
↪ggc
 dump files              :   0.02 ( 1%) usr   0.00 ( 0%) sys   0.01 ( 0%) wall       0 kB ( 0%)␣
↪ggc
 callgraph construction  :   0.02 ( 1%) usr   0.01 ( 6%) sys   0.01 ( 0%) wall     242 kB ( 1%)␣
↪ggc
 callgraph optimization  :   0.03 ( 2%) usr   0.00 ( 0%) sys   0.02 ( 0%) wall     142 kB ( 1%)␣
↪ggc
 trivially dead code     :   0.01 ( 1%) usr   0.00 ( 0%) sys   0.00 ( 0%) wall       0 kB ( 0%)␣
↪ggc
 df scan insns           :   0.01 ( 1%) usr   0.00 ( 0%) sys   0.00 ( 0%) wall       9 kB ( 0%)␣
↪ggc
 df live regs            :   0.01 ( 1%) usr   0.00 ( 0%) sys   0.01 ( 0%) wall       0 kB ( 0%)␣
↪ggc
 inline parameters       :   0.02 ( 1%) usr   0.00 ( 0%) sys   0.01 ( 0%) wall      82 kB ( 0%)␣
↪ggc
 tree CFG cleanup        :   0.01 ( 1%) usr   0.00 ( 0%) sys   0.00 ( 0%) wall       0 kB ( 0%)␣
↪ggc
 tree PHI insertion      :   0.01 ( 1%) usr   0.00 ( 0%) sys   0.02 ( 0%) wall      64 kB ( 0%)␣
↪ggc
 tree SSA other          :   0.01 ( 1%) usr   0.00 ( 0%) sys   0.01 ( 0%) wall      18 kB ( 0%)␣
↪ggc
 expand                  :   0.01 ( 1%) usr   0.00 ( 0%) sys   0.00 ( 0%) wall     398 kB ( 2%)␣
↪ggc
 jump                    :   0.01 ( 1%) usr   0.00 ( 0%) sys   0.00 ( 0%) wall       0 kB ( 0%)␣
↪ggc
 loop init               :   0.01 ( 0%) usr   0.00 ( 0%) sys   0.00 ( 0%) wall      67 kB ( 0%)␣
↪ggc
 integrated RA           :   0.02 ( 1%) usr   0.00 ( 0%) sys   0.00 ( 0%) wall    2468 kB (12%)␣
↪ggc
 thread pro- & epilogue  :   0.01 ( 1%) usr   0.00 ( 0%) sys   0.00 ( 0%) wall     162 kB ( 1%)␣
↪ggc
 final                   :   0.01 ( 1%) usr   0.00 ( 0%) sys   0.00 ( 0%) wall     216 kB ( 1%)␣
↪ggc
 rest of compilation     :   1.37 (69%) usr   0.00 ( 0%) sys   1.13 (18%) wall    1391 kB ( 6%)␣
↪ggc
 assemble JIT code       :   0.01 ( 1%) usr   0.00 ( 0%) sys   4.04 (66%) wall       0 kB ( 0%)␣
↪ggc
 load JIT result         :   0.02 ( 1%) usr   0.00 ( 0%) sys   0.00 ( 0%) wall       0 kB ( 0%)␣
↪ggc
 JIT client code         :   0.00 ( 0%) usr   0.01 ( 6%) sys   0.00 ( 0%) wall       0 kB ( 0%)␣
↪ggc
Client items:
 create_code             :   0.00 ( 0%) usr   0.01 ( 6%) sys   0.00 ( 0%) wall       0 kB ( 0%)␣
↪ggc
 compile                 :   0.36 (18%) usr   0.15 (83%) sys   0.86 (14%) wall   14939 kB (70%)␣
↪ggc
 running code            :   0.00 ( 0%) usr   0.00 ( 0%) sys   0.00 ( 0%) wall       0 kB ( 0%)␣
↪ggc
 TOTAL                   :   2.00             0.18             6.12             21444 kB
```

The exact format is intended to be human-readable, and is subject to change.

LIBGCCJIT_HAVE_TIMING_API

The timer API was added to libgccjit in GCC 6. This macro is only defined in versions of libgccjit.h which have the timer API, and so can be used to guard code that may need to compile against earlier releases:

```
#ifdef LIBGCCJIT_HAVE_TIMING_API
gcc_jit_timer *t = gcc_jit_timer_new ();
gcc_jit_context_set_timer (ctxt, t);
#endif
```

type **gcc_jit_timer**

gcc_jit_timer ***gcc_jit_timer_new**(void)

Create a gcc_jit_timer instance, and start timing:

```
gcc_jit_timer *t = gcc_jit_timer_new ();
```

This API entrypoint was added in LIBGCCJIT_ABI_4; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_TIMING_API
```

void **gcc_jit_timer_release**(gcc_jit_timer *timer)

Release a gcc_jit_timer instance:

```
gcc_jit_timer_release (t);
```

This should be called exactly once on a timer.

This API entrypoint was added in LIBGCCJIT_ABI_4; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_TIMING_API
```

void **gcc_jit_context_set_timer**(gcc_jit_context *ctxt, gcc_jit_timer *timer)

Associate a gcc_jit_timer instance with a context:

```
gcc_jit_context_set_timer (ctxt, t);
```

A timer instance can be shared between multiple gcc_jit_context instances.

Timers have no locking, so if you have a multithreaded program, you must provide your own locks if more than one thread could be working with the same timer via timer-associated contexts.

This API entrypoint was added in LIBGCCJIT_ABI_4; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_TIMING_API
```

gcc_jit_timer ***gcc_jit_context_get_timer**(gcc_jit_context *ctxt)

Get the timer associated with a context (if any).

This API entrypoint was added in LIBGCCJIT_ABI_4; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_TIMING_API
```

void **gcc_jit_timer_push**(gcc_jit_timer *timer, const char *item_name)

Push the given item onto the timer's stack:

```
gcc_jit_timer_push (t, "running code");
run_the_code (ctxt, result);
gcc_jit_timer_pop (t, "running code");
```

This API entrypoint was added in LIBGCCJIT_ABI_4; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_TIMING_API
```

void **gcc_jit_timer_pop**(gcc_jit_timer *timer, const char *item_name)

Pop the top item from the timer's stack.

If "item_name" is provided, it must match that of the top item. Alternatively, `NULL` can be passed in, to suppress checking.

This API entrypoint was added in LIBGCCJIT_ABI_4; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_TIMING_API
```

void **gcc_jit_timer_print**(gcc_jit_timer *timer, FILE *f_out)

Print timing information to the given stream about activity since the timer was started.

This API entrypoint was added in LIBGCCJIT_ABI_4; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_TIMING_API
```

2.11 Using Assembly Language with libgccjit

libgccjit has some support for directly embedding assembler instructions. This is based on GCC's support for inline `asm` in C code, and the following assumes a familiarity with that functionality. See How to Use Inline Assembly Language in C Code in GCC's documentation, the "Extended Asm" section in particular.

These entrypoints were added in LIBGCCJIT_ABI_15; you can test for their presence using

```
#ifdef LIBGCCJIT_HAVE_ASM_STATEMENTS
```

2.11.1 Adding assembler instructions within a function

type **gcc_jit_extended_asm**

A *gcc_jit_extended_asm* represents an extended `asm` statement: a series of low-level instructions inside a function that convert inputs to outputs.

To avoid having an API entrypoint with a very large number of parameters, an extended `asm` statement is made in stages: an initial call to create the `gcc_jit_extended_asm`, followed by calls to add operands and set other properties of the statement.

There are two API entrypoints for creating a `gcc_jit_extended_asm`:

- `gcc_jit_block_add_extended_asm()` for an `asm` statement with no control flow, and
- `gcc_jit_block_end_with_extended_asm_goto()` for an `asm goto`.

For example, to create the equivalent of:

```
      asm ("mov %1, %0\n\t"
           "add $1, %0"
           : "=r" (dst)
           : "r" (src));
```

the following API calls could be used:

```
gcc_jit_extended_asm *ext_asm
  = gcc_jit_block_add_extended_asm (block, NULL,
                                    "mov %1, %0\n\t"
                                    "add $1, %0");
gcc_jit_extended_asm_add_output_operand (ext_asm, NULL, "=r", dst);
gcc_jit_extended_asm_add_input_operand (ext_asm, NULL, "r",
                                        gcc_jit_lvalue_as_rvalue (src));
```

Warning: When considering the numbering of operands within an extended `asm` statement (e.g. the `%0` and `%1` above), the equivalent to the C syntax is followed i.e. all output operands, then all input operands, regardless of what order the calls to `gcc_jit_extended_asm_add_output_operand()` and `gcc_jit_extended_asm_add_input_operand()` were made in.

As in the C syntax, operands can be given symbolic names to avoid having to number them. For example, to create the equivalent of:

```
      asm ("bsfl %[aMask], %[aIndex]"
           : [aIndex] "=r" (Index)
           : [aMask] "r" (Mask)
           : "cc");
```

the following API calls could be used:

```
gcc_jit_extended_asm *ext_asm
  = gcc_jit_block_add_extended_asm (block, NULL,
                                    "bsfl %[aMask], %[aIndex]");
gcc_jit_extended_asm_add_output_operand (ext_asm, "aIndex", "=r", index);
gcc_jit_extended_asm_add_input_operand (ext_asm, "aMask", "r",
                                        gcc_jit_param_as_rvalue (mask));
gcc_jit_extended_asm_add_clobber (ext_asm, "cc");
```

gcc_jit_extended_asm *__gcc_jit_block_add_extended_asm__(gcc_jit_block *block, gcc_jit_location *loc, const char *asm_template)

Create a gcc_jit_extended_asm for an extended `asm` statement with no control flow (i.e. without the `goto` qualifier).

The parameter `asm_template` corresponds to the *AssemblerTemplate* within C's extended `asm` syntax. It must be non-NULL. The call takes a copy of the underlying string, so it is valid to pass in a pointer to an on-stack buffer.

gcc_jit_extended_asm *__gcc_jit_block_end_with_extended_asm_goto__(gcc_jit_block *block, gcc_jit_location *loc, const char *asm_template, int num_goto_blocks, gcc_jit_block **goto_blocks, gcc_jit_block *fallthrough_block)

Create a gcc_jit_extended_asm for an extended `asm` statement that may perform jumps, and use it to terminate the given block. This is equivalent to the `goto` qualifier in C's extended `asm` syntax.

For example, to create the equivalent of:

```
      asm goto ("btl %1, %0\n\t"
                "jc %l[carry]"
                : // No outputs
                : "r" (p1), "r" (p2)
                : "cc"
                : carry);
```

the following API calls could be used:

```
const char *asm_template =
  (use_name
   ? /* Label referred to by name: "%l[carry]".  */
     ("btl %1, %0\n\t"
      "jc %l[carry]")
   : /* Label referred to numerically: "%l2".  */
     ("btl %1, %0\n\t"
      "jc %l2"));
```

(continues on next page)

(continued from previous page)

```
  gcc_jit_extended_asm *ext_asm
    = gcc_jit_block_end_with_extended_asm_goto (b_start, NULL,
                                                asm_template,
                                                1, &b_carry,
                                                b_fallthru);
  gcc_jit_extended_asm_add_input_operand (ext_asm, NULL, "r",
                                          gcc_jit_param_as_rvalue (p1));
  gcc_jit_extended_asm_add_input_operand (ext_asm, NULL, "r",
                                          gcc_jit_param_as_rvalue (p2));
  gcc_jit_extended_asm_add_clobber (ext_asm, "cc");
```

here referencing a gcc_jit_block named “carry”.

num_goto_blocks must be >= 0.

goto_blocks must be non-NULL. This corresponds to the GotoLabels parameter within C’s extended asm syntax. The block names can be referenced within the assembler template.

fallthrough_block can be NULL. If non-NULL, it specifies the block to fall through to after the statement.

Note: This is needed since each gcc_jit_block must have a single exit point, as a basic block: you can’t jump from the middle of a block. A “goto” is implicitly added after the asm to handle the fallthrough case, which is equivalent to what would have happened in the C case.

---

void **gcc_jit_extended_asm_set_volatile_flag**(gcc_jit_extended_asm *ext_asm, int flag)

Set whether the gcc_jit_extended_asm has side-effects, equivalent to the volatile qualifier in C’s extended asm syntax.

For example, to create the equivalent of:

```
asm volatile ("rdtsc\n\t"    // Returns the time in EDX:EAX.
              "shl $32, %%rdx\n\t"  // Shift the upper bits left.
              "or %%rdx, %0"        // 'Or' in the lower bits.
              : "=a" (msr)
              :
              : "rdx");
```

the following API calls could be used:

```
  gcc_jit_extended_asm *ext_asm
    = gcc_jit_block_add_extended_asm
        (block, NULL,
         "rdtsc\n\t"  /* Returns the time in EDX:EAX.  */
         "shl $32, %%rdx\n\t"  /* Shift the upper bits left.  */
         "or %%rdx, %0");  /* 'Or' in the lower bits.  */
  gcc_jit_extended_asm_set_volatile_flag (ext_asm, 1);
  gcc_jit_extended_asm_add_output_operand (ext_asm, NULL, "=a", msr);
  gcc_jit_extended_asm_add_clobber (ext_asm, "rdx");
```

where the gcc_jit_extended_asm is flagged as volatile.

void **gcc_jit_extended_asm_set_inline_flag**(gcc_jit_extended_asm *ext_asm, int flag)

Set the equivalent of the inline qualifier in C's extended `asm` syntax.

void **gcc_jit_extended_asm_add_output_operand**(gcc_jit_extended_asm *ext_asm, const char *asm_symbolic_name, const char *constraint, gcc_jit_lvalue *dest)

Add an output operand to the extended `asm` statement. See the Output Operands section of the documentation of the C syntax.

`asm_symbolic_name` corresponds to the `asmSymbolicName` component of C's extended `asm` syntax. It can be NULL. If non-NULL it specifies the symbolic name for the operand.

`constraint` corresponds to the `constraint` component of C's extended `asm` syntax. It must be non-NULL.

`dest` corresponds to the `cvariablename` component of C's extended `asm` syntax. It must be non-NULL.

```
// Example with a NULL symbolic name, the equivalent of:
//   : "=r" (dst)
gcc_jit_extended_asm_add_output_operand (ext_asm, NULL, "=r", dst);

// Example with a symbolic name ("aIndex"), the equivalent of:
//   : [aIndex] "=r" (index)
gcc_jit_extended_asm_add_output_operand (ext_asm, "aIndex", "=r", index);
```

This function can't be called on an `asm goto` as such instructions can't have outputs; see the Goto Labels section of GCC's "Extended Asm" documentation.

void **gcc_jit_extended_asm_add_input_operand**(gcc_jit_extended_asm *ext_asm, const char *asm_symbolic_name, const char *constraint, gcc_jit_rvalue *src)

Add an input operand to the extended `asm` statement. See the Input Operands section of the documentation of the C syntax.

`asm_symbolic_name` corresponds to the `asmSymbolicName` component of C's extended `asm` syntax. It can be NULL. If non-NULL it specifies the symbolic name for the operand.

`constraint` corresponds to the `constraint` component of C's extended `asm` syntax. It must be non-NULL.

`src` corresponds to the `cexpression` component of C's extended `asm` syntax. It must be non-NULL.

```
// Example with a NULL symbolic name, the equivalent of:
//   : "r" (src)
gcc_jit_extended_asm_add_input_operand (ext_asm, NULL, "r",
                                        gcc_jit_lvalue_as_rvalue (src));

// Example with a symbolic name ("aMask"), the equivalent of:
//   : [aMask] "r" (Mask)
```

(continues on next page)

(continued from previous page)

```
gcc_jit_extended_asm_add_input_operand (ext_asm, "aMask", "r",
                                        gcc_jit_lvalue_as_rvalue (mask));
```

void **gcc_jit_extended_asm_add_clobber**(gcc_jit_extended_asm *ext_asm, const char *victim)

Add *victim* to the list of registers clobbered by the extended `asm` statement. It must be non-NULL. See the Clobbers and Scratch Registers section of the documentation of the C syntax.

Statements with multiple clobbers will require multiple calls, one per clobber.

For example:

```
gcc_jit_extended_asm_add_clobber (ext_asm, "r0");
gcc_jit_extended_asm_add_clobber (ext_asm, "cc");
gcc_jit_extended_asm_add_clobber (ext_asm, "memory");
```

A `gcc_jit_extended_asm` is a `gcc_jit_object` "owned" by the block's context. The following upcast is available:

gcc_jit_object ***gcc_jit_extended_asm_as_object**(gcc_jit_extended_asm *ext_asm)

Upcast from extended `asm` to object.

2.11.2 Adding top-level assembler statements

In addition to creating extended `asm` instructions within a function, there is support for creating "top-level" assembler statements, outside of any function.

void **gcc_jit_context_add_top_level_asm**(gcc_jit_context *ctxt, gcc_jit_location *loc, const char *asm_stmts)

Create a set of top-level asm statements, analogous to those created by GCC's "basic" `asm` syntax in C at file scope.

For example, to create the equivalent of:

```
    asm ("\t.pushsection .text\n"
         "\t.globl add_asm\n"
         "\t.type add_asm, @function\n"
         "add_asm:\n"
         "\tmovq %rdi, %rax\n"
         "\tadd %rsi, %rax\n"
         "\tret\n"
         "\t.popsection\n");
```

the following API calls could be used:

```
  gcc_jit_context_add_top_level_asm (ctxt, NULL,
                                     "\t.pushsection .text\n"
                                     "\t.globl add_asm\n"
                                     "\t.type add_asm, @function\n"
```

(continues on next page)

(continued from previous page)

```
                                "add_asm:\n"
                                "\tmovq %rdi, %rax\n"
                                "\tadd %rsi, %rax\n"
                                "\tret\n"
                                "\t# some asm here\n"
                                "\t.popsection\n");
```

CHAPTER

THREE

C++ BINDINGS FOR LIBGCCJIT

This document describes the C++ bindings to libgccjit, an API for embedding GCC inside programs and libraries.

The C++ bindings consist of a single header file `libgccjit++.h`.

This is a collection of "thin" wrapper classes around the C API. Everything is an inline function, implemented in terms of the C API, so there is nothing extra to link against.

Contents:

3.1 Tutorial

3.1.1 Tutorial part 1: "Hello world"

Before we look at the details of the API, let's look at building and running programs that use the library.

Here's a toy "hello world" program that uses the library's C++ API to synthesize a call to *printf* and uses it to write a message to stdout.

Don't worry about the content of the program for now; we'll cover the details in later parts of this tutorial.

```
/* Smoketest example for libgccjit.so C++ API
   Copyright (C) 2014-2022 Free Software Foundation, Inc.

This file is part of GCC.

GCC is free software; you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 3, or (at your option)
any later version.

GCC is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.
```

(continues on next page)

(continued from previous page)

```
You should have received a copy of the GNU General Public License
along with GCC; see the file COPYING3.  If not see
<http://www.gnu.org/licenses/>.  */

#include <libgccjit++.h>

#include <stdlib.h>
#include <stdio.h>

static void
create_code (gccjit::context ctxt)
{
  /* Let's try to inject the equivalent of this C code:
     void
     greet (const char *name)
     {
        printf ("hello %s\n", name);
     }
  */
  gccjit::type void_type = ctxt.get_type (GCC_JIT_TYPE_VOID);
  gccjit::type const_char_ptr_type =
    ctxt.get_type (GCC_JIT_TYPE_CONST_CHAR_PTR);
  gccjit::param param_name =
    ctxt.new_param (const_char_ptr_type, "name");
  std::vector<gccjit::param> func_params;
  func_params.push_back (param_name);
  gccjit::function func =
    ctxt.new_function (GCC_JIT_FUNCTION_EXPORTED,
                       void_type,
                       "greet",
                       func_params, 0);

  gccjit::param param_format =
    ctxt.new_param (const_char_ptr_type, "format");
  std::vector<gccjit::param> printf_params;
  printf_params.push_back (param_format);
  gccjit::function printf_func =
    ctxt.new_function (GCC_JIT_FUNCTION_IMPORTED,
                       ctxt.get_type (GCC_JIT_TYPE_INT),
                       "printf",
                       printf_params, 1);

  gccjit::block block = func.new_block ();
  block.add_eval (ctxt.new_call (printf_func,
                                 ctxt.new_rvalue ("hello %s\n"),
                                 param_name));
  block.end_with_return ();
}

int
main (int argc, char **argv)
```

(continues on next page)

(continued from previous page)

```
{
  gccjit::context ctxt;
  gcc_jit_result *result;

  /* Get a "context" object for working with the library.  */
  ctxt = gccjit::context::acquire ();

  /* Set some options on the context.
     Turn this on to see the code being generated, in assembler form.  */
  ctxt.set_bool_option (GCC_JIT_BOOL_OPTION_DUMP_GENERATED_CODE, 0);

  /* Populate the context.  */
  create_code (ctxt);

  /* Compile the code.  */
  result = ctxt.compile ();
  if (!result)
    {
      fprintf (stderr, "NULL result");
      exit (1);
    }

  ctxt.release ();

  /* Extract the generated code from "result".  */
  typedef void (*fn_type) (const char *);
  fn_type greet =
    (fn_type)gcc_jit_result_get_code (result, "greet");
  if (!greet)
    {
      fprintf (stderr, "NULL greet");
      exit (1);
    }

  /* Now call the generated function: */
  greet ("world");
  fflush (stdout);

  gcc_jit_result_release (result);
  return 0;
}
```

Copy the above to *tut01-hello-world.cc*.

Assuming you have the jit library installed, build the test program using:

```
$ gcc \
    tut01-hello-world.cc \
    -o tut01-hello-world \
    -lgccjit
```

You should then be able to run the built program:

```
$ ./tut01-hello-world
hello world
```

3.1.2 Tutorial part 2: Creating a trivial machine code function

Consider this C function:

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

How can we construct this at run-time using libgccjit's C++ API?

First we need to include the relevant header:

```
#include <libgccjit++.h>
```

All state associated with compilation is associated with a `gccjit::context`, which is a thin C++ wrapper around the C API's `gcc_jit_context`*.

Create one using `gccjit::context::acquire()`:

```
gccjit::context ctxt;
ctxt = gccjit::context::acquire ();
```

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`, using `gccjit::context::get_type()`:

```
gccjit::type int_type = ctxt.get_type (GCC_JIT_TYPE_INT);
```

`gccjit::type` is an example of a "contextual" object: every entity in the API is associated with a `gccjit::context`.

Memory management is easy: all such "contextual" objects are automatically cleaned up for you when the context is released, using `gccjit::context::release()`:

```
ctxt.release ();
```

so you don't need to manually track and cleanup all objects, just the contexts.

All of the C++ classes in the API are thin wrappers around pointers to types in the C API.

The C++ class hierarchy within the `gccjit` namespace looks like this:

```
+- object
    +- location
    +- type
       +- struct
    +- field
```

(continues on next page)

(continued from previous page)

```
   +- function
   +- block
   +- rvalue
       +- lvalue
          +- param
```

One thing you can do with a `gccjit::object` is to ask it for a human-readable description as a `std::string`, using `gccjit::object::get_debug_string()`:

```
printf ("obj: %s\n", obj.get_debug_string ().c_str ());
```

giving this text on stdout:

```
obj: int
```

This is invaluable when debugging.

Let's create the function. To do so, we first need to construct its single parameter, specifying its type and giving it a name, using `gccjit::context::new_param()`:

```
gccjit::param param_i = ctxt.new_param (int_type, "i");
```

and we can then make a vector of all of the params of the function, in this case just one:

```
std::vector<gccjit::param> params;
params.push_back (param_i);
```

Now we can create the function, using `gccjit::context::new_function()`:

```
gccjit::function func =
  ctxt.new_function (GCC_JIT_FUNCTION_EXPORTED,
                     int_type,
                     "square",
                     params,
                     0);
```

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:

```
gccjit::block block = func.new_block ();
```

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

We can build the expression using `gccjit::context::new_binary_op()`:

```
gccjit::rvalue expr =
  ctxt.new_binary_op (
    GCC_JIT_BINARY_OP_MULT, int_type,
    param_i, param_i);
```

A gccjit::rvalue is another example of a gccjit::object subclass. As before, we can print it with gccjit::object::get_debug_string().

```
printf ("expr: %s\n", expr.get_debug_string ().c_str ());
```

giving this output:

```
expr: i * i
```

Note that gccjit::rvalue provides numerous overloaded operators which can be used to dramatically reduce the amount of typing needed. We can build the above binary operation more directly with this one-liner:

```
gccjit::rvalue expr = param_i * param_i;
```

Creating the expression 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 using gccjit::context::compile():

```
gcc_jit_result *result;
result = ctxt.compile ();
```

and get a gcc_jit_result*.

We can now use gcc_jit_result_get_code() to look up a specific machine code routine within the result, in this case, the function we created above.

```
void *fn_ptr = gcc_jit_result_get_code (result, "square");
if (!fn_ptr)
  {
    fprintf (stderr, "NULL fn_ptr");
    goto error;
  }
```

We can now cast the pointer to an appropriate function pointer type, and then call it:

```
typedef int (*fn_type) (int);
fn_type square = (fn_type)fn_ptr;
printf ("result: %d", square (5));
```

```
result: 25
```

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 `GCC_JIT_BOOL_OPTION_DUMP_INITIAL_GIMPLE` will dump a C-like representation to stderr when you compile (GCC's "GIMPLE" representation):

```
ctxt.set_bool_option (GCC_JIT_BOOL_OPTION_DUMP_INITIAL_GIMPLE, 1);
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 `GCC_JIT_BOOL_OPTION_DUMP_GENERATED_CODE` on the context before compiling:

```
ctxt.set_bool_option (GCC_JIT_BOOL_OPTION_DUMP_GENERATED_CODE, 1);
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)"
      .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 `GCC_JIT_INT_OPTION_OPTIMIZATION_LEVEL`:

```
ctxt.set_int_option (GCC_JIT_INT_OPTION_OPTIMIZATION_LEVEL, 3);
```

```
      .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)"
      .section        .note.GNU-stack,"",@progbits
```

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

Full example

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

```
/* Usage example for libgccjit.so's C++ API
   Copyright (C) 2014-2022 Free Software Foundation, Inc.

This file is part of GCC.

GCC is free software; you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 3, or (at your option)
any later version.

GCC is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with GCC; see the file COPYING3.  If not see
<http://www.gnu.org/licenses/>.  */

#include <libgccjit++.h>

#include <stdlib.h>
```

(continues on next page)

(continued from previous page)

```
#include <stdio.h>

void
create_code (gccjit::context ctxt)
{
  /* Let's try to inject the equivalent of this C code:

     int square (int i)
     {
       return i * i;
     }
  */
  gccjit::type int_type = ctxt.get_type (GCC_JIT_TYPE_INT);
  gccjit::param param_i = ctxt.new_param (int_type, "i");
  std::vector<gccjit::param> params;
  params.push_back (param_i);
  gccjit::function func = ctxt.new_function (GCC_JIT_FUNCTION_EXPORTED,
                                             int_type,
                                             "square",
                                             params, 0);

  gccjit::block block = func.new_block ();

  gccjit::rvalue expr =
    ctxt.new_binary_op (GCC_JIT_BINARY_OP_MULT, int_type,
                        param_i, param_i);

  block.end_with_return (expr);
}

int
main (int argc, char **argv)
{
  /* Get a "context" object for working with the library.  */
  gccjit::context ctxt = gccjit::context::acquire ();

  /* Set some options on the context.
     Turn this on to see the code being generated, in assembler form.  */
  ctxt.set_bool_option (
    GCC_JIT_BOOL_OPTION_DUMP_GENERATED_CODE,
    0);

  /* Populate the context.  */
  create_code (ctxt);

  /* Compile the code.  */
  gcc_jit_result *result = ctxt.compile ();

  /* We're done with the context; we can release it: */
  ctxt.release ();
```

(continues on next page)

(continued from previous page)

```
  if (!result)
    {
      fprintf (stderr, "NULL result");
      return 1;
    }

  /* Extract the generated code from "result".  */
  void *fn_ptr = gcc_jit_result_get_code (result, "square");
  if (!fn_ptr)
     {
       fprintf (stderr, "NULL fn_ptr");
       gcc_jit_result_release (result);
       return 1;
     }

  typedef int (*fn_type) (int);
  fn_type square = (fn_type)fn_ptr;
  printf ("result: %d\n", square (5));

  gcc_jit_result_release (result);
  return 0;
}
```

Building and running it:

```
$ gcc \
    tut02-square.cc \
    -o tut02-square \
    -lgccjit

# Run the built program:
$ ./tut02-square
result: 25
```

3.1.3 Tutorial part 3: Loops and variables

Consider this C function:

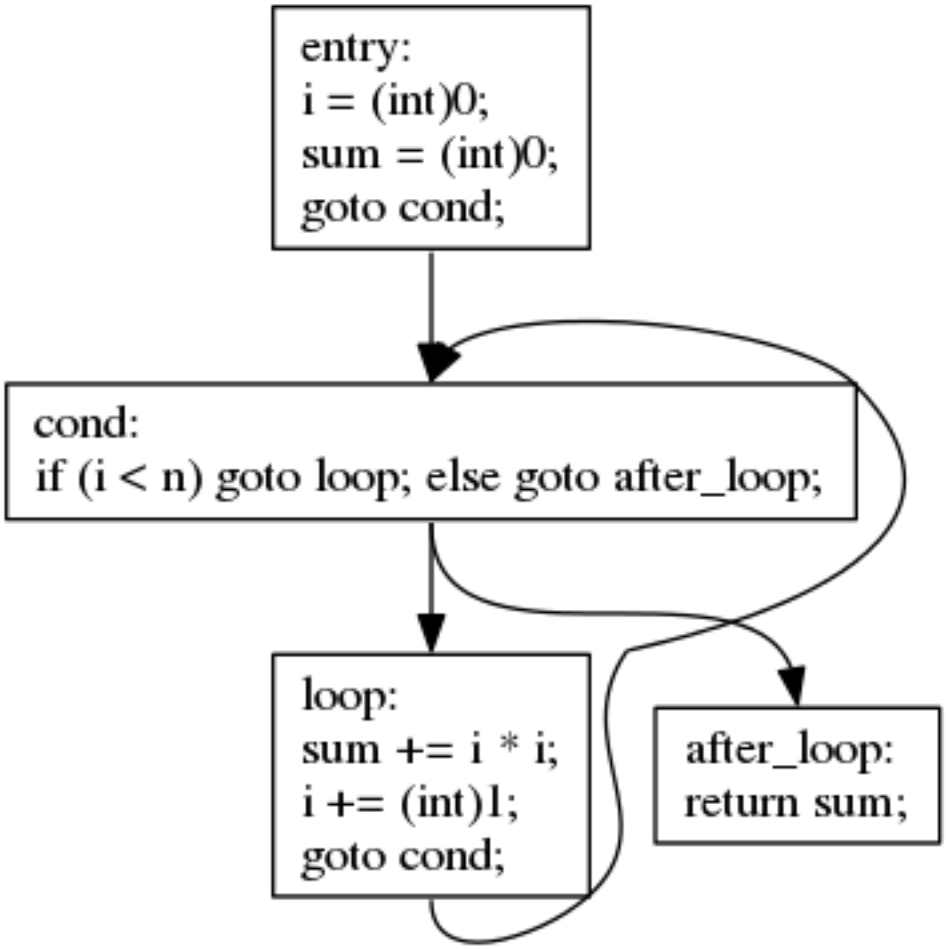
```
int loop_test (int n)
{
  int sum = 0;
  for (int i = 0; i < n; i++)
    sum += i * i;
  return sum;
}
```

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

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 include the libgccjit++ header and make a `gccjit::context`.

```
#include <libgccjit++.h>

void test (void)
```

(continues on next page)

(continued from previous page)

```
{
  gccjit::context ctxt;
  ctxt = gccjit::context::acquire ();
```

The function works with the C *int* type.

In the previous tutorial we acquired this via

```
gccjit::type the_type = ctxt.get_type (ctxt, GCC_JIT_TYPE_INT);
```

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

```
gccjit::type the_type = ctxt.get_type (ctxt, GCC_JIT_TYPE_DOUBLE);
```

For integer types we can use `gccjit::context::get_int_type` to directly bind a specific type:

```
gccjit::type the_type = ctxt.get_int_type <int> ();
```

Let's build the function:

```
gcc_jit_param n = ctxt.new_param (the_type, "n");
std::vector<gccjit::param> params;
params.push_back (n);
gccjit::function func =
  ctxt.new_function (GCC_JIT_FUNCTION_EXPORTED,
                     return_type,
                     "loop_test",
                     params, 0);
```

Expressions: lvalues and rvalues

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`.

Another 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 from the previous tutorial:

1. the multiplication *i * i*:

```
gccjit::rvalue expr =
  ctxt.new_binary_op (
    GCC_JIT_BINARY_OP_MULT, int_type,
    param_i, param_i);

/* Alternatively, using operator-overloading: */
gccjit::rvalue expr = param_i * param_i;
```

   which is a `gccjit::rvalue`, and

2. 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 new kind of expression: we have two local variables. We create them by calling `gccjit::function::new_local()`, supplying a type and a name:

```
/* Build locals:  */
gccjit::lvalue i = func.new_local (the_type, "i");
gccjit::lvalue sum = func.new_local (the_type, "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.

Control flow

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`:

```
gccjit::block b_initial = func.new_block ("initial");
gccjit::block b_loop_cond = func.new_block ("loop_cond");
gccjit::block b_loop_body = func.new_block ("loop_body");
gccjit::block b_after_loop = func.new_block ("after_loop");
```

We now populate each block with statements.

The entry block *b_initial* 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:

```
/* sum = 0; */
b_initial.add_assignment (sum, ctxt.zero (the_type));

/* i = 0; */
b_initial.add_assignment (i, ctxt.zero (the_type));
```

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

```
b_initial.end_with_jump (b_loop_cond);
```

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 could build the comparison using `gccjit::context::new_comparison()`:

```
gccjit::rvalue guard =
  ctxt.new_comparison (GCC_JIT_COMPARISON_GE,
                       i, n);
```

and can then use this to add *b_loop_cond*'s sole statement, via `gccjit::block::end_with_conditional()`:

```
b_loop_cond.end_with_conditional (guard,
                                  b_after_loop, // on_true
                                  b_loop_body); // on_false
```

However `gccjit::rvalue` has overloaded operators for this, so we express the conditional as

```
gccjit::rvalue guard = (i >= n);
```

and hence we can write the block more concisely as:

```
b_loop_cond.end_with_conditional (
  i >= n,
  b_after_loop, // on_true
  b_loop_body); // 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:

```
/* sum += i * i */
b_loop_body.add_assignment_op (sum,
                               GCC_JIT_BINARY_OP_PLUS,
                               i * i);
```

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

```
/* i++ */
b_loop_body.add_assignment_op (i,
                               GCC_JIT_BINARY_OP_PLUS,
                               ctxt.one (the_type));
```

Note: For numeric constants other than 0 or 1, we could use `gccjit::context::new_rvalue()`, which has overloads for both `int` and `double`.

The loop body completes by jumping back to the conditional:

```
b_loop_body.end_with_jump (b_loop_cond);
```

Finally, we populate the *b_after_loop* block, reached when the loop conditional is false. We want to generate the equivalent of:

```
return sum;
```

so the block is just one statement:

```
/* return sum */
b_after_loop.end_with_return (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:

```
gcc_jit_result *result;
result = ctxt.compile ();

ctxt.release ();

if (!result)
  {
    fprintf (stderr, "NULL result");
    return 1;
  }

typedef int (*loop_test_fn_type) (int);
loop_test_fn_type loop_test =
 (loop_test_fn_type)gcc_jit_result_get_code (result, "loop_test");
if (!loop_test)
  {
```

(continues on next page)

(continued from previous page)

```
    fprintf (stderr, "NULL loop_test");
    gcc_jit_result_release (result);
    return 1;
  }
printf ("result: %d", loop_test (10));
```

```
result: 285
```

Visualizing the control flow graph

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

```
func.dump_to_dot ("/tmp/sum-of-squares.dot");
```

giving a .dot file in GraphViz format.

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

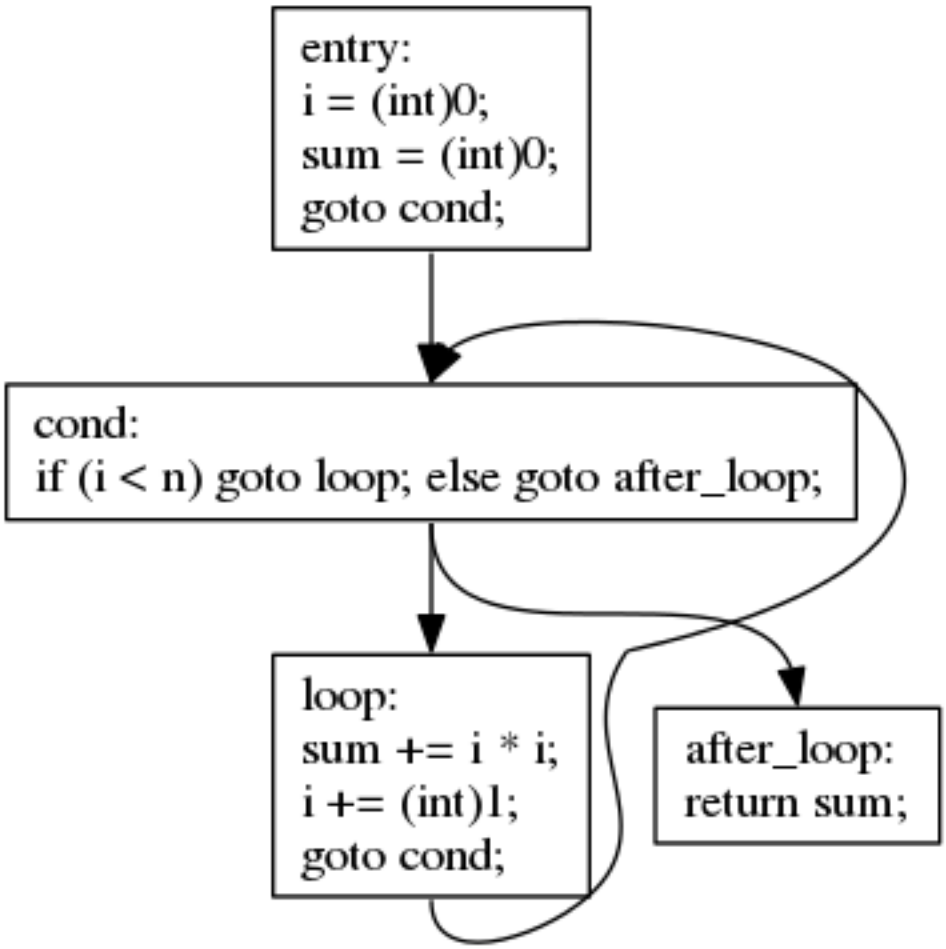
```
$ dot -Tpng /tmp/sum-of-squares.dot -o /tmp/sum-of-squares.png
```

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*):

Full example

```
/* Usage example for libgccjit.so's C++ API
   Copyright (C) 2014-2022 Free Software Foundation, Inc.

This file is part of GCC.

GCC is free software; you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 3, or (at your option)
any later version.

GCC is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with GCC; see the file COPYING3.  If not see
<http://www.gnu.org/licenses/>.  */

#include <libgccjit++.h>
```

(continues on next page)


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;

(continued from previous page)

```
#include <stdlib.h>
#include <stdio.h>

void
create_code (gccjit::context ctxt)
{
  /*
    Simple sum-of-squares, to test conditionals and looping

    int loop_test (int n)
    {
      int i;
      int sum = 0;
      for (i = 0; i < n ; i ++)
      {
        sum += i * i;
      }
      return sum;
   */
  gccjit::type the_type = ctxt.get_int_type <int> ();
  gccjit::type return_type = the_type;

  gccjit::param n = ctxt.new_param (the_type, "n");
  std::vector<gccjit::param> params;
  params.push_back (n);
  gccjit::function func =
    ctxt.new_function (GCC_JIT_FUNCTION_EXPORTED,
                       return_type,
                       "loop_test",
                       params, 0);

  /* Build locals:  */
  gccjit::lvalue i = func.new_local (the_type, "i");
  gccjit::lvalue sum = func.new_local (the_type, "sum");

  gccjit::block b_initial = func.new_block ("initial");
  gccjit::block b_loop_cond = func.new_block ("loop_cond");
  gccjit::block b_loop_body = func.new_block ("loop_body");
  gccjit::block b_after_loop = func.new_block ("after_loop");

  /* sum = 0; */
  b_initial.add_assignment (sum, ctxt.zero (the_type));

  /* i = 0; */
  b_initial.add_assignment (i, ctxt.zero (the_type));

  b_initial.end_with_jump (b_loop_cond);

  /* if (i >= n) */
  b_loop_cond.end_with_conditional (
    i >= n,
```

(continues on next page)

(continued from previous page)

```
    b_after_loop,
    b_loop_body);

  /* sum += i * i */
  b_loop_body.add_assignment_op (sum,
                                 GCC_JIT_BINARY_OP_PLUS,
                                 i * i);

  /* i++ */
  b_loop_body.add_assignment_op (i,
                                 GCC_JIT_BINARY_OP_PLUS,
                                 ctxt.one (the_type));

  b_loop_body.end_with_jump (b_loop_cond);

  /* return sum */
  b_after_loop.end_with_return (sum);
}

int
main (int argc, char **argv)
{
  gccjit::context ctxt;
  gcc_jit_result *result = NULL;

  /* Get a "context" object for working with the library.  */
  ctxt = gccjit::context::acquire ();

  /* Set some options on the context.
     Turn this on to see the code being generated, in assembler form.  */
  ctxt.set_bool_option (GCC_JIT_BOOL_OPTION_DUMP_GENERATED_CODE,
                        0);

  /* Populate the context.  */
  create_code (ctxt);

  /* Compile the code.  */
  result = ctxt.compile ();

  ctxt.release ();

  if (!result)
    {
      fprintf (stderr, "NULL result");
      return 1;
    }

  /* Extract the generated code from "result".  */
  typedef int (*loop_test_fn_type) (int);
  loop_test_fn_type loop_test =
    (loop_test_fn_type)gcc_jit_result_get_code (result, "loop_test");
```

(continues on next page)

(continued from previous page)

```
  if (!loop_test)
    {
      fprintf (stderr, "NULL loop_test");
      gcc_jit_result_release (result);
      return 1;
    }

  /* Run the generated code.  */
  int val = loop_test (10);
  printf("loop_test returned: %d\n", val);

  gcc_jit_result_release (result);
  return 0;
}
```

Building and running it:

```
$ gcc \
    tut03-sum-of-squares.cc \
    -o tut03-sum-of-squares \
    -lgccjit

# Run the built program:
$ ./tut03-sum-of-squares
loop_test returned: 285
```

3.1.4 Tutorial part 4: Adding JIT-compilation to a toy interpreter

In this example we construct a "toy" interpreter, and add JIT-compilation to it.

Our toy interpreter

It's a stack-based interpreter, and is intended as a (very simple) example of the kind of bytecode interpreter seen in dynamic languages such as Python, Ruby etc.

For the sake of simplicity, our toy virtual machine is very limited:

- The only data type is *int*
- It can only work on one function at a time (so that the only function call that can be made is to recurse).
- Functions can only take one parameter.
- Functions have a stack of *int* values.
- We'll implement function call within the interpreter by calling a function in our implementation, rather than implementing our own frame stack.
- The parser is only good enough to get the examples to work.

Naturally, a real interpreter would be much more complicated that this.

The following operations are supported:

Operation	Meaning	Old Stack	New Stack
DUP	Duplicate top of stack.	`[..., x]`	`[..., x, x]`
ROT	Swap top two elements of stack.	`[..., x, y]`	`[..., y, x]`
BINARY_ADD	Add the top two elements on the stack.	`[..., x, y]`	`[..., (x+y)]`
BINARY_SUBTRACT	Likewise, but subtract.	`[..., x, y]`	`[..., (x-y)]`
BINARY_MULT	Likewise, but multiply.	`[..., x, y]`	`[..., (x*y)]`
BINARY_COMPARE_LT	Compare the top two elements on the stack and push a nonzero/zero if (x<y).	`[..., x, y]`	`[..., (x<y)]`
RECURSE	Recurse, passing the top of the stack, and popping the result.	`[..., x]`	`[..., fn(x)]`
RETURN	Return the top of the stack.	`[x]`	`[]`
PUSH_CONST *arg*	Push an int const.	`[...]`	`[..., arg]`
JUMP_ABS_IF_TRUE *arg*	Pop; if top of stack was nonzero, jump to `arg`.	`[..., x]`	`[...]`

Programs can be interpreted, disassembled, and compiled to machine code.

The interpreter reads `.toy` scripts. Here's what a simple recursive factorial program looks like, the script `factorial.toy`. The parser ignores lines beginning with a #.

```
# Simple recursive factorial implementation, roughly equivalent to:
#
#  int factorial (int arg)
#  {
#    if (arg < 2)
#      return arg
#    return arg * factorial (arg - 1)
#  }

# Initial state:
# stack: [arg]

# 0:
DUP
# stack: [arg, arg]

# 1:
PUSH_CONST 2
# stack: [arg, arg, 2]

# 2:
BINARY_COMPARE_LT
# stack: [arg, (arg < 2)]
```

(continues on next page)

(continued from previous page)

```
# 3:
JUMP_ABS_IF_TRUE 9
# stack: [arg]

# 4:
DUP
# stack: [arg, arg]

# 5:
PUSH_CONST 1
# stack: [arg, arg, 1]

# 6:
BINARY_SUBTRACT
# stack: [arg,  (arg - 1)

# 7:
RECURSE
# stack: [arg, factorial(arg - 1)]

# 8:
BINARY_MULT
# stack: [arg * factorial(arg - 1)]

# 9:
RETURN
```

The interpreter is a simple infinite loop with a big `switch` statement based on what the next opcode is:

```
int
toyvm_function::interpret (int arg, FILE *trace)
{
  toyvm_frame frame;
#define PUSH(ARG) (frame.push (ARG))
#define POP(ARG) (frame.pop ())

  frame.frm_function = this;
  frame.frm_pc = 0;
  frame.frm_cur_depth = 0;

  PUSH (arg);

  while (1)
    {
      toyvm_op *op;
      int x, y;
      assert (frame.frm_pc < fn_num_ops);
      op = &fn_ops[frame.frm_pc++];
```

(continues on next page)

(continued from previous page)

```
      if (trace)
        {
          frame.dump_stack (trace);
          disassemble_op (op, frame.frm_pc, trace);
        }

      switch (op->op_opcode)
        {
          /* Ops taking no operand.  */
        case DUP:
          x = POP ();
          PUSH (x);
          PUSH (x);
          break;

        case ROT:
          y = POP ();
          x = POP ();
          PUSH (y);
          PUSH (x);
          break;

        case BINARY_ADD:
          y = POP ();
          x = POP ();
          PUSH (x + y);
          break;

        case BINARY_SUBTRACT:
          y = POP ();
          x = POP ();
          PUSH (x - y);
          break;

        case BINARY_MULT:
          y = POP ();
          x = POP ();
          PUSH (x * y);
          break;

        case BINARY_COMPARE_LT:
          y = POP ();
          x = POP ();
          PUSH (x < y);
          break;

        case RECURSE:
          x = POP ();
          x = interpret (x, trace);
          PUSH (x);
          break;
```

(continues on next page)

(continued from previous page)

```
        case RETURN:
          return POP ();

          /* Ops taking an operand.  */
        case PUSH_CONST:
          PUSH (op->op_operand);
          break;

        case JUMP_ABS_IF_TRUE:
          x = POP ();
          if (x)
            frame.frm_pc = op->op_operand;
          break;

        default:
          assert (0); /* unknown opcode */

        } /* end of switch on opcode */
    } /* end of while loop */

#undef PUSH
#undef POP
}
```

Compiling to machine code

We want to generate machine code that can be cast to this type and then directly executed in-process:

```
typedef int (*toyvm_compiled_func) (int);
```

Our compiler isn't very sophisticated; it takes the implementation of each opcode above, and maps it directly to the operations supported by the libgccjit API.

How should we handle the stack? In theory we could calculate what the stack depth will be at each opcode, and optimize away the stack manipulation "by hand". We'll see below that libgccjit is able to do this for us, so we'll implement stack manipulation in a direct way, by creating a `stack` array and `stack_depth` variables, local within the generated function, equivalent to this C code:

```
int stack_depth;
int stack[MAX_STACK_DEPTH];
```

We'll also have local variables `x` and `y` for use when implementing the opcodes, equivalent to this:

```
int x;
int y;
```

This means our compiler has the following state:

```
  toyvm_function &toyvmfn;

  gccjit::context ctxt;

  gccjit::type int_type;
  gccjit::type bool_type;
  gccjit::type stack_type; /* int[MAX_STACK_DEPTH] */

  gccjit::rvalue const_one;

  gccjit::function fn;
  gccjit::param param_arg;
  gccjit::lvalue stack;
  gccjit::lvalue stack_depth;
  gccjit::lvalue x;
  gccjit::lvalue y;

  gccjit::location op_locs[MAX_OPS];
  gccjit::block initial_block;
  gccjit::block op_blocks[MAX_OPS];
```

Setting things up

First we create our types:

```
void
compilation_state::create_types ()
{
  /* Create types.  */
  int_type = ctxt.get_type (GCC_JIT_TYPE_INT);
  bool_type = ctxt.get_type (GCC_JIT_TYPE_BOOL);
  stack_type = ctxt.new_array_type (int_type, MAX_STACK_DEPTH);
```

along with extracting a useful *int* constant:

```
  const_one = ctxt.one (int_type);

}
```

We'll implement push and pop in terms of the `stack` array and `stack_depth`. Here are helper functions for adding statements to a block, implementing pushing and popping values:

```
void
compilation_state::add_push (gccjit::block block,
                             gccjit::rvalue rvalue,
                             gccjit::location loc)
{
  /* stack[stack_depth] = RVALUE */
  block.add_assignment (
    /* stack[stack_depth] */
```

(continues on next page)

(continued from previous page)

```
    ctxt.new_array_access (
      stack,
      stack_depth,
      loc),
    rvalue,
    loc);

  /* "stack_depth++;".  */
  block.add_assignment_op (
    stack_depth,
    GCC_JIT_BINARY_OP_PLUS,
    const_one,
    loc);
}

void
compilation_state::add_pop (gccjit::block block,
                            gccjit::lvalue lvalue,
                            gccjit::location loc)
{
  /* "--stack_depth;".  */
  block.add_assignment_op (
    stack_depth,
    GCC_JIT_BINARY_OP_MINUS,
    const_one,
    loc);

  /* "LVALUE = stack[stack_depth];".  */
  block.add_assignment (
    lvalue,
    /* stack[stack_depth] */
    ctxt.new_array_access (stack,
                           stack_depth,
                           loc),
    loc);
}
```

We will support single-stepping through the generated code in the debugger, so we need to create `gccjit::location` instances, one per operation in the source code. These will reference the lines of e.g. `factorial.toy`.

```
void
compilation_state::create_locations ()
{
  for (int pc = 0; pc < toyvmfn.fn_num_ops; pc++)
    {
      toyvm_op *op = &toyvmfn.fn_ops[pc];

      op_locs[pc] = ctxt.new_location (toyvmfn.fn_filename,
                                       op->op_linenum,
                                       0); /* column */
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Let's create the function itself. As usual, we create its parameter first, then use the parameter to create the function:

```
void
compilation_state::create_function (const char *funcname)
{
  std::vector <gccjit::param> params;
  param_arg = ctxt.new_param (int_type, "arg", op_locs[0]);
  params.push_back (param_arg);
  fn = ctxt.new_function (GCC_JIT_FUNCTION_EXPORTED,
                          int_type,
                          funcname,
                          params, 0,
                          op_locs[0]);
```

We create the locals within the function.

```
  stack = fn.new_local (stack_type, "stack");
  stack_depth = fn.new_local (int_type, "stack_depth");
  x = fn.new_local (int_type, "x");
  y = fn.new_local (int_type, "y");
```

Populating the function

There's some one-time initialization, and the API treats the first block you create as the entrypoint of the function, so we need to create that block first:

```
  initial_block = fn.new_block ("initial");
```

We can now create blocks for each of the operations. Most of these will be consolidated into larger blocks when the optimizer runs.

```
  for (int pc = 0; pc < toyvmfn.fn_num_ops; pc++)
    {
      char buf[100];
      sprintf (buf, "instr%i", pc);
      op_blocks[pc] = fn.new_block (buf);
    }
```

Now that we have a block it can jump to when it's done, we can populate the initial block:

```
  /* "stack_depth = 0;".  */
  initial_block.add_assignment (stack_depth,
                                ctxt.zero (int_type),
                                op_locs[0]);
```

(continues on next page)

(continued from previous page)

```
/* "PUSH (arg);".  */
add_push (initial_block,
          param_arg,
          op_locs[0]);

/* ...and jump to insn 0.  */
initial_block.end_with_jump (op_blocks[0],
                             op_locs[0]);
```

We can now populate the blocks for the individual operations. We loop through them, adding instructions to their blocks:

```
for (int pc = 0; pc < toyvmfn.fn_num_ops; pc++)
  {
    gccjit::location loc = op_locs[pc];

    gccjit::block block = op_blocks[pc];
    gccjit::block next_block = (pc < toyvmfn.fn_num_ops
                                ? op_blocks[pc + 1]
                                : NULL);

    toyvm_op *op;
    op = &toyvmfn.fn_ops[pc];
```

We're going to have another big `switch` statement for implementing the opcodes, this time for compiling them, rather than interpreting them. It's helpful to have macros for implementing push and pop, so that we can make the `switch` statement that's coming up look as much as possible like the one above within the interpreter:

```
#define X_EQUALS_POP()\
      add_pop (block, x, loc)
#define Y_EQUALS_POP()\
      add_pop (block, y, loc)
#define PUSH_RVALUE(RVALUE)\
      add_push (block, (RVALUE), loc)
#define PUSH_X()\
      PUSH_RVALUE (x)
#define PUSH_Y() \
      PUSH_RVALUE (y)
```

Note: A particularly clever implementation would have an *identical* `switch` statement shared by the interpreter and the compiler, with some preprocessor "magic". We're not doing that here, for the sake of simplicity.

When I first implemented this compiler, I accidentally missed an edit when copying and pasting the `Y_EQUALS_POP` macro, so that popping the stack into `y` instead erroneously assigned it to `x`, leaving `y` uninitialized.

To track this kind of thing down, we can use `gccjit::block::add_comment()` to add descriptive

comments to the internal representation. This is invaluable when looking through the generated IR for, say `factorial`:

```
      block.add_comment (opcode_names[op->op_opcode], loc);
```

We can now write the big `switch` statement that implements the individual opcodes, populating the relevant block with statements:

```
      switch (op->op_opcode)
        {
        case DUP:
          X_EQUALS_POP ();
          PUSH_X ();
          PUSH_X ();
          break;

        case ROT:
          Y_EQUALS_POP ();
          X_EQUALS_POP ();
          PUSH_Y ();
          PUSH_X ();
          break;

        case BINARY_ADD:
          Y_EQUALS_POP ();
          X_EQUALS_POP ();
          PUSH_RVALUE (
           ctxt.new_binary_op (
             GC_JIT_BINARY_OP_PLUS,
             int_type,
             x, y,
             loc));
          break;

        case BINARY_SUBTRACT:
          Y_EQUALS_POP ();
          X_EQUALS_POP ();
          PUSH_RVALUE (
           ctxt.new_binary_op (
             GC_JIT_BINARY_OP_MINUS,
             int_type,
             x, y,
             loc));
          break;

        case BINARY_MULT:
          Y_EQUALS_POP ();
          X_EQUALS_POP ();
          PUSH_RVALUE (
           ctxt.new_binary_op (
             GC_JIT_BINARY_OP_MULT,
             int_type,
             x, y,
```

(continues on next page)

(continued from previous page)

```
            loc));
          break;

        case BINARY_COMPARE_LT:
          Y_EQUALS_POP ();
          X_EQUALS_POP ();
          PUSH_RVALUE (
            /* cast of bool to int */
            ctxt.new_cast (
              /* (x < y) as a bool */
              ctxt.new_comparison (
                GCC_JIT_COMPARISON_LT,
                x, y,
                loc),
              int_type,
              loc));
          break;

        case RECURSE:
          {
            X_EQUALS_POP ();
            PUSH_RVALUE (
              ctxt.new_call (
                fn,
                x,
                loc));
            break;
          }

        case RETURN:
          X_EQUALS_POP ();
          block.end_with_return (x, loc);
          break;

          /* Ops taking an operand.  */
        case PUSH_CONST:
          PUSH_RVALUE (
            ctxt.new_rvalue (int_type, op->op_operand));
          break;

        case JUMP_ABS_IF_TRUE:
          X_EQUALS_POP ();
          block.end_with_conditional (
            /* "(bool)x".  */
            ctxt.new_cast (x, bool_type, loc),
            op_blocks[op->op_operand], /* on_true */
            next_block, /* on_false */
            loc);
          break;

        default:
```

(continues on next page)

(continued from previous page)

```
        assert(0);
      } /* end of switch on opcode */
```

Every block must be terminated, via a call to one of the `gccjit::block::end_with_` entrypoints. This has been done for two of the opcodes, but we need to do it for the other ones, by jumping to the next block.

```
    if (op->op_opcode != JUMP_ABS_IF_TRUE
        && op->op_opcode != RETURN)
      block.end_with_jump (next_block, loc);
```

This is analogous to simply incrementing the program counter.

Verifying the control flow graph

Having finished looping over the blocks, the context is complete.

As before, we can verify that the control flow and statements are sane by using `gccjit::function::dump_to_dot()`:

```
fn.dump_to_dot ("/tmp/factorial.dot");
```

and viewing the result. Note how the label names, comments, and variable names show up in the dump, to make it easier to spot errors in our compiler.

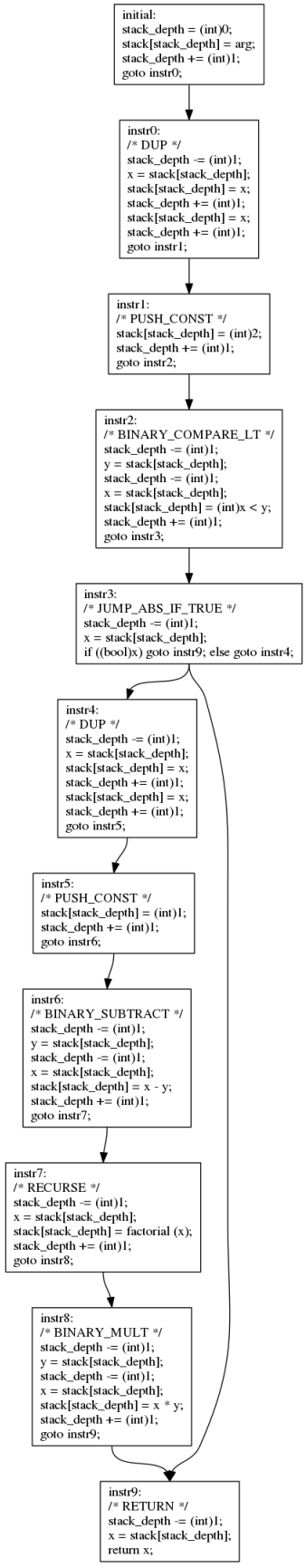
Compiling the context

Having finished looping over the blocks and populating them with statements, the context is complete.

We can now compile it, extract machine code from the result, and run it:

```
class compilation_result
{
public:
  compilation_result (gcc_jit_result *result) :
    m_result (result)
  {
  }
  ~compilation_result ()
  {
    gcc_jit_result_release (m_result);
  }

  void *get_code (const char *funcname)
  {
    return gcc_jit_result_get_code (m_result, funcname);
```

(continues on next page)


initial:
stack_depth = (int)0;
stack[stack_depth] = arg;
stack_depth += (int)1;
goto instr0;
instr0:
/* DUP */
stack_depth -= (int)1;
x = stack[stack_depth];
stack[stack_depth] = x;
stack_depth += (int)1;
stack[stack_depth] = x;
stack_depth += (int)1;
goto instr1;
instr1:
/* PUSH_CONST */
stack[stack_depth] = (int)2;
stack_depth += (int)1;
goto instr2;
instr2:
/* BINARY_COMPARE_LT */
stack_depth -= (int)1;
y = stack[stack_depth];
stack_depth -= (int)1;
x = stack[stack_depth];
stack[stack_depth] = (int)x < y;
stack_depth += (int)1;
goto instr3;
instr3:
/* JUMP_ABS_IF_TRUE */
stack_depth -= (int)1;
x = stack[stack_depth];
if ((bool)x) goto instr9; else goto instr4;
instr4:
/* DUP */
stack_depth -= (int)1;
x = stack[stack_depth];
stack[stack_depth] = x;
stack_depth += (int)1;
stack[stack_depth] = x;
stack_depth += (int)1;
goto instr5;
instr5:
/* PUSH_CONST */
stack[stack_depth] = (int)1;
stack_depth += (int)1;
goto instr6;
instr6:
/* BINARY_SUBTRACT */
stack_depth -= (int)1;
y = stack[stack_depth];
stack_depth -= (int)1;
x = stack[stack_depth];
stack[stack_depth] = x - y;
stack_depth += (int)1;
goto instr7;
instr7:
/* RECURSE */
stack_depth -= (int)1;
x = stack[stack_depth];
stack[stack_depth] = factorial (x);
stack_depth += (int)1;
goto instr8;
instr8:
/* BINARY_MULT */
stack_depth -= (int)1;
y = stack[stack_depth];
stack_depth -= (int)1;
x = stack[stack_depth];
stack[stack_depth] = x * y;
stack_depth += (int)1;
goto instr9;
instr9:
/* RETURN */
stack_depth -= (int)1;
x = stack[stack_depth];
return x;

(continued from previous page)

```
  }

private:
  gcc_jit_result *m_result;
};
```

```
  compilation_result compiler_result = fn->compile ();

  const char *funcname = fn->get_function_name ();
  toyvm_compiled_func code
    = (toyvm_compiled_func)compiler_result.get_code (funcname);

  printf ("compiler result: %d\n",
          code (atoi (argv[2])));
```

Single-stepping through the generated code

It's possible to debug the generated code. To do this we need to both:

- Set up source code locations for our statements, so that we can meaningfully step through the code. We did this above by calling `gccjit::context::new_location()` and using the results.
- Enable the generation of debugging information, by setting `GCC_JIT_BOOL_OPTION_DEBUGINFO` on the `gccjit::context` via `gccjit::context::set_bool_option()`:

```
ctxt.set_bool_option (GCC_JIT_BOOL_OPTION_DEBUGINFO, 1);
```

Having done this, we can put a breakpoint on the generated function:

```
$ gdb --args ./toyvm factorial.toy 10
(gdb) break factorial
Function "factorial" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (factorial) pending.
(gdb) run
Breakpoint 1, factorial (arg=10) at factorial.toy:14
14    DUP
```

We've set up location information, which references `factorial.toy`. This allows us to use e.g. `list` to see where we are in the script:

```
(gdb) list
9
10    # Initial state:
11    # stack: [arg]
12
13    # 0:
14    DUP
15    # stack: [arg, arg]
```

(continues on next page)

(continued from previous page)

```
16
17    # 1:
18    PUSH_CONST 2
```

and to step through the function, examining the data:

```
(gdb) n
18    PUSH_CONST 2
(gdb) n
22    BINARY_COMPARE_LT
(gdb) print stack
$5 = {10, 10, 2, 0, -7152, 32767, 0, 0}
(gdb) print stack_depth
$6 = 3
```

You'll see that the parts of the `stack` array that haven't been touched yet are uninitialized.

Note: Turning on optimizations may lead to unpredictable results when stepping through the generated code: the execution may appear to "jump around" the source code. This is analogous to turning up the optimization level in a regular compiler.

Examining the generated code

How good is the optimized code?

We can turn up optimizations, by calling `gccjit::context::set_int_option()` with `GCC_JIT_INT_OPTION_OPTIMIZATION_LEVEL`:

```
ctxt.set_int_option (GCC_JIT_INT_OPTION_OPTIMIZATION_LEVEL, 3);
```

One of GCC's internal representations is called "gimple". A dump of the initial gimple representation of the code can be seen by setting:

```
ctxt.set_bool_option (GCC_JIT_BOOL_OPTION_DUMP_INITIAL_GIMPLE, 1);
```

With optimization on and source locations displayed, this gives:

```
factorial (signed int arg)
{
  <unnamed type> D.80;
  signed int D.81;
  signed int D.82;
  signed int D.83;
  signed int D.84;
  signed int D.85;
  signed int y;
  signed int x;
  signed int stack_depth;
```

(continues on next page)

(continued from previous page)

```
  signed int stack[8];

  try
    {
      initial:
      stack_depth = 0;
      stack[stack_depth] = arg;
      stack_depth = stack_depth + 1;
      goto instr0;
      instr0:
      /* DUP */:
      stack_depth = stack_depth + -1;
      x = stack[stack_depth];
      stack[stack_depth] = x;
      stack_depth = stack_depth + 1;
      stack[stack_depth] = x;
      stack_depth = stack_depth + 1;
      goto instr1;
      instr1:
      /* PUSH_CONST */:
      stack[stack_depth] = 2;
      stack_depth = stack_depth + 1;
      goto instr2;

      /* etc */
```

You can see the generated machine code in assembly form via:

```
ctxt.set_bool_option (GCC_JIT_BOOL_OPTION_DUMP_GENERATED_CODE, 1);
result = ctxt.compile ();
```

which shows that (on this x86_64 box) the compiler has unrolled the loop and is using MMX instructions to perform several multiplications simultaneously:

```
        .file   "fake.c"
        .text
.Ltext0:
        .p2align 4,,15
        .globl  factorial
        .type   factorial, @function
factorial:
.LFB0:
        .file 1 "factorial.toy"
        .loc 1 14 0
        .cfi_startproc
.LVL0:
.L2:
        .loc 1 26 0
        cmpl    $1, %edi
        jle     .L13
        leal    -1(%rdi), %edx
```

(continues on next page)

(continued from previous page)

```
        movl    %edx, %ecx
        shrl    $2, %ecx
        leal    0(,%rcx,4), %esi
        testl   %esi, %esi
        je      .L14
        cmpl    $9, %edx
        jbe     .L14
        leal    -2(%rdi), %eax
        movl    %eax, -16(%rsp)
        leal    -3(%rdi), %eax
        movd    -16(%rsp), %xmm0
        movl    %edi, -16(%rsp)
        movl    %eax, -12(%rsp)
        movd    -16(%rsp), %xmm1
        xorl    %eax, %eax
        movl    %edx, -16(%rsp)
        movd    -12(%rsp), %xmm4
        movd    -16(%rsp), %xmm6
        punpckldq       %xmm4, %xmm0
        movdqa  .LC1(%rip), %xmm4
        punpckldq       %xmm6, %xmm1
        punpcklqdq      %xmm0, %xmm1
        movdqa  .LC0(%rip), %xmm0
        jmp     .L5
        # etc - edited for brevity
```

This is clearly overkill for a function that will likely overflow the `int` type before the vectorization is worthwhile - but then again, this is a toy example.

Turning down the optimization level to 2:

```
ctxt.set_int_option (GCC_JIT_INT_OPTION_OPTIMIZATION_LEVEL, 2);
```

yields this code, which is simple enough to quote in its entirety:

```
        .file   "fake.c"
        .text
        .p2align 4,,15
        .globl  factorial
        .type   factorial, @function
factorial:
.LFB0:
        .cfi_startproc
.L2:
        cmpl    $1, %edi
        jle     .L8
        movl    $1, %edx
        jmp     .L4
        .p2align 4,,10
        .p2align 3
.L6:
```

(continues on next page)

(continued from previous page)

```
        movl    %eax, %edi
.L4:
.L5:
        leal    -1(%rdi), %eax
        imull   %edi, %edx
        cmpl    $1, %eax
        jne     .L6
.L3:
.L7:
        imull   %edx, %eax
        ret
.L8:
        movl    %edi, %eax
        movl    $1, %edx
        jmp     .L7
        .cfi_endproc
.LFE0:
        .size   factorial, .-factorial
        .ident  "GCC: (GNU) 4.9.0 20131023 (Red Hat 0.2)"
        .section        .note.GNU-stack,"",@progbits
```

Note that the stack pushing and popping have been eliminated, as has the recursive call (in favor of an iteration).

Putting it all together

The complete example can be seen in the source tree at `gcc/jit/docs/examples/tut04-toyvm/toyvm.cc`

along with a Makefile and a couple of sample .toy scripts:

```
$ ls -al
drwxrwxr-x. 2 david david   4096 Sep 19 17:46 .
drwxrwxr-x. 3 david david   4096 Sep 19 15:26 ..
-rw-rw-r--. 1 david david    615 Sep 19 12:43 factorial.toy
-rw-rw-r--. 1 david david    834 Sep 19 13:08 fibonacci.toy
-rw-rw-r--. 1 david david    238 Sep 19 14:22 Makefile
-rw-rw-r--. 1 david david  16457 Sep 19 17:07 toyvm.cc

$ make toyvm
g++ -Wall -g -o toyvm toyvm.cc -lgccjit

$ ./toyvm factorial.toy 10
interpreter result: 3628800
compiler result: 3628800

$ ./toyvm fibonacci.toy 10
interpreter result: 55
compiler result: 55
```

Behind the curtain: How does our code get optimized?

Our example is done, but you may be wondering about exactly how the compiler turned what we gave it into the machine code seen above.

We can examine what the compiler is doing in detail by setting:

```
state.ctxt.set_bool_option (GCC_JIT_BOOL_OPTION_DUMP_EVERYTHING, 1);
state.ctxt.set_bool_option (GCC_JIT_BOOL_OPTION_KEEP_INTERMEDIATES, 1);
```

This will dump detailed information about the compiler's state to a directory under `/tmp`, and keep it from being cleaned up.

The precise names and their formats of these files is subject to change. Higher optimization levels lead to more files. Here's what I saw (edited for brevity; there were almost 200 files):

```
intermediate files written to /tmp/libgccjit-KPQbGw
$ ls /tmp/libgccjit-KPQbGw/
fake.c.000i.cgraph
fake.c.000i.type-inheritance
fake.c.004t.gimple
fake.c.007t.omplower
fake.c.008t.lower
fake.c.011t.eh
fake.c.012t.cfg
fake.c.014i.visibility
fake.c.015i.early_local_cleanups
fake.c.016t.ssa
# etc
```

The gimple code is converted into Static Single Assignment form, with annotations for use when generating the debuginfo:

```
$ less /tmp/libgccjit-KPQbGw/fake.c.016t.ssa
```

```
;; Function factorial (factorial, funcdef_no=0, decl_uid=53, symbol_order=0)

factorial (signed int arg)
{
  signed int stack[8];
  signed int stack_depth;
  signed int x;
  signed int y;
  <unnamed type> _20;
  signed int _21;
  signed int _38;
  signed int _44;
  signed int _51;
  signed int _56;

initial:
  stack_depth_3 = 0;
```

(continues on next page)

(continued from previous page)

```
  # DEBUG stack_depth => stack_depth_3
  stack[stack_depth_3] = arg_5(D);
  stack_depth_7 = stack_depth_3 + 1;
  # DEBUG stack_depth => stack_depth_7
  # DEBUG instr0 => NULL
  # DEBUG /* DUP */ => NULL
  stack_depth_8 = stack_depth_7 + -1;
  # DEBUG stack_depth => stack_depth_8
  x_9 = stack[stack_depth_8];
  # DEBUG x => x_9
  stack[stack_depth_8] = x_9;
  stack_depth_11 = stack_depth_8 + 1;
  # DEBUG stack_depth => stack_depth_11
  stack[stack_depth_11] = x_9;
  stack_depth_13 = stack_depth_11 + 1;
  # DEBUG stack_depth => stack_depth_13
  # DEBUG instr1 => NULL
  # DEBUG /* PUSH_CONST */ => NULL
  stack[stack_depth_13] = 2;

  /* etc; edited for brevity */
```

We can perhaps better see the code by turning off `GCC_JIT_BOOL_OPTION_DEBUGINFO` to suppress all those `DEBUG` statements, giving:

```
$ less /tmp/libgccjit-1Hywc0/fake.c.016t.ssa
```

```
;; Function factorial (factorial, funcdef_no=0, decl_uid=53, symbol_order=0)

factorial (signed int arg)
{
  signed int stack[8];
  signed int stack_depth;
  signed int x;
  signed int y;
  <unnamed type> _20;
  signed int _21;
  signed int _38;
  signed int _44;
  signed int _51;
  signed int _56;

initial:
  stack_depth_3 = 0;
  stack[stack_depth_3] = arg_5(D);
  stack_depth_7 = stack_depth_3 + 1;
  stack_depth_8 = stack_depth_7 + -1;
  x_9 = stack[stack_depth_8];
  stack[stack_depth_8] = x_9;
  stack_depth_11 = stack_depth_8 + 1;
  stack[stack_depth_11] = x_9;
```

(continues on next page)

(continued from previous page)

```
  stack_depth_13 = stack_depth_11 + 1;
  stack[stack_depth_13] = 2;
  stack_depth_15 = stack_depth_13 + 1;
  stack_depth_16 = stack_depth_15 + -1;
  y_17 = stack[stack_depth_16];
  stack_depth_18 = stack_depth_16 + -1;
  x_19 = stack[stack_depth_18];
  _20 = x_19 < y_17;
  _21 = (signed int) _20;
  stack[stack_depth_18] = _21;
  stack_depth_23 = stack_depth_18 + 1;
  stack_depth_24 = stack_depth_23 + -1;
  x_25 = stack[stack_depth_24];
  if (x_25 != 0)
    goto <bb 4> (instr9);
  else
    goto <bb 3> (instr4);

instr4:
/* DUP */:
  stack_depth_26 = stack_depth_24 + -1;
  x_27 = stack[stack_depth_26];
  stack[stack_depth_26] = x_27;
  stack_depth_29 = stack_depth_26 + 1;
  stack[stack_depth_29] = x_27;
  stack_depth_31 = stack_depth_29 + 1;
  stack[stack_depth_31] = 1;
  stack_depth_33 = stack_depth_31 + 1;
  stack_depth_34 = stack_depth_33 + -1;
  y_35 = stack[stack_depth_34];
  stack_depth_36 = stack_depth_34 + -1;
  x_37 = stack[stack_depth_36];
  _38 = x_37 - y_35;
  stack[stack_depth_36] = _38;
  stack_depth_40 = stack_depth_36 + 1;
  stack_depth_41 = stack_depth_40 + -1;
  x_42 = stack[stack_depth_41];
  _44 = factorial (x_42);
  stack[stack_depth_41] = _44;
  stack_depth_46 = stack_depth_41 + 1;
  stack_depth_47 = stack_depth_46 + -1;
  y_48 = stack[stack_depth_47];
  stack_depth_49 = stack_depth_47 + -1;
  x_50 = stack[stack_depth_49];
  _51 = x_50 * y_48;
  stack[stack_depth_49] = _51;
  stack_depth_53 = stack_depth_49 + 1;

  # stack_depth_1 = PHI <stack_depth_24(2), stack_depth_53(3)>
instr9:
/* RETURN */:
```

(continues on next page)

(continued from previous page)

```
  stack_depth_54 = stack_depth_1 + -1;
  x_55 = stack[stack_depth_54];
  _56 = x_55;
  stack ={v} {CLOBBER};
  return _56;

}
```

Note in the above how all the `gccjit::block` instances we created have been consolidated into just 3 blocks in GCC's internal representation: `initial`, `instr4` and `instr9`.

Optimizing away stack manipulation

Recall our simple implementation of stack operations. Let's examine how the stack operations are optimized away.

After a pass of constant-propagation, the depth of the stack at each opcode can be determined at compile-time:

```
$ less /tmp/libgccjit-1Hywc0/fake.c.021t.ccp1
```

```
;; Function factorial (factorial, funcdef_no=0, decl_uid=53, symbol_order=0)

factorial (signed int arg)
{
  signed int stack[8];
  signed int stack_depth;
  signed int x;
  signed int y;
  <unnamed type> _20;
  signed int _21;
  signed int _38;
  signed int _44;
  signed int _51;

initial:
  stack[0] = arg_5(D);
  x_9 = stack[0];
  stack[0] = x_9;
  stack[1] = x_9;
  stack[2] = 2;
  y_17 = stack[2];
  x_19 = stack[1];
  _20 = x_19 < y_17;
  _21 = (signed int) _20;
  stack[1] = _21;
  x_25 = stack[1];
  if (x_25 != 0)
    goto <bb 4> (instr9);
```

(continues on next page)

(continued from previous page)

```
  else
    goto <bb 3> (instr4);

instr4:
/* DUP */:
  x_27 = stack[0];
  stack[0] = x_27;
  stack[1] = x_27;
  stack[2] = 1;
  y_35 = stack[2];
  x_37 = stack[1];
  _38 = x_37 - y_35;
  stack[1] = _38;
  x_42 = stack[1];
  _44 = factorial (x_42);
  stack[1] = _44;
  y_48 = stack[1];
  x_50 = stack[0];
  _51 = x_50 * y_48;
  stack[0] = _51;

instr9:
/* RETURN */:
  x_55 = stack[0];
  x_56 = x_55;
  stack ={v} {CLOBBER};
  return x_56;

}
```

Note how, in the above, all those `stack_depth` values are now just constants: we're accessing specific stack locations at each opcode.

The "esra" pass ("Early Scalar Replacement of Aggregates") breaks out our "stack" array into individual elements:

```
$ less /tmp/libgccjit-1Hywc0/fake.c.024t.esra
```

```
;; Function factorial (factorial, funcdef_no=0, decl_uid=53, symbol_order=0)

Created a replacement for stack offset: 0, size: 32: stack$0
Created a replacement for stack offset: 32, size: 32: stack$1
Created a replacement for stack offset: 64, size: 32: stack$2

Symbols to be put in SSA form
{ D.89 D.90 D.91 }
Incremental SSA update started at block: 0
Number of blocks in CFG: 5
Number of blocks to update: 4 ( 80%)
```

(continues on next page)

(continued from previous page)

```
factorial (signed int arg)
{
  signed int stack$2;
  signed int stack$1;
  signed int stack$0;
  signed int stack[8];
  signed int stack_depth;
  signed int x;
  signed int y;
  <unnamed type> _20;
  signed int _21;
  signed int _38;
  signed int _44;
  signed int _51;

initial:
  stack$0_45 = arg_5(D);
  x_9 = stack$0_45;
  stack$0_39 = x_9;
  stack$1_32 = x_9;
  stack$2_30 = 2;
  y_17 = stack$2_30;
  x_19 = stack$1_32;
  _20 = x_19 < y_17;
  _21 = (signed int) _20;
  stack$1_28 = _21;
  x_25 = stack$1_28;
  if (x_25 != 0)
    goto <bb 4> (instr9);
  else
    goto <bb 3> (instr4);

instr4:
/* DUP */:
  x_27 = stack$0_39;
  stack$0_22 = x_27;
  stack$1_14 = x_27;
  stack$2_12 = 1;
  y_35 = stack$2_12;
  x_37 = stack$1_14;
  _38 = x_37 - y_35;
  stack$1_10 = _38;
  x_42 = stack$1_10;
  _44 = factorial (x_42);
  stack$1_6 = _44;
  y_48 = stack$1_6;
  x_50 = stack$0_22;
  _51 = x_50 * y_48;
  stack$0_1 = _51;

  # stack$0_52 = PHI <stack$0_39(2), stack$0_1(3)>
```

(continues on next page)

(continued from previous page)

```
instr9:
/* RETURN */:
  x_55 = stack$0_52;
  x_56 = x_55;
  stack ={v} {CLOBBER};
  return x_56;

}
```

Hence at this point, all those pushes and pops of the stack are now simply assignments to specific temporary variables.

After some copy propagation, the stack manipulation has been completely optimized away:

```
$ less /tmp/libgccjit-1Hywc0/fake.c.026t.copyprop1
```

```
;; Function factorial (factorial, funcdef_no=0, decl_uid=53, symbol_order=0)

factorial (signed int arg)
{
  signed int stack$2;
  signed int stack$1;
  signed int stack$0;
  signed int stack[8];
  signed int stack_depth;
  signed int x;
  signed int y;
  <unnamed type> _20;
  signed int _21;
  signed int _38;
  signed int _44;
  signed int _51;

initial:
  stack$0_39 = arg_5(D);
  _20 = arg_5(D) <= 1;
  _21 = (signed int) _20;
  if (_21 != 0)
    goto <bb 4> (instr9);
  else
    goto <bb 3> (instr4);

instr4:
/* DUP */:
  _38 = arg_5(D) + -1;
  _44 = factorial (_38);
  _51 = arg_5(D) * _44;
  stack$0_1 = _51;

  # stack$0_52 = PHI <arg_5(D)(2), _51(3)>
instr9:
```

(continues on next page)

(continued from previous page)

```
/* RETURN */:
  stack ={v} {CLOBBER};
  return stack$0_52;

}
```

Later on, another pass finally eliminated `stack_depth` local and the unused parts of the *stack*' array altogether:

```
$ less /tmp/libgccjit-1Hywc0/fake.c.036t.release_ssa
```

```
;; Function factorial (factorial, funcdef_no=0, decl_uid=53, symbol_order=0)

Released 44 names, 314.29%, removed 44 holes
factorial (signed int arg)
{
  signed int stack$0;
  signed int mult_acc_1;
  <unnamed type> _5;
  signed int _6;
  signed int _7;
  signed int mul_tmp_10;
  signed int mult_acc_11;
  signed int mult_acc_13;

  # arg_9 = PHI <arg_8(D)(0)>
  # mult_acc_13 = PHI <1(0)>
initial:

  <bb 5>:
  # arg_4 = PHI <arg_9(2), _7(3)>
  # mult_acc_1 = PHI <mult_acc_13(2), mult_acc_11(3)>
  _5 = arg_4 <= 1;
  _6 = (signed int) _5;
  if (_6 != 0)
    goto <bb 4> (instr9);
  else
    goto <bb 3> (instr4);

instr4:
/* DUP */:
  _7 = arg_4 + -1;
  mult_acc_11 = mult_acc_1 * arg_4;
  goto <bb 5>;

  # stack$0_12 = PHI <arg_4(5)>
instr9:
/* RETURN */:
  mul_tmp_10 = mult_acc_1 * stack$0_12;
  return mul_tmp_10;
```

(continues on next page)

(continued from previous page)

```
}
```

Elimination of tail recursion

Another significant optimization is the detection that the call to `factorial` is tail recursion, which can be eliminated in favor of an iteration:

```
$ less /tmp/libgccjit-1Hywc0/fake.c.030t.tailr1
```

```
;; Function factorial (factorial, funcdef_no=0, decl_uid=53, symbol_order=0)


Symbols to be put in SSA form
{ D.88 }
Incremental SSA update started at block: 0
Number of blocks in CFG: 5
Number of blocks to update: 4 ( 80%)


factorial (signed int arg)
{
  signed int stack$2;
  signed int stack$1;
  signed int stack$0;
  signed int stack[8];
  signed int stack_depth;
  signed int x;
  signed int y;
  signed int mult_acc_1;
  <unnamed type> _20;
  signed int _21;
  signed int _38;
  signed int mul_tmp_44;
  signed int mult_acc_51;

  # arg_5 = PHI <arg_39(D)(0), _38(3)>
  # mult_acc_1 = PHI <1(0), mult_acc_51(3)>
initial:
  _20 = arg_5 <= 1;
  _21 = (signed int) _20;
  if (_21 != 0)
    goto <bb 4> (instr9);
  else
    goto <bb 3> (instr4);

instr4:
/* DUP */:
  _38 = arg_5 + -1;
  mult_acc_51 = mult_acc_1 * arg_5;
```

(continues on next page)

(continued from previous page)

```
  goto <bb 2> (initial);

  # stack$0_52 = PHI <arg_5(2)>
instr9:
/* RETURN */:
  stack ={v} {CLOBBER};
  mul_tmp_44 = mult_acc_1 * stack$0_52;
  return mul_tmp_44;

}
```

3.2 Topic Reference

3.2.1 Compilation contexts

class `gccjit::context`

The top-level of the C++ API is the `gccjit::context` type.

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 `gcc_jit_result*`.

It is a thin wrapper around the C API's `gcc_jit_context*`.

Lifetime-management

Contexts are the unit of lifetime-management within the API: objects have their lifetime bounded by the context they are created within, and cleanup of such objects is done for you when the context is released.

gccjit::context `gccjit::context::acquire()`

This function acquires a new `gccjit::context` instance, which is independent of any others that may be present within this process.

void `gccjit::context::release()`

This function releases all resources associated with the given context. Both the context itself and all of its `gccjit::object*` instances are cleaned up. It should be called exactly once on a given context.

It is invalid to use the context or any of its "contextual" objects after calling this.

```
ctxt.release ();
```

gccjit::context `gccjit::context::new_child_context()`

Given an existing JIT context, create a child context.

The child inherits a copy of all option-settings from the parent.

The child can reference objects created within the parent, but not vice-versa.

The lifetime of the child context must be bounded by that of the parent: you should release a child context before releasing the parent context.

If you use a function from a parent context within a child context, you have to compile the parent context before you can compile the child context, and the gccjit::result of the parent context must outlive the gccjit::result of the child context.

This allows caching of shared initializations. For example, you could create types and declarations of global functions in a parent context once within a process, and then create child contexts whenever a function or loop becomes hot. Each such child context can be used for JIT-compiling just one function or loop, but can reference types and helper functions created within the parent context.

Contexts can be arbitrarily nested, provided the above rules are followed, but it's probably not worth going above 2 or 3 levels, and there will likely be a performance hit for such nesting.

Thread-safety

Instances of `gccjit::context` created via `gccjit::context::acquire()` are independent from each other: only one thread may use a given context at once, but multiple threads could each have their own contexts without needing locks.

Contexts created via `gccjit::context::new_child_context()` are related to their parent context. They can be partitioned by their ultimate ancestor into independent "family trees". Only one thread within a process may use a given "family tree" of such contexts at once, and if you're using multiple threads you should provide your own locking around entire such context partitions.

Error-handling

You can only compile and get code from a context if no errors occur.

In general, if an error occurs when using an API entrypoint, it returns NULL. You don't have to check everywhere for NULL results, since the API gracefully handles a NULL being passed in for any argument.

Errors are printed on stderr and can be queried using `gccjit::context::get_first_error()`.

const char *`gccjit::context::get_first_error`(gccjit::context *ctxt)

Returns the first error message that occurred on the context.

The returned string is valid for the rest of the lifetime of the context.

If no errors occurred, this will be NULL.

Debugging

void gccjit::context::**dump_to_file**(const std::string &path, int update_locations)

To help with debugging: dump a C-like representation to the given path, describing what's been set up on the context.

If "update_locations" is true, then also set up gccjit::location information throughout the context, pointing at the dump file as if it were a source file. This may be of use in conjunction with GCC_JIT_BOOL_OPTION_DEBUGINFO to allow stepping through the code in a debugger.

void gccjit::context::**dump_reproducer_to_file**(gcc_jit_context *ctxt, const char *path)

This is a thin wrapper around the C API gcc_jit_context_dump_reproducer_to_file(), and hence works the same way.

Note that the generated source is C code, not C++; this might be of use for seeing what the C++ bindings are doing at the C level.

Options

String Options

void gccjit::context::**set_str_option**(enum gcc_jit_str_option, const char *value)

Set a string option of the context.

This is a thin wrapper around the C API gcc_jit_context_set_str_option(); the options have the same meaning.

Boolean options

void gccjit::context::**set_bool_option**(enum gcc_jit_bool_option, int value)

Set a boolean option of the context.

This is a thin wrapper around the C API gcc_jit_context_set_bool_option(); the options have the same meaning.

void gccjit::context::**set_bool_allow_unreachable_blocks**(int bool_value)

By default, libgccjit will issue an error about unreachable blocks within a function.

This entrypoint can be used to disable that error; it is a thin wrapper around the C API gcc_jit_context_set_bool_allow_unreachable_blocks().

This entrypoint was added in LIBGCCJIT_ABI_2; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_gcc_jit_context_set_bool_allow_unreachable_blocks
```

void gccjit::context::**set_bool_use_external_driver**(int bool_value)

libgccjit internally generates assembler, and uses "driver" code for converting it to other formats (e.g. shared libraries).

By default, libgccjit will use an embedded copy of the driver code.

This option can be used to instead invoke an external driver executable as a subprocess; it is a thin wrapper around the C API `gcc_jit_context_set_bool_use_external_driver()`.

This entrypoint was added in LIBGCCJIT_ABI_5; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_gcc_jit_context_set_bool_use_external_driver
```

Integer options

void `gccjit::context::set_int_option`(enum gcc_jit_int_option, int value)

Set an integer option of the context.

This is a thin wrapper around the C API `gcc_jit_context_set_int_option()`; the options have the same meaning.

Additional command-line options

void `gccjit::context::add_command_line_option`(const char *optname)

Add an arbitrary gcc command-line option to the context for use when compiling.

This is a thin wrapper around the C API `gcc_jit_context_add_command_line_option()`.

This entrypoint was added in LIBGCCJIT_ABI_1; you can test for its presence using

```
#ifdef LIBGCCJIT_HAVE_gcc_jit_context_add_command_line_option
```

3.2.2 Objects

class `gccjit::object`

Almost every entity in the API (with the exception of `gccjit::context` and `gcc_jit_result*`) is a "contextual" object, a `gccjit::object`.

A JIT object:

- is associated with a `gccjit::context`.
- is automatically cleaned up for you when its context is released so you don't need to manually track and cleanup all objects, just the contexts.

The C++ class hierarchy within the `gccjit` namespace looks like this:

```
+- object
    +- location
    +- type
       +- struct
    +- field
```

(continues on next page)

(continued from previous page)

```
   +- function
   +- block
   +- rvalue
       +- lvalue
          +- param
   +- case_
```

The gccjit::object base class has the following operations:

gccjit::context gccjit::object::**get_context**() const

Which context is the obj within?

std::string gccjit::object::**get_debug_string**() const

Generate a human-readable description for the given object.

For example,

```
printf ("obj: %s\n", obj.get_debug_string ().c_str ());
```

might give this text on stdout:

```
obj: 4.0 * (float)i
```

3.2.3 Types

class gccjit::**type**

gccjit::type represents a type within the library. It is a subclass of gccjit::object.

Types can be created in several ways:

- fundamental types can be accessed using gccjit::context::get_type():

```
gccjit::type int_type = ctxt.get_type (GCC_JIT_TYPE_INT);
```

or using the gccjit::context::get_int_type template:

```
gccjit::type t = ctxt.get_int_type <unsigned short> ();
```

See gcc_jit_context_get_type() for the available types.

- derived types can be accessed by using functions such as gccjit::type::get_pointer() and gccjit::type::get_const():

```
gccjit::type const_int_star = int_type.get_const ().get_pointer ();
gccjit::type int_const_star = int_type.get_pointer ().get_const ();
```

- by creating structures (see below).

Standard types

gccjit::type **gccjit::context::get_type**(enum gcc_jit_types)

Access a specific type. This is a thin wrapper around `gcc_jit_context_get_type()`; the parameter has the same meaning.

gccjit::type **gccjit::context::get_int_type**(size_t num_bytes, int is_signed)

Access the integer type of the given size.

template<typename **T**>
gccjit::type **gccjit::context::get_int_type**<T>()

Access the given integer type. For example, you could map the `unsigned short` type into a gccjit::type via:

```
gccjit::type t = ctxt.get_int_type <unsigned short> ();
```

Pointers, *const*, and *volatile*

gccjit::type **gccjit::type::get_pointer**()

Given type "T", get type "T*".

gccjit::type **gccjit::type::get_const**()

Given type "T", get type "const T".

gccjit::type **gccjit::type::get_volatile**()

Given type "T", get type "volatile T".

gccjit::type **gccjit::type::get_aligned**(size_t alignment_in_bytes)

Given type "T", get type:

```
T __attribute__ ((aligned (ALIGNMENT_IN_BYTES)))
```

The alignment must be a power of two.

gccjit::type **gccjit::context::new_array_type**(gccjit::type element_type, int num_elements, gccjit::location loc)

Given type "T", get type "T[N]" (for a constant N). Param "loc" is optional.

Vector types

gccjit::type **gccjit::type::get_vector**(size_t num_units)

Given type "T", get type:

```
T  __attribute__ ((vector_size (sizeof(T) * num_units))
```

T must be integral or floating point; num_units must be a power of two.

Structures and unions

class gccjit::**struct_**

A compound type analagous to a C *struct*.

gccjit::struct_ is a subclass of gccjit::type (and thus of gccjit::object in turn).

class gccjit::**field**

A field within a gccjit::struct_.

gccjit::field is a subclass of gccjit::object.

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:

  ```
  gccjit::field field_x = ctxt.new_field (double_type, "x");
  gccjit::field field_y = ctxt.new_field (double_type, "y");
  std::vector fields;
  fields.push_back (field_x);
  fields.push_back (field_y);
  gccjit::struct_ coord = ctxt.new_struct_type ("coord", fields);
  ```

- 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:

  ```
  gccjit::struct_ node = ctxt.new_opaque_struct_type ("node");
  gccjit::type node_ptr = node.get_pointer ();
  gccjit::field field_hash = ctxt.new_field (int_type, "m_hash");
  gccjit::field field_next = ctxt.new_field (node_ptr, "m_next");
  std::vector fields;
  fields.push_back (field_hash);
  fields.push_back (field_next);
  node.set_fields (fields);
  ```

gccjit::field gccjit::context::**new_field**(gccjit::type type, const char *name, gccjit::location loc)

Construct a new field, with the given type and name.

gccjit::struct_ gccjit::context::**new_struct_type**(const std::string &name, std::vector<field> &fields, gccjit::location loc)

Construct a new struct type, with the given name and fields.

gccjit::struct_ **gccjit::context::new_opaque_struct**(const std::string &name, gccjit::location loc)

Construct a new struct type, with the given name, but without specifying the fields. The fields can be omitted (in which case the size of the struct is not known), or later specified using `gcc_jit_struct_set_fields()`.

3.2.4 Expressions

Rvalues

class **gccjit::rvalue**

A `gccjit::rvalue` is an expression that can be computed. It is a subclass of `gccjit::object`, and is a thin wrapper around `gcc_jit_rvalue*` from the C API.

It can be simple, e.g.:

- an integer value e.g. *0* or *42*
- a string literal e.g. *"Hello world"*
- a variable e.g. *i*. These are also lvalues (see below).

or compound e.g.:

- a unary expression e.g. *!cond*
- a binary expression e.g. *(a + b)*
- a function call e.g. *get_distance (&player_ship, &target)*
- etc.

Every rvalue has an associated type, and the API will check to ensure that types match up correctly (otherwise the context will emit an error).

gccjit::type **gccjit::rvalue::get_type**()

Get the type of this rvalue.

Simple expressions

gccjit::rvalue **gccjit::context::new_rvalue**(gccjit::type numeric_type, int value) const

Given a numeric type (integer or floating point), build an rvalue for the given constant `int` value.

gccjit::rvalue **gccjit::context::new_rvalue**(gccjit::type numeric_type, long value) const

Given a numeric type (integer or floating point), build an rvalue for the given constant `long` value.

gccjit::rvalue **gccjit::context::zero**(gccjit::type numeric_type) const

Given a numeric type (integer or floating point), get the rvalue for zero. Essentially this is just a shortcut for:

```
ctxt.new_rvalue (numeric_type, 0)
```

gccjit::rvalue **gccjit::context::one**(gccjit::type numeric_type) const

Given a numeric type (integer or floating point), get the rvalue for one. Essentially this is just a shortcut for:

```
ctxt.new_rvalue (numeric_type, 1)
```

gccjit::rvalue **gccjit::context::new_rvalue**(gccjit::type numeric_type, double value) const

Given a numeric type (integer or floating point), build an rvalue for the given constant `double` value.

gccjit::rvalue **gccjit::context::new_rvalue**(gccjit::type pointer_type, void *value) const

Given a pointer type, build an rvalue for the given address.

gccjit::rvalue **gccjit::context::new_rvalue**(const std::string &value) const

Generate an rvalue of type `GCC_JIT_TYPE_CONST_CHAR_PTR` for the given string. This is akin to a string literal.

Vector expressions

gccjit::rvalue **gccjit::context::new_rvalue**(gccjit::type vector_type, std::vector<gccjit::rvalue> elements) const

Given a vector type, and a vector of scalar rvalue elements, generate a vector rvalue.

The number of elements needs to match that of the vector type.

Unary Operations

gccjit::rvalue **gccjit::context::new_unary_op**(enum gcc_jit_unary_op, gccjit::type result_type, gccjit::rvalue rvalue, gccjit::location loc)

Build a unary operation out of an input rvalue.

Parameter `loc` is optional.

This is a thin wrapper around the C API's `gcc_jit_context_new_unary_op()` and the available unary operations are documented there.

There are shorter ways to spell the various specific kinds of unary operation:

gccjit::rvalue **gccjit::context::new_minus**(gccjit::type result_type, gccjit::rvalue a, gccjit::location loc)

Negate an arithmetic value; for example:

```
gccjit::rvalue negpi = ctxt.new_minus (t_double, pi);
```

builds the equivalent of this C expression:

```
-pi
```

gccjit::rvalue **new_bitwise_negate**(gccjit::type result_type, gccjit::rvalue a, gccjit::location loc)

Bitwise negation of an integer value (one's complement); for example:

```
gccjit::rvalue mask = ctxt.new_bitwise_negate (t_int, a);
```

builds the equivalent of this C expression:

```
~a
```

gccjit::rvalue **new_logical_negate**(gccjit::type result_type, gccjit::rvalue a, gccjit::location loc)

Logical negation of an arithmetic or pointer value; for example:

```
gccjit::rvalue guard = ctxt.new_logical_negate (t_bool, cond);
```

builds the equivalent of this C expression:

```
!cond
```

The most concise way to spell them is with overloaded operators:

gccjit::rvalue **operator-**(gccjit::rvalue a)

```
gccjit::rvalue negpi = -pi;
```

gccjit::rvalue **operator~**(gccjit::rvalue a)

```
gccjit::rvalue mask = ~a;
```

gccjit::rvalue **operator!**(gccjit::rvalue a)

```
gccjit::rvalue guard = !cond;
```

Binary Operations

gccjit::rvalue `gccjit::context::`**new_binary_op**(enum gcc_jit_binary_op, gccjit::type result_type, gccjit::rvalue a, gccjit::rvalue b, gccjit::location loc)

Build a binary operation out of two constituent rvalues.

Parameter `loc` is optional.

This is a thin wrapper around the C API's `gcc_jit_context_new_binary_op()` and the available binary operations are documented there.

There are shorter ways to spell the various specific kinds of binary operation:

gccjit::rvalue `gccjit::context::`**new_plus**(gccjit::type result_type, gccjit::rvalue a, gccjit::rvalue b, gccjit::location loc)

gccjit::rvalue **gccjit::context::new_minus**(gccjit::type result_type, gccjit::rvalue a, gccjit::rvalue b, gccjit::location loc)

gccjit::rvalue **gccjit::context::new_mult**(gccjit::type result_type, gccjit::rvalue a, gccjit::rvalue b, gccjit::location loc)

gccjit::rvalue **gccjit::context::new_divide**(gccjit::type result_type, gccjit::rvalue a, gccjit::rvalue b, gccjit::location loc)

gccjit::rvalue **gccjit::context::new_modulo**(gccjit::type result_type, gccjit::rvalue a, gccjit::rvalue b, gccjit::location loc)

gccjit::rvalue **gccjit::context::new_bitwise_and**(gccjit::type result_type, gccjit::rvalue a, gccjit::rvalue b, gccjit::location loc)

gccjit::rvalue **gccjit::context::new_bitwise_xor**(gccjit::type result_type, gccjit::rvalue a, gccjit::rvalue b, gccjit::location loc)

gccjit::rvalue **gccjit::context::new_bitwise_or**(gccjit::type result_type, gccjit::rvalue a, gccjit::rvalue b, gccjit::location loc)

gccjit::rvalue **gccjit::context::new_logical_and**(gccjit::type result_type, gccjit::rvalue a, gccjit::rvalue b, gccjit::location loc)

gccjit::rvalue **gccjit::context::new_logical_or**(gccjit::type result_type, gccjit::rvalue a, gccjit::rvalue b, gccjit::location loc)

The most concise way to spell them is with overloaded operators:

gccjit::rvalue **operator+**(gccjit::rvalue a, gccjit::rvalue b)

```
gccjit::rvalue sum = a + b;
```

gccjit::rvalue **operator-**(gccjit::rvalue a, gccjit::rvalue b)

```
gccjit::rvalue diff = a - b;
```

gccjit::rvalue **operator***(gccjit::rvalue a, gccjit::rvalue b)

```
gccjit::rvalue prod = a * b;
```

gccjit::rvalue **operator/**(gccjit::rvalue a, gccjit::rvalue b)

```
gccjit::rvalue result = a / b;
```

gccjit::rvalue **operator%**(gccjit::rvalue a, gccjit::rvalue b)

```
gccjit::rvalue mod = a % b;
```

gccjit::rvalue **operator&**(gccjit::rvalue a, gccjit::rvalue b)

```
gccjit::rvalue x = a & b;
```

gccjit::rvalue **operator^**(gccjit::rvalue a, gccjit::rvalue b)

```
gccjit::rvalue x = a ^ b;
```

gccjit::rvalue **operator|**(gccjit::rvalue a, gccjit::rvalue b)

```
gccjit::rvalue x = a | b;
```

gccjit::rvalue **operator&&**(gccjit::rvalue a, gccjit::rvalue b)

```
gccjit::rvalue cond = a && b;
```

gccjit::rvalue **operator||**(gccjit::rvalue a, gccjit::rvalue b)

```
gccjit::rvalue cond = a || b;
```

These can of course be combined, giving a terse way to build compound expressions:

```
gccjit::rvalue discriminant = (b * b) - (four * a * c);
```

Comparisons

gccjit::rvalue `gccjit::context::`**new_comparison**(enum gcc_jit_comparison, gccjit::rvalue a, gccjit::rvalue b, gccjit::location loc)

Build a boolean rvalue out of the comparison of two other rvalues.

Parameter `loc` is optional.

This is a thin wrapper around the C API's `gcc_jit_context_new_comparison()` and the available kinds of comparison are documented there.

There are shorter ways to spell the various specific kinds of binary operation:

gccjit::rvalue `gccjit::context::`**new_eq**(gccjit::rvalue a, gccjit::rvalue b, gccjit::location loc)

gccjit::rvalue `gccjit::context::`**new_ne**(gccjit::rvalue a, gccjit::rvalue b, gccjit::location loc)

gccjit::rvalue `gccjit::context::`**new_lt**(gccjit::rvalue a, gccjit::rvalue b, gccjit::location loc)

gccjit::rvalue `gccjit::context::`**new_le**(gccjit::rvalue a, gccjit::rvalue b, gccjit::location loc)

gccjit::rvalue `gccjit::context::`**new_gt**(gccjit::rvalue a, gccjit::rvalue b, gccjit::location loc)

gccjit::rvalue `gccjit::context::`**new_ge**(gccjit::rvalue a, gccjit::rvalue b, gccjit::location loc)

The most concise way to spell them is with overloaded operators:

gccjit::rvalue **operator==**(gccjit::rvalue a, gccjit::rvalue b)

```
gccjit::rvalue cond = (a == ctxt.zero (t_int));
```

gccjit::rvalue **operator!=**(gccjit::rvalue a, gccjit::rvalue b)

```
gccjit::rvalue cond = (i != j);
```

gccjit::rvalue **operator<**(gccjit::rvalue a, gccjit::rvalue b)

```
gccjit::rvalue cond = i < n;
```

gccjit::rvalue **operator<=**(gccjit::rvalue a, gccjit::rvalue b)

```
gccjit::rvalue cond = i <= n;
```

gccjit::rvalue **operator>**(gccjit::rvalue a, gccjit::rvalue b)

```
gccjit::rvalue cond = (ch > limit);
```

gccjit::rvalue **operator>=**(gccjit::rvalue a, gccjit::rvalue b)

```
gccjit::rvalue cond = (score >= ctxt.new_rvalue (t_int, 100));
```

Function calls

gcc_jit_rvalue ***gcc_jit_context_new_call**(gcc_jit_context *ctxt, gcc_jit_location *loc, gcc_jit_function *func, int numargs, gcc_jit_rvalue **args)

Given a function and the given table of argument rvalues, construct a call to the function, with the result as an rvalue.

Note: `gccjit::context::new_call()` merely builds a `gccjit::rvalue` i.e. an expression that can be evaluated, perhaps as part of a more complicated expression. The call *won't* happen unless you add a statement to a function that evaluates the expression.

For example, if you want to call a function and discard the result (or to call a function with `void` return type), use `gccjit::block::add_eval()`:

```
/* Add "(void)printf (arg0, arg1);".  */
block.add_eval (ctxt.new_call (printf_func, arg0, arg1));
```

Function pointers

gccjit::rvalue `gccjit::function::`**get_address**(gccjit::location loc)

Get the address of a function as an rvalue, of function pointer type.

Type-coercion

gccjit::rvalue **gccjit::context::new_cast**(gccjit::rvalue rvalue, gccjit::type type, gccjit::location loc)

Given an rvalue of T, construct another rvalue of another type.

Currently only a limited set of conversions are possible:

- int <-> float
- int <-> bool
- P* <-> Q*, for pointer types P and Q

Lvalues

class **gccjit::lvalue**

An lvalue is something that can of the *left*-hand side of an assignment: a storage area (such as a variable). It is a subclass of `gccjit::rvalue`, where the rvalue is computed by reading from the storage area.

It iss a thin wrapper around `gcc_jit_lvalue*` from the C API.

gccjit::rvalue **gccjit::lvalue::get_address**(gccjit::location loc)

Take the address of an lvalue; analogous to:

```
&(EXPR)
```

in C.

Parameter "loc" is optional.

Global variables

gccjit::lvalue **gccjit::context::new_global**(enum gcc_jit_global_kind, gccjit::type type, const char *name, gccjit::location loc)

Add a new global variable of the given type and name to the context.

This is a thin wrapper around `gcc_jit_context_new_global()` from the C API; the "kind" parameter has the same meaning as there.

Working with pointers, structs and unions

gccjit::lvalue **gccjit::rvalue::dereference**(gccjit::location loc)

Given an rvalue of pointer type `T *`, dereferencing the pointer, getting an lvalue of type `T`. Analogous to:

```
*(EXPR)
```

in C.

Parameter "loc" is optional.

If you don't need to specify the location, this can also be expressed using an overloaded operator:

gccjit::lvalue **gccjit::rvalue::operator*()**

```
gccjit::lvalue content = *ptr;
```

Field access is provided separately for both lvalues and rvalues:

gccjit::lvalue **gccjit::lvalue::access_field**(gccjit::field field, gccjit::location loc)

Given an lvalue of struct or union type, access the given field, getting an lvalue of the field's type. Analogous to:

```
(EXPR).field = ...;
```

in C.

gccjit::rvalue **gccjit::rvalue::access_field**(gccjit::field field, gccjit::location loc)

Given an rvalue of struct or union type, access the given field as an rvalue. Analogous to:

```
(EXPR).field
```

in C.

gccjit::lvalue **gccjit::rvalue::dereference_field**(gccjit::field field, gccjit::location loc)

Given an rvalue of pointer type `T *` where T is of struct or union type, access the given field as an lvalue. Analogous to:

```
(EXPR)->field
```

in C, itself equivalent to `(*EXPR).FIELD`.

gccjit::lvalue **gccjit::context::new_array_access**(gccjit::rvalue ptr, gccjit::rvalue index, gccjit::location loc)

Given an rvalue of pointer type `T *`, get at the element T at the given index, using standard C array indexing rules i.e. each increment of `index` corresponds to `sizeof(T)` bytes. Analogous to:

```
PTR[INDEX]
```

in C (or, indeed, to `PTR + INDEX`).

Parameter "loc" is optional.

For array accesses where you don't need to specify a `gccjit::location`, two overloaded operators are available:

gccjit::lvalue gccjit::rvalue::operator[] (gccjit::rvalue index)

```
gccjit::lvalue element = array[idx];
```

gccjit::lvalue gccjit::rvalue::operator[] (int index)

```
gccjit::lvalue element = array[0];
```

3.2.5 Creating and using functions

Params

class `gccjit::param`

A *gccjit::param* represents a parameter to a function.

gccjit::param `gccjit::context::new_param`(gccjit::type type, const char *name, gccjit::location loc)

In preparation for creating a function, create a new parameter of the given type and name.

`gccjit::param` is a subclass of `gccjit::lvalue` (and thus of `gccjit::rvalue` and `gccjit::object`). It is a thin wrapper around the C API's `gcc_jit_param`*.

Functions

class `gccjit::function`

A *gccjit::function* represents a function - either one that we're creating ourselves, or one that we're referencing.

gccjit::function `gccjit::context::new_function`(enum gcc_jit_function_kind, gccjit::type return_type, const char *name, std::vector<param> &params, int is_variadic, gccjit::location loc)

Create a gcc_jit_function with the given name and parameters.

Parameters "is_variadic" and "loc" are optional.

This is a wrapper around the C API's `gcc_jit_context_new_function()`.

gccjit::function `gccjit::context::get_builtin_function`(const char *name)

This is a wrapper around the C API's `gcc_jit_context_get_builtin_function()`.

gccjit::param `gccjit::function::get_param`(int index) const

Get the param of the given index (0-based).

void gccjit::function::**dump_to_dot**(const char *path)

Emit the function in graphviz format to the given path.

gccjit::lvalue gccjit::function::**new_local**(gccjit::type type, const char *name, gccjit::location loc)

Create a new local variable within the function, of the given type and name.

Blocks

class gccjit::**block**

A *gccjit::block* represents a basic block within a function i.e. a sequence of statements with a single entry point and a single exit point.

gccjit::block is a subclass of gccjit::object.

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, a return, or a switch.

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

gccjit::block gccjit::function::**new_block**(const char *name)

Create a basic block of the given name. The name may be NULL, but providing meaningful names is often helpful when debugging: it may show up in dumps of the internal representation, and in error messages.

Statements

void gccjit::block::**add_eval**(gccjit::rvalue rvalue, gccjit::location loc)

Add evaluation of an rvalue, discarding the result (e.g. a function call that "returns" void).

This is equivalent to this C code:

```
(void)expression;
```

void gccjit::block::**add_assignment**(gccjit::lvalue lvalue, gccjit::rvalue rvalue, gccjit::location loc)

Add evaluation of an rvalue, assigning the result to the given lvalue.

This is roughly equivalent to this C code:

```
lvalue = rvalue;
```

void gccjit::block::**add_assignment_op**(gccjit::lvalue lvalue, enum gcc_jit_binary_op, gccjit::rvalue rvalue, gccjit::location loc)

Add evaluation of an rvalue, using the result to modify an lvalue.

This is analogous to "+=" and friends:

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

etc. For example:

```
/* "i++" */
loop_body.add_assignment_op (
  i,
  GCC_JIT_BINARY_OP_PLUS,
  ctxt.one (int_type));
```

void gccjit::block::**add_comment**(const char *text, gccjit::location loc)

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 `GCC_JIT_BOOL_OPTION_DUMP_INITIAL_TREE` and `GCC_JIT_BOOL_OPTION_DUMP_INITIAL_GIMPLE`, and thus may be of use when debugging how your project's internal representation gets converted to the libgccjit IR.

Parameter "loc" is optional.

void gccjit::block::**end_with_conditional**(gccjit::rvalue boolval, gccjit::block on_true, gccjit::block on_false, gccjit::location loc)

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;
```

block, boolval, on_true, and on_false must be non-NULL.

void gccjit::block::**end_with_jump**(gccjit::block target, gccjit::location loc)

Terminate a block by adding a jump to the given target block.

This is roughly equivalent to this C code:

```
goto target;
```

void gccjit::block::**end_with_return**(gccjit::rvalue rvalue, gccjit::location loc)

Terminate a block.

Both params are optional.

An rvalue must be provided for a function returning non-void, and must not be provided by a function "returning" *void*.

If an rvalue is provided, the block is terminated by evaluating the rvalue and returning the value.

This is roughly equivalent to this C code:

```
return expression;
```

If an rvalue is not provided, the block is terminated by adding a valueless return, for use within a function with "void" return type.

This is equivalent to this C code:

```
return;
```

class `gccjit::case_`

A *gccjit::case_* represents a case within a switch statement, and is created within a particular `gccjit::context` using `gccjit::context::new_case()`. It is a subclass of `gccjit::object`.

Each case expresses a multivalued range of integer values. You can express single-valued cases by passing in the same value for both *min_value* and *max_value*.

gccjit::case_ *`gccjit::context::new_case`(gccjit::rvalue min_value, gccjit::rvalue max_value, gccjit::block dest_block)

Create a new gccjit::case for use in a switch statement. *min_value* and *max_value* must be constants of an integer type, which must match that of the expression of the switch statement.

dest_block must be within the same function as the switch statement.

void `gccjit::block::end_with_switch`(gccjit::rvalue expr, gccjit::block default_block, std::vector<gccjit::case_> cases, gccjit::location loc)

Terminate a block by adding evalation of an rvalue, then performing a multiway branch.

This is roughly equivalent to this C code:

```
switch (expr)
  {
  default:
    goto default_block;

  case C0.min_value ... C0.max_value:
    goto C0.dest_block;

  case C1.min_value ... C1.max_value:
    goto C1.dest_block;

  ...etc...

  case C[N - 1].min_value ... C[N - 1].max_value:
    goto C[N - 1].dest_block;
}
```

`expr` must be of the same integer type as all of the `min_value` and `max_value` within the cases.

The ranges of the cases must not overlap (or have duplicate values).

The API entrypoints relating to switch statements and cases:

- `gccjit::block::end_with_switch()`

- gccjit::context::new_case()

were added in LIBGCCJIT_ABI_3; you can test for their presence using

```
#ifdef LIBGCCJIT_HAVE_SWITCH_STATEMENTS
```

Here's an example of creating a switch statement:

```
void
create_code (gcc_jit_context *c_ctxt, void *user_data)
{
  /* Let's try to inject the equivalent of:
      int
      test_switch (int x)
      {
        switch (x)
          {
          case 0 ... 5:
             return 3;

          case 25 ... 27:
             return 4;

          case -42 ... -17:
             return 83;

          case 40:
             return 8;

          default:
             return 10;
          }
      }
   */
  gccjit::context ctxt (c_ctxt);
  gccjit::type t_int = ctxt.get_type (GCC_JIT_TYPE_INT);
  gccjit::type return_type = t_int;
  gccjit::param x = ctxt.new_param (t_int, "x");
  std::vector <gccjit::param> params;
  params.push_back (x);
  gccjit::function func = ctxt.new_function (GCC_JIT_FUNCTION_EXPORTED,
                                             return_type,
                                             "test_switch",
                                             params, 0);

  gccjit::block b_initial = func.new_block ("initial");

  gccjit::block b_default = func.new_block ("default");
  gccjit::block b_case_0_5 = func.new_block ("case_0_5");
  gccjit::block b_case_25_27 = func.new_block ("case_25_27");
  gccjit::block b_case_m42_m17 = func.new_block ("case_m42_m17");
  gccjit::block b_case_40 = func.new_block ("case_40");
```

(continues on next page)

(continued from previous page)

```
  std::vector <gccjit::case_> cases;
  cases.push_back (ctxt.new_case (ctxt.new_rvalue (t_int, 0),
                                  ctxt.new_rvalue (t_int, 5),
                                  b_case_0_5));
  cases.push_back (ctxt.new_case (ctxt.new_rvalue (t_int, 25),
                                  ctxt.new_rvalue (t_int, 27),
                                  b_case_25_27));
  cases.push_back (ctxt.new_case (ctxt.new_rvalue (t_int, -42),
                                  ctxt.new_rvalue (t_int, -17),
                                  b_case_m42_m17));
  cases.push_back (ctxt.new_case (ctxt.new_rvalue (t_int, 40),
                                  ctxt.new_rvalue (t_int, 40),
                                  b_case_40));
  b_initial.end_with_switch (x,
                             b_default,
                             cases);

  b_case_0_5.end_with_return (ctxt.new_rvalue (t_int, 3));
  b_case_25_27.end_with_return (ctxt.new_rvalue (t_int, 4));
  b_case_m42_m17.end_with_return (ctxt.new_rvalue (t_int, 83));
  b_case_40.end_with_return (ctxt.new_rvalue (t_int, 8));
  b_default.end_with_return (ctxt.new_rvalue (t_int, 10));
}
```

3.2.6 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.

gccjit::location instances are optional: you can always omit them from any C++ API entrypoint accepting one.

You can construct them using `gccjit::context::new_location()`.

You need to enable `GCC_JIT_BOOL_OPTION_DEBUGINFO` on the `gccjit::context` for these locations to actually be usable by the debugger:

```
ctxt.set_bool_option (GCC_JIT_BOOL_OPTION_DEBUGINFO, 1);
```

gccjit::location `gccjit::context::new_location`(const char *filename, int line, int column)

Create a *gccjit::location* instance representing the given source location.

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 ("/tmp/something.c",
                   1 /* update_locations */);
```

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.

3.2.7 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.

In-memory compilation

gcc_jit_result *`gccjit::context::compile`()

This calls into GCC and builds the code, returning a *gcc_jit_result **.

This is a thin wrapper around the `gcc_jit_context_compile()` API entrypoint.

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()` method.

void `gccjit::context::compile_to_file`(enum gcc_jit_output_kind, const char *output_path)

Compile the `gccjit::context` to a file of the given kind.

This is a thin wrapper around the `gcc_jit_context_compile_to_file()` API entrypoint.

3.2.8 Using Assembly Language with libgccjit++

libgccjit has some support for directly embedding assembler instructions. This is based on GCC's support for inline `asm` in C code, and the following assumes a familiarity with that functionality. See How to Use Inline Assembly Language in C Code in GCC's documentation, the "Extended Asm" section in particular.

These entrypoints were added in LIBGCCJIT_ABI_15; you can test for their presence using

```
#ifdef LIBGCCJIT_HAVE_ASM_STATEMENTS
```

Adding assembler instructions within a function

class gccjit::**extended_asm**

A *gccjit::extended_asm* represents an extended `asm` statement: a series of low-level instructions inside a function that convert inputs to outputs.

`gccjit::extended_asm` is a subclass of `gccjit::object`. It is a thin wrapper around the C API's `gcc_jit_extended_asm*`.

To avoid having an API entrypoint with a very large number of parameters, an extended `asm` statement is made in stages: an initial call to create the `gccjit::extended_asm`, followed by calls to add operands and set other properties of the statement.

There are two API entrypoints for creating a `gccjit::extended_asm`:

- `gccjit::block::add_extended_asm()` for an `asm` statement with no control flow, and
- `gccjit::block::end_with_extended_asm_goto()` for an `asm goto`.

For example, to create the equivalent of:

```
      asm ("mov %1, %0\n\t"
           "add $1, %0"
           : "=r" (dst)
           : "r" (src));
```

the following API calls could be used:

```
   block.add_extended_asm ("mov %1, %0\n\t"
                           "add $1, %0")
   .add_output_operand ("=r", dst)
   .add_input_operand ("r", src);
```

Warning: When considering the numbering of operands within an extended `asm` statement (e.g. the `%0` and `%1` above), the equivalent to the C syntax is followed i.e. all output operands, then all input operands, regardless of what order the calls to `gccjit::extended_asm::add_output_operand()` and `gccjit::extended_asm::add_input_operand()` were made in.

As in the C syntax, operands can be given symbolic names to avoid having to number them. For example, to create the equivalent of:

```
      asm ("bsfl %[aMask], %[aIndex]"
           : [aIndex] "=r" (Index)
           : [aMask] "r" (Mask)
           : "cc");
```

the following API calls could be used:

```
  block.add_extended_asm ("bsfl %[aMask], %[aIndex]")
  .add_output_operand ("aIndex", "=r", index)
  .add_input_operand ("aMask", "r", mask)
  .add_clobber ("cc");
```

extended_asm **gccjit::block::add_extended_asm**(const std::string &asm_template, gccjit::location loc = location())

Create a `gccjit::extended_asm` for an extended `asm` statement with no control flow (i.e. without the `goto` qualifier).

The parameter `asm_template` corresponds to the *AssemblerTemplate* within C's extended `asm` syntax. It must be non-NULL. The call takes a copy of the underlying string, so it is valid to pass in a pointer to an on-stack buffer.

extended_asm **gccjit::block::end_with_extended_asm_goto**(const std::string &asm_template, std::vector<block> goto_blocks, block *fallthrough_block, location loc = location())

Create a `gccjit::extended_asm` for an extended `asm` statement that may perform jumps, and use it to terminate the given block. This is equivalent to the `goto` qualifier in C's extended `asm` syntax.

For example, to create the equivalent of:

```
      asm goto ("btl %1, %0\n\t"
                "jc %l[carry]"
                : // No outputs
                : "r" (p1), "r" (p2)
                : "cc"
                : carry);
```

the following API calls could be used:

```
const char *asm_template =
  (use_name
   ? /* Label referred to by name: "%l[carry]".  */
     ("btl %1, %0\n\t"
      "jc %l[carry]")
   : /* Label referred to numerically: "%l2".  */
     ("btl %1, %0\n\t"
      "jc %l2"));

std::vector<gccjit::block> goto_blocks ({b_carry});
gccjit::extended_asm ext_asm
  = (b_start.end_with_extended_asm_goto (asm_template,
                                         goto_blocks,
                                         &b_fallthru)
     .add_input_operand ("r", p1)
     .add_input_operand ("r", p2)
     .add_clobber ("cc"));
```

here referencing a `gcc_jit_block` named "carry".

`num_goto_blocks` corresponds to the `GotoLabels` parameter within C's extended `asm` syntax. The block names can be referenced within the assembler template.

`fallthrough_block` can be NULL. If non-NULL, it specifies the block to fall through to after the statement.

Note: This is needed since each `gccjit::block` must have a single exit point, as a basic block: you can't jump from the middle of a block. A "goto" is implicitly added after the asm to handle the fallthrough case, which is equivalent to what would have happened in the C case.

---

gccjit::extended_asm &**gccjit::extended_asm::set_volatile_flag**(bool flag)

Set whether the `gccjit::extended_asm` has side-effects, equivalent to the volatile qualifier in C's extended asm syntax.

For example, to create the equivalent of:

```
asm volatile ("rdtsc\n\t"    // Returns the time in EDX:EAX.
              "shl $32, %%rdx\n\t"  // Shift the upper bits left.
              "or %%rdx, %0"        // 'Or' in the lower bits.
              : "=a" (msr)
              :
              : "rdx");
```

the following API calls could be used:

```
  gccjit::extended_asm ext_asm
    = block.add_extended_asm
        ("rdtsc\n\t"  /* Returns the time in EDX:EAX.  */
         "shl $32, %%rdx\n\t"  /* Shift the upper bits left.  */
         "or %%rdx, %0")  /* 'Or' in the lower bits.  */
    .set_volatile_flag (true)
    .add_output_operand ("=a", msr)
    .add_clobber ("rdx");
```

where the `gccjit::extended_asm` is flagged as volatile.

gccjit::extended_asm &**gccjit::extended_asm::set_inline_flag**(bool flag)

Set the equivalent of the inline qualifier in C's extended `asm` syntax.

gccjit::extended_asm &**gccjit::extended_asm::add_output_operand**(const std::string &asm_symbolic_name, const std::string &constraint, gccjit::lvalue dest)

Add an output operand to the extended `asm` statement. See the Output Operands section of the documentation of the C syntax.

`asm_symbolic_name` corresponds to the `asmSymbolicName` component of C's extended `asm` syntax, and specifies the symbolic name for the operand. See the overload below for an alternative

that does not supply a symbolic name.

`constraint` corresponds to the `constraint` component of C's extended `asm` syntax.

`dest` corresponds to the `cvariablename` component of C's extended `asm` syntax.

```
// Example with a symbolic name ("aIndex"), the equivalent of:
//   : [aIndex] "=r" (index)
ext_asm.add_output_operand ("aIndex", "=r", index);
```

This function can't be called on an `asm goto` as such instructions can't have outputs; see the Goto Labels section of GCC's "Extended Asm" documentation.

gccjit::extended_asm &**gccjit::extended_asm::add_output_operand**(const std::string &constraint, gccjit::lvalue dest)

As above, but don't supply a symbolic name for the operand.

```
// Example without a symbolic name, the equivalent of:
//   : "=r" (dst)
ext_asm.add_output_operand ("=r", dst);
```

gccjit::extended_asm &**gccjit::extended_asm::add_input_operand**(const std::string &asm_symbolic_name, const std::string &constraint, gccjit::rvalue src)

Add an input operand to the extended `asm` statement. See the Input Operands section of the documentation of the C syntax.

`asm_symbolic_name` corresponds to the `asmSymbolicName` component of C's extended `asm` syntax. See the overload below for an alternative that does not supply a symbolic name.

`constraint` corresponds to the `constraint` component of C's extended `asm` syntax.

`src` corresponds to the `cexpression` component of C's extended `asm` syntax.

```
// Example with a symbolic name ("aMask"), the equivalent of:
//   : [aMask] "r" (Mask)
ext_asm.add_input_operand ("aMask", "r", mask);
```

gccjit::extended_asm &**gccjit::extended_asm::add_input_operand**(const std::string &constraint, gccjit::rvalue src)

As above, but don't supply a symbolic name for the operand.

```
// Example without a symbolic name, the equivalent of:
//   : "r" (src)
ext_asm.add_input_operand ("r", src);
```

gccjit::extended_asm &**gccjit::extended_asm::add_clobber**(const std::string &victim)

Add *victim* to the list of registers clobbered by the extended `asm` statement. See the Clobbers and Scratch Registers section of the documentation of the C syntax.

Statements with multiple clobbers will require multiple calls, one per clobber.

For example:

```
ext_asm.add_clobber ("r0").add_clobber ("cc").add_clobber ("memory");
```

Adding top-level assembler statements

In addition to creating extended `asm` instructions within a function, there is support for creating "top-level" assembler statements, outside of any function.

void `gccjit::context::add_top_level_asm`(const char *asm_stmts, gccjit::location loc = location())

Create a set of top-level asm statements, analogous to those created by GCC's "basic" `asm` syntax in C at file scope.

For example, to create the equivalent of:

```
    asm ("\t.pushsection .text\n"
         "\t.globl add_asm\n"
         "\t.type add_asm, @function\n"
         "add_asm:\n"
         "\tmovq %rdi, %rax\n"
         "\tadd %rsi, %rax\n"
         "\tret\n"
         "\t.popsection\n");
```

the following API calls could be used:

```
ctxt.add_top_level_asm ("\t.pushsection .text\n"
                        "\t.globl add_asm\n"
                        "\t.type add_asm, @function\n"
                        "add_asm:\n"
                        "\tmovq %rdi, %rax\n"
                        "\tadd %rsi, %rax\n"
                        "\tret\n"
                        "\t# some asm here\n"
                        "\t.popsection\n");
```

CHAPTER

FOUR

INTERNALS

4.1 Working on the JIT library

Having checked out the source code (to "src"), you can configure and build the JIT library like this:

```
mkdir build
mkdir install
PREFIX=$(pwd)/install
cd build
../src/configure \
   --enable-host-shared \
   --enable-languages=jit,c++ \
   --disable-bootstrap \
   --enable-checking=release \
   --prefix=$PREFIX
nice make -j4 # altering the "4" to however many cores you have
```

This should build a libgccjit.so within jit/build/gcc:

```
[build] $ file gcc/libgccjit.so*
gcc/libgccjit.so:       symbolic link to `libgccjit.so.0'
gcc/libgccjit.so.0:     symbolic link to `libgccjit.so.0.0.1'
gcc/libgccjit.so.0.0.1: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically␣
↪linked, not stripped
```

Here's what those configuration options mean:

--enable-host-shared

Configuring with this option means that the compiler is built as position-independent code, which incurs a slight performance hit, but it necessary for a shared library.

--enable-languages=jit,c++

This specifies which frontends to build. The JIT library looks like a frontend to the rest of the code.

The C++ portion of the JIT test suite requires the C++ frontend to be enabled at configure-time, or you may see errors like this when running the test suite:

```
xgcc: error: /home/david/jit/src/gcc/testsuite/jit.dg/test-quadratic.cc: C++ compiler not␣
↪installed on this system
c++: error trying to exec 'cc1plus': execvp: No such file or directory
```

--disable-bootstrap

For hacking on the "jit" subdirectory, performing a full bootstrap can be overkill, since it's unused by a bootstrap. However, when submitting patches, you should remove this option, to ensure that the compiler can still bootstrap itself.

--enable-checking=release

The compile can perform extensive self-checking as it runs, useful when debugging, but slowing things down.

For maximum speed, configure with `--enable-checking=release` to disable this self-checking.

4.2 Running the test suite

```
[build] $ cd gcc
[gcc] $ make check-jit RUNTESTFLAGS="-v -v -v"
```

A summary of the tests can then be seen in:

```
jit/build/gcc/testsuite/jit/jit.sum
```

and detailed logs in:

```
jit/build/gcc/testsuite/jit/jit.log
```

The test executables are normally deleted after each test is run. For debugging, they can be preserved by setting `PRESERVE_EXECUTABLES` in the environment. If so, they can then be seen as:

```
jit/build/gcc/testsuite/jit/*.exe
```

which can be run independently.

You can compile and run individual tests by passing "jit.exp=TESTNAME" to RUNTESTFLAGS e.g.:

```
[gcc] $ PRESERVE_EXECUTABLES= \
          make check-jit \
            RUNTESTFLAGS="-v -v -v jit.exp=test-factorial.c"
```

and once a test has been compiled, you can debug it directly:

```
[gcc] $ PATH=.:$PATH \
        LD_LIBRARY_PATH=. \
        LIBRARY_PATH=. \
          gdb --args \
            testsuite/jit/test-factorial.c.exe
```

4.2.1 Running under valgrind

The jit testsuite detects if `RUN_UNDER_VALGRIND` is present in the environment (with any value). If it is present, it runs the test client code under valgrind, specifcally, the default memcheck tool with –leak-check=full.

It automatically parses the output from valgrind, injecting XFAIL results if any issues are found, or PASS results if the output is clean. The output is saved to `TESTNAME.exe.valgrind.txt`.

For example, the following invocation verbosely runs the testcase `test-sum-of-squares.c` under valgrind, showing an issue:

```
$ RUN_UNDER_VALGRIND= \
    make check-jit \
      RUNTESTFLAGS="-v -v -v jit.exp=test-sum-of-squares.c"

(...verbose log contains detailed valgrind errors, if any...)

                === jit Summary ===

# of expected passes            28
# of expected failures          2

$ less testsuite/jit/jit.sum
(...other results...)
XFAIL: jit.dg/test-sum-of-squares.c: test-sum-of-squares.c.exe.valgrind.txt: definitely lost: 8␣
↪bytes in 1 blocks
XFAIL: jit.dg/test-sum-of-squares.c: test-sum-of-squares.c.exe.valgrind.txt: unsuppressed␣
↪errors: 1
(...other results...)

$ less testsuite/jit/test-sum-of-squares.c.exe.valgrind.txt
(...shows full valgrind report for this test case...)
```

When running under valgrind, it's best to have configured gcc with `--enable-valgrind-annotations`, which automatically suppresses various known false positives.

4.3 Environment variables

When running client code against a locally-built libgccjit, three environment variables need to be set up:

LD_LIBRARY_PATH

> *libgccjit.so* is dynamically linked into client code, so if running against a locally-built library, `LD_LIBRARY_PATH` needs to be set up appropriately. The library can be found within the "gcc" subdirectory of the build tree:
>
> ```
> $ file libgccjit.so*
> libgccjit.so:       symbolic link to `libgccjit.so.0'
> ```

(continues on next page)

(continued from previous page)

```
libgccjit.so.0:     symbolic link to `libgccjit.so.0.0.1'
libgccjit.so.0.0.1: ELF 64-bit LSB shared object, x86-64, version 1 (GNU/Linux),␣
↪dynamically linked, not stripped
```

PATH

The library uses a driver executable for converting from .s assembler files to .so shared libraries. Specifically, it looks for a name expanded from `${target_noncanonical}-gcc-${gcc_BASEVER}${exeext}` such as `x86_64-unknown-linux-gnu-gcc-5.0.0`.

Hence `PATH` needs to include a directory where the library can locate this executable.

The executable is normally installed to the installation bindir (e.g. /usr/bin), but a copy is also created within the "gcc" subdirectory of the build tree for running the testsuite, and for ease of development.

LIBRARY_PATH

The driver executable invokes the linker, and the latter needs to locate support libraries needed by the generated code, or you will see errors like:

```
ld: cannot find crtbeginS.o: No such file or directory
ld: cannot find -lgcc
ld: cannot find -lgcc_s
```

Hence if running directly from a locally-built copy (without installing), `LIBRARY_PATH` needs to contain the "gcc" subdirectory of the build tree.

For example, to run a binary that uses the library against a non-installed build of the library in LIBGCCJIT_BUILD_DIR you need an invocation of the client code like this, to preprend the dir to each of the environment variables:

```
$ LD_LIBRARY_PATH=$(LIBGCCJIT_BUILD_DIR):$(LD_LIBRARY_PATH) \
  PATH=$(LIBGCCJIT_BUILD_DIR):$(PATH) \
  LIBRARY_PATH=$(LIBGCCJIT_BUILD_DIR):$(LIBRARY_PATH) \
    ./jit-hello-world
hello world
```

4.4 Packaging notes

The configure-time option `--enable-host-shared` is needed when building the jit in order to get position-independent code. This will slow down the regular compiler by a few percent. Hence when packaging gcc with libgccjit, please configure and build twice:

- once without `--enable-host-shared` for most languages, and
- once with `--enable-host-shared` for the jit

For example:

```
# Configure and build with --enable-host-shared
# for the jit:
mkdir configuration-for-jit
pushd configuration-for-jit
$(SRCDIR)/configure \
  --enable-host-shared \
  --enable-languages=jit \
  --prefix=$(DESTDIR)
make
popd

# Configure and build *without* --enable-host-shared
# for maximum speed:
mkdir standard-configuration
pushd standard-configuration
$(SRCDIR)/configure \
  --enable-languages=all \
  --prefix=$(DESTDIR)
make
popd

# Both of the above are configured to install to $(DESTDIR)
# Install the configuration with --enable-host-shared first
# *then* the one without, so that the faster build
# of "cc1" et al overwrites the slower build.
pushd configuration-for-jit
make install
popd

pushd standard-configuration
make install
popd
```

4.5 Overview of code structure

The library is implemented in C++.

- `libgccjit.cc` implements the API entrypoints. It performs error checking, then calls into classes of the gcc::jit::recording namespace within `jit-recording.cc` and `jit-recording.h`.
- The gcc::jit::recording classes (within `jit-recording.cc` and `jit-recording.h`) record the API calls that are made:

```
/* Indentation indicates inheritance: */
class context;
class memento;
  class string;
  class location;
  class type;
    class function_type;
```

(continues on next page)

(continued from previous page)

```
    class compound_type;
      class struct_;
      class union_;
    class vector_type;
  class field;
    class bitfield;
  class fields;
  class function;
  class block;
  class rvalue;
    class lvalue;
      class local;
      class global;
      class param;
    class base_call;
    class function_pointer;
  class statement;
    class extended_asm;
  class case_;
class top_level_asm;
```

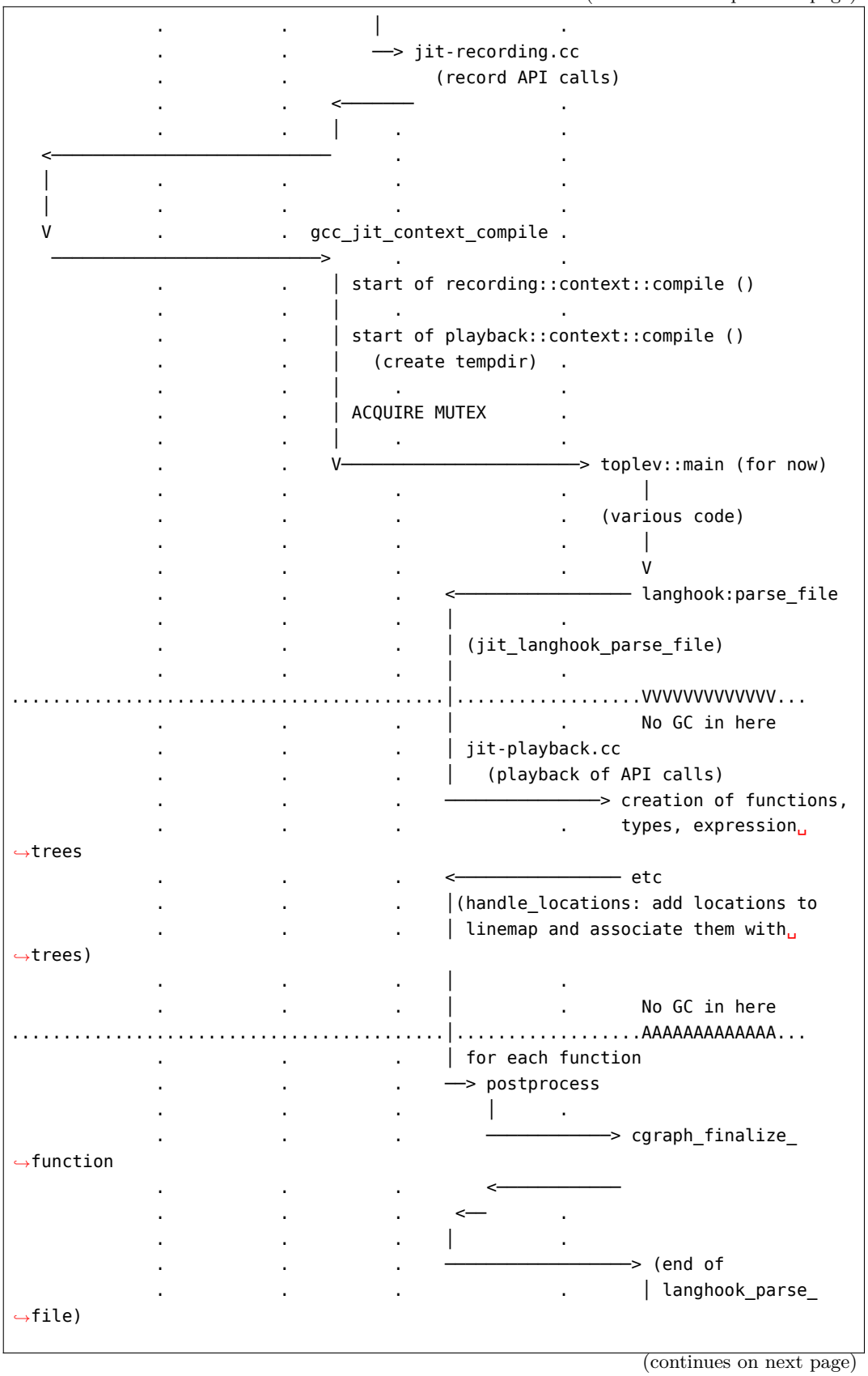
- When the context is compiled, the gcc::jit::playback classes (within `jit-playback.cc` and `jit-playback.h`) replay the API calls within langhook:parse_file:

```
/* Indentation indicates inheritance: */
class context;
class wrapper;
  class type;
    class compound_type;
  class field;
  class function;
  class block;
  class rvalue;
    class lvalue;
      class param;
  class source_file;
  class source_line;
  class location;
  class case_;
```

```
Client Code   . Generated .            libgccjit.so
              . code      .
              .           . JIT API  . JIT "Frontend". (libbackend.a)
.....................................................................................
↪...
   |          .           .          .               .
    ------------------------------>   .               .
              .           .    |     .               .
              .           .    V     .               .
              .           .    --> libgccjit.cc       .
              .           .        | (error-checking).
```

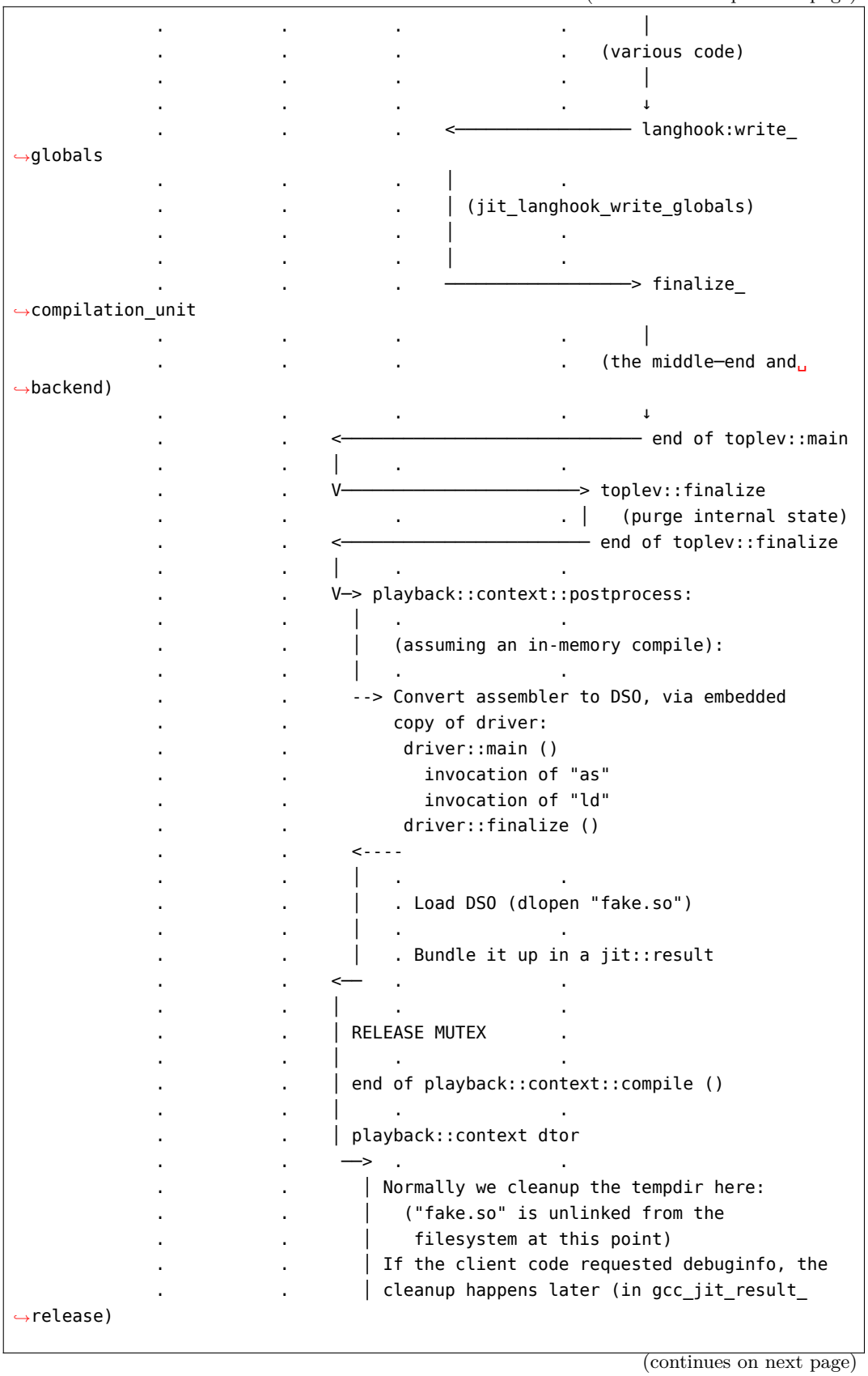
(continues on next page)

(continued from previous page)



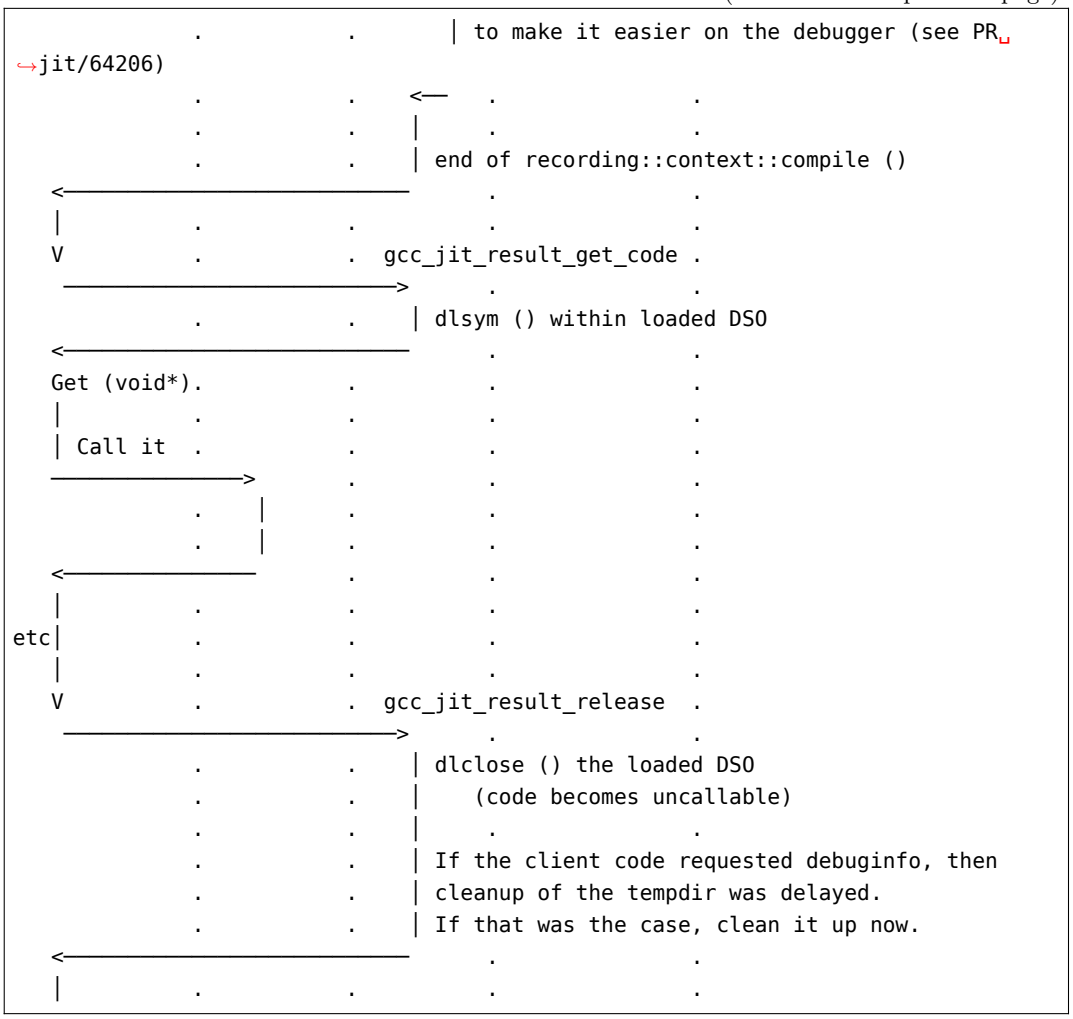
(continues on next page)

(continued from previous page)



(continues on next page)

(continued from previous page)



Here is a high-level summary from `jit-common.h`:

> In order to allow jit objects to be usable outside of a compile whilst working with the existing structure of GCC's code the C API is implemented in terms of a gcc::jit::recording::context, which records the calls made to it.
>
> When a gcc_jit_context is compiled, the recording context creates a playback context. The playback context invokes the bulk of the GCC code, and within the "frontend" parsing hook, plays back the recorded API calls, creating GCC tree objects.
>
> So there are two parallel families of classes: those relating to recording, and those relating to playback:
>
> - Visibility: recording objects are exposed back to client code, whereas playback objects are internal to the library.
> - Lifetime: recording objects have a lifetime equal to that of the recording context that created them, whereas playback objects only exist within the frontend hook.
> - Memory allocation: recording objects are allocated by the recording context, and automatically freed by it when the context is released, whereas playback objects are

allocated within the GC heap, and garbage-collected; they can own GC-references.

- Integration with rest of GCC: recording objects are unrelated to the rest of GCC, whereas playback objects are wrappers around "tree" instances. Hence you can't ask a recording rvalue or lvalue what its type is, whereas you can for a playback rvalue of lvalue (since it can work with the underlying GCC tree nodes).
- Instancing: There can be multiple recording contexts "alive" at once (albeit it only one compiling at once), whereas there can only be one playback context alive at one time (since it interacts with the GC).

Ultimately if GCC could support multiple GC heaps and contexts, and finer-grained initialization, then this recording vs playback distinction could be eliminated.

During a playback, we associate objects from the recording with their counterparts during this playback. For simplicity, we store this within the recording objects, as `void *m_playback_obj`, casting it to the appropriate playback object subclass. For these casts to make sense, the two class hierarchies need to have the same structure.

Note that the playback objects that `m_playback_obj` points to are GC-allocated, but the recording objects don't own references: these associations only exist within a part of the code where the GC doesn't collect, and are set back to NULL before the GC can run.

Another way to understand the structure of the code is to enable logging, via `gcc_jit_context_set_logfile()`. Here is an example of a log generated via this call:

```
JIT: libgccjit (GCC) version 6.0.0 20150803 (experimental) (x86_64-pc-linux-gnu)
JIT:        compiled by GNU C version 4.8.3 20140911 (Red Hat 4.8.3-7), GMP version 5.1.2, MPFR␣
↪version 3.1.2, MPC version 1.0.1
JIT: entering: gcc_jit_context_set_str_option
JIT:  GCC_JIT_STR_OPTION_PROGNAME: "./test-hello-world.c.exe"
JIT: exiting: gcc_jit_context_set_str_option
JIT: entering: gcc_jit_context_set_int_option
JIT:  GCC_JIT_INT_OPTION_OPTIMIZATION_LEVEL: 3
JIT: exiting: gcc_jit_context_set_int_option
JIT: entering: gcc_jit_context_set_bool_option
JIT:  GCC_JIT_BOOL_OPTION_DEBUGINFO: true
JIT: exiting: gcc_jit_context_set_bool_option
JIT: entering: gcc_jit_context_set_bool_option
JIT:  GCC_JIT_BOOL_OPTION_DUMP_INITIAL_TREE: false
JIT: exiting: gcc_jit_context_set_bool_option
JIT: entering: gcc_jit_context_set_bool_option
JIT:  GCC_JIT_BOOL_OPTION_DUMP_INITIAL_GIMPLE: false
JIT: exiting: gcc_jit_context_set_bool_option
JIT: entering: gcc_jit_context_set_bool_option
JIT:  GCC_JIT_BOOL_OPTION_SELFCHECK_GC: true
JIT: exiting: gcc_jit_context_set_bool_option
JIT: entering: gcc_jit_context_set_bool_option
JIT:  GCC_JIT_BOOL_OPTION_DUMP_SUMMARY: false
JIT: exiting: gcc_jit_context_set_bool_option
JIT: entering: gcc_jit_context_get_type
JIT: exiting: gcc_jit_context_get_type
```

(continues on next page)

(continued from previous page)

```
JIT: entering: gcc_jit_context_get_type
JIT: exiting: gcc_jit_context_get_type
JIT: entering: gcc_jit_context_new_param
JIT: exiting: gcc_jit_context_new_param
JIT: entering: gcc_jit_context_new_function
JIT: exiting: gcc_jit_context_new_function
JIT: entering: gcc_jit_context_new_param
JIT: exiting: gcc_jit_context_new_param
JIT: entering: gcc_jit_context_get_type
JIT: exiting: gcc_jit_context_get_type
JIT: entering: gcc_jit_context_new_function
JIT: exiting: gcc_jit_context_new_function
JIT: entering: gcc_jit_context_new_string_literal
JIT: exiting: gcc_jit_context_new_string_literal
JIT: entering: gcc_jit_function_new_block
JIT: exiting: gcc_jit_function_new_block
JIT: entering: gcc_jit_block_add_comment
JIT: exiting: gcc_jit_block_add_comment
JIT: entering: gcc_jit_context_new_call
JIT: exiting: gcc_jit_context_new_call
JIT: entering: gcc_jit_block_add_eval
JIT: exiting: gcc_jit_block_add_eval
JIT: entering: gcc_jit_block_end_with_void_return
JIT: exiting: gcc_jit_block_end_with_void_return
JIT: entering: gcc_jit_context_dump_reproducer_to_file
JIT:  entering: void gcc::jit::recording::context::dump_reproducer_to_file(const char*)
JIT:  exiting: void gcc::jit::recording::context::dump_reproducer_to_file(const char*)
JIT: exiting: gcc_jit_context_dump_reproducer_to_file
JIT: entering: gcc_jit_context_compile
JIT:  in-memory compile of ctxt: 0x1283e20
JIT:  entering: gcc::jit::result* gcc::jit::recording::context::compile()
JIT:   GCC_JIT_STR_OPTION_PROGNAME: "./test-hello-world.c.exe"
JIT:   GCC_JIT_INT_OPTION_OPTIMIZATION_LEVEL: 3
JIT:   GCC_JIT_BOOL_OPTION_DEBUGINFO: true
JIT:   GCC_JIT_BOOL_OPTION_DUMP_INITIAL_TREE: false
JIT:   GCC_JIT_BOOL_OPTION_DUMP_INITIAL_GIMPLE: false
JIT:   GCC_JIT_BOOL_OPTION_DUMP_GENERATED_CODE: false
JIT:   GCC_JIT_BOOL_OPTION_DUMP_SUMMARY: false
JIT:   GCC_JIT_BOOL_OPTION_DUMP_EVERYTHING: false
JIT:   GCC_JIT_BOOL_OPTION_SELFCHECK_GC: true
JIT:   GCC_JIT_BOOL_OPTION_KEEP_INTERMEDIATES: false
JIT:   gcc_jit_context_set_bool_allow_unreachable_blocks: false
JIT:   gcc_jit_context_set_bool_use_external_driver: false
JIT:   entering: void gcc::jit::recording::context::validate()
JIT:   exiting: void gcc::jit::recording::context::validate()
JIT:   entering: gcc::jit::playback::context::context(gcc::jit::recording::context*)
JIT:   exiting: gcc::jit::playback::context::context(gcc::jit::recording::context*)
JIT:   entering: gcc::jit::playback::compile_to_memory::compile_to_
↪memory(gcc::jit::recording::context*)
JIT:   exiting: gcc::jit::playback::compile_to_memory::compile_to_
↪memory(gcc::jit::recording::context*)
```

(continues on next page)

(continued from previous page)

```
JIT:   entering: void gcc::jit::playback::context::compile()
JIT:    entering: gcc::jit::tempdir::tempdir(gcc::jit::logger*, int)
JIT:    exiting: gcc::jit::tempdir::tempdir(gcc::jit::logger*, int)
JIT:    entering: bool gcc::jit::tempdir::create()
JIT:     m_path_template: /tmp/libgccjit-XXXXXX
JIT:     m_path_tempdir: /tmp/libgccjit-CKq1M9
JIT:    exiting: bool gcc::jit::tempdir::create()
JIT:    entering: void gcc::jit::playback::context::acquire_mutex()
JIT:    exiting: void gcc::jit::playback::context::acquire_mutex()
JIT:    entering: void gcc::jit::playback::context::make_fake_args(vec<char*>*, const char*, vec
↪<gcc::jit::recording::requested_dump>*)
JIT:     reusing cached configure-time options
JIT:     configure_time_options[0]: -mtune=generic
JIT:     configure_time_options[1]: -march=x86-64
JIT:    exiting: void gcc::jit::playback::context::make_fake_args(vec<char*>*, const char*, vec
↪<gcc::jit::recording::requested_dump>*)
JIT:    entering: toplev::main
JIT:     argv[0]: ./test-hello-world.c.exe
JIT:     argv[1]: /tmp/libgccjit-CKq1M9/fake.c
JIT:     argv[2]: -fPIC
JIT:     argv[3]: -O3
JIT:     argv[4]: -g
JIT:     argv[5]: -quiet
JIT:     argv[6]: --param
JIT:     argv[7]: ggc-min-expand=0
JIT:     argv[8]: --param
JIT:     argv[9]: ggc-min-heapsize=0
JIT:     argv[10]: -mtune=generic
JIT:     argv[11]: -march=x86-64
JIT:     entering: bool jit_langhook_init()
JIT:     exiting: bool jit_langhook_init()
JIT:     entering: void gcc::jit::playback::context::replay()
JIT:      entering: void gcc::jit::recording::context::replay_into(gcc::jit::replayer*)
JIT:      exiting: void gcc::jit::recording::context::replay_into(gcc::jit::replayer*)
JIT:      entering: void gcc::jit::recording::context::disassociate_from_playback()
JIT:      exiting: void gcc::jit::recording::context::disassociate_from_playback()
JIT:      entering: void gcc::jit::playback::context::handle_locations()
JIT:      exiting: void gcc::jit::playback::context::handle_locations()
JIT:      entering: void gcc::jit::playback::function::build_stmt_list()
JIT:      exiting: void gcc::jit::playback::function::build_stmt_list()
JIT:      entering: void gcc::jit::playback::function::build_stmt_list()
JIT:      exiting: void gcc::jit::playback::function::build_stmt_list()
JIT:      entering: void gcc::jit::playback::function::postprocess()
JIT:      exiting: void gcc::jit::playback::function::postprocess()
JIT:      entering: void gcc::jit::playback::function::postprocess()
JIT:      exiting: void gcc::jit::playback::function::postprocess()
JIT:     exiting: void gcc::jit::playback::context::replay()
JIT:    exiting: toplev::main
JIT:    entering: void gcc::jit::playback::context::extract_any_requested_dumps(vec
↪<gcc::jit::recording::requested_dump>*)
JIT:    exiting: void gcc::jit::playback::context::extract_any_requested_dumps(vec
```

(continues on next page)

(continued from previous page)

```
↪<gcc::jit::recording::requested_dump>*)
JIT:    entering: toplev::finalize
JIT:    exiting: toplev::finalize
JIT:    entering: virtual void gcc::jit::playback::compile_to_memory::postprocess(const char*)
JIT:     entering: void gcc::jit::playback::context::convert_to_dso(const char*)
JIT:      entering: void gcc::jit::playback::context::invoke_driver(const char*, const char*,␣
↪const char*, timevar_id_t, bool, bool)
JIT:       entering: void gcc::jit::playback::context::add_multilib_driver_arguments(vec<char*>*)
JIT:       exiting: void gcc::jit::playback::context::add_multilib_driver_arguments(vec<char*>*)
JIT:       argv[0]: x86_64-unknown-linux-gnu-gcc-6.0.0
JIT:       argv[1]: -m64
JIT:       argv[2]: -shared
JIT:       argv[3]: /tmp/libgccjit-CKq1M9/fake.s
JIT:       argv[4]: -o
JIT:       argv[5]: /tmp/libgccjit-CKq1M9/fake.so
JIT:       argv[6]: -fno-use-linker-plugin
JIT:       entering: void gcc::jit::playback::context::invoke_embedded_driver(const vec<char*>*)
JIT:       exiting: void gcc::jit::playback::context::invoke_embedded_driver(const vec<char*>*)
JIT:      exiting: void gcc::jit::playback::context::invoke_driver(const char*, const char*,␣
↪const char*, timevar_id_t, bool, bool)
JIT:     exiting: void gcc::jit::playback::context::convert_to_dso(const char*)
JIT:     entering: gcc::jit::result* gcc::jit::playback::context::dlopen_built_dso()
JIT:      GCC_JIT_BOOL_OPTION_DEBUGINFO was set: handing over tempdir to jit::result
JIT:      entering: gcc::jit::result::result(gcc::jit::logger*, void*, gcc::jit::tempdir*)
JIT:      exiting: gcc::jit::result::result(gcc::jit::logger*, void*, gcc::jit::tempdir*)
JIT:     exiting: gcc::jit::result* gcc::jit::playback::context::dlopen_built_dso()
JIT:    exiting: virtual void gcc::jit::playback::compile_to_memory::postprocess(const char*)
JIT:    entering: void gcc::jit::playback::context::release_mutex()
JIT:    exiting: void gcc::jit::playback::context::release_mutex()
JIT:   exiting: void gcc::jit::playback::context::compile()
JIT:   entering: gcc::jit::playback::context::~context()
JIT:   exiting: gcc::jit::playback::context::~context()
JIT:  exiting: gcc::jit::result* gcc::jit::recording::context::compile()
JIT:  gcc_jit_context_compile: returning (gcc_jit_result *)0x12f75d0
JIT: exiting: gcc_jit_context_compile
JIT: entering: gcc_jit_result_get_code
JIT:  locating fnname: hello_world
JIT:  entering: void* gcc::jit::result::get_code(const char*)
JIT:  exiting: void* gcc::jit::result::get_code(const char*)
JIT:  gcc_jit_result_get_code: returning (void *)0x7ff6b8cd87f0
JIT: exiting: gcc_jit_result_get_code
JIT: entering: gcc_jit_context_release
JIT:  deleting ctxt: 0x1283e20
JIT:  entering: gcc::jit::recording::context::~context()
JIT:  exiting: gcc::jit::recording::context::~context()
JIT: exiting: gcc_jit_context_release
JIT: entering: gcc_jit_result_release
JIT:  deleting result: 0x12f75d0
JIT:  entering: virtual gcc::jit::result::~result()
JIT:   entering: gcc::jit::tempdir::~tempdir()
JIT:    unlinking .s file: /tmp/libgccjit-CKq1M9/fake.s
```

(continues on next page)

(continued from previous page)

```
JIT:    unlinking .so file: /tmp/libgccjit-CKq1M9/fake.so
JIT:    removing tempdir: /tmp/libgccjit-CKq1M9
JIT:   exiting: gcc::jit::tempdir::~tempdir()
JIT:  exiting: virtual gcc::jit::result::~result()
JIT: exiting: gcc_jit_result_release
JIT: gcc::jit::logger::~logger()
```

4.6 Design notes

It should not be possible for client code to cause an internal compiler error. If this *does* happen, the root cause should be isolated (perhaps using `gcc_jit_context_dump_reproducer_to_file()`) and the cause should be rejected via additional checking. The checking ideally should be within the libgccjit API entrypoints in libgccjit.cc, since this is as close as possible to the error; failing that, a good place is within `recording::context::validate ()` in jit-recording.cc.

4.7 Submitting patches

Please read the contribution guidelines for gcc at https://gcc.gnu.org/contribute.html.

Patches for the jit should be sent to both the gcc-patches@gcc.gnu.org and jit@gcc.gnu.org mailing lists, with "jit" and "PATCH" in the Subject line.

You don't need to do a full bootstrap for code that just touches the `jit` and `testsuite/jit.dg` subdirectories. However, please run `make check-jit` before submitting the patch, and mention the results in your email (along with the host triple that the tests were run on).

A good patch should contain the information listed in the gcc contribution guide linked to above; for a `jit` patch, the patch shold contain:

- the code itself (for example, a new API entrypoint will typically touch `libgccjit.h` and `.c`, along with support code in `jit-recording.[ch]` and `jit-playback.[ch]` as appropriate)
- test coverage
- documentation for the C API
- documentation for the C++ API

A patch that adds new API entrypoints should also contain:

- a feature macro in `libgccjit.h` so that client code that doesn't use a "configure" mechanism can still easily detect the presence of the entrypoint. See e.g. `LIBGCCJIT_HAVE_SWITCH_STATEMENTS` (for a category of entrypoints) and `LIBGCCJIT_HAVE_gcc_jit_context_set_bool_allow_unreachable_blocks` (for an individual entrypoint).
- a new ABI tag containing the new symbols (in `libgccjit.map`), so that we can detect client code that uses them

- Support for gcc_jit_context_dump_reproducer_to_file(). Most jit testcases attempt to dump their contexts to a .c file; jit.exp then sanity-checks the generated c by compiling them (though not running them). A new API entrypoint needs to "know" how to write itself back out to C (by implementing gcc::jit::recording::memento::write_reproducer for the appropriate memento subclass).
- C++ bindings for the new entrypoints (see libgccjit++.h); ideally with test coverage, though the C++ API test coverage is admittedly spotty at the moment
- documentation for the new C entrypoints
- documentation for the new C++ entrypoints
- documentation for the new ABI tag (see topics/compatibility.rst).

Depending on the patch you can either extend an existing test case, or add a new test case. If you add an entirely new testcase: jit.exp expects jit testcases to begin with test-, or test-error- (for a testcase that generates an error on a gcc_jit_context).

Every new testcase that doesn't generate errors should also touch gcc/testsuite/jit.dg/all-non-failing-tests.h:

- Testcases that don't generate errors should ideally be added to the testcases array in that file; this means that, in addition to being run standalone, they also get run within test-combination.c (which runs all successful tests inside one big gcc_jit_context), and test-threads.c (which runs all successful tests in one process, each one running in a different thread on a different gcc_jit_context).

  Note: Given that exported functions within a gcc_jit_context must have unique names, and most testcases are run within test-combination.c, this means that every jit-compiled test function typically needs a name that's unique across the entire test suite.

- Testcases that aren't to be added to the testcases array should instead add a comment to the file clarifying why they're not in that array. See the file for examples.

Typically a patch that touches the .rst documentation will also need the texinfo to be regenerated. You can do this with Sphinx 1.0 or later by running make texinfo within SRCDIR/gcc/jit/docs. Don't do this within the patch sent to the mailing list; it can often be relatively large and inconsequential (e.g. anchor renumbering), rather like generated "configure" changes from configure.ac. You can regenerate it when committing to svn.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- search

INDEX

Symbols

C

E

G

L

N

O

P

R