tensorflow / **tensorflow** Public

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

a1320ec1ea ▾

tensorflow / tensorflow / core / framework / **shape_inference.h**

jpienaar Document ShapeHandle & DimensionHandle ownership ... ✓ **History**

18 contributors +6

902 lines (761 sloc) | 36.6 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.
==============================================================================*/
#ifndef TENSORFLOW_CORE_FRAMEWORK_SHAPE_INFERENCE_H_
#define TENSORFLOW_CORE_FRAMEWORK_SHAPE_INFERENCE_H_

#include <vector>

#include "absl/memory/memory.h"
#include "tensorflow/core/framework/full_type.pb.h"
#include "tensorflow/core/framework/node_def_util.h"
#include "tensorflow/core/framework/tensor.h"
#include "tensorflow/core/lib/core/errors.h"
#include "tensorflow/core/lib/core/status.h"
#include "tensorflow/core/lib/gtl/inlined_vector.h"
#include "tensorflow/core/platform/macros.h"

namespace tensorflow {
```

```cpp

namespace grappler {
class GraphProperties;
class SymbolicShapeManager;
}  // namespace grappler

namespace shape_inference {

struct DimensionOrConstant;
class InferenceContext;

// This header contains the InferenceContext that is used to infer the shape of
// the results of an operation or flag an operation with invalid inputs (e.g.,
// mismatched shapes for elementwise operation) by ShapeRefiner. The shape of an
// operation is computed using the OpShapeInferenceFn set via SetShapeFn in op
// registration. The OpShapeInferenceFn uses a per op InferenceContext populated
// with input shapes to compute resultant shape (including resource shapes).
//
// The shapes created in the InferenceContext are bound to the lifetime of the
// InferenceContext in which it was created. E.g., in
//
// ```c++
//  InferenceContext c;
//  // Below a ShapeHandle is returned by MakeShape, while UnknownDim returns a
//  // DimensionHandle.
//  ShapeHandle in0 = c.MakeShape({10, c.UnknownDim()});
// ```
//
// the ShapeHandle `in0` (and the nested unknown dim inside) is only valid while
// `c` is in scope, as ShapeHandle and DimensionHandle are effectively
// wrappers around pointers stored inside the context with the lifetime of the
// value pointed to managed by the context. The result from one operation's
// inference context will be passed as input to the inference of consumer
// operations. Hence it is possible for ShapeHandles produced by inference on a
// node to consist of ShapeHandles owned by different InferenceContexts. While
// inferring the shapes of a Graph, the InferenceContext of all nodes/operations
// in the Graph remain resident for the lifetime of the Graph (e.g, there is a
// map from each node to its InferenceContext, technically its
// ExtendedInferencContext which additionally stores the element types of inputs
// & outputs, which remains resident).
//
// For functions, the body of the function is instantiated as a Graph while
// inferring the result shapes of a function call node. The rules above apply
// while the function's shape is being inferred, but the contexts associated
// with nodes in the function body are released once the function call's
// resultant shapes are inferred. The shapes of results returned by a function
// are propagated to the InferenceContext of the function call's op (which is
// associated with a Graph of nodes whose shape is being inferred) as the return
// values of a function call node are the inputs of its consumer, but the return
```

```
// values are produced by nodes inside the function whose InferenceContexts
// (which owns the values pointed to by ShapeHandle and DimensionHandle) are
// reclaimed after inferring function result shapes. Recursive user-defined
// function are not supported hence inference of functions are fully nested with
// the InferenceContext's of function calls forming a stack.
//
// For example, consider the following call and function:
//
// ```python
// @tf.function
// def g(st):
//   d = tf.add(st, st)
//   return d
//
// @tf.function
// def f():
//   st = tf.A()
//   result = g(st)
//   return h(result)
// ```
//
// During inference of f, the shape of `A` will be inferred and the results from
// its InferenceContext used as inputs to function call `g(st)`. The call node
// will have an InferenceContext created (call it outer context) and the graph
// corresponding to function `g` will be instantiated. The result shape of the
// Arg nodes of the function will be associated with input from outer context.
// During inference of `g` (for the callsite `g(st)` in `f`), the
// InferenceContext of all nodes inside `g` will remain alive. Thus, when shape
// of `tf.add` is computed it may rely on all inputs. Once the RetVal nodes of a
// function is reached, we know the shape of its input may correspond to a shape
// queried in the outer context and it is explicitly copied to outer context. In
// this case that means that the shape of `d` is copied to the InferenceContext
// of `g(st)` and so when `h(result)` is executed this shape may be queried.
// Furthermore, no shapes computed due to call `g(st)` can be queried post this
// point and, as the RetVal shapes have been coppied into outer context, all
// InferenceContexts associated with nodes in function `g` instantiated for
// `g(st)` may be and are released.

// Dimension values are accessed through InferenceContext.
class Dimension {
 private:
  Dimension();
  Dimension(int64_t value);
  ~Dimension() {}

  const int64_t value_;

  friend class InferenceContext;
  friend class ShapeManager;
```

```
  TF_DISALLOW_COPY_AND_ASSIGN(Dimension);
};

class DimensionHandle {
 public:
  DimensionHandle() {}
  bool SameHandle(DimensionHandle d) const { return ptr_ == d.ptr_; }
  std::size_t Handle() const { return reinterpret_cast<std::size_t>(ptr_); }

 private:
  DimensionHandle(const Dimension* dim) { ptr_ = dim; }

  const Dimension* operator->() const { return ptr_; }
  bool IsSet() const { return ptr_ != nullptr; }

  const Dimension* ptr_ = nullptr;

  friend struct DimensionOrConstant;
  friend class InferenceContext;
  friend class ShapeInferenceTest;
  friend class ShapeInferenceTestutil;
  friend class ::tensorflow::grappler::GraphProperties;
  friend class ::tensorflow::grappler::SymbolicShapeManager;

  // Intentionally copyable.
};

// Shape rank and dimensions are accessed through InferenceContext.
class Shape {
 private:
  Shape();
  Shape(const std::vector<DimensionHandle>& dims);
  ~Shape() {}

  const int32 rank_;
  const std::vector<DimensionHandle> dims_;

  friend class InferenceContext;
  friend class ::tensorflow::grappler::SymbolicShapeManager;

  TF_DISALLOW_COPY_AND_ASSIGN(Shape);
};

class ShapeHandle {
 public:
  ShapeHandle() {}
  bool SameHandle(ShapeHandle s) const { return ptr_ == s.ptr_; }
  std::size_t Handle() const { return reinterpret_cast<std::size_t>(ptr_); }

 private:
  ShapeHandle(const Shape* shape) { ptr_ = shape; }
  const Shape* operator->() const { return ptr_; }
  bool IsSet() const { return ptr_ != nullptr; }

  const Shape* ptr_ = nullptr;

  friend class InferenceContext;
  friend class ShapeInferenceTest;
  friend class ShapeInferenceTestutil;
  friend class ::tensorflow::grappler::SymbolicShapeManager;

  // Intentionally copyable.
};

// Struct used to allow functions to take DimensionHandle or a dimension value.
// Not meant to be constructed directly.
struct DimensionOrConstant {
 public:
  // Intentionally not explicit.
  DimensionOrConstant(DimensionHandle dim);

  // val must be non-negative or InferenceContext::kUnknownDim.
  DimensionOrConstant(int64_t val);

  // dim takes precedence. If dim != nullptr, val is ignored.
  DimensionHandle dim;
  int64_t val;

 private:
  DimensionOrConstant();
};

struct ShapeAndType {
  ShapeAndType() {}
  ShapeAndType(ShapeHandle s, DataType t) : shape(s), dtype(t) {}
  // TODO(mdan): Remove dtype from constructor, and use type_ instead.
  // dtype is kept here for backward compatibiity. Its information should
  // be redundant to that in type;
  ShapeAndType(ShapeHandle s, DataType t, FullTypeDef type_)
      : shape(s), dtype(t), type(type_) {}

  ShapeHandle shape;
  DataType dtype = DT_INVALID;
  FullTypeDef type;
};

// Shape inference functions registered on ops in REGISTER_OP implement
// their shape functions in terms of this InferenceContext.  An InferenceContext
// is created by the framework and passed to a shape inference function.  The
// shape inference function calls functions on the context, and should call
// set_output() to set the shape on all outputs.
//
// To infer shapes for user-defined functions see ShapeRefiner.
//
// All Shape* and Dimension* returned by functions of InferenceContext are owned
// by the InferenceContext.
class InferenceContext {
 public:
  static constexpr int64_t kUnknownDim = -1;
  static constexpr int32_t kUnknownRank = -1;

  // <input_tensors> is NULL-padded to be the same size as <input_shapes>.
  //
  // Elements of <input_tensors_as_shapes> are used for when a shape function
  // makes a call to MakeShapeFromShapeTensor; in particular, when the
  // input_tensors[i] is nullptr but the shape represented by it is partially
  // known from analysis of the graph.
  // <input_tensors_as_shapes> can have fewer elements than <input_shapes>.
  // Values of <input_tensors_as_shapes> do not need to outlive the context.
  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);

  // <input_tensors> is NULL-padded to be the same size as <input_shapes>.
  //
  // Elements of <input_tensors_as_shapes> are used for when a shape
  // function makes a call to MakeShapeFromShapeTensor; in particular, when
  // the input_tensors[i] is nullptr but the shape represented by it is
  // partially known from analysis of the graph. <input_tensors_as_shapes>
  // can have fewer elements than <input_shapes>. Values of
  // <input_tensors_as_shapes> do not need to outlive the context.
  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);

  ~InferenceContext();

  // Runs the shape inference function 'fn' with 'this' as the
  // argument, returns the status of the inference.
  //
  // On error, additional context is provided in the error message.
  Status Run(
      const std::function<Status(shape_inference::InferenceContext* c)>& fn);

  // Merge the stored shape of the input in position idx with <shape> according
  // to the following rules:
  //
  // - If the ShapeHandles are the same or <shape> is unknown, there will be no
  //   change. Otherwise if the stored shape is unknown, the new shape will be
  //   <shape>.
  // - If both shapes are known, then they must have the same rank.
  // - For any one dimension, if the values for that dimension in both shapes
  //   are known, then the values must match.
  // - If one shape has equal or more information than the other shape in every
  //   dimension, the new shape will become the shape with more information.
  // - Example: merging [2,?] and [?,2] results in [2,2]
  // - Example: [2,2] cannot be merged with [1,2]
  //
  // This requires idx to be in the [0, num_inputs) range. If the merge is
  // successful, return true. Return false otherwise.
  bool MergeInput(int idx, ShapeHandle shape) {
    ShapeHandle new_shape;
    if (!Merge(inputs_[idx], shape, &new_shape).ok()) return false;
    inputs_[idx] = new_shape;
    return true;
  }

  // Relax the stored shape of the input in position idx with <shape> according
  // to the following rules:
  //
  // - If the ShapeHandles are the same then the stored shape will be returned.
  // - If either of the ShapeHandles are unknown, then a new UnknownShape will
  //   be returned. A new shape must be returned because we cannot claim that
  //   the resulting shape is necessarily the same as either of the input
  //   shapes.
  // - If the shapes both have known ranks but their ranks are different, a new
  //   UnknownShape will be returned.
  // - For any one dimension, if the value for that dimension in either of the
  //   shapes is unknown, a new shape will be returned with a new UnknownDim in
  //   that dimension.
  // - For any one dimension, if the values for that dimension in both shapes
  //   are known but do not match, a new shape will be returned with a new
  //   UnknownDim in that dimension.
  // - If both shapes have the same known rank and match in every dimension,
  //   the stored shape will be returned.
  // - Example: relaxing [2,?] and [?,2] results in [?,?]
  // - Example: relaxing [2,2] and [3,2] results in [?,2]
```

```cpp
  // - Example: relaxing [2,2] with [1,2,3] results in ?
  //
  // This requires idx to be in the [0, num_inputs) range. If the relax is
  // successful and the new shape differs from the old one, store the new
  // shape and return true. Return false otherwise.
  bool RelaxInput(int idx, ShapeHandle shape) {
    ShapeHandle new_shape;
    Relax(inputs_[idx], shape, &new_shape);
    if (inputs_[idx].SameHandle(new_shape)) {
      return false;
    }
    inputs_[idx] = new_shape;
    return true;
  }

  void SetInput(int idx, ShapeHandle shape) { inputs_[idx] = shape; }

  ShapeHandle input(int64_t idx) const { return inputs_[idx]; }
  Status input(StringPiece input_name, std::vector<ShapeHandle>* output) const;
  int num_inputs() const { return inputs_.size(); }

  // Returns the input tensor at index <idx>, or nullptr if the input tensor is
  // not available at the time of shape inference.
  const Tensor* input_tensor(int idx) {
    // Mark that this idx was requested.
    request_input_tensor(idx);
    return input_tensors_[idx];
  }

  // Notifies the shape refiner that the value of the tensor at index <idx>
  // is needed. The shape refiner tries to statically compute this tensor,
  // and if successful re-runs the  shape function with this tensor available
  // in the call to 'input_tensor(idx)'.
  void request_input_tensor(int idx) { requested_input_tensor_[idx] = true; }

  // Returns true iff input_tensor(idx) was called by the shape function.
  bool requested_input_tensor(int idx) const {
    return requested_input_tensor_[idx];
  }

  // Notifies the shape refiner that the value of the tensor at index <idx>
  // as a partial shape is needed. The shape refiner tries to statically compute
  // this, and if successful re-runs the  shape function with the
  // computed PartialTensorShape available in the call to
  // 'MakeShapeFromShapeTensor(idx, handle)' or
  // 'MakeShapeFromShapeTensorTreatScalarAsUnknownShape(idx, handle)'.
  void request_input_tensor_as_partial_shape(int idx) {
    requested_input_tensor_as_partial_shape_[idx] = true;
  }
  // Returns true if MakeShapeFromInputTensor was called but the constant
  // input_tensor was not present.
  bool requested_input_tensor_as_partial_shape(int idx) const {
    return requested_input_tensor_as_partial_shape_[idx];
  }

  void set_input_tensors(const std::vector<const Tensor*>& input_tensors) {
    input_tensors_ = input_tensors;
  }

  void set_input_tensors_as_shapes(
      const std::vector<ShapeHandle>& input_tensors_as_shapes) {
    input_tensors_as_shapes_ = input_tensors_as_shapes;
  }

  const std::vector<ShapeHandle>& input_tensors_as_shapes() const {
    return input_tensors_as_shapes_;
  }

  ShapeHandle output(int64_t idx) const { return outputs_.at(idx); }
  void set_output(int idx, ShapeHandle shape) { outputs_.at(idx) = shape; }
  Status set_output(StringPiece output_name,
                    const std::vector<ShapeHandle>& shapes);

  int num_outputs() const { return outputs_.size(); }
  ShapeHandle output(int idx) const { return outputs_.at(idx); }
  Status output(StringPiece output_name,
                std::vector<ShapeHandle>* output) const;

  // Returns the value for attribute named `attr_name`.
  Status GetAttr(StringPiece attr_name, const AttrValue** attr_value) const {
    return attrs_.Find(attr_name, attr_value);
  }
  const AttrValue* GetAttr(StringPiece attr_name) const {
    return attrs_.Find(attr_name);
  }

  const FullTypeDef& ret_types() const { return ret_types_; }

  // idx can be negative for an offset from end of dimensions.
  // idx must be in the range [-1 * s.rank, s.rank).
  DimensionHandle Dim(ShapeHandle s, int64_t idx) {
    if (!s.Handle() || s->rank_ == kUnknownRank) {
      return UnknownDim();
    }
    return DimKnownRank(s, idx);
  }
  // As above, but asserts that the rank of the shape is known.
```

```
static DimensionHandle DimKnownRank(ShapeHandle s, int64_t idx) {
  CHECK_NE(s->rank_, kUnknownRank);
  if (idx < 0) {
    return s->dims_[s->dims_.size() + idx];
  }
  return s->dims_[idx];
}

static int32 Rank(ShapeHandle s) {
  return s.IsSet() ? s->rank_ : kUnknownRank;
}
static bool RankKnown(ShapeHandle s) {
  return (s.IsSet() && (Rank(s) != kUnknownRank));
}
static inline int64_t Value(DimensionOrConstant d) {
  return d.dim.IsSet() ? d.dim->value_ : d.val;
}
static inline bool ValueKnown(DimensionOrConstant d) {
  return Value(d) != kUnknownDim;
}

// Fills the output proto with the shape defined by the handle.
// "proto" is expected to be empty prior to the call.
void ShapeHandleToProto(ShapeHandle handle, TensorShapeProto* proto);

// Returns true if the rank and all dimensions of the Shape are known.
bool FullyDefined(ShapeHandle s);

// Returns the total number of elements, or an unknown dimension for an
// incomplete shape.
DimensionHandle NumElements(ShapeHandle s);

std::string DebugString(ShapeHandle s);
std::string DebugString(DimensionHandle d);
std::string DebugString(const ShapeAndType& shape_and_type);
std::string DebugString(gtl::ArraySlice<ShapeAndType> shape_and_types);

// Describes the whole context, for debugging purposes.
std::string DebugString() const;

// If <shape> has rank <rank>, or its rank is unknown, return OK and return
// the shape with asserted rank in <*out>. Otherwise return an error.
//
// Note that <*out> may be set to <shape>.
Status WithRank(ShapeHandle shape, int64_t rank,
                ShapeHandle* out) TF_MUST_USE_RESULT;
Status WithRankAtLeast(ShapeHandle shape, int64_t rank,
                       ShapeHandle* out) TF_MUST_USE_RESULT;
Status WithRankAtMost(ShapeHandle shape, int64_t rank,
                      ShapeHandle* out) TF_MUST_USE_RESULT;

// If <dim> has value <value>, or its value is unknown, returns OK and returns
// the dimension with asserted value in <*out>. Otherwise returns an error.
//
// Note that <*out> may be set to <dim>.
Status WithValue(DimensionHandle dim, int64_t value,
                 DimensionHandle* out) TF_MUST_USE_RESULT;

// Merges <s0> and <s1> and returns the merged shape in <*out>. See
// 'MergeInput' function for full details and examples.
Status Merge(ShapeHandle s0, ShapeHandle s1,
             ShapeHandle* out) TF_MUST_USE_RESULT;

// Asserts that <s>'s rank >= <prefix>'s rank, and the first
// <prefix.rank> dimensions of <s> are compatible with the dimensions of
// <prefix>.
// Returns the merged results in <*s_out> and <*prefix_out>.
Status MergePrefix(ShapeHandle s, ShapeHandle prefix, ShapeHandle* s_out,
                   ShapeHandle* prefix_out) TF_MUST_USE_RESULT;

// Merges <d0> and <d1> and returns the merged dimension in <*out>. If <d0>
// and <d1> have incompatible values, returns an error.
//
// Note that <*out> may be set to <d0> or <d1>.
Status Merge(DimensionHandle d0, DimensionHandle d1,
             DimensionHandle* out) TF_MUST_USE_RESULT;

// Returns in <*out> a sub-shape of <s> with dimensions [start:].
// <start> can be negative to index from the end of the shape. If <start> >
// rank of <s>, then an empty subshape is returned.
Status Subshape(ShapeHandle s, int64_t start,
                ShapeHandle* out) TF_MUST_USE_RESULT;

// Returns in <*out> a sub-shape of <s>, with dimensions [start:end].
// <start> and <end> can be negative, to index from the end of the shape.
// <start> and <end> are set to the rank of <s> if > rank of <s>.
Status Subshape(ShapeHandle s, int64_t start, int64_t end,
                ShapeHandle* out) TF_MUST_USE_RESULT;

// Returns in <*out> a sub-shape of <s>, with dimensions [start:end:stride].
// <start> and <end> can be negative, to index from the end of the shape.
// <start> and <end> are set to the rank of <s> if > rank of <s>.
// <stride> can be negative, to reverse the <s>.
Status Subshape(ShapeHandle s, int64_t start, int64_t end, int64_t stride,
                ShapeHandle* out) TF_MUST_USE_RESULT;

// Returns in <*out> the result of appending the dimensions of <s2> to those
// of <s1>.
Status Concatenate(ShapeHandle s1, ShapeHandle s2,
                   ShapeHandle* out) TF_MUST_USE_RESULT;

// Returns in <out> the shape from replacing <s.dim[dim_index]> with
// <new_dim>.
Status ReplaceDim(ShapeHandle s, int64_t dim_index, DimensionHandle new_dim,
                  ShapeHandle* out) TF_MUST_USE_RESULT;

// Returns a new shape with the given dims. The returned value is owned by
// this context.
ShapeHandle MakeShape(const std::vector<DimensionHandle>& dims);
ShapeHandle MakeShape(std::initializer_list<DimensionOrConstant> dims);

// Returns a new unknown shape.
ShapeHandle UnknownShape();

// Returns a shape with specified rank but unknown dims.
ShapeHandle UnknownShapeOfRank(int64_t rank);

// Returns a new shape of zero dimensions.
ShapeHandle Scalar();

// Returns a new shape of one dimension.
ShapeHandle Vector(DimensionOrConstant dim);

// Returns a new shape of two dimensions.
ShapeHandle Matrix(DimensionOrConstant dim1, DimensionOrConstant dim2);

// Returns in <out> a new shape whose dimension sizes come from input tensor
// <input_idx>. The tensor must be a 1-dimensional int32 or int64 tensor.  If
// the input tensor is NULL, then an unknown shape is returned.
Status MakeShapeFromShapeTensor(int input_idx, ShapeHandle* out);

// Like the function above, but treats scalar values as unknown
// shapes.  **NOTE** If the scalar is statically known, its value
// must be -1 or an error is returned.
Status MakeShapeFromShapeTensorTreatScalarAsUnknownShape(int input_idx,
                                                         ShapeHandle* out);

// Returns in <out> a new shape corresponding to <proto>.
Status MakeShapeFromShapeProto(const TensorShapeProto& proto,
                               ShapeHandle* out);

// Returns in <out> a new shape corresponding to <partial_shape>.
Status MakeShapeFromPartialTensorShape(
    const PartialTensorShape& partial_shape, ShapeHandle* out);

// Returns in <out> a new shape corresponding to <shape>.
Status MakeShapeFromTensorShape(const TensorShape& shape, ShapeHandle* out);
```

```

// Returns a new dimension of the given size.  The returned value is owned by
// this context.
inline DimensionHandle MakeDim(DimensionOrConstant d) {
  return shape_manager_.MakeDim(d);
}

inline DimensionHandle UnknownDim() { return MakeDim(kUnknownDim); }

// Returns in <val> a scalar value from an input tensor <t>.  The input tensor
// must be a 0-dimensional int32 or int64 tensor.  Caller must ensure that the
// input tensor is not NULL.
Status GetScalarFromTensor(const Tensor* t, int64_t* val);

// Returns in <val> a scalar value from a 1D input tensor <t> with int32 or
// int64 elements. Caller must ensure that the input tensor is not NULL.
Status GetScalarFromTensor(const Tensor* t, int64_t idx, int64_t* val);

// Returns a new dimension whose value is given by a scalar input tensor.
// The input tensor must be in host memory, since it is dereferenced to get
// the value.
Status MakeDimForScalarInput(int idx, DimensionHandle* out);

// Returns a new dimension whose value is given by a scalar input tensor.
// This allows for a negative input dimension given the rank of a separate
// tensor.  This rank can be negative if unknown.
// The input tensor must be in host memory, since it is dereferenced to get
// the value.
Status MakeDimForScalarInputWithNegativeIndexing(int idx, int input_rank,
                                                 DimensionHandle* out);

// Look up the attr being evaluated with name attr_name and set *value to its
// value. If no attr with attr_name is found in def(), or the attr does not
// have a matching type, a non-ok status will be returned.
template <class T>
Status GetAttr(StringPiece attr_name, T* value) const;

// Returns in <out> the result of dividing <dividend> by <divisor>.
// Returns an error if <divisor>  is not positive or if <evenly_divisible>
// and <divisor> does not evenly divide <dividend>.
Status Divide(DimensionHandle dividend, DimensionOrConstant divisor,
              bool evenly_divisible, DimensionHandle* out);

// Returns in <out> the sum of <first> and <second>.
Status Add(DimensionHandle first, DimensionOrConstant second,
           DimensionHandle* out);

// Returns in <out> the dimension that is <first> minus <second>.
Status Subtract(DimensionHandle first, DimensionOrConstant second,
```

```cpp
                  DimensionHandle* out);

  // Returns in <out> the product of <first> and <second>.
  Status Multiply(DimensionHandle first, DimensionOrConstant second,
                  DimensionHandle* out);

  // Returns in <out> the minimum of <first> and <second>. If either <first> or
  // <second> is zero the results is zero. Otherwise, if either <first> or
  // <second> is unknown the results is unknown.
  Status Min(DimensionHandle first, DimensionOrConstant second,
             DimensionHandle* out);

  // Returns in <out> the maximum of <first> and <second>. If either <first> or
  // <second> is unknown the results is unknown.
  Status Max(DimensionHandle first, DimensionOrConstant second,
             DimensionHandle* out);

  Status construction_status() const { return construction_status_; }

  // Methods to propagate shape and dtype on edges of handles. Handles are the
  // dtype DT_RESOURCE which can be used to access state stored in a
  // ResourceManager. When ops (such as variables) consume these handles to
  // produce tensors they might need to know side-information about the shapes
  // and dtypes of tensors which can be accessed via the handle. These methods
  // propagate that information. Output handle dtypes and shapes are ignored if
  // the output tensor is not of type DT_RESOURCE.

  // Merge the stored shapes and types corresponding to the input handle in
  // position idx with the specified shapes and types. This requires idx to be
  // in the [0, num_inputs) range.
  //
  // If the merge is successful and any of the new shapes differs from the old
  // one, or any of the old dtypes was DT_INVALID, store the new shapes and
  // return true.  Return false otherwise.
  //
  // See 'MergeInput' function for full details and examples.
  bool MergeInputHandleShapesAndTypes(
      int idx,
      const std::vector<ShapeAndType>& shapes_and_types) TF_MUST_USE_RESULT;

  // As MergeInputHandleShapesAndTypes, but for an output.
  bool MergeOutputHandleShapesAndTypes(
      int idx,
      const std::vector<ShapeAndType>& shapes_and_types) TF_MUST_USE_RESULT;

  // Relaxes the stored shapes and types corresponding to the input handle in
  // position idx with the specified shapes and types. This requires idx to be
  // in the [0, num_inputs) range.
  //
```

```cpp
// If the relax is successful (sizes are the same, old dtypes match new ones
// or are DT_INVALID), then store the relaxed shapes and return true.
// Return false otherwise.
//
// See 'RelaxInput' function for full details and examples.
bool RelaxInputHandleShapesAndMergeTypes(
    int idx,
    const std::vector<ShapeAndType>& shapes_and_types) TF_MUST_USE_RESULT;

// As RelaxInputHandleShapesAndTypes, but for an output.
bool RelaxOutputHandleShapesAndMergeTypes(
    int idx,
    const std::vector<ShapeAndType>& shapes_and_types) TF_MUST_USE_RESULT;

void set_input_handle_shapes_and_types(
    int idx, const std::vector<ShapeAndType>& shapes_and_types) {
  input_handle_shapes_and_types_[idx] =
      absl::make_unique<std::vector<ShapeAndType>>(shapes_and_types);
}

// Returns the output handle shapes and types, for the resource tensor output
// at index <idx>. Returns NULL if the shape and types were never set.
const std::vector<ShapeAndType>* output_handle_shapes_and_types(int idx) {
  return output_handle_shapes_and_types_[idx].get();
}

// Returns the inputs handle shapes and types, for the resource tensor output
// at index <idx>. Returns NULL if the shape and types were not available.
const std::vector<ShapeAndType>* input_handle_shapes_and_types(int idx) {
  return input_handle_shapes_and_types_[idx].get();
}

void set_output_handle_shapes_and_types(
    int idx, const std::vector<ShapeAndType>& shapes_and_types) {
  output_handle_shapes_and_types_[idx].reset(
      new std::vector<ShapeAndType>(shapes_and_types));
}

// Note that shape functions should usually call MakeShapeFromShapeTensor,
// as it does more analysis to provide partial shapes.
//
// Returns in <out> a new shape whose dimension sizes come from tensor <t>.
// The tensor must be a 1-dimensional int32 or int64 tensor.  If <t> is NULL,
// then an unknown shape is returned.
Status MakeShapeFromTensor(const Tensor* t, ShapeHandle tensor_shape,
                           ShapeHandle* out);

int graph_def_version() const { return graph_def_version_; }

```

```
  const std::vector<std::pair<ShapeHandle, ShapeHandle>>& MergedShapes() const {
    return merged_shapes_;
  }
  const std::vector<std::pair<DimensionHandle, DimensionHandle>>& MergedDims()
      const {
    return merged_dims_;
  }

  // Adds new outputs; useful when mutating the graph.
  Status ExpandOutputs(int new_output_size);

 private:
  // Creates and stores shapes for use in InferenceContext.
  class ShapeManager {
   public:
    ShapeManager();
    ~ShapeManager();

    // Returns a new shape with the given dims. The returned value is owned by
    // this class.
    ShapeHandle MakeShape(const std::vector<DimensionHandle>& dims);

    // Returns a new unknown shape.
    ShapeHandle UnknownShape();

    // Returns a new dimension of the given size.  The returned value
    // is owned by this class.
    inline DimensionHandle MakeDim(DimensionOrConstant d) {
      if (d.dim.IsSet()) {
        return d.dim;
      } else {
        all_dims_.push_back(new Dimension(d.val));
        return all_dims_.back();
      }
    }

   private:
    std::vector<Shape*> all_shapes_;    // values are owned.
    std::vector<Dimension*> all_dims_;  // values are owned.
  };

  friend class ::tensorflow::grappler::GraphProperties;

  friend class ShapeInferenceTest;      // For testing Relax functions.
  friend class ShapeInferenceTestutil;  // For testing shapes.

  // Shared initialization across the two constructors.  Remove
  // once we get rid of one of them.
  void PreInputInit(const OpDef& op_def,
                    const std::vector<const Tensor*>& input_tensors,
                    const std::vector<ShapeHandle>& input_tensors_as_shapes);
  void PostInputInit(std::vector<std::unique_ptr<std::vector<ShapeAndType>>>
                         input_handle_data);

  Status ReturnUnknownShape(ShapeHandle* out) {
    *out = UnknownShape();
    return Status::OK();
  }
  Status ReturnCreatedShape(const std::vector<DimensionHandle>& dims,
                            ShapeHandle* out) {
    *out = MakeShape(dims);
    return Status::OK();
  }

  // Adds additional context to the given status.
  Status AttachContext(const Status& status);

  // Relaxes an existing value <d_old> with a new value <d_new> and returns the
  // relaxed dimension in <*out>. If <d_old> and <d_new> have incompatible
  // values, returns an error.
  //
  // Note that <*out> may be set to <d_old> or <d_new>.
  void Relax(DimensionHandle d_old, DimensionHandle d_new,
             DimensionHandle* out);
  // Relaxes an existing shape <s_old> with a new shape <s_new> and returns the
  // relaxed shape in <*out>. See 'RelaxInput' function for full details and
  // examples.
  void Relax(ShapeHandle s_old, ShapeHandle s_new, ShapeHandle* out);

  // Used to implement MergeInputHandleShapesAndTypes and
  // MergeOutputHandleShapesAndTypes.
  bool MergeHandleShapesAndTypes(
      const std::vector<ShapeAndType>& shapes_and_types,
      std::vector<ShapeAndType>* to_update) TF_MUST_USE_RESULT;
  // Used to implement RelaxInputHandleShapesAndMergeTypes and
  // RelaxOutputHandleShapesAndMergeTypes.
  bool RelaxHandleShapesAndMergeTypes(
      const std::vector<ShapeAndType>& shapes_and_types,
      std::vector<ShapeAndType>* to_update) TF_MUST_USE_RESULT;

  // Forget all the previous merged shapes and dims.
  void ForgetMerges() {
    merged_shapes_.clear();
    merged_dims_.clear();
  }

  // Helper method for MakeShapeFromTensor and MakeShapeFromShapeTensor.
  Status InternalMakeShapeFromTensor(
```

```cpp
      bool treat_unknown_scalar_tensor_as_unknown_shape, const Tensor* t,
      ShapeHandle tensor_shape, ShapeHandle* out);

  ShapeManager shape_manager_;

  // inputs_, outputs_, and input_tensors_as_shapes_ refer to values from
  // `shape_manager_`.
  std::vector<ShapeHandle> inputs_;
  std::vector<const Tensor*> input_tensors_;
  std::vector<bool> requested_input_tensor_;
  std::vector<ShapeHandle> outputs_;
  // Can have fewer elements than inputs_.
  std::vector<ShapeHandle> input_tensors_as_shapes_;
  std::vector<bool> requested_input_tensor_as_partial_shape_;

  // input_handle_shapes_and_types_[i] is the list of shape/type pairs available
  // through the resource handle passed along input i of the node.
  //
  // Values may be NULL.
  std::vector<std::unique_ptr<std::vector<ShapeAndType>>>
      input_handle_shapes_and_types_;

  // output_handle_shapes_and_types_[i] is the list of shape/type pairs
  // available through the resource handle passed along output i of the node.
  //
  // Values may be NULL.
  std::vector<std::unique_ptr<std::vector<ShapeAndType>>>
      output_handle_shapes_and_types_;

  // Return types for the node this context is associated with. This information
  // is to eventually consolidate all the dtype and shape info, allowing for
  // output_handle_shapes_and_types_ to be removed.
  FullTypeDef ret_types_;

  const int graph_def_version_;
  AttrSlice attrs_;
  NameRangeMap input_name_map_;
  NameRangeMap output_name_map_;

  // An error set during construction. TODO(cwhipkey): remove when test
  // constructor is removed.
  Status construction_status_;

  // Pair of shape or dim handles that are equivalent, ie that represent the
  // same underlying shape of dimension. Note that for each pair at least one of
  // the handles must contain an unknown shape, since we don't keep track of
  // known shapes or dims here.
  std::vector<std::pair<ShapeHandle, ShapeHandle>> merged_shapes_;
  std::vector<std::pair<DimensionHandle, DimensionHandle>> merged_dims_;
  TF_DISALLOW_COPY_AND_ASSIGN(InferenceContext);
};

// -----------------------------------------------------------------------------
// Template and inline method implementations, please ignore

inline Dimension::Dimension() : value_(InferenceContext::kUnknownDim) {}
inline Dimension::Dimension(int64_t value) : value_(value) {
  DCHECK(value >= 0 || value == InferenceContext::kUnknownDim)
      << "Dimension must be non-negative or equal to "
         "InferenceContext::kUnknownDim but got "
      << value;
}

inline Shape::Shape() : rank_(InferenceContext::kUnknownRank) {}
inline Shape::Shape(const std::vector<DimensionHandle>& dims)
    : rank_(dims.size()), dims_(dims) {}

inline DimensionOrConstant::DimensionOrConstant(DimensionHandle dim)
    : dim(dim) {
  DCHECK(dim.IsSet()) << "Internal error: Got nullptr for Dimension.";
}

inline DimensionOrConstant::DimensionOrConstant(int64_t val) : val(val) {
  DCHECK(val >= 0 || val == InferenceContext::kUnknownDim)
      << "Dimension must be non-negative or equal to "
         "InferenceContext::kUnknownDim but got "
      << val;
}

template <class T>
Status InferenceContext::GetAttr(StringPiece attr_name, T* value) const {
  return GetNodeAttr(attrs_, attr_name, value);
}

}  // namespace shape_inference
}  // namespace tensorflow

#endif  // TENSORFLOW_CORE_FRAMEWORK_SHAPE_INFERENCE_H_
```