mihaimaruseac Fix abort caused by allocating a too large vector. ... ✓ History

22 contributors +10

1322 lines (1209 sloc) | 44 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 <cstdint>
18
19 #include "tensorflow/core/framework/bounds_check.h"
20 #include "tensorflow/core/framework/full_type_util.h"
21 #include "tensorflow/core/framework/node_def.pb.h"
22 #include "tensorflow/core/framework/op_def.pb.h"
23 #include "tensorflow/core/framework/partial_tensor_shape.h"
24 #include "tensorflow/core/framework/tensor_shape.pb.h"
25 #include "tensorflow/core/lib/core/errors.h"
26 #include "tensorflow/core/lib/strings/numbers.h"
27 #include "tensorflow/core/lib/strings/scanner.h"
28 #include "tensorflow/core/lib/strings/str_util.h"
29
```

```
namespace tensorflow {
namespace shape_inference {

constexpr int32_t InferenceContext::kUnknownRank;
constexpr int64_t InferenceContext::kUnknownDim;

// Same as above, but with PartialTensorShape instead of TensorShapeProto
InferenceContext::InferenceContext(
    int graph_def_version, const AttrSlice& attrs, const OpDef& op_def,
    const std::vector<PartialTensorShape>& input_shapes,
    const std::vector<const Tensor*>& input_tensors,
    const std::vector<PartialTensorShape>& input_tensors_as_shapes,
    const std::vector<
        std::unique_ptr<std::vector<std::pair<PartialTensorShape, DataType>>>>&
        input_handle_shapes_and_types)
    : graph_def_version_(graph_def_version), attrs_(attrs) {
  std::vector<ShapeHandle> input_tensors_as_shape_handles;
  input_tensors_as_shape_handles.reserve(input_tensors_as_shapes.size());
  for (const PartialTensorShape& p : input_tensors_as_shapes) {
    ShapeHandle shape;
    construction_status_.Update(MakeShapeFromPartialTensorShape(p, &shape));
    if (!construction_status_.ok()) {
      return;
    }
    input_tensors_as_shape_handles.push_back(shape);
  }
  PreInputInit(op_def, input_tensors, input_tensors_as_shape_handles);
  if (!construction_status_.ok()) return;
  inputs_.reserve(input_shapes.size());
  for (const PartialTensorShape& p : input_shapes) {
    ShapeHandle shape;
    construction_status_.Update(MakeShapeFromPartialTensorShape(p, &shape));
    if (!construction_status_.ok()) {
      return;
    }
    inputs_.push_back(shape);
  }
  std::vector<std::unique_ptr<std::vector<ShapeAndType>>> handle_data(
      input_shapes.size());
  for (int i = 0, end = input_handle_shapes_and_types.size(); i < end; ++i) {
    const auto& v = input_handle_shapes_and_types[i];
    if (v == nullptr) {
      continue;
    }
    handle_data[i].reset(new std::vector<ShapeAndType>(v->size()));
    auto& new_v = *handle_data[i];
    for (int j = 0, end = v->size(); j < end; ++j) {
      const auto& p = (*v)[j];
      construction_status_.Update(
```

```
          MakeShapeFromPartialTensorShape(p.first, &new_v[j].shape));
      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;
    const int shapes_size = shapes.size();
    if (size != shapes_size) {
      return errors::InvalidArgument("Must have exactly ", shapes.size(),
                                     " shapes.");
    }
    for (int i = 0; i < shapes_size; ++i) {
      outputs_[i + start] = shapes[i];
    }
  }
  return Status::OK();
}

Status InferenceContext::input(StringPiece input_name,
                               std::vector<ShapeHandle>* output) const {
  const auto result = input_name_map_.find(input_name);
  if (result == input_name_map_.end()) {
    return errors::InvalidArgument("Unknown input name: ", input_name);
  } else {
    output->clear();
    for (int i = result->second.first; i < result->second.second; ++i) {
      output->push_back(inputs_[i]);
    }
  }
  return Status::OK();
}

Status InferenceContext::output(StringPiece output_name,
                                std::vector<ShapeHandle>* output) const {
  const auto result = output_name_map_.find(output_name);
  if (result == output_name_map_.end()) {
    return errors::InvalidArgument("Unknown output name: ", output_name);
  } else {
    output->clear();
    for (int i = result->second.first; i < result->second.second; ++i) {
      output->push_back(outputs_[i]);
    }
  }
  return Status::OK();
}

void InferenceContext::PreInputInit(
    const OpDef& op_def, const std::vector<const Tensor*>& input_tensors,
    const std::vector<ShapeHandle>& input_tensors_as_shapes) {
  // TODO(mdan): This is also done at graph construction. Run only here instead?
  const auto ret = full_type::SpecializeType(attrs_, op_def);
  if (!ret.status().ok()) {
    construction_status_ = ret.status();
```

```
    return;
  }
  ret_types_ = ret.ValueOrDie();

  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");
    return;
  }

  CHECK_LE(input_tensors_.size(), inputs_.size());
  input_tensors_.resize(inputs_.size());
  requested_input_tensor_.resize(inputs_.size());
  requested_input_tensor_as_partial_shape_.resize(inputs_.size());
}

void InferenceContext::ShapeHandleToProto(ShapeHandle handle,
                                          TensorShapeProto* proto) {
  if (!RankKnown(handle)) {
    proto->set_unknown_rank(true);
    return;
  }

  for (int32_t i = 0; i < Rank(handle); ++i) {
    DimensionHandle dim = Dim(handle, i);
    auto* dim_shape = proto->add_dim();
    if (ValueKnown(dim)) {
      dim_shape->set_size(Value(dim));
    } else {
      dim_shape->set_size(-1);
    }
  }
}

bool InferenceContext::FullyDefined(ShapeHandle s) {
  if (!RankKnown(s)) return false;
  for (int i = 0; i < Rank(s); ++i) {
    if (!ValueKnown(Dim(s, i))) return false;
  }
  return true;
}

DimensionHandle InferenceContext::NumElements(ShapeHandle s) {
  const auto rank = Rank(s);
  if (rank == kUnknownRank) return UnknownDim();
  bool found_unknown = false;
  int64_t size = 1;
  for (int i = 0; i < rank; ++i) {
    int64_t dim_val = Value(Dim(s, i));
    if (dim_val == kUnknownDim) {
      found_unknown = true;
```

```
    } else if (dim_val == 0) {
      return MakeDim(0);
    } else {
      size *= dim_val;
    }
  }
  if (found_unknown) {
    return UnknownDim();
  } else {
    return MakeDim(size);
  }
}

string InferenceContext::DebugString(ShapeHandle s) {
  if (RankKnown(s)) {
    std::vector<string> vals;
    for (auto d : s->dims_) vals.push_back(DebugString(d));
    return strings::StrCat("[", absl::StrJoin(vals, ","), "]");
  } else {
    return "?";
  }
}

string InferenceContext::DebugString(DimensionHandle d) {
  return ValueKnown(d) ? strings::StrCat(Value(d)) : "?";
}

string InferenceContext::DebugString() const {
  return strings::StrCat("InferenceContext for node: ", attrs_.SummarizeNode());
}

string InferenceContext::DebugString(const ShapeAndType& shape_and_type) {
  return strings::StrCat(DebugString(shape_and_type.shape), ":",
                         DataTypeString(shape_and_type.dtype));
}

string InferenceContext::DebugString(
    gtl::ArraySlice<ShapeAndType> shape_and_types) {
  std::vector<string> pieces;
  for (const ShapeAndType& s : shape_and_types) {
    pieces.push_back(DebugString(s));
  }
  return strings::StrCat("[", absl::StrJoin(pieces, ","), "]");
}

Status InferenceContext::WithRank(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) {
    *out = shape;
    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);
  if (existing == value) {
    *out = dim;
    return Status::OK();
  }
  if (existing == kUnknownDim) {
    DimensionHandle d = MakeDim(value);
    return Merge(dim, d, out);
  }
  *out = nullptr;
  return errors::InvalidArgument("Dimension must be ", value, " but is ",
                                 existing);
}

void InferenceContext::Relax(DimensionHandle d_old, DimensionHandle d_new,
                             DimensionHandle* out) {
  if (d_old.SameHandle(d_new)) {
    *out = d_old;
  } else if (!ValueKnown(d_old) && !ValueKnown(d_new)) {
    // The node will be fed by the dimension d_new instead of d_old: any
    // equality assertion between d_old and other input dimension on this node
    // may not be true anymore, so forget them all.
    ForgetMerges();
    // Return the new shape handle to force the relaxation to propagate to the
    // fanout of the context.
    *out = d_new;
  } else if (!ValueKnown(d_new)) {
    ForgetMerges();
    *out = d_new;
  } else if (Value(d_old) == Value(d_new)) {
    // Return the old shape handle. This will stop the relaxation in the fanout
    // of the context.
    *out = d_old;
  } else {
    // Return a new handle that encodes a different unknown dim.
    ForgetMerges();
    *out = UnknownDim();
  }
}

Status InferenceContext::Merge(DimensionHandle d0, DimensionHandle d1,
                               DimensionHandle* out) {
  if (d0.SameHandle(d1)) {
    *out = d0;
    return Status::OK();
```

```
  } else if (!ValueKnown(d1)) {
    *out = d0;
    merged_dims_.emplace_back(d0, d1);
    return Status::OK();
  } else if (!ValueKnown(d0)) {
    *out = d1;
    merged_dims_.emplace_back(d0, d1);
    return Status::OK();
  } else if (Value(d0) == Value(d1)) {
    *out = d0;
    return Status::OK();
  } else {
    *out = nullptr;
    return errors::InvalidArgument("Dimensions must be equal, but are ",
                                   Value(d0), " and ", Value(d1));
  }
}

Status InferenceContext::MergePrefix(ShapeHandle s, ShapeHandle prefix,
                                     ShapeHandle* s_out,
                                     ShapeHandle* prefix_out) {
  *s_out = *prefix_out = nullptr;
  if (!RankKnown(prefix) || !RankKnown(s)) {
    *s_out = s;
    *prefix_out = prefix;
    return Status::OK();
  }
  const int32_t rank = Rank(prefix);
  TF_RETURN_IF_ERROR(WithRankAtLeast(s, rank, &s));

  // Merge the prefix dims and create the new output shapes.
  const int32_t rank_s = Rank(s);
  std::vector<DimensionHandle> dims;
  dims.reserve(std::max(rank, rank_s));
  dims.resize(rank);
  for (int i = 0; i < rank; ++i) {
    TF_RETURN_IF_ERROR(Merge(Dim(s, i), Dim(prefix, i), &dims[i]));
  }
  *prefix_out = MakeShape(dims);
  for (int i = rank; i < rank_s; ++i) dims.push_back(Dim(s, i));
  *s_out = MakeShape(dims);
  return Status::OK();
}

void InferenceContext::Relax(ShapeHandle s_old, ShapeHandle s_new,
                             ShapeHandle* out) {
  if (s_old.SameHandle(s_new)) {
    *out = s_old;
    return;
```

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

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

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

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

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

```
}

ShapeHandle InferenceContext::Scalar() { return MakeShape({}); }

ShapeHandle InferenceContext::Vector(DimensionOrConstant dim) {
  return MakeShape({dim});
}

ShapeHandle InferenceContext::Matrix(DimensionOrConstant dim1,
                                     DimensionOrConstant dim2) {
  return MakeShape({dim1, dim2});
}

Status InferenceContext::MakeShapeFromShapeTensorTreatScalarAsUnknownShape(
    int input_idx, ShapeHandle* out) {
  ShapeHandle input_shape;
  TF_RETURN_IF_ERROR(WithRankAtMost(input(input_idx), 1, &input_shape));

  request_input_tensor_as_partial_shape(input_idx);
  const int input_tensors_as_shapes_size = input_tensors_as_shapes_.size();
  if (input_idx < input_tensors_as_shapes_size &&
      input_tensors_as_shapes_[input_idx].IsSet() &&
      RankKnown(input_tensors_as_shapes_[input_idx])) {
    *out = input_tensors_as_shapes_[input_idx];
    return Status::OK();
  }

  return InternalMakeShapeFromTensor(
      true /* treat_unknown_scalar_tensor_as_unknown_shape */,
      input_tensor(input_idx), input_shape, out);
}

Status InferenceContext::MakeShapeFromShapeTensor(int input_idx,
                                                  ShapeHandle* out) {
  ShapeHandle input_shape;
  TF_RETURN_IF_ERROR(WithRank(input(input_idx), 1, &input_shape));

  request_input_tensor_as_partial_shape(input_idx);
  const int input_tensors_as_shapes_size = input_tensors_as_shapes_.size();
  if (input_idx < input_tensors_as_shapes_size &&
      input_tensors_as_shapes_[input_idx].IsSet() &&
      RankKnown(input_tensors_as_shapes_[input_idx])) {
    *out = input_tensors_as_shapes_[input_idx];
    return Status::OK();
  }

  return InternalMakeShapeFromTensor(
      false /* treat_unknown_scalar_tensor_as_unknown_shape */,
      input_tensor(input_idx), input_shape, out);
```

```
765 }
766
767 Status InferenceContext::MakeShapeFromTensor(const Tensor* t,
768                                              ShapeHandle tensor_shape,
769                                              ShapeHandle* out) {
770   return InternalMakeShapeFromTensor(
771       false /* treat_unknown_scalar_tensor_as_unknown_shape */, t, tensor_shape,
772       out);
773 }
774
775 Status InferenceContext::InternalMakeShapeFromTensor(
776     bool treat_unknown_scalar_tensor_as_unknown_shape, const Tensor* t,
777     ShapeHandle tensor_shape, ShapeHandle* out) {
778   // Only callers who have set
779   if (!treat_unknown_scalar_tensor_as_unknown_shape) {
780     TF_RETURN_IF_ERROR(WithRank(tensor_shape, 1, &tensor_shape));
781   }
782   if (t == nullptr) {
783     // This is guarded by the check above.
784     if (Rank(tensor_shape) == 0) {
785       return ReturnUnknownShape(out);
786     }
787     // Shape tensor is not known, but if the shape of the shape tensor is then
788     // the right number of unknown dims can be created.
789     DimensionHandle shape_dim = Dim(tensor_shape, 0);
790     if (!ValueKnown(shape_dim)) {
791       return ReturnUnknownShape(out);
792     }
793     const auto num_dims = Value(shape_dim);
794     // TODO(mihaimaruseac): Should be `TensorShape::MaxDimensions()` as we are
795     // not able to materialize shapes with more than this number of dimensions
796     // but then shape inference would fail for operations such as
797     // `tf.range`/`tf.ones`, etc. where the shape is not really materialized,
798     // only used during the inference. Hence, just prevent doing a `reserve`
799     // with a very large argument.
800     const int64_t max_dimensions = 1 << 20;
801     if (num_dims >= max_dimensions) {
802       return errors::Internal(
803           "Cannot create a tensor with ", num_dims,
804           " dimensions, as these would be more than maximum of ",
805           max_dimensions);
806     }
807     std::vector<DimensionHandle> dims;
808     dims.reserve(num_dims);
809     for (int i = 0; i < num_dims; i++) dims.push_back(UnknownDim());
810     return ReturnCreatedShape(dims, out);
811   }
812
813   if (t->shape().dims() == 0) {
```

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

```
      // -1 will become an unknown dim.
      dims.push_back(MakeDim(val));
    }
  } else if (t->dtype() == DataType::DT_INT64) {
    auto flat_t = t->flat<int64_t>();
    for (int i = 0; i < flat_t.size(); ++i) {
      const int64_t val = flat_t(i);
      if (val < -1) {
        return errors::InvalidArgument(
            "Invalid value in tensor used for shape: ", val);
      }
      // -1 will become an unknown dim.
      dims.push_back(MakeDim(val));
    }
  } else {
    *out = nullptr;
    return errors::InvalidArgument(
        "Input tensor must be int32 or int64, but was ",
        DataTypeString(t->dtype()));
  }

  return ReturnCreatedShape(dims, out);
}

Status InferenceContext::MakeShapeFromPartialTensorShape(
    const PartialTensorShape& partial_shape, ShapeHandle* out) {
  *out = nullptr;
  if (partial_shape.dims() == -1) {
    return ReturnUnknownShape(out);
  }
  const int num_dims = partial_shape.dims();
  std::vector<DimensionHandle> dims(num_dims);
  for (int i = 0; i < num_dims; ++i) {
    // -1 is unknown in PartialTensorShape and in InferenceContext, so this size
    // can be passed directly to MakeDim.
    dims[i] = MakeDim(partial_shape.dim_size(i));
  }
  return ReturnCreatedShape(dims, out);
}

Status InferenceContext::MakeShapeFromTensorShape(const TensorShape& shape,
                                                  ShapeHandle* out) {
  return MakeShapeFromPartialTensorShape(PartialTensorShape(shape.dim_sizes()),
                                         out);
}

Status InferenceContext::MakeShapeFromShapeProto(const TensorShapeProto& proto,
                                                 ShapeHandle* out) {
  *out = nullptr;
```

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

```
961     }
962   }
963
964   // Returns a new dimension whose value is given by a scalar input tensor.
965   Status InferenceContext::MakeDimForScalarInput(int idx, DimensionHandle* out) {
966     int64_t val;
967     const Tensor* t = input_tensor(idx);
968     if (t == nullptr) {
969       *out = UnknownDim();
970       return Status::OK();
971     }
972     TF_RETURN_IF_ERROR(GetScalarFromTensor(t, &val));
973     if (val < 0) {
974       return errors::InvalidArgument("Dimension size, given by scalar input ",
975                                      idx, ", must be non-negative but is ", val);
976     }
977     *out = MakeDim(val);
978     return Status::OK();
979   }
980
981   Status InferenceContext::MakeDimForScalarInputWithNegativeIndexing(
982       int idx, int input_rank, DimensionHandle* out) {
983     int64_t val;
984     const Tensor* t = input_tensor(idx);
985     if (t == nullptr) {
986       *out = UnknownDim();
987       return Status::OK();
988     }
989     TF_RETURN_IF_ERROR(GetScalarFromTensor(t, &val));
990     if (val < 0) {
991       if (input_rank < 0) {
992         *out = UnknownDim();
993         return Status::OK();
994       } else if (val + input_rank < 0) {
995         return errors::InvalidArgument("Dimension size, given by scalar input ",
996                                        val, " must be in range [-", input_rank,
997                                        ", ", input_rank, ")");
998       } else {
999         val += input_rank;
1000      }
1001    } else if (input_rank >= 0 && val >= input_rank) {
1002      return errors::InvalidArgument("Dimension size, given by scalar input ",
1003                                     val, " must be in range [-", input_rank,
1004                                     ", ", input_rank, ")");
1005    }
1006    *out = MakeDim(val);
1007    return Status::OK();
1008  }
1009
```

```
1010 Status InferenceContext::Divide(DimensionHandle dividend,
1011                                 DimensionOrConstant divisor,
1012                                 bool evenly_divisible, DimensionHandle* out) {
1013   const int64_t divisor_value = Value(divisor);
1014   if (divisor_value == 1) {
1015     *out = dividend;
1016   } else if (!ValueKnown(dividend) ||
1017              (divisor.dim.IsSet() && !ValueKnown(divisor.dim))) {
1018     *out = UnknownDim();
1019   } else {
1020     const int64_t v = Value(dividend);
1021     if (divisor_value <= 0) {
1022       return errors::InvalidArgument("Divisor must be positive but is ",
1023                                      divisor_value);
1024     }
1025     if (evenly_divisible && (v % divisor_value) != 0) {
1026       return errors::InvalidArgument(
1027           "Dimension size must be evenly divisible by ", divisor_value,
1028           " but is ", v);
1029     }
1030     *out = MakeDim(v / divisor_value);
1031   }
1032   return Status::OK();
1033 }
1034
1035 Status InferenceContext::Add(DimensionHandle first, DimensionOrConstant second,
1036                              DimensionHandle* out) {
1037   const int64_t first_value = Value(first);
1038   const int64_t second_value = Value(second);
1039   // Special cases.
1040   if (first_value == 0) {
1041     *out = MakeDim(second);
1042   } else if (second_value == 0) {
1043     *out = first;
1044   } else if (first_value == kUnknownDim || second_value == kUnknownDim) {
1045     *out = UnknownDim();
1046   } else {
1047     // Invariant: Both values are known and positive. Still in run-time we can
1048     // get pair of values which cannot be store in output. Check below will
1049     // report error. We still need to avoid undefined behavior of signed
1050     // overflow and use unsigned addition.
1051     const int64_t sum = static_cast<uint64>(first_value) + second_value;
1052     if (sum < 0) {
1053       return errors::InvalidArgument("Dimension size overflow from adding ",
1054                                      first_value, " and ", second_value);
1055     }
1056     *out = MakeDim(sum);
1057   }
1058   return Status::OK();
```

```
}

Status InferenceContext::Subtract(DimensionHandle first,
                                  DimensionOrConstant second,
                                  DimensionHandle* out) {
  const int64_t first_value = Value(first);
  const int64_t second_value = Value(second);
  // Special cases.
  if (second_value == 0) {
    *out = first;
  } else if (first_value == kUnknownDim || second_value == kUnknownDim) {
    *out = UnknownDim();
  } else {
    // Invariant: Both values are known, first_value is non-negative, and
    // second_value is positive.
    if (first_value < second_value) {
      return errors::InvalidArgument(
          "Negative dimension size caused by subtracting ", second_value,
          " from ", first_value);
    }
    *out = MakeDim(first_value - second_value);
  }
  return Status::OK();
}

Status InferenceContext::Multiply(DimensionHandle first,
                                  DimensionOrConstant second,
                                  DimensionHandle* out) {
  const int64_t first_value = Value(first);
  const int64_t second_value = Value(second);
  // Special cases.
  if (first_value == 0) {
    *out = first;
  } else if (second_value == 0) {
    *out = MakeDim(second);
  } else if (first_value == 1) {
    *out = MakeDim(second);
  } else if (second_value == 1) {
    *out = first;
  } else if (first_value == kUnknownDim || second_value == kUnknownDim) {
    *out = UnknownDim();
  } else {
    // Invariant: Both values are known and greater than 1.
    const int64_t product = first_value * second_value;
    if (product < 0) {
      return errors::InvalidArgument(
          "Negative dimension size caused by overflow when multiplying ",
          first_value, " and ", second_value);
    }
```

```
1108      *out = MakeDim(product);
1109    }
1110    return Status::OK();
1111  }
1112
1113  Status InferenceContext::Min(DimensionHandle first, DimensionOrConstant second,
1114                               DimensionHandle* out) {
1115    const int64_t first_value = Value(first);
1116    const int64_t second_value = Value(second);
1117    if (first_value == 0) {
1118      *out = first;
1119    } else if (second_value == 0) {
1120      *out = MakeDim(second);
1121    } else if (first_value == kUnknownDim || second_value == kUnknownDim) {
1122      *out = UnknownDim();
1123    } else {
1124      if (first_value <= second_value) {
1125        *out = first;
1126      } else {
1127        *out = MakeDim(second);
1128      }
1129    }
1130    return Status::OK();
1131  }
1132
1133  Status InferenceContext::Max(DimensionHandle first, DimensionOrConstant second,
1134                               DimensionHandle* out) {
1135    const int64_t first_value = Value(first);
1136    const int64_t second_value = Value(second);
1137    if (first_value == kUnknownDim || second_value == kUnknownDim) {
1138      *out = UnknownDim();
1139    } else {
1140      if (first_value >= second_value) {
1141        *out = first;
1142      } else {
1143        *out = MakeDim(second);
1144      }
1145    }
1146    return Status::OK();
1147  }
1148
1149  Status InferenceContext::AttachContext(const Status& status) {
1150    std::vector<string> input_shapes;
1151    input_shapes.reserve(inputs_.size());
1152    for (const ShapeHandle& input_shape : inputs_) {
1153      input_shapes.emplace_back(DebugString(input_shape));
1154    }
1155
1156    // Add information about the input tensors and partial tensor shapes used.
```

```
1157    std::vector<string> input_from_tensors_str;
1158    std::vector<string> input_from_tensors_as_shape_str;
1159    input_from_tensors_as_shape_str.reserve(inputs_.size());
1160    for (int i = 0, end = inputs_.size(); i < end; ++i) {
1161      const int input_tensors_as_shapes_size = input_tensors_as_shapes_.size();
1162      const int input_tensors_size = input_tensors_.size();
1163      if (requested_input_tensor_as_partial_shape_[i] &&
1164          i < input_tensors_as_shapes_size &&
1165          input_tensors_as_shapes_[i].IsSet() &&
1166          RankKnown(input_tensors_as_shapes_[i])) {
1167        input_from_tensors_as_shape_str.push_back(strings::StrCat(
1168            "input[", i, "] = ", DebugString(input_tensors_as_shapes_[i])));
1169      } else if (requested_input_tensor_[i] && i < input_tensors_size &&
1170                 input_tensors_[i] != nullptr) {
1171        input_from_tensors_str.push_back(strings::StrCat(
1172            "input[", i, "] = <",
1173            input_tensors_[i]->SummarizeValue(256 /* max_values */), ">"));
1174      }
1175    }
1176
1177    string error_context = strings::StrCat(
1178        " for '", attrs_.SummarizeNode(),
1179        "' with input shapes: ", absl::StrJoin(input_shapes, ", "));
1180    if (!input_from_tensors_str.empty()) {
1181      strings::StrAppend(&error_context, " and with computed input tensors: ",
1182                         absl::StrJoin(input_from_tensors_str, ", "));
1183    }
1184    if (!input_from_tensors_as_shape_str.empty()) {
1185      strings::StrAppend(&error_context,
1186                         " and with input tensors computed as partial shapes: ",
1187                         absl::StrJoin(input_from_tensors_as_shape_str, ","));
1188    }
1189
1190    strings::StrAppend(&error_context, ".");
1191    return errors::CreateWithUpdatedMessage(
1192        status, strings::StrCat(status.error_message(), error_context));
1193  }
1194
1195  bool InferenceContext::MergeHandleShapesAndTypes(
1196      const std::vector<ShapeAndType>& shapes_and_types,
1197      std::vector<ShapeAndType>* to_update) {
1198    if (shapes_and_types.size() != to_update->size()) {
1199      return false;
1200    }
1201    std::vector<ShapeAndType> new_values(shapes_and_types.size());
1202    bool refined = false;
1203    for (int i = 0, end = shapes_and_types.size(); i < end; ++i) {
1204      const ShapeAndType& existing = (*to_update)[i];
1205      if (shapes_and_types[i].dtype == existing.dtype) {
```

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

```
1255  bool InferenceContext::RelaxHandleShapesAndMergeTypes(
1256      const std::vector<ShapeAndType>& shapes_and_types,
1257      std::vector<ShapeAndType>* to_update) {
1258    if (shapes_and_types.size() != to_update->size()) {
1259      return false;
1260    }
1261    std::vector<ShapeAndType> new_values(shapes_and_types.size());
1262    for (int i = 0, end = shapes_and_types.size(); i < end; ++i) {
1263      const ShapeAndType& existing = (*to_update)[i];
1264      if (shapes_and_types[i].dtype == existing.dtype) {
1265        new_values[i].dtype = existing.dtype;
1266      } else {
1267        if (existing.dtype != DT_INVALID) {
1268          return false;
1269        } else {
1270          new_values[i].dtype = shapes_and_types[i].dtype;
1271        }
1272      }
1273      Relax(existing.shape, shapes_and_types[i].shape, &new_values[i].shape);
1274    }
1275    to_update->swap(new_values);
1276    return true;
1277  }
1278
1279  bool InferenceContext::RelaxOutputHandleShapesAndMergeTypes(
1280      int idx, const std::vector<ShapeAndType>& shapes_and_types) {
1281    if (output_handle_shapes_and_types_[idx] == nullptr) {
1282      output_handle_shapes_and_types_[idx].reset(
1283          new std::vector<ShapeAndType>(shapes_and_types));
1284      return true;
1285    }
1286    return RelaxHandleShapesAndMergeTypes(
1287        shapes_and_types, output_handle_shapes_and_types_[idx].get());
1288  }
1289
1290  bool InferenceContext::RelaxInputHandleShapesAndMergeTypes(
1291      int idx, const std::vector<ShapeAndType>& shapes_and_types) {
1292    if (input_handle_shapes_and_types_[idx] == nullptr) {
1293      input_handle_shapes_and_types_[idx].reset(
1294          new std::vector<ShapeAndType>(shapes_and_types));
1295      return true;
1296    }
1297    return RelaxHandleShapesAndMergeTypes(
1298        shapes_and_types, input_handle_shapes_and_types_[idx].get());
1299  }
1300
1301  // ----------------------------------------------------------------------
1302  // ShapeManager
1303  // ----------------------------------------------------------------------
```

```
1304  InferenceContext::ShapeManager::ShapeManager() {}
1305  InferenceContext::ShapeManager::~ShapeManager() {
1306    for (auto* s : all_shapes_) delete s;
1307    for (auto* d : all_dims_) delete d;
1308  }
1309
1310  ShapeHandle InferenceContext::ShapeManager::MakeShape(
1311      const std::vector<DimensionHandle>& dims) {
1312    all_shapes_.push_back(new Shape(dims));
1313    return all_shapes_.back();
1314  }
1315
1316  ShapeHandle InferenceContext::ShapeManager::UnknownShape() {
1317    all_shapes_.push_back(new Shape());
1318    return all_shapes_.back();
1319  }
1320
1321  }  // namespace shape_inference
1322  }  // namespace tensorflow
```