rdzhabarov [NFC] Remove unused AnnotateOutputShapes method. ... History

24 contributors +12

2851 lines (2605 sloc) | 107 KB

```
/* Copyright 2017 The TensorFlow Authors. All Rights Reserved.

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
==============================================================================*/

#include "tensorflow/core/grappler/costs/graph_properties.h"

#include "absl/types/optional.h"
#include "tensorflow/core/common_runtime/function.h"
#include "tensorflow/core/common_runtime/graph_constructor.h"
#include "tensorflow/core/framework/common_shape_fns.h"
#include "tensorflow/core/framework/function.pb.h"
#include "tensorflow/core/framework/node_def_util.h"
#include "tensorflow/core/framework/tensor.pb.h"
#include "tensorflow/core/framework/tensor_shape.pb.h"
#include "tensorflow/core/framework/types.h"
#include "tensorflow/core/framework/types.pb.h"
#include "tensorflow/core/framework/versions.pb.h"
#include "tensorflow/core/graph/tensor_id.h"
```

```
#include "tensorflow/core/grappler/costs/utils.h"
#include "tensorflow/core/grappler/mutable_graph_view.h"
#include "tensorflow/core/grappler/op_types.h"
#include "tensorflow/core/grappler/optimizers/evaluation_utils.h"
#include "tensorflow/core/grappler/utils.h"
#include "tensorflow/core/grappler/utils/functions.h"
#include "tensorflow/core/grappler/utils/topological_sort.h"
#include "tensorflow/core/lib/gtl/cleanup.h"
#include "tensorflow/core/lib/gtl/flatset.h"
#include "tensorflow/core/lib/strings/str_util.h"

namespace tensorflow {
namespace grappler {

namespace {

using shape_inference::DimensionHandle;
using shape_inference::InferenceContext;
using shape_inference::ShapeAndType;
using shape_inference::ShapeHandle;
using TensorVector = gtl::InlinedVector<TensorValue, 4>;

// A large value for UnknownDim from Const used as a dim value in shape.
// Some ops treat "-1" specially, different from UnknownDim:
// e.g., shape input to Reshape op.
const int64_t kUnknownDimFromConst = INT64_MAX;

// Skip const value instantiation if the number of elements in a const tensor
// is greater than this threshold.
const int kThresholdToSkipConstTensorInstantiation = 128;

template <typename Handle>
struct HashHandle {
  std::size_t operator()(const Handle& h) const { return h.Handle(); }
};
template <typename Handle>
struct CompareHandle {
  bool operator()(const Handle& h1, const Handle& h2) const {
    return h1.SameHandle(h2);
  }
};

template <typename Handle>
struct HandleToObject {};
template <>
struct HandleToObject<ShapeHandle> {
  typedef ShapeHandle Object;

  static ShapeHandle Unknown() { return ShapeHandle(); }
```

```
};

template <>
struct HandleToObject<DimensionHandle> {
  typedef int64_t Object;

  static int64_t Unknown() { return -1; }
};

template <typename Handle>
struct Processor {};

template <>
struct Processor<ShapeHandle> {
  // Extract the shape or dim denoted by the handle.
  void ExtractValue(ShapeHandle h, ShapeHandle* result) { *result = h; }
  // Merge the shapes or dims.
  Status Merge(ShapeHandle h1, ShapeHandle h2, ShapeHandle* result) {
    if (InferenceContext::RankKnown(*result)) {
      // The result was initialized in a previous merge to a shape of known
      // rank, make sure we preserve that information.
      return Status::OK();
    }
    if (InferenceContext::RankKnown(h1)) {
      *result = h1;
    } else {
      *result = h2;
    }
    return Status::OK();
  }
};

template <>
struct Processor<DimensionHandle> {
  // Assign a negative id to unknown dimensions, starting at -2 (the -1 id
  // reserved by TensorFlow).
  void ExtractValue(DimensionHandle d, int64_t* result) {
    if (!InferenceContext::ValueKnown(d)) {
      *result = -counter;
      counter++;
    } else {
      int64_t val = InferenceContext::Value(d);
      if (val >= 0) {
        *result = val;
      } else {
        // A shape inference function generated an invalid dimension handle.
        // Use a symbolic dimension to encode this.
        *result = -counter;
        counter++;
```

```
      }
    }
  }

  // Merge the dimensions d1 and d2. Return the known shape if there is one,
  // otherwise look for a symbolic shape. If there is no symbolic shape and no
  // known shape, the shape if fully unknown so return -1.
  Status Merge(DimensionHandle d1, DimensionHandle d2, int64_t* result) {
    const int64_t dim1 = InferenceContext::Value(d1);
    const int64_t dim2 = InferenceContext::Value(d2);

    if (dim1 >= 0 && dim2 >= 0) {
      CHECK_EQ(dim1, dim2);
      return RefineDim(dim1, result);
    } else if (dim1 >= 0 && dim2 < 0) {
      return RefineDim(dim1, result);
    } else if (dim1 < 0 && dim2 >= 0) {
      return RefineDim(dim2, result);
    } else if (dim1 < -1) {
      return RefineDim(dim1, result);
    } else if (dim2 < -1) {
      return RefineDim(dim2, result);
    } else {
      CHECK_EQ(dim1, dim2);
      CHECK_EQ(-1, dim1);
      return RefineDim(-1, result);
    }
    return Status::OK();
  }

 private:
  Status RefineDim(int64_t dim, int64_t* result) {
    if (*result >= 0) {
      if (!(*result == dim || dim < 0)) {
        return errors::InvalidArgument("Inconsistent dimensions detected");
      }
    } else if (dim >= 0) {
      *result = dim;
    } else if (dim < *result) {
      *result = dim;
    }
    return Status::OK();
  }

  int64_t counter = 2;
};

// Traditional Disjoint-Set datastructure with path compression.
// (https://en.wikipedia.org/wiki/Disjoint-set_data_structure)
```

```
template <typename Handle>
class DisjointSet {
 public:
  DisjointSet() {}
  ~DisjointSet() {
    for (auto rep : nodes_) {
      delete rep.second;
    }
  }

  Status Merge(Handle x, Handle y);
  const typename HandleToObject<Handle>::Object GetMergedValue(Handle value);

 private:
  // All the handles that belong to the same set are part of the same tree, and
  // utimately represented by the root of that tree.
  struct Rep {
    // Parent in the tree used to encode the set.
    Rep* parent;
    // Rank in the tree, used to figure out how to compress the path to the root
    // of the tree.
    int rank;
    // The handle.
    typename HandleToObject<Handle>::Object value;
  };

  // Create a new set for the value if none exists, or return its representative
  // node otherwise.
  Rep* Find(Handle value);

 private:
  Processor<Handle> processor_;
  absl::flat_hash_map<Handle, Rep*, HashHandle<Handle>, CompareHandle<Handle>>
      nodes_;
};

template <typename Handle>
const typename HandleToObject<Handle>::Object
DisjointSet<Handle>::GetMergedValue(Handle value) {
  Rep* rep = Find(value);
  if (!rep) {
    // We don't know anything about this handle.
    return HandleToObject<Handle>::Unknown();
  }
  return rep->value;
}

template <typename Handle>
Status DisjointSet<Handle>::Merge(Handle x, Handle y) {
```

```
  Rep* x_root = Find(x);
  Rep* y_root = Find(y);

  // x and y are already in the same set
  if (x_root == y_root) {
    return Status::OK();
  }
  // x and y are not in same set, so we merge them
  // Use the occasion to strengthen what we know about the handle by merging the
  // information about the 2 subsets.
  if (x_root->rank < y_root->rank) {
    TF_RETURN_IF_ERROR(processor_.Merge(y, x, &y_root->value));
    x_root->parent = y_root;
  } else if (x_root->rank > y_root->rank) {
    TF_RETURN_IF_ERROR(processor_.Merge(x, y, &x_root->value));
    y_root->parent = x_root;
  } else {
    TF_RETURN_IF_ERROR(processor_.Merge(x, y, &x_root->value));
    // Arbitrarily make one root the new parent
    y_root->parent = x_root;
    x_root->rank = x_root->rank + 1;
  }
  return Status::OK();
}

template <typename Handle>
typename DisjointSet<Handle>::Rep* DisjointSet<Handle>::Find(Handle value) {
  auto it = nodes_.find(value);
  if (it == nodes_.end()) {
    // This is the first time we process this handle, create an entry for it.
    Rep* node = new Rep;
    node->parent = node;
    node->rank = 0;
    processor_.ExtractValue(value, &node->value);
    nodes_[value] = node;
    return node;
  }
  // Return the representative for the set, which is the root of the tree. Apply
  // path compression to speedup future queries.
  Rep* node = it->second;
  Rep* root = node->parent;
  while (root != root->parent) {
    root = root->parent;
  }
  while (node->parent != root) {
    Rep* next = node->parent;
    node->parent = root;
    node = next;
  }
```

```
  return root;
}

// TODO(dyoon): Move many helper functions in this file (including those within
// SymbolicShapeRefiner class) to shared utils.
bool IsEnqueue(const NodeDef& n) {
  return (n.op().find("Enqueue") != string::npos &&
          n.op().find("EnqueueMany") == string::npos);
}

bool IsDequeue(const NodeDef& n) {
  return (n.op().find("Dequeue") != string::npos &&
          n.op().find("DequeueMany") == string::npos);
}

bool HasAnyUnknownDimensions(const TensorShapeProto& proto) {
  if (proto.unknown_rank()) {
    return true;
  }
  for (const auto& dim : proto.dim()) {
    if (dim.size() < 0) {
      return true;
    }
  }
  return false;
}

// This really should be done in an external debugging tool
void VerboseLogUnknownDimensionSources(
    const GraphDef& graph,
    const absl::flat_hash_map<string, std::vector<OpInfo::TensorProperties>>&
        input_properties_map,
    const absl::flat_hash_map<string, std::vector<OpInfo::TensorProperties>>&
        output_properties_map) {
  if (!VLOG_IS_ON(2)) {
    return;
  }

  VLOG(2) << "Nodes with known inputs, but with unknown output dimensions:";

  // Find all nodes in the graph for which we
  // do not have any unknown dimensions in their inputs, but
  // we have some unknown dimensions in their outputs.
  std::map<string, int> op_to_count;
  for (const NodeDef& node : graph.node()) {
    const auto& input_properties = input_properties_map.at(node.name());
    const auto& output_properties = output_properties_map.at(node.name());

    bool has_unknown_inputs = false;
```

```
    for (const auto& input_prop : input_properties) {
      if (HasAnyUnknownDimensions(input_prop.shape())) {
        has_unknown_inputs = true;
        break;
      }
    }

    if (has_unknown_inputs) {
      continue;
    }

    for (const auto& output_prop : output_properties) {
      if (HasAnyUnknownDimensions(output_prop.shape())) {
        string inputs = "input_shapes=[";
        for (const auto& input_prop : input_properties) {
          inputs += PartialTensorShape::DebugString(input_prop.shape());
        }
        inputs += "]";

        string outputs = "output_shapes=[";
        for (const auto& output_prop : output_properties) {
          outputs += PartialTensorShape::DebugString(output_prop.shape());
        }
        outputs += "]";

        VLOG(2) << "Node: " << node.name() << ", Op: " << node.op() << ", "
                << inputs << ", " << outputs;

        op_to_count[node.op()]++;

        // don't log again for this node
        break;
      }
    }
  }
  VLOG(2) << "Op types with known inputs, but with unknown output dimensions "
          << "(format: <op_type> (<count>)):";
  for (const auto& p : op_to_count) {
    VLOG(2) << p.first << " (" << p.second << ")";
  }
}

// Helper function to convert kUnknownDimFromConst into UnknownDim.
std::vector<ShapeHandle> ReplaceUnknownDimFromConstWithUnknownDim(
    InferenceContext* ic, const std::vector<ShapeHandle>& shapes) {
  std::vector<ShapeHandle> converted_shapes(shapes.size());
  for (int i = 0, shapes_size = shapes.size(); i < shapes_size; i++) {
    const auto& shape = shapes[i];
    if (!ic->RankKnown(shape)) {
```

```
      converted_shapes[i] = shape;
      continue;
    }
    bool just_copy = true;
    std::vector<DimensionHandle> dims;
    for (int32_t i = 0; i < ic->Rank(shape); ++i) {
      DimensionHandle dim = ic->Dim(shape, i);
      if (ic->ValueKnown(dim) && ic->Value(dim) == kUnknownDimFromConst) {
        just_copy = false;
        dims.push_back(ic->UnknownDim());
      } else {
        dims.push_back(dim);
      }
    }
    if (just_copy) {
      converted_shapes[i] = shape;
      continue;
    }
    converted_shapes[i] = ic->MakeShape(dims);
  }
  return converted_shapes;
}

// Returned tensor's shape is like `shape`, and its values and dtype are from
// `tensor_as_shape` and `dtype`.
TensorProto MakeTensorProtoFromShape(InferenceContext* ic,
                                     const ShapeHandle& shape,
                                     const ShapeHandle& tensor_as_shape,
                                     const DataType& dtype) {
  TensorProto tensor_proto;
  tensor_proto.set_dtype(dtype);
  auto* shape_proto = tensor_proto.mutable_tensor_shape();
  if (ic->Rank(shape) == 1) {
    shape_proto->add_dim()->set_size(ic->Rank(tensor_as_shape));
  }
  // For a scalar tensor, tensor_shape field will be left empty; no dim.
  for (int i = 0; i < ic->Rank(tensor_as_shape); i++) {
    int64_t value = ic->Value(ic->Dim(tensor_as_shape, i));
    if (dtype == DT_INT32) {
      tensor_proto.add_int_val(value);
    } else {
      tensor_proto.add_int64_val(value);
    }
  }
  return tensor_proto;
}

// Returns a Const NodeDef with tensor `tensor_proto` and dtype = `dtype`.
NodeDef MakeConstNodeDefFromTensorProto(InferenceContext* ic,
```

```
                                                const TensorProto& tensor_proto,
                                                const DataType& dtype) {
  NodeDef const_node;
  const_node.set_name("const_from_shape");
  const_node.set_op("Const");
  auto* attr = const_node.mutable_attr();
  (*attr)["dtype"].set_type(dtype);
  auto* tensor = (*attr)["value"].mutable_tensor();
  *tensor = tensor_proto;
  return const_node;
}

// Returns a Const NodeDef with shape = `shape`, values = `tensor_as_shape`,
// and dtype = `dtype`.
NodeDef MakeConstNodeDefFromShape(InferenceContext* ic,
                                  const ShapeHandle& shape,
                                  const ShapeHandle& tensor_as_shape,
                                  const DataType& dtype) {
  return MakeConstNodeDefFromTensorProto(
      ic, MakeTensorProtoFromShape(ic, shape, tensor_as_shape, dtype), dtype);
}

bool IsNumericType(const DataType dtype) {
  static const gtl::FlatSet<DataType>* const kRealNumberTypes =
      CHECK_NOTNULL((new gtl::FlatSet<DataType>{
          // Floating point.
          DT_BFLOAT16,
          DT_HALF,
          DT_FLOAT,
          DT_DOUBLE,
          // Int / UInt.
          DT_INT8,
          DT_INT16,
          DT_INT32,
          DT_INT64,
          DT_UINT8,
          DT_UINT16,
          DT_UINT32,
          DT_UINT64,
          // Quantized Int.
          DT_QINT8,
          DT_QUINT8,
          DT_QINT16,
          DT_QUINT16,
          DT_QINT32,
          // Bool.
          DT_BOOL,
      }));
  return kRealNumberTypes->find(dtype) != kRealNumberTypes->end();
```

```
}

// Returns the number of elements in the input (const) tensor.
// -1 if the tensor has no shape or unknown rank.
uint64 NumElementsFromTensorProto(const TensorProto& tensor_proto) {
  if (!tensor_proto.has_tensor_shape()) {
    return -1;
  }
  const auto& tensor_shape_proto = tensor_proto.tensor_shape();
  if (tensor_shape_proto.unknown_rank()) {
    return -1;
  }
  int64_t num_elements = 1;
  for (const auto& dim : tensor_shape_proto.dim()) {
    // Note that in some cases, dim.size() can be zero (e.g., empty vector).
    num_elements *= dim.size();
  }
  return num_elements;
}

}  // namespace

// Note that tensor_as_shape input should not include kUnknownDimFromConst.
// This function check kUnknownDimFromConst, but will log WARNING.
// If checking input_tensors_as_shape_to_propgate or output_tensors_as_shape,
// which may include kUnknownDimFromConst, run
// convert it using ReplaceUnknownDimFromConstWithUnknownDim() before.
bool IsShapeFullyDefinedIntegerVectorOrScalar(
    InferenceContext* ic, const ShapeHandle& shape,
    const ShapeHandle& tensor_as_shape, const DataType& dtype) {
  if (!ic->FullyDefined(shape) || ic->Rank(shape) > 1 ||
      !ic->FullyDefined(tensor_as_shape) ||
      (dtype != DT_INT32 && dtype != DT_INT64)) {
    return false;
  }
  // Also check whether any dim in tensor_as_shape is kUnknownDimFromConst.
  for (int32_t i = 0; i < ic->Rank(tensor_as_shape); ++i) {
    DimensionHandle dim = ic->Dim(tensor_as_shape, i);
    if (ic->Value(dim) == kUnknownDimFromConst) {
      LOG(WARNING) << "IsShapeFullyDefinedIntegerVectorOrScalar(): "
                   << "tensor_as_shape input includes kUnknownDimFromConst -- "
                   << ic->DebugString(tensor_as_shape);
      return false;
    }
  }
  return true;
}

// Queue of nodes to process. Nodes can be enqueued in any order, but will be
```

```cpp
// dequeued in (roughly) topological order. Propagating shapes following a
// topological ordering isn't required for correctness but helps speed things up
// since it avoids processing the same node multiple times as its inputs
// information is refined.
class TopoQueue {
 public:
  explicit TopoQueue(const std::vector<const NodeDef*>& topo_order)
      : topo_order_(TopoOrder(topo_order)) {}

  void push(const NodeDef* n) { queue_.emplace(n, topo_order_.at(n)); }

  const NodeDef* pop() {
    CHECK(!empty());
    auto it = queue_.begin();
    const NodeDef* n = it->first;
    queue_.erase(it);
    return n;
  }

  bool empty() const { return queue_.empty(); }
  std::size_t size() const { return queue_.size(); }

 private:
  using NodeAndId = std::pair<const NodeDef*, int>;
  // Graph nodes are created in (roughly) topological order. Therefore we can
  // use their id to ensure they're sorted topologically.
  struct OrderByIdAscending {
    bool operator()(const NodeAndId& lhs, const NodeAndId& rhs) const {
      return lhs.second < rhs.second;
    }
  };

  const absl::flat_hash_map<const NodeDef*, int> TopoOrder(
      const std::vector<const NodeDef*>& topo_order) const {
    absl::flat_hash_map<const NodeDef*, int> map;
    map.reserve(topo_order.size());
    for (int i = 0, topo_order_size = topo_order.size(); i < topo_order_size;
         ++i) {
      map.emplace(topo_order[i], i);
    }
    return map;
  }

  const absl::flat_hash_map<const NodeDef*, int> topo_order_;
  std::set<NodeAndId, OrderByIdAscending> queue_;
};


bool IsAllowListedOpTypeForEvaluateNode(const string& op_type) {
```

```
  static const gtl::FlatSet<string>* const kOpTpeAllowlist =
      CHECK_NOTNULL((new gtl::FlatSet<string>{
          // Unary arithmetic ops
          "Floor",
          "Round",
          "Sqrt",
          "Square",
          "Sign",
          // Binary arithmetic ops
          "Add",
          "AddV2",
          "Div",
          "FloorDiv",
          "FloorMod",
          "Greater",
          "GreaterEqual",
          "Less",
          "LessEqual",
          "LogicalAnd",
          "LogicalNot",
          "LogicalOr",
          "Maximum",
          "Minimum",
          "Mod",
          "Mul",
          "NotEqual",
          "QuantizedAdd",
          "QuantizedMul",
          "SquareDifference",
          "Sub",
          "TruncateDiv",
          "TruncateMod",
          "RealDiv",
          // N-ary arithmetic ops
          "AddN",
          // Others
          "StridedSlice",
          "OnesLike",
          "ZerosLike",
          "Concat",
          "ConcatV2",
          "Split",
          "Range",
          "Fill",
          "Cast",
          "Prod",
          "Unpack",
          "GatherV2",
          "Pack",
```

```
        // Used in batch_gather_nd: tensorflow/python/ops/array_ops.py
        "ExpandDims",
    }));
  return kOpTpeAllowlist->find(op_type) != kOpTpeAllowlist->end();
}

// Negative shape size of '-1' represents unknown, while negative shape sizes
// less than -1 represent unknown symbolic shapes (e.g. the shape of [-5, 5, -1,
// -5] really means [x, 5, ?, x]). Before we can output the tensors as shapes,
// we need to normalize them: mark all values <-1 as "unknown" (-1).
static void NormalizeShapeForOutput(TensorShapeProto* shape) {
  for (int i = 0; i < shape->dim_size(); i++) {
    if (shape->dim(i).size() < -1) {
      VLOG(2) << "Normalizing dimension: " << i << " from "
              << shape->dim(i).size() << " to -1";
      shape->mutable_dim(i)->set_size(-1);
    }
  }
}

// Processes symbolic shapes.
// Each symbolic shape or dimension is represented by a handle. Unlike the TF
// shape refiner which creates new handles every time it processes an unknown
// shape/dimension, the symbolic shape refiner assigns a specific handle to each
// unknown shape/dimension of a given node.
class SymbolicShapeRefiner {
 public:
  explicit SymbolicShapeRefiner(
      const GraphView& graph,
      const absl::flat_hash_map<string, absl::flat_hash_set<int>>& fed_ports,
      const bool aggressive_shape_inference)
      : graph_(graph),
        function_library_(OpRegistry::Global(), graph.graph()->library()),
        fed_ports_(fed_ports),
        aggressive_shape_inference_(aggressive_shape_inference) {
    graph_def_version_ = graph.graph()->versions().producer();
    node_to_context_.reserve(graph.graph()->node_size());
  }

  const GraphView& graph() const { return graph_; }

  struct NodeContext {
    const OpRegistrationData* op_data;
    DataTypeVector input_types;
    DataTypeVector output_types;
    std::unique_ptr<InferenceContext> inference_context;
    // Additional info for propagating tensor values and tensor shapes.
    std::vector<const TensorProto*> input_tensor_protos;
    std::vector<const TensorProto*> output_tensor_protos;
```

```
  // This is the same to inference_context->input_tensors_as_shapes, except
  // that some UnknownDims (-1) can be kUnknownDimFromConst.
  std::vector<ShapeHandle> input_tensors_as_shapes_to_propagate;
  std::vector<ShapeHandle> output_tensors_as_shapes;

  // Output shapes incompatible between annotation and shape inference.
  bool shape_incompatible = false;

  // Similar to DebugString() in InferenceContext, but prints out
  // kUnknownDimFromConst properly.
  std::string StringifyShapeHandle(ShapeHandle s) {
    auto* ic = inference_context.get();
    if (ic->RankKnown(s)) {
      std::vector<std::string> vals;
      for (int i = 0; i < ic->Rank(s); i++) {
        DimensionHandle d = ic->Dim(s, i);
        if (ic->ValueKnown(d) && ic->Value(d) == kUnknownDimFromConst) {
          vals.push_back("?(Const)");
        } else {
          vals.push_back(ic->DebugString(d));
        }
      }
      return strings::StrCat("[", absl::StrJoin(vals, ","), "]");
    } else {
      return "?";
    }
  }

  std::string DebugString(const NodeDef& node) {
    std::string output;
    auto* ic = inference_context.get();
    absl::StrAppend(
        &output, node.name(), " [", node.op(), "] has ", ic->num_inputs(),
        (ic->num_inputs() > 1 ? " inputs and " : " input and "),
        ic->num_outputs(), (ic->num_outputs() > 1 ? " outputs" : " output"));
    if (op_data->is_function_op) {
      absl::StrAppend(&output, " (function op)");
    }
    absl::StrAppend(&output, ": \n");

    for (int i = 0; i < ic->num_inputs(); i++) {
      absl::StrAppend(&output, " input [", i, "] ", node.input(i),
                      " -- type: ", DataTypeString(input_types.at(i)),
                      ", shape: ", ic->DebugString(ic->input(i)),
                      ", tensor: ");
      Tensor t1;
      int input_tensor_protos_size = input_tensor_protos.size();
      if (input_tensor_protos_size > i &&
          input_tensor_protos.at(i) != nullptr &&
```

```
          t1.FromProto(*input_tensor_protos.at(i))) {
        absl::StrAppend(&output, t1.DebugString(), ", tensor_as_shape: ");
      } else {
        absl::StrAppend(&output, " null, tensor_as_shape: ");
      }
      int input_tensors_as_shapes_to_propagate_size =
          input_tensors_as_shapes_to_propagate.size();
      if (input_tensors_as_shapes_to_propagate_size > i) {
        absl::StrAppend(
            &output,
            StringifyShapeHandle(input_tensors_as_shapes_to_propagate.at(i)),
            "\n");
      } else {
        absl::StrAppend(&output, " null\n");
      }
    }
    for (int i = 0; i < ic->num_outputs(); i++) {
      absl::StrAppend(&output, " output [", i,
                      "] -- type: ", DataTypeString(output_types.at(i)),
                      ", shape: ", ic->DebugString(ic->output(i)),
                      ", tensor: ");
      Tensor t2;
      int output_tensor_protos_size = output_tensor_protos.size();
      if (output_tensor_protos_size > i &&
          output_tensor_protos.at(i) != nullptr &&
          t2.FromProto(*output_tensor_protos.at(i))) {
        absl::StrAppend(&output, t2.DebugString(), ", tensor_as_shape: ");
      } else {
        absl::StrAppend(&output, " null, tensor_as_shape: ");
      }
      int output_tensors_as_shapes_size = output_tensors_as_shapes.size();
      if (output_tensors_as_shapes_size > i) {
        absl::StrAppend(&output,
                        StringifyShapeHandle(output_tensors_as_shapes.at(i)),
                        "\n");
      } else {
        absl::StrAppend(&output, " null\n");
      }
    }
    return output;
  }
};

NodeContext* GetNodeContext(const NodeDef* node) {
  auto it = node_to_context_.find(node);
  if (it == node_to_context_.end()) {
    return nullptr;
  }
  return &it->second;
```

```
  }

  InferenceContext* GetContext(const NodeDef* node) {
    auto it = node_to_context_.find(node);
    if (it == node_to_context_.end()) {
      return nullptr;
    }
    return it->second.inference_context.get();
  }

  // Forward the shapes from the function input nodes, PartitionedCalls or
  // StatefulPartitionedCall to
  // the argument nodes (which are Placeholder nodes), then
  // perform shape inference on the function body.
  //
  // Propagate shape information of final function body node
  // to function node `function_node`.
  //
  // In the event of an error, UpdateNode will simply set `function_node`'s
  // output shape to be Unknown.
  Status UpdateFunction(const NodeDef* function_node) {
    NameAttrList function;
    TF_RETURN_IF_ERROR(NameAndAttrsFromFunctionCall(*function_node, &function));
    auto it = fun_to_grappler_function_item_.find(function.name());
    if (it == fun_to_grappler_function_item_.end()) {
      return errors::InvalidArgument(
          function.name(),
          " was not previously added to SymbolicShapeRefiner.");
    }

    const absl::optional<GrapplerFunctionItem>& maybe_grappler_function_item =
        it->second;
    if (!maybe_grappler_function_item.has_value()) {
      VLOG(3) << "Skip failed to instantiate function call: function_name="
              << function.name();

      auto* ctx = GetNodeContext(function_node);
      auto* ic = ctx->inference_context.get();
      for (int i = 0; i < ic->num_outputs(); ++i) {
        TF_RETURN_IF_ERROR(SetUnknownShape(function_node, i));
      }

      return Status::OK();
    }

    // Copy (not reference) so that changes we make here (e.g., replacing
    // _Arg with Const and _Retval with Identity) don't affect one in
    // fun_to_grappler_function_item_.
    GrapplerFunctionItem grappler_function_item = *maybe_grappler_function_item;
```

```
  MutableGraphView gv(&grappler_function_item.graph);

  // Forward shapes from function input nodes to argument nodes.
  for (int i = 0, end = grappler_function_item.inputs().size(); i < end;
       ++i) {
    auto& fun_input = grappler_function_item.input(i);
    NodeDef* fun_node = gv.GetNode(fun_input.node_name);
    const TensorId input_tensor = ParseTensorName(function_node->input(i));

    if (IsControlInput(input_tensor)) {
      return errors::FailedPrecondition(
          "Function inputs should not contain control nodes.");
    }

    const NodeDef* input_node = graph_.GetNode(input_tensor.node());
    if (input_node == nullptr) {
      return errors::FailedPrecondition(input_tensor.node(),
                                        " was not found in the graph.");
    }

    InferenceContext* input_ic = GetContext(input_node);
    if (input_ic == nullptr) {
      return errors::FailedPrecondition(
          "Inference context has not been created for ", input_tensor.node());
    }

    int output_port_num = input_tensor.index();
    AttrValue attr_output_shape;
    TensorShapeProto proto;
    const auto handle = input_ic->output(output_port_num);
    input_ic->ShapeHandleToProto(handle, &proto);
    // There may be dim.size < -1 in SymbolicShapeRefiner. Change those to -1.
    NormalizeShapeForOutput(&proto);
    // _Arg op's output shape uses _output_shapes attr.
    AttrValue output_attr;
    output_attr.mutable_list()->add_shape()->Swap(&proto);
    (*fun_node->mutable_attr())["_output_shapes"] = output_attr;

    // If dtype is DT_RESOURCE, ops that read _Arg op use _handle_dtypes and
    // _handle_shapes attr for its shapes and dtypes.
    if (fun_input.data_type == DT_RESOURCE) {
      auto* shapes_and_types =
          input_ic->output_handle_shapes_and_types(output_port_num);
      if (shapes_and_types != nullptr && !shapes_and_types->empty()) {
        AttrValue dtype_attr;
        AttrValue shape_attr;
        for (const auto& shape_and_type : *shapes_and_types) {
          const auto& dtype = shape_and_type.dtype;
          const auto& shape_handle = shape_and_type.shape;
```

```
          dtype_attr.mutable_list()->add_type(dtype);
          input_ic->ShapeHandleToProto(
              shape_handle, shape_attr.mutable_list()->add_shape());
        }
        (*fun_node->mutable_attr())["_handle_dtypes"] = dtype_attr;
        (*fun_node->mutable_attr())["_handle_shapes"] = shape_attr;
      } else {
        // Note that we do not return error here, even if the input node does
        // not have shapes_and_types. Within the function, we cannot infer the
        // output shape of the DT_RESOURCE input; hence, potentially unknown
        // shapes/dims in the function output shapes.
        VLOG(2)
            << "A function node (" << function_node->name()
            << ") has input with DT_RESOURCE, but the input node does not "
            << "have shapes_and_types information: \n"
            << "function_node: " << function_node->ShortDebugString() << "\n"
            << "function input: " << i
            << ", input node's output: " << output_port_num << "\n"
            << "input node: " << input_node->ShortDebugString();
      }
    }
  }

  // ReplaceInputWithConst() may break GraphView's internal node mapping
  // structure; hence, we separately build node name to NodeDef* map, for the
  // output nodes (before GraphView becomes invalid). Note that we use string,
  // not string_view.
  absl::flat_hash_map<std::string, NodeDef*> output_nodes;
  for (const auto& output_arg : grappler_function_item.outputs()) {
    output_nodes[output_arg.node_name] = gv.GetNode(output_arg.node_name);
  }

  // Replace input nodes with Consts, if values are known. Note that
  // we don't check exceptions here as it's done in the above loop.
  auto* ctx = GetNodeContext(function_node);
  auto* ic = ctx->inference_context.get();
  for (int i = grappler_function_item.inputs().size() - 1; i >= 0; --i) {
    const string& input = function_node->input(i);
    const string node_name = NodeName(input);
    const NodeDef* input_node = graph_.GetNode(node_name);
    if (IsConstant(*input_node)) {
      TF_CHECK_OK(
          ReplaceInputWithConst(*input_node, i, &grappler_function_item));
    } else if (static_cast<int>(ctx->input_tensor_protos.size()) > i &&
               ctx->input_tensor_protos[i] != nullptr) {
      NodeDef const_input_node = MakeConstNodeDefFromTensorProto(
          ic, *ctx->input_tensor_protos[i], ctx->input_types[i]);
      TF_CHECK_OK(ReplaceInputWithConst(const_input_node, i,
                                        &grappler_function_item));
```

```
      } else if (static_cast<int>(ic->input_tensors_as_shapes().size()) > i &&
                 IsShapeFullyDefinedIntegerVectorOrScalar(
                     ic, ic->input(i), ic->input_tensors_as_shapes()[i],
                     ctx->input_types[i])) {
        // We have fully defined input_tensors_as_shapes for this input; use it
        // as a const input to the function node.
        NodeDef const_input_node = MakeConstNodeDefFromShape(
            ic, ic->input(i), ic->input_tensors_as_shapes()[i],
            ctx->input_types[i]);
        TF_CHECK_OK(ReplaceInputWithConst(const_input_node, i,
                                          &grappler_function_item));
      }
    }
    // node_name to NodeDef* map in GraphView gv can be broken due to
    // ReplaceInputWithConst(). gv should not be used after this.

    // Replace output _Retval nodes with Identity nodes. _Retval is a system op
    // without outputs and registered shape function.
    for (const auto& output_arg : grappler_function_item.outputs()) {
      NodeDef* output_node = output_nodes[output_arg.node_name];
      DCHECK_EQ(output_node->op(), "_Retval");
      output_node->set_op("Identity");
      output_node->mutable_attr()->erase("index");
    }

    // Perform inference on function body.
    GraphProperties gp(grappler_function_item);
    TF_RETURN_IF_ERROR(gp.InferStatically(
        /*assume_valid_feeds=*/true,
        /*aggressive_shape_inference=*/aggressive_shape_inference_,
        /*include_tensor_values=*/true));

    // Add return nodes for output shapes.
    int output = 0;
    ctx->output_tensors_as_shapes.resize(grappler_function_item.output_size());
    ctx->output_tensor_protos.resize(grappler_function_item.output_size(),
                                     nullptr);
    for (auto const& out_arg : grappler_function_item.outputs()) {
      // It is guaranteed that output_tensors does not contain any control
      // inputs, so port_id >= 0.
      TensorId out_tensor = ParseTensorName(out_arg.node_name);

      if (output_nodes.count(out_tensor.node()) <= 0) {
        return errors::FailedPrecondition(
            "Unable to find return function_node ", out_tensor.node(), " for ",
            function_node->name());
      }
      const NodeDef* retnode = output_nodes[out_tensor.node()];

```

```
      auto output_properties = gp.GetOutputProperties(retnode->name());
      int output_properties_size = output_properties.size();
      if (out_tensor.index() >= output_properties_size) {
        return errors::InvalidArgument(
            out_tensor.ToString(), " has invalid position ", out_tensor.index(),
            " (output_properties.size() = ", output_properties.size(), ").");
      }
      auto& outprop = output_properties[out_tensor.index()];
      TensorShapeProto shape = outprop.shape();
      NormalizeShapeForOutput(&shape);
      ShapeHandle out;
      TF_RETURN_IF_ERROR(ic->MakeShapeFromShapeProto(shape, &out));
      ic->set_output(output, out);
      if (outprop.has_value()) {
        // Forward tensor value to output_tensors_as_shape.
        MaybeTensorProtoToShape(ic, outprop.value(),
                                &ctx->output_tensors_as_shapes[output]);
        const_tensors_to_propagate_.push_back(outprop.value());
        ctx->output_tensor_protos[output] = &const_tensors_to_propagate_.back();
      }
      output++;
    }

    return Status::OK();
  }

  // Prepares input shapes/values/handles, then runs shape inference, and
  // finally sets output shapes/values/handles.
  Status UpdateNode(const NodeDef* node, bool* refined) {
    NodeContext* ctx = GetNodeContext(node);
    if (ctx == nullptr) {
      TF_RETURN_IF_ERROR(AddNode(node));
      ctx = CHECK_NOTNULL(GetNodeContext(node));
      *refined = true;
    }

    // Check if the shapes of the nodes in the fan-in of this node have changed,
    // and if they have, update the node input shapes.
    InferenceContext* ic = ctx->inference_context.get();
    ctx->input_tensors_as_shapes_to_propagate.resize(ic->num_inputs());
    ctx->input_tensor_protos.resize(ic->num_inputs(), nullptr);

    for (int dst_input = 0; dst_input < ic->num_inputs(); ++dst_input) {
      const GraphView::InputPort port(node, dst_input);
      const GraphView::OutputPort fanin = graph_.GetRegularFanin(port);
      int src_output = fanin.port_id;
      const NodeDef* src = fanin.node;
      NodeContext* src_ctx = GetNodeContext(src);
      if (src_ctx == nullptr) {
```

```
      return errors::FailedPrecondition(
          "Input ", dst_input, " for '", node->name(),
          "' was not previously added to SymbolicShapeRefiner.");
    }

    InferenceContext* src_ic = src_ctx->inference_context.get();
    if (src_output >= src_ic->num_outputs()) {
      return errors::OutOfRange("src_output = ", src_output,
                                ", but num_outputs is only ",
                                src_ic->num_outputs());
    }

    // Propagate input node's NodeContext info to the current node's
    // NodeContext:
    // output_tensor_protos to input_tensor_protos and input_tensors, and
    // output_tensors_as_shapes to input_tensors_as_shapes.
    if (static_cast<int>(src_ctx->output_tensors_as_shapes.size()) >
        src_output) {
      ctx->input_tensors_as_shapes_to_propagate[dst_input] =
          src_ctx->output_tensors_as_shapes[src_output];
    }

    if (static_cast<int>(src_ctx->output_tensor_protos.size()) > src_output) {
      const auto* tensor_proto = src_ctx->output_tensor_protos[src_output];
      if (tensor_proto != nullptr) {
        ctx->input_tensor_protos[dst_input] = tensor_proto;
      }
    }

    // NOTE: we check only shape is refined; we do not (yet) check whether
    // tensor value is refined.
    if (!*refined &&
        !ic->input(dst_input).SameHandle(src_ic->output(src_output))) {
      *refined = true;
    }
    ic->SetInput(dst_input, src_ic->output(src_output));

    if (!*refined && ic->requested_input_tensor_as_partial_shape(dst_input)) {
      // The input value may have changed. Since we have no way to know if
      // that's indeed the case, err on the safe side.
      *refined = true;
    }

    // Also propagate handle shape and dtype of edges which are carrying
    // resource handles.
    if (ctx->input_types[dst_input] == DT_RESOURCE) {
      auto* outputs = src_ic->output_handle_shapes_and_types(src_output);
      if (!outputs) continue;
      auto* inputs = ic->input_handle_shapes_and_types(dst_input);
```

```
      if (!inputs || !EquivalentShapesAndTypes(*outputs, *inputs))
        *refined = true;
      ic->set_input_handle_shapes_and_types(dst_input, *outputs);
    }
  }

  // Make sure we schedule the fanout of resources (which have no input)
  // whenever the resources are updated.
  *refined |= ic->num_inputs() == 0;

  if (!*refined) {
    // No input shape has changed, we're done.
    return Status::OK();
  }

  // Convert all kUnknownDimFromConst to -1 for shape inference.
  ic->set_input_tensors_as_shapes(ReplaceUnknownDimFromConstWithUnknownDim(
      ic, ctx->input_tensors_as_shapes_to_propagate));
  // Note: UpdateFunction uses input_tensors_as_shapes and
  // input_tensor_protos (not the Tensor object) for input values.
  // so for function nodes, we don't need to convert TensorProtos
  // to Tensors here. If the current op is not a function op, we convert
  // TensorProtos to Tensors before calling InferShapes.

  // Properly handle function nodes.
  if (ctx->op_data && ctx->op_data->is_function_op) {
    // TODO(jmdecker): Detect if the input shapes have changed for this
    // function. Note that when we hit a function call node, refined will be
    // true, as the updates to the call node will have changed, even if it's
    // the same function being called twice with the same input shapes.
    // Example: simple_function.pbtxt
    if (aggressive_shape_inference_) {
      // If output shapes are annotated, use it and skip UpdateFunction();
      // it can be very expensive when a function node has nested function
      // nodes internally. One downside with this approach is that we do not
      // get output values or output shapes as tensor from function node.
      auto s = UpdateOutputShapesUsingAnnotatedInformation(*node, ctx);
      if (s.ok() && AllOutputShapesKnown(ctx)) {
        return Status::OK();
      }
      // If shape annotation was not available, incomplete, or incompatible,
      // fall through to call UpdateFunction().
    }
    auto s = UpdateFunction(node);
    if (s.ok()) {
      return Status::OK();
    } else {
      VLOG(1) << "UpdateFunction failed for " << node->op()
```

```
              << ". Defaulting to ShapeUnknown.\n"
              << s.ToString();
    }
  }

  //  Construct Tensors for constant inputs used by shape functions.
  std::vector<Tensor> const_values(ic->num_inputs());
  std::vector<const Tensor*> input_tensors(ic->num_inputs(), nullptr);
  for (int dst_input = 0; dst_input < ic->num_inputs(); ++dst_input) {
    const TensorProto* tensor_proto = ctx->input_tensor_protos[dst_input];
    if (tensor_proto != nullptr &&
        // Skip if the const tensor is too large.
        NumElementsFromTensorProto(*tensor_proto) <=
            kThresholdToSkipConstTensorInstantiation &&
        const_values[dst_input].FromProto(*tensor_proto)) {
      input_tensors[dst_input] = &const_values[dst_input];
    }
  }
  ic->set_input_tensors(input_tensors);

  // Update the shapes of the outputs.
  return InferShapes(*node, ctx);
}

Status SetUnknownShape(const NodeDef* node, int output_port) {
  shape_inference::ShapeHandle shape =
      GetUnknownOutputShape(node, output_port);
  InferenceContext* ctx = GetContext(node);
  if (ctx == nullptr) {
    return errors::InvalidArgument("Missing context");
  }
  ctx->set_output(output_port, shape);
  return Status::OK();
}

struct ShapeId {
  const NodeDef* node;
  int port_id;
  bool operator==(const ShapeId& other) const {
    return node == other.node && port_id == other.port_id;
  }
};
struct HashShapeId {
  std::size_t operator()(const ShapeId& shp) const {
    return std::hash<const NodeDef*>{}(shp.node) + shp.port_id;
  }
};

struct DimId {
```

```
    const NodeDef* node;
    int port_id;
    int dim_index;
    bool operator==(const DimId& other) const {
      return node == other.node && port_id == other.port_id &&
             dim_index == other.dim_index;
    }
  };

  struct HashDimId {
    std::size_t operator()(const DimId& dim) const {
      return std::hash<const NodeDef*>{}(dim.node) + dim.port_id +
             dim.dim_index;
    }
  };

  // 'port_index' as the union of shape1 and shape2.
  ShapeHandle OutputAsUnion(const NodeDef* node, int port_index,
                            ShapeHandle shape1, ShapeHandle shape2) {
    if (shape1.SameHandle(shape2)) {
      return shape1;
    }
    InferenceContext* ctx = GetContext(node);
    ShapeHandle relaxed = shape1;
    const int rank = ctx->Rank(shape1);
    if (!ctx->RankKnown(shape2) || ctx->Rank(shape2) != rank) {
      relaxed = GetUnknownOutputShape(node, port_index);
    } else {
      for (int d = 0; d < rank; ++d) {
        if (!ctx->Dim(shape1, d).SameHandle(ctx->Dim(shape2, d))) {
          int64_t val1 = ctx->Value(ctx->Dim(shape1, d));
          int64_t val2 = ctx->Value(ctx->Dim(shape2, d));
          if (val1 != val2 || (val1 < 0 && val2 < 0)) {
            DimensionHandle new_dim = GetUnknownOutputDim(node, port_index, d);
            TF_CHECK_OK(ctx->ReplaceDim(relaxed, d, new_dim, &relaxed));
          }
        }
      }
    }
    return relaxed;
  }

  bool EquivalentShapes(ShapeHandle s1, ShapeHandle s2) const {
    if (s1.SameHandle(s2)) {
      return true;
    }
    if (InferenceContext::Rank(s1) != InferenceContext::Rank(s2)) {
      return false;
    }
```

```
    if (!InferenceContext::RankKnown(s1) && !InferenceContext::RankKnown(s2)) {
      return true;
    }
    const int rank = InferenceContext::Rank(s1);
    for (int i = 0; i < rank; ++i) {
      if (!InferenceContext::DimKnownRank(s1, i).SameHandle(
              InferenceContext::DimKnownRank(s2, i))) {
        int64_t val1 =
            InferenceContext::Value(InferenceContext::DimKnownRank(s1, i));
        int64_t val2 =
            InferenceContext::Value(InferenceContext::DimKnownRank(s2, i));
        if (val1 >= 0 && val2 >= 0 && val1 == val2) {
          continue;
        }
        return false;
      }
    }
    return true;
  }

  // Return true if the annotated shape is compatible with shape inference
  // result. Examples:
  // Inferred shape: ?, annotated shape: [10, 10] -> true;
  // Inferred shape: [-1, 10], annotated shape: [10, 10] -> true;
  // Inferred shape: [-1, 100], annotated shape: [10, 10] -> false;
  // Inferred shape: [-1, 10, 10], annotated shape: [10, 10] -> false.
  bool CompatibleShapes(ShapeHandle inferred_shape,
                        ShapeHandle annotated_shape) const {
    if (inferred_shape.SameHandle(annotated_shape)) {
      return true;
    }
    if (!InferenceContext::RankKnown(inferred_shape)) {
      return true;
    }
    if (InferenceContext::Rank(inferred_shape) !=
        InferenceContext::Rank(annotated_shape)) {
      return false;
    }
    const int rank = InferenceContext::Rank(inferred_shape);
    for (int i = 0; i < rank; ++i) {
      if (!InferenceContext::DimKnownRank(inferred_shape, i)
               .SameHandle(
                   InferenceContext::DimKnownRank(annotated_shape, i))) {
        int64_t val1 = InferenceContext::Value(
            InferenceContext::DimKnownRank(inferred_shape, i));
        int64_t val2 = InferenceContext::Value(
            InferenceContext::DimKnownRank(annotated_shape, i));
        if (val1 >= 0 && val1 != val2) {
          return false;
```

```
          }
        }
      }
      return true;
    }

    bool SameShapes(ShapeHandle inferred_shape,
                    ShapeHandle annotated_shape) const {
      if (inferred_shape.SameHandle(annotated_shape)) {
        return true;
      }
      if (InferenceContext::Rank(inferred_shape) !=
          InferenceContext::Rank(annotated_shape)) {
        return false;
      }
      const int rank = InferenceContext::Rank(inferred_shape);
      for (int i = 0; i < rank; ++i) {
        int64_t val1 = InferenceContext::Value(
            InferenceContext::DimKnownRank(inferred_shape, i));
        int64_t val2 = InferenceContext::Value(
            InferenceContext::DimKnownRank(annotated_shape, i));
        if (val1 != val2) {
          return false;
        }
      }
      return true;
    }

    bool EquivalentShapesAndTypes(const std::vector<ShapeAndType>& st1,
                                  const std::vector<ShapeAndType>& st2) const {
      if (st1.size() != st2.size()) {
        return false;
      }
      for (int i = 0, st1_size = st1.size(); i < st1_size; ++i) {
        const ShapeAndType& s1 = st1[i];
        const ShapeAndType& s2 = st2[i];
        if (s1.dtype != s2.dtype) {
          return false;
        }
        if (!EquivalentShapes(s1.shape, s2.shape)) {
          return false;
        }
      }
      return true;
    }

    Status AddFunction(const NodeDef* function_node, NameAttrList function) {
      auto it = fun_to_grappler_function_item_.find(function.name());
      if (it != fun_to_grappler_function_item_.end()) {
```

```
      return Status::OK();
    }

    const FunctionDef* function_def =
        CHECK_NOTNULL(function_library_.Find(function.name()));
    GrapplerFunctionItem grappler_function_item;
    Status function_instantiated =
        MakeGrapplerFunctionItem(*function_def, function_library_,
                                 graph_def_version_, &grappler_function_item);

    // If function instantiation failed we will skip it during shape inference.
    if (!function_instantiated.ok()) {
      VLOG(3) << "Failed to instantiate a function. Error: "
              << function_instantiated.error_message();
      fun_to_grappler_function_item_[function_def->signature().name()] =
          absl::nullopt;
      return Status::OK();
    }

    if (static_cast<int>(grappler_function_item.inputs().size()) >
        function_node->input_size()) {
      return errors::FailedPrecondition(
          "Function input size should be smaller than node input size.");
    }

    for (int i = grappler_function_item.inputs().size(),
             end = function_node->input_size();
         i < end; ++i) {
      const string& input = function_node->input(i);
      if (!IsControlInput(input)) {
        return errors::FailedPrecondition(
            "Found regular input (", input,
            ") instead of control nodes for node ", function_node->name());
      }
    }

    fun_to_grappler_function_item_[function_def->signature().name()] =
        grappler_function_item;

    return Status::OK();
  }

  Status AddNode(const NodeDef* node) {
    NodeContext& node_ctx = node_to_context_[node];
    NameAttrList function;
    TF_RETURN_IF_ERROR(NameAndAttrsFromFunctionCall(*node, &function));

    // For PartitionedCall, op_data represents the function info.
    TF_RETURN_IF_ERROR(
```

```
        function_library_.LookUp(function.name(), &node_ctx.op_data));

    if (node_ctx.op_data->is_function_op) {
      TF_RETURN_IF_ERROR(AddFunction(node, function));
    }

    TF_RETURN_IF_ERROR(InOutTypesForNode(*node, node_ctx.op_data->op_def,
                                         &node_ctx.input_types,
                                         &node_ctx.output_types));

    // Create the inference context for this node.
    const int num_inputs = node_ctx.input_types.size();
    std::vector<ShapeHandle> input_shapes(num_inputs);
    std::vector<std::unique_ptr<std::vector<ShapeAndType>>>
        input_handle_shapes_and_types(num_inputs);
    std::vector<const Tensor*> input_tensors(num_inputs, nullptr);
    std::vector<ShapeHandle> input_tensors_as_shapes;

    node_ctx.inference_context.reset(new InferenceContext(
        graph_def_version_, *node, node_ctx.op_data->op_def, input_shapes,
        input_tensors, input_tensors_as_shapes,
        std::move(input_handle_shapes_and_types)));
    const Status s = node_ctx.inference_context->construction_status();
    if (!s.ok()) {
      node_ctx.inference_context.reset(nullptr);
    }
    return s;
  }

 private:
  // Return the one ShapeHandle used to denote a fully unknown shape for a node
  // output.
  ShapeHandle GetUnknownOutputShape(const NodeDef* node, int index) {
    ShapeId id{node, index};
    auto it = unknown_shapes_.find(id);
    if (it != unknown_shapes_.end()) {
      return it->second;
    }
    InferenceContext* c = GetContext(node);
    ShapeHandle shp = c->UnknownShape();
    unknown_shapes_[id] = shp;
    return shp;
  }
  // Return the one ShapeHandle used to denote a fully unknown dimension for a
  // node output.
  DimensionHandle GetUnknownOutputDim(const NodeDef* node, int index,
                                      int dim_id) {
    DimId id{node, index, dim_id};
    auto it = unknown_dims_.find(id);
```

```
    if (it != unknown_dims_.end()) {
      return it->second;
    }
    InferenceContext* c = GetContext(node);
    DimensionHandle dim = c->UnknownDim();
    unknown_dims_[id] = dim;
    return dim;
  }

  // Returns true if all the output tensors have known values.
  bool AllOutputValuesKnown(NodeContext* c) {
    InferenceContext* ic = c->inference_context.get();
    int c_output_tensors_as_shapes_size = c->output_tensors_as_shapes.size();
    int c_output_tensor_protos_size = c->output_tensor_protos.size();
    if (c_output_tensors_as_shapes_size < ic->num_outputs() &&
        c_output_tensor_protos_size < ic->num_outputs()) {
      return false;
    } else {
      // Checks if we can get output value via either output_tensor_proto or
      // output_tensors_as_shapes.
      for (int i = 0; i < ic->num_outputs(); i++) {
        if (c_output_tensor_protos_size > i &&
            c->output_tensor_protos[i] != nullptr) {
          continue;
        }
        if (c_output_tensors_as_shapes_size > i &&
            ic->FullyDefined(c->output_tensors_as_shapes[i])) {
          bool no_unknown_dim_from_const = true;
          for (int32_t j = 0; j < ic->Rank(c->output_tensors_as_shapes[i]);
               ++j) {
            const auto dim = ic->Dim(c->output_tensors_as_shapes[i], j);
            if (ic->ValueKnown(dim) && ic->Value(dim) == kUnknownDimFromConst) {
              no_unknown_dim_from_const = false;
              break;
            }
          }
          if (no_unknown_dim_from_const) {
            continue;
          }
        }
        return false;
      }
    }
    return true;
  }

  // Returns true if all the output shapes are known.
  bool AllOutputShapesKnown(NodeContext* c) {
    InferenceContext* ic = c->inference_context.get();
```

```
    // Checks if all the output shapes are fully defined.
    for (int i = 0; i < ic->num_outputs(); i++) {
      if (!ic->FullyDefined(ic->output(i))) {
        return false;
      }
    }
    return true;
  }

  // Returns true if we can infer output tensors' values -- we know values of
  // all the input tensors.
  bool AllInputValuesKnown(NodeContext* c) {
    InferenceContext* ic = c->inference_context.get();

    // Check inputs are fully defined and values are known.
    for (int i = 0; i < ic->num_inputs(); i++) {
      const Tensor* tensor = ic->input_tensor(i);
      // Note that we don't check c->input_tensor_protos[i], as UpdateNode()
      // already converted it to ic->input_tensor(i);
      const ShapeHandle& input_tensors_as_shape =
          ic->input_tensors_as_shapes()[i];
      // Either input_tensor is valid or input_tensors_as_shape, which has
      // value of input tensors as shape format, should be fully defined.
      if (tensor == nullptr && !ic->FullyDefined(input_tensors_as_shape)) {
        return false;
      }
    }
    return true;
  }

  // Returns true if we want to update output shapes and values with running
  // EvaluateNode() for this op, based on op type, data type, and size.
  bool ShouldUpdateOutputShapesAndValues(NodeContext* c, int64_t max_size) {
    InferenceContext* ic = c->inference_context.get();

    // Due to the cost of running EvaluateNode(), we limit only to white listed
    // op types.
    if (!IsAllowListedOpTypeForEvaluateNode(c->op_data->op_def.name())) {
      return false;
    }

    // Check input dtypes are number types.
    for (const auto& input_type : c->input_types) {
      if (!IsNumericType(input_type)) {
        return false;
      }
    }

    // Check output dtypes are number types.
```

```
    for (const auto& output_type : c->output_types) {
      if (!IsNumericType(output_type)) {
        return false;
      }
    }

    // Check if the number of elements of each of input tensor is no larger than
    // the given max size.
    for (int i = 0; i < ic->num_inputs(); i++) {
      const Tensor* tensor = ic->input_tensor(i);
      const ShapeHandle& input_shape_handle = ic->input(i);
      if (tensor != nullptr) {
        if (tensor->NumElements() > max_size) {
          return false;
        }
      } else if (ic->Value(ic->NumElements(input_shape_handle)) > max_size) {
        return false;
      }
    }

    // Check if we know the shape of each output tensor, and the number of
    // elements is larger than the given max size.
    for (int i = 0; i < ic->num_outputs(); i++) {
      const ShapeHandle& shape_handle = ic->output(i);
      if (!ic->FullyDefined(shape_handle) ||
          ic->Value(ic->NumElements(shape_handle)) > max_size) {
        return false;
      }
    }
    return true;
  }

  // Create input tensors from the NodeContext.
  void CreateInputTensors(NodeContext* c,
                          std::vector<Tensor>* input_tensor_vector,
                          TensorVector* inputs) {
    InferenceContext* ic = c->inference_context.get();
    for (int i = 0; i < ic->num_inputs(); i++) {
      if (ic->input_tensor(i)) {
        input_tensor_vector->at(i) = *ic->input_tensor(i);
        inputs->emplace_back(&input_tensor_vector->at(i));
        // Note that we don't check c->input_tensor_protos[i], as UpdateNode()
        // already converted it to ic->input_tensor(i);
      } else {
        // Create Tensor from input_tensors_as_shapes, and then emplace it
        // back to inputs.
        // Note that input_tensors_as_shapes is scalar or vector.
        const ShapeHandle& shape_handle = ic->input_tensors_as_shapes()[i];
        const DataType& data_type = c->input_types[i];
```

```
        int32_t rank = ic->Rank(shape_handle);
        if (rank < 1) {
          input_tensor_vector->at(i) = Tensor(data_type, {});
        } else {
          input_tensor_vector->at(i) = Tensor(data_type, {rank});
        }
        auto* tensor = &input_tensor_vector->at(i);
        if (data_type == DT_INT32) {
          auto flat = tensor->flat<int32>();
          for (int j = 0; j < rank; j++) {
            int32_t dim = ic->Value(ic->Dim(shape_handle, j));
            flat(j) = dim;
          }
        } else {
          auto flat = tensor->flat<int64_t>();
          for (int j = 0; j < rank; j++) {
            int64_t dim = ic->Value(ic->Dim(shape_handle, j));
            flat(j) = dim;
          }
        }
        inputs->emplace_back(tensor);
      }
    }
  }

  // Run a node to infer output shapes and values, and add it to the
  // NodeContext.
  Status UpdateOutputShapesAndValues(const NodeDef& node, NodeContext* c) {
    InferenceContext* ic = c->inference_context.get();

    // Input to EvaluateNode()
    TensorVector inputs;
    // Container for temporarily created tensor object.
    std::vector<Tensor> input_tensor_vector(ic->num_inputs());
    CreateInputTensors(c, &input_tensor_vector, &inputs);

    // Output for EvaluateNode() and output tensor clean up object.
    TensorVector outputs;
    auto outputs_cleanup = gtl::MakeCleanup([&outputs] {
      for (const auto& output : outputs) {
        if (output.tensor) {
          delete output.tensor;
        }
      }
    });

    TF_RETURN_IF_ERROR(EvaluateNode(node, inputs, /*cpu_device=*/nullptr,
                                    &resource_mgr_, &outputs));
    c->output_tensors_as_shapes.resize(outputs.size());
```

```
    c->output_tensor_protos.resize(outputs.size(), nullptr);
    for (int k = 0, outputs_size = outputs.size(); k < outputs_size; k++) {
      const auto& t = outputs[k];
      // Override output shape.
      ShapeHandle output_shape;
      TF_RETURN_IF_ERROR(
          ic->MakeShapeFromTensorShape(t->shape(), &output_shape));
      if (ic->FullyDefined(ic->output(k)) &&
          !EquivalentShapes(ic->output(k), output_shape)) {
        LOG(WARNING) << "UpdateOutputShapesAndValues() -- node: " << node.name()
                     << ", inferred output shape "
                     << "doesn't match for k=" << k << ": "
                     << "ic->output(k): " << ic->DebugString(ic->output(k))
                     << ", output_shape: " << ic->DebugString(output_shape)
                     << " -- " << node.DebugString();
      }
      ic->set_output(k, output_shape);
      // Set output_tensors_as_shape.
      MaybeTensorValueToShape(ic, *t.tensor, &c->output_tensors_as_shapes[k]);

      // Set output_tensor_protos.
      TensorProto tensor_proto;
      t->AsProtoTensorContent(&tensor_proto);
      const_tensors_to_propagate_.push_back(tensor_proto);
      c->output_tensor_protos[k] = &const_tensors_to_propagate_.back();
    }
    return Status::OK();
  }

  // Update output shapes with annotated information.
  // Currently only handle nodes with static shapes, i.e. shapes do not change
  // during execution.
  // TODO(andiryxu): Use annotated shapes in Enter/Merge etc as well.
  Status UpdateOutputShapesUsingAnnotatedInformation(const NodeDef& node,
                                                     NodeContext* c) const {
    const auto& attr = node.attr();
    if (attr.count(kOutputSame) == 0 || !attr.at(kOutputSame).b() ||
        attr.count(kOutputShapes) == 0)
      return Status::OK();

    InferenceContext* ic = c->inference_context.get();
    int output_size = attr.at(kOutputShapes).list().shape_size();

    for (int i = 0; i < ic->num_outputs(); i++) {
      // Annotated Switch node has only one output. Propagate the shape to all
      // the outputs.
      int shape_index = IsSwitch(node) ? 0 : i;
      if (shape_index >= output_size) {
        LOG(WARNING)
```

```
            << "UpdateOutputShapesUsingAnnotatedInformation() -- node: "
            << node.name() << ", inferred output shape size "
            << ic->num_outputs() << ", annotated output shape size "
            << output_size;
        break;
      }

      const TensorShapeProto& shape =
          attr.at(kOutputShapes).list().shape(shape_index);
      if (shape.dim().empty()) continue;

      ShapeHandle output_shape;
      TF_RETURN_IF_ERROR(ic->MakeShapeFromShapeProto(shape, &output_shape));

      // Check if annotated shapes are incompatible with inferred shapes.
      if ((ic->FullyDefined(ic->output(i)) &&
           !SameShapes(ic->output(i), output_shape)) ||
          (!ic->FullyDefined(ic->output(i)) &&
           !CompatibleShapes(ic->output(i), output_shape))) {
        LOG(WARNING)
            << "UpdateOutputShapesUsingAnnotatedInformation() -- node: "
            << node.name() << ", inferred output shape "
            << "doesn't match for i=" << i << ": "
            << "ic->output(k): " << ic->DebugString(ic->output(i))
            << ", annotated output shape: " << ic->DebugString(output_shape)
            << " -- " << node.DebugString();
        c->shape_incompatible = true;
      }

      // Only use annotated shapes if the inference shape is unknown and
      // compatible with annotated shapes.
      if (!ic->FullyDefined(ic->output(i)) &&
          CompatibleShapes(ic->output(i), output_shape)) {
        VLOG(3) << "UpdateOutputShapesUsingAnnotatedInformation() -- node: "
                << node.name() << ", inferred output shape " << i << ": "
                << "ic->output(i): " << ic->DebugString(ic->output(i))
                << ", annotated output shape: " << ic->DebugString(output_shape)
                << " -- " << node.ShortDebugString();
        ic->set_output(i, output_shape);
      }
    }

    return Status::OK();
  }

  Status MaybeUpdateNodeContextOutput(const NodeDef& node, const bool is_fed,
                                      NodeContext* c) {
    // Propagate tensors and shape tensors unless the node is fed.
    // TODO(bsteiner) We should still propagate the shapes to the ports that
```

```
    // aren't fed in the case of a ShapeN node.

    // Note that when propagating tensors_as_shapes, we use
    // c->input_tensors_as_shapes_to_progate instead of
    // ic->input_tensors_as_shapes. The former uses kUnknownDimFromConst if
    // UnknownDim is from Const tensor, and it is propagated through shape
    // inference. Before calling shape functions, we convert it to UnknownDim,
    // but instantiate a new UnknownDim to prevent incorrect symbolic shape
    // inference through UnknownDim from Const.
    InferenceContext* ic = c->inference_context.get();
    if (!is_fed) {
      if (IsConstant(node)) {
        const TensorProto& tensor_proto = node.attr().at("value").tensor();
        c->output_tensor_protos.resize(1);
        c->output_tensor_protos[0] = &tensor_proto;
        c->output_tensors_as_shapes.resize(1);
        MaybeTensorProtoToShape(ic, tensor_proto,
                                &c->output_tensors_as_shapes[0]);
      } else if (IsRank(node)) {
        if (ic->RankKnown(ic->input(0))) {
          // Propagate rank value.
          int32_t rank = ic->Rank(ic->input(0));
          const_tensors_to_propagate_.push_back(
              MakeIntegerScalarTensorProto(DT_INT32, rank));
          c->output_tensor_protos.resize(1);
          c->output_tensor_protos[0] = &const_tensors_to_propagate_.back();
        }
      } else if (IsSize(node)) {
        DimensionHandle size = ic->NumElements(ic->input(0));
        if (ic->ValueKnown(size)) {
          // Propagate size value.
          int64_t sz = ic->Value(size);
          bool valid = false;
          if (node.attr().at("out_type").type() == DT_INT32) {
            if (sz < std::numeric_limits<int32>::max()) {
              const_tensors_to_propagate_.push_back(
                  MakeIntegerScalarTensorProto(DT_INT32, sz));
              valid = true;
            }
          } else {
            const_tensors_to_propagate_.push_back(
                MakeIntegerScalarTensorProto(DT_INT64, sz));
            valid = true;
          }
          if (valid) {
            c->output_tensor_protos.resize(1);
            c->output_tensor_protos[0] = &const_tensors_to_propagate_.back();
          }
        }
```

```
    } else if (IsShape(node)) {
      c->output_tensors_as_shapes.resize(1);
      c->output_tensors_as_shapes[0] = c->inference_context->input(0);
    } else if (IsShapeN(node)) {
      c->output_tensors_as_shapes.resize(c->inference_context->num_inputs());
      for (int i = 0; i < c->inference_context->num_inputs(); ++i) {
        c->output_tensors_as_shapes[i] = c->inference_context->input(i);
      }
    } else if (node.op() == "ConcatV2") {
      bool valid = true;
      ShapeHandle result;
      for (int i = 0; i < ic->num_inputs() - 1; ++i) {
        ShapeHandle input = c->input_tensors_as_shapes_to_propagate[i];
        if (!ic->RankKnown(input)) {
          valid = false;
          break;
        } else if (i == 0) {
          result = input;
        } else {
          TF_RETURN_IF_ERROR(ic->Concatenate(result, input, &result));
        }
      }
      if (valid) {
        c->output_tensors_as_shapes.resize(1);
        c->output_tensors_as_shapes[0] = result;
      }
    } else if (IsPack(node)) {
      // A Pack node concatenating scalars is often used to generate a shape.
      std::vector<DimensionHandle> dims;
      bool valid = true;
      for (int i = 0; i < ic->num_inputs(); ++i) {
        const Tensor* t = ic->input_tensor(i);
        if (t) {
          if (t->dims() != 0 ||
              (t->dtype() != DT_INT32 && t->dtype() != DT_INT64)) {
            valid = false;
            break;
          }
          int64_t size = t->dtype() == DT_INT32 ? t->scalar<int32>()()
                                                : t->scalar<int64_t>()();
          dims.push_back(size < 0 ? ic->MakeDim(kUnknownDimFromConst)
                                  : ic->MakeDim(size));
        } else {
          // Don't have tensor value, but use input_tensors_as_shapes, if
          // possible.
          const ShapeHandle& shape_handle =
              c->input_tensors_as_shapes_to_propagate[i];
          if (ic->RankKnown(shape_handle) && ic->Rank(shape_handle) >= 1 &&
              ic->ValueKnown(ic->Dim(shape_handle, 0))) {
```

```
            dims.push_back(ic->Dim(shape_handle, 0));
          } else {
            // This is not from Const, but as it shouldn'be used as symbolic
            // unknown dim for different ops, we use kUnknownDimFromConst.
            dims.push_back(ic->MakeDim(kUnknownDimFromConst));
          }
        }
      }
      if (valid) {
        c->output_tensors_as_shapes.resize(1);
        c->output_tensors_as_shapes[0] = ic->MakeShape(dims);
      }
    } else if (IsIdentity(node) || IsIdentityNSingleInput(node)) {
      c->output_tensors_as_shapes.resize(1);
      c->output_tensors_as_shapes[0] =
          c->input_tensors_as_shapes_to_propagate[0];
      if (c->input_tensor_protos[0] != nullptr) {
        c->output_tensor_protos.resize(1);
        c->output_tensor_protos[0] = c->input_tensor_protos[0];
      }
    } else if (IsSlice(node)) {
      ShapeHandle input = c->input_tensors_as_shapes_to_propagate[0];
      bool valid = ic->RankKnown(input);
      const Tensor* slice_offset = ic->input_tensor(1);
      valid &= slice_offset != nullptr && slice_offset->NumElements() == 1;
      const Tensor* slice_size = ic->input_tensor(2);
      valid &= slice_size != nullptr && slice_size->NumElements() == 1;
      if (valid) {
        int64_t start = slice_offset->dtype() == DT_INT32
                            ? slice_offset->flat<int32>()(0)
                            : slice_offset->flat<int64_t>()(0);
        int64_t size = (slice_size->dtype() == DT_INT32
                            ? slice_size->flat<int32>()(0)
                            : slice_size->flat<int64_t>()(0));
        ShapeHandle result;
        if (size == -1) {
          TF_RETURN_IF_ERROR(ic->Subshape(input, start, &result));
        } else {
          int64_t end = start + size;
          TF_RETURN_IF_ERROR(ic->Subshape(input, start, end, &result));
        }
        c->output_tensors_as_shapes.resize(1);
        c->output_tensors_as_shapes[0] = result;
      }
    } else if (IsStridedSlice(node)) {
      ShapeHandle input = c->input_tensors_as_shapes_to_propagate[0];
      bool valid = ic->RankKnown(input);
      const Tensor* slice_begin = ic->input_tensor(1);
      valid &= slice_begin != nullptr && slice_begin->NumElements() == 1;
```

```
        const Tensor* slice_end = ic->input_tensor(2);
        valid &= slice_end != nullptr && slice_end->NumElements() == 1;
        const Tensor* slice_stride = ic->input_tensor(3);
        valid &= slice_stride != nullptr && slice_stride->NumElements() == 1;

        if (node.attr().count("ellipsis_mask") > 0 &&
            node.attr().at("ellipsis_mask").i() != 0) {
          valid = false;
        }
        if (node.attr().count("new_axis_mask") > 0 &&
            node.attr().at("new_axis_mask").i() != 0) {
          valid = false;
        }
        if (node.attr().count("shrink_axis_mask") > 0 &&
            node.attr().at("shrink_axis_mask").i() != 0) {
          valid = false;
        }
        int begin_mask = 0;
        if (node.attr().count("begin_mask") > 0) {
          begin_mask = node.attr().at("begin_mask").i();
        }
        int end_mask = 0;
        if (node.attr().count("end_mask") > 0) {
          end_mask = node.attr().at("end_mask").i();
        }
        if (begin_mask < 0 || begin_mask > 1 || end_mask < 0 || end_mask > 1) {
          valid = false;
        }
        if (valid) {
          int64_t begin = 0;
          if (begin_mask == 0) {
            begin = slice_begin->dtype() == DT_INT32
                        ? slice_begin->flat<int32>()(0)
                        : slice_begin->flat<int64_t>()(0);
          }
          int64_t end = std::numeric_limits<int64_t>::max();
          if (end_mask == 0) {
            end = (slice_end->dtype() == DT_INT32
                       ? slice_end->flat<int32>()(0)
                       : slice_end->flat<int64_t>()(0));
          }
          int64_t stride = slice_stride->dtype() == DT_INT32
                               ? slice_stride->flat<int32>()(0)
                               : slice_stride->flat<int64_t>()(0);
          ShapeHandle result;
          TF_RETURN_IF_ERROR(ic->Subshape(input, begin, end, stride, &result));
          c->output_tensors_as_shapes.resize(1);
          c->output_tensors_as_shapes[0] = result;
        }
```

```
      }
    }

    if (aggressive_shape_inference_) {
      // Update output shapes with annotated information. This is optional.
      UpdateOutputShapesUsingAnnotatedInformation(node, c).IgnoreError();

      // Update output tensor values using EvaluateNode() if we can.
      // Due to the cost of EvaluateNode(), we run it only for certain op types
      // (white listed) and small integer tensors.

      const int max_element_size = 17;  // Max up to 4x4 matrix or similar.
      if (AllOutputValuesKnown(c) || !AllInputValuesKnown(c) ||
          !ShouldUpdateOutputShapesAndValues(c, max_element_size)) {
        return Status::OK();
      }
      UpdateOutputShapesAndValues(node, c).IgnoreError();  // This is optional.
    }
    return Status::OK();
  }

  Status InferShapes(const NodeDef& node, NodeContext* c) {
    // Infer the shapes of output tensors.
    if (!c->op_data || c->op_data->shape_inference_fn == nullptr ||
        !c->inference_context->Run(c->op_data->shape_inference_fn).ok()) {
      // Annotate outputs with unknown shapes. Update output shapes with
      // annotated information later on if available.
      // Note that shape inference function may return an error, but we ignore
      // it, and use UnknownShape in that case.
      TF_RETURN_IF_ERROR(
          c->inference_context->Run(shape_inference::UnknownShape));
    }
    Status status = Status::OK();
    auto it = fed_ports_.find(node.name());
    const bool is_fed = it != fed_ports_.end();
    if (is_fed) {
      // It is possible to feed node output ports with tensors of any shape: as
      // a result, the shape of a fed port is completely unknown.
      for (const int output_port : it->second) {
        status.Update(SetUnknownShape(&node, output_port));
      }
    }

    // Update NodeContext output fields after shape inference function runs.
    status.Update(MaybeUpdateNodeContextOutput(node, is_fed, c));

    return status;
  }
```

```
 private:
  bool IsIntegerVector(const Tensor& tensor) {
    if (tensor.dims() == 1 &&
        (tensor.dtype() == DT_INT32 || tensor.dtype() == DT_INT64)) {
      return true;
    }
    return false;
  }

  bool IsIntegerScalar(const Tensor& tensor) {
    if (tensor.dims() == 0 &&
        (tensor.dtype() == DT_INT32 || tensor.dtype() == DT_INT64) &&
        tensor.NumElements() == 1) {
      return true;
    }
    return false;
  }

  TensorProto MakeIntegerScalarTensorProto(const DataType dtype,
                                           const int64_t val) {
    TensorProto tensor_proto;
    tensor_proto.set_dtype(dtype);
    // Scalar TensorProto has an empty tensor_shape; no dim, no dim.size.
    tensor_proto.mutable_tensor_shape();
    if (dtype == DT_INT32) {
      tensor_proto.add_int_val(val);
    } else if (dtype == DT_INT64) {
      tensor_proto.add_int64_val(val);
    }
    return tensor_proto;
  }

  bool MaybeTensorProtoToShape(InferenceContext* ic,
                               const TensorProto& tensor_proto,
                               ShapeHandle* tensors_as_shapes) {
    // Skip if dtype is not integer.
    if (tensor_proto.dtype() != DT_INT32 && tensor_proto.dtype() != DT_INT64) {
      return false;
    }
    // Skip if the const tensor is too large.
    if (NumElementsFromTensorProto(tensor_proto) >
        kThresholdToSkipConstTensorInstantiation) {
      return false;
    }
    // Skip if shape is neither scalar nor vector.
    if (tensor_proto.tensor_shape().unknown_rank() ||
        tensor_proto.tensor_shape().dim_size() > 1) {
      return false;
    }
```

```
    Tensor tensor;
    if (!tensor.FromProto(tensor_proto)) {
      return false;
    }
    return MaybeTensorValueToShape(ic, tensor, tensors_as_shapes);
  }

  bool MaybeTensorValueToShape(InferenceContext* ic, const Tensor& tensor,
                               ShapeHandle* tensors_as_shapes) {
    // Integer tensors of rank one can also be interpreted as a shape
    // provided all their values are >= -1.

    if (IsIntegerVector(tensor)) {
      bool has_values_smaller_than_minus_1 = false;
      std::vector<DimensionHandle> dims;
      for (int i = 0; i < tensor.NumElements(); i++) {
        int64_t value = tensor.dtype() == DT_INT32 ? tensor.flat<int32>()(i)
                                                   : tensor.flat<int64_t>()(i);
        has_values_smaller_than_minus_1 |= (value < -1);
        // Mark this as UnknownDim from Const.
        dims.push_back(value < 0 ? ic->MakeDim(kUnknownDimFromConst)
                                 : ic->MakeDim(value));
      }

      if (!has_values_smaller_than_minus_1) {
        *tensors_as_shapes = ic->MakeShape(dims);
        return true;
      }
    } else if (IsIntegerScalar(tensor)) {
      // Scalar constant.
      int64_t value = tensor.dtype() == DT_INT32 ? tensor.flat<int32>()(0)
                                                 : tensor.flat<int64_t>()(0);
      if (value == -1) {
        // Scalar value -1 represents an unknown shape. If we would try to
        // MakeShape(MakeDim) with it, we would get vector of unknown size.
        *tensors_as_shapes = ic->UnknownShape();
        return true;
      } else if (value >= 0) {
        // Ideally, values can be < -1, but MakeDim() fails with a value < -1.
        // It's a limitation as we use ShapeHandle as a means to pass values.
        *tensors_as_shapes = ic->MakeShape({ic->MakeDim(value)});
        return true;
      }
    }
    return false;
  }

  const GraphView& graph_;
  int graph_def_version_;
```

```
  absl::flat_hash_map<const NodeDef*, NodeContext> node_to_context_;
  absl::flat_hash_map<ShapeId, ShapeHandle, HashShapeId> unknown_shapes_;
  absl::flat_hash_map<DimId, DimensionHandle, HashDimId> unknown_dims_;
  // Store function instantiations only for valid function. If function
  // instantiation failed it will have an `absl::nullopt`.
  absl::flat_hash_map<string, absl::optional<GrapplerFunctionItem>>
      fun_to_grappler_function_item_;
  FunctionLibraryDefinition function_library_;
  const absl::flat_hash_map<string, absl::flat_hash_set<int>>& fed_ports_;
  // Store TensorProtos for tensor value propagation. Note that we use deque,
  // not vector, as we use pointers to the TensorProtos in this container.
  // Vector may resize and copy the objects into a new buffer, then the existing
  // pointers become dangling pointers.
  std::deque<TensorProto> const_tensors_to_propagate_;

  // For more aggressive shape and value inference.
  bool aggressive_shape_inference_;
  ResourceMgr resource_mgr_;
};

// Keep track of shapes and dimensions in a graph.
// In particular, use disjoint sets to track equivalence between shapes and
// dims, and consolidate the information globally.
class SymbolicShapeManager {
 public:
  SymbolicShapeManager() {}

  Status Merge(ShapeHandle s1, ShapeHandle s2) {
    if (!s1.IsSet() || !s2.IsSet()) {
      return Status::OK();
    }
    TF_RETURN_IF_ERROR(shapes_.Merge(s1, s2));
    if (InferenceContext::Rank(s1) > 0 && InferenceContext::Rank(s2) > 0) {
      CHECK_EQ(InferenceContext::Rank(s1), InferenceContext::Rank(s2));
      for (int i = 0; i < InferenceContext::Rank(s1); ++i) {
        TF_RETURN_IF_ERROR(dims_.Merge(InferenceContext::DimKnownRank(s1, i),
                                       InferenceContext::DimKnownRank(s2, i)));
      }
    }
    return Status::OK();
  }
  Status Merge(DimensionHandle d1, DimensionHandle d2) {
    if (!d1.IsSet() || !d2.IsSet()) {
      return Status::OK();
    }
    return dims_.Merge(d1, d2);
  }

  void AsTensorProperties(const ShapeHandle& shape, const DataType& type,
```

```
                                OpInfo::TensorProperties* properties) {
    properties->set_dtype(type);
    ShapeHandle actual_shape = shapes_.GetMergedValue(shape);
    if (!InferenceContext::RankKnown(actual_shape)) {
      properties->mutable_shape()->set_unknown_rank(true);
    } else {
      for (int j = 0; j < InferenceContext::Rank(actual_shape); ++j) {
        shape_inference::DimensionHandle dim =
            InferenceContext::DimKnownRank(actual_shape, j);
        int64_t d = dims_.GetMergedValue(dim);
        properties->mutable_shape()->add_dim()->set_size(d);
      }
    }
  }

  // Returns merged shape with merged dimensions.
  ShapeHandle GetMergedShape(InferenceContext* ic, ShapeHandle s) {
    const auto& actual_shape = shapes_.GetMergedValue(s);
    if (!InferenceContext::RankKnown(actual_shape)) {
      return ic->UnknownShape();
    } else {
      std::vector<DimensionHandle> dims;
      for (int j = 0; j < InferenceContext::Rank(actual_shape); ++j) {
        shape_inference::DimensionHandle dim =
            InferenceContext::DimKnownRank(actual_shape, j);
        int64_t d = dims_.GetMergedValue(dim);
        // Symbolic shape manager may made some dims < -1, which causes errors
        // in creating Dimension.
        if (d < -1) {
          d = -1;
        }
        dims.push_back(ic->MakeDim(d));
      }
      return ic->MakeShape(dims);
    }
  }

 private:
  DisjointSet<shape_inference::ShapeHandle> shapes_;
  DisjointSet<shape_inference::DimensionHandle> dims_;
};

// Checks whether there is any conflict in merged shapes and dims in
// SymbolicShapeManager.
Status ValidateSymbolicShapeManager(const GraphDef& graph_def,
                                    SymbolicShapeRefiner* refiner,
                                    SymbolicShapeManager* shape_manager) {
  if (!VLOG_IS_ON(1)) {
    return Status::OK();
```

```
  }

  VLOG(1) << "Checking any conflics in shapes and dimensions ...";
  int64_t num_incompatible_shapes = 0;
  for (const NodeDef& node : graph_def.node()) {
    auto ctx = refiner->GetNodeContext(&node);
    if (!ctx) {
      continue;
    }
    auto* ic = ctx->inference_context.get();
    for (int i = 0; i < ic->num_inputs(); ++i) {
      const auto& shape = ic->input(i);
      const auto& merged_shape = shape_manager->GetMergedShape(ic, shape);
      if (!refiner->CompatibleShapes(shape, merged_shape)) {
        num_incompatible_shapes++;
        VLOG(1) << "**** Incompatible shape from SymbolicShapeManager "
                << "for node " << node.name() << " input (" << i << ") "
                << ic->DebugString(shape)
                << " vs. merged: " << ic->DebugString(merged_shape);
      }
    }
    for (int i = 0; i < ic->num_outputs(); ++i) {
      const auto& shape = ic->output(i);
      const auto& merged_shape = shape_manager->GetMergedShape(ic, shape);
      if (!refiner->CompatibleShapes(shape, merged_shape)) {
        num_incompatible_shapes++;
        VLOG(1) << "**** Incompatible shape from SymbolicShapeManager "
                << "for node " << node.name() << " output (" << i << ") "
                << ic->DebugString(shape)
                << " vs. merged: " << ic->DebugString(merged_shape);
      }
    }
  }
  if (num_incompatible_shapes > 0) {
    VLOG(1) << "**** WARNING: " << num_incompatible_shapes
            << " incompatible shapes from SymbolicShapeManager.";
  } else {
    VLOG(1) << "**** No incompatible shape found from SymbolicShapeManager.";
  }

  return Status::OK();
}

// Log shape inference and its merged shapes.
Status VerboseShapeInferenceLogging(const GraphDef& graph_def,
                                    SymbolicShapeRefiner* refiner,
                                    SymbolicShapeManager* shape_manager) {
  // As logging all the nodes would generate too many lines, we by default
  // skip this detailed logging. Users may add nodes of interest to
```

```
  // node_names_for_logging to enable detailed logging.
  absl::flat_hash_set<std::string> node_names_for_logging = {};
  if (!VLOG_IS_ON(3) || node_names_for_logging.empty()) {
    return Status::OK();
  }

  auto should_log = [&node_names_for_logging](std::string node_name) {
    return node_names_for_logging.find(node_name) !=
           node_names_for_logging.end();
  };

  for (const NodeDef& node : graph_def.node()) {
    if (!should_log(node.name())) {
      continue;
    }
    auto ctx = refiner->GetNodeContext(&node);
    if (!ctx) {
      continue;
    }
    auto* ic = ctx->inference_context.get();
    VLOG(3) << "Shape inference for node : " << node.name();
    VLOG(3) << ctx->DebugString(node);
    std::string merged_shapes = "Merged shapes from SymbolicShapManager:\n";
    for (int i = 0; i < ic->num_inputs(); ++i) {
      absl::StrAppend(
          &merged_shapes, " input[", i, "] -- ",
          ic->DebugString(shape_manager->GetMergedShape(ic, ic->input(i))),
          "\n");
    }
    for (int i = 0; i < ic->num_outputs(); ++i) {
      absl::StrAppend(
          &merged_shapes, " output[", i, "] -- ",
          ic->DebugString(shape_manager->GetMergedShape(ic, ic->output(i))),
          "\n");
    }
    VLOG(3) << merged_shapes;
    VLOG(3) << "--------------------------------";
    VLOG(3) << "";
  }

  return Status::OK();
}

Status GraphProperties::RelaxEnqueueShapesAndMergeTypes(
    SymbolicShapeRefiner* shape_refiner, const NodeDef* qnode,
    const std::vector<ShapeAndType>& shapes_and_types,
    std::vector<ShapeAndType>* queue_shapes_and_types) {
  if (shapes_and_types.size() != queue_shapes_and_types->size()) {
    return errors::InvalidArgument(
```

```
        "Enqueue nodes mixed number of tensors: ", shapes_and_types.size(),
        "  vs ", queue_shapes_and_types->size());
  }
  for (size_t i = 0; i < shapes_and_types.size(); ++i) {
    const ShapeAndType& a = shapes_and_types[i];
    ShapeAndType& b = (*queue_shapes_and_types)[i];
    if (a.dtype != b.dtype) {
      return errors::InvalidArgument("Enqueue nodes mixed dtypes for tensor ",
                                     i, ": ", DataTypeString(a.dtype), " vs ",
                                     DataTypeString(b.dtype));
    }

    b.shape = shape_refiner->OutputAsUnion(qnode, i, a.shape, b.shape);
  }
  return Status::OK();
}

// Compute the output shape of the merge node as the union of the available
// input shapes.
Status GraphProperties::UpdateMerge(SymbolicShapeRefiner* shape_refiner,
                                    const NodeDef* node,
                                    bool* new_shapes) const {
  InferenceContext* ic = shape_refiner->GetContext(node);
  if (!ic) {
    // Now we can run shape inference
    TF_RETURN_IF_ERROR(shape_refiner->AddNode(node));
    ic = CHECK_NOTNULL(shape_refiner->GetContext(node));
    *new_shapes = true;

    // Infer the shape of the second output once and for all since it never
    // changes.
    ShapeHandle out1 = ic->Scalar();
    if (ic->num_outputs() >= 2) ic->set_output(1, out1);
  }

  ShapeHandle out;
  const std::vector<ShapeAndType>* out_handle = nullptr;
  bool out_initialized = false;
  for (const GraphView::Edge fanin : shape_refiner->graph().GetFaninEdges(
           *node, /*include_controlling_edges=*/false)) {
    InferenceContext* src_ic = shape_refiner->GetContext(fanin.src.node);
    if (!src_ic) {
      // Handling a loop for the first time, the back edge won't have any shape
      // info.
      continue;
    }
    ShapeHandle input = src_ic->output(fanin.src.port_id);
    ic->SetInput(fanin.dst.port_id, input);
    auto* input_handle =
```

```
          src_ic->output_handle_shapes_and_types(fanin.src.port_id);
      if (input_handle)
        ic->set_input_handle_shapes_and_types(fanin.dst.port_id, *input_handle);
      if (!out_initialized) {
        out_initialized = true;
        out = input;
        out_handle = input_handle;
      } else {
        // Note here only out, not out_handle, is modified.
        out = shape_refiner->OutputAsUnion(node, 0, input, out);
      }
    }

  if (*new_shapes || !shape_refiner->EquivalentShapes(out, ic->output(0))) {
    ic->set_output(0, out);
    if (out_handle) ic->set_output_handle_shapes_and_types(0, *out_handle);
    *new_shapes = true;
  }

  return Status::OK();
}

// Manually propagate the input shape for Enter nodes.
Status GraphProperties::UpdateEnter(SymbolicShapeRefiner* shape_refiner,
                                    const NodeDef* node, bool* new_shapes) {
  InferenceContext* ic = shape_refiner->GetContext(node);
  if (!ic) {
    TF_RETURN_IF_ERROR(shape_refiner->UpdateNode(node, new_shapes));
    ic = shape_refiner->GetContext(node);
  }

  GraphView::InputPort port(node, 0);
  GraphView::OutputPort fanin = shape_refiner->graph().GetRegularFanin(port);

  InferenceContext* src_ic = shape_refiner->GetContext(fanin.node);
  ShapeHandle input = src_ic->output(fanin.port_id);
  if (!ic->output(0).SameHandle(input)) {
    ic->SetInput(0, input);
    ic->set_output(0, input);
    *new_shapes = true;
  }
  auto* outputs = src_ic->output_handle_shapes_and_types(fanin.port_id);
  if (outputs) {
    ic->set_input_handle_shapes_and_types(0, *outputs);
    ic->set_output_handle_shapes_and_types(0, *outputs);
    *new_shapes = true;
  }
  return Status::OK();
}
```

```

Status GraphProperties::UpdateShapes(
    SymbolicShapeRefiner* shape_refiner,
    const absl::flat_hash_map<const NodeDef*, const NodeDef*>& resource_handles,
    const NodeDef* n, bool* new_shapes) const {
  if (IsEnter(*n)) {
    // The Enter shape function always forwards an UnknownShape, so do the right
    // thing here.
    TF_RETURN_IF_ERROR(UpdateEnter(shape_refiner, n, new_shapes));
  } else if (IsMerge(*n)) {
    // Properly handle merge nodes.
    TF_RETURN_IF_ERROR(UpdateMerge(shape_refiner, n, new_shapes));
  } else if (IsEnqueue(*n)) {
    // Make sure the shapes of enqueued tensors are propagated to the queue
    // itself.
    TF_RETURN_IF_ERROR(
        UpdateEnqueue(n, resource_handles, shape_refiner, new_shapes));
  } else if (IsQueue(*n)) {
    // Set shapes and types of Queue ops, if needed.
    TF_RETURN_IF_ERROR(UpdateQueue(n, shape_refiner, new_shapes));
  } else {
    // Rely on regular TF shape refinement for all the other nodes.
    // UpdateNode calls UpdateFunction if a function node is detected.
    TF_RETURN_IF_ERROR(shape_refiner->UpdateNode(n, new_shapes));
  }

  return Status::OK();
}

// Propagates the shapes in the transitive fan-out of <new_shapes>.
Status GraphProperties::PropagateShapes(
    SymbolicShapeRefiner* shape_refiner, TopoQueue* new_shapes,
    const absl::flat_hash_map<const NodeDef*, const NodeDef*>& resource_handles,
    int num_loops) const {
  // Limit the number of iterations to prevent infinite loops in the presence of
  // incorrect shape functions. The algorithm should converge in at most
  // num_nested_loops^2 * max_rank. We approximate max_rank with the constant 4.
  // The same applies to resources.
  VLOG(1) << "Propagating " << new_shapes->size() << " new shapes through "
          << num_loops << " loops and " << resource_handles.size()
          << " resources" << std::endl;

  const int64_t max_loop_length = item_.graph.node_size();
  const int64_t max_rank = 4;
  const int64_t max_loop_iterations =
      max_rank * max_loop_length * std::max<int64_t>(1, num_loops * num_loops);
  const int64_t num_queues = resource_handles.size();
  const int64_t max_resource_iterations = num_queues * num_queues * max_rank;

```

```
  int64_t num_resource_iterations = 0;
  do {
    int64_t num_loop_iterations = 0;
    while (!new_shapes->empty() &&
           num_loop_iterations++ < max_loop_iterations) {
      const NodeDef* n = new_shapes->pop();
      bool updated = false;
      TF_RETURN_IF_ERROR(
          UpdateShapes(shape_refiner, resource_handles, n, &updated));
      if (updated) {
        for (const auto& fanout : shape_refiner->graph().GetFanouts(
                 *n, /*include_controlled_nodes=*/false)) {
          new_shapes->push(fanout.node);
        }
        // Make sure the corresponding queue nodes are (re)processed.
        if (IsEnqueue(*n)) {
          auto it = resource_handles.find(n);
          if (it != resource_handles.end()) {
            new_shapes->push(it->second);
          }
        }
      }
    }
  } while (!new_shapes->empty() &&
           num_resource_iterations++ < max_resource_iterations);

  if (!new_shapes->empty()) {
    return errors::Internal("Shape inference failed to converge");
  }

  return Status::OK();
}

Status GraphProperties::UpdateQueue(const NodeDef* queue_node,
                                    SymbolicShapeRefiner* shape_refiner,
                                    bool* new_shapes) {
  auto* ctx = shape_refiner->GetNodeContext(queue_node);
  if (!ctx) {
    TF_RETURN_IF_ERROR(shape_refiner->AddNode(queue_node));
    ctx = CHECK_NOTNULL(shape_refiner->GetNodeContext(queue_node));
  }
  auto* ic = ctx->inference_context.get();

  auto* outputs = ic->output_handle_shapes_and_types(0);
  if (outputs) {
    // Shapes and types are already set, presumably by Enqueue ops.
    return shape_refiner->UpdateNode(queue_node, new_shapes);
  }

```

```
  if (queue_node->attr().count("shapes") <= 0 ||
      queue_node->attr().count("component_types") <= 0 ||
      queue_node->attr().at("shapes").list().shape_size() !=
          queue_node->attr().at("component_types").list().type_size()) {
    // Errors in shapes and component_types attr.
    return shape_refiner->UpdateNode(queue_node, new_shapes);
  }

  // Extract types and shapes from Queue attr.
  const auto& shapes = queue_node->attr().at("shapes").list().shape();
  const auto& types = queue_node->attr().at("component_types").list().type();
  std::vector<ShapeAndType> shapes_and_types;
  for (int i = 0; i < types.size(); i++) {
    const auto& shape = shapes[i];
    ShapeHandle shape_handle;
    TF_RETURN_IF_ERROR(
        ic->MakeShapeFromPartialTensorShape(shape, &shape_handle));
    DataType data_type =
        queue_node->attr().at("component_types").list().type(i);
    ShapeAndType shape_and_type(shape_handle, data_type);
    shapes_and_types.push_back(shape_and_type);
  }
  ic->set_output_handle_shapes_and_types(0, shapes_and_types);

  // Queue node is updated with output_handle_shapes_and_types, so set
  // new_shapes and ignore it from UpdateNoe().
  *new_shapes = true;
  bool dummy_new_shapes = false;
  return shape_refiner->UpdateNode(queue_node, &dummy_new_shapes);
}

Status GraphProperties::UpdateEnqueue(
    const NodeDef* enqueue_node,
    const absl::flat_hash_map<const NodeDef*, const NodeDef*>& resource_handles,
    SymbolicShapeRefiner* shape_refiner, bool* new_shapes) {
  auto ctx = shape_refiner->GetNodeContext(enqueue_node);
  if (!ctx) {
    TF_RETURN_IF_ERROR(shape_refiner->AddNode(enqueue_node));
    ctx = CHECK_NOTNULL(shape_refiner->GetNodeContext(enqueue_node));
  }

  auto it = resource_handles.find(enqueue_node);
  if (it == resource_handles.end()) {
    // The corresponding queue was not found, there isn't much we can do.
    return Status::OK();
  }
  const NodeDef* qnode = it->second;
  auto qctx = shape_refiner->GetContext(qnode);
  if (!qctx) {
```

```
    return Status::OK();
  }
  auto* queue_handle_data = qctx->output_handle_shapes_and_types(0);

  // TODO(bsteiner): handle EnqueueMany as well.
  std::vector<ShapeAndType> shapes_and_types;
  for (int i = 1, end = ctx->input_types.size(); i < end; ++i) {
    GraphView::InputPort inp(enqueue_node, i);
    GraphView::OutputPort fanin = shape_refiner->graph().GetRegularFanin(inp);
    InferenceContext* in = shape_refiner->GetContext(fanin.node);
    ShapeHandle input = in->output(fanin.port_id);
    ctx->inference_context->SetInput(i, input);
    shapes_and_types.push_back({input, ctx->input_types[i]});
  }

  if (queue_handle_data == nullptr) {
    qctx->set_output_handle_shapes_and_types(0, shapes_and_types);
    *new_shapes = true;
  } else {
    TF_RETURN_IF_ERROR(RelaxEnqueueShapesAndMergeTypes(
        shape_refiner, qnode, *queue_handle_data, &shapes_and_types));
    *new_shapes |= !shape_refiner->EquivalentShapesAndTypes(*queue_handle_data,
                                                             shapes_and_types);
    qctx->set_output_handle_shapes_and_types(0, shapes_and_types);
  }

  return Status::OK();
}

Status GraphProperties::InferStatically(bool assume_valid_feeds,
                                        bool aggressive_shape_inference,
                                        bool include_input_tensor_values,
                                        bool include_output_tensor_values) {
  FunctionLibraryDefinition function_library(OpRegistry::Global(),
                                             item_.graph.library());
  absl::flat_hash_map<string, absl::flat_hash_set<int>> fed_ports;
  if (!assume_valid_feeds) {
    for (const auto& feed : item_.feed) {
      SafeTensorId tensor_id = ParseTensorName(feed.first);
      fed_ports[tensor_id.node()].insert(tensor_id.index());
    }
  }

  GraphView graph_view(&item_.graph);

  // List the resources and the nodes using them. Also collect the Merge nodes,
  // fed nodes, and primary inputs.
  absl::flat_hash_map<const NodeDef*,
                      std::pair<absl::flat_hash_set<const NodeDef*>,
```

```
                                  absl::flat_hash_set<const NodeDef*>>>
      resources;
  absl::flat_hash_set<const NodeDef*> merge_nodes;
  absl::flat_hash_set<const NodeDef*> fed_nodes;
  absl::flat_hash_set<const NodeDef*> primary_inputs;
  int num_loops = 0;
  for (const NodeDef& node : item_.graph.node()) {
    if (IsQueue(node)) {
      for (const GraphView::InputPort& fanout :
           graph_view.GetFanouts(node, false)) {
        if (IsEnter(*fanout.node)) {
          const NodeDef& enter = *fanout.node;
          for (const GraphView::InputPort& fanout :
               graph_view.GetFanouts(enter, false)) {
            if (IsEnqueue(*fanout.node)) {
              resources[&node].first.insert(fanout.node);
            } else if (IsDequeue(*fanout.node)) {
              resources[&node].second.insert(fanout.node);
            }
          }
        } else {
          if (IsEnqueue(*fanout.node)) {
            resources[&node].first.insert(fanout.node);
          } else if (IsDequeue(*fanout.node)) {
            resources[&node].second.insert(fanout.node);
          }
        }
      }
    }
    if (!HasRegularInputs(node)) {
      primary_inputs.insert(&node);
    } else if (IsMerge(node)) {
      merge_nodes.insert(&node);
    } else if (IsNextIteration(node)) {
      ++num_loops;
    }
    if (fed_ports.find(node.name()) != fed_ports.end()) {
      fed_nodes.insert(&node);
    }
  }

  absl::flat_hash_map<const NodeDef*, const NodeDef*> resource_handles;
  std::vector<TopologicalDependency> extra_deps;
  for (const auto& resource : resources) {
    for (const NodeDef* src : resource.second.first) {
      resource_handles[src] = resource.first;
      for (const NodeDef* dst : resource.second.second) {
        // Add control edges from enqueue to dequeue nodes to ensure they are
        // processed in their logical order.
```

```
        extra_deps.emplace_back(src, dst);
      }
    }
  }

  std::vector<const NodeDef*> topo_order;
  Status s = ComputeTopologicalOrder(item_.graph, extra_deps, &topo_order);
  if (!s.ok()) {
    if (extra_deps.empty()) {
      return s;
    } else {
      // There is a loop between queues: we'll just use the graph topological
      // order. This will make the shape inference less precise but since this
      // isn't common it's not worth to figure out where to break the loop and
      // do a proper relaxation.
      TF_RETURN_IF_ERROR(ComputeTopologicalOrder(item_.graph, &topo_order));
    }
  }

  // Heap-allocate SymbolicShapeRefiner in order to not consume a large amount
  // of stack space.
  auto refiner = absl::make_unique<SymbolicShapeRefiner>(
      graph_view, fed_ports, aggressive_shape_inference);

  TopoQueue new_shapes(topo_order);
  // Also seed the propagation of shapes in the fanout of primary inputs.
  for (const NodeDef* node : primary_inputs) {
    new_shapes.push(node);
  }
  // Also seed the propagation of shapes in the fanout of fed nodes.
  for (const NodeDef* node : fed_nodes) {
    new_shapes.push(node);
  }
  // Propagate shapes normally.
  TF_RETURN_IF_ERROR(
      PropagateShapes(refiner.get(), &new_shapes, resource_handles, num_loops));

  // Track shapes globally across the graph.
  std::unique_ptr<SymbolicShapeManager> shape_manager =
      absl::make_unique<SymbolicShapeManager>();
  bool found_error = false;
  for (const NodeDef& node : item_.graph.node()) {
    auto node_ctx = refiner->GetContext(&node);
    if (!node_ctx) {
      continue;
    }
    // Skip any information that comes from fed nodes.
    if (fed_ports.find(node.name()) != fed_ports.end()) {
      VLOG(2) << "Skipping feed node shape: " << node.name();
```

```
        continue;
      }
      for (const auto& merged_shapes : node_ctx->MergedShapes()) {
        if (!shape_manager->Merge(merged_shapes.first, merged_shapes.second)
                 .ok()) {
          found_error = true;
          break;
        }
      }
      for (const auto& merged_dims : node_ctx->MergedDims()) {
        if (!shape_manager->Merge(merged_dims.first, merged_dims.second).ok()) {
          found_error = true;
          break;
        }
      }
      if (found_error) {
        // The shapes aren't consistent, we can't infer safely: discard all the
        // information discovered so far.
        shape_manager = absl::make_unique<SymbolicShapeManager>();
        break;
      }
    }

    TF_RETURN_IF_ERROR(ValidateSymbolicShapeManager(item_.graph, refiner.get(),
                                                    shape_manager.get()));

    for (const NodeDef& node : item_.graph.node()) {
      VLOG(4) << "Filling in graph properties for node: " << node.name();
      auto ctx = refiner->GetNodeContext(&node);
      if (!ctx) {
        continue;
      }

      auto* ic = ctx->inference_context.get();

      // Fill input properties.
      {
        auto& input_properties = input_properties_[node.name()];

        // Should always be empty, node names in graph are supposed to be unique.
        CHECK_EQ(input_properties.size(), 0);

        input_properties.resize(ic->num_inputs());
        GraphView::InputPort input(&node, -1);
        for (int i = 0; i < ic->num_inputs(); ++i) {
          shape_manager->AsTensorProperties(ic->input(i), ctx->input_types[i],
                                            &input_properties[i]);
          input.port_id = i;
          GraphView::OutputPort fanin = graph_view.GetRegularFanin(input);
```

```
      if (include_input_tensor_values) {
        // Export tensor value to input_properties.value.
        if (IsConstant(*fanin.node)) {
          const TensorProto& raw_val =
              fanin.node->attr().at("value").tensor();
          *input_properties[i].mutable_value() = raw_val;
        } else if (static_cast<int>(ctx->input_tensor_protos.size()) > i &&
                   ctx->input_tensor_protos[i] != nullptr) {
          *input_properties[i].mutable_value() = *ctx->input_tensor_protos[i];
        } else if (static_cast<int>(ic->input_tensors_as_shapes().size()) >
                       i &&
                   IsShapeFullyDefinedIntegerVectorOrScalar(
                       ic, ic->input(i), ic->input_tensors_as_shapes()[i],
                       ctx->input_types[i])) {
          *input_properties[i].mutable_value() = MakeTensorProtoFromShape(
              ic, ic->input(i), ic->input_tensors_as_shapes()[i],
              ctx->input_types[i]);
        }
      }
    }
  }

  // Fill output properties.
  {
    auto& output_properties = output_properties_[node.name()];

    // Should always be empty, node names in graph are supposed to be unique.
    CHECK_EQ(output_properties.size(), 0);

    output_properties.resize(ic->num_outputs());
    for (int i = 0; i < ic->num_outputs(); ++i) {
      shape_manager->AsTensorProperties(ic->output(i), ctx->output_types[i],
                                        &output_properties[i]);
      auto converted_output_tensors_as_shapes =
          ReplaceUnknownDimFromConstWithUnknownDim(
              ic, ctx->output_tensors_as_shapes);
      if (include_output_tensor_values) {
        // Export tensor value to output_properties.value.
        if (IsConstant(node)) {
          // TODO(rmlarsen): Eliminate this copy.
          const TensorProto& raw_val = node.attr().at("value").tensor();
          *output_properties[i].mutable_value() = raw_val;
        } else if (static_cast<int>(ctx->output_tensor_protos.size()) > i &&
                   ctx->output_tensor_protos[i] != nullptr) {
          *output_properties[i].mutable_value() =
              *ctx->output_tensor_protos[i];
        } else if (static_cast<int>(
                       converted_output_tensors_as_shapes.size()) > i &&
                   IsShapeFullyDefinedIntegerVectorOrScalar(
```

```
                           ic, ic->output(i),
                           converted_output_tensors_as_shapes[i],
                           ctx->output_types[i])) {
             *output_properties[i].mutable_value() = MakeTensorProtoFromShape(
                 ic, ic->output(i), converted_output_tensors_as_shapes[i],
                 ctx->output_types[i]);
           }
         }
       }
     }

     if (aggressive_shape_inference && ctx->shape_incompatible)
       incompatible_shape_nodes_.insert(node.name());
   }

   if (aggressive_shape_inference && !incompatible_shape_nodes_.empty())
     LOG(WARNING) << incompatible_shape_nodes_.size()
                  << " nodes have incompatible output shapes.";

   // Help trace the unknown dimensions to their origins.
   VerboseLogUnknownDimensionSources(item_.graph, input_properties_,
                                     output_properties_);

   TF_RETURN_IF_ERROR(VerboseShapeInferenceLogging(item_.graph, refiner.get(),
                                                   shape_manager.get()));

   return Status::OK();
 }

 Status GraphProperties::InferDynamically(Cluster* cluster) {
   TF_RETURN_IF_ERROR(cluster->Initialize(item_));

   // Runs the model once to collect the shapes in the cost model.
   RunMetadata metadata;
   TF_RETURN_IF_ERROR(
       cluster->Run(item_.graph, item_.feed, item_.fetch, &metadata));

   return InferFromCostGraph(metadata.cost_graph());
 }

 Status GraphProperties::AnnotateOutputShapes(GraphDef* output_graph_def) const {
   *output_graph_def = item_.graph;
   for (int i = 0; i < output_graph_def->node_size(); i++) {
     auto node = output_graph_def->mutable_node(i);
     AttrValue attr_output_shape;
     auto tensor_properties = GetOutputProperties(node->name());
     for (const auto& tensor_property : tensor_properties) {
       TensorShapeProto* proto = attr_output_shape.mutable_list()->add_shape();
       *proto = tensor_property.shape();
```

```
      NormalizeShapeForOutput(proto);
    }
    (*node->mutable_attr())["_output_shapes"] = std::move(attr_output_shape);
  }
  return Status::OK();
}

Status GraphProperties::InferFromCostGraph(const CostGraphDef& cost_graph) {
  if (cost_graph.node_size() == 0) {
    LOG(WARNING) << "cost_graph is empty: nothing can be inferred!";
  }
  std::unordered_map<string, const CostGraphDef::Node*> name_to_cost;
  std::unordered_map<string, const NodeDef*> name_to_node;  // Empty
  for (auto& node : cost_graph.node()) {
    name_to_cost[node.name()] = &node;

    std::vector<OpInfo::TensorProperties> output_properties;
    for (const auto& out : node.output_info()) {
      OpInfo::TensorProperties properties;
      properties.set_dtype(out.dtype());
      *properties.mutable_shape() = out.shape();
      output_properties.push_back(properties);
    }
    output_properties_[node.name()] = output_properties;
  }

  for (const auto& node : item_.graph.node()) {
    // Skip the nodes that are not in the cost graph: these are nodes that
    // aren't run, because they aren't in the intersection of transitive fan-in
    // of a fetch node and the transitive fan-out of an input, or nodes that
    // were optimized away by the optimizer.
    auto it = name_to_cost.find(node.name());
    if (it == name_to_cost.end()) {
      continue;
    }
    std::vector<OpInfo::TensorProperties> inputs =
        FindInputFeatures(node, name_to_cost, name_to_node);

    input_properties_[node.name()] = inputs;
  }
  return Status::OK();
}

bool GraphProperties::HasInputProperties(const string& node_name) const {
  return input_properties_.find(node_name) != input_properties_.end();
}

bool GraphProperties::HasOutputProperties(const string& node_name) const {
  return output_properties_.find(node_name) != output_properties_.end();
```

```
}

const std::vector<OpInfo::TensorProperties>&
GraphProperties::GetInputProperties(const string& node_name) const {
  auto it = input_properties_.find(node_name);
  if (it != input_properties_.end()) {
    return it->second;
  }
  return missing_properties_;
}

const std::vector<OpInfo::TensorProperties>&
GraphProperties::GetOutputProperties(const string& node_name) const {
  auto it = output_properties_.find(node_name);
  if (it != output_properties_.end()) {
    return it->second;
  }
  return missing_properties_;
}

void GraphProperties::ClearInputProperties(const string& node_name) {
  input_properties_.erase(node_name);
}
void GraphProperties::ClearOutputProperties(const string& node_name) {
  output_properties_.erase(node_name);
}

}  // end namespace grappler
}  // end namespace tensorflow
```