tensorflow / **tensorflow** Public

Code Issues 2.1k Pull requests 283 Actions Projects 1 ...

5100e359ae ▾ ...

tensorflow / tensorflow / core / framework / **shape_inference.cc**

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

22 contributors +10

1322 lines (1209 sloc) | 44 KB ...

```
/* Copyright 2016 The TensorFlow Authors. All Rights Reserved.

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

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

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

#include <cstdint>

#include "tensorflow/core/framework/bounds_check.h"
#include "tensorflow/core/framework/full_type_util.h"
#include "tensorflow/core/framework/node_def.pb.h"
#include "tensorflow/core/framework/op_def.pb.h"
#include "tensorflow/core/framework/partial_tensor_shape.h"
#include "tensorflow/core/framework/tensor_shape.pb.h"
#include "tensorflow/core/lib/core/errors.h"
#include "tensorflow/core/lib/strings/numbers.h"
#include "tensorflow/core/lib/strings/scanner.h"
#include "tensorflow/core/lib/strings/str_util.h"

```

```
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();
```

```cpp
    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;
  } else if (!RankKnown(s_new) || !s_old.IsSet()) {
    ForgetMerges();
    *out = s_new;
    return;
  }

  const int32_t rank = Rank(s_old);
  if (rank != Rank(s_new)) {
    ForgetMerges();
    *out = UnknownShape();
    return;
  }

  bool return_s_old = true;
  for (int i = 0; i < rank; ++i) {
    auto d0 = Dim(s_old, i);
    auto d1 = Dim(s_new, i);
    if (d0.SameHandle(d1)) continue;

    auto v0 = Value(d0);
    auto v1 = Value(d1);
    if (v0 == kUnknownDim || v1 == kUnknownDim || v0 != v1) {
      return_s_old = false;
      break;
    }
  }
  if (return_s_old) {
    *out = s_old;
    return;
  }

  // Relax dims.
  std::vector<DimensionHandle> dims(rank);
  for (int i = 0; i < rank; ++i) {
    Relax(Dim(s_old, i), Dim(s_new, i), &dims[i]);
  }
  ForgetMerges();
  *out = MakeShape(dims);
}

Status InferenceContext::Merge(ShapeHandle s0, ShapeHandle s1,
                               ShapeHandle* out) {
  if (s0.SameHandle(s1)) {
    *out = s0;
    return Status::OK();
  } else if (!RankKnown(s1)) {
    *out = s0;
    merged_shapes_.emplace_back(s0, s1);
    return Status::OK();
  } else if (!RankKnown(s0)) {
    *out = s1;
    merged_shapes_.emplace_back(s0, s1);
    return Status::OK();
  }

  const int32_t rank = Rank(s0);
  if (rank != Rank(s1)) {
    *out = nullptr;
    return errors::InvalidArgument("Shapes must be equal rank, but are ", rank,
                                   " and ", Rank(s1));
  }

  bool return_s0 = true;
  bool return_s1 = true;
  for (int i = 0; i < rank; ++i) {
    auto d0 = Dim(s0, i);
    auto d1 = Dim(s1, i);
    if (d0.SameHandle(d1)) continue;

    auto v0 = Value(d0);
    auto v1 = Value(d1);
    if (v0 == kUnknownDim) {
      if (v1 != kUnknownDim) {
        return_s0 = false;
      }
    } else if (v1 == kUnknownDim) {
      return_s1 = false;
    } else if (v0 != v1) {
      *out = nullptr;
      return errors::InvalidArgument(
          "Dimension ", i, " in both shapes must be equal, but are ", Value(d0),
          " and ", Value(d1), ". Shapes are ", DebugString(s0), " and ",
          DebugString(s1), ".");
    }
  }

  merged_shapes_.emplace_back(s0, s1);

  if (return_s0 || return_s1) {
    *out = return_s0 ? s0 : s1;
    return Status::OK();
  }

  // Merge dims.
  std::vector<DimensionHandle> dims(rank, nullptr);
  for (int i = 0; i < rank; ++i) {
    // Invariant for merge was checked earlier, so CHECK is ok.
    TF_CHECK_OK(Merge(Dim(s0, i), Dim(s1, i), &dims[i]));
  }

  Status s = ReturnCreatedShape(dims, out);
  if (s.ok()) {
    // Merge the new shape with s0. Since s0 and s1 are merged, this implies
    // that s1 and out are also merged.
    merged_shapes_.emplace_back(s0, *out);
  }
  return s;
}

Status InferenceContext::Subshape(ShapeHandle s, int64_t start,
                                  ShapeHandle* out) {
  return Subshape(s, start, std::numeric_limits<int64_t>::max() /* end */, out);
}

Status InferenceContext::Subshape(ShapeHandle s, int64_t start, int64_t end,
                                  ShapeHandle* out) {
  return Subshape(s, start, end, 1 /* stride */, out);
}

Status InferenceContext::Subshape(ShapeHandle s, int64_t start, int64_t end,
                                  int64_t stride, ShapeHandle* out) {
  int64_t start_in = start;
  int64_t end_in = end;

  const int32_t rank = Rank(s);
  if (start == 0 && stride == 1 &&
      ((RankKnown(s) && end >= rank) ||
       end == std::numeric_limits<int64_t>::max())) {
    *out = s;
    return Status::OK();
  }
  if (!RankKnown(s)) {
    return ReturnUnknownShape(out);
  }

  if (start > rank) start = rank;
  if (end > rank) end = rank;

  if (stride < 0 && start == rank) --start;

  if (start < 0) {
    start = rank + start;
    if (start < 0) {
      *out = nullptr;
      return errors::InvalidArgument("Subshape start out of bounds: ", start_in,
                                     ", for shape with rank ", rank);
    }
  }

  if (end < 0) {
    end = rank + end;
    if (end < 0) {
      *out = nullptr;
      return errors::InvalidArgument("Subshape end out of bounds: ", end_in,
                                     ", for shape with rank ", rank);
    }
  }
  if (stride > 0 && start > end) {
    *out = nullptr;
    return errors::InvalidArgument(
        "Subshape must have computed start <= end, but is ", start, " and ",
        end, " (computed from start ", start_in, " and end ", end_in,
        " over shape with rank ", rank, ")");
  } else if (stride < 0 && start < end) {
    *out = nullptr;
    return errors::InvalidArgument(
        "Subshape must have computed start >= end since stride is negative, "
        "but is ",
        start, " and ", end, " (computed from start ", start_in, " and end ",
        end_in, " over shape with rank ", rank, " and stride", stride, ")");
  }

  std::vector<DimensionHandle> dims;
  for (int i = start; stride > 0 ? i < end : i > end; i += stride) {
    dims.push_back(Dim(s, i));
  }
  return ReturnCreatedShape(dims, out);
}

Status InferenceContext::Concatenate(ShapeHandle s1, ShapeHandle s2,
                                     ShapeHandle* out) {
  if (!RankKnown(s1) || !RankKnown(s2)) {
    return ReturnUnknownShape(out);
  }
  const int32_t s1_rank = Rank(s1);
  const int32_t s2_rank = Rank(s2);
  const int32_t rank = s1_rank + s2_rank;
  std::vector<DimensionHandle> dims;
  dims.reserve(rank);
  for (int i = 0; i < s1_rank; ++i) dims.push_back(Dim(s1, i));
  for (int i = 0; i < s2_rank; ++i) dims.push_back(Dim(s2, i));
  return ReturnCreatedShape(dims, out);
}

Status InferenceContext::ReplaceDim(ShapeHandle s, int64_t dim_index_in,
                                    DimensionHandle new_dim, ShapeHandle* out) {
```

```
  if (!RankKnown(s)) {
    return ReturnUnknownShape(out);
  }
  int64_t dim_index = dim_index_in;
  if (dim_index < 0) {
    dim_index = s->dims_.size() + dim_index;
  }
  if (!FastBoundsCheck(dim_index, s->dims_.size())) {
    *out = nullptr;
    return errors::InvalidArgument("Out of range dim_index ", dim_index_in,
                                   " for shape with ", s->dims_.size(),
                                   " dimensions");
  }
  std::vector<DimensionHandle> dims(s->dims_);
  dims[dim_index] = new_dim;
  return ReturnCreatedShape(dims, out);
}

ShapeHandle InferenceContext::MakeShape(
    const std::vector<DimensionHandle>& dims) {
  return shape_manager_.MakeShape(dims);
}

ShapeHandle InferenceContext::MakeShape(
    std::initializer_list<DimensionOrConstant> dims) {
  std::vector<DimensionHandle> dims_actual;
  dims_actual.reserve(dims.size());
  for (const DimensionOrConstant& d : dims) {
    dims_actual.push_back(MakeDim(d));
  }

  return shape_manager_.MakeShape(dims_actual);
}

ShapeHandle InferenceContext::UnknownShape() {
  return shape_manager_.UnknownShape();
}

ShapeHandle InferenceContext::UnknownShapeOfRank(int64_t rank) {
  CHECK_LE(rank, kint32max) << "rank must be less than kint32max";
  if (rank == kUnknownRank) {
    return UnknownShape();
  }
  CHECK_GE(rank, 0) << "rank must not be negative";
  std::vector<DimensionHandle> dims(rank);
  for (int32_t i = 0; i < rank; ++i) {
    dims[i] = UnknownDim();
  }
  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);
```

```
}

Status InferenceContext::MakeShapeFromTensor(const Tensor* t,
                                             ShapeHandle tensor_shape,
                                             ShapeHandle* out) {
  return InternalMakeShapeFromTensor(
      false /* treat_unknown_scalar_tensor_as_unknown_shape */, t, tensor_shape,
      out);
}

Status InferenceContext::InternalMakeShapeFromTensor(
    bool treat_unknown_scalar_tensor_as_unknown_shape, const Tensor* t,
    ShapeHandle tensor_shape, ShapeHandle* out) {
  // Only callers who have set
  if (!treat_unknown_scalar_tensor_as_unknown_shape) {
    TF_RETURN_IF_ERROR(WithRank(tensor_shape, 1, &tensor_shape));
  }
  if (t == nullptr) {
    // This is guarded by the check above.
    if (Rank(tensor_shape) == 0) {
      return ReturnUnknownShape(out);
    }
    // Shape tensor is not known, but if the shape of the shape tensor is then
    // the right number of unknown dims can be created.
    DimensionHandle shape_dim = Dim(tensor_shape, 0);
    if (!ValueKnown(shape_dim)) {
      return ReturnUnknownShape(out);
    }
    const auto num_dims = Value(shape_dim);
    // TODO(mihaimaruseac): Should be `TensorShape::MaxDimensions()` as we are
    // not able to materialize shapes with more than this number of dimensions
    // but then shape inference would fail for operations such as
    // `tf.range`/`tf.ones`, etc. where the shape is not really materialized,
    // only used during the inference. Hence, just prevent doing a `reserve`
    // with a very large argument.
    const int64_t max_dimensions = 1 << 20;
    if (num_dims >= max_dimensions) {
      return errors::Internal(
          "Cannot create a tensor with ", num_dims,
          " dimensions, as these would be more than maximum of ",
          max_dimensions);
    }
    std::vector<DimensionHandle> dims;
    dims.reserve(num_dims);
    for (int i = 0; i < num_dims; i++) dims.push_back(UnknownDim());
    return ReturnCreatedShape(dims, out);
  }

  if (t->shape().dims() == 0) {
    if (t->dtype() == DataType::DT_INT32) {
      auto flat_t = t->scalar<int32>();
      if (flat_t() != -1) {
        *out = nullptr;
        return errors::InvalidArgument(
            "Input tensor must be rank 1, or if its rank 0 it must have value "
            "-1 "
            "(representing an unknown shape).  Saw value: ",
            flat_t());
      }
      return ReturnUnknownShape(out);
    } else if (t->dtype() == DataType::DT_INT64) {
      auto flat_t = t->scalar<int64_t>();
      if (flat_t() != -1) {
        *out = nullptr;
        return errors::InvalidArgument(
            "Input tensor must be rank 1, or if its rank 0 it must have value "
            "-1 "
            "(representing an unknown shape).  Saw value: ",
            flat_t());
      }
      return ReturnUnknownShape(out);
    } else {
      *out = nullptr;
      return errors::InvalidArgument(
          "Input tensor must be int32 or int64, but was ",
          DataTypeString(t->dtype()));
    }
  }

  if (t->shape().dims() != 1) {
    *out = nullptr;
    return errors::InvalidArgument(
        "Input tensor must be rank 1, but was rank ", t->shape().dims(), ".",
        ((t->shape().dims() == 0)
             ? "If it is rank 0 rank 0 it must have statically known value -1 "
               "(representing an unknown shape). "
             : " "),
        "Saw tensor shape ", t->shape().DebugString());
  }
  std::vector<DimensionHandle> dims;
  if (t->dtype() == DataType::DT_INT32) {
    auto flat_t = t->flat<int32>();
    for (int i = 0; i < flat_t.size(); ++i) {
      const int32_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 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;
  TF_RETURN_IF_ERROR(PartialTensorShape::IsValidShape(proto));
  PartialTensorShape partial_shape(proto);
  return MakeShapeFromPartialTensorShape(partial_shape, out);
}

Status InferenceContext::GetScalarFromTensor(const Tensor* t, int64_t* val) {
  // Caller must ensure that <t> is not NULL.
  const int rank = t->dims();
  if (rank != 0) {
    return errors::InvalidArgument("Input must be scalar but has rank ", rank);
  }

  if (t->dtype() == DataType::DT_INT32) {
    *val = t->scalar<int32>()();
    return Status::OK();
  } else if (t->dtype() == DataType::DT_INT64) {
    *val = t->scalar<int64_t>()();
    return Status::OK();
  } else {
    return errors::InvalidArgument("Scalar input must be int32 or int64.");
  }
}

Status InferenceContext::GetScalarFromTensor(const Tensor* t, int64_t idx,
                                             int64_t* val) {
  // Caller must ensure that <t> is not NULL.
  const int rank = t->dims();
  if (rank != 1) {
    return errors::InvalidArgument("Input must be 1D but has rank ", rank);
  }

  if (t->dtype() == DataType::DT_INT32) {
    auto flat_t = t->flat<int32>();
    if (idx < 0 || idx >= flat_t.size()) {
      return errors::InvalidArgument("Invalid index ", idx,
                                     " for Tensor of size ", flat_t.size());
    }
    *val = flat_t(idx);
    return Status::OK();
  } else if (t->dtype() == DataType::DT_INT64) {
    auto flat_t = t->flat<int64_t>();
    if (idx < 0 || idx >= flat_t.size()) {
      return errors::InvalidArgument("Invalid index ", idx,
                                     " for Tensor of size ", flat_t.size());
    }
    *val = flat_t(idx);
    return Status::OK();
  } else {
    return errors::InvalidArgument("Tensor input must be int32 or int64.");
  }
}

// Returns a new dimension whose value is given by a scalar input tensor.
Status InferenceContext::MakeDimForScalarInput(int idx, DimensionHandle* out) {
  int64_t val;
  const Tensor* t = input_tensor(idx);
  if (t == nullptr) {
    *out = UnknownDim();
    return Status::OK();
  }
  TF_RETURN_IF_ERROR(GetScalarFromTensor(t, &val));
  if (val < 0) {
    return errors::InvalidArgument("Dimension size, given by scalar input ",
                                   idx, ", must be non-negative but is ", val);
  }
  *out = MakeDim(val);
  return Status::OK();
}

Status InferenceContext::MakeDimForScalarInputWithNegativeIndexing(
    int idx, int input_rank, DimensionHandle* out) {
  int64_t val;
  const Tensor* t = input_tensor(idx);
  if (t == nullptr) {
    *out = UnknownDim();
    return Status::OK();
  }
  TF_RETURN_IF_ERROR(GetScalarFromTensor(t, &val));
  if (val < 0) {
    if (input_rank < 0) {
      *out = UnknownDim();
      return Status::OK();
    } else if (val + input_rank < 0) {
      return errors::InvalidArgument("Dimension size, given by scalar input ",
                                     val, " must be in range [-", input_rank,
                                     ", ", input_rank, ")");
    } else {
      val += input_rank;
    }
  } else if (input_rank >= 0 && val >= input_rank) {
    return errors::InvalidArgument("Dimension size, given by scalar input ",
                                   val, " must be in range [-", input_rank,
                                   ", ", input_rank, ")");
  }
  *out = MakeDim(val);
  return Status::OK();
}

```

```
Status InferenceContext::Divide(DimensionHandle dividend,
                                DimensionOrConstant divisor,
                                bool evenly_divisible, DimensionHandle* out) {
  const int64_t divisor_value = Value(divisor);
  if (divisor_value == 1) {
    *out = dividend;
  } else if (!ValueKnown(dividend) ||
             (divisor.dim.IsSet() && !ValueKnown(divisor.dim))) {
    *out = UnknownDim();
  } else {
    const int64_t v = Value(dividend);
    if (divisor_value <= 0) {
      return errors::InvalidArgument("Divisor must be positive but is ",
                                     divisor_value);
    }
    if (evenly_divisible && (v % divisor_value) != 0) {
      return errors::InvalidArgument(
          "Dimension size must be evenly divisible by ", divisor_value,
          " but is ", v);
    }
    *out = MakeDim(v / divisor_value);
  }
  return Status::OK();
}

Status InferenceContext::Add(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 = MakeDim(second);
  } else if (second_value == 0) {
    *out = first;
  } else if (first_value == kUnknownDim || second_value == kUnknownDim) {
    *out = UnknownDim();
  } else {
    // Invariant: Both values are known and positive. Still in run-time we can
    // get pair of values which cannot be store in output. Check below will
    // report error. We still need to avoid undefined behavior of signed
    // overflow and use unsigned addition.
    const int64_t sum = static_cast<uint64>(first_value) + second_value;
    if (sum < 0) {
      return errors::InvalidArgument("Dimension size overflow from adding ",
                                     first_value, " and ", second_value);
    }
    *out = MakeDim(sum);
  }
  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);
    }
    *out = MakeDim(product);
  }
  return Status::OK();
}

Status InferenceContext::Min(DimensionHandle first, DimensionOrConstant second,
                             DimensionHandle* out) {
  const int64_t first_value = Value(first);
  const int64_t second_value = Value(second);
  if (first_value == 0) {
    *out = first;
  } else if (second_value == 0) {
    *out = MakeDim(second);
  } else 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::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 */), ">"));
    }
  }

  string error_context = strings::StrCat(
      " for '", attrs_.SummarizeNode(),
      "' with input shapes: ", absl::StrJoin(input_shapes, ", "));
  if (!input_from_tensors_str.empty()) {
    strings::StrAppend(&error_context, " and with computed input tensors: ",
                       absl::StrJoin(input_from_tensors_str, ", "));
  }
  if (!input_from_tensors_as_shape_str.empty()) {
    strings::StrAppend(&error_context,
                       " and with input tensors computed as partial shapes: ",
                       absl::StrJoin(input_from_tensors_as_shape_str, ","));
  }

  strings::StrAppend(&error_context, ".");
  return errors::CreateWithUpdatedMessage(
      status, strings::StrCat(status.error_message(), error_context));
}

bool InferenceContext::MergeHandleShapesAndTypes(
    const std::vector<ShapeAndType>& shapes_and_types,
    std::vector<ShapeAndType>* to_update) {
  if (shapes_and_types.size() != to_update->size()) {
    return false;
  }
  std::vector<ShapeAndType> new_values(shapes_and_types.size());
  bool refined = false;
  for (int i = 0, end = shapes_and_types.size(); i < end; ++i) {
    const ShapeAndType& existing = (*to_update)[i];
    if (shapes_and_types[i].dtype == existing.dtype) {
      new_values[i].dtype = existing.dtype;
    } else {
      if (existing.dtype != DT_INVALID) {
        return false;
      } else {
        new_values[i].dtype = shapes_and_types[i].dtype;
        refined = true;
      }
    }
    if (!Merge(existing.shape, shapes_and_types[i].shape, &new_values[i].shape)
             .ok()) {
      // merge failed, ignore the new value.
      new_values[i].shape = existing.shape;
    }
    if (!existing.shape.SameHandle(new_values[i].shape)) {
      refined = true;
    }
  }
  if (!refined) {
    return false;
  }
  for (int i = 0, end = new_values.size(); i < end; ++i) {
    (*to_update)[i] = new_values[i];
  }
  return true;
}

bool InferenceContext::MergeOutputHandleShapesAndTypes(
    int idx, const std::vector<ShapeAndType>& shapes_and_types) {
  if (output_handle_shapes_and_types_[idx] == nullptr) {
    output_handle_shapes_and_types_[idx].reset(
        new std::vector<ShapeAndType>(shapes_and_types));
    return true;
  }
  return MergeHandleShapesAndTypes(shapes_and_types,
                                   output_handle_shapes_and_types_[idx].get());
}

bool InferenceContext::MergeInputHandleShapesAndTypes(
    int idx, const std::vector<ShapeAndType>& shapes_and_types) {
  if (input_handle_shapes_and_types_[idx] == nullptr) {
    input_handle_shapes_and_types_[idx].reset(
        new std::vector<ShapeAndType>(shapes_and_types));
    return true;
  }
  return MergeHandleShapesAndTypes(shapes_and_types,
                                   input_handle_shapes_and_types_[idx].get());
}

```

```
bool InferenceContext::RelaxHandleShapesAndMergeTypes(
    const std::vector<ShapeAndType>& shapes_and_types,
    std::vector<ShapeAndType>* to_update) {
  if (shapes_and_types.size() != to_update->size()) {
    return false;
  }
  std::vector<ShapeAndType> new_values(shapes_and_types.size());
  for (int i = 0, end = shapes_and_types.size(); i < end; ++i) {
    const ShapeAndType& existing = (*to_update)[i];
    if (shapes_and_types[i].dtype == existing.dtype) {
      new_values[i].dtype = existing.dtype;
    } else {
      if (existing.dtype != DT_INVALID) {
        return false;
      } else {
        new_values[i].dtype = shapes_and_types[i].dtype;
      }
    }
    Relax(existing.shape, shapes_and_types[i].shape, &new_values[i].shape);
  }
  to_update->swap(new_values);
  return true;
}

bool InferenceContext::RelaxOutputHandleShapesAndMergeTypes(
    int idx, const std::vector<ShapeAndType>& shapes_and_types) {
  if (output_handle_shapes_and_types_[idx] == nullptr) {
    output_handle_shapes_and_types_[idx].reset(
        new std::vector<ShapeAndType>(shapes_and_types));
    return true;
  }
  return RelaxHandleShapesAndMergeTypes(
      shapes_and_types, output_handle_shapes_and_types_[idx].get());
}

bool InferenceContext::RelaxInputHandleShapesAndMergeTypes(
    int idx, const std::vector<ShapeAndType>& shapes_and_types) {
  if (input_handle_shapes_and_types_[idx] == nullptr) {
    input_handle_shapes_and_types_[idx].reset(
        new std::vector<ShapeAndType>(shapes_and_types));
    return true;
  }
  return RelaxHandleShapesAndMergeTypes(
      shapes_and_types, input_handle_shapes_and_types_[idx].get());
}

// -----------------------------------------------------------------------------
// ShapeManager
// -----------------------------------------------------------------------------
```

```cpp
InferenceContext::ShapeManager::ShapeManager() {}
InferenceContext::ShapeManager::~ShapeManager() {
  for (auto* s : all_shapes_) delete s;
  for (auto* d : all_dims_) delete d;
}

ShapeHandle InferenceContext::ShapeManager::MakeShape(
    const std::vector<DimensionHandle>& dims) {
  all_shapes_.push_back(new Shape(dims));
  return all_shapes_.back();
}

ShapeHandle InferenceContext::ShapeManager::UnknownShape() {
  all_shapes_.push_back(new Shape());
  return all_shapes_.back();
}

}  // namespace shape_inference
}  // namespace tensorflow
```