sagunb Hash attributes instead of eliding them with ellipses to reduce the c... ... History

40 contributors  +25

2034 lines (1851 sloc) | 68.7 KB ...

```
1  /* Copyright 2015 The TensorFlow Authors. All Rights Reserved.
2
3  Licensed under the Apache License, Version 2.0 (the "License");
4  you may not use this file except in compliance with the License.
5  You may obtain a copy of the License at
6
7      http://www.apache.org/licenses/LICENSE-2.0
8
9  Unless required by applicable law or agreed to in writing, software
10 distributed under the License is distributed on an "AS IS" BASIS,
11 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 See the License for the specific language governing permissions and
13 limitations under the License.
14 ==============================================================================*/
15
16 #include "tensorflow/core/framework/function.h"
17
18 #include <ctype.h>
19
20 #include <map>
21 #include <unordered_map>
22 #include <utility>
23 #include <vector>
24
25 #include "absl/container/flat_hash_set.h"
26 #include "absl/strings/escaping.h"
27 #include "absl/strings/str_cat.h"
28 #include "absl/strings/str_join.h"
29 #include "tensorflow/core/framework/allocator.h"
```

```cpp
#include "tensorflow/core/framework/common_shape_fns.h"
#include "tensorflow/core/framework/function.pb.h"
#include "tensorflow/core/framework/graph.pb.h"
#include "tensorflow/core/framework/node_def.pb.h"
#include "tensorflow/core/framework/node_def_util.h"
#include "tensorflow/core/framework/op.h"
#include "tensorflow/core/graph/graph.h"
#include "tensorflow/core/lib/core/errors.h"
#include "tensorflow/core/lib/gtl/inlined_vector.h"
#include "tensorflow/core/lib/gtl/map_util.h"
#include "tensorflow/core/lib/strings/proto_serialization.h"
#include "tensorflow/core/platform/fingerprint.h"
#include "tensorflow/core/util/device_name_utils.h"
#include "tensorflow/core/util/equal_graph_def.h"

namespace tensorflow {

/* static */ constexpr const char* const FunctionLibraryDefinition::kArgOp;
/* static */ constexpr const char* const
    FunctionLibraryDefinition::kDeviceArgOp;
/* static */ constexpr const char* const FunctionLibraryDefinition::kRetOp;
/* static */ constexpr const char* const
    FunctionLibraryDefinition::kDeviceRetOp;
/* static */ constexpr const char* const
    FunctionLibraryDefinition::kIntsOnDeviceAttr;
/* static */ constexpr const char* const FunctionLibraryDefinition::kGradientOp;
/* static */ constexpr const char* const FunctionLibraryDefinition::kFuncAttr;

// Extracts the actual type from "attr_values" based on its definition
// "arg_def".
//
// If "arg_def" is a N*T type, *is_type_list is set to false, and
// *dtypes is set to be a vector of size N and each element is T.
//
// If "arg_def" is a list(type), *is_type_list is set to true, and
// *dtypes is set to be a vector of types specified in attrs for
// arg_def.
//
// Otherwise (arg_def is a simple type T), *is_type_list is set to
// false, and *dtypes is set to a single element vector, whose only
// element is T.
Status ArgNumType(AttrSlice attrs, const OpDef::ArgDef& arg_def,
                  bool* is_type_list, DataTypeVector* dtypes) {
  dtypes->clear();
  if (!arg_def.type_list_attr().empty()) {
    const AttrValue* v = attrs.Find(arg_def.type_list_attr());
    if (v == nullptr) {
      return errors::NotFound("type attr not found: ",
                              arg_def.type_list_attr());
```

```
    }
    *is_type_list = true;
    for (int i = 0; i < v->list().type_size(); ++i) {
      dtypes->push_back(v->list().type(i));
    }
    return Status::OK();
  }

  *is_type_list = false;
  int num = 1;
  if (!arg_def.number_attr().empty()) {
    const AttrValue* v = attrs.Find(arg_def.number_attr());
    if (v == nullptr) {
      return errors::NotFound("type attr not found: ", arg_def.type_attr());
    }
    num = v->i();
  }

  DataType dtype;
  if (arg_def.type() != DT_INVALID) {
    dtype = arg_def.type();
  } else if (arg_def.type_attr().empty()) {
    dtype = DT_INVALID;
  } else {
    const AttrValue* v = attrs.Find(arg_def.type_attr());
    if (v == nullptr) {
      return errors::NotFound("type attr not found: ", arg_def.type_attr());
    }
    dtype = v->type();
  }
  dtypes->resize(num, dtype);
  return Status::OK();
}

namespace {

template <typename T>
void AddAttr(const string& name, const T& val, NodeDef* ndef) {
  SetAttrValue(val, &((*ndef->mutable_attr())[name]));
}

Status ValidateSignatureWithAttrs(const OpDef& sig, AttrSlice attr_values) {
  // attr_values should specify all attrs defined in fdef, except for those
  // which have a default value
  for (const auto& attr : sig.attr()) {
    const AttrValue* attr_value = attr_values.Find(attr.name());
    if (attr_value) {
      Status status = AttrValueHasType(*attr_value, attr.type());
      if (!status.ok()) {
```

```
        errors::AppendToMessage(&status, "for attr '", attr.name(), "'");
        return status;
      }
    } else if (!attr.has_default_value()) {
      return errors::NotFound("Attr ", attr.name(), " is not found from ",
                              SummarizeOpDef(sig));
    }
  }

// TODO(josh11b): Enable this code once it works with function gradients.
// Right now the C++ function gradient code assumes it can pass
// all the attrs of the function to the gradient, and any attrs that
// the gradient doesn't care about will be ignored.
#if 0
  if (attr_values.size() != sig.attr_size()) {
    for (const auto& a : attr_values) {
      // TODO(josh11b): Possibly should ignore attrs that start with "_" here?
      bool found = false;
      for (const auto& s : sig.attr()) {
        if (a.first == s.name()) {
          found = true;
          break;
        }
      }
      if (!found) {
        return errors::NotFound("Attr ", a.first, " is not found in ",
                                SummarizeOpDef(sig));
      }
    }
  }
#endif

  return Status::OK();
}

// A helper class for instantiating functions. This contains shared information
// like the resulting graph and node name index.
class FunctionInstantiationHelper {
 public:
  FunctionInstantiationHelper(GetFunctionSignature get_function,
                              InstantiationResult* result)
      : get_function_(std ::move(get_function)), result_(*result) {
    result_.nodes.clear();
  }

  // Builds index for nodes that can be used as node's input arguments.
  // `resource_arg_unique_id`: if non-negative, will be populated to the
  // "_resource_arg_unique_id" attribute of the arg node.
  Status BuildInputArgIndex(const OpDef::ArgDef& arg_def, AttrSlice attr_values,
```

```
177                           const FunctionDef::ArgAttrs* arg_attrs,
178                           bool ints_on_device,
179                           int64_t resource_arg_unique_id) {
180     bool is_type_list;
181     DataTypeVector dtypes;
182     TF_RETURN_IF_ERROR(
183         ArgNumType(attr_values, arg_def, &is_type_list, &dtypes));
184     CHECK_GE(dtypes.size(), size_t{1});
185     int arg_index = result_.nodes.size();
186     TF_RETURN_IF_ERROR(
187         AddItem(arg_def.name(), {true, arg_index, 0, is_type_list, dtypes}));
188     // Creates dtypes.size() nodes in the graph.
189     for (size_t i = 0; i < dtypes.size(); ++i) {
190       TF_RETURN_IF_ERROR(AddItem(strings::StrCat(arg_def.name(), ":", i),
191                                  {true, arg_index, 0, false, {dtypes[i]}}));
192       DCHECK_EQ(arg_index, result_.nodes.size());
193       string name = arg_def.name();
194       if (dtypes.size() > 1) {
195         strings::StrAppend(&name, "_", i);
196       }
197       NodeDef* gnode = AddNode(name);
198       if (ints_on_device && dtypes[i] == DataType::DT_INT32) {
199         gnode->set_op(FunctionLibraryDefinition::kDeviceArgOp);
200       } else {
201         gnode->set_op(FunctionLibraryDefinition::kArgOp);
202       }
203       DataType dtype = arg_def.is_ref() ? MakeRefType(dtypes[i]) : dtypes[i];
204       AddAttr("T", dtype, gnode);
205       AddAttr("index", arg_index, gnode);
206       if (resource_arg_unique_id >= 0) {
207         AddAttr("_resource_arg_unique_id", resource_arg_unique_id, gnode);
208       }
209       if (arg_attrs) {
210         for (const auto& arg_attr : arg_attrs->attr()) {
211           AddAttr(arg_attr.first, arg_attr.second, gnode->mutable_attr());
212         }
213       }
214       result_.arg_types.push_back(dtypes[i]);
215       ++arg_index;
216     }
217     return Status::OK();
218   }
219
220   Status BuildNodeOutputIndex(const NodeDef& node, AttrSlice attrs,
221                               const int arg_index) {
222     const OpDef* node_sig = nullptr;
223     TF_RETURN_IF_ERROR(get_function_(node.op(), &node_sig));
224     if (node_sig->output_arg_size() == 0) {
225       return AddItem(node.name(), {false, arg_index, 0, false, {}});
```

```
226       }
227       const int num_retval = node_sig->output_arg_size();
228       int start = 0;
229       bool is_type_list;
230       DataTypeVector dtypes;
231       for (int i = 0; i < num_retval; ++i) {
232         TF_RETURN_IF_ERROR(
233             ArgNumType(attrs, node_sig->output_arg(i), &is_type_list, &dtypes));
234         // Note that we rely on the backwards-compatibility test enforcing
235         // that output_arg(*).name() doesn't change here.
236         const string base_name =
237             strings::StrCat(node.name(), ":", node_sig->output_arg(i).name());
238         TF_RETURN_IF_ERROR(
239             AddItem(base_name, {false, arg_index, start, is_type_list, dtypes}));
240         for (int j = 0; j < static_cast<int>(dtypes.size()); ++j) {
241           TF_RETURN_IF_ERROR(
242               AddItem(strings::StrCat(base_name, ":", j),
243                       {false, arg_index, start + j, false, {dtypes[j]}}));
244         }
245         start += dtypes.size();
246       }
247       return Status::OK();
248     }
249
250     Status InstantiateNode(const NodeDef& fnode, AttrSlice attrs) {
251       const OpDef* fnode_sig = nullptr;
252       TF_CHECK_OK(get_function_(fnode.op(), &fnode_sig));
253       NodeDef* gnode = AddNode(fnode.name());
254       gnode->set_op(fnode.op());
255       gnode->set_device(fnode.device());
256       int gnode_idx = nodes_.size() - 1;
257
258       // Input
259       const int num_args = fnode_sig->input_arg_size();
260       bool is_type_list;  // ignored
261       DataTypeVector dtypes;
262       int fnode_arg_index = 0;
263       for (int i = 0; i < num_args; ++i) {
264         TF_RETURN_IF_ERROR(
265             ArgNumType(attrs, fnode_sig->input_arg(i), &is_type_list, &dtypes));
266         // Consume inputs (indexed by fnode_arg_index) until we have
267         // matched each element of dtypes (indexed by j).
268         for (size_t j = 0; j < dtypes.size(); ++fnode_arg_index) {
269           if (fnode_arg_index >= fnode.input_size()) {
270             // Should never happen if we computed dtypes correctly.
271             return errors::InvalidArgument(
272                 "Attempt to access beyond input size: ", fnode_arg_index,
273                 " >= ", fnode.input_size());
274           }
```

```
275         // Look up the next input.
276         const string& input_name = fnode.input(fnode_arg_index);
277         const auto* item = GetItemOrNull(input_name);
278         if (item == nullptr) {
279           return errors::InvalidArgument(
280               "input ", input_name,
281               " is not found: ", FormatNodeDefForError(fnode));
282         }
283         if (item->dtypes.size() > dtypes.size() - j) {
284           return errors::InvalidArgument("Input ", input_name, " too long for ",
285                                          fnode_sig->input_arg(i).name());
286         }
287         // Match up all the elements of this input (indexed by k) with
288         // elements of dtypes (advancing j).
289         for (int k = 0; k < item->dtypes.size(); ++k, ++j) {
290           if (item->dtypes[k] != dtypes[j]) {
291             return errors::InvalidArgument(
292                 "input ", fnode_sig->input_arg(i).name(), "[", j,
293                 "] expected type ", DataTypeString(dtypes[j]),
294                 " != ", DataTypeString(item->dtypes[k]), ", the type of ",
295                 input_name, "[", k, "]");
296           }
297           if (item->is_func_arg) {
298             AddInput(gnode_idx, item->nid + k, 0);
299           } else {
300             AddInput(gnode_idx, item->nid, item->idx + k);
301           }
302         }
303       }
304     }
305
306     // Control deps.
307     for (int i = fnode_arg_index; i < fnode.input_size(); ++i) {
308       const string& input = fnode.input(i);
309       if (input.empty() || input[0] != '^') {
310         return errors::InvalidArgument("Expected input[", i, "] == '", input,
311                                        "' to be a control input.");
312       }
313       int nid = -1;
314       const string node_name = input.substr(1);
315       const string node_colon = node_name + ":";
316       const string node_colon_bound = node_name + ";";
317       // index_ is a map sorted lexicographically, so the key we are looking for
318       // must lie in the range [node_name, node_colon_bound).
319       auto it = index_.lower_bound(node_name);
320       while (it != index_.end() && it->first <= node_colon_bound) {
321         if (it->first == node_name || absl::StartsWith(it->first, node_colon)) {
322           nid = it->second.nid;
323           break;
```

```
324         }
325         ++it;
326       }
327       if (nid == -1) {
328         return errors::InvalidArgument("input[", i, "] == '", input,
329                                        "', is not found.");
330       }
331       AddDep(gnode_idx, nid);
332     }
333
334     // Attrs.
335     for (const auto& p : attrs) {
336       (*gnode->mutable_attr())[p.first] = p.second;
337     }
338
339     // Experimental_debug_info.
340     if (fnode.has_experimental_debug_info()) {
341       gnode->mutable_experimental_debug_info()->MergeFrom(
342           fnode.experimental_debug_info());
343     }
344
345     // Tye info.
346     // TODO(mdan): Might this need adjustment at instantiation?
347     if (fnode.has_experimental_type()) {
348       *gnode->mutable_experimental_type() = fnode.experimental_type();
349     }
350
351     return Status::OK();
352   }
353
354   Status AddReturnNode(
355       const OpDef::ArgDef& ret_def, AttrSlice attrs,
356       const ::tensorflow::protobuf::Map<string, string>& ret_map,
357       bool ints_on_device, int* ret_index) {
358     auto ret_iter = ret_map.find(ret_def.name());
359     if (ret_iter == ret_map.end()) {
360       return errors::InvalidArgument("Return ", ret_def.name(), " missing.");
361     }
362     bool is_type_list;
363     DataTypeVector dtypes;
364     TF_RETURN_IF_ERROR(ArgNumType(attrs, ret_def, &is_type_list, &dtypes));
365     CHECK_GE(dtypes.size(), size_t{1});
366     const auto* item = GetItemOrNull(ret_iter->second);
367     if (item == nullptr) {
368       return errors::InvalidArgument("Return ", ret_def.name(), " -> ",
369                                      ret_iter->second, " is not found.");
370     }
371     if (dtypes != item->dtypes) {
372       return errors::InvalidArgument("Invalid ret types ", ret_def.name(),
```

```
373                                     " : ", DataTypeVectorString(dtypes),
374                                     " vs. ",
375                                     DataTypeVectorString(item->dtypes));
376       }
377       for (size_t i = 0; i < dtypes.size(); ++i) {
378         string name = strings::StrCat(ret_def.name(), "_RetVal");
379         if (dtypes.size() > 1) {
380           strings::StrAppend(&name, "_", i);
381         }
382         NodeDef* gnode = AddNode(name);
383         if (ints_on_device && dtypes[i] == DataType::DT_INT32) {
384           gnode->set_op(FunctionLibraryDefinition::kDeviceRetOp);
385         } else {
386           gnode->set_op(FunctionLibraryDefinition::kRetOp);
387         }
388         AddInput(nodes_.size() - 1, item->nid, item->idx + i);
389         DataType dtype = ret_def.is_ref() ? MakeRefType(dtypes[i]) : dtypes[i];
390         AddAttr("T", dtype, gnode);
391         AddAttr("index", (*ret_index)++, gnode);
392         result_.ret_types.push_back(dtypes[i]);
393       }
394       return Status::OK();
395     }
396
397     // Adds the actual node inputs to the result graph by converting indexes to
398     // the node names.
399     void AddNodeInputs() {
400       for (int i = 0; i < result_.nodes.size(); i++) {
401         NodeInfo& node_info = nodes_[i];
402         for (const auto& p : node_info.data_inputs) {
403           result_.nodes[i].add_input(Name(p.first, p.second));
404         }
405         for (int index : node_info.control_inputs) {
406           result_.nodes[i].add_input(Dep(index));
407         }
408       }
409     }
410
411   private:
412     // This is used to build a small index for all names that can be used as a
413     // node's input arguments.
414     //
415     // If is_func_arg is true, the name is a function's argument.  In
416     // this case, the produced graph def has node[nid:nid + dtype.size()].
417     //
418     // Otherwise, the name is a function body's node return value.  In
419     // this case, the produced graph def has one node node[nid] and
420     // the node's output index [idx ... idx + num) corresponds to the
421     // named outputs.
```

```
  //
  // In all cases, "dtype" specifies the data type.
  struct NameInfoItem {
    bool is_func_arg;
    int nid;
    int idx;
    bool is_type_list;
    DataTypeVector dtypes;
  };

  // Adds an item into the input name index.
  Status AddItem(const string& name, const NameInfoItem& item) {
    if (!index_.insert({name, item}).second) {
      return errors::InvalidArgument(
          strings::StrCat("Duplicated ", item.is_func_arg ? "arg" : "ret",
                          " name: "),
          name);
    }
    return Status::OK();
  }

  const NameInfoItem* GetItemOrNull(const string& name) const {
    return gtl::FindOrNull(index_, name);
  }

  string Dep(int node_index) const {
    return strings::StrCat("^", Name(node_index));
  }

  string Name(int node_index) const {
    CHECK_LT(node_index, nodes_.size());
    return nodes_[node_index].name;
  }

  string Name(int node_index, int output_index) const {
    if (output_index == 0) {
      return Name(node_index);
    } else {
      return strings::StrCat(Name(node_index), ":", output_index);
    }
  }

  NodeDef* AddNode(const string& name) {
    result_.nodes.emplace_back();
    NodeDef* gnode = &result_.nodes.back();
    gnode->set_name(name);
    nodes_.push_back({name, {}, {}});
    CHECK_EQ(result_.nodes.size(), nodes_.size());
    return gnode;
```

```
471     }
472
473     void AddInput(int node_index, int output_node, int output_index) {
474       CHECK_LT(node_index, nodes_.size());
475       nodes_[node_index].data_inputs.push_back(
476           std::make_pair(output_node, output_index));
477     }
478
479     void AddDep(int node_index, int dep_index) {
480       CHECK_LT(node_index, nodes_.size());
481       nodes_[node_index].control_inputs.push_back(dep_index);
482     }
483
484     GetFunctionSignature get_function_;
485     InstantiationResult& result_;
486     // A small index for all names that can be used as a node's input arguments.
487     std::map<string, NameInfoItem> index_;
488     // This contains information about a node in the new graph including the node
489     // names and input nodes' indexes.
490     struct NodeInfo {
491       string name;
492       // Data inputs where <n, k> means arg k of node n.
493       std::vector<std::pair<int, int>> data_inputs;
494       // Control inputs (dependencies).
495       std::vector<int> control_inputs;
496     };
497     // nodes_[i] is the information about result_.nodes[i].
498     std::vector<NodeInfo> nodes_;
499   };
500
501   // Various helpers Print(proto) to print relevant protos to ascii.
502   string Print(const OpDef::ArgDef& arg) {
503     string out;
504     strings::StrAppend(&out, arg.name(), ":");
505     if (arg.is_ref()) strings::StrAppend(&out, "Ref(");
506     if (!arg.number_attr().empty()) {
507       strings::StrAppend(&out, arg.number_attr(), "*");
508     }
509     if (arg.type() != DT_INVALID) {
510       strings::StrAppend(&out, DataTypeString(arg.type()));
511     } else {
512       strings::StrAppend(&out, arg.type_attr());
513     }
514     if (arg.is_ref()) strings::StrAppend(&out, ")");
515     return out;
516   }
517
518   // TODO(josh11b): Merge this with SummarizeAttrValue().
519   // When hash_string_attrs = true, string attributes are hashed instead of being
```

```
// truncated with ellipses. This is done to reduce the chance of collisions when
// looking up functions using the canonical representation.
string Print(const AttrValue& attr_value,
             const bool hash_string_attrs = false) {
  if (attr_value.value_case() == AttrValue::kType) {
    return DataTypeString(attr_value.type());
  } else if ((attr_value.value_case() == AttrValue::kList) &&
             (attr_value.list().type_size() > 0)) {
    string ret = "{";
    for (int i = 0; i < attr_value.list().type_size(); ++i) {
      if (i > 0) strings::StrAppend(&ret, ", ");
      strings::StrAppend(&ret, DataTypeString(attr_value.list().type(i)));
    }
    strings::StrAppend(&ret, "}");
    return ret;
  } else if (attr_value.value_case() == AttrValue::kFunc) {
    if (attr_value.func().attr_size() == 0) {
      return attr_value.func().name();
    }
    std::vector<string> entries;
    for (const auto& p : attr_value.func().attr()) {
      entries.push_back(strings::StrCat(p.first, "=", Print(p.second)));
    }
    std::sort(entries.begin(), entries.end());
    return strings::StrCat(attr_value.func().name(), "[",
                           absl::StrJoin(entries, ", "), "]");
  } else if (attr_value.value_case() == AttrValue::kS && hash_string_attrs) {
    return strings::StrCat(Fingerprint64(attr_value.s()));
  }
  return SummarizeAttrValue(attr_value);
}

// TODO(josh11b): Merge this with SummarizeNodeDef().
string Print(const NodeDef& n) {
  string out;
  strings::StrAppend(&out, n.name(), " = ", n.op());
  if (n.attr_size() > 0) {
    std::vector<string> entries;
    for (auto& a : n.attr()) {
      entries.push_back(strings::StrCat(a.first, "=", Print(a.second)));
    }
    std::sort(entries.begin(), entries.end());
    // Add a short device string at the end of all attributes.
    if (!n.device().empty()) {
      DeviceNameUtils::ParsedName parsed;
      if (DeviceNameUtils::ParseFullName(n.device(), &parsed)) {
        entries.push_back(
            strings::StrCat("device=", parsed.type, ":", parsed.id));
      } else {
```

```
569           entries.push_back("device=<FAILED_TO_PARSE>");
570         }
571       }
572       strings::StrAppend(&out, "[", absl::StrJoin(entries, ", "), "]");
573     }
574     strings::StrAppend(&out, "(");
575     std::vector<StringPiece> dat;
576     std::vector<string> dep;
577     for (StringPiece s : n.input()) {
578       if (absl::ConsumePrefix(&s, "^")) {
579         dep.emplace_back(s);
580       } else {
581         dat.push_back(s);
582       }
583     }
584     strings::StrAppend(&out, absl::StrJoin(dat, ", "), ")");
585     if (!dep.empty()) {
586       strings::StrAppend(&out, " @ ", absl::StrJoin(dep, ", "));
587     }
588     return out;
589   }
590
591   string Print(const FunctionDef& fdef) {
592     string out;
593     const OpDef& sig = fdef.signature();
594     strings::StrAppend(&out, "\n", sig.name());
595     if (sig.attr_size() > 0) {
596       strings::StrAppend(&out, "[");
597       for (int i = 0; i < sig.attr_size(); ++i) {
598         const auto& a = sig.attr(i);
599         if (i > 0) strings::StrAppend(&out, ", ");
600         if (a.type() == "type") {
601           strings::StrAppend(&out, a.name(), ":", Print(a.allowed_values()));
602         } else {
603           strings::StrAppend(&out, a.name(), ":", a.type());
604         }
605       }
606       strings::StrAppend(&out, "]");
607     }
608     strings::StrAppend(&out, "(");
609     for (int i = 0; i < sig.input_arg_size(); ++i) {
610       if (i > 0) strings::StrAppend(&out, ", ");
611       strings::StrAppend(&out, Print(sig.input_arg(i)));
612     }
613     strings::StrAppend(&out, ") -> (");
614     for (int i = 0; i < sig.output_arg_size(); ++i) {
615       if (i > 0) strings::StrAppend(&out, ", ");
616       strings::StrAppend(&out, Print(sig.output_arg(i)));
617     }
```

```
  strings::StrAppend(&out, ") {\n");
  for (const auto& n : fdef.node_def()) {
    strings::StrAppend(&out, "  ", Print(n), "\n");
  }
  for (const auto& cr : fdef.control_ret()) {
    strings::StrAppend(&out, "  @return ", cr.first, " = ", cr.second, "\n");
  }
  for (const auto& r : fdef.ret()) {
    strings::StrAppend(&out, "  return ", r.first, " = ", r.second, "\n");
  }
  strings::StrAppend(&out, "}\n");
  return out;
}

string Print(gtl::ArraySlice<const NodeDef*> nodes) {
  std::vector<const NodeDef*> arg;
  std::vector<const NodeDef*> ret;
  std::vector<const NodeDef*> body;
  for (const NodeDef* n : nodes) {
    if (n->op() == FunctionLibraryDefinition::kArgOp ||
        n->op() == FunctionLibraryDefinition::kDeviceArgOp) {
      arg.push_back(n);
    } else if (n->op() == FunctionLibraryDefinition::kRetOp ||
               n->op() == FunctionLibraryDefinition::kDeviceRetOp) {
      ret.push_back(n);
    } else {
      body.push_back(n);
    }
  }
  auto comp = [](const NodeDef* x, const NodeDef* y) {
    int xi;
    TF_CHECK_OK(GetNodeAttr(*x, "index", &xi));
    int yi;
    TF_CHECK_OK(GetNodeAttr(*y, "index", &yi));
    return xi < yi;
  };
  std::sort(arg.begin(), arg.end(), comp);
  std::sort(ret.begin(), ret.end(), comp);
  string out;
  strings::StrAppend(&out, "\n(");
  auto get_type_and_device = [](const NodeDef& n) {
    DataType dt;
    if (!TryGetNodeAttr(n, "T", &dt)) {
      dt = DT_INVALID;
    }
    if (!n.device().empty()) {
      DeviceNameUtils::ParsedName parsed;
      if (DeviceNameUtils::ParseFullName(n.device(), &parsed)) {
        return strings::StrCat(DataTypeString(dt), "@", parsed.type, ":",
```

```
                                parsed.id);
      } else {
        LOG(WARNING) << "Failed to parse device \"" << n.device() << "\" in "
                     << n.op() << ":" << n.name();
        return strings::StrCat(DataTypeString(dt), "@",
                               "<FAILED_TO_PARSE_DEVICE>");
      }
    }
    return DataTypeString(dt);
  };
  for (size_t i = 0; i < arg.size(); ++i) {
    const NodeDef* n = arg[i];
    if (i > 0) strings::StrAppend(&out, ", ");
    CHECK_GE(n->attr_size(), 2);
    strings::StrAppend(&out, n->name(), ":", get_type_and_device(*n));
  }
  strings::StrAppend(&out, ") -> (");
  for (size_t i = 0; i < ret.size(); ++i) {
    const NodeDef* n = ret[i];
    if (i > 0) strings::StrAppend(&out, ", ");
    CHECK_LE(2, n->attr_size());

    // The _RetVal op should have a unique non-control input. We assert that
    // here and add it to the output.
    bool found_non_control_input = false;
    for (const string& input : n->input()) {
      if (!input.empty() && input[0] != '^') {
        DCHECK_EQ(found_non_control_input, false)
            << "RetVal node has more than one non-control input: "
            << absl::StrJoin(n->input(), ", ");
        strings::StrAppend(&out, n->input(0), ":", get_type_and_device(*n));
        found_non_control_input = true;
      }
    }
    DCHECK_EQ(found_non_control_input, true)
        << "RetVal did not have any non-control inputs: "
        << absl::StrJoin(n->input(), ", ");
  }
  strings::StrAppend(&out, ") {\n");
  for (size_t i = 0; i < body.size(); ++i) {
    strings::StrAppend(&out, "  ", Print(*body[i]), "\n");
  }
  strings::StrAppend(&out, "}\n");
  return out;
}

Status AddDefaultAttrs(const string& op,
                       const GetFunctionSignature& get_function,
                       AttrValueMap* attrs) {
```

```
  const OpDef* op_def = nullptr;
  TF_RETURN_IF_ERROR(get_function(op, &op_def));
  AttrSlice attr_slice(attrs);
  for (const auto& attr_def : op_def->attr()) {
    if (attr_def.has_default_value() && !attr_slice.Find(attr_def.name())) {
      if (!attrs->insert({attr_def.name(), attr_def.default_value()}).second) {
        return errors::Internal("Somehow duplicated: ", attr_def.name());
      }
    }
  }
  return Status::OK();
}

}  // end namespace

Status InstantiateFunction(const FunctionDef& fdef, AttrSlice attr_values,
                           GetFunctionSignature get_function,
                           InstantiationResult* result) {
  if (VLOG_IS_ON(5)) {
    const auto& signature = fdef.signature();
    VLOG(5) << "Instantiate function definition: name=" << signature.name()
            << " #input_args=" << signature.input_arg_size()
            << " #output_args=" << signature.output_arg_size()
            << " #control_output=" << signature.control_output_size();
    for (const auto& line : str_util::Split(Print(fdef), '\n')) {
      VLOG(5) << "|| " << line;
    }
  }

  const OpDef& sig = fdef.signature();
  TF_RETURN_IF_ERROR(ValidateSignatureWithAttrs(sig, attr_values));

  bool ints_on_device =
      fdef.attr().count(FunctionLibraryDefinition::kIntsOnDeviceAttr) != 0 &&
      fdef.attr().at(FunctionLibraryDefinition::kIntsOnDeviceAttr).b();

  FunctionInstantiationHelper helper(get_function, result);
  Status s;
  for (int i = 0, e = sig.input_arg_size(); i < e; ++i) {
    const OpDef::ArgDef& arg_def = sig.input_arg(i);
    auto it = fdef.arg_attr().find(i);
    const FunctionDef::ArgAttrs* arg_attrs =
        it != fdef.arg_attr().end() ? &it->second : nullptr;
    auto resource_id_it = fdef.resource_arg_unique_id().find(i);
    int64_t resource_arg_unique_id =
        resource_id_it != fdef.resource_arg_unique_id().end()
            ? resource_id_it->second
            : -1LL;
    s = helper.BuildInputArgIndex(arg_def, attr_values, arg_attrs,
```

```
765                                 ints_on_device, resource_arg_unique_id);
766
767     if (!s.ok()) {
768       errors::AppendToMessage(&s, "In ", Print(arg_def));
769       return s;
770     }
771   }
772
773   auto substitute = [attr_values, &sig](StringPiece name, AttrValue* val) {
774     // Look for a specified value...
775     if (const AttrValue* v = attr_values.Find(name)) {
776       *val = *v;
777       return true;
778     }
779     // .. and if not, then check for a default value.
780     if (const OpDef::AttrDef* attr = FindAttr(name, sig)) {
781       if (attr->has_default_value()) {
782         *val = attr->default_value();
783         return true;
784       }
785     }
786     // No luck finding a substitution.
787     return false;
788   };
789
790   // Makes a copy of all attrs in fdef and substitutes placeholders.
791   // After this step, every attr is bound to a concrete value.
792   std::vector<AttrValueMap> node_attrs;
793   node_attrs.resize(fdef.node_def_size());
794   for (int i = 0; i < fdef.node_def_size(); ++i) {
795     for (auto attr : fdef.node_def(i).attr()) {
796       if (!SubstitutePlaceholders(substitute, &attr.second)) {
797         return errors::InvalidArgument("Failed to bind all placeholders in ",
798                                        SummarizeAttrValue(attr.second));
799       }
800       if (!node_attrs[i].insert(attr).second) {
801         return errors::Internal("Somehow duplicated: ", attr.first);
802       }
803     }
804     TF_RETURN_IF_ERROR(
805         AddDefaultAttrs(fdef.node_def(i).op(), get_function, &node_attrs[i]));
806   }
807
808   for (int i = 0; i < fdef.node_def_size(); ++i) {
809     s = helper.BuildNodeOutputIndex(fdef.node_def(i), AttrSlice(&node_attrs[i]),
810                                     result->nodes.size() + i);
811     if (!s.ok()) {
812       errors::AppendToMessage(&s, "In ",
813                               FormatNodeDefForError(fdef.node_def(i)));
```

```
814       return s;
815     }
816   }
817   // Emits one node for each fdef.node_def.
818   for (int i = 0; i < fdef.node_def_size(); ++i) {
819     s = helper.InstantiateNode(fdef.node_def(i), AttrSlice(&node_attrs[i]));
820     if (!s.ok()) {
821       errors::AppendToMessage(&s, "In ",
822                               FormatNodeDefForError(fdef.node_def(i)));
823       return s;
824     }
825   }
826
827   // Emits nodes for the function's return values.
828   int ret_index = 0;
829   for (const OpDef::ArgDef& ret_def : sig.output_arg()) {
830     s = helper.AddReturnNode(ret_def, attr_values, fdef.ret(), ints_on_device,
831                              &ret_index);
832     if (!s.ok()) {
833       errors::AppendToMessage(&s, "In function output ", Print(ret_def));
834       return s;
835     }
836   }
837
838   // Adds the actual node inputs using the input indexes.
839   helper.AddNodeInputs();
840
841   return Status::OK();
842 }
843
844 string DebugString(const FunctionDef& func_def) { return Print(func_def); }
845
846 string DebugString(const GraphDef& instantiated_func_def) {
847   std::vector<const NodeDef*> ptrs;
848   for (const NodeDef& n : instantiated_func_def.node()) {
849     ptrs.push_back(&n);
850   }
851   return Print(ptrs);
852 }
853
854 string DebugString(gtl::ArraySlice<NodeDef> instantiated_func_nodes) {
855   std::vector<const NodeDef*> ptrs;
856   for (const NodeDef& n : instantiated_func_nodes) {
857     ptrs.push_back(&n);
858   }
859   return Print(ptrs);
860 }
861
862 string DebugStringWhole(const GraphDef& gdef) {
```

```
863   string ret;
864   for (const auto& fdef : gdef.library().function()) {
865     strings::StrAppend(&ret, Print(fdef));
866   }
867   strings::StrAppend(&ret, "\n");
868   for (const auto& ndef : gdef.node()) {
869     strings::StrAppend(&ret, Print(ndef), "\n");
870   }
871   return ret;
872 }
873
874 namespace {
875
876 // Returns the name -> attr mapping of fdef's attrs that have a value set. In
877 // Python, it's possible to access unset attrs, which returns a default value
878 // and adds an unset attr to the map.
879 std::map<string, AttrValue> GetSetAttrs(const FunctionDef& fdef) {
880   std::map<string, AttrValue> set_attrs;
881   for (const auto& pair : fdef.attr()) {
882     if (pair.second.value_case() != AttrValue::VALUE_NOT_SET) {
883       set_attrs[pair.first] = pair.second;
884     }
885   }
886   return set_attrs;
887 }
888
889 }  // end namespace
890
891 bool FunctionDefsEqual(const FunctionDef& f1, const FunctionDef& f2) {
892   if (!OpDefEqual(f1.signature(), f2.signature())) return false;
893
894   std::map<string, AttrValue> f1_attrs = GetSetAttrs(f1);
895   std::map<string, AttrValue> f2_attrs = GetSetAttrs(f2);
896   if (f1_attrs.size() != f2_attrs.size()) return false;
897   for (const auto& iter1 : f1_attrs) {
898     auto iter2 = f2_attrs.find(iter1.first);
899     if (iter2 == f2_attrs.end()) return false;
900     if (!AreAttrValuesEqual(iter1.second, iter2->second)) return false;
901   }
902
903   if (!EqualRepeatedNodeDef(f1.node_def(), f2.node_def(), nullptr)) {
904     return false;
905   }
906
907   std::map<string, string> ret1(f1.ret().begin(), f1.ret().end());
908   std::map<string, string> ret2(f2.ret().begin(), f2.ret().end());
909   if (ret1 != ret2) return false;
910
911   std::map<string, string> control_ret1(f1.control_ret().begin(),
```

```
                                        f1.control_ret().end());
  std::map<string, string> control_ret2(f2.control_ret().begin(),
                                        f2.control_ret().end());
  if (control_ret1 != control_ret2) return false;

  return true;
}

uint64 FunctionDefHash(const FunctionDef& fdef) {
  // signature
  uint64 h = OpDefHash(fdef.signature());

  // attrs
  std::map<string, AttrValue> attrs = GetSetAttrs(fdef);
  for (const auto& p : attrs) {
    h = Hash64(p.first.data(), p.first.size(), h);
    h = Hash64Combine(AttrValueHash(p.second), h);
  }

  // node defs
  h = Hash64Combine(RepeatedNodeDefHash(fdef.node_def()), h);

  // output names
  std::map<string, string> ret(fdef.ret().begin(), fdef.ret().end());
  for (const auto& p : ret) {
    h = Hash64(p.first.data(), p.first.size(), h);
    h = Hash64(p.second.data(), p.second.size(), h);
  }

  // control output names
  std::map<string, string> control_ret(fdef.control_ret().begin(),
                                       fdef.control_ret().end());
  for (const auto& p : control_ret) {
    h = Hash64(p.first.data(), p.first.size(), h);
    h = Hash64(p.second.data(), p.second.size(), h);
  }

  return h;
}

static constexpr const char* const kExecutorAttr = "_executor";

/* static */
string FunctionLibraryRuntime::ExecutorType(const InstantiateOptions& options,
                                            AttrSlice attrs) {
  if (!options.executor_type.empty()) {
    return options.executor_type;
  } else if (const AttrValue* executor_attr = attrs.Find(kExecutorAttr)) {
    return executor_attr->s();
```

```
  } else {
    return string();
  }
}

namespace {
class AttrKeyAndValue {
 public:
  enum ValueRepresentationOp {
    kRaw,
    kCEscape,
  };
  AttrKeyAndValue(absl::string_view key_name, int key_suffix, string value,
                  ValueRepresentationOp value_op = kRaw)
      : key_name_(key_name),
        key_suffix_(key_suffix),
        value_op_(value_op),
        value_(std::move(value)) {}

  bool operator<(const AttrKeyAndValue& b) const {
    if (key_name_ != b.key_name_) {
      return key_name_ < b.key_name_;
    } else if (key_suffix_ != b.key_suffix_) {
      return key_suffix_ < b.key_suffix_;
    } else {
      return value_ < b.value_;
    }
  }

  void AppendTo(bool first, string* s) const {
    absl::string_view v;
    bool add_escaped = false;
    if ((value_op_ == kCEscape) && NeedsEscaping(value_)) {
      // Use CEscape call below
      add_escaped = true;
    } else {
      // Add raw value contents directly
      v = value_;
    }
    if (key_suffix_ >= 0) {
      strings::StrAppend(s, first ? "" : ",", key_name_, key_suffix_, "=", v);
    } else {
      strings::StrAppend(s, first ? "" : ",", key_name_, "=", v);
    }
    if (add_escaped) {
      strings::StrAppend(s, absl::CEscape(value_));
    }
  }

```

```
 private:
  static bool NeedsEscaping(const string& s) {
    for (auto c : s) {
      if (!isalnum(c) && (c != ' ')) {
        return true;
      }
    }
    return false;
  }

  absl::string_view key_name_;
  int key_suffix_;  // -1 if missing
  ValueRepresentationOp value_op_;
  string value_;
};
}  // namespace

string GetFunctionResourceInputDevice(
    const Tensor& input, const int arg_index, const FunctionDef& function_def,
    absl::flat_hash_map<string, std::vector<string>>* composite_devices) {
  const auto& handles = input.flat<ResourceHandle>();
  const ResourceHandle& handle0 = handles(0);
  string composite_device;
  auto iter = function_def.arg_attr().find(arg_index);
  if (iter != function_def.arg_attr().end()) {
    auto arg_attr = iter->second.attr().find("_composite_device");
    if (arg_attr != iter->second.attr().end()) {
      composite_device = arg_attr->second.s();
    }
  }
  if (!composite_device.empty()) {
    if (composite_devices->find(composite_device) == composite_devices->end()) {
      for (int i = 0; i < handles.size(); ++i) {
        (*composite_devices)[composite_device].push_back(handles(i).device());
      }
    }
    return composite_device;
  } else {
    return handle0.device();
  }
}

string Canonicalize(const string& funcname, AttrSlice attrs,
                    const FunctionLibraryRuntime::InstantiateOptions& options) {
  absl::InlinedVector<AttrKeyAndValue, 8> entries;
  entries.reserve(attrs.size() + static_cast<int>(!options.target.empty()) +
                  options.input_devices.size());
  for (const auto& p : attrs) {
    if (p.first != kExecutorAttr) {
```

```
1059        entries.push_back(AttrKeyAndValue(
1060            p.first, -1, Print(p.second, /*hash_string_attrs=*/true)));
1061      }
1062    }
1063    if (!options.target.empty()) {
1064      entries.push_back(AttrKeyAndValue("_target", -1, options.target,
1065                                        AttrKeyAndValue::kCEscape));
1066    }
1067    for (int i = 0; i < options.input_devices.size(); ++i) {
1068      entries.push_back(AttrKeyAndValue("_input_dev", i, options.input_devices[i],
1069                                        AttrKeyAndValue::kCEscape));
1070    }
1071    for (int i = 0; i < options.output_devices.size(); ++i) {
1072      entries.push_back(AttrKeyAndValue("_output_dev", i,
1073                                        options.output_devices[i],
1074                                        AttrKeyAndValue::kCEscape));
1075    }
1076    for (const auto& iter : options.input_resource_dtypes_and_shapes) {
1077      entries.push_back(AttrKeyAndValue("_input_resource_dtype", iter.first,
1078                                        DataTypeString(iter.second.dtype)));
1079      entries.push_back(AttrKeyAndValue("_input_resource_shape", iter.first,
1080                                        iter.second.shape.DebugString(),
1081                                        AttrKeyAndValue::kCEscape));
1082    }
1083    if (options.lib_def) {
1084      entries.push_back(AttrKeyAndValue(
1085          "_lib_def", -1,
1086          absl::StrCat("", reinterpret_cast<uintptr_t>(options.lib_def))));
1087    }
1088    if (!options.state_handle.empty()) {
1089      entries.push_back(
1090          AttrKeyAndValue("_state_handle", -1, options.state_handle));
1091    }
1092    string executor_type = FunctionLibraryRuntime::ExecutorType(options, attrs);
1093    if (!executor_type.empty()) {
1094      entries.push_back(AttrKeyAndValue(kExecutorAttr, -1, executor_type));
1095    }
1096    if (options.config_proto.ByteSize() > 0) {
1097      string config_proto_serialized;
1098      SerializeToStringDeterministic(options.config_proto,
1099                                     &config_proto_serialized);
1100      entries.push_back(AttrKeyAndValue("_config_proto", -1,
1101                                        config_proto_serialized,
1102                                        AttrKeyAndValue::kCEscape));
1103    }
1104    std::sort(entries.begin(), entries.end());
1105    string result = strings::StrCat(funcname, "[");
1106    bool first = true;
1107    for (const auto& entry : entries) {
```

```
1108      entry.AppendTo(first, &result);
1109      first = false;
1110    }
1111    result += "]";
1112    return result;
1113  }
1114
1115  string Canonicalize(const string& funcname, AttrSlice attrs) {
1116    static const FunctionLibraryRuntime::InstantiateOptions* kEmptyOptions =
1117        new FunctionLibraryRuntime::InstantiateOptions;
1118    return Canonicalize(funcname, attrs, *kEmptyOptions);
1119  }
1120
1121  FunctionCallFrame::FunctionCallFrame(DataTypeSlice arg_types,
1122                                       DataTypeSlice ret_types)
1123      : arg_types_(arg_types.begin(), arg_types.end()),
1124        ret_types_(ret_types.begin(), ret_types.end()) {
1125    args_.resize(arg_types_.size());
1126    rets_.resize(ret_types_.size());
1127  }
1128
1129  FunctionCallFrame::~FunctionCallFrame() {}
1130
1131  Status FunctionCallFrame::SetArgs(gtl::ArraySlice<Tensor> args) {
1132    // Input type checks.
1133    if (args.size() != arg_types_.size()) {
1134      return errors::InvalidArgument("Expects ", arg_types_.size(),
1135                                     " arguments, but ", args.size(),
1136                                     " is provided");
1137    }
1138    for (size_t i = 0; i < args.size(); ++i) {
1139      if (arg_types_[i] != args[i].dtype()) {
1140        return errors::InvalidArgument(
1141            "Expects arg[", i, "] to be ", DataTypeString(arg_types_[i]), " but ",
1142            DataTypeString(args[i].dtype()), " is provided");
1143      }
1144      args_[i] = args[i];
1145    }
1146    return Status::OK();
1147  }
1148
1149  Status FunctionCallFrame::GetRetvals(std::vector<Tensor>* rets) const {
1150    rets->clear();
1151    rets->reserve(rets_.size());
1152    for (size_t i = 0; i < rets_.size(); ++i) {
1153      const auto& item = rets_[i];
1154      if (item.has_val) {
1155        rets->push_back(item.val);
1156      } else {
```

```
1157        return errors::Internal("Retval[", i, "] does not have value");
1158      }
1159    }
1160    return Status::OK();
1161  }
1162
1163  Status FunctionCallFrame::ConsumeRetvals(std::vector<Tensor>* rets,
1164                                           bool allow_dead_tensors) {
1165    rets->clear();
1166    rets->reserve(rets_.size());
1167    for (size_t i = 0; i < rets_.size(); ++i) {
1168      if (rets_[i].has_val) {
1169        rets->emplace_back(std::move(rets_[i].val));
1170      } else if (allow_dead_tensors) {
1171        rets->emplace_back();
1172      } else {
1173        return errors::Internal("Retval[", i, "] does not have value");
1174      }
1175    }
1176    return Status::OK();
1177  }
1178
1179  Status FunctionCallFrame::GetArg(int index, const Tensor** val) {
1180    if (index < 0 || static_cast<size_t>(index) >= args_.size()) {
1181      return errors::InvalidArgument("GetArg ", index, " is not within [0, ",
1182                                     args_.size(), ")");
1183    }
1184    *val = &args_[index];
1185    return Status::OK();
1186  }
1187
1188  Status FunctionCallFrame::SetRetval(int index, const Tensor& val) {
1189    if (index < 0 || static_cast<size_t>(index) >= rets_.size()) {
1190      return errors::InvalidArgument("SetRetval ", index, " is not within [0, ",
1191                                     rets_.size(), ")");
1192    }
1193    if (val.dtype() != ret_types_[index]) {
1194      return errors::InvalidArgument(
1195          "Expects ret[", index, "] to be ", DataTypeString(ret_types_[index]),
1196          ", but ", DataTypeString(val.dtype()), " is provided.");
1197    }
1198    Retval* item = &rets_[index];
1199    if (!item->has_val) {
1200      item->has_val = true;
1201      item->val = val;
1202    } else {
1203      return errors::Internal("Retval[", index, "] has already been set.");
1204    }
1205    return Status::OK();
```

```
}

FunctionLibraryDefinition::FunctionDefAndOpRegistration::
    FunctionDefAndOpRegistration(const FunctionDef& fdef_in,
                                 const StackTracesMap& stack_traces)
    : fdef(fdef_in),
      // Exact shape inference for functions is handled by ShapeRefiner.
      // Here we pass a dummy shape inference function for legacy code paths.
      op_registration_data(fdef.signature(), shape_inference::UnknownShape,
                           true /* is_function */),
      stack_traces(stack_traces) {}

FunctionLibraryDefinition::FunctionLibraryDefinition(
    const FunctionLibraryDefinition& other)
    : default_registry_(other.default_registry_) {
  tf_shared_lock l(other.mu_);
  function_defs_ = other.function_defs_;
  func_grad_ = other.func_grad_;
}

FunctionLibraryDefinition::FunctionLibraryDefinition(
    const OpRegistryInterface* default_registry,
    const FunctionDefLibrary& def_lib)
    : default_registry_(default_registry),
      function_defs_(def_lib.function_size()) {
  for (const auto& fdef : def_lib.function()) {
    // The latter function definition wins.
    auto& ptr = function_defs_[fdef.signature().name()];
    ptr.reset(new FunctionDefAndOpRegistration(fdef));
  }
  for (const auto& grad : def_lib.gradient()) {
    func_grad_[grad.function_name()] = grad.gradient_func();
  }
}

FunctionLibraryDefinition::~FunctionLibraryDefinition() {}

bool FunctionLibraryDefinition::Contains(const string& func) const {
  tf_shared_lock l(mu_);
  return function_defs_.find(func) != function_defs_.end();
}

const FunctionDef* FunctionLibraryDefinition::Find(const string& func) const {
  tf_shared_lock l(mu_);
  auto result = FindHelper(func);
  if (result) {
    return &result->fdef;
  } else {
    return nullptr;
```

```
1255    }
1256  }
1257
1258  std::shared_ptr<FunctionLibraryDefinition::FunctionDefAndOpRegistration>
1259  FunctionLibraryDefinition::FindHelper(const string& func) const {
1260    auto iter = function_defs_.find(func);
1261    if (iter == function_defs_.end()) {
1262      return nullptr;
1263    } else {
1264      return iter->second;
1265    }
1266  }
1267
1268  Status FunctionLibraryDefinition::AddFunctionDef(
1269      const FunctionDef& fdef, const StackTracesMap& stack_traces) {
1270    mutex_lock l(mu_);
1271    bool added;
1272    return AddFunctionDefHelper(fdef, stack_traces, &added);
1273  }
1274
1275  Status FunctionLibraryDefinition::AddFunctionDefHelper(
1276      const FunctionDef& fdef, const StackTracesMap& stack_traces, bool* added) {
1277    *added = false;
1278    std::shared_ptr<FunctionDefAndOpRegistration>& entry =
1279        function_defs_[fdef.signature().name()];
1280    if (entry) {
1281      if (!FunctionDefsEqual(entry->fdef, fdef)) {
1282        return errors::InvalidArgument(
1283            "Cannot add function '", fdef.signature().name(),
1284            "' because a different function with the same name already "
1285            "exists.");
1286      }
1287      // Ignore duplicate FunctionDefs.
1288      return Status::OK();
1289    }
1290    const OpDef* op_def;
1291    if (default_registry_->LookUpOpDef(fdef.signature().name(), &op_def).ok()) {
1292      return errors::InvalidArgument(
1293          "Cannot add function '", fdef.signature().name(),
1294          "' because an op with the same name already exists.");
1295    }
1296    entry = std::make_shared<FunctionDefAndOpRegistration>(fdef, stack_traces);
1297    *added = true;
1298    return Status::OK();
1299  }
1300
1301  Status FunctionLibraryDefinition::AddHelper(
1302      std::shared_ptr<FunctionDefAndOpRegistration> registration, bool* added) {
1303    *added = false;
```

```
  std::shared_ptr<FunctionDefAndOpRegistration>& entry =
      function_defs_[registration->fdef.signature().name()];
  if (entry) {
    if (!FunctionDefsEqual(entry->fdef, registration->fdef)) {
      return errors::InvalidArgument(
          "Cannot add function '", registration->fdef.signature().name(),
          "' because a different function with the same name already "
          "exists.");
    }
    // Ignore duplicate FunctionDefs.
    return Status::OK();
  }
  const OpDef* op_def;
  if (default_registry_
          ->LookUpOpDef(registration->fdef.signature().name(), &op_def)
          .ok()) {
    return errors::InvalidArgument(
        "Cannot add function '", registration->fdef.signature().name(),
        "' because an op with the same name already exists.");
  }
  entry = std::move(registration);
  *added = true;
  return Status::OK();
}

Status FunctionLibraryDefinition::CopyFunctionDefFrom(
    const string& func, const FunctionLibraryDefinition& other) {
  if (default_registry_ != other.default_registry_) {
    return errors::InvalidArgument(
        "Cannot copy function '", func,
        "' because CopyFunctionDefFrom() requires that both libraries have the "
        "same default registry.");
  }
  std::shared_ptr<FunctionDefAndOpRegistration> function_def;
  {
    tf_shared_lock l(other.mu_);
    function_def = other.FindHelper(func);
  }
  if (!function_def) {
    return errors::InvalidArgument(
        "Cannot copy function '", func,
        "' because no function with that name exists in the other library.");
  }
  {
    mutex_lock l(mu_);
    std::shared_ptr<FunctionDefAndOpRegistration>& entry = function_defs_[func];
    if (entry) {
      if (!FunctionDefsEqual(entry->fdef, function_def->fdef)) {
        return errors::InvalidArgument(
```

```
1353              "Cannot copy function '", func,
1354              "' because a different function with the same name already "
1355              "exists.");
1356        }
1357      } else {
1358        entry = std::move(function_def);
1359      }
1360    }
1361    return Status::OK();
1362  }
1363
1364  Status FunctionLibraryDefinition::AddGradientDef(const GradientDef& grad) {
1365    mutex_lock l(mu_);
1366    bool added;
1367    return AddGradientDefHelper(grad, &added);
1368  }
1369
1370  Status FunctionLibraryDefinition::AddGradientDefHelper(const GradientDef& grad,
1371                                                         bool* added) {
1372    *added = false;
1373    string* entry = &func_grad_[grad.function_name()];
1374    if (!entry->empty()) {
1375      if (*entry != grad.gradient_func()) {
1376        return errors::InvalidArgument(
1377            "Cannot assign gradient function '", grad.gradient_func(), "' to '",
1378            grad.function_name(), "' because it already has gradient function ",
1379            "'", *entry, "'");
1380      }
1381      // Ignore duplicate GradientDefs
1382      return Status::OK();
1383    }
1384    *entry = grad.gradient_func();
1385    *added = true;
1386    return Status::OK();
1387  }
1388
1389  Status FunctionLibraryDefinition::AddLibrary(
1390      const FunctionLibraryDefinition& other) {
1391    // Clone `other` to ensure thread-safety (grabbing `other`'s lock for
1392    // the duration of the function could lead to deadlock).
1393    FunctionLibraryDefinition clone(other);
1394    mutex_lock l(mu_);
1395    mutex_lock l2(clone.mu_);
1396    // Remember the funcs and grads that we added successfully so that
1397    // we can roll them back on error.
1398    std::vector<string> funcs;
1399    std::vector<string> funcs_with_grads;
1400    Status s;
1401    bool added;
```

```
1402    for (auto iter : clone.function_defs_) {
1403      s = AddHelper(iter.second, &added);
1404      if (!s.ok()) {
1405        Status remove_status = Remove(funcs, funcs_with_grads);
1406        if (!remove_status.ok()) {
1407          return remove_status;
1408        }
1409        return s;
1410      }
1411      if (added) {
1412        funcs.push_back(iter.second->fdef.signature().name());
1413      }
1414    }
1415    for (auto iter : clone.func_grad_) {
1416      GradientDef grad;
1417      grad.set_function_name(iter.first);
1418      grad.set_gradient_func(iter.second);
1419      s = AddGradientDefHelper(grad, &added);
1420      if (!s.ok()) {
1421        Status remove_status = Remove(funcs, funcs_with_grads);
1422        if (!remove_status.ok()) {
1423          return remove_status;
1424        }
1425        return s;
1426      }
1427      if (added) {
1428        funcs_with_grads.push_back(grad.function_name());
1429      }
1430    }
1431    return Status::OK();
1432  }
1433
1434  Status FunctionLibraryDefinition::AddLibrary(
1435      const FunctionDefLibrary& lib_def) {
1436    // Remember the funcs and grads that we added successfully so that
1437    // we can roll them back on error.
1438    mutex_lock l(mu_);
1439    std::vector<string> funcs;
1440    std::vector<string> funcs_with_grads;
1441    Status s;
1442    bool added;
1443    for (const FunctionDef& fdef : lib_def.function()) {
1444      s = AddFunctionDefHelper(fdef, /*stack_traces=*/{}, &added);
1445      if (!s.ok()) {
1446        Status remove_status = Remove(funcs, funcs_with_grads);
1447        if (!remove_status.ok()) {
1448          return remove_status;
1449        }
1450        return s;
```

```
1451      }
1452      if (added) {
1453        funcs.push_back(fdef.signature().name());
1454      }
1455    }
1456    for (const GradientDef& grad : lib_def.gradient()) {
1457      s = AddGradientDefHelper(grad, &added);
1458      if (!s.ok()) {
1459        Status remove_status = Remove(funcs, funcs_with_grads);
1460        if (!remove_status.ok()) {
1461          return remove_status;
1462        }
1463        return s;
1464      }
1465      if (added) {
1466        funcs_with_grads.push_back(grad.function_name());
1467      }
1468    }
1469    return Status::OK();
1470  }
1471
1472  Status FunctionLibraryDefinition::ReplaceFunction(
1473      const string& func, const FunctionDef& fdef,
1474      const StackTracesMap& stack_traces) {
1475    mutex_lock l(mu_);
1476    bool added;
1477    TF_RETURN_IF_ERROR(RemoveFunctionHelper(func));
1478    TF_RETURN_IF_ERROR(AddFunctionDefHelper(fdef, stack_traces, &added));
1479    return Status::OK();
1480  }
1481
1482  Status FunctionLibraryDefinition::ReplaceGradient(const GradientDef& grad) {
1483    mutex_lock l(mu_);
1484    bool added;
1485    TF_RETURN_IF_ERROR(RemoveGradient(grad.function_name()));
1486    TF_RETURN_IF_ERROR(AddGradientDefHelper(grad, &added));
1487    return Status::OK();
1488  }
1489
1490  Status FunctionLibraryDefinition::RemoveFunction(const string& func) {
1491    mutex_lock l(mu_);
1492    TF_RETURN_IF_ERROR(RemoveFunctionHelper(func));
1493    return Status::OK();
1494  }
1495
1496  Status FunctionLibraryDefinition::RemoveFunctionHelper(const string& func) {
1497    const auto& i = function_defs_.find(func);
1498    if (i == function_defs_.end()) {
1499      return errors::InvalidArgument("Tried to remove non-existent function '",
```

```
1500                                     func, "'.");
1501   }
1502   function_defs_.erase(i);
1503   return Status::OK();
1504 }
1505
1506 void FunctionLibraryDefinition::Clear() {
1507   mutex_lock l(mu_);
1508   function_defs_.clear();
1509   func_grad_.clear();
1510 }
1511
1512 Status FunctionLibraryDefinition::RemoveGradient(const string& func) {
1513   const auto& i = func_grad_.find(func);
1514   if (i == func_grad_.end()) {
1515     return errors::InvalidArgument("Tried to remove non-existent gradient '",
1516                                     func, "'.");
1517   }
1518   func_grad_.erase(i);
1519   return Status::OK();
1520 }
1521
1522 Status FunctionLibraryDefinition::Remove(
1523     const std::vector<string>& funcs,
1524     const std::vector<string>& funcs_with_grads) {
1525   Status s;
1526   for (const string& f : funcs) {
1527     s = RemoveFunctionHelper(f);
1528     if (!s.ok()) {
1529       return s;
1530     }
1531   }
1532   for (const string& f : funcs_with_grads) {
1533     s = RemoveGradient(f);
1534     if (!s.ok()) {
1535       return s;
1536     }
1537   }
1538   return Status::OK();
1539 }
1540
1541 string FunctionLibraryDefinition::FindGradient(const string& func) const {
1542   tf_shared_lock l(mu_);
1543   return gtl::FindWithDefault(func_grad_, func, "");
1544 }
1545
1546 string FunctionLibraryDefinition::FindGradientHelper(const string& func) const {
1547   return gtl::FindWithDefault(func_grad_, func, "");
1548 }
```

```
1549
1550  Status FunctionLibraryDefinition::LookUp(
1551      const string& op, const OpRegistrationData** op_reg_data) const {
1552    tf_shared_lock l(mu_);
1553    auto iter = function_defs_.find(op);
1554    if (iter != function_defs_.end()) {
1555      *op_reg_data = &iter->second->op_registration_data;
1556      return Status::OK();
1557    }
1558    return default_registry_->LookUp(op, op_reg_data);
1559  }
1560
1561  string FunctionLibraryDefinition::UniqueFunctionName(StringPiece prefix) const {
1562    tf_shared_lock l(mu_);
1563    int index = 0;
1564    string name = strings::StrCat(prefix, index);
1565    while (function_defs_.find(name) != function_defs_.end()) {
1566      ++index;
1567      name = strings::StrCat(prefix, index);
1568    }
1569    return name;
1570  }
1571
1572  const FunctionDef* FunctionLibraryDefinition::GetAttrImpl(
1573      const NodeDef& ndef) const {
1574    if (ndef.op() != kGradientOp) {
1575      // If 'ndef' calls a function and the function's def has the attr,
1576      // returns it.
1577      return Find(ndef.op());
1578    }
1579
1580    // If ndef is SymbolicGradient[f=Foo], we use Foo's gradient or
1581    // Foo's attributes.
1582    const NameAttrList* forward_func_attrs;
1583    if (!TryGetNodeAttr(ndef, kFuncAttr, &forward_func_attrs)) {
1584      return nullptr;
1585    }
1586    const string& func_name = forward_func_attrs->name();
1587    {
1588      tf_shared_lock l(mu_);
1589      const string& grad_name = FindGradientHelper(func_name);
1590      // If 'func' has a user-defined gradient function, uses the grad
1591      // function's attrs to see if noinline is specified. Otherwise,
1592      // uses func's attrs.
1593      if (!grad_name.empty()) {
1594        if (const auto helper = FindHelper(grad_name)) {
1595          return &(helper->fdef);
1596        } else {
1597          return nullptr;
```

```
      }
    }
    if (const auto helper = FindHelper(func_name)) {
      return &(helper->fdef);
    } else {
      return nullptr;
    }
  }
}

std::vector<string> FunctionLibraryDefinition::ListFunctionNames() const {
  std::vector<string> function_names;
  tf_shared_lock l(mu_);
  function_names.reserve(function_defs_.size());
  for (const auto& it : function_defs_) {
    function_names.emplace_back(it.first);
  }
  return function_names;
}

FunctionDefLibrary FunctionLibraryDefinition::ToProto() const {
  FunctionDefLibrary lib;
  tf_shared_lock l(mu_);
  for (const auto& f : function_defs_) {
    *lib.add_function() = f.second->fdef;
  }
  for (const auto& g : func_grad_) {
    GradientDef* gd = lib.add_gradient();
    gd->set_function_name(g.first);
    gd->set_gradient_func(g.second);
  }
  return lib;
}

template <typename T>
Status FunctionLibraryDefinition::GetAttr(const NodeDef& ndef,
                                          const string& attr, T* value) const {
  const FunctionDef* fdef = GetAttrImpl(ndef);
  if (fdef && TryGetNodeAttr(AttrSlice(&fdef->attr()), attr, value)) {
    return Status::OK();
  }
  return errors::InvalidArgument("Attr ", attr, " is not defined.");
}

template <typename T>
Status FunctionLibraryDefinition::GetAttr(const Node& node, const string& attr,
                                          T* value) const {
  return GetAttr(node.def(), attr, value);
}
```

```
1647
1648  #define GET_ATTR(T)                                                    \
1649    template Status FunctionLibraryDefinition::GetAttr(const Node&,      \
1650                                                       const string&, T*) const; \
1651    template Status FunctionLibraryDefinition::GetAttr(const NodeDef&,   \
1652                                                       const string&, T*) const;
1653  GET_ATTR(string)
1654  GET_ATTR(bool)
1655  #undef GET_ATTR
1656
1657  namespace {
1658
1659  constexpr char kApiImplements[] = "api_implements";
1660
1661  std::set<string> ReachableFunctions(
1662      const FunctionLibraryDefinition& flib,
1663      const protobuf::RepeatedPtrField<NodeDef>& nodes) {
1664    // Functions that are reachable from the graph.
1665    std::set<string> reachable_funcs;
1666
1667    // For any functions, if it has attribute "api_implements" =
1668    // "some_interface" and it is reachable, then it means any other
1669    // function with same attribute name and value could also be potentially
1670    // reachable, eg via implementation_selector swapping the nodedef.
1671    absl::flat_hash_set<string> reachable_api_interface;
1672
1673    // Functions might be reachable from the nested function calls, so we keep a
1674    // queue of functions that we have to check.
1675    gtl::InlinedVector<const FunctionDef*, 4> func_queue;
1676
1677    // Add reachable and not already processed functions to the functions queue.
1678    const auto add_to_func_queue = [&](const string& func_name) {
1679      const FunctionDef* func = flib.Find(func_name);
1680      if (func && reachable_funcs.find(func_name) == reachable_funcs.end()) {
1681        func_queue.push_back(func);
1682      }
1683    };
1684
1685    // If any function with certain API name is reachable, all the other functions
1686    // with same API name should also be checked.
1687    const auto add_function_with_api_interface = [&](const string& api_name) {
1688      if (!reachable_api_interface.contains(api_name)) {
1689        reachable_api_interface.insert(api_name);
1690        for (const auto& func_name : flib.ListFunctionNames()) {
1691          const auto& func_def = flib.Find(func_name);
1692          const auto attr_it = func_def->attr().find(kApiImplements);
1693          if (attr_it != func_def->attr().end() &&
1694              attr_it->second.s() == api_name) {
1695            add_to_func_queue(func_name);
```

```
1696        }
1697      }
1698    }
1699  };
1700
1701  // Add all the functions that are reachable from the given node to the queue.
1702  const auto process_node = [&](const NodeDef& node) {
1703    // Node itself can be a call to the function.
1704    add_to_func_queue(node.op());
1705
1706    // Or node can have an attribute referencing a function.
1707    for (const auto& attr : node.attr()) {
1708      const auto& attr_value = attr.second;
1709
1710      // 1. AttrValue.func
1711      if (attr_value.has_func()) {
1712        add_to_func_queue(attr_value.func().name());
1713      }
1714
1715      // 2. AttrValue.ListValue.func
1716      if (attr_value.has_list()) {
1717        for (const auto& func : attr_value.list().func()) {
1718          add_to_func_queue(func.name());
1719        }
1720      }
1721    }
1722  };
1723
1724  // Add all functions that are directly called from the optimized graph.
1725  std::for_each(nodes.begin(), nodes.end(), process_node);
1726
1727  // Process all reachable functions.
1728  while (!func_queue.empty()) {
1729    const FunctionDef* func = func_queue.back();
1730    func_queue.pop_back();
1731
1732    const string& func_name = func->signature().name();
1733    reachable_funcs.insert(func_name);
1734
1735    const auto attr_it = func->attr().find(kApiImplements);
1736    if (attr_it != func->attr().end()) {
1737      add_function_with_api_interface(attr_it->second.s());
1738    }
1739
1740    // Find all the functions called from the function body.
1741    const auto& func_body = func->node_def();
1742    std::for_each(func_body.begin(), func_body.end(), process_node);
1743
1744    // Check if the function has a registered gradient.
```

```
1745      const string grad_func_name = flib.FindGradient(func_name);
1746      if (!grad_func_name.empty()) add_to_func_queue(grad_func_name);
1747    }
1748
1749    return reachable_funcs;
1750  }
1751
1752  FunctionLibraryDefinition ReachableFunctionLibraryDefinition(
1753      const FunctionLibraryDefinition& flib,
1754      const protobuf::RepeatedPtrField<NodeDef>& nodes) {
1755    std::set<string> reachable_funcs = ReachableFunctions(flib, nodes);
1756
1757    FunctionLibraryDefinition reachable_flib(flib.default_registry(),
1758                                             FunctionDefLibrary());
1759
1760    for (const string& func_name : reachable_funcs) {
1761      // This should never fail, because we copy functions from a valid flib and
1762      // use the same default registry.
1763      Status added = reachable_flib.CopyFunctionDefFrom(func_name, flib);
1764      TF_DCHECK_OK(added);
1765
1766      const string grad_func_name = flib.FindGradient(func_name);
1767      if (!grad_func_name.empty()) {
1768        GradientDef grad;
1769        grad.set_function_name(func_name);
1770        grad.set_gradient_func(grad_func_name);
1771        // It can only fail if function already has a gradient function.
1772        const Status added_grad = reachable_flib.AddGradientDef(grad);
1773        TF_DCHECK_OK(added_grad);
1774      }
1775    }
1776
1777    return reachable_flib;
1778  }
1779
1780  string AllocatorAttributesToString(
1781      const std::vector<AllocatorAttributes>& attrs) {
1782    string result("[");
1783    // AllocatorAttribute::DebugString produces around 85 bytes now.
1784    result.reserve(100 * attrs.size());
1785    for (const AllocatorAttributes& attr : attrs) {
1786      result.append(attr.DebugString());
1787      result.append(", ");
1788    }
1789    if (!attrs.empty()) {
1790      result.resize(result.size() - 2);
1791    }
1792    result.append("]");
1793    return result;
```

```
}

const char* IsSet(void* ptr) { return ptr == nullptr ? "unset" : "set"; }

}  // namespace

FunctionLibraryDefinition FunctionLibraryDefinition::ReachableDefinitions(
    const GraphDef& graph) const {
  return ReachableFunctionLibraryDefinition(*this, graph.node());
}

FunctionLibraryDefinition FunctionLibraryDefinition::ReachableDefinitions(
    const FunctionDef& func) const {
  return ReachableFunctionLibraryDefinition(*this, func.node_def());
}

string FunctionLibraryRuntime::Options::DebugString() const {
  return absl::StrCat(
      "FLR::Options(step_id=", step_id, " rendezvous=", IsSet(rendezvous),
      " cancellation_manager=", IsSet(cancellation_manager),
      " collective_executor=", IsSet(collective_executor),
      " step_container=", IsSet(step_container),
      " stats_collector=", IsSet(stats_collector), " runner=", IsSet(runner),
      " remote_execution=", remote_execution, " source_device=", source_device,
      " create_rendezvous=", create_rendezvous,
      " allow_dead_tensors=", allow_dead_tensors,
      " args_alloc_attrs=", AllocatorAttributesToString(args_alloc_attrs),
      " rets_alloc_attrs=", AllocatorAttributesToString(rets_alloc_attrs), ")");
}

void FunctionDefHelper::AttrValueWrapper::InitFromString(StringPiece val) {
  if (val.size() >= 2 && val[0] == '$') {
    proto.set_placeholder(val.data() + 1, val.size() - 1);
  } else {
    SetAttrValue(val, &proto);
  }
}

FunctionDefHelper::AttrValueWrapper FunctionDefHelper::FunctionRef(
    const string& name,
    gtl::ArraySlice<std::pair<string, AttrValueWrapper>> attrs) {
  AttrValueWrapper ret;
  ret.proto.mutable_func()->set_name(name);
  for (const auto& a : attrs) {
    ret.proto.mutable_func()->mutable_attr()->insert({a.first, a.second.proto});
  }
  return ret;
}

```

```
1843 NodeDef FunctionDefHelper::Node::ToNodeDef() const {
1844   NodeDef n;
1845   n.set_op(this->op);
1846   n.set_name(GetName());
1847   for (const auto& a : this->attr) {
1848     n.mutable_attr()->insert({a.first, a.second.proto});
1849   }
1850   for (const string& a : this->arg) {
1851     n.add_input(a);
1852   }
1853   for (const string& d : this->dep) {
1854     n.add_input(strings::StrCat("^", d));
1855   }
1856   if (!this->device.empty()) {
1857     n.set_device(this->device);
1858   }
1859   if (!this->original_node_names.empty()) {
1860     *n.mutable_experimental_debug_info()->mutable_original_node_names() = {
1861         this->original_node_names.begin(), this->original_node_names.end()};
1862   }
1863   if (!this->original_func_names.empty()) {
1864     *n.mutable_experimental_debug_info()->mutable_original_func_names() = {
1865         this->original_func_names.begin(), this->original_func_names.end()};
1866   }
1867   return n;
1868 }
1869
1870 /* static */
1871 FunctionDef FunctionDefHelper::Create(
1872     const string& function_name, gtl::ArraySlice<string> in_def,
1873     gtl::ArraySlice<string> out_def, gtl::ArraySlice<string> attr_def,
1874     gtl::ArraySlice<Node> node_def,
1875     gtl::ArraySlice<std::pair<string, string>> ret_def,
1876     gtl::ArraySlice<std::pair<string, string>> control_ret_def) {
1877   FunctionDef fdef;
1878
1879   // Signature
1880   OpDefBuilder b(function_name);
1881   for (const auto& i : in_def) b.Input(i);
1882   for (const auto& o : out_def) b.Output(o);
1883   for (const auto& a : attr_def) b.Attr(a);
1884   for (const auto& c : control_ret_def) b.ControlOutput(c.first);
1885
1886   OpRegistrationData op_reg_data;
1887   TF_CHECK_OK(b.Finalize(&op_reg_data));
1888   fdef.mutable_signature()->Swap(&op_reg_data.op_def);
1889
1890   // Function body
1891   for (const auto& n : node_def) {
```

```
1892      *(fdef.add_node_def()) = n.ToNodeDef();
1893    }
1894
1895    // Returns
1896    for (const auto& r : ret_def) {
1897      fdef.mutable_ret()->insert({r.first, r.second});
1898    }
1899
1900    // Control returns
1901    for (const auto& cr : control_ret_def) {
1902      fdef.mutable_control_ret()->insert({cr.first, cr.second});
1903    }
1904
1905    auto* op_def_registry = OpRegistry::Global();
1906    // Check if any op is stateful.
1907    for (const auto& n : node_def) {
1908      const OpDef* op_def = nullptr;
1909      auto status = op_def_registry->LookUpOpDef(n.op, &op_def);
1910      // Lookup can fail if e.g. we are calling a function that was not yet
1911      // defined.  If it happens, conservatively assume the op is stateful.
1912      if (!status.ok() || op_def->is_stateful()) {
1913        fdef.mutable_signature()->set_is_stateful(true);
1914      }
1915    }
1916
1917    return fdef;
1918  }
1919
1920  /* static */
1921  FunctionDef FunctionDefHelper::Create(
1922      const string& function_name, gtl::ArraySlice<string> in_def,
1923      gtl::ArraySlice<string> out_def, gtl::ArraySlice<string> attr_def,
1924      gtl::ArraySlice<Node> node_def,
1925      gtl::ArraySlice<std::pair<string, string>> ret_def) {
1926    return Create(function_name, in_def, out_def, attr_def, node_def, ret_def,
1927                  /*control_ret_def=*/{});
1928  }
1929
1930  /* static */
1931  FunctionDef FunctionDefHelper::Define(const string& name,
1932                                        gtl::ArraySlice<string> arg_def,
1933                                        gtl::ArraySlice<string> ret_def,
1934                                        gtl::ArraySlice<string> attr_def,
1935                                        gtl::ArraySlice<Node> node_def) {
1936    FunctionDef fdef;
1937    OpDefBuilder b(name);
1938    for (const auto& a : arg_def) b.Input(a);
1939    for (const auto& r : ret_def) b.Output(r);
1940    for (const auto& a : attr_def) b.Attr(a);
```

```
1941
1942    OpRegistrationData op_reg_data;
1943    TF_CHECK_OK(b.Finalize(&op_reg_data));
1944    fdef.mutable_signature()->Swap(&op_reg_data.op_def);
1945
1946    // Mapping from legacy output names to NodeDef outputs.
1947    std::unordered_map<string, string> ret_index;
1948    for (const auto& a : fdef.signature().input_arg()) {
1949      ret_index[a.name()] = a.name();
1950    }
1951
1952    // For looking up OpDefs
1953    auto* op_def_registry = OpRegistry::Global();
1954
1955    // Function body
1956    for (const auto& src : node_def) {
1957      NodeDef* n = fdef.add_node_def();
1958      n->set_op(src.op);
1959      n->set_name(src.GetName());
1960      for (const auto& a : src.attr) {
1961        n->mutable_attr()->insert({a.first, a.second.proto});
1962      }
1963      for (const string& a : src.arg) {
1964        const auto iter = ret_index.find(a);
1965        CHECK(iter != ret_index.end())
1966            << "Node input '" << a << "' in '" << n->name() << "' of " << name;
1967        n->add_input(iter->second);
1968      }
1969      for (const string& d : src.dep) {
1970        n->add_input(strings::StrCat("^", d));
1971      }
1972
1973      // Add the outputs of this node to ret_index.
1974      const OpDef* op_def = nullptr;
1975      TF_CHECK_OK(op_def_registry->LookUpOpDef(n->op(), &op_def)) << n->op();
1976      CHECK(op_def != nullptr) << n->op();
1977      NameRangeMap output_names;
1978      TF_CHECK_OK(NameRangesForNode(*n, *op_def, nullptr, &output_names));
1979      for (const auto& o : output_names) {
1980        CHECK_LE(o.second.second, src.ret.size())
1981            << "Missing ret for output '" << o.first << "' in '" << n->name()
1982            << "' of " << name;
1983        for (int i = o.second.first; i < o.second.second; ++i) {
1984          ret_index[src.ret[i]] =
1985              strings::StrCat(n->name(), ":", o.first, ":", i - o.second.first);
1986        }
1987      }
1988      if (op_def->is_stateful()) fdef.mutable_signature()->set_is_stateful(true);
1989    }
```

```
1990
1991    // Returns
1992    for (const auto& r : fdef.signature().output_arg()) {
1993      const auto iter = ret_index.find(r.name());
1994      CHECK(iter != ret_index.end()) << "Return '" << r.name() << "' in " << name;
1995      fdef.mutable_ret()->insert({r.name(), iter->second});
1996    }
1997    return fdef;
1998  }
1999
2000  FunctionDef FunctionDefHelper::Define(gtl::ArraySlice<string> arg_def,
2001                                        gtl::ArraySlice<string> ret_def,
2002                                        gtl::ArraySlice<string> attr_def,
2003                                        gtl::ArraySlice<Node> node_def) {
2004    return Define("_", arg_def, ret_def, attr_def, node_def);
2005  }
2006
2007  namespace gradient {
2008
2009  typedef std::unordered_map<string, Creator> OpGradFactory;
2010
2011  OpGradFactory* GetOpGradFactory() {
2012    static OpGradFactory* factory = new OpGradFactory;
2013    return factory;
2014  }
2015
2016  bool RegisterOp(const string& op, Creator func) {
2017    CHECK(GetOpGradFactory()->insert({op, func}).second)
2018        << "Duplicated gradient for " << op;
2019    return true;
2020  }
2021
2022  Status GetOpGradientCreator(const string& op, Creator* creator) {
2023    auto fac = GetOpGradFactory();
2024    auto iter = fac->find(op);
2025    if (iter == fac->end()) {
2026      return errors::NotFound("No gradient defined for op: ", op);
2027    }
2028    *creator = iter->second;
2029    return Status::OK();
2030  }
2031
2032  }  // end namespace gradient
2033
2034  }  // namespace tensorflow
```