faizan-m Avoid losing payloads at Status recreation ... History

21 contributors +9

1304 lines (1192 sloc) | 43.3 KB ...

```
1  /* Copyright 2016 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 #include "tensorflow/core/framework/shape_inference.h"
16
17 #include "tensorflow/core/framework/bounds_check.h"
18 #include "tensorflow/core/framework/full_type_util.h"
19 #include "tensorflow/core/framework/node_def.pb.h"
20 #include "tensorflow/core/framework/op_def.pb.h"
21 #include "tensorflow/core/framework/partial_tensor_shape.h"
22 #include "tensorflow/core/framework/tensor_shape.pb.h"
23 #include "tensorflow/core/lib/core/errors.h"
24 #include "tensorflow/core/lib/strings/numbers.h"
25 #include "tensorflow/core/lib/strings/scanner.h"
26 #include "tensorflow/core/lib/strings/str_util.h"
27
28 namespace tensorflow {
29 namespace shape_inference {
```

```
30
31  constexpr int32_t InferenceContext::kUnknownRank;
32  constexpr int64_t InferenceContext::kUnknownDim;
33
34  // Same as above, but with PartialTensorShape instead of TensorShapeProto
35  InferenceContext::InferenceContext(
36      int graph_def_version, const AttrSlice& attrs, const OpDef& op_def,
37      const std::vector<PartialTensorShape>& input_shapes,
38      const std::vector<const Tensor*>& input_tensors,
39      const std::vector<PartialTensorShape>& input_tensors_as_shapes,
40      const std::vector<
41          std::unique_ptr<std::vector<std::pair<PartialTensorShape, DataType>>>>&
42          input_handle_shapes_and_types)
43      : graph_def_version_(graph_def_version), attrs_(attrs) {
44    std::vector<ShapeHandle> input_tensors_as_shape_handles;
45    input_tensors_as_shape_handles.reserve(input_tensors_as_shapes.size());
46    for (const PartialTensorShape& p : input_tensors_as_shapes) {
47      ShapeHandle shape;
48      construction_status_.Update(MakeShapeFromPartialTensorShape(p, &shape));
49      if (!construction_status_.ok()) {
50        return;
51      }
52      input_tensors_as_shape_handles.push_back(shape);
53    }
54    PreInputInit(op_def, input_tensors, input_tensors_as_shape_handles);
55    if (!construction_status_.ok()) return;
56    inputs_.reserve(input_shapes.size());
57    for (const PartialTensorShape& p : input_shapes) {
58      ShapeHandle shape;
59      construction_status_.Update(MakeShapeFromPartialTensorShape(p, &shape));
60      if (!construction_status_.ok()) {
61        return;
62      }
63      inputs_.push_back(shape);
64    }
65    std::vector<std::unique_ptr<std::vector<ShapeAndType>>> handle_data(
66        input_shapes.size());
67    for (int i = 0, end = input_handle_shapes_and_types.size(); i < end; ++i) {
68      const auto& v = input_handle_shapes_and_types[i];
69      if (v == nullptr) {
70        continue;
71      }
72      handle_data[i].reset(new std::vector<ShapeAndType>(v->size()));
73      auto& new_v = *handle_data[i];
74      for (int j = 0, end = v->size(); j < end; ++j) {
75        const auto& p = (*v)[j];
76        construction_status_.Update(
77            MakeShapeFromPartialTensorShape(p.first, &new_v[j].shape));
78        if (!construction_status_.ok()) {
```

```
        return;
      }
      new_v[j].dtype = p.second;
    }
  }
  PostInputInit(std::move(handle_data));
}

InferenceContext::InferenceContext(
    int graph_def_version, const AttrSlice& attrs, const OpDef& op_def,
    const std::vector<ShapeHandle>& input_shapes,
    const std::vector<const Tensor*>& input_tensors,
    const std::vector<ShapeHandle>& input_tensors_as_shapes,
    std::vector<std::unique_ptr<std::vector<ShapeAndType>>>
        input_handle_shapes_and_types)
    : graph_def_version_(graph_def_version), attrs_(attrs) {
  PreInputInit(op_def, input_tensors, input_tensors_as_shapes);
  if (!construction_status_.ok()) return;
  inputs_ = input_shapes;

  PostInputInit(std::move(input_handle_shapes_and_types));
}

InferenceContext::~InferenceContext() {}

Status InferenceContext::Run(
    const std::function<Status(shape_inference::InferenceContext* c)>& fn) {
  ForgetMerges();
  Status s = fn(this);
  if (!s.ok()) {
    ForgetMerges();
    return AttachContext(s);
  }
#ifndef NDEBUG
  for (int i = 0; i < num_outputs(); ++i) {
    DCHECK(output(i).IsSet()) << i << " for " << attrs_.SummarizeNode();
  }
#endif  // NDEBUG
  return s;
}

Status InferenceContext::set_output(StringPiece output_name,
                                    const std::vector<ShapeHandle>& shapes) {
  auto result = output_name_map_.find(output_name);
  if (result == output_name_map_.end()) {
    return errors::InvalidArgument("Unknown output name: ", output_name);
  } else {
    const int start = result->second.first;
    const int size = result->second.second - start;
```

```
128     const int shapes_size = shapes.size();
129     if (size != shapes_size) {
130       return errors::InvalidArgument("Must have exactly ", shapes.size(),
131                                      " shapes.");
132     }
133     for (int i = 0; i < shapes_size; ++i) {
134       outputs_[i + start] = shapes[i];
135     }
136   }
137   return Status::OK();
138 }
139
140 Status InferenceContext::input(StringPiece input_name,
141                                std::vector<ShapeHandle>* output) const {
142   const auto result = input_name_map_.find(input_name);
143   if (result == input_name_map_.end()) {
144     return errors::InvalidArgument("Unknown input name: ", input_name);
145   } else {
146     output->clear();
147     for (int i = result->second.first; i < result->second.second; ++i) {
148       output->push_back(inputs_[i]);
149     }
150   }
151   return Status::OK();
152 }
153
154 Status InferenceContext::output(StringPiece output_name,
155                                 std::vector<ShapeHandle>* output) const {
156   const auto result = output_name_map_.find(output_name);
157   if (result == output_name_map_.end()) {
158     return errors::InvalidArgument("Unknown output name: ", output_name);
159   } else {
160     output->clear();
161     for (int i = result->second.first; i < result->second.second; ++i) {
162       output->push_back(outputs_[i]);
163     }
164   }
165   return Status::OK();
166 }
167
168 void InferenceContext::PreInputInit(
169     const OpDef& op_def, const std::vector<const Tensor*>& input_tensors,
170     const std::vector<ShapeHandle>& input_tensors_as_shapes) {
171   // TODO(mdan): This is also done at graph construction. Run only here instead?
172   const auto ret = full_type::SpecializeType(attrs_, op_def);
173   DCHECK(ret.status().ok()) << "while instantiating types: " << ret.status();
174   ret_types_ = ret.ValueOrDie();
175
176   input_tensors_ = input_tensors;
```

```
  input_tensors_as_shapes_ = input_tensors_as_shapes;

  construction_status_ =
      NameRangesForNode(attrs_, op_def, &input_name_map_, &output_name_map_);
  if (!construction_status_.ok()) return;

  int num_outputs = 0;
  for (const auto& e : output_name_map_) {
    num_outputs = std::max(num_outputs, e.second.second);
  }
  outputs_.assign(num_outputs, nullptr);
  output_handle_shapes_and_types_.resize(num_outputs);
}

Status InferenceContext::ExpandOutputs(int new_output_size) {
  const int outputs_size = outputs_.size();
  if (new_output_size < outputs_size) {
    return errors::InvalidArgument("Trying to reduce number of outputs of op.");
  }
  outputs_.resize(new_output_size, nullptr);
  output_handle_shapes_and_types_.resize(new_output_size);
  return Status::OK();
}

void InferenceContext::PostInputInit(
    std::vector<std::unique_ptr<std::vector<ShapeAndType>>> input_handle_data) {
  int num_inputs_from_node_def = 0;
  for (const auto& e : input_name_map_) {
    num_inputs_from_node_def =
        std::max(num_inputs_from_node_def, e.second.second);
  }

  // Allow passing empty shapes/dtypes to avoid changing every single test.
  if (input_handle_data.empty()) {
    input_handle_shapes_and_types_.resize(inputs_.size());
  } else {
    if (input_handle_data.size() != inputs_.size()) {
      construction_status_ = errors::InvalidArgument(
          "Wrong number of handle shapes passed; expected ", inputs_.size(),
          " got ", input_handle_data.size());
      return;
    }
    input_handle_shapes_and_types_ = std::move(input_handle_data);
  }
  const int inputs_size = inputs_.size();
  if (inputs_size != num_inputs_from_node_def) {
    construction_status_ = errors::InvalidArgument(
        "Wrong number of inputs passed: ", inputs_.size(), " while ",
        num_inputs_from_node_def, " expected based on NodeDef");
```

```
226       return;
227     }
228
229     CHECK_LE(input_tensors_.size(), inputs_.size());
230     input_tensors_.resize(inputs_.size());
231     requested_input_tensor_.resize(inputs_.size());
232     requested_input_tensor_as_partial_shape_.resize(inputs_.size());
233   }
234
235   void InferenceContext::ShapeHandleToProto(ShapeHandle handle,
236                                             TensorShapeProto* proto) {
237     if (!RankKnown(handle)) {
238       proto->set_unknown_rank(true);
239       return;
240     }
241
242     for (int32_t i = 0; i < Rank(handle); ++i) {
243       DimensionHandle dim = Dim(handle, i);
244       auto* dim_shape = proto->add_dim();
245       if (ValueKnown(dim)) {
246         dim_shape->set_size(Value(dim));
247       } else {
248         dim_shape->set_size(-1);
249       }
250     }
251   }
252
253   bool InferenceContext::FullyDefined(ShapeHandle s) {
254     if (!RankKnown(s)) return false;
255     for (int i = 0; i < Rank(s); ++i) {
256       if (!ValueKnown(Dim(s, i))) return false;
257     }
258     return true;
259   }
260
261   DimensionHandle InferenceContext::NumElements(ShapeHandle s) {
262     const auto rank = Rank(s);
263     if (rank == kUnknownRank) return UnknownDim();
264     bool found_unknown = false;
265     int64_t size = 1;
266     for (int i = 0; i < rank; ++i) {
267       int64_t dim_val = Value(Dim(s, i));
268       if (dim_val == kUnknownDim) {
269         found_unknown = true;
270       } else if (dim_val == 0) {
271         return MakeDim(0);
272       } else {
273         size *= dim_val;
274       }
```

```
275     }
276     if (found_unknown) {
277       return UnknownDim();
278     } else {
279       return MakeDim(size);
280     }
281   }
282
283   string InferenceContext::DebugString(ShapeHandle s) {
284     if (RankKnown(s)) {
285       std::vector<string> vals;
286       for (auto d : s->dims_) vals.push_back(DebugString(d));
287       return strings::StrCat("[", absl::StrJoin(vals, ","), "]");
288     } else {
289       return "?";
290     }
291   }
292
293   string InferenceContext::DebugString(DimensionHandle d) {
294     return ValueKnown(d) ? strings::StrCat(Value(d)) : "?";
295   }
296
297   string InferenceContext::DebugString() const {
298     return strings::StrCat("InferenceContext for node: ", attrs_.SummarizeNode());
299   }
300
301   string InferenceContext::DebugString(const ShapeAndType& shape_and_type) {
302     return strings::StrCat(DebugString(shape_and_type.shape), ":",
303                            DataTypeString(shape_and_type.dtype));
304   }
305
306   string InferenceContext::DebugString(
307       gtl::ArraySlice<ShapeAndType> shape_and_types) {
308     std::vector<string> pieces;
309     for (const ShapeAndType& s : shape_and_types) {
310       pieces.push_back(DebugString(s));
311     }
312     return strings::StrCat("[", absl::StrJoin(pieces, ","), "]");
313   }
314
315   Status InferenceContext::WithRank(ShapeHandle shape, int64_t rank,
316                                     ShapeHandle* out) {
317     if (rank > kint32max) {
318       return errors::InvalidArgument("Rank cannot exceed kint32max");
319     }
320     const int32_t existing = Rank(shape);
321     if (existing == rank) {
322       *out = shape;
323       return Status::OK();
```

```
  }
  if (existing == kUnknownRank) {
    std::vector<DimensionHandle> dims;
    dims.reserve(rank);
    for (int i = 0; i < rank; ++i) {
      dims.push_back(UnknownDim());
    }
    ShapeHandle shp = shape_manager_.MakeShape(dims);
    return Merge(shape, shp, out);
  }
  *out = nullptr;

  return errors::InvalidArgument("Shape must be rank ", rank, " but is rank ",
                                 existing);
}

Status InferenceContext::WithRankAtLeast(ShapeHandle shape, int64_t rank,
                                         ShapeHandle* out) {
  if (rank > kint32max) {
    return errors::InvalidArgument("Rank cannot exceed kint32max");
  }
  const int32_t existing = Rank(shape);
  if (existing >= rank || existing == kUnknownRank) {
    *out = shape;
    return Status::OK();
  }
  *out = nullptr;
  return errors::InvalidArgument("Shape must be at least rank ", rank,
                                 " but is rank ", existing);
}

Status InferenceContext::WithRankAtMost(ShapeHandle shape, int64_t rank,
                                        ShapeHandle* out) {
  if (rank > kint32max) {
    return errors::InvalidArgument("Rank cannot exceed kint32max");
  }
  const int32_t existing = Rank(shape);
  if (existing <= rank || existing == kUnknownRank) {
    *out = shape;
    return Status::OK();
  }
  *out = nullptr;
  return errors::InvalidArgument("Shape must be at most rank ", rank,
                                 " but is rank ", existing);
}

Status InferenceContext::WithValue(DimensionHandle dim, int64_t value,
                                   DimensionHandle* out) {
  const int64_t existing = Value(dim);
```

```
373   if (existing == value) {
374     *out = dim;
375     return Status::OK();
376   }
377   if (existing == kUnknownDim) {
378     DimensionHandle d = MakeDim(value);
379     return Merge(dim, d, out);
380   }
381   *out = nullptr;
382   return errors::InvalidArgument("Dimension must be ", value, " but is ",
383                                  existing);
384 }
385
386 void InferenceContext::Relax(DimensionHandle d_old, DimensionHandle d_new,
387                              DimensionHandle* out) {
388   if (d_old.SameHandle(d_new)) {
389     *out = d_old;
390   } else if (!ValueKnown(d_old) && !ValueKnown(d_new)) {
391     // The node will be fed by the dimension d_new instead of d_old: any
392     // equality assertion between d_old and other input dimension on this node
393     // may not be true anymore, so forget them all.
394     ForgetMerges();
395     // Return the new shape handle to force the relaxation to propagate to the
396     // fanout of the context.
397     *out = d_new;
398   } else if (!ValueKnown(d_new)) {
399     ForgetMerges();
400     *out = d_new;
401   } else if (Value(d_old) == Value(d_new)) {
402     // Return the old shape handle. This will stop the relaxation in the fanout
403     // of the context.
404     *out = d_old;
405   } else {
406     // Return a new handle that encodes a different unknown dim.
407     ForgetMerges();
408     *out = UnknownDim();
409   }
410 }
411
412 Status InferenceContext::Merge(DimensionHandle d0, DimensionHandle d1,
413                                DimensionHandle* out) {
414   if (d0.SameHandle(d1)) {
415     *out = d0;
416     return Status::OK();
417   } else if (!ValueKnown(d1)) {
418     *out = d0;
419     merged_dims_.emplace_back(d0, d1);
420     return Status::OK();
421   } else if (!ValueKnown(d0)) {
```

```
422       *out = d1;
423       merged_dims_.emplace_back(d0, d1);
424       return Status::OK();
425     } else if (Value(d0) == Value(d1)) {
426       *out = d0;
427       return Status::OK();
428     } else {
429       *out = nullptr;
430       return errors::InvalidArgument("Dimensions must be equal, but are ",
431                                      Value(d0), " and ", Value(d1));
432     }
433   }
434
435   Status InferenceContext::MergePrefix(ShapeHandle s, ShapeHandle prefix,
436                                        ShapeHandle* s_out,
437                                        ShapeHandle* prefix_out) {
438     *s_out = *prefix_out = nullptr;
439     if (!RankKnown(prefix) || !RankKnown(s)) {
440       *s_out = s;
441       *prefix_out = prefix;
442       return Status::OK();
443     }
444     const int32_t rank = Rank(prefix);
445     TF_RETURN_IF_ERROR(WithRankAtLeast(s, rank, &s));
446
447     // Merge the prefix dims and create the new output shapes.
448     const int32_t rank_s = Rank(s);
449     std::vector<DimensionHandle> dims;
450     dims.reserve(std::max(rank, rank_s));
451     dims.resize(rank);
452     for (int i = 0; i < rank; ++i) {
453       TF_RETURN_IF_ERROR(Merge(Dim(s, i), Dim(prefix, i), &dims[i]));
454     }
455     *prefix_out = MakeShape(dims);
456     for (int i = rank; i < rank_s; ++i) dims.push_back(Dim(s, i));
457     *s_out = MakeShape(dims);
458     return Status::OK();
459   }
460
461   void InferenceContext::Relax(ShapeHandle s_old, ShapeHandle s_new,
462                                ShapeHandle* out) {
463     if (s_old.SameHandle(s_new)) {
464       *out = s_old;
465       return;
466     } else if (!RankKnown(s_new) || !s_old.IsSet()) {
467       ForgetMerges();
468       *out = s_new;
469       return;
470     }
```

```
471
472   const int32_t rank = Rank(s_old);
473   if (rank != Rank(s_new)) {
474     ForgetMerges();
475     *out = UnknownShape();
476     return;
477   }
478
479   bool return_s_old = true;
480   for (int i = 0; i < rank; ++i) {
481     auto d0 = Dim(s_old, i);
482     auto d1 = Dim(s_new, i);
483     if (d0.SameHandle(d1)) continue;
484
485     auto v0 = Value(d0);
486     auto v1 = Value(d1);
487     if (v0 == kUnknownDim || v1 == kUnknownDim || v0 != v1) {
488       return_s_old = false;
489       break;
490     }
491   }
492   if (return_s_old) {
493     *out = s_old;
494     return;
495   }
496
497   // Relax dims.
498   std::vector<DimensionHandle> dims(rank);
499   for (int i = 0; i < rank; ++i) {
500     Relax(Dim(s_old, i), Dim(s_new, i), &dims[i]);
501   }
502   ForgetMerges();
503   *out = MakeShape(dims);
504 }
505
506 Status InferenceContext::Merge(ShapeHandle s0, ShapeHandle s1,
507                                ShapeHandle* out) {
508   if (s0.SameHandle(s1)) {
509     *out = s0;
510     return Status::OK();
511   } else if (!RankKnown(s1)) {
512     *out = s0;
513     merged_shapes_.emplace_back(s0, s1);
514     return Status::OK();
515   } else if (!RankKnown(s0)) {
516     *out = s1;
517     merged_shapes_.emplace_back(s0, s1);
518     return Status::OK();
519   }
```

```
520
521   const int32_t rank = Rank(s0);
522   if (rank != Rank(s1)) {
523     *out = nullptr;
524     return errors::InvalidArgument("Shapes must be equal rank, but are ", rank,
525                                    " and ", Rank(s1));
526   }
527
528   bool return_s0 = true;
529   bool return_s1 = true;
530   for (int i = 0; i < rank; ++i) {
531     auto d0 = Dim(s0, i);
532     auto d1 = Dim(s1, i);
533     if (d0.SameHandle(d1)) continue;
534
535     auto v0 = Value(d0);
536     auto v1 = Value(d1);
537     if (v0 == kUnknownDim) {
538       if (v1 != kUnknownDim) {
539         return_s0 = false;
540       }
541     } else if (v1 == kUnknownDim) {
542       return_s1 = false;
543     } else if (v0 != v1) {
544       *out = nullptr;
545       return errors::InvalidArgument(
546           "Dimension ", i, " in both shapes must be equal, but are ", Value(d0),
547           " and ", Value(d1), ". Shapes are ", DebugString(s0), " and ",
548           DebugString(s1), ".");
549     }
550   }
551
552   merged_shapes_.emplace_back(s0, s1);
553
554   if (return_s0 || return_s1) {
555     *out = return_s0 ? s0 : s1;
556     return Status::OK();
557   }
558
559   // Merge dims.
560   std::vector<DimensionHandle> dims(rank, nullptr);
561   for (int i = 0; i < rank; ++i) {
562     // Invariant for merge was checked earlier, so CHECK is ok.
563     TF_CHECK_OK(Merge(Dim(s0, i), Dim(s1, i), &dims[i]));
564   }
565
566   Status s = ReturnCreatedShape(dims, out);
567   if (s.ok()) {
568     // Merge the new shape with s0. Since s0 and s1 are merged, this implies
```

```
569       // that s1 and out are also merged.
570       merged_shapes_.emplace_back(s0, *out);
571     }
572     return s;
573   }
574
575   Status InferenceContext::Subshape(ShapeHandle s, int64_t start,
576                                     ShapeHandle* out) {
577     return Subshape(s, start, std::numeric_limits<int64_t>::max() /* end */, out);
578   }
579
580   Status InferenceContext::Subshape(ShapeHandle s, int64_t start, int64_t end,
581                                     ShapeHandle* out) {
582     return Subshape(s, start, end, 1 /* stride */, out);
583   }
584
585   Status InferenceContext::Subshape(ShapeHandle s, int64_t start, int64_t end,
586                                     int64_t stride, ShapeHandle* out) {
587     int64_t start_in = start;
588     int64_t end_in = end;
589
590     const int32_t rank = Rank(s);
591     if (start == 0 && stride == 1 &&
592         ((RankKnown(s) && end >= rank) ||
593          end == std::numeric_limits<int64_t>::max())) {
594       *out = s;
595       return Status::OK();
596     }
597     if (!RankKnown(s)) {
598       return ReturnUnknownShape(out);
599     }
600
601     if (start > rank) start = rank;
602     if (end > rank) end = rank;
603
604     if (stride < 0 && start == rank) --start;
605
606     if (start < 0) {
607       start = rank + start;
608       if (start < 0) {
609         *out = nullptr;
610         return errors::InvalidArgument("Subshape start out of bounds: ", start_in,
611                                        ", for shape with rank ", rank);
612       }
613     }
614
615     if (end < 0) {
616       end = rank + end;
617       if (end < 0) {
```

```
618       *out = nullptr;
619       return errors::InvalidArgument("Subshape end out of bounds: ", end_in,
620                                      ", for shape with rank ", rank);
621     }
622   }
623   if (stride > 0 && start > end) {
624     *out = nullptr;
625     return errors::InvalidArgument(
626         "Subshape must have computed start <= end, but is ", start, " and ",
627         end, " (computed from start ", start_in, " and end ", end_in,
628         " over shape with rank ", rank, ")");
629   } else if (stride < 0 && start < end) {
630     *out = nullptr;
631     return errors::InvalidArgument(
632         "Subshape must have computed start >= end since stride is negative, "
633         "but is ",
634         start, " and ", end, " (computed from start ", start_in, " and end ",
635         end_in, " over shape with rank ", rank, " and stride", stride, ")");
636   }
637
638   std::vector<DimensionHandle> dims;
639   for (int i = start; stride > 0 ? i < end : i > end; i += stride) {
640     dims.push_back(Dim(s, i));
641   }
642   return ReturnCreatedShape(dims, out);
643 }
644
645 Status InferenceContext::Concatenate(ShapeHandle s1, ShapeHandle s2,
646                                      ShapeHandle* out) {
647   if (!RankKnown(s1) || !RankKnown(s2)) {
648     return ReturnUnknownShape(out);
649   }
650   const int32_t s1_rank = Rank(s1);
651   const int32_t s2_rank = Rank(s2);
652   const int32_t rank = s1_rank + s2_rank;
653   std::vector<DimensionHandle> dims;
654   dims.reserve(rank);
655   for (int i = 0; i < s1_rank; ++i) dims.push_back(Dim(s1, i));
656   for (int i = 0; i < s2_rank; ++i) dims.push_back(Dim(s2, i));
657   return ReturnCreatedShape(dims, out);
658 }
659
660 Status InferenceContext::ReplaceDim(ShapeHandle s, int64_t dim_index_in,
661                                     DimensionHandle new_dim, ShapeHandle* out) {
662   if (!RankKnown(s)) {
663     return ReturnUnknownShape(out);
664   }
665   int64_t dim_index = dim_index_in;
666   if (dim_index < 0) {
```

```
667       dim_index = s->dims_.size() + dim_index;
668     }
669     if (!FastBoundsCheck(dim_index, s->dims_.size())) {
670       *out = nullptr;
671       return errors::InvalidArgument("Out of range dim_index ", dim_index_in,
672                                      " for shape with ", s->dims_.size(),
673                                      " dimensions");
674     }
675     std::vector<DimensionHandle> dims(s->dims_);
676     dims[dim_index] = new_dim;
677     return ReturnCreatedShape(dims, out);
678   }
679
680   ShapeHandle InferenceContext::MakeShape(
681       const std::vector<DimensionHandle>& dims) {
682     return shape_manager_.MakeShape(dims);
683   }
684
685   ShapeHandle InferenceContext::MakeShape(
686       std::initializer_list<DimensionOrConstant> dims) {
687     std::vector<DimensionHandle> dims_actual;
688     dims_actual.reserve(dims.size());
689     for (const DimensionOrConstant& d : dims) {
690       dims_actual.push_back(MakeDim(d));
691     }
692
693     return shape_manager_.MakeShape(dims_actual);
694   }
695
696   ShapeHandle InferenceContext::UnknownShape() {
697     return shape_manager_.UnknownShape();
698   }
699
700   ShapeHandle InferenceContext::UnknownShapeOfRank(int64_t rank) {
701     CHECK_LE(rank, kint32max) << "rank must be less than kint32max";
702     if (rank == kUnknownRank) {
703       return UnknownShape();
704     }
705     CHECK_GE(rank, 0) << "rank must not be negative";
706     std::vector<DimensionHandle> dims(rank);
707     for (int32_t i = 0; i < rank; ++i) {
708       dims[i] = UnknownDim();
709     }
710     return MakeShape(dims);
711   }
712
713   ShapeHandle InferenceContext::Scalar() { return MakeShape({}); }
714
715   ShapeHandle InferenceContext::Vector(DimensionOrConstant dim) {
```

```
716   return MakeShape({dim});
717 }
718
719 ShapeHandle InferenceContext::Matrix(DimensionOrConstant dim1,
720                                      DimensionOrConstant dim2) {
721   return MakeShape({dim1, dim2});
722 }
723
724 Status InferenceContext::MakeShapeFromShapeTensorTreatScalarAsUnknownShape(
725     int input_idx, ShapeHandle* out) {
726   ShapeHandle input_shape;
727   TF_RETURN_IF_ERROR(WithRankAtMost(input(input_idx), 1, &input_shape));
728
729   request_input_tensor_as_partial_shape(input_idx);
730   const int input_tensors_as_shapes_size = input_tensors_as_shapes_.size();
731   if (input_idx < input_tensors_as_shapes_size &&
732       input_tensors_as_shapes_[input_idx].IsSet() &&
733       RankKnown(input_tensors_as_shapes_[input_idx])) {
734     *out = input_tensors_as_shapes_[input_idx];
735     return Status::OK();
736   }
737
738   return InternalMakeShapeFromTensor(
739       true /* treat_unknown_scalar_tensor_as_unknown_shape */,
740       input_tensor(input_idx), input_shape, out);
741 }
742
743 Status InferenceContext::MakeShapeFromShapeTensor(int input_idx,
744                                                   ShapeHandle* out) {
745   ShapeHandle input_shape;
746   TF_RETURN_IF_ERROR(WithRank(input(input_idx), 1, &input_shape));
747
748   request_input_tensor_as_partial_shape(input_idx);
749   const int input_tensors_as_shapes_size = input_tensors_as_shapes_.size();
750   if (input_idx < input_tensors_as_shapes_size &&
751       input_tensors_as_shapes_[input_idx].IsSet() &&
752       RankKnown(input_tensors_as_shapes_[input_idx])) {
753     *out = input_tensors_as_shapes_[input_idx];
754     return Status::OK();
755   }
756
757   return InternalMakeShapeFromTensor(
758       false /* treat_unknown_scalar_tensor_as_unknown_shape */,
759       input_tensor(input_idx), input_shape, out);
760 }
761
762 Status InferenceContext::MakeShapeFromTensor(const Tensor* t,
763                                              ShapeHandle tensor_shape,
764                                              ShapeHandle* out) {
```

```
765   return InternalMakeShapeFromTensor(
766       false /* treat_unknown_scalar_tensor_as_unknown_shape */, t, tensor_shape,
767       out);
768 }
769
770 Status InferenceContext::InternalMakeShapeFromTensor(
771     bool treat_unknown_scalar_tensor_as_unknown_shape, const Tensor* t,
772     ShapeHandle tensor_shape, ShapeHandle* out) {
773   // Only callers who have set
774   if (!treat_unknown_scalar_tensor_as_unknown_shape) {
775     TF_RETURN_IF_ERROR(WithRank(tensor_shape, 1, &tensor_shape));
776   }
777   if (t == nullptr) {
778     // This is guarded by the check above.
779     if (Rank(tensor_shape) == 0) {
780       return ReturnUnknownShape(out);
781     }
782     // Shape tensor is not known, but if the shape of the shape tensor is then
783     // the right number of unknown dims can be created.
784     DimensionHandle shape_dim = Dim(tensor_shape, 0);
785     if (!ValueKnown(shape_dim)) {
786       return ReturnUnknownShape(out);
787     }
788     const auto num_dims = Value(shape_dim);
789     std::vector<DimensionHandle> dims;
790     dims.reserve(num_dims);
791     for (int i = 0; i < num_dims; i++) dims.push_back(UnknownDim());
792     return ReturnCreatedShape(dims, out);
793   }
794
795   if (t->shape().dims() == 0) {
796     if (t->dtype() == DataType::DT_INT32) {
797       auto flat_t = t->scalar<int32>();
798       if (flat_t() != -1) {
799         *out = nullptr;
800         return errors::InvalidArgument(
801             "Input tensor must be rank 1, or if its rank 0 it must have value "
802             "-1 "
803             "(representing an unknown shape).  Saw value: ",
804             flat_t());
805       }
806       return ReturnUnknownShape(out);
807     } else if (t->dtype() == DataType::DT_INT64) {
808       auto flat_t = t->scalar<int64_t>();
809       if (flat_t() != -1) {
810         *out = nullptr;
811         return errors::InvalidArgument(
812             "Input tensor must be rank 1, or if its rank 0 it must have value "
813             "-1 "
```

```
814             "(representing an unknown shape).  Saw value: ",
815             flat_t());
816       }
817       return ReturnUnknownShape(out);
818     } else {
819       *out = nullptr;
820       return errors::InvalidArgument(
821           "Input tensor must be int32 or int64, but was ",
822           DataTypeString(t->dtype()));
823     }
824   }
825
826   if (t->shape().dims() != 1) {
827     *out = nullptr;
828     return errors::InvalidArgument(
829         "Input tensor must be rank 1, but was rank ", t->shape().dims(), ".",
830         ((t->shape().dims() == 0)
831              ? "If it is rank 0 rank 0 it must have statically known value -1 "
832                "(representing an unknown shape). "
833              : " "),
834         "Saw tensor shape ", t->shape().DebugString());
835   }
836   std::vector<DimensionHandle> dims;
837   if (t->dtype() == DataType::DT_INT32) {
838     auto flat_t = t->flat<int32>();
839     for (int i = 0; i < flat_t.size(); ++i) {
840       const int32_t val = flat_t(i);
841       if (val < -1) {
842         return errors::InvalidArgument(
843             "Invalid value in tensor used for shape: ", val);
844       }
845       // -1 will become an unknown dim.
846       dims.push_back(MakeDim(val));
847     }
848   } else if (t->dtype() == DataType::DT_INT64) {
849     auto flat_t = t->flat<int64_t>();
850     for (int i = 0; i < flat_t.size(); ++i) {
851       const int64_t val = flat_t(i);
852       if (val < -1) {
853         return errors::InvalidArgument(
854             "Invalid value in tensor used for shape: ", val);
855       }
856       // -1 will become an unknown dim.
857       dims.push_back(MakeDim(val));
858     }
859   } else {
860     *out = nullptr;
861     return errors::InvalidArgument(
862         "Input tensor must be int32 or int64, but was ",
```

```
863         DataTypeString(t->dtype()));
864   }
865
866   return ReturnCreatedShape(dims, out);
867 }
868
869 Status InferenceContext::MakeShapeFromPartialTensorShape(
870     const PartialTensorShape& partial_shape, ShapeHandle* out) {
871   *out = nullptr;
872   if (partial_shape.dims() == -1) {
873     return ReturnUnknownShape(out);
874   }
875   const int num_dims = partial_shape.dims();
876   std::vector<DimensionHandle> dims(num_dims);
877   for (int i = 0; i < num_dims; ++i) {
878     // -1 is unknown in PartialTensorShape and in InferenceContext, so this size
879     // can be passed directly to MakeDim.
880     dims[i] = MakeDim(partial_shape.dim_size(i));
881   }
882   return ReturnCreatedShape(dims, out);
883 }
884
885 Status InferenceContext::MakeShapeFromTensorShape(const TensorShape& shape,
886                                                   ShapeHandle* out) {
887   return MakeShapeFromPartialTensorShape(PartialTensorShape(shape.dim_sizes()),
888                                          out);
889 }
890
891 Status InferenceContext::MakeShapeFromShapeProto(const TensorShapeProto& proto,
892                                                  ShapeHandle* out) {
893   *out = nullptr;
894   TF_RETURN_IF_ERROR(PartialTensorShape::IsValidShape(proto));
895   PartialTensorShape partial_shape(proto);
896   return MakeShapeFromPartialTensorShape(partial_shape, out);
897 }
898
899 Status InferenceContext::GetScalarFromTensor(const Tensor* t, int64_t* val) {
900   // Caller must ensure that <t> is not NULL.
901   const int rank = t->dims();
902   if (rank != 0) {
903     return errors::InvalidArgument("Input must be scalar but has rank ", rank);
904   }
905
906   if (t->dtype() == DataType::DT_INT32) {
907     *val = t->scalar<int32>()();
908     return Status::OK();
909   } else if (t->dtype() == DataType::DT_INT64) {
910     *val = t->scalar<int64_t>()();
911     return Status::OK();
```

```
912     } else {
913       return errors::InvalidArgument("Scalar input must be int32 or int64.");
914     }
915   }
916
917   Status InferenceContext::GetScalarFromTensor(const Tensor* t, int64_t idx,
918                                                int64_t* val) {
919     // Caller must ensure that <t> is not NULL.
920     const int rank = t->dims();
921     if (rank != 1) {
922       return errors::InvalidArgument("Input must be 1D but has rank ", rank);
923     }
924
925     if (t->dtype() == DataType::DT_INT32) {
926       auto flat_t = t->flat<int32>();
927       if (idx < 0 || idx >= flat_t.size()) {
928         return errors::InvalidArgument("Invalid index ", idx,
929                                        " for Tensor of size ", flat_t.size());
930       }
931       *val = flat_t(idx);
932       return Status::OK();
933     } else if (t->dtype() == DataType::DT_INT64) {
934       auto flat_t = t->flat<int64_t>();
935       if (idx < 0 || idx >= flat_t.size()) {
936         return errors::InvalidArgument("Invalid index ", idx,
937                                        " for Tensor of size ", flat_t.size());
938       }
939       *val = flat_t(idx);
940       return Status::OK();
941     } else {
942       return errors::InvalidArgument("Tensor input must be int32 or int64.");
943     }
944   }
945
946   // Returns a new dimension whose value is given by a scalar input tensor.
947   Status InferenceContext::MakeDimForScalarInput(int idx, DimensionHandle* out) {
948     int64_t val;
949     const Tensor* t = input_tensor(idx);
950     if (t == nullptr) {
951       *out = UnknownDim();
952       return Status::OK();
953     }
954     TF_RETURN_IF_ERROR(GetScalarFromTensor(t, &val));
955     if (val < 0) {
956       return errors::InvalidArgument("Dimension size, given by scalar input ",
957                                      idx, ", must be non-negative but is ", val);
958     }
959     *out = MakeDim(val);
960     return Status::OK();
```

```
961 }
962
963 Status InferenceContext::MakeDimForScalarInputWithNegativeIndexing(
964     int idx, int input_rank, DimensionHandle* out) {
965   int64_t val;
966   const Tensor* t = input_tensor(idx);
967   if (t == nullptr) {
968     *out = UnknownDim();
969     return Status::OK();
970   }
971   TF_RETURN_IF_ERROR(GetScalarFromTensor(t, &val));
972   if (val < 0) {
973     if (input_rank < 0) {
974       *out = UnknownDim();
975       return Status::OK();
976     } else if (val + input_rank < 0) {
977       return errors::InvalidArgument("Dimension size, given by scalar input ",
978                                      val, " must be in range [-", input_rank,
979                                      ", ", input_rank, ")");
980     } else {
981       val += input_rank;
982     }
983   } else if (input_rank >= 0 && val >= input_rank) {
984     return errors::InvalidArgument("Dimension size, given by scalar input ",
985                                    val, " must be in range [-", input_rank,
986                                    ", ", input_rank, ")");
987   }
988   *out = MakeDim(val);
989   return Status::OK();
990 }
991
992 Status InferenceContext::Divide(DimensionHandle dividend,
993                                 DimensionOrConstant divisor,
994                                 bool evenly_divisible, DimensionHandle* out) {
995   const int64_t divisor_value = Value(divisor);
996   if (divisor_value == 1) {
997     *out = dividend;
998   } else if (!ValueKnown(dividend) ||
999              (divisor.dim.IsSet() && !ValueKnown(divisor.dim))) {
1000    *out = UnknownDim();
1001  } else {
1002    const int64_t v = Value(dividend);
1003    if (divisor_value <= 0) {
1004      return errors::InvalidArgument("Divisor must be positive but is ",
1005                                     divisor_value);
1006    }
1007    if (evenly_divisible && (v % divisor_value) != 0) {
1008      return errors::InvalidArgument(
1009          "Dimension size must be evenly divisible by ", divisor_value,
```

```
1010            " but is ", v);
1011      }
1012      *out = MakeDim(v / divisor_value);
1013    }
1014    return Status::OK();
1015  }
1016
1017  Status InferenceContext::Add(DimensionHandle first, DimensionOrConstant second,
1018                               DimensionHandle* out) {
1019    const int64_t first_value = Value(first);
1020    const int64_t second_value = Value(second);
1021    // Special cases.
1022    if (first_value == 0) {
1023      *out = MakeDim(second);
1024    } else if (second_value == 0) {
1025      *out = first;
1026    } else if (first_value == kUnknownDim || second_value == kUnknownDim) {
1027      *out = UnknownDim();
1028    } else {
1029      // Invariant: Both values are known and positive. Still in run-time we can
1030      // get pair of values which cannot be store in output. Check below will
1031      // report error. We still need to avoid undefined behavior of signed
1032      // overflow and use unsigned addition.
1033      const int64_t sum = static_cast<uint64>(first_value) + second_value;
1034      if (sum < 0) {
1035        return errors::InvalidArgument("Dimension size overflow from adding ",
1036                                       first_value, " and ", second_value);
1037      }
1038      *out = MakeDim(sum);
1039    }
1040    return Status::OK();
1041  }
1042
1043  Status InferenceContext::Subtract(DimensionHandle first,
1044                                    DimensionOrConstant second,
1045                                    DimensionHandle* out) {
1046    const int64_t first_value = Value(first);
1047    const int64_t second_value = Value(second);
1048    // Special cases.
1049    if (second_value == 0) {
1050      *out = first;
1051    } else if (first_value == kUnknownDim || second_value == kUnknownDim) {
1052      *out = UnknownDim();
1053    } else {
1054      // Invariant: Both values are known, first_value is non-negative, and
1055      // second_value is positive.
1056      if (first_value < second_value) {
1057        return errors::InvalidArgument(
1058            "Negative dimension size caused by subtracting ", second_value,
```

```
1059            " from ", first_value);
1060      }
1061      *out = MakeDim(first_value - second_value);
1062    }
1063    return Status::OK();
1064  }
1065
1066  Status InferenceContext::Multiply(DimensionHandle first,
1067                                    DimensionOrConstant second,
1068                                    DimensionHandle* out) {
1069    const int64_t first_value = Value(first);
1070    const int64_t second_value = Value(second);
1071    // Special cases.
1072    if (first_value == 0) {
1073      *out = first;
1074    } else if (second_value == 0) {
1075      *out = MakeDim(second);
1076    } else if (first_value == 1) {
1077      *out = MakeDim(second);
1078    } else if (second_value == 1) {
1079      *out = first;
1080    } else if (first_value == kUnknownDim || second_value == kUnknownDim) {
1081      *out = UnknownDim();
1082    } else {
1083      // Invariant: Both values are known and greater than 1.
1084      const int64_t product = first_value * second_value;
1085      if (product < 0) {
1086        return errors::InvalidArgument(
1087            "Negative dimension size caused by overflow when multiplying ",
1088            first_value, " and ", second_value);
1089      }
1090      *out = MakeDim(product);
1091    }
1092    return Status::OK();
1093  }
1094
1095  Status InferenceContext::Min(DimensionHandle first, DimensionOrConstant second,
1096                               DimensionHandle* out) {
1097    const int64_t first_value = Value(first);
1098    const int64_t second_value = Value(second);
1099    if (first_value == 0) {
1100      *out = first;
1101    } else if (second_value == 0) {
1102      *out = MakeDim(second);
1103    } else if (first_value == kUnknownDim || second_value == kUnknownDim) {
1104      *out = UnknownDim();
1105    } else {
1106      if (first_value <= second_value) {
1107        *out = first;
```

```
    } else {
      *out = MakeDim(second);
    }
  }
  return Status::OK();
}

Status InferenceContext::Max(DimensionHandle first, DimensionOrConstant second,
                             DimensionHandle* out) {
  const int64_t first_value = Value(first);
  const int64_t second_value = Value(second);
  if (first_value == kUnknownDim || second_value == kUnknownDim) {
    *out = UnknownDim();
  } else {
    if (first_value >= second_value) {
      *out = first;
    } else {
      *out = MakeDim(second);
    }
  }
  return Status::OK();
}

Status InferenceContext::AttachContext(const Status& status) {
  std::vector<string> input_shapes;
  input_shapes.reserve(inputs_.size());
  for (const ShapeHandle& input_shape : inputs_) {
    input_shapes.emplace_back(DebugString(input_shape));
  }

  // Add information about the input tensors and partial tensor shapes used.
  std::vector<string> input_from_tensors_str;
  std::vector<string> input_from_tensors_as_shape_str;
  input_from_tensors_as_shape_str.reserve(inputs_.size());
  for (int i = 0, end = inputs_.size(); i < end; ++i) {
    const int input_tensors_as_shapes_size = input_tensors_as_shapes_.size();
    const int input_tensors_size = input_tensors_.size();
    if (requested_input_tensor_as_partial_shape_[i] &&
        i < input_tensors_as_shapes_size &&
        input_tensors_as_shapes_[i].IsSet() &&
        RankKnown(input_tensors_as_shapes_[i])) {
      input_from_tensors_as_shape_str.push_back(strings::StrCat(
          "input[", i, "] = ", DebugString(input_tensors_as_shapes_[i])));
    } else if (requested_input_tensor_[i] && i < input_tensors_size &&
               input_tensors_[i] != nullptr) {
      input_from_tensors_str.push_back(strings::StrCat(
          "input[", i, "] = <",
          input_tensors_[i]->SummarizeValue(256 /* max_values */), ">"));
    }
```

```
1157    }
1158
1159    string error_context = strings::StrCat(
1160        " for '", attrs_.SummarizeNode(),
1161        "' with input shapes: ", absl::StrJoin(input_shapes, ", "));
1162    if (!input_from_tensors_str.empty()) {
1163      strings::StrAppend(&error_context, " and with computed input tensors: ",
1164                         absl::StrJoin(input_from_tensors_str, ", "));
1165    }
1166    if (!input_from_tensors_as_shape_str.empty()) {
1167      strings::StrAppend(&error_context,
1168                         " and with input tensors computed as partial shapes: ",
1169                         absl::StrJoin(input_from_tensors_as_shape_str, ","));
1170    }
1171
1172    strings::StrAppend(&error_context, ".");
1173    return errors::CreateWithUpdatedMessage(
1174        status, strings::StrCat(status.error_message(), error_context));
1175  }
1176
1177  bool InferenceContext::MergeHandleShapesAndTypes(
1178      const std::vector<ShapeAndType>& shapes_and_types,
1179      std::vector<ShapeAndType>* to_update) {
1180    if (shapes_and_types.size() != to_update->size()) {
1181      return false;
1182    }
1183    std::vector<ShapeAndType> new_values(shapes_and_types.size());
1184    bool refined = false;
1185    for (int i = 0, end = shapes_and_types.size(); i < end; ++i) {
1186      const ShapeAndType& existing = (*to_update)[i];
1187      if (shapes_and_types[i].dtype == existing.dtype) {
1188        new_values[i].dtype = existing.dtype;
1189      } else {
1190        if (existing.dtype != DT_INVALID) {
1191          return false;
1192        } else {
1193          new_values[i].dtype = shapes_and_types[i].dtype;
1194          refined = true;
1195        }
1196      }
1197      if (!Merge(existing.shape, shapes_and_types[i].shape, &new_values[i].shape)
1198                .ok()) {
1199        // merge failed, ignore the new value.
1200        new_values[i].shape = existing.shape;
1201      }
1202      if (!existing.shape.SameHandle(new_values[i].shape)) {
1203        refined = true;
1204      }
1205    }
```

```
1206    if (!refined) {
1207      return false;
1208    }
1209    for (int i = 0, end = new_values.size(); i < end; ++i) {
1210      (*to_update)[i] = new_values[i];
1211    }
1212    return true;
1213  }
1214
1215  bool InferenceContext::MergeOutputHandleShapesAndTypes(
1216      int idx, const std::vector<ShapeAndType>& shapes_and_types) {
1217    if (output_handle_shapes_and_types_[idx] == nullptr) {
1218      output_handle_shapes_and_types_[idx].reset(
1219          new std::vector<ShapeAndType>(shapes_and_types));
1220      return true;
1221    }
1222    return MergeHandleShapesAndTypes(shapes_and_types,
1223                                     output_handle_shapes_and_types_[idx].get());
1224  }
1225
1226  bool InferenceContext::MergeInputHandleShapesAndTypes(
1227      int idx, const std::vector<ShapeAndType>& shapes_and_types) {
1228    if (input_handle_shapes_and_types_[idx] == nullptr) {
1229      input_handle_shapes_and_types_[idx].reset(
1230          new std::vector<ShapeAndType>(shapes_and_types));
1231      return true;
1232    }
1233    return MergeHandleShapesAndTypes(shapes_and_types,
1234                                     input_handle_shapes_and_types_[idx].get());
1235  }
1236
1237  bool InferenceContext::RelaxHandleShapesAndMergeTypes(
1238      const std::vector<ShapeAndType>& shapes_and_types,
1239      std::vector<ShapeAndType>* to_update) {
1240    if (shapes_and_types.size() != to_update->size()) {
1241      return false;
1242    }
1243    std::vector<ShapeAndType> new_values(shapes_and_types.size());
1244    for (int i = 0, end = shapes_and_types.size(); i < end; ++i) {
1245      const ShapeAndType& existing = (*to_update)[i];
1246      if (shapes_and_types[i].dtype == existing.dtype) {
1247        new_values[i].dtype = existing.dtype;
1248      } else {
1249        if (existing.dtype != DT_INVALID) {
1250          return false;
1251        } else {
1252          new_values[i].dtype = shapes_and_types[i].dtype;
1253        }
1254      }
```

```
1255      Relax(existing.shape, shapes_and_types[i].shape, &new_values[i].shape);
1256    }
1257    to_update->swap(new_values);
1258    return true;
1259  }
1260
1261  bool InferenceContext::RelaxOutputHandleShapesAndMergeTypes(
1262      int idx, const std::vector<ShapeAndType>& shapes_and_types) {
1263    if (output_handle_shapes_and_types_[idx] == nullptr) {
1264      output_handle_shapes_and_types_[idx].reset(
1265          new std::vector<ShapeAndType>(shapes_and_types));
1266      return true;
1267    }
1268    return RelaxHandleShapesAndMergeTypes(
1269        shapes_and_types, output_handle_shapes_and_types_[idx].get());
1270  }
1271
1272  bool InferenceContext::RelaxInputHandleShapesAndMergeTypes(
1273      int idx, const std::vector<ShapeAndType>& shapes_and_types) {
1274    if (input_handle_shapes_and_types_[idx] == nullptr) {
1275      input_handle_shapes_and_types_[idx].reset(
1276          new std::vector<ShapeAndType>(shapes_and_types));
1277      return true;
1278    }
1279    return RelaxHandleShapesAndMergeTypes(
1280        shapes_and_types, input_handle_shapes_and_types_[idx].get());
1281  }
1282
1283  // ----------------------------------------------------------------------
1284  // ShapeManager
1285  // ----------------------------------------------------------------------
1286  InferenceContext::ShapeManager::ShapeManager() {}
1287  InferenceContext::ShapeManager::~ShapeManager() {
1288    for (auto* s : all_shapes_) delete s;
1289    for (auto* d : all_dims_) delete d;
1290  }
1291
1292  ShapeHandle InferenceContext::ShapeManager::MakeShape(
1293      const std::vector<DimensionHandle>& dims) {
1294    all_shapes_.push_back(new Shape(dims));
1295    return all_shapes_.back();
1296  }
1297
1298  ShapeHandle InferenceContext::ShapeManager::UnknownShape() {
1299    all_shapes_.push_back(new Shape());
1300    return all_shapes_.back();
1301  }
1302
1303  }  // namespace shape_inference
```

```
1304 }  // namespace tensorflow
```