Supplementary Material for AG: Staged Programming For Easier Graphs

1 Dynamic RNN Implementation

Below is the hand-written graph implementation of the `tf.dynamic_rnn` cell.

```
def dynamic_rnn(rnn_cell, input_data,
  initial_state, sequence_len=None):
  input_data = tf.transpose(input_data,
    (1, 0, 2))
  outputs = tf.TensorArray(
      tf.float32, input_data.shape[0])
  if sequence_length is None:
    max_len = input_data.shape[0]
  else:
    max_len = tf.reduce_max(sequence_len)
  def while_body(i, state, outputs):
    prev_state = state
    output, state = rnn_cell(
        input_data[i], state)
    state = tf.where(
        i < sequence_len,
        state,
        prev_state)
    outputs = outputs.write(i, output)
    return i + 1, state, outputs
  def while_cond(i, state, outputs):
    return i < max_len
  _, state, outputs = tf.while_loop(
      while_cond,
      while_body,
      loop_vars=(tf.constant(0),
                 initial_state,
                 outputs))
  outputs = outputs.stack()
  outputs = tf.transpose(outputs, (1, 0, 2))
  return outputs, state
```

2 Error Handling

In AG, there are three distinct steps of execution, in addition to the usual syntax verifications performed by the Python runtime:

- Conversion
- Staging (e.g. TensorFlow graph construction)
- Runtime (e.g. TensorFlow graph execution)

The latter two steps can be associated with the two stages in the multi-stage programming model that platforms like TensorFlow and PyTorch's JIT model implement. Each of these steps has distinct requirements for error handling, but principally make use of these two technologies:

- *Source map construction*. Each node in the AST, even after several passes of SCT, is associated to an original line of the user's Python code.
- *Error rewriting*. Several frames in the stack trace of TensorFlow code, especailly AG-generated TensorFlow code, points to lines of code written by the AG compiler system, and not the user. We are able to reassociate temporary files (used when generating code in AG) to the user's original source files.

Conversion Errors Conversion errors may occur due to code that is otherwise legal Python, but is unsupported by AG. These errors usually originate inside AG internal code.

For usability, such errors must indicate the location in the converted code of the idiom that caused the error. In addition, the error message must provide sufficient information to allow the developer to remedy the error. Lastly, the error stack trace should avoid references to internal code, as they are typically uninformative to the user.

For example, the code snippet below will result in a conversion error indicating that the unary `+` operator is not supported:

```
def f(y):
  return +y
```

Currently, we facilitate this requirement by generating a stack-trace-like message that indicates the location of the unary plus operator, and including the message that the operator is not supported. In the future, we plan to further improve the conciseness of error messages of this type.

Staging Errors Staging errors can occur in successfully converted code and are typically raised because of disallowed or invalid argument types, shapes, hyperparameter values, or other conditions that are only detectable at runtime. Errors in auto-generated code at this stage are also intermingled with all other errors that occur at Python runtime.

For example, the converted code snippet below will result in a staging error indicating that the `mean` argument is of an invalid type:

```
def f(x):
  return tf.random_normal(
      (2,),
      mean=0.0, # tf.float32
      dtype=tf.int32)
```

Naively generated errors raised from inside of auto-generated code at staging time are difficult to interpret. To address this, we generate a stack-trace-like message that frames from the original code that the intermediate code was generated from, and excludes frames from internal code like AG. This is facilitated by the AST source map that we maintain between each node in the generated AST and the user's original source code.

Another challenge is that error messages may refer to generated symbols or to contexts specific to generated code. Addressing this shortcoming is a subject of future work.

Runtime Errors The name of these errors refers to the staged IR runtime.

For example, integer division by zero errors in TensorFlow:

```
def f(n):
  return tf.constant(10, dtype=tf.int32) / n
```

The IR execution environment typically includes facilities to trace the source of the error to user code, however, in the case of AG that will be generated code. To remedy this, we offer the possibility to intercept these errors and attach information that helps the user further trace the source of error to original, pre-conversion code.

The main challenge with intercepting runtime errors is that staged programs are executed at an unknown time and unknown place in the user code. For example, TensorFlow AG returns one or mode `Operation` or `Tensor` objects that the user can execute later. We offer a separate API, `AG.improved_errors` that the user can call when running staged computations, and which will make the necessary rewrites on any errors that may encounter.

The API for TensorFlow is used as a context manager that wraps the calls to `sess.run`:

```
converted_f = AG.to_graph(f)
tensor_result = converted_f(inputs)

with tf.Session() as sess:
  with sn.improved_errors(graph_f):
    sess.run(result)
```

We plan to further refine the error messages that this mechanism offers. We also plan a more seamless error handling process in TensorFlow Eager.

3 Useful Utilities

In order to build the system as described, we created a large library of source code transformation tools that we anticipate will be useful to the broader Python community.

Easy Code Quoting and Unquoting A few of the utility functions are listed below:

- `parser.parse_entity(fn_or_class)` takes a Python class or function and returns the corresponding AST node, wrapped in a containing `Module` node.
- `parser.parse_str(code_string)` is dentical to `parse_entity`, except takes a string of Python code as input. The string may contain any valid Python code.
- `pretty_printer.fmt(ast_node)` returns a pretty-printable string representing the AST.
- `compiler.ast_to_source(ast_node)` unparses an AST into the equivalent Python code, returned as a string.
- `compiler.ast_to_object(ast_node)` compiles an AST into an equivalent Python entity, returned as a module.

For example:

```
node = parse_str('a = b')
print(fmt(node))

# Output:
Module:
| body=[
| | Assign:
| | | targets=[
| | | | Name:
| | | | | id="a"
| | | | | ctx=Store()
| | | | | annotation=None
| | | ]
| | | value=Name:
| | | | id="b"
| | | | ctx=Load()
| | | | annotation=None
| ]
```

These utilities make it easy to make small modifications to the AST.

```
node = parse_str('a = b')
node.body[0].value.id = 'c'
print(ast_to_source(node))
```

```
# Output:
a = c
```

Templated Code Rewriting Example:

```
code_quote = '''
def fn(args):
  body
'''
new_body = textwrap.dedent('''
  a = x
  b = y
  return a + b
''')
node = templates.replace(
  code_quote,
  fn='my_function',
  args=('x', 'y'),
  body=parser.parse_str(new_body).body
)
print(compiler.ast_to_source(node))

# Output:
def my_function(x, y):
  a = x
  b = y
  return a + b
```

The function inserts string symbols or AST nodes into the quoted code template, and performs additional integrity checks. This allows for the easy construction of complicated code blocks, especially with respect to building the AST manually.

4 Extended Details on Side Effect Guards

Based on the heuristics for detecting and converting side effect guards described in the main text, lone function calls generate control dependencies.

```
# Before conversion
def f():
  print(1)
  print(2)
  return x + 1
x = f()
print(3)
x = x + 1

# After conversion (simplified)
def f():
  with ag.control_deps([ag.print_(1)]):
    with ag.control_deps([ag.print_(2)]):
      return x + 1
with ag.control_deps([ag.print_(3)]):
  x = x + 1
```

Note that control dependencies only affect the statements inside the block that contains them. In the example above, `print(3)` is not gated by the completion of `f`, and may execute in parallel. Improving this behavior is also a topic for future work.

Another special case arises when a function with side effects is called last in its block. In this case, we insert a dummy computation at the end of the block. This ensures the block itself matches the heuristic for ”function with side effects”.

```
# Before conversion
def f():
  print(1)

# After conversion (simplified)
def f():
  with ag.control_deps([ag.print_(1)]):
    return tf.constant(1)  # dummy value
```

If the block does have return values, then we wrap any returned `Tensor` with an `tf.identity` op.

```
# Before conversion
def f():
  print(1)
  return x

# After conversion (simplified)
def f():
  with ag.control_deps([ag.print_(1)]):
    return tf.identity(x)
```