

a1320ec1ea ▾

...

tensorflow / tensorflow / core / grappler / costs / graph_properties.cc



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

History

24 contributors



2851 lines (2605 sloc) | 107 KB

...

```

1  /* Copyright 2017 The TensorFlow Authors. All Rights Reserved.
2
3  Licensed under the Apache License, Version 2.0 (the "License");
4  you may not use this file except in compliance with the License.
5  You may obtain a copy of the License at
6
7      http://www.apache.org/licenses/LICENSE-2.0
8
9  Unless required by applicable law or agreed to in writing, software
10 distributed under the License is distributed on an "AS IS" BASIS,
11 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 See the License for the specific language governing permissions and
13 limitations under the License.
14 =====*/
15
16 #include "tensorflow/core/grappler/costs/graph_properties.h"
17
18 #include "absl/types/optional.h"
19 #include "tensorflow/core/common_runtime/function.h"
20 #include "tensorflow/core/common_runtime/graph_constructor.h"
21 #include "tensorflow/core/framework/common_shape_fns.h"
22 #include "tensorflow/core/framework/function.pb.h"
23 #include "tensorflow/core/framework/node_def_util.h"
24 #include "tensorflow/core/framework/tensor.pb.h"
25 #include "tensorflow/core/framework/tensor_shape.pb.h"
26 #include "tensorflow/core/framework/types.h"
27 #include "tensorflow/core/framework/types.pb.h"
28 #include "tensorflow/core/framework/versions.pb.h"
29 #include "tensorflow/core/graph/tensor_id.h"

```

```

30 #include "tensorflow/core/grappler/costs/utils.h"
31 #include "tensorflow/core/grappler/mutable_graph_view.h"
32 #include "tensorflow/core/grappler/op_types.h"
33 #include "tensorflow/core/grappler/optimizers/evaluation_utils.h"
34 #include "tensorflow/core/grappler/utils.h"
35 #include "tensorflow/core/grappler/utils/functions.h"
36 #include "tensorflow/core/grappler/utils/topological_sort.h"
37 #include "tensorflow/core/lib/gtl/cleanup.h"
38 #include "tensorflow/core/lib/gtl/flatset.h"
39 #include "tensorflow/core/lib/strings/str_util.h"
40
41 namespace tensorflow {
42 namespace grappler {
43
44 namespace {
45
46 using shape_inference::DimensionHandle;
47 using shape_inference::InferenceContext;
48 using shape_inference::ShapeAndType;
49 using shape_inference::ShapeHandle;
50 using TensorVector = gtl::InlinedVector<TensorValue, 4>;
51
52 // A large value for UnknownDim from Const used as a dim value in shape.
53 // Some ops treat "-1" specially, different from UnknownDim:
54 // e.g., shape input to Reshape op.
55 const int64_t kUnknownDimFromConst = INT64_MAX;
56
57 // Skip const value instantiation if the number of elements in a const tensor
58 // is greater than this threshold.
59 const int kThresholdToSkipConstTensorInstantiation = 128;
60
61 template <typename Handle>
62 struct HashHandle {
63     std::size_t operator()(const Handle& h) const { return h.Handle(); }
64 };
65 template <typename Handle>
66 struct CompareHandle {
67     bool operator()(const Handle& h1, const Handle& h2) const {
68         return h1.SameHandle(h2);
69     }
70 };
71
72 template <typename Handle>
73 struct HandleToObject {};
74 template <>
75 struct HandleToObject<ShapeHandle> {
76     typedef ShapeHandle Object;
77
78     static ShapeHandle Unknown() { return ShapeHandle(); }

```

```

79 };
80
81 template <>
82 struct HandleToObject<DimensionHandle> {
83     typedef int64_t Object;
84
85     static int64_t Unknown() { return -1; }
86 };
87
88 template <typename Handle>
89 struct Processor {};
90
91 template <>
92 struct Processor<ShapeHandle> {
93     // Extract the shape or dim denoted by the handle.
94     void ExtractValue(ShapeHandle h, ShapeHandle* result) { *result = h; }
95     // Merge the shapes or dims.
96     Status Merge(ShapeHandle h1, ShapeHandle h2, ShapeHandle* result) {
97         if (InferenceContext::RankKnown(*result)) {
98             // The result was initialized in a previous merge to a shape of known
99             // rank, make sure we preserve that information.
100             return Status::OK();
101         }
102         if (InferenceContext::RankKnown(h1)) {
103             *result = h1;
104         } else {
105             *result = h2;
106         }
107         return Status::OK();
108     }
109 };
110
111 template <>
112 struct Processor<DimensionHandle> {
113     // Assign a negative id to unknown dimensions, starting at -2 (the -1 id
114     // reserved by TensorFlow).
115     void ExtractValue(DimensionHandle d, int64_t* result) {
116         if (!InferenceContext::ValueKnown(d)) {
117             *result = -counter;
118             counter++;
119         } else {
120             int64_t val = InferenceContext::Value(d);
121             if (val >= 0) {
122                 *result = val;
123             } else {
124                 // A shape inference function generated an invalid dimension handle.
125                 // Use a symbolic dimension to encode this.
126                 *result = -counter;
127                 counter++;

```

```

128     }
129 }
130 }
131
132 // Merge the dimensions d1 and d2. Return the known shape if there is one,
133 // otherwise look for a symbolic shape. If there is no symbolic shape and no
134 // known shape, the shape is fully unknown so return -1.
135 Status Merge(DimensionHandle d1, DimensionHandle d2, int64_t* result) {
136     const int64_t dim1 = InferenceContext::Value(d1);
137     const int64_t dim2 = InferenceContext::Value(d2);
138
139     if (dim1 >= 0 && dim2 >= 0) {
140         CHECK_EQ(dim1, dim2);
141         return RefineDim(dim1, result);
142     } else if (dim1 >= 0 && dim2 < 0) {
143         return RefineDim(dim1, result);
144     } else if (dim1 < 0 && dim2 >= 0) {
145         return RefineDim(dim2, result);
146     } else if (dim1 < -1) {
147         return RefineDim(dim1, result);
148     } else if (dim2 < -1) {
149         return RefineDim(dim2, result);
150     } else {
151         CHECK_EQ(dim1, dim2);
152         CHECK_EQ(-1, dim1);
153         return RefineDim(-1, result);
154     }
155     return Status::OK();
156 }
157
158 private:
159 Status RefineDim(int64_t dim, int64_t* result) {
160     if (*result >= 0) {
161         if (!(*result == dim || dim < 0)) {
162             return errors::InvalidArgument("Inconsistent dimensions detected");
163         }
164     } else if (dim >= 0) {
165         *result = dim;
166     } else if (dim < *result) {
167         *result = dim;
168     }
169     return Status::OK();
170 }
171
172 int64_t counter = 2;
173 };
174
175 // Traditional Disjoint-Set datastructure with path compression.
176 // (https://en.wikipedia.org/wiki/Disjoint-set_data_structure)

```

```

177 template <typename Handle>
178 class DisjointSet {
179 public:
180     DisjointSet() {}
181     ~DisjointSet() {
182         for (auto rep : nodes_) {
183             delete rep.second;
184         }
185     }
186
187     Status Merge(Handle x, Handle y);
188     const typename HandleToObject<Handle>::Object GetMergedValue(Handle value);
189
190 private:
191     // All the handles that belong to the same set are part of the same tree, and
192     // ultimately represented by the root of that tree.
193     struct Rep {
194         // Parent in the tree used to encode the set.
195         Rep* parent;
196         // Rank in the tree, used to figure out how to compress the path to the root
197         // of the tree.
198         int rank;
199         // The handle.
200         typename HandleToObject<Handle>::Object value;
201     };
202
203     // Create a new set for the value if none exists, or return its representative
204     // node otherwise.
205     Rep* Find(Handle value);
206
207 private:
208     Processor<Handle> processor_;
209     absl::flat_hash_map<Handle, Rep*, HashHandle<Handle>, CompareHandle<Handle>>
210         nodes_;
211 };
212
213 template <typename Handle>
214 const typename HandleToObject<Handle>::Object
215 DisjointSet<Handle>::GetMergedValue(Handle value) {
216     Rep* rep = Find(value);
217     if (!rep) {
218         // We don't know anything about this handle.
219         return HandleToObject<Handle>::Unknown();
220     }
221     return rep->value;
222 }
223
224 template <typename Handle>
225 Status DisjointSet<Handle>::Merge(Handle x, Handle y) {

```

```

226     Rep* x_root = Find(x);
227     Rep* y_root = Find(y);
228
229     // x and y are already in the same set
230     if (x_root == y_root) {
231         return Status::OK();
232     }
233     // x and y are not in same set, so we merge them
234     // Use the occasion to strengthen what we know about the handle by merging the
235     // information about the 2 subsets.
236     if (x_root->rank < y_root->rank) {
237         TF_RETURN_IF_ERROR(processor_.Merge(y, x, &y_root->value));
238         x_root->parent = y_root;
239     } else if (x_root->rank > y_root->rank) {
240         TF_RETURN_IF_ERROR(processor_.Merge(x, y, &x_root->value));
241         y_root->parent = x_root;
242     } else {
243         TF_RETURN_IF_ERROR(processor_.Merge(x, y, &x_root->value));
244         // Arbitrarily make one root the new parent
245         y_root->parent = x_root;
246         x_root->rank = x_root->rank + 1;
247     }
248     return Status::OK();
249 }
250
251 template <typename Handle>
252 typename DisjointSet<Handle>::Rep* DisjointSet<Handle>::Find(Handle value) {
253     auto it = nodes_.find(value);
254     if (it == nodes_.end()) {
255         // This is the first time we process this handle, create an entry for it.
256         Rep* node = new Rep;
257         node->parent = node;
258         node->rank = 0;
259         processor_.ExtractValue(value, &node->value);
260         nodes_[value] = node;
261         return node;
262     }
263     // Return the representative for the set, which is the root of the tree. Apply
264     // path compression to speedup future queries.
265     Rep* node = it->second;
266     Rep* root = node->parent;
267     while (root != root->parent) {
268         root = root->parent;
269     }
270     while (node->parent != root) {
271         Rep* next = node->parent;
272         node->parent = root;
273         node = next;
274     }

```

```

275     return root;
276 }
277
278 // TODO(dyoon): Move many helper functions in this file (including those within
279 // SymbolicShapeRefiner class) to shared utils.
280 bool IsEnqueue(const NodeDef& n) {
281     return (n.op().find("Enqueue") != string::npos &&
282            n.op().find("EnqueueMany") == string::npos);
283 }
284
285 bool IsDequeue(const NodeDef& n) {
286     return (n.op().find("Dequeue") != string::npos &&
287            n.op().find("DequeueMany") == string::npos);
288 }
289
290 bool HasAnyUnknownDimensions(const TensorShapeProto& proto) {
291     if (proto.unknown_rank()) {
292         return true;
293     }
294     for (const auto& dim : proto.dim()) {
295         if (dim.size() < 0) {
296             return true;
297         }
298     }
299     return false;
300 }
301
302 // This really should be done in an external debugging tool
303 void VerboseLogUnknownDimensionSources(
304     const GraphDef& graph,
305     const absl::flat_hash_map<string, std::vector<OpInfo::TensorProperties>>&
306         input_properties_map,
307     const absl::flat_hash_map<string, std::vector<OpInfo::TensorProperties>>&
308         output_properties_map) {
309     if (!VLOG_IS_ON(2)) {
310         return;
311     }
312
313     VLOG(2) << "Nodes with known inputs, but with unknown output dimensions:";
314
315     // Find all nodes in the graph for which we
316     // do not have any unknown dimensions in their inputs, but
317     // we have some unknown dimensions in their outputs.
318     std::map<string, int> op_to_count;
319     for (const NodeDef& node : graph.node()) {
320         const auto& input_properties = input_properties_map.at(node.name());
321         const auto& output_properties = output_properties_map.at(node.name());
322
323         bool has_unknown_inputs = false;

```

```

324     for (const auto& input_prop : input_properties) {
325         if (HasAnyUnknownDimensions(input_prop.shape())) {
326             has_unknown_inputs = true;
327             break;
328         }
329     }
330
331     if (has_unknown_inputs) {
332         continue;
333     }
334
335     for (const auto& output_prop : output_properties) {
336         if (HasAnyUnknownDimensions(output_prop.shape())) {
337             string inputs = "input_shapes=[";
338             for (const auto& input_prop : input_properties) {
339                 inputs += PartialTensorShape::DebugString(input_prop.shape());
340             }
341             inputs += "];";
342
343             string outputs = "output_shapes=[";
344             for (const auto& output_prop : output_properties) {
345                 outputs += PartialTensorShape::DebugString(output_prop.shape());
346             }
347             outputs += "];";
348
349             VLOG(2) << "Node: " << node.name() << ", Op: " << node.op() << ", "
350                 << inputs << ", " << outputs;
351
352             op_to_count[node.op()]++;
353
354             // don't log again for this node
355             break;
356         }
357     }
358 }
359 VLOG(2) << "Op types with known inputs, but with unknown output dimensions "
360     << "(format: <op_type> (<count>)):";
361 for (const auto& p : op_to_count) {
362     VLOG(2) << p.first << " (" << p.second << ")";
363 }
364 }
365
366 // Helper function to convert kUnknownDimFromConst into UnknownDim.
367 std::vector<ShapeHandle> ReplaceUnknownDimFromConstWithUnknownDim(
368     InferenceContext* ic, const std::vector<ShapeHandle>& shapes) {
369     std::vector<ShapeHandle> converted_shapes(shapes.size());
370     for (int i = 0, shapes_size = shapes.size(); i < shapes_size; i++) {
371         const auto& shape = shapes[i];
372         if (!ic->RankKnown(shape)) {

```



```

373     converted_shapes[i] = shape;
374     continue;
375 }
376 bool just_copy = true;
377 std::vector<DimensionHandle> dims;
378 for (int32_t i = 0; i < ic->Rank(shape); ++i) {
379     DimensionHandle dim = ic->Dim(shape, i);
380     if (ic->ValueKnown(dim) && ic->Value(dim) == kUnknownDimFromConst) {
381         just_copy = false;
382         dims.push_back(ic->UnknownDim());
383     } else {
384         dims.push_back(dim);
385     }
386 }
387 if (just_copy) {
388     converted_shapes[i] = shape;
389     continue;
390 }
391 converted_shapes[i] = ic->MakeShape(dims);
392 }
393 return converted_shapes;
394 }
395
396 // Returned tensor's shape is like `shape`, and its values and dtype are from
397 // `tensor_as_shape` and `dtype`.
398 TensorProto MakeTensorProtoFromShape(InferenceContext* ic,
399                                     const ShapeHandle& shape,
400                                     const ShapeHandle& tensor_as_shape,
401                                     const DataType& dtype) {
402     TensorProto tensor_proto;
403     tensor_proto.set_dtype(dtype);
404     auto* shape_proto = tensor_proto.mutable_tensor_shape();
405     if (ic->Rank(shape) == 1) {
406         shape_proto->add_dim()->set_size(ic->Rank(tensor_as_shape));
407     }
408     // For a scalar tensor, tensor_shape field will be left empty; no dim.
409     for (int i = 0; i < ic->Rank(tensor_as_shape); i++) {
410         int64_t value = ic->Value(ic->Dim(tensor_as_shape, i));
411         if (dtype == DT_INT32) {
412             tensor_proto.add_int_val(value);
413         } else {
414             tensor_proto.add_int64_val(value);
415         }
416     }
417     return tensor_proto;
418 }
419
420 // Returns a Const NodeDef with tensor `tensor_proto` and dtype = `dtype`.
421 NodeDef MakeConstNodeDefFromTensorProto(InferenceContext* ic,

```

```

422         const TensorProto& tensor_proto,
423         const DataType& dtype) {
424     NodeDef const_node;
425     const_node.set_name("const_from_shape");
426     const_node.set_op("Const");
427     auto* attr = const_node.mutable_attr();
428     (*attr)["dtype"].set_type(dtype);
429     auto* tensor = (*attr)["value"].mutable_tensor();
430     *tensor = tensor_proto;
431     return const_node;
432 }
433
434 // Returns a Const NodeDef with shape = `shape`, values = `tensor_as_shape`,
435 // and dtype = `dtype`.
436 NodeDef MakeConstNodeDefFromShape(InferenceContext* ic,
437                                   const ShapeHandle& shape,
438                                   const ShapeHandle& tensor_as_shape,
439                                   const DataType& dtype) {
440     return MakeConstNodeDefFromTensorProto(
441         ic, MakeTensorProtoFromShape(ic, shape, tensor_as_shape, dtype), dtype);
442 }
443
444 bool IsNumericType(const DataType dtype) {
445     static const gtl::FlatSet<DataType>* const kRealNumberTypes =
446         CHECK_NOTNULL((new gtl::FlatSet<DataType>{
447             // Floating point.
448             DT_BFLOAT16,
449             DT_HALF,
450             DT_FLOAT,
451             DT_DOUBLE,
452             // Int / UInt.
453             DT_INT8,
454             DT_INT16,
455             DT_INT32,
456             DT_INT64,
457             DT_UINT8,
458             DT_UINT16,
459             DT_UINT32,
460             DT_UINT64,
461             // Quantized Int.
462             DT_QINT8,
463             DT_QUINT8,
464             DT_QINT16,
465             DT_QUINT16,
466             DT_QINT32,
467             // Bool.
468             DT_BOOL,
469         }));
470     return kRealNumberTypes->find(dtype) != kRealNumberTypes->end();

```

```

471 }
472
473 // Returns the number of elements in the input (const) tensor.
474 // -1 if the tensor has no shape or unknown rank.
475 uint64 NumElementsFromTensorProto(const TensorProto& tensor_proto) {
476     if (!tensor_proto.has_tensor_shape()) {
477         return -1;
478     }
479     const auto& tensor_shape_proto = tensor_proto.tensor_shape();
480     if (tensor_shape_proto.unknown_rank()) {
481         return -1;
482     }
483     int64_t num_elements = 1;
484     for (const auto& dim : tensor_shape_proto.dim()) {
485         // Note that in some cases, dim.size() can be zero (e.g., empty vector).
486         num_elements *= dim.size();
487     }
488     return num_elements;
489 }
490
491 } // namespace
492
493 // Note that tensor_as_shape input should not include kUnknownDimFromConst.
494 // This function check kUnknownDimFromConst, but will log WARNING.
495 // If checking input_tensors_as_shape_to_propgate or output_tensors_as_shape,
496 // which may include kUnknownDimFromConst, run
497 // convert it using ReplaceUnknownDimFromConstWithUnknownDim() before.
498 bool IsShapeFullyDefinedIntegerVectorOrScalar(
499     InferenceContext* ic, const ShapeHandle& shape,
500     const ShapeHandle& tensor_as_shape, const DataType& dtype) {
501     if (!ic->FullyDefined(shape) || ic->Rank(shape) > 1 ||
502         !ic->FullyDefined(tensor_as_shape) ||
503         (dtype != DT_INT32 && dtype != DT_INT64)) {
504         return false;
505     }
506     // Also check whether any dim in tensor_as_shape is kUnknownDimFromConst.
507     for (int32_t i = 0; i < ic->Rank(tensor_as_shape); ++i) {
508         DimensionHandle dim = ic->Dim(tensor_as_shape, i);
509         if (ic->Value(dim) == kUnknownDimFromConst) {
510             LOG(WARNING) << "IsShapeFullyDefinedIntegerVectorOrScalar(): "
511                 << "tensor_as_shape input includes kUnknownDimFromConst -- "
512                 << ic->DebugString(tensor_as_shape);
513             return false;
514         }
515     }
516     return true;
517 }
518
519 // Queue of nodes to process. Nodes can be enqueued in any order, but will be

```

```

520 // dequeued in (roughly) topological order. Propagating shapes following a
521 // topological ordering isn't required for correctness but helps speed things up
522 // since it avoids processing the same node multiple times as its inputs
523 // information is refined.
524 class TopoQueue {
525 public:
526     explicit TopoQueue(const std::vector<const NodeDef*>& topo_order)
527         : topo_order_(TopoOrder(topo_order)) {}
528
529     void push(const NodeDef* n) { queue_.emplace(n, topo_order_.at(n)); }
530
531     const NodeDef* pop() {
532         CHECK(!empty());
533         auto it = queue_.begin();
534         const NodeDef* n = it->first;
535         queue_.erase(it);
536         return n;
537     }
538
539     bool empty() const { return queue_.empty(); }
540     std::size_t size() const { return queue_.size(); }
541
542 private:
543     using NodeAndId = std::pair<const NodeDef*, int>;
544     // Graph nodes are created in (roughly) topological order. Therefore we can
545     // use their id to ensure they're sorted topologically.
546     struct OrderByIdAscending {
547         bool operator()(const NodeAndId& lhs, const NodeAndId& rhs) const {
548             return lhs.second < rhs.second;
549         }
550     };
551
552     const absl::flat_hash_map<const NodeDef*, int> TopoOrder(
553         const std::vector<const NodeDef*>& topo_order) const {
554         absl::flat_hash_map<const NodeDef*, int> map;
555         map.reserve(topo_order.size());
556         for (int i = 0, topo_order_size = topo_order.size(); i < topo_order_size;
557             ++i) {
558             map.emplace(topo_order[i], i);
559         }
560         return map;
561     }
562
563     const absl::flat_hash_map<const NodeDef*, int> topo_order_;
564     std::set<NodeAndId, OrderByIdAscending> queue_;
565 };
566
567
568 bool IsAllowListedOpTypeForEvaluateNode(const string& op_type) {

```

```
569 static const gtl::FlatSet<string>* const kOpTypeAllowlist =
570     CHECK_NOTNULL((new gtl::FlatSet<string>{
571         // Unary arithmetic ops
572         "Floor",
573         "Round",
574         "Sqrt",
575         "Square",
576         "Sign",
577         // Binary arithmetic ops
578         "Add",
579         "AddV2",
580         "Div",
581         "FloorDiv",
582         "FloorMod",
583         "Greater",
584         "GreaterEqual",
585         "Less",
586         "LessEqual",
587         "LogicalAnd",
588         "LogicalNot",
589         "LogicalOr",
590         "Maximum",
591         "Minimum",
592         "Mod",
593         "Mul",
594         "NotEqual",
595         "QuantizedAdd",
596         "QuantizedMul",
597         "SquareDifference",
598         "Sub",
599         "TruncateDiv",
600         "TruncateMod",
601         "RealDiv",
602         // N-ary arithmetic ops
603         "AddN",
604         // Others
605         "StridedSlice",
606         "OnesLike",
607         "ZerosLike",
608         "Concat",
609         "ConcatV2",
610         "Split",
611         "Range",
612         "Fill",
613         "Cast",
614         "Prod",
615         "Unpack",
616         "GatherV2",
617         "Pack",
```

```

618         // Used in batch_gather_nd: tensorflow/python/ops/array_ops.py
619         "ExpandDims",
620     }));
621     return kOpTpeAllowlist->find(op_type) != kOpTpeAllowlist->end();
622 }
623
624 // Negative shape size of '-1' represents unknown, while negative shape sizes
625 // less than -1 represent unknown symbolic shapes (e.g. the shape of [-5, 5, -1,
626 // -5] really means [x, 5, ?, x]). Before we can output the tensors as shapes,
627 // we need to normalize them: mark all values <-1 as "unknown" (-1).
628 static void NormalizeShapeForOutput(TensorShapeProto* shape) {
629     for (int i = 0; i < shape->dim_size(); i++) {
630         if (shape->dim(i).size() < -1) {
631             VLOG(2) << "Normalizing dimension: " << i << " from "
632                 << shape->dim(i).size() << " to -1";
633             shape->mutable_dim(i)->set_size(-1);
634         }
635     }
636 }
637
638 // Processes symbolic shapes.
639 // Each symbolic shape or dimension is represented by a handle. Unlike the TF
640 // shape refiner which creates new handles every time it processes an unknown
641 // shape/dimension, the symbolic shape refiner assigns a specific handle to each
642 // unknown shape/dimension of a given node.
643 class SymbolicShapeRefiner {
644 public:
645     explicit SymbolicShapeRefiner(
646         const GraphView& graph,
647         const absl::flat_hash_map<string, absl::flat_hash_set<int>>& fed_ports,
648         const bool aggressive_shape_inference)
649         : graph_(graph),
650         function_library_(OpRegistry::Global(), graph.graph()->library()),
651         fed_ports_(fed_ports),
652         aggressive_shape_inference_(aggressive_shape_inference) {
653         graph_def_version_ = graph.graph()->versions().producer();
654         node_to_context_.reserve(graph.graph()->node_size());
655     }
656
657     const GraphView& graph() const { return graph_; }
658
659     struct NodeContext {
660         const OpRegistrationData* op_data;
661         DataTypeVector input_types;
662         DataTypeVector output_types;
663         std::unique_ptr<InferenceContext> inference_context;
664         // Additional info for propagating tensor values and tensor shapes.
665         std::vector<const TensorProto*> input_tensor_protos;
666         std::vector<const TensorProto*> output_tensor_protos;

```

```

667 // This is the same to inference_context->input_tensors_as_shapes, except
668 // that some UnknownDims (-1) can be kUnknownDimFromConst.
669 std::vector<ShapeHandle> input_tensors_as_shapes_to_propagate;
670 std::vector<ShapeHandle> output_tensors_as_shapes;
671
672 // Output shapes incompatible between annotation and shape inference.
673 bool shape_incompatible = false;
674
675 // Similar to DebugString() in InferenceContext, but prints out
676 // kUnknownDimFromConst properly.
677 std::string StringifyShapeHandle(ShapeHandle s) {
678     auto* ic = inference_context.get();
679     if (ic->RankKnown(s)) {
680         std::vector<std::string> vals;
681         for (int i = 0; i < ic->Rank(s); i++) {
682             DimensionHandle d = ic->Dim(s, i);
683             if (ic->ValueKnown(d) && ic->Value(d) == kUnknownDimFromConst) {
684                 vals.push_back("(Const)");
685             } else {
686                 vals.push_back(ic->DebugString(d));
687             }
688         }
689         return strings::StrCat("[", absl::StrJoin(vals, ","), "]");
690     } else {
691         return "?";
692     }
693 }
694
695 std::string DebugString(const NodeDef& node) {
696     std::string output;
697     auto* ic = inference_context.get();
698     absl::StrAppend(
699         &output, node.name(), " [", node.op(), "] has ", ic->num_inputs(),
700         (ic->num_inputs() > 1 ? " inputs and " : " input and "),
701         ic->num_outputs(), (ic->num_outputs() > 1 ? " outputs" : " output"));
702     if (op_data->is_function_op) {
703         absl::StrAppend(&output, " (function op)");
704     }
705     absl::StrAppend(&output, ": \n");
706
707     for (int i = 0; i < ic->num_inputs(); i++) {
708         absl::StrAppend(&output, " input [", i, "] ", node.input(i),
709             " -- type: ", DataTypeString(input_types.at(i)),
710             ", shape: ", ic->DebugString(ic->input(i)),
711             ", tensor: ");
712         Tensor t1;
713         int input_tensor_protos_size = input_tensor_protos.size();
714         if (input_tensor_protos_size > i &&
715             input_tensor_protos.at(i) != nullptr &&

```

```

716         t1.FromProto(*input_tensor_protos.at(i))) {
717             absl::StrAppend(&output, t1.DebugString(), ", tensor_as_shape: ");
718         } else {
719             absl::StrAppend(&output, " null, tensor_as_shape: ");
720         }
721         int input_tensors_as_shapes_to_propagate_size =
722             input_tensors_as_shapes_to_propagate.size();
723         if (input_tensors_as_shapes_to_propagate_size > i) {
724             absl::StrAppend(
725                 &output,
726                 StringifyShapeHandle(input_tensors_as_shapes_to_propagate.at(i)),
727                 "\n");
728         } else {
729             absl::StrAppend(&output, " null\n");
730         }
731     }
732     for (int i = 0; i < ic->num_outputs(); i++) {
733         absl::StrAppend(&output, " output [" , i,
734             " -- type: ", DataTypeString(output_types.at(i)),
735             ", shape: ", ic->DebugString(ic->output(i)),
736             ", tensor: ");
737         Tensor t2;
738         int output_tensor_protos_size = output_tensor_protos.size();
739         if (output_tensor_protos_size > i &&
740             output_tensor_protos.at(i) != nullptr &&
741             t2.FromProto(*output_tensor_protos.at(i))) {
742             absl::StrAppend(&output, t2.DebugString(), ", tensor_as_shape: ");
743         } else {
744             absl::StrAppend(&output, " null, tensor_as_shape: ");
745         }
746         int output_tensors_as_shapes_size = output_tensors_as_shapes.size();
747         if (output_tensors_as_shapes_size > i) {
748             absl::StrAppend(&output,
749                 StringifyShapeHandle(output_tensors_as_shapes.at(i)),
750                 "\n");
751         } else {
752             absl::StrAppend(&output, " null\n");
753         }
754     }
755     return output;
756 }
757 };
758
759 NodeContext* GetNodeContext(const NodeDef* node) {
760     auto it = node_to_context_.find(node);
761     if (it == node_to_context_.end()) {
762         return nullptr;
763     }
764     return &it->second;

```



```

765 }
766
767 InferenceContext* GetContext(const NodeDef* node) {
768     auto it = node_to_context_.find(node);
769     if (it == node_to_context_.end()) {
770         return nullptr;
771     }
772     return it->second.inference_context.get();
773 }
774
775 // Forward the shapes from the function input nodes, PartitionedCalls or
776 // StatefulPartitionedCall to
777 // the argument nodes (which are Placeholder nodes), then
778 // perform shape inference on the function body.
779 //
780 // Propagate shape information of final function body node
781 // to function node `function_node`.
782 //
783 // In the event of an error, UpdateNode will simply set `function_node`'s
784 // output shape to be Unknown.
785 Status UpdateFunction(const NodeDef* function_node) {
786     NameAttrList function;
787     TF_RETURN_IF_ERROR(NameAndAttrsFromFunctionCall(*function_node, &function));
788     auto it = fun_to_grappler_function_item_.find(function.name());
789     if (it == fun_to_grappler_function_item_.end()) {
790         return errors::InvalidArgument(
791             function.name(),
792             " was not previously added to SymbolicShapeRefiner.");
793     }
794
795     const absl::optional<GrapplerFunctionItem>& maybe_grappler_function_item =
796         it->second;
797     if (!maybe_grappler_function_item.has_value()) {
798         VLOG(3) << "Skip failed to instantiate function call: function_name="
799             << function.name();
800
801         auto* ctx = GetNodeContext(function_node);
802         auto* ic = ctx->inference_context.get();
803         for (int i = 0; i < ic->num_outputs(); ++i) {
804             TF_RETURN_IF_ERROR(SetUnknownShape(function_node, i));
805         }
806
807         return Status::OK();
808     }
809
810     // Copy (not reference) so that changes we make here (e.g., replacing
811     // _Arg with Const and _RetVal with Identity) don't affect one in
812     // fun_to_grappler_function_item_.
813     GrapplerFunctionItem grappler_function_item = *maybe_grappler_function_item;

```

```

814     MutableGraphView gv(&grappler_function_item.graph);
815
816     // Forward shapes from function input nodes to argument nodes.
817     for (int i = 0, end = grappler_function_item.inputs().size(); i < end;
818         ++i) {
819         auto& fun_input = grappler_function_item.input(i);
820         NodeDef* fun_node = gv.GetNode(fun_input.node_name);
821         const TensorId input_tensor = ParseTensorName(function_node->input(i));
822
823         if (IsControlInput(input_tensor)) {
824             return errors::FailedPrecondition(
825                 "Function inputs should not contain control nodes.");
826         }
827
828         const NodeDef* input_node = graph_.GetNode(input_tensor.node());
829         if (input_node == nullptr) {
830             return errors::FailedPrecondition(input_tensor.node(),
831                 " was not found in the graph.");
832         }
833
834         InferenceContext* input_ic = GetContext(input_node);
835         if (input_ic == nullptr) {
836             return errors::FailedPrecondition(
837                 "Inference context has not been created for ", input_tensor.node());
838         }
839
840         int output_port_num = input_tensor.index();
841         AttrValue attr_output_shape;
842         TensorShapeProto proto;
843         const auto handle = input_ic->output(output_port_num);
844         input_ic->ShapeHandleToProto(handle, &proto);
845         // There may be dim.size < -1 in SymbolicShapeRefiner. Change those to -1.
846         NormalizeShapeForOutput(&proto);
847         // _Arg op's output shape uses _output_shapes attr.
848         AttrValue output_attr;
849         output_attr.mutable_list()->add_shape()->Swap(&proto);
850         (*fun_node->mutable_attr())["_output_shapes"] = output_attr;
851
852         // If dtype is DT_RESOURCE, ops that read _Arg op use _handle_dtypes and
853         // _handle_shapes attr for its shapes and dtypes.
854         if (fun_input.data_type == DT_RESOURCE) {
855             auto* shapes_and_types =
856                 input_ic->output_handle_shapes_and_types(output_port_num);
857             if (shapes_and_types != nullptr && !shapes_and_types->empty()) {
858                 AttrValue dtype_attr;
859                 AttrValue shape_attr;
860                 for (const auto& shape_and_type : *shapes_and_types) {
861                     const auto& dtype = shape_and_type.dtype;
862                     const auto& shape_handle = shape_and_type.shape;

```

```

863         dtype_attr.mutable_list()->add_type(dtype);
864         input_ic->ShapeHandleToProto(
865             shape_handle, shape_attr.mutable_list()->add_shape());
866     }
867     (*fun_node->mutable_attr())["_handle_dtypes"] = dtype_attr;
868     (*fun_node->mutable_attr())["_handle_shapes"] = shape_attr;
869 } else {
870     // Note that we do not return error here, even if the input node does
871     // not have shapes_and_types. Within the function, we cannot infer the
872     // output shape of the DT_RESOURCE input; hence, potentially unknown
873     // shapes/dims in the function output shapes.
874     VLOG(2)
875         << "A function node (" << function_node->name()
876         << ") has input with DT_RESOURCE, but the input node does not "
877         << "have shapes_and_types information: \n"
878         << "function_node: " << function_node->ShortDebugString() << "\n"
879         << "function input: " << i
880         << ", input node's output: " << output_port_num << "\n"
881         << "input node: " << input_node->ShortDebugString();
882     }
883 }
884 }
885
886 // ReplaceInputWithConst() may break GraphView's internal node mapping
887 // structure; hence, we separately build node name to NodeDef* map, for the
888 // output nodes (before GraphView becomes invalid). Note that we use string,
889 // not string_view.
890 absl::flat_hash_map<std::string, NodeDef*> output_nodes;
891 for (const auto& output_arg : grappler_function_item.outputs()) {
892     output_nodes[output_arg.node_name] = gv.GetNode(output_arg.node_name);
893 }
894
895 // Replace input nodes with Consts, if values are known. Note that
896 // we don't check exceptions here as it's done in the above loop.
897 auto* ctx = GetNodeContext(function_node);
898 auto* ic = ctx->inference_context.get();
899 for (int i = grappler_function_item.inputs().size() - 1; i >= 0; --i) {
900     const string& input = function_node->input(i);
901     const string node_name = NodeName(input);
902     const NodeDef* input_node = graph_.GetNode(node_name);
903     if (IsConstant(*input_node)) {
904         TF_CHECK_OK(
905             ReplaceInputWithConst(*input_node, i, &grappler_function_item));
906     } else if (static_cast<int>(ctx->input_tensor_protos.size()) > i &&
907         ctx->input_tensor_protos[i] != nullptr) {
908         NodeDef const_input_node = MakeConstNodeDefFromTensorProto(
909             ic, *ctx->input_tensor_protos[i], ctx->input_types[i]);
910         TF_CHECK_OK(ReplaceInputWithConst(const_input_node, i,
911             &grappler_function_item));

```

```

912     } else if (static_cast<int>(ic->input_tensors_as_shapes().size()) > i &&
913               IsShapeFullyDefinedIntegerVectorOrScalar(
914                 ic, ic->input(i), ic->input_tensors_as_shapes()[i],
915                 ctx->input_types[i])) {
916         // We have fully defined input_tensors_as_shapes for this input; use it
917         // as a const input to the function node.
918         NodeDef const_input_node = MakeConstNodeDefFromShape(
919             ic, ic->input(i), ic->input_tensors_as_shapes()[i],
920             ctx->input_types[i]);
921         TF_CHECK_OK(ReplaceInputWithConst(const_input_node, i,
922                                           &grappler_function_item));
923     }
924 }
925 // node_name to NodeDef* map in GraphView gv can be broken due to
926 // ReplaceInputWithConst(). gv should not be used after this.
927
928 // Replace output _Retval nodes with Identity nodes. _Retval is a system op
929 // without outputs and registered shape function.
930 for (const auto& output_arg : grappler_function_item.outputs()) {
931     NodeDef* output_node = output_nodes[output_arg.node_name];
932     DCHECK_EQ(output_node->op(), "_Retval");
933     output_node->set_op("Identity");
934     output_node->mutable_attr()->erase("index");
935 }
936
937 // Perform inference on function body.
938 GraphProperties gp(grappler_function_item);
939 TF_RETURN_IF_ERROR(gp.InferStatically(
940     /*assume_valid_feeds=*/true,
941     /*aggressive_shape_inference=*/aggressive_shape_inference_,
942     /*include_tensor_values=*/true));
943
944 // Add return nodes for output shapes.
945 int output = 0;
946 ctx->output_tensors_as_shapes.resize(grappler_function_item.output_size());
947 ctx->output_tensor_protos.resize(grappler_function_item.output_size(),
948                                 nullptr);
949 for (auto const& out_arg : grappler_function_item.outputs()) {
950     // It is guaranteed that output_tensors does not contain any control
951     // inputs, so port_id >= 0.
952     TensorId out_tensor = ParseTensorName(out_arg.node_name);
953
954     if (output_nodes.count(out_tensor.node()) <= 0) {
955         return errors::FailedPrecondition(
956             "Unable to find return function_node ", out_tensor.node(), " for ",
957             function_node->name());
958     }
959     const NodeDef* retnode = output_nodes[out_tensor.node()];
960

```

```

961     auto output_properties = gp.GetOutputProperties(retnode->name());
962     int output_properties_size = output_properties.size();
963     if (out_tensor.index() >= output_properties_size) {
964         return errors::InvalidArgument(
965             out_tensor.ToString(), " has invalid position ", out_tensor.index(),
966             " (output_properties.size() = ", output_properties.size(), ").");
967     }
968     auto& outprop = output_properties[out_tensor.index()];
969     TensorShapeProto shape = outprop.shape();
970     NormalizeShapeForOutput(&shape);
971     ShapeHandle out;
972     TF_RETURN_IF_ERROR(ic->MakeShapeFromShapeProto(shape, &out));
973     ic->set_output(output, out);
974     if (outprop.has_value()) {
975         // Forward tensor value to output_tensors_as_shape.
976         MaybeTensorProtoToShape(ic, outprop.value(),
977                                 &ctx->output_tensors_as_shapes[output]);
978         const_tensors_to_propagate_.push_back(outprop.value());
979         ctx->output_tensor_protos[output] = &const_tensors_to_propagate_.back();
980     }
981     output++;
982 }
983
984 return Status::OK();
985 }
986
987 // Prepares input shapes/values/handles, then runs shape inference, and
988 // finally sets output shapes/values/handles.
989 Status UpdateNode(const NodeDef* node, bool* refined) {
990     NodeContext* ctx = GetNodeContext(node);
991     if (ctx == nullptr) {
992         TF_RETURN_IF_ERROR(AddNode(node));
993         ctx = CHECK_NOTNULL(GetNodeContext(node));
994         *refined = true;
995     }
996
997     // Check if the shapes of the nodes in the fan-in of this node have changed,
998     // and if they have, update the node input shapes.
999     InferenceContext* ic = ctx->inference_context.get();
1000     ctx->input_tensors_as_shapes_to_propagate.resize(ic->num_inputs());
1001     ctx->input_tensor_protos.resize(ic->num_inputs(), nullptr);
1002
1003     for (int dst_input = 0; dst_input < ic->num_inputs(); ++dst_input) {
1004         const GraphView::InputPort port(node, dst_input);
1005         const GraphView::OutputPort fanin = graph_.GetRegularFanin(port);
1006         int src_output = fanin.port_id;
1007         const NodeDef* src = fanin.node;
1008         NodeContext* src_ctx = GetNodeContext(src);
1009         if (src_ctx == nullptr) {

```

```

1010     return errors::FailedPrecondition(
1011         "Input ", dst_input, " for '", node->name(),
1012         "' was not previously added to SymbolicShapeRefiner.");
1013 }
1014
1015 InferenceContext* src_ic = src_ctx->inference_context.get();
1016 if (src_output >= src_ic->num_outputs()) {
1017     return errors::OutOfRange("src_output = ", src_output,
1018                               ", but num_outputs is only ",
1019                               src_ic->num_outputs());
1020 }
1021
1022 // Propagate input node's NodeContext info to the current node's
1023 // NodeContext:
1024 // output_tensor_protos to input_tensor_protos and input_tensors, and
1025 // output_tensors_as_shapes to input_tensors_as_shapes.
1026 if (static_cast<int>(src_ctx->output_tensors_as_shapes.size()) >
1027     src_output) {
1028     ctx->input_tensors_as_shapes_to_propagate[dst_input] =
1029         src_ctx->output_tensors_as_shapes[src_output];
1030 }
1031
1032 if (static_cast<int>(src_ctx->output_tensor_protos.size()) > src_output) {
1033     const auto* tensor_proto = src_ctx->output_tensor_protos[src_output];
1034     if (tensor_proto != nullptr) {
1035         ctx->input_tensor_protos[dst_input] = tensor_proto;
1036     }
1037 }
1038
1039 // NOTE: we check only shape is refined; we do not (yet) check whether
1040 // tensor value is refined.
1041 if (!*refined &&
1042     !ic->input(dst_input).SameHandle(src_ic->output(src_output))) {
1043     *refined = true;
1044 }
1045 ic->SetInput(dst_input, src_ic->output(src_output));
1046
1047 if (!*refined && ic->requested_input_tensor_as_partial_shape(dst_input)) {
1048     // The input value may have changed. Since we have no way to know if
1049     // that's indeed the case, err on the safe side.
1050     *refined = true;
1051 }
1052
1053 // Also propagate handle shape and dtype of edges which are carrying
1054 // resource handles.
1055 if (ctx->input_types[dst_input] == DT_RESOURCE) {
1056     auto* outputs = src_ic->output_handle_shapes_and_types(src_output);
1057     if (!outputs) continue;
1058     auto* inputs = ic->input_handle_shapes_and_types(dst_input);

```

```

1059
1060     if (!inputs || !EquivalentShapesAndTypes(*outputs, *inputs))
1061         *refined = true;
1062     ic->set_input_handle_shapes_and_types(dst_input, *outputs);
1063 }
1064 }
1065
1066 // Make sure we schedule the fanout of resources (which have no input)
1067 // whenever the resources are updated.
1068 *refined |= ic->num_inputs() == 0;
1069
1070 if (!*refined) {
1071     // No input shape has changed, we're done.
1072     return Status::OK();
1073 }
1074
1075 // Convert all kUnknownDimFromConst to -1 for shape inference.
1076 ic->set_input_tensors_as_shapes(ReplaceUnknownDimFromConstWithUnknownDim(
1077     ic, ctx->input_tensors_as_shapes_to_propagate));
1078 // Note: UpdateFunction uses input_tensors_as_shapes and
1079 // input_tensor_protos (not the Tensor object) for input values.
1080 // so for function nodes, we don't need to convert TensorProtos
1081 // to Tensors here. If the current op is not a function op, we convert
1082 // TensorProtos to Tensors before calling InferShapes.
1083
1084 // Properly handle function nodes.
1085 if (ctx->op_data && ctx->op_data->is_function_op) {
1086     // TODO(jmdecker): Detect if the input shapes have changed for this
1087     // function. Note that when we hit a function call node, refined will be
1088     // true, as the updates to the call node will have changed, even if it's
1089     // the same function being called twice with the same input shapes.
1090     // Example: simple_function.pbtxt
1091     if (aggressive_shape_inference_) {
1092         // If output shapes are annotated, use it and skip UpdateFunction();
1093         // it can be very expensive when a function node has nested function
1094         // nodes internally. One downside with this approach is that we do not
1095         // get output values or output shapes as tensor from function node.
1096         auto s = UpdateOutputShapesUsingAnnotatedInformation(*node, ctx);
1097         if (s.ok() && AllOutputShapesKnown(ctx)) {
1098             return Status::OK();
1099         }
1100         // If shape annotation was not available, incomplete, or incompatible,
1101         // fall through to call UpdateFunction().
1102     }
1103     auto s = UpdateFunction(node);
1104     if (s.ok()) {
1105         return Status::OK();
1106     } else {
1107         VLOG(1) << "UpdateFunction failed for " << node->op()

```

```

1108         << ". Defaulting to ShapeUnknown.\n"
1109         << s.ToString();
1110     }
1111 }
1112
1113 // Construct Tensors for constant inputs used by shape functions.
1114 std::vector<Tensor> const_values(ic->num_inputs());
1115 std::vector<const Tensor*> input_tensors(ic->num_inputs(), nullptr);
1116 for (int dst_input = 0; dst_input < ic->num_inputs(); ++dst_input) {
1117     const TensorProto* tensor_proto = ctx->input_tensor_protos[dst_input];
1118     if (tensor_proto != nullptr &&
1119         // Skip if the const tensor is too large.
1120         NumElementsFromTensorProto(*tensor_proto) <=
1121             kThresholdToSkipConstTensorInstantiation &&
1122         const_values[dst_input].FromProto(*tensor_proto)) {
1123         input_tensors[dst_input] = &const_values[dst_input];
1124     }
1125 }
1126 ic->set_input_tensors(input_tensors);
1127
1128 // Update the shapes of the outputs.
1129 return InferShapes(*node, ctx);
1130 }

```

...

```

1131
1132 Status SetUnknownShape(const NodeDef* node, int output_port) {
1133     shape_inference::ShapeHandle shape =
1134         GetUnknownOutputShape(node, output_port);
1135     InferenceContext* ctx = GetContext(node);
1136     if (ctx == nullptr) {
1137         return errors::InvalidArgument("Missing context");
1138     }
1139     ctx->set_output(output_port, shape);
1140     return Status::OK();
1141 }

```

```

1142
1143 struct ShapeId {
1144     const NodeDef* node;
1145     int port_id;
1146     bool operator==(const ShapeId& other) const {
1147         return node == other.node && port_id == other.port_id;
1148     }
1149 };
1150 struct HashShapeId {
1151     std::size_t operator()(const ShapeId& shp) const {
1152         return std::hash<const NodeDef*>{}(shp.node) + shp.port_id;
1153     }
1154 };
1155
1156 struct DimId {

```



```

1157     const NodeDef* node;
1158     int port_id;
1159     int dim_index;
1160     bool operator==(const DimId& other) const {
1161         return node == other.node && port_id == other.port_id &&
1162             dim_index == other.dim_index;
1163     }
1164 };
1165
1166 struct HashDimId {
1167     std::size_t operator()(const DimId& dim) const {
1168         return std::hash<const NodeDef*>{}(dim.node) + dim.port_id +
1169             dim.dim_index;
1170     }
1171 };
1172
1173 // 'port_index' as the union of shape1 and shape2.
1174 ShapeHandle OutputAsUnion(const NodeDef* node, int port_index,
1175     ShapeHandle shape1, ShapeHandle shape2) {
1176     if (shape1.SameHandle(shape2)) {
1177         return shape1;
1178     }
1179     InferenceContext* ctx = GetContext(node);
1180     ShapeHandle relaxed = shape1;
1181     const int rank = ctx->Rank(shape1);
1182     if (!ctx->RankKnown(shape2) || ctx->Rank(shape2) != rank) {
1183         relaxed = GetUnknownOutputShape(node, port_index);
1184     } else {
1185         for (int d = 0; d < rank; ++d) {
1186             if (!ctx->Dim(shape1, d).SameHandle(ctx->Dim(shape2, d))) {
1187                 int64_t val1 = ctx->Value(ctx->Dim(shape1, d));
1188                 int64_t val2 = ctx->Value(ctx->Dim(shape2, d));
1189                 if (val1 != val2 || (val1 < 0 && val2 < 0)) {
1190                     DimensionHandle new_dim = GetUnknownOutputDim(node, port_index, d);
1191                     TF_CHECK_OK(ctx->ReplaceDim(relaxed, d, new_dim, &relaxed));
1192                 }
1193             }
1194         }
1195     }
1196     return relaxed;
1197 }
1198
1199 bool EquivalentShapes(ShapeHandle s1, ShapeHandle s2) const {
1200     if (s1.SameHandle(s2)) {
1201         return true;
1202     }
1203     if (InferenceContext::Rank(s1) != InferenceContext::Rank(s2)) {
1204         return false;
1205     }

```

```

1206     if (!InferenceContext::RankKnown(s1) && !InferenceContext::RankKnown(s2)) {
1207         return true;
1208     }
1209     const int rank = InferenceContext::Rank(s1);
1210     for (int i = 0; i < rank; ++i) {
1211         if (!InferenceContext::DimKnownRank(s1, i).SameHandle(
1212             InferenceContext::DimKnownRank(s2, i))) {
1213             int64_t val1 =
1214                 InferenceContext::Value(InferenceContext::DimKnownRank(s1, i));
1215             int64_t val2 =
1216                 InferenceContext::Value(InferenceContext::DimKnownRank(s2, i));
1217             if (val1 >= 0 && val2 >= 0 && val1 == val2) {
1218                 continue;
1219             }
1220             return false;
1221         }
1222     }
1223     return true;
1224 }
1225
1226 // Return true if the annotated shape is compatible with shape inference
1227 // result. Examples:
1228 // Inferred shape: ?, annotated shape: [10, 10] -> true;
1229 // Inferred shape: [-1, 10], annotated shape: [10, 10] -> true;
1230 // Inferred shape: [-1, 100], annotated shape: [10, 10] -> false;
1231 // Inferred shape: [-1, 10, 10], annotated shape: [10, 10] -> false.
1232 bool CompatibleShapes(ShapeHandle inferred_shape,
1233                       ShapeHandle annotated_shape) const {
1234     if (inferred_shape.SameHandle(annotated_shape)) {
1235         return true;
1236     }
1237     if (!InferenceContext::RankKnown(inferred_shape)) {
1238         return true;
1239     }
1240     if (InferenceContext::Rank(inferred_shape) !=
1241         InferenceContext::Rank(annotated_shape)) {
1242         return false;
1243     }
1244     const int rank = InferenceContext::Rank(inferred_shape);
1245     for (int i = 0; i < rank; ++i) {
1246         if (!InferenceContext::DimKnownRank(inferred_shape, i)
1247             .SameHandle(
1248                 InferenceContext::DimKnownRank(annotated_shape, i))) {
1249             int64_t val1 = InferenceContext::Value(
1250                 InferenceContext::DimKnownRank(inferred_shape, i));
1251             int64_t val2 = InferenceContext::Value(
1252                 InferenceContext::DimKnownRank(annotated_shape, i));
1253             if (val1 >= 0 && val1 != val2) {
1254                 return false;

```

```

1255     }
1256 }
1257 }
1258 return true;
1259 }
1260
1261 bool SameShapes(ShapeHandle inferred_shape,
1262                 ShapeHandle annotated_shape) const {
1263     if (inferred_shape.SameHandle(annotated_shape)) {
1264         return true;
1265     }
1266     if (InferenceContext::Rank(inferred_shape) !=
1267         InferenceContext::Rank(annotated_shape)) {
1268         return false;
1269     }
1270     const int rank = InferenceContext::Rank(inferred_shape);
1271     for (int i = 0; i < rank; ++i) {
1272         int64_t val1 = InferenceContext::Value(
1273             InferenceContext::DimKnownRank(inferred_shape, i));
1274         int64_t val2 = InferenceContext::Value(
1275             InferenceContext::DimKnownRank(annotated_shape, i));
1276         if (val1 != val2) {
1277             return false;
1278         }
1279     }
1280     return true;
1281 }
1282
1283 bool EquivalentShapesAndTypes(const std::vector<ShapeAndType>& st1,
1284                               const std::vector<ShapeAndType>& st2) const {
1285     if (st1.size() != st2.size()) {
1286         return false;
1287     }
1288     for (int i = 0, st1_size = st1.size(); i < st1_size; ++i) {
1289         const ShapeAndType& s1 = st1[i];
1290         const ShapeAndType& s2 = st2[i];
1291         if (s1.dtype != s2.dtype) {
1292             return false;
1293         }
1294         if (!EquivalentShapes(s1.shape, s2.shape)) {
1295             return false;
1296         }
1297     }
1298     return true;
1299 }
1300
1301 Status AddFunction(const NodeDef* function_node, NameAttrList function) {
1302     auto it = fun_to_grappler_function_item_.find(function.name());
1303     if (it != fun_to_grappler_function_item_.end()) {

```

```

1304     return Status::OK();
1305 }
1306
1307 const FunctionDef* function_def =
1308     CHECK_NOTNULL(function_library_.Find(function.name()));
1309 GrapplerFunctionItem grappler_function_item;
1310 Status function_instantiated =
1311     MakeGrapplerFunctionItem(*function_def, function_library_,
1312                             graph_def_version_, &grappler_function_item);
1313
1314 // If function instantiation failed we will skip it during shape inference.
1315 if (!function_instantiated.ok()) {
1316     VLOG(3) << "Failed to instantiate a function. Error: "
1317             << function_instantiated.error_message();
1318     fun_to_grappler_function_item_[function_def->signature().name()] =
1319         absl::nullopt;
1320     return Status::OK();
1321 }
1322
1323 if (static_cast<int>(grappler_function_item.inputs().size()) >
1324     function_node->input_size()) {
1325     return errors::FailedPrecondition(
1326         "Function input size should be smaller than node input size.");
1327 }
1328
1329 for (int i = grappler_function_item.inputs().size(),
1330      end = function_node->input_size();
1331      i < end; ++i) {
1332     const string& input = function_node->input(i);
1333     if (!IsControlInput(input)) {
1334         return errors::FailedPrecondition(
1335             "Found regular input (" + input,
1336             ") instead of control nodes for node " + function_node->name());
1337     }
1338 }
1339
1340 fun_to_grappler_function_item_[function_def->signature().name()] =
1341     grappler_function_item;
1342
1343 return Status::OK();
1344 }
1345
1346 Status AddNode(const NodeDef* node) {
1347     NodeContext& node_ctx = node_to_context_[node];
1348     NameAttrList function;
1349     TF_RETURN_IF_ERROR(NameAndAttrsFromFunctionCall(*node, &function));
1350
1351     // For PartitionedCall, op_data represents the function info.
1352     TF_RETURN_IF_ERROR(

```

```

1353         function_library_.Lookup(function.name(), &node_ctx.op_data));
1354
1355     if (node_ctx.op_data->is_function_op) {
1356         TF_RETURN_IF_ERROR(AddFunction(node, function));
1357     }
1358
1359     TF_RETURN_IF_ERROR(InOutTypesForNode(*node, node_ctx.op_data->op_def,
1360                                         &node_ctx.input_types,
1361                                         &node_ctx.output_types));
1362
1363     // Create the inference context for this node.
1364     const int num_inputs = node_ctx.input_types.size();
1365     std::vector<ShapeHandle> input_shapes(num_inputs);
1366     std::vector<std::unique_ptr<std::vector<ShapeAndType>>>
1367         input_handle_shapes_and_types(num_inputs);
1368     std::vector<const Tensor*> input_tensors(num_inputs, nullptr);
1369     std::vector<ShapeHandle> input_tensors_as_shapes;
1370
1371     node_ctx.inference_context.reset(new InferenceContext(
1372         graph_def_version_, *node, node_ctx.op_data->op_def, input_shapes,
1373         input_tensors, input_tensors_as_shapes,
1374         std::move(input_handle_shapes_and_types)));
1375     const Status s = node_ctx.inference_context->construction_status();
1376     if (!s.ok()) {
1377         node_ctx.inference_context.reset(nullptr);
1378     }
1379     return s;
1380 }
1381
1382 private:
1383     // Return the one ShapeHandle used to denote a fully unknown shape for a node
1384     // output.
1385     ShapeHandle GetUnknownOutputShape(const NodeDef* node, int index) {
1386         ShapeId id{node, index};
1387         auto it = unknown_shapes_.find(id);
1388         if (it != unknown_shapes_.end()) {
1389             return it->second;
1390         }
1391         InferenceContext* c = GetContext(node);
1392         ShapeHandle shp = c->UnknownShape();
1393         unknown_shapes_[id] = shp;
1394         return shp;
1395     }
1396     // Return the one ShapeHandle used to denote a fully unknown dimension for a
1397     // node output.
1398     DimensionHandle GetUnknownOutputDim(const NodeDef* node, int index,
1399                                         int dim_id) {
1400         DimId id{node, index, dim_id};
1401         auto it = unknown_dims_.find(id);

```

```

1402     if (it != unknown_dims_.end()) {
1403         return it->second;
1404     }
1405     InferenceContext* c = GetContext(node);
1406     DimensionHandle dim = c->UnknownDim();
1407     unknown_dims_[id] = dim;
1408     return dim;
1409 }
1410
1411 // Returns true if all the output tensors have known values.
1412 bool AllOutputValuesKnown(NodeContext* c) {
1413     InferenceContext* ic = c->inference_context.get();
1414     int c_output_tensors_as_shapes_size = c->output_tensors_as_shapes.size();
1415     int c_output_tensor_protos_size = c->output_tensor_protos.size();
1416     if (c_output_tensors_as_shapes_size < ic->num_outputs() &&
1417         c_output_tensor_protos_size < ic->num_outputs()) {
1418         return false;
1419     } else {
1420         // Checks if we can get output value via either output_tensor_proto or
1421         // output_tensors_as_shapes.
1422         for (int i = 0; i < ic->num_outputs(); i++) {
1423             if (c_output_tensor_protos_size > i &&
1424                 c->output_tensor_protos[i] != nullptr) {
1425                 continue;
1426             }
1427             if (c_output_tensors_as_shapes_size > i &&
1428                 ic->FullyDefined(c->output_tensors_as_shapes[i])) {
1429                 bool no_unknown_dim_from_const = true;
1430                 for (int32_t j = 0; j < ic->Rank(c->output_tensors_as_shapes[i]);
1431                     ++j) {
1432                     const auto dim = ic->Dim(c->output_tensors_as_shapes[i], j);
1433                     if (ic->ValueKnown(dim) && ic->Value(dim) == kUnknownDimFromConst) {
1434                         no_unknown_dim_from_const = false;
1435                         break;
1436                     }
1437                 }
1438                 if (no_unknown_dim_from_const) {
1439                     continue;
1440                 }
1441             }
1442             return false;
1443         }
1444     }
1445     return true;
1446 }
1447
1448 // Returns true if all the output shapes are known.
1449 bool AllOutputShapesKnown(NodeContext* c) {
1450     InferenceContext* ic = c->inference_context.get();

```

```

1451 // Checks if all the output shapes are fully defined.
1452 for (int i = 0; i < ic->num_outputs(); i++) {
1453     if (!ic->FullyDefined(ic->output(i))) {
1454         return false;
1455     }
1456 }
1457 return true;
1458 }
1459
1460 // Returns true if we can infer output tensors' values -- we know values of
1461 // all the input tensors.
1462 bool AllInputValuesKnown(NodeContext* c) {
1463     InferenceContext* ic = c->inference_context.get();
1464
1465     // Check inputs are fully defined and values are known.
1466     for (int i = 0; i < ic->num_inputs(); i++) {
1467         const Tensor* tensor = ic->input_tensor(i);
1468         // Note that we don't check c->input_tensor_protos[i], as UpdateNode()
1469         // already converted it to ic->input_tensor(i);
1470         const ShapeHandle& input_tensors_as_shape =
1471             ic->input_tensors_as_shapes()[i];
1472         // Either input_tensor is valid or input_tensors_as_shape, which has
1473         // value of input tensors as shape format, should be fully defined.
1474         if (tensor == nullptr && !ic->FullyDefined(input_tensors_as_shape)) {
1475             return false;
1476         }
1477     }
1478     return true;
1479 }
1480
1481 // Returns true if we want to update output shapes and values with running
1482 // EvaluateNode() for this op, based on op type, data type, and size.
1483 bool ShouldUpdateOutputShapesAndValues(NodeContext* c, int64_t max_size) {
1484     InferenceContext* ic = c->inference_context.get();
1485
1486     // Due to the cost of running EvaluateNode(), we limit only to white listed
1487     // op types.
1488     if (!IsAllowListedOpTypeForEvaluateNode(c->op_data->op_def.name())) {
1489         return false;
1490     }
1491
1492     // Check input dtypes are number types.
1493     for (const auto& input_type : c->input_types) {
1494         if (!IsNumericType(input_type)) {
1495             return false;
1496         }
1497     }
1498
1499     // Check output dtypes are number types.

```

```

1500     for (const auto& output_type : c->output_types) {
1501         if (!IsNumericType(output_type)) {
1502             return false;
1503         }
1504     }
1505
1506     // Check if the number of elements of each of input tensor is no larger than
1507     // the given max size.
1508     for (int i = 0; i < ic->num_inputs(); i++) {
1509         const Tensor* tensor = ic->input_tensor(i);
1510         const ShapeHandle& input_shape_handle = ic->input(i);
1511         if (tensor != nullptr) {
1512             if (tensor->NumElements() > max_size) {
1513                 return false;
1514             }
1515         } else if (ic->Value(ic->NumElements(input_shape_handle)) > max_size) {
1516             return false;
1517         }
1518     }
1519
1520     // Check if we know the shape of each output tensor, and the number of
1521     // elements is larger than the given max size.
1522     for (int i = 0; i < ic->num_outputs(); i++) {
1523         const ShapeHandle& shape_handle = ic->output(i);
1524         if (!ic->FullyDefined(shape_handle) ||
1525             ic->Value(ic->NumElements(shape_handle)) > max_size) {
1526             return false;
1527         }
1528     }
1529     return true;
1530 }
1531
1532 // Create input tensors from the NodeContext.
1533 void CreateInputTensors(NodeContext* c,
1534                         std::vector<Tensor*> input_tensor_vector,
1535                         TensorVector* inputs) {
1536     InferenceContext* ic = c->inference_context.get();
1537     for (int i = 0; i < ic->num_inputs(); i++) {
1538         if (ic->input_tensor(i)) {
1539             input_tensor_vector->at(i) = *ic->input_tensor(i);
1540             inputs->emplace_back(&input_tensor_vector->at(i));
1541             // Note that we don't check c->input_tensor_protos[i], as UpdateNode()
1542             // already converted it to ic->input_tensor(i);
1543         } else {
1544             // Create Tensor from input_tensors_as_shapes, and then emplace it
1545             // back to inputs.
1546             // Note that input_tensors_as_shapes is scalar or vector.
1547             const ShapeHandle& shape_handle = ic->input_tensors_as_shapes()[i];
1548             const DataType& data_type = c->input_types[i];

```



```

1549     int32_t rank = ic->Rank(shape_handle);
1550     if (rank < 1) {
1551         input_tensor_vector->at(i) = Tensor(data_type, {});
1552     } else {
1553         input_tensor_vector->at(i) = Tensor(data_type, {rank});
1554     }
1555     auto* tensor = &input_tensor_vector->at(i);
1556     if (data_type == DT_INT32) {
1557         auto flat = tensor->flat<int32>();
1558         for (int j = 0; j < rank; j++) {
1559             int32_t dim = ic->Value(ic->Dim(shape_handle, j));
1560             flat(j) = dim;
1561         }
1562     } else {
1563         auto flat = tensor->flat<int64_t>();
1564         for (int j = 0; j < rank; j++) {
1565             int64_t dim = ic->Value(ic->Dim(shape_handle, j));
1566             flat(j) = dim;
1567         }
1568     }
1569     inputs->emplace_back(tensor);
1570 }
1571 }
1572 }
1573
1574 // Run a node to infer output shapes and values, and add it to the
1575 // NodeContext.
1576 Status UpdateOutputShapesAndValues(const NodeDef& node, NodeContext* c) {
1577     InferenceContext* ic = c->inference_context.get();
1578
1579     // Input to EvaluateNode()
1580     TensorVector inputs;
1581     // Container for temporarily created tensor object.
1582     std::vector<Tensor> input_tensor_vector(ic->num_inputs());
1583     CreateInputTensors(c, &input_tensor_vector, &inputs);
1584
1585     // Output for EvaluateNode() and output tensor clean up object.
1586     TensorVector outputs;
1587     auto outputs_cleanup = gtl::MakeCleanup([&outputs] {
1588         for (const auto& output : outputs) {
1589             if (output.tensor) {
1590                 delete output.tensor;
1591             }
1592         }
1593     });
1594
1595     TF_RETURN_IF_ERROR(EvaluateNode(node, inputs, /*cpu_device=*/nullptr,
1596                                     &resource_mgr_, &outputs));
1597     c->output_tensors_as_shapes.resize(outputs.size());

```

```

1598     c->output_tensor_protos.resize(outputs.size(), nullptr);
1599     for (int k = 0, outputs_size = outputs.size(); k < outputs_size; k++) {
1600         const auto& t = outputs[k];
1601         // Override output shape.
1602         ShapeHandle output_shape;
1603         TF_RETURN_IF_ERROR(
1604             ic->MakeShapeFromTensorShape(t->shape(), &output_shape));
1605         if (ic->FullyDefined(ic->output(k)) &&
1606             !EquivalentShapes(ic->output(k), output_shape)) {
1607             LOG(WARNING) << "UpdateOutputShapesAndValues() -- node: " << node.name()
1608                 << ", inferred output shape "
1609                 << "doesn't match for k=" << k << ": "
1610                 << "ic->output(k): " << ic->DebugString(ic->output(k))
1611                 << ", output_shape: " << ic->DebugString(output_shape)
1612                 << " -- " << node.DebugString();
1613         }
1614         ic->set_output(k, output_shape);
1615         // Set output_tensors_as_shape.
1616         MaybeTensorValueToShape(ic, *t.tensor, &c->output_tensors_as_shapes[k]);
1617
1618         // Set output_tensor_protos.
1619         TensorProto tensor_proto;
1620         t->AsProtoTensorContent(&tensor_proto);
1621         const_tensors_to_propagate_.push_back(tensor_proto);
1622         c->output_tensor_protos[k] = &const_tensors_to_propagate_.back();
1623     }
1624     return Status::OK();
1625 }
1626
1627 // Update output shapes with annotated information.
1628 // Currently only handle nodes with static shapes, i.e. shapes do not change
1629 // during execution.
1630 // TODO(andiryxu): Use annotated shapes in Enter/Merge etc as well.
1631 Status UpdateOutputShapesUsingAnnotatedInformation(const NodeDef& node,
1632                                                    NodeContext* c) const {
1633     const auto& attr = node.attr();
1634     if (attr.count(kOutputSame) == 0 || !attr.at(kOutputSame).b() ||
1635         attr.count(kOutputShapes) == 0)
1636         return Status::OK();
1637
1638     InferenceContext* ic = c->inference_context.get();
1639     int output_size = attr.at(kOutputShapes).list().shape_size();
1640
1641     for (int i = 0; i < ic->num_outputs(); i++) {
1642         // Annotated Switch node has only one output. Propagate the shape to all
1643         // the outputs.
1644         int shape_index = IsSwitch(node) ? 0 : i;
1645         if (shape_index >= output_size) {
1646             LOG(WARNING)

```

```

1647         << "UpdateOutputShapesUsingAnnotatedInformation() -- node: "
1648         << node.name() << ", inferred output shape size "
1649         << ic->num_outputs() << ", annotated output shape size "
1650         << output_size;
1651     break;
1652 }
1653
1654     const TensorShapeProto& shape =
1655         attr.at(kOutputShapes).list().shape(shape_index);
1656     if (shape.dim().empty()) continue;
1657
1658     ShapeHandle output_shape;
1659     TF_RETURN_IF_ERROR(ic->MakeShapeFromShapeProto(shape, &output_shape));
1660
1661     // Check if annotated shapes are incompatible with inferred shapes.
1662     if ((ic->FullyDefined(ic->output(i)) &&
1663         !SameShapes(ic->output(i), output_shape)) ||
1664         (!ic->FullyDefined(ic->output(i)) &&
1665         !CompatibleShapes(ic->output(i), output_shape))) {
1666         LOG(WARNING)
1667             << "UpdateOutputShapesUsingAnnotatedInformation() -- node: "
1668             << node.name() << ", inferred output shape "
1669             << "doesn't match for i=" << i << ": "
1670             << "ic->output(k): " << ic->DebugString(ic->output(i))
1671             << ", annotated output shape: " << ic->DebugString(output_shape)
1672             << " -- " << node.DebugString();
1673         c->shape_incompatible = true;
1674     }
1675
1676     // Only use annotated shapes if the inference shape is unknown and
1677     // compatible with annotated shapes.
1678     if (!ic->FullyDefined(ic->output(i)) &&
1679         CompatibleShapes(ic->output(i), output_shape)) {
1680         VLOG(3) << "UpdateOutputShapesUsingAnnotatedInformation() -- node: "
1681             << node.name() << ", inferred output shape " << i << ": "
1682             << "ic->output(i): " << ic->DebugString(ic->output(i))
1683             << ", annotated output shape: " << ic->DebugString(output_shape)
1684             << " -- " << node.ShortDebugString();
1685         ic->set_output(i, output_shape);
1686     }
1687 }
1688
1689     return Status::OK();
1690 }
1691
1692 Status MaybeUpdateNodeContextOutput(const NodeDef& node, const bool is_fed,
1693                                     NodeContext* c) {
1694     // Propagate tensors and shape tensors unless the node is fed.
1695     // TODO(bsteiner) We should still propagate the shapes to the ports that

```

```

1696 // aren't fed in the case of a ShapeN node.
1697
1698 // Note that when propagating tensors_as_shapes, we use
1699 // c->input_tensors_as_shapes_to_propagate instead of
1700 // ic->input_tensors_as_shapes. The former uses kUnknownDimFromConst if
1701 // UnknownDim is from Const tensor, and it is propagated through shape
1702 // inference. Before calling shape functions, we convert it to UnknownDim,
1703 // but instantiate a new UnknownDim to prevent incorrect symbolic shape
1704 // inference through UnknownDim from Const.
1705 InferenceContext* ic = c->inference_context.get();
1706 if (!is_fed) {
1707     if (IsConstant(node)) {
1708         const TensorProto& tensor_proto = node.attr().at("value").tensor();
1709         c->output_tensor_protos.resize(1);
1710         c->output_tensor_protos[0] = &tensor_proto;
1711         c->output_tensors_as_shapes.resize(1);
1712         MaybeTensorProtoToShape(ic, tensor_proto,
1713                                 &c->output_tensors_as_shapes[0]);
1714     } else if (IsRank(node)) {
1715         if (ic->RankKnown(ic->input(0))) {
1716             // Propagate rank value.
1717             int32_t rank = ic->Rank(ic->input(0));
1718             const_tensors_to_propagate_.push_back(
1719                 MakeIntegerScalarTensorProto(DT_INT32, rank));
1720             c->output_tensor_protos.resize(1);
1721             c->output_tensor_protos[0] = &const_tensors_to_propagate_.back();
1722         }
1723     } else if (IsSize(node)) {
1724         DimensionHandle size = ic->NumElements(ic->input(0));
1725         if (ic->ValueKnown(size)) {
1726             // Propagate size value.
1727             int64_t sz = ic->Value(size);
1728             bool valid = false;
1729             if (node.attr().at("out_type").type() == DT_INT32) {
1730                 if (sz < std::numeric_limits<int32_t>::max()) {
1731                     const_tensors_to_propagate_.push_back(
1732                         MakeIntegerScalarTensorProto(DT_INT32, sz));
1733                     valid = true;
1734                 }
1735             } else {
1736                 const_tensors_to_propagate_.push_back(
1737                     MakeIntegerScalarTensorProto(DT_INT64, sz));
1738                 valid = true;
1739             }
1740             if (valid) {
1741                 c->output_tensor_protos.resize(1);
1742                 c->output_tensor_protos[0] = &const_tensors_to_propagate_.back();
1743             }
1744         }
1745     }
1746 }

```

```

1745     } else if (IsShape(node)) {
1746         c->output_tensors_as_shapes.resize(1);
1747         c->output_tensors_as_shapes[0] = c->inference_context->input(0);
1748     } else if (IsShapeN(node)) {
1749         c->output_tensors_as_shapes.resize(c->inference_context->num_inputs());
1750         for (int i = 0; i < c->inference_context->num_inputs(); ++i) {
1751             c->output_tensors_as_shapes[i] = c->inference_context->input(i);
1752         }
1753     } else if (node.op() == "ConcatV2") {
1754         bool valid = true;
1755         ShapeHandle result;
1756         for (int i = 0; i < ic->num_inputs() - 1; ++i) {
1757             ShapeHandle input = c->input_tensors_as_shapes_to_propagate[i];
1758             if (!ic->RankKnown(input)) {
1759                 valid = false;
1760                 break;
1761             } else if (i == 0) {
1762                 result = input;
1763             } else {
1764                 TF_RETURN_IF_ERROR(ic->Concatenate(result, input, &result));
1765             }
1766         }
1767         if (valid) {
1768             c->output_tensors_as_shapes.resize(1);
1769             c->output_tensors_as_shapes[0] = result;
1770         }
1771     } else if (IsPack(node)) {
1772         // A Pack node concatenating scalars is often used to generate a shape.
1773         std::vector<DimensionHandle> dims;
1774         bool valid = true;
1775         for (int i = 0; i < ic->num_inputs(); ++i) {
1776             const Tensor* t = ic->input_tensor(i);
1777             if (t) {
1778                 if (t->dims() != 0 ||
1779                     (t->dtype() != DT_INT32 && t->dtype() != DT_INT64)) {
1780                     valid = false;
1781                     break;
1782                 }
1783                 int64_t size = t->dtype() == DT_INT32 ? t->scalar<int32>()()
1784                     : t->scalar<int64_t>()();
1785                 dims.push_back(size < 0 ? ic->MakeDim(kUnknownDimFromConst)
1786                     : ic->MakeDim(size));
1787             } else {
1788                 // Don't have tensor value, but use input_tensors_as_shapes, if
1789                 // possible.
1790                 const ShapeHandle& shape_handle =
1791                     c->input_tensors_as_shapes_to_propagate[i];
1792                 if (ic->RankKnown(shape_handle) && ic->Rank(shape_handle) >= 1 &&
1793                     ic->ValueKnown(ic->Dim(shape_handle, 0))) {

```

```

1794         dims.push_back(ic->Dim(shape_handle, 0));
1795     } else {
1796         // This is not from Const, but as it shouldn't be used as symbolic
1797         // unknown dim for different ops, we use kUnknownDimFromConst.
1798         dims.push_back(ic->MakeDim(kUnknownDimFromConst));
1799     }
1800 }
1801 }
1802 if (valid) {
1803     c->output_tensors_as_shapes.resize(1);
1804     c->output_tensors_as_shapes[0] = ic->MakeShape(dims);
1805 }
1806 } else if (IsIdentity(node) || IsIdentityNSingleInput(node)) {
1807     c->output_tensors_as_shapes.resize(1);
1808     c->output_tensors_as_shapes[0] =
1809         c->input_tensors_as_shapes_to_propagate[0];
1810     if (c->input_tensor_protos[0] != nullptr) {
1811         c->output_tensor_protos.resize(1);
1812         c->output_tensor_protos[0] = c->input_tensor_protos[0];
1813     }
1814 } else if (IsSlice(node)) {
1815     ShapeHandle input = c->input_tensors_as_shapes_to_propagate[0];
1816     bool valid = ic->RankKnown(input);
1817     const Tensor* slice_offset = ic->input_tensor(1);
1818     valid &= slice_offset != nullptr && slice_offset->NumElements() == 1;
1819     const Tensor* slice_size = ic->input_tensor(2);
1820     valid &= slice_size != nullptr && slice_size->NumElements() == 1;
1821     if (valid) {
1822         int64_t start = slice_offset->dtype() == DT_INT32
1823             ? slice_offset->flat<int32>()(0)
1824             : slice_offset->flat<int64_t>()(0);
1825         int64_t size = (slice_size->dtype() == DT_INT32
1826             ? slice_size->flat<int32>()(0)
1827             : slice_size->flat<int64_t>()(0));
1828         ShapeHandle result;
1829         if (size == -1) {
1830             TF_RETURN_IF_ERROR(ic->Subshape(input, start, &result));
1831         } else {
1832             int64_t end = start + size;
1833             TF_RETURN_IF_ERROR(ic->Subshape(input, start, end, &result));
1834         }
1835         c->output_tensors_as_shapes.resize(1);
1836         c->output_tensors_as_shapes[0] = result;
1837     }
1838 } else if (IsStridedSlice(node)) {
1839     ShapeHandle input = c->input_tensors_as_shapes_to_propagate[0];
1840     bool valid = ic->RankKnown(input);
1841     const Tensor* slice_begin = ic->input_tensor(1);
1842     valid &= slice_begin != nullptr && slice_begin->NumElements() == 1;

```

```

1843     const Tensor* slice_end = ic->input_tensor(2);
1844     valid &= slice_end != nullptr && slice_end->NumElements() == 1;
1845     const Tensor* slice_stride = ic->input_tensor(3);
1846     valid &= slice_stride != nullptr && slice_stride->NumElements() == 1;
1847
1848     if (node.attr().count("ellipsis_mask") > 0 &&
1849         node.attr().at("ellipsis_mask").i() != 0) {
1850         valid = false;
1851     }
1852     if (node.attr().count("new_axis_mask") > 0 &&
1853         node.attr().at("new_axis_mask").i() != 0) {
1854         valid = false;
1855     }
1856     if (node.attr().count("shrink_axis_mask") > 0 &&
1857         node.attr().at("shrink_axis_mask").i() != 0) {
1858         valid = false;
1859     }
1860     int begin_mask = 0;
1861     if (node.attr().count("begin_mask") > 0) {
1862         begin_mask = node.attr().at("begin_mask").i();
1863     }
1864     int end_mask = 0;
1865     if (node.attr().count("end_mask") > 0) {
1866         end_mask = node.attr().at("end_mask").i();
1867     }
1868     if (begin_mask < 0 || begin_mask > 1 || end_mask < 0 || end_mask > 1) {
1869         valid = false;
1870     }
1871     if (valid) {
1872         int64_t begin = 0;
1873         if (begin_mask == 0) {
1874             begin = slice_begin->dtype() == DT_INT32
1875                 ? slice_begin->flat<int32>()(0)
1876                 : slice_begin->flat<int64_t>()(0);
1877         }
1878         int64_t end = std::numeric_limits<int64_t>::max();
1879         if (end_mask == 0) {
1880             end = (slice_end->dtype() == DT_INT32
1881                 ? slice_end->flat<int32>()(0)
1882                 : slice_end->flat<int64_t>()(0));
1883         }
1884         int64_t stride = slice_stride->dtype() == DT_INT32
1885             ? slice_stride->flat<int32>()(0)
1886             : slice_stride->flat<int64_t>()(0);
1887         ShapeHandle result;
1888         TF_RETURN_IF_ERROR(ic->Subshape(input, begin, end, stride, &result));
1889         c->output_tensors_as_shapes.resize(1);
1890         c->output_tensors_as_shapes[0] = result;
1891     }

```

```

1892     }
1893 }
1894
1895 if (aggressive_shape_inference_) {
1896     // Update output shapes with annotated information. This is optional.
1897     UpdateOutputShapesUsingAnnotatedInformation(node, c).IgnoreError();
1898
1899     // Update output tensor values using EvaluateNode() if we can.
1900     // Due to the cost of EvaluateNode(), we run it only for certain op types
1901     // (white listed) and small integer tensors.
1902
1903     const int max_element_size = 17; // Max up to 4x4 matrix or similar.
1904     if (AllOutputValuesKnown(c) || !AllInputValuesKnown(c) ||
1905         !ShouldUpdateOutputShapesAndValues(c, max_element_size)) {
1906         return Status::OK();
1907     }
1908     UpdateOutputShapesAndValues(node, c).IgnoreError(); // This is optional.
1909 }
1910 return Status::OK();
1911 }
1912
1913 Status InferShapes(const NodeDef& node, NodeContext* c) {
1914     // Infer the shapes of output tensors.
1915     if (!c->op_data || c->op_data->shape_inference_fn == nullptr ||
1916         !c->inference_context->Run(c->op_data->shape_inference_fn).ok()) {
1917         // Annotate outputs with unknown shapes. Update output shapes with
1918         // annotated information later on if available.
1919         // Note that shape inference function may return an error, but we ignore
1920         // it, and use UnknownShape in that case.
1921         TF_RETURN_IF_ERROR(
1922             c->inference_context->Run(shape_inference::UnknownShape));
1923     }
1924     Status status = Status::OK();
1925     auto it = fed_ports_.find(node.name());
1926     const bool is_fed = it != fed_ports_.end();
1927     if (is_fed) {
1928         // It is possible to feed node output ports with tensors of any shape: as
1929         // a result, the shape of a fed port is completely unknown.
1930         for (const int output_port : it->second) {
1931             status.Update(SetUnknownShape(&node, output_port));
1932         }
1933     }
1934
1935     // Update NodeContext output fields after shape inference function runs.
1936     status.Update(MaybeUpdateNodeContextOutput(node, is_fed, c));
1937
1938     return status;
1939 }
1940

```



```

1941 private:
1942 bool IsIntegerVector(const Tensor& tensor) {
1943     if (tensor.dims() == 1 &&
1944         (tensor.dtype() == DT_INT32 || tensor.dtype() == DT_INT64)) {
1945         return true;
1946     }
1947     return false;
1948 }
1949
1950 bool IsIntegerScalar(const Tensor& tensor) {
1951     if (tensor.dims() == 0 &&
1952         (tensor.dtype() == DT_INT32 || tensor.dtype() == DT_INT64) &&
1953         tensor.NumElements() == 1) {
1954         return true;
1955     }
1956     return false;
1957 }
1958
1959 TensorProto MakeIntegerScalarTensorProto(const DataType dtype,
1960                                         const int64_t val) {
1961     TensorProto tensor_proto;
1962     tensor_proto.set_dtype(dtype);
1963     // Scalar TensorProto has an empty tensor_shape; no dim, no dim.size.
1964     tensor_proto.mutable_tensor_shape();
1965     if (dtype == DT_INT32) {
1966         tensor_proto.add_int_val(val);
1967     } else if (dtype == DT_INT64) {
1968         tensor_proto.add_int64_val(val);
1969     }
1970     return tensor_proto;
1971 }
1972
1973 bool MaybeTensorProtoToShape(InferenceContext* ic,
1974                             const TensorProto& tensor_proto,
1975                             ShapeHandle* tensors_as_shapes) {
1976     // Skip if dtype is not integer.
1977     if (tensor_proto.dtype() != DT_INT32 && tensor_proto.dtype() != DT_INT64) {
1978         return false;
1979     }
1980     // Skip if the const tensor is too large.
1981     if (NumElementsFromTensorProto(tensor_proto) >
1982         kThresholdToSkipConstTensorInstantiation) {
1983         return false;
1984     }
1985     // Skip if shape is neither scalar nor vector.
1986     if (tensor_proto.tensor_shape().unknown_rank() ||
1987         tensor_proto.tensor_shape().dim_size() > 1) {
1988         return false;
1989     }

```

```

1990     Tensor tensor;
1991     if (!tensor.FromProto(tensor_proto)) {
1992         return false;
1993     }
1994     return MaybeTensorValueToShape(ic, tensor, tensors_as_shapes);
1995 }
1996
1997 bool MaybeTensorValueToShape(InferenceContext* ic, const Tensor& tensor,
1998                             ShapeHandle* tensors_as_shapes) {
1999     // Integer tensors of rank one can also be interpreted as a shape
2000     // provided all their values are >= -1.
2001
2002     if (IsIntegerVector(tensor)) {
2003         bool has_values_smaller_than_minus_1 = false;
2004         std::vector<DimensionHandle> dims;
2005         for (int i = 0; i < tensor.NumElements(); i++) {
2006             int64_t value = tensor.dtype() == DT_INT32 ? tensor.flat<int32>()(i)
2007                                                         : tensor.flat<int64_t>()(i);
2008             has_values_smaller_than_minus_1 |= (value < -1);
2009             // Mark this as UnknownDim from Const.
2010             dims.push_back(value < 0 ? ic->MakeDim(kUnknownDimFromConst)
2011                                     : ic->MakeDim(value));
2012         }
2013
2014         if (!has_values_smaller_than_minus_1) {
2015             *tensors_as_shapes = ic->MakeShape(dims);
2016             return true;
2017         }
2018     } else if (IsIntegerScalar(tensor)) {
2019         // Scalar constant.
2020         int64_t value = tensor.dtype() == DT_INT32 ? tensor.flat<int32>()(0)
2021                                                         : tensor.flat<int64_t>()(0);
2022         if (value == -1) {
2023             // Scalar value -1 represents an unknown shape. If we would try to
2024             // MakeShape(MakeDim) with it, we would get vector of unknown size.
2025             *tensors_as_shapes = ic->UnknownShape();
2026             return true;
2027         } else if (value >= 0) {
2028             // Ideally, values can be < -1, but MakeDim() fails with a value < -1.
2029             // It's a limitation as we use ShapeHandle as a means to pass values.
2030             *tensors_as_shapes = ic->MakeShape({ic->MakeDim(value)});
2031             return true;
2032         }
2033     }
2034     return false;
2035 }
2036
2037 const GraphView& graph_;
2038 int graph_def_version_;

```

```

2039     absl::flat_hash_map<const NodeDef*, NodeContext> node_to_context_;
2040     absl::flat_hash_map<ShapeId, ShapeHandle, HashShapeId> unknown_shapes_;
2041     absl::flat_hash_map<DimId, DimensionHandle, HashDimId> unknown_dims_;
2042     // Store function instantiations only for valid function. If function
2043     // instantiation failed it will have an `absl::nullopt`.
2044     absl::flat_hash_map<string, absl::optional<GrapplerFunctionItem>>
2045         fun_to_grappler_function_item_;
2046     FunctionLibraryDefinition function_library_;
2047     const absl::flat_hash_map<string, absl::flat_hash_set<int>>& fed_ports_;
2048     // Store TensorProtos for tensor value propagation. Note that we use deque,
2049     // not vector, as we use pointers to the TensorProtos in this container.
2050     // Vector may resize and copy the objects into a new buffer, then the existing
2051     // pointers become dangling pointers.
2052     std::deque<TensorProto> const_tensors_to_propagate_;
2053
2054     // For more aggressive shape and value inference.
2055     bool aggressive_shape_inference_;
2056     ResourceMgr resource_mgr_;
2057 };
2058
2059 // Keep track of shapes and dimensions in a graph.
2060 // In particular, use disjoint sets to track equivalence between shapes and
2061 // dims, and consolidate the information globally.
2062 class SymbolicShapeManager {
2063 public:
2064     SymbolicShapeManager() {}
2065
2066     Status Merge(ShapeHandle s1, ShapeHandle s2) {
2067         if (!s1.IsSet() || !s2.IsSet()) {
2068             return Status::OK();
2069         }
2070         TF_RETURN_IF_ERROR(shapes_.Merge(s1, s2));
2071         if (InferenceContext::Rank(s1) > 0 && InferenceContext::Rank(s2) > 0) {
2072             CHECK_EQ(InferenceContext::Rank(s1), InferenceContext::Rank(s2));
2073             for (int i = 0; i < InferenceContext::Rank(s1); ++i) {
2074                 TF_RETURN_IF_ERROR(dims_.Merge(InferenceContext::DimKnownRank(s1, i),
2075                                                 InferenceContext::DimKnownRank(s2, i)));
2076             }
2077         }
2078         return Status::OK();
2079     }
2080     Status Merge(DimensionHandle d1, DimensionHandle d2) {
2081         if (!d1.IsSet() || !d2.IsSet()) {
2082             return Status::OK();
2083         }
2084         return dims_.Merge(d1, d2);
2085     }
2086
2087     void AsTensorProperties(const ShapeHandle& shape, const DataType& type,

```

```

2088         OpInfo::TensorProperties* properties) {
2089     properties->set_dtype(type);
2090     ShapeHandle actual_shape = shapes_.GetMergedValue(shape);
2091     if (!InferenceContext::RankKnown(actual_shape)) {
2092         properties->mutable_shape()->set_unknown_rank(true);
2093     } else {
2094         for (int j = 0; j < InferenceContext::Rank(actual_shape); ++j) {
2095             shape_inference::DimensionHandle dim =
2096                 InferenceContext::DimKnownRank(actual_shape, j);
2097             int64_t d = dims_.GetMergedValue(dim);
2098             properties->mutable_shape()->add_dim()->set_size(d);
2099         }
2100     }
2101 }
2102
2103 // Returns merged shape with merged dimensions.
2104 ShapeHandle GetMergedShape(InferenceContext* ic, ShapeHandle s) {
2105     const auto& actual_shape = shapes_.GetMergedValue(s);
2106     if (!InferenceContext::RankKnown(actual_shape)) {
2107         return ic->UnknownShape();
2108     } else {
2109         std::vector<DimensionHandle> dims;
2110         for (int j = 0; j < InferenceContext::Rank(actual_shape); ++j) {
2111             shape_inference::DimensionHandle dim =
2112                 InferenceContext::DimKnownRank(actual_shape, j);
2113             int64_t d = dims_.GetMergedValue(dim);
2114             // Symbolic shape manager may made some dims < -1, which causes errors
2115             // in creating Dimension.
2116             if (d < -1) {
2117                 d = -1;
2118             }
2119             dims.push_back(ic->MakeDim(d));
2120         }
2121         return ic->MakeShape(dims);
2122     }
2123 }
2124
2125 private:
2126     DisjointSet<shape_inference::ShapeHandle> shapes_;
2127     DisjointSet<shape_inference::DimensionHandle> dims_;
2128 };
2129
2130 // Checks whether there is any conflict in merged shapes and dims in
2131 // SymbolicShapeManager.
2132 Status ValidateSymbolicShapeManager(const GraphDef& graph_def,
2133                                     SymbolicShapeRefiner* refiner,
2134                                     SymbolicShapeManager* shape_manager) {
2135     if (!VLOG_IS_ON(1)) {
2136         return Status::OK();

```

```

2137     }
2138
2139     VLOG(1) << "Checking any conflicts in shapes and dimensions ...";
2140     int64_t num_incompatible_shapes = 0;
2141     for (const NodeDef& node : graph_def.node()) {
2142         auto ctx = refiner->GetNodeContext(&node);
2143         if (!ctx) {
2144             continue;
2145         }
2146         auto* ic = ctx->inference_context.get();
2147         for (int i = 0; i < ic->num_inputs(); ++i) {
2148             const auto& shape = ic->input(i);
2149             const auto& merged_shape = shape_manager->GetMergedShape(ic, shape);
2150             if (!refiner->CompatibleShapes(shape, merged_shape)) {
2151                 num_incompatible_shapes++;
2152                 VLOG(1) << "**** Incompatible shape from SymbolicShapeManager "
2153                     << "for node " << node.name() << " input (" << i << ") "
2154                     << ic->DebugString(shape)
2155                     << " vs. merged: " << ic->DebugString(merged_shape);
2156             }
2157         }
2158         for (int i = 0; i < ic->num_outputs(); ++i) {
2159             const auto& shape = ic->output(i);
2160             const auto& merged_shape = shape_manager->GetMergedShape(ic, shape);
2161             if (!refiner->CompatibleShapes(shape, merged_shape)) {
2162                 num_incompatible_shapes++;
2163                 VLOG(1) << "**** Incompatible shape from SymbolicShapeManager "
2164                     << "for node " << node.name() << " output (" << i << ") "
2165                     << ic->DebugString(shape)
2166                     << " vs. merged: " << ic->DebugString(merged_shape);
2167             }
2168         }
2169     }
2170     if (num_incompatible_shapes > 0) {
2171         VLOG(1) << "**** WARNING: " << num_incompatible_shapes
2172             << " incompatible shapes from SymbolicShapeManager.";
2173     } else {
2174         VLOG(1) << "**** No incompatible shape found from SymbolicShapeManager.";
2175     }
2176
2177     return Status::OK();
2178 }
2179
2180 // Log shape inference and its merged shapes.
2181 Status VerboseShapeInferenceLogging(const GraphDef& graph_def,
2182                                     SymbolicShapeRefiner* refiner,
2183                                     SymbolicShapeManager* shape_manager) {
2184     // As logging all the nodes would generate too many lines, we by default
2185     // skip this detailed logging. Users may add nodes of interest to

```

```

2186 // node_names_for_logging to enable detailed logging.
2187 absl::flat_hash_set<std::string> node_names_for_logging = {};
2188 if (!VLOG_IS_ON(3) || node_names_for_logging.empty()) {
2189     return Status::OK();
2190 }
2191
2192 auto should_log = [&node_names_for_logging](std::string node_name) {
2193     return node_names_for_logging.find(node_name) !=
2194         node_names_for_logging.end();
2195 };
2196
2197 for (const NodeDef& node : graph_def.node()) {
2198     if (!should_log(node.name())) {
2199         continue;
2200     }
2201     auto ctx = refiner->GetNodeContext(&node);
2202     if (!ctx) {
2203         continue;
2204     }
2205     auto* ic = ctx->inference_context.get();
2206     VLOG(3) << "Shape inference for node : " << node.name();
2207     VLOG(3) << ctx->DebugString(node);
2208     std::string merged_shapes = "Merged shapes from SymbolicShapManager:\n";
2209     for (int i = 0; i < ic->num_inputs(); ++i) {
2210         absl::StrAppend(
2211             &merged_shapes, " input[" , i, "]" -- ",
2212             ic->DebugString(shape_manager->GetMergedShape(ic, ic->input(i))),
2213             "\n");
2214     }
2215     for (int i = 0; i < ic->num_outputs(); ++i) {
2216         absl::StrAppend(
2217             &merged_shapes, " output[" , i, "]" -- ",
2218             ic->DebugString(shape_manager->GetMergedShape(ic, ic->output(i))),
2219             "\n");
2220     }
2221     VLOG(3) << merged_shapes;
2222     VLOG(3) << "-----";
2223     VLOG(3) << "";
2224 }
2225
2226 return Status::OK();
2227 }
2228
2229 Status GraphProperties::RelaxEnqueueShapesAndMergeTypes(
2230     SymbolicShapeRefiner* shape_refiner, const NodeDef* qnode,
2231     const std::vector<ShapeAndType>& shapes_and_types,
2232     std::vector<ShapeAndType>* queue_shapes_and_types) {
2233     if (shapes_and_types.size() != queue_shapes_and_types->size()) {
2234         return errors::InvalidArgument(

```

```

2235     "Enqueue nodes mixed number of tensors: ", shapes_and_types.size(),
2236     " vs ", queue_shapes_and_types->size());
2237 }
2238 for (size_t i = 0; i < shapes_and_types.size(); ++i) {
2239     const ShapeAndType& a = shapes_and_types[i];
2240     ShapeAndType& b = (*queue_shapes_and_types)[i];
2241     if (a.dtype != b.dtype) {
2242         return errors::InvalidArgument("Enqueue nodes mixed dtypes for tensor ",
2243             i, ": ", DataTypeString(a.dtype), " vs ",
2244             DataTypeString(b.dtype));
2245     }
2246
2247     b.shape = shape_refiner->OutputAsUnion(qnode, i, a.shape, b.shape);
2248 }
2249 return Status::OK();
2250 }
2251
2252 // Compute the output shape of the merge node as the union of the available
2253 // input shapes.
2254 Status GraphProperties::UpdateMerge(SymbolicShapeRefiner* shape_refiner,
2255     const NodeDef* node,
2256     bool* new_shapes) const {
2257     InferenceContext* ic = shape_refiner->GetContext(node);
2258     if (!ic) {
2259         // Now we can run shape inference
2260         TF_RETURN_IF_ERROR(shape_refiner->AddNode(node));
2261         ic = CHECK_NOTNULL(shape_refiner->GetContext(node));
2262         *new_shapes = true;
2263
2264         // Infer the shape of the second output once and for all since it never
2265         // changes.
2266         ShapeHandle out1 = ic->Scalar();
2267         if (ic->num_outputs() >= 2) ic->set_output(1, out1);
2268     }
2269
2270     ShapeHandle out;
2271     const std::vector<ShapeAndType>* out_handle = nullptr;
2272     bool out_initialized = false;
2273     for (const GraphView::Edge fanin : shape_refiner->graph().GetFaninEdges(
2274         *node, /*include_controlling_edges=*/false)) {
2275         InferenceContext* src_ic = shape_refiner->GetContext(fanin.src.node);
2276         if (!src_ic) {
2277             // Handling a loop for the first time, the back edge won't have any shape
2278             // info.
2279             continue;
2280         }
2281         ShapeHandle input = src_ic->output(fanin.src.port_id);
2282         ic->SetInput(fanin.dst.port_id, input);
2283         auto* input_handle =

```

```

2284     src_ic->output_handle_shapes_and_types(fanin.src.port_id);
2285     if (input_handle)
2286         ic->set_input_handle_shapes_and_types(fanin.dst.port_id, *input_handle);
2287     if (!out_initialized) {
2288         out_initialized = true;
2289         out = input;
2290         out_handle = input_handle;
2291     } else {
2292         // Note here only out, not out_handle, is modified.
2293         out = shape_refiner->OutputAsUnion(node, 0, input, out);
2294     }
2295 }
2296
2297 if (*new_shapes || !shape_refiner->EquivalentShapes(out, ic->output(0))) {
2298     ic->set_output(0, out);
2299     if (out_handle) ic->set_output_handle_shapes_and_types(0, *out_handle);
2300     *new_shapes = true;
2301 }
2302
2303 return Status::OK();
2304 }
2305
2306 // Manually propagate the input shape for Enter nodes.
2307 Status GraphProperties::UpdateEnter(SymbolicShapeRefiner* shape_refiner,
2308                                     const NodeDef* node, bool* new_shapes) {
2309     InferenceContext* ic = shape_refiner->GetContext(node);
2310     if (!ic) {
2311         TF_RETURN_IF_ERROR(shape_refiner->UpdateNode(node, new_shapes));
2312         ic = shape_refiner->GetContext(node);
2313     }
2314
2315     GraphView::InputPort port(node, 0);
2316     GraphView::OutputPort fanin = shape_refiner->graph().GetRegularFanin(port);
2317
2318     InferenceContext* src_ic = shape_refiner->GetContext(fanin.node);
2319     ShapeHandle input = src_ic->output(fanin.port_id);
2320     if (!ic->output(0).SameHandle(input)) {
2321         ic->SetInput(0, input);
2322         ic->set_output(0, input);
2323         *new_shapes = true;
2324     }
2325     auto* outputs = src_ic->output_handle_shapes_and_types(fanin.port_id);
2326     if (outputs) {
2327         ic->set_input_handle_shapes_and_types(0, *outputs);
2328         ic->set_output_handle_shapes_and_types(0, *outputs);
2329         *new_shapes = true;
2330     }
2331     return Status::OK();
2332 }

```



```

2333
2334 Status GraphProperties::UpdateShapes(
2335     SymbolicShapeRefiner* shape_refiner,
2336     const absl::flat_hash_map<const NodeDef*, const NodeDef*>& resource_handles,
2337     const NodeDef* n, bool* new_shapes) const {
2338     if (IsEnter(*n)) {
2339         // The Enter shape function always forwards an UnknownShape, so do the right
2340         // thing here.
2341         TF_RETURN_IF_ERROR(UpdateEnter(shape_refiner, n, new_shapes));
2342     } else if (IsMerge(*n)) {
2343         // Properly handle merge nodes.
2344         TF_RETURN_IF_ERROR(UpdateMerge(shape_refiner, n, new_shapes));
2345     } else if (IsEnqueue(*n)) {
2346         // Make sure the shapes of enqueued tensors are propagated to the queue
2347         // itself.
2348         TF_RETURN_IF_ERROR(
2349             UpdateEnqueue(n, resource_handles, shape_refiner, new_shapes));
2350     } else if (IsQueue(*n)) {
2351         // Set shapes and types of Queue ops, if needed.
2352         TF_RETURN_IF_ERROR(UpdateQueue(n, shape_refiner, new_shapes));
2353     } else {
2354         // Rely on regular TF shape refinement for all the other nodes.
2355         // UpdateNode calls UpdateFunction if a function node is detected.
2356         TF_RETURN_IF_ERROR(shape_refiner->UpdateNode(n, new_shapes));
2357     }
2358
2359     return Status::OK();
2360 }
2361
2362 // Propagates the shapes in the transitive fan-out of <new_shapes>.
2363 Status GraphProperties::PropagateShapes(
2364     SymbolicShapeRefiner* shape_refiner, TopoQueue* new_shapes,
2365     const absl::flat_hash_map<const NodeDef*, const NodeDef*>& resource_handles,
2366     int num_loops) const {
2367     // Limit the number of iterations to prevent infinite loops in the presence of
2368     // incorrect shape functions. The algorithm should converge in at most
2369     // num_nested_loops^2 * max_rank. We approximate max_rank with the constant 4.
2370     // The same applies to resources.
2371     VLOG(1) << "Propagating " << new_shapes->size() << " new shapes through "
2372         << num_loops << " loops and " << resource_handles.size()
2373         << " resources" << std::endl;
2374
2375     const int64_t max_loop_length = item_.graph.node_size();
2376     const int64_t max_rank = 4;
2377     const int64_t max_loop_iterations =
2378         max_rank * max_loop_length * std::max<int64_t>(1, num_loops * num_loops);
2379     const int64_t num_queues = resource_handles.size();
2380     const int64_t max_resource_iterations = num_queues * num_queues * max_rank;
2381

```

```

2382     int64_t num_resource_iterations = 0;
2383     do {
2384         int64_t num_loop_iterations = 0;
2385         while (!new_shapes->empty() &&
2386             num_loop_iterations++ < max_loop_iterations) {
2387             const NodeDef* n = new_shapes->pop();
2388             bool updated = false;
2389             TF_RETURN_IF_ERROR(
2390                 UpdateShapes(shape_refiner, resource_handles, n, &updated));
2391             if (updated) {
2392                 for (const auto& fanout : shape_refiner->graph().GetFanouts(
2393                     *n, /*include_controlled_nodes=*/false)) {
2394                     new_shapes->push(fanout.node);
2395                 }
2396                 // Make sure the corresponding queue nodes are (re)processed.
2397                 if (IsEnqueue(*n)) {
2398                     auto it = resource_handles.find(n);
2399                     if (it != resource_handles.end()) {
2400                         new_shapes->push(it->second);
2401                     }
2402                 }
2403             }
2404         }
2405     } while (!new_shapes->empty() &&
2406         num_resource_iterations++ < max_resource_iterations);
2407
2408     if (!new_shapes->empty()) {
2409         return errors::Internal("Shape inference failed to converge");
2410     }
2411
2412     return Status::OK();
2413 }
2414
2415 Status GraphProperties::UpdateQueue(const NodeDef* queue_node,
2416     SymbolicShapeRefiner* shape_refiner,
2417     bool* new_shapes) {
2418     auto* ctx = shape_refiner->GetNodeContext(queue_node);
2419     if (!ctx) {
2420         TF_RETURN_IF_ERROR(shape_refiner->AddNode(queue_node));
2421         ctx = CHECK_NOTNULL(shape_refiner->GetNodeContext(queue_node));
2422     }
2423     auto* ic = ctx->inference_context.get();
2424
2425     auto* outputs = ic->output_handle_shapes_and_types(0);
2426     if (outputs) {
2427         // Shapes and types are already set, presumably by Enqueue ops.
2428         return shape_refiner->UpdateNode(queue_node, new_shapes);
2429     }
2430

```

```

2431     if (queue_node->attr().count("shapes") <= 0 ||
2432         queue_node->attr().count("component_types") <= 0 ||
2433         queue_node->attr().at("shapes").list().shape_size() !=
2434             queue_node->attr().at("component_types").list().type_size()) {
2435         // Errors in shapes and component_types attr.
2436         return shape_refiner->UpdateNode(queue_node, new_shapes);
2437     }
2438
2439     // Extract types and shapes from Queue attr.
2440     const auto& shapes = queue_node->attr().at("shapes").list().shape();
2441     const auto& types = queue_node->attr().at("component_types").list().type();
2442     std::vector<ShapeAndType> shapes_and_types;
2443     for (int i = 0; i < types.size(); i++) {
2444         const auto& shape = shapes[i];
2445         ShapeHandle shape_handle;
2446         TF_RETURN_IF_ERROR(
2447             ic->MakeShapeFromPartialTensorShape(shape, &shape_handle));
2448         DataType data_type =
2449             queue_node->attr().at("component_types").list().type(i);
2450         ShapeAndType shape_and_type(shape_handle, data_type);
2451         shapes_and_types.push_back(shape_and_type);
2452     }
2453     ic->set_output_handle_shapes_and_types(0, shapes_and_types);
2454
2455     // Queue node is updated with output_handle_shapes_and_types, so set
2456     // new_shapes and ignore it from UpdateNoe().
2457     *new_shapes = true;
2458     bool dummy_new_shapes = false;
2459     return shape_refiner->UpdateNode(queue_node, &dummy_new_shapes);
2460 }
2461
2462 Status GraphProperties::UpdateEnqueue(
2463     const NodeDef* enqueue_node,
2464     const absl::flat_hash_map<const NodeDef*, const NodeDef*>& resource_handles,
2465     SymbolicShapeRefiner* shape_refiner, bool* new_shapes) {
2466     auto ctx = shape_refiner->GetNodeContext(enqueue_node);
2467     if (!ctx) {
2468         TF_RETURN_IF_ERROR(shape_refiner->AddNode(enqueue_node));
2469         ctx = CHECK_NOTNULL(shape_refiner->GetNodeContext(enqueue_node));
2470     }
2471
2472     auto it = resource_handles.find(enqueue_node);
2473     if (it == resource_handles.end()) {
2474         // The corresponding queue was not found, there isn't much we can do.
2475         return Status::OK();
2476     }
2477     const NodeDef* qnode = it->second;
2478     auto qctx = shape_refiner->GetContext(qnode);
2479     if (!qctx) {

```

```

2480     return Status::OK();
2481 }
2482 auto* queue_handle_data = qctx->output_handle_shapes_and_types(0);
2483
2484 // TODO(bsteiner): handle EnqueueMany as well.
2485 std::vector<ShapeAndType> shapes_and_types;
2486 for (int i = 1, end = ctx->input_types.size(); i < end; ++i) {
2487     GraphView::InputPort inp(enqueue_node, i);
2488     GraphView::OutputPort fanin = shape_refiner->graph().GetRegularFanin(inp);
2489     InferenceContext* in = shape_refiner->GetContext(fanin.node);
2490     ShapeHandle input = in->output(fanin.port_id);
2491     ctx->inference_context->SetInput(i, input);
2492     shapes_and_types.push_back({input, ctx->input_types[i]});
2493 }
2494
2495 if (queue_handle_data == nullptr) {
2496     qctx->set_output_handle_shapes_and_types(0, shapes_and_types);
2497     *new_shapes = true;
2498 } else {
2499     TF_RETURN_IF_ERROR(RelaxEnqueueShapesAndMergeTypes(
2500         shape_refiner, qnode, *queue_handle_data, &shapes_and_types));
2501     *new_shapes |= !shape_refiner->EquivalentShapesAndTypes(*queue_handle_data,
2502         shapes_and_types);
2503     qctx->set_output_handle_shapes_and_types(0, shapes_and_types);
2504 }
2505
2506 return Status::OK();
2507 }
2508
2509 Status GraphProperties::InferStatically(bool assume_valid_feeds,
2510                                         bool aggressive_shape_inference,
2511                                         bool include_input_tensor_values,
2512                                         bool include_output_tensor_values) {
2513     FunctionLibraryDefinition function_library(OpRegistry::Global(),
2514                                               item_.graph.library());
2515     absl::flat_hash_map<string, absl::flat_hash_set<int>> fed_ports;
2516     if (!assume_valid_feeds) {
2517         for (const auto& feed : item_.feed) {
2518             SafeTensorId tensor_id = ParseTensorName(feed.first);
2519             fed_ports[tensor_id.node()].insert(tensor_id.index());
2520         }
2521     }
2522
2523     GraphView graph_view(&item_.graph);
2524
2525     // List the resources and the nodes using them. Also collect the Merge nodes,
2526     // fed nodes, and primary inputs.
2527     absl::flat_hash_map<const NodeDef*,
2528         std::pair<absl::flat_hash_set<const NodeDef*>,

```



```

2578     extra_deps.emplace_back(src, dst);
2579 }
2580 }
2581 }
2582
2583 std::vector<const NodeDef*> topo_order;
2584 Status s = ComputeTopologicalOrder(item_.graph, extra_deps, &topo_order);
2585 if (!s.ok()) {
2586     if (extra_deps.empty()) {
2587         return s;
2588     } else {
2589         // There is a loop between queues: we'll just use the graph topological
2590         // order. This will make the shape inference less precise but since this
2591         // isn't common it's not worth to figure out where to break the loop and
2592         // do a proper relaxation.
2593         TF_RETURN_IF_ERROR(ComputeTopologicalOrder(item_.graph, &topo_order));
2594     }
2595 }
2596
2597 // Heap-allocate SymbolicShapeRefiner in order to not consume a large amount
2598 // of stack space.
2599 auto refiner = absl::make_unique<SymbolicShapeRefiner>(
2600     graph_view, fed_ports, aggressive_shape_inference);
2601
2602 TopoQueue new_shapes(topo_order);
2603 // Also seed the propagation of shapes in the fanout of primary inputs.
2604 for (const NodeDef* node : primary_inputs) {
2605     new_shapes.push(node);
2606 }
2607 // Also seed the propagation of shapes in the fanout of fed nodes.
2608 for (const NodeDef* node : fed_nodes) {
2609     new_shapes.push(node);
2610 }
2611 // Propagate shapes normally.
2612 TF_RETURN_IF_ERROR(
2613     PropagateShapes(refiner.get(), &new_shapes, resource_handles, num_loops));
2614
2615 // Track shapes globally across the graph.
2616 std::unique_ptr<SymbolicShapeManager> shape_manager =
2617     absl::make_unique<SymbolicShapeManager>();
2618 bool found_error = false;
2619 for (const NodeDef& node : item_.graph.node()) {
2620     auto node_ctx = refiner->GetContext(&node);
2621     if (!node_ctx) {
2622         continue;
2623     }
2624     // Skip any information that comes from fed nodes.
2625     if (fed_ports.find(node.name()) != fed_ports.end()) {
2626         VLOG(2) << "Skipping feed node shape: " << node.name();

```

```

2627     continue;
2628 }
2629 for (const auto& merged_shapes : node_ctx->MergedShapes()) {
2630     if (!shape_manager->Merge(merged_shapes.first, merged_shapes.second)
2631         .ok()) {
2632         found_error = true;
2633         break;
2634     }
2635 }
2636 for (const auto& merged_dims : node_ctx->MergedDims()) {
2637     if (!shape_manager->Merge(merged_dims.first, merged_dims.second).ok()) {
2638         found_error = true;
2639         break;
2640     }
2641 }
2642 if (found_error) {
2643     // The shapes aren't consistent, we can't infer safely: discard all the
2644     // information discovered so far.
2645     shape_manager = absl::make_unique<SymbolicShapeManager>();
2646     break;
2647 }
2648 }
2649
2650 TF_RETURN_IF_ERROR(ValidateSymbolicShapeManager(item_.graph, refiner.get(),
2651     shape_manager.get()));
2652
2653 for (const NodeDef& node : item_.graph.node()) {
2654     VLOG(4) << "Filling in graph properties for node: " << node.name();
2655     auto ctx = refiner->GetNodeContext(&node);
2656     if (!ctx) {
2657         continue;
2658     }
2659
2660     auto* ic = ctx->inference_context.get();
2661
2662     // Fill input properties.
2663     {
2664         auto& input_properties = input_properties_[node.name()];
2665
2666         // Should always be empty, node names in graph are supposed to be unique.
2667         CHECK_EQ(input_properties.size(), 0);
2668
2669         input_properties.resize(ic->num_inputs());
2670         GraphView::InputPort input(&node, -1);
2671         for (int i = 0; i < ic->num_inputs(); ++i) {
2672             shape_manager->AsTensorProperties(ic->input(i), ctx->input_types[i],
2673                 &input_properties[i]);
2674             input.port_id = i;
2675             GraphView::OutputPort fanin = graph_view.GetRegularFanin(input);

```

```

2676     if (include_input_tensor_values) {
2677         // Export tensor value to input_properties.value.
2678         if (IsConstant(*fanin.node)) {
2679             const TensorProto& raw_val =
2680                 fanin.node->attr().at("value").tensor();
2681             *input_properties[i].mutable_value() = raw_val;
2682         } else if (static_cast<int>(ctx->input_tensor_protos.size()) > i &&
2683             ctx->input_tensor_protos[i] != nullptr) {
2684             *input_properties[i].mutable_value() = *ctx->input_tensor_protos[i];
2685         } else if (static_cast<int>(ic->input_tensors_as_shapes().size()) >
2686             i &&
2687             IsShapeFullyDefinedIntegerVectorOrScalar(
2688                 ic, ic->input(i), ic->input_tensors_as_shapes()[i],
2689                 ctx->input_types[i])) {
2690             *input_properties[i].mutable_value() = MakeTensorProtoFromShape(
2691                 ic, ic->input(i), ic->input_tensors_as_shapes()[i],
2692                 ctx->input_types[i]);
2693         }
2694     }
2695 }
2696 }
2697
2698 // Fill output properties.
2699 {
2700     auto& output_properties = output_properties_[node.name()];
2701
2702     // Should always be empty, node names in graph are supposed to be unique.
2703     CHECK_EQ(output_properties.size(), 0);
2704
2705     output_properties.resize(ic->num_outputs());
2706     for (int i = 0; i < ic->num_outputs(); ++i) {
2707         shape_manager->AsTensorProperties(ic->output(i), ctx->output_types[i],
2708             &output_properties[i]);
2709         auto converted_output_tensors_as_shapes =
2710             ReplaceUnknownDimFromConstWithUnknownDim(
2711                 ic, ctx->output_tensors_as_shapes);
2712         if (include_output_tensor_values) {
2713             // Export tensor value to output_properties.value.
2714             if (IsConstant(node)) {
2715                 // TODO(rmlarsen): Eliminate this copy.
2716                 const TensorProto& raw_val = node.attr().at("value").tensor();
2717                 *output_properties[i].mutable_value() = raw_val;
2718             } else if (static_cast<int>(ctx->output_tensor_protos.size()) > i &&
2719                 ctx->output_tensor_protos[i] != nullptr) {
2720                 *output_properties[i].mutable_value() =
2721                     *ctx->output_tensor_protos[i];
2722             } else if (static_cast<int>(
2723                 converted_output_tensors_as_shapes.size()) > i &&
2724                 IsShapeFullyDefinedIntegerVectorOrScalar(

```



```

2725         ic, ic->output(i),
2726         converted_output_tensors_as_shapes[i],
2727         ctx->output_types[i])) {
2728     *output_properties[i].mutable_value() = MakeTensorProtoFromShape(
2729         ic, ic->output(i), converted_output_tensors_as_shapes[i],
2730         ctx->output_types[i]);
2731     }
2732 }
2733 }
2734 }
2735
2736 if (aggressive_shape_inference && ctx->shape_incompatible)
2737     incompatible_shape_nodes_.insert(node.name());
2738 }
2739
2740 if (aggressive_shape_inference && !incompatible_shape_nodes_.empty())
2741     LOG(WARNING) << incompatible_shape_nodes_.size()
2742         << " nodes have incompatible output shapes.";
2743
2744 // Help trace the unknown dimensions to their origins.
2745 VerboseLogUnknownDimensionSources(item_.graph, input_properties_,
2746     output_properties_);
2747
2748 TF_RETURN_IF_ERROR(VerboseShapeInferenceLogging(item_.graph, refiner.get(),
2749     shape_manager.get()));
2750
2751 return Status::OK();
2752 }
2753
2754 Status GraphProperties::InferDynamically(Cluster* cluster) {
2755     TF_RETURN_IF_ERROR(cluster->Initialize(item_));
2756
2757     // Runs the model once to collect the shapes in the cost model.
2758     RunMetadata metadata;
2759     TF_RETURN_IF_ERROR(
2760         cluster->Run(item_.graph, item_.feed, item_.fetch, &metadata));
2761
2762     return InferFromCostGraph(metadata.cost_graph());
2763 }
2764
2765 Status GraphProperties::AnnotateOutputShapes(GraphDef* output_graph_def) const {
2766     *output_graph_def = item_.graph;
2767     for (int i = 0; i < output_graph_def->node_size(); i++) {
2768         auto node = output_graph_def->mutable_node(i);
2769         AttrValue attr_output_shape;
2770         auto tensor_properties = GetOutputProperties(node->name());
2771         for (const auto& tensor_property : tensor_properties) {
2772             TensorShapeProto* proto = attr_output_shape.mutable_list()->add_shape();
2773             *proto = tensor_property.shape();

```

```

2774     NormalizeShapeForOutput(proto);
2775 }
2776 (*node->mutable_attr())["_output_shapes"] = std::move(attr_output_shape);
2777 }
2778 return Status::OK();
2779 }
2780
2781 Status GraphProperties::InferFromCostGraph(const CostGraphDef& cost_graph) {
2782     if (cost_graph.node_size() == 0) {
2783         LOG(WARNING) << "cost_graph is empty: nothing can be inferred!";
2784     }
2785     std::unordered_map<string, const CostGraphDef::Node*> name_to_cost;
2786     std::unordered_map<string, const NodeDef*> name_to_node; // Empty
2787     for (auto& node : cost_graph.node()) {
2788         name_to_cost[node.name()] = &node;
2789
2790         std::vector<OpInfo::TensorProperties> output_properties;
2791         for (const auto& out : node.output_info()) {
2792             OpInfo::TensorProperties properties;
2793             properties.set_dtype(out.dtype());
2794             *properties.mutable_shape() = out.shape();
2795             output_properties.push_back(properties);
2796         }
2797         output_properties_[node.name()] = output_properties;
2798     }
2799
2800     for (const auto& node : item_.graph.node()) {
2801         // Skip the nodes that are not in the cost graph: these are nodes that
2802         // aren't run, because they aren't in the intersection of transitive fan-in
2803         // of a fetch node and the transitive fan-out of an input, or nodes that
2804         // were optimized away by the optimizer.
2805         auto it = name_to_cost.find(node.name());
2806         if (it == name_to_cost.end()) {
2807             continue;
2808         }
2809         std::vector<OpInfo::TensorProperties> inputs =
2810             FindInputFeatures(node, name_to_cost, name_to_node);
2811
2812         input_properties_[node.name()] = inputs;
2813     }
2814     return Status::OK();
2815 }
2816
2817 bool GraphProperties::HasInputProperties(const string& node_name) const {
2818     return input_properties_.find(node_name) != input_properties_.end();
2819 }
2820
2821 bool GraphProperties::HasOutputProperties(const string& node_name) const {
2822     return output_properties_.find(node_name) != output_properties_.end();

```

```
2823 }
2824
2825 const std::vector<OpInfo::TensorProperties>&
2826 GraphProperties::GetInputProperties(const string& node_name) const {
2827     auto it = input_properties_.find(node_name);
2828     if (it != input_properties_.end()) {
2829         return it->second;
2830     }
2831     return missing_properties_;
2832 }
2833
2834 const std::vector<OpInfo::TensorProperties>&
2835 GraphProperties::GetOutputProperties(const string& node_name) const {
2836     auto it = output_properties_.find(node_name);
2837     if (it != output_properties_.end()) {
2838         return it->second;
2839     }
2840     return missing_properties_;
2841 }
2842
2843 void GraphProperties::ClearInputProperties(const string& node_name) {
2844     input_properties_.erase(node_name);
2845 }
2846 void GraphProperties::ClearOutputProperties(const string& node_name) {
2847     output_properties_.erase(node_name);
2848 }
2849
2850 } // end namespace grappler
2851 } // end namespace tensorflow
```