# GNU Octave JIT Compilation - Report

Max Brister

August 18, 2012

## 1 Goal

My final goal for Google Summer of Code as stated in my project proposal is as follows.

> Code generation will work for octave_scalar and loops. There should be a noticeable speed up for compiled code. Furthermore, there should be no noticeable slow down in code that can not be compiled. Code speed up will be measured using some simple numerical integration tests that deal only with scalars. Code slow down will be measured by comparing the runtime of *make check*.
>
> At this point in the project the code should have some practical use in GNU Octave. JIT compilation will be much more useful once matrices are supported. However, it should still be interesting to merge the code into the main GNU Octave development branch. This is because JIT compilation will make some code much faster without a major impact to the rest of the code.

## 2 Progress

My goals have been both met and exceeded. In addition to supporting scalars JIT supports matrix indexing, complex scalars number, and a few built in functions. Furthermore, JIT has been merged into the main development branch and will be part of the next Octave release as an experimental feature.

## 3 Performance

### 3.1 make check

Running make check provides a good way of seeing what sort of slowdown JIT causes. Ideally, make check should test all the features in Octave on small data sets. This means that there is little benefit to using JIT, as loops will only run

| | |
|---|---|
| JIT | 104s |
| Interpter | 103s |

Table 1: make check runtime

| | Compile time (s) | Runtime (s) | Total (s) |
|---|---|---|---|
| JIT | 0.024 | 0.088 | 0.112 |
| Interpter | Na | 34 | 34 |

Table 2: euler benchmark

for a few iterations.

As we can see from table 1 JIT slows down make check by about 1%. In practice, this slowdown is not noticable as make check will vary by more then 1% between runs.

### 3.2 Simple Integration

The next benchmark is a simple integration test using euler integration to integrate $y' = 1/3x^2$ on $[0, 1]$ with $y_0 = 0$. The following Octave code was used

```
function result = euler(inc=1e-7)
  result = 0;
  tic;
  for ii = 0:inc:1
    result += inc * (1/3 * ii * ii);
  endfor
  toc;
endfunction
```

For JIT, the compile time indicates the overhead of the first function call. As we can see from table 2. From this we get a factor of 386 speedup.

## 4 Bugs

The only difference between JIT and the interpreter should be speed. Here I outline a few issues that might be difficult or time consuming to solve.

### 4.1 Error Messages

The error messages printed by JIT and the interpreter differ. For example

```
warning ("error", "Octave:divide-by-zero");
a = zeros (5);
for i = 1:1e2
```

```
  a((1/0):5) = 0;
endfor
```

When run in JIT the following output is produced

```
error: division by zero
```

However, the interpreter produces

```
error: division by zero
error: invalid base value in colon expression
error: evaluating argument list element number 1
error: invalid empty index list
```

I find the output of JIT to be cleaner, however the fact that there is a difference is still a bug.

### 4.2 Profiler

JIT does not generate any calls to the profiler.

### 4.3 Debugging

Supporting debugging JIT generated code is complex. Additionally, the Octave debugger allows for all variables in the current context to be inspected and modified. This means that in the debugger the user may break assumptions that the compiler made during type inference.

For this reason, I think the best approach is to disallow debugging while running JIT compiled code. This means we need to refuse to compile any code that has breakpoints in it.

## 5 Extending the Octave JIT Compiler

### 5.1 Types

When defining a new type the first task is to come up with a LLVM representation of the type. The LLVM representation should be a simple structure. For example, a range is represented as follows

```
struct
jit_range
{
  double base, limit, inc;
  octave_idx_type nelem;
};
```

The next task is to create a jit_type for the new type. jit_types are initialized in the jit_typeinfo constructor.

```
llvm::StructType *range_t = llvm::StructType::create (context, "range");
std::vector<llvm::Type *> range_contents (4, scalar_t);
range_contents[3] = index_t;
range_t->setBody (range_contents);

// type name - "range"
// parent type - matrix
// range_t - LLVM representation
range = new_type ("range", matrix, range_t);
```

Here range's parent type is matrix. The parent type indicates a range is a specific type of matrix.

Next, calling conventions must be established. Calling conventions determine how the type should be passed in LLVM. For example, range is marked as follows

```
range->mark_sret ();
range->mark_pointer_arg ();
```

mark_sret and mark_pointer_arg forces pointers to ranges to be passed to C++.

Finally, cast operations from any to the new type and from the new type to any must be defined.

## 5.2 Operations

Operations are defined in jit_typeinfo and initialized in the jit_typeinfo constructor. Each operation is overloaded with several jit_functions. An individual jit_function defines how the operation should behave given a set of types and what type the operation produces.

For example, take the cast to any operation. We need to implement this operation in C++ because we can not directly access an octave_value from LLVM. The C++ conversion functions is as follows

```
extern "C" double
octave_jit_cast_scalar_any (octave_base_value *obv)
{
  double ret = obv->double_value ();
  obv->release ();
  return ret;
}
```

Then, in the jit_typeinfo constructor, the function is added to the cast to scalar operation.

```
  fn = create_function (jit_convention::external, "octave_jit_cast_scalar_any",
                        scalar, any);
```

```
fn.add_mapping (engine, &octave_jit_cast_scalar_any);
casts[scalar->type_id ()].add_overload (fn);
```

Small functions should be implemented in LLVM. This allows LLVM optimizations to apply. For example, ++ for scalars

```
fn = create_function (jit_convention::internal, "octave_jit_++", scalar,
                      scalar);
body = fn.new_block ();
builder.SetInsertPoint (body);
{
  llvm::Value *one = llvm::ConstantFP::get (scalar_t, 1);
  llvm::Value *val = fn.argument (builder, 0);
  val = builder.CreateFAdd (val, one);
  fn.do_return (builder, val);
}
unary_ops[octave_value::op_incr].add_overload (fn);
```

### 5.3 Functions

Currently, support for builtin functions is added in the jit_typeinfo constructor. For example det is defined as follows

```
add_builtin ("det");
register_generic ("det", scalar, matrix);
```

No extra code is needed, as the internal Octave det function will be called.

## 6 Future Work

### 6.1 Support C++ Objects

Generating code to directly interact with C++ objects is not a simple task. Even with LLVM it is tricky to support in a cross platform way. However, supporting C++ objects directly in JIT might reduce the code duplication between JIT and the interpreter.

### 6.2 Improve Hand LLVM IR Creation

Operations can currently be specified in LLVM, however this is very awkward. For example, scalar division

```
fn = create_function (jit_convention::internal,
                      "octave_jit_div_scalar_scalar", scalar, scalar, scalar);
fn.mark_can_error ();

llvm::BasicBlock *body = fn.new_block ();
builder.SetInsertPoint (body);
```

```
{
  llvm::BasicBlock *warn_block = fn.new_block ("warn");
  llvm::BasicBlock *normal_block = fn.new_block ("normal");

  llvm::Value *zero = llvm::ConstantFP::get (scalar_t, 0);
  llvm::Value *check = builder.CreateFCmpUEQ (zero, fn.argument (builder, 1));
  builder.CreateCondBr (check, warn_block, normal_block);

  builder.SetInsertPoint (warn_block);
  gripe_div0.call (builder);
  builder.CreateBr (normal_block);

  builder.SetInsertPoint (normal_block);
  llvm::Value *ret = builder.CreateFDiv (fn.argument (builder, 0),
                                         fn.argument (builder, 1));
  fn.do_return (builder, ret);
}
```

Using an embedded domain specific language, it might look something like

```
fn = if_(arg1 == 0) (gripe_div0 ())
     (arg0 / arg1);
```

Additionally, if the language targeted the low level Octave IR, the code generated would be slightly better as the error check could be placed after the gripe_div0.

### 6.3 User Functions and Function Handles

There should be some way to compile an entire function and deal with function handles. This is important for speeding up functions like fsolve and arrayfun.

### 6.4 More

There are several other ways which JIT can be improved including adding support for multiple return values, cell arrays, structs, and objects.

## 7 Acknowledgments

I would like to thank my mentor John W. Eaton for providing interesting insights into the Octave interpreter and its development. I would also like to thank Jordi Gutirrez Hermoso for hosting OctConf 2012, it was a great experience. I would like to thank the Octave project for providing travel funding to attend OctConf 2012 and Google for funding my project. Finally, I would like to thank the Octave community for providing feedback and testing.